SPRINT book

Паоло Феррагина

Разработка алгоритмов

инженерный подход

элегантные решения алгоритмических задач,
возникающих при создании приложений
для работы с большими данными


SPRINT
book

Pearls of Algorithm Engeneering

PAOLO FERRAGINA
University of Pisa

Разработка алгоритмов

инженерный подход

ПАОЛО ФЕРРАГИНА

Пизанский университет

SPRINT book 2026

Паоло Феррагина

Разработка алгоритмов. Инженерный подход

Перевела с английского И. Рузмайкина

ББК 32.973.2-018
УДК 004.421

Феррагина Паоло

Ф43 Разработка алгоритмов. Инженерный подход. — Астана: «Спринт Бук», 2026. — 352 с.: ил.

ISBN 978-601-12-3657-7

Большинство книг об алгоритмах фокусируются на нотации «О большое» и основных принципах проектирования, однако эта книга предлагает уникальный подход, выводя разработку и анализ на уровень предсказуемой практической эффективности. В ней обсуждаются базовые и классические алгоритмические задачи, возникающие при создании приложений больших данных, для которых демонстрируются элегантные решения постепенно возрастающей сложности. Анализ решений дается в рамках как классической RAM-модели, так и более значимой с практической точки зрения модели с использованием внешней памяти, позволяющей оценивать сложность ввода-вывода.

В книге рассматриваются различные типы данных, включая целые числа, строки, деревья и графы, разные алгоритмические инструменты, такие как выборка, сортировка, сжатие данных и поиск по словарям и текстам. Наконец, вы найдете здесь информацию о последних разработках, связанных со сжатыми структурами данных. Алгоритмические решения сопровождаются подробным псевдокодом и множеством работающих примеров, что позволит обогатить инструментарий студентов, исследователей и профессионалов, заинтересованных в результативной и экономичной обработке больших данных.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1009123280 англ.

ISBN 978-601-12-3657-7

Изготовлено в России. Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес: 010000, Казахстан, город Астана, район Алматы, Проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 11.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 19.09.25. Формат 70x100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 1000. Заказ 0000.

КРАТКОЕ СОДЕРЖАНИЕ

ОГЛАВЛЕНИЕ

Отзывы о книге

В 2000 году, когда я пришел работать в Google, алгоритмические проблемы возникали каждый день. В то время даже опытные инженеры не имели всеобъемлющей подготовки, необходимой для проектирования эффективных алгоритмов. Хорошо написанная и в то же время лаконичная книга Паоло Феррагины помогает восполнить этот пробел. Инженер-программист, освоивший этот материал, станет ценным сотрудником для любой компании.

Мартин Фарах-Колтон, Ратгерский университет

Написано множество книг по теории проектирования алгоритмов, но далеко не во всех рассказывается об их реализации, оптимизации и внедрении в реальные системы. Перед вами одна из тех редких книг, где больше внимания уделяется практическим аспектам. А именно они выходят на первый план, когда требуется действительная производительность, ведь то и дело оказывается, что некоторые теоретически привлекательные алгоритмы бесполезны в реальной жизни. Надеюсь, чтение этой книги будет для вас таким же приятным и вдохновляющим, каким оно было для меня.

Гонсало Наварро, Чилийский университет

Паоло Феррагина совмещает навыки инженера-программиста, математика, специализирующегося на алгоритмах, и новатора в педагогике, что позволило создать прекрасное ожерелье, состоящее из жемчужин-алгоритмов. Красота этого ожерелья сочетается с вычислительной эффективностью. Эта книга должна быть в библиотеке каждого, кто интересуется красотой кода и кодом красоты.

Бад Мишра, Курантовский институт
при Нью-Йоркском университете

Существует множество учебников по алгоритмам, которые фокусируются на нотации «О большое» и общих принципах разработки. В этой книге автор выводит проектирование и анализ на уровень предсказуемой практической эффективности. Преимущества рандомизации элегантно используются для получения простых алгоритмов, информативный анализ которых дает читателям полезные инструменты для применения в других условиях. Эта книга бесценна для расширения учебной программы по информатике курсом проектирования алгоритмов.

Вели Мякинен, Хельсинкский университет

ОБ АВТОРЕ

Паоло Феррагина — профессор Пизанского университета и школы передовых исследований Сант'Анна, «гуру алгоритмов». После получения докторской степени стажировался в Институте информатики Макса Планка. Занимал должность проректора по информационно-коммуникационным технологиям (2019–2022) и прикладным исследованиям и инновациям (2010–2016), а также возглавлял программу подготовки аспирантов в сфере Computer Science (2018–2020). Его исследования связаны с разработкой алгоритмов и структур данных для анализа и обработки Big Data. В 2022 году он стал одним из лауреатов престижной премии Париса Канеллакиса за теоретические и практические достижения; также является обладателем множества других международных наград. Ранее Феррагина сотрудничал с компаниями AT&T, Bloomberg, Google, ST microelectronics, Tiscali и Yahoo. В результате его исследований было получено несколько патентов и написано более 170 статей. Он вел исследовательскую работу в Институте информатики Общества Макса Планка, Университете Северного Техаса, Курантовском институте при Нью-Йоркском университете, медицинской школе Гарвардского университета, AT&T, Google, IBM Research и Yahoo.

ПРЕДИСЛОВИЕ

В этой книге собраны советы для программистов и разработчиков: как бы вы ни были сведущи в проектировании алгоритмов, вряд ли сможете быстро придумать разумное решение реальных задач, ведь эти задачи настолько разрослись, компьютеры стали такими сложными, пользователи — такими требовательными, приложения — такими жадными до ресурсов, а алгоритмические инструменты — такими продвинутыми, что инженеры — проектировщики алгоритмов уже не могут полагаться на импровизацию. Им требуется сначала пройти обучение.

Многочисленные подтверждения этих слов вы найдете, когда будете читать про сложные задачи и элегантные и эффективные алгоритмические методы их решения. При выборе тем я руководствовался двумя целями. С одной стороны, хотелось дать читателям *инструментарий для разработки алгоритмов*, помогающий в решении задач, связанных с большими наборами данных, с другой — тянуло систематизировать научный материал, которого лично мне не хватало, когда я был студентом магистратуры и аспирантом. Некоторые разделы, которые, как правило (хотя и не всегда), расположены в конце глав, обозначены значком ∞. Они содержат более сложную информацию, которую можно пропустить без ущерба для понимания остального текста. Для читателей, увлеченных программированием, отмечу, что индексы массивов у меня начинаются с 1, а не с 0, как принято. Это сделано для того, чтобы не усложнять формулы в алгоритмах наличием ±1.

Стиль и содержание этих глав выкристаллизовались в результате многочасовых, иногда вдохновляющих, а иногда тяжелых и утомительных дискуссий с коллегами-исследователями и студентами. В некоторые из этих лекций вошли курсы по поиску информации и продвинутым алгоритмам, которые я читаю в Университете Пизы и различных международных аспирантских школах с 2004 года. В частности, предварительный проект этих заметок был подготовлен студентами курса «Алгоритмическая инженерия» в магистратуре по компьютерным наукам и сетям в сентябре — декабре 2009 года в рамках сотрудничества между Университетом Пизы и Школой перспективных исследований Святой Анны. Некоторые тексты подготовлены аспирантами, посещавшими курс «Продвинутые алгоритмы для массивных наборов данных», который я вел в Международной весенней школе Бертиноро (BISS), проходившей в марте 2010 года (Бертиноро, Италия). Все накопившиеся за время работы черновики я использовал в качестве основы некоторых глав. Конечно, за последующие годы в эти заметки было внесено множество изменений благодаря исправлениям и предложениям, поступавшим от многочисленных студентов курсов по алгоритмической инженерии.

Особая благодарность Антонио Боффе, Андреа Гуэрре, Франческо Тозони и Джорджио Винчигуэрре за внимательное прочтение последней версии этой книги,

Джемме Мартини за вклад в главу 15 и Рикардо Манетти за создание иллюстраций в tikz. Огромное спасибо за многие часы увлекательных и сложных дискуссий на темы, которые в итоге легли в основу этой книги. Я хочу поблагодарить за это моих аспирантов и коллег: Юрки Алакуйалу, Рикардо Баэса-Йейтса, Лоренцо Белломо, Масси Чиарамиту, Марко Корнолти, Мартина Фарах-Колтона, Андреа Фарруджиа, Рафаэле Джанкарло, Роберто Гросси, Антонио Гулли, Луиджи Лауру, Вели Мякинена, Джованни Манзини, Курта Мельхорна, Улли Майера, Бада Мишру, С. Мутукришнана, Гонсало Наварро, Игоря Нитто, Линду Пали, Франческо Пиччинну, Луку Пинелло, Марко Понцу, Прабхакара Рагхавана, Питера Сандерса, Россано Вентурини и Джеффа С. Виттера. И наконец, самую горячую благодарность я хочу выразить моему наставнику Фабрицио Луччио, который постоянно стимулировал мою исследовательскую страсть и привил мне желание преподавать и писать настолько просто и ясно, насколько это возможно... но не слишком просто. Вам судить, удалось ли мне достичь этой цели в этой книге.

Я горячо надеюсь, что при чтении книги вы будете испытывать те же удовольствие и волнение, которые наполняли меня, когда я впервые встретил эти алгоритмические решения. Если это так, пожалуйста, читайте как можно больше об алгоритмах, чтобы найти вдохновение для своей академической и профессиональной деятельности. *Программирование по-прежнему остается искусством*, но, чтобы выразить свои идеи максимально красиво, вам потребуются хорошие инструменты.

П. Ф.

ОТ ИЗДАТЕЛЬСТВА

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научных редакторах русскоязычного издания

Георгий Курячий окончил факультет вычислительной математики и кибернетики МГУ им. М. В. Ломоносова по специальности «Прикладная математика». Имеет большой опыт преподавания в вузах, участвовал в разработке образовательных курсов для ВМК МГУ и ВШЭ ФКН.

Антон Русанов — ИТ-предприниматель и Ruby-разработчик с более чем 25-летним опытом. Ранее — заместитель руководителя направления ЦОД в компании АРСИЭНТЕК, участвовал в масштабных проектах по построению и развитию корпоративных информационных систем. Специализируется на разработке ПО, архитектуре приложений и стратегическом развитии ИТ-инфраструктуры.

Глава 1
ВВЕДЕНИЕ

> Это сложные вещи, но не нужно быть гением,
> чтобы ими пользоваться.
>
> *Джек Нунен, президент корпорации SPSS*

Эта книга посвящена *алгоритмам*. Чтобы глубоко почувствовать их красоту и осознать все сложности их разработки, первым делом следует понять, что же это такое. Оксфордский словарь английского языка сообщает, что алгоритмом называется *«обычно записываемый в алгебраической нотации процесс или набор правил, который в настоящее время используется, в частности, в вычислениях, машинном переводе и лингвистике»*. Современное значение этого термина сходно со значением слов *«рецепт»*, *«метод»*, *«процедура»* или *«функция»*, но в информатике ему дается более точное определение. За последние 200 лет множество авторитетных исследователей пытались закрепить значение этого термина, предлагая все более сложные и подробные определения, считая их более точными и элегантными[1]. Мы, как проектировщики алгоритмов и инженеры, будем пользоваться определением Дональда Кнута, сформулированным в конце 1960-х годов [7][2], согласно которому алгоритмом называется *«конечная, определенная, эффективная процедура, дающая какой-то результат»*. Эти интуитивно понятные характеристики широко принимаются в качестве требований к последовательности составляющих алгоритм шагов. Но они настолько глубоки, что следует рассмотреть каждую из них более подробно. Это наглядно продемонстрирует не только сценарии и проблемы, возникающие при проектировании и разработке алгоритмов, но и цель написания этой книги.

- **Конечность.** «Алгоритм всегда должен завершаться после конечного числа шагов... разумно конечного числа». Очевидно, что уточняющее наречие «разумно» связано с *эффективностью* алгоритма, ведь, по утверждению Кнута [7], «на практике требуются не просто алгоритмы, а годные алгоритмы». Годность зависит от того, как алгоритм поступает с такими ценными *вычислительными ресурсами*, как время работы процессора, используемая память, загрузка сети и подсистемы ввода-вывода, потребляемая энергия, а также от простоты и элегантности самого алгоритма. Последнее влияет на величину затрат на написание кода, его отладку и поддержку.

---

[1] См. статью «Характеристики алгоритмов»: https://en.wikipedia.org/wiki/Algorithm_characterizations.

[2] Список литературы, упоминаемой в тексте главы, приводится в ее конце. — *Примеч. ред.*

- **Определенность.** «Каждый шаг алгоритма должен быть строго определен путем досконального и однозначного указания всех действий, которые необходимо выполнить для каждого случая». Продемонстрировать этот принцип Кнут попытался, описав машинный язык для своего гипотетического компьютера MIX. Сегодня существует множество языков программирования, таких как C/C++, Java, Python и т. д. Для каждого из них предусмотрен набор команд, позволяющих программистам однозначно описывать лежащие в основе алгоритмов процедуры. Однозначность в данном случае обеспечивается формальной семантикой, закрепленной за каждой командой. Именно поэтому любой читающий алгоритм человек сможет правильно его интерпретировать.
- **Эффективность.** «Все составляющие алгоритм операции должны быть достаточно простыми, чтобы их в принципе мог реализовать человек с помощью бумаги и карандаша». Поэтому упомянутое ранее понятие «шаг» подразумевает необходимость полностью и глубоко понимать решаемую задачу и четко определять пошаговую структуру решения.
- **Процедура.** «Логически упорядоченная последовательность шагов».
- **Входные данные.** «Параметры, которые задаются перед началом работы алгоритма. Они берутся из заданных наборов объектов». Соответственно, поведение алгоритма не уникально, а зависит от предназначенных для обработки наборов объектов.
- **Выходные данные.** «Величины, определенным образом связанные с входными данными». По сути, это ответ, возвращаемый алгоритмом для заданных входных данных.

В этой книге я постарался избежать формального подхода к описанию алгоритмов, так как мне хотелось сосредоточиться на элегантных в теории и эффективных на практике идеях, составляющих основу алгоритмического решения некоторых интересных задач. При этом крайне важно было не завязнуть во множестве технических деталей программирования. Поэтому каждая глава посвящена разбору интересной задачи, порожденной каким-то практическим сценарием. Я буду предлагать решения все большей сложности и повышенной эффективности, стараясь, чтобы это не увеличивало сложность описания алгоритма. Я старался выбирать задачи, допускающие удивительно элегантные решения, которые можно записать несколькими строками кода. Поэтому предпочтение было отдано принятой в настоящее время практике проектирования алгоритмов, когда их поясняют на словах или с помощью имитирующего наиболее известные языки *псевдокода*. Но все описания будут достаточно строгими, чтобы соответствовать шести признакам Кнута.

Разумеется, *элегантность* будет не единственным критерием при проектировании алгоритмов — нашей целью станет еще и *эффективность*, как правило связанная с *временной и пространственной сложностью*. Традиционно временная сложность оценивается как функция от размера входных данных n путем подсчета максимального количества шагов $T(n)$, необходимого алгоритму для завершения работы. Поскольку при оценке учитывается максимальное по всем входным данным

размером n время работы алгоритма, говорят про *время в худшем случае*. Понятно, что чем больше n, тем больше будет $T(n)$, то есть это неубывающая положительная функция. Аналогично наихудшая пространственная сложность алгоритма определяется как максимальное количество ячеек памяти, используемых для вычислений над входными данными размером n.

Такой подход к *проектированию* и *анализу* алгоритмов базируется на очень простой вычислительной модели, известной как *модель фон Неймана* (она же машина с произвольным доступом к памяти, или RAM-машина). Она состоит из процессора и памяти бесконечного размера с постоянным временем доступа к каждой из ячеек. Это означает, что каждый шаг занимает фиксированное время, одинаковое для любой операции — арифметической, логической или просто операции доступа к памяти (чтение/запись). Соответственно, для точной оценки времени выполнения алгоритма на ПК достаточно *подсчитать* количество его шагов. После чего сравнивается асимптотическое поведение функций временной сложности разных алгоритмов при $n \to +\infty$: чем быстрее с ростом размера входных данных растет временная сложность, тем хуже алгоритм. Надежность этого подхода долго была предметом дискуссий, тем не менее RAM-машина доминировала на алгоритмической сцене десятилетиями (и до сих пор не утратила своих позиций), ведь она достаточно проста, что влияет на разработку и оценку алгоритмов. Кроме того, на старых ПК и при малых размерах входных данных она дает довольно точную оценку производительности. Поэтому неудивительно, что в большинстве книг, знакомящих с алгоритмами, для оценки их производительности используется именно модель фон Неймана [6].

Но последние десятилетия стали временем существенных перемен, что вызвало необходимость изменений в разработке и анализе алгоритмов. Во-первых, усложнилась архитектура современных ПК (это давно уже не один ЦП с однородной оперативной памятью). Во-вторых, резко возрос размер входных данных и теперь случай $n \to +\infty$ перестал быть только теорией, ведь в наше время данные в изобилии генерируются множеством источников, таких как расшифровка последовательностей ДНК, банковские трансакции, мобильная связь, навигация и поиск в Интернете, аукционы и пр. Для современных ПК RAM-машина стала абстракцией, которую нельзя применить, зато рост количества данных так повлиял на бизнес [2], общество [1] и науку в целом [3], что разработка алгоритмов, подходящих для предельных случаев, стала интересовать не только теоретиков, но и гораздо более широкую аудиторию профессионалов. В результате мы наблюдаем не только заново вспыхнувший интерес к этой теме, но и появление термина «алгоритм» даже в обычной речи.

В новых условиях потребовались новые вычислительные модели, способные лучше абстрагироваться от особенностей современных компьютеров и приложений и точнее оценивать производительность алгоритмов. Современный ПК состоит из одного или нескольких ЦП (многоядерных, графических, тензорных и т. п.) и имеет сложную иерархию уровней памяти. У каждого уровня свои технологические особенности (рис. 1.1). Это кэши L1 и L2, RAM, один или несколько HDD или SSD;

возможно, данные распределены по целому набору вычислительных узлов в сети (хостов) со своей иерархией (которая зависит и от их географического положения) — так называемое облако. Каждый из этих уровней памяти имеет собственные стоимость, емкость, задержку, пропускную способность и метод доступа. Чем ближе к ЦП, тем он меньше, быстрее и дороже. Для доступа к кэшам достаточно наносекунд, тогда как извлечение данных с дисков (ввод-вывод на внешнем устройстве) занимает миллисекунды. Это так называемое *узкое место в подсистеме ввода-вывода* с поразительным коэффициентом замедления 10^5–10^6, который прекрасно иллюстрирует цитата, приписываемая Томасу Кормену: «Разницу в скоростях доступа к оперативной и дисковой памяти можно сравнить со скоростью заточки карандаша в случаях, когда точилка лежит на вашем столе и когда за ней требуется лететь на другой конец света».

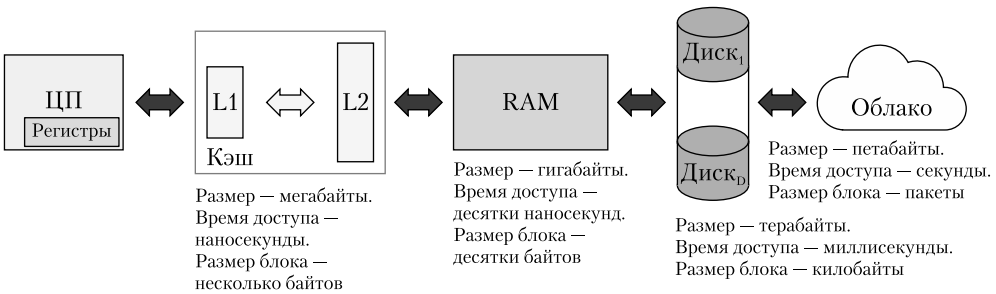


Рис. 1.1. Пример иерархии запоминающих устройств современного ПК

Разумеется, в ходе инженерных исследований ищут способы снизить влияние узкого места в подсистеме ввода-вывода на эффективность приложений, работающих с большими наборами данных. Но хорошие разработка и проектирование алгоритмов дают результаты, превосходящие лучшие технологические достижения. Сейчас на простом примере я покажу, почему так происходит[1].

Рассмотрим три алгоритма оценки времени ввода-вывода с возрастающей сложностью: $C_1(n) = n$, $C_2(n) = n^2$ и $C_3(n) = 2^n$. Здесь $C_i(n)$ —количество обращений к диску, к которому i-й алгоритм прибегает для обработки n входных данных. В первых двух случаях мы видим *полиномиальный* рост количества операций ввода-вывода, в последнем же случае оно растет *экспоненциально*. Для упрощения вычислений взяты максимально простые и поэтому нереалистичные формулы, так как в данном случае моя задача — просто продемонстрировать принцип. Оценим количество данных, которое каждый алгоритм обрабатывает за время t, если каждая операция ввода-вывода занимает время c. Для этого нужно найти n из уравнения $C_i(n)c = t$. Для первого алгоритма этот параметр рассчитывается по формуле t/c, для второго — $\sqrt{t/c}$, а для третьего — $\log_2(t/c)$, что наглядно показывает, почему полиномиальные алгоритмы

[1] Я использовал объяснение Фабрицио Луччо [8], заменив шаги алгоритма операциями ввода-вывода.

считаются *эффективными*, а экспоненциальные — нет, ведь увеличение времени t не сильно меняет объем обрабатываемых экспоненциальным алгоритмом данных.

Разумеется, это утверждение допускает исключения, например, для задач с ограниченным размером входных данных. Бывает и так, что распределение данных благоприятствует эффективному выполнению задачи. Но такие ситуации возникают довольно редко, так что меньшее время выполнения полиномиальных алгоритмов означает, что они считаются *доказуемо* эффективным и предпочтительным способом решения задач. Опыт показывает, что в большинстве случаев экспоненциальное время появляется при решении задачи методом полного перебора, тогда как полиномиального времени можно добиться только благодаря более глубокому пониманию сути задачи. Именно поэтому полиномиальные алгоритмы со многих точек зрения можно считать оптимальным вариантом.

Предположим, что наши алгоритмы удалось запустить на компьютере, подсистема ввода-вывода которого работает быстрее в k раз. С каким объемом данных сможет справиться такой компьютер? Для ответа на этот вопрос нужно поменять временной интервал на $k \times t$, обозначив, что теперь для выполнения алгоритмов доступно в k раз больше времени, чем в предыдущем случае. Мы увидим, что первый алгоритм будет работать в k раз быстрее, скорость второго домножится на $\sqrt{k}$, а к скорости последнего *всего лишь прибавится небольшое число* $\log_2 k$. То есть рост вычислительных мощностей в k раз незначительно сказывается на времени выполнения экспоненциального алгоритма. На алгоритмы со степенной зависимостью от времени, такие как второй из упомянутых ранее алгоритмов, рост мощности влияет позитивно, но чем выше степень, тем меньшее улучшение производительности мы увидим. А именно, для алгоритма $C(n) = n^{\alpha}$ в k раз более мощный компьютер даст прирост скорости в $\sqrt[\alpha]{k}$ раз. Так что можно с достаточной уверенностью утверждать: правильный выбор алгоритма влияет на скорость выполнения намного сильнее, чем любой рост производительности HDD или SSD[1].

Теперь, когда я показал вам важность корректного проектирования алгоритмов, вернемся к анализу их производительности. Для примера рассмотрим простую задачу — нахождение суммы целых чисел, хранящихся в массиве $A[1, n]$. Первое, что приходит в голову, — это поэлементный просмотр и суммирование элементов массива с сохранением промежуточного результата во временной переменной. Этот алгоритм состоит из n шагов, так как операция сложения выполняется после обращения к каждому элементу массива A.

Для перехода к общему случаю рассмотрим семейство алгоритмов $A_{s, b}$, в которых шаблон доступа к элементам задается параметрами s и b. В частности, каждый массив будет логически разделен на блоки из b элементов, например $A_j = A[j \times b + 1, (j + 1) \times b]$ для $j = 0, 1, 2, ..., n/b - 1$[2]. После суммирования элементов в блоке A_j происходит переход к блоку A_{j+s}, находящемуся на s блоков правее. Будем считать массив A циклическим. В этом случае, если переход к следующему блоку приводит

[1] Более подробно эта тема рассматривается в книге Джеффри С. Виттера [11].

[2] Для удобства предположим, что n и b являются степенями двойки, то есть b — делитель n.

к выходу за границы массива, алгоритм начинает просмотр с начала, рассчитывая индекс блока по формуле $(j + s) \bmod (n/b)$[1]. Очевидно, что далеко не при всех s мы сможем учесть все блоки массива A и таким образом суммировать содержащиеся в нем целые числа. Но в случае, когда s и n/b являются взаимно простыми числами, последовательность индексов посещенных блоков, то есть $j = s \times i \bmod (n/b)$ для $i = 0, 1... n/b - 1$, будет перестановкой целых чисел $\{0, 1... n/b - 1\}$. Соответственно, шаблон $A_{s,b}$ затронет все блоки массива A, и мы получим сумму всех содержащихся в нем целых чисел. Зачем нужна такая параметризация? Дело в том, что, меняя s и b, мы сможем суммировать числа в массиве A в соответствии с различными шаблонами доступа к памяти: от упомянутого ранее последовательного перебора (случай $s = b = 1$) до последовательного доступа к блокам (при большем b) и случайного доступа к блокам (при большем s). С точки зрения вычислений все алгоритмы $A_{s,b}$ эквивалентны, так как каждый из них читает и суммирует ровно n элементов. Но на практике они имеют разную производительность, причем эта разница увеличивается с ростом n, ведь чем больше размер массива, тем по большему количеству уровней памяти будут распределяться данные. При этом каждый из уровней имеет собственные емкость, задержку, пропускную способность и метод доступа. Соответственно, эквивалентная эффективность, полученная ранее на модели фон Неймана, не дает представления о том, сколько же времени на самом деле займет суммирование элементов массива A.

Нужна другая модель, лучше описывающая работу реальных компьютеров и при этом достаточно простая для того, чтобы мы по-прежнему могли разрабатывать и анализировать алгоритмы. Ранее я утверждал, что с учетом большого разрыва в производительности дисковой и оперативной памяти хорошей оценкой временной сложности алгоритма может служить количество операций ввода-вывода. Это отражено в *двухуровневой модели памяти*, которая абстрагирует компьютер до внутренней памяти ограниченного размера M и неограниченной дисковой памяти, которая работает путем записи данных в блоки размером B, называемые *страницами диска*, и чтения с них. Модель может включать в себя D дисков неограниченного размера. В этом случае при каждом вводе-выводе на D страниц, расположенных на разных дисках, записывается в общей сложности $D \times B$ элементов и столько же читается с них. Следует отметить, что двухуровневое представление не ограничивает модель абстрактными вычислениями на основе дисковой памяти. Мы можем выбрать любые два уровня иерархии памяти с корректно заданными параметрами M и B. Производительность алгоритма в этой модели оценивается путем подсчета:

- обращений к страницам диска (далее будем называть их вводом-выводом);
- времени выполнения (процессорного времени);
- страниц диска, используемых алгоритмом в качестве рабочего пространства.

Кроме того, для проектирования хороших алгоритмов, работающих с большими наборами данных, предлагается учитывать принципы *пространственной* и *временной*

[1] Функция деления по модулю для двух положительных целых чисел x и $m > 1$ определяется как остаток от деления x на m.

локальности. Первый предписывает структурировать данные на диске (-ах) таким образом, чтобы каждая считанная оттуда страница содержала как можно больше полезных данных. Второй требует, чтобы перед записью на диск с данными во внутренней памяти было проделано как можно больше полезной работы.

Проанализируем временную сложность алгоритмов $A_{s,\,b}$ для новой модели. Будем считать, что процессорное время равно n, а занятое пространство составляет n/B страниц диска при любых значениях s и b. Начнем с простейшего случая $s = 1$. Здесь все очевидно. Алгоритмы $A_{1,\,b}$ перемещаются по элементам массива A вправо, суммируя элементы по одному блоку за раз, выполняя n/B операций ввода-вывода при любых b. При изменении s и b ситуация немного усложняется. Пусть $s = 2$. Выберем $b < B$, которое для простоты будет делителем B. То есть каждый блок размером B состоит из B/b (логических) блоков размером b и алгоритмы $A_{2,\,b}$ проверяют только половину из них, ведь мы установили параметр $s = 2$. Фактически это означает, что каждая страница размером B участвует в процессе суммирования наполовину, индуцируя в общей сложности $2n/B$ операций ввода-вывода. Эту формулу несложно обобщить, записав затраты на операции ввода-вывода как $\min\{s, B/b\} \times (n/B)$, что для больших скачков по массиву A дает n/b. Она намного лучше оценивает сложность алгоритмов $A_{s,\,b}$ в реальном времени, хотя и не учитывает все особенности диска. Все операции ввода-вывода считаются равноценными независимо от их распределения, что не соответствует действительности, потому что на реальных дисках последовательный ввод-вывод выполняется быстрее случайного[1].

Таким образом, мы выяснили, что все алгоритмы $A_{s,\,b}$ имеют одинаковую сложность ввода-вывода n/B независимо от s, хотя при использовании механических дисков их поведение сильно различается, ведь с ростом s растет и время поиска на диске. Можно сделать вывод, что даже двухуровневая модель памяти достаточно хорошо описывает поведение алгоритмов на реальных компьютерах. Именно поэтому она широко применяется для оценки производительности алгоритмов на больших наборах данных. Для достижения максимальной точности мы будем учитывать не только количество операций ввода-вывода, но и характеристику их *распределения* (случайного или последовательного) по диску.

Здесь можно вспомнить, что за последние годы размер внутренней памяти M увеличился настолько, что теоретически в нее может поместиться большая часть рабочего набора алгоритма, то есть набора страниц, к которым он будет обращаться в ближайшем будущем. Это позволяет значительно сократить количество ошибок ввода-вывода. Но сейчас я вам покажу, как замедляет работу алгоритма даже небольшая часть данных, находящихся на диске, и вы поймете, почему не стоит пренебрегать организацией данных и в чрезвычайно благоприятных ситуациях.

Предположим, размер входных данных $n = (1 + \varepsilon)M$ больше, чем размер внутренней памяти, умноженный на $\varepsilon > 0$. Насколько это повлияет на среднюю стоимость

[1] Эта разница незначительна в случае DRAM и твердотельных накопителей, где распределение обращений к памяти не оказывает существенного влияния на пропускную способность.

шага алгоритма, который обращается к данным, находящимся как во внутренней памяти, так и на диске? Чтобы упростить анализ, введем в рассмотрение такой параметр, как вероятность *промаха* $p(\varepsilon)$ (непопадания в дисковый кэш в оперативной памяти). Когда алгоритм работает только с дисковыми данными, $p(\varepsilon) = 1$, если же набор данных целиком помещается в оперативной памяти, то $p(\varepsilon) = 0$. Для случаев, когда алгоритм работает с данными, хранящимися как в памяти, так и на диске, $p(\varepsilon) = \frac{\varepsilon M}{(1+\varepsilon)M} = \frac{\varepsilon}{1+\varepsilon}$. Другими словами, этот параметр можно считать мерой нелокальности запросов алгоритма к памяти.

Перечислю остальные нужные нам параметры. Пусть c — это соотношение скоростей одной операции ввода-вывода и одного доступа к внутренней памяти. На практике $c \approx 10^5 ... 10^6$, выше я это уже упоминал. Долю шагов, для которых алгоритму требуется доступ к памяти, обозначим f. Согласно [5], этот параметр, как правило, составляет 30–40 %. Средние временные затраты на такой доступ к памяти обозначим t_m, затраты же на каждый шаг вычисления или на доступ к внутренней памяти пусть будут равны 1. Чтобы получить формулу для расчета t_m, следует выделить два случая: доступ к памяти, происходящий с вероятностью $1 - p(\varepsilon)$, и доступ к диску, происходящий с вероятностью $p(\varepsilon)$. В результате мы получим $t_m = 1 \times (1 - p(\varepsilon)) + c \times p(\varepsilon)$.

Теперь для нашего алгоритма можно оценить *средние временные затраты на один шаг*. Они рассчитываются по формуле $1 \times (1 - f) + t_m \times f$, где $1 - f$ — это доля вычислительных шагов, а f — доля обращений к памяти, как внутренней, так и дисковой. Подставив вычисленное значение t_m, мы можем понизить эту оценку до $3 \times 10^4 \times p(\varepsilon)$. Эта формула ясно показывает, что даже для алгоритмов, использующих локальность обращений, то есть небольшую $p(\varepsilon)$, замедление может оказаться значительным — где-то на четыре порядка выше ожидаемой величины, то есть $p(\varepsilon)$. Для примера возьмем алгоритм, в котором локальность обращений к памяти строго ограниченна: скажем, всего 1 из 1000 обращений идет к данным, хранящимся на диске, то есть $p(\varepsilon) = 0{,}001$. В результате его производительность будет более чем в 30 раз ниже, чем производительность алгоритма, ведущего вычисления полностью во внутренней памяти.

Разумеется, это лишь вершина айсберга. Чем больше объем данных, которые должен обработать алгоритм, тем больше уровней памяти будет задействовано в их хранении и тем разнообразнее будут типы промахов, с которыми придется бороться для повышения эффективности. Надеюсь, я смог продемонстрировать вам, что в системах с иерархической памятью игнорирование затрат на обращения к памяти может помешать использовать алгоритм для больших входных данных.

В этой книге я разобрал несколько сложных задач, имеющих элегантные алгоритмические решения, ведь именно эффективность алгоритмов обеспечивает возможность работы с большими наборами данных, которые встречаются во многих реальных приложениях. Я продемонстрирую подробности разработки каждого алгоритма, отдельно прокомментировав трудности, с которыми пришлось столкнуться. Мы поговорим о способах превращения теоретически эффективных алгоритмов

в практически эффективный код. Как теоретик, я слишком часто слышал фразу: «Ваш алгоритм вряд ли допускает эффективную реализацию!» Кроме того, внося свой вклад в недавний всплеск исследований в области *проектирования алгоритмов* [10], мы углубленно рассмотрим вычислительные особенности некоторых из них, прибегнув к другим успешным моделям вычислений — в основном к потоковой [9] и кэш-независимой модели [4]. Эти модели позволят зафиксировать и выделить некоторые интересные проблемы базовых вычислений, такие как проходы по диску (потоковая модель) и универсальная масштабируемость (модель с независимостью от кэша). Я постараюсь максимально просто описать все эти проблемы, однако вряд ли смогу превратить эту высшую математику для неспециалистов в науку для «чайников» [2], ведь для создания рабочего алгоритма необходимо учитывать гораздо больше вещей. В ведущих ИТ-компаниях, таких как Amazon, Facebook, Google, IBM, Microsoft, Oracle, Spotify и т. д., прекрасно понимают, как сложно найти людей с навыками, подходящими для проектирования и разработки хороших алгоритмов. Моя книга лишь поверхностно коснется проектирования и разработки алгоритмов, ее главная цель — вдохновить вас на повседневную работу в качестве разработчика программного обеспечения.

Список литературы

1. Person of the year // Time Magazine, 168: 27, December 2006.
2. Business by numbers // The Economist, September 2007.
3. *Butler D.* 2020 computing: Everything, everywhere // Nature, 440 (7083): 402–405, 2006.
4. *Fagerberg R.* Cache-oblivious model // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 264–269, 2016.
5. *Hennessy J. L., Patterson D. A.* Computer architecture: A quantitative approach. Morgan Kaufmann, fourth edition, 2006.
6. *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 2016.
7. *Knuth D.* The Art of computer programming: Fundamental algorithms, Vol. 1. Addison-Wesley, 1973.
8. *Luccio F.* La struttura degli algoritmi. Boringhieri, 1982.
9. *Muthukrishnan S.* Data streams: Algorithms and applications // Foundations and Trends in Theoretical Computer Science, 1 (2): 117–236, 2005.
10. *Sanders P.* Algorithm engineering — an attempt at a definition // *Albers S., Alt H., Näher S.* Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, 5760, Springer, 321–323, 2009.
11. *Vitter J. S.* External memory algorithms and data structures // ACM Computing Surveys, 33 (2): 209–271, 2001.

Глава 2
РАЗМИНКА

> Все следует упрощать до тех пор, пока это возможно, но не более того.
>
> *Альберт Эйнштейн*

Рассмотрим простую на первый взгляд задачу, найти оптимальное решение которой совсем не легко.

> **Задача.** Даны показатели эффективности акций на Нью-Йоркской фондовой бирже (NYSE) в виде последовательности ежедневной разницы их котировок. Нужно определить лучшую стратегию купли-продажи этих акций, а именно пару дней (b, s) с максимальным доходом. Покупка акций происходит в начале дня b, а продажа — в конце дня s.

При всей простоте формулировки эта задача имеет много вариаций и приложений. Некоторые из них я прокомментирую в конце главы, пока же достаточно упомянуть, что я выбрал именно ее, так как она допускает последовательность алгоритмических решений возрастающей сложности и элегантности, позволяющих значительно уменьшить временную сложность. В конце мы получим алгоритм, линейно зависящий от числа n котировок акций, причем *оптимальный* по количеству шагов. Он учитывает каждое изменение цены акции и определяет, нужно ли включать его в оптимальное решение, ведь доход может принести даже однократное колебание цены. Удивительно, но наш алгоритм будет иметь простейшую схему доступа к памяти. Он однократно сканирует доступные котировки акций, то есть использует *потоковое поведение*, особенно полезное, когда из-за колебания котировок требуется на лету вычислять оптимальное временное окно. В главе 1 я уже упоминал, что такая алгоритмическая схема оптимальна с точки зрения процедуры ввода-вывода и *единообразна* на всех уровнях иерархии памяти. Фактически именно благодаря потоковому поведению количество вводов-выводов будет равняться n/B независимо от размера страницы диска B. Это типичная особенность *кэш-независимых алгоритмов* (cache-oblivious algorithms) [4], о которых пойдет речь в разделе 2.3.

Итак, рассмотрим случай с массивом $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Это котировки одной акции за 11 дней. Нетрудно сообразить, что результат покупки акции в начале дня x и ее продажи в конце дня y определяется как сумма

значений в подмассиве $D[x, y]$, или сумма всех колебаний ее курса. Для начала рассмотрим $x = 1$ и $y = 2$ и получим $+4 - 6 = -2$. То есть, купив акцию утром первого дня и продав ее в конце второго, мы потеряем два доллара. Обратите внимание на то, что начальная стоимость акций не имеет никакого значения. При поиске лучшего временного интервала для инвестиций важны только изменения ее стоимости. Это известная из литературы задача о *максимальной сумме подмассива*.

Обобщенная формулировка. Дан массив $D[1, n]$ положительных и отрицательных чисел. Найти подмассив $D[b, s]$ с самой большой суммой элементов.

Очевидно, что массив положительных элементов оптимален целиком. Цена акции все время растет, и нет причин продавать ее до последнего дня. Если, наоборот, все элементы массива отрицательны, имеет смысл выбрать элемент с наибольшим значением: если вы вынуждены торговать при постоянно снижающемся курсе, лучше покупать акции в день, когда они минимально теряют в цене, а затем быстро продать. Во всех остальных случаях расположение оптимального подмассива непонятно. В нашем примере оптимумом является подмассив $D[3, 7] = [+3, +1, +3, -2, +3]$, дающий доход 8 долларов. Как видите, в последовательность не попал день с максимальным ростом цены, то есть +6, кроме того, в ней присутствуют не только положительные значения. Найти оптимальный подмассив непросто, но, как ни удивительно, не очень сложно.

2.1. Алгоритм кубической сложности

Для начала рассмотрим неэффективное решение, просто чтобы перевести нашу задачу в псевдокод. В алгоритме 2.1 пара переменных $\langle b_0, s_0 \rangle$ обозначает подмассив, а сумма его значений хранится в переменной *MaxSum*. Начальное значение этой переменной равно $-\infty$, и при каждом выполнении шага 8 оно меняется. Ядро алгоритма составляют два вложенных цикла `for`, которые за шаги 2–3 проверяют все возможные подмассивы $D[b, s]$, вычисляя для каждого из них сумму элементов (шаги 4–7). Если во время шагов 8–9 обнаруживается сумма, превышающая текущее максимальное значение, то значение переменной *TmpSum* и соответствующие ей экстремумы подмассива сохраняются в *MaxSum* и $\langle b_0, s_0 \rangle$ соответственно.

Корректность алгоритма не вызывает сомнений. Он, как нам и требуется, проверяет все возможные подмассивы $D[1, n]$ и выбирает подмассив с максимальной суммой элементов (шаг 8). Покажем, что временная сложность этого алгоритма кубическая, — обозначим этот факт как $\Theta(n^3)$. Очевидно, что из элементов $\langle b, s \rangle$ можно сформировать не более чем $n^2/2$ пар, а верхняя граница стоимости вычисления суммы элементов каждого подмассива равна n[1] — таким образом, верхняя оценка

[1] Для каждой пары $\langle b_0, s_0 \rangle$ с $b \leqslant s$ возможно получить подмассив $D[b, s]$, но не подмассив $D[s, b]$.

сложности — $O(n^3)$. Теперь убедимся, что при наилучшем возможном сценарии зависимость тоже кубическая. Чтобы оценить нижнюю границу $\Omega(n^3)$, обратим внимание на то, что $D[1, n]$ состоит из $(n - L + 1)$ подмассивов длиной L, то есть затраты на вычисление суммы всех их элементов составят $(n - L + 1) \times L$. Суммирование по всем L даст точную временную сложность. Но нас интересует нижняя граница, так что оценку можно провести только для подмножества подмассивов длиной L в диапазоне $[n/4, n/2]$. То есть мы берем только $L \geqslant n/4$ и получаем $n - L + 1 > n/2$. В результате временные затраты составят $(n - L + 1) \times L > n^2/8$. Поскольку таких подмассивов у нас $n/2 - n/4 + 1 > n/4$, то общие временные затраты на анализ будут ограничены снизу[1] как $n^3/32 = \Omega(n^3)$.

Алгоритм 2.1. Алгоритм кубической сложности

```
 1: MaxSum = −∞;
 2: for (b = 1; b ⩽ n; b++) do
 3:     for (s = b; s ⩽ n; s++) do
 4:         TmpSum = 0;
 5:         for (i = b; i ⩽ s; i++) do
 6:             TmpSum + = D[i];
 7:         end for
 8:         if MaxSum < TmpSum then
 9:             MaxSum = TmpSum; b_0 = b; s_0 = s;
10:       end if
11:     end for
12: end for
13: return ⟨MaxSum, b_0, s_0⟩;
```

Так насколько же алгоритм 2.1 полезен с практической точки зрения? К сожалению, при переходе к очень большим наборам котировок акций он начинает работать крайне медленно.

2.2. Алгоритм квадратичного времени

Основную неэффективность алгоритму кубического времени 2.1 обеспечивают шаги 4–7. Именно здесь с нуля пересчитывается сумма элементов в подмассиве $D[b, s]$ при каждом изменении границы на шагах 2–3. Если внимательно рассмотреть

[1] Мы показали, что и верхняя граница сложности $O(n^3)$, и нижняя $\Omega(n^3)$ — кубические. Следовательно, какова бы ни была точная оценка, она тоже кубическая — $\Theta(n^3)$. — *Примеч. науч. ред.*

цикл `for` на шаге 3, нетрудно заметить, что размер s за один прогон цикла увеличивается на единицу — от значения b (одноэлементный подмассив) до значения n (самый длинный возможный подмассив, начинающийся с b). То есть при переходе к следующей итерации подлежащий суммированию подмассив меняется с $D[b, s]$ на $D[b, s + 1]$. Очевидно, что сумму для $D[b, s + 1]$ не нужно каждый раз вычислять с нуля, достаточно прибавить значение нового элемента $D[s + 1]$ к переменной *TmpSum*, в которой уже хранится результат суммирования $D[b, s]$. Именно такую схему реализует псевдокод алгоритма 2.2. Мы получили его из предыдущего алгоритма, отредактировав шаг 3. Теперь переменная *TmpSum* обнуляется при каждом изменении b, ведь очередной подмассив тоже начинается с длины 1, а именно с $D[b, b]$. Отредактирован в новой версии и шаг 5, реализующий инкрементное обновление суммы. Эти небольшие изменения экономят нам $\Theta(n)$ операций суммирования на шаге 2, снижая временную сложность алгоритма до $\Theta(n^2)$.

Алгоритм 2.2. Алгоритм квадратичного времени

```
1:  MaxSum = −∞;
2:  for (b = 1; b ⩽ n; b++) do
3:      TmpSum = 0;
4:      for (s = b; s ⩽ n; s++) do
5:          TmpSum + = D[s];
6:          if MaxSum < TmpSum then
7:              MaxSum = TmpSum; b0 = b; s0 = s;
8:          end if
9:      end for
10: end for
11: return ⟨MaxSum, b0, s0⟩;
```

Давайте точно подсчитаем количество операций сложения в алгоритме 2.2. Именно это позволит оценить общее количество шагов алгоритма. Оно вычисляется так[1]:

$$\sum_{b=1}^{n}\left(1+\sum_{s=b}^{n}1\right)=\sum_{b=1}^{n}\left(1+\left(n-b+1\right)\right)=n\left(n+2\right)-\sum_{b=1}^{n}b=n^2+2n-\frac{n\left(n-1\right)}{2}=O\left(n^2\right).$$

На практике улучшение становится заметным сразу же. Например, массив D размером $n = 10^3$ алгоритм 2.1, реализованный на языке Python, обрабатывает примерно за 17 с. Это данные для обычного ПК с процессором Intel i5. Выполнение

[1] Для суммы по b мы применили известную формулу сложения первых n положительных чисел, которую, согласно легенде, изобрел Гаусс, еще будучи школьником.

алгоритма 2.2 на той же машине занимает менее 1 с. Эта на первый взгляд небольшая разница становится значительной при размере массива $n = 10^4$. В этом случае алгоритм 2.1 работает примерно 17 000 с (почти в 10^3 раз дольше), в то время как алгоритм 2.2 дает результат где-то за 7 с. Это означает, что, в отличие от кубического алгоритма, квадратичный способен обработать большое количество элементов за разумное время. Понятно, что эти значения изменятся при переходе к другому языку программирования, другой операционной системе или на компьютер с другим процессором (в нашем примере мы использовали Python на компьютере под управлением MacOS с процессором Intel Core i5). Но они очень наглядно демонстрируют, что такое асимптотическое улучшение и насколько большую роль оно играет в реальной ситуации. Работа программиста нелегка, поскольку теоретически хорошие алгоритмы часто имеют так много скрытых особенностей, что их проектирование становится сложным, а нотация «большое O» далеко не всегда дает реалистичную оценку. Но не волнуйтесь, я постараюсь подробно рассмотреть все эти вопросы.

2.3. Алгоритм линейного времени

Ну и наконец, я хотел бы показать, что поиск подмассива с максимальной суммой можно реализовать с помощью элегантного алгоритма, который обрабатывает элементы $D[1, n]$ в потоковом режиме и занимает *оптимальное* время $O(n)$. И это лучший из возможных результатов.

Чтобы разработать такой алгоритм, нужно понимать конструктивные особенности оптимального подмассива. Для наглядности используем рис. 2.1, где предполагается, что оптимальный подмассив в диапазоне $[1, n]$ расположен между позициями $b_0 \leqslant s_0$.

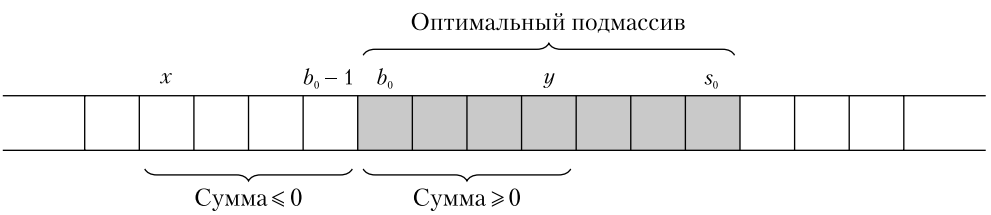


Рис. 2.1. Иллюстрация свойств 1 и 2

Рассмотрим подмассив, который начинается до b_0 и заканчивается в позиции $b_0 - 1$, например $D[x, b_0 - 1]$, где $x < b_0$. Сумма его элементов не может быть положительной, поскольку в противном случае его можно было бы объединить со смежным оптимальным подмассивом и таким образом получить более длинный подмассив $D[x, s_0]$, сумма элементов которого *превысила бы* сумму элементов заявленного оптимального подмассива $D[b_0, s_0]$. Таким образом, можно утверждать следующее.

Свойство 1. Сумма элементов подмассива $D[x, b_0 - 1]$, где $x < b_0$, не может быть строго положительной.

Аналогичным образом рассмотрим подмассив, с которого начинается оптимальный $D[b_0, s_0]$, то есть подмассив $D[b_0, y]$, где $y \leqslant s_0$. Сумма его элементов не может быть отрицательной, поскольку в противном случае его можно было бы исключить из оптимального решения, получив более короткий массив $D[y + 1, s_0]$, сумма элементов которого *превышает* сумму элементов заявленного оптимального подмассива $D[b_0, s_0]$. Это позволяет сформулировать второе свойство.

Свойство 2. Сумма элементов подмассива $D[b_0, y]$, где $y \leqslant s_0$, не может быть строго отрицательной.

Сумма элементов любого из рассматриваемых ранее подмассивов может оказаться равной нулю. От этого подмассив $D[b_0, s_0]$ не перестанет быть оптимальным, но могут появиться другие оптимальные решения большей или меньшей длины.

Чтобы проиллюстрировать эти свойства, снова используем массив $D[1, 11]$, который выглядел вот так: [+4, –6, +3, +1, +3, –2, +3, –4, +1, –9, +6]. Оптимальным в этом случае был подмассив $D[3, 7]$ = [+3, +1, +3, –2, +3]. Сумма элементов подмассива $D[x, 2]$ всегда отрицательна, как указано в свойстве 1. Для $x = 1$ сумма равна +4 – 6 = –2, а для $x = 2$ сумма равна –6. В то же время сумма всех элементов подмассива $D[3, y]$ положительна для всех префиксов оптимального подмассива, а именно для $y \leqslant 7$, как указано в свойстве 2. Более того, эта сумма положительна даже для некоторых $y > 7$. Например, суммы элементов подмассивов $D[3, 8]$ и $D[3, 9]$ равны 4 и 5 соответственно.

Эти два свойства приводят нас к простому линейному алгоритму 2.3. Он содержит всего один цикл `for` (шаг 3), который сохраняет в переменную *TmpSum* сумму подмассива, начинающегося с позиции b и заканчивающегося в рассматриваемой позиции s, где $b \leqslant s$. На каждой итерации цикла рассматриваемый подмассив расширяется на одну позицию вправо (операция `s++`), а к значению *TmpSum* прибавляется текущий элемент $D[s]$ (шаг 4). Поскольку рассматриваемый подмассив является кандидатом в оптимальные, результат суммирования его элементов сравнивается с текущим оптимальным значением (шаг 5). Затем мы используем свойство 1: если сумма элементов подмассива оказывается отрицательной, процесс перезапускается с новым подмассивом, начинающимся с позиции $b = s + 1$ (шаги 8–9). А в случае положительной суммы подмассив расширяется вправо, увеличивая s. Не так-то легко увидеть, что оптимальность подмассива проверяется на шаге 5, после чего сохраняется в $\langle b_0, s_0 \rangle$. Это совсем неинтуитивно, так как алгоритм проверяет n подмассивов из $\Theta(n^2)$ возможных, и я хочу показать, что это минимальное подмножество кандидатов на самом деле содержит оптимальное решение. Подмножество *минимально*, потому что подмассивы образуют разбиение $D[1, n]$, где каждый элемент принадлежит одному и только одному проверенному подмассиву. Более того, из-за необходимости анализировать каждый элемент нельзя отбросить ни один подмассив этого разбиения, не проверив его сумму.

Алгоритм 2.3. Алгоритм линейного времени

```
1:  MaxSum = −∞;
2:  TmpSum = 0; b = 1;
3:  for (s = 1; s ⩽ n; s++) do
4:      TmpSum + = D[s];
5:     if MaxSum < TmpSum then
6:         MaxSum = TmpSum; b_0 = b; s_0 = s;
7:      end if
8:      if TmpSum < 0 then
9:          TmpSum = 0; b = s + 1;
10:    end if
11: end for
12: return ⟨MaxSum, b_0, s_0⟩;
```

Прежде чем перейти к формальному доказательству корректности, проверим работу алгоритма на примере все того же массива $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Напомню, что оптимальный подмассив у нас $D[3, 7] = [+3, +1, +3, -2, +3]$. Поскольку сумма элементов подмассива $D[x, 2]$ отрицательна для $x = 1, 2$, при $s = 2$ алгоритм обнуляет переменную *TmpSum* на шагах 8–9. В это время b достигает значения 3, а переменная *TmpSum* становится равной 0. Последующее сканирование элементов $s = 3...7$ увеличивает значение *TmpSum*, ведь, как мы видели ранее, ее значение будет только положительным. При $s = 7$ рассматриваемый подмассив совпадает с оптимальным и переменная *TmpSum* достигает значения 8. Эти координаты на шаге 5 будут сохранены в $\langle b_0, s_0 \rangle$. Интересно отметить, что после перехода к следующей позиции $s = 8$ алгоритм не перезапустит значение *TmpSum*, поскольку оно все еще останется положительным, а именно *TmpSum* = 4. То есть он проверит более длинные подмассивы, но с меньшими суммами элементов. Перезапуск произойдет только в позиции $s = 10$, в которой *TmpSum* = –4.

Очевидно, что этот алгоритм имеет временную сложность $O(n)$, ведь каждый элемент проверяется только один раз. Сложнее показать корректность алгоритма в общей форме, то есть тот факт, что именно на шагах 4–5 происходят вычисление и проверка оптимальной суммы подмассива. Для этого нужно доказать две вещи:

- при $s = b_0 - 1$, на шаге 8 переменной b присваивается значение b_0;
- для всех последующих $s = b_0 ... s_0$ на шаге 8 такого присваивания не происходит, что и позволяет поместить в переменную *TmpSum* сумму всех элементов подмассива $D[b_0, s_0]$ при любых $s = s_0$.

Нетрудно увидеть, что первое доказательство вытекает из свойства 1, а второе — из свойства 2.

В сценарии из раздела 2.2 этот алгоритм работает очень быстро — обработка миллионов котировок акций занимает менее секунды. Так что это хорошо масштабируемый алгоритм со множеством приятных функций, которые делают его привлекательным в случае иерархической памяти. Фактически он сканирует массив D слева направо, по разу проверяя каждый из его элементов. Если D хранится на диске, элементы постранично извлекаются во внутреннюю память. То есть алгоритм выполняет n/B операций ввода-вывода, что оказывается *оптимальным*. Интересно также отметить следующее: алгоритм спроектирован таким образом, что никак не зависит от размера страницы B (это никак не проявляется в псевдокоде). Разрыв связи с таким параметром, как размер страницы, — ключевой момент разработки и анализа *кэш-независимых алгоритмов*. В рассматриваемом случае этот результат был достигнут за счет подхода на основе сканирования. Но в литературе [4] предлагаются более сложные варианты решения этой задачи.

2.4. Еще один алгоритм линейного времени

Для поиска подмассива с максимальной суммой элементов существует еще одно оптимальное решение. Обозначим сумму элементов подмассива $D[y', y'']$ как $\text{Sum}_D[y', y'']$. Рассмотрим все подмассивы, которые заканчиваются на времени продажи s. То есть нас интересуют подмассивы вида $D[x, s]$, где $x \leqslant s$. Значение $\text{Sum}_D[x, s]$ можно выразить как разницу между $\text{Sum}_D[1, s]$ и $\text{Sum}_D[1, x-1]$. Это две префиксные суммы по массиву D, допускающие вычисление за линейное время. В результате нашу задачу можно записать так:

$$\max_s \max_{b \leq s} \text{Sum}_D[b, s] = \max_s \max_{b \leq s} \left(\text{Sum}_D[1, s] - \text{Sum}_D[1, b-1]\right).$$

При $b = 1$ второй член относится к пустому подмассиву $D[1, 0]$, поэтому можно предположить, что $\text{Sum}_D[1, 0] = 0$. Это тот случай, когда $D[1, s]$ является подмассивом с максимальной суммой среди всех подмассивов, заканчивающихся на элементе s, поэтому ни один префиксный подмассив $D[1, b-1]$ из него не удаляется.

Следующий шаг — предварительное вычисление всех префиксных сумм $P[i] = \text{Sum}_D[1, i]$. Эта операция выполняется путем сканирования массива D и имеет временную и пространственную сложность $O(n)$. Пусть $P[i] = P[i-1] + D[i]$, где мы устанавливаем $P[0] = 0$, чтобы учесть рассматриваемый выше частный случай. Теперь задачу поиска подмассива с максимальной суммой вместо $\text{Sum}_D[1, s] - \text{Sum}_D[1, b-1]$ можно записать так: $P[s] - P[b-1]$. Это позволит разложить вычисление $\max_s \max_{b \leq s}$ на вычисление минимума/максимума следующим образом:

$$\max_s \max_{b \leq s} \left(P[s] - P[b-1]\right) = \max_s \left(P[s] - \min_{b \leq s} P[b-1]\right).$$

Фактически $P[s]$ как не зависящее от переменной b можно вынести за пределы внутреннего вычисления максимума, а затем изменить максимум на минимум из-за отрицательного знака. После этого остается предварительно вычислить минимум

$\min_{b \leqslant s} P[b-1]$ для всех s и сохранить его в массиве $M[0, n-1]$. Обратите внимание: в этом случае вычисление $M[i]$ также имеет временную и пространственную сложность $O(n)$, так как может быть выполнено простым сканированием P. Мы присвоим $M[0]$ значение 0 и установим $M[i]$ как $\min\{M[i-1], P[i]\}$. Все это позволит переписать предыдущую формулу как:

$$\max_s\Big(P[s]-\min_{b\leq s}P[b-1]\Big)=\max_s\big(P[s]-M[s-1]\big).$$

Для массивов P и M ее вычисление имеет временную сложность $O(n)$, как и у алгоритма 2.3, но данному алгоритму требуется дополнительное пространство $\Theta(n)$.

В качестве примера снова рассмотрим массив $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Посчитаем массив префиксной суммы $P[0, 11] = [0, +4, -2, +1, +2, +5, +3, +6, +2, +3, -6, 0]$ и минимальный массив $M[0, 10] = [0, 0, -2, -2, -2, -2, -2, -2, -2, -2, -6]$. После нахождения $P[s] - M[s-1]$ для всех $s = 1... n$ получим последовательность значений $[+4, -2, +3, +4, +7, +5, +8, +4, +5, -4, +6]$, максимум которой равен +8. Этот максимум находится в корректной конечной позиции $s = 7$. Интересно отметить, что левое крайнее значение b_0 оптимального подмассива можно вывести, найдя позицию $b_0 - 1$, где $P[b_0 - 1]$ является минимумом — в этом примере $P[2] = -2$ и, таким образом, $b_0 = 3$.

Реализация этой идеи показана в алгоритме 2.4, причем там используется хитрый трюк, позволяющий обойтись однократным сканированием и задействовать дополнительное пространство всего лишь $O(1)$, как и для алгоритма 2.3. Мы используем ассоциативность функций min/max и задействуем две переменные, которые на каждой итерации сохраняют значения $P[s]$, то есть *TmpSum*, и $M[s-1]$, то есть *MinTmpSum*, поскольку массив D сканируется слева направо. Таким образом, формула $\max_s(P[s] - M[s-1])$ инкрементно вычисляется для $s = 1... n$.

Алгоритм 2.4. Еще один алгоритм линейного времени

```
1:  MaxSum = −∞; b_tmp = 1;
2:  TmpSum = 0; MinTmpSum = 0;
3:  for (s = 1; s ⩽ n; s++) do
4:      TmpSum + = D[s];
5:      if MaxSum < TmpSum − MinTmpSum then
6:          MaxSum = TmpSum − MinTmpSum; s_0 = s; b_0 = b_tmp;
7:      end if
8:      if TmpSum < MinTmpSum then
9:          MinTmpSum = TmpSum; b_tmp = s + 1;
10:     end if
11: end for
12: return ⟨MaxSum, b_0, s_0⟩;
```

2.5. Несколько интересных вариантов∞

А теперь, как я и обещал в начале главы, обсудим несколько интересных вариантов задачи по поиску подмассива с максимальной суммой элементов. Прочие детали построения алгоритмов вы найдете в [1] и [2]. А сейчас будет сложный раздел, где мы рассмотрим алгоритм, конструирование и анализ которого требуют специальных знаний. Я обозначил его символом ∞. Он и далее будет применяться для маркировки таких разделов.

В литературе по биоинформатике вместо термина «подмассив» используется термин «сегмент», и задача состоит в том, чтобы идентифицировать внутри ДНК-последовательностей строки, составленные из стандартного генетического четырехбуквенного алфавита {A, T, G, C}, где A — аденин, T — тимин, G — гуанин, C — цитозин) сегменты с высоким содержанием нуклеотидов гуанина (G) и цитозина (C). В биологии считается, что именно такие сегменты преимущественно содержат генетическую информацию. Превратить ДНК-последовательности в *массивы чисел*, а потом перейти к абстрактной формулировке задачи можно несколькими способами в зависимости от целевой функции, моделирующей *GC-богатство* сегмента. Вот два интересных варианта.

- Назначим нуклеотидам A и T штраф $-p$, а нуклеотидам C и G — вознаграждение $1 - p$. В этом случае для сегмента длиной l, содержащего x вхождений нуклеотидов C или G, сумма будет равна $x - p \times l$. Что интересно, при такой целевой функции для поиска в ДНК-последовательности CG-богатых сегментов можно использовать любой из описанных ранее алгоритмов. Биологи часто указывают диапазон отсечки по длине сегментов, для которых ищется максимальная сумма, тем самым убирая из рассмотрения слишком короткие или слишком длинные сегменты. В этом случае алгоритмы из предыдущих разделов уже не годятся, но существуют другие оптимальные решения, сходящиеся за линейное время (см., например, [2]).
- Присвоим нуклеотидам A и T значение 0, а нуклеотидам C и G — значение 1. В этом случае в сегменте длиной l, содержащем x вхождений нуклеотидов C или G, *плотность* этих нуклеотидов будет равна x/l. Очевидно, что $0 \leqslant x/l \leqslant 1$, а максимальная плотность достигается как минимум в каждом единичном сегменте, состоящем из нуклеотида C или G. Биологи считают это единицей измерения CG-богатства сегмента в ситуации, когда на длину сегментов накладываются ограничения. Такая задача сложнее упомянутой в предыдущем пункте, тем не менее она тоже имеет оптимальные квазилинейные по времени решения. В силу их чрезвычайной сложности они не будут рассматриваться в этой книге, но заинтересованные читатели могут самостоятельно ознакомиться с ними, например, в [1], [3], [5].

Эти два примера наглядно демонстрируют *опасную ловушку*, подстерегающую нас при попытках сформулировать реальную задачу в абстрактном виде: даже небольшие изменения в формулировке могут привести к резкому усложнению проектирования эффективных алгоритмов. Например, для функции плотности, которую мы ввели

во втором случае, необходимо дополнительно ограничивать снизу длину сегмента. Без этого дополнительного параметра невозможно было бы избежать тривиального решения, состоящего из *одиночных* нуклеотидов C или G. Но это небольшое изменение резко усложнило решение задачи.

Есть и другие тонкости, причем менее очевидные. Предположим, что результата в виде одиночных нуклеотидов мы пытаемся избежать, отыскивая *самый длинный* сегмент, плотность которого x/l *не меньше* фиксированного значения t. Фактически мы еще сильнее усложнили задачу, ведь теперь ищется сегмент максимальной длины при ограничениях, наложенных на значение плотности. Удивительно, но такую задачу можно свести к задаче из первого пункта этого раздела, аналогичной уже решенным в предыдущих разделах. Это распространенная практика, хорошо экономящая время, так как проще взять уже известное решение, чем с нуля изобретать велосипед.

Чтобы доказать возможность такой редукции, достаточно заметить, что для любого подмассива $D[a, b]$:

$$\frac{\mathrm{sum}_D[a,b]}{b-a+1} = \sum_{k=a}^{b} \frac{D[k]}{b-a+1} \geq t \Leftrightarrow \sum_{k=a}^{b} \big(D[k]-t\big) \geq 0.$$

То есть, вычитая пороговое значение плотности t из всех элементов массива D, мы переходим от задачи, базирующейся на таком параметре, как плотность, к задаче по поиску *самого длинного сегмента, сумма элементов которого больше или равна* 0.

> **Задача.** Есть массив $D[1, n]$ положительных и отрицательных чисел. Требуется найти в нем самый длинный сегмент с плотностью, *превышающей* фиксированное пороговое значение t *или равной* ему.

Имейте в виду, что, если поменять запрос с *самого длинного* сегмента на *самый короткий* с плотностью, превышающей t, задача снова станет тривиальной, так как ее решением будут единичные вхождения нуклеотида C или G. Если же вместо нижней границы зафиксировать для суммы элементов сегмента верхнюю границу u, достаточно будет изменить знак всех элементов в D, чтобы вернуться к задаче с нижней границей $t = -u$. Наконец, отмечу, что формулировка этой задачи в некотором смысле является дополнением к формулировке задачи из первого пункта. Здесь мы максимизируем длину сегмента и принудительно устанавливаем нижнюю границу для суммы его элементов, а там максимизировали сумму элементов сегмента при условии, что его длина находится в заданном диапазоне. Приятно отметить, что обе задачи имеют сходную структуру алгоритмического решения, поэтому подробно рассмотрим только первую. Решение второй задачи вы сможете найти самостоятельно в литературе.

Алгоритм решения этой задачи должен на каждой итерации $i = 1, 2... n$ искать в диапазоне $D[1, i-1]$ самый длинный подмассив, сумма элементов которого превышает t. Обозначим решение на шаге i как $D[l_{i-1}, r_{i-1}]$. Изначально $i = 1$. В этом случае мы получаем пустое решение, то есть длину 0. При переходе к шагу $i + 1$ нужно вычислить $D[l_i, r_i]$. У нас уже есть известно решение, которым можно воспользоваться, что дает хорошее преимущество.

Очевидно, что новый сегмент или находится внутри диапазона $D[1, i-1]$ (при $r_i < i$), или заканчивается в позиции $D[i]$ (при $r_i = i$). В первом случае можно взять одно из решений, полученных на предыдущей итерации, а именно $D[l_{i-1}, r_{i-1}]$. Тут ничего делать не нужно, достаточно установить $r_i = r_{i-1}$ и $l_i = l_{i-1}$. Второй случай менее тривиален, и, чтобы показать, что предлагаемый алгоритм асимптотически оптимален, так как имеет временную и пространственную сложность $O(n)$, потребуется прибегнуть к специальным структурам данных.

Для начала отметим простую, но впечатляющую вещь.

Факт 2.1. При $r_i = i$ отрезок $D[l_i, r_i]$ должен быть строго длиннее отрезка $D[l_{i-1}, r_{i-1}]$. В частности, это означает, что l_i появляется левее позиции $L_i = i - (r_{i-1} - l_{i-1})$.

Этот факт следует из того, что если $r_i = i$, то на шаге i найден сегмент самой большой длины. Поэтому можно отбросить все позиции в диапазоне $[L_i, i]$, так как там фигурируют сегменты, чья длина меньше предыдущего решения $D[l_{i-1}, r_{i-1}]$ или равна ему.

> **Переформулированная задача.** В массиве $D[1, n]$ положительных и отрицательных чисел на каждом шаге найти *наименьший* индекс $l_i \in [1, L_i)$, подходящий под условие $Sum_D[l_i, i] \geqslant t$.

Индексов l_i, подходящих под условие $Sum_D[l_i, i] \geqslant t$, может оказаться больше одного, и среди них требуется определить *самый маленький*, поскольку мы ищем самый *длинный сегмент*.

На этом этапе полезно вспомнить, что сумму $Sum_D[l_i, i]$ можно переписать в терминах префиксных сумм массива D, а именно $Sum_D[1, i] - Sum_D[1, l_i - 1] = P[i] - P[l_i - 1]$. Массив P был введен в рассмотрение в разделе 2.4 и предварительно вычислен с линейной сложностью по времени и пространству. Поэтому остается найти наименьший индекс $l_i \in [1, L_i)$, подходящий под условие $P[i] - P[l_i - 1] \geqslant t$.

Надо отметить, что поиск l_i можно выполнить путем сканирования массива $P[1, L_i - 1]$ на предмет *самого левого* индекса x, подходящего под условие $P[i] - P[x] \geqslant t$ или эквивалентное условие $P[x] \leqslant P[i] - t$. После этого останется выполнить присваивание

$l_i = x + 1$, и на этом процесс завершается. К сожалению, это неэффективный подход, поскольку на каждой итерации i требуется заново сканировать все элементы массива P, что приводит к квадратичному алгоритму, в то время как наша цель — линейный алгоритм.

Чтобы избежать сканирования всего префикса $P[1, L_i - 1]$, нужно определить *подмножество позиций-кандидатов* для x. Обозначим такую позицию для итерации i как $C_{i,j}$, где $j = 0, 1...$ Эти позиции задаются следующим образом: $C_{i,0} = L_i$ (это фиктивное значение), а $C_{i,j}$ определяется на каждой итерации как *самый левый минимум* подмассива $P[1, C_{i,j-1} - 1]$. Этот подмассив находится слева от текущего минимума и слева от L_i. Обозначим количество позиций-кандидатов на шаге i как $c(i)$, где $c(i) \leqslant L_i$ (равенство здесь появляется при убывании $P[1, L_i]$).

Для наглядности я добавил рис. 2.2, на котором $c(i) = 3$, а позиции-кандидаты соединены стрелками.

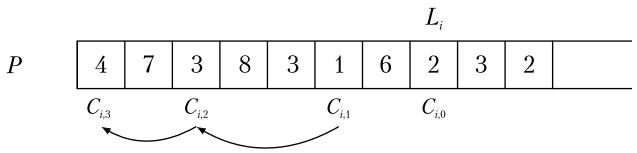


Рис. 2.2. Позиции-кандидаты $C_{i,j}$ относительно позиции L_i в массиве префиксных сумм P

Из рис. 2.2 можно вывести три ключевых свойства, доказательство которых я оставляю вам для самостоятельной работы, поскольку они вытекают непосредственно из определения $C_{i,j}$.

- **Свойство 1.** Последовательность позиций-кандидатов $C_{i,j}$ находится в диапазоне $[1, L_i)$ и прирастает слева, то есть $C_{i,j} < C_{i,j-1} < ... < C_{i,1} < C_{i,0} = L_i$, где $j = c(i)$.
- **Свойство 2.** На каждой итерации i последовательность значений позиций-кандидатов $P[C_{i,j}]$ возрастает при $j = 1, 2, ..., c(i)$. Точнее, $P[C_{i,j}] > P[C_{i,j-1}]$, а $C_{i,j} < C_{i,j-1}$, то есть значения возрастают по мере сдвига позиции $C_{i,j}$ влево.
- **Свойство 3.** Значение $P[C_{i,j}]$ меньше, чем у любого элемента, расположенного слева от него в массиве P, так как это самый левый минимум префиксного массива $P[1, C_{i,j-1} - 1]$.

Теперь важно показать, что для поиска нужного нам индекса l_i достаточно рассмотреть только эти позиции-кандидаты $C_{i,j}$ и соответствующие значения префиксной суммы $P[C_{i,j}]$. Из факта 2.1 следует, что нас интересуют сегменты вида $D[x, i]$, где $x < L_i$ и сумма $Sum_D[x, i] \geqslant t$. Среди этих x мы и будем искать наименьший, то есть l_i. Поскольку $Sum_D[x, i] = P[i] - P[x - 1]$, свойство 3 позволяет заключить, что при $Sum_D[C_{i,j} + 1, i] < t$ все более длинные сегменты будут иметь сумму меньше t. Отсюда вытекает факт 2.2.

Факт 2.2. На каждой итерации i самый длинный сегмент будет иметь наибольший индекс j^*, удовлетворяющий условию $Sum_D[C_{i,j^*} + 1, i] \geqslant t$ (если таковой существует).

Для поиска позиций-кандидатов $C_{i,j}$, необходимых для вычисления l_i, нужно понять две вещи: как вычислять $C_{i,j}$ по мере увеличения i и как искать индекс j^*. К счастью, вычисление $C_{i,j}$ зависит *не* от индексов i или j, а только от позиции предыдущего кандидата $C_{i,j-1}$. Поэтому можно задать вспомогательный массив $LMin[1, n]$ таким образом, чтобы $LMin[i]$ была самой левой позицией минимума в диапазоне $P[1, i-1]$. Нетрудно увидеть, что $C_{i,1} = LMin[L_i]$, а согласно определению позиций-кандидатов $C_{i,2} = LMin[LMin[L_i]]$, то есть $LMin^2[L_i]$. В общем случае мы получим $C_{i,k} = LMin^k[L_i]$. Теперь можно выполнить инкрементное вычисление:

$$LMin[x] = \begin{cases} 0, \text{ если } x = 0; \\ x-1, \text{ если } P[x-1] < P[LMin[x-1]]; \\ LMin[x-1] \text{ в остальных случаях.} \end{cases}$$

Эта формула легко объясняется с помощью *индукции*. Сначала мы присваиваем функции $LMin[0]$ фиктивное значение 0. Для нахождения $LMin[x]$ нужно определить самый левый минимум в диапазоне $P[1, x-1]$. Он может быть равен $x-1$ (со значением $P[x-1]$) или в позиции $LMin[x-1]$ определен для $P[1, x-2]$ (со значением $P[L\min[x-1]]$). Сравнение этих двух значений позволяет вычислить $LMin[x]$ за постоянное время. Следовательно, вычисление всех позиций-кандидатов $LMin[1, n]$ занимает время $O(n)$.

Остается проблема эффективного определения индекса j^*. Мы не сможем на каждой итерации i вычислять его за постоянное время, но можно показать, что выполнение $s_i > 1$ шагов увеличивает длину текущего решения на $\Theta(s_i)$ единиц. Поскольку самый длинный сегмент не может быть длиннее n, получится, что сумма дополнительных затрат не может быть больше $O(n)$, что и требуется доказать. Это называется *амортизационным подходом*[1], потому что мы в некотором смысле взимаем стоимость дорогих итераций с самых дешевых. Вычисление j^* на итерации i требует проверки $LMin^k[L_i]$ позиций для $k = 1, 2...$ Проверка осуществляется до выполнения условия из факта 2.2. При этом мы знаем, что все $j > j^*$ не удовлетворяют этому условию. Так что поиск занимает j^* шагов и находит новый сегмент, длина которого, учитывая свойство 1, *увеличивается* по крайней мере на j^* единиц. Сегмент не может быть длиннее всей последовательности $D[1, n]$, так что можно сделать вывод, что общее дополнительное время, затраченное на поиск j^*, не может быть больше $O(n)$.

Псевдокод для этого алгоритма я оставляю вам в качестве самостоятельного упражнения, благо методы, лежащие в основе его элегантной конструкции и анализа, достаточно просты для того, чтобы к ним можно было подойти без каких-либо затруднений.

---

[1] Амортизационный анализ вычислительной сложности применяется, если традиционный подход — умножить сложность наихудшего случая на количество итераций — дает неоправданно большое значение. В нашем случае вычисление j^* на каждом шаге требует неконстантного времени и утверждение о линейности общего поиска j^* требует более детального анализа, который приведен ниже. — *Примеч. науч. ред.*

Список литературы

1. *Chao K.-M.* Maximum-density segment // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 502–504, 2008.
2. *Chao K.-M.* Maximum-scoring segment with length restrictions // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 506–7, 2008.
3. *Cheng C.-H., Liu H.-F., Chao K.-M.* Optimal algorithms for the average-constrained maximum-sum segment problem // Information Processing Letters, 109 (3): 171–174, 2009.
4. *Fagerberg R.* Cache-oblivious model // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 264–269, 2016.
5. *Fukuda T., Morimoto Y., Morishita S., Tokuyama T.* Mining optimized association rules for numeric attributes // Journal of Computer System Sciences, 58 (1): 1–12, 1999.

Глава 3
СЛУЧАЙНАЯ ВЫБОРКА

> Мне кажется, очень многое в жизни зависит
> от случайностей.
>
> *Сидни Пуатье*

Задача с очень простой формулировкой, которую мы рассмотрим в этой главе, лежит в основе множества алгоритмов с использованием случайных чисел и допускает сложные для алгоритмического проектирования и анализа решения.

Задача. Даны последовательность из n элементов $S = (i_1, i_2, ..., i_n)$ и положительное целое число $m \leqslant n$. Нужно равномерным и случайным образом выбрать из последовательности S подмножество, состоящее из m элементов.

Равномерность здесь означает, что вероятность выбора любого элемента составляет $1/n$. В качестве элементов могут выступать числа, строки и даже сложные объекты, которые либо хранятся на диске, либо передаются по сети. В первом варианте размер входных данных n известен, а элементы занимают непрерывную последовательность страниц. Во втором размер входных данных может быть неизвестен, но гарантирована *однородность* процесса выборки. Мы рассмотрим оба сценария, стремясь к эффективности с точки зрения ввода-вывода, дополнительного пространства для вычислений в дополнение к вводу и объема используемой случайности, выраженного через количество случайно сгенерированных целых чисел. Мы прибегнем к встроенной процедуре `RAND`(a, b), которая случайным образом выбирает число из диапазона (a, b]. Действительное это или целое число, будет понятно из контекста. Проектирование хорошей функции `RAND` — непростая задача, и мы ее рассматривать не будем, поскольку хотим сосредоточиться на процессе выборки, а не на деталях генерации случайных чисел. Если вам интересна эта тема, массу материала можно найти в литературе по генераторам псевдослучайных и случайных чисел.

Напоследок отмечу: желательно, чтобы выборка производилась последовательно (то есть чтобы *позиции* — индексы выбираемых элементов — были упорядочены); это ускорит их извлечение из S как в случае хранения на диске (меньше время поиска), так и при обработке потока (меньше проходов по данным). Более того, это уменьшит и занимаемую память, поскольку позволит эффективно извлекать элементы путем сканирования, а не с помощью вспомогательного массива указателей на

выбранные элементы. Я не буду подробно останавливаться на проблеме сортировки, которая усложняется всякий раз при $m > M$, когда наши элементы не помещаются во внутреннюю память. В этом случае требуется сортировщик на самом диске. Это сложная тема, которая будет рассмотрена в главе 5. А вот когда $m \leqslant M$, можно воспользоваться тем, что позиции являются целыми числами в фиксированном диапазоне, и прибегнуть к поразрядной сортировке или любой другой быстрой процедуре сортировки во внутренней памяти.

3.1. Дисковая модель и известная длина последовательности

Предположим, нам известен размер входных данных n, причем $S[1, n]$ хранится в непрерывном наборе страниц на диске. Менять эти страницы нельзя, поскольку они могут быть входными данными более сложной задачи, включающей текущую в качестве подзадачи. Алгоритм 3.1 — это первое решение, которое я хочу вам предложить. Оно очень простое и потому имеет ряд важных недостатков, с которыми мы будем разбираться в последующих решениях.

Алгоритм 3.1. Извлечение невыбранных элементов

1: Инициализация вспомогательного массива $S'[1, n] = S[1, n]$;
2: **for** $s = 0, 1, ..., m - 1$ **do**
3: p = `RAND`$(1, n - s)$;
4: добавляем в выборку элемент, на который в данный момент указывает $S'[p]$;
5: меняем местами $S'[p]$ и $S'[n - s]$;
6: **end for**

На каждом шаге s алгоритм сохраняет следующий инвариант: подмассив $S'[n - s + 1, n]$ *с уже выбранными элементами, в то время как невыбранные элементы массива* S *содержатся в подмассиве* $S'[1, n - s]$. Изначально (при $s = 0$) этот инвариант выполняется, поскольку $S'[n - s + 1, n] = S'[n + 1, n]$ — это пустой массив. На шаге s алгоритм случайным образом выбирает элемент из последовательности $S'[1, n - s]$ и меняет его местами с последним элементом последовательности $S'[n - s]$. Инвариант выполняется и для $s + 1$. В конце (при $s = m$) выбранные элементы оказываются в массиве $S'[n - m + 1, n]$.

Точной копией массива S массив S' быть не может — он должен быть реализован как *массив указателей* на элементы S. Дело в том, что эти элементы могут иметь переменную длину, например, если они представляют собой строки или сложные объекты. В этом случае извлечение за постоянное время нельзя реализовать с помощью арифметических операций. Более того, иногда операция замены оказывается невозможной из-за разницы в длине между элементами в $S'[p]$ и $S'[n - s]$. Указатели позволяют избежать этих проблем, но занимают дополнительное пространство $\Theta(n \log n)$ бит. При больших n это становится существенным. Более того, если

средняя длина элементов массива S короче чем $\log n$ бит, они могут занять больше места, чем есть в S. Другой недостаток этого подхода связан с его шаблоном доступа к памяти. У нас есть $\Theta(n)$ ячеек, к которым мы обращаемся в совершенно случайном порядке и соответственно выполняем $\Theta(m)$ операций ввода-вывода. При $m \approx n$ это может оказаться слишком медленным. В этом случае мы хотели бы иметь $O(n/B)$ операций ввода-вывода, что является стоимостью сканирования всего массива S.

Эти недостатки я буду исправлять, предлагая алгоритмы, которые постепенно улучшают такие параметры, как количество вводов-выводов и размер используемого пространства. Итогом станет результат с дополнительным пространством $O(m)$, ожидаемым временем $O(m)$ и $O(\min\{m, n/B\})$ вводов-выводов. Для начала обратите внимание на то, что на шаге 5 алгоритма 3.1 происходит не только генерация отдельного элемента, но и дублирование массива S. Поэтому давайте посмотрим на алгоритм 3.2, который уменьшает пространственную сложность и количество операций ввода-вывода с помощью вспомогательной структуры данных — в ней в отсортированном виде хранятся позиции выбранных элементов, а пространства в памяти она занимает всего $O(m)$.

Алгоритм 3.2. Словарь выбранных позиций

1: Инициализируем словарь $\mathcal{D} = \emptyset$;
2: **while** $|\mathcal{D}| < m$ **do**
3: $p = \texttt{RAND}(1, n)$;
4: **if** $p \notin \mathcal{D}$, вставляем эту позицию в $\mathcal{D}$;
5: **end while**

Алгоритм 3.2 останавливается, когда в словаре[1] $\mathcal{D}$ оказывается m (различных) целых чисел — позиций элементов для выборки. Эффективность алгоритма в основном зависит от реализации словаря $\mathcal{D}$, который позволяет обнаруживать дублирующиеся позиции, а следовательно, дубликаты элементов. В литературе предлагается множество структур данных, которые эффективно поддерживают операции проверки принадлежности и вставки, базирующихся на хешировании или деревьях. Мы рассмотрим только решение, в котором словарь $\mathcal{D}$ реализован в виде хеш-таблицы размером $\Theta(m)$ с разрешением коллизий с помощью цепочек, а также универсальной хеш-функции для доступа к этой таблице (подробнее о хешировании поговорим в главе 8). Таким образом, каждый запрос принадлежности и операция вставки в словарь $\mathcal{D}$ займут ожидаемое время $O(1)$ (хеш-таблицы имеют константный коэффициент заполнения $O(1)$) и общее пространство $O(m)$. Временную сложность можно улучшить, воспользовавшись динамическим идеальным хешированием или кукушкиным хешированием, но оценку по времени можно производить только

[1] Здесь термин «словарь» не имеет прямого отношения к одноименным встроенным типам данных (в Python, JavaScript и т. д.) — например, автор рассчитывает на то, что элементы такого словаря упорядочены. — *Примеч. науч. пер.*

в терминах *математического ожидания*, так как мы имеем дело с повторной выборкой. Дело в том, что наш алгоритм может генерировать *одинаковые* позиции, которые следует игнорировать и *снова производить выборку*. В оценку работы такого подхода следует включать затраты на повторную выборку, и в этом его недостаток: повторные выборки замедляют ожидаемое время в несколько раз. На практике это весьма привлекательное решение — для малых m. Дело в том, что вероятность выбрать элемент, уже имеющийся в словаре $\mathcal{D}$, составляет $|\mathcal{D}|/n \leqslant m/n < 1/2$, и для $m << n$ это неплохой показатель. Без потери общности можно предположить, что $m < n/2$. Если это не так, мы можем решать *дополнение* текущей задачи: случайным образом отмечать позиции элементов, *не* выбранных из S. Таким образом, вероятность, что нам понадобится повторная выборка, не превосходит $1/2$, то есть математическое ожидание количества повторных выборок перед тем, как мы получим новый элемент для словаря $\mathcal{D}$, — константа. В целом мы доказали следующее.

Факт 3.1. Базирующемуся на хешировании цепочками алгоритму 3.2 для равномерного выбора случайных m позиций из набора $[1, n]$ требуется ожидаемое время $O(m)$ и дополнительное пространство $O(m)$. Если мы хотим извлекать элементы из S в потоковом режиме, необходим дополнительный шаг для сортировки. В этом случае общий процесс выборки занимает $O(\min\{m, n/B\})$ операций ввода-вывода.

Если заменить хеширование сбалансированным деревом поиска и предположить, что мы работаем в модели RAM (следовательно, $m < M$), то мы сможем избежать этапа сортировки, выполнив обход дерева за время $O(m)$. Однако оценка времени работы алгоритма 3.2 все равно будет $O(m \log m)$, поскольку каждая операция вставки/проверки принадлежности займет время $O(\log m)$. Можно было бы добиться большего, развернув *целочисленную* словарную структуру данных, такую как дерево ван Эмде Боаса [1]. В этом случае временные затраты на каждую операцию со словарем, которая выполняется на целочисленных ключах по $\Theta(\log n)$ бит каждый, составили бы $O(\log \log n)$. Обратите внимание: эти две оценки несопоставимы, так как многое зависит от отношения m и n. Возможны и другие компромиссы, достичь которых можно изменением базовой структуры данных словаря, — я оставлю это вам в качестве упражнения. В заключение напомню, что при $m \leqslant M$ генерация случайных чисел и управление словарем $\mathcal{D}$ могут выполняться в основной памяти без каких-либо операций ввода-вывода. Иногда это полезно, поскольку вероятностный алгоритм, который вызывает подпрограмму случайной выборки, интересуют не соответствующие элементы, а только их позиции в массиве S.

3.2. Потоковая модель и известная длина последовательности

Рассмотрим случай, когда массив S необходимо получить через канал передачи данных, а размер входных данных n известен и велик (например, это могут быть интернет-трафик или журналы запросов). В такой модели предварительная обработка невозможна. То есть мы не можем поступить как в разделе 3.1, где позиции элементов можно было повторно выбрать и/или отсортировать. Теперь каждый

элемент массива S рассматривается всего один раз, а алгоритм должен немедленно и бесповоротно принять решение, следует ли включать этот элемент в набор. Более поздние элементы могут выкидывать из набора ранее попавшие туда элементы, но ни один выброшенный не может быть рассмотрен повторно. Даже в этом случае алгоритмы имеют довольно простую конструкцию, а вот их вероятностный анализ будет немного сложнее. В предыдущем разделе мы рассматривали алгоритмы с *ожидаемой* временной сложностью, так как там приходилось иметь дело с проблемой повторной выборки — некоторые образцы приходилось исключать, поскольку они дублировались. Избежать повторной выборки можно, гарантировав, что каждый элемент рассматривается всего один раз. Простейшим образом реализуют эту идею алгоритмы, которые сканируют входную последовательность S, однократно рассматривая каждый ее элемент. Здесь нас поджидают две основные трудности, связанные с такими требованиями, как равномерная выборка из диапазона $[1, n]$ и возврат выборки размером m.

Для начала рассмотрим алгоритм, извлекающий из S только один элемент (следовательно, $m = 1$). Затем обобщим его на случай подмножества из $m > 1$ элементов. Этот алгоритм выбирает элемент $S[j]$ с вероятностью $\mathcal{P}(j)$, которая определена таким образом, чтобы гарантировать равномерный случайный выбор одного элемента[1]. В частности, мы устанавливаем $\mathcal{P}(1) = 1/n$, $\mathcal{P}(2) = 1/(n-1)$, $\mathcal{P}(3) = 1/(n-2)$ и т. д., и когда алгоритм выбирает элемент j — с вероятностью $\mathcal{P}(j) = 1/(n-j+1)$, — он останавливается. В конце выбирается элемент $S[n]$, потому что вероятность его извлечения $\mathcal{P}(n) = 1$.

Итак, предложенный алгоритм обеспечивает выполнение условия для выборки размером $m = 1$. То, что вероятность отбора произвольного элемента $S[j]$ составляет $1/n$ для любого j, доказать сложнее, ведь мы определили $\mathcal{P}(j) = 1/(n-j+1)$. В данном случае срабатывает простой вероятностный аргумент, ведь $n-j+1$ — это количество элементов, оставшихся в последовательности, и все они имеют равную вероятность быть выбранными. По индукции первые $j-1$ элементов последовательности имеют равную вероятность $1/n$ быть выбранными, поэтому вероятность того, что ни один из них не будет выбран, равна $1-(j-1)/n$. В результате получим вот такую формулу:

$$\mathcal{P}(\text{отбор } i_j) = \mathcal{P}(\text{не выбор } i_1 \ldots i_{j-1})\mathcal{P}(\text{выбор } i_j) = \\ = \left(1 - \frac{j-1}{n}\right) \times \frac{1}{n-j+1} = \frac{1}{n}.$$

Алгоритм 3.3 демонстрирует псевдокод для этого подхода, обобщенный для работы с произвольно большой выборкой $m \geqslant 1$.

---

[1] Для выбора элемента с вероятностью $\mathcal{P}$ достаточно сгенерировать случайное действительное число $p = \text{RAND}(0, 1) \in (0, 1]$ и сравнить его с $\mathcal{P}$. Если $p \leqslant \mathcal{P}$, элемент выбран, в противном случае — нет. Тогда вне зависимости от p элемент никогда не будет выбран при $\mathcal{P} = 0$ и гарантированно выбран при $\mathcal{P} = 1$.

Алгоритм 3.3. Сканирование и выбор

```
1: s = 0;
2: for (j = 1; (j ⩽ n) and (s < m); j++) do
3:     p = RAND(0, 1);
4:     if p ⩽ (m − s)/(n − j + 1) then
5:         select S[j];
6:         s++;
7:     end if
8: end for
```

Отличие от ранее описанного алгоритма для $m = 1$ заключается в изменившейся вероятности $\mathcal{P}(j)$ выборки элемента $S[j]$, которая теперь составляет $\mathcal{P}(j) = (m - s) / / (n - j + 1)$, где s — количество элементов, выбранных перед элементом $S[j]$. Обратите внимание на то, что, если все образцы элементов выбраны, то есть $s = m$ и $\mathcal{P}(j) = 0$, это означает, что больше чем m образцов алгоритм 3.3 не генерирует. При этом он не может генерировать меньше чем m образцов. Скажем, невозможно сгенерировать y образцов, где $y < m$, ведь у последних $m - y$ элементов массива S вероятность быть выбранными равна 1, поэтому они неминуемо попадут в окончательную выборку (согласно шагу 4 алгоритма и тому, что сказано в сноске). Что касается равномерности выборки, то в нашем случае $\mathcal{P}(j)$ — это вероятность включения элемента $S[j]$ в случайную выборку размером m при условии, что s элементов находятся в диапазоне $S[1, j - 1]$. По-другому это можно назвать вероятностью попадания элемента $S[j]$ в случайную выборку размером $m - s$, взятую из диапазона $S[j, n]$, то есть из $n - j + 1$ элементов. Эта вероятность получается подсчетом количества сочетаний, включающих элемент $S[j]$, то есть $\binom{n-j}{m-s-1}$, и деления на количество сочетаний, которые либо включают, либо не включают в себя $S[j]$, то есть $\binom{n-j+1}{m-s}$. Поскольку $\binom{b}{a} = \frac{b!}{a!(b-a)!}$, мы получаем формулу для $\mathcal{P}(j)$.

Факт 3.2. Алгоритм 3.3 выполняет $O(n/B)$ операций ввода-вывода за время $O(n)$, генерирует n случайных чисел и использует дополнительное пространство $O(m)$ для равномерной выборки m элементов из последовательности $S[1, n]$ потоковым способом.

В заключение я хочу упомянуть решение, предложенное Джеффри Виттером [4], которое уменьшает количество случайно сгенерированных чисел с n до m, тем самым снижая временную сложность и количество операций ввода-вывода до $O(m)$. Это

решение (с произвольным доступом к входным данным) также может быть вписано в рамки предыдущего раздела, в этом случае мы сможем избежать повторной выборки. Здесь генерируются не случайные *индикаторы*, определяющие, следует ли выбирать элемент $S[j]$, а случайные *скачки*, указывающие, сколько элементов нужно пропустить перед выбором следующего. Виттер вводит случайную переменную $G(v, V)$, где v — количество элементов, которые осталось выбрать, а V — общее количество элементов, которые осталось проверить в массиве S. В соответствии с введенными ранее обозначениями на шаге j переменная $v = m - s$, а $V = n - j + 1$. Элемент $S[G(v, V) + 1]$ становится следующим, формирующим равномерную выборку из того, *что осталось*. Понятно, что при таком подходе дубликаты образцов не генерируются, но он все равно влияет на ожидаемое ограничение по времени из-за стоимости генерации скачков G в соответствии со следующим распределением:

$$\mathcal{P}(G = g) = \binom{V - g - 1}{v - 1} \Big/ \binom{V}{v}.$$

Основная трудность заключается в том, что мы не можем заранее представить в табличной форме и сохранить значения всех биномиальных коэффициентов, поскольку это потребовало бы слишком много времени и места, ведь $V \leqslant n$ и $v \leqslant m$. Удивительно, но Виттер решил эту проблему за ожидаемое время $O(1)$, элегантно адаптировав метод выборки с отклонением фон Неймана к дискретному случаю, вызванному скачками G. Подробности этого решения вы сможете найти в книге [4].

3.3. Потоковая модель и неизвестная длина последовательности

Само собой разумеется, что для вычисления $\mathcal{P}(j)$ в алгоритме 3.3 решающее значение имело знание n. Когда эта информация отсутствует, нужно действовать по-другому. Двум возможным решениям для такого сценария и посвящен остаток этой главы.

Первое решение довольно простое — на основе min-кучи $\mathcal{H}$ размером m и генератора случайных чисел RAND(0, 1). Основная идея заключается в связывании с каждым элементом $S[j]$ случайного приоритета r_j, после чего с помощью кучи $\mathcal{H}$ выбираются элементы с m максимальными приоритетами. Эту идею реализует псевдокод, приведенный в алгоритме 3.4. В нем минимальный среди m максимальных приоритетов, хранящихся в куче $\mathcal{H}$ (он находится на ее вершине), сравнивается с приоритетом r_j текущего рассматриваемого элемента $S[j]$. Если r_j оказывается больше этого минимального приоритета, элемент вставляется в кучу, а минимальный приоритет удаляется из нее и тем самым обновляется набор максимальных приоритетов и связанных с ними элементов.

Алгоритм 3.4. Куча и случайные приоритеты

```
1:  Инициализация ℋ с m фиктивными элементами с приоритетами −∞;
2:  for каждого элемента S[j] do
3:     r_j = RAND(0, 1);
4:     y = минимальный приоритет в куче ℋ;
5:     if r_j > y then
6:        извлечь элемент с приоритетом y из ℋ;
7:        вставить элемент S[j] в ℋ, назначая приоритет r_j;
8:     end if
9:  end for
10: return m элементов, содержащихся в куче ℋ;
```

Поскольку куча $\mathcal{H}$ имеет размер m, именно столько элементов получит окончательный образец. Вставка в $\mathcal{H}$ каждого приоритета или их удаление из него требует времени $O(\log m)$. Случайность приоритетов гарантирует, что каждый элемент имеет одинаковую вероятность включения в набор. Таким образом, мы доказали следующее.

Факт 3.3. Алгоритм 3.4 выполняет $O(n/B)$ операций ввода-вывода за время $O(n \log m)$, генерирует n случайных чисел и использует дополнительное пространство $O(m)$ для равномерной выборки случайных m элементов из последовательности $S[1, n]$ потоковым способом и без знания n.

Ну и напоследок покажу вам элегантный алгоритм *резервуарной выборки*, который приписывает Алану Уотерману [5]. Он лучше алгоритма 3.4 как по временной, так и по пространственной сложности. Идея похожа на ту, что использовалась для алгоритма 3.3, и состоит в корректном определении вероятности выбора элемента. Но проблема в том, как принять однозначное решение относительно $S[j]$, если нам неизвестна длина S. Соответственно, пока продолжается сканирование этого массива, требуется некоторая свобода менять принятые решения.

В псевдокоде алгоритма 3.5 для хранения образцов-кандидатов применяется массив резервуара $R[1, m]$. Изначально R подготавливается для хранения первых m элементов входной последовательности. На каждой итерации j алгоритм выбирает, следует ли включать $S[j]$ в текущий образец. Вероятность выбора $\mathcal{P}(j) = m/j$, причем после выбора из R должен быть выброшен какой-то из уже присутствующих там элементов. Кандидат на выброс выбирается случайным образом, то есть с вероятностью $1/m$. Такой *двойной выбор* в алгоритме 3.5 реализуется с помощью генерации случайного целого числа h, лежащего в диапазоне $[1, j]$. Замена элементов происходит при условии $h \leqslant m$. Вероятность этого события m/j — именно такой результат мы хотели получить для $\mathcal{P}(j)$.

Алгоритм 3.5. Резервуарная выборка

1: Инициализация массива $R[1, m] = S[1, m]$;
2: **for** каждого элемента $S[j]$ **do**
3: $h = \texttt{RAND}(1, j)$;
4: **if** $h \leqslant m$ **then**
5: присваиваем $R[h] = S[j]$;
6: **end if**
7: **end for**
8: **return** массив R;

Чтобы проверить корректность этого алгоритма, нужно показать, что m элементов выбираются из массива S равномерно и случайным образом, то есть что вероятность выбора каждого из них составляет m/n. Изначально это не совсем очевидно. Воспользуемся индукцией по последовательности n неизвестной длины. Базовый случай, в котором $n = m$, очевиден: каждый элемент выбирается с вероятностью $m/n = 1$. Именно это происходит на шаге 1: первые m элементов массива S записываются в резервуар R. Чтобы доказать шаг индукции (переход от $n - 1$ к n элементов), отметим, что для $S[n]$ выполняется свойство равномерной выборки, поскольку по определению этот элемент вставляется в резервуар R с вероятностью $\mathcal{P}(n) = m/n$ (шаг 4). Сложнее вычислить вероятность попасть в выборку для предыдущих элементов массива $S[1, n - 1]$. Элемент $S[j]$, где $j < n$, будет находиться в резервуаре R на шаге n алгоритма 3.5 только в случае, когда на шаге $(n - 1)$ он уже присутствовал там и не был выброшен на следующей итерации. Последнее может произойти, либо если элемент $S[n]$ не выбран (и, таким образом, резервуар R не изменился), либо если $S[n]$ выбран и $S[j]$ не выброшен из R (эти два события независимы друг от друга). В виде формулы эта вероятность выглядит так:

$$\mathcal{P}(S[j] \in R) \times [\mathcal{P}(S[n] \text{ не выбран}) + \mathcal{P}(S[n] \text{ выбран}) \times \\ \times \mathcal{P}(S[j]) \text{ не выброшен из } R)].$$

По индукции вероятность любого элемента $S[j]$, предшествующего $S[n]$, находиться в резервуаре R до того, как будет обработан $S[n]$, составляет $m/(n - 1)$. При этом элемент $S[j]$ остается в резервуаре, когда $S[n]$ не выбран (вероятность этого $1 - m/n$) или когда он не выброшен оттуда выбранным элементом $S[n]$ (в этом случае вероятность равна $(m - 1)/m$). Объединив эти условия, получим:

$$\mathcal{P}(\text{элемент } S[j] \in R \text{ после } n \text{ элементов при } j < n) = \\ = \frac{m}{n-1}\left[\left(1-\frac{m}{n}\right)+\left(\frac{m}{n}\frac{m-1}{m}\right)\right] = \frac{m}{n-1}\frac{n-1}{n} = \frac{m}{n}.$$

Чтобы понять эту формулу, предположим, что есть резервуар R из 10 элементов, в который на шаге 1 вставляются первые 10 элементов массива S. Элемент $S[11]$

попадает в резервуар с вероятностью 10/11, элемент $S[12]$ — с вероятностью 10/12 и т. д. При каждой такой вставке из резервуара выталкивается случайный элемент. Вероятность этого события 1/10. Через n шагов в резервуаре оказывается 10 элементов, каждый из которых был выбран из массива S с вероятностью $10/n$.

Факт 3.4. Алгоритму 3.5 требуются $O(n/B)$ операций ввода-вывода, время $O(n)$, дополнительное пространство — ровно m, и он генерирует n случайных чисел для равномерной выборки m случайных элементов из последовательности $S[1, n]$ потоковым способом и без знания n. Следовательно, этот алгоритм оптимален по времени, пространству и количеству операций ввода-вывода в рассматриваемой модели вычислений.

К недостаткам этого алгоритма можно отнести количество исполнений шага 3, на котором происходит генерация случайного целого числа, причем все большего по мере обработки каждого очередного элемента $S[j]$. Задача уменьшения этого числа неоднократно рассматривалась различными исследователями. Например, существует асимптотически оптимальное решение, которое в [3] называется алгоритмом L. Ожидаемое время работы этого алгоритма $O(m(1 + \log(n/m)))$, и он тоже связан с генерацией случайных чисел, что оптимально с точностью до постоянного множителя. Тем, кто интересуется подробностями, могу посоветовать самостоятельно почитать [3], я же ограничусь несколькими важными замечаниями.

Начнем с ожидаемого количества вставок в резервуар R. Туда гарантированно попадают первые m элементов, а каждый последующий $S[j]$ выполняет шаг 5 с вероятностью m/j при $j > m$, соответственно, ожидаемое число вставок в R можно оценить как:

$$m + \sum_{j=m+1}^{n} (m/j) = m + \left(\sum_{j=1}^{n} (m/j) - \sum_{j=1}^{m} (m/j) \right) = m(1 + H_n - H_m),$$

где $H_n = \sum_{j=1}^{n} (1/j)$ — это n-е гармоническое число, которое в пределе можно оценить как $O(\log n)$. Ровно так же ограничено ожидаемое число выполнений шага 5 алгоритма 3.5. Такое же число шагов выполняет алгоритм L при условии, что генерация прыжков занимает постоянное время.

Вторая ключевая идея, лежащая в основе алгоритма L, состоит в вычислении количества элементов в S, которые отбрасываются до попадания в резервуар R следующего элемента. Именно это выполняется на шаге 5 и реализовано на базе подхода из алгоритма 3.4, когда с каждым элементом $S[j]$ связывается случайный приоритет из диапазона (0, 1), после чего выбираются элементы, соответствующие верхним m приоритетам. Но в нашем случае явно сгенерировать приоритеты для всех элементов невозможно, это делается только для отмеченных элементов. Можно показать, что количество пропущенных элементов соответствует геометрическому распределению и, следовательно, может быть вычислено за постоянное время [2].

Список литературы

1. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Hashing // Introduction to Algorithms. Chapter 11, Hash tables, 253–283. The MIT Press, third edition, 2009.
2. *Devroye L.* Random sampling // Non-uniform Random Variate Generation. Chapter 12, Random Sampling, 611–641, Springer, 611–41, 1986.
3. *Li K.-H.* Reservoir-sampling algorithms of time Complexity O(n(1 + log(N/n))) // ACM Transactions on Mathematical Software, 20 (4): 481–493, 1994.
4. *Vitter J. S.* Faster methods for random sampling // ACM Computing Surveys, 27 (7): 703–718, 1984.
5. *Vitter J. S.* Random sampling with a reservoir // ACM Transactions on Mathematical Software, 11 (1): 37–57, 1985.

Глава 4
РАНЖИРОВАНИЕ СПИСКОВ

> При работе с дисковой памятью указатели небезопасны!

В этой главе рассматривается простая задача, связанная со списками — базовой структурой данных, которая лежит в основе проектирования многих алгоритмов, управляющих взаимосвязанными элементами. Сначала я покажу вам простое по формулировке, но неэффективное решение. Оно получено из оптимального решения, разработанного для RAM-модели. Затем мы обсудим постепенно усложняющиеся варианты решений, которые, несмотря на свою элегантность и эффективность или оптимальность, не могут быть реализованы в виде нескольких строк кода. Все это позволит подчеркнуть тонкую связь между *параллельными вычислениями* и *вычислениями во внешней памяти*, которые могут быть использованы для разработки эффективных решений, работающих с дисковой памятью, на базе параллельных алгоритмов.

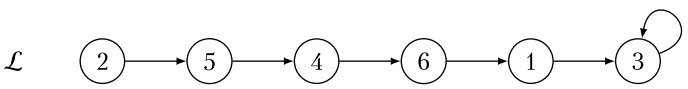
Задача. В однонаправленном списке $\mathcal{L}$, состоящем из n элементов, определить расстояние каждого из элементов от хвоста списка.

Эта задача играет важную роль в параллельных вычислениях и имеет собственное название — *задача ранжирования списка*. Элементы представлены идентификаторами — целыми числами от 1 до n. Список моделируется[1] с помощью массива $\mathtt{Succ}[1, n]$. В записи $\mathtt{Succ}[i]$ этот массив хранит индекс j, когда элемент i указывает на элемент j. Для индекса хвоста списка t $\mathtt{Succ}[t] = t$, то есть указатель, исходящий из t, образует *петлю*. Все это проиллюстрировано на рис. 4.1, где слева приводится графическое представление списка, а справа — его моделирование с помощью массива Succ и результат решения задачи ранжирования списков в массиве $\mathtt{Rank}[1, n]$.

Такая задача легко решается в модели RAM благодаря доступу к внутренней памяти за постоянное время. Навскидку можно придумать три простых алгоритмических

[1] Здесь используется одна из возможных моделей списка — структуры данных с произвольной вставкой и удалением элементов; возможны и другие модели — например, помощью простого массива элементов (в этом случае вставка и удаление будут иметь неэффективную линейную сложность) или так называемого связного списка, в котором индексами являются указатели (адреса) динамически создаваемых элементов. — *Примеч. науч. ред.*

решения, каждое из которых занимает время $O(n)$. Это оптимальная временная сложность, ведь алгоритму приходится сканировать все элементы списка $\mathcal{L}$ для определения их рангов.



id	Succ	Rank
1	3	1
2	5	5
3	3	0
4	6	3
5	4	4
6	1	2

Рис. 4.1. Ввод и вывод для задачи ранжирования на примере списка $\mathcal{L}$ из $n = 6$ элементов. Голова списка — единственный элемент h в {1... 6}, не встречающийся в массиве Succ (здесь $h = 2$); хвост списка — единственный элемент t в {1... 6}, для которого Succ[t] = t (здесь $t = 3$). Соответственно, Rank[t] = 0, а Rank[h] = $n - 1 = 5$

Можно начать сканирование с головы списка и вычислить количество n его элементов, а при повторном сканировании присвоить голове ранг $n - 1$, а каждому последующему элементу в списке — ранг, на каждой итерации уменьшающийся еще на единицу. В другом варианте можно вычислить массив предшествующих элементов вида `Pred[Succ[`i`]]` = i, а затем сканировать список в обратном направлении, начиная с хвоста t, для которого устанавливается `Rank[`t`]` = 0. В этом случае значение `Rank` для каждого элемента увеличивается по мере продвижения обхода от t. Третий способ решения задачи — *рекурсивный*, определяющий функцию `ListRank(`i`)`, которая работает следующим образом: `Rank[`i`]` = 0, если `Succ[`i`]` = i (и, следовательно, $i = t$), в противном случае `Rank[`i`]` = `ListRank(Succ[`i`])` + 1.

Если эти алгоритмы выполнить со списком, хранящимся на диске (когда работа с элементами массива `Succ` за пределами блока приводит к дисковой операции), то из-за произвольного распределения ссылок потребуется $\Theta(n)$ операций ввода-вывода. Произвольное распределение ссылок означает, что мы не можем предсказать, в какой момент потребуется доступ к записям массивов `Rank` и `Succ` (в худшем варианте — при каждом обращении к элементу массива). Затраты на ввод-вывод в данном случае сильно отличаются от очевидной нижней границы $\Omega(n/B)$ — ее можно вывести с помощью тех же рассуждений, коими мы пользовались для RAM-модели. Такая граница кажется очень низкой, но в этой главе мы сможем подойти к ней довольно близко. В процессе этого я познакомлю вас с некоторыми сложными методами, достаточно универсальными для того, чтобы найти применение во многих других, на первый взгляд непохожих сценариях.

Мораль этого раздела такова: чтобы добиться высокой эффективности ввода-вывода в связанных структурах данных, нужно по возможности избегать обхода указателей. Кроме того, рекомендую изучить обширную литературу по параллельным алгоритмам, например [2], где также определена модель параллельной оперативной памяти (PRAM), поскольку эффективный параллелизм, как ни странно, можно использовать для достижения высокой эффективности ввода-вывода.

4.1. Техника перескока указателей

Существует хорошо известный вариант решения задачи ранжирования списков в параллельных вычислениях, основанный на *технике перескока указателя* (pointer-jumping technique). Алгоритмическая идея довольно проста. Берутся n процессоров, каждый из которых имеет дело с одним элементом списка $\mathcal{L}$. Процессор i инициализирует Rank[i] = 0, если $i = t$, в противном случае присваивается Rank[i] = 1. Затем он выполняет две инструкции: Rank[i] = Rank[i] + Rank[Succ[i]] и Succ[i] = Succ[Succ[i]]. Такое двойное обновление фактически сохраняет следующий инвариант: Rank[i] *измеряет расстояние (то есть количество элементов) в исходном списке между* i *и элементом, который в данный момент хранится в* Succ[i]. Формальное доказательство можно вывести по индукции, и я его пропущу, проиллюстрировав примером на рис. 4.2. Пунктирные стрелки указывают на новые связи, вычисленные за один шаг перехода указателя, а таблица справа от каждого списка содержит значения массива Rank[1, n], которые пересчитываются после каждого шага. Значения, выделенные жирным шрифтом, — это окончательные/корректные значения. Обратите внимание на то, что расстояния растут не линейно (то есть 1, 2, 3...), а как степень двойки (то есть 1, 2, 4...), вплоть до шага, на котором следующий переход достигает хвоста списка t. Это означает, что общее количество выполнений этого шага параллельным алгоритмом — то есть *время* выполнения — составляет $O(\log n)$. То есть мы получили экспоненциальное улучшение по сравнению с временем работы последовательного алгоритма.

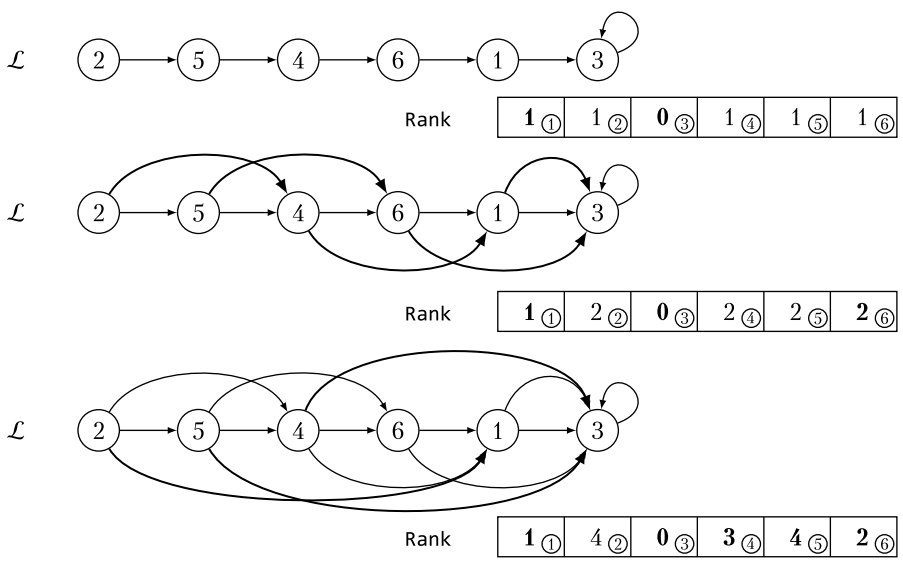


Рис. 4.2. Перескок указателя на примере списка $\mathcal{L}$ (см. рис. 4.1). Сплошные черные стрелки — текущее состояние массива Succ, пунктирные — результат выполнения одного перескока. Когда стрелка начинает указывать на хвост списка t, значение Rank становится корректным (выделено жирным шрифтом)

При n процессорах переход указателя выполняет в общей сложности $O(n \log n)$ операций, что неэффективно по сравнению с $O(n)$ операций оптимального RAM-алгоритма.

Лемма 4.1. *Параллельный алгоритм, использующий n процессоров и технику перескока указателя, требует для решения задачи ранжирования списков время* $O(\log n)$ *и* $O(n \log n)$ *операций.*

Этот результат можно улучшить, постаравшись приблизиться к оптимальному количеству операций. Например, имеет смысл отключать процессоры, когда элементы, с которыми они работают, достигают конца списка. Не буду углубляться в детали, поскольку эта тема подробно рассмотрена в литературе, например [2]. Я же хочу показать вам реализацию техники перескока указателя в условиях нашей задачи, где имеются один процессор и двухуровневая память. Вы увидите, что при наличии эффективного параллельного алгоритма очень просто получить алгоритм, эффективный по вводу-выводу. Простота обеспечивается алгоритмической схемой, развертывающей две основные примитивные операции: сканирование и сортировку множества кортежей, которые в настоящее время доступны почти на каждой распределенной платформе, такой как Apache Hadoop.

4.2. Моделирование параллельного алгоритма в двухуровневой памяти

Основная трудность применения техники перескока указателя в двухуровневой памяти связана с произвольным расположением списка на диске. В результате доступ к памяти для обновления указателей `Succ` и значений `Rank` происходит по произвольному шаблону, что порождает множество избыточных операций ввода-вывода. Обойти эту проблему позволяют два ключевых шага техники перескока указателя, которые реализуются с помощью фиксированного количества примитивов `Sort` и `Scan` на n тройках целых чисел. Базовый примитив `Sort` очень сложно реализовать эффективно с точки зрения ввода-вывода, и в главе 5 мы будем разбирать его очень подробно. Для наглядности обозначим его сложность ввода-вывода $\tilde{O}(n/B)$, что указывает на наличие скрытого логарифмического коэффициента, зависящего от основных параметров модели, а именно от M, n и B. На практике этот коэффициент пренебрежимо мал, поэтому его можно безопасно ограничить сверху очень малой константой, например 4 или меньше. Пока мы его *скроем*, чтобы упростить обсуждение. Зато второй примитив, `Scan`, прост и для обработки непрерывной части диска, занятой n тройками, требует $O(n/B)$ операций ввода-вывода.

Общую алгоритмическую структуру определим в двух шагах техники перескока указателя. Каждый шаг состоит из операции копирования или суммирования над двумя записями массива — `Succ` или `Rank`. Для наглядности будем рассматривать обобщенный массив A и смоделируем параллельную операцию на диске следующим образом.

> Предположим, параллельный проход имеет следующий вид: $A[a_i]$ `op` $A[b_i]$, где `op` — это операция, выполняемая над двумя элементами массива, $A[a_i]$ и $A[b_i]$, параллельно всеми процессорами $i = 1, 2, ..., n$, которые фактически считывают $A[b_i]$ и используют это значение для обновления $A[a_i]$.

Операция `op` для массива `Rank` (когда A = `Rank`) — это суммирование и присваивание, а для массива `Succ` (когда A = `Succ`) — копирование. Что касается индексов массива, то для обоих шагов они равны $a_i = i$ и b_i = `Succ`$[i]$. Ключевая задача — показать, что $A[a_i]$ `op` $A[b_i]$ допускает одновременную реализацию для всех $i = 1, 2, 3, ..., n$, используя фиксированное число примитивов `Sort` и `Scan` и выполняя в общей сложности $\tilde{O}(n/B)$ операций ввода-вывода. Эта реализация состоит из пяти шагов.

1. `Scan`: сканируем диск и создаем последовательность троек в форме $\langle a_i, b_i, 0\rangle$. Каждая тройка несет информацию об исходном адресе записи массива, участвующей в операции b_i, ее целевом адресе a_i, и значении, которое мы перемещаем (это третий компонент с начальным значением 0).
2. `Sort`: сортируем тройки по второму компоненту, то есть исходному адресу b_i. Здесь мы «выравниваем» каждую тройку $\langle a_i, b_i, 0\rangle$ с ячейкой памяти $A[b_i]$.
3. Сканируем тройки и массив A, используя два итератора — один по тройкам, второй по массиву A. Поскольку тройки сортируются по второму компоненту, то есть по b_i, новые тройки $\langle a_i, b_i, A[b_i]\rangle$ в процессе сканирования создаются достаточно эффективно. Обратите внимание, что не все ячейки памяти A обязательно присутствуют как второй компонент тройки, тем не менее их согласованный порядок позволяет в рамках операции `Scan` так же согласованно копировать $A[b_i]$ в тройку для b_i.
4. Сортируем тройки по первому компоненту, то есть целевому адресу a_i. Таким образом, тройку $\langle a_i, b_i, A[b_i]\rangle$ мы «выравниваем» с ячейкой памяти $A[a_i]$.
5. Снова сканируем тройки и массив A двумя итераторами: один проходит по тройкам, второй — по массиву A. Для каждой тройки $\langle a_i, b_i, A[b_i]\rangle$ обновляем содержимое ячейки памяти $A[a_i]$ в соответствии с семантикой `op` и значением $A[b_i]$. Обновление можно эффективно выполнить в рамках согласованной операции `Scan`.

Сложность ввода-вывода в данном случае определить легко, поскольку алгоритм выполняет две операции `Sort` и три операции `Scan`, в которых задействовано n троек. Поэтому можно утверждать следующее.

Теорема 4.1. *Параллельное выполнение n операций $A[a_i]$* `op` *$A[b_i]$ в двухуровневой модели памяти можно смоделировать, используя фиксированное число примитивов* `Sort` *и* `Scan`*, что в общей сложности потребует $\tilde{O}(n/B)$ операций ввода-вывода.*

В алгоритме, где используются параллельные перескоки указателя, они выполняются $O(\log n)$ раз, поэтому у нас есть такая теорема.

Теорема 4.2. *Параллельный алгоритм перескока указателя в двухуровневой модели памяти можно смоделировать таким образом, что он потребует* $\tilde{O}((n/B)\log n)$ *операций ввода-вывода.*

Оказывается, эта граница оценивается как $o(n)$, то есть этот результат лучше, чем прямое выполнение RAM-алгоритма на диске, когда $B = \omega(\log n)$. Такое условие повсеместно выполняется на практике, поскольку $B \approx 10^4$ байт и $\log n \leqslant 80$ для любого набора данных реального размера 2^{80} (это предполагаемое число атомов во Вселенной)[1].

Пример такой модели показан на рис. 4.3. Таблица слева содержит данные массивов `Rank` и `Succ` для списка из $n = 6$ элементов. Указатели этих элементов показаны сплошными стрелками. Таблица справа содержит данные этих двух массивов после одного шага алгоритма, использующего технику перескока указателя. Новые указатели имеют вид пунктирных стрелок. Четыре столбца троек демонстрируют их изменение в процессе описанных ранее пяти фаз сканирования/сортировки. Первый столбец троек создается при первом сканировании как $\langle i, \texttt{Succ}[i], 0\rangle$, поскольку $a_i = i$ и $b_i = \texttt{Succ}[i]$. Второй столбец троек получается в результате сортировки по второму компоненту. Третий столбец троек появляется после согласованной процедуры `Scan` троек и массива `Rank`. В результате появляются новые тройки $\langle i, \texttt{Succ}[i], \texttt{Rank}[\texttt{Succ}[i]]\rangle$ Четвертый столбец троек возникает в результате сортировки по первому компоненту, а именно i. А последняя согласованная процедура `Scan` массива `Rank` дает нам третий компонент этих троек, то есть $\texttt{Rank}[i] = \texttt{Rank}[i] + \texttt{Rank}[\texttt{Succ}[i]]$.

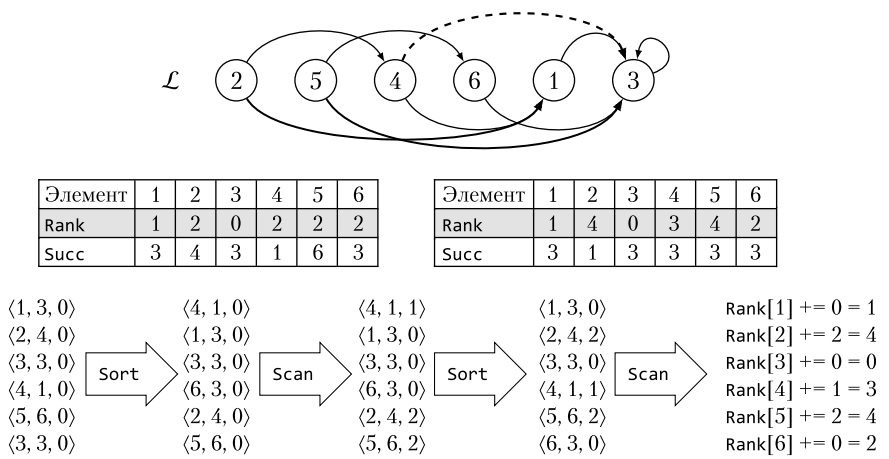


Рис. 4.3. Пример параллельного использования примитивов Scan и Sort для вычисления массива Rank с конфигурацией, указанной на схеме и в таблицах. Сплошные стрелки — указатели элементов, описанных в таблице *слева*, пунктирные стрелки — обновленные указатели после одного перескока, описанного в таблице *справа*

[1] Например, см.: https://ru.wikipedia.org/wiki/Большие_числа.

Итак, я показал вам пример обновления массива `Rank`, массив `Succ` может быть обновлен аналогичным образом. Но важно понимать, что обновление этих двух массивов можно осуществить одновременно, вместо троек взяв четверки, включающие значения `Rank[Succ[`i`]]` и `Succ[Succ[`i`]]`. Это возможно, потому что ссылки на начальный и конечный адреса у обоих значений одни и те же — i и `Succ[`i`]`.

Фактически показанную в этом разделе схему можно обобщить на любой параллельный алгоритм, что приводит к следующему важному и полезному результату (см. [1]).

Теорема 4.3. *В системе с p процессорами и двухуровневой моделью памяти любой параллельный алгоритм, выполняющий T шагов, можно смоделировать в виде последовательного алгоритма, учитывающего характеристики диска, который выполняет* $\tilde{O}(n/B)$ *операций ввода-вывода и требует пространство* $O(n)$.

Такое моделирование выгодно при $T = o(B)$, что подразумевает сублинейное число операций ввода-вывода, а именно $o(n)$. Это происходит во всех случаях, когда параллельный алгоритм имеет низкую *полилогарифмическую* временную сложность. Такая ситуация типична для параллельных алгоритмов, разработанных для так называемой модели вычислений P-RAM, которая предполагает, что все процессоры работают независимо друг от друга и могут за постоянное время получать доступ к неограниченной общей памяти. Это идеальная модель, широко известная в 1980–1990-х годах [2], которая позволила разработать много мощных параллельных методов, применявшихся как к распределенным, так и к дисковым алгоритмам. Ее основные недостатки: она не учитывала конфликты между процессорами, обращающимися к общей памяти[1], а связи между ними имели упрощенную структуру. Тем не менее простота P-RAM позволила исследователям сосредоточиться на алгоритмических аспектах параллельных вычислений и разработать параллельные схемы, такие как схема с перескоком указателя и другие, описанные в оставшейся части этой главы.

4.3. Подход «разделяй и властвуй»

Сейчас вы увидите, что задача ранжирования списков имеет более эффективные решения, чем с помощью техники перескока указателя. Я покажу вам алгоритм на основе принципа *«разделяй и властвуй»*, в данном случае адаптированный для работы с однонаправленным списком элементов.

Прежде чем углубляться в технические детали, кратко перечислю основные идеи, лежащие в основе разработки нашего алгоритма (назовем его $\mathcal{A}_{dc}$). Он будет

[1] Речь идет не о классической для параллельного программирования *зависимости* потоков по данным, а о том, что в P-RAM предполагается *одновременный* доступ нескольких процессоров к одной и той же ячейке оперативной памяти, чего в реальных компьютерах достичь очень сложно. — *Примеч. науч. ред.*

решать задачу $\mathcal{P}$ на n входных данных [3]. Этот алгоритм состоит из трех основных этапов.

- **Разделение.** Алгоритм $\mathcal{A}_{\text{dc}}$ создает набор из k подзадач, скажем $\mathcal{P}_1$, $\mathcal{P}_2$... $\mathcal{P}_k$, размером n_1, n_2... n_k соответственно. Они идентичны исходной задаче $\mathcal{P}$, но сформулированы на меньших входных данных, то есть $n_i < n$.
- **Овладение.** Алгоритм $\mathcal{A}_{\text{dc}}$ рекурсивно вызывается для подзадач $\mathcal{P}_i$, получая решения s_i.
- **Слияние.** Алгоритм $\mathcal{A}_{\text{dc}}$ объединяет решения s_i, чтобы получить решение s для исходной задачи $\mathcal{P}$. Параметр s возвращается в качестве выходных данных $\mathcal{A}_{\text{dc}}$.

Очевидно, что техника «разделяй и властвуй» генерирует рекурсивный алгоритм $\mathcal{A}_{\text{dc}}$, которому для завершения работы требуется базовый случай. Обычно основанием рекурсии выступает остановка $\mathcal{A}_{\text{dc}}$ всякий раз, когда входные данные состоят из малого числа элементов, например $n \leqslant 1$. Для небольших входных данных решение можно вычислить за постоянное время легко и напрямую, например, путем перебора.

Временная сложность $T(n)$ алгоритма $\mathcal{A}_{\text{dc}}$ описывается рекуррентным соотношением, в котором основанием рекурсии является $T(n) = O(1)$ для $n \leqslant 1$, а для прочих случаев:

$$T(n) = D(n) + R(n) + \sum_{i=1,\ldots,k} T(n_i),$$

где $D(n)$ — стоимость второго шага, а $R(n)$ — третьего. Последний член учитывает стоимость всех k рекурсивных вызовов. Этой информации вполне достаточно. Тех же, кому интересно более глубокое обсуждение метода «разделяй и властвуй» и основной теоремы, которая дает математическое решение для большинства рекуррентных соотношений, таких как приведенное здесь, я отсылаю к главе 4 в [3].

У нас все готово для адаптации метода «разделяй и властвуй» к задаче ранжирования списков. Предлагаемый алгоритм довольно прост и начинается с присваивания $\texttt{Rank}[t] = 0$ и $\texttt{Rank}[i] = 1$ для всех элементов $i \neq t$. Выполняются три основных шага.

- **Разделение.** Определяем набор элементов $I = \{i_1, i_2 \ldots i_h\}$, взятых из входного списка I. Набор I должен быть *независимым*, что означает: элемент, следующий в списке I за элементом, добавленным в I, сам туда не попадает. Такое условие явно гарантирует, что $|I| \leqslant n/2$, поскольку из двух последовательных элементов может быть выбран всего один. Также алгоритм гарантирует, что $|I| \geqslant n/c$, где $c > 2$, чтобы сделать подход эффективным по времени.
- **Овладение.** Формируем список $\mathcal{L}^* = \mathcal{L} - I$, удаляя элементы I из списка $\mathcal{L}$. Это реализуется техникой перескока указателя только к предшественникам удаленных элементов. Поэтому для каждого элемента $x \in \mathcal{L}^*$ такого, что $\texttt{Succ}[x] \in I$, мы устанавливаем $\texttt{Rank}[x] = \texttt{Rank}[x] + \texttt{Rank}[\texttt{Succ}[x]]$ и $\texttt{Succ}[x] = \texttt{Succ}[\texttt{Succ}[x]]$. Это означает, что при любом рекурсивном вызове $\texttt{Rank}[x]$ учитывает количество элементов исходного входного списка, которые лежат между x

(включительно) и текущим Succ[x]. Затем рекурсивно решаем задачу ранжирования списка $\mathcal{L}^*$. Обратите внимание на то, что $n/2 \leqslant |\mathcal{L}^*| \leqslant (1 - 1/c)n$, поэтому рекурсия действует на подсписок $\mathcal{L}^*$, размер которого является дробной частью $\mathcal{L}$. Это важно для обеспечения эффективности рекурсивных вызовов.

- **Слияние.** К этому моменту рекурсивные вызовы правильно ранжировали все элементы подсписка $\mathcal{L}^*$. Ранг каждого элемента $x \in I$ рассчитывается как Rank[x] = Rank[x] + Rank[Succ[x]]. В данном случае обновление происходит по уже знакомой вам схеме, которая использовалась при перескоке указателя. Корректность этой формулы обеспечивается двумя фактами:
  - свойство независимого множества I гарантирует, что Succ[x] $\notin I$, таким образом, Succ[x] $\in \mathcal{L}^*$ и нам доступно его значение Rank;
  - по индукции Rank[Succ[x]] учитывает расстояние Succ[x] от хвоста $\mathcal{L}$, а Rank[x] учитывает в исходном входном списке количество элементов между x (включительно) и Succ[x], как наблюдалось на втором этапе. Фактически удаление x (из-за его переноса в I) может произойти на любом рекурсивном шаге, поэтому при рассмотрении исходного списка x может быть далеко от текущего Succ[x]. Это означает, что возможен случай Rank[x] >> 1, который предыдущий шаг суммирования учтет корректно. В результате все элементы в $\mathcal{L} = \mathcal{L}^* \cup I$ будут иметь правильно вычисленные значения Rank, следовательно, индукция сохраняется и алгоритм может вернуться к своей вызывающей функции.

Рисунок 4.4 иллюстрирует, как из списка $\mathcal{L}$ удаляется независимый набор (обозначен жирными узлами) и как обновляются ссылки на элементы Succ. Обратите внимание на то, что *указатели перескакивают* только на предшественников удаленных элементов (то есть предшественников элементов, попавших в I), при этом указатели других элементов не меняются. Очевидно, если на следующем шаге будет выбрано $I = \{6\}$, то список окажется составленным из трех элементов $\mathcal{L} = (2, 4, 3)$, чьи окончательные ранги равны $(5, 3, 0)$ соответственно. На этапе слияния произойдет повторная вставка 6 в $\mathcal{L} = (2, 4, 3)$ сразу после 4 и будет вычислен Rank[6] = Rank[6] + + Rank[3] = 2 + 0 = 2, потому что в текущем списке Succ[6] = 3. И наоборот, если не принять во внимание, что элемент 6 в исходном списке может находиться далеко от Succ[6] = 3, и прибавить 1, расчет для Rank[6] будет неверным.

Очевидно, что эффективность ввода-вывода этого алгоритма зависит от первого шага. Поскольку второй шаг является рекурсивным, количество вводов-выводов для него можно оценить как $T((1 - 1/c)n)$. На третьем шаге все вставки происходят одновременно, учитывая, что удаленные элементы не являются смежными (по определению независимого множества), следовательно, его можно реализовать за $\tilde{O}(n/B)$ вводов-выводов (см. теорему 4.1).

Теорема 4.4. *Задача ранжирования для списка $\mathcal{L}$ длиной n решается методом «разделяй и властвуй» с количеством вводов-выводов $T(n) = I(n) + \tilde{O}(n/B) + T((1 - 1/c)n)$, где $I(n)$ — количество вводов-выводов в процессе выбора из списка $\mathcal{L}$ независимого набора размером не менее n/c (и конечно, не более $n/2$).*

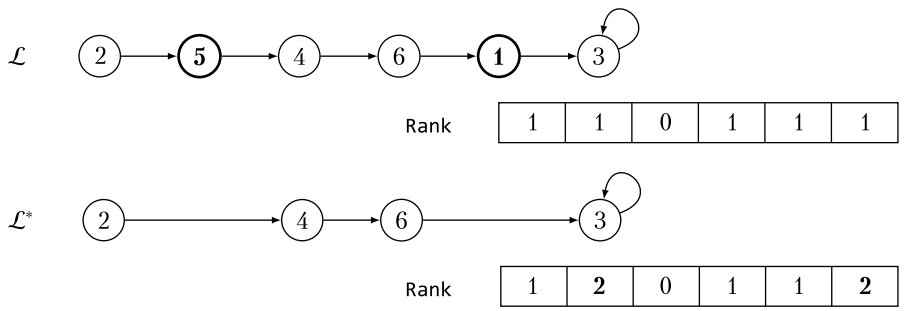


Рис. 4.4. Удаление из списка элементов, переходящих в независимый набор, выделенных жирным шрифтом. В результате получается нижний список, массив Rank пересчитывается с учетом удаленных элементов. Обновленные ранги, а именно ранги элементов 2 и 6 (из-за удаления элементов 5 и 1 соответственно), показаны жирным шрифтом

Если мы можем последовательно обойти список, задача формирования большого независимого набора тривиальна: достаточно выбрать один из каждых двух элементов. Но, так как мы работаем с дисковой памятью, последовательный обход списка неэффективен с точки зрения ввода-вывода — это именно то, чего нужно избежать, и именно это вызвало затруднения.

Поэтому сосредоточимся на том, как сформировать *большой* независимый набор элементов $\mathcal{L}$ эффективным с точки зрения ввода-вывода способом, то есть используя внутри списка только локальную информацию. Я покажу два решения этой задачи: одно — простое рандомизированное, другое — более сложное детерминированное. Что примечательно, последнее решение, называемое *детерминированным подбрасыванием монеты*, нашло применение во множестве других задач, таких как сжатие данных, поиск сходства текстов и проверка равенства строк. Это универсальная и очень мощная техника, определенно заслуживающая вашего внимания.

4.3.1. Рандомизированное решение

Идея алгоритма проста: для каждого элемента списка $\mathcal{L}$ подбрасывается симметричная монета, после чего требуется выбрать элементы i, для которых `coin`(i) = `H`, но `coin`(`Succ`[i]) = `T`[1].

Вероятность выбора элемента i составляет 1/4, поскольку из четырех возможных конфигураций это происходит для одной (`HT`). Таким образом, ожидаемое количество элементов, выбранных для I, составляет $n/4$. Используя сложные вероятностные инструменты, такие как оценка Чернова[2], можно доказать, что количество

[1] Алгоритм также работает, если поменять местами роли орла (H) и решки (T), но не работает для конфигураций HH или TT. Почему?

[2] См.: https://ru.wikipedia.org/wiki/Оценка_Чернова.

выбранных элементов тесно сконцентрировано[1] вокруг $n/4$. Это означает, что для некоторого $c > 4$ алгоритм может повторять подбрасывание монеты до тех пор, пока $|I| \geqslant n/c$. Высокая концентрация гарантирует, что это повторение будет выполнено небольшое постоянное число раз.

Отмечу, что проверку сторон монеты для выбора элементов в I по теореме 4.1 можно смоделировать с помощью нескольких примитивов `Sort` и `Scan` с количеством ожидаемых вводов-выводов $I(n) = \tilde{O}(n/B)$. Подставляя это значение в теорему 4.4, мы получаем следующее рекуррентное соотношение для сложности ввода-вывода предлагаемого алгоритма: $T(n) = \tilde{O}(n/B) + T(((c-1)n)/c)$, где $c > 4$. Используя основную теорему (см. главу 4 в [3]), можно доказать следующий результат.

Теорема 4.5. *Существует рандомизированный алгоритм, который решает задачу ранжирования списков, сформулированную для списка длиной n, за* $\tilde{O}(n/B)$ *ожидаемых операций ввода-вывода.*

4.3.2. Детерминированное подбрасывание монеты*

Ключевым свойством описанного ранее рандомизированного алгоритма была *локальность* структуры I. Именно она позволяла выбирать элемент i, просто просматривая результаты подбрасывания монет для элемента i и следующего за ним элемента $\texttt{Succ}[i]$. В этом разделе мы смоделируем этот процесс с помощью так называемой техники *детерминированного подбрасывания монеты*. Теперь вместо присваивания каждому элементу двух значений монеты (то есть `H` или `T`) будем присваивать n значений (далее обозначаются целыми числами $0, 1 \ldots n-1$), которые в конечном счете сведем к трем, а именно 0, 1, 2. После чего останется выбрать в I элементы, значение которых минимально среди соседних элементов в списке $\mathcal{L}$. То есть алгоритм будет сравнивать три элемента, смежные в $\mathcal{L}$, для чего опять же требуется выполнение фиксированного числа примитивов `Sort` и `Scan`. Рассмотрим этот процесс более подробно.

- **Инициализация.** Каждому элементу i присваивается значение $\texttt{coin}(i) = i - 1$. Таким образом, все элементы принимают разные значения $\texttt{coin} < n$. Эти значения мы представим в $b = \lceil \log n \rceil$ битах и обозначим двоичное представление $\texttt{coin}(i)$, занимающее b бит, как $\texttt{bit}_b(i)$.
- **Сокращение до шести значений монет.** Следующие шаги повторяются до тех пор, пока для всех элементов $i \in \mathcal{L}$ не станет $\texttt{coin}(i) < 6$.
  - Вычисляем $\pi(i)$: первую позицию справа, в которой различаются биты $\texttt{bit}_b(i)$ и $\texttt{bit}_b(\texttt{Succ}[i])$, и обозначаем $z(i)$ значение бита $\texttt{bit}_b(i)$ в этой позиции для всех элементов i, которые не относятся к хвосту списка.
  - Новое значение для i вычисляем как $\texttt{coin}(i) = 2\pi(i) + z(i)$. Для его представления используем $\lceil \log b \rceil + 1$ бит. Если i — последний элемент в списке и, следовательно, за ним ничего нет, то $\texttt{coin}(i)$ определяется как минимальное значение, отличное от всех остальных назначенных монет.

[1] https://ru.wikipedia.org/wiki/Неравенство_концентрации_меры.

- **Сокращение до трех значений монет.** Для каждого $v \in \{3, 4, 5\}$ берем все элементы i, для которых $\texttt{coin}(i) = v$, и меняем их значение на $\{0, 1, 2\} \setminus \{\texttt{coin}(\texttt{Succ}[i]), \texttt{coin}(\texttt{Pred}[i])\}$.
- **Выбор I.** Выбираем элементы i, для которых $\texttt{coin}(i)$ является локальным минимумом, то есть он меньше, чем $\texttt{coin}(\texttt{Pred}[i])$ и $\texttt{coin}(\texttt{Succ}[i])$.

Проверим корректность этого алгоритма. Изначально все значения монет различны и находятся в диапазоне $\{0, 1... n - 1\}$. Соответственно, вычисление $\pi(i)$ корректно и $2\pi(i) + z(i) \leqslant 2(b - 1) + 1 = 2b - 1$, поскольку значение $\texttt{coin}(i)$ было представлено b битами и, следовательно, $\pi(i) \leqslant b - 1$ (отсчитывая от 0). Поэтому новое значение $\texttt{coin}(i)$ может быть представлено $\lceil \log b \rceil + 1$ битами, таким образом, обновление b тоже корректно.

Важно отметить, что новое значение $\texttt{coin}(i)$ все еще отличается от значений его соседей в списке $\mathcal{L}$, а именно $\texttt{coin}(\texttt{Succ}[i])$ и $\texttt{coin}(\texttt{Pred}[i])$. Докажем это от противного. Предположим, что $\texttt{coin}(i) = \texttt{coin}(\texttt{Succ}[i])$ (второй случай доказывается аналогично), тогда $2\pi(i) + z(i) = 2\pi(\texttt{Succ}[i]) + z(\texttt{Succ}[i])$. Поскольку z — это битовое значение, два значения coin равны только при условии, что $\pi(i) = \pi(\texttt{Succ}[i])$, а $z(i) = z(\texttt{Succ}[i])$. Но если это условие выполняется, то двухбитовые последовательности $\texttt{bit}_b(i)$ и $\texttt{bit}_b(\texttt{Succ}[i])$ не могут различаться в битовой позиции $\pi(i)$, как мы и предположили изначально.

Это наглядно демонстрирует корректность шага, на котором происходит переход от n значений монет к шести, а затем и к трем значениям. Очевидно также, что выбранные элементы образуют независимый набор из-за минимальности $\texttt{coin}(i)$ и различимости значений соседних монет.

Чтобы оценить сложность ввода-вывода, введем функцию $\log^* n$, которая определяется как $\min\{j \mid \log^{(j)} n \leqslant 1\}$, где $\log^{(j)} n$ — это функция логарифма, j раз примененная к n[1]. В качестве примера возьмем $n = 16$ и вычислим $\log^{(0)} 16 = 16$, $\log^{(1)} 16 = 4$, $\log^{(2)} 16 = 2$, $\log^{(3)} 16 = 1$, таким образом, $\log^* 16 = 3$. Нетрудно убедиться, что $\log^* n$ растет очень медленно, в частности для $n = 2^{65\,536}$ его значение равно 5.

Теперь нужно ограничить число итераций, необходимых для уменьшения номиналов монет до $\{0, 1, ..., 5\}$. Оно равно $\log^* n$, потому что на каждом шаге число битов, используемых для представления значений монет, уменьшается логарифмически от b до $\lceil \log b \rceil + 1$. Согласно теореме 4.1, каждый отдельный шаг может быть реализован с помощью нескольких примитивов Sort и Scan, что потребует $\tilde{O}(n/B)$ операций ввода-вывода. Соответственно, построение независимого набора потребует $I(n) = \tilde{O}((n/B) \log^* n)$, что согласно определению $\tilde{O}()$ будет равно $\tilde{O}(n/B)$. Снизу размер I может быть ограничен как $|I| \geqslant n/4$, поскольку расстояние между двумя последовательными выбранными элементами (локальными минимумами) становится максимальным, когда значения монет образуют *битоническую последовательность* вида ..., 0, 1, 2, 1, 0, 1, 2, 1, 0, ...

[1] Напомню, что все логарифмы имеют основание 2, если не указано иное.

Подставляя это значение в теорему 4.4, мы получаем то же самое рекуррентное соотношение, что и для рандомизированного алгоритма, представленного в подразделе 4.3.1, за исключением того, что теперь алгоритм детерминированный, а его ограничение ввода-вывода соответствует наихудшему случаю.

Теорема 4.6. *Существует детерминированный алгоритм, решающий задачу ранжирования списков для списка длиной n за* $\tilde{O}(n/B)$ *операций ввода-вывода в худшем случае.*

В заключение хотелось бы отметить, что логарифмический член, скрытый в $\tilde{O}()$-нотации, имеет форму $(\log^* n)(\log_{M/B} n)$. Это будет показано в главе 5. Можно смело считать, что он меньше 15, поскольку на практике для n до 1 Пбайт при использовании обычной машины с несколькими гигабайтами внутренней памяти $\log_{M/B} n \leqslant 3$ и $\log^* n \leqslant 5$.

Список литературы

1. *Chiang Y.-J., Goodrich M. T., Grove E. F. et al.* External-memory graph algorithms // Proceedings of the 6th ACM — SIAM Symposium on Algorithms (SODA), 139–149, 1995.
2. *JaJa J.* An introduction to parallel algorithms. Addison-Wesley, 1992.
3. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to algorithms. The MIT Press, third edition, 2009.

Глава 5
СОРТИРОВКА АТОМАРНЫХ ЭЛЕМЕНТОВ

> Мы обожаем хаос, потому что любим наводить порядок.
>
> *Приписывается М. К. Эшеру*

В этой главе рассматривается широко известная задача сортировки набора *атомарных* элементов. Об элементах *переменной длины* (то есть строки) мы поговорим в следующей главе. Термин «атомарный» означает, что элементы занимают фиксированное количество ячеек памяти. Обычно это целые или действительные числа, представленные фиксированным количеством байтов, скажем, по 4 байта (32 бита) или 8 байт (64 бита) каждый.

Задача сортировки. Даны последовательность из n атомарных элементов $S[1, n]$ и общий порядок $\leqslant$ между каждой парой. Требуется отсортировать S по возрастанию.

Я расскажу вам про две взаимодополняющие парадигмы сортировки: сортировку путем *слияния*, которая лежит в основе конструкции `MERGESORT`, и сортировку путем *распределения*, лежащую в основе конструкции `QUICKSORT`. Мы адаптируем их для работы в двухуровневой модели памяти, проанализируем сложность ввода-вывода и познакомимся с полезными инструментами, ускоряющими их выполнение, такими как техника снегоочистителя и сжатие данных. Я покажу оптимальность этих адаптаций к работе с дисками, вычислив нижнюю границу количества операций ввода-вывода, которые в ходе работы с внешней памятью должен выполнить любой сортировщик для создания упорядоченной последовательности S. В этом контексте задачу сортировки можно связать с так называемой задачей *перестановки*, которая обычно игнорируется при сортировке в RAM-модели. Затем я покажу интересную эквивалентность между сложностью ввода-вывода у этих двух задач. Именно на этом базируется математическое обоснование повсеместного применения сортировщиков в разработке эффективных с точки зрения ввода-вывода решений для задач, связанных с большими наборами данных.

Задача перестановки. Даны последовательность из n атомарных элементов $S[1, n]$ и перестановка $\pi[1, n]$ целых чисел $\{1, 2... n\}$. Требуется поменять порядок элементов в S в соответствии с π и получить новую последовательность $S[\pi[1]], S[\pi[2]]... S[\pi[n]]$.

Очевидно, что сортировка включает в себя перестановку как подзадачу. Для упорядочения последовательности S нужно определить ее отсортированную перестановку, а затем реализовать ее (в алгоритме обе эти стадии могут изощренно переплетаться). Соответственно, сортировка должна быть сложнее перестановки. И действительно, известно, что в RAM-модели для сортировки путем сравнения n атомарных элементов с помощью алгоритмов `MERGESORT` или `HEAPSORT` [3] требуется время $\Theta(n \log n)$, тогда как их перестановка занимает время $\Theta(n)$. Последнее ограничение по времени можно получить, просто перемещая один элемент за раз в соответствии с порядком, определенным в массиве π. Как ни странно, в дисковой модели нет такой большой разницы в сложности. При некоторых разумных условиях для входных и модельных параметров n, M, B обе эти задачи демонстрируют одинаковую сложность ввода-вывода. Такой элегантный и глубокий результат, полученный Аггарвалом и Виттером в 1988 году [1], несомненно, стимулировал написание огромного количества алгоритмической литературы по теме ввода-вывода. С философской точки зрения этот результат формально подтверждает интуитивно понятную вещь: реально узким местом является *перемещение* элементов на диске, а не *поиск* отсортированной перестановки. И в самом деле, исследователи и инженеры-программисты обычно говорят про *узкое место ввода-вывода*, когда характеризуют эту проблему в своих медленных алгоритмах.

В заключение упомяну еще два решения задачи сортировки элементов на дисках. Во-первых, это метод чередования данных, лежащий в основе систем RAID (массив независимых дисков с избыточностью), который превращает любой эффективный/оптимальный алгоритм с одним диском в эффективный алгоритм с D дисками, обычно теряя при этом оптимальность, если таковая была. Во-вторых, алгоритм `GREEDSORT`, который специально разработан для сортировки на D дисках и обеспечивает оптимальность ввода-вывода.

5.1. Сортировка на основе слияния

Напомню основные характеристики модели вычислений во внешней памяти, которую я упоминал в главе 1. Модель состоит из внутренней памяти размером M и допускает поблочный доступ к диску путем одновременного чтения/записи B элементов *(страницы диска)*.

Сортировка слиянием — один из самых известных алгоритмов сортировки, базирующийся на парадигме «разделяй и властвуй» [3]. Ее псевдокод приведен

в алгоритме 5.1. На шаге 1 проверяется, состоит ли массив хотя бы из двух элементов, ведь когда в нем всего один элемент, сортировка не требуется. Если же обнаруживается несколько элементов, алгоритм разбивает входной массив S пополам и для каждой части выполняет рекурсию. По завершении рекурсии обе части, $S[i, m-1]$ и $S[m, j]$, упорядочиваются таким образом, что на шаге 5 процедура `MERGE` объединяет их в массив $S[i, j]$. `HEAPSORT` и некоторые варианты `QUICKSORT` изменяют массив S непосредственно, однако `MERGESORT` не относится к такому классу алгоритмов: для работы процедуры `MERGE` нужен вспомогательный массив размером n, и стало быть, `MERGESORT` требуется дополнительное рабочее пространство $\Theta(n)$.

Алгоритм 5.1. Двоичная сортировка слиянием — MERGESORT(S, i, j)

```
1: if i < j then
2:     m = (i + j)/2;
3:     MERGESORT(S, i, m − 1);
4:     MERGESORT(S, m, j);
5:     MERGE(S, i, m, j);
6: end if
```

С учетом того, что при каждом рекурсивном вызове размер входного массива для сортировки уменьшается вдвое, общее количество рекурсивных вызовов составит $O(\log n)$. Процедуру `MERGE` можно реализовать за время $O(j - i + 1)$ с помощью двух указателей, скажем x и y, которые начинаются с головы каждой половины, $S[i, m-1]$ и $S[m, j]$. Затем $S[x]$ сравнивается с $S[y]$, меньший элемент записывается в объединенную последовательность, и соответствующий указатель сдвигается. Так как каждое сравнение двигает один указатель, общее количество шагов ограничено сверху общим количеством перемещений указателя, которое, в свою очередь, ограничено сверху длиной массива $S[i, j]$. Таким образом, временную сложность алгоритма `MERGESORT`(S, 1, n) можно смоделировать с помощью рекуррентного соотношения $T(n) = 2T(n/2) + O(n) = O(n \log n)$. Это хорошо известно из любого базового курса алгоритмов, например [3][1].

Теперь предположим, что $n > M$, то есть массив S должен храниться на диске. В этом случае самым важным вычислительным ресурсом для анализа и минимизации становятся операции ввода-вывода. На практике на каждый ввод-вывод в среднем затрачивается 5 мс. Если предположить, что каждое сравнение элементов требует одной операции ввода-вывода, время выполнения алгоритма `MERGESORT` для большого массива S можно оценить как 5 мс × ($n \log n$), немного округлив нотацию «О большое». При n порядка нескольких гигабайт, скажем $n \approx 2^{30}$, что на самом деле не так уж и много для размера памяти обычных современных ПК, предыдущая оценка времени составит не менее $5 \times (2^{30} \times 30) > 10^8$ мс. То есть вычисления должны занять

[1] Везде в книге, если основание логарифма явно не указано, его следует считать равным 2.

более одного дня. Однако если запустить алгоритм MERGESORT на обычном ПК, выяснится, что он завершится менее чем за час. Это неудивительно, поскольку в своей оценке я полностью игнорировал существование внутренней памяти размером M, а также последовательный шаблон доступа к памяти, используемый алгоритмом MERGESORT. Чтобы получить более реальный результат, проанализируем этот алгоритм в рамках двухуровневой модели памяти.

Прежде всего отмечу, что $O(z/B)$ операций ввода-вывода — это стоимость слияния двух упорядоченных последовательностей, состоящих в общей сложности из z элементов. Это справедливо при $M \geqslant 3B$, поскольку процедура MERGE в алгоритме 5.1 сохраняет во внутренней памяти, во-первых, две страницы диска с двумя элементами, на которые нацелены указатели, проходящие по массиву $S[i, j]$, где $z = j - i + 1$, а во-вторых, еще одну страницу диска для записи отсортированной выходной последовательности, которая сбрасывается на диск после каждого заполнения. Переход указателя на другую страницу диска вызывает промах. Во внутреннюю память извлекается новая страница, и слияние продолжается. Поскольку S хранится на диске постоянно, массив $S[i, j]$ занимает $O(z/B)$ страниц диска, и это граница ввода-вывода для слияния двух подпоследовательностей общего размера z. Аналогично как $O(z/B)$ оценивается количество операций ввода-вывода при записи объединенной последовательности, поскольку эта процедура выполняется от наименьшего к наибольшему элементу $S[i, j]$ через вспомогательный массив размером z. В результате рекуррентное соотношение для сложности ввода-вывода алгоритма MERGESORT можно записать как $T(n) = 2T(n/2) + O(n/B) = O(n/B \log n)$.

Но эта формула не до конца объясняет, почему на практике алгоритм MERGESORT так хорошо себя ведет, ведь в ней не учитывается иерархия памяти. При рекурсивном разбиении последовательности S генерируются все меньшие и меньшие подпоследовательности, требующие сортировки. Как только подпоследовательность длиной z начинает помещаться во внутреннюю память, то есть складывается ситуация $z = O(M)$, она полностью кэшируется операционной системой за $O(z/B)$ операций ввода-вывода. После этого для последующих шагов сортировки операций ввода-вывода уже не требуется. В результате количество операций ввода-вывода при сортировке подпоследовательности из $z = O(M)$ элементов равно не $\Theta(z/B \log z)$, как получено в предыдущем рекуррентном соотношении, а $O(z/B)$, ведь теперь учитывается только стоимость загрузки подпоследовательности во внутреннюю память. Такая экономия происходит на всех подпоследовательностях S размером $\Theta(M)$, для которых рекурсивно выполняется алгоритм MERGESORT, что в сумме составляет $\Theta(n/M)$. Таким образом, общая экономия составляет $\Theta(n/B \log M)$, что дает для алгоритма MERGESORT оценку в $\Theta(n/B \log n/M)$ операций ввода-вывода.

Результат особенно интересен тем, что полученная граница связывает сложность ввода-вывода алгоритма MERGESORT не только с размером страницы на диске B, но и с размером внутренней памяти M, то есть с функцией *кэширования*, предоставляемой программе сортировки операционной системой. Более того, эта граница предполагает три очевидные оптимизации классического псевдокода алгоритма 5.1, которые я вам сейчас покажу.

5.1.1. Остановка рекурсии

Первая оптимизация — ввести пороговый размер подпоследовательности, например $j - i < cM$, где $c \leqslant 1$. Это вызывает остановку рекурсии, извлечение всей подпоследовательности во внутреннюю память и применение к ней сортировщика, работающего во внутренней памяти (рис. 5.1). Параметр c зависит от того, какое пространство занимает сам сортировщик. Напомню, что он должен работать полностью во внутренней памяти. Например, для алгоритмов `INSERTIONSORT` и `HEAPSORT` параметр c равен 1, а для `QUICKSORT` — близок к 1 (из-за использования *стека вызовов* во время рекурсии; см. подраздел 5.3.3). Для алгоритма `MERGESORT` значение этого параметра меньше 0,5 (из-за дополнительного массива, используемого функцией `MERGE`). В результате в приведенной ранее оценке количества операций ввода-вывода нужно заменить M на cM, поскольку рекурсия останавливается на cM элементах. Это дает оценку $\Theta(n/B \log n/cM)$. При асимптотическом анализе эта замена бесполезна, потому что c является константой, но она важна при оценке реальной производительности алгоритмов в сценариях с использованием внешней памяти, поскольку $c < 1$. Поэтому желательно, чтобы параметр c был как можно ближе к 1. Это уменьшает влияние логарифмического фактора на сложность ввода-вывода. Так что лучше отдавать предпочтение сортировщикам, работающим во внутренней памяти, таким как `HEAPSORT`.

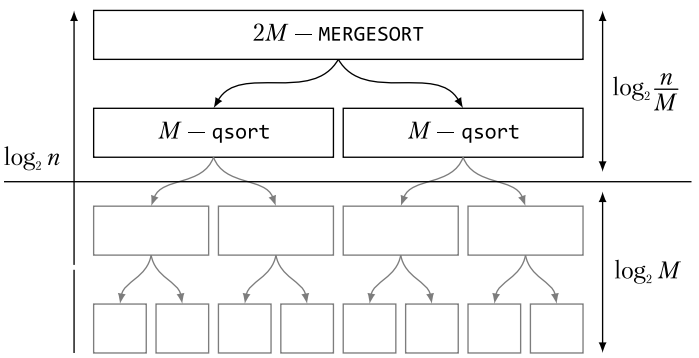


Рис. 5.1. Рекурсивный вызов алгоритма MERGESORT для подмассива размером $2M$

На рис. 5.1 происходит разделение на два подмассива размером M, к которым предлагается применить функцию `qsort` — эффективную реализацию алгоритма `QUICKSORT` во внутренней памяти. Ниже горизонтальной пунктирной линии изображены рекурсивные вызовы для подмассивов короче M. Они не порождают операций ввода-вывода, поскольку выполняются во внутренней памяти. Слева указано общее количество рекурсивных вызовов, выполненных алгоритмом `MERGESORT` (это $O(\log n)$, нотация «О большое» скрыта для простоты). Справа это количество делится между операциями с диском ($O(\log (n/M))$) и протекающими во внутренней памяти ($O(\log M)$).

Впрочем, при небольших M хорошим выбором может стать алгоритм `INSERTIONSORT` (и это действительно так), ведь в этом случае n элементов помещаются во внутреннюю память, то есть M соответствует размеру двух кэшей, L1 и L2, и составляет несколько мегабайт.

5.1.2. Техника снегоуборщика∞

Из формулы сложности ввода-вывода двоичного (бинарного) алгоритма `MERGESORT`, которая выглядит как $\Theta(n/B \log n/M)$, легко понять, что чем больше M, тем меньше число проходов слияния по данным. Очевидно, что эти проходы являются узким местом, снижающим эффективность алгоритма, особенно на дисках с невысокой пропускной способностью. Для борьбы с этим недостатком можно нарастить память или попытаться максимально использовать имеющийся ресурс. Проектировщики алгоритмов выбирают второй вариант. Поэтому сейчас я познакомлю вас с двумя методами, которые можно объединить, чтобы виртуально увеличить M.

Первая техника базируется на сжатии данных и опирается на тот факт, что последовательности становятся все более отсортированными. Поэтому вместо представления элементов с помощью кодов фиксированной длины, например 4 или 8 байт, мы воспользуемся методами *сжатия целых чисел*, которые позволяют упаковать элементы в меньшее количество битов, а значит, увеличить их количество во внутренней памяти или на странице диска. О том, как это делается, я подробно расскажу в главе 11, а пока упомяну некоторые из подходов к решению этой задачи: гамма- и дельта-коды Элиаса, коды Райса, коды Голомба и т. д. Также, если вспомнить, что чем меньше целое число, тем меньше битов требуется для его представления, можно обеспечить в отсортированных последовательностях малые целые числа, кодируя не столько абсолютные значения, сколько *разницу* между ними в отсортированной последовательности *(кодирование промежутков)*. Эта разница гарантированно неотрицательна, и она меньше, чем подлежащий кодированию элемент. Именно такой подход к кодированию целочисленных последовательностей используется в современных поисковых системах. Об этом мы тоже поговорим в главе 11.

В основу второй техники, позволяющей *виртуально* увеличить память *в среднем* в два раза, легла аналогия со снегоуборщиком, приписываемая Дональду Кнуту [5]. Мы сканируем входную последовательность S, генерируя отсортированные последовательности переменного размера. Размер каждой такой последовательности должен быть не меньше M, а в среднем составляет $2M$. Схема сортировки в данном случае будет другой, поскольку алгоритм сначала создает отсортированные последовательности переменной длины, а затем многократно применяет к ним процедуру `MERGE`. Несмотря на разницу длин последовательностей, `MERGE`, как обычно, потратит оптимальное количество операций ввода-вывода. Чтобы вдвое сократить число итераций для слияния, хватит $O(n/B)$ операций ввода-вывода, то есть для создания полностью упорядоченной последовательности их количество будет оцениваться как $O(n/B \log n/2M)$. По сути, мы экономим один проход по данным, что существенно для очень длинных S.

Для простоты предположим, что с диска в память элементы передаются по одному, а не блоками. Поскольку алгоритм сканирует входную последовательность, очевидно, что этот процесс по-прежнему использует оптимальное количество операций ввода-вывода $O(n/B)$.

На каждом этапе генерируется отсортированная последовательность, что наглядно представлено на рис. 5.2, а реализующий такой подход псевдокод вы найдете в алгоритме 5.2. Сначала внутренняя память заполняется M элементами (несортированными), которые хранятся в *куче* $\mathcal{H}$. Поскольку реализация куч на основе массива не требует дополнительного пространства, количество элементов, которые там можно разместить, ограничено только количеством доступных ячеек памяти. Мы сканируем входную последовательность S, на каждом шаге снимаем с вершины кучи $\mathcal{H}$ минимальный элемент — назовем его `min` — и загружаем в память следующий элемент — назовем его `next`. Поскольку нам нужен отсортированный вывод, при `next` < `min` сохранять `next` в $\mathcal{H}$ нельзя, иначе в куче появится новый минимум, на следующем шаге его потребуется снять и вывести, и порядок выходных данных нарушится. Поэтому элемент `next` сохраняется во вспомогательный массив $\mathcal{U}$, который остается несортированным и также хранится во внутренней памяти. На протяжении всей итерации общий размер $\mathcal{H}$ и $\mathcal{U}$ равен M. Итерация заканчивается при пустой куче $\mathcal{H}$. Это означает, что в массиве $\mathcal{U}$ оказывается M несортированных элементов. Следующая итерация может начаться с перемещения M элементов из $\mathcal{U}$ в новую min-кучу $\mathcal{H}$, после чего массив $\mathcal{U}$ оказывается пустым.

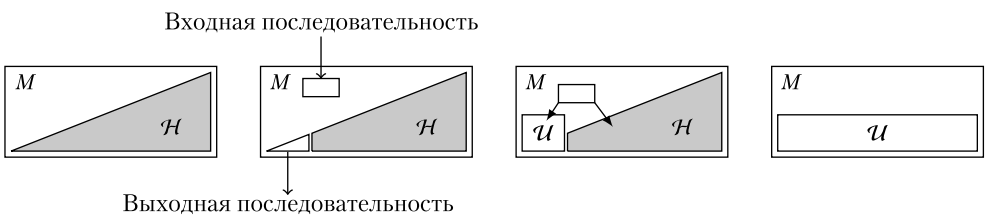


Рис. 5.2. Четыре основных шага одной итерации техники снегоочистителя

Крайний левый фрагмент на рис. 5.2 показывает M несортированных элементов, присутствующих в памяти, которые сгруппированы в min-кучу $\mathcal{H}$. На следующем базовом шаге ввода-вывода минимальный элемент кучи $\mathcal{H}$ записывается на диск, а из S извлекается новый элемент. Этот элемент в зависимости от текущего минимума кучи может попасть в $\mathcal{U}$ либо в $\mathcal{H}$. Справа показано условие завершения итерации — пустая куча $\mathcal{H}$, в то время как массив $\mathcal{U}$ целиком заполняет внутреннюю память.

Следует отметить два момента. Во-первых, при выполнении входящих в итерацию операций минимум $\mathcal{H}$ не уменьшается, соответственно, не убывает и выходная последовательность. Во-вторых, элементы, сохраненные в кучу $\mathcal{H}$ в начале итерации, в итоге выводятся до ее окончания. Первое наблюдение демонстрирует корректность алгоритма, а из второго следует, что в результате генерируются отсортированные последовательности длиннее чем M, что делает конечный алгоритм не менее эффективным, чем `MERGESORT`.

Алгоритм 5.2. Одна итерация техники снегоочистителя

Требуется: неотсортированный массив $\mathcal{U}$ из M элементов.

```
1:  Строим из элементов U min-кучу H;
2:  Присваиваем U = Ø;
3:  while H ≠ Ø do
4:    min = извлекаем из H минимум;
5:    Записываем min в выходную последовательность;
6:    next = читаем из входной последовательности следующий элемент;
7:    if next < min then
8:      вставляем next в U;
9:    else
10:     вставляем next в H;
11:   end if
12: end while
```

Более того, в среднем этот алгоритм эффективнее алгоритма MERGESORT. Пусть на одной итерации из последовательности S считывается τ элементов. Инициализация цикла while, который начинается на шаге 3, перемещает M элементов массива $\mathcal{U}$ в min-кучу $\mathcal{H}$ (шаг 1 алгоритма 5.2). Цикл while заканчивается, когда эта куча оказывается пустой, и мы снова имеем $|\mathcal{U}| = M$. Известно, что следующие τ элементов последовательности S частично попадают в кучу $\mathcal{H}$, а частично — во вспомогательный массив $\mathcal{U}$. При этом попадающие в массив $\mathcal{U}$ элементы на этой же итерации никогда из него не удаляются, а так как $|\mathcal{U}| \leqslant M$, можно сделать вывод, что M из τ элементов оказываются в $\mathcal{U}$. Следовательно, в кучу $\mathcal{H}$ попадает $(\tau - M)$ элементов, и именно они в конечном счете записываются в выходную (отсортированную) последовательность. Таким образом, длина этой последовательности в конце итерации составляет $M + (\tau - M) = \tau$. Первое слагаемое здесь учитывает элементы кучи в начале итерации, а второе — элементы, прочитанные из последовательности S и вставленные в кучу в процессе итерации. Осталось определить среднее значение τ, что легко сделать, если предположить случайное распределение входных элементов. В этом случае вероятность того, что next < min, составит $1/2$. Таким образом, каждый элемент имеет равную вероятность попасть в кучу или во вспомогательный массив, и в среднем как в $\mathcal{H}$, так и в $\mathcal{U}$ попадает $\tau/2$ элементов. Но, как мы помним, элементы в $\mathcal{U}$ равны M, поэтому из равенства $M = \tau/2$ получим $\tau = 2M$.

Факт 5.1. Техника снегоочистителя создает $O(n/M)$ отсортированных последовательностей, каждая из которых длиннее M и фактически имеет среднюю длину $2M$. Использующий эту технику для генерации таких последовательностей алгоритм MERGESORT имеет среднюю сложность ввода-вывода $O(n/B \log n/2M)$[1].

[1] Напоминаем, что MERGESORT продолжает оставаться очень эффективным по вводу-выводу в ситуации, когда размер отсортированной подпоследовательности превышает M. — *Примеч. науч. пер.*

5.1.3. От бинарного слияния к множественному

Во время предыдущей оптимизации количество уровней рекурсии сокращалось за счет увеличения размера начальных (отсортированных) последовательностей при условии использования внутренней памяти размером M. Слияние при этом было *бинарным*, поскольку оно обрабатывало две входные последовательности одновременно. Именно это обеспечило основание 2 логарифма сложности ввода-вывода в алгоритме `MERGESORT`. Попробуем существенно увеличить основание логарифма, увеличив количество одновременно обрабатываемых последовательностей так, чтобы для этого было достаточно памяти размером M. При слиянии двух последовательностей во внутренней памяти используется только три блока размером B: два для кэширования текущих страниц диска, содержащих сравниваемые элементы $S[x]$ и $S[y]$, и один для кэширования выходных элементов, которые сбрасываются на диск после заполнения блока, чтобы обеспечить поблочную запись на диск объединенной последовательности. Но во внутренней памяти гораздо больше блоков, то есть $M/B >> 3$, которые в процессе слияния никак не используются.

Это означает, что есть возможность дальнейшей оптимизации: нужен алгоритм, который обрабатывает одновременно k последовательностей при $k >> 2$. В частности, пусть $k = (M/B) - 1$, то есть для поблочного чтения k отсортированных входных последовательностей будет доступно k блоков, а еще один блок зарезервируем для поблочной записи объединенной отсортированной последовательности на диск. Это сложная алгоритмическая задача, поскольку на каждой итерации нужно выбирать минимальный элемент из k входных отсортированных последовательностей, и, если сделать это методом перебора за время $\Theta(k)$, мы потеряем в производительности. Нужно решение поэффективнее, и здесь нам снова поможет min-куча. Запись одного элемента в выходной блок в итоге будет занимать время $O(\log k)$. Наша куча содержит k пар (по одной на входную последовательность), где первый компонент — это элемент для сравнения, а второй — индекс последовательности. Изначально это минимальные элементы из k последовательностей, поэтому пары имеют вид $\langle R_i[1], i\rangle$, где $R_i[1]$ обозначает первый элемент i-й отсортированной последовательности при $i = 1, 2... k$.

Алгоритм множественного слияния проиллюстрирован на рис. 5.3. На каждой итерации извлекается пара, содержащая текущий наименьший элемент кучи (задан первым компонентом), который записывается в выходной блок, после чего в кучу попадает следующий элемент последовательности. Например, для минимальной пары $\langle R_m[x], m\rangle$ алгоритм записывает $R_m[x]$ в выходной блок во внутренней памяти и вставляет в кучу следующую пару $\langle R_m[x + 1], m\rangle$ из той же последовательности R_m, если, конечно, в ней еще остались элементы. Когда элементов больше нет, извлеченную пару ничем не заменяют. Некэшированная страница диска с элементом $R_m[x + 1]$ вызывает промах. Такая страница извлекается во внутреннюю память, гарантируя, что для следующих B считываний из R_m операций ввода-вывода не потребуется. Очевидно, что для слияния k последовательностей общей длиной z требуются время $O(\log_2 k)$ на элемент и $O(z/B)$ операций ввода-вывода.

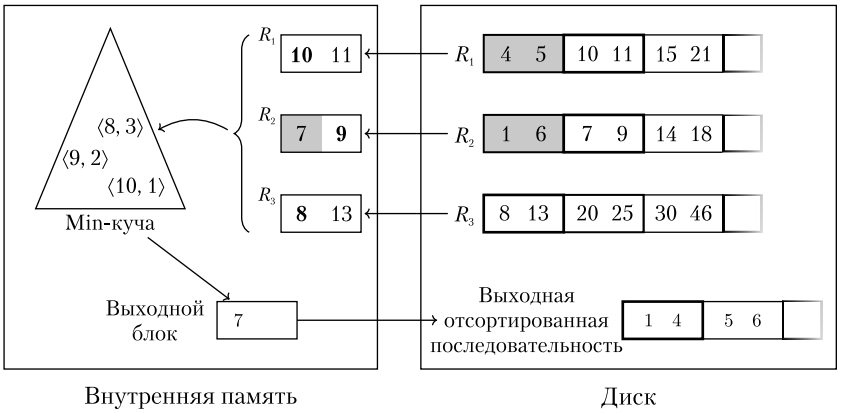


Рис. 5.3. Алгоритм множественного слияния для $k = 3$ отсортированных запусков, хранящихся на диске, страницы которого содержат $B = 2$ элемента

На рис. 5.3 серым выделены уже обработанные блоки, элементы которых записаны в выходную отсортированную последовательность на диске. Жирным обведены прямоугольники страниц, которые обрабатываются алгоритмом, то есть уже извлечены во внутреннюю память. В частности, первый элемент страницы R_2, то есть 7, уже записан в выходной блок во внутренней памяти, но пока не сброшен на диск, поскольку эта страница еще не заполнена. Куча хранит во внутренней памяти еще не обработанный первый элемент каждой последовательности (выделен жирным шрифтом) и его индекс в виде пары ⟨элемент, индекс⟩. Учитывая содержимое кучи, следующим в выходной блок внутренней памяти будет записан элемент 8, а в куче его заменит элемент из этой же последовательности 13. То есть в кучу попадет пара ⟨13, 3⟩.

В результате схема слияния k последовательностей напоминает дерево порядка k с $O(n/M)$ листьями (то есть с отсортированными последовательностями длиной не менее M), сформированное, например, с помощью техники снегоуборщика. Соответственно, общее количество уровней слияния оценивается как $O(\log_{M/B} n/M)$, а общий объем операций ввода-вывода — как $O(n/B \log_{M/B} n/M)$. Отмечу, что иногда в литературе эта формула выглядит как $O(n/B \log_{M/B} n/B)$, потому что $\log_{M/B} M$ можно записать как $\log_{M/B}(B(M/B)) = (\log_{M/B} B) + 1 = \Theta(\log_{M/B} B)$. Асимптотически это не имеет никакого значения, поэтому $\log_{M/B} n/M = \Theta(\log_{M/B} n/B)$.

Теорема 5.1. *При множественном слиянии алгоритм* `MERGESORT` *требует* $O(n/B \log_{M/B} n/M)$ *операций ввода-вывода и* $O(n \log n)$ *сравнений и времени для сортировки n атомарных элементов в двухуровневой модели памяти, в которой внутренняя память имеет размер M, а страница на диске — размер B.*

На практике количество уровней слияния будет небольшим. Скажем, для блока $B = 4$ Кбайт и памяти $M = 4$ Гбайт мы получим $M/B = 2^{32}/2^{12} = 2^{20}$, так что количество проходов будет в 20 раз меньше, чем при слиянии двух последовательностей алгоритмом `MERGESORT`. Куда интереснее тот факт, что за один проход можно индивидуально сортировать последовательности из M элементов, в то время как два

прохода позволяют сортировать последовательности из $(M/B)M = M^2/B$ элементов благодаря (M/B)-слияниям, что уже оказывается большим числом. Само собой разумеется, что на практике внутреннее пространство памяти, которое может быть выделено для сортировки, меньше доступной *физической* памяти (обычно мегабайт против гигабайта). Тем не менее для M = 128 Мбайт и B = 4 Кбайт размер M^2/B будет уже порядка терабайта. Кроме того, мы отмечаем, что использование техники снегоочистителя или сжатия целых чисел может виртуально увеличить значение M, дав *двукратное преимущество* в конечной сложности ввода-вывода, поскольку в границах ввода-вывода этот параметр фигурирует два раза.

5.2. Нижние границы

В начале этой главы я упоминал, что в RAM-модели задача сортировки сложнее задачи перестановки. Разница во временной сложности задается логарифмическим множителем. Сейчас же мне хотелось бы ответить на вопрос, существует ли аналогичная разница в количестве операций ввода-вывода. Я покажу, что, как ни странно, для большинства ситуаций с разумными параметрами n, M и B сортировка с точки зрения объема ввода-вывода *эквивалентна* перестановке. Этот удивительный результат можно интерпретировать как утверждение, что стоимость ввода-вывода при сортировке складывается не из затрат на *вычисление* отсортированной перестановки, а из затрат на *перемещение* данных на диске. В этом разделе вы найдете математическое доказательство и количественную оценку популярного выражения «бутылочное горлышко ввода-вывода».

Прежде чем углубляться в доказательство этой нижней границы, посмотрим, как с помощью сортировщика поменять порядок элементов в последовательности $S[1, n]$ в соответствии с заданной перестановкой $\pi[1, n]$. Это позволит оценить верхнюю границу операций ввода-вывода, достаточных для того, чтобы решить задачу перестановки для любого экземпляра $\langle S, \pi\rangle$. Напомню, что это означает генерацию последовательности $S[\pi[1]], S[\pi[2]]... S[\pi[n]]$. В RAM-модели есть возможность, перемещаясь между элементами S в соответствии с перестановкой π, получить новую последовательность $S[\pi[i]]$, для i = 1, 2... n, за оптимальное время $\Theta(n)$. На диске же фактически работают два разных алгоритма с несопоставимым количеством операций ввода-вывода. Первый имитирует происходящее в оперативной памяти, тратя на перемещение одного элемента одну операцию ввода-вывода. В худшем случае он производит $\Theta(n)$ таких операций. Второй алгоритм отвечает за генерацию надлежащего набора кортежей и последующую их сортировку. Он создает последовательность $\mathcal{P}$ из пар $\langle i, \pi[i]\rangle$, где i — это позиция элемента $S[\pi[i]]$, заданного вторым компонентом. Затем алгоритм сортирует эти пары по π и при параллельном сканировании массивов S и $\mathcal{P}$ меняет $\pi[i]$ на $S[\pi[i]]$, создавая новые пары $\langle i, S[\pi[i]]\rangle$. После этого происходит еще одна сортировка, на этот раз по первому компоненту этих пар. Здесь параллельное сканирование массивов S и $\mathcal{P}$ применяется для записи $S[\pi[i]]$ в $S[i]$, что дает последовательность упорядоченных нужным образом элементов. В целом второй алгоритм проводит два сканирования и две операции сортировки, поэтому, согласно теореме 5.1, он осуществляет $O(n/B \log_{M/B} n/M)$ операций ввода-вывода.

Это означает, что в зависимости от параметров n, M и B пользователь может выбрать алгоритм с минимальным количеством операций ввода-вывода. Эти соображения суммированы в табл. 5.1.

Таблица 5.1. Временная сложность и сложность ввода-вывода для задач перестановки и сортировки в двухуровневой модели памяти, где M — размер внутренней памяти, B — размер страницы диска, а $D = 1$ — количество доступных дисков. При большем количестве дисков n заменяется на n/D

Задача	Временная сложность (RAM-модель)	Сложность ввода-вывода (двухуровневая модель памяти)
Перестановка	$O(n)$	$O\left(\min\left\{n, \frac{n}{B}\log_{M/B}\frac{n}{M}\right\}\right)$
Сортировка	$O(n\log_2 n)$	$O\left(\frac{n}{B}\log_{M/B}\frac{n}{M}\right)$

Теорема 5.2. *В двухуровневой модели памяти с внутренней памятью размером M и страницей диска размером B перестановка n элементов требует $O(\min\{n, n/B \log_{M/B} n/M\})$ операций ввода-вывода.*

Далее я покажу, что при всей своей простоте с точки зрения ввода-вывода этот подход оптимален. Верхние границы для сортировки и перестановки асимптотически равны друг другу при $n = \Omega(n/B \log_{M/B} n/M)$. Так происходит при $B = \Omega(\log_{M/B} n/M)$, что всегда выполняется на практике, поскольку этот логарифмический член представляет собой очень маленькую константу для n входных данных объемом вплоть до эксабайтов. Поэтому программистам не нужно искать сложные стратегии, достаточно воспользоваться приведенной тут сортировкой.

5.2.1. Нижняя граница для сортировки

Остаются несколько тонких моментов, на которых мне не хотелось бы останавливаться слишком подробно, поэтому ограничусь кратким описанием того, на чем базируются нижние границы оценки ввода-вывода как для сортировки, так и для перестановки[1]. Для начала напомню, что для оценки нижних границ путем сравнения в RAM-модели использовалось *дерево решений* [3]. Алгоритм соответствует семейству таких деревьев, по одному на один размер входных данных (то есть их бесконечное число). Каждый узел представляет собой операцию сравнения двух элементов с двумя возможными результатами, соответственно, у каждого внутреннего узла будет два ветвления, то есть речь идет о бинарном дереве. Каждый лист дерева соответствует

[1] В процессе доказательства, как правило, исходят из двух предположений. Во-первых, что элементы не допускают разбиения, следовательно, не позволяется и хеширование, во-вторых, что их нельзя создавать/уничтожать/копировать, можно только перемещать что фактически подразумевает наличие единственной копии каждого элемента во время сортировки или перестановки.

решению базовой задачи. Например, при сортировке n элементов любой возможной перестановке входных данных соответствует один лист, то есть дерево содержит $n!$ листьев[1]. Каждый путь от корня к листу в дереве решений соответствует одному вычислению, поэтому самый длинный путь дает наихудшее число сравнений, выполняемых алгоритмом. Для оценки нижней границы достаточно определить глубину самого мелкого двоичного дерева с указанным количеством листьев. Самое мелкое двоичное дерево с ℓ листьями — это идеально сбалансированное дерево[2], высота h которого такова, что $2^h \geqslant \ell$, следовательно, $h \geqslant \log_2 \ell$. В случае сортировки мы имеем $\ell = n!$, поэтому классическую нижнюю границу $h = \Omega(n\log_2 n)$ легко вывести, взяв логарифм обеих сторон уравнения и воспользовавшись формулой Стирлинга для факториальной функции (см. предыдущую сноску).

В двухуровневой модели памяти воспользоваться таким деревом решений уже сложнее. Здесь нужно учитывать вводы-выводы, помня, что информация, находящаяся во внутренней памяти, этих операций не требует. Каждый узел дерева решений в этом случае по-прежнему будет соответствовать одному вводу-выводу, а количество листьев все так же будет $n!$. А вот разветвление каждого внутреннего узла будет равно количеству *различных результатов сравнения*, которые один ввод-вывод может сгенерировать среди элементов как считываемых с диска (B), так и доступных во внутренней памяти $(M - B)$. Эти B элементов можно распределить среди других $M - B$ элементов, присутствующих во внутренней памяти[3], не более чем $\binom{M}{B}$ способами. Следовательно, для этих сравнений один ввод-вывод может сгенерировать не более чем $\binom{M}{B}$ различных результатов.

Но это не окончательный ответ, потому что пока не учтены перестановки среди этих элементов. Подсчитаны только перестановки элементов, которые уже перешли во внутреннюю память, то есть были извлечены во время предыдущих вводов-выводов. Еще нужно подсчитать перестановки среди новых элементов, которые пока находятся на страницах диска. Существует n/B таких страниц, а следовательно, n/B операций ввода-вывода при обращении к новым элементам. Эти операции могут генерировать $\binom{M}{B}(B!)$ различных результатов, сравнивая новые B элементов (подлежащие сортировке) с $M - B$ элементами во внутренней памяти.

---

[1] Напомню, что факториал определяется как $n! = n(n-1)(n-2)\times\times\times 2 \times 1$. Известно, что $n! = \Theta\left((n/e)^n \sqrt{2\pi n}\right)$ — это формула Стирлинга для приближенного вычисления факториала.

[2] В литературе встречается два определения идеально сбалансированного дерева: в одном требуется, чтобы глубина поддеревьев отличалась не более чем на 1, в другом — чтобы количество узлов в поддеревьях отличалось не более чем на 1. Нижняя оценка справедлива для обоих определений. — *Примеч. науч. ред.*

[3] Чтобы убедиться в верности биномиального коэффициента, предположим, что $M - B$ элементов во внутренней памяти уже отсортированы. То есть читаемые с диска B элементов распределяются среди $M - B$ отсортированных элементов таким образом, чтобы получилась последовательность из M отсортированных элементов. Чтобы подсчитать, сколькими способами это может произойти, достаточно учесть тот факт, что это соответствует выбору B позиций из M доступных.

Рассмотрим задачу с t вводами-выводами, то есть путь в дереве решений с t узлами. Для генерации окончательной перестановки доступ к входным элементам, которые нужно прочитать, должны получить n/B узлов. Остальные $t - n/B$ узлов читают страницы с уже обработанными элементами. Такую форму имеет любой путь от корня к листьям, поэтому дерево решений можно рассматривать как имеющее наверху узлы для новых вводов-выводов, а внизу — все прочие узлы. Для дерева глубиной t количество листьев составит по крайней мере $\binom{M}{B}^t (B!)^{n/B}$. Исходя из предположения, что это число больше или равно $n!$, и взяв логарифм обеих сторон неравенства, получим $t = \Omega(n/B \log_{M/B} n/M)$. Несложно повторить эти рассуждения для случая с D дисками и прийти к следующей теореме.

Теорема 5.3. *В двухуровневой модели памяти с внутренней памятью размером M, размером страницы диска B и D дисками алгоритм сортировки на основе сравнения будет выполнять $\Omega(n/DB \log_{M/B} n/DB)$ операций ввода-вывода.*

Интересно отметить, что число доступных дисков D не фигурирует в знаменателе основания логарифма, хотя в знаменателе всех остальных членов мы его наблюдаем. Его наличие там уменьшило бы основание логарифма, соответственно, выросла бы нижняя граница сложности ввода-вывода. В свете сказанного в теореме 5.1 при одном диске оптимальным по количеству операций ввода-вывода и времени работы будет алгоритм `MERGESORT` со множественным слиянием. Но, согласно теореме 5.3, при нескольких дисках этот алгоритм теряет свою оптимальность, поскольку одновременное слияние $k > 2$ последовательностей оптимально при $O(n/(DB))$ операциях ввода-вывода. Такое возможно, только если алгоритм извлекает за один ввод-вывод D страниц, то есть по одной с каждого диска. Текущая схема слияния не может гарантировать такого на всех итерациях, каким бы ни было распределение k последовательностей по D дискам. Даже если мы знаем, какие элементы будут загружены в кучу $\mathcal{H}$ следующими, может случиться так, что на одном диске окажется больше чем B этих элементов, что потребует более одного ввода-вывода с этого диска, препятствуя D-мерному параллелизму в операции чтения.

В разделе 5.4 я покажу метод решения этой проблемы с помощью *чередования дисков*, которое позволяет приблизиться к оптимальному пределу ввода-вывода только за счет размещения данных на дисках, а также применения алгоритма `GREEDSORT`, обеспечивающего полную оптимальность благодаря элегантной и сложной схеме слияния.

5.2.2. Нижняя граница для перестановки

Предположим, что в любой момент времени глобальная память нашей модели (то есть внутренняя память размером M и неограниченный диск) содержит перестановку входных элементов, возможно перемежающихся пустыми ячейками. Во время выполнения алгоритма непустыми будут не более n блоков, поскольку n шагов (и, следовательно, операций ввода-вывода) — очевидная верхняя граница

сложности ввода-вывода в случае перестановки. Эта граница получена путем имитации на диске алгоритма перестановки для RAM-модели в начале этой главы. Пусть P_t — максимальное количество перестановок, генерируемых любым алгоритмом с t операций ввода-вывода. Учитывая предыдущее наблюдение, можно утверждать, что $t \leqslant n$ и $P_0 = 1$, поскольку начальной перестановкой у нас служит порядок ввода. Оценим P_t и примем, что $P_t \geqslant n!$. Это позволит определить минимальное количество шагов t, необходимое любому алгоритму для реализации любой возможной перестановки n выходных элементов.

Напомню, что в отличие от сортировки образец перестановки, которую нужно реализовать, предоставляется на входе, так что вычислять в этом случае ничего не требуется. Причем у нас есть три типа операций ввода-вывода, которые вносят разный вклад в количество сгенерированных перестановок.

- **Чтение новой страницы.** В этом случае требуется учитывать перестановки среди прочитанных с этой страницы элементов. Это дает значение $B!$. Кроме того, должны учитываться и перестановки, которые эти B элементов могут образовать в процессе своего распределения среди $M - B$ элементов, уже присутствующих во внутренней памяти (вы видели это в примере с сортировкой). Таким образом, P_t может увеличиться в $O\left(\binom{M}{B}(B!)\right)$ раз. Количество «нетронутых» страниц равно n/B. После чтения со страницы она становится «затронутой».
- **Повторное чтение страницы.** Ранее произведенные операции чтения-записи уже учены в P_t, соответственно, новая операция чтения может увеличить этот параметр только на коэффициент $O\binom{M}{B}$, появляющийся из-за перемешивания элементов B с уже имевшимися во внутренней памяти $M - B$ элементами. Количество затронутых страниц не может превышать n, так как это верхняя граница количества шагов, выполняемых алгоритмом перестановки.
- **Запись на страницу.** Страницу, сбрасываемую из внутренней памяти на диск, можно расположить не более чем $n + 1$ возможными способами среди доступных на диске непустых страниц, количество которых не превышает n. Таким образом, операция записи может увеличить P_t в $O(n)$ раз. Любая страница, на которую была сделана запись, относится к затронутым. Напомню, что в любой момент процесса перестановки количество таких страниц не превышает n.

Если t_r — это количество выполненных алгоритмом перестановки операций чтения, а t_w — количество операций записи, где $t = t_r + t_w$, то P_t можно ограничить следующим образом («О большое» опущено для удобства чтения формул):

$$P_t \leq \left(\frac{n}{B}\binom{M}{B}(B!)\right)^{n/B}\left(n\binom{M}{B}\right)^{t_r - n/B} n^{t_w} \leqslant \left(n\binom{M}{B}\right)^{t}(B!)^{n/B}. \qquad (5.1)$$

В этой формуле каждый коэффициент умножен на количество возможных способов, которыми страница может участвовать в операции чтения или записи. Это n/B

для чтения с нетронутых страниц и максимум n для чтения с затронутых страниц и записи.

Чтобы сгенерировать все возможные перестановки n входных элементов, нужно соблюдение условия $P_t \geqslant n!$. С учетом уравнения (5.1) это означает, что должно выполняться неравенство $\left(n\binom{M}{B}\right)^t (B!)^{n/B} \geqslant n!$. Разрешив его относительно t, получим:

$$t = \Omega\left(\frac{n\log\frac{n}{B}}{B\log\frac{M}{B}+\log n}\right).$$

Следует различать два случая. При $B\log M/B \leqslant \log n$ это уравнение превращается в $t = \Omega\left(\frac{n\log n/B}{\log n}\right) = \Omega(n)$. В противном случае оно имеет вид $t = \Omega\left(\frac{n\log n/B}{B\log M/B}\right) =$

$= \Omega\left(n/B\log_{M/B} n/M\right)$. Как и в случае с алгоритмом сортировки, это доказательство несложно распространить на вариант с D дисками. В целом это означает, что мы доказали следующее.

Теорема 5.4. *В двухуровневой модели памяти с внутренней памятью размером M, размером страницы диска B и D дисками перестановка n элементов требует $\Omega\left(\min\left\{n/D, n/DB\log_{M/B} n/DB\right\}\right)$ операций ввода-вывода.*

Теоремы 5.2–5.4 доказывают асимптотическую оптимальность верхних границ ввода-вывода, представленных в табл. 5.1, для сортировки и перестановки. На самом деле, как я уже отмечал, при $B = \Omega(\log_{M/B} n/M)$ они асимптотически эквивалентны. С учетом значений B и M на современных компьютерах (это десятки килобайтов и по крайней мере десятки гигабайтов соответственно) это равенство справедливо практически для любых, даже очень больших значений n. Поэтому неудивительно, что проектировщики алгоритмов при настройке ввода-вывода обычно рассматривают сортировку и перестановку как аналогичные операции.

5.3. Сортировка на основе распределения

Основу алгоритма `QUICKSORT`, как и `MERGESORT`, составляет парадигма «разделяй и властвуй», потому что он работает, разделяя сортируемый массив на две части, которые затем сортируются рекурсивно. Но, в отличие от алгоритма `MERGESORT`, здесь дополнительное рабочее пространство в явном виде не выделяется, отсутствует этап *слияния*, а процесс *разделения* сложен и влияет на общую эффективность. Алгоритм 5.3 содержит псевдокод, которым я буду пользоваться для оценки сложности `QUICKSORT` и демонстрации некоторых оптимизаций и сложностей, возникающих при реализации этой сортировки в иерархической памяти.

Алгоритм 5.3. Бинарный поиск QUICKSORT(S, i, j)

```
1: if i < j then
2:     r = выбираем положение хорошего опорного элемента;
3:     меняем местами S[r] и S[i];
4:     p = PARTITION(S, i, j);
5:     QUICKSORT(S, i, p − 1);
6:     QUICKSORT(S, p + 1, j);
7: end if
```

Основная идея заключается в разбиении входного массива $S[i,j]$ на две части таким образом, чтобы элементы одной части были *меньше* элементов другой. Это разбиение с сохранением порядка, поскольку после двух рекурсивных вызовов дополнительных шагов для повторного объединения упорядоченных частей не требуется. Процесс начинается с выбора так называемого опорного элемента, после сравнения с которым остальные элементы распределяются в два подмассива (шаг 4). Элементы, равные опорному, могут попасть в любой из них. В псевдокоде опорный элемент принудительно размещен в первой позиции $S[i]$ сортируемого массива (шаги 2 и 3)[1]. Для этого перед вызовом процедуры `PARTITION`(S, i, j) выбранный опорный элемент $S[r]$ меняется местами с $S[i]$. Процесс выбора опорного элемента пока оставлен за кадром, так как подробно он будет рассматриваться в подразделе 5.3.2.

Обратите внимание на то, что процедура `PARTITION`(S, i, j) возвращает позицию p, занимаемую опорным элементом после разбиения $S[i,j]$, — именно она будет управлять двумя последующими рекурсивными вызовами.

Есть два тонких момента, влияющих на эффективность алгоритма `QUICKSORT`. Это реализация процедуры `PARTITION`(S, i, j) и соотношение размеров двух подмассивов. Чем лучше они *сбалансированы*, тем ближе `QUICKSORT` к `MERGESORT`, а значит, и к оптимальной временной сложности $\Theta(n \log n)$. В случае полного дисбаланса, например при одном пустом подмассиве (то есть при $p = i$ или $p = j$), временная сложность `QUICKSORT` составляет $\Theta(n^2)$ — почти как у алгоритма `INSERTIONSORT`. Далее я подробно прокомментирую оба этих момента.

5.3.1. Разбиение на три части

Процедура `PARTITION`(S, i, j) делит входной массив на две части, одна из которых содержит элементы меньше опорного, а вторая — элементы больше опорного. Элементы, равные опорному, могут быть произвольно распределены между двумя частями. Поэтому входной массив подвергается такой перестановке, чтобы меньшие элементы оказались перед опорным, который, в свою очередь, должен

[1] Тем самым, строго говоря, нарушая порядок. — *Примеч. науч. ред.*

предшествовать большим элементам. В конце процедуры PARTITION(S, i, j) опорный элемент оказывается в $S[p]$, меньшие элементы — в $S[i, p-1]$, а большие — в $S[p+1, j]$. Существует несколько способов такого разделения за оптимальное время $O(n)$, но каждый из них по-разному пользуется кэшем, поэтому на практике эти способы имеют разную производительность. Алгоритм 5.4 демонстрирует сложный псевдокод, который учитывает наличие элементов, равных опорному, и фактически реализует разбиение на три части. Такие элементы сохраняются в специальном подмассиве, расположенном посередине.

Алгоритм 5.4. Разбиение на три части: PARTITION(S, i, j)

```
1:  P = S[i]; l = i; r = i + 1;
2:  for (c = r; c ⩽ j; c++) do
3:    if S[c] = P then
4:        меняем местами S[c] и S[r];
5:        r++;
6:      else if S[c] < P then
7:        меняем местами S[c] и S[l];
8:        меняем местами S[c] и S[r];
9:        r++; l++;
10:   end if
11: end for
12: return ⟨l, r − 1⟩;
```

Очевидно, что центральный подмассив с элементами, равными опорному, можно убрать из последующих рекурсивных вызовов аналогично тому, как мы отбрасываем опорный элемент. Это уменьшает количество подлежащих сортировке элементов, но требует изменений в классическом варианте псевдокода алгоритма 5.3, поскольку процедура PARTITION теперь должна возвращать не только позицию p опорного элемента, но и пару индексов, которые ограничивают центральный подмассив.

Алгоритм 5.4 демонстрирует реализацию тройного разбиения $S[i, j]$. Мы видим три указателя, которые перемещаются по этому массиву вправо и поддерживают следующий инвариант: P — *опорная точка,* $S[c]$ — *элемент, в данный момент сравниваемый с* P, *а* $S[i, c-1]$ — *уже обработанная часть входного массива*. В частности, $S[i, c-1]$ состоит из трех частей: $S[i, l-1]$ содержит элементы, меньшие чем P, $S[l, r-1]$ — элементы, равные P, а $S[r, c-1]$ — элементы, большие чем P. Любой из этих подмассивов может оказаться пустым[1]. В первом шаге псевдокода 5.4 P

[1] В нашем варианте алгоритма центральный подмассив (с опорной точкой) пустым быть не может, однако такое случается в других модификациях QUICKSORT. — *Примеч. науч. ред.*

инициализируется первым элементом разделяемого массива, а l и r получают значения, гарантирующие пустоту меньшей/большей части, в то время как центральная часть состоит только из элемента P. Затем алгоритм сканирует $S[i, j]$, пытаясь сохранить инвариант. При $S[c] > P$ для этого достаточно расширить часть больших элементов, увеличив r. В остальных случаях, то есть при $S[c] \leqslant P$, нужно поставить $S[c]$ на правильное место среди элементов $S[i, r-1]$, тогда инвариант — разбиение $S[i, c]$ на три части — сохранится. Здесь у меня для вас хорошая новость: это можно реализовать за постоянное время с помощью максимум двух перестановок, как показано на рис. 5.4 и продемонстрировано в шагах 3–9 алгоритма 5.4.

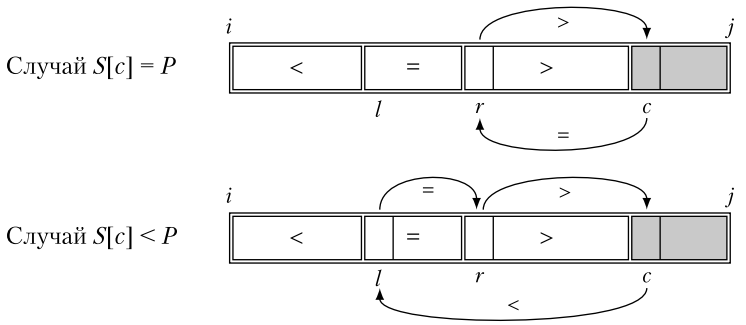


Рис. 5.4. Операции обмена в двух случаях. Стрелки показывают, в каком порядке надо менять местами опорный элемент с другими элементами массива

Алгоритм трехчастного разбиения занимает время $O(n)$ и обладает двумя положительными свойствами. Во-первых, это потоковый доступ к массиву S, благодаря которому процессор может предварительно считывать элементы заранее. Во-вторых, мы можем исключить из последующих рекурсивных вызовов элементы подмассива $S[l, r-1]$, равные опорному, поскольку они уже находятся в корректной позиции.

5.3.2. Выбор опорного элемента

Выбор опорного элемента имеет решающее значение для получения сбалансированных подмассивов, сокращения количества рекурсивных вызовов и достижения оптимальной временной сложности $O(n \log n)$. В псевдокоде алгоритма 5.3 не было выбора опорного элемента, поскольку это можно сделать по-разному и каждый вариант имеет свои плюсы и минусы. Например, если взять в качестве опорного первый элемент входного массива (случай $r = i$), выбор будет быстрым, но легко придумать экземпляр входного массива, порождающий несбалансированные подмассивы, — достаточно взять S как возрастающую или убывающую упорядоченную последовательность элементов. К такой проблеме может привести любой детерминированный выбор.

Чтобы не попасть в такую ситуацию, выбирать опорный элемент в $S[i, j]$ можно *случайным образом*. Правда, при этом поведение алгоритма становится непредсказуемым,

так как от этого случайного выбора зависят многие вещи. Можно показать, что *ожидаемая* временная сложность является оптимальной $O(n \log_2 n)$ с мультипликативной константой меньше 2. Если к этому присовокупить эффективность использования пространства алгоритмом `QUICKSORT` (об этом мы поговорим в подразделе 5.3.3), то на практике описанный подход становится очень привлекательным.

Теорема 5.5. *Случайный выбор опорного элемента приводит к тому, что в соответствии с математическим ожиданием алгоритм* `QUICKSORT` *будет сравнивать не более чем* $2n \ln n$ *элементов.*

Доказательство. При правильном подходе доказать эту теорему несложно. Нас интересует количество сравнений, которые процедура `PARTITION` выполняет над входной последовательностью S. Введем случайную бинарную переменную $X_{u,v}$, показывающую, сравнивает ли процедура `PARTITION` элементы $S[u]$ и $S[v]$. Вероятность этого события обозначим $p_{u,v}$. Учитывая линейность математического ожидания, количество сравнений, выполненных алгоритмом `QUICKSORT`, будет определяться формулой:

$$E\left[\sum_{u,v} X_{u,v}\right] = \sum_{u=1}^{n} \sum_{v=u+1} E\left[X_{u,v}\right] = \sum_{u} \sum_{v>u} 1 p_{u,v} + 0 \cdot \left(1 - p_{u,v}\right) = \sum_{u=1}^{n} \sum_{v=u+1}^{n} p_{u,v}.$$

Для оценки $p_{u,v}$ рассмотрим ситуацию со случайным выбором опорного элемента $S[r]$, ведь процедура `PARTITION` выполняет сравнение только с опорным элементом. Требуется рассмотреть три случая. Когда $S[r]$ меньше или больше, чем $S[u]$ и $S[v]$, эти два элемента не сравниваются друг с другом, а передаются в тот же рекурсивный вызов `QUICKSORT`. В результате проблема возникает снова, но уже на меньшем подмножестве элементов, содержащем как $S[u]$, так и $S[v]$. Такой случай для оценки $p_{u,v}$ неинтересен, поскольку в этой рекурсивной точке невозможно сделать вывод о том, сравнивались ли $S[u]$ и $S[v]$. Если же какой-то из элементов, $S[u]$ или $S[v]$, является опорным, они гарантированно сравниваются процедурой `PARTITION`. Во всех остальных случаях опорный элемент берется из элементов массива S, значение которых оказывается строго между $S[u]$ и $S[v]$. Поэтому эти два элемента попадают в два разных подмассива (отсюда два разных рекурсивных вызова `QUICKSORT`) и никогда не будут сравниваться.

Все это означает, что для вычисления $p_{u,v}$ нужно рассматривать две последние ситуации. В них два варианта выбора предоставляют «хорошие» случаи (то есть $S[u]$ и $S[v]$ сравниваются), а b выборов остается на «плохие» случаи. Здесь b — количество элементов в массиве S, значение которых попадает строго между $S[u]$ и $S[v]$ (сравнения не происходит). Чтобы оценить b, рассмотрим отсортированную версию массива S, которую обозначим S'. Существует очевидное взаимно однозначное отображение между парами элементов в S' и S. Предположим, $S[u]$ отображается в $S'[u']$, а $S[v]$ — в $S'[v']$. Тогда b легко вывести как $v' - u' - 1$. Таким образом, вероятность сравнения элементов $S[u]$ и $S[v]$ составит $p_{u,v} = 2/(b + 2) = 2/(v' - u' + 1)$.

Эта формула может показаться сложной, поскольку слева у нас u, v, а справа — v', u'. Учитывая взаимно однозначное отображение между S и S', можно изменить утверждение «рассматривая все пары (u, v) в S» на «рассматривая все пары (u', v') в S'» и, таким образом, записать предыдущее суммирование как:

$$\sum_{u=1}^{n}\sum_{v=u+1}^{n} p_{u,v} = \sum_{u'=1}^{n}\sum_{v'>u'}^{n} \frac{2}{v'-u'+1} = 2\sum_{u'=1}^{n}\sum_{k=2}^{n-u'+1}\frac{1}{k} \leqslant 2\sum_{u'=1}^{n}\sum_{k=2}^{n}\frac{1}{k} \leqslant 2n\ln n,$$

где окончательное неравенство вытекает из свойств n-го гармонического числа. ■

Следующий вопрос заключается в том, как обеспечить ожидаемое поведение. Это можно сделать, например, выборкой более чем одного опорного элемента. Обычно из S случайным образом выбираются три опорных элемента и берется медиана, что требует всего двух сравнений. Более трех опорных элементов делают «хороший» выбор более надежным [2], как доказывает теорема 5.6.

Теорема 5.6. *Если алгоритм* `QUICKSORT` *разделяет массив по медиане* $2s + 1$ *случайно выбранных элементов, то n различных элементов он сортирует за* $\frac{2nH_n}{H_{2s+2} - H_{s+1}} + O(n)$ *ожидаемых сравнений, где* H_n *— номер n-го гармонического числа* $\sum_{i=1}^{n} 1/i$.

Увеличивая s, можно приблизить ожидаемое число сравнений к $n \log n + O(n)$, однако выбор медианы требует дополнительных ресурсов. Фактически его можно реализовать или путем сортировки s выборок за время $O(s \log s)$ и выбора медианы, находящейся в средней позиции $s + 1$ упорядоченной последовательности, или с помощью сложного алгоритма (о нем упоминается в [3]) за время $O(s)$ в худшем случае. Рандомизация упрощает выборку и по-прежнему гарантирует ожидаемое время выполнения $O(s)$. Этот подход я рассмотрю подробно, поскольку его анализ элегантен, а алгоритмическая структура довольно универсальна и может применяться для выбора не только медианы неупорядоченной последовательности, но и элемента любого ранга k.

Алгоритм 5.5 рандомизирован и выбирает элемент неупорядоченного массива S с рангом k. Что интересно, алгоритмическая схема в данном случае похожа на ту, которую мы применяли на этапе разбиения в алгоритме `QUICKSORT`. Здесь выбранный элемент $S[r]$ играет ту же роль, что и опорный элемент в `QUICKSORT`. Он используется для разбиения входной последовательности S на три части, состоящие из элементов, меньших чем $S[r]$, или равных $S[r]$, или больших чем $S[r]$. Правда, в отличие от `QUICKSORT` алгоритм `RANDSELECT` рекурсивно выполняет только одну из этих трех частей, а именно ту, которая содержит элемент с рангом k. Это можно определить, просто посмотрев на размеры этих частей, как сделано на шагах 6 и 8. Есть два момента, на которые стоит обратить внимание. Во-первых, нет нужды рекурсивно обходить $S_=$, потому что он полностью состоит из элементов, равных $S[r]$, соответственно, именно такое значение имеет элемент ранга k. Во-вторых, в процессе рекурсии на подмассиве $S_>$ требуется обновлять ранг k, поскольку из исходной

последовательности удаляются элементы, принадлежащие множеству $S_> \cup S_=$. Корректность алгоритма в данном случае очевидна, поэтому остается вычислить его ожидаемую временную сложность. Здесь мы имеем оптимальный вариант $O(n)$, ведь последовательность S не отсортирована, соответственно, для поиска элемента с рангом k нужно проверять все n ее элементов.

Алгоритм 5.5. Выбор элемента с рангом k: RANDSELECT(S, k)

```
1:  r = случайное положение в {1, 2... n};
2:  S< = элементы S, которые меньше чем S[r];
3:  S> = элементы S, которые больше чем S[r];
4:  n< = |S<|;
5:  n= = |S| − (|S<| + |S>|);
6:  if k ⩽ n< then
7:     return RANDSELECT(S<, k);
8:  else if k ⩽ n< + n= then
9:     return S[r];
10: else
11:    return RANDSELECT(S>, k − n< − n=);
12: end if
```

Теорема 5.7. *Выбор элемента с рангом k в неупорядоченной последовательности размера n занимает $O(n)$ ожидаемого времени в RAM-модели и $O(n/B)$ ожидаемых операций ввода-вывода в двухуровневой модели памяти.*

Доказательство. Назовем хорошим выбор, который индуцирует разбиение с $n_<$ и $n_>$ не больше $2n/3$. Размер $S_=$ нас не волнует, поскольку, когда этот подмассив содержит искомый элемент, тот немедленно возвращается как $S[r]$. Нетрудно заметить, что для соблюдения условий $n_< \leqslant 2n/3$ и $n_> \leqslant 2n/3$ элемент $S[r]$ должен иметь ранг в диапазоне $[n/3, 2n/3]$. Вероятность этого составляет $1/3$, так как элемент $S[r]$ выбирается из S равномерно случайным образом (шаг 1). Обозначим ожидаемую временную сложность алгоритма RANDSELECT при запуске на массиве $S[1, n]$ как $\hat{T}(n)$. Можно записать:

$$\hat{T}(n) \leq O(n) + \frac{1}{3}\hat{T}(2n/3) + \frac{2}{3}\hat{T}(n),$$

где первый линейный член учитывает временную сложность шагов 2–5, второй член учитывает ожидаемую временную сложность рекурсивного вызова при хорошем выборе опорной точки, а третий член является грубой верхней границей ожидаемой временной сложности рекурсивного вызова при плохом выборе опорной

точки, что на самом деле тоже относится к случаю рекурсии по всему массиву S. Это особое рекуррентное соотношение, поскольку член $\hat{T}(n)$ встречается с обеих сторон неравенства, при этом он содержит различные константы. Соотношение можно упростить, вычтя одинаковые члены и получив $\frac{1}{3}\hat{T}(n) \leqslant O(n) + \frac{1}{3}\hat{T}(2n/3)$, что дает $\hat{T}(n) \leqslant O(n) + \hat{T}(2n/3) = O(n)$. В двухуровневой модели памяти уравнение принимает вид $\hat{T}(n) \leqslant O(n/B) + \hat{T}(2n/3) = O(n/B)$, так как построение трех подмножеств можно выполнить за один проход по n входным элементам, для чего будет достаточно $O(n/B)$ операций ввода-вывода. ■

Алгоритм `RANDSELECT` можно разными способами использовать в алгоритме `QUICKSORT`. Например, выбрать опорным элементом медиану массива S, установив $k = n/2$, или выбрать медиану среди избыточного набора опорных элементов $2s + 1$, установив $k = s + 1$, где $s << n/2$. Наконец, с помощью этого алгоритма можно выбрать опорный элемент, который генерирует сбалансированное разбиение при разных размерах частей, являющихся долями n, скажем, αn и $(1 - \alpha)n$ при $\alpha < 0{,}5$. Последний вариант $k = \lfloor \alpha n \rfloor$ кажется бесполезным, поскольку трехстороннее разбиение по-прежнему занимает время $O(n)$, но проблема в том, что оно увеличивает количество рекурсивных вызовов с $\log_2 n$ до $\log_{1/(1-\alpha)} n$. В то же время в рассуждении не учитывался тот факт, что при отсутствии событий, которые *прерывают* поток инструкций, значительно замедляя вычисления, современные процессоры реализуют конвейерный параллелизм или параллелизм на уровне инструкций. Особенно сильно влияет *ошибочное предсказание ветвлений* при выполнении процедуры `PARTITION`(S, i, j), которое может возникнуть при появлении элемента, меньшего или равного опорному. Сокращение количества таких предсказаний увеличивает параллелизм на уровне команд современных ЦП [4].

Исходя из этих соображений в 2012 году для библиотеки среды выполнения Java 7 от Oracle в качестве стандартного метода сортировки простых типов был выбран новый вариант алгоритма `QUICKSORT`. Решение об изменении было принято после того, как эмпирические исследования показали, что в среднем новый алгоритм работает быстрее старого. Улучшение обеспечила новая стратегия трехстороннего разбиения, основанная *на паре опорных точек, правильно размещенных* во входной последовательности S. Исследования показали, что в результате ожидаемое количество сравнений сократилось за счет увеличения числа перестановок [9]. Несмотря на этот компромисс, стратегия с двумя опорными точками более чем на 10 % превысила по скорости работы классическую реализацию `QUICKSORT`, потому что в то время неправильные предсказания ветвлений требовали большего количества ресурсов, чем доступ к памяти.

Этот пример замечательно показывает, как классические алгоритмы и проблемы, известные десятилетиями и считающиеся устаревшими, могут оказаться предвестниками инноваций и глубокого/нового теоретического анализа. Так что никогда не теряйте любопытства и не ленитесь исследовать и анализировать новые алгоритмические схемы!

5.3.3. Ограничение дополнительного рабочего пространства

Алгоритм QUICKSORT часто называют сортировщиком *на месте*, потому что ему не требуется дополнительное пространство для упорядочения массива S. Для псевдокода из алгоритма 5.3 это действительно так, но если посчитать ресурсы, затрачиваемые на рекурсивные вызовы, ситуация меняется. Фактически при каждом рекурсивном вызове локальные переменные вызывающей функции должны сохраняться до тех пор, пока не случится возврат из него; память для этого выделяет операционная система. Каждому рекурсивному вызову требуется пространство $\Theta(1)$, и эту оценку необходимо умножить на количество вложенных вызовов, которые могут понадобиться алгоритму QUICKSORT для массива $S[1, n]$. В худшем случае эта оценка может быть $\Theta(n)$, что означает необходимость дополнительного рабочего пространства $\Theta(n)$ на плохих входных данных (например, уже отсортированных, которые вызывают полностью несбалансированные разбиения).

Способ обхода такого поведения показан в алгоритме 5.6. На первый взгляд он кажется загадочным, но в его основе лежит довольно остроумный и элегантный принцип проектирования. В первую очередь отмечу, что цикл while выполняется только для входного массива длиннее n_0. Если это не так, то на шаге 13 вызывается известный своей эффективностью для очень коротких последовательностей алгоритм INSERTIONSORT. Значение n_0 обычно устанавливается в несколько десятков элементов.

Алгоритм 5.6. Двоичный алгоритм QUICKSORT с ограниченной рекурсией: BOUNDEDQS(S, i, j)

```
1:  while j – i > n0 do
2:     r = выбираем положение хорошего опорного элемента;
3:     меняем местами S[r] и S[i];
4:     p = PARTITION(S, i, j);
5:     if p ⩽ (i + j)/2 then
6:        BOUNDEDQS(S, i, p – 1);
7:        i = p + 1;
8:     else
9:        BOUNDEDQS(S, p + 1, j);
10:       j = p – 1;
11:    end if
12: end while
13: INSERTIONSORT(S, i, j);
```

Для более длинных входных массивов выполняется модифицированная версия классического бинарного алгоритма QUICKSORT, комбинирующая один рекурсивный вызов с итеративным циклом while. Обоснованием для такого рефакторинга

служит тот факт, что правильность классического алгоритма `QUICKSORT` не зависит от порядка рекурсивных вызовов, поэтому их можно перетасовать[1] таким образом, чтобы первый всегда выполнялся для меньшей из двух интересующих нас частей трехстороннего разбиения. Именно это гарантирует оператор `if` на шаге 5. Более того, длинная часть вообще не обрабатывается на этой итерации цикла `while` и остается до следующей, в которой параметры i и j оказываются равными границам этой более длинной части. Такая модификация рекурсивных алгоритмов хорошо известна в литературе по компиляторам и называется *устранением хвостовой рекурсии*. В результате рекурсивный вызов выполняется на подмассиве, размер которого не превышает половины входного массива. Это гарантирует верхнюю границу $O(\log n)$ для количества рекурсивных вызовов и, следовательно, для размера дополнительного пространства, необходимых для управления ими.

Теорема 5.8. *В RAM-модели алгоритм* `BOUNDEDQS` *сортирует n атомарных элементов за ожидаемое время* $O(n \log n)$ *и требует для этого дополнительное рабочее пространство* $O(\log n)$.

В заключение отмечу, что стандарты C89 и C99 ANSI определяют алгоритм сортировки `qsort`, реализация которого включает большинство алгоритмических трюков, о которых шла речь ранее[2]. Это дополнительно демонстрирует эффективность сортировки на основе распределения в двухуровневой модели (кэш и оперативная память).

5.3.4. От бинарного к множественному алгоритму QUICKSORT

Сортировка на основе распределения противоположна сортировке на основе слияния, поскольку первая выполняется путем разбиения последовательностей с помощью опорных элементов и их последующего рекурсивного упорядочения, а вторая объединяет рекурсивно упорядоченные последовательности. При работе с набором дисков эффективность алгоритма `MERGESORT` достигалась слиянием нескольких отсортированных последовательностей. То же самое в этих условиях применяется для проектирования алгоритма `QUICKSORT`, который разбивает входную последовательность на $k = \Theta(M/B)$ подпоследовательностей с помощью $k - 1$ опорных элементов. Поскольку $k >> 1$, выбор этих опорных элементов представляет собой нетривиальную задачу, ведь получаемые в итоге k частей должны быть гарантированно сбалансированными, то есть содержать $\Theta(n/k)$ элементов каждая. В подразделе 5.3.2 я упоминал, как непросто выбрать даже один опорный элемент, так что сейчас мы рассмотрим еще более сложный случай.

---

[1] Третью часть, в которой все элементы равны опорному значению, менять уже не надо. — *Примеч. науч. ред.*

[2] На самом деле в основе алгоритма qsort лежит другая схема разбиения, использующая два итератора. Один движется по массиву S вперед, а другой — назад, при каждом обнаружении двух неупорядоченных элементов происходит обмен. Асимптотическая временная сложность при этом не меняется, но на практике это более практичный алгоритм, ведь количество обменов уменьшается, поскольку равные элементы не перемещаются.

Пусть $s_1, ..., s_{k-1}$ — опорные точки, используемые алгоритмом для разделения входной последовательности $S[1, n]$ на k частей, называемых также *корзинами* (buckets). Для наглядности возьмем две фиктивные опорные точки $s_0 = -\infty$ и $s_k = +\infty$ и обозначим i-й сегмент $B_i = \{S[j]: s_{i-1} < S[j] \leqslant s_i\}$. Нужно, чтобы $|B_i| = \Theta(n/k)$ для всех k сегментов. Это гарантировало бы достаточность $\log_k n/M$ фаз разбиения для получения подпоследовательностей короче M, которые сортируются во внутренней памяти и поэтому не требуют дополнительных операций ввода-вывода. Каждую фазу разбиения можно реализовать за $O(n/B)$ операций ввода-вывода. Организация памяти при этом будет противоположна той, которая используется для множественного алгоритма `MERGESORT`. Сейчас мы задействуем один входной блок для чтения из входной последовательности, подлежащей разделению, и k выходных блоков для записи в k формирующихся разделов. Из требования $k = \Theta(M/B)$ следует, что число фаз разбиения равно $\log_k n/M = \Theta(\log_{M/B} n/M)$, соответственно, если каждый шаг разбиения равномерно распределяет входные элементы между k блоками, для множественного алгоритма `QUICKSORT` ожидаемая оптимальная граница ввода-вывода будет $\Theta(n/B \log_{M/B} n/M)$.

Для эффективного поиска $(k - 1)$ хороших опорных точек применим быструю и простую рандомизированную стратегию на базе *избыточной выборки*. Ее псевдокод приведен в алгоритме 5.7. Объем выборки контролируется параметром $a \geqslant 0$, который на шаге 2 влияет на надежность процесса выбора, а также эффективность по времени. Если взять оптимальный сортировщик в памяти, такой как `HEAPSORT` или `MERGESORT`, по $\Theta(ak)$ выбранным элементам, получим оценку $O((ak) \log(ak))$.

Алгоритм 5.7. Поиск $k - 1$ хороших опорных точек через избыточную выборку

1: Берем $(a + 1)k - 1$ случайных элементов из входной последовательности;
2: Получаем из них отсортированную последовательность A;
3: Для $i = 1, ..., k - 1$ выбираем опорную точку $s_i = A[(a + 1)i]$;
4: **return** опорные точки s_i;

Из отобранных $\Theta(ak)$ кандидатов на опорные элементы нужно выбрать $(k - 1)$, причем они должны быть равномерно распределены, то есть отстоять друг от друга на $(a + 1)$. Я утверждаю, что эти $\Theta(ak)$ элементов дают точную картину распределения во всей входной последовательности, поэтому сбалансированный выбор $s_i = A[(a + 1)i]$ должен дать нам хорошие опорные элементы. Чем больше a, тем ближе размер всех блоков к $\Theta(n/k)$, но тем выше окажется стоимость сортировки выборок. В крайнем случае, при $a = n/k$, мы уже не сможем сортировать выборки во внутренней памяти. В то же время чем ближе a к нулю, тем быстрее происходит выбор опорного элемента, но тем выше вероятность получения несбалансированных разделов. Как будет показано в лемме 5.1, вариант $a = \Theta(\log k)$ достаточен для получения сбалансированных разделов с временными затратами на выбор опорной точки $O(k \log^2 k)$. Отмечу, что корзины будут не идеально сбалансированными,

а квазисбалансированными, поскольку с разумной вероятностью в них окажется не более чем $4n/k = O(n/k)$ элементов (множитель 4 не влияет на оценку целевого асимптотического времени и сложности ввода-вывода).

Лемма 5.1. *Пусть* $k \geqslant 2$ *и* $a + 1 = 12 \ln k$. *Выборки размером* $(a + 1)k - 1$ *достаточно для того, чтобы с вероятностью не менее* $1/2$ *гарантировать попадание во все корзины менее чем* $4n/k$ *элементов.*

Доказательство. Верхняя граница вероятности дополнительного события, указанного в лемме, а именно существования одной корзины размером больше $4n/k$, оценивается как $1/2$. Это соответствует выборке *по отказам*, которые вызываются несбалансированным разбиением. Чтобы получить такую оценку вероятности отказа, рассмотрим каскад событий, увеличивающих вероятность его возникновения. Для последнего события в этой цепочке можно зафиксировать явную верхнюю границу $1/2$. Как следствие, эта же верхняя граница будет корректной и для исходного события.

Как и в доказательстве теоремы 5.5, рассмотрим отсортированную версию входной последовательности S, которую обозначим S'. Логически разобьем S' на $k/2$ сегментов длиной $2n/k$ каждый. Нас интересует, существует ли для некоторого индекса i корзина B_i с попавшими в нее по крайней мере $4n/k$ элементами. Как показано на рис. 5.5, эта большая корзина полностью охватывает по крайней мере один сегмент. В рассматриваемом случае это сегмент t_2 (но это может быть любой сегмент последовательности S'), потому что первый содержит по крайней мере $4n/k$ элементов, тогда как в последнем $2n/k$ элементов.

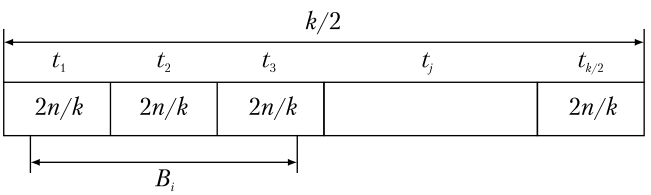


Рис. 5.5. Разбиение на сегменты отсортированной последовательности S'

Согласно определению корзины, ограничивающие B_i опорные точки s_{i-1} и s_i выходят за пределы сегмента t_2. Следовательно, по алгоритму 5.7 в сегмент, который перекрывает корзина B_i, попадает менее $(a + 1)$ образцов. В результате мы имеем:

$$\mathcal{P}(\exists B_i : |B_i| \geqslant 4n/k) \leqslant \mathcal{P}(\exists t_j : t_j \text{ содержит менее } (a + 1) \text{ образцов}) \leqslant$$
$$\leqslant (k/2)\mathcal{P}(\text{определенный сегмент содержит менее } (a + 1) \text{ образцов}), \quad (5.2)$$

где последнее неравенство вытекает из *неравенства Буля*, ведь $k/2$ — это число сегментов, составляющих последовательность S'. Поэтому в дальнейшем мы сосредоточимся на доказательстве верхней границы последнего члена.

Вероятность того, что один выбранный элемент заканчивается в данном сегменте, равна $\frac{(2n/k)}{n} = \frac{2}{k}$, поскольку предполагается, что из S (и, следовательно, из S') элементы извлекаются равномерно случайным образом. Пусть количество таких элементов X. Нас интересует $\mathcal{P}(X < a + 1)$. Поскольку берется $((a + 1)k - 1)$ элементов, $E[X] = ((a + 1)k - 1) \times 2/k = 2(a + 1) - 2/k$. Лемма предполагает, что $k \geqslant 2$, поэтому $E[X] \geqslant 2(a + 1) - 1$, что по крайней мере $3/2(a + 1)$ для всех $a \geqslant 1$. Разрешая неравенство $E[X] \geqslant 3/2\ (a + 1)$ относительно $(a + 1)$, получаем $a + 1 \leqslant (2/3)\ E[X] = (1 - (1/3))E[X]$. Эта форма заставляет вспомнить оценку Чернова:

$$\mathcal{P}(X < (1 - \delta)E[X]) \leqslant e^{-\left(\delta^2/2\right)E[X]}.$$

Приравняв $\delta = 1/3$, получим:

$$\mathcal{P}(X < (a + 1) \leqslant \mathcal{P}(X < (1 - (1/3))E[X]) \leqslant$$

$$\leqslant e^{-(E[X]/2)(1/3)^2} = e^{-E[X]/18} \leqslant e^{-(3/2)(a+1)/18} = e^{-(a+1)/12} = e^{-\ln k} = \frac{1}{k}. \qquad (5.3)$$

Здесь мы использовали неравенство $E[X] \geqslant (3/2)(a + 1)$ и предположение из леммы, что $a + 1 = 12 \ln k$. Подставляя результат уравнения (5.3) в уравнение (5.2), получаем $\mathcal{P}(\exists B_i : |B_i| \geqslant 4n/k) \leqslant (k/2)(1/k) = 1/2$, таким образом, лемма доказана. ■

5.4. Сортировка в случае нескольких дисков∞

Узким местом сортировки, использующей дисковую память, является время, необходимое для выполнения ввода-вывода. Для смягчения этой проблемы можно взять D параллельно работающих дисков, чтобы передавать DB элементов за один ввод-вывод. Это увеличивает пропускную способность подсистемы ввода-вывода, но сильно усложняет разработку эффективных с точки зрения ввода-вывода алгоритмов. Посмотрим, как это выглядит в контексте сортировки n атомарных элементов.

Самый простой подход к управлению параллельными дисками называется *чередованием дисков* и заключается в рассмотрении D дисков как *одного* с размером страницы $B' = DB$. С одной стороны, такой подход дает возможность использовать любой алгоритм, разработанный для одного диска, просто с размером страницы B', с другой — теряется независимость дисков, что отрицательно сказывается на сложности ввода-вывода в алгоритмах сортировки:

$$O\left(\frac{n}{B'}\log_{M/B'}\frac{n}{M}\right) = O\left(\frac{n}{DB}\log_{M/DB}\frac{n}{M}\right).$$

Такая граница далеко не оптимальна, поскольку основание логарифма в этом случае в D раз меньше, чем у нижней границы из теоремы 5.3. Соотношение между оптимальной границей и границей при чередовании дисков составит $1 - \log_{M/B} D$. Это означает, что по мере увеличения количества дисков, а именно $D \to M/B$, техника чередования дисков становится все менее эффективной.

Эффективно использовать *независимость* между D дисками сложно. Потребовалось несколько лет, чтобы разработать оптимальные алгоритмы, работающие с несколькими дисками и достигающие границ, указанных в теореме 5.3. Вопрос был в том, как гарантировать, что при обращении к дисковой подсистеме каждая из D операций чтения-записи будет выполняться со своего диска, ведь именно это обеспечивает скорость передачи элементов DB. При сортировке такая трудность возникает вне зависимости от того, каким механизмом пользуется сортировщик — распределения данных или слияния.

Для примера рассмотрим алгоритм `QUICKSORT`. Гарантировать пропускную способность позволяет равномерное распределение входных элементов между D дисками, например, по кругу, как показано на рис. 5.6. В этом случае для сканирования входных элементов хватит оптимальных $O(n/DB)$ операций ввода-вывода. На фазе распределения входная последовательность уже может считываться с этой скоростью ввода-вывода. А вот при записи выходных подпоследовательностей, полученных в процессе разбиения, возникают проблемы, ведь, чтобы сохранить инвариант для следующей фазы распределения, которая будет выполняться независимо для всех этих подпоследовательностей, нужно распределить подпоследовательности по дискам тоже по кругу. В случае D дисков у нас есть D выходных блоков, которые заполняются результатами разбиения. Заполненные блоки нужно записать на D отдельных дисков, чтобы обеспечить полный параллелизм, то есть единую операцию ввода-вывода. С учетом чередования последовательностей все выходные блоки, принадлежащие одной и той же последовательности, можно записать за одну операцию ввода-вывода. Но как правило, они принадлежат разным последовательностям, что может вызвать конфликты при попытке записи на один диск.

	Блок 1	Блок 2	Блок 3	Блок 4	Блок 5	
Диск 1	1 2	9 10	17 18	25 26	33 34	...
Диск 2	3 4	11 12	19 20	27 28	35 36	...
Диск 3	5 6	13 14	21 22	29 30	37 38	...
Диск 4	7 8	15 16	23 24	31 32	39 40	...

Рис. 5.6. Пример распределения последовательности элементов по $D = 4$ дискам, где $B = 2$

Рисунок 5.7 иллюстрирует ситуацию, когда на фазе разбиения в алгоритме `QUICKSORT` формируются три последовательности, предназначенные для записи на три диска. Последовательности чередуются по кругу, а затененные блоки соответствуют их начальным частям, уже записанным на диски. Стрелками обозначены следующие *свободные* блоки каждой последовательности, причем все они находятся на диске D_2. Это неудачная ситуация, потому что, если на фазе разбиения многопоточному

алгоритму QUICKSORT нужно записывать за один проход один блок, возникает конфликт и подсистеме ввода-вывода приходится *сериализовать* операцию записи в $D = 3$ отдельных ввода-вывода, теряя весь параллелизм.

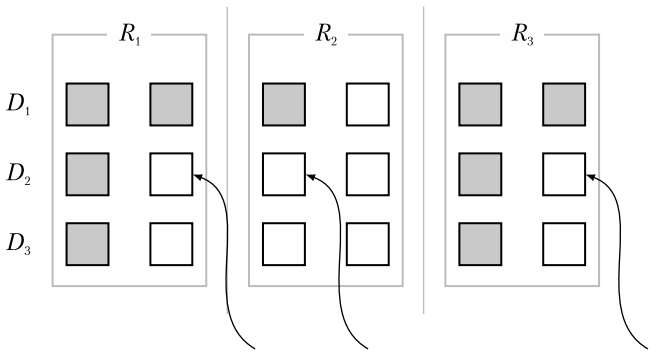


Рис. 5.7. Конфликт при записи $D = 3$ блоков, принадлежащих трем разным последовательностям

Для предотвращения такой ситуации были предложены варианты рандомизированных и детерминированных сортировщиков, выполняющих оптимальное количество операций ввода-вывода [8]. Я кратко расскажу про детерминированный многопоточный сортировщик GREEDSORT [7], который преодолевает описанные трудности с помощью элегантного подхода на основе слияния. На первом этапе происходит примерная сортировка элементов с помощью эффективной с точки зрения ввода-вывода процедуры многопоточного слияния, которая независимо работает с $R = \Theta\left(\sqrt{M/B}\right)$ отсортированными последовательностями, обеспечивая параллельную работу с дисками. Сортировку входной последовательности завершает алгоритм, который в ходе работы с *короткими* последовательностями длиной $O(M^{3/2})$ обходится линейным количеством операций ввода-вывода. Это разработанный Т. Лейтоном в 1985 году алгоритм COLUMNSORT. Корректность распределения достигается за счет того, что после первого этапа расстояние между неотсортированными элементами и их правильной отсортированной позицией оказывается меньше размера последовательностей, управляемых COLUMNSORT. На втором этапе это дает возможность полностью упорядочить приблизительно отсортированную последовательность за один оптимальный с точки зрения количества операций ввода-вывода проход.

Получить приблизительно отсортированную последовательность эффективным с точки зрения ввода-вывода способом помогает алгоритм GREEDSORT. Те, кому интересны подробности его функционирования, могут обратиться к литературе [7], я же опишу его вкратце. Так как предполагается, что отсортированные последовательности хранятся в чередующемся виде на D дисках (см. рис. 5.6), чтение D идущих подряд блоков из каждой последовательности происходит за одну операцию ввода-вывода. Как и в случае с уже знакомым вам алгоритмом QUICKSORT, при чтении чередующихся последовательностей снова возможны конфликты ввода-вывода.

Алгоритм GREEDSORT позволяет избежать этой проблемы, так как с каждым диском он работает независимо, извлекая по два *лучших* доступных блока. Слово «лучший» означает, что эти два блока содержат *наименьший минимальный элемент*, скажем m_1, и *наименьший максимальный элемент*, скажем m_2, причем два блока могут быть на самом деле одним и тем же. Очевидно, что такой выбор может независимо выполняться на D дисках, а для отслеживания минимальных/максимальных элементов в дисковых блоках требуется правильная структура данных. В литературе [7] показано, что эта структура данных может поместиться во внутренней памяти, так что дополнительных операций ввода-вывода для этого выбора не потребуется.

На рис. 5.8 показан пример диска *j*, который из-за хранения с чередованием содержит блоки нескольких последовательностей. Последовательность *1* содержит блок с наименьшим минимальным элементом 1, а последовательность *2* — блок с наименьшим максимальным элементом 7. Очевидно, что остальные блоки с фрагментами последовательностей *1* и *2* содержат элементы, превышающие 7. А блоки с фрагментами других последовательностей — минимум больше 1 и максимум больше 7. Алгоритм GREEDSORT объединяет эти два лучших блока диска *j* и создает два новых отсортированных блока. Первый записывается в следующий свободный блок выходной последовательности на диске *j* (он содержит элементы {1, 2, 3, 4}), а второй — обратно в последовательность с наименьшим минимумом m_1, а именно в последовательность *1* (он содержит элементы {5, 6, 7, 8}). Эта последняя запись не нарушает упорядоченную подпоследовательность, поскольку второй лучший блок содержит элементы, которые меньше максимума исходного блока m_1.

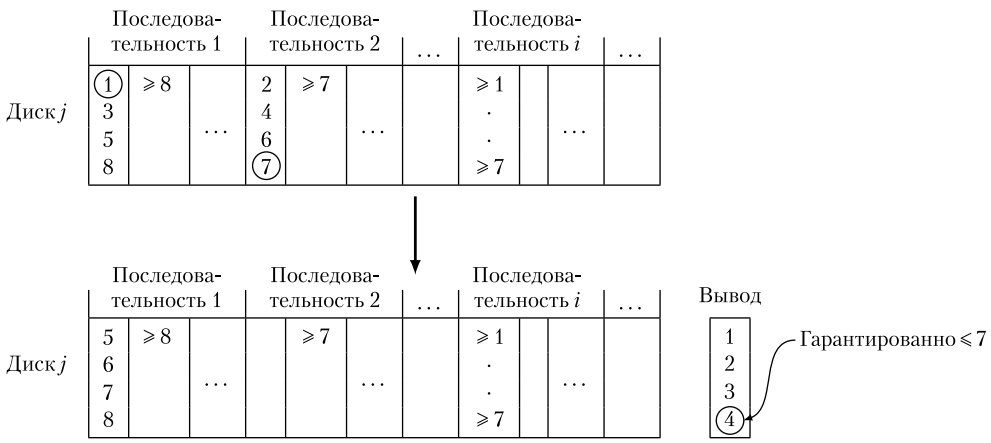


Рис. 5.8. Пример из [7]. *Верхний фрагмент:* диск *j* содержит блоки нескольких последовательностей, и лучшие два блока находятся в последовательности *1* (с наименьшим минимальным элементом) и последовательности *2* (с наименьшим максимальным элементом). *Нижний фрагмент:* после объединения лучших двух блоков диска *j* полученный первый блок записан в следующий свободный блок выходной последовательности на диске *j*, а второй блок — в последовательность *1* на диске *j*

Выбор двух лучших блоков идет независимо по всем дискам до тех пор, пока не будут проверены все входные последовательности, а их блоки не запишут в выходную последовательность, распределенную с чередованием по D дискам. В рассматриваемом примере стоит обратить внимание на тот факт, что элементы, записанные в выходных данных на диск j, могут и не являться четырьмя наименьшими элементами всех блоков на этом диске. Фактически в другой последовательности (отличной от *1* и *2*, но все еще находящейся на диске j) может оказаться блок с элементом в пределах, скажем, 2,5, минимум которого больше 1, а максимум больше 7. Таким образом, этот блок совместим с выбором двух лучших, но содержит элементы, которые должны быть сохранены в первом блоке отсортированной последовательности. Поэтому конечная последовательность, полученная в результате слияния, не сортируется, но при ее чтении с чередованием по всем D дискам происходит ее приблизительная сортировка. Это указано в следующей лемме, а доказано в [7].

Лемма 5.2. *Последовательность называется L-регрессивной, если расстояние между любой парой несортированных записей, скажем, ...y...x..., где y > x, меньше L. Предыдущий алгоритм сортировки создает L-регрессивный вывод* $L = RDB = D\sqrt{MB}$.

Так как $L = D\sqrt{MB} < DB\sqrt{M} < M^{3/2}$, алгоритм `COLUMNSORT`, примененный к скользящему окну из $2L$ элементов, которое на каждой итерации перемещается на L шагов вперед, даст полностью отсортированную последовательность, разбросанную по D дискам. Следовательно, инвариант для следующей фазы слияния сохраняется, требуя $O(n/DB)$ операций ввода-вывода. Поскольку количество упорядоченных последовательностей уменьшено в $R = \Theta(\sqrt{M/B})$ раз, общее количество фаз слияния оценивается как $O(\log_R n/M) = O(\log_{M/B} n/M)$, из чего следует оптимальное количество операций ввода-вывода при работе с D дисками.

Список литературы

1. *Aggarwal A., Vitter J. S.* The input/output complexity of sorting and related problems // Communications of the ACM, 31 (9): 1116–1127, 1988.
2. *Bentley J. L., Sedgewick R.* Fast algorithms for sorting and searching strings // Proceedings of the 8th ACM — SIAM Symposium on Discrete Algorithms (SODA), 360–369, 1997.
3. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to algorithms. The MIT Press, third edition, 2009.
4. *Kaligosi K., Sanders P.* How branch mispredictions affect quicksort // Proceedings of the 14th European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 4168, Springer, 780–791, 2006.
5. *Knuth D. E.* The art of computer programming. Vol. 3. Addison-Wesley, second edition, 1998.

6. *Leighton F. T.* Tight bounds on the complexity of parallel sorting // IEEE Transactions on Computers, C-34 (4), Special Issue on Sorting, 1985.
7. *Nodine M. H., Vitter J. S.* Greed sort: Optimal deterministic sorting on parallel disks // Journal of the ACM, 42 (4): 919–933, 1995.
8. *Vitter J. S.* External memory algorithms and data structures // ACM Computing Surveys, 33 (2): 209–271, 2001.
9. *Wild S., Nebel M. E.* Average case analysis of Java 7's Dual Pivot QuickSort // Proceedings of the 20th European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 7501, Springer, 826–836, 201.

Глава 6
ПЕРЕСЕЧЕНИЕ МНОЖЕСТВ

Делиться — значит заботиться!

В этой главе будет рассмотрена простая задача, связанная с множествами, которая в любой поисковой системе лежит в основе обработчика запросов. Такая система представляет собой инструмент, предназначенный для поиска информации в коллекции $\mathcal{D}$ документов. Я ограничусь *текстовыми* документами, подразумевая под документом $d_i \in \mathcal{D}$ книгу, новость, твит или любой файл, содержащий последовательность языковых токенов (слов). Для эффективных ответов на запросы пользователей поисковая система наряду со множеством других вспомогательных структур данных создает *индекс*. Запрос Q обычно структурируется как *мешок слов* (bag of words), скажем, $w_1w_2...w_k$, и цель поисковой системы — эффективно извлечь из $\mathcal{D}$ *наиболее релевантные* документы, содержащие все слова запроса. Специалисты в этой области сразу назовут это определение очень упрощенным, ведь современные поисковые системы ищут документы, включающие в себя *большинство* входящих в запрос слов, которые могут быть точными, содержать несколько опечаток или ссылаться на *синонимы* или *родственные слова*. Желательно, чтобы результаты были *релевантными* и имели отношение к введенному пользователем запросу Q. Впрочем, релевантность — довольно субъективное и меняющееся со временем понятие.

Так как поиск информации не является центральной и основной темой этой книги, тех читателей, которые хотели бы подробнее ознакомиться с этим вопросом, я отошлю к специализированной литературе ([4], [7]). Здесь же будет рассматриваться наиболее общий алгоритмический процесс работы с запросом в виде мешка слов.

> **Задача.** Даны последовательность слов $Q = w_1w_2...w_k$ и набор документов $\mathcal{D}$. Найдите в $\mathcal{D}$ документы, содержащие все слова w_i.

Первое, что приходит на ум, — сканировать каждый документ в наборе, отыскивая все указанные слова. Проблема в том, что для такого поиска требуется время, пропорциональное длине набора документов. Учитывая количество индексированных данных в Интернете, поиск займет уйму времени даже для суперкомпьютера или центра обработки данных. Поэтому для современных поисковых систем создана очень простая, но эффективная структура данных, называемая *инвертированным индексом*, ускоряющая поток миллиардов ежедневных пользовательских запросов.

Инвертированный индекс состоит из трех частей: словаря слов w, списка вхождений для каждого словарного слова, называемого *списком позиций*, который я обозначу $\mathcal{L}[w]$, и дополнительной информации, указывающей на важность каждого вхождения. Последнее требуется на этапах определения релевантности документа, но в этой главе обсуждаться не будет (см., например, [4], [7]). Термин «инвертированный» относится к тому факту, что вхождения слов сортируются в соответствии с алфавитным порядком слов, к которым они относятся, а не с их положением в документе. То есть инвертированные индексы похожи на классический глоссарий, который встречается в книгах. В данном случае он расширен для представления вхождений *всех* слов, присутствующих в коллекции документов, а не только самых важных.

Каждый список позиций $\mathcal{L}[w]$ хранится непрерывно в одном массиве, возможно на диске. Имена индексированных документов, фактически идентифицирующие их URL-адреса, помещаются в отдельную таблицу, где им сопоставляются уникальные положительные целые числа, известные как *docID*. Обычно эти идентификаторы произвольным образом назначает поисковая система[1]. Словарь хранится в таблице, которая содержит сопутствующую информацию и указатели на списки позиций. Основная структура инвертированного индекса показана на рис. 6.1.

Словарь	Список позиций
…	…
`abaco`	50, 23, 10
`abiura`	131, 100, 90, 132
`ball`	20, 21, 90
`mathematics`	15, 1, 3, 23, 30, 7, 10, 18, 40, 70
`zoo`	5, 1000
…	…

Рис. 6.1. Пример инвертированного индекса для фрагмента словаря. Списки позиций пока не упорядочены

Предположим, что запрос Q состоит из двух слов, `abaco` и `mathematics`. Поиск в наборе D документов, которые содержат оба слова, сводится к поиску *docID*, общих для инвертированных списков L[`abaco`] и L[`mathematics`], а именно 10 и 23. Это задача поиска *пересечения множеств*, которая и является ключевой темой главы.

В обоих списках позиций целые числа расположены в произвольном порядке, поэтому вычислить пересечение можно сравнением каждого *docID* $a \in \mathcal{L}$[`abaco`] со всеми *docID* $b \in \mathcal{L}$[`mathematics`]. Если $a = b$, то a вставляется в результирующий набор. Для списков длиной n и m полный перебор осуществляется за nm шагов/сравнений. В реальности, когда порядок n и m составляет миллионы, число шагов/сравнений будет иметь порядок $10^6 \times 10^6 = 10^{12}$. Даже на современном компьютере,

[1] Процесс назначения *docID* крайне важен для экономии места в хранилище списков позиций, но его реализация слишком сложна, чтобы обсуждать ее здесь (см. [6]).

способном выполнять миллиард сравнений в секунду (10^9 сравнений/с), запрос из двух слов такой тривиальный алгоритм будет обрабатывать 10^3 с, то есть около 10 мин, что слишком долго даже для терпеливого пользователя!

Впрочем, *docID*, встречающиеся в двух списках позиций, можно переставить в соответствии с некоторой структурой, ускоряющей поиск общих целых чисел. Ключевая идея заключается в однократной сортировке списков позиций (рис. 6.2). В результате формулировка задачи по поиску пересечения на двух упорядоченных множествах $A = \mathcal{L}$[abaco] и $B = \mathcal{L}$[mathematics] будет выглядеть следующим образом.

Поиск пересечения (упорядоченных) множеств. *Даны две отсортированные целочисленные последовательности* $A = a_1a_2...a_n$ *и* $B = b_1b_2...b_m$, *такие, что* $a_i < a_{i+1}$ *и* $b_i < b_{i+1}$. *Найдите общие для обоих множеств целые числа.*

Словарь	Список позиций
…	…
abaco	10, 23, 50
abiura	90, 100, 131, 132
ball	20, 21, 90
mathematics	1, 3, 7, 10, 15, 18, 23, 30, 40, 70
zoo	5, 1000
…	…

Рис. 6.2. Пример инвертированного индекса для части словаря с отсортированными списками позиций

Отмечу, что подходы, с которыми я вас сейчас познакомлю, допускают расширение на любые упорядоченные последовательности элементов, то есть работают не только с целыми числами. Целочисленные последовательности я взял для простоты.

6.1. Подход на основе слияния

Упорядоченность двух последовательностей позволяет разработать обманчиво простой, элегантный и быстрый алгоритм поиска пересечения множеств. Он сканирует A и B слева направо, на каждом шаге сравнивая *docID* двух списков. Назовем эти *docID* a_i и b_j, причем сначала $i = j = 1$. При $a_i < b_j$ увеличивается итератор i, а при $a_i > b_j$ — итератор j. В случае же, когда $a_i = b_j$, обнаружен общий *docID* и увеличиваются оба итератора.

Корректность такого подхода может быть доказана индуктивно на базе следующего наблюдения: если $a_i < b_j$ (второй случай симметричен), то a_i меньше, чем все элементы, следующие в B за b_j (благодаря упорядоченности), поэтому $a_i \notin B$. Для оценки временной сложности следует обратить внимание на тот факт, что на каждом шаге алгоритм выполняет одно сравнение и увеличивает по крайней мере один итератор.

Учитывая, что $n = |A|$ и $m = |B|$ — это количество элементов в двух последовательностях, очевидно, что i (соответственно, и j) может продвигаться вперед не более n раз (соответственно, не более m раз). Это означает, что наш алгоритм выполняет не более чем $n + m$ сравнений/шагов. *Не более*, потому что одна последовательность может исчерпаться раньше другой и оставшиеся элементы сравнивать будет уже не с чем. Как видите, временные затраты в этом случае значительно меньше, чем для неупорядоченных последовательностей (для которых было nm), так что реальное преимущество такого подхода очевидно. Фактически если снова взять n и m с порядком *docID* 10^6, то на компьютере, выполняющем 10^9 сравнений/с, для вычисления $A \cap B$ новому алгоритму потребуется 10^{-3} с, то есть порядка миллисекунд. Именно такую скорость мы наблюдаем у современных поисковых систем.

Возможно, внимательный читатель уже заметил, что этот алгоритм имитирует процедуру `MERGE` из алгоритма `MERGESORT`, которая в данном случае адаптирована для поиска общих элементов двух множеств A и B, а не для их слияния.

Теорема 6.1. *Алгоритм, основанный на парадигме слияния, ищет пересечение отсортированных множеств за время* $O(m + n)$.

Алгоритм оптимален при $n = \Theta(m)$, поскольку в этом случае обрабатывается самый маленький набор, соответственно, $\Omega(\min\{n, m\})$ — очевидная нижняя граница. Более того, эта основанная на сканировании парадигма будет оптимальна и в модели с дисковой памятью, поскольку требует $O(n/B)$ операций ввода-вывода. Точнее, она оптимальна независимо от иерархии памяти, в которой происходят вычисления *(кэш-независимая модель)*.

Следующий вопрос — что делать, когда m сильно отличается от n, скажем, $m << n$, то есть в ситуации, когда одно слово намного специфичнее второго. Здесь может помочь классический *бинарный поиск* в том смысле, что можно разработать алгоритм, выполняющий бинарный поиск каждого элемента $b \in B$ (а их всего несколько штук) во множестве упорядоченных элементов A. В этом случае потребуется $O(m \log n)$ шагов/сравнений. Такая временная сложность лучше, чем $O(n + m)$, когда $m = o(n/\log n)$. Это менее строгое и более точное условие, чем $m << n$.

Теорема 6.2. *Алгоритм, основанный на парадигме бинарного поиска, находит пересечение упорядоченных множеств за время* $O(m \log n)$.

На этом этапе естественно задаться вопросом: можно ли разработать алгоритм, который вберет в себя лучшее из парадигм на базе слияния и поиска? Фактически в парадигме бинарного поиска кроется неэффективность, которая становится очевидной при одинаковом порядке m и n: при поиске элемента b_i в A мы можем снова и снова перепроверять одни и те же элементы этого множества. Это, безусловно, касается среднего элемента A, скажем $a_{n/2}$, который в первую очередь проверяется любым бинарным поиском. Но при $b_i > a_{n/2}$ бесполезно сравнивать с $a_{n/2}$ элемент b_{i+1}, так как он по умолчанию будет больше, ведь $b_{i+1} > b_i > a_{n/2}$. Это справедливо и для всех последующих элементов B. Подобная аргументация применима и к остальным элементам A, проверяемым бинарным поиском. Поэтому рассмотрим другой вариант поиска пересечения множеств, позволяющий избежать ненужных сравнений.

6.2. Взаимное разбиение

Подойдем к поиску пересечения множеств, взяв за основу другую классическую парадигму — *разбиение*. Вы уже видели пример ее применения при проектировании алгоритма `QUICKSORT`, а сейчас я воспользуюсь ею для многократного взаимного разбиения двух упорядоченных множеств, для которых ищется пересечение [1]. Предположим, что $m \leqslant n$, причем оба числа четные. В качестве *опорного* возьмем медианный элемент $b_{m/2}$ самой короткой последовательности B и попробуем найти его в более длинной последовательности A. Здесь возможны два варианта. Если $b_{m/2} \in A$, скажем, $b_{m/2} = a_j$ для некоторого j, то $b_{m/2}$ возвращается как один из элементов пересечения $A \cap B$. Во втором же случае $b_{m/2} \notin A$, скажем, $a_j < b_{m/2} < a_{j+1}$ (мы предполагаем, что $a_0 = -\infty$ и $a_{n+1} = +\infty$). В обоих случаях алгоритм пересечения выполняется *рекурсивно* для каждой части, полученной разбиением последовательностей A и B по опорному элементу. В результате рекурсивно вычисляется $A[1, j] \cap B[1, m/2 - 1]$ и $A[j + 1, n] \cap B[m/2 + 1, m]$. В первом случае возможна небольшая оптимизация, состоящая в отбрасывании элемента $b_{m/2} = a_j$ из первого рекурсивного вызова.

Пример реализации показан на рис. 6.3, а псевдокод приведен в алгоритме 6.1. Медианный элемент B, который используется для взаимного разбиения двух упорядоченных последовательностей A и B, равен 12. Опорный элемент разбивает последовательность A на две несбалансированные части, $A[1, 7]$ и $A[16, 100]$. А последовательность B разбивается им почти пополам: $B[2, 10]$ и $B[16, 32]$. Так как опорный элемент встречается и в A, и в B, он возвращается как элемент пересечения. Обратите внимание на то, что первая часть A короче первой части B, поэтому в рекурсивном вызове они меняются ролями.

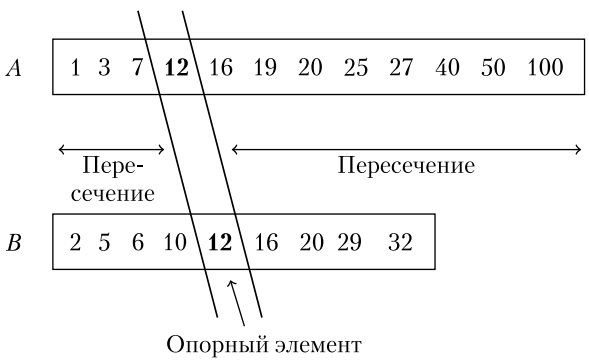


Рис. 6.3. Первый шаг парадигмы взаимного разбиения: опорный элемент 12 — это медианный элемент кратчайшей последовательности B. Он разбивает упорядоченную последовательность A на две части: {1, 3, 7} и {16, 19, 20, 25, 27, 40, 50, 100}. Эти две части рекурсивно пересекаются попарно с двумя частями B, разделенными опорным элементом 12

Алгоритм 6.1. Пересечение упорядоченных множеств на основе взаимного разбиения

1: Пусть $m = |B| \leqslant n = |A|$, в противном случае меняем A и B местами;
2: Выбираем в B медианный элемент $p = b_{\lfloor m/2 \rfloor}$;
3: Бинарный поиск положения p в A, например, $a_j \leqslant p < a_{j+1}$;
4: Рекурсивно вычисляем пересечение $A[1, j] \cap B[1, m/2 - 1]$;
5: **if** $p = a_j$ **then**
6: выводим p;
7: **end if**
8: Рекурсивно вычисляем пересечение $A[j + 1, n] \cap B[m/2 + 1, m]$;

Корректность в этом случае доказывается легко, а для оценки временной сложности нужно определить наихудший случай. Начнем с простейшей ситуации, когда опорная точка выходит за пределы A ($j = 0$ или $j = n$). Это означает, что одна из двух частей в A пуста и соответствующую часть B можно отбросить из последующих рекурсивных вызовов. То есть единственный бинарный поиск по A за время $O(\log n)$ убрал половину последовательности B. Если такое будет происходить во всех рекурсивных вызовах, общее их количество составит $O(\log m)$, таким образом, в целом работа алгоритма займет время $O(\log m \log n)$. Как видите, *несбалансированное* разбиение A приводит к отличной работе алгоритма поиска пересечения, несмотря на то что на несбалансированных разбиениях рекурсивные алгоритмы в целом работают хуже всего. Теперь предположим, что опорный элемент $b_{m/2}$ попадает внутрь последовательности A. Рассмотрим случай его совпадения с медианным элементом A, который назовем $a_{n/2}$. В этой конкретной ситуации разбиения сбалансированы в обеих последовательностях, поэтому временную сложность можно выразить через рекуррентное соотношение $T(n, m) = O(\log n) + 2T(n/2, m/2)$ с основанием рекурсии $T(n, m) = O(1)$ при любых $n, m \leqslant 1$. Можно доказать, что для любого $m \leqslant n$ это рекуррентное соотношение имеет решение $T(n, m) = O(m(1 + \log n/m))$. Интересно отметить, что в данном случае временная сложность включает в себя сложности предыдущих двух алгоритмов — на базе слияния и бинарного поиска. Фактически при $m = \Theta(n)$ имеем $T(n, m) = O(n)$, как при слиянии, а при $m << n$ имеем $T(n, m) = O(m \log n)$, как при бинарном поиске.

На самом деле временная сложность процедуры взаимного разбиения в модели сравнения оптимальна. Это следует из классического доказательства для двоичного дерева решений: существует по крайней мере $\binom{n}{m}$ решений задачи поиска пересечения множеств (здесь учитывается только случай $B \subseteq A$), поэтому каждый алгоритм на базе сравнения, вычисляющий любое из этих решений, должен выполнить $\Omega\left(\log\binom{n}{m}\right)$ шагов, что по определению биномиального коэффициента составляет $\Omega(m \log n/m)$.

Теорема 6.3. *Алгоритм, основанный на парадигме взаимного разбиения, ищет пересечение упорядоченных множеств за время* $O(m(1 + \log n/m))$. *Временная сложность оптимальна в модели сравнения.*

6.3. Поиск с удвоением

Имеющая оптимальную временную сложность парадигма взаимного разбиения в значительной степени базируется на рекурсивных вызовах и бинарном поиске, имеющих низкую производительность, когда работают с дисковой памятью. В случае длинных последовательностей требуется много рекурсивных вызовов (динамическое распределение памяти) и много шагов бинарного поиска (произвольный доступ к памяти). Чтобы частично решить эти проблемы, рассмотрим другой подход к поиску пересечения отсортированных множеств. Он базируется на интересной парадигме — *поиске с удвоением*, который называется также *галопирующим* или *экспоненциальным* поиском. Нагляднее всего можно объяснить его с помощью индуктивного рассуждения.

Предположим, что мы уже проверили наличие в A первых $j - 1$ элементов B и что элемент b_{j-1} в упорядоченной последовательности A находится сразу после a_i. Другими словами, $a_i \leqslant b_{j-1} < a_{i+1}$. Для проверки следующего элемента B, а именно b_j, достаточно поискать его в $A[i + 1, n]$. Блестящая идея, лежащая в основе этого подхода, состоит в том, что вместо бинарного поиска в этом подмассиве выполняется *поиск с удвоением*, то есть проверка элементов $A[i + 1, n]$ на расстояниях, растущих как степень двойки (отсюда и название). Мы сравниваем b_j с элементами $A[i + 2^k]$ при $k = 0, 1...$, пока не обнаружим, что для некоторого k элемент $b_j < A[i + 2^k]$ или что произошел выход за границу массива A, то есть $i + 2^k > n$. После этого выполняется бинарный поиск b_j в $A[i + 1, \min\{i + 2^k, n\}]$ и в случае успеха возвращается b_j. Таким образом обнаруживается позиция b_j в этом подмассиве, например $a'_i \leqslant b_j < a'_{i+1}$, после чего процесс можно повторить, отбросив $A[1, i']$ из поиска следующих элементов B. В алгоритме 6.2 показан псевдокод такого поиска, а на рис. 6.4 этот процесс представлен графически.

Предполагается, что последовательности A и B пересекаются до элемента $a_i = 12$. Для экспоненциального поиска в элементах последовательности A, находящихся после 12, берется следующий элемент в B, то есть $b_j = 41$ (на него указывает стрелка). Проверяются элементы на расстояниях, которые являются степенью двойки, а именно 1, 2, 4, 8, 16. Поиск длится до обнаружения элемента 60, который больше чем 41, что позволяет определить часть последовательности A, подходящую для бинарного поиска элемента 41. Обратите внимание на то, что подмассив, по которому осуществляется поиск, имеет размер 16, тогда как расстояние между 41 и 12 в A равно 11. Этот пример доказывает, что бинарный поиск выполняется на подмассиве, размер которого меньше удвоенного реального расстояния до искомого элемента.

Алгоритм 6.2. Упорядоченное пересечение множеств на основе поиска с удвоением

1: Пусть $m = |B| \leqslant n = |A|$, в противном случае меняем роли A и B;
2: $i = 0$;
3: **for** $j = 1, 2 \ldots m$ **do**
4: $k = 0$;
5: **while** $(i + 2^k \leqslant n)$ **and** $(B[j] > A[i + 2^k])$ **do**
6: $k = k + 1$;
7: **end while**
8: $i' =$ Бинарный поиск $B[j]$ в $A[i + 2^{k-1} + 1, \min\{i + 2^k, n\}]$;
9: **if** $a_{i'} = b_j$ **then**
10: выводим b_j;
11: **end if**
12: $i = i'$;
13: **end for**

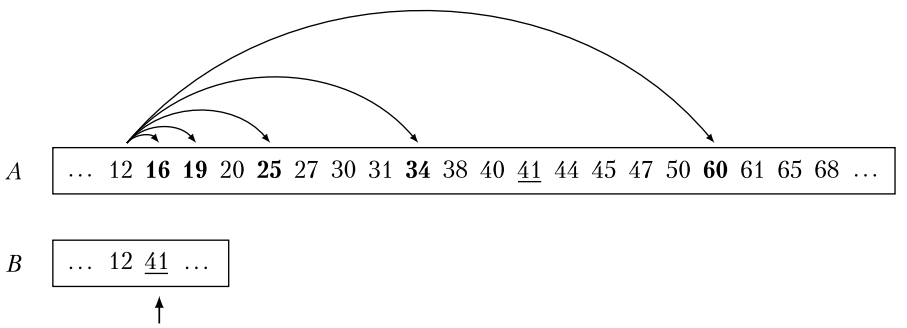


Рис. 6.4. Пример парадигмы поиска с удвоением

Корректность этого алгоритма очевидна, а вот оценка временной сложности потребует больших усилий. Обозначим как $i_j = i'$ позицию элемента b_j в последовательности A. По индукции позиция элемента b_{j-1} будет i_{j-1}, и очевидно, что $i_{j-1} \leqslant i_j$. Для наглядности сделаем $i_0 = 0$ и обозначим $\Delta_j = \min\{2^{k-1}, n\}$ размер подмассива, в котором согласно шагу 8 алгоритма 6.2 выполняется бинарный поиск b_j. Из условия цикла `while` на шаге 5 позиция b_j в последовательности A такова, что $i_j \geqslant i_{j-1} + 2^{k-1}$ (то есть b_j больше, чем ранее проверенный элемент в A) и $i_j < \min\{i_{j-1} + 2^k, n\}$ (то есть b_j меньше, чем $A[i_{j-1} + 2^k]$, или проверяемая позиция вышла за границы A). Объединяя неравенство $2^{k-1} \leqslant i_j - i_{j-1}$ с определением Δ_j, получим $\Delta_j \leqslant 2^{k-1} \leqslant i_j - i_{j-1}$. Теперь у нас есть все, что требуется для оценки общей длины искомых подмассивов последовательности A: $\sum_{j=1}^{m} \Delta_j \leqslant \sum_{j=1}^{m} \left(i_j - i_{j-1}\right) \leqslant n$, ведь здесь мы имеем телескопический

ряд, в котором при суммировании сокращаются последовательные члены, за исключением $i_0 = 0$ и $i_m \leqslant n$. Из-за оператора `while` на шагах 5–7 и бинарного поиска на шаге 8 для каждого j алгоритм 6.2 выполняет $O(1 + \log \Delta_j)$ шагов. Суммируя по $j = 1, 2, ..., m$, получаем общую временную сложность[1]:

$$\sum_{j=1}^{m} O\left(1+\log\Delta_j\right) = O\left(\sum_{j=1}^{m}\left(1+\log\Delta_j\right)\right) = O\left(m + m\log\sum_{j=1}^{m}\frac{\Delta_j}{m}\right) = O\left(m\left(1+\log\frac{n}{m}\right)\right).$$

Теорема 6.4. *Алгоритм, основанный на парадигме поиска с удвоением, находит пересечение отсортированных множеств за время* $O(m(1 + \log n/m))$. *Это оптимальная временная сложность в модели сравнения.*

Обратите внимание, что такую же временную сложность имел алгоритм, основанный на парадигме взаимного разбиения (см. теорему 6.3). Но алгоритм на базе поиска с удвоением — итеративный, то есть, в отличие от схемы с взаимным разбиением, не выполняет никаких рекурсивных вызовов.

6.4. Двухуровневый подход к хранению

Рассмотренный ранее подход помог избежать проблем, связанных с рекурсивным разбиением двух упорядоченных последовательностей A и B, но остались прыжки по массиву A, связанные со схемой удвоения, которые, как вы знаете, неэффективны при выполнении в иерархической памяти. Здесь на помощь придет *двухуровневая организация данных*. Основная идея такой схемы хранения, предназначенной для работы с подготовленной для поисковых систем коллекцией отсортированных списков, заключается в предварительной обработке всех этих списков. Каждый из них *логически* разбивается на блоки размером L (последний блок может оказаться короче), причем первый элемент каждого блока копируется во вспомогательную последовательность. Эта последовательность будет использоваться для ускорения поиска пересечения между любой парой наборов коллекции, фигурирующей в пользовательском запросе, состоящем из двух терминов. Рассмотрим набор A длиной n, причем эта длина кратна L (скажем, $n = hL$). После предварительной обработки набор A превратится в h блоков A_i размером L каждый. Первый элемент каждого блока $A_i[1] = A[(i - 1)L + 1]$ копируется во вспомогательную последовательность A' размером h. Такой предварительной обработке подвергаются все входные наборы. Графически этот процесс представлен на рис. 6.5.

На рис. 6.5 слева жирным шрифтом выделены элементы A, копируемые во вспомогательную последовательность $A' = (1, 4, 8, 15)$ при $L = 2$. Для B также генерируются два блока длиной L, начинающиеся с 5 и 9. Но для простоты разбиение B опущено, поскольку оно не используется в этом примере, где $|A| > |B|$. Справа показано

[1] Мы применяем неравенство Йенсена (https://ru.wikipedia.org/wiki/Неравенство_Йенсена).

разбиение элементов B путем слияния A' и B. При этом в соответствии с элементами $A_i[1]$ (фаза *1*) образуются подмножества B_i. Блок B_1 пуст, поскольку в B отсутствуют элементы между $A_1[1] = 1$ и $A_2[1] = 4$. В результате на фазе *2* A_1 не проверяется, а просто исключается из вычислений без каких-либо временных затрат. Последующие пересечения, обнаруживаемые на фазе *2* между парами подмножеств $A_i \cap B_i$ для $i = 2, 3, 4$, вернут нужное нам пересечение $A \cap B = \{5, 8\}$.

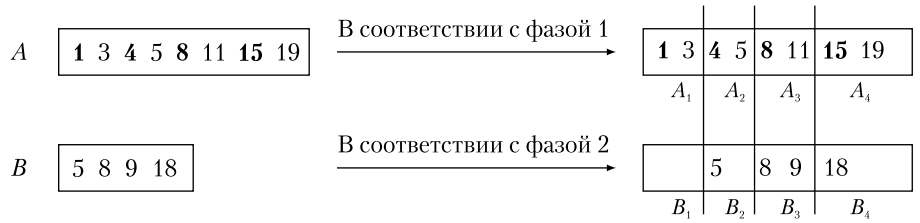


Рис. 6.5. Пример двухуровневого подхода к хранению

После запроса поиск пересечения наборов A и B предварительно обработанной коллекции можно разбить на две фазы. Предположим, что A длиннее B и, таким образом, $n = |A| > |B| = m$. Во время фазы *1* для объединения упорядоченных последовательностей A' и B применяется процедура `MERGE` из алгоритма `MERGESORT` (см. раздел 5.1). В результате получается уникальная последовательность из элементов B, перемежающихся элементами A'. Ее получение занимает время $O(n/L + m)$. Возьмем B_i — непрерывную подпоследовательность элементов B, которая попадает между двумя последовательными элементами A', скажем $A_i[1]$ и $A_{i+1}[1]$. Это означает, что элементы B_i могут встречаться в блоке A_i. Именно это позволяет на фазе *2* воспользоваться алгоритмом пересечения множеств на основе слияния из теоремы 6.1 для вычисления $A_i \cap B_i$ за время $O(|A_i| + |B_i|) = O(L + |B_i|)$. Этот алгоритм выполняется для всех пар A_i и B_i, которые включают в себя непустое подмножество B_i. Таких пар не более m, и поскольку $B = \cup_i B_i$, общее время прохождения фазы *2* составит $O(Lm + m)$. Суммируя временную сложность для двух фаз и учитывая тот факт, что последовательности сканируются оптимальным с точки зрения ввода-вывода способом, получим следующую теорему.

Теорема 6.5. *Алгоритм, использующий двухуровневый подход к хранению, находит пересечение отсортированных множеств за время $O(n/L + mL)$ и за $O(n/LB + mL/B + m)$ операций ввода-вывода, где B — размер страницы диска в двухуровневой модели памяти.*

Отмечу, что двухуровневый подход к хранению подходит для случая со сжатым хранением элементов с целью экономии места и, следовательно, дает повышение общей производительности из-за возможного сокращения объема обрабатываемых данных. В основе этого нового предложения лежат две идеи. Во-первых, возрастающие элементы каждого блока $A_i = (a'_1, a'_2, \ldots, a'_L)$ можно отобразить с помощью схемы сжатия, в которой $a'_0 = 0$, а a'_j для $j = 1, 2, \ldots, L$ представляют через различие

с предыдущим элементом a'_{j-1}. Каждое различие сохраняется (сжимается) с использованием $\left\lceil \log_2\left(1+\max_j\left\{a'_j - a'_{j-1}\right\}\right)\right\rceil$ бит вместо полного представления в 4/8 байтах. Поскольку двухуровневый подход к хранению распространяется по последовательностям слева направо, мы можем эффективно распаковывать разницы и соответствующие им элементы.

Вторая идея вытекает из наблюдения, показывающего, что такая схема сжатия выгодна при маленьких различиях. Во время предварительной обработки алгоритм может искусственно форсировать эту ситуацию, перетасовывая элементы во всех наборах индексированной коллекции посредством случайной перестановки. Это гарантирует наименьший максимальный ожидаемый зазор между соседними перетасованными элементами. Уже перетасованные наборы предварительно обрабатываются и используются для поиска ответов на запросы, как описано ранее. Более подробно варианты этого подхода рассматривались в [2], [3], [5].

Список литературы

1. *Baeza-Yates R.* A fast set intersection algorithm for sorted sequences // Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM), Lecture Notes in Computer Science 3109, Springer, 400–408, 2004.
2. *Barbay J., López-Ortiz A., Lu T., Salinger A.* An experimental investigation of set intersection algorithms for text searching // ACM Journal of Experimental Algorithmics, 14 (3), 7–24, 2009.
3. *Ding B., König A. C.* Fast set intersection in memory // Proceedings of the VLDB Endowment (PVLDB), 4 (4): 255–266, 2011.
4. *Manning C. D., Raghavan P., Schütze H.* Introduction to Information Retrieval. Cambridge University Press, 2008.
5. *Sanders P., Transier F.* Intersection in integer inverted indices // Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 71–83, 2007.
6. *Yan H., Ding S., Suel T.* Inverted index compression and query processing with optimized document ordering // Proceedings of the 18th International Conference on World Wide Web (WWW), Association for Computing Machinery, 401–410, 2009.
7. *Witten I. H., Moffat A., Bell T. C.* Managing Gigabytes. Morgan Kauffman, second edition, 1999.

Глава 7
СОРТИРОВКА СТРОК

В главе 5 рассматривалась сортировка *атомарных элементов*, то есть таких, которые или занимают фиксированное постоянное пространство, или допускают манипулирование только целиком, без разделения на составные части. Пришло время обобщить все изученные алгоритмы и добавить к ним новые, чтобы научиться работать с *элементами переменной длины*, то есть *строками*. С формальной точки зрения мы будем искать эффективное решение следующей задачи.

> **Задача сортировки строк.** Дана последовательность строк $S[1, n]$ общей длиной N, составленных из алфавита размером σ. Требуется расположить эти строки в возрастающем лексикографическом порядке.

Первое, что приходит в голову, — воспользоваться алгоритмами сортировки *на основе сравнения*, такими как `QUICKSORT` или `MERGESORT`, и сравнивать попарно строки с начала, символ за символом, определяя лексикографический порядок по обнаруженным несоответствиям. Обозначим $L = N/n$ среднюю длину строк в последовательности S. В этом случае оптимальный сортировщик на основе сравнения, работающий в оперативной памяти, справится с задачей в среднем за время $O(Ln \log n) = O(N \log n)$, поскольку каждое сравнение строк может включать рассмотрение в среднем $O(L)$ символов.

Помимо неоптимальной временной сложности (см. раздел 7.1), ключевым ограничением такого подхода является то, что S обычно реализуется как *массив указателей* на строки, которые распределены во внутренней памяти компьютера или при очень больших N хранятся на диске. Эти две ситуации иллюстрирует рис. 7.1. Какое бы распределение ни выбрал алгоритм, сортировщик будет опосредованно упорядочивать строки S, но перемещая не символы, а указатели. Этот момент программисты часто игнорируют, из-за чего на больших наборах строк сортировщик будет работать медленно. Причина лежит на поверхности: каждое сравнение строк начинается с разрешения двух указателей, скажем $S[i]$ и $S[j]$, и только потом выполняется посимвольное сравнение. В результате каждый раз мы получаем два промаха кэша, то есть две дополнительные операции ввода-вывода, и в целом алгоритм выполняет $\Theta(n \log n)$ операций ввода-вывода. Как я отмечал в главе 1, в таких случаях помогает виртуальная память операционной системы, ведь буферизация последних

строк, которые сравнили, позволяет сократить количество вводов-выводов. Но за это буферное пространство в нашем случае конкурируют два массива — массив указателей и массив строк. Кроме того, тратится время на *повторное сканирование* префиксов строк, уже принимавших участие в процедуре сравнения.

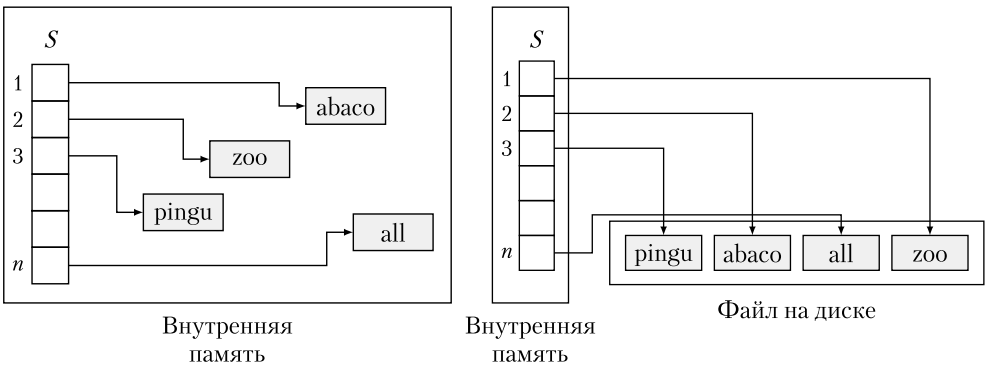


Рис. 7.1. Примеры размещения: *слева* — строк во внутренней памяти; *справа* — строк, последовательно записанных в файл на диске

Так что эту главу я посвятил алгоритмам, оптимальным по количеству выполняемых сравнений символов и предлагающим шаблоны доступа к памяти, учитывающие ввод-вывод, что делает их эффективными в случае иерархической памяти.

7.1. Нижняя граница

Пусть d_s — длина самого короткого префикса строки $s \in S$, который отличает ее от других строк набора. Значение d_s называется *отличительным префиксом* строки s. Сумма этих значений по всем строкам будет называться отличительным префиксом набора S. Ее мы обозначим $d = \sum_{s \in S} d_s$. На рис. 7.1, где S состоит из четырех строк, отличительным префиксом строки `all` будет `al`, поскольку эта подстрока не является префиксом никакой другой строки в S, тогда как `a` — является.

Очевидно, что любой сортировщик строк должен сравнивать начальные d_s символов строки s, ведь только таким способом он сможет отличить ее от других строк в S и найти ее лексикографическую позицию в упорядоченном наборе. Соответственно, в формуле для нижней границы сортировки строк должен присутствовать член $\Omega(d)$. Но он не учитывает затраты на сортировку n строк, составляющие $\Omega(n \log n)$ операций сравнения строк, а значит, по меньшей мере $\Omega(n \log n)$ операций сравнения символов, потому что сортировка невозможна без сравнения хотя бы одного символа на строку.

Лемма 7.1. *Любой алгоритм, решающий задачу сортировки строк, должен выполнять* $\Omega(d + n \log n)$ *сравнений символов.*

Здесь нужно сделать несколько комментариев. Предположим, что n — степень двойки. При этом n строк в S являются бинарными, имеют одинаковые начальные ℓ битов и различные остальные $\log n$ бит. В этом случае $d_s = \ell + \log n$, а $d = n(\ell + \log n) = N$. Нижняя граница будет равна $\Omega(N + n \log n) = \Omega(N)$, поскольку $N = n(\ell + \log n) \geqslant \geqslant n \log n$. Но работа сортировщиков строк на базе алгоритмов `MERGESORT` или `QUICKSORT` занимает время $\Theta(N \log n)$. Словом, для любого ℓ эти алгоритмы могут отличаться от оптимальных на коэффициент $\Theta(\log n)$, который увеличивается по мере роста набора S.

Возникает вопрос: нельзя ли реализовать сортировку строк, не рассматривая все их содержимое? Это возможно при $d < N$. Именно поэтому и был введен параметр d, позволяющий проводить более тонкий анализ алгоритмов, которые будут обсуждаться в дальнейшем.

7.2. RADIXSORT

Чтобы получить более конкурентоспособный алгоритм сортировки строк, рассмотрим строки как последовательности символов целочисленного алфавита {0, 1, 2... $\sigma - 1$}, то есть цифр[1]. Это легко сделать, заранее отсортировав символы из набора S, а затем присвоив каждому из них целое число (его разряд) в указанном диапазоне. Обычно такая процедура называется процессом *именования* и занимает время $O(N\log \sigma)$, ведь для нее можно использовать двоичное дерево поиска, построенное не более чем по σ различным символам, встречающимся в S.

В дальнейшем я буду исходить из предположения, что строки в наборе S составлены из целочисленного алфавита размером σ. Если это не так, к временной сложности предлагаемого алгоритма будет добавляться $O(N \log \sigma)$. Более того, легко увидеть, что каждый символ можно закодировать в $\lceil \log_2 \sigma \rceil$ бит, таким образом, измеряемый в битах размер входных данных составит $\Theta(N \log \sigma)$.

Для алгоритма `RADIXSORT` можно разработать два основных варианта, различающихся порядком обработки: MSD-сортировка обрабатывает строки, двигаясь вправо начиная с наиболее значимой цифры, а LSD-сортировка обрабатывает строки, двигаясь влево начиная с наименее значимой цифры.

7.2.1. MSD-сортировка

Этот алгоритм использует подход «разделяй и властвуй», ведь он посимвольно обрабатывает строки, рекурсивно распределяя их по σ корзинам. При этом каждый символ обрабатывается за постоянное время. На рис. 7.2 показан набор S из семи строк, составленных из букв алфавита размером $\sigma = 10$. Строки распределяются

[1] В системе счисления с основанием σ. — *Примеч. науч. ред.*

по десяти корзинам в соответствии с их первой цифрой (по старшему разряду числа). Поскольку корзины 1 и 7 содержат по одной строке, сортировка им не требуется. А вот содержимое корзин 0 и 9 нуждается в рекурсивной сортировке в соответствии со второй цифрой каждой строки. В конце мы получаем упорядоченный набор S путем конкатенации всех групп отдельных строк слева направо.

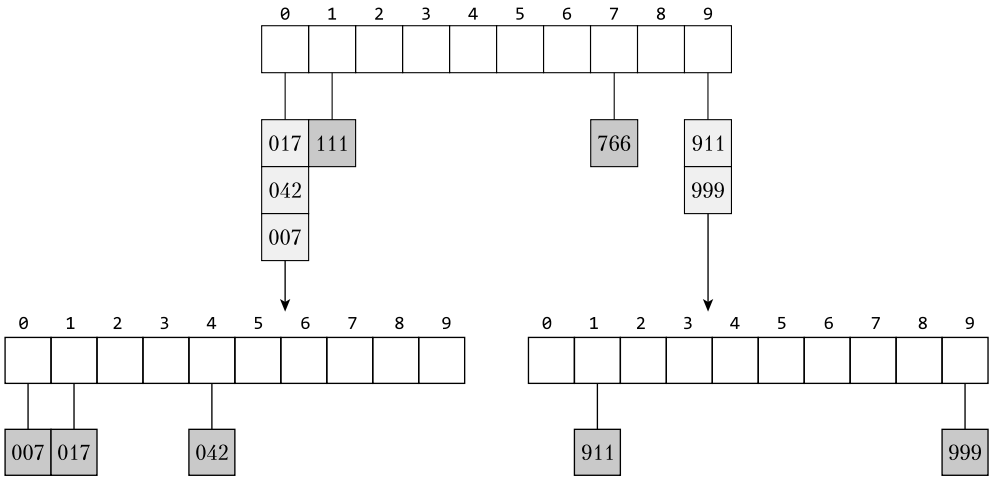


Рис. 7.2. *Вверху* — семь строк отсортированы по цифре старшего разряда. *Внизу* — результат рекурсивной сортировки блоков 0 и 9 по второй слева цифре

Нетрудно заметить, что подходы на базе распределения генерируют деревья поиска. Классический алгоритм `QUICKSORT` генерирует бинарное дерево поиска, в то время как MSD-сортировка алгоритма `RADIXSORT` генерирует σ-позиционное дерево из-за выполняемого на каждом шаге σ-арного разбиения строк S. Такое дерево в литературе носит название *префиксного дерева*, или *дерева цифрового поиска*, и применяется в основном для поиска строк (подробно поговорим о нем в главе 9).

Пример префиксного дерева для набора строк с рис. 7.2 демонстрируется на рис. 7.3. Каждый узел реализован как массив размером σ, по одной записи на возможный символ алфавита. Строки хранятся в листьях, соответственно, имеется n листьев. Внутренних узлов длиной меньше N по одному на символ, встречающийся в строках набора S. Нисходящий путь к узлу u от корня дерева составляет строку $s[u]$, которая получается конкатенацией символов, встречающихся при обходе пути. Например, путь к листу `017` проходит через три узла, по одному на возможный символ этой строки. При фиксированном узле u все выходящие из него строки будут иметь один и тот же префикс $s[u]$. Например, s[корень] — это пустая строка, которая, очевидно, является общей для всех строк набора S. Самый левый дочерний элемент корня образует строку `0`, поскольку для его достижения нужно обойти ребро, выходящее из 0-й записи массива.

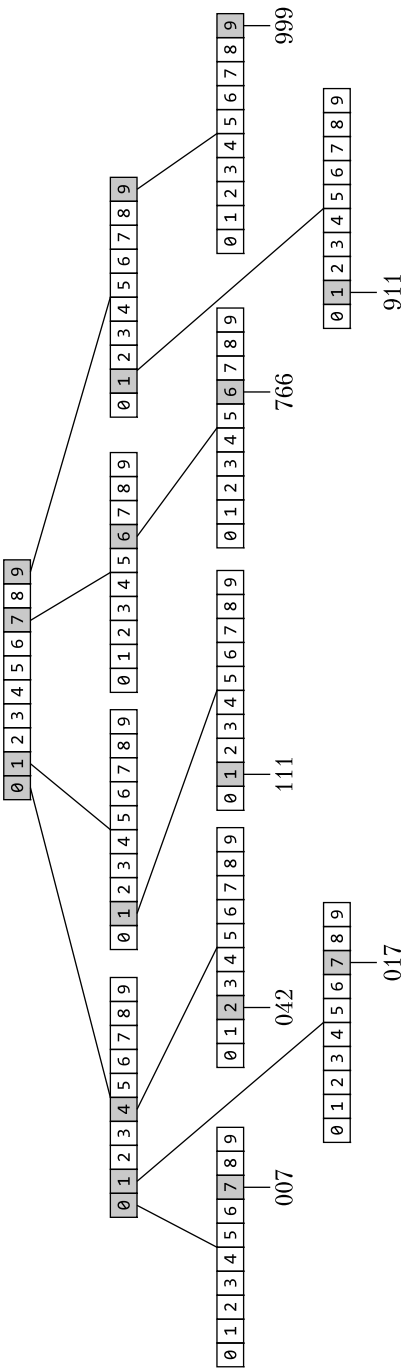


Рис. 7.3. Префиксное дерево MSD-сортировки алгоритма RADIXSORT для строк с рис. 7.2

Обратите внимание на то, что префиксное дерево может содержать *унарные узлы*, то есть узлы с единственным дочерним элементом. Например, это узлы на путях, ведущих к строкам `111` и `766`.

Все остальные внутренние узлы имеют по крайней мере двух потомков. Они называются *узлами ветвления*. На рис. 7.3 девять унарных узлов и три узла ветвления с $n = 7$ и $N = 21$. В общем случае префиксное дерево может содержать не более n узлов ветвления и не более N унарных узлов. Унарных узлов с нисходящим узлом ветвления будет не больше d. Они соответствуют символам, встречающимся в отличительных префиксах строк набора S, а самые нижние нисходящие узлы ветвления соответствуют символам, которыми завершаются отличительные префиксы. Унарные пути, начинающиеся с самых нижних узлов ветвления в префиксном дереве и ведущие к его листьям, соответствуют суффиксам строк, которые следуют за этими отличительными префиксами. С точки зрения алгоритма `RADIXSORT` унарные узлы выглядят как блоки из элементов, имеющих одинаковые сравниваемые символы в распределении MSD-сортировки, тогда как узлы ветвления соответствуют блокам, которые образованы элементами с различными сравниваемыми символами.

Если метки ребер располагаются в алфавитном порядке, как на рис. 7.3, чтение листьев в соответствии с предварительным порядком обхода префиксного дерева дает отсортированный набор S. Так работает простой сортировщик строк на основе префиксного дерева.

Идея состоит в том, чтобы по очереди вставлять строки в изначально пустое префиксное дерево. Вставка строки $s \in S$ предполагает отслеживание нисходящего пути до тех пор, пока символы s не будут сопоставлены с существующими метками ребер, то есть непустыми записями массивов размером σ. Если следующий символ s не сопоставляется ни с одним из ребер, выходящих из достигнутого узла u[1], значит, мы нашли несовпадение. Поэтому в дерево добавляется *специальный* узел ветвления с несовпадающим символом, который указывает на s. Особенность этой операции в отбрасывании еще не сопоставленного суффикса s. При этом указатель на строку неявно отслеживает его для последующих вставок. Если при вставке очередной строки s' встречается специальный узел u, нужно обратиться к связанной с ним строке s и создать для остальных символов общего префикса, выходящих из u строк s и s', унарный путь. Последний на маршруте узел разветвится на s и s', возможно снова игнорируя пути, соответствующие еще не сопоставленным суффиксам этих двух строк, и генерируя для каждой из них очередной специальный узел.

Каждое создание узла префиксного дерева сопровождается выделением памяти под массив размером σ, что занимает время и пространство $O(\sigma)$. Таким образом, можно доказать следующую теорему.

---

[1] Фактически это означает, что ячейка в массиве u размером Σ, соответствующая следующему символу строки s, имеет значение null.

Теорема 7.1. *Набор строк из букв целочисленного алфавита размером* σ *с отличающимся префиксом* d *MSD-сортировка алгоритма* `RADIXSORT` *упорядочивает при помощи префиксного дерева и массивов размером* σ*, используя для этого время и пространство* $O(d\sigma)$.

Доказательство. Каждая строка s описывает путь длиной d_s, перед этим создается специальный указывающий на эту строку узел. Под каждый такой узел выделяется пространство σ, причем процесс выделения занимает время σ. Временные затраты на обход одного узла префиксного дерева оцениваются как $O(1)$. Следовательно, на каждый пройденный/созданный узел будет затрачиваться время $O(\sigma)$. Для обхода префиксного дерева с чтением его листьев слева направо теорема доказана, если учесть, что листья лексикографически отсортированы, поскольку именование символов является лексикографической процедурой и, таким образом, отражает их порядок. ■

Как видите, алгоритму для работы требуется значительное пространство, которое неплохо было бы уменьшить. Для этого, к примеру, в каждом узле u массив размером σ можно заменить хеш-таблицей (со списком коллизий). Размер таблицы e_u пропорционален количеству выходящих из u ребер и индексируется цифрой, связанной с меткой ребра[1]. Это гарантирует ожидаемое время поиска и вставки одного ребра в каждый узел $O(1)$. Напомню, что количество ребер может быть ограничено количеством внутренних узлов, то есть $O(d)$, и количеством специальных узлов, то есть n листьев, так что $\sum_u e_u = O(d + n) = O(d)$. Таким образом мы можем вывести время построения. Вставка всех строк в префиксное дерево займет время $O(d)$ при постоянном времени доступа к каждому узлу. На сортировку ребер этого дерева по всем узлам потребуется время $O(\sum_u e_u \log e_u) = O(\sum_u e_u \log \sigma) = O(d \log \sigma)$. Кроме того, время $O(d)$ понадобится для сканирования листьев префиксного дерева слева направо в процессе предварительного обхода для получения словарных строк в лексикографическом порядке.

Теорема 7.2. *При использовании префиксного дерева с хешированием набор строк, построенных из целочисленного алфавита размером* σ*, с отличающимся префиксом* d *MSD-сортировка алгоритма* `RADIXSORT` *сортирует за ожидаемое время* $O(d \log \sigma)$*, занимая пространство* $O(d)$.

При малых σ невозможно сделать вывод, *лучше этот результат, чем нижняя граница* из леммы 7.1, или нет, поскольку она определена для алгоритмов на базе сравнения и, следовательно, не работает в случае хеширования или сортировки целочисленных значений.

---

[1] Более сложные решения на основе хеширования будут рассматриваться в главе 8. Здесь речь пойдет про хеш-таблицы с цепочками, поскольку это классическая тема каждого базового курса по алгоритмам, которой вполне достаточно для целей этой главы.

Пространство, выделенное для префиксного дерева, можно сократить до $O(n)$, а время построения — до $O(d + n \log \sigma)$, если воспользоваться *сжатыми* префиксными деревьями, в которых унарные пути сливаются в одиночные ребра. Длина этих ребер равна количеству символов, формирующих компактный унарный путь. Эта структура данных будет обсуждаться в главе 9.

7.2.2. LSD-сортировка

Рассматриваемый далее сортировщик был разработан Германом Холлеритом более века назад и послужил основой машины для обработки результатов переписи населения за 1890 год. Что интересно, именно Холлерит основал компанию, которая впоследствии стала называться IBM[1]. Предложенный им алгоритм был контринтуитивным, поскольку сортировал строки цифра за цифрой, начиная с наименее значимой и используя *устойчивый* сортировщик как черный ящик. Напомню, что сортировщик считается устойчивым только при условии, что одинаковые ключи сохраняют в конце исходный порядок. Возьмем в качестве такого сортировщика `COUNTINGSORT` (см., например, [3]) и предположим, что все строки имеют одинаковую длину L. Если это предположение неверно, строки *логически* дополняются спереди специальной цифрой, которая меньше любой другой цифры алфавита, чтобы обеспечить LSD-сортировщику алгоритма `RADIXSORT` упорядоченную лексикографическую последовательность.

LSD-сортировка алгоритма `RADIXSORT` состоит из L фаз, скажем $i = 1, 2... L$. На каждой фазе все строки сортируются по i-й наименее значимой цифре. Наглядный пример такой сортировки демонстрирует рис. 7.4. Жирным шрифтом выделены цифры, подлежащие сортировке на текущей фазе, а уже прошедшие сортировку — подчеркнуты. На каждой фазе порядок строк меняется в соответствии с порядком входной последовательности, полученной на предыдущих итерациях. Это происходит для упорядочения строк с одинаковыми цифрами в текущей сравниваемой позиции i. В качестве примера рассмотрим вторую фазу для строк `111` и `017`. Из-за одинаковой второй цифры после первого шага сортировки `111` оказалась перед `017`. Но на последней фазе, когда сравнивались значения в третьей позиции, их порядок поменялся.

Временную сложность можно легко оценить как длину строк L, умноженную на затраты сортировщика `COUNTINGSORT` на обработку n целых цифр из диапазона $\{0, 1... \sigma - 1\}$. Здесь мы получаем $O(L(n + \sigma))$. Приятным свойством этого сортировщика является то, что он функционирует непосредственно в памяти, если в ней работает сортировочный черный ящик, а именно $\sigma = O(1)$.

Лемма 7.2. *LSD-сортировка алгоритма* `RADIXSORT` *сортирует строки за время* $O(L(n + \sigma)) = O(N + L\sigma)$, *занимая пространство* $O(N)$. *Сортировщик работает непосредственно в памяти тогда и только тогда, когда используется работающий в памяти сортировщик цифр.*

---

[1] См.: https://ru.wikipedia.org/wiki/Холлерит_Герман.

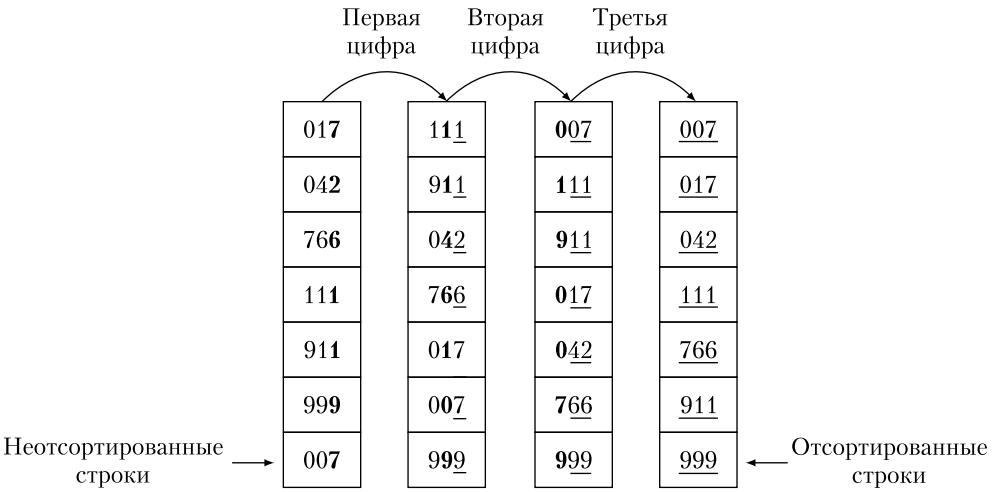


Рис. 7.4. Пример LSD-сортировки алгоритма RADIXSORT. Невыделенные цифры еще не обработаны, подчеркнутые — уже обработаны, выделенные жирным — обрабатываются стабильным сортировщиком. Для сортировки строк, состоящих из трех цифр, достаточно трех фаз

Доказательство. Временная и пространственная эффективность вытекает из предыдущих наблюдений. Для доказательства корректности нужно подтвердить устойчивость сортировщика `COUNTINGSORT`. Пусть α и β — две строки из набора S, причем в соответствии с их лексикографическим порядком $\alpha < \beta$. Поскольку все строки в S имеют одинаковую длину, допустимо разложение $\alpha = \gamma a \alpha_1$ и $\beta = \gamma b \beta_1$. Здесь γ — самый длинный общий префикс строк α и β (он может быть пустым), $a < b$ — первые несовпадающие символы, а α_1 и β_1 — два оставшихся суффикса, которые тоже могут быть пустыми.

В истории сравнений цифр α и β можно выделить три этапа в зависимости от положения сравниваемой цифры в нашем трехкомпонентном разложении. Алгоритм начинает с наименее значимой цифры, то есть первыми сравниваются цифры в суффиксах α_1 и β_1. Общий порядок после первых фаз $|\alpha_1| = |\beta_1|$ не имеет значения, потому что на следующем шаге строки α и β сортируются в соответствии с символами a и b. Поскольку $a < b$, строка α оказывается перед β. Все остальные шаги сортировки $|\gamma|$ будут сравнивать цифры, попадающие в γ. Поскольку это общий префикс, символы в обеих строках будут одинаковыми и из-за стабильности процедуры `COUNTINGSORT` порядок этих строк не изменится. В конце мы получим корректный результат $\alpha < \beta$. Это справедливо для любой пары строк в наборе S, что означает лексикографическую упорядоченность конечной последовательности, созданной LSD-сортировкой алгоритма `RADIXSORT`. ■

Хотелось бы сделать несколько замечаний по поводу предыдущей временной границы. Так как LSD-сортировка алгоритма `RADIXSORT` обрабатывает все цифры всех строк, при $d << N$ она кажется намного менее привлекательной, чем

MSD-сортировка. Но дело в том, что при LSD-сортировке можно сортировать не одиночные цифры, а сразу *группы*. Очевидно, что чем длиннее такая группа, тем выше временная сложность фазы, но тем меньше количество фаз. Фактически нам предоставляется компромисс, которым можно воспользоваться, если хорошо понимать связь между этими двумя параметрами. Без потери общности упростим ситуацию, предположив, что набор S содержит двоичные строки одинаковой длиной b бит, соответственно, $N \log \sigma = bn$. На практике это не будет ограничением, поскольку любая строка кодируется в памяти как последовательность битов, занимая $\log \sigma$ бит на одну цифру.

Лемма 7.3. *LSD-сортировке алгоритма* `RADIXSORT` *для сортировки n строк по b бит каждая требуются время* $\Theta(b/r(n + 2^r))$ *и пространство* $O(nb) = O(N \log \sigma)$. *Здесь* $r \leqslant b$ *— положительное, целое, заранее заданное число.*

Доказательство. Разделим каждую строку на $g = b/r$ групп по r бит. На каждой фазе происходит упорядочение строк в соответствии с группой из r бит. Следовательно, алгоритму `COUNTINGSORT` приходится упорядочивать n целых чисел от 0 до $2^r - 1$ включительно, для чего требуется время $O(n + 2^r)$. Так как это осуществляется за g фаз, общее время составит $O(g(n + 2^r)) = O((b/r)(n + 2^r))$. ■

При известных n и b нужно выбрать значение r, обеспечивающее минимальную временную сложность. Можно воспользоваться методами математического анализа, например, посмотреть на первую производную, хотя есть и более простой путь. Так как для сортировки каждой группы из r цифр алгоритму `COUNTINGSORT` требуется время $O(n + 2^r)$, нет смысла работать с группами короче чем $\log n$, ведь время $O(n)$ будет потрачено в любом случае. Поэтому r нужно выбирать из интервала $[\log n, b]$. Как только r превысит $\log n$, из-за соотношения $2^r/r$ возрастет временная сложность в лемме 7.3. В результате наилучшим выбором станет значение $r = \Theta(\log n)$, для которого временная сложность составляет $O((bn)/\log n)$.

Теорема 7.3. *Поразрядная LSD-сортировка упорядочивает n строк по b бит каждая за время* $O((bn)/\log n)$, *занимая пространство* $O(bn)$ *и применяя алгоритм* `COUNTINGSORT` *к группам из* $\Theta(\log n)$ *бит. Этот алгоритм не работает непосредственно в памяти, потому что ему требуется дополнительное пространство* $\Theta(n)$.

Теперь легко увидеть, что общая длина строк в наборе S в битах составит bn. По-другому ее можно выразить как $N \log \sigma$, поскольку для представления каждого символа требуется $\log \sigma$ бит.

Следствие 7.1. *Составленные из алфавита размером* Σ *n строк поразрядная LSD-сортировка упорядочивает за время* $O((N \log \sigma)/\log n)$, *требуя пространство* $O(N \log \sigma)$ *бит.*

Если $d = \Theta(N)$, а σ — константа (соответственно, $N = \Omega(n \log n)$ из-за разницы строк), то нижняя граница, основанная на сравнении (см. лемму 7.1), будет $\Omega(N)$. Таким образом, поразрядная LSD-сортировка превосходит эту нижнюю границу, что неудивительно, ведь сортировщик работает с целочисленным алфавитом и использует алгоритм `COUNTINGSORT`.

Сравнивая MSD-сортировку алгоритма `RADIXSORT` (см. теоремы 7.1 и 7.2) на основе префиксного дерева с LSD-сортировкой, легко увидеть, что для $d = O(N/\log n)$, то есть для большинства практических случаев, первый вариант всегда лучше второго. Фактически при LSD-сортировке весь набор строк сканируется в любом случае, тогда как конструкция с префиксным деревом может пропускать некоторые суффиксы строк при $d << N$. Зато при LSD-сортировке удается избежать динамического выделения памяти, связанного с конструкцией префиксного дерева, и дополнительного пространства для хранения этой структуры. На практике эти дополнительные затраты весьма значительны и могут неблагоприятно повлиять на производительность MSD-сортировки и даже помешать ее работе на больших наборах строк из-за ограниченного размера внутренней памяти.

7.3. Многомерная быстрая сортировка

Сейчас я покажу вариант уже известного вам алгоритма `QUICKSORT`, расширенный для управления элементами переменной длины. В основе этого сортировщика строк лежит процедура сравнения, а его нижняя граница $\Omega(d + n \log n)$ в соответствии с леммой 7.1. Освежить в памяти принципы его работы можно, вернувшись к главе 5. Сейчас же достаточно помнить, что алгоритм `QUICKSORT` зависит от двух основных факторов: процедуры выбора опорного элемента и алгоритма разбиения входного массива по выбранному опорному элементу. В этом разделе я ограничусь *случайным* выбором опорного элемента и разбиением входного массива на *три части*. Перейти к другим вариантам, которые обсуждались в главе 5, вы легко сможете, выбрав для строк соответствующие настройки.

Ключевой момент заключается в том, что теперь будут рассматриваться не атомарные элементы, а строки, которые следует делить на составляющие их символы. Опорной точкой также станет символ, и разбиение входных строк будет осуществляться по символу, занимающему определенную позицию. Алгоритм 7.1 подробно демонстрирует псевдокод многомерного варианта `QUICKSORT`. Предполагается, что набор входных строк R *не содержит префиксов*, соответственно, ни одна строка не будет служить префиксом ни для какой другой строки. Это условие легко выполняется, когда все строки в R различны и логически дополнены пустым символом, который меньше любого другого символа алфавита. Это гарантирует, что любая пара строк в R допускает ограниченный *самый длинный общий префикс* (longest common prefix, LCP) и у обеих строк есть следующий за LCP несовпадающий символ.

Алгоритм 7.1. MULTIKEYQS(R, i)

```
1:  if |R| ⩽ 1 then
2:      return R;
3:  else
4:      выбираем опорную строку p ∈ R;
5:      R_< = {s ∈ R | s[i] < p[i]};
6:      R_= = {s ∈ R | s[i] = p[i]};
7:      R_> = {s ∈ R | s[i] > p[i]};
8:      A = MULTIKEYQS(R_<, i);
9:      B = MULTIKEYQS(R_=, i + 1);
10:     C = MULTIKEYQS(R_>, i);
11:     return конкатенированную последовательность A, B, C;
12: end if
```

На входе алгоритм 7.1 получает последовательность из R строк и целочисленный параметр $i \geqslant 0$, который указывает смещение символа, управляющего трехсторонним разбиением. Опорным символом будет $p[i]$, где p — случайно выбранная строка. Реальная реализация такого трехстороннего разбиения может проходить в соответствии с процедурой `PARTITION` из главы 5. Алгоритм `MULTIKEYQS`(R, i) предполагает, что для входных параметров выполняется следующий инвариант: *все строки в R лексикографически упорядочиваются до их префикса длиной* ($i - 1$). Таким образом, сортировка входного набора строк $S[1, n]$ осуществляется вызовом `MULTIKEYQS`(S, 1), что гарантирует инвариант для исходной последовательности S. На шагах 5–7 последовательность строк из R разбивается на три подмножества, содержимое которых поясняет нотация. Все подмножества рекурсивно сортируются, а итоговые упорядоченные последовательности в конечном счете объединяются, формируя упорядоченный набор R. Сложность здесь заключается в выборе параметров, передаваемых трем рекурсивным вызовам.

- Процедуре сортировки строк в подмножествах $R_<$ и $R_>$ все еще приходится повторно рассматривать i-й символ, потому что мы только что проверили, что он меньше/больше, чем $p[i]$, а для упорядочения строк этого недостаточно. Таким образом, рекурсия не продвигает i и зависит от того, действителен ли инвариант.
- Сортировка строк в подмножестве $R_=$ может сдвинуть i вперед, поскольку, согласно инварианту, эти строки сортируются по префиксам длиной ($i - 1$) и, согласно построению $R_=$, имеют общий i-й символ, который равен $p[i]$. Соответственно, $p \in R_=$.

Пример, на котором можно убедиться в корректности приведенных рассуждений, демонстрируется на рис. 7.5. Осталось вычислить ожидаемую временную сложность

алгоритма `MULTIKEYQS`. Давайте возьмем одну строку $s \in R$ и подсчитаем, в каком количестве сравнений во время последовательности рекурсивных вызовов участвует один из ее символов. В общем случае при рекурсивном вызове `MULTIKEYQS`(R, i) возможны два варианта: $s \in R_< \cup R_>$ или $s \in R_=$. В первом случае символ $s[i]$ сравнивается с соответствующим символом опорной строки $p[i]$, а затем s включается в меньший набор $R_< \cup R_> \subset R$, при этом смещение i не меняется. Во втором случае $s[i]$ и $p[i]$ одинаковы, поэтому s включается в $R_=$, а смещение i увеличивается. При удачном выборе опорного элемента (см. главу 5) трехсторонние разбиения сбалансированы и, таким образом, $|R_< \cup R_>| \leqslant \alpha n$ для подходящей константы $\alpha < 1$. В результате в обоих случаях требуется время $O(1)$, но в первом случае набор строк уменьшается на постоянный множитель, а во втором увеличивается i. Поскольку исходный набор $R = S$ имеет размер n, а i ограничено сверху длиной строки $|s|$, количество сравнений с участием s составляет $O(|s| + \log n)$. Суммирование по всем строкам в S дает ограничение по времени $O(N + n \log n)$. Во втором случае i может быть ограничено сверху d_s символами, которые принадлежат отличительному префиксу s, потому что из-за этих символов s оказывается в одноэлементном наборе.

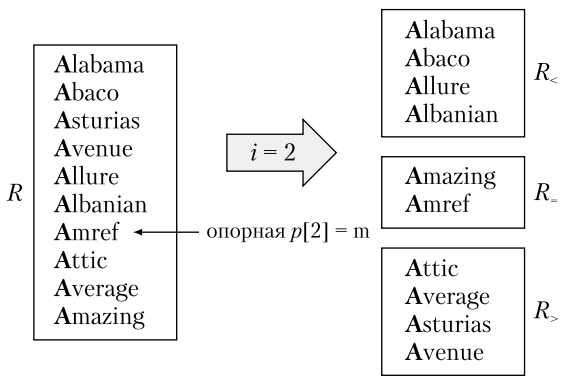


Рис. 7.5. Пример выполнения MULTIKEYQS(R, 2). Жирным шрифтом выделен префикс длиной 1, общий для всех строк в R согласно гарантированному инварианту. Опорная строка, выбранная случайным образом, — это p = Amref, а символ, взятый для разбиения набора строк R на три части, — $p[2]$ = m

Теорема 7.4. *При сортировке строк алгоритмом* `MULTIKEYQS` *ожидаемое количество сравнений символов составляет* $O(d + n \log n)$. *Для модели на базе сравнения это оптимальный вариант. Чтобы перейти к границе для наихудшего случая, для выбора в качестве опорной точки медианы набора R можно взять алгоритм линейного времени.*

Если сравнивать многомерную быструю сортировку с сортировщиками на основе префиксных деревьев, с практической точки зрения она выглядит более привлекательно. Она намного проще, не требует дополнительных структур данных (например, хеш-таблицы или массивов размером Σ), а скрытые в нотации «О большое» константы очень малы.

В заключение отмечу интересную параллель между многомерным алгоритмом `QUICKSORT` и *троичными* деревьями поиска [4]. Каждый узел такого дерева содержит *разделительный символ* и три типа указателей: левый, средний и правый. В некотором смысле троичное дерево поиска получается из префиксного путем объединения потомков, у которых первый спереди символ меньше/больше разделительного символа. Элегантность и практическую эффективность этой структуры данных обеспечивает именно трехстороннее ветвление, возникающее как упрощение Σ-арного ветвления классических префиксных деревьев, а также уменьшающее промахи кэша при каждом ветвлении узла.

Если узел разделяется на i-м символе (назовем его c), то у строк, нисходящих от левого (или правого) дочернего элемента, i-й символ будет меньше (или больше), чем c. Средний дочерний элемент указывает на строки, в которых i-й символ равен c. Более того, как и в случае многомерной быстрой сортировки, итератор i не продвигается при спуске по левому/правому дочерним элементам, он увеличивается только при спуске по среднему дочернему элементу.

Балансировка троичных деревьев осуществляется вставкой строк в случайном порядке либо с помощью различных известных схем. Поиск продолжается обходом по ребрам в соответствии с разделительным символом встреченных узлов. Пример троичного дерева поиска показан на рис. 7.6. Поиск шаблона P = “ir” начинается с корня, который помечен символом i, и инициализирует итератор $i = 1$. Поскольку $P[1]$ = `i`, поиск продолжается до среднего дочернего элемента, увеличивает i до 2 и, таким образом, достигает узла с разделительным символом s. Здесь $P[2]$ = `r` < `s`, поэтому мы переходим на левый дочерний элемент, а i остается неизменным. Поиск останавливается, потому что этот лист указывает на строку `in`. Таким образом, алгоритм приходит к выводу, что P не входит в набор строк, индексированный троичным деревом поиска, и фактически также может определить лексикографическую позицию P справа от `in`.

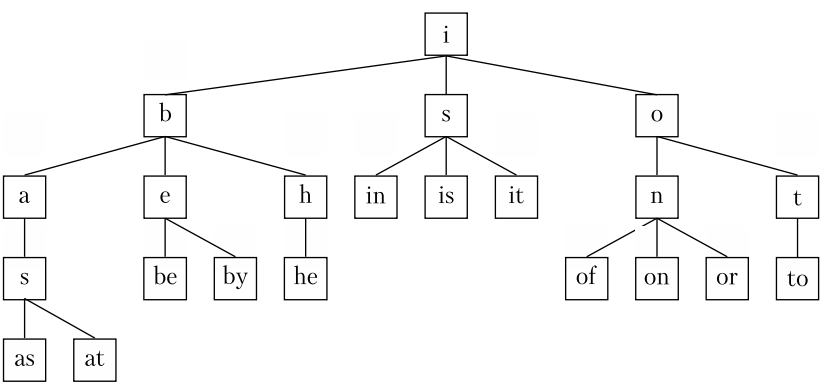


Рис. 7.6. Троичное дерево поиска для 12 двухбуквенных строк. Левые и правые указатели обозначены сплошными линиями, а указатели на равный разделительный символ потомка — пунктирными линиями. Разделительный символ указан во внутренних узлах

Теорема 7.5. *Поиск шаблона* $P[1, p]$ *в сбалансированном троичном дереве поиска, представляющем n строк, требует* $O(p + \log n)$ *сравнений символов. Это оптимально для модели на базе сравнения.*

7.4. Некоторые наблюдения для двухуровневой модели памяти∞

Сортировка строк на диске не так проста, как во внутренней памяти, и в литературе упоминается множество сложных алгоритмов для достижения эффективности ввода-вывода (см., например, [1], [2]). Проблема связана с переменной длиной строк и с тем, что их сравнение методом перебора требует множества лишних операций ввода-вывода. Обозначим n_s количество строк, которые короче страницы диска B и имеют общую длину N_s, а n_l — количество строк длиннее B с общей длиной N_l. Очевидно, что $n = n_s + n_l$ и $N = N_s + N_l$.

Известные алгоритмы можно классифицировать по способу управления строками в процессе их сортировки. Есть три основные модели вычислений [1].

- Модель A. Строки считаются *неделимыми* (то есть перемещаются целиком и не допускают разбиения на символы), за исключением того, что длинные строки можно делить на блоки размером B.
- Модель B. Допускает разделение строк на отдельные символы, но это может происходить *только во внутренней памяти*.
- Модель C. Допускает разделение строк *как во внутренней, так и во внешней памяти*.

Модель A заставляет использовать сортировщики на основе процедуры `MERGE`, границы ввода-вывода которых можно считать оптимальными.

Теорема 7.6. *В модели A сортировка строк требует* $\Theta\left(\frac{N_s}{B}\log_{M/B}\frac{N_s}{B} + n_l \log_{M/B} n_l + \frac{N_s + N_l}{B}\right)$ *операций ввода-вывода.*

В этой оценке первый член — это затраты на сортировку коротких строк, второй — затраты на сортировку длинных, а последний учитывает затраты на чтение всего списка ввода. Результат показывает, что при длине N_s сортировка коротких строк и сортировка их отдельных символов имеют одинаковую сложность, а сортировка длинных строк по сложности аналогична сортировке их первых B символов. Нижняя граница для небольших строк в теореме 7.6 доказывается расширением техники, которой я пользовался в главе 5, и рассмотрением особого случая, когда все n_s небольших строк имеют одинаковую длину N_s/n_s. Для длинных строк нижнюю границу можно доказать рассмотрением n_l небольших строк, полученных в процессе просмотра их первых B символов. Верхние границы в теореме 7.6

получены с помощью специального многомерного алгоритма MERGESORT, который для управления слиянием строк использует ленивое *префиксное дерево*, хранящееся во внутренней памяти.

Модель B представляет собой более сложную ситуацию, в которой длинные и короткие строки приходится обрабатывать по отдельности.

Теорема 7.7. *Сортировка коротких строк в модели B требует* $O\left(\min\left\{n_s \log_M n_s, \frac{N_s}{B}\log_{M/B}\frac{N_s}{B}\right\}\right)$ *операций ввода-вывода, тогда как для сортировки длинных строк их количество составляет* $\Theta\left(n_l \log_M n_l + \frac{N_l}{B}\right)$.

Граница для длинных строк оптимальна, тогда как про границу для коротких строк этого сказать нельзя. Сравнивая оптимальную границу для длинных строк с соответствующей границей из теоремы 7.6, легко заметить, что различие состоит в основании логарифма: в одном случае это M, в другом — M/B. Это показывает, что разбиение длинных строк во внутренней памяти доказуемо полезно для внешней сортировки строк. Верхняя граница получается объединением такой структуры данных, как B-дерево для строк (о ней пойдет речь в главе 9), с надлежащим методом буферизации. Что касается коротких строк, обратите внимание на то, что их граница ввода-вывода совпадает со стоимостью сортировки всех символов в строках средней длины N_s/n_s. Она равна $O\left(\frac{B}{\log_{M/B} M}\right)$. На практике в узком диапазоне $\frac{B}{\log_{M/B} M} < \frac{N_s}{n_s} < B$ затраты на сортировку коротких строк становятся равными $O(n_s \log_M n_s)$. В этом диапазоне сложность сортировки для модели B ниже, чем для модели A, что показывает возможную пользу от разбиения коротких строк во внутренней памяти.

Примечательно то, что лучший детерминированный алгоритм для модели C получен из алгоритма для модели B. Поскольку модель C также допускает разделение строк на диске, мы можем использовать рандомизацию и хеширование. Основная идея состоит в сжатии строк путем хеширования некоторых их частей. Хеширование не сохраняет лексикографический порядок, алгоритмы должны выбирать части строки для хеширования с помощью тщательно разработанного процесса сортировки, чтобы в итоге получить отсортированный результат. В [2] было доказано следующее (причем этот результат можно распространить и на более мощную модель, не зависящую от кэша).

Теорема 7.8. *В модели C задачу сортировки строк можно решить с помощью рандомизированного алгоритма с произвольно высокой вероятностью, использующего* $O\left(\frac{n}{B}\left(\log_{M/B}\frac{n}{M}\right)\left(\log\frac{N}{n}\right) + \frac{N}{B}\right)$ *операций ввода-вывода.*

Список литературы

1. *Arge L., Ferragina P., Grossi R., Vitter J. S.* On sorting strings in external memory // Proceedings of the 29th ACM Symposium on Theory of Computing (STOC), 540–548, 1997.
2. *Fagerberg R., Pagh A., Pagh R.* External string sorting: Faster and cacheoblivious // Proceedings of the 23rd Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science 3884, Springer, 68–79, 2006.
3. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to Algorithms. The MIT Press, third edition, 2009.
4. *Sedgewick R., Bentley J. L.* Fast algorithms for sorting and searching strings // Proceedings of the 8th Annual ACM — SIAM Symposium on Discrete Algorithms (SODA), 360–369, 19.

Глава 8
ЗАДАЧА О СЛОВАРЕ

> Слово «невозможно» можно найти только
> в лексиконе глупцов.
>
> *Наполеон Бонапарт*

В этой главе я познакомлю вас с рандомизированными и простыми, но при этом эффективными структурами данных, которые продуктивно решают классическую *задачу о словаре*. Эти решения дают ключ к некоторым задачам, которые в базовых курсах по алгоритмам обычно остаются без должного внимания.

Задача. Есть набор из n объектов $\mathcal{D}$, называемый *словарем*, уникально идентифицируемый ключами из *пространства* U. Нужно спроектировать структуру данных, эффективно поддерживающую следующие основные операции:

- `Search` (k). Проверяет, содержит ли словарь $\mathcal{D}$ объект x с ключом k = `key`[x], и в зависимости от результата возвращает `true` или `false`. В некоторых случаях целью может быть возврат объекта, связанного с определенным ключом, если таковой имеется. При отсутствии этого объекта возвращается значение `null`;
- `Insert` (x). Вставляет в словарь $\mathcal{D}$ объект x, индексированный ключом k = `key`[x]. Обычно предполагается, что до момента вставки объекты с таким ключом в словаре отсутствуют. Это условие легко проверяется предварительным запросом `Search` (k);
- `Delete` (k). Удаляет из словаря $\mathcal{D}$ объект, индексированный ключом k, если таковой имеется.

Если необходима поддержка всех трех операций, задача и соответствующая структура данных, которая ее решает, называются *динамическими*, в противном случае, когда достаточно поддерживать только операцию запроса, они называются *статическими*.

Бывают варианты словарей, в которых структура объекта x состоит из пары $\langle k, d \rangle$, где $k \in U$ — ключ, индексирующий x в словаре $\mathcal{D}$, а d — так называемые *дополнительные данные*, характеризующие x. Для простоты в этой главе такие словари я рассматривать не буду, ограничившись вариантами с обычными наборами ключей $S \subseteq U$, индексирующими объекты словаря. Это упростит обсуждение, так как операции поиска и обновления будут рассматриваться только для ключей, а не для целых объектов.

Но если контекст потребует дополнительных данных, речь снова пойдет об объектах и реализующих их парах. Более того, так как компьютеры представляют ключи в виде двоичных строк, без потери общности можно установить $U = \{0, 1, 2...\}$ как пространство неотрицательных целых чисел. Графически все это представлено на рис. 8.1.

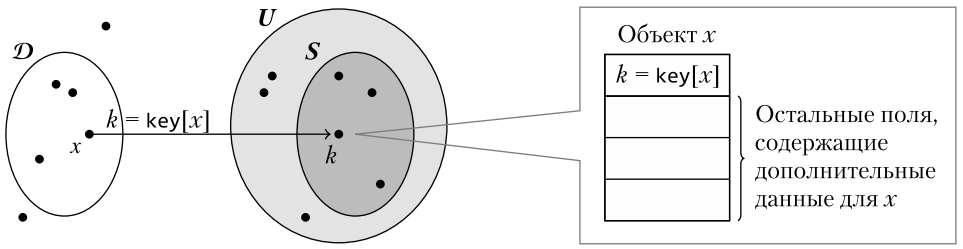


Рис. 8.1. Наглядный пример: *слева* — словарь объектов $\mathcal{D}$; *посередине* — его ключи k = key[x], образующие подмножество S большого пространства U. *Справа* графически показаны дополнительные данные, характеризующие объект x

В следующих разделах мы проанализируем три основные структуры данных: таблицы с прямой адресацией (или массивы), хеш-таблицы (и некоторые их сложные варианты) и фильтр Блума. Таблицы с прямой адресацией рассматриваются в учебных целях, потому что часто задача словаря допускает очень эффективные решения без использования сложных структур данных. При обсуждении хеш-таблиц я покажу, во-первых, как справиться с задачей проектирования хорошей хеш-функции (обычно об этом рассказывается в базовых курсах алгоритмов), во-вторых, как разработать так называемые *идеальные* хеш-таблицы, которые даже *в худшем случае оптимально* решают проблему статического словаря. В конце расскажу про элегантное кукушкино хеширование, с помощью которого можно эффективно управлять обновлениями словаря, гарантируя постоянное время запроса даже в худшем случае. Завершит эту главу рассказ про фильтр Блума — одну из наиболее широко используемых структур данных в контексте больших словарей и веб- или сетевых приложений. Его удивительная особенность состоит в том, что он гарантирует постоянное время операций запроса и обновления и, что еще более удивительно, его требования к пространству зависят от количества ключей n, но не от их длины. Возможность такого впечатляющего сжатия обеспечивается тем, что для каждого ключа сохраняется только *отпечаток* размером несколько битов. Правда, остается одна проблема — односторонняя ошибка при выполнении операции `Search` (k). Дело в том, что эта структура данных отвечает правильно, когда $k \in S$, но если это не так, может ответить неправильно (так называемый *ложноположительный результат*). Впрочем, я покажу, что вероятность этой ошибки можно математически ограничить функцией, которая экспоненциально убывает с зарезервированным для фильтра Блума пространством m или, что эквивалентно, с числом битов, выделенных для каждого ключа, то есть с его отпечатком. Достаточно взять m как постоянный множитель, немного превышающий n, чтобы вероятность ошибки стала пренебрежимо малой. Это делает фильтр Блума очень привлекательным в сканерах поисковых систем, системах хранения данных, системах P2P и т. п.

8.1. Таблицы с прямой адресацией

Простейшая структура данных для поддержки всех операций со словарем основана на двоичной таблице T размером $u = |U|$ бит. Между ключами и записями такой таблицы существует однозначное соответствие, поэтому запись $T[k]$ равна 1 тогда и только тогда, когда ключ $k \in S$. Если необходимо сохранить какие-то дополнительные данные для k, T реализуется как таблица указателей на эти данные. В этом случае мы имеем $T[k] \neq$ `NULL` тогда и только тогда, когда $k \in S$, а $T[k]$ указывает на область памяти, где хранятся дополнительные данные для k.

На базе такой таблицы довольно просто реализовать операции со словарем, в худшем случае выполняемые за постоянное оптимальное время. Основная проблема с этим решением заключается в том, что заполняемость таблицы зависит от размера пространства u, — подход оптимален при $n = \Theta(u)$. Когда словарь мал по сравнению с пространством ключей, много места тратится впустую и решение становится неприемлемым. Представим университет, который хранит данные студентов, индексированные по их идентификаторам. Если студентов миллионы, а идентификаторы закодированы целыми числами, например 4 байта, размер пространства ключей составит 2^{32}, то есть будет порядка миллиардов. Поэтому для уменьшения разреженности таблицы по отношению к размеру пространства ключей были разработаны более умные решения, по-прежнему гарантирующие эффективность операций со словарем. Среди предложенных вариантов выделяются хеш-таблицы и их многочисленные вариации.

8.2. Хеш-таблицы

Простейшие структуры данных для реализации словаря — это массивы и списки. Работая с массивами, мы имеем постоянный доступ к записям, но линейное время обновления, тогда как списки, наоборот, дают линейный доступ к элементам, но постоянное время обновления при каждом указании подлежащей обновлению позиции. Хеш-таблицы объединяют лучшее из этих подходов. Простейшая реализация такой таблицы — это *хеширование с цепочками*, основанное на *массиве списков*. Массив T размером m указывает на списки объектов словаря или в упрощенном случае на списки ключей. Отображение ключей на записи массива реализуется *хеш-функцией* $h : U \to \{0, 1, 2... m - 1\}$, то есть ключ k отображается в список, на который указывает $T[h(k)]$. Графический пример хеш-таблицы с цепочками приведен на рис. 8.2.

Давайте на мгновение предположим, что вычисление функции h занимает постоянное время. Этой теме посвящена значительная часть материала главы, поскольку общая эффективность предлагаемой схемы сильно зависит от того, насколько результативно функция h равномерно распределяет элементы между записями таблицы.

При наличии хорошей хеш-функции операции со словарем легко реализуются с хеш-таблицей, поскольку по своей сути это операции с массивом T и связанными с записями этого массива списками. Ключ k и связанный с ним объект отыскиваются

в списке $T[h(k)]$. Вставка объекта с ключом k состоит из добавления его в начало списка, на который указывает $T[h(k)]$. Удаление объекта с ключом k сводится к поиску k в списке $T[h(k)]$, а затем удалению результатов этого поиска. Операция вставки выполняется за постоянное время при условии, что $h(k)$ вычисляется за постоянное время. Время поиска и удаления линейно зависит от длины списка, на который указывает $T[h(k)]$. Следовательно, эффективность хеширования с цепочками зависит от способности хеш-функции h *равномерно распределять* ключи словаря среди m записей таблицы T. Чем равномернее распределены ключи, тем короче будет список для сканирования. Хуже всего, когда все ключи хешируются в одну и ту же запись массива T, формируя список длиной n. В этом случае хеш-таблица сводится к одному связанному списку и затраты на поиск составляют $\Theta(n)$.

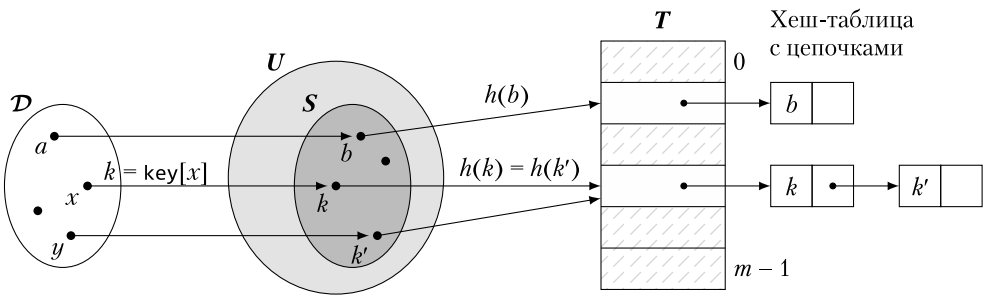


Рис. 8.2. Хеш-таблица с цепочками

Вот почему нас интересуют хорошие хеш-функции, которые случайным образом равномерно распределяют ключи по записям таблицы (так называемое *равномерное хеширование* [6]). Для таких хеш-функций каждый ключ $k \in S$ с *одинаковой вероятностью* будет хеширован в любой из m слотов в T *независимо* от местоположения остальных ключей словаря. Если h гарантирует это сильное свойство, можно легко доказать следующую теорему.

Теорема 8.1. *При равномерном хешировании существует хеш-таблица с цепочками размером m, в которой операция* `Search`(k) *по словарю из n объектов занимает ожидаемое время* $\Theta(1 + n/m)$. *Параметр* $\alpha = n/m$ *часто называют коэффициентом загрузки хеш-таблицы.*

Доказательство. В случае неудачного поиска (когда $k \notin S$) время выполнения процедуры `Search` (k) совпадает с временем полного сканирования списка $T[h(k)]$ и, соответственно, с его длиной. Учитывая равномерное распределение ключей, индуцированное хеш-функцией h, ожидаемая длина общего списка $T[i]$ составит $\sum_{k \in S} \mathcal{P}(h(k) = i) = |S| \times 1/m = n/m = \alpha$. Плюс 1 во временной сложности появился из-за постоянного времени вычисления $h(k)$.

При успешном поиске (когда $k \in S$) доказательство менее прямолинейно. Пусть k — это j-й ключ, вставленный в словарь S, в конец списка $\mathcal{L}(k) = T[h(k)]$. Для отслеживания такого ключа требуется всего один дополнительный указатель на список.

Количество элементов, проверяемых в процессе поиска, равно количеству элементов в $\mathcal{L}(k)$ плюс сам ключ k. Ожидаемую длину $\mathcal{L}(k)$ можно оценить как $(j-1)/m$ (ведь k — это j-й ключ, который нужно вставить), поэтому для успешного поиска ожидаемое время составит:

$$\frac{1}{n}\sum_{j=1}^{n}\left(1+\frac{j-1}{m}\right)=1+\frac{\alpha}{2}-\frac{1}{2m}.$$

Эту формулу можно переписать как $O\left(1+\frac{\alpha}{2}-\frac{1}{2m}\right)=O(1+\alpha)$. ■

Пространство, занимаемое хеш-таблицей, легко оценить, если учесть, что каждый указатель списка занимает $\Theta(\log n)$ бит, поскольку он должен индексировать один из n элементов, а каждый ключ словаря занимает $\Theta(\log u)$ бит, так как он извлекается из пространства U размером u. Интересно отметить, что по мере увеличения U хранилище ключей может превысить общее занимаемое таблицей пространство (например, если в роли ключей используются URL-адреса, для представления которых в среднем требуются сотни байтов). Это тонкое наблюдение пригодится при обсуждении конструкции фильтров Блума в разделе 8.7.

Следствие 8.1. *Хеш-таблица с цепочками занимает* $O((m+n)\log_2 n + n\log_2 u)$ *бит. Константы, скрытые в нотации «О большого», очень малы и близки к* 1.

Очевидно, что при известном размере словаря n можно спроектировать таблицу, состоящую из $m=\Theta(n)$ ячеек (следовательно, $\alpha=\Theta(1)$), и таким образом получить для всех операций со словарем ожидаемое постоянное время. Если n неизвестно, размер таблицы можно *менять* всякий раз, когда словарь становится слишком маленьким (много удалений) или слишком большим (много вставок). Идея состоит в том, чтобы начать с таблицы размером $m=2n_0$, где n_0 — начальное количество ключей словаря. Мы будем отслеживать текущее количество ключей словаря n в S (и, следовательно, в таблице T). Когда словарь становится слишком маленьким, то есть $n < n_0/2$, нужно уменьшить m вдвое и перестроить T, если же он становится слишком большим, то есть $n > 2n_0$, следует удвоить m и опять же перестроить T. *Перестроение* сводится к вставке текущих ключей словаря в новую таблицу надлежащего размера m и удалению старой таблицы.

Такая схема гарантирует пропорциональность размеров словаря n и таблицы m в любой момент, точнее, $n_0/2 = m/4 \leqslant n \leqslant m = 2n_0$. Это подразумевает, что $\alpha = m/n = \Theta(1)$. Поскольку на вставку одного ключа затрачивается время $O(1)$, а перестройка затрагивает n элементов, которые должны быть удалены из текущей таблицы, а затем вставлены в новую таблицу, общая стоимость перестроения составит $\Theta(n)$. Но эту цену приходится платить по крайней мере каждые $n_0/2 = \Theta(n)$ операций. В худшем случае эти операции будут полностью состоять только из вставок или удалений между последовательными перестроениями таблицы. Таким образом, стоимость перестроения $\Theta(n)$ может быть *амортизирована* за счет $\Omega(n)$ операций, которые привели к необходимости перестраивать таблицу. Поскольку на $\Omega(n)$ операций обновления сложностью $O(1)$ приходится не более одной операции перестроения

сложностью $\Theta(n)$, к фактической стоимости одного обновления добавляется $O(1)$. В целом это означает следующее.

Следствие 8.2. *При равномерном хешировании существует динамическая хеш-таблица с цепочками, поиск по которой занимает ожидаемое постоянное время, а операции вставки и удаления — ожидаемое амортизированное постоянное время, при этом используется пространство $O(n)$.*

8.2.1. Как спроектировать хорошую хеш-функцию

С вычислительной точки зрения равномерное хеширование гарантировать сложно, поскольку распределение вероятностей, в соответствии с которым размещаются ключи словаря, редко известно заранее, кроме того, может оказаться, что они размещаются не независимо друг от друга. Функция h отображает ключи из пространства размером u в целочисленный диапазон размером m, то есть ключи распределяются по m подмножествам $U_i = \{k \in U : h(k) = i\}$. Согласно принципу Дирихле, существует по крайней мере одно подмножество, размер которого превышает средний коэффициент загрузки u/m. Если пространство ключей достаточно велико для того, чтобы гарантировать неравенство $u/m \geqslant n$, выбрать словарь S можно как подмножество U_i, подходящее под условие $|U_i| \geqslant u/m$. В этом случае мы получаем наихудшее поведение хеш-таблицы, так как она превращается в связанный список длиной n.

Подобные рассуждения применимы к любой хеш-функции h. Это означает, что не существует достаточно надежной хеш-функции с *гарантированно* хорошим поведением. На практике для создания хеш-функций, которые *достаточно часто* работают хорошо, применяется набор эвристических правил. Принцип проектирования состоит в вычислении хеш-значения, не зависящего ни от одного регулярного шаблона, который может существовать среди ключей словаря. Две самые простые и известные схемы хеширования основаны на делении и умножении (более подробную информацию можно найти в любом классическом тексте по алгоритмам, например в [6]).

- **Хеширование с использованием деления.** Значение хеша вычисляется как остаток от деления ключа k на размер таблицы m, то есть $h(k) = k \bmod m$. Это довольно быстрая и хорошо ведущая себя функция, пока она зависит от большинства битов k. Поэтому при выборе значений m следует избегать степеней двойки, отдавая предпочтение простым числам. Для выбора больших простых чисел существуют как простые, но медленные (экспоненциальные по времени) алгоритмы, например знаменитый метод решета Эратосфена, так и быстрые алгоритмы, основанные на некотором рандомизированном или детерминированном тесте на простоту[1]. В общем случае затраты на выбор

[1] Самый известный рандомизированный тест на простоту — тест Миллера — Рабина. Недавно был предложен детерминированный тест, позволяющий доказать, что эта задача разрешима за детерминированное полиномиальное время. См.: https://ru.wikipedia.org/wiki/Простое_число.

простого числа незначительны по сравнению с затратами на построение хеш-таблицы.

- **Хеширование с использованием умножения.** Значение хеша вычисляется в два этапа: сначала ключ k умножается на константу A, где $0 < A < 1$, затем дробная часть kA умножается на m, а целая часть результата берется в качестве индекса для хеш-таблицы T. В этом случае выбор m некритичен и обычно в качестве этого множителя берут степень двойки, что упрощает этап умножения. В качестве A часто фигурирует значение $(\sqrt{5} - 1)/2 \cong 0{,}618$.

Понятно, что ни одна из предложенных схем хеширования не гарантирует хорошего поведения отображения — всегда может найтись словарь ключей, который сведет хеш-таблицу к связанному списку. Например, достаточно взять целочисленные ключи, кратные m, чтобы метод хеширования с использованием деления дал плохой результат. Впрочем, сейчас я покажу схему хеширования, которая достаточно надежна для того, чтобы для любого входного словаря гарантировать хорошее поведение в ожидании.

8.3. Универсальное хеширование

Воспользуемся методом подсчета, чтобы понять, почему трудно гарантировать такое свойство, как однородность, которое необходимо для *хороших* хеш-функций. Напомню, что нас интересуют хеш-функции, отображающие ключи из пространства U в целые числа в $\{0, 1... m - 1\}$. Каждый ключ среди $u = |U|$ допускает отображение в любую из m ячеек хеш-таблицы, поэтому общее количество таких хеш-функций составит m^u. Чтобы гарантировать равномерное распределение ключей и независимость между ними, хеш-функция должна быть любым из этих отображений. Но в этом случае для ее представления потребуется $\Omega(\log_2 m^u) = \Omega(u \log m)$ бит, что слишком много с точки зрения занимаемого пространства и времени вычислений. Компьютеру с длиной ячейки памяти 64 бита, то есть $u = 2^{64}$, для хранения хеш-функции потребуется пространство порядка зеттабайта.

С практической точки зрения хеш-функции имеют ряд недостатков, часть которых я уже упоминал. В этом разделе покажу мощную универсальную парадигму хеширования, которая преодолевает эти недостатки с помощью правильного проектирования *класса* возможных хеш-функций и применения *рандомизации* для выбора одной из них. Аналогичный процесс использовался в главе 5 для обеспечения большей надежности при выборе опорного элемента в процедуре `QUICKSORT`. Тогда от опорного элемента в фиксированной позиции отказывались в пользу *равномерного, случайным образом сделанного* выбора этого элемента из базового массива для сортировки. В результате никакие входные данные *в принципе* не могли оказаться плохими с точки зрения стратегии выбора опорного элемента, ведь рандомизация этого процесса позволяет распределить риски между различными вариантами, гарантируя, что в большинстве случаев выбор приведет к хорошо сбалансированному разбиению.

Универсальное хеширование использует этот же алгоритмический подход. Мы не устанавливаем хеш-функцию заранее, как не фиксировали положение опорной точки, но выбираем ее *равномерно случайным образом* из *правильно определенного* набора хеш-функций (сравните со случайным выбором опорной точки). Этот набор определен так, чтобы увеличивать вероятность выбора хорошего хеша для текущего входного набора ключей S (сравните со сбалансированным разбиением). Хорошей считается хеш-функция, которая может быть вычислена за постоянное время и минимизирует количество *коллизий* между парами ключей словаря. Благодаря рандомизации даже при фиксированном наборе S алгоритм каждый раз будет вести себя по-разному. Приятное свойство универсального хеширования заключается в том, что в среднем производительность оказывается ожидаемой. Теперь формализуем эти идеи.

Определение 8.1. Пусть $\mathcal{H}$ — конечный набор хеш-функций, отображающих заданное пространство ключей U в целые числа в диапазоне $\{0, 1... m - 1\}$. Класс хеш-функций называют *универсальным* тогда и только тогда, когда для любой фиксированной пары различных ключей $x, y \in U$ выполняется условие:

$$\left|\left\{h \in \mathcal{H} : h(x) = h(y)\right\}\right| \leqslant \frac{|\mathcal{H}|}{m}.$$

Другими словами, класс $\mathcal{H}$ определяется таким образом, что у случайно выбранной хеш-функции h из этого набора вероятность вызвать коллизию[1] двух фиксированных ключей x и y составляет не более $1/m$. Именно этим базовым свойством мы пользовались при проектировании хеширования с цепочками (см. доказательство теоремы 8.1). В теореме 8.2 я покажу, что теорему 8.1 и ее следствия 8.1 и 8.2 можно переформулировать, заменив *идеальное* простое равномерное хеширование *эффективным* универсальным хешированием. Определение «эффективное» появилось потому, что в подразделе 8.3.1 я покажу реально универсальный класс хеширования, который переведет все эти математические размышления в конкретную форму.

Теорема 8.2. *Дана хеш-таблица с цепочками $T[0, m - 1]$, причем хеш-функция h выбирается равномерно случайным образом из универсального класса H. В этом случае ожидаемая длина списков в T независимо от вида входного словаря ключей S по-прежнему не превышает $1 + \alpha$, где α — коэффициент загрузки T.*

Доказательство. В данном случае ожидание берется по результатам выбора h в $\mathcal{H}$ и не зависит от распределения ключей в S. Для каждой пары ключей $x, y \in S$ определим индикаторную случайную переменную I_{xy}, которая равна 1, если эти два ключа сталкиваются согласно заданному условию для $h \in \mathcal{H}$, а именно $h(x) = h(y)$. В противном случае эта переменная принимает значение 0. По определению

[1] Если ввести некоторую константу $c \geqslant 1$, это определение можно несколько ослабить до так называемого c-универсального хеширования, в котором вероятность коллизии будет ограничена сверху отношением c/m.

универсального класса, учитывая случайный выбор $h \in \mathcal{H}$, получим $\mathcal{P}(I_{xy} = 1) = \mathcal{P}(h(x) = h(y)) \leqslant 1/m$. В результате имеем $E[I_{xy}] = 1\mathcal{P}(I_{xy} = 1) + 0\mathcal{P}(I_{xy} = 0) = \mathcal{P}(I_{xy} = 1) \leqslant 1/m$, где ожидание вычисляется по случайно выбранному h.

Теперь определим для каждого ключа $x \in S$ случайные величины N_x, подсчитывающие количество ключей, которые отличны от x и при этом хешируются в запись $T[h(x)]$, то есть сталкиваются с x. Можно записать, что $N_x = \sum_{y \in S,\, y \neq x} I_{xy}$. В силу линейности математического ожидания предполагаемая длина списка, на который указывает $T[h(x)]$, будет $E[N_x] = \sum_{y \in S,\, y \neq x} E[I_{xy}] = (n-1)/m < \alpha$. Остается добавить 1 из-за x, и теорема доказана. ■

Обратите внимание на то, что мы оценили математическое ожидание временных ограничений для хеширования, но не исключено появление очень длинных списков, возможно содержащих до $\Theta(n)$ элементов. Это удовлетворяет теореме 8.2, но ситуация, конечно, не очень хорошая, ведь если распределение поисков отдаст предпочтение ключам из таких списков, время окажется значительно больше ожидаемого. Чтобы обойти эту проблему, следует также гарантировать маленькую верхнюю границу для длины *самого длинного* списка в T. Именно об этом идет речь в теореме 8.3.

Теорема 8.3. Пусть $T[0, n-1]$ — хеш-таблица с цепочками, хеш-функция которой выбрана из универсального класса H равномерно случайным образом. При вставке в T словаря S из n ключей длина самого длинного списка в T составит $O\left(\frac{\log n}{\log\log n}\right)$ с высокой вероятностью, не меньшей $1 - 1/n$.

Доказательство. Пусть h — хеш-функция, выбранная равномерно случайным образом из класса $\mathcal{H}$, и пусть $Q(k)$ — вероятность того, что ровно k ключей словаря S помещаются функцией h в определенную ячейку таблицы T. Учитывая универсальность h, вероятность попадания произвольного ключа в фиксированную ячейку ограничена сверху значением $1/n$, таким образом, ожидаемое количество ключей на ячейку будет равно 1.

Пусть X_j — случайная переменная, обозначающая количество ключей, сопоставленных слоту j при $j = 0, 1 \ldots n-1$. Выбрать k ключей из словаря S можно $\binom{n}{k}$ способами, поэтому вероятность того, что в ячейке окажется по крайней мере k ключей, ограничена сверху значением:

$$Q(k) = \binom{n}{k}\left(\frac{1}{n}\right)^k \leqslant \left(\frac{ne}{k}\right)^k \left(\frac{1}{n}\right)^k \leqslant \left(\frac{e}{k}\right)^k,$$

в котором для биномиального коэффициента используется известная верхняя граница $\binom{n}{k} \leqslant (ne/k)^k$.

Поскольку при $k > e$ вероятность $Q(k)$ убывает, нужно найти такое значение k_0, чтобы при $k \geqslant k_0$ было справедливо неравенство $Q(k) \leqslant 1/n^2$. Я покажу, что это значение $k_0 = \frac{c \ln n}{\ln \ln n}$, где константа $c \geqslant 4$. Достаточно заметить, что неравенство $Q(k) \leqslant 1/n^2$ соблюдается только при условии $(e/k)^k \leqslant 1/n^2$, то есть $(k/e)^k \geqslant n^2$. Взяв логарифм обеих сторон этого неравенства, получим $k \ln(k/e) \geqslant \ln n^2$. Простые алгебраические преобразования позволяют убедиться в том, что для k_0 неравенство соблюдается.

Утверждение теоремы справедливо, ведь можно объединить границы n ячеек таблицы T и таким образом показать, что вероятность попадания в одну ячейку более k_0 ключей не превышает $n\, Q(k_0) \leqslant n(1/n^2) = 1/n$.

Здесь следует сделать два замечания. Первое: *с высокой вероятностью* существует граница максимальной длины списка в таблице T, но ее можно превратить *в границу для наихудшего случая*, применив к построению T небольшое допущение. При равномерном случайном выборе хеш-функции $h \in \mathcal{H}$ каждый ключ словаря S хешируется в T, а затем мы проверяем, не превышает ли длина самого длинного списка значения $2\log n/\log\log n$. В случае положительного результата проверки мы используем T для последующего поиска, в противном случае происходят повторный выбор $h \in \mathcal{H}$ и повторная вставка всех ключей словаря в T. Нескольких попыток хватит, чтобы удостовериться в существовании этой границы[1], гарантируя ожидаемое время построения $O(n)$ и ожидаемое время поиска $O(\log n/\log\log n)$ в худшем случае. ■

Второе замечание состоит в том, что этот результат можно улучшить, используя две или более, например d, хеш-функции h_i и хеш-таблицу T, ячейки которой представляют собой корзины, содержащие все хешированные ключи словаря. Все дело тут в способе, которым d хеш-функций развертываются для заполнения корзин. Операция `Insert`(k) проверяет загрузку d ячеек $T[h_i(k)]$, а затем вставляет k в наименее заполненную ячейку. При равномерной загрузке ячеек алгоритм выбирает $T[h_1(k)]$. Операция `Search`(k) просматривает все d списков $T[h_i(k)]$, поскольку заранее не известно, насколько загружена каждая ячейка. Из-за своей алгоритмической структуры эту схему знают также как *d-кратное хеширование*. Время выполнения операции `Insert`(k) оценивается как $O(d)$, тогда как для `Search`(k) оно определяется общей длиной d списков, по которым выполняется поиск. Эту длину можно ограничить сверху произведением d и длины самого длинного списка в T, что, как ни странно, для $d \geqslant 2$ будет равно $\frac{\log\log n}{\log d} + O(1)$ (подробнее см. [5]). В литературе предлагается несколько вариантов этой идеи. Одним из самых известных и эффективных является *d-кратное левое хеширование*, в котором используются d таблиц размером m/d, причем каждая индексируется отдельной хеш-функцией, а связи управляются вставкой ключа в самую левую таблицу.

---

[1] Для утверждения того, что самый длинный список, превышающий среднее больше чем в два раза, может появиться с вероятностью $\leqslant 1/2$, я воспользовался неравенством Маркова (см.: https://ru.wikipedia.org/wiki/Неравенство_Маркова).

Интересно отметить, что двукратное хеширование предпочтительнее любого другого d-кратного при $d > 2$, поскольку в этом случае длина самого длинного списка в T уже достигает $O(\log \log n)$. Эту границу невозможно улучшить асимптотически, какая бы константа d ни была взята, а большие значения d замедляют операции поиска в d раз. Более того, сравнивая границы в случае двукратного хеширования и в классическом случае хеширования с цепочками (здесь $d = 1$), рассматриваемого в теореме 8.3, легко заметить, что экспоненциальное улучшение при двукратном хешировании достигается за счет использования всего лишь еще одной хеш-функции. Этот удивительный результат также известен в литературе как *сила двух выборов* (two-choice hashing), потому что предпочтение менее загруженной ячейки из двух случайно выбранных позволяет экспоненциально сократить длину самого длинного списка в T.

В заключение отмечу, что этот результат можно использовать для проектирования хеш-таблицы, которая экономит место под указатели и увеличивает локальность ссылок в операциях поиска и обновления, следовательно сокращая количество промахов кэша/ввода-вывода. Суть заключается в двукратном хешировании над таблицей *небольших корзин фиксированного размера*. Если сделать размер корзины равным $\gamma \log \log n$ для некоторой малой константы $\gamma > 1$, то *с высокой вероятностью* будет гарантировано, что, во-первых, ни одна корзина не будет переполнена, во-вторых, любая операция поиска и обновления не будет допускать более двух промахов кэша.

8.3.1. Существуют ли универсальные хеш-функции

Ответ на этот вопрос положительный — как ни удивительно, универсальные хеш-функции легко построить, что я и покажу в этом подразделе на трех конкретных примерах. Без потери общности можно предположить, что размер таблицы m — простое число, а ключи — целые числа, представленные в виде строк по $\log_2 |U|$ бит[1]. Примем $r = \dfrac{\log_2 |U|}{\log_2 m}$ и округлим его до целого. Разложим каждый ключ k на r частей по $\log_2 m$ бит каждая, так что $k = [k_0, k_1 \ldots k_{r-1}]$. Очевидно, что каждое k_i является целым числом, меньшим чем m, поскольку оно представлено в $\log_2 m$ битах. То же самое сделаем для общего целочисленного параметра $a = [a_0, a_1 \ldots a_{r-1}] \in [1, |U| - 1]$, который определяет универсальный класс хеш-функций $\mathcal{H}$ следующим образом: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \bmod m$. Размер $\mathcal{H}$ равен $|U| - 1$, ведь на каждое значение $a > 0$ приходится по одной функции.

Теорема 8.4. *Примером универсального класса может служить множество хеш-функций вида* $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \bmod m$, *где* m *— простое число,* a *— положительное целое число, меньшее чем* $|U| = m^r$.

---

[1] Для этого ключ предварительно дополняют нулями, чтобы сохранить его значение.

Доказательство. Предположим, что x и y — два ключа, различающихся хотя бы на 1 бит. Для простоты будем считать, что отличающийся бит попадает в самый значимый блок ($\log m$)-битов, то есть $x_0 \neq y_0$. Согласно определению 8.1, требуется подсчитать, сколько хеш-функций вида $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \bmod m$ вызывают коллизии этих двух ключей, или, что эквивалентно, сколько существует $a > 0$, для которых $h_a(x) = h_a(y)$. Так как $x_0 \neq y_0$, а мы работаем с остатком от деления на простое число m, должно существовать обратное $(x_0 - y_0)$ по модулю m число, которое является целым и лежит в диапазоне $[1, |U| - 1]$. Обозначим (несколько погрешив против традиции) это число $(x_0 - y_0)^{-1}$ и получим:

$$h_a(x) = h_a(y) \Leftrightarrow \sum_{i=0}^{r-1} a_i x_i \equiv \sum_{i=0}^{r-1} a_i y_i \bmod m \Leftrightarrow$$
$$\Leftrightarrow a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r-1} a_i (x_i - y_i) \bmod m \Leftrightarrow$$
$$\Leftrightarrow a_0 \equiv \left(-\sum_{i=1}^{r-1} a_i (x_i - y_i) \right) (x_0 - y_0)^{-1} \bmod m.$$

Окончательное уравнение фактически показывает, что при любом выборе $[a_0, a_1 \ldots a_{r-1}]$ — а это будет $m^{r-1} - 1$, поскольку нулевая конфигурация исключена, — только при одном варианте для $a_0 \neq 0$ (он указан в последней строке формулы) происходит коллизия x и y. Соответственно, существует $m^{r-1} - 1$ вариантов выбора для a, вызывающих коллизию $x \neq y$. Это число можно записать как $m^{r-1} - 1 = (m^r/m) - 1 = (|U|/m) - 1 \leqslant (|U| - 1)/m = |\mathcal{H}|/m$. Так что класс $\mathcal{H}$ попадает под определение 8.1 универсального класса хешей. ■

Предыдущее определение можно скорректировать таким образом, чтобы оно выполнялось для любого m, а не только для простых значений. Основная идея заключается в создании *вложенных вычислений по модулю* с использованием большого простого числа $p > |U|$ и целого числа $m << |U|$, равного размеру хеш-таблицы, которую мы хотим настроить. После чего определяется хеш-функция, параметризованная двумя целыми значениями, $a \neq 0$ и $b \geqslant 0$, такими, что $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$, а затем определяется множество $\mathcal{H}_{p,m} = \bigcup_{a>0,\, b\geq 0} \{h_{a,b}\}$, которое, как можно показать, универсально.

Эти два примера универсальных классов хеш-функций требуют операций по модулю над произвольными целыми числами. Существуют и другие универсальные классы хеш-функций, которые вычисляются быстрее, поскольку базируются на операциях по модулю с целыми числами, являющимися степенями двойки. Их легко реализовать с помощью быстрых сдвигов регистров. Интересный и эффективный пример дает *схема умножения — сложения — сдвига* (см. [7], [12]). В этой схеме предполагается, что $|U| = 2^w$, а размер таблицы $m = 2^\ell < |U|$. Затем определяется класс $\mathcal{H}_{w,\ell}$ как содержащий хеш-функции вида $h_{a,b}(k) = (ak + b \bmod 2^w) \operatorname{div} 2^{w-\ell}$, где a — нечетное целое число, меньшее чем $|U|$, а $b \geqslant 0$. Обратите внимание на то, что $h_{a,b}(k)$ сначала выбирает наименее значимые w бит (из-за остатка от деления на 2^w), а затем уже среди них выбирает наиболее значимые ℓ бит (из-за деления на $2^{w-\ell}$). В некотором смысле эти ℓ бит попадают в «середину» целого числа $(ak + b)$

и соответствуют неотрицательному целому числу, которое меньше 2^{ℓ} ($= m$). Можно доказать, что класс $\mathcal{H}_{w,\ell}$ универсален и допускает эффективную реализацию без операций по модулю посредством регистровых сдвигов по словам памяти из $2w$ бит.

8.4. Простая (статическая) идеальная хеш-таблица

Хеш-таблицы считаются очень эффективными на практике, но в теории их оценки являются *ожидаемыми* и зависят от свойств базового универсального класса хеш-функций и/или от распределения ключей словаря. Это неверное представление, так как в настоящее время существует множество конструкций хеш-таблиц, которые элегантны, просты в реализации и обеспечивают для запросов и обновлений хорошую производительность в худшем случае. Проектированию именно таких эффективных хеш-функций посвящен остаток этой главы. Подобные схемы хорошо подходят к ситуации, когда операция поиска сводится к *просмотру*, то есть когда требуется только узнать, существует ли в словаре S нужный ключ.

Начнем со статического словаря S, для которого не предполагается ни добавлять ключи, ни удалять их оттуда.

Определение 8.2. Хеш-функцию $h : U \to \{0, 1... m - 1\}$ называют *идеальной* для словаря S с n ключами тогда и только тогда, когда для любой пары ключей $k', k'' \in S$ соблюдается условие $h(k') \neq h(k'')$.

Простой подсчет аргументов показывает, что для достижения идеального хеширования требуется соблюдение условия $m \geqslant n$. При $m = n$, то есть минимально возможном значении, идеальная хеш-функция называется *минимальной* (minimal perfect hash function, MPHF). Хеш-таблица T, использующая `MPHF` h, гарантирует время поиска $O(1)$ в худшем случае и не расходует дополнительной памяти, ведь эта функция имеет размер словаря S, а ключи можно сохранять непосредственно в ячейках таблицы. Таким образом, идеальное хеширование является своего рода идеальным вариантом таблиц с прямой адресацией (см. раздел 8.1), ведь оно достигает постоянного времени поиска, как эти таблицы, но, в отличие от них, оптимального линейного пространства размером S.

Идеальная минимальная хеш-функция называется *сохраняющей порядок* (для краткости `OP(MP)HF`) только при условии, что $\forall k' < k'' \in S, h(k') < h(k'')$. Очевидно, что когда функция h также минимальна, то есть $m = n$, то $h(k)$ возвращает ранг ключа в упорядоченном словаре S. Разумеется, что свойство `OPMPHF` строго зависит от словаря S, на основе которого построена функция h: меняя S, можно разрушить это свойство, поэтому его трудно поддерживать в динамическом сценарии.

В этом разделе я ограничусь *статическими* словарями и, следовательно, фиксированным S. Затем в разделе 8.5 эта конструкция будет расширена до динамического словаря, и, наконец, в разделе 8.6 я вернусь к статическому словарю, чтобы показать вам проектирование идеальной, статической, минимальной и сохраняющей порядок схемы хеширования.

Ключевой алгоритмический метод проектирования статической идеальной схемы хеширования использует двухуровневый подход с развертыванием группы таблиц и универсальных хеш-функций: одна для первого уровня, остальные — для второго. Точнее, первый уровень представляет собой хеш-таблицу T_1 размером $m = n$, распределение ключей в которой происходит в соответствии с универсальной хеш-функцией, скажем $h_1(k) : U \to \{0, 1... n - 1\}$. Для управления ключами, попадающими в одну ячейку $T_1[j]$ при $j = 0, 1... n - 1$, спроектируем вторую хеш-таблицу $T_{2,j}$, к которой будет обращаться другая универсальная хеш-функция $h_{2,j}$. Она должна быть идеальной для ключей, сталкивающихся в $T_1[j]$, и, таким образом, не вызывать никаких коллизий в $T_{2,j}$. В целом получится максимум $1 + n$ универсальных хеш-функций: h_1 для первого уровня и n хеш-функций $h_{2,j}$ — для второго. Ключевой результат, который нужно доказать, заключается в том, что любую функцию $h_{2,j}$ можно спроектировать так, чтобы гарантировать отсутствие коллизий в $T_{2,j}$, и что общая занятость пространства всех хеш-таблиц второго уровня ограничена сверху как $O(n)$. С точки зрения эффективности запроса поиск ключа k сводится к двум просмотрам таблицы (следовательно, к времени $O(1)$ в худшем случае): во-первых, просмотру $T_1[j]$, где $j = h_1(k)$, во-вторых, просмотру таблицы $T_{2,j}$, на которую указывает $T_1[j]$. Псевдокод такой операции представлен в алгоритме 8.1.

Алгоритм 8.1. Процедура Search(k) в идеальной хеш-таблице T

```
1:  Пусть h1 и h2,j — универсальные хеш-функции, определяющие T;
2:  j = h1(k);
3:  if T1[j] = NULL then
4:      return false; // k ∉ S
5:  end if
6:  T2,j = T1[j];
7:  i = h2,j(k);
8:  if T2,j[i] = NULL or T2,j[i] ≠ k then
9:      return false; // k ∉ S
10: end if
11: return true; // k ∈ S
```

Следующая теорема важна для определения размера вторых хеш-таблиц.

Теорема 8.5. *Если q ключей сохранить с помощью универсальной хеш-функции в хеш-таблице размером $s = q^2$, то ожидаемое число коллизий среди них будет меньше* $1/2$. *Следовательно, вероятность хотя бы одной коллизии в этой таблице будет меньше* $1/2$.

Доказательство. Для q ключей существует $\binom{q}{2} < q^2/2$ допускающих коллизию пар. Если хеш-функция h равномерно и случайным образом выбирается из универсального класса, то согласно определению 8.1 вероятность коллизии каждой

конкретной пары составит $1/s$. Таким образом, при $s = q^2$ ожидаемое число коллизий ограничено сверху как $\binom{q}{2}\frac{1}{s} < q^2 \Big/ \left(2q^2\right) = \frac{1}{2}$. Для доказательства второго утверждения теоремы достаточно применить неравенство Маркова, то есть $\mathcal{P}(X \geqslant 2E[X]) \leqslant 1/2$, к случайной величине X, которая показывает общее число коллизий в хеш-таблице T. ■

Обозначим S_j ключи словаря, сопоставленные записи $T[j]$ хеш-функцией первого уровня h_1, при этом $n_j = |S_j|$. Теорема 8.5 доказывает, что если размер m_j хеш-таблицы T_j сделать равным $(n_j)^2$ (квадрату числа ключей, хешированных в $T[j]$), то случайно выбранная универсальная хеш-функция второго уровня $h_{2,j}$ идеально подойдет для ключей S_j с довольно заметной вероятностью для каждого $j = 0, 1... n - 1$. Следовательно, случайный выбор $h_{2,j}$ можно повторять до тех пор, пока это свойство не станет гарантированным. Фактически теорема 8.5 утверждает, что двух итераций достаточно, чтобы гарантировать его для каждой записи. Осталось доказать, что общий размер хеш-таблиц второго уровня ограничен сверху как $O(n)$ и, соответственно, общий размер двухуровневой схемы хеширования будет $n + O(n) = \Theta(n)$.

Теорема 8.6. *Если поместить n ключей в хеш-таблицу размером $m = n$, используя универсальную хеш-функцию, получим $E\left[\sum_{j=0}^{n-1}\left(n_j\right)^2\right] < 2n$, где ожидание относится к случайному выбору хеш-функции в ее универсальном классе. В терминах двухуровневой схемы хеширования это означает, что ожидаемый общий размер всех хеш-таблиц второго уровня $T_{2,j}$ меньше $2n$, учитывая, что мы установили $m_j = (n_j)^2$ для каждого $j = 0, 1... n - 1$.*

Доказательство. Рассмотрим справедливое для любого положительного целого числа a тождество $a^2 = a + 2\binom{a}{2}$ и отметим, что $n = \sum_{j=0}^{n-1} n_j$, поскольку каждый ключ словаря отображается функцией h_1 в одну запись таблицы. Воспользуемся этими двумя равенствами в следующих вычислениях:

$$E\left[\sum_{j=0}^{n-1}\left(n_j\right)^2\right] = E\left[\sum_{j=0}^{n-1}\left(n_j + 2\binom{n_j}{2}\right)\right] =$$

$$= E\left[\sum_{j=0}^{n-1} n_j\right] + 2E\left[\sum_{j=0}^{n-1}\binom{n_j}{2}\right] =$$

$$= n + 2E\left[\sum_{j=0}^{n-1}\binom{n_j}{2}\right].$$

В одной коллизии принимает участие пара ключей, поэтому число коллизий в ячейке $T[j]$ равно количеству отображаемых в эту ячейку пар, то есть $\binom{n_j}{2}$. Следовательно, сумма в последнем равенстве учитывает общее число коллизий, вызванных хеш-функцией h_1. Если рассуждения, которые применялись для доказательства

теоремы 8.5, повторить для $q = n$ ключей и таблицы T_1 размером $s = n$, то для общего числа коллизий получим ожидаемое значение $\binom{n}{2}\frac{1}{n} = \frac{n(n-1)}{2n} < \frac{n}{2}$. Подставляя эту верхнюю границу в сумму $\sum_{j=0}^{n-1}\binom{n_j}{2}$, получим утверждение теоремы. ■

Важно отметить, что каждая хеш-функция второго уровня $h_{2,j}$ выбирается независимо от других. Поэтому, если она оказывается неидеальной и вызывает коллизии среди ключей n_j в таблице $T_{2,j}[0, (n_j)^2 - 1]$, ее можно перевыбрать из ее универсального класса, никак не затрагивая другие хеш-функции второго уровня. Вы уже могли убедиться в том, что это справедливо и для хеш-функции первого уровня h_1. Псевдокод для построения такой двухуровневой схемы хеширования представлен в алгоритме 8.2, а теоремы 8.5 и 8.6 гарантируют, что ожидаемое количество перевыборов для одной хеш-функции является небольшой константой, поэтому ожидаемое общее время построения составляет $O(n)$.

Алгоритм 8.2. Создание идеальной хеш-таблицы

1: Равномерный случайный выбор хеш-функции из универсального класса h_1 : $U \to \{0, 1, ..., n - 1\}$;
2: Инициализируем $n_j = 0$, $S_j = \varnothing$, для всех $j = 0, 1, ..., n - 1$;
3: **for** $k \in S$ **do**
4: Добавляем k в набор S_j, где $j = h_1(k)$;
5: $n_j = n_j + 1$;
6: **end for**
7: $L = \sum_{j=0}^{n-1}(n_j)^2$;
8: **if** $L \geqslant 2n$ **then**
9: Повторяем алгоритм с шага 1;
10: **end if**
11: **for** $j = 0, 1... n - 1$ **do**
12: Выделяем таблицу $T_{2,j}$ размером $m_j = (n_j)^2$;
13: Из универсального класса выбираем равномерно случайным образом $h_{2,j}$: $U \to \{0, 1... m_j - 1\}$;
14: **for** $k \in S_j$ **do**
15: $i = h_{2,j}(k)$;
16: **if** $T_{2,j}[i] \neq$ NULL **then**
17: Удаляем $T_{2,j}$ и повторяем с шага 12;
18: **end if**
19: $T_{2,j}[i] = k$;
20: **end for**
21: **end for**

Пример двухуровневой схемы хеширования для набора S из $n = 9$ целочисленных ключей демонстрирует рис. 8.3. В нем используются универсальные хеш-функции вида $h(k) = (ak + b \bmod 19) \bmod s$, где s — размер таблицы, индексированной этой хеш-функцией. Для хеш-таблицы первого уровня T_1 задано, что $m = n = 9$ и $s = m$, $a = 1$ и $b = 0$. Для хеш-таблиц второго уровня $T_{2,j}$ задано, что $m_j = (n_j)^2$. В левой части рисунка показаны три параметра ($s = m_j$, a_j, b_j), участвующие в вычислении $h_{2,j}$, а в правой части — содержимое $T_{2,j}[0, m_j - 1]$. Условие для общего размера хеш-таблиц второго уровня из теоремы 8.6 выполнено: $L = 1 + 1 + 4 + 4 + 1 + 4 = 15 < 2n = 18$.

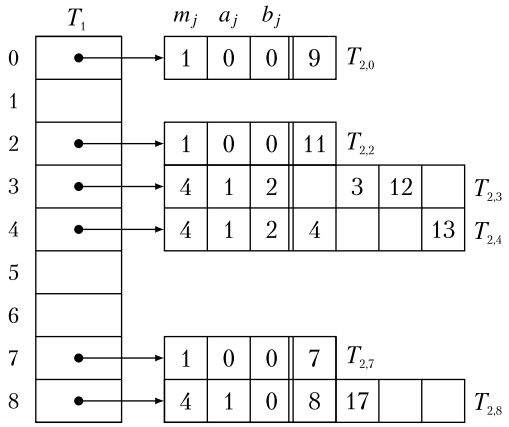


Рис. 8.3. Идеальная хеш-таблица для словаря $S = \{3, 4, 7, 8, 9, 11, 12, 13, 17\}$ с универсальными хеш-функциями вида $h(k) = (ak + b \bmod 19) \bmod s$, где s — размер индексированной этой хеш-функцией хеш-таблицы

Воспользовавшись псевдокодом алгоритма 8.1 для поиска ключа $k = 14$, получим $j = h_1(14) = (1 \times 14 + 0 \bmod 19) \bmod 9 = 5$. Это дает нам доступ к ячейке $T_1[5]$, в которой оказывается значение `NULL`, таким образом, получается, что $14 \neq S$. Теперь попробуем взять $k = 13$. В этом случае $j = h_1(13) = (1 \times 13 + 0 \bmod 19) \bmod 9 = 4$. Ячейка $T_1[4]$ указывает на таблицу $T_{2,4}$, а посещенная запись находится в позиции $h_{2,4}(13) = (1 \times 13 + 2 \bmod 19) \bmod 4 = 3$, которая не является `NULL`, потому что $T_{2,4}[3] = 13$. Таким образом, алгоритм убедился, что $13 \in S$. Последний случай иллюстрируется ключом $k = 18$, который обращается к хеш-таблице первого уровня в $T_1[0]$ и хеш-таблице второго уровня $T_{2,0}[0] = 9 \neq 18$, что приводит алгоритм к выводу: $18 \in S$.

8.5. Кукушкино хеширование

Для динамических словарей необходима другая схема идеального хеширования. Я покажу эффективное и элегантное решение, называемое *кукушкиным хешированием* [10], с ожидаемым постоянным временем для обновлений и постоянным временем поиска в худшем случае. Единственным недостатком этого подхода в его

предварительной формулировке было использование двух $O(\log n)$-независимых[1] хеш-функций. В более новых публикациях это требование смягчено до простого двойного универсального хеширования [1], но я буду придерживаться исходной схемы из-за ее простоты.

Кукушкино хеширование, если говорить коротко, добавляет к d-кратному хешированию, описанному в разделе 8.3, возможность перемещать ключи словаря между записями таблицы с целью устранения коллизий. В простейшей форме это две хеш-функции, h_1 и h_2, и одна таблица T размером m. Любой ключ k хранится либо в $T[h_1(k)]$, либо в $T[h_2(k)]$, что делает операции поиска и удаления тривиальными: достаточно найти k в обеих записях и при необходимости удалить его. Немного сложнее выглядит вставка ключа, поскольку она может спровоцировать *каскадное* перемещение ключей в таблице. Именно этот процесс и дал название подходу, напоминающему поведение отдельных видов кукушек, которые, вылупляясь в гнезде других птиц, выталкивают оттуда яйца или птенцов хозяев. При вставке ключа k алгоритм просматривает в таблице T позиции $h_1(k)$ и $h_2(k)$. Если обе пусты, выбирается $h_1(k)$. Когда пуста только одна из них, ключ сохраняется в этой позиции и вставка завершается. В противном случае место для k создается вытеснением одного из двух ключей, сохраненных в этих двух записях таблицы. Обычно вытесняется ключ из $T[h_1(k)]$. Вытесненный ключ начинает играть роль k, и процедура вставки повторяется.

На этом этапе следует учитывать один момент. Если вытесняемый ключ (назовем его y) был сохранен в $T[h_1(k)]$, то для $i = 1$ или $i = 2$ будет $T[h_i(y)] = T[h_1(k)]$. Это означает, что, когда заняты обе позиции, $T[h_1(y)]$ и $T[h_2(y)]$, подлежащий вытеснению на втором шаге ключ нельзя выбирать из записи, хранящей $T[h_1(k)]$, ведь там находится ключ k и такой выбор приведет к тривиальному бесконечному циклу вытеснений по этой записи между ключами k и y. Поэтому алгоритм тщательно избегает вытеснения ранее вставленного ключа. Тем не менее такие циклы могут возникать (например, в тривиальном случае, когда $\{h_1(k), h_2(k)\} = \{h_1(y), h_2(y)\}$). Более того, их длина может оказаться произвольной, поэтому крайне важно правильно задать *условия выхода*, добавив туда эту ситуацию. В этом случае алгоритм перевыбирает две хеш-функции и повторно хеширует все ключи словаря аналогично тому, как делалось для идеального хеширования в предыдущем разделе. Ключевым свойством, которое будет фигурировать в теореме 8.7, является ограниченная вероятность возникновения таких циклов, поэтому стоимость повторного хеширования $O(n)$ можно амортизировать путем взимания времени $O(1)$ за вставку, как это было доказано в следствии 8.3.

Алгоритм 8.3 демонстрирует псевдокод операции вставки. Возможны более компактные варианты, но я в учебных целях выбрал самый наглядный. На шагах 1–6

---

[1] Формальное, но краткое определение k-независимого хеширования можно найти в https://en.wikipedia.org/wiki/K-independent_hashing. На самом деле введенное в разделе 8.3 свойство универсальности можно иначе назвать *парной независимостью*, поскольку оно справедливо для пар ключей, таким образом, $k = 2$.

проверяется, пуста ли хотя бы одна из двух записей таблицы-кандидата для хранения k. В случае положительного результата проверки ключ сохраняется туда. В противном случае запускается цикл `while` с кодом каскадного вытеснения. Цикл останавливается либо после обнаружения пустой записи таблицы (шаги 14–16), либо после m шагов (если мы оказались в бесконечном цикле). Хитрость нашего кода заключается в шаге 10. Именно здесь гарантируется, что вытесненный ключ перемещается в запись таблицы, на которую указывает $h_s(k) \to h_d(k)$ и которую мы будем моделировать как направленное ребро соответствующим образом построенного графа (рис. 8.4).

Алгоритм 8.3. Вставка ключа $k \notin S$ в словарь, индексированный с помощью кукушкиного хеширования по таблице $T[0, m-1]$

1: **if** $T[h_1(k)]$ = `NULL` **then**
2: $T[h_1(k)] = k$; **return**;
3: **end if**
4: **if** $T[h_2(k)]$ = `NULL` **then**
5: $T[h_2(k)] = k$; **return**;
6: **end if**
7: $count = 0$;
8: $s = 2$; *// для вставленного ключа k используем h_1 из шага 10*
9: **while** $count < m$ **do**
10: $d = \{1, 2\} - \{s\}$; *// s — индекс (исходной) хеш-функции для k в T*
11: *// d — индекс (целевой) хеш-функции для k в T*
12: $pos = h_d(k)$; *// позиция в T, где должен оказаться k*
13: меняем k на $T[pos]$; *// k — новый ключ, подлежащий перемещению*
14: **if** k = `NULL` **then**
15: **return**;
16: **end if**
17: пусть s равно $h_s(k) = pos$; *// s — исходный индекс хеш-функции для k в T*
18: $count = count + 1$;
19: **end while**
20: Повторно хешируем ключи в T двумя новыми хеш-функциями, возможно, удвоив m;
21: Повторяем вставку k из шага 7.

Для анализа этой ситуации пригодится граф, узлы которого представляют записи таблицы T, а ребра связаны с ключами словаря и соединяют две записи таблицы, куда универсальные хеш-функции h_1 и h_2 могут сохранить ключ. Ребра ориентированы

с учетом места исходного хранения ключа и места его возможного нового сохранения. Такой кукушкин граф для таблицы из 10 записей и 6 ключей, а именно {A, B, C, D, E, F}, демонстрирует рис. 8.4.

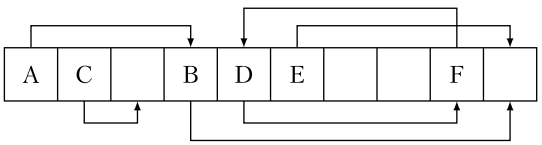


Рис. 8.4. Представление кукушкиного хеширования в виде графа

Ориентированные ребра дают простой способ идентифицировать последовательность ячеек таблицы (то есть путь из узлов), пройденных каскадом вытеснений, вызванных ключом k и начинающихся либо с записи (узла) $T[h_1(k)]$, либо с $T[h_2(k)]$. Назовем этот путь *корзиной* для k. Он напоминает список, который был связан с ячейкой $T[h(k)]$ в хешировании с цепочками (см. раздел 8.2), но в данном случае возможна куда более сложная структура, ведь кукушкин граф допускает циклы. Соответственно, путь может образовывать петли, как в случае с ключами D и F.

Рассмотрим вставку в граф с рис. 8.4 ключа G, для которого предполагается h_1(G) = 3 и h_2(G) = 0 (записи таблицы отсчитываются от 0). В этом случае G вытесняет либо A, либо B, как показано на рис. 8.5. Мы помещаем ключ G в $T[0]$, тем самым вытесняя ключ A, который в соответствии с направленным ребром пытается сохраниться в $T[3]$ = B. Затем A вытесняет B, который в соответствии с его направленным ребром уходит на последнее место таблицы. Поскольку здесь находится значение NULL, B сохраняется в эту ячейку и процедура вставки успешно завершается.

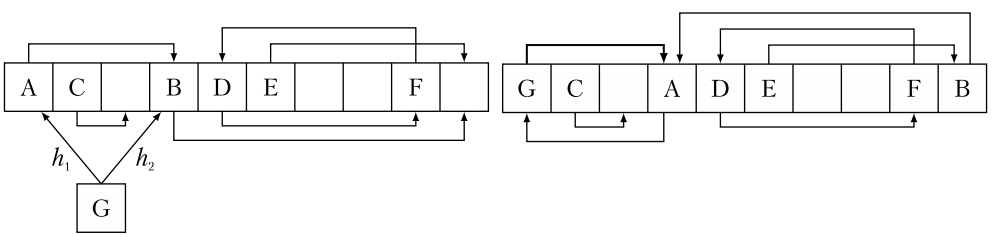


Рис. 8.5. Вставка ключа G: *слева* — два варианта входа; *справа* — окончательная конфигурация таблицы. Обратите внимание на то, что ребра, связанные с перемещенными ключами A и B, выделены пунктиром. Их также поменяли местами, чтобы отразить окончательное положение

Теперь рассмотрим более сложный случай вставки, включающий ключ Z, для которого мы изучим два возможных отображения хешей, h_1(Z) и h_2(Z). На рис. 8.6 представлены две ситуации, иллюстрирующие связь между циклом в ориентированном кукушкином графе и невозможностью завершения операции вставки.

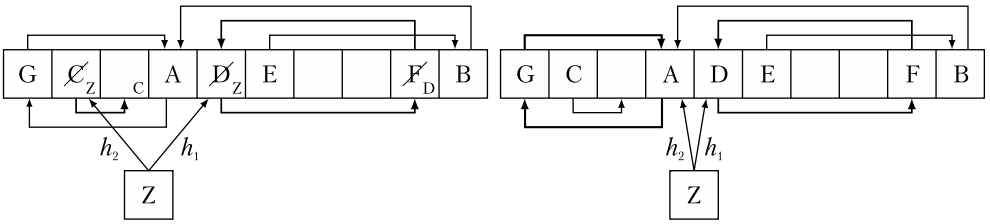


Рис. 8.6. Вставка ключа Z: *слева* — успешная; *справа* — неудачная. Жирным шрифтом выделены ребра, рассматриваемые процедурой вставки

Слева на рис. 8.6 ключ `Z` отображается в ячейку $T[1]$ или $T[4]$. Там уже находятся ключи `C` и `D` соответственно. Так как при вставке предпочтение отдается вытеснению ключа $T[h_1(\texttt{Z})] = \texttt{D}$, ключ `Z` замещает `D`, а тот, в свою очередь, перемещается, чтобы вытолкнуть `F` (в соответствии с его направленным ребром), который снова перемещается для замещения `Z`. Алгоритм находит цикл (на рисунке он показан жирными ребрами), который тем не менее не становится бесконечным, поскольку вытеснение ключа `Z` заставляет алгоритм проверить второе возможное местоположение, а именно $T[h_2(\texttt{Z})] = T[1] = \texttt{C}$. Отсюда ключ перемещается в свободную ячейку таблицы $T[2]$. Вставка завершается успешно даже при наличии одного цикла.

Справа на рис. 8.6 второе отображение $h_2(\texttt{Z})$ происходит уже не в ячейку 1, а в ячейку 3. Такое изменение обрекает процедуру вставки на неудачу, потому что обе ячейки, $T[3]$ и $T[4]$, являются частью двух направленных циклов (выделены на рисунке жирными линиями). Первый включает записи {0, 3}, а второй — {4, 8}. В результате алгоритм входит в бесконечный цикл вытеснений. Чтобы избежать подобной ситуации, нужно проверять, не заканчивается ли обход кукушкиного графа таким бесконечным циклом. Это легко реализуется, причем это можно сделать эффективно с точки зрения использования памяти, ограничив количество шагов вытеснения размером таблицы. В такой ситуации, как указано в шаге 20 алгоритма 8.3, таблица перестраивается с нуля, задействуя две новые хеш-функции и больший/меньший размер в зависимости от текущего коэффициента загрузки.

Проанализируем временную сложность операции вставки, которая неизбежно зависит от некоторых свойств кукушкиного графа. Предположим, граф состоит из m узлов, по одному на запись таблицы, и n ребер, по одному на ключ словаря. Обратите внимание на то, что роли этих двух параметров отличаются от принятых в классической нотации, используемой в литературе по графам, где, как правило, n — это количество узлов, а m — количество ребер. Но в данном случае мы работаем со *случайным* графом, поскольку конечные точки его ребер определяются в соответствии с двумя универсальными хеш-функциями h_1 и h_2. Чтобы легко ограничить вероятности событий, рассмотрим *неориентированную* версию графа. В ней ребра не направлены, соответственно, два узла, то есть записи таблицы, участвуют в операции вставки, если в графе есть соединяющий их неориентированный путь. Ключевое свойство, от которого зависит эффективность операции вставки, формулирует и доказывает теорема 8.7.

Теорема 8.7. *При $m \geqslant 2cn$ для любой пары узлов i и j и любой константы $c > 1$ вероятность наличия в неориентированном кукушкином графе, состоящем из m узлов и n ребер, кратчайшего пути из i в j длиной $L \geqslant 1$ не превышает c^{-L}/m.*

Доказательство. Проведем индукцию по длине пути $L \geqslant 1$. Базовый случай $L = 1$ означает наличие ненаправленного ребра между узлами i и j. Такое ребро может сгенерировать любой ключ с вероятностью не более $2/m^2$, поскольку ребра ненаправленные, а универсальные хеш-функции $h_1(k)$ и $h_2(k)$ обеспечивают равномерный случайный выбор среди m записей таблицы. Суммируя по всем n ключам словаря и вспоминая, что $m \geqslant 2cn$, получаем границу $\sum_{k \in S} 2/m^2 = 2n/m^2 \leqslant c^{-1}/m$.

Для индуктивного шага нужно ограничить вероятность существования кратчайшего пути длиной $L > 1$. Ни один кратчайший путь длиной меньше L не соединяет i с j и наоборот. Это произойдет только в случае выполнения для некоторой записи таблицы z следующих условий: во-первых, должен существовать кратчайший путь длиной $L - 1$ от i до z (который, очевидно, не проходит через j), во-вторых, должно существовать ребро из z в j.

Согласно индуктивной гипотезе вероятность соблюдения первого условия ограничена величиной $c^{-(L-1)}/m = c^{-L+1}/m$. Вероятность соблюдения второго условия мы уже вычисляли для базового случая, и она ограничена c^{-1}/m. Таким образом, вероятность существования кратчайшего пути, проходящего через z, не превышает $(1/cm)(c^{-L+1}/m) = c^{-L}/m^2$. Суммируя по всем m записям таблицы z, получаем для кратчайшего пути длиной L между i и j вероятность, не превышающую c^{-L}/m. ■

Другими словами, эта теорема утверждает, что если число m узлов в кукушкином графе достаточно велико по сравнению с числом ребер n, то есть $m \geqslant 2cn$, то вероятность того, что любые два узла i и j соединены длинным путем и, следовательно, могут участвовать в длинном каскаде вытеснений, низка. Более того, с ростом L эта вероятность экспоненциально убывает. Показателен случай с фиксированной длиной пути $L = \Theta(1)$, вероятность появления которой составляет $O(1/m)$. Это означает, что даже для этого ограниченного случая вероятность непостоянного числа выталкиваний мала. Ее можно связать с *вероятностью коллизии* при хешировании с цепочками. Для этого используется введенное в разделе 8.3 понятие корзины, куда помещаются все ключи словаря, хранящиеся в записях таблицы, соединенных путем обхода ненаправленного кукушкиного графа.

Большое количество операций вставки приводит к тому, что в конечном итоге таблица T заполняется и дальнейшая вставка становится невозможной. Более того, она может оказаться невозможной задолго до заполнения таблицы T, ведь теорема 8.7 справедлива только для коэффициента загрузки кукушкиной хеш-таблицы $\alpha = n/m \leqslant 1/2c < 50\ \%$. Недавно эту границу смогли улучшить, сделав кукушкино хеширование жизнеспособной, практичной и эффективной альтернативой другим схемам хеширования. Поэтому в завершение этого раздела я сосредоточусь на стоимости повторного хеширования ключей, хранящихся в таблице T, и на том, как этот процесс влияет на временную сложность каждой отдельной операции вставки.

Рассмотрим последовательность из εn вставок ключей, где ε — малая положительная константа. Предположим, что размер таблицы m достаточно велик для того, чтобы после этой последовательности вставок удовлетворять условиям из теоремы 8.7, а именно $m \geqslant 2cn + 2c(\varepsilon n) = 2cn(1 + \varepsilon)$. Переформатирование таблицы T происходит только в случае, когда вставка ключа индуцирует в кукушкином графе бесконечный цикл, то есть на шаге 20 алгоритма 8.3. Чтобы ограничить эту вероятность, рассмотрим граф после вставки всех εn ключей, состоящий из m узлов и $n(1 + \varepsilon)$ ребер. Наличие в этом графе замкнутого контура, безусловно, является фактором, способствующим появлению бесконечного цикла при вставке этих ключей, поэтому вероятность появления такого контура дает простую, но существенную верхнюю границу для вероятности неудачной вставки. Эту вероятность легко оценить в два шага. Во-первых, нужно посчитать вероятность того, что узел в кукушкином графе принадлежит циклу произвольной длины: согласно теореме 8.7, она не превышает суммы $\sum_{L=1}^{\infty} c^{-L}/m = \frac{m^{-1}}{c-1}$, ведь мы предположили, что $m \geqslant 2cn(1 + \varepsilon)$. Во-вторых, обратите внимание на то, что вероятность цикла в кукушкином графе можно ограничить суммированием предыдущей границы по всем m записям таблицы, а именно $\frac{m}{m(c-1)} = \frac{1}{c-1}$.

Следствие 8.3. *При $c = 3$ вставка εn ключей в словарь размером n, реализованная с помощью кукушкиного хеширования над таблицей размером $m \geqslant 6n(1 + \varepsilon)$, занимает амортизированное ожидаемое время $\Theta(n)$.*

Доказательство. Для $c = 3$ вероятность существования цикла в кукушкином графе из $m \geqslant 6n(1 + \varepsilon)$ узлов и $n(1 + \varepsilon)$ ключей не превышает $1/2$. Поэтому фиксированного числа перехеширований достаточно для того, чтобы гарантировать успешную вставку εn ключей в нашу таблицу. Так как ожидаемое время вставки составляет $O(1)$, при фиксированном ожидаемом размере корзины можно вывести, что одно повторное хеширование $n(1 + \varepsilon)$ ключей занимает ожидаемое время $O(n)$. Следовательно, можно заключить, что для успешной вставки εn ключей потребуется ожидаемое амортизированное время $O(n)$. ■

Чтобы сделать этот алгоритм рабочим для всех n и ε, можно воспользоваться идеей, которую я описывал в начале этой главы для хеширования с цепочками, сразу после следствия 8.1. Она называется *методом глобальной перестройки*. Всякий раз, когда размер словаря становится слишком маленьким по сравнению с размером хеш-таблицы, создается новая, меньшая хеш-таблица и, наоборот, всякий раз, когда хеш-таблица слишком заполняется, создается новая хеш-таблица большего размера. Для большей эффективности размер умножается или делится на постоянный коэффициент (больше 1), например удваивается или уменьшается вдвое. Стоимость повторного хеширования можно еще сильнее снизить за счет небольшого фиксированного дополнительного пространства, называемого *временным хранилищем* (stash). Как только во время вставки ключа k возникает сбой, то есть k вызывает бесконечный цикл, этот ключ помещается во временное хранилище без повторного хеширования. Это снижает вероятность перехеширования до $\Theta(1/n^{s+1})$, где s — размер временного

хранилища. Выбор параметра s связан с некоторыми структурными свойствами кукушкиного графа и универсальных хеш-функций, которые слишком сложны, чтобы обсуждать их здесь (подробности см. в [1] и далее по ссылкам).

8.6. Более подробно о статическом и идеальном хешировании

Напомню, что минимальная упорядоченная идеальная хеш-функция — это хеш-функция без коллизий между ключами словаря, отображаемая в область значений, размер которой совпадает с размером словаря ($m = n$), и сохраняющая у значений тот же самый порядок, что и у ключей, то есть $\forall k' < k'' \in S, h(k') < h(k'')$. Минимальность в данном случае подразумевает, что для $k \in S$ целое число $h(k)$ будет соответствовать рангу k в упорядоченном словаре S.

Проектирование функции h строго зависит от словаря S. Редактирование S может привести к потере или идеальности, или упорядоченности, или обоих свойств. Поэтому в динамическом сценарии поддерживать эти свойства трудно. Более того, даже при упорядоченном пространстве ключей U сохранение порядка гарантируется только для ключей. Выбрав ключ $\hat{k} \notin S$, мы ничего не сможем сказать о значении $h(\hat{k})$ и положении $\hat{k}$ в упорядоченном словаре S. Можно только заключить, что это значение в диапазоне $\{0, 1... m - 1\}$. Тем не менее для ключей словаря с помощью h можно реализовать таблицу прямого доступа, не сталкиваясь с ограничениями по занимаемому пространству, которые упоминались в разделе 8.1. Достаточно выделить место под таблицу размером n и сохранить ключ k и его сопутствующую информацию в $T[h(k)]$. Ключи будут храниться в отдельных ячейках таблицы (благодаря идеальности h) и будут упорядоченными (благодаря упорядоченности h), не требуя дополнительного пространства (благодаря минимальности h). Интересная особенность хеш-функций такого типа заключается в том, что значение $h(k)$ можно использовать в качестве *лексикографического (целочисленного) имени* для ключа k. Это позволяет менять лексикографические сравнения между парами ключей на сравнения «меньше, чем» между значениями хеш-функции, то есть применять сравнения целых чисел вместо неэффективных сравнений между, возможно, длинными ключами.

Кроме того, если ключи словаря закодированы как битовые строки переменной длины, то каждому ключу можно присвоить ранг, развернув такую структуру данных, как префиксное дерево (см. теорему 9.7). Но такой подход имеет два недостатка. Во-первых, для присвоения ранга требуется поиск по префиксному дереву, и количество операций ввода-вывода будет уже оцениваться для худшего случая не как $O(1)$. Во-вторых, занятость пространства станет линейной не по количеству элементов в словаре словаря, а по его общему размеру (включая динамически растущие дополнительные структуры данных). Это связано с тем, что предлагаемые минимальные упорядоченные совершенные хеш-функции не нуждаются в хранилище словаря S, им достаточно $\Theta(n)$ целых чисел. Но нужно учитывать, что эти хеш-функции не могут предоставить лексикографический ранг ключей, *не* входящих в словарь S, и, таким образом, они ограничены только так называемыми *поисковыми* запросами. Верно и обратное: если требуется поддержка еще и *лексикографических*

(поддерживающих сравнение) запросов для *не* входящих в словарь S ключей, вы не сможете обойтись без префиксных деревьев, требующих больших временных и пространственных затрат (см. главу 7).

Основу конструкции h составляют три вспомогательные функции, h_1, h_2 и g. Первые две — это универсальные хеш-функции с собственной целочисленной областью значений, большей чем n, а функция g определяется из первых двух путем решения набора из n уравнений по модулю с двумя переменными каждое, что, в свою очередь, сводится к решению интересной *задачи маркировки* для неориентированного, ациклического случайного графа. Напомню, что случайные графы используются для решения самых разных задач. Ранее я применял такой граф для проектирования и анализа кукушкиного хеширования, теперь же он поможет при проектировании и анализе минимальных и упорядоченных идеальных хеш-функций.

Дадим формальное определение трех вспомогательных функций, h_1, h_2 и g.

- Две универсальные хеш-функции, h_1 и h_2, отображают ключи из пространства U в целые числа набора $\{0, 1... m' - 1\}$. Здесь m' больше размера словаря $n = |S|$. Стоит отметить, что h_1 и h_2 не являются минимальными (на самом деле $m' > n$) и, кроме того, могут быть неидеальными, то есть вызывать коллизии между ключами словаря S. После выбора h_1 и h_2 из универсального класса хеш-функций алгоритм строит неориентированный граф G из n ребер и m' узлов[1] и проверяет, является ли он циклическим. В случае положительного результата проверки происходит повторный выбор h_1 и h_2. Если же G оказывается ациклическим, алгоритм переходит к построению функции g. От выбора m' зависит ожидаемое количество перевыборов и, следовательно, эффективность процесса построения. Для константы $c \geqslant 3$ обычно берется $m' = cn^2$.
- Функция g отображает целые числа из диапазона $\{0... m' - 1\}$ в целые числа в диапазоне $\{0... n - 1\}$. Так как $m' > n$, некоторые выходные значения g могут повторяться, и g определенно не идеальна.. Тем не менее функция g спроектирована таким образом, чтобы объединить h_1 с h_2 и получить нужное нам h. Функция g создается с помощью элегантного алгоритма, который назначает узлам графа G (а их у нас m') надлежащие целочисленные метки, которые соответствуют значениям из диапазона этой функции. Для ациклического графа G такая возможность всегда существует и может быть реализована за время, линейное по размеру G, путем обхода его узлов (см. далее в этом разделе).
- Компоновка функций h_1, h_2 и g, задающая функцию $h(k)$, происходит следующим образом: $[g(h_1(k)) + g(h_2(k))] \bmod n$ для каждого $k \in S$. Очевидно, что h будет минимальной, поскольку она возвращает значения в диапазоне $\{0... n - 1\}$. Осталось доказать, что она вдобавок идеальна и упорядоченна.

---

[1] Здесь, как и в случае с кукушкиным графом, поменялись местами роли букв n и m', обозначающих количество ребер и узлов соответственно.

[2] Выбор $c \geqslant 3$ дает ациклический граф G в $\left\lceil \sqrt{\frac{m'}{m' - 2n}} \right\rceil$ случаях. При $c = 3$ это около двух попыток (см. [11]).

Теперь давайте оценим временную и пространственную сложность h. Функция g кодируется целочисленным массивом из m' записей, тогда как универсальные хеш-функции h_1 и h_2 занимают фиксированное пространство (см. раздел 8.3). Учитывая, что m' оценивается как $\Theta(n)$, общее требуемое пространство будет $\Theta(n)$, то есть линейным по кардинальности словаря, а не его по длине. Оценка $h(t)$ дает постоянное время, ведь мы вычисляем две хеш-функции, h_1 и h_2, два раза обращаемся к массиву g и, наконец, выполняем два суммирования и одну операцию по модулю. Левая половина рис. 8.7 демонстрирует словарь из девяти строк, у которого m' равно 13.

Осталось определить детали алгоритма, который вычисляет функцию g, корректно определяя $h(k)$ в соответствии с приведенной ранее формулой. Формула фактически обеспечивает n ограничивающих равенств по n ключам словаря $k \in S$. Для каждого такого ограничения известны $h(k)$, $h_1(k)$ и $h_2(k)$ (после того как были выбраны две универсальные хеш-функции), поэтому неизвестными переменными остаются два вхождения $g()$. Возьмем в качестве примера рис. 8.7:

- первый ключ, abacus, задает уравнение $0 = [g(1) + g(6)] \bmod 9$;
- второй ключ, cat, задает уравнение $1 = [g(7) + g(2)] \bmod 9$;
- и так далее до самого последнего ключа, zoo, который задает уравнение $8 = [g(5) + g(3)] \bmod 9$.

а)	Ключ k	$h(k)$	$h_1(k)$	$h_2(k)$	б)	x	$g(x)$
	abacus	0	1	6		0	0
	cat	1	7	2		1	5
	dog	2	5	7		2	0
	flop	3	4	6		3	7
	home	4	1	10		4	8
	house	5	0	1		5	1
	son	6	8	11		6	4
	trip	7	11	9		7	1
	zoo	8	5	3		8	0
						9	1
						10	8
						11	6
						12	0

Рис. 8.7. Пример идеальной, минимальной и упорядоченной хеш-функций для словаря S из $n = 9$ ключей (в виде строк переменной длины), перечисленных в алфавитном порядке. Столбец $h(k)$ содержит лексикографический ранг каждого ключа в S. Хеш-функция h получена из трех функций. Это две случайные хеш-функции $h_1(k)$ и $h_2(k)$ для $k \in S$ с областью значений $\{0, 1, ..., m' - 1\}$, где $m' = 13 > 9 = n$. А также корректно выведенная функция $g(x)$, для $x \in \{0, 1, ..., m' - 1\}$, реализованная в виде массива. Ее конструкция объясняется в тексте

Легко заметить, что в этих уравнениях могут быть несколько вхождений одних и тех же значений: например, $g(1)$ встречается в уравнениях abacus, home и house.

Поэтому не совсем понятно, допускает ли идеальное решение такой набор уравнений. Но g вычисляется на удивление просто. Этот процесс сводится к построению неориентированного графа $G = (V, E)$ с m' помеченными узлами $\{0, 1, ..., m' - 1\}$ (здесь тот же диапазон, что и для областей значений h_1, h_2 и g) и n ребрами (по количеству ключей в словаре S). На один ключ приходится одно неориентированное ребро, то есть мы имеем одно неориентированное ребро на уравнение, а именно ребро $(h_1(k), h_2(k))$, для каждого ключа $k \in S$. Каждое такое ребро помечается *желаемым* значением $h(k)$. Из рис. 8.8 и приведенных здесь уравнений получим:

- первый ключ, abacus, задает уравнение $0 = [g(1) + g(6)] \bmod 9$, которое создает ненаправленное ребро (1, 6) и метку 0 для него;
- второй ключ, cat, задает уравнение $1 = [g(7) + g(2)] \bmod 9$, которое создает ненаправленное ребро (2, 7) и метку 1 для него;
- и так далее до последнего ключа, zoo, задающего уравнение $8 = [g(5) + g(3)] \bmod 9$, которое создает ненаправленное ребро (3, 5) и метку 8 для него.

Очевидно, что топология графа G зависит только от h_1 и h_2, то есть это *случайная топология* на базе двух случайных хеш-функций. Остается решить этот набор уравнений. Здесь возникает элегантная, но простая идея, которая возможна благодаря отсутствию циклов в графе G. Напомню, что, если граф оказывается циклическим, функции h_1 и h_2 перевыбираются до тех пор, пока он не превратится в ациклический.

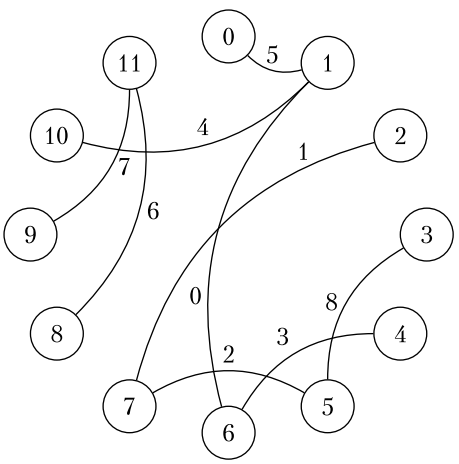


Рис. 8.8. Граф, соответствующий словарю с рис. 8.7. Узел 12 не показан, потому что это значение не встречается в области значений хеш-функций h_1 и h_2

Идея алгоритма заключается в том, что берется любой узел, скажем узел 0 с рис. 8.8, и с ним сопоставляется произвольное значение, например 0. Это соответствует равенству $g(0) = 0$. Затем берется связанное с этим узлом ребро, скажем (0, 1), имеющее метку 5. Оно соответствует шестому уравнению, связанному с ключом

house: $5 = [g(0) + g(1)] \bmod 9$. Поскольку $g(0) = 0$, легко определить, что $g(1) = 5$. Затем процесс повторяется: берется связанное с новым узлом ребро, например (1, 10) с меткой 4. Оно соответствует пятому уравнению, связанному с ключом home: $4 = [g(1) + g(10)] \bmod 9$. Так как ранее стало известно, что $g(1) = 5$, можно легко найти $g(10) = 8$. Постепенно доходит до ребра (u, v) с меткой $R(u, v)$. Для v определено значение $g(v)$, а для $g(u)$ — еще нет. Для $g(u) \geqslant 0$ получаем $g(u) = [R(u, v) - g(v)] \bmod 9$. Если $g(u)$ не было определено ранее, такое присваивание допустимо. В ациклическом графе это условие всегда соблюдается. При наличии ребра, связанного с двумя непомеченными узлами, мы сопоставляем с одним из этих узлов произвольное значение (скажем, значение 0 для простоты) и продолжаем действовать по описанной схеме.

Псевдокод для этих схем представлен в алгоритмах 8.4 и 8.5. Первый алгоритм запускает маркировку узлов графа, достижимых из узла v, который исходно не был помечен и получил начальное значение 0. Если при обходе графа алгоритм встречает уже посещенный узел, построение g останавливается, так как это указывает на наличие цикла. В остальных случаях переменной u присваивается значение, определенное путем разрешения соответствующего уравнения в терминах $g(u)$. Массив g удовлетворяет всем необходимым требованиям, а временная сложность его построения соответствует всего одному обходу G. Таким образом, мы доказали теорему 8.8.

Алгоритм 8.4. Процедура LabelGraph(G)

```
1: for v ∈ V do
2:     g[v] = undef
3: end for
4: for v ∈ V do
5:     if g[v] = undef then
6:         Label(v, 0)
7:     end if
8: end for
```

Алгоритм 8.5. Процедура Label(v, c)

```
1: if g[v] ≠ undef and g[v] ≠ c then
2:     return циклический граф; ОСТАНОВКА
3: end if
4: g[v] = c
5: for u ∈ Adj[v] do
6:     Label(u, R(u, v) – g(v) mod n)
7: end for
```

Теорема 8.8. *Минимальная упорядоченная идеальная хеш-функция для словаря с n ключами может быть построена за ожидаемое время $O(n)$ на таком же пространстве. Вычисление хеш-функции в худшем случае занимает время $O(1)$ и использует пространство $O(n)$.*

Вы можете самостоятельно убедиться в том, что, применив алгоритм 8.4 к графику с рис. 8.8, для массива g можно получить значения с рис. 8.7, *б*.

8.7. Фильтры Блума

Количество элементов и общий объем пространства ключей бывают очень большими. В этом случае после хеширования приходится сталкиваться с недостатком места для хранения не только таблицы и ее указателей, которые занимают $(n + m)$ $\log n$ бит (см. следствие 8.1), но и ключей, которые занимают $n \log_2 u$ бит. Например, представим словарь URL-адресов, с которым работают поисковые роботы. Длина URL-адреса может составлять сотни символов, поэтому при попытке хранить адреса целиком для индексируемого словаря довольно быстро перестанет хватать места во внутренней памяти [4]. Вместо кукушкиного хеширования или хеширования с цепочками поисковые роботы используют простую и рандомизированную, но эффективную структуру данных, называемую *фильтром Блума* [2].

Важнейшее свойство этой структуры заключается в том, что она не хранит ключи *в явном виде*. Хранится только небольшой *отпечаток* от них, в результате чего необходимое фильтрам Блума пространство зависит от количества ключей, а не от их общей длины. Это большой плюс, но, к сожалению, есть и минус — *ложноположительные срабатывания* при запросе членства. К запросам, включающим входящие в словарь ключи, этот минус не имеет отношения. В данном случае фильтр Блума всегда срабатывает правильно и ложноотрицательных ответов не бывает. Вот почему говорят, что фильтры Блума допускают *односторонние ошибки*. При этом ложноположительные ошибки можно контролировать, потому что вероятность их возникновения *уменьшается экспоненциально* по мере увеличения размера отпечатков ключей. С практической точки зрения десятков битов (нескольких байтов) на отпечаток достаточно, чтобы гарантировать крошечные вероятности ошибок[1] и сжатое заполнение пространства, что делает такое решение очень привлекательным в контексте больших данных. Полезно запомнить формулу: «Везде, где используется список или набор и имеет значение занимаемое пространство, следует подумать о фильтре Блума. При использовании фильтра Блума учитывайте потенциальные ложные срабатывания».

---

[1] Можно возразить, что ошибки все равно будут. Но программисты в этом случае отвечают, что таких ошибок намного меньше, чем возникающих из-за сбоев оборудования/сети в центрах обработки данных или на ПК, соответственно, их можно игнорировать.

Пусть $S = \{x_1, x_2 \ldots x_n\}$ — набор из n ключей, а B — битовый вектор длиной m. Изначально все биты этого вектора имеют значение 0. Предположим, у нас есть r универсальных хеш-функций $h_i : U \to \{0 \ldots m-1\}$ для $i = 1 \ldots r$. Также предполагается, что каждый ключ словаря k явно в B не представлен, а идентифицируется путем отображения r бит вектора B в 1 следующим образом: $B[h_i(k)] = 1, \forall\ 1 \leqslant i \leqslant r$. Следовательно, вставка ключа в фильтр Блума требует времени $O(r)$ и устанавливает не более r бит (некоторые хеши при этом могут конфликтовать, это называется *стандартным* фильтром Блума). Мы утверждаем, что ключ y принадлежит словарю S тогда и только тогда, когда все биты его отпечатка установлены в единицу, то есть $B[h_i(y)] = 1, \forall\ 1 \leqslant i \leqslant r$. На поиск, как и на вставку, затрачивается время $O(r)$. Операция удаления не поддерживается. В примере, приведенном на рис. 8.9, мы видим, что $y \notin S$, поскольку три бита установлены в 1, но самый правый бит равен нулю (четыре проверенных бита $B[h_i(y)]$ — это те, на которые указывают четыре стрелки).

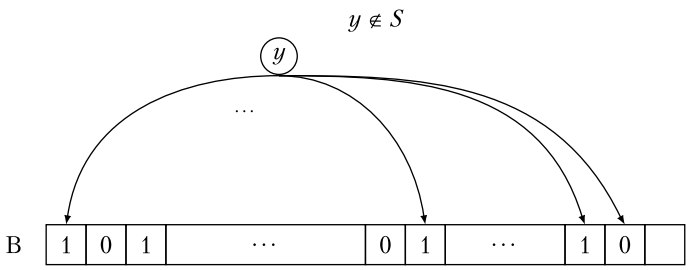


Рис. 8.9. Поиск ключа y в фильтре Блума. Четыре стрелки указывают на четыре проверенных элемента $B[h_i(y)]$, ведь в рассматриваемом случае $r = 4$ и, таким образом, $i = 1, 2, 3, 4$. Поскольку один проверенный элемент равен 0, ключ $y \notin S$

Очевидно, если $y \in S$, фильтр Блума определяет это правильно. Но бывает, что при $y \notin S$ все проверенные r бит установлены в 1. Так получается оттого, что на каком-то ключе не из набора (возможно, на нескольких) какие-то другие r хеш-функций выдали значения от 1 до r. Это тот самый ложноположительный результат, о котором упоминалось ранее. На запрос членства фильтр Блума возвращает положительный, но неверный ответ. Понятно, что возникает вопрос о вероятности такой ошибки, которая, как легко доказать, ограничена сверху удивительно простой формулой.

Вероятность того, что после вставки ключа $k \in S$ запись $B[j]$ осталась пустой, совпадает с вероятностью того, что r хеш-функций $h_i(k)$ вернут запись, отличную от j. Она равна $\left(\frac{m-1}{m}\right)^r$. Для достаточно больших m это значение можно аппроксимировать формулой $e^{-r/m}$. После вставки всех n ключей вероятность того, что запись $B[j]$ все еще пуста, можно оценить как $p_0 \approx \left(e^{-r/m}\right)^n = e^{-rn/m}$. При этом мы

исходим из предположения о независимости этих хеш-функций[1]. Следовательно, вероятность ложноположительного результата — это вероятность того, что все r бит, проверенных для не входящего в текущий словарь ключа, установлены в 1, то есть $p_{\text{err}} = \left(1 - p_0\right)^r \approx \left(1 - e^{-m/m}\right)^r$.

Неудивительно, что вероятность ошибки зависит от трех параметров, которые определяют структуру фильтра Блума: количества хеш-функций r, количества ключей словаря n и количества битов m в двоичном массиве B. Как и ожидалось, в этой формуле отсутствует общий размер словаря S, который также учитывает длину ключей. Интересно отметить, что дробь $f = m/n \geqslant 1$ можно рассматривать как выделенное в массиве B среднее количество битов на ключ словаря, отсюда и то, что мы назвали его размером *отпечатка* f. Чем больше f, тем меньше вероятность ошибки p_{err}, но тем больше необходимое для B пространство. Оптимизировать p_{err} для заданного размера отпечатка f можно, взяв от него производную первого порядка и приравняв ее к нулю, что дает $r_{\text{opt}} = (m/n) \ln 2 = f \ln 2$. Для этого r вероятность того, что какой-то бит в массиве B получит нулевое значение, составляет $p_0 = 1/2$. Фактически это означает, что массив заполнен наполовину единицами, а наполовину нулями. Другого результата здесь быть не может: чем больше r, тем больше единиц будет в B и, таким образом, больше вероятность p_{err}, а чем меньше r, тем больше в B нулей и, таким образом, больше пространства расходуется впустую. Оптимальный выбор r_{opt} попадает в середину. Для оптимального значения мы имеем $p_{\text{err}} = \left(1/2\right)^{r_{\text{opt}}} = \left(0{,}693\right)^{m/n}$, и эта вероятность экспоненциально уменьшается по мере роста размера отпечатка $f = m/n$.

Довольно хорошо, что для сильного уменьшения количества ложноположительных результатов нужны небольшие значения f: $p_{\text{err}} = 8 \times 10^{-6}$ для отпечатка из 32 бит (то есть $m = 32n$) и $p_{\text{err}} = 6{,}4 \times 10^{-11}$ для отпечатка из 64 бит (то есть $m = 64n$). Количество ложноположительных результатов как функцию количества хешей для фильтра Блума, разработанного под пространство $m = 32n$ бита, и, следовательно, отпечатка $f = 32$ бита на один ключ демонстрирует рис. 8.10. Здесь стоит отметить, что при количестве хеш-функций $r = 23$ показатель ложноположительных результатов падает примерно до 10^{-7}. Кроме того, для $r \geqslant 16$ вероятность ошибок меняется не слишком сильно, но на практике эти неоптимальные варианты с $16 < r < 23$ ускоряют выполнение запросов на членство.

---

[1] Можно провести более точный анализ, но он, при всей его сложности, не меняет асимптотический результат, поэтому я буду придерживаться более простых, хотя и более грубых вычислений [3], [8]. Кроме того, существует версия, называемая секционированным (partitioned) фильтром Блума [2], в которой r бит, установленных в 1 одним ключом, считаются разными, например, из-за рассмотрения r хеш-таблиц размером m/r каждая. У двух этих типов фильтров Блума одинаковое асимптотическое поведение, хотя второй из-за большего количества единиц имеет тенденцию к большему числу ложных срабатываний.

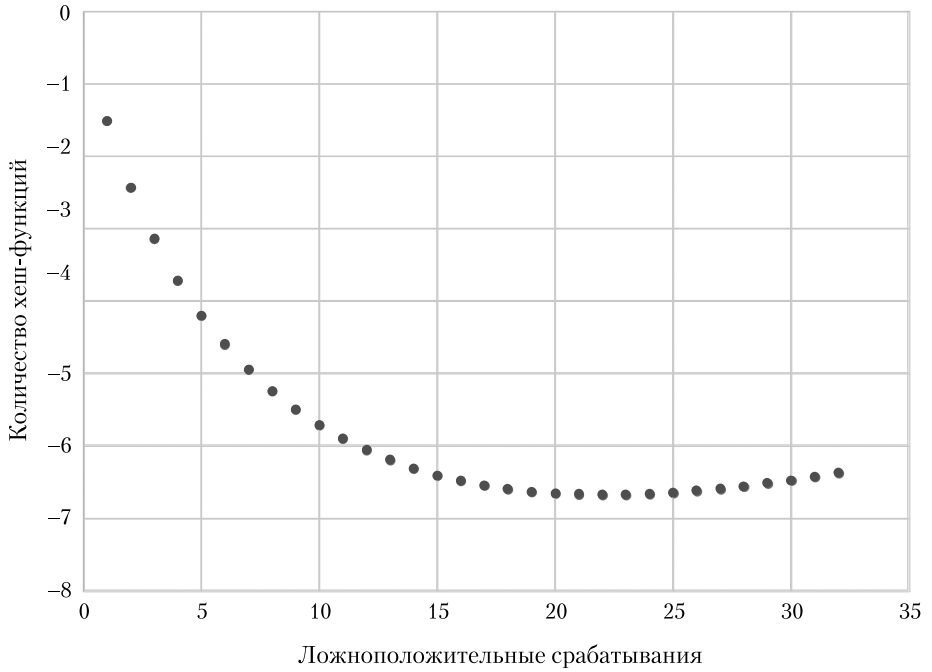


Рис. 8.10. График p_{err} (в логарифмическом масштабе) как функции количества хешей r для случая $m = 32n$

В литературе предлагается множество вариантов фильтра Блума. Два примечательных примера — фильтр Блума *со сжатием* (compressed Bloom filter) и *спектральный* фильтр Блума (spectral Bloom filter). Первый решает проблему дальнейшего сжатия занимаемого им пространства, поскольку во многих веб-приложениях для экономии полосы пропускания и времени передачи фильтр Блума должен передаваться между промежуточными серверами [9]. Второй полезен при управлении мультимножествами, так как позволяет подсчитывать кратность элементов при условии, что они ниже заданного порога *(спектра)*.

Попробуем оптимизировать частоту ложных срабатываний фильтра Блума при условии, что количество отправляемых после сжатия битов $z \leqslant m$. Для сжатия воспользуемся арифметическим кодированием (см. главу 12), которое хорошо аппроксимирует энтропию подлежащей сжатию строки. Удивительно, но больший, хотя более разреженный фильтр Блума может дать ту же частоту ложных срабатываний при меньшем количестве переданных битов. Другими словами, можно передавать столько же битов с меньшей частотой ложных срабатываний. Например, оптимальное количество хеш-функций в стандартном фильтре Блума с $m = 16n$ составляет $r_{opt} = 11$. Частота ложных срабатываний при этом равна 0,000459. В более разреженном фильтре Блума с $m = 48n$ и всего лишь $r = 3$ хеш-функциями результат сжимается до менее чем $f = 16$ бит на элемент, а частота ложных срабатываний уменьшается примерно в два

раза. Получается беспроигрышная ситуация с точки зрения частоты ложных срабатываний, занятого пространства и скорости запроса. Понятно, что сжатый фильтр Блума может потребовать больше памяти во время выполнения (из-за декомпрессии), кроме того, растут затраты на вычисления из-за дополнительных стадий сжатия/декомпрессии. Тем не менее возможны сложные подходы к индексированию сжатых данных, дающие доступ к ним без полной декомпрессии, — например, FM-индекс, обсуждаемый в главе 14, который может быть построен на базе двоичного массива B.

Спектральный фильтр Блума предназначен для отслеживания количества одинаковых ключей (с допустимой небольшой вероятностью ошибки ожидаемого значения), а также поддержки вставок и удалений с учетом этого количества. Рассмотрим его более подробно. Обозначим кратность ключа $k \in S$ как $f(k)$ — это целое число может быть числом вхождений k в поток или просто значением, связанным с k. В спектральном фильтре Блума битовый вектор B заменяется массивом C из m счетчиков, таким образом занимая пространство $\Theta(m \log u)$ бит, где u — максимальная кратность, которую может иметь ключ в S. Вставка ключа k в словарь S состоит из добавления значения $f(k)$ ко всем счетчикам, которые установлены r хеш-функциями $h_i(k)$. Процедура удаления выглядит симметрично. Запрос кратности ключа k сводится к вычислению *минимального* из значений, хранящихся в счетчиках $C[h_i(k)]$, где $i = 1, 2... r$. Минимум ищется потому, что из-за возможных конфликтов между хешами разных ключей каждый счетчик $C[i]$ дает завышенную оценку кратностей сопоставленных ему ключей. Следовательно, $f(k) \leqslant C[h_i(k)]$, где $i = 1, 2... r$. Можно легко доказать, что вероятность отличия минимума от $f(k)$ (что означает неверную оценку) равна вероятности коллизии по всем счетчикам $C[h_i(k)]$, а это, в свою очередь, равно частоте ошибок стандартного фильтра Блума, построенного на том же наборе параметров m, n, r. В литературе предлагаются варианты спектрального фильтра Блума, которые позволяют уменьшить занимаемое им пространство или частоту ошибок. Вероятно, самой известной является адаптация фильтра Блума второго уровня, хранящая ключи k, минимум которых однократно встречается среди счетчиков $C[h_i(k)]$, где $i = 1, 2... r$. Предполагается, что свойство повторяющихся минимумов обеспечивает точность оценок, в противном случае лучше использовать другой спектральный фильтр Блума — меньшего размера, который уточняет потенциально неверные ответы, данные при оценке с одним минимумом.

8.7.1. Нижняя граница занятого пространства

Интересный вопрос: насколько малым может быть битовый массив B, чтобы гарантировать заданную частоту ошибок для словаря из n ключей, взятых из пространства U размером u? Сейчас я докажу, что любая структура данных с аналогичными свойствами должна занимать не менее $n \log_2 (1/\varepsilon)$ бит. Такой результат позволяет сделать вывод, что фильтры Блума асимптотически оптимальны в пределах множителя $\log_2 e \approx 1{,}44$ от нижней границы этого пространства.

Чтобы доказать это, обозначим $F(m, \varepsilon, X)$ любую структуру данных, требующую пространство m бит и решающую запрос на членство в словаре $X \subseteq U$ с долей

ложноположительных результатов ε. Под это определение подходит фильтр Блума, который в предыдущем разделе рассматривался с $B = F(m, \varepsilon, S)$ для индексированного словаря S. Очевидно, что эта структура данных должна работать с любым возможным подмножеством $X \subset U$, и таких подмножеств $\binom{u}{n}$. Если X включает k, говорят, что структура данных $F(m, \varepsilon, X)$ принимает ключ k, в противном случае — что она этот ключ отвергает.

Любая структура данных, которая используется для представления словаря S из n элементов, должна принимать каждый из n ключей, поскольку ложноотрицательных результатов быть не должно. Кроме того, она может также принимать не более $(u - n)\varepsilon$ других ключей из пространства U, поскольку доля ложноположительных результатов равна ε. Следовательно, каждая такая структура данных принимает не более $n + \varepsilon(u - n)$ ключей и, таким образом, может использоваться для представления любого из $w = \binom{n+\varepsilon(u-n)}{n}$ подмножеств этих элементов размером n, но ее нельзя задействовать для представления любого другого множества.

Поскольку существует 2^m таких структур данных из m бит, можно заключить, что они представляют не более $2^m w$ подмножеств U, состоящих из n ключей и с долей ошибок, не превышающей ε. Учитывая, что из n ключей пространства U можно получить $\binom{u}{n}$ возможных словарей, гарантировать существование требуемой m-битной структуры данных позволяет следующее неравенство:

$$2^m \binom{n+\varepsilon(u-n)}{n} \geqslant \binom{u}{n}.$$

Разрешив его относительно m, получим:

$$m \geqslant \log_2 \left(\binom{u}{n} \Big/ \binom{n+\varepsilon(u-n)}{n} \right) \geqslant \log_2 \left(\binom{u}{n} \Big/ \binom{\varepsilon u}{n} \right) \approx \log_2 (1/\varepsilon)^n = n \log_2 (1/\varepsilon).$$

Здесь использовалась аппроксимация $\binom{a}{b} \approx \frac{a^b}{b!}$, справедливая при $a >> b$, что мы и имеем на практике для параметров u (размер пространства ключей) и n (размер словаря).

Теперь рассмотрим фильтр Блума в той же конфигурации, а именно: доля ошибок ε, размер словаря n и выделенное пространство m бит. Известно, что в оптимальной конфигурации $r_{opt} = (m/n)\ln 2$, что соответствует доле ошибок $\varepsilon = (1/2)^{(m/n)\ln 2}$. Решая это уравнение относительно m, получим:

$$m = n \frac{\log_2 (1/\varepsilon)}{\ln 2} \approx 1{,}44 n \log_2 (1/\varepsilon).$$

Это означает, что фильтры Блума асимптотически оптимальны по занимаемому пространству, а зазор относительно оптимальной границы пространства — это постоянный коэффициент 1,44.

8.7.2. Простой вариант применения

Фильтры Блума можно использовать для приблизительного пересечения двух множеств, A и B, хранящихся на двух разных машинах, M_A и M_B. Это делается путем обмена небольшим количеством битов. Такое требуется при проверке репликации данных и в распределенных поисковых системах. Вот схема эффективного решения такой задачи.

1. Машина M_A строит фильтр Блума $BF(A)$ из $m_A = \Theta(|A|)$ бит, оптимальное количество хеш-функций $r_{\text{opt}} = (m_A/|A|) \ln 2$.
2. Машина M_A отправляет $BF(A)$ машине M_B.
3. Машина M_B строит Q как подмножество элементов B, для которых $BF(A)$ дает положительный результат.

Очевидно, что $A \cap B \subseteq Q$, поэтому Q содержит ключи $|A \cap B|$ плюс количество элементов, принадлежащих только множеству B, которые, к сожалению, фильтр Блума $BF(A)$ ошибочно определил как тоже находящиеся в A. Следовательно, мы можно заключить, что $|Q| = |A \cap B| + |B|\varepsilon$, где $\varepsilon = 0{,}6185^{m_A/|A|}$, — это доля ошибок для рассматриваемой конструкции $BF(A)$. Это означает, что данные три шага определяют однораундовый протокол, который позволяет машине M_B вычислять *приблизительное значение* $A \cap B$ с долей ошибок ε. Чем больше m_A, тем меньше ε, но тем большим количеством битов придется обмениваться.

Добавив сюда еще два шага, мы получим двухраундовый протокол, который позволяет машине M_A корректно вычислять пересечение $A \cap B$.

1. Машина M_B отправляет Q обратно в M_A.
2. Машина M_A вычисляет $Q \cap A$.

На шаге 2 происходит обмен m_A битами, а на шаге 4 — $|Q| \log |U|$ битами. Соответственно, в целом двухраундовый протокол обменивает $m_A + \left(|A \cap B| + |B| \cdot 0{,}693^{m_A/|A|}\right) \log |U|$ бит. Верно и обратное: обычный протокол, отправляющий весь набор A в M_B, обменивает ровно $|A| \log |U|$ бит. Следовательно, двухраундовый протокол предпочтительнее выбирать всякий раз, когда $m_A = c|A|$, причем константа c намного меньше $\log |U|$, но не слишком мала. Это следует из связи между размером m фильтра Блума и его долей ошибок ε.

Список литературы

1. *Aumüller M., Dietzfelbinger M., Woelfel P.* Explicit and efficient hash families suffice for cuckoo hashing with a stash // Proceedings of the 20th European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 7501, Springer, 108–120, 2012.
2. *Bloom B. H.* Space/time trade-offs in hash coding with allowable errors // Communications of the ACM, 13 (7): 422–426, 1970.
3. *Bose P., Guo H., Kranakis E. et al.* On the false-positive rate of Bloom filters // Information Processing Letters, 108 (4): 210–213, 2008.
4. *Broder A. Z., Mitzenmacher M.* Survey: Network applications of Bloom filters: A survey // Internet Mathematics, 1 (4): 485–509, 2003.
5. *Azar Y., Broder A. Z., Karlin A. R., Upfal E.* Balanced allocations // SIAM Journal on Computing, 29 (1): 180–200, 1999.
6. *Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.* Introduction to Algorithms. The MIT Press, third edition, 2009.
7. *Dietzfelbinger M., Hagerup T., Katajainen J., Penttonen M.* A reliable randomized algorithm for the closest-pair problem // Journal of Algorithms, 25 (1): 19–51, 1997.
8. *Grandi F.* On the analysis of Bloom filters // Information Processing Letters, 129: 35–39, 2018.
9. *Mitzenmacher M.* Compressed Bloom filters // IEEE/ACM Transactions on Networks, 10 (5): 604–612, 2002.
10. *Pagh R., Rodler F. F.* Cuckoo hashing // Proceedings of the 9th European Symposium on Algorithms (ESA). Lecture Notes in Computer Science, 2161, 121–133, 2001.
11. *Witten I. H., Moffat A., Bell T. C.* Managing gigabytes: compressing and indexing documents and images, second edition. Morgan Kaufmann, 1999.
12. *Woelfel P.* Efficient strongly universal and optimally universal hashing // Proceedings of the 24th International Symposium on the Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, 1672, 262–272, 1999.

Глава 9
ПОИСК СТРОК ПО ПРЕФИКСУ

> Большинство открытий даже сегодня
> представляют собой сочетание счастливой
> случайности и целенаправленного поиска.
>
> *Приписывается Сиддхартхе Мукерджи*

Из-за новых приложений, базирующихся на поисковых системах в Интернете, с новой силой вспыхнул интерес к задаче поиска строк по префиксу. Вспомните про *автодополнение*, в настоящее время предлагаемое основными поисковыми системами, такими как Google и Bing, — это поиск по префиксу, на лету выполняемый по миллионам строк, соответствующих прошлым запросам пользователей, с применением шаблона запроса в качестве строки для поиска. Задачу усложняют такие моменты, как размер словаря и временные ограничения, связанные с терпением пользователей[1]. В этой главе я расскажу о множестве различных решений этой задачи с повышением проработанности и эффективности с точки зрения временной и пространственной сложности, а также сложности ввода-вывода.

Задача поиска по префиксу. Дан словарь $\mathcal{D}$, состоящий из n строк общей длиной N, составленных из алфавита размером Σ. Нужно выполнить предварительную обработку $\mathcal{D}$ с целью извлечения или просто подсчета строк с префиксом P.

Два типичных строковых запроса — это *точный* поиск и поиск *подстроки* в строках словаря $\mathcal{D}$. Для реализации первого лучше всего подходит хеширование из-за своих простоты и скорости. В главе 8 я подробно описал несколько решений. Вторая задача более сложна и требуется, например, в вычислительной геномике и в азиатских поисковых системах. Она состоит в поиске всех позиций, в которых шаблон запроса P

[1] Я упростил формулировку задачи до синтаксической формы, игнорируя моменты, касающиеся *ранжирования* ответов на основе таких характеристик, как частота возвращаемых запросов, географическое положение автора запроса и др., которые должна определять поисковая система, чтобы наилучшим образом удовлетворить основные потребности пользователя.

встречается как подстрока строк словаря. Что интересно, существует простой алгоритмической переход от поиска подстроки до поиска префикса по множеству всех *суффиксов* в строках словаря. Я расскажу о нем в главе 10, посвященной таким структурам данных, как массивы и деревья суффиксов. Фактически поиск префикса является основой других важных задач, связанных с поиском по строкам, поэтому структуры данных, представленные в этой главе, имеют варианты применения, далеко выходящие за рамки простых обсуждаемых приложений.

9.1. Массив строковых указателей

Начнем с простого распространенного решения задачи префиксного поиска — массива указателей на строки, хранящиеся в произвольных местах памяти (возможно, на диске). Пусть $A[1, n]$ — массив указателей, которые сортируются в соответствии со строками. Предположим, что каждый указатель занимает w байт памяти. Как правило, это 4 байта (32 бита) или 8 байтов (64 бита). Возможны и другие представления указателей, например, с переменной длиной. Но их обсуждение отложено до главы 11, где будут рассматриваться эффективные кодировки целых чисел.

На рис. 9.1 представлен пример словаря, строки которого хранятся в массиве S в произвольном порядке.

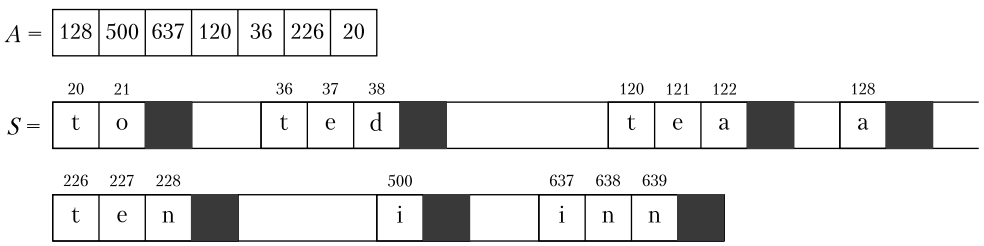


Рис. 9.1. Массив S = [*to*, *ted*, *tea*, *a*, *ten*, *i*, *inn*] несортированных строк и массив A косвенно отсортированных указателей на строки массива S

Массив указателей на отсортированные строки A удовлетворяет двум важнейшим свойствам.

- Все строки словаря с префиксом P после лексикографический сортировки становятся смежными. Поэтому их указатели занимают подмассив, скажем, $A[l, r]$, который окажется пустым, если ни у одной строки словаря не будет префикса P.
- Строка P лексикографически расположена между $A[l - 1]$ и $A[l]$.

Поскольку префиксный поиск возвращает либо количество строк словаря с префиксом P, отсюда значение $n_{occ} = r - l + 1$, либо сами эти строки, наша задача сводится

к тому, чтобы эффективно идентифицировать позиции l и r. Фактически задачу префиксного поиска мы сведем к задаче поиска лексикографической позиции нужной строки шаблона Q среди строк словаря $\mathcal{D}$. Эта задача называется *лексикографическим поиском*. Формируется Q — шаблон P или $P\#$, где # — специальный символ, который больше любого другого символа алфавита. Нетрудно убедиться, что $Q = P$ будет предшествовать строке $A[l]$, тогда как $Q = P\#$ — следовать за строкой $A[r]$. Фактически это означает, что двух лексикографических поисков шаблонов, которые короче, чем $p + 1$ символов, достаточно для решения задачи префиксного поиска.

Лексикографический поиск Q среди строк $\mathcal{D}$ может быть реализован как косвенный бинарный поиск по косвенно отсортированному массиву A. Он занимает $O(\log n)$ шагов, каждый из которых требует сравнения шаблона Q со строкой, на которую указывает запись, проверенная в A. Это лексикографическое сравнение, требующее времени $O(p)$ и $O(p/B)$ операций ввода-вывода, поскольку в худшем случае приходится сканировать все $\Theta(p)$ символов шаблона Q. Плохие временная сложность и сложность ввода-вывода вытекают из *косвенности*, которая не позволяет пользоваться локальным доступом к памяти/строкам бинарного поиска. Неэффективность этого подхода становится еще более очевидной, если попробовать получить все строки n_{occ} с префиксом P, а не просто подсчитать их количество. После идентификации подмассива $A[l, r]$ каждая сообщающая о себе строка влечет за собой по крайней мере одну операцию ввода-вывода, поскольку строки, на которые нацелены указатели в $A[l, r]$, могут не быть смежными в S.

Теорема 9.1. *Временная сложность префиксного поиска по массиву указателей строк составляет $O(p \log n)$, а количество операций ввода-вывода оценивается как $O(p/B \log n)$. Общее занимаемое пространство составляет $N + (1 + w)n$ байт. Извлечение n_{occ} строк с префиксом P требует $\Omega(n_{occ})$ операций ввода-вывода.*

Доказательство. Временная сложность и сложность ввода-вывода вытекают из сделанных ранее наблюдений. Что касается требований к пространству, то массиву A необходимы n указателей, каждый из которых занимает слово памяти w, а все строки словаря занимают N байт плюс однобайтовый разделитель для каждого из них (обычно `\0` в C). ■

При большом числе возвращаемых строк граница $\Omega(n_{occ})$ может стать основным узким местом. Именно это обычно и происходит в запросах, которые используют поиск по префиксу в качестве предварительного шага, выбирая *набор возможных ответов*, которые затем уточняются с помощью постфильтрации. Например, задача *поиска с помощью подстановочных знаков* подразумевает наличие в префиксе P наибольшего из возможных количества специальных символов *. Этот символ соответствует любой подстроке. Если $P = \alpha * \beta *...$, где α и $\beta...$ — это непустые строки, то сначала выполняется поиск в D по префиксу для α, а затем методом перебора проверяется, соответствует ли P возвращаемым строкам с учетом наличия подстановочных знаков. Понятно, что такой подход может оказаться очень затратным,

особенно когда α не является селективным префиксом и в качестве возможных соответствий возвращается много строк словаря. Так что этот вариант не подходит для запросов с подстановочными знаками в дисковой среде.

9.1.1. Непрерывное распределение строк

Некоторые из упомянутых ограничений можно обойти с помощью простого трюка — хранить на диске строки словаря, отсортированные лексикографически и смежные. Смежность указателя в A в дальнейшем отражается на смежности строк в S. Здесь мы имеем два основных преимущества.

- **Скорость.** Когда двоичный поиск ограничен несколькими строками, они будут храниться близко как в массиве A, так и в словаре S, потому что, скорее всего, буферизованы операционной системой во внутренней памяти.
- **Пространство.** К смежным строкам в S можно применять сжатие, ведь обычно они имеют общий префикс.

Учитывая, что словарь S хранится на диске, можно объединить строки в группы по B символов каждая, а затем сохранить в массиве A указатель на первую строку каждой группы. Выбранные строки обозначаются $D_B \subseteq D$, а их число n_B ограничено сверху как N/B, поскольку мы выбираем не более одной строки на блок. Поскольку массив A был сжат до индекса не более $n_B \leqslant n$ строк, поиск по нему нужно редактировать, чтобы отразить *двухуровневую структуру*, заданную массивом A и объединенными в блоки строками.

Идея состоит в разложении лексикографического поиска шаблона Q на двухэтапный процесс. На первом этапе ищется лексикографическая позиция Q в выбранных строках D_B, на втором — идентифицируется блок строк, в котором она находится, что позволяет просканировать эти строки и сравнить их с шаблоном на предмет соответствия префикса. Напомню, что при поиске префикса этот процесс выполняется для двух строк, P и $P\#$, поэтому мы доказали следующую теорему.

Теорема 9.2. *Поиск префикса по словарю D требует $O(p/B \log N/B)$ операций ввода-вывода. Извлечение строк с префиксом P потребует N_{occ}/B операций ввода-вывода, где N_{occ} — длина этих строк. Общее занимаемое пространство равно $N + n + n_B w$.*

Доказательство. После идентификации блока строк $A[l, r]$ с префиксом P нам потребуется $O(N_{occ}/B)$ операций ввода-вывода для сканирования непрерывной части словаря S, в которой содержатся эти строки. Занятость пространства можно оценить, вспомнив, что указатели хранятся только для n_B выбранных строк, где $n_B \leqslant N/B$. ■

Как правило, строки короче B, поэтому $N/B \leqslant n$, следовательно, это решение быстрее предыдущего. Кроме того, его можно эффективно объединить с техникой *кодирования префиксов*, чтобы еще больше снизить пространственную сложность и сложность ввода-вывода.

9.1.2. Кодирование префиксов

В последовательности отсортированных строк соседние строки часто имеют общий префикс. Обозначим количество общих символов ℓ. Эти символы можно заменить двоичным кодом переменной длины, который экономит несколько битов по сравнению с классическим кодом фиксированного размера на основе 4 или 8 байт. Конвертеры для кодирования и декодирования будут подробно описываться в главе 11, а здесь я покажу самый простой вариант. Конвертер добавляет к двоичному представлению ℓ нули, пока не заполнит целое число байтов, а затем отдает первые 2 бита дополнения (если таковые имеются, в противном случае добавляется еще 1 байт) под информацию о количестве используемых байтов[1]. Это кодирование не содержит префиксов, поэтому оно гарантирует уникальность декодирования, а выравнивание байтов обеспечивает высокую скорость декодирования на современных процессорах. Кроме того, начальные $\Theta(\ell \log_2 \sigma)$ бит, представляющие символы ℓ общего префикса, заменяются $O(\log \ell)$ битами целочисленного кодирования, что выгодно с точки зрения занимаемого пространства. Очевидно, что конечный эффект этого решения зависит от количества общих символов, которое в случае словаря URL-адресов может достигать 70 %.

Кодирование префиксов представляет собой алгоритм *дельта-кодирования*, который можно легко задать инкрементным способом. Для последовательности строк $(s_1, ..., s_n)$ строка s_i кодируется с помощью пары $(\ell_i, \hat{s}_i)$, где ℓ_i — длина общего префикса между s_i и его предшественником s_{i-1} (или 0, если $i = 1$), а $\hat{s}_i = s_i[\ell_i + 1, |s_i|]$ — оставшийся суффикс строки s_i. Например, словарь $\mathcal{D}$ = {`alcatraz`, `alcool`, `alcyone`, `anacleto`, `ananas`, `aster`, `astral`, `astronomy`} после кодирования префиксов будет выглядеть так: (0, `alcatraz`), (3, `ool`), (3, `yone`), (1, `nacleto`), (3, `nas`), (1, `ster`), (3, `ral`), (4, `onomy`).

Декодирование строки для пары $(\ell, \hat{s})$ выглядит симметрично. Символы l копируются из предыдущей строки, а затем к ним добавляется оставшийся суффикс $\hat{s}$. При условии, что предыдущая строка доступна, это занимает время $O(|s|)$ и требует $O(1 + |s|/B)$ операций ввода-вывода. В общем случае для реконструкции строки s_i может потребоваться обратное сканирование входной последовательности до первой строки s_1, которая доступна полностью. То есть, чтобы декодировать $(\ell_i, \hat{s}_i)$, возможно, придется просканировать $(\ell_{i-1}, \hat{s}_{i-1})$, ..., $(\ell_1, \hat{s}_1)$ и реконструировать s_1, ..., s_{i-1},. Следовательно, временные затраты на декодирование s_i могут оказаться намного выше оптимальных $\Theta(|s_i|)$[2].

Чтобы преодолеть этот недостаток, кодирование префиксов обычно применяют к блокам строк по двухуровневой схеме, которую я описывал в подразделе 9.1.1. Идея состоит в перезапуске кодирования префиксов в начале каждого блока, чтобы первая строка сохранялась *в исходном виде*. Процедуре префиксного поиска такой подход дает два преимущества: во-первых, несжатые строки участвуют в двоичном

---

[1] Предполагается, что ℓ можно закодировать в двоичном виде в 30 битах, а именно $\ell < 2^{30}$.

[2] Более разумно восстанавливать только первые ℓ символов предыдущих строк $s_1, s_2, ..., s_{i-1}$, поскольку в реконструкции s_i участвуют именно они.

поиске и их не нужно распаковывать для сравнения с шаблоном Q; во-вторых, каждый блок сжимается индивидуально, поэтому сканирование его строк для лексикографического поиска шаблона Q можно объединить с распаковкой этих строк. Такую схему хранения называют *кодированием префиксов с группировкой* (front coding with bucketing). Я обозначу ее FC_B. Наглядный пример демонстрируется на рис. 9.2. Строки alcatraz, alcyone, ananas и astral хранятся явно и целиком, поскольку они первые в каждом блоке.

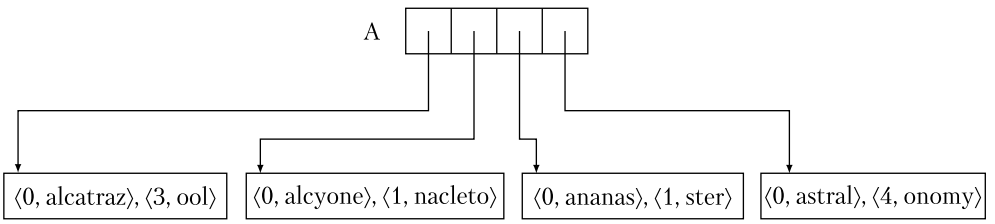


Рис. 9.2. Двухуровневая индексация набора строк $\mathcal{D}$ = {alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy}. Строки разбиваются на блоки по две строки в каждом, каждый блок хранится на одной странице диска и сжимается с помощью процедуры FC_B. Первая строка каждого блока доступна в несжатом виде (фактически для таких строк $l = 0$)

Положительный побочный эффект этого подхода заключается в том, что он уменьшает количество выбранных строк, так как позволяет увеличивать количество строк, хранящихся на одной странице диска. Начиная с s_1, мы по порядку сжимаем строки словаря $\mathcal{D}$. Когда сжатие строки s_i переполняет текущий блок, начинается новый блок. Последняя строка сохраняется в несжатом виде. Количество выборочных строк уменьшается примерно с N/B до примерно $FC_B(\mathcal{D})/B$, где $FC_B(\mathcal{D})$ — пространство, необходимое процедуре FC_B для хранения всех строк словаря в блоках размером B. Очевидно, что это положительно влияет на количество операций ввода-вывода, необходимых для префиксного поиска в ситуации, когда выполняется двоичный поиск по выбранным строкам. А вот пространства требуется больше, чем в случае $FC(\mathcal{D})$, так как первая строка каждого блока хранится несжатой. Но мы рассчитываем на то, что это увеличение незначительно, поскольку $B >> 1$.

Теорема 9.3. *Префиксный поиск по словарю D требует* $O\left(\frac{p}{B}\log\frac{FC_B(\mathcal{D})}{B}\right)$ *операций ввода-вывода. Извлечение строк с префиксом P требует* $O\left(\frac{FC_B(\mathcal{D}_{occ})}{B}\right)$ *операций ввода-вывода, где* $\mathcal{D}_{occ} \subseteq \mathcal{D}$ *— строки в наборе ответов.*

Итак, в общем случае сжатие строк является хорошей идеей, поскольку уменьшает как необходимое для хранения строк пространство, так и количество операций ввода-вывода. Однако следует учитывать, что FC-сжатие из-за необходимости распаковывать блоки может увеличивать временную сложность сканирования блока с $\Theta(B)$ до $\Theta(B^2)$. Для примера возьмем последовательность строк (a, aa, aaa...).

После префиксного сжатия она примет вот вид $(0, a)$, $(1, a)$, $(2, a)$, $(3, a)$... На одной странице диска можно хранить $\Theta(B)$ таких пар, представляющих $\Theta(B)$ строк общей длиной $\sum_{i=0}^{B}\Theta(i)=\Theta\left(B^2\right)$.

Несмотря на эти патологические случаи, на практике оказывается, что пространство сокращается на постоянный множитель, поэтому увеличение времени, вызванное сканированием блока, незначительно. Фактически мы имеем компромисс по времени/пространству, обусловленный размером блока B. Чем длиннее B, тем выше степень сжатия и тем быстрее выполняется бинарный поиск, но тем медленнее происходит сканирование блока. Кроме того, выбор B влияет на занятость массива A и возможное копирование во внутреннюю память несжатых строк, на которые этот массив указывает, что сокращает количество операций ввода-вывода во время бинарного поиска.

Чтобы разобраться с этим компромиссом, разделим задачи поиска и сжатия. Обращу ваше внимание на то, что предлагаемая структура данных состоит из двух уровней: верхний содержит ссылки на базу данных *выбранных строк* $\mathcal{D}_B$, а на нижнем находятся сами строки, сохраненные в виде блоков. Выбор алгоритмов и структур данных, используемых на этих уровнях, ортогонален друг другу, так что может осуществляться независимо. Само собой разумеется, что наша двухуровневая схема поиска и хранения словаря строк подходит для применения в двухуровневой иерархии памяти, такой как кэш и внутренняя память. Это типично для поиска в Интернете, где в роли $\mathcal{D}$ выступают запросы пользователей и необходимо избегать доступа к диску, чтобы каждый поиск по ключевым словам мог выполняться за несколько миллисекунд.

В следующих разделах я покажу вам четыре улучшения двухуровневого решения: первое касается хранения строк словаря, позволяющего обеспечить оптимальную с точки зрения ввода-вывода декомпрессию строк, остальные три — эффективной индексации выбранных строк. Они интересны и сами по себе, так что не стоит ограничивать их применение описанными в этой главе вариантами.

9.2. Кодирование префиксов с сохранением локальности∞

Рассмотрим элегантный вариант кодирования префиксов, обеспечивающий контролируемый компромисс между занятостью пространства и временем декодирования одной строки [2]. Основная идея проста и легко реализуема, а вот доказать ее гарантированные границы уже не так просто. Формулируется она так: *строка принимает участие в кодировании префиксов только в случае, когда время ее декодирования пропорционально ее длине, в противном случае она записывается несжатой*. С временной сложностью все очевидно: строка подвергается сжатию только в том случае, когда декодирование оптимально. Что поразительно, если детально рассмотреть оптимальность декодирования строки по времени, окажется, что ее «константа пропорциональности» контролирует также занятость пространства сжатыми строками. Это похоже на волшебство!

Предположим, что мы провели префиксное кодирование первых $i-1$ строк ($s_1 \ldots s_{i-1}$), превратив их в сжатую последовательность $F = (0, \hat{s}_1), (\ell_2, \hat{s}_2) \ldots (\ell_{i-1}, \hat{s}_{i-1})$. Чтобы сжать s_i, требуется обратное сканирование не более $c|s_i|$ символов F, чтобы проверить, хватит ли их для восстановления s_i. Фактически это означает, что несжатая строка входит в эти символы, потому что у нас есть первый символ для s_i. Если это так, мы выполняем кодирование префикса s_i в $(\ell_i, \hat{s}_i)$, в противном случае s_i без сжатия копируется в F, приводя к появлению пары $(0, s_i)$. Эти два случая наглядно демонстрируются на рис. 9.3.

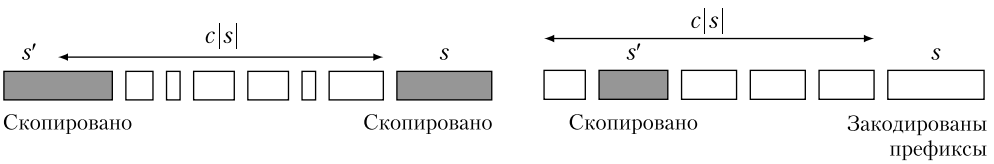


Рис. 9.3. Два варианта сжатия словарной строки s при кодировании префиксов с сохранением локальности (LPFC). Серые прямоугольники копируются без сжатия (в частности, s' — это скопированная строка, предшествующая s), белые прямоугольники представляют строки после префиксного кодирования

Основная сложность состоит в том, чтобы показать, что как строки, оставшиеся несжатыми, так и строки, сжатые классическим кодированием префиксов, имеют длину, которую можно контролировать с помощью параметра c, как показывает следующая теорема, в которой фигурирует параметр $\varepsilon = c/(c-2)$.

Теорема 9.4. *Кодирование префиксов с сохранением локальности (locality-preserving front coding, LPFC) занимает пространство не более* $(1+\varepsilon)FC(\mathcal{D})$ *и поддерживает декодирование любой словарной строки* s *за* $O(|s|/(\varepsilon B))$ *оптимальных операций ввода-вывода.*

Доказательство. Любую несжатую строку s будем считать *скопированной*, а исследованные в процессе обратного сканирования символы $c|s|$ назовем *левым экстентом* строки s. Скопированной строки, предшествующей s и начинающейся в ее левом экстенте, существовать не может. В противном случае s подверглась бы процедуре кодирования префикса. Более того, предшествующая s скопированная строка может заканчиваться в левом экстенте s (см. левую часть рис. 9.3). Для наглядности символы, принадлежащие выходному суффиксу $\hat{s}$ строки s с префиксным кодированием, называются *FC-символами*.

Очевидно, что пространство, которое строки занимают после кодирования префиксов, ограничено сверху как $FC(\mathcal{D})$. Я хочу показать, что пространство, занимаемое скопированными строками, которые, возможно, подверглись классическому кодированию префиксов, но остались несжатыми, в сумме составит $\varepsilon FC(\mathcal{D})$. Здесь, как вы увидите в конце доказательства, параметр ε зависит от соотношения $c/(c-2)$ и равен ему.

Рассмотрим для скопированных строк два случая, зависящих от количества FC-символов, находящихся между двумя последовательными их появлениями. Первый случай называется *разреженным* (uncrowded) и возникает при количестве FC-символов не менее $c|s|/2$ (показано в правой части рис. 9.4). Второй случай называется *плотным* (crowded) и возникает, когда количество FC-символов составляет не более $c|s|/2$ (показано в левой части рис. 9.4). Из рисунка легко сделать вывод, что при плотной скопированной строке s мы имеем $|s'| \geqslant c|s|/2$, потому что s' начинается до левого экстента s (в противном случае строка s не будет скопирована), а заканчивается в пределах последних $c|s|/2$ символов этого экстента (иначе s будет разреженной). И наоборот, из рис. 9.4 можно заключить, что разреженной строке s предшествуют по крайней мере $c|s|/2$ FC-символов.

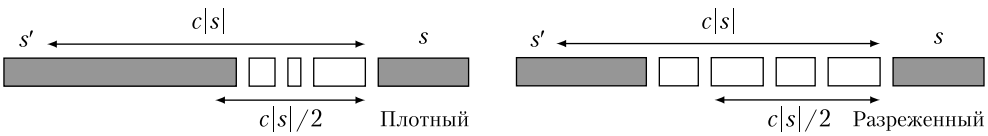


Рис. 9.4. Два случая, возникающих при LPFC. Белые прямоугольники обозначают строки после кодирования префиксов и, следовательно, их FC-символы, а серые прямоугольники соответствуют двум последовательным скопированным, а значит, несжатым строкам. Случаи плотных *(слева)* и разреженных *(справа)* копий разграничиваются

Все готово для оценки общей длины скопированных (несжатых) строк. Разобьем их на цепочки, состоящие из одной разреженной скопированной строки, за которой следует максимальная последовательность плотных скопированных строк. По определению считается, что первой в $\mathcal{D}$ идет разреженная строка, так как во всех случаях это скопированная строка. Нужно доказать, что общее количество символов каждой цепочки пропорционально длине ее первой скопированной строки. Возьмем цепочку $w_1 w_2 ... w_x$ последовательных скопированных строк, где w_1 — разреженная строка, а все последующие w_i — плотные. Для любой строки w_i, где $i > 1$, справедливо доказанное ранее неравенство, а именно $|w_{i-1}| \geqslant c|w_i|/2$, или, что эквивалентно, $|w_i| \leqslant 2|w_{i-1}|/c = ... = (2/c)^{i-1}|w_1|$ для всех $i = 2, 3... x$. При $c > 2$ получаем, что плотные скопированные строки уменьшаются на постоянный множитель. Поэтому сверху общее количество символов в цепочке можно ограничить вот так:

$$\sum_{i=1}^{x}|w_i| = |w_1| + \sum_{i=2}^{x}|w_i| \leqslant |w_1| + \sum_{i=2}^{x}(2/c)^{i-1}|w_1| =$$
$$= |w_1|\sum_{i=1}^{x}(2/c)^i < |w_1|\sum_{i\geq 0}(2/c)^i < \frac{c|w_1|}{c-2}.$$

По определению разреженной строки строке w_1 предшествует по крайней мере $c|w_1|/2$ FC-символов. Общее количество FC-символов ограничено значением $\mathrm{FC}(\mathcal{D})$, поэтому сверху общую длину разреженных строк можно ограничить значением $(2/c)\mathrm{FC}(\mathcal{D})$. Подставляя это в предыдущую границу для общей длины

цепочек, получим $(c/(c-2))(2\mathrm{FC}(\mathcal{D}))/c = 2/(c-2)\mathrm{FC}(\mathcal{D})$. Эта теорема вытекает из равенства $\varepsilon = 2/(c-2)$. ■

Итак, кодирование префиксов с сохранением локальности (LPFC) — это схема сжатого хранения строк. Причем переход от простого хранения к этой схеме происходит без какого-либо асимптотического замедления на доступ к сжатым строкам. Занятость пространства при этом такая же, как в случае классического префиксного кодирования с точностью до постоянного множителя. В этом смысле LPFC можно рассматривать как своего рода *увеличитель пространства* для любой техники индексации строк.

Двухуровневую структуру индексации данных из предыдущего раздела можно с выгодой использовать в LPFC, заставив массив A указывать на скопированные (несжатые) строки. Ограниченные этими строками группы будут иметь *переменную длину*, но распаковка каждой строки может быть выполнена за оптимальное время и оптимальное количество операций ввода-вывода. Так что мы имеем границы из теоремы 9.3, разумеется, если не рассматривать патологические случаи (сравните с размером группы $\Theta(B^2)$ в классической схеме FC_B). Количество операций ввода-вывода и время, необходимые для сканирования строк с префиксом P и идентифицированные на шаге бинарного поиска, по-прежнему пропорциональны их общей длине, следовательно, это оптимальный результат.

Остается вопрос, как ускорить поиск по массиву A. Здесь возможны два ограничения. Во-первых, шаг бинарного поиска может иметь временную сложность, зависящую от N или n. Во-вторых, если строки, на которые указывает массив A, не помещаются в выделенную под них внутреннюю память или недоступны в кэше, этап бинарного поиска будет сопровождаться множеством операций ввода-вывода и промахов кэша. В разделах 9.3–9.5 я продемонстрирую три подхода, в полной мере использующих преимущества распределения словарных строк или более сложной их индексации.

9.3. Интерполяционный поиск

Рассмотрим словарь строк, составленных из алфавита размером Σ. Длина строк не превышает b символов. Эти строки можно интерпретировать как целые числа в пространстве размером σ^b. В некоторых приложениях роль ключей могут играть короткие двоичные строки, помещающиеся в слово памяти, то есть $b = 4$ или 8 байт, а $\sigma = 256$.

Поиск лексикографической позиции строки P в словаре $\mathcal{D}$ сводится к поиску целочисленного кода этой строки среди упорядоченного набора целых чисел, представляющих все строки словаря. Выполнять преобразования целых чисел следует с осторожностью. Имеет смысл исходить из предположения об одинаковой длине строк. Для соблюдения этого условия более короткие строки логически дополняются символом, который меньше любого другого символа алфавита. После этого

поиск P и $P\#$ сведется к поиску двух подходящих целых чисел. Для определенного распределения сгенерированных из строк чисел существуют варианты поиска, работающие быстрее, чем бинарный поиск. Сейчас я покажу вам вариант классического *интерполяционного поиска*, предлагающий ряд интересных дополнительных свойств (подробно об этом написано в [4]).

Пусть $X[1, m] = x_1 ... x_m$ — массив отсортированных целых чисел, кодирующих строки словаря, так что при кодировании целого словаря $m = n$, а если кодируется только выборка $\mathcal{D}_B$, то $m = n_B$. Равномерно распределим диапазон $[x_1, x_m]$ по m корзинам $B_1 ... B_m$ размером $b = (x_m - x_1 + 1) / m$. В этом случае количество корзин совпадает с количеством строк словаря, а $B_i = [x_1 + (i - 1)b, x_1 + ib)$. Чтобы гарантировать постоянный доступ к этим ячейкам и упростить изложение, добавим дополнительный массив $S[1, m]$ указателей на первые и последние целые числа B_i в X.

На рис. 9.5 показан пример, в котором $m = 12$ целых чисел (и корзин). Первое целое число — $x_1 = 1$, последнее целое число — $x_{12} = 36$, таким образом, размер корзины $b = (36 - 1 + 1) / 12 = 3$. Показаны только непустые корзины, а целые числа массива X в них разделены длинными вертикальными чертами.

B_1			B_3		B_6	B_7		B_{10}		B_{11}	B_{12}
1	2	3	8	9	17	19	20	28	30	32	36

Рис. 9.5. Пример интерполяционного поиска по набору из 12 целых чисел. Корзины разделены длинными вертикальными линиями. Некоторые из них пусты, например B_2, B_4, B_5, B_8 и B_9, и поэтому не показаны

Дана строка P, по которой нужно провести лексикографический поиск среди строк словаря. Нужно вычислить соответствующее ей целое число, скажем y, и определить индекс j ячейки-кандидата, где может располагаться это число: $j = \left\lfloor \frac{y - x_1}{b} \right\rfloor + 1$. Затем остается выполнить двоичный поиск позиции y в B_j, обращаясь к подмассиву массива X, разделенному двумя хранящимися в $S[j]$ указателями. Эти действия требуют времени $O(1 + \log |B_j|) = O(\log b)$. Значение b зависит от величины целых чисел в индексированном словаре, но в любом случае оно составляет $|B_j| \leqslant n$, что дает для худшего случая оценку $O(\log b) = O(\log n)$. Как видите, асимптотически этот подход не быстрее, чем двоичный поиск.

У этого подхода есть два интересных свойства, касающихся распределения целых чисел X и влияния этого распределения на временную сложность. Определим параметр Δ как отношение максимального и минимального промежутков между двумя последовательными целыми числами входного словаря. Формально это выглядит так:

$$\Delta = \frac{\max_{i=2...m}\left(x_i - x_{i-1}\right)}{\min_{i=2...m}\left(x_i - x_{i-1}\right)}.$$

Обратите внимание, что алгоритм не зависит от значения Δ, тем не менее следующая теорема показывает, что его временная сложность может быть ограничена в терминах Δ. Следовательно, интерполяционный поиск не может быть медленнее бинарного, но может оказаться быстрее в зависимости от распределения ключей словаря.

Теорема 9.5. *Интерполяционный поиск по словарю из m целых чисел занимает время $O(\log \min\{\Delta, m\})$ и в худшем случае требует дополнительное пространство $O(m)$.*

Доказательство. Корректность этой теоремы очевидна. Для временной сложности нужно доказать, что максимальное количество целых чисел в любой корзине не может превышать значение b/g, где g определяется как минимальный промежуток между последовательными целыми числами X. Достаточно показать, что $b/g \leqslant \Delta$, и утверждение будет доказано.

Обратите внимание на то, что для ряда целых чисел максимальное значение будет по крайней мере таким же, как их среднее. Для промежутков $x_i - x_{i-1}$ между последовательными отсортированными целыми числами в X мы имеем:

$$\max_{i=2\ldots m}\left(x_i - x_{i-1}\right) \geqslant \frac{\sum_{i=2}^{m} x_i - x_{i-1}}{m-1} \geqslant \frac{x_m - x_1 + 1}{m}. \tag{9.1}$$

Последнее неравенство здесь вытекает из следующего арифметического свойства: $\frac{a'}{a''} \geqslant \frac{a'+1}{a''+1}$ всякий раз, когда $a' \geqslant a'' > 0$. Это свойство можно легко доказать, решив уравнение. Для конечного соотношения в уравнении (9.1) оно справедливо, поскольку целые числа в X положительны и различны, так что $x_m - x_1 \geqslant m - 1$.

Начав с предварительного наблюдения $|B_i| \leqslant b/g$, а затем подставив уравнение (9.1) и определение Δ, можно записать:

$$\left|B_i\right| \leqslant \frac{b}{g} \leqslant \frac{\max_{i=2\ldots m}\left(x_i - x_{i-1}\right)}{\min_{i=2\ldots m}\left(x_i - x_{i-1}\right)} = \Delta. \qquad \blacksquare$$

При равномерном распределении целых чисел X можно вывести очень интересную границу максимального размера ячейки, которая с высокой вероятностью будет соблюдаться. Фактически мы ищем максимальную загрузку m ячеек (B_is), среди которых равномерно случайным образом распределяются m шаров (целые числа X). Для задачи в такой формулировке максимальное значение хорошо известно и равно $O(\log m/\log \log m)$, что и было доказано в теореме 8.3.

Лемма 9.1. *Если из диапазона $[0, U-1]$ равномерно случайным образом выбираются m целых чисел множества X, то интерполяционный поиск с высокой вероятностью займет время $O(\log \log m)$.*

Проблема в том, что на практике равномерное распределение входных данных встречается редко. Тем не менее если снизить требования к запросу, ограничившись запросом членства, то равномерное распределение любых входных данных можно обеспечить искусственно, взяв случайную перестановку $\pi : U \to U$ и перетасовав X в соответствии с ней перед построением предлагаемой структуры данных. Необходимо внимательно составлять запрос, ведь мы ищем не y, а его образ $\pi(y)$ в наборе $\pi(X)$. Тогда с высокой вероятностью для *запроса членства* мы сохраним производительность, указанную в лемме 9.1, причем для *любого индексированного множества X*. О выборе π можно почитать в [8].

Если применить интерполяционный поиск в контексте строк, исходя из предположения о равномерности распределения выбранных строк $\mathcal{D}_B$ (а такое может быть после выборки, если B не слишком мало), то количество операций ввода-вывода, требующихся для префиксного поиска по строкам, составит $O(p/B \log\log N/B)$. Это экспоненциальное сокращение времени и производительности ввода-вывода поиска префикса, выполняемого с помощью бинарного поиска, как указано в теореме 9.2.

9.4. Сжатое префиксное дерево

В главе 7 я уже рассказывал о префиксных деревьях, пришло время более подробно рассмотреть их свойства как эффективных структур данных для поиска строк. В этом контексте они используются для индексации выборки строк $\mathcal{D}_B$ во внутренней памяти. Это ускоряет первый этап лексикографического поиска строки Q (P или $P\#$) в наборе $\mathcal{D}_B$ с временем от $O(\log(N/B))$ до $O(p)$, демонстрируя удивительную независимость от размера словаря. Причина кроется в свойствах RAM-модели, которая позволяет за постоянное время обращаться к ячейкам памяти размером $\Theta(\log N)$ бит. Давайте более детально рассмотрим применение префиксных деревьев для поиска префиксов.

Ребра префиксного дерева помечены символами индексированных строк. Внутренний узел u связан со строкой $s[u]$, которая представляет собой *префикс* словарной строки. Строка $s[u]$ получается конкатенацией символов на нисходящем пути, соединяющем корень префиксного дерева с узлом u. Каждый лист связан со строкой словаря. Все листья, которые происходят от узла u, имеют префикс $s[u]$. В префиксном дереве n листьев и самое большее N узлов, по одному на символ строки[1]. Префиксное дерево, построенное на семи строках, демонстрируется на рис. 9.6. Такую форму обычно называют *несжатой*, потому что она может иметь *унарные пути*, такие как путь к строке `inn`[2].

---

[1] Я говорю «самое большее», потому что некоторые пути (префиксы) могут быть общими для нескольких строк.

[2] Префиксное дерево не может индексировать строки, если одна из них является префиксом другой. Фактически первая строка оказывается во внутреннем узле. Чтобы избежать такой ситуации, каждая строка расширяется специальным символом \$, которого нет в алфавите.

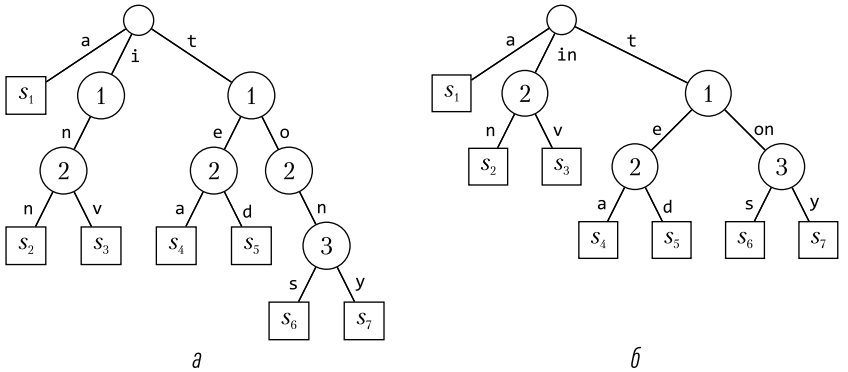


Рис. 9.6. Примеры префиксного дерева для $n = 7$ строк: *а* — несжатого; *б* — сжатого

На рис. 9.6 целое число в каждом внутреннем узле u обозначает длину строки на пути от корня к u. В несжатых деревьях эти целые числа не имеют особого смысла, поскольку соответствуют глубине u. В сжатых деревьях метки ребер представляют собой подстроки переменной длины, которые в постоянном пространстве можно представить как тройки целых чисел: например, on можно закодировать как $\langle 6, 2, 3\rangle$, поскольку шестая строка tons включает строку on с позиции *2* до позиции *3*.

Чтобы проверить, является ли строка P префиксом какой-то словарной строки, нужно убедиться в наличии нисходящего пути от корня префиксного дерева, который читается как P. Все посещаемые при обходе листья составляют словарную строку, которую остается сравнить с указанным префиксом, так как префиксные деревья не нуждаются в лексикографическом поиске с обязательным индексированием всего словаря $\mathcal{D}$.

Но если строки $\mathcal{D}$ хранятся на диске, то для индексации базы данных выборочных словарных строк $\mathcal{D}_B$ требуются двухуровневая индексация и несжатое дерево. В этом случае не обойтись без лексикографического поиска правильно построенной строки Q в несжатом дереве. Если Q корректно считывается, поиск заканчивается на некотором внутреннем узле u. В противном случае нисходящий обход дерева останавливается на узле v при незавершенном сканировании Q, скажем, на символе $Q[\ell]$. В первом случае лексикографическая позиция Q определяется путем обхода самого левого ребра поддерева с корнем в u. Во втором случае l дает самый длинный общий префикс между строками Q и $\mathcal{D}_B$, а лексикографическое положение Q в $\mathcal{D}_B$ определяется путем размещения $Q[\ell]$ среди меток ребер, исходящих из v, то есть среди поддеревьев, спускающихся из этих ребер.

В качестве примера рассмотрим лексикографический поиск шаблона Q = `to` в несжатом префиксном дереве (см. рис. 9.6, *а*). Этот шаблон предваряет строки s_6 и s_7 и, по сути, полностью считывается при нисходящем обходе дерева, при котором мы обнаруживаем, что унарный узел оказывается самым правым узлом с меткой 2. Назовем его v и рассмотрим новый шаблон для лексикографического поиска Q = `tod`. В этом случае после достижения v алгоритм начнет проверять наличие исходящего

ребра с пометкой d. Но такого ребра не существует, то есть лексикографическая позиция Q находится слева от нисходящего поддерева, потому что d меньше символа n, помечающего одиночное ребро, исходящее из v.

Большой проблемой является поиск подходящего ребра во время нисходящего обхода префиксного дерева, ведь именно этот выбор влияет на общую эффективность поиска. Успешность этого шага зависит от правильного хранения исходящих из узла ребер (и их маркеров). Простейшая структура данных для этой цели — это *связный список*. Его требования к пространству оптимальны, а именно пропорциональны количеству исходящих ребер. В худшем случае сюда добавляются затраты $\Theta(\sigma)$ на каждый пройденный узел. В результате мы получаем префиксный поиск, в худшем случае занимающий $O(p\sigma)$. Для больших алфавитов это слишком много. Если символы ветвления и их ребра сохранить в отсортированном массиве, двоичный поиск будет требовать времени на узел $O(\log \sigma)$. Для ускорения потребуется полноразмерный массив из σ записей, непустые записи которого будут хранить указатели на дочерние элементы, связанные с существующими символами ветвления. В этом случае время прохождения ветвления составит $O(1)$, а для поиска шаблона Q потребуется время $O(p)$. Правда, тогда пространство, занимаемое префиксным деревом, возрастает до $O(N\sigma)$, что в случае больших алфавитов может оказаться неприемлемым. Лучше всего обратиться к идеальной хеш-таблице, хранящей только существующие символы ветвления и связанные с ними указатели. Это гарантирует время ветвления $O(1)$ в худшем случае и оптимальное занимаемое пространство, тем самым объединяя лучшее из двух предыдущих решений. Подробно идеальные хеши рассматривались в главе 8.

Теорема 9.6. *Несжатое префиксное дерево решает задачу поиска префикса за время $O(p + n_{occ})$ и $O(p + n_{occ}/B)$ операций ввода-вывода, где n_{occ} — это количество строк с префиксом P. Извлечение этих строк занимает время $O(N_{occ})$ и (N_{occ}/B) операций ввода-вывода. В несжатом префиксном дереве $O(N)$ узлов и ровно n листьев, таким образом, оно занимает $O(N)$ места. Наконец, такое дерево поддерживает лексикографический поиск шаблона P среди индексированных строк за время $O(p + \log \sigma)$ в худшем случае и за $O(p + \log \sigma)$ операций ввода-вывода.*

Доказательство. Пусть u — узел, для которого $s[u] = P$. Все происходящие от него строки имеют префикс P, и их можно визуализировать, посетив поддерево с корнем в u. Для обхода, ведущего к u, сложность ввода-вывода составляет $O(p)$ из-за прыжков с постоянным временем между узлами дерева через идеальные хеш-таблицы. Извлечение происходящих от узла P листьев n_{occ} требует оптимальных $O(n_{occ}/B)$ операций ввода-вывода, поскольку мы предположили, что листья префиксного дерева хранятся на диске непрерывно и каждый узел хранит указатель на свои самые левые и самые правые нисходящие листья. В то же время при условии, что индексированные строки упорядочены и хранятся на диске непрерывно, для отображения связанных с этими листьями строк потребуются дополнительные $O(N_{occ}/B)$ операций ввода-вывода. Поиск среди меток ребер, исходящих из достигнутого узла, лексикографической позиции несовпадающего символа в P, то есть $Q[\ell]$, занимает

время $O(\log \sigma)$ при условии, что у нас реализован массив символов ветвления (сжатые и более быстрые решения подробно рассматриваются в главе 15). ■

Префиксное дерево с длинными строками с коротким общим префиксом может занимать слишком большое пространство, так как в нем появляется довольно много унарных узлов. Для экономии пространства можно сжимать унарные пути в одно ребро. В этом случае метки ребер из отдельных символов превращаются в подстроки (возможно, длинные), и такое префиксное дерево называется *сжатым*. Пример такого дерева был показан на рис. 9.6, *б*. Очевидно, что каждая метка ребра является подстрокой словарной строки, скажем $s[i,j]$, поэтому ее можно представить в виде тройки $\langle s, i, j\rangle$. Поскольку каждый узел является по крайней мере двоичным, количество внутренних узлов и ребер составит $O(n)$ (сравните с $O(N)$ у несжатых префиксных деревьев). Таким образом, общее пространство, необходимое сжатому префиксному дереву, также равно $O(n)$.

Теорема 9.7. *Сжатое префиксное дерево решает задачу поиска префикса за время* $O(p + n_{occ})$ *и* $O(p + n_{occ}/B)$ *операций ввода-вывода, где* n_{occ} *— количество строк с префиксом* P*. Эти отсортированные по алфавиту строки хранятся на диске непрерывно. Извлечение этих строк требует времени* $O(N_{occ})$ *и* $O(N_{occ}/B)$ *операций ввода-вывода. Сжатое префиксное дерево состоит из* $O(n)$ *узлов и листьев, поэтому для его хранения понадобится* $O(n)$ *места. Само собой разумеется, что такому дереву под хранение строк словаря для разрешения меток ребер требуется дополнительно* N *места. Наконец, сжатое префиксное дерево поддерживает лексикографический поиск шаблона* P *среди индексированных строк за время* $O(p + \log \sigma)$ *в худшем случае и за столько же операций ввода-вывода.*

Доказательство. Поиск префикса в данном случае реализован аналогично поиску по несжатым деревьям. Разница состоит в том, что он чередует символы-ветви из внутренних узлов и подстроки, соответствующие меткам ребер. Если символы ветвления и связанные указатели ребер, исходящие из внутренних узлов, снова реализованы с помощью идеальных хеш-таблиц, то заявленные границы времени и границы ввода-вывода легко доказываются.

Поиск лексикографической позиции шаблона Q среди индексированных строк можно реализовать аналогично тому, как это делалось в несжатом дереве. Достаточно пройти вниз, как можно чаще записывая символы, пока не встретится несоответствующий символ или не будет полностью прочитан шаблон Q. В отличие от поиска в несжатом дереве, обход может остановиться на середине ребра, но все выводы, сделанные для несжатых деревьев, остаются в силе (сжатые и более быстрые решения подробно рассматриваются в главе 15). ■

Таким образом, сжатое префиксное дерево является интересной заменой массива A в двухуровневой структуре индексации (см. подраздел 9.1.1) и может использоваться для поддержки поиска корзины, в которой встречается строка P, что в худшем случае займет время $O(p + \log \sigma)$. Поскольку проход каждого ребра требует одного

ввода-вывода на извлечение его подстроки маркировки для сравнения с соответствующей подстрокой в P, этот подход эффективен, когда префиксное дерево и его индексированные строки могут быть помещены во внутреннюю память. В противном случае (поскольку σ обычно не очень велико) он имеет два основных недостатка: линейную зависимость количества вводов-выводов от длины шаблона и зависимость занимаемого пространства от размера страницы диска, влияющего на выборку из $\mathcal{D}$ для формирования $\mathcal{D}_B$, и от длины выбранных строк.

Первую проблему решает представленное в следующем разделе patricia-дерево, а его сочетание с префиксным сжатием с сохранением локальности решает обе проблемы.

9.5. Patricia-дерево

Patricia-дерево, построенное на словаре $\mathcal{D}$, состоящем из n строк общей длиной N, является сжатым префиксным деревом, в котором метки ребер состоят только из их начальных *одиночных символов*, а внутренние узлы помечены целыми числами, обозначающими *длину* связанных строк. На рис. 9.7 показано, как преобразовать сжатое префиксное дерево *(слева)* в patricia-дерево *(справа)*.

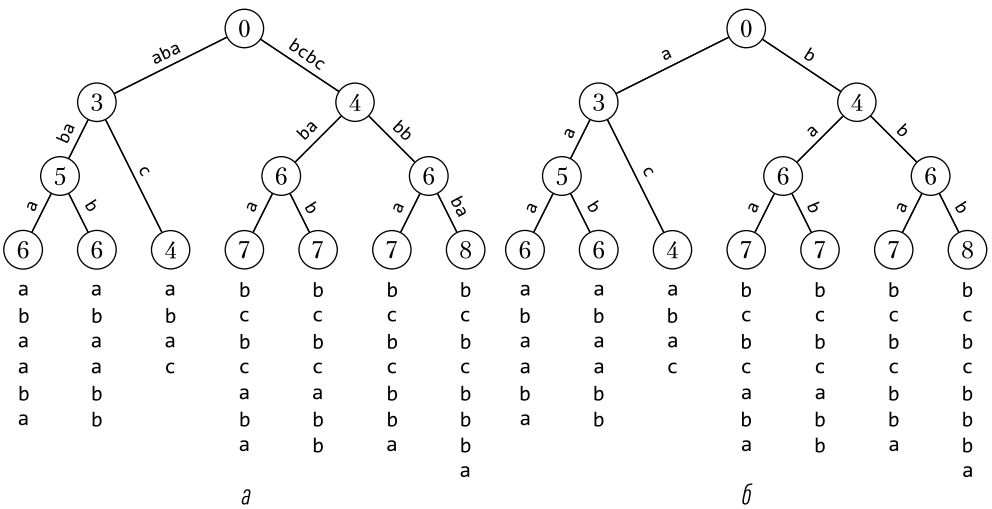


Рис. 9.7. Примеры: *а* — сжатого префиксного дерева; *б* — соответствующего ему patricia-дерева

Даже если patricia удаляет часть информации из сжатого префиксного дерева, оно все равно может поддерживать поиск лексикографической позиции шаблона P среди отсортированной последовательности строк. Более того, оно дает существенное преимущество, так как этот поиск требует доступа только к одной строке, а следовательно, обычно выполняет один ввод-вывод вместо p вводов-выводов, которые вызваны необходимостью разрешения ребер в сжатых префиксных деревьях.

В литературе [5] этот алгоритм называется *слепым поиском*. Он немного сложнее префиксного поиска в классических префиксных деревьях из-за того, что метка каждого ребра состоит всего из одного символа. С технической точки зрения слепой поиск можно разбить на три этапа.

- **Этап 1 — нисходящий обход.** Идем по patricia-дереву вниз в поиске листа l, который указывает на интересующую нас строку индексированного словаря. Эта строка не обязательно идентифицирует лексикографическую позицию P в словаре (что является нашей целью), но дает *достаточно информации* для дальнейшего поиска. Извлечение листа l выполняется обходом patricia-дерева от корня и сравнением символов префикса P с символами, которыми помечены пройденные ребра. Обход выполняется, пока не будет достигнут нужный лист или не закончится ветвление. В последнем случае в качестве l выбирается любой лист — потомок последнего пройденного узла.
- **Этап 2 — вычисление LCP.** Сравниваем P со строкой, на которую указывает лист l, чтобы определить их самый длинный общий префикс (longest common prefix, LCP). Обозначим длину этого общего префикса ℓ. Можно доказать (см. [5]), что лист l хранит индексированную patricia-деревом строку, которая разделяет самый длинный общий префикс с P. Два несовпадающих символа этих строк обозначим $P[\ell + 1]$ и $s[\ell + 1]$.
- **Этап 3 — восходящий обход.** Идем по patricia-дереву вверх от l в поиске ребра $e = (u, v)$, хранящего несовпадающий символ $s[\ell + 1]$. Это легко реализуется, потому что каждый узел на восходящем пути хранит целое число, которое обозначает длину соответствующего префикса s. Таким образом, мы имеем $|s[u]| < \ell + 1 \leqslant |s[v]|$. Если $s[\ell + 1]$ является символом ветвления, то есть $\ell = |s[u]|$, то лексикографическая позиция $P[\ell + 1]$ оказывается среди символов ветвления узла u. Допустим, это i-й дочерний элемент u. В этом случае лексикографическая позиция P окажется непосредственно слева от поддерева этого дочернего элемента. В противном случае (при $\ell > |s[u]|$) символ $s[\ell + 1]$ лежит внутри ребра e и располагается после его первого символа. Поэтому лексикографически P оказывается непосредственно справа от поддерева, нисходящего из e, при условии, что $P[\ell + 1] > s[\ell + 1]$, в противном случае он будет слева от этого поддерева.

Пример поиска показан на рис. 9.8. На этапе *1* происходит движение вниз по patricia-дереву, по самому правому пути (выделен жирным шрифтом), до листа s_7 = `bcbcbbba`. Так происходит потому, что символы, помечающие ребра пути, в позициях 1, 5, 7 соответствуют символам шаблона. Эти числа — +1 к числам, помечающим узлы в пройденном пути, принимая во внимание тот факт, что они обозначают позиции символов ветвления. На этапе *2* вычисляется самый длинный общий префикс между P = `bcbabcba` и s_8. Выясняется, что он равен 3, так как дальше идут несовпадения $s_7[4] = c$ и $P[4] = a$. На этапе *3* мы поднимаемся по дереву вверх от листа s_7 с остановкой на ребре, указанном на рис. 9.8, *б*. Выясняется, что P лежит слева от этого ребра, а следовательно, слева от его нисходящего поддерева. Это корректная лексикографическая позиция искомого шаблона среди проиндексированных строк словаря.

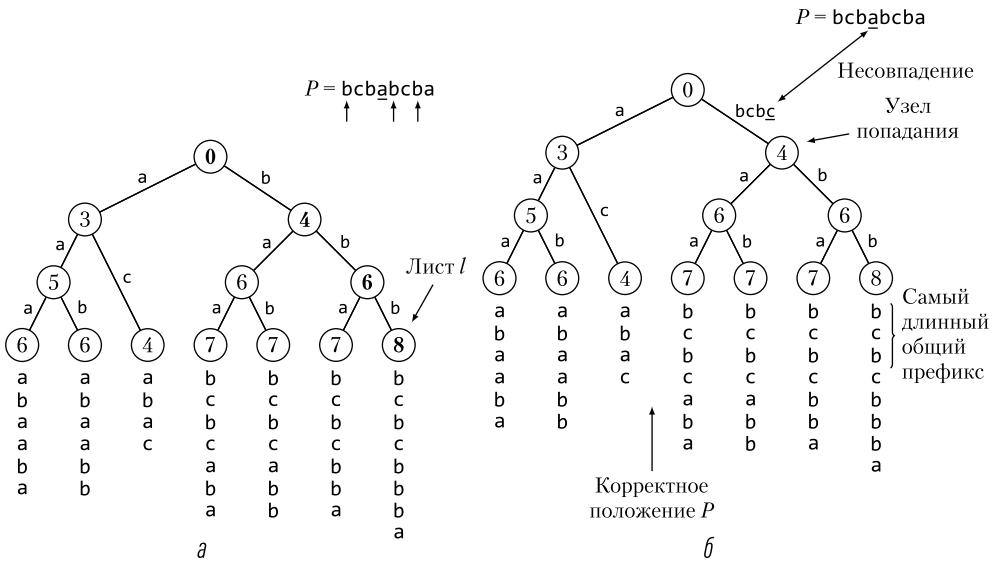


Рис. 9.8. Слепой поиск префикса P в словаре из семи строк: *а* — этап *1*; *б* — этапы *2* и *3*

Если взять другой шаблон P = ababbb, то этап *1* остановится на самом левом потомке корня, потому что все его символы ветвления отличны от $P[4] = b$. Затем в качестве листа l будет выбран произвольный лист, спускающийся с этого узла. Например, выберем лист, указывающий на s_1. Дальше будет вычислен самый длинный общий префикс между s_7 и P, и нам вернут значение 3. Наконец, на этапе *3* восходящий обход от s_1 остановится на самом левом потомке корня, обнаружив, что позиция $P[4] = b$ находится между двумя ребрами ветвления этого узла. Значит, лексикографическое положение P лежит между поддеревьями, спускающимися с этих двух ребер ветвления, то есть между строками s_2 и s_3.

Чтобы понять, почему этот алгоритм верен, рассмотрим путь, соответствующий префиксу $P[1, \ell]$ — самому длинному общему префиксу между P и строками словаря. Возможны два случая: мы достигли внутреннего узла u, такого, что $|s[u]| = \ell$, или находимся в середине ребра (u, v), а именно $|s[u]| < \ell < |s[v]|$. В первом случае все строки, нисходящие из u, имеют общие с шаблоном ℓ символов. При этом по определению LCP ни одна строка словаря не имеет большего количества общих символов. Правильная лексикографическая позиция, таким образом, попадает в середину этих строк или является смежной с ними, и ее можно найти, изучив символы ветвления ребер, исходящих из узла u. Корректно это делается слепым поиском, который останавливается на узле u (этап *1*), вычисляет l (этап *2*) и, наконец, определяет правильную позицию P, сравнивая символы ветвления u с $P[\ell + 1]$ (этап *3*).

Во втором случае слепой поиск достигает узла v, пропуская несовпадающий символ в метке ребра (u, v), и, вероятно, обход идет дальше вниз по дереву из-за возможного

совпадения между символами ветвления следующих пройденных ребер и символами P. В конце концов достигается или выбирается лист l, спускающийся из узла v (этап *1*), и корректно вычисляется значение ℓ, ведь все листья, выходящие из v, имеют общие символы $|s[v]| > \ell$. Это означает наличие общих символов у ℓ и P (этап *2*). Восходящий обход от листа l на этапе *3* достигает ребра (u, v), для которого $|s[u]| < \ell < |s[v]|$. Дальше несовпадающие символы $s_l[\ell + 1]$ и $P[\ell + 1]$ позволяют корректно определить лексикографическую позицию P. Она располагается либо слева ($P[\ell + 1] < s_l[\ell + 1]$), либо справа ($s_l[\ell + 1] < P[\ell + 1]$) от выходящих из v листьев.

Таким образом, мы предоставили математическое обоснование корректности слепого поиска, который обеспечивает превосходную производительность с точки зрения пространства, времени и ввода-вывода.

Теорема 9.8. *Предположим, что patricia-дерево, индексирующее словарь из n строк, может храниться во внутренней памяти компьютера, тогда как словарь настолько велик, что его приходится хранить на диске. Patricia-дерево занимает $\Theta(n)$ места, следовательно, под каждую индексированную строку выделяется постоянное место (независимо от их общей длины).*

Слепой поиск лексикографической позиции шаблона $P[1, p]$ среди строк словаря занимает время $O(p + \log \sigma)$ и не требует операций ввода-вывода для обхода префиксного дерева (этапы 1 и 3). Кроме того, необходимы также время $O(p)$ и $O(p/B)$ операций ввода-вывода для извлечения и сравнения одной строки, идентифицированной слепым поиском (этап 2).

При поиске P и $P\#$ слепой поиск определяет диапазон индексированных строк с префиксом P, если таковые имеются, в пределах того же времени и границ ввода-вывода.

Эта теорема фактически утверждает, что если $p \leqslant B$ и $n < M$, что на практике соответствует вполне разумным условиям, то префиксный поиск P в словаре $\mathcal{D}$ осуществляется всего за один ввод-вывод. При обязательном сжатии словаря весьма примечательна производительность, достигаемая сочетанием patricia-дерева со схемой LPFC, как указано в следующей теореме.

Теорема 9.9. *Двухуровневой индексной структуре данных, состоящей из patricia-дерева в качестве индекса во внутренней памяти (верхний уровень) и LPFC в качестве сжатого хранилища строк словаря на диске (нижний уровень), требуется пространство $O(n)$ в памяти и пространство $O((1 + \varepsilon)FC(\mathcal{D}))$ на диске, где ε — параметр, устанавливаемый LPFC и контролирующий компромисс между количеством операций ввода-вывода и занимаемым пространством.*

Префиксный поиск для шаблона $P[1, p]$ требует $O(p/B + |s|/B\varepsilon)$ операций ввода-вывода, где s — интересующая нас строка, определенная на этапе 1 слепого поиска. Извлечение префиксных строк происходит за $O\left(\frac{(1+\varepsilon)FC(\mathcal{D}_{occ})}{B}\right)$ операций ввода-вывода, где $\mathcal{D}_{occ} \subseteq \mathcal{D}$ — набор возвращенных строк.

Доказательство. Производительность ввода-вывода основана на следующем наблюдении: вычисление значения ℓ на этапе *2* слепого поиска требует декодирования выбранной строки s из ее представления LPFC (см. теорему 9.4), и для этого требуется $O(|s|/\varepsilon B)$ операций ввода-вывода. ■

Когда $n = \Omega(M)$, индексировать во внутренней памяти patricia-дерева весь словарь невозможно, поэтому приходится прибегать к сегментированию по его строкам и, таким образом, индексировать в дереве только первую строку каждой страницы диска. Если $N/B = O(M)$, наше дерево может индексировать во внутренней памяти все выбранные строки и таким образом поддерживать префиксный поиск для P в пределах, указанных в теореме 9.9, добавляя всего один ввод-вывод из-за сканирования корзины (то есть страницы диска), содержащей лексикографическую позицию P. Предыдущее условие можно переписать как $N = O(MB)$, что с практической точки зрения выглядит довольно разумно, учитывая текущие значения $M \approx 32$ Гбайт и $B \approx 32$ Кбайт, дающие произведение MB порядок сотен терабайт.

9.6. Управление огромными словарями∞

А как быть, если $N = \Omega(MB)$? В этом случае patricia-дерево слишком велико для размещения во внутренней памяти компьютера. Наверное, его можно сохранить на диске, не уделяя особого внимания расположению отдельных узлов. Но, к сожалению, в этом случае два обхода от корня к листьям, выполняемые на этапах *1* и *3* слепого поиска по шаблону, потребуют $\Theta(p)$ операций ввода-вывода. В качестве альтернативы можно представить для patricia-дерева «упаковку» на страницах диска, которая минимизирует необходимые для таких обходов операции ввода-вывода. В этом случае суть будет заключаться в постепенном наращивании корневой страницы путем многократного добавления туда еще не упакованных узлов. Выбор этих узлов определяется различными критериями, зависящими от вероятности доступа к узлу или его глубины. Как только на корневой странице оказывается B узлов, она записывается на диск и алгоритм рекурсивно выкладывает остальную часть дерева. Как ни удивительно, такая упаковка далека от оптимальной — превышает ее в $\Omega\left(\frac{\log B}{\log\log B}\right)$ раз, но можно уверенно утверждать, что она находится в пределах $O(\log B)$ раз от оптимальной [1].

Я покажу вам два оптимальных подхода к префиксному поиску по словарям *огромного размера*. Первое решение основано на такой структуре данных, как *строковое B-дерево* [5]. Именно так называется B-дерево, в котором таблица маршрутизации каждого узла является patricia-деревом. Основу второго решения составляет *корректное расположение* patricia-дерева на диске. Рассмотрим оба этих подхода.

9.6.1. Строковое B-дерево

Основная идея этого подхода состоит в разделении большого patricia-дерева на набор меньших деревьев, каждое из которых помещается на одну страницу диска. Затем все они связываются с помощью B-дерева. В этом разделе я дам конструктивное определение строкового B-дерева. Подробную же информацию об этой структуре и операциях с ней вы найдете в статье [5].

Строки словаря $\mathcal{D}$ хранятся на диске непрерывно и упорядочены в алфавитном порядке. Обозначим $\mathcal{D}^0 = \mathcal{D}$ указатели на эти строки, хранящиеся на уровне листьев строкового B-дерева. Эти указатели разделены на фрагменты одинакового размера $\mathcal{D}_1^0, \ldots, \mathcal{D}_m^0$, каждый из которых включает $\Theta(B)$ строк произвольной длины. Таким образом, $m = N/B$. После этого каждый фрагмент $\mathcal{D}_i^0$ можно индексировать с помощью patricia-дерева, которое помещается на одну страницу диска, и встроить его в лист B-дерева. Чтобы найти среди этого набора листовых узлов P, возьмем из каждого раздела $\mathcal{D}_i^0$ его первую и последнюю (лексикографически) строки, s_i^f и s_i^l, тем самым задав набор $\mathcal{D}^1 = \left\{s_1^f, s_1^l, \ldots, s_m^f, s_m^l\right\}$.

Напомню, что поиск префикса для P сводится к лексикографическому поиску должным образом определенного шаблона Q: $Q = P$ или $Q = P\#$. При поиске Q в наборе $\mathcal{D}^1$ возможны три случая.

1. Шаблон Q стоит перед первой или после последней строки $\mathcal{D}^1$, потому что $Q < s_1^f$ или $Q > s_m^l$.
2. Шаблон Q оказывается между двумя фрагментами, скажем $\mathcal{D}_i^0$ и $\mathcal{D}_{i+1}^0$, потому что $s_i^l < Q < s_{i+1}^f$. Тут лексикографическая позиция шаблона Q в целом словаре D находится между этими двумя соседними фрагментами.
3. Шаблон Q оказывается среди строк некоторого фрагмента $\mathcal{D}_i^0$, потому что $s_i^f \leqslant Q \leqslant s_i^l$. Поэтому поиск продолжается в patricia-дереве, которое индексирует $\mathcal{D}_i^0$.

Чтобы понять, какой из трех случаев имеет место, нужно эффективно найти в $\mathcal{D}^1$ лексикографическую позицию шаблона Q. Если набор $\mathcal{D}^1$ мал настолько, что допускает размещение в памяти, на нем можно построить patricia-дерево, и все будет готово (см. теорему 9.8). В противном случае процесс разбиения $\mathcal{D}^1$ следует повторить, чтобы получить меньшее множество $\mathcal{D}^2$. В нем мы снова выберем две строки для каждого B согласно равенству $\left|\mathcal{D}^2\right| = \frac{2\left|\mathcal{D}^1\right|}{B}$. Процесс разбиения продолжится k раз, пока мы не получим $\left|\mathcal{D}^k\right| = O(B)$ и patricia-дерево, построенное для k-го подмножества, не поместится на одну страницу диска[1].

[1] На самом деле остановиться можно уже при $\left|\mathcal{D}^k\right| = O(M)$, но для получения стандартной структуры B-дерева лучше подходит описанный вариант.

Обратите внимание на то, что при разбиении на каждой странице диска оказывается четное количество строк $\mathcal{D}^0, \ldots, \mathcal{D}^k$ и с каждой парой $\left\langle s_i^f, s_i^l \right\rangle$ связывается указатель на блок строк, который они разграничивают на нижнем уровне процесса разбиения. В качестве конечного результата мы получим B-дерево над указателями строк. *Кратность* этого дерева составляет $\Theta(B)$, поскольку в каждом узле мы индексируем $\Theta(B)$ строк. После этого узлы строкового B-дерева сохраняются на диске, поэтому его высота $k = \Theta(\log_B n)$. Пример такого дерева из 15 строк высотой 2 демонстрируется на рис. 9.9.

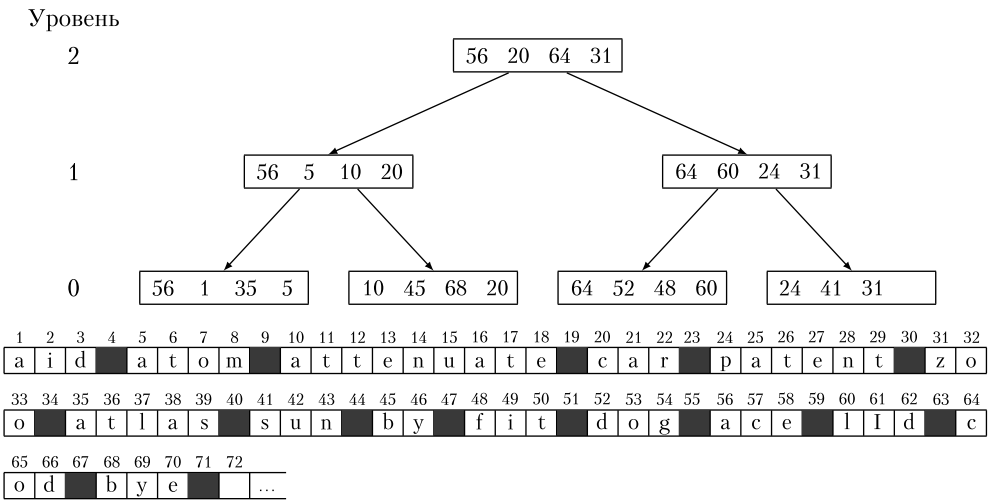


Рис. 9.9. Пример строкового B-дерева, построенного на словаре из 15 строк, хранящихся на диске в несортированном порядке. С помощью логических указателей строки были сохранены в листьях B-дерева в алфавитном порядке

Обратите внимание: сортировка строк на диске улучшит показатели ввода-вывода при их сканировании, более того, все наши теоремы действительны для упорядоченного словаря $\mathcal{D}$ на диске. Однако на рис. 9.9 строки не отсортированы, чтобы показать гибкость этой структуры данных при операциях обновления из-за вставок и удалений строк.

(Префиксный) поиск шаблона Q в строковом B-дереве представляет собой обычный обход B-дерева, при котором в каждом узле выполняется лексикографический поиск Q по соответствующему этому узлу patricia-дереву. При этом возможны три упоминавшихся ранее случая.

- Случай 1 может произойти только в корневом узле, лексикографическая позиция Q обнаруживается в начале или в конце словаря $\mathcal{D}$, поэтому поиск по B-дереву останавливается.

- В случае 2 лексикографическая позиция Q в словаре $\mathcal{D}$ обнаруживается между строками s_i^l и s_{i+1}^f, соседствующими в словаре $\mathcal{D}$, поэтому поиск по строковому B-дереву останавливается.
- Случай 3 подразумевает, что мы должны следовать указателю узла, связанному с идентифицированным фрагментом, ограниченным на нижнем уровне парой строк $\langle s_i^f, s_i^l \rangle$.

Определенная ранее структура данных имеет довольно хорошую сложность ввода-вывода. У B-дерева с кратностью $\Theta(B)$ есть $\Theta(\log_B n)$ уровней, поэтому поиск проходит по $\Theta(\log_B n)$ узлам. Поскольку в каждом узле нужно загрузить соответствующую ему страницу в память и выполнить слепой поиск шаблона Q по его patricia-дереву (во внутренней памяти), приходится выполнять $O(1 + p/B)$ операций ввода-вывода. Таким образом, общий префиксный поиск шаблона Q в словаре $\mathcal{D}$ вызывает $O(p/B \log_B n)$ операций ввода-вывода.

Теорема 9.10. *Префиксный поиск шаблона $P[1, p]$ в строковом B-дереве на базе состоящего из n строк словаря $\mathcal{D}$ требует $O\left(\frac{p}{B}\log_B n + \frac{N_{occ}}{B}\right)$ операций ввода-вывода, где N_{occ} — общая длина строк словаря с префиксом P. Строковое B-дерево занимает $O(n/B)$ страниц диска, а строки хранятся на диске в отсортированном, несжатом, смежном виде.*

Это хороший, но пока *не оптимальный* результат. Достичь оптимальности мешает *повторное сканирование шаблона*: при каждом слепом поиске шаблон Q посимвольно сравнивается с одной из строк, хранящихся в текущем посещаемом узле B-дерева. По мере спуска по B-дереву можно извлечь выгоду из того факта, что часть символов шаблона Q уже подвергалась сравнению. Это позволяет избежать их повторного сканирования при новых вычислениях LCP. Фактически в этом случае сравнить Q со строкой словаря можно, начиная с $(c + 1)$-го символа. Это позволит нам достичь оптимального ввода-вывода, но, к сожалению, строки должны храниться в несжатом виде для обеспечения эффективного доступа к этому $(c + 1)$-му символу. Прорабатывая все детали (см. [5]), можно показать следующее.

Теорема 9.11. *Префиксный поиск шаблона $P[1, p]$ в строковом B-дереве, построенном по словарю $\mathcal{D}$ из n строк, требует $O\left(\frac{p}{B} + \log_B n + \frac{N_{occ}}{B}\right)$ операций ввода-вывода, где N_{occ} — общая длина строк словаря, которые начинаются с P. Строковое B-дерево занимает $O(n/B)$ страниц диска, а строки хранятся упорядоченными, несжатыми и смежными.*

Для хранения на диске сжатых строк можно принять субоптимальный подход теоремы 9.10 и подключить к словарю $\mathcal{D}$ схему LPFC. Если все-таки хочется сохранить оптимальность ввода-вывода, придется принимать более сложное решение, например описанное в [2], [6].

9.6.2. Упаковка деревьев на диске

В случае несбалансированных деревьев хорошая компоновка на страницах диска (размером B) имеет настолько большое значение, что про нее обязательно помнить при проектировании решений для больших деревьев. В то время как сбалансированные деревья при отображении на диск дают сложность $O(\log B)$ (упаковка на страницу сбалансированных поддеревьев, состоящих из B узлов), отображение несбалансированных деревьев растет неравномерно. В крайней форме оно приближается к линейно сформированному дереву (то есть дереву, формой напоминающему путь) и к экономии в $\Theta(B)$ раз по сравнению с наивной структурой памяти.

В литературе эта задача известна как задача *упаковки дерева*. Ее цель — найти распределение узлов дерева по страницам диска, которое минимизирует число операций ввода-вывода при обходе от корня к листьям. Минимизация может касаться как общего числа загруженных во внутреннюю память страниц (то есть количества операций ввода-вывода), так и числа отдельных посещенных страниц (то есть размера рабочего набора, который может быть меньше, если некоторые страницы посещаются повторно). При таком подходе мы моделируем две крайние ситуации: случай одностраничной внутренней памяти (небольшого буфера) или случай неограниченной внутренней памяти (неограниченного буфера). Удивительно, но оптимальное решение задачи упаковки дерева *не зависит* от доступного размера буфера, так как после минимизации количества операций ввода-вывода или рабочего набора ни одна страница диска не будет посещаться дважды. Более того, оптимальное решение демонстрирует хорошую *разложимость*: оптимальная упаковка дерева образует дерево страниц диска. Эти два факта позволяют сосредоточиться на проблеме минимизации ввода-вывода и разработать рекурсивные решения для оптимального разложения дерева по дисковым страницам размером B.

Далее я покажу два решения возрастающей сложности, которые рассматривают два сценария. В первом случае целью будет минимизация *максимального количества операций ввода-вывода* при обходе вниз от корня к листьям. Во втором случае минимизации будет подвергаться *среднее количество* операций ввода-вывода, исходя из распределения доступа к листьям дерева и, соответственно, возможным вариантам обхода. В обоих случаях предполагается, что B известно. В литературе есть описания решений этой задачи в условиях независимости от кэша, но они слишком сложны для этой книги (подробности можно найти в [1], [7]).

Алгоритм минимакс. Это решение [3] использует жадный подход и двигается снизу вверх по дереву, которое нужно упаковать. Оно минимизирует максимальное количество операций ввода-вывода, выполняемых во время нисходящего обхода от корня дерева. Дерево предполагается бинарным, что не является ограничением для применения patricia-деревьев, ведь всегда можно закодировать символы алфавита двоичными строками. Каждому листу назначается собственная страница диска, а ее высота устанавливается равной 1. Двигаясь вверх, алгоритм 9.1 применяется к каждому узлу, пока не будет достигнут корень дерева.

Окончательный вариант упаковки далеко не всегда обеспечивает хорошее заполнение страниц, но в реальных ситуациях эту проблему можно смягчить разными способами, например таким.

1. После закрытия страницы ее дочерние страницы от самой маленькой до самой большой сканируются, чтобы посмотреть, нельзя ли их объединить с родительской.
2. Проектируются логические страницы диска, которые упаковываются в одну физическую страницу диска. При размещении логических страниц на диске границы физических страниц по возможности игнорируются.

Можно показать, что алгоритм 9.1 обеспечивает такую дисковую упаковку двоичного дерева из n узлов высотой H, что каждый путь от корня к листьям проходит через $O\left(\frac{H}{\sqrt{B}}+\log_B n\right)$ страниц.

Алгоритм 9.1. Алгоритм минимакс для бинарных деревьев (обобщенный вид)

Пусть u — посещаемый в текущий момент узел;

```
if у обоих потомков u одинаковая высота страницы d then
    if общее количество узлов для обеих дочерних страниц < B then
        Сливаем две дисковые страницы и добавляем u;
        Присваиваем высоте новой страницы значение d (как у ее потомков);
    else
        Заканчиваем страницы потомков u;
        Создаем для u новую страницу и присваиваем ей высоту d + 1;
    end if
else
    Завершаем страницу потомка u с меньшей высотой;
    По возможности сливаем страницу другого потомка с u и оставляем ее высоту неизменной;
    В противном случае создаем для u новую страницу высотой d + 1 и закрываем страницы обоих потомков;
    end if
```

Упаковка деревьев с учетом распределения. Предполагается, что нам известно распределение доступа к листьям patricia-дерева. Часто оно смещено в сторону листьев, к которым обращаются чаще, чем к другим, и по этой причине алгоритм минимакс может быть неэффективным. Поэтому рассмотрим алгоритм, базирующийся на схеме динамического программирования, которая минимизирует

ожидаемое количество операций ввода-вывода при любом обходе от корня к листьям в patricia-дереве [7].

Обозначим оптимальную упаковку дерева τ. Страница диска, на которую отображается узел u, будет обозначаться $\tau(u)$. Вероятностью доступа к листу f обозначим $w(f)$. Распределение по всем узлам дерева получается простым суммированием вероятностей доступа к расположенным ниже листьям. Можно предположить, что корень дерева r всегда отображается на фиксированную страницу $\tau(r) = R$. Рассмотрим V — множество узлов дерева, которые происходят от узлов R, но сами в них не находятся. Мы видим, что оптимальная упаковка τ порождает дерево страниц диска, следовательно, если упаковка τ оптимальна для текущего дерева T, то она будет оптимальной и для всех поддеревьев T_v с корнем в $v \in V$.

Этот результат позволяет сформулировать рекурсивное вычисление для τ, которое сначала определяет узлы, располагающиеся в R, а затем проделывает то же самое со всеми поддеревьями T_v. Динамическое программирование обеспечивает эффективную реализацию идеи, основанной на следующем определении: *i-ограниченной* называется упаковка дерева T, страница R которой содержит ровно i узлов, причем $i \leqslant B$. Теперь в оптимальной упаковке τ корневая страница R будет содержать корень r дерева T, $i*$ узлов из левого поддерева $T_{\text{left}(r)}$ и $(B - i* - 1)$ узлов из правого поддерева $T_{\text{right}(r)}$ для некоторого $i* < B$. Фактически τ является как оптимальной $i*$-ограниченной упаковкой для $T_{\text{left}(r)}$, так и оптимальной $(B - i* - 1)$-ограниченной упаковкой для $T_{\text{right}(r)}$.

Это свойство лежит в основе правила динамического программирования, согласно которому для общего узла v и целого числа $i \leqslant B$ вычисляется стоимость $A[v, i]$ оптимальной i-ограниченной упаковки поддерева T_v. Авторы статьи [7] показали, что $A[v, i]$ можно вычислить как вероятность доступа $w(v)$ к узлу v, которая учитывает, во-первых, ожидаемые операции ввода-вывода, возникающие при посещении страницы v, то есть $1w(v)$, во-вторых, минимум из следующих трех величин:

- $A[\texttt{left}(v), i - 1] + w(\texttt{right}(v)) + A[\texttt{right}(v), B]$ учитывает несбалансированный случай, когда i-ограниченная упаковка для v получается путем сохранения $i - 1$ узла из $T_{\text{left}(r)}$ на странице v и $\texttt{right}(v)$ — на другой странице;
- $A[\texttt{right}(v), i - 1] + w(\texttt{left}(v)) + A[\texttt{left}(v), B]$ симметрична предыдущему правилу, но специализируется на правом потомке v;
- $\min_{1 \leqslant j < i-1}\{A[\texttt{left}(v), j] + A[\texttt{right}(v), i - j - 1]\}$ учитывает случай, когда j узлов из $T_{\text{left}(r)}$ и $i - j - 1$ узлов из $T_{\text{right}(r)}$ хранятся на странице v для формирования оптимальной i-ограниченной упаковки поддерева T_v. (Частный случай $i = 1$ задается формулой $A[v, 1] = w(v) + A[\texttt{left}(v), B] + A[\texttt{right}(v), B]$.)

Можно показать, что оптимальную упаковку дерева алгоритм 9.2 вычисляет за время $O(nB^2)$, требуя пространство $O(nB)$. Оптимальность в этом случае относится к ожидаемому числу операций ввода-вывода, возникающих при любом обходе упакованного бинарного дерева от корня к листьям. Упаковка отображает бинарное дерево не более чем на $2\lfloor n/B \rfloor$ страниц диска.

Алгоритм 9.2. Упаковка деревьев на диске с учетом распределения

1: Инициализируем $A[v, i] = w(v)$ для всех листьев v и целых чисел $i \leqslant B$;
2: **while** существует немаркированный узел v **do**
3: маркируем v;
4: обновляем $A[v, 1] = w(v) + A[\texttt{left}(v), B] + A[\texttt{right}(v), B]$;
5: **for** $i = 2$ to B **do**
6: обновляем $A[v, i]$ согласно описанному в тексте правилу динамического программирования
7: **end for**
8: **end while**

Список литературы

1. *Alstrup S., Bender M. A., Demaine E. D. et al.* Efficient tree layout in a multilevel memory hierarchy, 2003 // Personal communication: corrected version of a paper that appeared in Proceedings of the 10th European Symposium on Algorithms (ESA), 2002.
2. *Bender M. A., Farach-Colton M., Kuszmaul B. C.* Cache-oblivious string B-trees // Proceedings of the 25th ACM Symposium on Principles of Database Systems, 223–242, 2006.
3. *Clark D. R., Munro J. I.* Efficient suffix trees on secondary storage // Proceedings of the 7th ACM — SIAM Symposium on Discrete Algorithms (SODA), 383–391, 1996.
4. *Demaine E. D., Jones T., Pătraşcu M.* Interpolation search for non-independent data // Proceedings of the 15th ACM — SIAM Symposium on Discrete Algorithms, 529–530, 2004.
5. *Ferragina P., Grossi R.* The String B-tree: A new data structure for string search in external memory and its applications // Journal of the ACM, 46 (2): 236–280, 1999.
6. *Ferragina P., Venturini R.* Compressed cache-oblivious String B-tree // ACM Transactions on Algorithms, 12 (4): 52:1–52:17, 2016.
7. *Gil J., Itai A.* How to pack trees // Journal of Algorithms, 32 (2): 108–132, 1999.
8. *Luby M., Rackoff C.* How to construct pseudorandom permutations from pseudorandom functions // SIAM Journal on Computing, 17: 373–386, 1988.

Глава 10
ПОИСК ПО ПОДСТРОКЕ

В этой главе будет рассматриваться решение задачи, известной как *полнотекстовый поиск* или *поиск подстрок*.

> **Задача.** Выполнить предварительную обработку текстовой строки $T[1, n]$, составленной из букв алфавита размером σ, чтобы получить возможность эффективно извлекать или просто подсчитывать все позиции, в которых шаблон запроса $P[1, p]$ встречается как подстрока текста T.

Очевидно, что решить такую задачу можно прямым сравнением P с каждой подстрокой T, что в худшем случае займет время $O(np)$. Не менее очевидно то, что для массивных текстовых коллекций, к которым выполняется большое количество запросов, например геномных баз данных или поисковых систем, такой подход будет неприемлемо медленным. Поэтому до начала поиска необходимо предварительно обработать текст T, построив структуру *индексных* данных. Потребуются ресурсы для настройки этой структуры, но они будут амортизироваться в ходе последующих поисков по шаблону, что приводит к квазистатической среде, в которой T меняется очень редко.

Я покажу два основных подхода к поиску подстрок, один на базе массивов, другой на базе деревьев, которые имитируют решение, найденное для поиска префиксов. В обоих подходах будут использоваться две фундаментальные структуры данных: *массив суффиксов* (SA) и *суффиксное дерево* (ST). Я опишу их очень подробно, поскольку их применение выходит далеко за рамки полнотекстового поиска.

10.1. Обозначения и терминология

Предполагается, что текст T заканчивается специальным символом $T[n] = \$$, который меньше любого другого символа алфавита. Это гарантирует, что не существует суффикса, который был бы префиксом другого суффикса. Переменная $suff_i$ соответствует i-му суффиксу текста T, а именно подстроке $T[i, n]$. Большое значение имеет следующее наблюдение.

При $P = T[i, i + p - 1]$ шаблон встречается в текстовой позиции i, таким образом, можно утверждать, что P является префиксом i-го текстового суффикса, то есть строки $suff_i$.

Например, если P = `siss`, а T = `mississippi$`, то на рис. 10.1 P встречается в текстовой позиции 4 и предваряет суффикс $suff_4 = T[4, 12]$ = `sissippi$`. Для простоты изложения и по историческим причинам этот текст и далее будет использоваться в качестве рабочего примера. Обратите внимание, что текст может состоять из произвольной последовательности символов, следовательно, не обязательно быть одним словом.

SUF(*T*)	Позиции
mississippi$	1
ississippi$	2
ssissippi$	3
sissippi$	4
issippi$	5
ssippi$	6
sippi$	7
ippi$	8
ppi$	9
pi$	10
i$	11
$	12

Отсортированный *SUF*(*T*)	*SA*	lsp
$	12	0
i$	11	1
ippi$	8	1
issippi$	5	4
ississippi$	2	0
mississippi$	1	0
pi$	10	1
ppi$	9	0
sippi$	7	2
sissippi$	4	1
ssippi$	6	3
ssissippi$	3	-

Рис. 10.1. Набор всех текстовых суффиксов *SUF*(*T*) и два массива, *SA* и lcp, для строки *T* = mississippi$

Учитывая это наблюдение, можно сформировать из всех текстовых суффиксов словарь *SUF*(*T*) и заявить, что *поиск P как подстроки T сводится к поиску P как префикса некоторой строки в SUF*(*T*). Кроме того, поскольку между текстовыми суффиксами с префиксом P и вхождениями шаблона в T существует биективное соответствие, то:

- суффиксы с префиксом P встречаются последовательно в лексикографически отсортированном словаре *SUF*(*T*);
- лексикографическая позиция P в словаре *SUF*(*T*) непосредственно предшествует блоку суффиксов, префиксом которых является P.

Внимательный читатель мог узнать свойства, которые в главе 9 разворачивались для эффективной поддержки префиксного поиска. И действительно, известные из литературы решения для эффективного поиска подстроки опираются или на

массивы (массив суффиксов), или на префиксные деревья (суффиксное дерево). Поэтому именно эти структуры данных первым делом приходят в голову, когда речь заходит о поиске по шаблону. Сложность заключается в том, как реализовать их построение и отображение на диск для достижения эффективной производительности ввода-вывода. Именно эти вопросы будут рассматриваться в следующих разделах.

10.2. Массив суффиксов

Массив суффиксов для текста T — это массив указателей на все текстовые суффиксы, упорядоченные лексикографически [14]. Для массива суффиксов, построенного над словарем T, используем обозначение $SA(T)$ или просто SA, если индексированный текст ясен из контекста. Из-за лексикографического упорядочения $SA[i]$ является i-м наименьшим текстовым суффиксом, поэтому $suff_{SA[1]} < suff_{SA[2]} < ... < suff_{SA[n]}$, где знак < показывает лексикографический порядок между строками. Для экономии места каждый суффикс будет представлен своей начальной позицией в T, то есть целым числом, поэтому массив SA состоит из n целых чисел в диапазоне $[1, n]$ и занимает $\Theta(n \log n)$ бит.

Еще одна полезная концепция — *самый длинный общий префикс* между двумя последовательными суффиксами $suff_{SA[i]}$ и $suff_{SA[i+1]}$, с которым я вас подробно познакомил в предыдущей главе. Его длина хранится в записи `lcp`$[i]$ массива из $n - 1$ записей, каждая из которых содержит значения меньше n. На рис. 10.1 представлен работающий пример с массивами `lcp` и SA при входном тексте $T =$ `mississippi$`. Для построения массива `lcp` существует оптимальный неочевидный алгоритм линейного времени, который будет подробно описан в подразделе 10.2.3. Этот массив крайне полезен при разработке эффективных/оптимальных алгоритмов для различных задач поиска и добычи строковых данных. Некоторые из этих алгоритмов будут рассматриваться в разделе 10.4.

10.2.1. Поиск подстроки

Надеюсь, вы заметили, что задачу поиска подстроки можно свести к поиску префикса по строковому словарю $SUF(T)$ и решить путем бинарного поиска P по массиву упорядоченных лексикографически текстовых суффиксов, а именно $SA(T)$. Классический бинарный поиск, приспособленный для сравнения строк, демонстрирует псевдокод алгоритма 10.1. Здесь происходит $O(\log n)$ сравнений строк, каждое из которых занимает время $O(p)$ в худшем случае. Таким образом, мы доказали следующую лемму.

Лемма 10.1. *Для подсчета вхождений шаблона $P[1, p]$ в текст $T[1, n]$, снабженный массивом суффиксов, в худшем случае потребуются время $O(p \log n)$ и $O(\log n)$ обращений к памяти. Извлечение позиций этих occ вхождений занимает дополнительное время $O(occ)$. Общее требуемое пространство составляет $\Theta(n (\log n + \log \sigma))$ бит, где первый член учитывает массив суффиксов, а второй член — текст.*

Алгоритм 10.1. SUBSTRINGSEARCH(P, T, SA)

```
1:  L = 1, R = n;
2:  while L ≠ R do
3:     M = ⌊(L + R)/2⌋;
4:     if strncmp(P, suff_M, p) > 0 then
5:        L = M + 1;
6:     else
7:     R = M;
8:     end if
9:  end while
10: if strncmp(P, suff_L, p) = 0 then
11:    return L;
12: else
13:    return – 1;
14: end if
```

Поскольку каждое сравнение шаблона и суффикса требует $O(p/B)$ операций ввода-вывода, то для подсчета вхождений шаблона $P[1, p]$ в словарь T понадобится $O(p/B \log n)$ операций ввода-вывода, а извлечение позиций occ вхождений потребует $O(occ/B)$ дополнительных операций ввода-вывода.

Рабочий пример для алгоритма 10.1, подчеркивающий интересное свойство: сравнение P и $suff_M$ не обязательно должно начинаться с их первого символа, демонстрирует рис. 10.2. Можно воспользоваться лексикографической сортировкой суффиксов и пропустить сравнения, которые уже были выполнены на предыдущих итерациях. В этом нам помогут три массива.

- Массив `lcp`$[1, n-1]$ хранит в `lcp`$[i]$ длину самого длинного общего префикса между суффиксами $SA[i]$ и $SA[i+1]$. Обозначим $lcp(suff_{SA[i]}, suff_{SA[i+1]})$ функцию, которая вычисляет эту длину.
- Два других массива, $Llcp[1, n-1]$ и $Rlcp[1, n-1]$, определены для каждой тройки (L, M, R), которая может появиться во внутреннем цикле бинарного поиска в диапазоне $[1, n]$. Мы задаем $Llcp[M] = lcp(suff_{SA[L]}, suff_{SA[M]})$ и $Rlcp[M] = lcp(suff_{SA[M]}, suff_{SA[R]})$. То есть $Llcp[M]$ учитывает длину самого длинного префикса, общего для самого левого суффикса $suff_{SA[L]}$ и среднего суффикса $suff_{SA[M]}$ диапазона, который в данный момент исследуется бинарным поиском. А $Rlcp[M]$ учитывает длину самого длинного префикса, общего для самого правого суффикса $suff_{SA[R]}$ и среднего суффикса $suff_{SA[M]}$ этого диапазона.

Обратите внимание на то, что каждая тройка (L, M, R) однозначно идентифицируется своей средней точкой M, поскольку процедура двоичного поиска фактически

определяет иерархическое разбиение массива *SA* на все меньшие и меньшие подмассивы, разделенные (*L*, *R*) и, таким образом, центрированные на *M*. Следовательно, всего мы имеем $\Theta(n)$ троек, и эти три массива в общей сложности занимают пространство $\Theta(n)$.

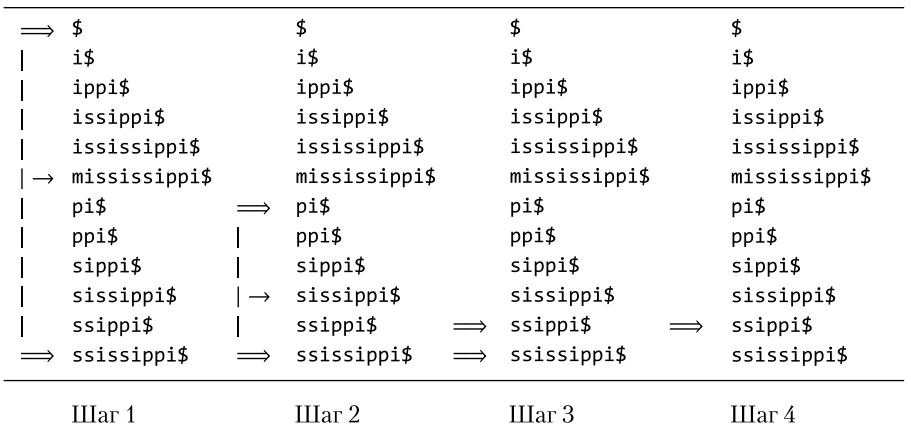


Рис. 10.2. Двоичный поиск для определения лексикографической позиции шаблона *P* = ssi среди суффиксов текстовой строки mississippi$. Двойные стрелки обозначают указатели *L* и *R*, а одинарная стрелка — указатель *M*

Массивы *Llcp* и *Rlcp* можно построить за время $O(n)$, используя два разных подхода. В первом случае мы эксплуатируем наблюдение, показывающее, что длина самого длинного общего префикса между любыми двумя суффиксами $suff_{SA[i]}$ и $suff_{SA[j]}$, где $i < j$, может быть вычислена как минимум из диапазона значений $\texttt{lcp}[i, j] := \min_{k=i,\ldots,j-1} \texttt{lcp}(suff_{SA[k]}, suff_{SA[k+1]}) = \min_{k=i,\ldots,j-1} \texttt{lcp}[k]$. Благодаря ассоциативности оператора min вычисление можно разделить следующим образом: $\texttt{lcp}[i, j] = \min\{\texttt{lcp}[i, k], \texttt{lcp}[k, j]\}$, где k — любой индекс в диапазоне $[i, j]$. Мы можем установить $\texttt{lcp}[L, R] = \min\{\texttt{lcp}[L, M], \texttt{lcp}[M, R]\}$. Это означает, что массивы *Llcp* и *Rlcp* можно вычислить путем обхода троек (*L*, *M*, *R*) снизу вверх за время $O(n)$. Второй подход заключается в вычислении $\texttt{lcp}[i, j]$ на лету с помощью структуры данных, выполняющей запрос минимального диапазона. Строится такая структура на основе массива `lcp`, а алгоритм ее построения вы найдете в подразделе 10.4.2. Оба этих подхода имеют пространственную и временную сложность $O(n)$ и, таким образом, асимптотически оптимальны.

Осталось показать, как ускорить бинарный поиск с помощью трех массивов, *SA*, *Llcp* и *Rlcp*. Рассмотрим шаг бинарного поиска в подмассиве $SA[L, R]$, и пусть *M* будет средней точкой этого диапазона (следовательно, $M = \lfloor (L + R) / 2 \rfloor$). Лексикографическое сравнение *P* и $suff_{SA[M]}$, выполняемое на шаге 4 алгоритма 10.1, выбирает следующий диапазон поиска между $SA[L, M]$ и $SA[M, R]$. Там сравнение строк начиналось с первого символа *P* и $suff_{SA[M]}$, а здесь мы сравниваем их, пропуская некоторые символы, с помощью предыдущих шагов бинарного поиска.

Удивительно, но это можно сделать. В дополнение к *Llcp* и *Rlcp* нужно знать значения $l = lcp(P, suff_{SA[L]})$ и $r = lcp(P, suff_{SA[R]})$, которые обозначают количество символов, разделяемых шаблоном P со строками в крайних точках диапазона, в настоящее время исследуемого бинарным поиском. Первоначально, то есть при $L = 1$ и $R = n$, эти два значения можно вычислить за время $O(p)$ посимвольным сравнением задействованных строк. В общем виде предполагается, что l и r определены индуктивно. Далее показывается, как шаг бинарного поиска может эффективно пересчитать их перед переходом либо к $SA[L, M]$, либо к $SA[M, R]$[1].

Фактически мы знаем, что P располагается между $suff_{SA[L]}$ и $suff_{SA[R]}$ и поэтому имеет общие `lcp`$[L, R]$ символов со всеми суффиксами в диапазоне $SA[L, R]$. Эти символы должны быть общими у любой строки в этом диапазоне, учитывая, что они лексикографически отсортированы. Поэтому мы можем заключить, что l и r больше или равны `lcp`$[L, R]$, а количество символов m, которые шаблон P разделяет с $suff_{SA[M]}$, также больше или равно `lcp`$[L, R]$. Это последнее неравенство позволяет сравнить P с $suff_{SA[M]}$, начиная с их (`lcp`$[L, R] + 1$)-го символа.

Но на самом деле есть и лучший вариант, ведь l и r известны и могут быть значительно больше, чем `lcp`$[L, R]$. Это означает, что в предыдущих сравнениях фигурировало больше символов шаблона P, поэтому они уже известны. Мы различаем три основных случая, предполагая, что $l \geqslant r$ (случай $l < r$ симметричен), и стремимся избежать сканирования уже известных символов P (это символы в $P[1, l]$). Алгоритм устроен таким образом, что порядок между P и $suff_{SA[M]}$ можно определить либо сравнением символов в $P[l + 1, n]$, либо сравнением значений l и $Llcp[M]$ (которые дают информацию о $P[1, l]$).

- Если $l < Llcp[M]$, то P больше, чем $suff_{SA[M]}$, и мы можем установить $l = m$. По индукции $P > suff_{SA[L]}$, и их несовпадающий символ занимает позицию $l + 1$. Согласно определению $Llcp[M]$ и гипотезе (то есть $l < Llcp[M]$), у $suff_{SA[L]}$ и $suff_{SA[M]}$ более l общих символов. Так что $suff_{SA[M]}[l + 1] = suff_{SA[L]}[l + 1]$ и несовпадение между P и этими двумя суффиксами одинаково, следовательно, их сравнение снова дает тот же ответ, то есть $P > suff_{SA[M]}$. Поиск может продолжаться в поддиапазоне $SA[M, R]$, причем нет необходимости сравнивать символы.
- Случай $l > Llcp[M]$ аналогичен предыдущему. Можно сделать вывод, что P меньше, чем $suff_{SA[M]}$, и что $m = Llcp[M]$. Поиск продолжится в поддиапазоне $SA[L, M]$ без сравнения символов.
- Если $l = Llcp[M]$, то у P будет l общих символов с $suff_{SA[L]}$ и $suff_{SA[M]}$, поэтому сравнение между P и $suff_{SA[M]}$ может начинаться с их $(l + 1)$-го символа. Выполнив ряд сравнений, мы определим m и их лексикографический порядок, попутно расширив *известную нам информацию* о символах P.

---

[1] Для упрощения представления правый диапазон $[M, R]$ взят вместо принятого в алгоритме 10.1 диапазона $[M + 1, R]$. Поскольку центральный элемент $SA[M]$ общий для левого и правого диапазонов, при каждом $R - L = 1$ мы рискуем оказаться в бесконечном цикле. Впрочем, можно легко изменить условие while этого алгоритма на $R - L = 1$ и явно проверить две строки-разделителя.

Очевидно, что на каждой итерации бинарного поиска или происходит дальнейшее сравнение символов P, или, если сравнение не выполняется, диапазон $[L, R]$ делится пополам. Первое может произойти не более p раз, а второе — $O(\log n)$ раз. Таким образом, мы пришли к следующей лемме.

Лемма 10.2. *С учетом трех массивов, lcp, Llcp и Rlcp, в дополнение к массиву суффиксов SA, построенному над текстом* $T[1, n]$, *подсчет вхождений шаблона* $P[1, p]$ *в текст T займет время* $O(p + \log n)$ *в худшем случае. Извлечение позиций этих вхождений occ займет дополнительное время* $O(occ)$. *Всего для поиска потребуется пространство* $O(n)$. *В модели на базе сравнения все эти границы оптимальны.*

Доказательство. Напомню, что для нахождения всех строк, у которых шаблон P выступает в качестве префикса, требуются два лексикографических поиска: для P и $P\#$, где # — специальный символ, больший, чем любой другой символ алфавита. Таким образом, $O(p + \log n)$ сравнений символов достаточно для того, чтобы установить диапазон $SA[i, j]$ суффиксов, имеющих P в качестве префикса. Нахождения шаблона за постоянное время легко подсчитываются как $occ = j - i + 1$, а вывести все их позиции можно за время $O(occ)$. ■

10.2.2. Построение массива LCP∞

Построение массива самых длинных общих префиксов `lcp`$[1, n - 1]$ кажется простой задачей: достаточно просканировать $n - 1$ смежную пару текстовых суффиксов в массиве SA и сравнить их посимвольно[1]. Это займет время $\Theta\left(\sum_{i=1}^{n-1}\left(\texttt{lcp}[i]+1\right)\right)$, где + 1 возникает из-за сравнения несовпадающего символа в $SA[i]$ и $SA[i + 1]$. Для некоторых входных данных, таких как $T = a^n$, ограничение по времени может оказаться $\Theta(n^2)$. В этом случае $SA[i]$ указывает на $T[n - i + 1, n]$, где $i = 1, ..., n$, поэтому `lcp`$[i] = i$, а временная сложность, как и заявлено, составит $\Theta\left(\sum_{i=1}^{n-1}\left(\texttt{lcp}[i]+1\right)\right) = \Theta\left(\sum_{i=1}^{n-1}\left(i+1\right)\right) = \Theta\left(n^2\right)$. В общем случае временная сложность составит $O(nl)$, где l — средний LCP среди всех суффиксов T.

В 2001 году Касаи и его коллеги предложили элегантный, обманчиво простой и линейный по времени алгоритм для вычисления массива `lcp` при условии, что нам известен массив суффиксов [12]. Линейность достигается за счет избегания повторного сканирования текстовых символов благодаря некоторым свойствам входного текста, доказанным на этапе разработки. Разбору этого алгоритма и посвящен остаток этого раздела.

Для наглядности обратимся к рис. 10.3, который иллюстрирует основную алгоритмическую идею. Сосредоточимся на двух последовательных суффиксах в тексте T, например $suff_{i-1} = T[i - 1, n]$ и $suff_i = T[i, n]$. В массиве суффиксов SA они занимают

[1] Напомню, что lcp$[i] = lcp(suffSA[i], suffSA[i + 1])$ для $i < n$.

позиции p и q, то есть $SA[p] = suff_{i-1}$ и $SA[q] = suff_i$. Рассмотрим текстовый суффикс, который лексикографически предшествует $SA[p]$, например $SA[p-1] = suff_{j-1}$ для некоторого j. Предположим, что мы индуктивно установили значение `lcp`$[p-1]$, хранящее длину самого длинного общего префикса между $SA[p-1]$ и $SA[p]$. Аналогично обозначим с помощью $SA[q-1] = suff_k = T[k, n]$ текстовый суффикс, который лексикографически предшествует $SA[q]$. По определению массива `lcp` запись `lcp`$[q-1]$ должна хранить их самый длинный общий префикс.

Отсортированные суффиксы	*SA*	Позиции *SA*
abcdef…	$j-1$	$p-1$
abchi…	$i-1$	p
.	.	.
.	.	.
.	.	.
bcdef…	j	
.	.	.
.	.	.
.	.	.
bch…	k	$q-1$
bchi…	i	q

Рис. 10.3. Связь между суффиксами и значениями lcp в алгоритме Касаи. Для простоты показаны только первые символы суффиксов

Алгоритм Касаи сканирует текстовые суффиксы слева направо (алгоритм 10.2), то есть элемент $T[i, n]$ проверяется после того, как обработан элемент $T[i-1, n]$. Наша цель — показать, что запись `lcp`$[q-1]$, которая ссылается на $T[i, n]$, может быть эффективно выведена из записи `lcp`$[p-1]$, которая ссылается на $T[i-1, n]$. Эффективность в данном случае означает, что вычисление `lcp`$[q-1]$ не требует полного повторного сканирования $T[i, n]$, так как мы можем воспользоваться сведениями о записи `lcp`$[p-1]$ и начать с места, где завершилось сравнение $SA[p-1]$ и $SA[p]$. В этом случае повторного сканирования текстовых символов не потребуется, и в результате мы получим линейную по времени сложность.

Начнем со свойства, которое упоминалось при рассмотрении префиксного поиска. Здесь я его просто переформулирую в контексте суффиксных массивов.

Факт 10.1. Самый длинный общий префикс (LCP) между последовательными суффиксами $SA[y-1]$ и $SA[y]$ не короче, чем LCP между $SA[y]$ и любым другим предыдущим суффиксом $SA[x]$, где $x = 1 \ldots y-1$.

Доказательство. Это свойство вытекает из лексикографической упорядоченности суффиксов в SA, в результате чего по мере удаления от $SA[y]$ мы уменьшаем длину их общего префикса. ■

Алгоритм 10.2. LCP-BUILD(*T*, *SA*)

```
1:  h = 0;
2:  for (i = 1; i ⩽ n, i++) do
3:    q = SA^{-1}[i];
4:    if q > 1 then
5:      k = SA[q − 1];
6:      if h > 0 then
7:        h = h − 1;
8:      end if
9:      while T[k + h] = T[i + h] do
10:       h = h + 1;
11:     end while
12:     lcp[q − 1] = h;
13:   end if
14: end for
```

Теперь обратимся к рис. 10.3 и сосредоточимся на общем шаге i алгоритма Касаи, который сравнил пару последовательных суффиксов $SA[p-1] = suff_{j-1}$ и $SA[p] = suff_{i-1}$, а затем перешел к сравнению пары последовательных суффиксов $SA[q-1] = suff_j$ и $SA[q] = suff_i$. Возможны два случая: если $\texttt{lcp}[p-1] > 0$, то у префикса двух соседних суффиксов $SA[p-1]$ и $SA[p]$ будут общие символы (как на рис. 10.3), а если $\texttt{lcp}[p-1] = 0$, общих символов не будет.

В первом случае можно заключить, что, поскольку лексикографически $suff_{j-1}$ предшествует $suff_{i-1}$, их следующие суффиксы в T сохранят этот лексикографический порядок, то есть $suff_j$ будет предшествовать $suff_i$. Более того, поскольку $suff_j$ (и, соответственно, $suff_i$) получается из $suff_{j-1}$ (и, соответственно, из $suff_{i-1}$) путем отбрасывания его первого символа, то $lcp(suff_j, suff_i) = lcp(suff_{j-1}, suff_{i-1}) - 1 = \texttt{lcp}[p-1] - 1$. На рис. 10.3 мы имеем $\texttt{lcp}[p-1] = 3$, а общий префикс — `abc`, поэтому у следующих суффиксов будет LCP `bc` длиной 2 и их порядок сохранится, поскольку $suff_j$ находится перед $suff_i$.

Таким образом, доказано следующее свойство, которое я перефразировал с учетом наблюдения, показывающего, что $j = SA[p-1] + 1$ и $i = SA[p] + 1$.

Факт 10.2. Если $lcp(suff_{SA[p-1]}, suff_{SA[p]}) > 0$, тогда $lcp(suff_{SA[p-1]+1}, suff_{SA[p]+1}) = lcp(suff_{SA[p-1]}, suff_{SA[p]}) - 1$.

Далее нужно отметить, что хотя $suff_{j-1}$ и $suff_{i-1}$ располагаются в массиве SA последовательно, для следующих за ними суффиксов $suff_j$ и $suff_i$ это может оказаться не так, как показано на рис. 10.3. Из фактов 10.1 и 10.2 можно вывести ключевое

свойство: `lcp`$[q-1] \geqslant \max\{$`lcp`$[p-1]-1, 0\}$. Оно алгоритмически развернуто в алгоритме Касаи для вычисления `lcp`$[q-1]$ и в полной мере использует сравнения, применявшиеся для вывода `lcp`$[p-1]$.

Приведенный в алгоритме 10.2 псевдокод задействует обратный массив суффиксов, обозначенный SA^{-1} и возвращающий для каждого суффикса его позицию в SA. Согласно рис. 10.3 $SA^{-1}[i] = q$ и $SA^{-1}[i-1] = p$. Основу алгоритма 10.2 составляет цикл `for`, который слева направо перебирает суффиксы $suff_i$, извлекая для каждого его позицию в SA, а именно $q = SA^{-1}[i]$, и в конце устанавливает содержимое записи `lcp`$[q-1]$ (см. шаг 12). Чтобы сделать процесс инициализации последовательным, на шаге 4 проверяется, занимает ли $suff_i$ первую позицию массива суффиксов, то есть верно ли $q = 1$. В этом случае LCP с предыдущим суффиксом не определен, поэтому алгоритм пропускает вычисление LCP и переходит к следующему i. В противном случае, то есть при $q > 1$, на шаге 5 вычисляется суффикс, лексикографически предшествующий $suff_i$, как $k = SA[q-1]$. Затем алгоритм посимвольно сравнивает LCP между $SA[q-1]$ и $SA[q]$, начиная со смещения $h =$ `lcp`$[p-1]$, которое надлежащим образом уменьшается на одну единицу на шаге 6, согласно факту 10.2.

Что касается временной сложности, обратите внимание на то, что смещение h уменьшается не более чем n раз, по одному разу на итерацию цикла `for`, и не может выйти за пределы T из-за шага 9, поскольку строка T завершается специальным символом, таким как `\0` в C. Это означает, что h допускает увеличение не более чем $2n$ раз, следовательно, это верхняя граница количества сравнений символов, выполняемых алгоритмом 10.2. В результате общая временная сложность составит $O(n)$. Очевидно, что этот алгоритм не очень эффективен по вводу-выводу, так как записи `lcp` стоят в произвольном порядке. Известны некоторые эвристические алгоритмы, позволяющие уменьшить количество вводов-выводов, но оптимальная граница ввода-вывода $O(n/B)$, если она действительно возможна, еще не достигнута.

Теорема 10.1. *Для строки* $T[1, n]$ *и ее массива суффиксов SAT можно получить соответствующий массив* `lcp` *за время* $O(n)$*, заняв при этом пространство* $O(n)$*. В двухуровневой модели памяти этот алгоритм может быть неэффективным, так как требует* $O(n)$ *операций ввода-вывода.*

10.2.3. Конструирование массива суффиксов

Так как массив SA представляет собой отсортированную последовательность суффиксов, наиболее интуитивный способ его построения — взять эффективный алгоритм сортировки на основе сравнений и отредактировать функцию сравнения таким образом, чтобы она вычисляла лексикографический порядок строк. Алгоритм 10.3 реализует эту идею в стиле языка C, используя в качестве сортировщика встроенную процедуру `QSORT` и подпрограмму `Suffix_cmp` для сравнения суффиксов:

```
Suffix_cmp(char **p, char **q){ return strcmp(*p,*q); };
```

Алгоритм 10.3. COMPARISON_BASED_CONSTRUCTION(char *T, int n, char **SA)

```
1: for (i = 0; i < n; i++) do
2:     SA[i] = T + i;
3: end for
4: QSORT(SA, n, sizeof(char *), Suffix_cmp);
```

Обратите внимание на то, что массив суффиксов инициализируется указателями на реальные начальные позиции в памяти сортируемых суффиксов, а не целочисленными смещениями от 1 до n, как указано в формальном описании SA в начале раздела 10.2. Дело в том, что процедуре `Suffix_cmp` не нужно знать позицию T в памяти, что потребовало бы использования глобального параметра, так как фактические параметры, переданные во время вызова, напрямую предоставляют начальные позиции памяти сравниваемых суффиксов. Более того, массив суффиксов SA имеет индексы, начинающиеся с 0, что типично для языка C.

Главный недостаток этого простого подхода состоит в его неэффективности по вводу-выводу, ведь оптимальное число сравнений $O(n \log n)$ теперь включает строки переменной длины, количество символов в которых может доходить до $\Theta(n)$. Кроме того, локальность массива SA не приводит к локальности в сравнениях суффиксов, потому что при сортировке переставляются указатели строк, а не сами строки. Эти моменты требуют дополнительных операций ввода-вывода, делая этот простой алгоритм медленным.

Теорема 10.2. *Для построения массива суффиксов строки $T[1, n]$ сортировщику, работающему на основе сравнения, в худшем случае требуются $O((n/B)n \log n)$ операций ввода-вывода и пространство $O(n \log n)$ битов.*

А сейчас я покажу вам два эффективных с точки зрения ввода-вывода подхода к построению массива суффиксов. Базирующийся на принципе «разделяй и властвуй» алгоритм DC3, предложенный Карккайненом и Сандерсом [11], — элегантный, простой в кодировании и достаточно гибкий для достижения оптимальной границы ввода-вывода в различных моделях вычислений. Второй алгоритм на базе сканирования, предложенный Гонне, Баеза-Йетсом и Снайдером [10], также довольно прост. И хотя он требует большего количества операций ввода-вывода, но все равно интересен, поскольку предлагает обработку входных данных на лету, подобно потоковой передаче, что дает возможность сделать предварительную выборку, допускает сжатие и, следовательно, подходит для медленных дисков.

Skew-алгоритм. В 2003 году Карккайнен и Сандерс [11] показали, что задачу построения массивов суффиксов можно *свести* к сортировке набора троек, компоненты которых являются целыми числами в диапазоне $[1, O(n)]$. Удивительно, но результат этого перехода занимает *линейное время и пространство*. Фактически мы спускаемся от сложности построения массива суффиксов к сложности сортировки атомарных элементов, которая подробно обсуждалась в предыдущих главах

и для которой уже известны оптимальные алгоритмы для случаев иерархической памяти и нескольких дисков. Более того, поскольку элементы, которые нужно отсортировать, — это целые числа, ограниченные по значению $O(n)$, сортировка троек в RAM-модели займет время $O(n)$. Фактически мы узнали, как в этой модели построить массив суффиксов с линейной временной сложностью. Очень впечатляюще!

Этот алгоритм ценен тем, что работает в любой модели вычислений, для которой доступен эффективный примитив сортировки. Он основан на подходе «разделяй и властвуй», который выполняет разбиение $\frac{2}{3}:\frac{1}{3}$, что крайне важно для упрощения последнего шага слияния. Предыдущие подходы, например [4], использовали более естественное разбиение $\frac{1}{2}:\frac{1}{2}$, но это порождало сложности на этапе слияния, так как требовалось прибегнуть к такой структуре данных, как суффиксное дерево (описано в разделе 10.3). Из-за *несбалансированности* базового разбиения алгоритм изначально назывался skew, а затем получил сокращенное название DC3 (от difference cover modulo 3 — «покрытие разностей по модулю 3»).

Обозначим входную строку $T[1, n] = t_1 t_2 ... t_n$. Предполагается, что она составлена из символов целочисленного алфавита размером $\Sigma = O(n)$. Если это не так, символы T можно отсортировать и переименовать в целые числа из диапазона $[0, n - 1]$, что в худшем случае потребует времени $O(n \log \Sigma)$. Таким образом, мы имеем текст из целых чисел, каждое из которых занимает $O(\log n)$ бит. Это верно для всех текстов, созданных при построении массива суффиксов. Кроме того, предполагается, что t_n = \$. Это специальный символ, который меньше любого другого символа алфавита и логически дополняет T бесконечным числом вхождений.

Теперь можно описать три основных шага нашего алгоритма.

Шаг 1. Конструируем массив суффиксов $SA^{2,0}$, ограниченный суффиксами, начинающимися с позиций $P_{2,0} = \{i : i \bmod 3 = 2 \text{ или } i \bmod 3 = 0\}$.

Этот шаг можно разбить на три этапа.

- Строим специальную строку $T^{2,0}$ длиной $(2/3)n$, которая компактно кодирует все суффиксы T, начинающиеся с позиций $P_{2,0}$.
- Рекурсивно строим массив суффиксов SA' для $T^{2,0}$.
- Выводим из SA' массив суффиксов $SA^{2,0}$.

Шаг 2. Конструируем из оставшихся суффиксов массив SA^1, начиная с позиции $P_1 = \{i : i \bmod 3 = 1\}$.

Эта процедура состоит из трех этапов.

- Предположим, что мы предварительно вычислили массив `pos`$[j]$, который показывает позицию j-го текстового суффикса $T[j, n]$ в массиве $SA^{2,0}$.
- Для каждого $i \in P_1$ суффикс $T[i, n]$ представляется с помощью пары $\langle T[i],$ `pos`$[i + 1]\rangle$, где $i + 1 \in P_{2,0}$.
- Применяем `RADIXSORT` к $O(n)$ парам.

Шаг 3. Объединяем два массива суффиксов $SA^{2,0}$ и SA^{1} через дополнительный шаг — выполняем разбиение $\frac{2}{3}:\frac{1}{3}$, которое обеспечивает лексикографическое сравнение любой пары суффиксов за постоянное время (подробности будут чуть позже).

Сейчас я подробно опишу этот алгоритм и проиллюстрирую его на примере входной строки $T[1, 12]$ = mississippi\$, массив суффиксов которой SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3). В этом примере $P_{2,0}$ = {2, 3, 5, 6, 8, 9, 11, 12}, а P_1 = {1, 4, 7, 10}.

Шаг 1. Первый шаг наиболее сложен, так как он лежит в основе всего рекурсивного процесса. Происходит лексикографическая сортировка суффиксов, начиная с позиций $P_{2,0}$. Результирующий массив $SA^{2,0}$ представляет собой *выборочную* версию конечного массива суффиксов SA, поскольку он ограничен суффиксами, начинающимися с позиций $P_{2,0}$.

Чтобы эффективно получить $SA^{2,0}$, сведем задачу к построению массива суффиксов для строки $T^{2,0}$ длиной примерно $2n/3$. Текст этой строки состоит из символов, которые являются целыми числами с максимальным значением примерно $2n/3$. Мы снова имеем дело с текстом из целых чисел, длина которого пропорционально меньше n, а значит, можем построить для него массив суффиксов, вызвав *рекурсивно* уже знакомую процедуру.

Основная сложность заключается в том, как определить $T^{2,0}$ таким образом, чтобы его массив суффиксов позволял легко вывести $SA^{2,0}$. Нам нужна отсортированная последовательность текстовых суффиксов, начинающихся с позиций в $P_{2,0}$. В [11] было предложено элегантное решение, в котором два текстовых суффикса, $T[2, n]$ и $T[3, n]$, дополнялись специальным символом \$ для получения длины, кратной 3. Затем полученные строки делили на тройки символов, то есть $T[2, \cdot]$ = = $[t_2, t_3, t_4][t_5, t_6, t_7][t_8, t_9, t_{10}]$... и $T[3, \cdot] = [t_3, t_4, t_5][t_6, t_7, t_8][t_9, t_{10}, t_{11}]$... Точка обозначает, что мы рассматриваем наименьшее целое число, большее чем n, которое позволяет этим строкам иметь длину, кратную 3.

Возвращаясь к предыдущему примеру, имеем:

$$T[2, \cdot] = [\underset{2}{\texttt{i s s}}][\underset{5}{\texttt{i s s}}][\underset{8}{\texttt{i p p}}][\underset{11}{\texttt{i \$ \$}}];$$

$$T[3, \cdot] = [\underset{3}{\texttt{s s i}}][\underset{6}{\texttt{s s i}}][\underset{9}{\texttt{p p i}}][\underset{12}{\texttt{\$ \$ \$}}].$$

Объединим эти две строки и получим:

$$R = T[2, \cdot\ ]\ T[3, \cdot]:$$

$$R = [\underset{2}{\texttt{i s s}}][\underset{5}{\texttt{i s s}}][\underset{8}{\texttt{i p p}}][\underset{11}{\texttt{i \$ \$}}][\underset{3}{\texttt{s s i}}][\underset{6}{\texttt{s s i}}][\underset{9}{\texttt{p p i}}][\underset{12}{\texttt{\$ \$ \$}}].$$

Вот ключевое свойство, на котором базируется первый шаг skew-алгоритма.

Свойство 1. Каждый суффикс $T[i, n]$, начинающийся с позиции $i \in P_{2,0}$, можно поставить в соответствие суффиксу строки R, состоящей из целостной последовательности триплетов. В частности, при $i \bmod 3 = 0$ суффикс $T[i, n]$ точно совпадает

с суффиксом R, а если $i \bmod 3 = 2$, то $T[i, n]$ предшествует суффиксу R, который в любом случае заканчивается специальным символом \$.

В предыдущем примере возьмем $i = 6 \in P_{2,0}$, так как $i \bmod 3 = 0$, и обратим внимание на то, что суффикс $T[6, 12]$ = `ssippi$` встречается во второй тройке $T[3, \cdot]$, которая является шестым триплетом R. Аналогично возьмем $i = 8 \in P_{2,0}$, так как $i \bmod 3 = 2$, и заметим, что суффикс $T[8, 12]$ = `ippi$` встречается в третьей тройке $T[2, \cdot]$, которая является третьим триплетом R. Обратите внимание: даже если $T[8, 12]$ не является полным суффиксом R, он заканчивается двумя знаками \$, которые послужат своего рода конечным разделителем.

С формальной точки зрения корректность этого свойства можно легко вывести, заметив, что любой суффикс $T[i, n]$, начинающийся с позиции $i \in P_{2,0}$, является суффиксом либо $T[2, \cdot]$, либо $T[3, \cdot]$, когда $i > 0$, а $i \bmod 3$ равно 0 или 2. Более того, поскольку $i \in P_{2,0}$, он имеет вид $i = 3 + 3k$ или $i = 2 + 3k$ для $k \geqslant 0$. Таким образом, $T[i, n]$ встречается в R, выровненном по началу какой-то тройки.

Последняя операция для получения строки $T^{2,0}$, состоящей из $2n/3$ целочисленных символов, заключается в кодировании троек с помощью целых чисел. Его следует реализовать таким образом, чтобы лексикографическое сравнение двух троек вытекало из сравнения соответствующих целых чисел. В литературе это называется *лексикографическим именованием* и легко реализуется с помощью процедуры `RADIXSORT` над тройками в R и связывания с каждой тройкой ее ранга в лексикографическом порядке. Поскольку у нас $O(n)$ троек, состоящих из символов в диапазоне $[0, n]$, процедура `RADIXSORT` для них занимает время $O(n)$.

В нашем примере отсортированные тройки (вверху) помечены следующими рангами (внизу):

$$\underset{0}{[\$\ \$\ \$]}\ \underset{1}{[i\ \$\ \$]}\ \underset{2}{[i\ p\ p]}\ \underset{3}{[i\ s\ s]}\ \underset{3}{[i\ s\ s]}\ \underset{4}{[p\ p\ i]}\ \underset{5}{[s\ s\ i]}\ \underset{5}{[s\ s\ i]}.$$

Это позволяет сконструировать следующую строку троек $T^{2,0}$:

$$R = \underset{3}{[i\ s\ s]}\ \underset{3}{[i\ s\ s]}\ \underset{2}{[i\ p\ p]}\ \underset{1}{[i\ \$\ \$]}\ \underset{5}{[s\ s\ i]}\ \underset{5}{[s\ s\ i]}\ \underset{4}{[p\ p\ i]}\ \underset{0}{[\$\ \$\ \$]}.$$

После именования троек в R образуется новый текст $T^{2,0} = 33215540$ длиной $2n/3$. Здесь важно то, что наш текст $T^{2,0}$ снова является набором целых чисел, как T, занимая $O(\log n)$ бит на одно целое число, как и раньше. Но $T^{2,0}$ имеет меньшую длину, чем T, поэтому для него можно рекурсивно вызвать процедуру построения суффиксного массива.

Очевидно, поскольку ранги назначаются в порядке, равном лексикографическому порядку их троек, лексикографическое сравнение суффиксов R, выровненных по тройкам, эквивалентно лексикографическому сравнению суффиксов $T^{2,0}$. Здесь в игру вступает свойство 1, определяющее биекцию между суффиксами R, выровненными по началам троек, а следовательно, и суффиксами $T^{2,0}$, с текстовыми суффиксами, начинающимися с $P_{2,0}$. Затем это соответствие развертывается для получения $SA^{2,0}$ из массива суффиксов $T^{2,0}$.

В нашем примере с $T^{2,0} = 33215540$ алгоритм применяется рекурсивно, давая массив суффиксов (8, 4, 3, 2, 1, 7, 6, 5). Его можно превратить в $SA^{2,0}$, преобразовав позиции в $T^{2,0}$ в позиции в T. Учитывая расположение троек в $T^{2,0}$, это можно сделать с помощью обычных арифметических операций. Здесь результатом будет массив суффиксов $SA^{2,0} = (12, 11, 8, 5, 2, 9, 6, 3)$.

В конце хотелось бы сделать два замечания. Во-первых, если все символы в $T^{2,0}$ различны, рекурсия не требуется, так как в этом случае суффиксы можно отсортировать по их первым символам. Во-вторых, программистам для получения массива $SA^{2,0}$ следует осторожно подходить к преобразованию позиций суффиксов в $T^{2,0}$ в позиции суффиксов в T, так как при этом должно учитываться расположение троек R.

Шаг 2. После рекурсивного построения массива суффиксов $SA^{2,0}$ оставшиеся суффиксы T, которые начинаются с позиций текста $i \bmod 3 = 1$, можно отсортировать лексикографически. Для этого суффикс $T[i, n]$ разлагается как пара, состоящая из его первого символа $T[i]$ и его оставшегося суффикса $T[i + 1, n]$. Так как $i \in P_1$, следующая позиция $i + 1 \in P_{2,0}$, таким образом, суффикс $T[i + 1, n]$ встречается в $SA^{2,0}$. Суффикс $T[i, n]$ можно закодировать в виде пары целых чисел $\langle T[i], \texttt{pos}[i + 1]\rangle$, где $\texttt{pos}[i + 1]$ обозначает лексикографический ранг суффикса $T[i + 1, n]$ в массиве $SA^{2,0}$. Если $i + 1 = n + 1$, то мы устанавливаем $\texttt{pos}[n + 1] = 0$, помня о том, что символ \$ меньше любого другого символа алфавита.

Из этого наблюдения следует, что два текстовых суффикса, начинающихся с позиций в P_1, можно сравнить за постоянное время, сопоставляя их соответствующие пары. Поэтому SA^1 можно вычислить за время $O(n)$ с помощью процедуры `RADIXSORT`, примененной к $O(n)$ парам, кодирующим его суффиксы.

В нашем примере процедура `RADIXSORT` применяется к парам $\langle T[i], \texttt{pos}[i + 1]\rangle$, в результате чего получается массив суффиксов $SA^1 = (1, 10, 7, 4)$.

Начальная позиция в P_1	1	4	7	10
Пары: символ + текстовый суффикс	$\langle T[1], T[2,\cdot]\rangle$ $\langle \texttt{m}, T[2,\cdot]\rangle$	$\langle T[4], T[5, 12]\rangle$ $\langle \texttt{s}, T[5, 12]\rangle$	$\langle T[7], T[8, 12]\rangle$ $\langle \texttt{s}, T[8, 12]\rangle$	$\langle T[10], T[11, 12]\rangle$ $\langle \texttt{p}, T[11, 12]\rangle$
Пары: символ + позиция суффикса в $SA^{2,0}$	$\langle \texttt{m}, \texttt{pos}[2]\rangle$ $\langle \texttt{m}, 4\rangle$	$\langle \texttt{s}, \texttt{pos}[5]\rangle$ $\langle \texttt{s}, 3\rangle$	$\langle \texttt{s}, \texttt{pos}[8]\rangle$ $\langle \texttt{s}, 2\rangle$	$\langle \texttt{p}, \texttt{pos}[11]\rangle$ $\langle \texttt{p}, 1\rangle$
Отсортированные пары	$\langle \texttt{m}, 4\rangle$ <	$\langle \texttt{p}, 1\rangle$ <	$\langle \texttt{s}, 2\rangle$ <	$\langle \texttt{s}, 3\rangle$
SA^1	1	10	7	4

Шаг 3. Пришло время объединить два отсортированных массива, SA^1 и $SA^{2,0}$. Это можно сделать за линейное время $O(n)$ благодаря наблюдению, которое послужило основанием для разделения $\frac{2}{3} : \frac{1}{3}$. Возьмем два суффикса, $T[i, n] \in SA^1$ и $T[j, n] \in SA^{2,0}$, которые нужно лексикографически сравнить для слияния. Поскольку они находятся

в разных массивах, мы не знаем, как они *связаны лексикографически*, а посимвольное сравнение повлечет за собой очень высокие затраты. Поэтому прибегнем к разложению примерно так же, как делали на шаге 2. Для этого будем считать, что суффикс состоит *из одного или двух символов* плюс лексикографический ранг его оставшейся части. Такое разложение эффективно, когда оставшиеся части сравниваемых суффиксов находятся в одном и том же массиве, соответственно, их ранга достаточно, чтобы узнать их порядок за постоянное время. Это довольно элегантное решение возможно при разделении $\frac{2}{3}:\frac{1}{3}$, но нереализуемо при разделении $\frac{1}{2}:\frac{1}{2}$.

1. При $j \bmod 3 = 2$ мы сравниваем $T[j, n]$ с $T[i, n]$, рассматривая их как пары $\langle T[j], T[j+1, n]\rangle$ и $\langle T[i], T[i+1, n]\rangle$. Оба суффикса, $T[j+1, n]$ и $T[i+1, n]$, встречаются в $SA^{2,0}$ при условии, что их начальные позиции конгруэнтны 0 или 2 mod 3 соответственно. Это дает возможность выполнить лексикографическое сравнение путем сравнения пар $\langle T[i], \texttt{pos}[i+1]\rangle$ и $\langle T[j], \texttt{pos}[j+1]\rangle$. Такое сравнение, если массив нам доступен[1], занимает время $O(1)$.
2. При $j \bmod 3 = 0$ мы сравниваем $T[j, n]$ с $T[i, n]$ путем сопоставления троек $\langle T[j], T[j+1], T[j+2, n]\rangle$ и $\langle T[i], T[i+1], T[i+2, n]\rangle$. Оба суффикса, $T[j+2, n]$ и $T[i+2, n]$, присутствуют в $SA^{2,0}$ при условии, что их начальные позиции конгруэнтны 0 или 2 mod 3 соответственно. В этом случае лексикографическое сравнение выполняется с помощью сравнения троек $\langle T[i], T[i+1], \texttt{pos}[i+2]\rangle$ и $\langle T[j], T[j+1], \texttt{pos}[j+2]\rangle$. При условии доступности массива `pos` это сравнение занимает время $O(1)$.

В нашем примере $T[8, 11] < T[10, 11]$ и фактически $\langle i, 5\rangle < \langle p, 1\rangle$. Кроме того, $T[7, 11] < T[6, 11]$ и фактически $\langle s, i, 5\rangle < \langle s, s, 2\rangle$. В таблице показаны все возможные пары или тройки массива SA^1, доступные для сравнения. Символами (*) и (**) обозначены пары и тройки для приведенных ранее правил 1 и 2 соответственно. Так как мы не знаем, какой суффикс $SA^{2,0}$ будет сравниваться с суффиксом SA^1 во время слияния, для каждого из последних суффиксов нужно вычислить оба представления, (*) и (**), то есть как пару и как тройку[2]. Слияние дает нам окончательный массив суффиксов $SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$.

SA^1				$SA^{2,0}$								Позиции суффиксов
1	10	7	4	12	11	8	5	2	9	6	3	
⟨m, 4⟩	⟨p, 1⟩	⟨s, 2⟩	⟨s, 3⟩		⟨i, 0⟩	⟨i, 5⟩	⟨i, 6⟩	⟨i, 7⟩				(*)
⟨m, i, 7⟩	⟨p, i, 0⟩	⟨s, i, 5⟩	⟨s, i, 6⟩	⟨$, $, −1⟩					⟨p, p, 1⟩	⟨s, s, 2⟩	⟨s, s, 3⟩	(**)

[1] Массив pos является обратным по отношению к $SA^{2,0}$, поэтому может быть получен из $SA^{2,0}$ за линейное время.

[2] Напомню, что $\text{pos}[n] = 0$ и что для соблюдения лексикографического порядка для всех $j > n$ мы сделали $\text{pos}[j] = -1$.

Из приведенного описания ясно, что каждый шаг может быть реализован сортировкой или сканированием набора из n атомарных элементов, которые, возможно, являются тройками целых чисел и занимают $O(\log n)$ бит каждая, то есть $O(1)$ слов памяти. Поэтому предлагаемый метод можно рассматривать как *алгоритмическое сведение* задачи построения суффиксного массива к классической задаче сортировки n элементов.

В RAM-модели временная сложность skew-алгоритма может быть смоделирована с помощью рекуррентного соотношения $T(n) = T(2n/3) + O(n)$, ведь шаги 2 и 3 требуют времени $O(n)$, а рекурсивный вызов выполняется над строкой $T^{2,0}$, длина которой равна $(2/3)n$. У рекуррентного соотношения есть решение $T(n) = O(n)$, которое, очевидно, оптимально. В двухуровневой модели памяти skew-алгоритм может быть реализован за $O(n/B \log_{M/B} n/M)$ операций ввода-вывода, что совпадает со сложностью ввода-вывода сортировки n атомарных элементов.

Теорема 10.3. *Skew-алгоритм строит массив суффиксов строки $T[1, n]$ за $O(\mathtt{Sort}(n))$ операций ввода-вывода, используя $O(n/B)$ страниц диска. При полиномиальном по n размере алфавита σ процессорное время оценивается как $O(n)$.*

Алгоритм на базе сканирования$^{\infty}$

До появления skew-алгоритма наиболее известным алгоритмом для работы с хранящимися на диске данными был вариант, предложенный Гонне, Баеза-Йетсом и Снайдером в 1992 году [10]. Он работал по принципу «разделяй и властвуй» и имел сильно несбалансированный шаг разбиения, поэтому количество сравнений суффиксов у него было квадратичным, что давало кубическую временную сложность. Но на практике он работал довольно быстро, поскольку обрабатывал данные на лету, тем самым используя высокую пропускную способность современных дисков.

Возьмем положительную константу $\ell < 1$ для построения во внутренней памяти массива суффиксов текстового фрагмента из $m = \ell M$ символов. Предположим, что текст $T[1, n]$ логически разделен на блоки по m символов. Эти блоки нумеруются вправо, а именно $T = T_1 T_2 ... T_{n/m}$, где $T_h = T[hm + 1, (h + 1)m]$ для $h = 0, 1...$ Алгоритм *пошагово* вычисляет массив суффиксов T за $\Theta(n/M)$ этапов, а не за логарифмическое число этапов, как в skew-алгоритме. В начале этапа h мы предполагаем, что на диске есть *массив SA^h, содержащий отсортированную последовательность первых hm суффиксов T*. Изначально $h = 0$ и массив SA' пуст. На h-м этапе алгоритм загружает во внутреннюю память следующий фрагмент текста T^{h+1}, строит SA' как отсортированную последовательность суффиксов, начиная с T^{h+1}, а затем вычисляет новый массив SA^{h+1} путем слияния двух отсортированных последовательностей SA^h и SA'.

При реализации этой алгоритмической идеи возникают две основные проблемы:

- как эффективно построить SA', поскольку его суффиксы начинаются с T^{h+1}, но до конца текста T могут выходить за пределы этого блока символов;
- как эффективно объединить два отсортированных массива, SA^h и SA', если длина содержащихся в них суффиксов может доходить до $\Theta(n)$ символов.

Первую проблему алгоритм решает в лоб посимвольным сравнением пар суффиксов, что занимает время $O(n)$ и требует $O(n/B)$ операций ввода-вывода. Это позволяет расширить процесс сравнения суффиксов за пределы загруженного во внутреннюю память блока. За $O(n/M)$ этапов конструирование массива SA' требует $O\left(\frac{n}{B}\frac{n}{m}m\log m\right)=O\left(\frac{n^2}{B}\log m\right)$ операций ввода-вывода.

Для решения второй проблемы алгоритм применяет умный подход к слиянию SA' и SA^h, задействуя вспомогательный массив $C[1, m + 1]$. Он подсчитывает в $C[j]$ количество суффиксов SA^h, которые лексикографически больше, чем $SA'[j - 1]$-й текстовый суффикс, и меньше, чем $SA'[j]$-й текстовый суффикс. В данном случае $SA'[0]$ логически устанавливается как пустая строка, а $SA'[m + 1]$ — как специальная строка, превышающая любую другую. Массив SA^h постоянно растет, так что его невозможно поместить во внутреннюю память, и поэтому его обрабатывают потоковым способом, сканируя текст T вправо (с начала) и выполняя двоичный поиск каждого суффикса в SA'. Если лексикографическая позиция искомого суффикса равна j, запись $C[j]$ увеличивается. Двоичный поиск в массиве SA^h может включать сравнение некоторых символов суффикса за пределами находящегося во внутренней памяти блока T^{h+1}, поэтому на один шаг этого поиска нужно брать худший случай $O(n/B)$ операций ввода-вывода. На всех n/M этапах вычисление массива C займет $O\left(\sum_{h=0}^{n/m-1}\frac{n}{B}(hm)\log m\right)=O\left(\frac{n^3}{MB}\log M\right)$ операций ввода-вывода (напомню, что $m = \ell M$).

На следующем подэтапе массив C используется для быстрого слияния двух массивов: SA', который находится во внутренней памяти, и SA^h, находящегося на диске. Запись $C[j]$ указывает, сколько последовательных суффиксов SA^h лексикографически располагаются после $SA'[j - 1]$ и перед $SA'[j]$. Следовательно, сканирования диска достаточно для выполнения процесса слияния за $O(n/B)$ операций ввода-вывода.

Теорема 10.4. *Алгоритм на базе сканирования создает массив суффиксов строки* $T[1, n]$ *за* $O\left(\frac{n^3}{MB}\log M\right)$ *операций ввода-вывода, используя* $O(n/B)$ *страниц диска.*

Поскольку в наихудшем случае мы видим кубическое число операций ввода-вывода, алгоритм выглядит неперспективным. Но нельзя забывать о том, что при каждом сравнении суффиксов во внутренней памяти ищут все присутствующие в этих суффиксах символы. Поэтому на практике количество операций ввода-вывода для этого алгоритма лучше описывает формула $O\left(\frac{n^2}{MB}\right)$. Кроме того, все операции ввода-вывода в этом случае являются последовательными, а фактическое число случайных поисков составляет всего $O(n/M)$ (не более постоянного числа на один этап). Так что алгоритм в полной мере использует большую пропускную способность современных дисков и высокую скорость современных процессоров. Напоследок отмечу, что последовательное сканирование массивов суффиксов SA^h и текста T

позволяет применить некоторую форму сжатия для уменьшения объема операций ввода-вывода, что способствует дальнейшему ускорению базового алгоритма.

На примере следующей текстовой строки посмотрим, как работает алгоритм на базе сканирования.

$$T[1,12] = \overset{1}{\texttt{m}}\ \overset{2}{\texttt{i}}\ \overset{3}{\texttt{s}}\ \overset{4}{\texttt{s}}\ \overset{5}{\texttt{i}}\ \overset{6}{\texttt{s}}\ \overset{7}{\texttt{s}}\ \overset{8}{\texttt{i}}\ \overset{9}{\texttt{p}}\ \overset{10}{\texttt{p}}\ \overset{11}{\texttt{i}}\ \overset{12}{\texttt{\$}}.$$

Предположим, что $m = 3$ и в начале этапа $h = 1$. Алгоритм уже обработал текстовый блок $T^0 = T[1, 3] =$ `mis`, то есть вычислил и сохранил на диск массив $SA^1 = (2, 1, 3)$, который соответствует лексикографическому порядку трех начинающихся в этом блоке текстовых суффиксов. Это суффиксы `mississippi$`, `ississippi$` и `ssissippi$`. На первом этапе алгоритм загружает во внутреннюю память блок $T^1 = T[4, 6] =$ `sis`, лексикографически сортирует текстовые суффиксы, которые начинаются в позициях [4, 6] и продолжаются до конца T, как показано на рис. 10.4. Обратите внимание на то, что на втором шаге в сравнении текстовых суффиксов $T[4, 12] =$ `sissippi$` и $T[6, 12] =$ `ssippi$` участвуют символы, лежащие за пределами доступной во внутренней памяти текстовой части $T[4, 6]$, что порождает дополнительные операции ввода-вывода. Последний, третий шаг вычисляет новый массив SA^2 путем слияния хранящегося на диске массива $SA^1 = (2, 1, 3)$ с массивом $SA' = (5, 4, 6)$, доступным во внутренней памяти. Рисунок 10.4 показывает это слияние на примере массива $C[1, 4] = [0, 2, 0, 1]$. В этом случае $C[2] = 2$, потому что суффиксы $T[1, 12] =$ `mississippi$` и $T[2, 12] =$ `ississippi$` находятся между суффиксом номер $SA'[1]$ ($T[5, 12] =$ `issippi$`) и суффиксом номер $SA'[2]$ ($T[4, 12] =$`sissippi$`), а $C[4] = 1$, потому что суффикс $T[3, 12] =$ `ssissippi$` располагается после суффикса номер $SA'[3]$ ($T[6, 12] =$ `ssippi$`).

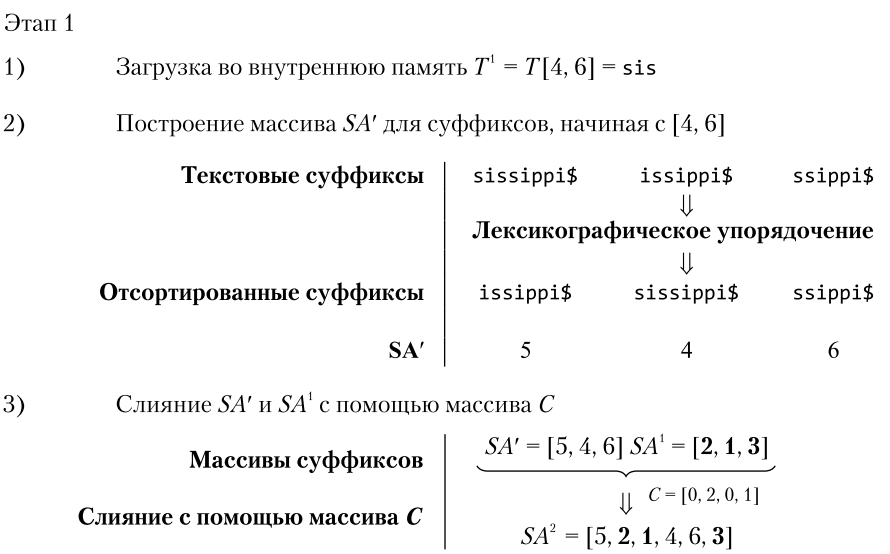


Рис. 10.4. Первый этап алгоритма на базе сканирования

Следующие два этапа обработки текстовых подстрок $T^2 = T[7, 9]$ = `sip` и $T^3 =$ $= T[10, 12]$ = `pi$` иллюстрируют рис. 10.5 и 10.6 соответственно. В частности, на этапе 2 в память загружается подстрока $T^2 = T[7, 9]$ и строится массив SA' для суффиксов, начинающихся с позиций [7, 9]. Затем этот массив объединяется с находящимся на диске массивом SA^2, в котором содержатся суффиксы, начинающиеся с $T[1, 6]$ (см. рис. 10.4). Во время этапа 3, последнего, в память загружается подстрока $T^3 = T[10, 12]$ и строится массив SA' для суффиксов, начинающихся с позиций [10, 12]. Этот массив суффиксов затем объединяется с массивом SA^3, находящимся на диске и содержащим суффиксы, начинающиеся с $T[1, 9]$ (см. рис. 10.6). Объединенный массив является массивом суффиксов всей строки $T[1, 12]$.

Этап 2

1) Загрузка во внутреннюю память $T^2 = T[7, 9]$ = `sip`

2) Построение массива SA' для суффиксов, начиная с [7, 9]

Текстовые суффиксы	`sippi$`	`ippi$`	`ppi$`
	⇓		
	Лексикографическое упорядочение		
	⇓		
Отсортированные суффиксы	`ippi$`	`ppi$`	`sippi$`
SA′	8	9	7

3) Слияние SA' и SA^2 с помощью массива C

Массивы суффиксов	$SA' = [8, 9, 7]$ $SA^2 = [\mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$
	⇓ $C = [0, 3, 0, 3]$
Слияние с помощью массива *C*	$SA^3 = [8, \mathbf{5}, \mathbf{2}, \mathbf{1}, 9, 7, \mathbf{4}, \mathbf{6}, \mathbf{3}]$

Рис. 10.5. Второй этап алгоритма на базе сканирования

Есть несложный прием, который позволит улучшить заявленную в теореме 10.4 асимптотическую производительность алгоритма на базе сканирования. Предположим, что в начале этапа h вдобавок к массиву SA^h на диске имеется битовый массив $\mathtt{gt}_h$, причем $\mathtt{gt}_h[i] = 1$ тогда и только тогда, когда суффикс $T[(hm + 1) + i, n]$ больше суффикса $T[(hm + 1), n]$. Другими словами, текстовый суффикс, начинающийся с i-го символа подстроки T^h, больше текстового суффикса, начинающегося с ее первого символа. Вычисление `gt` можно сделать эффективным с точки зрения ввода-вывода. Технические подробности вы найдете в оригинальной статье [7], здесь я на них останавливаться не буду.

На этапе h алгоритм загружает во внутреннюю память подстроку $t[1, 2m] = T^h T^{h+1}$ (то есть вдвое большую, чем раньше) и двоичный массив $\mathtt{gt}_{h+1}[1, m - 1]$ (он относится ко второму блоку текста, загруженного во внутреннюю память). Что интересно, построение SA' для суффиксов, начинающихся в подстроке T^h, можно выполнить,

развернув два массива без каких-либо операций ввода-вывода, кроме необходимых для загрузки $t[1, 2m]$ и $\texttt{gt}_{h+1}[1, m-1]$. Это вытекает из того факта, что любые два текстовых суффикса, начинающихся в позициях i и j в подстроке T^h, где $i < j$, можно сравнить лексикографически, первым делом рассмотрев некоторые из символов подстроки t, а именно подстроки $t[i, m]$ и $t[j, j + m - i]$. Эти две подстроки имеют одинаковую длину и целиком находятся внутри $t[1, 2m]$, то есть во внутренней памяти. Если эти строки различны, мы определяем их порядок и процесс завершается. В противном случае их порядок определяется порядком остальной части их суффиксов, начиная с символов $t[m + 1]$ и $t[j + m - i + 1]$. Он задается битом, хранящимся в $\texttt{gt}_{h+1}[j - i]$, также доступным во внутренней памяти. Массивы t и $\texttt{gt}_{h+1}$ содержат всю информацию, необходимую для построения SA^{h+1} исключительно во внутренней памяти, то есть не прибегая к дополнительным операциям ввода-вывода.

Этап 3

1) Загрузка во внутреннюю память $T^3 = T[10, 12] = \texttt{pi\$}$

2) Построение массива SA' для суффиксов, начиная с [10, 12]

Текстовые суффиксы	pi$	i$	$
	⇓		
	Лексикографическое упорядочение		
	⇓		
Отсортированные суффиксы	$	i$	pi$
SA′	8	9	7

3) Слияние SA' и SA^3 с помощью массива C

Массивы суффиксов	$SA' = [12, 11, 10]$ $SA^3 = [\mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$
Слияние с помощью массива *C*	⇓ $C = [0, 0, 4, 5]$
	$SA^4 = [12, 11, \mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, 10, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$

Рис. 10.6. Третий этап алгоритма на базе сканирования

Теорема 10.5. *Новый вариант алгоритма на базе сканирования создает массив суффиксов строки $T[1, n]$ за $O\left(\frac{n^2}{MB}\right)$ операций ввода-вывода, используя $O(n/B)$ страниц диска.*

В качестве примера рассмотрим этап $h = 1$, во время которого во внутреннюю память загружаются подстрока $t = T^1T^2 = T[4, 9] = \texttt{sis sip}$ и массив $\texttt{gt}_2 = (0, 0)$. Содержимое $\texttt{gt}_2$ определяется тем, что $\texttt{gt}_2[1] = 0$, поскольку $T^2[1 + 1, \cdot] = \texttt{ippi\$} < T^2[1, \cdot] = \texttt{sippi\$}$, и $\texttt{gt}_2[2] = 0$, поскольку $T^2[1 + 2, \cdot] = \texttt{pi\$} < T^2[1, \cdot] = \texttt{sippi\$}$. Теперь сравним текстовые

суффиксы в t, начинающиеся с позиций $i = 1$ и $j = 2$. Для этого возьмем подстроки $t[1, 3] = T[4, 6] =$ `sis` и $t[3, 5] = T[6, 9] =$ `ssi`. Так как они различны, определить их порядок можно, не обращаясь к диску. А вот позиции $i = 3$ и $j = 4$ в t не будут учитываться алгоритмом, потому что блок имеет размер 3. Тем не менее рассмотрим их для наглядности. Для сравнения начинающихся с этих позиций текстовых суффиксов возьмем подстроки $t[3, 3] =$ `s` и $t[4, 4] =$ `s`. Поскольку они совпадают, мы используем $\texttt{gt}_2[j - i] = \texttt{gt}_2[1] = 0$. Следовательно, оставшийся $(j - i)$-й суффикс $T[8, \cdot] =$ `ippi$` лексикографически меньше первого суффикса $T[7, \cdot] =$ `sippi$`. В данном случае нам снова не потребовались никакие дополнительные операции ввода-вывода.

10.3. Суффиксное дерево

Суффиксным деревом называется фундаментальная структура данных, используемая во многих алгоритмах, обрабатывающих строки переменной длины [9]. По сути, это сжатое префиксное дерево, которое хранит все суффиксы входной строки. Каждый суффикс представлен уникальным путем от корня дерева к одному из листьев. Сжатые префиксные деревья обсуждались в главе 9, теперь поговорим о них в контексте словаря строк, которые являются суффиксами одной-единственной строки.

Обозначим суффиксное дерево, построенное по входной строке $T[1, n]$, как ST_T (или просто ST, если ввод ясен из контекста) и предположим, что, как это бывает в массивах суффиксов, последний символ этой строки — специальный символ \$, который меньше любого другого символа алфавита. Суффиксное дерево обладает следующими свойствами.

1. Все суффиксы строки T представлены *уникальными* путями, нисходящими от корня ST к одному из листьев. То есть мы имеем n листьев, по одному на каждый текстовый суффикс, причем каждый лист помечен начальной позицией в строке T соответствующего ему суффикса.
2. Каждый внутренний узел ST имеет по крайней мере два исходящих ребра, поскольку это сжатое префиксное дерево. То есть у нас менее n внутренних узлов и менее $2n - 1$ ребер. Каждый внутренний узел u описывает текстовую подстроку $s[u]$, которая является префиксом для всех суффиксов дерева, происходящих от u. Обычно значение $|s[u]|$ хранится как сопутствующая информация узла u, а для указания на листья и их текстовые позиции используем обозначение $occ[u]$.
3. Метки ребер представляют собой непустые подстроки строки T. Метки ребер, выходящих из внутренних узлов, начинаются с разных символов, называемых *символами ветвления*. Предполагается, что ребра упорядочены по алфавиту в соответствии с их символами ветвления и каждый узел имеет не более чем σ исходящих ребер[1].

[1] Специальный символ \$ включен в алфавит Σ.

Суффиксное дерево, построенное по образцу текста $T[1, 12]$ = `mississippi$`, демонстрирует рис. 10.7. Специальный символ $T[12]$ = \$ гарантирует, что ни один суффикс не является префиксом другого суффикса строки T, то есть все пары суффиксов различаются в каких-то символах и с каждым суффиксом связан свой лист. Соответственно, пути от корня к листьям двух разных суффиксов совпадают до их самого длинного общего префикса, который заканчивается во внутреннем узле ST.

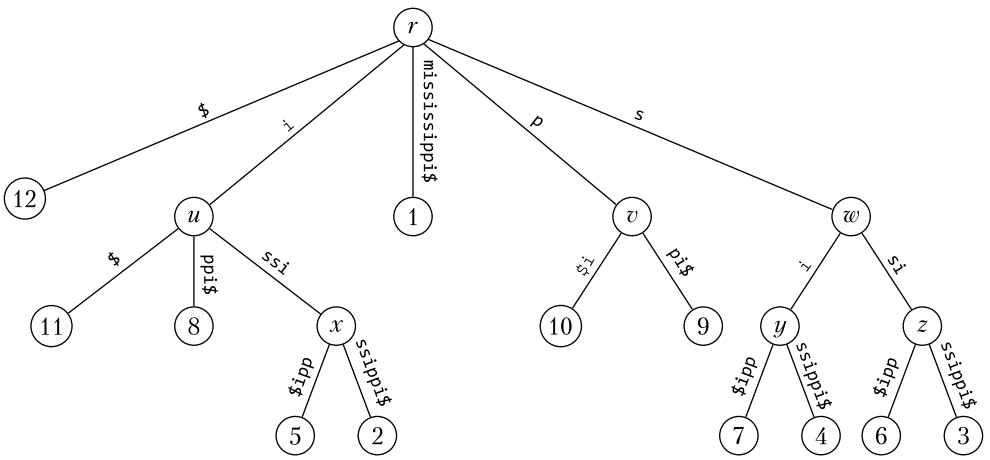


Рис. 10.7. Суффиксное дерево строки T = mississippi\$

Очевидно, что подстроки, помечающие ребра суффиксного дерева, невозможно хранить в явном виде, потому что это может занять пространство $\Theta(n^2)$. Если строка T состоит из n различных символов, то суффиксное дерево будет состоять из корня, соединенного с n листьями с помощью ребер, представляющих все суффиксы.

Для уменьшения занимаемого пространства можно кодировать метки ребер парами целых чисел: начальной позицией маркированной подстроки и ее длиной. Например, на рис. 10.7 метка ребра, ведущего к листу 5, то есть $T[9, 12]$ = `ppi$`, может быть представлена как $\langle 9, 4\rangle$, где 9 — смещение в строке T, а 4 — длина метки ребра. Возможны и другие очевидные пары, например $\langle 9, 12\rangle$, указывающая начальную и конечную позиции метки ребра, но ее я подробно рассматривать не буду. В любом случае, какую бы кодировку меток ребер мы ни приняли, она будет занимать постоянное пространство. Соответственно, для хранения всех меток ребер потребуется пространство $O(n)$ независимо от рассматриваемой строки.

Факт 10.3. Суффиксное дерево строки $T[1, n]$ состоит из n листьев, не более чем $n - 1$ внутренних узлов и не более чем $2n - 2$ ребер. При условии корректной кодировки меток ребер, имеющих постоянный размер, занимаемое деревом пространство оценивается как $\Theta(n)$.

В заключение познакомлю вас с еще двумя терминами. *Местоположением* (locus) текстовой подстроки t называется узел v, отвечающий условию $s[v] = t$. *Расширенным местоположением* (extended locus) текстовой подстроки t' называется позиция ее кратчайшего расширения с определенным местоположением в дереве ST. Другими словами, путь, записывающий строку t' в ST, заканчивается внутри метки ребра, скажем метки (u, v). Тогда $s[u]$ добавляет префикс t', который, в свою очередь, добавляет префикс $s[v]$. Конечно, если подстрока t' расположена в дереве ST, это совпадает с ее расширенным местоположением. Например, узел z суффиксного дерева на рис. 10.7 — это местоположение подстроки `ssi` и расширенное местоположение подстроки `ss`.

У суффиксного дерева есть и другие важные свойства, они используются в большинстве алгоритмов, зависящих от этой мощной структуры данных. Вот некоторые из них.

Свойство 2. Пусть α — подстрока текста T. Внутренний узел u такой, что $s[u] = \alpha$ (следовательно, u — местоположение α), существует только при условии, что строка T имеет по крайней мере два вхождения α, за которыми следуют различные символы.

В качестве примера возьмем узел x на рис. 10.7: подстрока $s[x]$ = `issi` встречается в T дважды в позициях 2 и 5, за ней следуют символы `i` и `p` соответственно.

Свойство 3. Если α — подстрока текста T с расширенным местоположением в суффиксном дереве, то за каждым ее вхождением в T следует один и тот же символ.

В качестве примера возьмем подстроку `iss`, для которой узел x на рис. 10.7 является расширенным местоположением. Эта подстрока встречается в T дважды в позициях 2 и 5, оба раза сопровождаясь символом `i`.

Свойство 4. Каждый внутренний узел u определяет подстроку $s[u]$ слова T, которая встречается в позициях $occ[u]$ и является *максимальной* в том смысле, что в этих позициях она не допускает расширения ни на один символ.

Теперь введем понятие *наименьшего общего предка* (lowest common ancestor, `lca`), который определяется для каждой пары листьев и обозначает самый глубокий узел, который является их общим предком. Например, на рис. 10.7 узел u — это `lca` листьев 8 и 2. Его можно превратить в самый длинный общий префикс (longest common prefix, lcp) между соответствующими суффиксами.

Свойство 5. Если $a(i, j)$ — наименьший общий предок между листьями, соответствующими суффиксам $T[i, n]$ и $T[j, n]$, то $s[a(i, j)]$ равен их префиксу, имеющему длину $lcp(T[i, n], T[j, n])$.

В качестве примера возьмем суффиксы $T[11, 12]$ = `i$` и $T[5, 12]$ = `issippi$`. Их самый длинный общий префикс состоит из единственного символа i, ведь `lca` между их листьями — это узел u, которому соответствует строка $s[u]$ = `i`.

10.3.1. Поиск подстрок

Поиск шаблона $P[1, p]$ как подстроки текста $T[1, n]$ с помощью суффиксного дерева ST состоит из обхода дерева, стартующего в корне и идущего вниз по мере сопоставления символов шаблона с метками ребер дерева (рис. 10.8). Обратите внимание на то, что из-за разницы в первых символах ребер, исходящих из каждого пройденного узла, сопоставление с шаблоном P может проходить только по одному нисходящему пути. Обнаружение несовпадающего символа означает, что шаблон P не встречается в строке T, в противном случае шаблон сопоставляется полностью, находится расширенное местоположение шаблона P, например узел u, и все спускающиеся из этого узла листья дерева ST идентифицируют все начинающиеся с P текстовые суффиксы. Связанные с этими листьями позиции $occ[u]$ указывают на вхождения occ шаблона P в подстроку T. Эти позиции можно извлечь за оптимальное линейное время, обойдя поддерево с корнем в узле u. Его размер — $O(occ)$, поскольку состоит из occ листьев, а его внутренние узлы имеют как минимум двоичное ветвление.

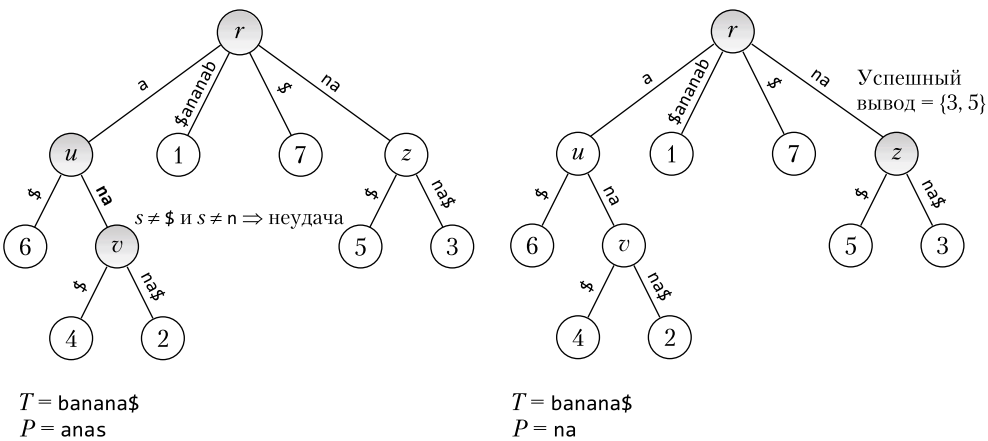


Рис. 10.8. Два примера поиска подстрок по суффиксному дереву для текста T = banana$. *Слева* при поиске шаблона P = anas происходит сбой в узле v, *справа* поиск шаблона P = na успешно завершается в узле z

В примере на рис. 10.8 шаблон P = na встречается в T дважды, поэтому обход дерева ST обнаруживает полное совпадение с P и останавливается в узле z, который представляет собой местоположение P. Из этого узла спускаются два листа с метками 3 и 5. Это позиции в T, в которых встречается шаблон P. Он предшествует двум суффиксам, $T[3, 12]$ = nana$ и $T[5, 12]$ = na$.

На такой поиск в худшем случае требуется время $O(pt_\sigma + occ)$, где t_σ — время, затрачиваемое на выбор ветви при выходе из узла в процессе обхода дерева. Затраты зависят от размера алфавита σ и структуры данных, используемой для хранения символов ветвления ребер. Этот вопрос обсуждался в главе 9 при рассмотрении

префиксного поиска с помощью сжатых префиксных деревьев. Оказалось, что если использовать для хранения символов ветвления идеальную хеш-таблицу, то $t_\sigma = O(1)$. А вот если они хранятся в обычном массиве, а ветвление реализуется бинарным поиском, то $t_\sigma = O(\log \sigma)$. Но в обоих случаях занимаемое пространство будет оптимальным, то есть линейно зависящим от количества ребер ветвления, таким образом, в целом $O(n)$. Впрочем, как я уже упоминал, идеальные хеш-таблицы не поддерживают эффективный лексикографический поиск.

Факт 10.4. Если поиск вхождений *occ* шаблона $P[1, p]$ в текст $T[1, n]$ реализуется с помощью суффиксного дерева, в котором символы ветвления в каждом узле индексируются с помощью идеальной хеш-таблицы, его временная и пространственная сложность будет оцениваться как $O(p + occ)$ и $O(n)$ соответственно.

10.3.2. Построение массивов суффиксов из суффиксных деревьев и наоборот

Нетрудно заметить, что массив суффиксов *SA* текста *T* и соответствующий ему массив `lcp` можно получить обходом его суффиксного дерева *ST*. Достаточно записать индекс суффикса каждого листа в массив *SA*, а связанное с внутренним узлом *u* значение ($|s[u]|$) — в массив `lcp`.

Факт 10.5. Из суффиксного дерева строки $T[1, n]$ можно построить соответствующий массив суффиксов *SA* и массив самых длинных общих префиксов `lcp` за время $O(n)$, используя при этом пространство $O(n)$.

Возможен и обратный процесс. Из двух массивов, *SA* и `lcp`, за время $O(n)$ строится суффиксное дерево *ST*. Алгоритм создает его, начиная, например, с поддерева ST_1, которое содержит корень с пустой строкой и лист $SA[1]$ с наименьшим суффиксом строки *T*. На шаге $i > 1$ индуктивно будет получено частичное дерево ST_{i-1} со всеми $(i - 1)$-ми наименьшими суффиксами строки *T*. Они будут в лексикографическом порядке содержаться в массиве $SA[1, i - 1]$. На шаге *i* алгоритм вставит в дерево ST_{i-1} *i*-й наименьший суффикс, добавив для этого дополнительный лист $SA[i]$. Кроме того, как я покажу позже, будет добавлен максимум один внутренний узел, который станет предком листа $SA[i]$. Через *n* итераций сформируется дерево ST_n, которое и будет суффиксным деревом строки $T[1, n]$.

Здесь важно показать, что вставка листа $SA[i]$ в дерево ST_{i-1} происходит за постоянное амортизированное время. Этого достаточно, чтобы обеспечить общую временную сложность $O(n)$ для всего процесса построения. Сложнее всего найти узел *u*, который является предком листа $SA[i]$. Если он уже существует в дереве ST_{i-1}, то к нему присоединен лист $SA[i]$. В противном случае этот узел придется создавать расщеплением ребра ST_{i-1}. Существует *u* или нет, выясняется при обходе вверх дерева ST_{i-1}, начиная с самого правого листа $SA[i - 1]$. Обход останавливается при достижении узла *x*, удовлетворяющего условию $\texttt{lcp}[i - 1] \leqslant |s[x]|$. Напомню, что $\texttt{lcp}[i - 1]$ — это количество общих символов у текущего вставленного суффикса $suff_{SA[i]}$ и ранее

вставленного суффикса $suff_{SA[i-1]}$. Соответствующие этим двум суффиксам листья последовательно посещаются при упорядоченном обходе дерева *ST*. Если в этот момент выполняется условие $\texttt{lcp}[i-1] = |s[x]|$, значит, узел x является предком листа $SA[i]$. Их соединение дает новое частичное дерево ST_i. Если же $\texttt{lcp}[i-1] < |s[x]|$, то ведущее к узлу x ребро следует разделить путем вставки узла u, имеющего двух потомков: левый потомок — узел x, а правый — лист $SA[i]$ (потому что лексикографически он больше, чем $SA[i-1]$). Этот узел связан со значением $\texttt{lcp}[i-1]$. Попробуйте самостоятельно проверить этот алгоритм для строки $T[1, 12] = \texttt{mississippi\$}$ и убедиться, что окончательное дерево суффиксов ST_{12} именно то, которое показано на рис. 10.7.

Временную сложность этого алгоритма можно оценить подсчетом ребер, пройденных при восходящем обходе дерева *ST*. Поскольку суффикс $suff_{SA[i]}$ лексикографически больше суффикса $suff_{SA[i-1]}$, лист $SA[i]$ находится справа от листа $SA[i-1]$. Поэтому после прохода каждого ребра оно или удаляется из следующих проходов и движение вверх продолжается, или делится для вставки нового листа. В частности, все ребра от $SA[i-1]$ до x никогда не будут снова проходиться, потому что располагаются слева от вставленного ребра $(u, SA[i])$. Общее количество этих ребер ограничено общим количеством ребер в дереве *ST*, которое, как следует из факта 10.3, равно $O(n)$. Общее количество делений ребер равно количеству вставленных листьев, что снова составляет $O(n)$.

Теорема 10.6. *Из массива суффиксов и массива самых длинных общих префиксов строки* $T[1, n]$ *соответствующее суффиксное дерево можно получить с временными и пространственными затратами* $O(n)$.

Таким образом, для построения суффиксного дерева строки $T[1, n]$ следует с помощью skew-алгоритма получить массив суффиксов T за время $O(n)$ (см. теорему 10.3, где применяется процедура `RADIXSORT` в памяти). Затем нужно сформировать массив `lcp` за время $O(n)$ (см. теорему 10.1) и, наконец, воспользоваться описанным в этом разделе алгоритмом для построения дерева *ST* (см. теорему 10.6). Для целочисленного алфавита или алфавита постоянного размера весь процесс происходит за оптимальное время $O(n)$, а в модели на базе сравнения осуществляется за оптимальные $O(\texttt{Sort}(n))$ шагов.

Рассмотрим классический алгоритм *прямого* построения суффиксных деревьев, который, согласно теореме 10.7, имеет временную сложность $O(n \log \sigma)$, но не очень эффективен с точки зрения сложности ввода-вывода. В настоящее время компактность массивов суффиксов и наличие skew-алгоритма побуждают программистов строить суффиксные деревья с помощью массивов суффиксов. Но при небольшом *среднем* LCP среди суффиксов прямое построение суффиксного дерева все еще может быть выгодным как во внутренней памяти, так и на диске. Более глубокий анализ этих вопросов вы найдете в [6], а [4] содержит описание оптимального по вводу-выводу алгоритма прямого построения суффиксных деревьев, который слишком сложен для обсуждения на этих страницах.

10.3.3. Алгоритм Маккрейта∞

Наивный алгоритм построения суффиксного дерева входной строки $T[1, n]$ можно начать с итеративной вставки текстовых суффиксов в пустое префиксное дерево. При этом каждое промежуточное префиксное дерево — это сжатое префиксное дерево ранее вставленных суффиксов. В худшем случае, когда у нас есть строка с большим количеством повторений $T[1, n] = a^{n-1}\$$, работа алгоритма может занять время $O(n^2)$. Причина такого плохого поведения заключается в *повторном сканировании* некоторых подстрок T, уже проверенных в процессе вставки предыдущих суффиксов. В 1976 году Маккрейт предложил ныне хорошо известный алгоритм [15], который с помощью добавленных к суффиксному дереву специальных указателей избегает повторного сканирования.

Эти специальные указатели называются *суффиксными ссылками* (suffix link) и определяются следующим образом. Суффиксная ссылка $SL(z)$ соединяет узел z с узлом z' таким образом, что $s[z] = as[z']$, где a — произвольный символ алфавита. То есть строка, которая формируется по пути к узлу z', получается из строки $s[z]$ отбрасыванием ее первого символа. Существование в дереве ST узла z' вовсе не очевидно. Конечно, так как в T имеется подстрока $s[z]$, то $s[z']$ тоже будет подстрокой строки T. Таким образом, в дереве ST существует путь, который заканчивается расширенным местоположением $s[z']$. Но, похоже, ничто не гарантирует, что местоположение подстроки $s[z']$ действительно находится в дереве ST и узел z' существует. Это свойство вытекает из того, что существование узла z подразумевает существование по крайней мере двух суффиксов, скажем $suff_i$ и $suff_j$, для которых узел z является наименьшим общим предком в дереве ST, и, таким образом, $s[z]$ — это их самый длинный общий префикс (см. свойство 5). Отбросив первый символ в $s[z]$, мы получим подстроку $s[z']$, которая по построению, несомненно, будет самым длинным общим префиксом $suff_{i+1}$ и $suff_{j+1}$. Таким образом, узел z' оказывается наименьшим общим предком листьев, соответствующих этим двум суффиксам.

Для узла z с подстрокой $s[z]$ = `ssi` (см. рис. 10.7) возьмем суффиксы $suff_3$ и $suff_6$, у которых z — наименьший общий предок. Рассмотрим следующие за ними суффиксы, то есть $suff_4$ и $suff_7$. Их самый длинный общий префикс `si`, ведь мы отбросили первый символ подстроки $s[z]$, а в суффиксном дереве присутствует узел y, играющий роль z', такой, что $s[y]$ = `si`, и это их наименьший общий предок. В заключение скажу, что у каждого узла z есть корректно определенная суффиксная ссылка. При этом совокупность всех суффиксных ссылок образует дерево, корень которого совпадает с корнем ST. Обратите внимание на то, что $|s[z']| < |s[z]|$, поэтому они не могут образовывать циклы и в итоге оказываются в корне суффиксного дерева, формируя пустую строку.

Алгоритм Маккрейта работает за n шагов. Он начинается с суффиксного дерева ST_1, которое состоит из корневого узла с пустой строкой и единственного листа, обозначенного $suff_1 = T[1, n]$ (это весь наш текст). На итерации $i > 1$ текущее суффиксное дерево ST_{i-1} представляет собой сжатое префиксное дерево, построенное по всем суффиксам $suff_j$ таким образом, что $j = 1, 2... i - 1$. Следовательно, суффиксы

вставляются в ST от самого длинного до самого короткого и на любом шаге дерево ST_{i-1} индексирует $(i-1)$ самых длинных суффиксов строки T.

Для упрощения описания алгоритма введем параметр $head_i$, обозначающий самый длинный префикс суффикса $suff_i$, встречающийся в ST_{i-1}. Так как ST_{i-1} — это частичное суффиксное дерево, $head_i$ будет самым длинным общим префиксом для $suff_i$ и любого из его предыдущих суффиксов в T, а именно $suff_j$, где $j = 1, 2... i-1$. Обозначим h_i расширенное местоположение строки $head_i$ в текущем суффиксном дереве ST_{i-1}, поскольку суффикс $suff_i$ еще не вставлен. После его вставки получим, что $head_i = s[h_i]$ в дереве ST_i, поэтому h_i является местоположением $head_i$ и установлен как предок связанного с суффиксом $suff_i$ листа. В качестве примера рассмотрим вставку суффикса $suff_5$ = `byabz$` в частичное суффиксное дерево ST_4 (рис. 10.9). Этот суффикс разделяет с предыдущими четырьмя суффиксами T только символ `b`, поэтому в частичном дереве ST_4 общий префикс $head_5$ = `b` и имеет расширенное местоположение, заданное листом 2. Но после вставки суффикса $suff_5$ мы получим новое суффиксное дерево ST_5, в котором $h_5 = v$, и это местоположение $head_5$.

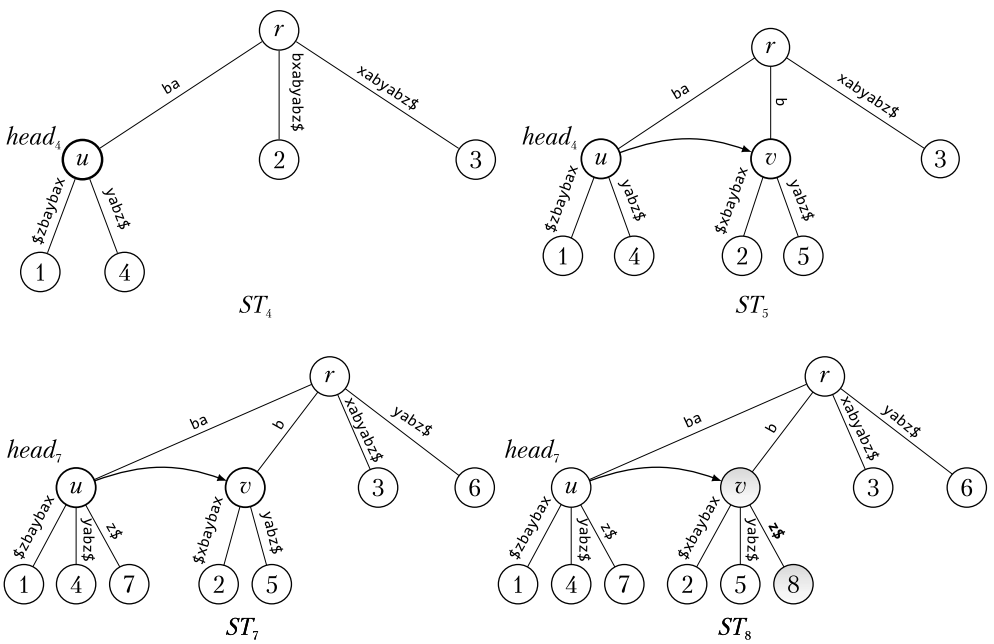


Рис. 10.9. Некоторые шаги алгоритма Маккрейта для строки T = abxabyabz$

Теперь все готово для того, чтобы подробно описать алгоритм Маккрейта. Для получения ST_i нужно найти в частичном дереве ST_{i-1} местоположение (в том числе расширенное) h_i для $head_i$. Если местоположение расширенное, то связанное с этим узлом ребро будет разбито вставкой внутреннего узла, который соответствует h_i и формирует строку $head_i$ с прикрепленным к ней листом для суффикса $suff_i$. Оказалось, что в наивном алгоритме $head_i$ и h_i, перемещаясь вниз по частичному

дереву ST_{i-1}, посимвольно совпадают с суффиксом $suff_i$. Но, как я уже говорил, при таком подходе мы имеем квадратичную в худшем случае временную сложность. Поэтому алгоритм Маккрейта определяет $head_i$ и h_i с помощью информации, индуктивно доступной для строки $head_{i-1}$ и ее местоположения h_{i-1}, а также с помощью уже доступных в ST_{i-1} *суффиксных ссылок*.

Факт 10.6. В частичном дереве ST_{i-1} суффиксная ссылка $SL(u)$ определена для всех узлов $u \neq h_{i-1}$. Может оказаться, что суффиксная ссылка $SL(h_{i-1})$ также уже определена, потому что этот узел присутствовал в ST_{i-1} до вставки суффикса $suff_{i-1}$.

Доказательство. Первое утверждение вытекает из способа индуктивного построения частичного дерева ST_{i-1}, а вопрос о возможном существовании $SL(h_{i-1})$ — из того, что h_{i-1} — местоположение префикса $head_{i-1}$, предшествующего суффиксу $suff_{i-1}$. Таким образом, второй суффикс для $head_{i-1}$ начинается с позиции i и предшествует суффиксу $suff_i$. Обозначу второй суффикс $head_{i-1}^-$, чтобы подчеркнуть, что мы отбросили первый символ из $head_{i-1}$. Строка $head_i$ по определению является самым длинным префиксом, общим для $suff_i$ и любого из предыдущих текстовых суффиксов, так что строка $head_{i-1}^-$ предшествует $head_i$ и поэтому уже находится в ST_{i-1} и может иметь местоположение в этом сжатом префиксном дереве. ■

Алгоритм Маккрейта начинается с ST_1, состоящего из двух узлов: корня и листа для суффикса $suff_1$. На первом шаге $head_1$ — пустая строка, h_1 — корень, а суффиксная ссылка $SL(root)$ указывает на корневой узел. На общем шаге $i > 1$ известны префикс $head_{i-1}$ и его местоположение h_{i-1} (предок суффикса $suff_{i-1}$) и нужно определить $head_i$ и h_i, чтобы вставить лист для суффикса $suff_i$ как дочернего элемента h_i. Эти данные можно получить так.

1. Если суффиксная ссылка $SL(h_{i-1})$ определена, присваиваем ее местоположению w и переходим к этапу 3.
2. Если она не определена, выполняем *повторное сканирование*, чтобы найти/создать местоположение w для строки, полученной отбрасыванием первого символа префикса $head_{i-1}$. Она обозначена $head_{i-1}^-$. Для поиска w берем предка f для h_{i-1}, переходя по его суффиксной ссылке $f' = SL(f)$ (согласно факту 10.6, она определена), и идем вниз от f', начиная с $(|s[f']| + 1)$-го символа суффикса $suff_i$. Так как суффикс $head_{i-1}^-$ встречается в *строке* T и предшествует суффиксу $suff_i$, для нисходящей трассировки достаточно сравнить с ним только символы ветвления пройденных ребер. Если конечный узел этого обхода является местоположением $head_{i-1}^-$, то именно он — искомое местоположение w. Если это не так, то он будет расширенным местоположением $head_{i-1}^-$, а значит, нужно разбить последнее пройденное ребро и вставить узел w таким образом, чтобы $s[w] = head_{i-1}^-$. Во всех случаях мы устанавливаем $SL(h_{i-1}) = w$.
3. Наконец, начиная с местоположения w, мы локализуем $head_i$ и сканируем остаток суффикса $suff_i$. Если местоположение $head_i$ существует, присваиваем его h_i, в противном случае сканирование $head_i$ останавливается на каком-то ребре, которое мы разбиваем, вставляя h_i как местоположение $head_i$. Процесс завершается тем, что лист для $suff_i$ устанавливается как дочерний элемент h_i.

Пример преимущества, которое дают суффиксные ссылки, демонстрируется на рис. 10.9. На шаге 8 мы имеем частичное суффиксное дерево ST_7, $head_7$ = `ab`, его местоположение $h_7 = u$. Нужно вставить суффикс $suff_8$ = `bz$`. Алгоритм Маккрейта показывает, что суффиксная ссылка $SL(h_7)$ определена и равна v, то есть до следующего за ней узла можно добраться без повторного сканирования $head_{i-1}^-$. Остается просканировать оставшуюся часть $suff_8$, а именно `z$`, чтобы найти местоположение $head_8$. Но на самом деле $head_8 = head_7^-$, а значит, $h_8 = v$ и туда можно прикрепить лист 8.

С точки зрения временной сложности при первом и повторном сканировании выполняются различные типы обходов. В первом случае мы идем по ребрам, сравнивая только символы ветвления, поскольку алгоритм повторно сканирует строку $head_{i-1}^-$, которая стала известна на предыдущем, i – 1-м шаге. Во втором случае алгоритм проходит по ребрам, сравнивая их метки целиком, ведь ему требуется определить $head_i$. Этот последний тип обхода всегда сопровождается продвижением по строке T, поэтому затраты на сканирование составляют $O(n)$. Трудность состоит в том, чтобы показать, что стоимость повторного сканирования также составляет $O(n)$. Доказательство вытекает из структуры суффиксных ссылок и суффиксных деревьев. Когда $SL(u) = v$, все предки u указывают на определенного предка v. Это следует из факта 10.6 (существование всех суффиксных ссылок) и определения суффиксных ссылок (обеспечение иерархических связей). Следовательно, глубина дерева $v = SL(u)$, скажем $d[v]$, больше, чем $d[u] - 1$ (–1 появилась из-за отбрасывания первого символа). Повторное сканирование может уменьшить текущую глубину максимум на 2 (один для достижения предка h_{i-1} и один для прохода через $SL(h_{i-1})$). Глубина дерева ST не превышает n, а каждый проход по суффиксной ссылке убирает не более двух уровней, соответственно, количество ребер, пройденных при повторном сканировании, будет равно $O(n)$, а на каждый обход ребра уйдет время $O(1)$, поскольку сопоставляется только символ ветвления.

Напоследок нужно рассмотреть затраты, связанные с ветвлением узла во время повторного и первичного сканирования. Ранее я утверждал, что они постоянны благодаря идеальным хеш-таблицам, построенным на базе символов ветвления каждого внутреннего узла ST. Поскольку мы строим суффиксное дерево, причем оно динамическое, нужно использовать динамические идеальные хеш-таблицы, что довольно сложно. Намного проще взять для хранения символов ветвления и связанных с ними ребер двоичное дерево поиска. В этом случае временные затраты на ветвление составят $O(\log \sigma)$. На практике требование к временной сложности в худшем случае ослабляют, прибегая к хеш-таблицам с цепочками, кукушкиному хешированию (см. главу 8), словарным структурам данных для целочисленных значений (дерево Ван Эмде Боаса с временной сложностью поиска $O(\log \log \sigma)$), ведь символы можно рассматривать как последовательности битов, то есть как целые числа.

Теорема 10.7. *В худшем случае временная и пространственная сложность построения суффиксного дерева строки $T[1, n]$ алгоритмом Маккрейта составляет $O(n \log \sigma)$ и $O(n)$ соответственно.*

При использовании внешней памяти этот алгоритм неэффективен, так как обход каждого ребра может сопровождаться одной операцией ввода-вывода. Но, как я показывал ранее, распределение длин $head_i$ можно сместить в сторону малых значений, сделав эту конструкцию эффективной с точки зрения ввода-вывода. Если кэшировать верхнюю часть суффиксного дерева во внутреннюю память, никакие операции ввода-вывода во время сканирования и повторного сканирования не понадобятся. Подробно этот вопрос освещался в [6].

10.4. Несколько интересных задач

10.4.1. Нечеткий поиск

Задачу нечеткого поиска, или приближенного сопоставления с шаблоном, можно сформулировать следующим образом: *найти все подстроки текста* $T[1, n]$, *совпадающие с шаблоном* $P[1, p]$ *максимум с* k *ошибками*. В этом разделе я ограничусь простейшим типом ошибок, которые называются *несовпадениями* (mismatches) или *подстановками* (substitutions). Здесь текстовые подстроки длиной p, имеющие k отличий от искомого образца P, совпадают с образцом везде, кроме некоторых символов, максимальное количество которых k. На рис. 10.10 представлены две строки ДНК, состоящие из четырех нуклеотидных оснований {A, T, G, C}, ведь интерес к нечеткому сопоставлению строк зародился именно в биоинформатике. Здесь шаблон P встречается в строке T, в позиции 1, с двумя несовпадениями, указанными стрелками.

```
C  C  G  T  A  C  G  A  T  C  A  G  T  A
C  C  G  A  A  C  T
         ⇑        ⇑
```

Рис. 10.10. Пример нечеткого соответствия между текстовой строкой *T (вверху)* и шаблоном *P (внизу)* с $k = 2$ несовпадениями

Простейшее решение сводится к сопоставлению шаблона P с каждой возможной подстрокой длиной p из T, подсчету несовпадений и возвращению позиций, в которых количество несовпадений меньше k. Это занимает время $O(pn)$ при любом значении k. Но такое решение неэффективно, потому что каждое сравнение шаблона и подстроки выполняется с начала P, в результате в худшем случае этот процесс требует времени $O(p)$. Поэтому сейчас я покажу вам сложное решение, зависящее от элегантной структуры данных, которая применяется, в частности, для *поиска минимума на отрезке* (range minimum query, RMQ). Эта структура данных легла в основу алгоритмических решений множества задач в области интеллектуального анализа данных, поиска информации, вычислительной биологии и т. п.

Алгоритм 10.4 находит варианты с максимум k несовпадениями за время $O(nk)$ благодаря следующей особенности. Если шаблон P встречается в T с $j \leqslant k$ несовпадениями,

то его можно выровнять с подстрокой T, имеющей ту же длину p, причем таким образом, что совпадут j или $j - 1$ подстрок, а j символов будут несовпадающими. Дело в том, что совпадающие подстроки и несовпадения чередуются. Например, на рис. 10.10 шаблон встречается в позиции 1 в тексте T с двумя несовпадениями, и две подстроки P на самом деле совпадают с соответствующими им подстроками T. Если в крайних позициях P нет ни одного совпадения, получим три совпадающие подстроки.

Алгоритм 10.4. Нечеткий поиск шаблона на основе вычислений LCP

```
совпадения = {}
for i = 1 to n do
  m = 0, j = 1;
  while m ⩽ k and j ⩽ p do
    l = lcp(T[i + j − 1, n], P[j, p]);
    j = j + l;
    if j ⩽ p then
      m = m + 1; j = j + 1;
    end if
  end while
  if m ⩽ k then
    совпадения = совпадения ∪ {i};
  end if
end for
return совпадения;
```

Это наблюдение позволяет прийти к следующему выводу: если сравнение шаблона с текстовой подстрокой происходит за постоянное время, то время, затрачиваемое на простейшее решение, будет $O(nk)$ вместо $O(np)$. Чтобы это наблюдение стало рабочим, перефразирую его следующим образом: если одна из совпадающих подстрок описывается выражением $T[i, i + \ell] = P[j, j + \ell]$, то ℓ — самый длинный общий префикс (LCP) между шаблоном и суффиксом, начинающимся с совпадающих позиций i и j. В алгоритме 10.4 показан вариант решения, время выполнения которого оценивается как $O(nk)$, когда вычисление `lcp` происходит за время $O(1)$. Подсчитав `lcp` для примера, приведенного на рис. 10.10, увидим, что шаблон $P[1, 7]$ встречается в позиции 1, поскольку он равен $T[1, 7]$ с двумя несовпадениями.

- Самый длинный общий префикс между $T[1, 14]$ = `CCGTACGATCAGTA` и $P[1, 7]$ = = `CCGAACT` — это `CCG` (`lcp`($T[1, 14]$, $P[1, 7]$) = 3), то есть $T[1, 3] = P[1, 3]$ с несовпадением в позиции 4.

- Самый длинный общий префикс между $T[5, 14]$ = ACGATCAGTA и $P[5, 7]$ = ACT — это AC (lcp($T[5, 14]$, $P[5, 7]$) = 2), то есть $T[5, 6] = P[5, 6]$ с несовпадением в позиции 7.

Каким образом в алгоритме 10.4 $lcp(T[i + j - 1, n], P[j, p])$ вычисляется за постоянное время? Как известно, суффиксные деревья и массивы суффиксов имеют встроенную информацию об LCP, но напомню, что эти структуры данных были построены из одной-единственной строки. Я имею в виду совместные суффиксы P и T. Для обхода этой трудности достаточно построить массив или дерево суффиксов на базе строки $X = T\#P$, где # — новый символ, в других местах не встречающийся. Таким образом, каждый расчет вида $lcp(T[i + j - 1, n], P[j, p])$ можно свести к вычислению LCP между суффиксами X, а именно $lcp(T[i + j - 1, n], P[n + 1 + j, n + 1 + p])$, ведь P начинается с позиции $n + 1$ строки X. Остается показать, как вычисления LCP могут быть выполнены за постоянное время для любой пары сравниваемых суффиксов. Этому посвящен следующий раздел.

10.4.2. Наименьший общий предок, запрос минимума на отрезке и декартово дерево

Рассмотрим суффиксное дерево ST_X и массив суффиксов SA_X на базе строки X = CCGTACGATCAGTA. Она не подпадает под формулу $X = T\#P$, так что в этом разделе я хочу показать ход рассуждений и алгоритмические решения для произвольной строки X.

Отталкиваться буду от свойства 5 из раздела 10.3, согласно которому существует сильная связь между вычислением самого длинного общего префикса для суффиксов строки X и поиском наименьшего общего предка (lca) для листьев суффиксного дерева ST_X. Рассмотрим процесс вычисления $lcp(X[i, x], X[j, x])$, где x — длина строки X. Узел u = lca($X[i, x]$, $X[j, x]$) в дереве суффиксов ST_X описывает LCP между двумя указанными суффиксами X. Соответственно, хранящееся в этом узле значение $|s[u]|$ — это искомая длина LCP. Ее можно получить также из массива суффиксов SA_X. В частности, возьмем лексикографические позиции i_p и j_p этих двух суффиксов в SA_X, скажем $SA_X[i_p] = i$ и $SA_X[j_p] = j$ (для простоты предполагается, что $i_p < j_p$). Из-за лексикографического порядка суффиксов строки X в подмассиве[1] lcp$[i_p, j_p - 1]$ *минимальное значение* будет равно $|s[u]|$, потому что именно в этом подмассиве хранятся значения из узлов суффиксного поддерева, спускающегося из u (см. подраздел 10.3.2). На самом деле нас интересует наименьшее значение, которое соответствует узлу с минимальной глубиной (то есть корню u) этого поддерева, именно поэтому вычисления минимальны. В результате узнать длину lcp за постоянное время можно двумя способами: с помощью вычисления lca в дереве ST_X или с помощью

[1] Напомню, что массив lcp имеет размер $x - 1$ и хранит в записи lcp$[i]$ длину самого длинного общего префикса между суффиксом $SA[i]$ и смежным суффиксом $SA[i + 1]$, где $i = 1, 2, ..., x - 1$.

вычисления RMQ в массиве `lcp` (и в заданном массиве суффиксов SA_X). Я покажу элегантное решение для второго случая, которое, в свою очередь, порождает элегантное решение для первого, так как связь между ними тесная. Пример, который будет обсуждаться в дальнейшем, приведен на рис. 10.11.

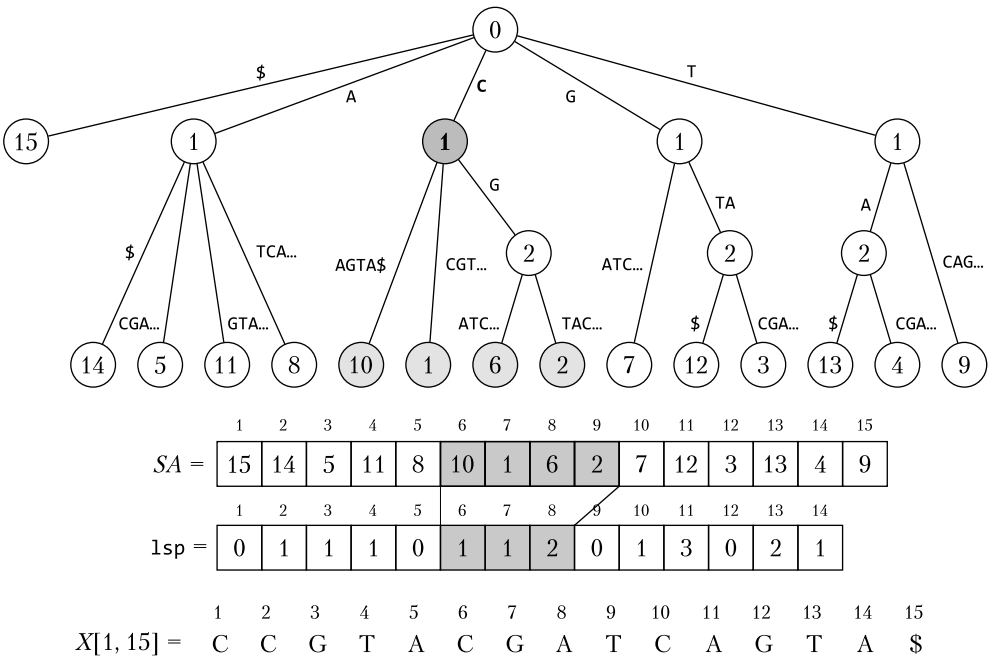


Рис. 10.11. Дерево суффиксов, массив суффиксов и массив lcp для строки X

На рис. 10.11 в суффиксном дереве указаны только префиксы для длинных меток ребер. Число во внутреннем узле u обозначает $|s[u]|$, тогда как числа в листьях обозначают начальные позиции соответствующих суффиксов. Показано, что вычисление $lcp(X[2, 14], X[10, 14]) = 1$, то есть совпадающего символа C, сводится к нахождению глубины узла lca в ST_X между листьями 2 и 10, а также поиску минимального диапазона для подмассива lcp[6, 8], потому что $SA_X[6] = 10$ и $SA_X[9] = 2$.

В общем виде RMQ-задача формулируется так.

> **Запрос минимума на отрезке** (range minimum query problem, RMQ). Из массива $A[1, n]$ элементов, взятых из упорядоченного пространства, построить структуру данных $\mathtt{RMQ}_A$, эффективно вычисляющую позицию одного из минимальных элементов в $A[i, j]$ для любого диапазона (i, j). Выражение «одного из» обусловлено тем фактом, что в подмассиве может оказаться несколько минимальных элементов.

Подчеркну, что задача состоит в определении *позиции* минимального элемента в подмассиве, а не его значения. В такой формулировке задача более универсальна, потому что, зная позицию элемента, получить его значение совсем не сложно.

Самое простое и наивное решение, обеспечивающее постоянное время выполнения RMQ-запросов, — это таблица, хранящая индекс минимальной записи для всех возможных диапазонов (i, j), где $1 \leqslant i \leqslant j \leqslant n$. Временная и пространственная сложность построения такой таблицы оценивается как $\Theta(n^2)$.

Лучшее решение базируется на возможности разложить любой диапазон (i, j) на два диапазона (возможно, перекрывающихся), размер которых является степенями двойки, а именно $(i, i + 2^L - 1)$ и $(j - 2^L + 1, j)$, где $L = \lfloor \log(j - i) \rfloor$. Это позволяет сделать предыдущую таблицу *разреженной*, сохранив только диапазоны размера, выраженного степенями двойки. В результате для каждой позиции i будут сохраняться ответы на запросы $\texttt{RMQ}_A(i, i + 2^l - 1)$ для $0 \leqslant l \leqslant \lfloor \log_2(n - i) \rfloor$. Такая разреженная таблица занимает пространство $O(n \log n)$ и отвечает на RMQ-запросы за постоянное время — достаточно вычислить $\texttt{RMQ}_A(i, j)$ как хранящую минимальное значение позицию между $\texttt{RMQ}_A(i, i + 2^L - 1)$ и $\texttt{RMQ}_A(j - 2^L + 1, j)$, где $L = \lfloor \log(j - i) \rfloor$.

Чтобы достичь оптимального заполнения пространства $O(n)$, нужно углубиться в структуру RMQ-задачи и провести две редукции, туда и обратно, между вычислениями RMQ и `lca`[3]. Во-первых, нужно перейти от вычисления RMQ для массива `lcp` к вычислению `lca` для декартовых деревьев (их определение приведу позже). Во-вторых, выполнить переход от вычисления `lca` для декартовых деревьев к вычислению RMQ для корректно определенного двоичного массива. Для решения последней задачи требуются пространство $O(n)$ и постоянное время. Очевидно, что второй переход возможен для любого дерева, в том числе для суффиксных деревьев при поиске `lca`. Это позволяет напрямую решить задачу нечеткого поиска с k несовпадениями.

Первый шаг редукции: от RMQ к lca. Задача $\texttt{RMQ}_A$ преобразуется в вычисление `lca` для специального дерева, построенного на основе записей массива $A[1, n]$. Это так называемое *декартово дерево* C_A — двоичное дерево из n узлов, каждый из которых помечен одним из элементов A, то есть значением и позицией в A. Маркировка определяется рекурсивно следующим образом: корень C_A помечен минимальным элементом из $A[1, n]$ и его позицией, например $\langle A[m], m \rangle$. Затем левое поддерево корня рекурсивно определяется как декартово дерево подмассива $A[1, m - 1]$, а правое поддерево — как декартово дерево подмассива $A[m + 1, n]$. Простой пример для массива из пяти позиций приведен на рис. 10.12.

Рисунок 10.13 демонстрирует декартово дерево на основе массива `lcp`, приведенного на рис. 10.11. Учитывая детали построения, можно утверждать, что диапазоны массива `lcp` соответствуют поддеревьям в декартовом дереве, поэтому вычисление $\texttt{RMQ}_A(i, j)$ сводится к запросу `lca` между узлами C_A, связанными с записями i и j. В отличие от запросов `lca` в ST_X, где аргументами выступают листья суффиксного дерева, в декартовом дереве запрос может выполняться к внутренним узлам, возможно, иерархически связанным. В этом случае запрос `lca` становится тривиальным. Например, выполнение $\texttt{RMQ}_{\text{lcp}}(6, 8)$ равно выполнению $\texttt{lca}(6, 8)$ для декартова

дерева C_{lcp}. Узлы, к которым делается запрос, выделены, и в результате получаем узел ⟨lsp[7], 7⟩ = ⟨1, 7⟩. Обращу ваше внимание на то, что у нас есть еще одно минимальное значение в lcp[6, 8] для lcp[6] = 1, но алгоритм его не обнаруживает, потому что выводится положение только одного минимума.

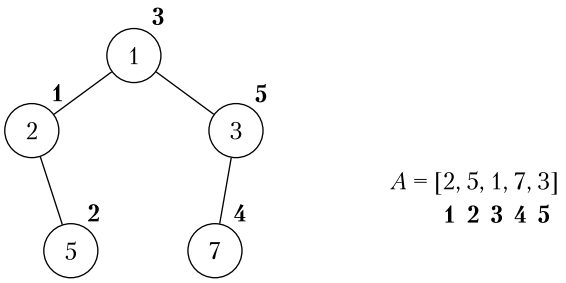
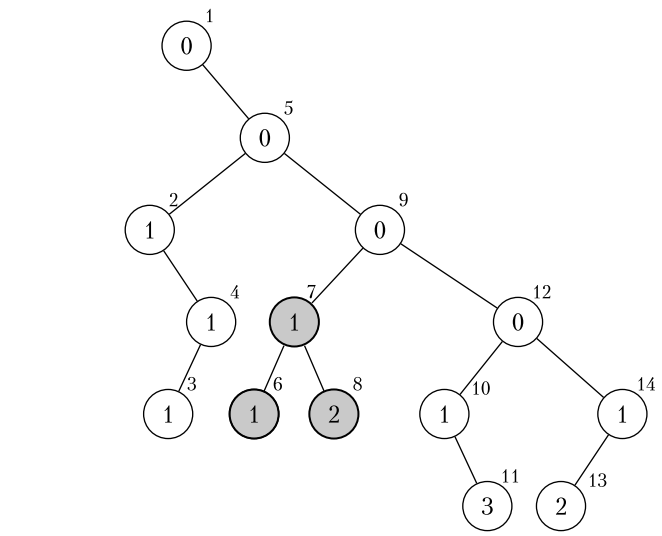


Рис. 10.12. Декартово дерево для массива $A[1, 5] = \{2, 5, 1, 7, 3\}$. Обратите внимание на то, что узлы C_A хранят в качестве первого компонента метки узла значение A (внутри узла), а в качестве второго компонента — его положение в A (снаружи узла, выделено жирным шрифтом)



Эйлеров обход = 1 5 2 4 3 4 2 5 9 7 6 7 8 7 9 12 10 11 10 12 14 13 14 12 9 5 1
D = 1 2 3 4 5 4 3 2 3 4 5 4 5 4 3 4 5 6 5 4 5 6 5 4 3 2 1

Рис. 10.13. Декартово дерево для массива lcp (см. рис. 10.11). Внутри узлов — значения LCP, снаружи — соответствующая позиция в массиве lcp (в префиксных деревьях делается произвольный выбор). В качестве примера потомок корня помечен как ⟨lsp[5], 5⟩ = ⟨0, 5⟩ . В нижней части рисунка приведены позиции узлов, посещенных во время эйлерова обхода, и массив D с глубиной этих узлов

Второй шаг редукции: от lca к RMQ. От поиска `lca` в декартовом дереве C_A переходим обратно к вычислению RMQ в специальном двоичном массиве $\Delta[1, 2e + 1]$, где e — количество ребер в декартовом дереве (разумеется, $e = O(n)$). Кажется странным, что последовательные редукции вернули нас к RMQ-задаче. Но дело в том, что теперь мы будем работать с двоичным массивом, который допускает оптимальное решение с использованием пространства $O(n)$.

Для создания двоичного массива $\Delta[1, 2e + 1]$ сначала строится массив $D[1, 2e + 1]$, определенный следующим образом. *Эйлеров обход* декартова дерева C_A — это последовательность узлов, полученных предварительным посещением C_A, при котором записывается каждый посещенный узел как при нисходящем, так и при восходящем обходе ребер. Таким образом, ребро посещается дважды, что оправдывает аддитивный член $2e$ в размере массива D, тогда как узел посещается и, таким образом, записывается столько раз, сколько ребер с ним связано. Исключение тут только корень, который записывается на один раз больше, чем количество связанных с ним ребер (отсюда +1 в размере массива D). Мы строим массив $D[1, 2e + 1]$, сохраняя в записи $D[i]$ глубину i-го узла при эйлеровом обходе декартова дерева C_A. В качестве примера см. рис. 10.13, где массив A — это массив `lcp`, приведенный на рис. 10.11.

Учитывая, каким образом массив D создается в процессе эйлерова обхода, можно сделать вывод, что запрос $\texttt{lca}(i, j)$ в декартовом дереве C_A сводится к нахождению в подмассиве $D[i', j']$ узла с минимальной глубиной, где i' (j') — это позиция первого (последнего) вхождения узла i (j) в процедуре обхода. Фактически диапазон (i', j') соответствует части обхода, которая начинается в узле i и заканчивается в узле j. Обнаруженный в этой подпоследовательности узел минимальной глубины и будет искомым наименьшим общим предком $\texttt{lca}(i, j)$.

Итак, поиск `lca` в декартовом дереве C_A сведен к RMQ-запросу по массиву D, содержащему данные о глубине узлов. В примере, приведенном на рис. 10.13, такая редукция преобразует запрос $\texttt{lca}(6, 8)$ в запрос $\texttt{RMQ}_D(11, 13)$, который выделен прямоугольником. Переход от узлов к диапазонам можно выполнить за постоянное время, сохраняя на узел декартова дерева по два целых числа, соответствующих их первому/последнему появлению в эйлеровом обходе. Это требует дополнительного пространства $O(n)$.

Вернемся к RMQ-запросу над целочисленным массивом. Особенность текущего массива D состоит в том, что его последовательные записи различаются на 1, ведь они ссылаются на глубины последовательных узлов в эйлеровом обходе. В этой ситуации два последовательных узла соединены ребром, соответственно, один узел является предком другого, а их глубины различаются на единицу. Именно это позволяет найти RMQ для $D[1, 2e + 1]$, заняв пространство $O(n)$ за время $O(1)$. Для этого требуются две структуры данных.

Сначала массив D разбивается на подмассивы D_k, имеющие размер $d = (1/2) \log e$. В каждом подмассиве D_k ищется минимальный элемент, чья позиция сохраняется в запись $M[k]$ нового массива, размер которого составит $(2e + 1)/d = O(e/\log e)$. Наконец, на базе массива M строится описанная ранее разреженная таблица,

занимающая пространство $O\left(\left(\frac{e}{\log e}\right)\log\frac{e}{\log e}\right) = O(e) = O(n)$ и за постоянное время выполняющая RMQ-запросы, выровненные по экстремумам этих подмассивов.

Вторая структура данных создана для того, чтобы давать эффективные ответы на RMQ-запросы, в которых i и j находятся в одном блоке D_k. Очевидно, что создать таблицу с ответами на все возможные пары индексов невозможно, потому что для этого потребуется пространство $O(nd) = O(n \log n)$. Поэтому решение, которое я описываю, вытекает из двух простых вещей, доказательство которых вполне очевидно, поэтому я оставлю его вам.

- **Двоичные записи.** Каждый блок D_k можно преобразовать в пару из первого элемента $D_k[1]$ и двоичного массива $\Delta_k[i] = D_k[i] - D_k[i-1]$, где $i = 2...d$. Записи Δ_k равны либо -1, либо $+1$ из-за разницы в единицу между соседними записями массива D.
- **Местоположение минимума.** Расположение минимального значения D_k зависит только от содержимого двоичной последовательности Δ_k и не зависит от начального значения $D_k[1]$.

Вообще, каждый блок D_k допускает бесконечное количество конфигураций, так как число способов, которыми можно создать экземпляры входного массива A для RMQ-запросов, неограниченно. А вот количество возможных конфигураций массива Δ_k конечно и составляет 2^{d-1}. Это указывает на необходимость применения *алгоритма четырех русских* ко всем двоичным массивам Δ_k. Для этого нужно составить таблицу всех их возможных двоичных конфигураций и для каждой сохранить позицию минимального значения. Длина блоков Δ_k равна $d - 1 < d = \log e/2$, поэтому общее число их возможных двоичных конфигураций не превышает $2^{d-1} = O\left(2^{\log e/2}\right) = O\left(\sqrt{e}\right) = O\left(\sqrt{n}\right)$ Более того, так как у индексов запросов i и j, находящихся внутри блока D_k, может быть не более чем $d = \log e/2$ возможных значений, количество запросов этого типа не будет превышать $O(\log^2 e) = O(\log^2 n)$. Следовательно, строится таблица поиска $T[i_o, j_o, \Delta_k]$, которая индексируется возможными смещениями запроса i_o и j_o в блоке D_k и его двоичной конфигурацией Δ_k. Таблица T хранит в этой записи позицию минимального значения в D_k. Предполагается также, что для каждого k сохранено Δ_k, что позволяет извлекать двоичное представление Δ_k из D_k за постоянное время. Каждый из параметров индексации занимает пространство $O(\log e) = O(\log n)$ бит, то есть одно слово памяти, так как управление им требует времени и пространства $O(1)$. Итого целиком таблица T состоит из $O\left(\sqrt{n}\left(\log n\right)^2\right) = o\left(n\right)$ записей. Для ее построения требуется время $O(n)$. Мощность преобразования D_k в Δ_k уже очевидна: каждая запись $T[i_o, j_o, \Delta_k]$ кодирует ответ для бесконечного числа блоков D_k, которые можно преобразовать в одну и ту же двоичную конфигурацию Δ_k.

Теперь можно приступать к разработке алгоритма, который, используя две упоминавшиеся ранее структуры данных, за постоянное время отвечает на запрос $\texttt{RMQ}_D(i,j)$. Если i, j находятся внутри одного блока D_k, ответ извлекается в два этапа. Сначала вычисляются смещения i_o и j_o относительно начала D_k и из k определяется двоичная конфигурация Δ_k. Эта тройка используется для доступа к правильной записи T.

Если же диапазон (i, j) охватывает по крайней мере два блока, его можно разбить на три части: суффикс блока $D_{i'}$, последовательность блоков $D_{i'+1}...D_{j'-1}$ (возможно, пустую) и, наконец, префикс блока $D_{j'}$. Минимум суффикса $D_{i'}$ и префикса $D_{j'}$ извлекается из T, так как эти диапазоны находятся внутри двух блоков. Минимум диапазона, охватываемого блоками $D_{i'+1}...D_{j'-1}$ (если они непустые), хранится в M. Всю эту информацию можно получить за постоянное время, соответственно, за постоянное время определяется и конечное минимальное положение путем сравнения этих трех минимальных значений.

Теорема 10.8. *Запрос минимума на отрезке по массиву $A[1, n]$ элементов, взятых из упорядоченного пространства, выполняется за постоянное время с помощью структуры данных, которая занимает пространство $O(n)$.*

С учетом редукций, проиллюстрированных ранее, можно сделать вывод, что теорема 10.8 применима и к вычислению `lca` в обобщенных деревьях: достаточно вместо декартова дерева взять дерево входных данных.

Теорема 10.9. *Запросы наименьшего общего предка по произвольному дереву размером n выполняются за постоянное время с помощью структуры данных, которая занимает пространство $O(n)$.*

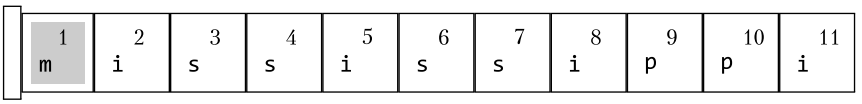
10.4.3. Сжатие текста

Подробно сжатие данных будет рассматриваться в главах 12–15, а в этом разделе приведу пример сжатия текста с помощью простого алгоритма, составляющего основу известной утилиты `gzip`. Он называется *LZ77* по инициалам его изобретателей (Авраам Лемпель и Якоб Зив [13]) и по году публикации (1977). Я продемонстрирую оптимальную реализацию этого алгоритма, которая использует суффиксные деревья и занимает время и пространство $O(n)$. (Подробный разбор семейства алгоритмов LZ приведен в главе 13.)

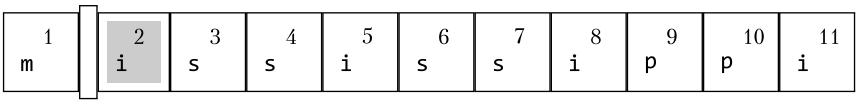
Текстовую строку $T[1, n]$ алгоритм LZ77 *разбирает* на подстроки, которые определяются следующим образом. Предположим, что он уже проанализировал префикс $T[1, i-1]$ (изначально пустой). После этого он разбивает оставшийся текстовый суффикс $T[i, n]$ на три части: самую длинную подстроку $T[i, i+\ell-1]$, которая начинается с i и повторяется до этого в тексте T, следующий символ $T[i+\ell]$ и оставшийся суффикс $T[i+\ell+1, n]$. Следующей к анализу нужно добавить подстроку $T[i, i+\ell]$, которая соответствует самой короткой строке. Это *новая* подстрока среди начинающихся с $T[1, i-1]$. Напоследок анализируется оставшийся суффикс $T[i+\ell+1, n]$, если таковой имеется.

Сжатие достигается путем краткого кодирования тройки целых чисел $\langle d, \ell, T[i+\ell]\rangle$, где d — расстояние (в символах) от i до предыдущей копии $T[i, i+l-1]$, ℓ — длина скопированной строки, а $T[i+\ell]$ — добавленный символ. Под определением «предыдущая копия» $T[i, i+\ell-1]$ подразумевается то, что она начинается до позиции i, но может продолжаться и после нее, то есть это может быть $d < \ell$. Кроме

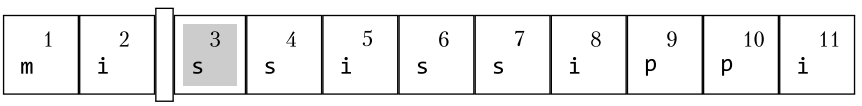
того, предыдущая копия может быть любым предыдущим вхождением $T[i, i + \ell - 1]$, хотя необходимость эффективно использовать пространство предполагает, что мы берем ближайшую копию (и, следовательно, наименьшее d)[1]. Обратите внимание на то, что символ $T[i + \ell]$ добавляется к тройке, потому что в случаях, когда копирование невозможно, ведет себя как *механизм выхода* и, соответственно, $d = 0$ и $\ell = 0$. В частности, это происходит, когда в процессе анализа алгоритм LZ обнаруживает в T новый символ, как показывает пример для строки $T =$ `mississippi`.



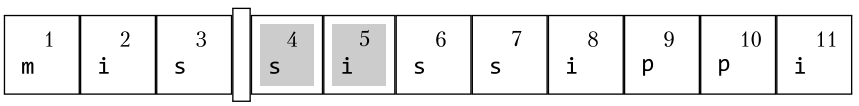
Вывод: ⟨0, 0, m⟩



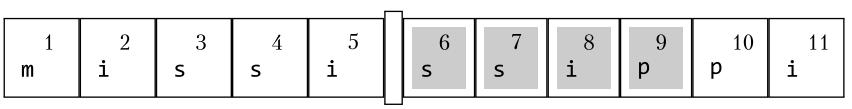
Вывод: ⟨0, 0, *i*⟩



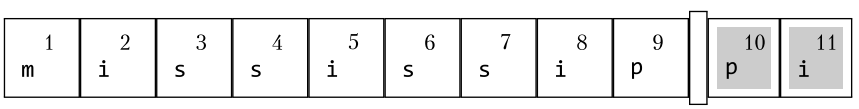
Вывод: ⟨0, 0, *s*⟩



Вывод: ⟨1, 1, *i*⟩



Вывод: ⟨3, 3, *p*⟩



Вывод: ⟨1, 1, *i*⟩

[1] Проблему целочисленного кодирования здесь я обсуждать не буду, потому что это тема главы 11. Упомяну только, что эффективность использования пространства классической утилитой gzip достигается путем взятия самой правой копии и кодирования значений d и ℓ с помощью алгоритма Хаффмана. Однако недавние исследования [5], [8] показали, что лучшая предыдущая копия не обязательно ближайшая. Именно это реализует Brotli — алгоритм сжатия данных от Google с открытым исходным кодом [1].

Синтаксический разбор алгоритм LZ77 выполняет за время $O(n)$ на таком же пространстве с помощью элегантного механизма развертывания суффиксного дерева *ST*. Трудность состоит в том, чтобы найти π_i — самую длинную подстроку, которая встречается в позиции i и повторяется ранее в тексте T, скажем, на расстоянии d от i. Соответственно, имеем $\ell = |\pi_i|$ и можем утверждать, что это самый длинный общий префикс суффиксов $suff_i$ и $suff_{i-d}$. Согласно свойствам суффиксных деревьев суффиксов, наименьший общий предок листьев i и $i - d$ формирует строку π_i. Но вычислить `lca`$(i, i - d)$ путем запроса к структуре данных теоремы 10.9 невозможно, потому что нам неизвестно d. Более того, это именно та информация, которую мы хотим вычислить. Аналогично мы не можем проследить нисходящий путь от корня *ST*, совпадающий с суффиксом $suff_i$, поскольку все суффиксы T индексируются в дереве суффиксов, так что есть вероятность обнаружить более длинную копию, которая следует за позицией i, а не предшествует ей.

Чтобы этого избежать, выполняется постфиксный обход дерева *ST*, в процессе которого для каждого внутреннего узла u определяется минимальный нисходящий лист $\min(u)$. Очевидно, что $\min(u)$ — это самая левая позиция, из которой можно скопировать подстроку $s[u]$. На основе этой информации легко определяется π_i: достаточно проследить нисходящий путь от корня *ST*, сканируя $suff_i$, и остановиться в узле v, для которого $\min(v) = i$. В этот момент мы берем узел u как предка узла v и устанавливаем $\pi_i = s[u]$, а $d = i - \min(u)$. Очевидно, что выбранная копия подстроки π_i расположена дальше всего от позиции i, а не ближе всего к ней. Но на количество итераций алгоритма LZ77, анализирующего строку T, это не влияет, зато, возможно, влияет на расстояния между копиями и, соответственно, на их краткое кодирование. Разработка алгоритма LZ77, эффективно определяющего ближайшую копию каждой подстроки π_i, нетривиальна и требует гораздо более сложных структур данных, рассмотрение которых выходит за рамки тематики данной книги. Но этот процесс освещался в литературе, например в [8], [2].

Предположим, что в процессе синтаксического анализа суффиксного дерева *ST* (см. рис. 10.7) обработан префикс `missi` и выделены тройки $\langle 0, 0,$ `m`$\rangle$, $\langle 0, 0,$ `i`$\rangle$, $\langle 0, 0,$ `s`$\rangle$, $\langle 1, 1,$ `i`$\rangle$. Трассировка суффикса $suff_6$ = `ssippi$` вниз по дереву *ST* останавливается в узле z, поскольку $\min(z) = 3 < 6$, и любой дополнительный символ из $suff_6$ приводит к листу 6, который не может быть меньше самого себя. Следовательно, сгенерированная тройка корректно равна $\langle 6 - \min(z), |s[z]|, T[6 + |s[z]|]\rangle = \langle 3, 3,$ `p`$\rangle$.

Временная сложность этого алгоритма линейна по длине текста T, поскольку обход суффиксного дерева продвигается по этой строке и может произойти только n раз. Ветвление узлов с помощью идеальных хеш-таблиц реализуется за время $O(1)$, как было в задаче поиска подстроки. Построение суффиксного дерева с использованием одного из алгоритмов из предыдущих разделов занимает время $O(n)$. На вычисление значений $\min(u)$ по всем узлам u при постфиксном обходе *ST* затрачивается время $O(n)$.

Теорема 10.10. *Для синтаксического анализа строки $T[1, n]$ с помощью алгоритма LZ77 требуется время и пространство $O(n)$. Предложенный алгоритм копирует каждую анализируемую подстроку из ее самого дальнего предыдущего появления.*

10.4.4. Интеллектуальный анализ текста

В этом разделе будут кратко рассмотрены два примера использования массивов суффиксов и массивов `lcp` для решения интересных задач интеллектуального анализа текста.

Давайте проверим, *существует ли подстрока* $T[1, n]$ *длиной* L, *которая повторяется по крайней мере дважды*. Такую задачу можно решить методом перебора, то есть просто брать каждую подстроку длиной L и подсчитывать количество ее вхождений в T. Количество подстрок оценивается как $\Theta(n)$, на поиск каждой из них затрачивается время $O(nL)$, следовательно, общая временная сложность такого алгоритма составит $O(n^2L)$. Более эффективное и быстрое, фактически оптимальное решение возможно при использовании либо суффиксного дерева, либо массива `lcp`, построенного на входном тексте T.

Воспользоваться суффиксным деревом очень просто. Предположим, что описанная строка существует и встречается в позициях x и y. Для двух листов суффиксного дерева, которые соответствуют $suff_x$ и $suff_y$, вычислим наименьшего общего предка $a(x, y)$. Поскольку $T[x, x + L - 1] = T[y, y + L - 1]$, то $|s[a(x, y)]| \geqslant L$. Знак «больше или равно» обусловлен тем, что у более длинной подстроки в позициях x и y может находиться общий символ. Фактически значение L задается только условием задачи. Из этого следует, что в ST_T существует *внутренний* узел u, метка которого больше или равна L, а именно $|s(u)| \geqslant L$. Следовательно, обхода суффиксного дерева достаточно для поиска такого узла u и он занимает время $O(n)$.

Использование массива суффиксов немного сложнее, но базируется на аналогичных рассуждениях. Напомню, что суффиксы в SA лексикографически упорядочены, поэтому самый длинный префикс, общий для суффикса $SA[i]$, — это префикс с его соседями, то есть либо с суффиксом $SA[i - 1]$, либо с суффиксом $SA[i + 1]$. Длина этих LCP хранится в записях `lcp`$[i - 1, i]$. Если повторяющаяся подстрока длиной L существует и встречается, например, в позициях x и y, то $lcp(T[x, n], T[y, n]) \geqslant L$. Эти два суффикса не обязательно смежны в массиве SA (так происходит, например, в случае, когда подстрока встречается более двух раз), тем не менее все суффиксы, встречающиеся среди них в SA, наверняка будут иметь общий префикс длиной L из-за их лексикографического порядка. Следовательно, если суффикс $T[x, n]$ встречается в позиции i массива суффиксов, а именно $SA[i] = x$, то `lcp`$[i - 1] \geqslant L$ (если $T[y, n] < T[x, n]$) или `lcp`$[i] \geqslant L$ (если $T[y, n] > T[x, n]$). Следовательно, для решения задачи из этого раздела нужно просканировать массив `lcp` в поиске записи, значение которой больше или равно L. Это занимает оптимальное время $O(n)$.

Теперь рассмотрим более сложную задачу. Нужно *проверить, существует ли подстрока* $T[1, n]$ *длиной* L, *которая повторяется не менее* C *раз*. Это типичный запрос в сценарии интеллектуального анализа текста, где нас интересует не просто повторяющееся событие, а событие, подтвержденное *статистическими данными*. Такую задачу тоже можно решить, подсчитав вхождения всех возможных подстрок, но более быстрым будет решение с использованием суффиксного дерева или массива `lcp`. Следуя той же логике, что и в предыдущем случае, легко заметить, что если

подстрока длиной L встречается не менее C раз, то существует не менее C суффиксов, у которых будут общими не менее L символов. Таким образом, в суффиксном дереве есть узел u, такой, что $|s[u]| \geqslant L$ и количество нисходящих листьев $|occ[u]| \geqslant C$. Аналогично существует подмассив `lcp` $\geqslant C - 1$, элементы которого будут больше или равны L. Оба подхода дают ответ на задачу за время $O(n)$.

И в заключение рассмотрим задачу, подходящую для сценария поисковой системы: *даны два шаблона, $P[1, p]$ и $Q[1, q]$, и положительное целое число k; нужно проверить, существует ли вхождение шаблона P, расстояние от которого до вхождения шаблона Q во входном тексте $T[1, n]$ не превышает k*. Это так называемый *поиск с учетом близости слов* по тексту T, который проводится в рамках предварительной обработки. Для решения можно воспользоваться любой структурой данных для поиска строк, будь то суффиксное дерево или массив суффиксов, и некоторыми этапами сортировки/сканирования. В массиве суффиксов или в суффиксном дереве, построенном для текста T, ищут вхождения P и Q и извлекают их в несортированном виде. Затем эти вхождения *occ* сортируются, упорядоченный список сканируется и для каждой пары *последовательных* позиций вхождений P и Q проверяется, не превышает ли их разница k. В общей сложности это требует времени $O(p + q + occ \log occ)$, что явно выгодно при небольшом наборе кандидатов. Таким образом, запросы P и Q довольно селективны.

Список литературы

1. *Alakuijala J., Farruggia A., Ferragina P. et al.* Brotli: A general-purpose data compressor // ACM Transactions on Information Systems, 37 (1): article 4, 2019.
2. *Belazzougui Djamal, Puglisi S. J.* Range predecessor and Lempel — Ziv parsing // Proceedings of the 27th Annual ACM — SIAM Symposium on Discrete Algorithms (SODA), 2053–2071, 2016.
3. *Bender M. A., Farach-Colton M.* The LCA problem revisited // Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN), 88–94, 2000.
4. *Farach-Colton M., Ferragina P., Muthukrishnan S.* On the sorting-complexity of suffix tree construction // Journal of the ACM, 47 (6): 987–1011, 2000.
5. *Farruggia A., Ferragina P., Frangioni A., Venturini R.* Bicriteria data compression // SIAM Journal on Computing, 48 (5): 1603–1642, 2019.
6. *Ferragina P.* String search in external memory: Algorithms and data structures // *Aluru S.* Handbook of Computational Molecular Biology. Chapman & Hall/CRC Computer and Information Science Series, 35-1-35-48, 2005.
7. *Ferragina P., Gagie T., Manzini G.* Lightweight data indexing and compression in external memory // Algorithmica: Special Issue on Selected Papers of LATIN 2010, 63 (3): 707–730, 2012.
8. *Ferragina P., Nitto I., Venturini R.* On the bit-complexity of Lempel — Ziv compression // SIAM Journal on Computing, 42 (4): 1521–1541, 2013.

9. *Gusfield D.* Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.
10. *Gonnet G. H., Baeza-Yates R. A., Snider T.* New indices for text: PAT trees and PAT arrays // *B. Frakes, R. A. Baeza-Yates.* Information Retrieval: Data Structures and Algorithms, Prentice Hall, 66–82, 1992.
11. *Kärkkäinen J., Sanders P.* Simple linear work suffix array construction // Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science 2791, Springer, 943–955, 2003.
12. *Kasai T., Lee G., Arimura H., Arikawa S., Park K.* Linear-time longest-common-prefix computation in suffix arrays and its applications // Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM), Lecture Notes in Computer Science 2089, Springer, 181–192, 2001.
13. *Ziv J., Lempel A.* A universal algorithm for sequential data compression // IEEE Transactions on Information Theory, 23 (3): 337–243, 1977.
14. *Manber U., Myers G.* Suffix arrays: A new method for on-line string searches // SIAM Journal on Computing, 22 (5): 935–948, 1993.
15. *McCreight E. M.* A space-economical suffix tree construction algorithm // Journal of the ACM, 23 (2): 262–272, 1976.

Глава 11
ЦЕЛОЧИСЛЕННОЕ КОДИРОВАНИЕ

> С целыми числами проблема в том, что рассмотрены только очень маленькие числа. Может быть, все самое интересное происходит с действительно большими числами, которых мы даже не можем себе представить.
>
> *Рональд Грэм*

Эта глава посвящена рассмотрению базовой задачи кодирования, которая возникает во многих контекстах [4], [9] и влияние которой на общий объем памяти и быстродействие базового приложения слишком легко недооценить или проигнорировать.

Задача. Последовательность $S = s_1 ... s_n$ положительных целых чисел s_i, возможно повторяющихся, нужно представить в виде занимающих небольшое пространство двоичных последовательностей, в которых различимо каждое кодовое слово.

Требование того, чтобы s_i были положительными целыми числами, можно ослабить, найдя в S минимальное значение среди отрицательных целых чисел и прибавив его модуль ко всем s_i.

Первым делом я хотел бы показать вам два примера применения этой задачи на практике. Поисковые системы хранят для каждого термина t список документов (веб-страниц, сообщений в блогах, твитов и т. д.), в которых он встречается. Называется такая структура данных *списком позиций t*. Документы в ней обычно представлены целочисленными идентификаторами, которые назначаются во время веб-сканирования. Ответ на запрос пользователя, сформулированный как последовательность ключевых слов $t_1 t_2 ... t_k$, сводится к поиску идентификаторов docID, где встречаются все t_i. Это реализуется с помощью пересечения списков позиций для k терминов. Но хранение набора целых чисел в двоичном представлении фиксированной длины, то есть 4 или 8 байт, требует значительного пространства, а следовательно, и времени для их извлечения, учитывая, что современные поисковые системы индексируют миллиарды документов. Чтобы уменьшить занимаемое дисковое пространство, а пропускную способность и объем кэшированных списков во внутренней памяти, наоборот, увеличить, обычно применяются два вида

сжатия. Во-первых, сортируются идентификаторы документов в каждом списке позиций, а затем каждый docID представляется разницей между ним и его предшественником, то есть выполняется так называемое *кодирование промежутков*[1]. Во-вторых, каждый промежуток можно закодировать в виде последовательности битов переменной длины, которая для небольших целых чисел получится очень короткой ([3], [7], [9]).

Второе применение нашей задачи относится к сжатию данных. В главе 10 вы видели, как алгоритм LZ77 превращает входные файлы в последовательности троек, в которых первые два компонента — целые числа. Другие известные алгоритмы сжатия, такие как MTF, MPEG, RLE, BWT и т. п., в качестве промежуточного вывода генерируют один или несколько наборов целых чисел, причем чем меньше значение, тем выше вероятность его появления. Затем эти целые числа преобразуются в поток битов таким образом, чтобы их общее количество получилось минимальным ([4], [9]).

Последний пример, подтверждающий универсальность задачи сжатия целых чисел, можно получить, рассматривая любой текст T как последовательность токенов, роль которых могут играть как слова, так и отдельные символы. Каждый токен можно представить целым числом, известным также как *tokenID*, так что задача сводится к сжатию последовательности его *tokenID*. Для оптимизации схемы целочисленного кодирования токены можно отсортировать по убыванию частоты их появления в T, а затем назначить им *ранг*. В итоге чем чаще токен встречается в T, тем меньше будет его *tokenID* и тем короче кодовое слово, назначенное ему схемой целочисленного кодирования. Хорошо известно, что слова в лингвистических текстах следуют закону Ципфа [3]: i-е по частотности появления слово в T имеет частотность $c(1/i)^{\alpha}$, где c — коэффициент нормализации, α — параметр, зависящий от входного текста. Затем для кодирования идентификаторов токенов T можно прибегнуть к любому из *универсальных* алгоритмов целочисленного кодирования (например, приведенным в разделе 11.1), достигая производительности сжатия, близкой к *энтропии* входного текста.

Поэтому основной вопрос, который будет рассматриваться в этой главе, заключается в том, как разработать для последовательности неограниченных целых чисел двоичное представление переменной длины, занимающее как можно меньше битов и не содержащее префиксов. То есть нужно, чтобы двоичное кодирование всех целых чисел допускало конкатенацию для создания выходного потока битов, сохраняющего возможность декодирования в том смысле, чтобы декодер умел определять начало и конец каждого отдельного представления целого числа в потоке битов и, таким образом, возвращал его в исходное, несжатое состояние.

Первая и самая простая идея — взять $m = \max_j s_j$ и закодировать каждое целое число $s_i \in S$, используя $\lceil \log_2(m+1) \rceil$ бит. Такое кодирование фиксированного размера эффективно, когда множество S не очень сильно рассредоточено и сконцентрировано вокруг значения 0. Но это нестандартная ситуация, потому что в общем случае

[1] Разумеется, первый в списке позиций docID хранится полностью.

$m >> s_i$, так что в этом случае много битов из выходного потока будет расходоваться впустую. Так почему бы не сохранить каждое s_i в двоичном виде с использованием $\lceil \log_2(s_i + 1) \rceil$ бит? Проблема в том, что невозможно объединить двоичное представление всех s_i и потом различать каждое кодовое слово. Например, рассмотрим $S = \{1, 2, 3\}$ и выходную битовую последовательность `11011`, полученную из двоичных представлений этих чисел: `1`, `10` и `11`. Очевидно, `11011` можно интерпретировать, как, например, {6, 1, 1} или {1, 2, 1, 1}. Есть и другие варианты.

Ясно, что простая с виду задача кодирования на самом деле нетривиальна и заслуживает того внимания, которое я уделил ей в этой главе. Для начала рассмотрим самый простой и известный из целочисленных кодов — *унарный код* $U(x)$. Для целого числа $x \geqslant 1$ он задается последовательностью $x - 1$ бит, установленных в 0, заканчивающейся (разделительным) битом, установленным в 1. Корректность условия $x \neq 0$ легко проверяется, хотя этот код можно заставить работать и со всеми неотрицательными целыми числами, просто установив 0 для x бит, а не для $x - 1$. Далее я буду игнорировать все эти тривиальные технические моменты, сосредоточившись на строго положительных числах s_i. Ясно, что $U(x)$ требует x бит, что *экспоненциально больше* длины $\Theta(\log x)$ его двоичного кода. Так что этот код эффективен для очень маленьких целых чисел, но по мере роста x быстро становится неэффективным по пространству.

Это утверждение можно сделать более точным, вспомнив основной факт, вытекающий из теоремы Шеннона, которая гласит, что *идеальная длина кода* $L(c)$ *для символа* c *составляет* $\log_2 1/\mathcal{P}[c]$ *бит, где* $\mathcal{P}[c]$ *— вероятность появления символа* c. Эта вероятность может быть известна заранее при наличии достаточной информации об источнике c, или же ее можно оценить эмпирически, подсчитав появления символа c в последовательности S. Надеюсь, вы помните, что в этой главе рассматривается сценарий, в котором символы представляют собой положительные целые числа, поэтому идеальный код для целого числа x состоит из $\log_2 1/\mathcal{P}[x]$ битов. Решение уравнения $|U(x)| = \log_2 1/\mathcal{P}[x]$ относительно $\mathcal{P}[x]$ дает распределение целых чисел x, для которых унарный код является оптимальным. В рассматриваемом случае $\mathcal{P}[x] = 2^{-x}$. Что касается эффективности, то унарный код требует большого количества битовых сдвигов, которые современные процессоры выполняют довольно медленно. Это еще одна причина отдавать предпочтение небольшим целым числам.

Теорема 11.1. *Унарный код положительного целого числа* x *занимает* x *бит и, таким образом, оптимален для распределения* $\mathcal{P}[x] = 2^{-x}$.

С помощью таких же рассуждений можно прийти к выводу, что двоичное представление фиксированной длины, использующее $\lceil \log_2(m + 1) \rceil$ бит, оптимально, когда целые числа последовательности S *равномерно распределены* по диапазону $\{1, 2, ..., m\}$.

Теорема 11.2. *Каждое число в наборе* S *целых чисел максимальной длиной* m *в двоичном представлении фиксированной длины занимает* $\lceil \log_2(m + 1) \rceil$ *бит, что оптимально для равномерного распределения* $\mathcal{P}[x] = 1/m$.

В общем случае целые числа распределены неравномерно, и фактически следует рассматривать двоичные представления переменной длины, улучшающие простой унарный код. В литературе описано много вариантов, каждый из которых предлагает свой *компромисс между пространством, занимаемым двоичным кодом, и эффективностью по времени его декодирования*. В следующих разделах будут подробно описаны самые полезные и наиболее применимые двоичные коды переменной длины, начиная с самых простых, на базе *фиксированных моделей кодирования* целых чисел в S (например, γ- и δ-коды), и заканчивая более сложными интерполяционными кодами и кодами Элиаса — Фано на базе динамических моделей, умеющих адаптироваться к распределению целых чисел в S и таким образом обеспечивающих более компактное кодирование.

Довольно неожиданно, но, как я докажу, в некоторых случаях интерполяционный код может оказаться даже короче оптимального кода Хаффмана, описанного в главе 12. Это противоречивое с виду утверждение исходит из того факта, что код Хаффмана *оптимален* для семейства *статических* кодов без префиксов, а именно тех, которые используют фиксированный код для всех вхождений x в S. Интерполяционный код, наоборот, задействует динамическую модель, которая кодирует x в соответствии с распределением окружающих его целых чисел, что может давать различные представления для разных вхождений x в S. Для некоторых вариантов распределения целых чисел такое контекстно зависимое поведение интерполяционного кода может дать гораздо более компактное представление выходного потока битов.

11.1. Гамма- и дельта-коды Элиаса

В 1960-х годах Элиас представил два очень простых *универсальных* кода для целых чисел на базе фиксированной модели [2]. Определение «универсальный» здесь указывает на то, что для любого целого числа x длина кода равна $O(\log x)$. Это на постоянный множитель длиннее двоичного кода $B(x)$, который имеет длину $\lceil \log(x + 1) \rceil$, желательно вдобавок при отсутствии префиксов.

В γ-коде целое число x представлено как двоичная последовательность, состоящая из двух частей: набора из $|B(x)| - 1$ нулей и следующего за ним двоичного представления $B(x)$. Набор нулей ограничен первой 1, которая также соответствует первому биту двоичного представления $B(x)$. Это обеспечивает простоту декодирования $\gamma(x)$ — достаточно подсчитать количество нулей до первой 1 (предположим, оно равно c). После этого остается извлечь следующие $c + 1$ бит, включая конечную 1, и интерпретировать двоичную последовательность длиной c как целое число x. Графическое представление γ-кода для произвольного положительного целого числа x и его реализацию для $x = 9$ демонстрирует рис. 11.1.

γ-код требует $2|B(x)| - 1$ бит, что равно $2\lceil \log_2(x + 1) \rceil - 1$. То есть γ-код целого числа 9 требует $2\lceil \log_2(9 + 1) \rceil - 1 = 7$ бит. Из условия Шеннона для идеальных кодов следует, что γ-код оптимален всякий раз, когда распределение целых чисел в S описывается формулой $\mathcal{P}[x] \approx 1/x^2$.

Рис. 11.1. Графическое представление: *слева* — $\gamma(x)$ при $x > 0$ и $\ell = |B(x)|$, темно-серый прямоугольник обозначает двоичную цифру 1, общую для $U(\ell)$ и $B(x)$; *справа* — $\gamma(9)$. Обратите внимание на темно-серую цифру 1, общую для унарного кода $U(4)$ и двоичного кода $B(9)$

Теорема 11.3. *Гамма-код положительного целого числа x занимает $2\lceil \log_2(x+1) \rceil - 1$ бит, таким образом, это оптимальный вариант для распределения $\mathcal{P}[x] \approx 1/x^2$. От длины в битах $|B(x)| = \lceil \log_2(x+1) \rceil$ двоичного кода x он отличается не более чем в два раза.*

Неэффективность γ-кода заключается в унарном кодировании длиной $|B(x)|$, которое по мере увеличения x становится все более затратным. Чтобы сгладить ситуацию, Элиас ввел δ-код, который вместо унарного кода использует γ-код. Таким образом, $\delta(x)$ состоит из двух частей: первая — гамма-код $\gamma(|B(x)|)$, вторая — x в двоичном представлении. Так как для длины $B(x)$ используется γ-код, первая и вторая части не имеют общих битов. Кроме того, γ применяется к $|B(x)|$, которое гарантированно больше нуля. Декодировать $\delta(x)$ просто: сначала декодируется $\gamma(|B(x)|)$, после чего извлекаются следующие биты $|B(x)|$, соответствующие значению x в двоичном виде. Интересно отметить, что δ-код может кодировать ноль: $\delta(0) =$ `1 0`, где первый бит соответствует $\gamma(1) = 1$ — длине (специального) двоичного представления значения 0. Графическое представление δ-кода для произвольного положительного целого числа x и его реализацию для $x = 14$ демонстрирует рис. 11.2.

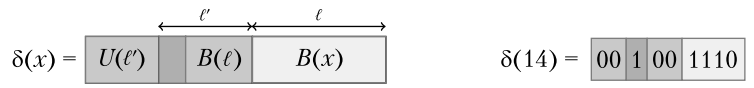


Рис. 11.2. Графическое представление: *слева* — $\delta(x)$ при $x > 0$, $\ell = |B(x)|$ и $\ell' = |B(\ell)|$, темно-серый прямоугольник обозначает двоичную цифру 1, общую для $U(\ell')$ и $B(\ell)$; *справа* — $\delta(14)$, где $\ell = 4$ и $\ell' = 3$, темно-серая цифра 1, общая для унарного кода $U(\ell') = U(3)$ и $B(v) = B(4)$, учитывая, что $|B(x)| = |B(14)| = 4$

Количество битов, занимаемых $\delta(x)$, равно $|\gamma(l)| + l = 2\lceil \log_2(\ell + 1) \rceil - 1 + \ell \approx$ $\approx 2 \log \log x + \log x + 1$. Таким образом, этот вариант кодирования отличается от длины $l = |B(x)|$ двоичного кода x на множитель $1 + o(1)$, следовательно, он универсален.

Теорема 11.4. *δ-код положительного целого числа x занимает примерно $1 + \log_2 x +$ $+ 2\log_2 \log_2 x$ бит, таким образом, он оптимален для распределения $\mathcal{P}[x] \approx \dfrac{1}{x(\log x)^2}$. От длины в битах $|B(x)| = 2\lceil \log_2(x+1) \rceil$ двоичного кода x он отличается множителем $1 + o(1)$.*

В заключение скажу, что γ- и δ-коды универсальны и довольно эффективны, когда целые числа в наборе S сосредоточены вокруг 1. Известно, что их декодирование требует большого количества битовых сдвигов, таким образом, оно крайне медленно работает с большими целыми числами. Теперь посмотрим на варианты кодов, которые жертвуют эффективностью использования пространства ради скорости декодирования и, по сути, более предпочтительны в практических приложениях.

11.2. Код Райса

Бывает так, что целые числа концентрируются вокруг некоторого значения, отличного от нуля. Чем оно больше, тем ниже производительность γ- и δ-кодов. В такой ситуации выгодным как по степени сжатия, так и по скорости декодирования становится код Райса. Это *параметрический код*. Он зависит от положительного целого числа k, которое может быть зафиксировано в соответствии с распределением целых чисел в множестве S. Код Райса $R_k(x)$ целого числа $x > 0$ состоит из двух частей: частного $q = \left\lfloor \frac{x-1}{2^k} \right\rfloor$ и остатка $r = x - 1 - 2^k q$.

Вычитание 1 из частного и остатка преобразует строго положительную целую последовательность в последовательность на базе 0. Для последовательностей, в которых $x \geqslant 0$, необходимости в этой операции нет. Частное хранится в унарной форме, занимая $q + 1$ бит (+1 необходимо, потому что q может быть равно 0, а мы определяем унарный код для положительных целых чисел). Остаток r находится в диапазоне $[0, 2^k)$ и хранится в двоичной форме, занимая k бит, обозначаемых $B_k(r)$. Это означает, что код частного имеет переменную длину, тогда как длина кода остатка фиксирована. Чем ближе 2^k к x, тем короче представление q, а значит, тем быстрее его декодирование. По этой причине k выбирается таким образом, чтобы 2^k концентрировалось вокруг среднего значения элементов S. Графическое представление кода Райса для произвольного положительного целого числа x с параметром k, а также пример его реализации для $x = 83$ и $k = 4$ демонстрирует рис. 11.3.

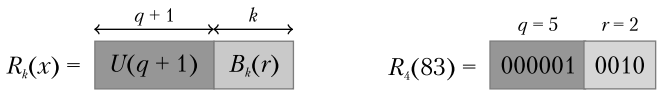


Рис. 11.3. Графическое представление: *слева* — кода Райса с параметром k; *справа* — для $R_4(83)$, где $k = 4$, $q = \lfloor (83 - 1)/2^4 \rfloor = 5$ и $r = 83 - 1 - 5 \times 2^4 = 2$

Битовая длина $R_k(x)$ равна $q + k + 1$. Код Райса представляет собой частный случай кода Голомба [9] — он оптимален, когда кодируемые значения следуют геометрическому распределению с параметром p, а именно $\mathcal{P}[x] = p(1 - p)^{x-1}$. В этом случае, если $2^k \simeq \ln(2)/p \simeq 0{,}69 \times \texttt{mean}(S)$, коды Райса и все коды Голомба генерируют оптимальный префиксный код [9].

Факт 11.1. Код Райса с параметром k для положительного целого числа x занимает $\left\lfloor \frac{x-1}{2^k} \right\rfloor + 1 + k$ бит и оптимален для геометрического распределения $\mathcal{P}[x] = p(1-p)^{x-1}$.

11.3. Алгоритм PForDelta

Существует метод сжатия целых чисел, который поддерживает чрезвычайно быструю распаковку и обеспечивает вывод небольшого размера, если целые числа S следуют нормальному распределению. Предположим, большинство целых чисел в S попадает в интервал [*base*, *base* + $2^b - 2$]. Чтобы закодировать их в b битах, их нужно перевести в интервал [0, $2^b - 2$]. Целые числа за пределами этого диапазона называются *исключениями*. В сжатом списке они фигурируют с *escape-символом* и хранятся в отдельном списке, имея фиксированный размер w бит (целое слово памяти). В двоичном виде *escape-символ* может занимать b бит, представляющих конфигурацию $2^b - 1$, которая не является частью диапазона кодируемых целых чисел (см. пример на рис. 11.4). Хорошее свойство этого кодирования заключается в том, что в результате все целые числа в S имеют фиксированную длину b или $w + b$ бит, поэтому они допускают быстрое декодирование, возможно, в параллельном режиме, если несколько чисел упакованы в одно слово памяти.

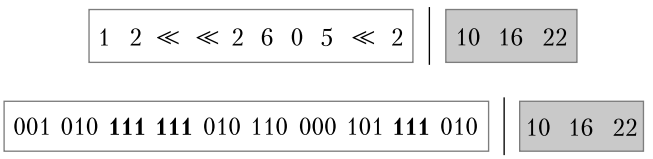


Рис. 11.4. Пример кодирования алгоритмом PForDelta последовательности S = (1, 2, 10, 16, 2, 6, 0, 5, 22, 2) при b = 3 и *base* = 0. *Вверху* показано, что целые числа S из диапазона [0, $2^b - 2$] = [0, 6] явно кодируются в белом прямоугольнике и занимают b бит. А целые числа за пределами диапазона кодируются дважды с помощью escape-символа << в белом прямоугольнике, занимая вместе с полноразмерным представлением в сером прямоугольнике b бит. *Внизу* показан окончательный результат кодирования, в котором escape-символ << представлен в трех битах с использованием зарезервированной двоичной последовательности $2^b - 1 = 7 = (111)_2$

Факт 11.2. Результат кодирования положительного целого числа x алгоритмом PForDelta занимает b или $b + w$ бит в зависимости от того, соблюдается ли условие $x \in$ [*base*, *base* + $2^b - 2$].

При проектировании кода PForDelta для целочисленной последовательности S важен выбор b. Существует эмпирическое правило, согласно которому около 90 % целых чисел S должны попасть в диапазон [*base*, *base* + $2^b - 2$], чтобы результат их кодирования занимал b бит. Альтернативным решением становится компромисс между расходованием пространства впустую (выбор большего b ради уменьшения

количества исключений) и экономией пространства (выбор меньшего b при росте количества исключений). Авторы [8] предложили алгоритм на базе динамического программирования, который вычисляет максимальное для желаемой степени сжатия значение b. Это обеспечивает самую быструю распаковку сжатой последовательности S при заданном ограничении по занимаемому пространству. Скорость декодирования алгоритма PForDelta особенно ценится в сообществе разработчиков программного обеспечения: фактически он принудительно выравнивает слова памяти для групп целых чисел w/b и допускает реализацию, предотвращающую неправильные предсказания ветвлений.

11.4. Код с переменной длиной байтов и (s, c)-плотные коды

Другой класс кодов, в которых приходится искать компромисс между скоростью и сжатием, — это (*s*, *c*)-*плотные коды*. Их простейшая реализация, которая была введена поисковой системой AltaVista, — это *код с переменной длиной байтов* (variable-byte code). Для представления целого числа он использует последовательность байтов с байтовым выравниванием, что обеспечивает значительную скорость декодирования. Для числа x он строится следующим образом: слева двоичное представление $B(x)$ дополняется нулями, чтобы гарантировать длину, кратную 7, затем эта двоичная последовательность разбивается на группы по 7 бит в каждой, и, наконец, к каждой такой группе добавляется бит-флаг, указывающий, завершает она представление x (бит установлен в 0) или нет (бит установлен в 1). Графическое представление кода с переменной длиной байтов для произвольного положительного целого числа и его реализацию для $x = 2^{16}$ демонстрирует рис. 11.5.

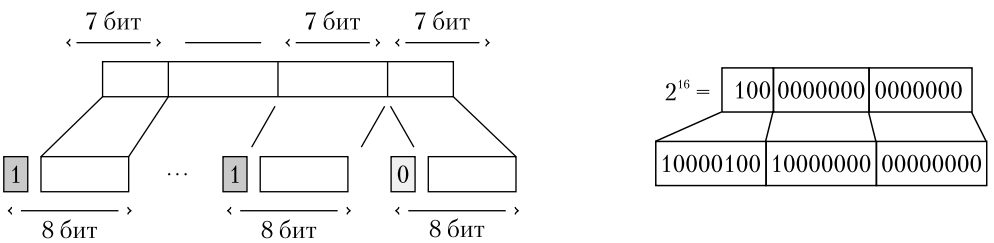


Рис. 11.5. Графическое представление кода с переменной длиной байтов: *слева* — для обобщенного целого числа; *справа* — для числа 2^{16}

Декодирование осуществляется просто: последовательность байтов сканируется до тех пор, пока не будет найден байт со значением меньше 128 (следовательно, флаг в его старшем бите равен 0). После этого все флаги удаляются и полученная двоичная последовательность интерпретируется как положительное целое число. Минимальное количество битов, необходимое для кодирования целого числа x, равно 8, и в среднем из-за выравнивания 4 бита расходуются впустую. Так что этот метод вполне подходит для больших значений x.

Факт 11.3. Код с переменной длиной байтов положительного целого числа x занимает $8\left\lceil \frac{B(x)}{7} \right\rceil$ бит и, таким образом, оптимален для распределения $\mathcal{P}[x] \approx \sqrt[7]{1/x^8}$.

Но использование флагов порождает неочевидную проблему, о которой я расскажу позже, представляя конструкцию более эффективного семейства целочисленных кодеров — (s, c)-плотных кодов.

Бит с флагом разделяет $2^8 = 256$ двоичных конфигураций каждого байта на два набора: конфигурации, представляющие целые числа меньше 128 (флаг равен 0), и конфигурации, представляющие целые числа, больше или равные 128 (флаг равен 1). Первые конфигурации (их 128) называются *стопперами*, потому что они ограничивают конец представления целого числа. Вторых конфигураций также 128, и называются они *продолжателями*, потому что находятся в середине представления целого числа, выровненного по байтам. Для наглядности обозначим мощности этих двух наборов s и c соответственно. В случае кода с переменной длиной байтов $s + c = 2^8 = 256$ и $s = c = 128$. В процессе декодирования при каждой коллизии с продолжателем чтение продолжается, в противном случае полученная двоичная последовательность декодируется в соответствии с проиллюстрированными ранее шагами.

К сожалению, при таком подходе для любого $x < 128$ используется один байт. И когда набор S состоит из очень маленьких целых чисел, биты расходуются впустую. А если S состоит из целых чисел больше 128, но меньше 256, то, возможно, лучше расширить набор стопперов, чтобы по-прежнему использовать для них один байт вместо двух. Имеет смысл изучить конструкцию кодов, в которых набор стопперов/продолжателей меняется при условии $s + c = 256$. Первым делом нужно понять, как выбор s и c влияет на количество целых чисел, которые могут быть закодированы одним или несколькими байтами:

- один байт может кодировать первые s целых чисел;
- два байта могут кодировать последующие sc целых чисел;
- три байта могут кодировать последующие sc^2 целых чисел;
- последовательность из k байт может кодировать sc^{k-1} целых чисел.

Легко вывести замкнутую формулу для количества целых чисел, которые можно закодировать максимум с помощью k байт и, таким образом, с максимум $(k - 1)$ продолжателями и одним стоппером:

$$s\sum_{i=0}^{k-1} c^i = s\frac{c^k - 1}{c - 1}.$$

Отсюда легко вывести количество байтов, необходимых для кодирования целого числа x с помощью (s, c)-кода при $s + c = 256$: достаточно найти наименьшее k такое, что $c^k > x((c - 1)/s) + 1$.

На этом этапе очевидно, что предыдущая конструкция не ограничена последовательностями байтов и, следовательно, конфигурациями более 8 бит. Но ее можно сформировать так, чтобы она соответствовала любому числу b бит. В этом случае общее количество 2^b конфигураций можно произвольно разделить на s стопперов и c продолжателей при условии, что $s + c = 2^b$. Для простоты проектирования предполагается, что первые s конфигураций являются стопперами, а оставшиеся c конфигураций — продолжателями, как было для кода с переменной длиной байтов. Лучший выбор s и c зависит от *распределения* кодируемых целых чисел. Например, чтобы получить (s, c)-код для $b = 3$ бит (вместо 8 бит), нужно выбрать число стопперов и продолжателей таким образом, чтобы $s + c = 2^3 = 8$ (вместо 256). На рис. 11.6 для s и c показаны два варианта, таких, что $s + c = 8$: в первом случае количество стопперов и продолжателей равно 4; во втором — стопперов 6, а продолжателей 2. Изменение s (и, таким образом, $c = 8 - s$) меняет количество целых чисел, которые могут быть закодированы с помощью 3, 6, 9... бит.

Значения	$s = c = 4$	$s = 6,\ c = 2$
0	000	000
1	001	001
2	010	010
3	011	011
4	100 000	100
5	100 001	101
6	100 010	110 000
7	100 011	110 001
8	101 000	110 010
9	101 001	110 011
10	101 010	110 100
11	101 011	110 101
12	110 000	111 000
13	110 001	111 001
14	110 010	111 010
15	110 011	111 011
16	111 000	111 100
17	111 001	111 101
18	111 010	
19	111 011	

	$s = c = 4$	$s = 6,\ c = 2$
s – от	000	000
s – до	011	101
c – от	100	110
c – до	111	111

$0, ..., s - 1$	0, ..., 3	0, ..., 5
$s, ..., c \times s - 1$	4, ..., 19	0, ..., 17
$c \times s, ..., c^2 \times s$	20, ..., 85	18, ..., 41

Рис. 11.6. Пример (s, c)-кода, использующего две разные пары значений для s и c

Обратите внимание на то, что если (4, 4)-код может кодировать только первые четыре целых числа тремя битами, то (6, 2)-код кодирует тремя битами еще два целых числа. Это означает, что второй вариант может дать более сжатую целочисленную последовательность в соответствии с асимметрией распределения целых чисел в {0, ..., 5}. Поэтому может оказаться выгодным адаптировать количество стопперов и продолжателей к распределению вероятностей целых чисел в S. В частности, если это распределение концентрируется вокруг 0, имеет смысл выбирать малое s, в то же время чем более плоское распределение, тем больше смысл рассматривать большие значения s. Авторы [1] предлагают эффективный алгоритм для вычисления оптимального s для заданного распределения целых чисел.

11.5. Интерполяционное кодирование

Существует метод целочисленного кодирования, который можно применять к *возрастающей* последовательности положительных целых чисел. Это означает, что от исходной формулировки задачи этой главы, то есть от целочисленного кодирования последовательности S целых, возможно, повторяющихся чисел мы переходим к работе с последовательностью S', состоящей из возрастающих положительных целых чисел. Для преобразования достаточно установить $S'[i] = \sum_{j=1}^{i} S[j]$, чтобы каждое целое число из S' представляло собой префиксную сумму целых чисел в S.

Давайте сосредоточимся на возрастающей последовательности $S = s'_1, \ldots, s'_n$, где $s'_i < s'_{i+1}$. Интерполяционный код очень эффективен в сжатии пространства, когда S' показывает *сгруппированные* вхождения целых чисел, то есть подпоследовательности, сконцентрированные в небольших диапазонах. Это типичная ситуация, которая возникает при хранении списков позиций поисковых систем [8].

Такая схема целочисленного кодирования строится *рекурсивно*. На каждой итерации алгоритм обрабатывает несжатую подпоследовательность $S'_{l,r}$ и *индуктивно* определяет четыре связанных с ней параметра:

- левый индекс l и правый индекс r разграничивают кодируемую подпоследовательность, то есть $S'_{l,r} = \{s'_l, s'_{l+1}, \ldots, s'_r\}$;
- для самого низкого значения в $S'_{l,r}$ определена нижняя граница *low*, а для самого высокого значения в $S'_{l,r}$ — верхняя граница *hi*, соответственно, $low \leqslant s'_l < \ldots < s'_r \leqslant hi$. Значения *low* и *hi* могут не совпадать с s'_i и s'_r. Это просто нижние и верхние оценки, полученные во время рекурсивных вызовов как кодером, так и декодером.

Первоначально кодируемая подпоследовательность представляет собой полную последовательность $S'[1, n]$. Поэтому $l = 1$, $r = n$, $low = s'_1$ и $hi = s'_n$. Эти четыре значения хранятся в сжатом файле, чтобы декодер смог прочитать их в начале фазы распаковки.

При каждом рекурсивном вызове алгоритм сначала кодирует средний элемент s'_m, где $m = \left\lfloor \frac{l+r}{2} \right\rfloor$, учитывая информацию, имеющуюся в кортеже из четырех

элементов $\langle l, r, low, hi \rangle$, а затем рекурсивно кодирует две подпоследовательности, $s'_l, \ldots, s'_{m-1}$ и $s'_{m+1}, \ldots, s'_r$, используя для каждой из них правильно пересчитанный кортеж.

- Для подпоследовательности $s'_l, \ldots, s'_{m-1}$ параметр *low* такой же, как и на предыдущем шаге, поскольку s'_l не меняется, а параметру *hi* можно присвоить значение $s'_m - 1$, поскольку $s'_{m-1} < s'_m$, ведь целые числа в S' различны и возрастают.
- Для подпоследовательности $s'_{m+1}, \ldots, s'_r$ параметр *hi* такой же, как и раньше, поскольку s'_r не меняется, а параметру *low* можно присвоить значение $s'_m + 1$, поскольку $s'_{m+1} > s'_m$.
- Параметры l, r и n пересчитываются соответствующим образом.

Чтобы кратко закодировать s'_m, алгоритм использует всю информацию, которую может извлечь из кортежа $\langle l, r, low, hi \rangle$. В частности, он знает, что $s'_m \geqslant low + m - l$, потому что слева от s'_m находятся $m - l$ различных элементов S' и наименьший из них больше *low*. Аналогично можно заключить, что $s'_m \leqslant hi - (r - m)$. Таким образом, алгоритм делает вывод, что s'_m лежит в диапазоне $[low + m - l, hi - r + m]$, поэтому кодирование происходит не явно, а в виде разницы между значением s'_m и его известной нижней границей ($low + m - l$). Это занимает $\lceil \log_2 B \rceil$ бит, где $B = hi - low - r + l + 1$ — размер интервала, охватывающего s'_m. Таким образом, для любой плотной последовательности интерполяционный код может использовать очень мало битов на каждый элемент s'_m. В качестве еще одной особенности этой схемы кодирования следует отметить, что всякий раз, когда кодируемая подпоследовательность имеет вид (low, $low + 1 \ldots low + n - 1$), алгоритм не генерирует никаких битов, тем самым достигая существенного преимущества в степени сжатия.

Алгоритм 11.1 раскрывает детали реализации этой схемы. В частности, процедура `BinaryCode`(x, a, b) используется для генерации двоичного кодирования целого числа ($x - a$) в $\lceil \log_2 (b - a + 1) \rceil$ битах при условии, что $x \in \{a, a + 1 \ldots b - 1, b\}$.

Алгоритм 11.1. Интерполяционное кодирование $\langle S', l, r, low, hi \rangle$

1: **if** $r < l$ **then**
2: **return** пустую строку
3: **end if**
4: **if** $l = r$ **then**
5: **return** `BinaryCode`($S'[l]$, low, hi);
6: **end if**
7: Вычисляем $m = \left\lfloor \frac{l + r}{2} \right\rfloor$;
8: Вычисляем A_1 = `BinaryCode`($S'[m]$, $low + m - 1$, $hi - r + m$);
9: Вычисляем A_2 = интерполяционное кодирование $\langle S', l, m - 1, low, S'[m] - 1 \rangle$;
10: Вычисляем A_3 = интерполяционное кодирование $\langle S', m + 1, r, S'[m] + 1, hi \rangle$;
11: **return** контанкенацию A_1, A_2 и A_3;

На рис. 11.7 показан пример интерполяционного кодирования для возрастающей последовательности из 12 положительных целых чисел. Обратите внимание на два случая, в которых алгоритм не генерирует никаких битов (показаны как жирно очерченные светло-серые прямоугольники), поскольку два диапазона целых чисел, а именно {1, 2} и {19, 20, 21}, полностью плотные.

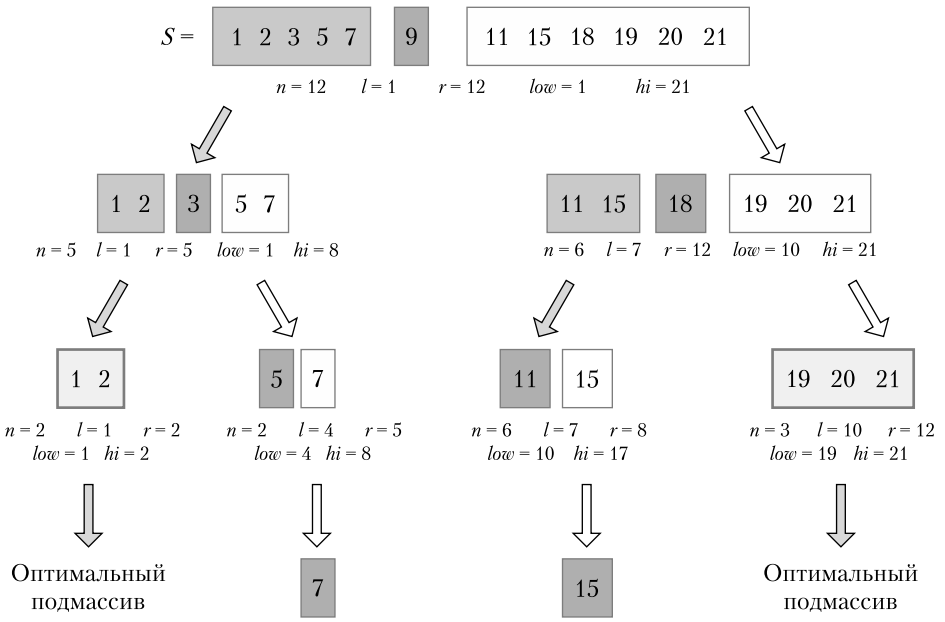


Рис. 11.7. Пример интерполяционного кодирования для возрастающей последовательности из 12 положительных целых чисел

Серые и белые поля на рис. 11.7 — левая и правая подпоследовательности каждой рекурсивной итерации алгоритма интерполяционного кодирования. Темно-серые поля выделяют целое кодируемое число s'_m. Явно показаны количество $n = r - l + 1$ кодируемых целых чисел, а также кортеж из четырех элементов, передаваемый каждому рекурсивному вызову. Два жирно очерченных светло-серых поля показывают подпоследовательности, для которых интерполяционный код не генерирует никаких битов. Процедура выполняет предварительный обход сбалансированного двоичного дерева, листьями которого являются целые числа в S'. Элементы кодируются в следующем порядке (фактическое закодированное число указано в скобках): 9 (3), 3 (0), 5 (1), 7 (1), 18 (6), 11 (1), 15 (4).

В заключение отмечу, что интерполяционное кодирование целого числа s'_i зависит от распределения других целых чисел в последовательности S'. Следовательно, это *адаптивный* код, который вдобавок оказывается *не*префиксным. Эти две особенности сильно отличают его от кодов, с которыми я познакомил вас раньше, и от рассматриваемого в следующей главе кода Хаффмана, оптимального среди класса

статических префиксных кодов. В результате для плотных целочисленных последовательностей интерполяционный код может оказаться намного более кратким, чем код Хаффмана, но это неудивительно, ведь теперь вы знаете, почему так получается.

11.6. Код Элиаса — Фано

Рассмотрим код, который, в отличие от интерполяционного кодирования, обеспечивает занятость пространства вне зависимости от распределения входных данных и, что более важно, может быть *проиндексирован* с помощью соответствующих сжатых структур данных для обеспечения эффективного *случайного* доступа к закодированным целым числам. Первая особенность — позитивный фактор в контексте хранения инвертированных списков поисковых систем и списков смежности больших графов, но негативный — в случае, когда целые числа *сгруппированы*, а использование пространства является ключевой проблемой базового приложения.

Недавно был предложен подход из своего рода динамического программирования, превращающий кодирование Элиаса — Фано в код, зависящий от распределения, как в случае интерполяционного кодирования. Это позволило объединить эффективность случайного доступа к закодированным целым числам первого с компактностью второго при сжатии сгруппированных целочисленных подпоследовательностей [6]. Эксперименты показали, что интерполяционное кодирование всего на 2–8 % меньше оптимизированного кода Элиаса — Фано и при этом работает медленнее в 5,5 раза, а код с переменной длиной байтов на 10–40 % быстрее оптимизированного кода Элиаса — Фано, но требует в 2,5 раза больше пространства. Это делает код Элиаса — Фано конкурентоспособным в ситуации, когда требуется сжать целочисленную последовательность, дав возможность получать случайный доступ к ее элементам.

Как и интерполяционный код, код Элиаса — Фано работает на *монотонно возрастающей* последовательности $S' = s'_1, \ldots, s'_n$, где $s'_i < s'_{i+1}$. Для наглядности пусть размер пространства $u = s'_n + 1$, а двоичное представление каждого целого числа s'_i занимает $b = \lceil \log_2 u \rceil$ бит. Двоичное представление s'_i разбивается на два блока: блок $L(s'_i)$ состоит из $\ell = \lceil \log_2(u/n) \rceil$ младших бит (самых правых), а блок $H(s'_i)$ — из $h = b - \ell$ старших бит (самых левых). Очевидно, что $b = \ell + h$.

Соответственно, код Элиаса — Фано состоит из двух двоичных последовательностей.

- Последовательность L получена конкатенацией блоков $L(s'_i)$ в порядке $i = 1, 2, \ldots, n$, в результате чего ее длина $n\ell = n\lceil \log_2(u/n) \rceil$ бит.
- Последовательность H получена перебором всех возможных конфигураций h бит, а именно от $j = 0^h = 0$ до $j = 1^h = 2^h - 1$, и использованием *отрицательного* унарного представления для кодирования числа x элементов s'_i, для которых $H(s'_i) = j$. В частности, значение x кодируется как 1^x0, то есть `1` повторяется x раз, поэтому двоичное представление `0` кодирует $x = 0$ — это случай, когда

не существует $H(s'_i) = j$. Каждое такое отрицательное унарное кодирование называется *корзиной* (bucket), потому что под этим понимается корзина целых чисел s'_i с определенной конфигурацией j. Последовательность H имеет n бит, установленных в `1`, потому что каждое число s'_i в отрицательном унарном представлении генерирует один бит, установленный в `1`, и число нулей равно количеству корзин, потому что каждый `0` устанавливает границы их кодирования. Так как максимальное значение сегмента равно $\lfloor u/2^{\ell} \rfloor$, количество нулей ограничено соотношением $\lfloor u/2^{\lceil \log_2 u/n \rceil} \rfloor \leqslant u/2^{\log_2 (u/n)} = n$. Следовательно, двоичная последовательность H имеет размер $2n$ бит и идеально сбалансированное количество нулей и единиц.

Пример кодирования набора S' из $n = 8$ целых чисел в пространстве размером $u = 32$ демонстрирует рис. 11.8. То есть $b = \lceil \log_2 32 \rceil = 5$ бит используются для представления целых чисел последовательности S', $\ell = \lceil \log_2 (u/n) \rceil = \lceil \log_2 32/8 \rceil = 2$ бита заняты под представление их наименее значимой части, а $h = b - \ell = 3$ бита отданы под представление наиболее значимой части. Соответственно, размер двоичной последовательности L составляет $ln = 2 \times 8 = 16$ бит, и двоичная последовательность H также состоит из 16 бит, поскольку у нас есть $2^3 = 8$ блоков (конфигураций $h = 3$ бита) и $n = 8$ кодируемых целых чисел. Для наглядности на рисунке показано, какая отрицательная унарная последовательность соответствует каждой конфигурации j из трех бит ($j = 0, 1... 7$). В частности, конфигурация `001` ($j = 1$) встречается как $H(s'_i)$ два раза — для целых чисел 4 и 7. Дело в том, что двоичная последовательность H кодирует эти два появления как `110`. При этом конфигурация `011` ($j = 3$) не встречается как $H(s'_i)$, поэтому двоичная последовательность H кодирует это событие как `0`.

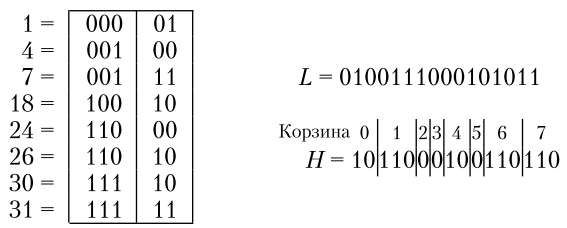


Рис. 11.8. Код Элиаса — Фано для целочисленной последовательности $S' = 1, 4, 7, 18, 24, 26, 30, 31$

Декодирование кода Элиаса — Фано происходит просто — достаточно поменять направление процесса кодирования S'. Из L выбираются группы ℓ, формирующие наименее значимую часть целых чисел S'. По сути, i-я группа обеспечивает l наименее значимые битов числа s'_i. Для получения h наиболее значимых бит числа s'_i последовательно просматривается двоичная последовательность H, и если i-й бит, установленный в `1`, принадлежит j-й отрицательной унарной последовательности, то j кодируется в двоичном виде, используя h бит. Таким образом, доказано следующее.

Теорема 11.5. *Код Элиаса — Фано для монотонно возрастающей последовательности из n целых чисел в диапазоне* $[0, u)$ *занимает менее* $2n + n\left\lceil \log_2 \frac{u}{n} \right\rceil$ бит *независимо от их распределения. Сжатие и распаковка такой целочисленной последовательности занимают время* $O(n)$. *Размер используемого пространства почти оптимален, когда целые числа равномерно распределены в* $[0, u)$. *Если быть точным, это кодирование требует менее* 2 *бит на целое число в дополнение к оптимальному коду из* $\lceil \log_2(u/n) \rceil$ *бит.*

Самое интересное свойство кода Элиаса — Фано — возможность дополнить его структурами данных для эффективной поддержки следующих операций:

- `Access`(i), которая при индексе $1 \leqslant i \leqslant n$ возвращает s'_i;
- `NextGEQ`(x), которая при целом $0 \leqslant x < u$ возвращает наименьший элемент s'_i, больше (**G**reater) или равный (**EQ**ual) x.

Основная идея, лежащая в основе дополнения H, заключается в использовании вспомогательной структуры данных, которая эффективно во времени и в пространстве реализует хорошо известный примитив $\texttt{Select}_1(p, H)$, возвращающий позицию в H p-го бита, установленного в `1` (или в `0` в случае $\texttt{Select}_0(p, H)$). Реализация этих двух операций и краткой структуры данных для примитива `Select` подробно описывается в главе 15, здесь же упомяну только то, что формирование примитива `Select` происходит за постоянное время и требует $o(|H|) = o(n)$ бит в дополнение к H (см., например, [5]). С учетом затрат на примитив `Select` две операции, `Access(i)` и `NextGEQ(x)`, выполняются за время $O(1)$ и $O(\log(u/n))$ соответственно, как будет показано в главе 15.

Здесь уместно прокомментировать обозначенную в начале этого раздела проблему. Поскольку код Элиаса — Фано представляет собой монотонную последовательность целых чисел, сгруппированные последовательности он сжимает значительно хуже, чем интерполяционный код. Рассмотрим последовательность $S' = (1, 2 \ldots n - 1, u - 1)$ из n целых чисел. Она хорошо сжимается, поскольку и длина первого прогона, и значение $u - 1$ допускают кодирование в $O(\log u)$ бит, в то время как код Элиаса — Фано требует $2 + \lceil \log_2(u/n) \rceil$ бит на элемент. Некоторые авторы искали способ сделать код Элиаса — Фано *чувствительным к распределению*, чтобы воспользоваться структурными преимуществами входной последовательности S' [6]. Были предложены два подхода. Один основывался на двухуровневой схеме хранения с разбиением последовательности S' на n/m фрагментов по m целых чисел в каждом, где параметр m определялся пользователем. Затем на первом уровне кодирование Элиаса — Фано применялось для последнего целого числа каждого фрагмента, то есть ему подвергались n/m целых чисел. На втором уровне определенное кодирование Элиаса — Фано применялось к каждому фрагменту, целые числа которого были закодированы с зазором относительно последнего целого числа предыдущего фрагмента, доступного на первом уровне. Если u_j — расстояние между первым и последним целыми числами j-го фрагмента, то на втором уровне кодирование Элиаса — Фано будет использовать при сжатии целых чисел $2 + \lceil \log(u_j/m) \rceil$ бит на

каждое. Это выгодно по сравнению с затратами на кодирование Элиаса — Фано всей последовательности S', ведь (u_j/m) — это среднее расстояние в пределах блока, тогда как (u/n) — среднее расстояние по всей последовательности S'. К сожалению, это преимущество частично утрачивается из-за пространства, занимаемого первым уровнем индексации. В целом благодаря этой несложной схеме целочисленного кодирования улучшение заполнения пространства классическим кодом Элиаса — Фано, который работает со всей последовательностью S', может достичь 30 %, но замедление времени распаковки может дойти до 10 %. По сравнению с интерполяционным кодом двухуровневая схема ухудшает заполнение пространства на 10 %, но достигает в 3–4 раза более быстрой распаковки. Второй подход, предложенный авторами [6], более сложен и зависит от интерпретации кодирования Элиаса — Фано S' как вычисления кратчайшего пути по графу, который по требованиям к пространству ближе к интерполяционному коду и по-прежнему обеспечивает очень быструю распаковку.

Список литературы

1. *Brisaboa N. R., Farina A., Navarro G., Paramá J. R.* Lightweight natural language text compression // Information Retrieval, 10: 1–33, 2007.
2. *Fenwick P.* Universal codes // *Sayood K.* Lossless Compression Handbook. Academic Press, 55–64, 2002.
3. *Manning C. D., Raghavan P., Schütze H.* Introduction to information retrieval. Cambridge University Press, 2008.
4. *Moffat A.* Compressing integer sequences and sets // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 178–183, 2009.
5. *Navarro G.* Compact Data Structures: A Practical Approach. Cambridge University Press, New York, 2016.
6. *Ottaviano G., Venturini R.* Partitioned Elias — Fano indexes // Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, 273–282, 2014.
7. *Pibiri G. E., Venturini R.* Techniques for inverted index compression // ACM Computing Surveys, 53(6): 125:1–125:36, 2021.
8. *Yan H., Ding S., Suel T.* Inverted index compression and query processing with optimized document ordering // Proceedings of the 18th International Conference on World Wide Web (WWW), 401–410, 2009.
9. *Witten I. H., Moffat A., Bell T. C.* Managing gigabytes. Morgan Kauffman, second edition, 1999.

Глава 12
СТАТИСТИЧЕСКОЕ КОДИРОВАНИЕ

> Информация — это устранение
> неопределенности.
>
> *Клод Шеннон*

Тема этой главы — статистическое кодирование последовательности S символов (текстов), составленных из алфавита Σ. Символы могут быть буквами, в этом случае задача называется *сжатием текста*, или нуклеотидами ДНК, для которых задача — сжатие геномной базы данных, или они являются битами, и в этом случае речь заходит о классическом сжатии данных. Когда в роли символов выступают целые числа, мы приходим к рассмотренной в предыдущей главе задаче целочисленного кодирования, которая решается в том числе с помощью статистического кодера. Для этого нужно получить статистическую информацию о целых числах в последовательности S. Созданный в этом случае код будет оптимальным префиксным кодом для целых чисел S, который в любом случае работает медленнее решений из предыдущей главы как в фазах сжатия, так и в фазах распаковки.

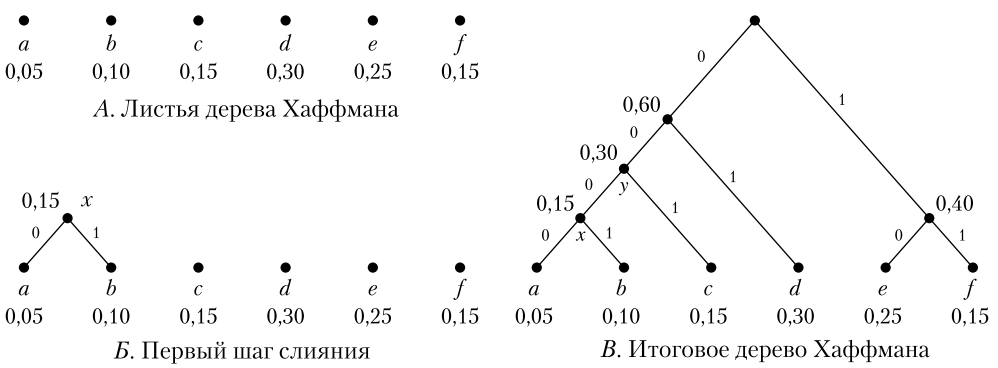
Концептуально статистическое сжатие можно рассматривать как процесс из двух фаз: *моделирования* и *кодирования*. Сначала вычисляются статистические свойства входной последовательности и строится *модель*, на базе которой генерируются кодовые слова из символов Σ, в дальнейшем применяющиеся для сжатия входной последовательности. В первых двух разделах этой главы будет рассматриваться только фаза кодирования и мы разберем наиболее известные статистические компрессоры: алгоритм Хаффмана, арифметическое и интервальное кодирование. И только в разделе 12.3 речь пойдет о сложной технике моделирования, которая будет использоваться для ознакомления с алгоритмом *предсказания по частичному совпадению* (prediction by partial matching, PPM). Этот алгоритм даст вам довольно полную картину возможностей, которыми обладают статистические компрессоры. В конечном счете от производительности сжатия, ограниченной в терминах энтропии нулевого порядка, а именно функции энтропии, зависящей от вероятности отдельных символов, которые считаются распределенными независимо, мы перейдем к более компактной энтропии k-го порядка, которая зависит от вероятности блоков символов размером k, — таким образом, моделирует, например, случай марковских источников.

12.1. Алгоритм Хаффмана

Опубликованный в начале 1950-х годов алгоритм Хаффмана на протяжении десятилетий считался одним из лучших методов сжатия данных, пока в конце 1960-х арифметическое кодирование не сделало возможным получение более высоких показателей сжатия.

Кодирование Хаффмана базируется на *жадной* алгоритмической схеме, конструирующей бинарное дерево, листья которого являются символами σ алфавита Σ. Вероятность появления в требующей сжатия входной последовательности листа σ обозначена $\mathcal{P}[\sigma]$. Все листья составляют *набор кандидатов*, обновляемый в процессе построения дерева Хаффмана. На общем шаге алгоритм выбирает из набора кандидатов два узла с наименьшими вероятностями и создает для них родительский узел. Вероятность для родительского узла равна сумме вероятностей узлов-потомков. Родительский узел вставляется в набор кандидатов, а дочерние узлы удаляются оттуда. Соответственно, на каждом шаге происходят добавление одного узла и удаление двух, и после $|\Sigma| - 1$ шагов процесс останавливается. В этот момент в наборе кандидатов остается только корень дерева. В конце жадного процесса дерево Хаффмана состоит из $t = |\Sigma| + (|\Sigma| - 1) = 2|\Sigma| - 1$ узлов, где $|\Sigma|$ — листья, а $(|\Sigma| - 1)$ — внутренние узлы.

Пример дерева Хаффмана для алфавита $\Sigma = \{a, b, c, d, e, f\}$ демонстрирует рис. 12.1. Первое слияние *(слева)* присоединяет символы *a* и *b* как дочерние элементы узла *x*, вероятность которого равна 0,05 + 0,1 = 0,15. Этот узел добавляется в набор кандидатов, а листья *a* и *b* оттуда удаляются. На втором этапе (см. рис. 12.1, *б*) двумя узлами с наименьшими вероятностями становятся лист *c* и только что вставленный узел *x*. Их слияние обновляет набор кандидатов путем удаления *x* и *c* и добавления родительского для них узла *y* с вероятностью 0,15 + 0,15 = 0,3. Алгоритм продолжается, пока не останется единственный узел (корень) с вероятностью 1.



А. Листья дерева Хаффмана

Б. Первый шаг слияния

В. Итоговое дерево Хаффмана

Рис. 12.1. Построение дерева Хаффмана для алфавита $\Sigma = \{a, b, c, d, e, f\}$. Под каждым символом указана его вероятность. Ребра дерева аннотированы битами 0 *(левый край)* и 1 *(правый край)*. Готовое дерево Хаффмана состоит из шести листьев и пяти внутренних узлов

Чтобы получить код Хаффмана для символов алфавита Σ, ребрам дерева нужно назначить бинарные метки. Как правило, 0 сопоставляется левому, а 1 — правому ребру каждого внутреннего узла. Но это всего один из множества возможных вариантов. Фактически дерево Хаффмана может генерировать $2^{|\Sigma|-1}$ деревьев с метками, поскольку для двух ребер, выходящих из каждого из $|\Sigma| - 1$ внутренних узлов, возможны два варианта маркировки, 0 – 1 или 1 – 0. При наличии меток кодовое слово для символа Σ получается чтением этих бинарных меток на нисходящем пути от корня к связанному с этим символом листу. Длина кодового слова равна глубине листа σ в дереве Хаффмана и обозначается $L(\sigma)$. В коде Хаффмана каждый символ связан с отдельным листом, таким образом, ни одно кодовое слово *не является префиксом* другого.

Понятно, что выбор двух узлов с минимальной вероятностью может быть не единственным, и доступные варианты дают различные кодовые слова с *одинаковой оптимальной средней* длиной, но, возможно, *разной максимальной* длиной. Минимизация этого значения полезна для уменьшения размера буфера сжатия/распаковки. Пример с двумя вариантами демонстрируется на рис. 12.2.

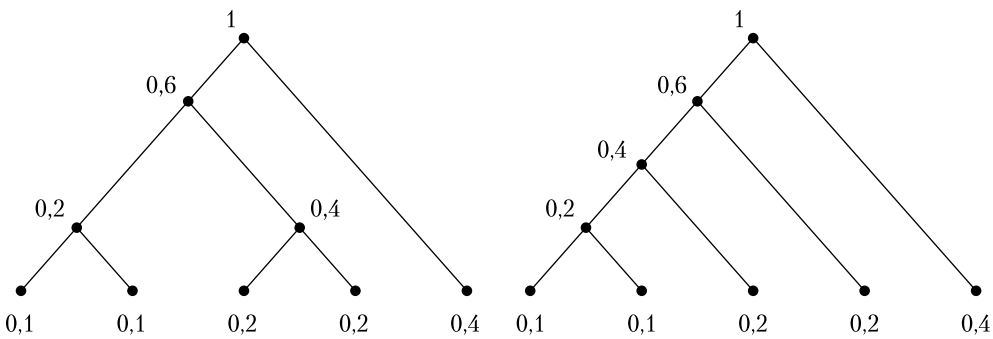


Рис. 12.2. Два кода Хаффмана с одинаковой средней длиной кодового слова 2,2 бита, но максимальными длинами кодового слова 3 и 4 бита

Стратегия минимизации максимальной длины кодового слова заключается в выборе из текущего набора кандидатов среди узлов с одинаковой вероятностью двух *самых старых*. Определение «*самые старые узлы*» означает, что эти листья или внутренние узлы были созданы раньше других. Для реализации этой стратегии используются две очереди: в первой находятся листья, упорядоченные по возрастанию вероятности, во второй внутренние узлы, расположенные в порядке их создания алгоритмом Хаффмана. Нетрудно заметить, что вторая очередь также отсортирована по возрастанию вероятности. При наличии более чем двух узлов с минимальной вероятностью алгоритм просматривает узлы в первой очереди, а затем во второй. На рис. 12.2 *слева* показано дерево, сгенерированное по этому принципу, а *справа* — дерево, полученное произвольным выбором.

Сжатый файл, сгенерированный алгоритмом Хаффмана, состоит из двух частей: *заголовка* (preamble), содержащего таблицу кодирования дерева Хаффмана и вероятности символов (его размер оценивается как $\Theta(|\Sigma|)$), и *тела* (body), содержащего кодовые слова символов входной последовательности S. При оценке длины сжатого файла размер заголовка обычно игнорируется, поскольку предполагается, что $|\Sigma| << |S|$. Но на практике встречаются ситуации, когда размер алфавита имеет значение, соответственно, требуется учитывать и размер заголовка. Оставшаяся часть этого раздела посвящена оценке размера сжатого тела файла в битах, после чего я приведу удачный пример кодирования дерева на базе элегантного *канонического кода Хаффмана*, который обеспечивает эффективность по занимаемому пространству и очень высокую скорость декодирования.

Пусть $L_C = \sum_{\sigma \in \Sigma} L(\sigma)\mathcal{P}[\sigma]$ — средняя длина кодовых слов, полученных с помощью префиксного кода C, который сжимает каждый символ алфавита Σ до $L(\Sigma)$ бит. Следующая теорема утверждает оптимальность алгоритма Хаффмана.

Теорема 12.1. *Если C — код Хаффмана, то L_C — наименьшая возможная средняя длина среди всех префиксных кодов C', то есть $L_C \leqslant L_{C'}$.*

Чтобы доказать это, первым делом отмечу, что префиксный код можно рассматривать как бинарное дерево (точнее, если вспомнить терминологию из главы 9, бинарное префиксное дерево). Это значит, что по-другому оптимальность кода Хаффмана можно описать как *минимальность средней глубины* соответствующего бинарного дерева. Это свойство доказывается с помощью ключевой леммы, что я оставляю вам в качестве упражнения.

Лемма 12.1. *Пусть $\mathcal{F}$ — набор взвешенных бинарных деревьев, листьям которых сопоставлены определенные вероятности, а их средняя глубина минимальна среди глубин всех бинарных деревьев с $|\Sigma|$ листьями. В наборе $\mathcal{F}$ существует дерево T, у которого два листа с минимальными вероятностями находятся на наибольшей глубине и являются потомками одного и того же родительского узла.*

На самой большой глубине все деревья в наборе $\mathcal{F}$ будут иметь два листа с минимальными вероятностями, хотя они могут быть прикреплены к разным родителям. Фактически чем глубже лист, тем больше его вес при вычислении средней глубины дерева. Поэтому для получения дерева с минимальной средней глубиной вниз лучше опустить листья с наименьшей вероятностью. В частности, если такой лист находится не в самой глубине, его можно поменять местами с листом, который там располагается, уменьшив тем самым среднюю глубину дерева. Таким образом, максимально глубоко должны находиться по крайней мере два листа с наименьшей вероятностью, но для всех таких листьев это не обязательно. Возьмем четыре символа $\{a, b, c, d\}$ с вероятностями $[.1, .1, .1, .7]$. Дерево Хаффмана может сначала объединить (a, b), а результирующий узел — с узлом c, в результате чего они окажутся на разной глубине. Более того, напомню, что два листа с наименьшей вероятностью

могут быть потомками разных родителей. Рассмотрим пять символов $\{a, b, c, d, e\}$ с вероятностями [.1, .1, .11, .11, .58]. Алгоритм Хаффмана сначала объединит (a, b), а затем (c, d) и, наконец, сделает их потомками одного узла, который является родителем узла e. Можно построить и другое дерево с минимальной средней глубиной, объединив сначала узлы (a, c), затем — узлы (b, d) и, наконец, сделав их потомками узла, который является родителем узла e. В таком дереве два листа с минимальными вероятностями будут иметь разных родителей, несмотря на то что оба находятся на наибольшей глубине. В любом случае по крайней мере одно дерево набора F будет удовлетворять обоим свойствам, указанным в лемме 12.1.

Прежде чем перейти к доказательству теоремы 12.1, рассмотрим еще одну техническую лемму. Пусть алфавит Σ состоит из n символов, а символы x и y имеют наименьшую вероятность. На основе этого алфавита кодом C сгенерировано бинарное дерево T_C. *Сокращенное дерево*, полученное отбрасыванием листьев для символов x и y, обозначим R_C. В этом случае предок листьев x и y (это z) будет листом R_C с вероятностью $\mathcal{P}[z] = \mathcal{P}[x] + \mathcal{P}[y]$. Соответственно, R_C — бинарное взвешенное дерево с $n - 1$ листьями для алфавита $\Sigma - \{x, y\} \cup \{z\}$. Бинарные деревья T_C и R_C показаны на рис. 12.3, а лемма 12.2 устанавливает связь между их средними глубинами.

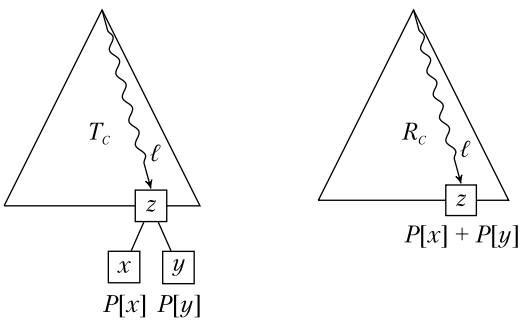


Рис. 12.3. Связь между бинарным взвешенным деревом T и соответствующим ему сокращенным взвешенным деревом R, как указано в лемме 12.2

Лемма 12.2. *Связь между средней глубиной бинарного взвешенного дерева T и средней глубиной его сокращенного бинарного взвешенного дерева R задается формулой $L_T = L_R + (\mathcal{P}[x] + \mathcal{P}[y])$, где x и y — два листа с наименьшей вероятностью.*

Доказательство. Достаточно записать L_T и L_R как суммы длин всех путей от корня к листу, умноженные на вероятность целевого листа. В первом случае $L_T = (\sum_{\sigma \neq x, y} \mathcal{P}[\Sigma] L(\Sigma)) + (\mathcal{P}[x] + \mathcal{P}[y])(L(z) + 1)$, где z — родительский элемент x и y, таким образом, $L(x) = L(y) = L(z) + 1$. Аналогично можно записать $L_R = (\sum_{\sigma \neq x, y} \mathcal{P}[\sigma] L(\sigma)) + L(z)(\mathcal{P}[x] + \mathcal{P}[y])$. Лемма доказана. ■

Теперь оптимальность кода Хаффмана, заявленную в теореме 12.1, можно доказать методом математической индукции. Базовый случай $n = 2$ очевиден, потому что любой код без префиксов должен сопоставлять каждому из двух символов алфавита Σ по крайней мере один бит, следовательно, код Хаффмана оптимален, ведь одному символу он назначает бит 0, а второму — бит 1.

Теперь предположим, что код Хаффмана оптимален для алфавита из $n - 1$ символов, где $n > 2$. Пусть $|\Sigma| = n$, а C — оптимальный код для Σ и его базового распределения. Нужно показать, что $L_C = L_H$, то есть что код Хаффмана будет оптимален и для n символов. Очевидно, что $L_C \leqslant L_H$, ведь C предполагается оптимальным кодом для алфавита Σ. Рассмотрим два сокращенных дерева, R_C и R_H, полученных из T_C и T_H соответственно путем отбрасывания листьев x и y с наименьшей вероятностью и оставления их родителя z. Это можно сделать в соответствии с леммой 12.1 (для оптимального кода C) и структурой алгоритма Хаффмана. Сокращенные деревья определяют префиксный код для алфавита из $n - 1$ символов. По индукции код, определяемый деревом R_H, оптимален для редуцированного алфавита $\Sigma \cup \{z\} - \{x, y\}$. Следовательно, для него будет $L_{R_H} \leqslant L_{R_C}$. Согласно лемме 12.2 можно записать, что средняя глубина T_H составляет $L_H = L_{R_H} + \mathcal{P}[x] + \mathcal{P}[y]$, а средняя глубина T_C — $L_C = L_{R_C} + \mathcal{P}[x] + \mathcal{P}[y]$. Таким образом, получаем $L_H \leqslant L_C$, что в сочетании с предыдущим (обратным) неравенством из-за оптимальности C дает $L_H = L_C$. Фактически это означает, что код Хаффмана будет оптимальным и для алфавита из n символов, и далее по индукции для алфавита произвольного размера.

Это утверждение не означает, что $C = H$. Существуют оптимальные префиксные коды, которые нельзя получить с помощью алгоритма Хаффмана (рис. 12.4). Скорее, оно указывает на равенство средних длин кодовых слов C и H. Количественную верхнюю границу средней длины определяет фундаментальная теорема 12.2.

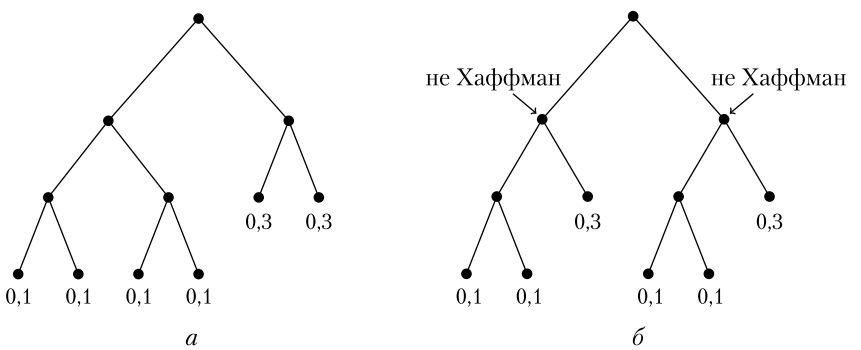


Рис. 12.4. Пример оптимального кода, который невозможно получить с помощью алгоритма Хаффмана: *а* — дерево Хаффмана; *б* — получено не алгоритмом Хаффмана

Теорема 12.2. *Пусть $\mathcal{H}$ — энтропия источника символов алфавита* Σ. *Следовательно,* $\mathcal{H} = \Sigma_{\sigma \in \Sigma} \mathcal{P}[\sigma] \log_2 1/\mathcal{P}[\sigma]$. *Средняя длина кодового слова кода Хаффмана удовлетворяет неравенствам* $\mathcal{H} \leqslant L_H < \mathcal{H} + 1$.

Доказательство. Первое неравенство вытекает из теоремы Шеннона для канала без шума, сложное доказательство которой можно найти в его прекрасной оригинальной статье [6] или любом другом классическом тексте по теории информации. Чтобы доказать второе неравенство, определим $\ell_\sigma = \left\lceil \log_2 \left(1/\mathcal{P}(\sigma)\right) \right\rceil$. Это наименьшее целое число, ограничивающее сверху оптимальную длину кодового слова Шеннона для символа σ, заданного энтропийным компонентом. Простым арифметическим преобразованием можно получить неравенство $\sum_{\sigma \in \Sigma} 2^{-\ell_\sigma} \leqslant 1$. Согласно неравенству Крафта — Макмиллана существует бинарное дерево из $|\Sigma|$ листьев с длиной ℓ_σ пути от корня к листьям для каждого σ. Именно оно обеспечивает код C для символов алфавита Σ со средней длиной кодового слова $L_C = \Sigma_{\sigma \in \Sigma} \mathcal{P}[\sigma] \ell_\sigma$. Согласно теореме 12.1, код Хаффмана оптимален и известно, что $L_H \leqslant L_C$, это доказывает тезис по определению энтропии $\mathcal{H}$ и неравенства $\ell_\sigma < 1 + \log_2 (1/\mathcal{P}[\sigma])$. ■

Эта теорема утверждает, что код Хаффмана может терять относительно энтропии $\mathcal{H}$ базового источника до одного бита на сжатый символ. В зависимости от значения $\mathcal{H}$ этот дополнительный бит может быть как важным, так и без проблем игнорируемым. Очевидно, что $\mathcal{H} \geqslant 0$, ведь $\mathcal{P}(\sigma) \in [0, 1]$ и, таким образом, $\log_2 1/\mathcal{P}[\sigma] \geqslant 0$. Энтропия равна нулю, когда источник генерирует один символ с вероятностью 1, а все остальные символы — с вероятностью 0. Более того, функция энтропии вогнутая, а ее максимум достигается, когда все символы равновероятны, что дает значение $\mathcal{H} = \log_2 |\Sigma|$. В результате для большого алфавита почти равномерно распределенных символов $\mathcal{H}$ может иметь произвольно большое значение. В этом случае (при $\mathcal{H} >> 1$) код Хаффмана эффективен, а дополнительным битом можно пренебречь. В противном случае (при $\mathcal{H} \approx 0$) распределение *асимметрично* в сторону одного или нескольких символов алфавита, и, возможно, потраченный впустую дополнительный бит делает код Хаффмана неэффективным, поскольку, как и для любого кода без префиксов, алгоритм Хаффмана не может использовать менее одного бита на символ. Таким образом, наилучшую степень сжатия алгоритм Хаффмана дает при кодировании каждого символа одним битом, начиная с его полного представления $\log_2 |\Sigma|$ бит. Такое идеальное кодирование позволяет достичь коэффициента сжатия $1/\log_2 |\Sigma|$. В стандарте ASCII $|\Sigma| = 256$, и алгоритм Хаффмана для любой последовательности S не может получить степени сжатия, меньшей чем $1/8 = 12{,}57$ %.

Обход этого ограничения Шеннон предложил в своей знаменитой статье 1948 года [6]. Это была простая *схема группировки*, использовавшая расширенный алфавит, символы которого представляли собой подстроки, состоящие из k символов каждая. Размер нового алфавита был равен $|\Sigma|^k$. Если применить к нему алгоритм Хаффмана, один дополнительный бит будет впустую расходоваться на блок, а не на

символ. Соответственно, потери на один символ составляют $1/k$ часть бита, что при больших k составляет очень незначительную величину. Почему большие значения k выбираются не всегда? Ведь часто это увеличивает коэффициент сжатия благодаря группировке, фиксирующей связи между соседними символами. Но, во-первых, k не может превышать длину текста. И во-вторых, что еще важнее, чем больше k, тем больше дерево Хаффмана, ведь количество листьев/символов в дереве растет как $|\Sigma|^k$, которое должно храниться в заголовке сжатого файла. Вот почему «умный» компрессор всегда должен тщательно выбирать правильное значение k, которое может меняться в зависимости от сжимаемого текста. Так что, несмотря на свою жизнеспособность, эта стратегия в любом случае неоптимальна, как я покажу в следующих разделах.

Последнее замечание касается случая с очень длинными кодовыми словами. Если длина кодового слова превысит 32 бита, операции могут стать дорогостоящими, ведь кодовые слова будет невозможно хранить как одно машинное слово. Когда возникает такой патологический случай? Если учесть тот факт, что оптимальный код должен сопоставлять символу σ кодовое слово длиной $\lceil \log_2(1/\mathcal{P}(\sigma)) \rceil$ бит, можно сделать вывод, что для $L(\sigma) > 32$ вероятность $\mathcal{P}[\sigma]$ должна быть около 2^{-33}. То есть плохая ситуация возникает только после обработки около 2^{33} символов. К сожалению, это чрезмерная верхняя граница. Достаточно рассмотреть асимметричное дерево со смещением влево Хаффмана, у которого лист i имеет частоту $F(i) < F(i + 1)$. Более того, предполагается, что заставить алгоритм Хаффмана соединять $F(i + 1)$ с последним созданным внутренним узлом, а не с листом $i + 2$ (или любым другим листом $i + 3$, $i + 4$...) можно только при соблюдении неравенства $\sum_{j=1}^{i} F(j) < F(i+2)$. Нетрудно заметить, что последовательность $F(i)$ можно принять за последовательность Фибоначчи, возможно, с другими начальными условиями $F(1) = F(2) = F(3) = 1$. В этом случае $F(33) = 3{,}01 \times 10^6$ и $\sum_{i=1}^{33} F(i) = 1{,}28 \times 10^7$. Накопленная сумма показывает, какой объем текста необходимо прочитать, чтобы принудительно получить кодовое слово длиной 33 бита.

Таким образом, патологический случай может возникнуть всего через 10 млн символов, что значительно ниже предыдущей оценки! Существуют методы уменьшения длины кодового слова, гарантирующие хорошую производительность сжатия [8], например *итеративное масштабирование* вероятностей символов. Классический алгоритм Хаффмана применяется к символам, вероятности которых приблизительно оцениваются в соответствии с количеством появлений каждого из них во входном тексте. Если самое длинное кодовое слово оказывается больше максимально допустимого количества битов L, все счетчики символов уменьшаются на фиксированное число, например 2 или золотое сечение 1,618, и для нового распределения вероятностей символов алфавита создается новый код Хаффмана. Процесс продолжается до тех пор, пока не будет сгенерирован код с максимальной длиной кодового слова L или меньше. В предельном случае счетчики всех символов будут иметь значение 1, то есть все придет к коду фиксированной длины.

12.1.1. Канонический алгоритм Хаффмана

Напомню два основных ограничения, накладываемых алгоритмом Хаффмана.

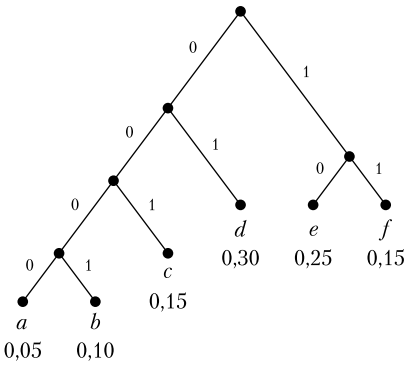
- Необходимо хранить структуру дерева, что в случае большого алфавита Σ может оказаться затратным, например, при кодировании блоков символов или слов.
- Декодирование медленное, потому что для каждого кодового слова приходится обходить все дерево. Кроме того, каждое ребро (бит кодового слова) может вызвать промах кэша. В результате общее количество промахов кэша может оказаться равным общему количеству битов, составляющих сжатый файл.

Существует элегантный вариант алгоритма Хаффмана, называемый *каноническим*, который устраняет эти проблемы путем реструктуризации дерева Хаффмана для обеспечения быстрого декодирования и малого объема занимаемой памяти. Суть его состоит в переходе к другому дереву Хаффмана, которое с точки зрения длин кодовых слов, соответствующих символам алфавита, эквивалентно исходному, но структурировано так, что по мере продвижения от самого левого к самому правому листу длины путей не увеличиваются. Такая реструктуризация достигается за пять шагов без манипулирования указателями дерева, а только с помощью массивов и основных арифметических операций.

1. Вычисляем длину кодового слова $L(\sigma)$ для каждого символа $\sigma \in \Sigma$ по классическому алгоритму Хаффмана. Обозначаем `max` максимальную длину кодового слова (в битах).
2. Создаем массив `symb`[1, `max`], хранящий в записи `symb`[ℓ] список символов, у которых кодовое слово Хаффмана состоит из ℓ бит.
3. Создаем массив `num`[1, `max`], хранящий в записи `num`[ℓ] количество символов, для которых кодовое слово Хаффмана состоит из ℓ бит.
4. Получаем из массива `num` (потом он уничтожается) массив `fc`[1, `max`], хранящий в записи `fc`[ℓ] первое кодовое слово всех символов, закодированных ℓ битами. Это сложный переход, в котором никак не используется структура дерева Хаффмана. Он осуществляется с помощью арифметических операций, которые будут подробно описаны в дальнейшем.
5. Неявно назначаем идущие подряд кодовые слова, каждое длиной ℓ бит, символам в массиве `symb`[ℓ], начиная с кодового слова `fc`[ℓ]. Неявно в данном случае означает, что они не хранятся в памяти, а создаются на лету в процессе кодирования/декодирования, как подробно описано в алгоритме 12.1.

В конце канонический алгоритм Хаффмана хранит только массивы `fc` и `symb`. Пространственная сложность массива `fc` не превышает `max`^{2} бит. А для таблицы кодирования `symb` она составит не более $(|\Sigma| + \texttt{max}) \log_2 (|\Sigma| + 1)$ бит (обратите внимание на то, что `max` $< |\Sigma|$). Следовательно, первое ключевое преимущество канонического алгоритма Хаффмана состоит в отсутствии необходимости явно хранить структуру дерева с помощью указателей, что экономит $\Theta(|\Sigma| \log_2 |\Sigma|)$ бит. Более того, в заголовке сжатого файла можно хранить только длины кодовых слов символов алфавита $|\Sigma|$, что в общей сложности составляет $|\Sigma| \log_2 |\Sigma|$ бит, в то время

как хранение информации о частоте символов заняло бы $|\Sigma| \log_2 n$ бит. Этой информации достаточно для перестроения канонического кода Хаффмана. Пример дерева Хаффмана, удовлетворяющего каноническому свойству, демонстрирует рис. 12.5.



symb	код	ℓ
a	0000	4
b	0001	4
c	001	3
d	01	2
e	10	2
f	11	2

ℓ	symb			num	fc
	0	1	2		
1				0	2
2	d	e	f	3	1
3	c			1	1
4	a	b		2	0

Рис. 12.5. Пример применения канонического алгоритма Хаффмана

Обратите внимание: дерево на рис. 12.5 сформировано таким образом, что при переходе от самого левого к самому правому листу длины путей не увеличиваются. Верхняя таблица демонстрирует результат работы алгоритма на шести символах $\{a, b, c, d, e, f\}$, вероятности которых указаны под соответствующими листьями. Нижняя таблица определяет все массивы, рассчитанные каноническим алгоритмом Хаффмана, а двоичное дерево является его графическим представлением.

Другое важное преимущество канонического алгоритма Хаффмана заключается в том, что процедура декодирования не нуждается в перемещениях по дереву. Она работает только с двумя доступными массивами, тем самым вызывая максимум два промаха кэша на один декодированный символ[1]. Псевдокод этой процедуры представлен в алгоритме 12.1. Корректность вычисления массива `fc` на шаге 4 будет доказана позже.

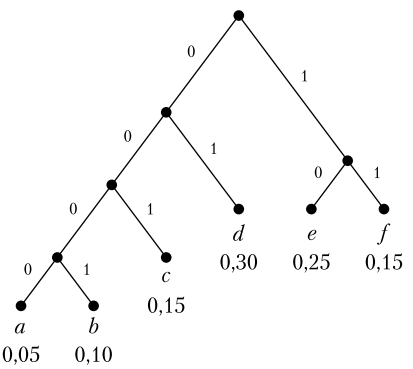
Алгоритм 12.1. Декодирование одного символа каноническим алгоритмом Хаффмана

```
1: v = next_bit();
2: ℓ = 1;
3: while v < fc[ℓ] do
4:     v = 2v + next_bit();
5:     ℓ++;
6: end while
7: return symb[ℓ, v − fc[ℓ]];
```

[1] Разумно предположить, что число промахов кэша равно 1, поскольку массив fc мал и может быть помещен в кэш.

Корректность процедуры декодирования вытекает из структуры канонического дерева Хаффмана. Фактически условие продолжения цикла `while` $v <$ `fc`$[\ell]$ проверяет, находится ли текущее кодовое слово v, занимающее ℓ бит, левее первого кодового слова этого уровня, а именно `fc`$[\ell]$. Если это так, то из-за наклона канонического дерева Хаффмана влево кодовое слово v находится левее *всех* закодированных символов длиной ℓ бит. Это означает большую длину декодируемого кодового слова, поэтому в теле цикла `while` извлекается еще один бит. Если же $v \geqslant$ `fc`$[\ell]$, то текущее кодовое слово больше по значению, чем первое кодовое слово длиной ℓ бит, следовательно, v соответствует листу канонического дерева Хаффмана на этом уровне ℓ и этот лист имеет смещение $v -$ `fc`$[\ell]$.

Чтобы лучше понять эти два случая, проанализируем декодирование сжатой последовательности 01, которое демонстрирует рис. 12.6. Функция `next_bit()` считывает первый входящий бит, а именно 0. Изначально мы имеем $\ell = 1$, $v = 0$ и `fc`$[1] = 2$, то есть условие продолжения цикла соблюдается ($v = 0 < 2 =$ `fc`$[1]$) и алгоритм знает, что кодовое слово для декодирования длиннее. Таким образом, ℓ увеличивается до значения 2 (следующий уровень канонического дерева Хаффмана), а v получает следующий бит 1, таким образом, $v = 01 = 1$. Условие продолжения цикла перестает выполняться — $v = 1 \geqslant$ `fc`$[2] = 1$. Алгоритм обнаруживает кодовое слово длиной $\ell = 2$ и, поскольку $v -$ `fc`$[2] = 0$, возвращает первый символ списка, на который указывает `symb`$[2]$, а именно `symb`$[2, 0] = d$.



symb	код	ℓ
a	0000	4
b	0001	4
c	001	3
d	01	2
e	10	2
f	11	2

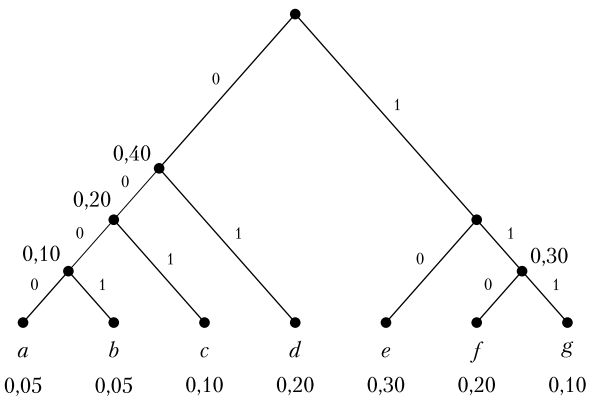
ℓ	symb			fc
	0	1	2	
1				2
2	d	e	f	1
3	c			1
4	a	b		0

ℓ	symb			fc
	0	1	2	
1				2
2	d	e	f	1
3	c			1
4	a	b		0

Рис. 12.6. Каноническое дерево Хаффмана (см. рис. 12.5) и все кодовые слова для символов алфавита. Выделено кодовое слово, соответствующее входным битам 01. Левая таблица относится к записи массива fc, проверенной в процессе декодирования на первой итерации цикла while при извлечении из ввода бита 0. Правая таблица относится к записи массива fc, проверенной на второй итерации цикла while при извлечении бита 1

Обратите внимание на то, что значение `fc[1]` = 2 кажется невозможным, ведь значение 2 нельзя представить с помощью кодового слова из одного бита. Но это *специальное значение*, заставляющее алгоритм переходить на следующий уровень, так как `fc[1]` больше любого кодового слова из одного бита.

Теперь все готово для того, чтобы описать построение канонического дерева Хаффмана в случае, если исходное распределение символов не приводит к дереву с этим свойством. Каноническое дерево Хаффмана, которое было построено благодаря распределению входных символов, демонстрирует рис. 12.5. Но так получается далеко не всегда. К примеру, дерево Хаффмана, изображенное на рис. 12.7, неканоническое, но его можно сделать каноническим с помощью шести строк псевдокода из алгоритма 12.2. Я подробно опишу для него процесс вычисления массива `fc`. Напомню, что *max* — это наибольшая длина кодового слова (в рассматриваемом примере *max* = 4).



symb	код	ℓ
a	0000	4
b	0001	4
c	001	3
d	01	2
e	10	2
f	110	3
g	111	3

ℓ	symb			num	fc
	0	1	2		
1				0	2
2	*d*	*e*		2	2
3	*c*	*f*	*g*	3	1
4	*a*	*b*		2	0

Рис. 12.7. От дерева Хаффмана к каноническому дереву Хаффмана

Алгоритм 12.2. Вычисление массива fc в каноническом алгоритме Хаффмана

```
1: fc[max] = 0;
2: i = max – 1;
3: while i >= 1 do
4:    fc[i] = (fc[i + 1] + num[i + 1])/2;
5:    i = i – 1;
6: end while
```

Прежде чем доказывать корректность алгоритма 12.2, нужно сделать два важных замечания. Во-первых, напомню, что `fc`[ℓ] — это значение кодового слова из ℓ бит, поэтому, если двоичное представление значения, хранящегося в `fc`[ℓ], короче ℓ, его следует дополнить нулями. Во-вторых, алгоритм устанавливает `fc`[*max*] = 0,

поэтому самое длинное кодовое слово представляет собой последовательность из max нулей и все дерево, построенное каноническим алгоритмом Хаффмана, наклонено влево. Теперь проанализируем формулу, по которой псевдокод вычисляет `fc`[ℓ]. По индукции на уровне $\ell + 1$ первое кодовое слово `fc`[$\ell + 1$], а сам уровень состоит из `num`[$\ell + 1$] листьев. Так что все кодовые слова от `fc`[$\ell + 1$] до `fc`[$\ell + 1$] + `num`[$\ell + 1$] – 1 могут быть зарезервированы для всех символов, хранящихся в `symb`[$\ell + 1$]. Первое *неиспользуемое* кодовое слово из $\ell + 1$ бит задается значением `fc`[$\ell + 1$] + `num`[$\ell + 1$]. Согласно формуле в псевдокоде это значение делится на 2, что соответствует отбрасыванию из двоичного кода этого числа самого старшего бита. В контексте бинарного дерева это эквивалентно выбору предка для узла, прописывающего битовую строку `fc`[$\ell + 1$] + `num`[$\ell + 1$], которая находится на глубине ℓ и не является префиксом ни одного кодового слова длиной $\ell + 1$. Таким образом, эту последовательность битов можно взять в качестве первого кодового слова `fc`[ℓ]. Для примера, приведенного на рис. 12.7, этот алгоритм дает `fc`[1] = 2, то есть оговоренное ранее специальное значение.

12.2. Арифметическое кодирование

Главное преимущество арифметического кодирования, введенного Элиасом в 1960-х годах, заключается в возможности кодировать символы произвольно близко к энтропии нулевого порядка, возможно тратя на один символ долю бита, что позволяет достичь гораздо лучшего сжатия, чем дает алгоритм Хаффмана на асимметричных распределениях. Так что с точки зрения теории Шеннона оно оптимально.

Для наглядности рассмотрим входной алфавит $\Sigma = \{a, b\}$ с асимметричным распределением $P[a] = 99/100$ и $P[b] = 1/100$. Согласно Шеннону энтропия такого источника составляет $\mathcal{H} = \mathcal{P}(a) \log_2 (1/\mathcal{P}(a)) + \mathcal{P}(b) \log_2 (1/\mathcal{P}(b)) \simeq 0{,}08056$ бита. Напротив, алгоритм Хаффмана, как и любой префиксный кодировщик, применяемый к каждому символу сгенерированного этим источником входного текста, должен использовать по крайней мере *один бит на символ*, что почти в 10 раз превышает энтропию источника. Так что в смысле степени сжатия алгоритм Хаффмана далек от энтропии нулевого порядка, и очевидно, что чем более асимметрично распределение символов, тем дальше этот алгоритм от оптимальности. Как я уже отмечал, код Хаффмана не позволяет получить коэффициент сжатия выше чем $1/\log_2 |\Sigma|$. Лучший результат достижим, когда один символ, закодированный в $\log_2 |\Sigma|$ битах, заменяется всего одним битом. Это 1/8 = 12,5 % в случае, когда используются 256 символов кода ASCII.

Чтобы решить эту проблему, арифметическое кодирование смягчает требование определять для каждого отдельного символа кодовое слово без префикса, принимая стратегию, в которой каждый бит сжатого вывода может представлять *более одного* входного символа. Это улучшает сжатие ценой замедления алгоритма и потери возможности доступа/декодирования сжатого вывода из любой позиции. Еще одна интересная особенность арифметического кодирования заключается

в том, что оно легко работает для *динамической* модели распределения вероятностей, в которой вероятности $\mathcal{P}(\Sigma)$ обновляются по мере обработки входной последовательности S. Для этого достаточно установить $\mathcal{P}(\sigma) = (\ell_\sigma + 1)/(\ell + |\Sigma|)$, где ℓ — длина префикса S, обработанного на данный момент, а ℓ_σ — количество символов Σ в этом префиксе. Вы можете самостоятельно удостовериться, что это корректное распределение вероятностей, изначально заданное как равномерное, так как $\ell = 0$ и $\ell_\sigma = 0$ для всех символов $\sigma \in \Sigma$. Динамические вероятности довольно легко обновляются алгоритмом декомпрессии, так что и компрессор, и декомпрессор смотрят на одно и то же входное распределение и кодируют/декодируют одни и те же символы.

12.2.1. Потоки битов и двоично-рациональные дроби

Поток битов (возможно, бесконечный) $b_1 b_2 b_3 \ldots b_k$ можно интерпретировать как действительное число в диапазоне [0, 1), добавив к нему 0:

$$0.b_1 b_2 b_3 \ldots b_k = \sum_{i=1}^{k} b_i \cdot 2^{-i}.$$

Верно и обратное: действительное число x в диапазоне [0, 1) допускает преобразование в последовательность битов (возможно, бесконечную). Это делается с помощью алгоритма `CONVERTER`, псевдокод которого приведен в алгоритме 12.3. Алгоритм состоит из цикла, в котором переменная *output* отдана под выходной битовый поток, а оператор `::` выполняет конкатенацию битов. Цикл завершается при достижении определенного уровня точности представления x. Остановиться можно, когда на выход подано определенное количество битов, или когда выясняется, что представление x периодическое, или когда кодируемое значение равно нулю.

Алгоритм 12.3. Converter (x, k)

На вход: Действительное число $x \in [0, 1)$, положительное целое число k.

Получить: строку, представляющую число x в k битах.

```
1: repeat
2:     x = 2x
3:     if x < 1 then
4:         output = output :: 0
5:     else
6:         output = output :: 1
7:         x = x − 1
8:     end if
9: until не будет сгенерировано k бит
```

Чтобы прояснить, как работает алгоритм `CONVERTER`, рассмотрим пример с $x = 1/3$, не фиксируя верхнюю границу для k:

$$\frac{1}{3}\cdot 2 = \frac{2}{3} < 1 \rightarrow \text{вывод} = 0;$$
$$\frac{2}{3}\cdot 2 = \frac{4}{3} \geqslant 1 \rightarrow \text{вывод} = 1.$$

На второй итерации x оказывается больше 1, поэтому алгоритм `CONVERTER` генерирует бит 1 и обновляет x, выполняя шаг 7: $4/3 - 1 = 1/3$. С таким значением x мы уже сталкивались, поэтому делаем вывод, что выходным значением является периодическое представление 01.

Рассмотрим еще один пример, скажем `CONVERTER`$(3/32, 5)$:

$$\frac{3}{32}\cdot 2 = \frac{6}{32} < 1 \rightarrow \text{вывод} = 0;$$
$$\frac{6}{32}\cdot 2 = \frac{12}{32} < 1 \rightarrow \text{вывод} = 0;$$
$$\frac{12}{32}\cdot 2 = \frac{24}{32} < 1 \rightarrow \text{вывод} = 0;$$
$$\frac{24}{32}\cdot 2 = \frac{48}{32} \geqslant 1 \rightarrow \text{вывод} = 1;$$
$$\left(\frac{48}{32} - 1\right)\cdot 2 = 1 \geqslant 1 \rightarrow \text{вывод} = 1;$$
$$(1-1)\cdot 2 = 0 \rightarrow \text{конец}.$$

В конце алгоритм `CONVERTER` даст значение `00011`. Такой же результат в этом случае можно получить, обратив внимание на то, что подлежащее кодированию число представлено *двоичной рациональной дробью*, а именно дробью вида $v/2^k$, где v и k — положительные целые числа. Двоичную рациональную дробь можно закодировать напрямую, без алгоритма `CONVERTER`, сгенерировав битовую последовательность $.bin_k(v)$, где $bin_k(v)$ — двоичное представление целого числа v в виде строки из k бит, в конце дополненное нулями. В предыдущем примере $k = 5$ и $v = 3$, соответственно, $bin_5(3)$ = `00011`, что совпадает с результатом, который дал алгоритм `CONVERTER`.

12.2.2. Алгоритм сжатия

Сжатие арифметическим кодированием выполняется итеративно. На каждом шаге в качестве входных данных принимается подынтервал [0, 1), представляющий собой префикс уже сжатой входной последовательности, *вероятности* символов алфавита и их *накопленные вероятности*, после чего обрабатывается следующий

входной символ[1]. Потом входной подынтервал делится на меньшие интервалы, по одному на каждый символ Σ алфавита Σ, длина которого пропорциональна его вероятности $\mathcal{P}(\sigma)$. На этом этапе в качестве выходных данных создается новый подынтервал, связанный с прочитанным входным символом и содержащийся внутри предыдущего подынтервала. Количество шагов равно количеству кодируемых символов, то есть длине входной последовательности.

Другими словами, первым делом алгоритм рассматривает интервал [0, 1), а после обработки всего ввода выдает интервал $[l, l + s)$, связанный с последним символом входной последовательности. Сложность в том, что на выходе мы получаем не пару $\langle l, s\rangle$ (два действительных числа), а действительное число $x \in [l, l + s)$, выбранное таким образом, чтобы это была двоично-рациональная дробь, плюс длину входной последовательности.

В следующем разделе я покажу, как выбрать это значение, минимизировав количество выходных битов. Пока же сосредоточимся на обобщенной процедуре сжатия, псевдокод которой представлен в алгоритме 12.4. Переменные l_i и s_i — это соответственно левый экстремум и длина интервала, кодирующего префикс входной последовательности длиной i. Когда для оценки вероятностей символов алфавита арифметическое кодирование использует полустатическую модель, входные данные алгоритма 12.4 могут состоять только из входной последовательности S, поскольку вероятности $\mathcal{P}(\sigma)$ оцениваются посредством сканирования как частота появления символа σ в S. Такое сканирование требует времени $O(n)$.

Алгоритм 12.4. AC-Coding (S, $\mathcal{P}$)

На вход: последовательность $S[1, n]$ и вероятности $\mathcal{P}(\Sigma)$.

Получить: подынтервал $[l, l + s)$ от [0, 1).

1: Вычисление накопленных вероятностей $f(\Sigma) = \sum_{c<\sigma}\mathcal{P}(c)$ для каждого символа алфавита $\sigma \in \Sigma$;
2: $s_0 = 1, l_0 = 0, i = 1$;
3: **while** $i \leqslant n$ **do**
4: $\quad s_i = s_{i-1}\mathcal{P}[S[i]]$;
5: $\quad l_i = l_{i-1} + s_{i-1}f(S[i])$;
6: $\quad i = i + 1$;
7: **end while**
8: **return** $\langle x \in [l_n, l_n + s_n], n\rangle$;

---

[1] Напомню, что накопленная вероятность символа $\sigma \in \Sigma$ вычисляется как $\sum_{c<\sigma}\mathcal{P}(c)$ и предоставляется статистической моделью, построенной в ходе фазы моделирования процесса сжатия. В динамической модели вероятности и накопленные вероятности меняются по мере сканирования входной последовательности.

Рассмотрим входную последовательность $S = abac$. Для оценки вероятностей $\mathcal{P}(a) = 1/2$ и $\mathcal{P}(b) = \mathcal{P}(c) = 1/4$ используется полустатическое моделирование. Результирующие накопленные вероятности: $f(a) = 0$, $f(b) = \mathcal{P}(a) = 1/2$ и $f(c) = \mathcal{P}(a) + \mathcal{P}(b) = 3/4$. Следуя псевдокоду `AC-Coding`($S$), получим $n = 4$, то есть внутренний цикл `while` повторяется четыре раза. На первой итерации ($i = 1$) рассматривается первый символ последовательности $S[1] = a$ и вычисляется новый интервал $[l_1, l_1 + s_1)$ с учетом значений $\mathcal{P}(a)$ и $f(a)$ из вероятностной модели:

$$s_1 = s_0\mathcal{P}(S[1]) = 1\mathcal{P}(a) = 1/2;$$
$$l_1 = l_0 + s_0 f(S[1]) = 0 + 1f(a) = 0.$$

На второй итерации рассматриваются второй символ, $S[2] = b$, накопленная вероятность $\mathcal{P}(b)$, а также $f(b)$ и определяется второй интервал $[l_2, l_2 + s_2)$:

$$s_2 = s_1\mathcal{P}(S[2]) = (1/2)\mathcal{P}(b) = 1/8;$$
$$l_2 = l_1 + s_1 f(S[2]) = 0 + (1/2)f(b) = 1/4.$$

Продолжая действовать таким же образом для третьего и четвертого символов, а именно $S[3] = a$ и $S[4] = c$, получим окончательный интервал:

$$\left[l_4, l_4 + s_4\right) = \left[\frac{19}{64}, \frac{19}{64} + \frac{1}{64}\right) = \left[\frac{19}{64}, \frac{20}{64}\right) = \left[\frac{19}{64}, \frac{5}{16}\right).$$

Выполнение этого алгоритма иллюстрирует рис. 12.8. Каждый шаг увеличивает связанный с текущим символом подынтервал. Последний шаг возвращает действительное число из последнего подынтервала, следовательно, оно находится *внутри всех* ранее сгенерированных интервалов. Как я покажу в следующем разделе, этого числа вместе с длиной n входной последовательности S достаточно для ее реконструкции. Фактически все входные последовательности фиксированной длиной n связаны с различными подынтервалами, которые не пересекаются друг с другом, но покрывают диапазон $[0, 1)$. В то же время последовательности разной длины могут быть вложенными, поэтому для уникальной реконструкции S необходимо знать n, и действительно, этот параметр возвращается алгоритмом `AC-Coding`.

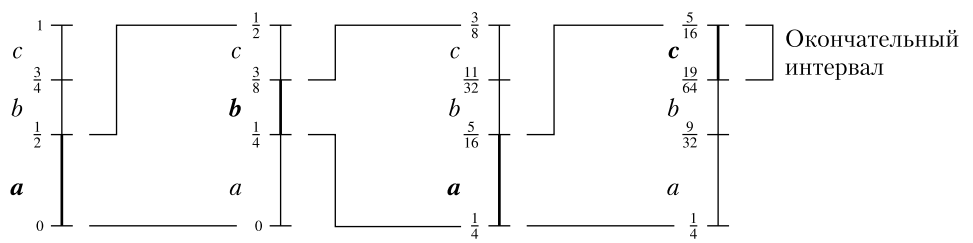


Рис. 12.8. Графическое представление идеи, лежащей в основе арифметического кодирования

12.2.3. Алгоритм декомпрессии

Входные данные состоят из потока битов, полученного путем сжатия, длины распаковываемой входной последовательности n и вероятностей $\mathcal{P}(\sigma)$ для всех символов алфавита $\sigma \in \Sigma$. На выходе нужно получить исходную последовательность $S[1, n]$, учитывая то, что арифметическое кодирование осуществляет *сжатие без потерь*.

Декодирование выполняется правильно, потому что для разбиения текущего интервала кодировщик и декодер задействуют одну и ту же статистическую модель ($\mathcal{P}$ и f) и оба начинают с интервала [0, 1). Разница в том, что кодировщик выбирает подынтервал, используя символы, тогда как декодер выбирает для увеличения тот же подынтервал, используя действительное число $0{,}b$.

В качестве примера возьмем пару $\langle 39/128, 4\rangle$. Предположим, что входное распределение составляет $\mathcal{P}(a) = 1/2$ и $\mathcal{P}(b) = \mathcal{P}(c) = 1/4$. Результирующие накопленные вероятности — это $f(a) = 0$, $f(b) = \mathcal{P}(a) = 1/2$ и $f(c) = \mathcal{P}(a) + \mathcal{P}(b) = 3/4$. Декодер выполняет псевдокод из алгоритма 12.5, начав с интервала [0, 1) и присваивания $b = 39/128$ и $n = 4$. В конце этого раздела вы увидите, что это пара, возвращаемая арифметическим кодированием для уже знакомого текста $S = abac$. Поэтому я предлагаю вернуться к рис. 12.8 и параллельно со сжатием произвести декомпрессию, просто чтобы убедиться в синхронизации кодировщика и декодера.

На первой итерации ($i = 1$) интервал [0, 1) делится на три части, по одной на символ алфавита. Новые интервалы располагаются в предопределенном порядке. В частности, в рассматриваемом примере первый снизу интервал связан с символом a, второй — с символом b, а последний — с символом c. Алгоритм 12.5 вычисляет размер каждого подынтервала пропорционально вероятности соответствующего символа, что дает [0, 1/2) для a, [1/2, 3/4) для b и [3/4, 1) для c. Алгоритм декодирования выводит символ a, поскольку входное значение 39/128 находится именно в интервале [0, 1/2). После этого он обновляет подынтервал как [0, 1/2). Разбиение на три подынтервала и выбор интервала, который содержит значение $0{,}b$ (шаги 4 и 5), реализуется повторением шагов 7 и 8 для всех символов Σ до обнаружения корректного варианта. То есть $0.b \in [l_1, l_1 + s_1)$, где $s_1 = s_0 \times \mathcal{P}(a) = 1 \times (1/2) = 1/2$ и $l_1 = l_0 + s_0 \times f(a) = 0 + 1 \times 0 = 0$.

На второй итерации ($i = 2$) текущий интервал [0, 1/2) делится на три, размеры которых пропорциональны вероятностям трех символов алфавита: снова 1/2 для a и по 1/4 для b и c. Это означает, что они равны $\left[0, \frac{1}{4}\right), \left[\frac{1}{4}, \frac{3}{8}\right), \left[\frac{3}{8}, \frac{1}{2}\right)$. В результате вторым возвращенным символом будет b, потому что $\frac{39}{128} \in \left[\frac{1}{4}, \frac{3}{8}\right)$. Для проверки корректности алгоритма 12.5 на шагах 7 и 8 вычисляется $s_2 = s_1 \times \mathcal{P}(b) = (1/2)(1/4) = 1/8$ и $l_2 = l_1 + s_1 \times f(b) = 0 + (1/2) \times (1/2) = 1/4$.

На третьей и четвертой итерациях выводятся символы *a* и *c* соответственно, то есть исходная последовательность реконструируется правильно. После четвертой итерации алгоритм 12.5 останавливается, потому что на вход декодеру была передана длина исходной последовательности $n = 4$.

Алгоритм 12.5. AC-Decoding(b, n, $\mathcal{P}$)

На вход: двоичная сжатая последовательность b, длина n последовательности S, вероятности $\mathcal{P}(\sigma)$.

Получить: исходную последовательность S.

1: Вычисляем накопленные вероятности $f(\sigma) = \sum_{c<\sigma}\mathcal{P}(c)$ для каждого символа алфавита $\sigma \in \Sigma$;
2: $s_0 = 1, l_0 = 0, i = 1$;
3: **while** $i \leqslant n$ **do**
4: Делим интервал $[l_{i-1}, l_{i-1} + s_{i-1})$ на подынтервалы, длина которых пропорциональна вероятностям символов алфавита, взятым в предустановленном порядке;
5: Определяем символ Σ, соответствующий подынтервалу, в котором находится значение $0.b$;
6: $S = S :: \sigma$;
7: $s_i = s_{i-1}\mathcal{P}(\sigma)$;
8: $l_i = l_{i-1} + s_{i-1}f(\sigma)$;
9: $i = i + 1$;
10: **end while**
11: **return** S;

12.2.4. Эффективность

На первый взгляд эта схема показывает хорошие результаты, так как связывает большие подынтервалы с частыми символами, ведь размер интервала s_i пропорционален $\mathcal{P}(S[i])$, а большой конечный интервал требует меньше битов для определения числа внутри него. Из шага 4 алгоритма 12.4 легко вывести формулу для размера s_n связанного с входной последовательностью S конечного интервала:

$$s_n = s_{n-1}\mathcal{P}\big(S[n]\big) = s_{n-2}\mathcal{P}\big(S[n-1]\big)\mathcal{P}\big(S[n]\big) = \ldots = \\ = s_0\mathcal{P}\big(S[1]\big)\cdots\mathcal{P}\big(S[n]\big) = 1\times\prod_{i=1}^{n}\mathcal{P}\big(S[i]\big). \quad (12.1)$$

Согласно этой формуле s_n зависит от формирующих последовательность S символов, но не от их порядка. Поэтому размер интервала, возвращаемого для S арифметическим кодированием, будет одинаковым при любом порядке. Но, так как он влияет

на количество битов в сжатом выводе, получается, что размер вывода *независим* от перестановки символов внутри последовательности S. Это не противоречит предыдущему утверждению, что арифметическое кодирование достигает производительности, близкой к энтропии последовательности S, когда формула для энтропии также не зависит от порядка символов S.

Остается проблема выбора числа в интервале $[l_n, l_n + s_n)$. Оно представляет собой двоично-рациональную дробь $v/2^k$ и может быть закодировано в нескольких битах (маленьким значением k). Следующая лемма крайне важна для установления эффективности и корректности арифметического кодирования.

Лемма 12.3. *Усечение действительного числа x, представленного в виде двоичной последовательности $0.b_1b_2...$, до первых d бит дает действительное число* $\text{trunc}_d(x) \in [x - 2^{-d}, x]$.

Доказательство. Действительное число x может отличаться от усеченной формы битами, которые следуют за позицией d. В ходе операции $\text{trunc}_d(x)$ эти биты сбрасываются в 0. Поэтому имеем:

$$x - \text{trunc}_d(x) = \sum_{i=1}^{\infty} b_{d+1} 2^{-(d+i)} \leqslant \sum_{i=1}^{\infty} 1 \cdot 2^{-(d+i)} = 2^{-d} \sum_{i=1}^{\infty} \frac{1}{2^i} = 2^{-d}.$$

В то же время $\text{trunc}_d(x) \leqslant x$, ведь биты 1 в двоичном представлении x можно превратить в биты 0 в двоичном представлении $\text{trunc}_d(x)$. ■

Следствие 12.1. *Усечение $l + s/2$ до первых* $\left\lceil \log_2 \frac{2}{s} \right\rceil$ *бит попадает в интервал* $[l, l + s)$.

Доказательство. Достаточно предположить, что в лемме 12.3 $d = \left\lceil \log_2 \frac{2}{s} \right\rceil$, и обратить внимание на неравенство $2^{-d} \leqslant s/2$. ■

На этом этапе последний, восьмой шаг в алгоритме `AC-Coding`(S) можно заменить вызовом алгоритма `CONVERTER` для действительного числа $l_n + s_n/2$, попросив сгенерировать только $\left\lceil \log_2 \frac{2}{s_n} \right\rceil$ бит. Алгоритм `CONVERTER` позволяет генерировать эти биты пошагово.

Для наглядности продолжим использовать предыдущий пример, взяв конечный интервал $[l_4, l_4 + s_4) = [19/64, 20/64)$, вычисленный на этапе сжатия. Выходным значением будет:

$$l_4 + \frac{s_4}{2} = \frac{19}{64} + \frac{1}{64} \cdot \frac{1}{2} = \frac{39}{128},$$

усеченное до первых $\left\lceil \log_2 \frac{2}{s_4} \right\rceil = \log_2 128 = 7$ бит. Связанный с этим значением результирующий поток битов получается выполнением алгоритма `CONVERTER` за семь шагов.

Вот результаты всех итераций:

$$\frac{39}{128}\cdot 2=\frac{78}{128}<1\rightarrow \text{вывод}=0;$$
$$\frac{78}{128}\cdot 2=\frac{156}{128}\geqslant 1\rightarrow \text{вывод}=1;$$
$$\left(\frac{156}{128}-1\right)\cdot 2=\frac{56}{128}<1\rightarrow \text{вывод}=0;$$
$$\frac{56}{128}\cdot 2=\frac{112}{128}<1\rightarrow \text{вывод}=0;$$
$$\frac{112}{128}\cdot 2=\frac{224}{128}\geqslant 1\rightarrow \text{вывод}=1;$$
$$\left(\frac{224}{128}-1\right)\cdot 2=\frac{192}{128}\geqslant 1\rightarrow \text{вывод}=1;$$
$$\left(\frac{192}{128}-1\right)\cdot 2=1\geqslant 1\rightarrow \text{вывод}=1.$$

В конце кодировщик возвращает пару $\langle 0100111_2, 4\rangle$. Напомню, что для декомпрессии декодер должен получить не только эту пару, но и алфавит $\Sigma = \{a, b, c\}$, и вероятности символов $\mathcal{P}(a) = 1/2$, $\mathcal{P}(b) = \mathcal{P}(c) = 1/4$.

Теперь все готово для того, чтобы доказать основную теорему этого раздела, которая связывает возможную при арифметическом кодировании степень сжатия с эмпирической энтропией входной строки S. Энтропию называют эмпирической, когда вероятности символов алфавита оцениваются по частоте их появления в S.

Теорема 12.3. *При арифметическом кодировании последовательности S, состоящей из n символов, генерируется не более $2 + n\mathcal{H}$ бит, где $\mathcal{H}$ — эмпирическая энтропия S.*

Доказательство. Из следствия 12.1 и уравнения 12.1 известно, что количество выходных битов составляет:

$$\left\lceil \log_2 \frac{2}{s_n} \right\rceil < 2 - \log_2 s_n = 2 - \log_2 \left(\prod_{i=1}^{n} \mathcal{P}\left(S[i]\right) \right) = 2 - \sum_{i=1}^{n} \log_2 \mathcal{P}\left(S[i]\right).$$

Суммирование можно провести не по позициям i в S, а по символам Σ, группируя их одинаковые вхождения:

$$2 - \sum_{\sigma\in\Sigma} n_\sigma \log_2 \mathcal{P}(\sigma) = 2 - n\left(\sum_{\sigma\in\Sigma} \frac{n_\sigma}{n} \log_2 \mathcal{P}(\sigma) \right),$$

где $n\Sigma$ — число вхождений символа Σ в S. При этом, когда арифметическое кодирование использует для оценки вероятностей символов полустатическую модель,

то есть $\mathcal{P}(\sigma) = n_\sigma/n$, эту формулу можно переписать *в точности* как $2 + n\mathcal{H}$, где $\mathcal{H}$ — эмпирическая энтропия S. Что и требовалось доказать. В статической модели вероятности фиксируются заранее, и в соответствии с ними генерируется последовательность S. Но по закону больших чисел при достаточно больших значениях n отношение n_σ/n сходится к $\mathcal{P}(\sigma)$, что дает указанную в теореме границу для энтропии источника S. ■

Из этого результата следует вот что.

- На всю входную последовательность S расходуется всего два бита, то есть по $2/n$ бита на символ. По мере роста входной последовательности это число становится пренебрежимо малым.
- Размер выходных данных зависит от набора символов в S и от их повторяемости, но не от порядка их следования.

В разделе 12.1 было показано, что для сжатия последовательности из n символов кодирование Хаффмана требует $n + n\mathcal{H}$ бит. В этом отношении арифметическое кодирование намного лучше, ведь аддитивный линейный член n превращается в константу 2. Другое преимущество заключается в том, что сжатие выполняется в реальном времени и легко поддерживает использование динамического моделирования. К сожалению, эффективность арифметического кодирования достигается с помощью арифметических операций с бесконечной точностью, требующих больших временных и пространственных затрат. Существует несколько предложений об использовании вычислений с *конечной* точностью, которые снижают коэффициент сжатия до $n\mathcal{H}_0 + (2/100)n$ (см., например, [3], [9]). Но даже в этом случае арифметическое кодирование дает лучшее сжатие, чем алгоритм Хаффмана, — 2/100 в отличие от потери одного бита. В следующем разделе описывается практическая реализация арифметического кодирования, предложенная Виттеном, Нилом и Клирли [9], иногда называемая *интервальным кодированием*. Математически оно эквивалентно арифметическому кодированию, но работает с подынтервалами, имеющими *целочисленные границы*.

Позднее было предложено еще одно решение, на практике использующее арифметику конечной точности, очень интересное сжатие и временные характеристики, — *асимметричные системы счисления* (asymmetric numeral systems, ANS) [2]. Как и арифметическое кодирование, ANS преобразует последовательность входных символов в число, которое инкапсулирует описание последовательности, но генерирует при этом *целочисленное* значение, которое *растет* по мере считывания новых символов входной последовательности. Таким образом, кодирование ANS добавляет к этому целому числу малозначимые биты, причем доказано, что это число отклоняется от нуля в соответствии с обратными вероятностями символов. Поэтому среднее число генерируемых битов продолжает расти как эмпирическая энтропия входной последовательности. (Более подробное объяснение и сравнение ANS с кодированием Хаффмана и арифметическим кодированием можно найти в [5].)

В целом (канонический) алгоритм Хаффмана имеет ряд преимуществ перед арифметическим кодированием и кодированием ANS: он работает быстрее и позволяет распаковать любую часть сжатого файла при условии, что известно ее начальное кодовое слово. Эти свойства оправдывают частое использование канонического кодирования Хаффмана в контексте текстовых коллекций, где важна эффективная распаковка частей данных, а распределение символов не очень асимметрично. Для таких вариантов применения интересна версия канонического алгоритма Хаффмана, использующая большой алфавит, который состоит из слов или токенов, называемых Huffword [8].

12.2.5. Интервальное кодирование∞

Представление действительных чисел в вычислениях с конечной точностью для интервального кодирования зависит от следующих шагов.

1. Для каждого символа $\sigma \in \Sigma$ создается целочисленный счетчик его появлений во входной последовательности $c[\sigma]$, а также кумулятивный счетчик $C[\sigma]$, суммирующий счетчики всех символов, предшествующих Σ в Σ, то есть $C[\sigma] = \sum_{\alpha < \sigma} c[\alpha]$. Это позволяет приблизительно оценить вероятность $\mathcal{P}(\sigma)$ и кумулятивную вероятность $f(\sigma)$ как:

$$\mathcal{P}(\sigma) = \frac{c[\sigma]}{C\left[|\Sigma|+1\right]}; \quad f(\sigma) = \frac{C[\sigma]}{C\left[|\Sigma|+1\right]}.$$

2. Интервал [0, 1) отображается в целочисленный интервал [0, M), где $M = 2^w$ зависит от длины слова памяти w в битах.
3. На итерации i процесса сжатия или декомпрессии текущий подынтервал (ранее $[l_i, l_i + s_i)$) выбирается так, чтобы конечные точки $[L_i, H_i)$ были целочисленными. Это делается по следующим формулам:

$$L_i = L_{i-1} + \left\lfloor f\left(S[i]\right)\left(H_{i-1} - L_{i-1}\right) \right\rfloor;$$
$$H_i = L_i + \left\lfloor \mathcal{P}\left(S[i]\right)\left(H_{i-1} - L_{i-1}\right) \right\rfloor.$$

Такие приближения приводят к потере сжатия, эмпирически оцененной как 10^{-4} бит на входной символ. Чтобы понять, как это работает, подробно разберем процессы сжатия и декомпрессии.

Сжатие. Гарантировать в каждом интервале $[L_i, H_i)$ непустые подынтервалы можно при условии, что начальная точка каждого следующего подынтервала больше начальной точки предыдущего. Из определений начальных точек получаем:

$$L_{i+1} = L_i + \left\lfloor f\left(S[i+1]\right)\left(H_i - L_i\right) \right\rfloor = L_i + \left\lfloor \frac{C\left[S[i+1]\right]}{C\left[|\Sigma|+1\right]}\left(H_i - L_i\right) \right\rfloor.$$

Поскольку значения кумулятивного счетчика $C[i]$ строго возрастают, условие $\frac{H_i - L_i}{C\left[|\Sigma|+1\right]} \geqslant 1$ гарантирует возрастание начальных точек подынтервалов. С учетом того, что результат суммирования должен быть меньше M, достаточно, чтобы соблюдалось неравенство:

$$C\left[|\Sigma|+1\right] \leqslant \frac{M}{4} + 2 \leq H_i - L_i. \tag{12.2}$$

Это означает, что *адаптивное* интервальное кодирование должно сбрасывать счетчики через каждые $M/4 + 2$ входных символа или масштабировать их, например деля пополам, через каждые $M/8 + 1$ входных символа.

Масштабирование. Чтобы гарантировать соблюдение неравенства (12.2), на каждой итерации нужно проверять следующие *правила расширения* (цикл продолжается до тех пор, пока все они не перестанут соблюдаться).

1. $[L_i, H_i) \subseteq [0, M/2) \rightarrow$ на выходе 0, а новый интервал:

$$[L_{i+1}, H_{i+1}) = [2L_i, 2(H_i - 1) + 2).$$

2. $[L_i, H_i) \subseteq [M/2, M) \rightarrow$ на выходе 1, а новый интервал:

$$\left[L_{i+1}, H_{i+1}\right) = \left[2\left(L_i - \frac{M}{2}\right), 2\left(H_i - 1 - \frac{M}{2}\right) + 2\right).$$

3. Если $M/4 \leqslant L_i < M/2 < H_i \leqslant 3M/4$, то это ситуация *потери значимости* и на выход невозможно подать ни одного бита.

При потере значимости невозможно вывести ни одного бита до попадания интервала в одну из половин диапазона $[0, M)$, что соответствует случаю 1 или 2. Если продолжить работу с интервалом $[M/4, 3M/4)$, просто изменив условия 1 и 2, интервал может стать меньше чем $M/8$ и проблема возникнет снова. Введем параметр m, куда будет записываться, сколько раз возникла ситуация потери значимости, в результате которой текущий интервал оказывается в пределах $\left[\frac{M}{2} - \frac{M}{2^{m+1}}, \frac{M}{2} + \frac{M}{2^{m+1}}\right)$. Обратите внимание: когда в итоге интервал перестает включать $M/2$, выводится 01^m, если результат находится в первой половине, или 10^m, если во второй. После этого интервал можно расширить вокруг его средней точки и подсчитать количество этих расширений.

- Математически, если $M/4 \leqslant L_i < M/2 < H_i \leqslant 3M/4$, нужно увеличить число потерь значимости m и рассмотреть новый интервал:

$$\left[L_{i+1}, H_{i+1}\right) = \left[2\left(L_i - \frac{M}{4}\right), 2\left(H_i - 1 - \frac{M}{4}\right) + 2\right).$$

- Если используются правила расширения 1 или 2, то после вывода бита также выводятся m копий дополнения этого бита, а m сбрасывается в 0.

Конец входной последовательности. Когда входная последовательность заканчивается из-за расширений интервалов, текущий интервал удовлетворяет по крайней мере одному из следующих неравенств:

$$L_n < \frac{M}{4} < \frac{M}{2} < H_n;\ L_n < \frac{M}{2} < \frac{3M}{4} < H_n. \tag{12.3}$$

При $m > 0$ интервальное кодирование завершает выходной поток битов следующим образом.

- Если выполняется первое неравенство, можно сгенерировать 01^{m+1} (при $m = 0$ это означает кодирование $M/4$).
- Если выполняется второе неравенство, можно сгенерировать 10^{m+1} (при $m = 0$ это означает кодирование $3M/4$).

Декомпрессия. Декодер должен воспроизводить вычисления, выполнявшиеся в процессе сжатия. Он поддерживает сдвиговый регистр v длиной $\lceil \log_2 M \rceil$ бит, который играет роль x (в классическом арифметическом кодировании) и, таким образом, используется для поиска следующего подынтервала в результатах разбиения текущего. При расширении интервала v меняется и функция `next_bit` (которая, как предполагается, извлекает бит 0 при завершении входного потока) загружает из сжатого потока очередной бит. Как и на этапе сжатия, на каждой итерации проверяются следующие правила расширения (цикл продолжается до тех пор, пока все они не перестанут соблюдаться).

1. $[L_i, H_i) \subseteq [0, M/2) \rightarrow$ рассматриваем новый интервал:

$$[L_{i+1}, H_{i+1}) = [2L_i, 2(H_i - 1) + 2);$$
$$v = 2v + \texttt{next\_bit}.$$

2. $[L_i, H_i) \subseteq [M/2, M) \rightarrow$ рассматриваем новый интервал:

$$\left[L_{i+1}, H_{i+1}\right) = \left[2\left(L_i - \frac{M}{2}\right), 2\left(H_i - 1 - \frac{M}{2}\right) + 2\right);$$
$$v = 2\left(v - \frac{M}{2}\right) + \texttt{next\_bit}.$$

3. При $M/4 \leqslant L_i < M/2 < H_i \leqslant 3M/4$ рассматриваем новый интервал:

$$\left[L_{i+1}, H_{i+1}\right) = \left[2\left(L_i - \frac{M}{4}\right), 2\left(H_i - 1 - \frac{M}{4}\right) + 2\right);$$
$$v = 2\left(v - \frac{M}{4}\right) + \texttt{next\_bit}.$$

Чтобы лучше понять процесс интервального кодирования, возьмем в качестве примера уже знакомую по предыдущим разделам последовательность $S = abac$ длиной $n = 4$.

Напомню, что она состоит из букв упорядоченного алфавита $\Sigma = \{a, b, c\}$ с вероятностями $\mathcal{P}(a) = 1/2$, $\mathcal{P}(b) = \mathcal{P}(c) = 1/4$ и накопленными вероятностями $f(a) = 0$, $f(b) = 1/2$ и $f(c) = 3/4$. Перепишем вероятности, используя аппроксимации из начала этого раздела. В результате получим $C[|\Sigma| + 1] = 4$ и начальный интервал $[L_0, H_0) = [0, M)$, где M выбрано таким образом, чтобы удовлетворять неравенству (12.2):

$$C\left[|\Sigma|+1\right] \leqslant \frac{M}{4}+2 \Leftrightarrow 4 \leqslant \frac{M}{4}+2.$$

Возьмем $M = 16$, соответственно, $M/4 = 4$, $M/2 = 8$, а $3M/4 = 12$ (это значение M не имеет отношения к реальной длине машинного слова, просто оно удобно для данного примера). На данном этапе начальный интервал будет $[L_0, H_0) = [0, 16)$. Все готово для сжатия первого символа $S[1] = a$ с помощью формул для конечных точек, приведенных в начале этого раздела:

$$L_1 = L_0 + \left\lfloor f(a)\left(H_0 - L_0\right)\right\rfloor = 0 + \left\lfloor 0 \cdot 16 \right\rfloor = 0;$$

$$H_1 = L_1 + \left\lfloor \mathcal{P}(a)\left(H_0 - L_0\right)\right\rfloor = 0 + \left\lfloor \frac{2}{4} \times 16 \right\rfloor = 8.$$

Новый интервал $[L_1, H_1) = [0, 8)$ удовлетворяет первому правилу расширения $[L_0, H_0) \subseteq [0, M/2)$, то есть процедура интервального кодирования сгенерирует бит 1 и расширит текущий интервал как $[L_1, H_1) = [2L_1, 2(H_1 - 1) + 2) = [0, 16)$.

На второй итерации будет рассматриваться следующий символ входной последовательности $S[2] = b$. Конечными точками нового интервала будут:

$$L_2 = L_1 + \left\lfloor f(b)\left(H_1 - L_1\right)\right\rfloor = 8;$$

$$H_2 = L_2 + \left\lfloor \mathcal{P}(b)\left(H_1 - L_1\right)\right\rfloor = 12.$$

Этот интервал удовлетворяет второму правилу расширения $[L_2, H_2) \subseteq [M/2, M)$, процедура интервального кодирования сгенерирует бит 1 и расширит текущий интервал как:

$$\left[L_2, H_2\right) = \left[2\left(L_2 - \frac{M}{2}\right), 2\left(H_2 - 1 - \frac{M}{2}\right) + 2\right) = \left[0, 8\right).$$

Этот интервал удовлетворяет первому правилу расширения, поэтому интервальное кодирование будет применено до считывания следующего символа входной последовательности. Генерируется 0, и создается новый расширенный интервал $[L_2, H_2) = [2L_1, 2(H_2 - 1) + 2) = [0, 16)$.

Третий символ входной последовательности $S[3] = a$ совпадает с первым и кодируется с тем же самым интервалом. Поэтому сразу можно сказать, что на выходе появится 0, а новый интервал будет равен $[L_3, H_3) = [0, 16)$. Для последнего, четвертого символа входной последовательности $S[4] = c$ интервал вычисляется как:

$$\left[L_4, H_4\right) = \left[L_3 + \left\lfloor f(c)\left(H_3 - L_3\right)\right\rfloor, L_4 + \left\lfloor \mathcal{P}(c)\left(H_3 - L_3\right)\right\rfloor\right) = \left[12, 16\right).$$

Этот интервал находится после $M/2$, поэтому процедура интервального кодирования генерирует 1 и применяет второе правило расширения:

$$[L_4, H_4) = \left[2\left(L_4 - \frac{M}{2}\right), 2\left(H_4 - 1 - \frac{M}{2}\right) + 2\right) = [8, 16).$$

В соответствии со вторым правилом этот интервал следует расширить, поэтому процедура интервального кодирования генерирует 1 и получает окончательный интервал:

$$[L_4, H_4) = \left[2\left(L_4 - \frac{M}{2}\right), 2\left(H_4 - 1 - \frac{M}{2}\right) + 2\right) = [0, 16).$$

Вы можете самостоятельно убедиться, что этот окончательный интервал удовлетворяет неравенству (12.3), а сжатая битовая последовательность `010011` на один бит короче, чем последовательность, сгенерированная при классическом арифметическом кодировании. Эта битовая последовательность кодирует двоично-рациональное число 19/64, попадающее внутрь определенного арифметическим кодированием диапазона [19/64, 5/16), как показано на рис. 12.8.

При декодировании на первой итерации сдвиговый регистр v длиной $\lceil \log_2 M \rceil = \lceil \log_2 16 \rceil = 4$ инициализируется первыми $\lceil \log_2 16 \rceil = 4$ битами сжатой последовательности, то есть $v = 0100_2 = 4_{10}$[1]. Начальный интервал $[L_0, H_0) = [0, 16)$ подразделяется на три подынтервала, по одному на каждый символ алфавита. Как было посчитано ранее, это [0, 8), [8, 12) и [12, 16). На выход подается символ a, потому что $v = 4 \in [0, 8)$. Так как в этот момент $[L_1, H_1) = [0, 8) \subseteq [0, M/2)$, интервальное кодирование применяет к процедуре декомпрессии первое правило расширения, получая:

$$[L_1, H_1) = [L_1, 2(H_1 - 1) + 2) = [0, 16);$$
$$v = 2v + \texttt{next\_bit} = \texttt{shift}_{sx}(0100_2) + 1_2 = 1000_2 + 1_2 = 1001_2 = 9_{10}.$$

На второй итерации интервал [0, 16) подразделяется на те же диапазоны, что и раньше, а на выход подается символ b, потому что $v = 9 \in [8, 12)$. Этот интервал удовлетворяет второму правилу расширения, которое создает новый интервал:

$$[L_2, H_2) = [2(L_2 - M/2), 2(H_2 - 1 - M/2) + 2) = [0, 8);$$
$$v = 2(v - M/2) + \texttt{next\_bit} = \texttt{shift}_{sx}(1001_2 - 1000_2) + 1_2 = 0011_2 = 3_{10}.$$

Применяем первое правило расширения, так как текущий интервал [0, 8) удовлетворяет именно ему (функция `next_bit` возвращает 0, если сжатая последовательность больше не содержит битов):

$$[L_2, H_2) = [2L_2, 2(H_2 - 1) + 2) = [0, 16);$$
$$v = 2v + \texttt{next\_bit} = \texttt{shift}_{sx}(0011_2) + 0_2 = 0110_2 = 6_{10}.$$

[1] Нотация x_2 или x_{10} означает, что x записан в двоичной или десятичной системе счисления соответственно.

Разбиение этого интервала происходит так же, как и на первой итерации. Интервальное кодирование выводит символ *a*, потому что $v = 6 \in [0, 8)$. Повторяя вычисления, которые проделывались на первой итерации, получаем новый интервал $[L_3, H_3) = [0, 16)$ и:

$$v = 2v + \texttt{next\_bit} = \texttt{shift}_{\text{sx}}(0110_2) + 0_2 = 1100_2 = 12_{10}.$$

Последним на выход подается символ *c*, так как $v = 12 \in [12, 16)$. Входная последовательность корректно реконструирована. Алгоритм может остановиться, потому что он сгенерировал четыре символа, а именно 4 было предоставлено в качестве входных данных декодеру как длина последовательности *S*.

12.3. Предсказания по частичному совпадению∞

Для повышения эффективности сжатия необходимо точнее моделировать вероятности символов. Как правило, они оцениваются не только по отдельности как независимые друг от друга, но и с точки зрения *условной вероятности* их появления в последовательности *S* с учетом нескольких предыдущих символов или так называемого *контекста*. Этот раздел посвящен *адаптивному* методу построения контекстной модели, который можно успешно комбинировать с арифметическим кодированием, поскольку он генерирует асимметричные вероятности и, следовательно, высокое сжатие. Этот метод называется *предсказанием по частичному совпадению* (prediction by partial matching, PPM) и позволяет перейти в алгоритмах кодирования от энтропии порядка 0 к энтропии порядка *k*. Недавнее появление таких сложных моделей распределения вероятностей, как *глубокие нейронные сети*, открыло новые подходы к моделированию, которые в сочетании со статистическими кодировщиками позволяют достичь еще более высокой производительности сжатия. Но их временная и пространственная эффективность пока недостаточно изучена.

При реализации PPM возникают две проблемы. Во-первых, требуются структуры данных для хранения всех условных вероятностей, обновляемых в процессе сканирования входной последовательности, причем они должны быть эффективными по времени и пространству. Во-вторых, поначалу оценка контекстно ориентированных вероятностей дает плохой результат, и необходимы корректировки, чтобы быстро получить хорошую статистику для всех входных символов. В этом разделе будет рассмотрена вторая проблема, а о попытках решения первой можно почитать, например, в [4] и [8]. Кроме того, я расскажу о результатах, недавно полученных в области кодирования со сжатием древовидных структур данных, которые могут использоваться для решения первой проблемы. Это еще одна ключевая проблема эффективности при росте размера алфавита и длины контекста (см. также главу 15).

Для предсказания следующего символа в PPM используется набор конечных контекстов длиной $\ell \leqslant K$ (контексты порядка ℓ). Максимальная длина контекста *K* влияет на временную и пространственную эффективность структур данных, используемых для индексации этих контекстов. На каждой итерации *i* PPM состоит из двух фаз: кодирования текущего символа $\sigma = S[i]$ в соответствии с оценками вероятности, предоставленными для него набором контекстов, и обновления счетчиков для σ

и всех его предыдущих контекстов $\alpha = S[i-\ell, i-1]$ длиной меньше K. Такой счетчик отслеживает, сколько раз строка α встречается в уже обработанном префиксе S.

Выбрать для кодирования $S[i]$ *наилучший* контекст непросто. Начинается этот процесс с самого длинного возможного контекста K, который предшествует $S[i]$, то есть с $S[i-K, i-1]$. Если *статистической информации* недостаточно для прогнозирования $S[i]$ после $S[i-K, i-1]$, алгоритм переключается на все более короткие контексты порядка l, пока не найдет подходящий — скажем, $S[i-\ell, i-1]$ с $\ell < K$, позволяющий предсказывать $S[i]$ с достаточной статистической значимостью. Для такой проверки используется статистика (счетчики), которую PPM собрал для всех символов, встречающихся в уже обработанном префиксе последовательности S после всех контекстов длиной меньше K. Это позволяет оценить вероятность того, что $S[i]$ следует за $S[i-\ell, i-1]$. Нужно только использовать соответствующий счетчик, если он больше 0. Контекст нулевого порядка соответствует вероятностям, оцененным только по счетчикам частот отдельных символов, точно так же, как в классическом арифметическом кодировании. В противном случае (при нулевом значении счетчика для $S[i]$, следующего за $S[i-\ell, i-1]$) PPM переходит к более короткому контексту порядка l. В самом начале некоторые контексты могут отсутствовать. Например, при $i \leqslant K$ максимальная длина контекста, доступного кодировщику и декодеру, составляет $(i-1) < K$. Чтобы избежать случаев, когда для всех $0 \leqslant l \leqslant K$ отсутствуют статистические данные по $S[i]$ после $S[i-\ell, i-1]$, поддерживается *специальный контекст* порядка (-1), соответствующий модели, в которой все символы имеют одинаковую вероятность. Он гарантирует, что при любом масштабировании контекста символ $S[i]$ всегда будет иметь ненулевую вероятность.

Ключевой вопрос состоит в том, как закодировать длину ℓ модели, принятой для сжатия $S[i]$, чтобы этим контекстом мог пользоваться декодер. Если задействовать целочисленное кодирование (см. главу 11), то на символ потребуется целое число битов, что сведет на нет все усилия по выбору хорошей модели. Для решения этого вопроса в PPM ввели знак перехода (`esc`), генерируемый при каждом переходе к более короткому контексту. *Процесс перехода* продолжается до тех пор, пока не будет достигнута модель, в которой символ не является новым, то есть его счетчик больше 0. В конечном счете это будет специальный контекст порядка (-1). При такой стратегии с символом связывается вероятность, вычисленная в самом длинном контексте, где он ранее встречался. Она куда точнее вероятностей на базе индивидуальных подсчетов числа появления символа в алфавите Σ, принятых в арифметическом кодировании и алгоритме Хаффмана.

Рассмотрим работу PPM на примере входной последовательности S = *abracadabra*. Определим размер самого длинного контекста, используемого при вычислении *условных вероятностей* для всех символов алфавита, как $K = 2$. Как уже упоминалось, в начале работы алгоритма доступна только модель порядка (-1). Поэтому при считывании первого символа a не нужно генерировать символ перехода, а контекст порядка (-1) используется для назначения $S[1] = a$ равномерной вероятности $1/|\Sigma|$ (обычно символы кодируются байтами, соответственно, $|\Sigma| = 256$). Одновременно обновляются частоты, и вероятность становится равной $\mathcal{P}(a) = 1/2$ и $\mathcal{P}(\texttt{esc}) = 1/2$. В этом примере предполагается, что счетчик символа перехода равен общему

количеству различных символов в модели. Другие стратегии определения его вероятности я рассмотрю чуть позже.

После считывания символа $S[2] = b$ и проверяется модель нулевого порядка, потому что, как объяснялось ранее, она самая длинная из доступных. Так как символ b прежде не считывался, передается символ перехода. Для сжатия b используется модель порядка (−1), после чего обновляются модели первого и нулевого порядка. В модели нулевого порядка получим $\mathcal{P}(a) = 1/4$, $\mathcal{P}(b) = 1/4$, $\mathcal{P}(\texttt{esc}) = 2/4 = 1/2$ (считано два различных символа). В модели первого порядка вероятности составят $\mathcal{P}(b|a) = 1/2$ и $\mathcal{P}(\texttt{esc}|a) = 1/2$ (считан только один отдельный символ).

Кодирование первых пяти символов пропустим и рассмотрим кодирование $S[6] = d$. Это первое появление символа d в последовательности S, поэтому для перехода от модели второго порядка к модели порядка (−1) будут сгенерированы три символа перехода. Статистику, вычисленную PPM после обработки всей последовательности S, демонстрирует рис. 12.9. Затем эти оценки вероятности могут использоваться для сжатия последовательности символов алфавита $\Sigma \cup \{\texttt{esc}\}$ алгоритмом арифметического кодирования или любым другим статистическим алгоритмом кодирования. Вот почему PPM часто рассматривается не как компрессор, а как метод моделирования контекста.

Порядок $k = 2$					Порядок $k = 1$					Порядок $k = 0$				Порядок $k = 1$			
Предсказания			c	$\mathcal{P}$	Предсказания			c	$\mathcal{P}$	Предсказания			c	$\mathcal{P}$	Предсказания			c	$\mathcal{P}$
ab	→	r	2	$\frac{2}{3}$	a	→	b	2	$\frac{2}{7}$	→	a		5	$\frac{5}{16}$	→	$\forall\sigma[i]$		1	$\frac{1}{\lvert\Sigma\rvert}$
	→	esc	1	$\frac{1}{3}$		→	c	1	$\frac{1}{7}$	→	b		2	$\frac{2}{16}$					
						→	d	1	$\frac{1}{7}$	→	c		1	$\frac{1}{16}$					
ac	→	a	1	$\frac{1}{2}$		→	esc	3	$\frac{2}{7}$	→	d		1	$\frac{1}{16}$					
	→	esc	1	$\frac{1}{2}$						→	r		2	$\frac{2}{16}$					
					b	→	r	2	$\frac{2}{3}$	→	esc		5	$\frac{5}{16}$					
ad	→	a	1	$\frac{1}{2}$		→	esc	1	$\frac{1}{3}$										
	→	esc	1	$\frac{1}{2}$															
					c	→	a	1	$\frac{1}{2}$										
br	→	a	2	$\frac{2}{3}$		→	esc	1	$\frac{1}{2}$										
	→	esc	1	$\frac{1}{3}$															
					d	→	a	1	$\frac{1}{2}$										
ca	→	d	1	$\frac{1}{2}$		→	esc	1	$\frac{1}{2}$										
	→	esc	1	$\frac{1}{2}$															
					r	→	a	2	$\frac{2}{3}$										
da	→	b	1	$\frac{1}{2}$		→	esc	1	$\frac{1}{3}$										
	→	esc	1	$\frac{1}{2}$															
ra	→	c	1	$\frac{1}{2}$															
	→	esc	1	$\frac{1}{2}$															

Рис. 12.9. Метод C модели PPM после обработки строки S = *abracadabra*

12.3.1. Оценка вероятностей символов

Информацию о частотах символов в моделях порядка l можно применять для улучшения скорости сжатия при масштабировании через все более короткие контексты. Предположим, что входная последовательность S из предыдущего примера обработана целиком и для кодирования предлагается следующий символ c, при этом $K = 2$. Текущий контекст второго порядка — ra. На рис. 12.9 это запись $ra \to c$, соответственно, c кодируется с вероятностью $1/2$, используя один бит.

Теперь предположим, что за последовательностью $S = abracadabra$ вместо c идет символ d. Записи $ra \to d$ на рис. 12.9 не существует, поэтому с вероятностью $1/2$ генерируется символ `esc` и происходит переход к контексту первого порядка a. Запись $a \to d$ существует, то есть d может быть закодировано с вероятностью $1/7$. Улучшить оценку вероятности $\mathcal{P}(d|a)$ позволяет так называемый *принцип исключения*. После отбрасывания модели второго порядка кодировщик и декодер делают вывод, что текущий символ не может быть ни одним из перечисленных на рис. 12.9 после контекста ra. Более короткая модель первого порядка знает, что текущим символом не может быть c (из-за записи $ra \to c$), что позволяет исключить запись $a \to c$ и на единицу уменьшить частоту контекста a (частоту появления сочетания ac). В результате символ d кодируется с вероятностью $1/6$.

Предположим, что после последовательности *abracadabra* появляется новый символ e. В этом случае генерируется последовательность символов `esc`, заставляя декодер переключиться на контекст порядка (-1). Без *исключения* кодирование нового символа происходит с вероятностью $1/|\Sigma|$. Вместо этого PPM может исключить из контекста порядка (-1) все символы, встречавшиеся после предыдущих более длинных контекстов, таких как ra, a и пустой контекст. Все эти символы, а именно $\{a, b, c, d, r\}$, появлялись раньше, поэтому новый символ получает вероятность $1/|\Sigma| - 5$. Такая техника требует немного больше времени, но дает разумную отдачу с точки зрения дополнительного сжатия, ведь все неисключенные символы имеют повышенную вероятность.

Последний комментарий касается максимальной длины контекстов K в PPM и кодирования символа `esc`. Может показаться, что с ростом K производительность PPM должна улучшаться, но так происходит не всегда. Входная последовательность конечна, и при более длинных контекстах появляется большая вероятность сгенерировать столько символов перехода, что будет достигнута длина контекста, для которой доступны ненулевые предсказания. Поэтому K не может быть очень большим (обычно выбирается $K \leqslant 5$), и оценке вероятности символа `esc` необходимо проявлять большую осторожность, поскольку это влияет на общий размер сжатого пространства, достигаемого в PPM.

В заключение я хочу прокомментировать некоторые известные методы оценки этой вероятности, например, из [1], [4], [7]. Пусть α обозначает контекст, σ — произвольный символ, $c(\sigma)$ — количество появлений σ после контекста α, $n\alpha$ — количество появлений контекста α, q — общее количество различных символов, прочитанных в контексте α.

Метод A. Вероятность появления символа Σ, то есть $c(\Sigma) > 0$, в контексте α оценивается как $P(\sigma|\alpha) = \frac{c(\sigma)}{1+n_\alpha}$. Вероятность появления нового символа в контексте α оценивается как:

$$P(\mathtt{esc}|\alpha) = 1 - \sum_{\sigma\in\Sigma,\ c(\sigma)>0} \mathcal{P}(\sigma|\alpha) = 1 - \sum_{\sigma\in\Sigma,\ c(\sigma)>0} \frac{c(\sigma)}{1+n_\alpha} = 1 - \frac{n_\alpha}{1+n_\alpha} = \frac{1}{1+n_\alpha},$$

где равенство $\sum_{\sigma\in\Sigma} c(\sigma) = n_\alpha$ используется исходя из определения этих параметров.

Метод B. Символ классифицируется как новый, если он не встретился два раза. Такой подход связан с тем, что однократное появление символа может быть аномалией. Вероятность появления символа в контексте в этом случае оценивается по формуле $\mathcal{P}(\sigma|\alpha) = \frac{c(\sigma)-1}{n_\alpha}$. Количество различных символов, уже обнаруженных во входной последовательности, обозначим q. Вероятность того, что в контексте α появится новый символ, оценивается по формуле:

$$\mathcal{P}(\mathtt{esc}|\alpha) = 1 - \sum_{\sigma\in\Sigma,\ c(\sigma)>0} \frac{c(\sigma)-1}{n_\alpha} = 1 - \frac{1}{n_\alpha}\left(\sum_{\sigma\in\Sigma,\ c(\sigma)>0} c(\sigma) - \sum_{\sigma\in\Sigma,\ c(\sigma)>0} 1\right) = 1 - \frac{1}{n_\alpha}(n_\alpha - q) = \frac{q}{n_\alpha}.$$

Метод C. Гибрид двух предыдущих методов. При появлении нового символа на 1 увеличивается как количество переходов, так и количество новых символов, соответственно, общее количество увеличивается на 2. Таким образом, вероятность символа σ в контексте α этот метод оценивает как $\mathcal{P}(\sigma\,|\,\alpha) = c(\sigma)/(n_\alpha + q)$, а вероятность перехода — как:

$$\mathcal{P}(\mathtt{esc}|\alpha) = 1 - \sum_{\sigma\in\Sigma,\ c(\sigma)>0} \frac{c(\sigma)}{n_\alpha + q} = \frac{q}{n_\alpha + q}.$$

Метод D. Небольшая модификация метода C, в которой появление нового символа обрабатывается более единообразно: к количеству переходов и к количеству новых символов вместо 1 добавляется 1/2. Следовательно, этот метод оценивает вероятность символа Σ в контексте α как $\mathcal{P}(\sigma\,|\,\alpha) = (2c(\sigma) - 1)/(2n_\alpha)$, а вероятность перехода — как $\mathcal{P}(\mathtt{esc}\,|\,\alpha) = (q/2n_\alpha)$.

Ни в одном из этих методов не делалось никаких предположений о распределении появления символов в каком-либо контексте. Но предположим, к примеру, что символы появляются в соответствии с процессом Пуассона. Пусть t_i — количество различных символов, встречающихся в выборке размером n_α ровно i раз. Вероятность того, что следующий символ будет новым, приближенно оценивается по формуле $t_1 - t_2 + t_3 - \ldots$ Для простоты можно ограничиться первым членом ряда $\mathcal{P}(\mathtt{esc}\,|\,\alpha) = t_1/n_\alpha$. Дело в том, что в большинстве случаев $n\alpha$ очень велико, а t_i быстро

уменьшается с ростом i. Это соответствует подсчету только символов, однократно появившихся после контекста α. Существуют и более сложные варианты оценки $\mathcal{P}(\texttt{esc} \mid \alpha)$. С ними можно познакомиться в [1], [4], [7], [8].

Список литературы

1. *Clearly J. G., Witten I. H.* Data compression using adaptive coding and partial string matching // IEEE Transactions on Communications, 32: 396–402, 1984.
2. *Duda J.* Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. CoRR abs/1311.2540, 2013.
3. *Howard P., Vitter J. S.* Arithmetic coding for data compression // Proceedings of the IEEE, 857–865, 1994.
4. *Moffat A.* Implementing the PPM data compression scheme // IEEE Transactions on Communications, 38: 1917–1921, 1990.
5. *Moffat A., Petri M.* Large-alphabet semi-static entropy coding via asymmetric numeral systems // ACM Transactions on Information Systems, 38 (4): 33.1–33.33, 2020.
6. *Shannon C. E.* A mathematical theory of communication // Reprinted with corrections from The Bell System Technical Journal, 27: 379–423, 623–656, 1948. https://people.math.harvard.edu/ ctm/home/text/others/shannon/entropy/entropy.pdf.
7. *Witten I. H., Bell T. C.* The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. IEEE Transactions on Information Theory, 37: 1085–94, 1991.
8. *Witten I. H., Moffat A., Bell T. C.* Managing Gigabytes. Morgan Kauffman, second edition, 1999.
9. *Witten I. H., Neal R. M., Clearly J. G.* Arithmetic coding for data compression // Communications of the ACM, 30 (6): 520–540, 1987.

Глава 13
СЖАТИЕ С ИСПОЛЬЗОВАНИЕМ СЛОВАРЕЙ

> Первым шагом стала попытка понять, как изучить что-то, если вы не знаете, что такое статистика.
>
> *Якоб Зив*

В этой главе я продолжу рассказ про методы сжатия данных, но теперь будет рассматриваться подход, отличный от статистического. Алгоритмы, которые я вам продемонстрирую, не извлекают статистическую информацию об источнике входной последовательности *S*. Вместо этого они превращают *S* в *словарь строк*, который используют для замены вхождений этих строк в *S* с помощью соответствующих *токенов* (идентификаторов), служащих словарными *индексами*. Понятно, что решающее влияние на степень сжатия файла оказывает выбор словаря. Например, с английским словарем трудно сжать итальянский текст и уж совсем неуместно сжимать исполняемый файл. Так что, хотя *статический* словарь может отлично справляться со сжатием определенных, заранее известных типов файлов, его невозможно использовать как *универсальный компрессор*. Более того, неэффективно передавать с каждым сжатым файлом полный словарь и нельзя исходить из предположения, что у получателя уже есть его копия.

В конце 1970-х Авраам Лемпель и Якоб Зив представили семейство компрессоров, которые успешно решили эти проблемы. Два алгоритма получили названия LZ77 и LZ78 по инициалам изобретателей и годам появления. Они используют сжимаемую входную последовательность в качестве словаря и заменяют каждое последующее вхождение строки либо смещением ее предыдущей позиции, либо инкрементным идентификатором. Словарь *строится динамически*. Изначально он пуст и наполняется по мере обработки входной последовательности. Вначале степень сжатия невелика, но после нескольких килобайтов при условии некоторой степени *повторяемости* во входной последовательности можно ожидать хороших коэффициентов сжатия. Типичные текстовые файлы этими методами можно сжать примерно на треть. Компрессоры Лемпеля — Зива очень популярны благодаря своей реализации в формате `gzip` и послужили основой более сложных компрессоров, таких как `7zip`, `Brotli`, `LZ4`, `LZMA` и `ZSTD`. Далее я расскажу о них и о других интересных вариантах более подробно.

13.1. LZ77

В знаковой статье 1977 года [7] Лемпель и Зив описали свой вклад следующим образом: «Универсальная схема кодирования, применимая к любому дискретному источнику и производительность которой сопоставима с некоторыми оптимальными фиксированными схемами кодирования с помощью словаря, разработанными для конкретных источников». Ключевые слова здесь — «сопоставима с <...> схемами кодирования с помощью словаря, разработанными для конкретных источников», так как авторы сравнивают свою схему с ранее разработанными статистическими алгоритмами сжатия, такими как алгоритм Хаффмана и арифметическое кодирование, для которых требовалась статистическая характеристика источника. Их компрессоры от этой характеристики отказались. Она получается *неявно* путем наблюдения за *повторяемостью подстрок* с помощью полностью синтаксического подхода.

Я не буду углубляться в математическое обоснование этих комментариев (см. [1], [7]), а рассмотрю алгоритмические вопросы, лежащие в основе разработки. Основу алгоритма LZ77 составляют *скользящее окно* $W[1, w]$, содержащее уже обработанную часть входной последовательности, обычно состоящую из последних w символов, и *буфер опережающего просмотра* B, который включает в себя суффикс подлежащего обработке текста. В следующем примере окно $W = aabbbababb$ имеет размер 9 (взято в рамку), а остальная часть входной последовательности равна $B = baababaabbaa\$$:

$$\leftarrow..\boxed{aabbbababb}\,baababaabbaa\$\rightarrow.$$

Алгоритм работает индуктивно, предполагая, что все до B обработано и сжато, а W изначально присвоена пустая строка. Выделяются два основных этапа: *анализ* и *кодирование*. В процессе анализа входная последовательность S преобразуется в последовательность троек целых чисел, называемых *фразами*. Кодирование превращает эти тройки в сжатый поток битов, применяя к каждому компоненту по отдельности либо статистический компрессор (например, алгоритм Хаффмана или арифметическое кодирование), либо схему целочисленного кодирования.

Синтаксический анализ работает следующим образом. Алгоритм LZ77 ищет в части B самый длинный префикс α, который встречается как подстрока $W \times B$. Запись конкатенации $W \times B$ вместо одной строки W обусловлена тем, что искомое предыдущее вхождение может начинаться в W и продолжаться до B. Допустим, префикс α располагается на расстоянии d от начала B, за ним в B следует символ c. В этом случае алгоритм LZ77 сгенерирует тройку $\langle d, |\alpha|, c\rangle$, где $|\alpha|$ — длина *скопированной* строки. Если совпадение не найдено, выходная тройка принимает вид $\langle 0, 0, B[1]\rangle$. Замечу, что за любым вхождением α в $W \times B$ должен следовать символ, отличный от c, в противном случае α *не будет* самым длинным префиксом B, который повторяется в $W \times B$.

После генерации этой тройки алгоритм LZ77 продвигается в B на позицию $|\alpha| + 1$ и соответствующим образом сдвигает W. Если $|W| = +\infty$, то окно не ограничено, и таким образом механизм анализа LZ77 может копировать вплоть до начала сжимаемого файла. Нетрудно убедиться, что процесс анализа минимизирует количество генерируемых фраз.

Алгоритм LZ77 называют компрессором на основе словаря, потому что словарь не хранится в явном виде, скорее он формируется всеми подстроками S, которые начинаются в W и продолжаются вправо, возможно заканчиваясь в B. Каждая из подстрок представлена парой $\langle d, |\alpha| \rangle$. Это *динамический* словарь, так как при каждом сдвиге он требует обновления путем удаления подстрок, начинающихся в $W[1, |\alpha| + 1]$, и добавления подстрок, начинающихся в $B[1, |\alpha| + 1]$.

Роль скользящего окна объяснить легко: оно ограничивает размер словаря, который зависит от длины W и S, поскольку включает строки $|W|$ длиной до $|S|$. Таким образом, W существенно влияет на временные затраты на поиск α. В качестве примера рассмотрим следующие этапы анализа (черта | разделяет здесь W и B):

|aabbabab⇒ копии не найдены, генерируется тройка $\langle 0, 0, a \rangle$;

a|abbabab⇒ копирует a, затем генерирует $\langle 1, 1, b \rangle$;

aab|babab⇒ копирует b, затем генерирует $\langle 1, 1, a \rangle$;

aabba|bab⇒ обнаружено перекрывающееся копирование,
алгоритм генерирует $\langle 2, 3, EOF \rangle$.

Интересно отметить, что последняя фраза, $\langle 2, 3, EOF \rangle$, указывает на то, что длина копирования больше расстояния копирования. Фактически это особая ситуация, о которой я упоминал ранее, когда α начинается в W и заканчивается в B. Но даже такое *перекрывание* не влияет на шаг копирования, который алгоритм LZ77 будет выполнять на этапе распаковки, при условии, что он выполняется последовательно в соответствии со следующим фрагментом кода:

```
for i = 0 to L-1 do { S[s + i] = S[s-d + i]; }
s = s + L;
```

где декодируемая тройка — $\langle d, L, c \rangle$, а $S[1, s - 1]$ — префикс уже распакованной входной последовательности. Так как $d \leqslant |W|$, а размер окна доходит до нескольких мегабайтов, операция копирования не вызывает промахов кэша, что делает процесс распаковки очень быстрым. Чем больше окно W, тем длиннее могут быть фразы, тем меньше их количество и, следовательно, возможно, быстрее сжатый вывод. Но, к сожалению, больше времени требуется для поиска самого длинного скопированного префикса α. И наоборот, чем короче W, тем хуже коэффициент сжатия, но тем быстрее оно выполняется. Величина этого компромисса зависит от входной последовательности. Удивительно, но на современных компьютерах производительность этапа распаковки противоположна производительности

этапа сжатия, поскольку количество фраз влияет на количество выполняемых копирований и, как следствие, на эффективность кэширования и предварительной выборки.

Немного улучшить сжатие смогли Сторер и Шиманский [5] в 1982 году. Дело в том, что в процессе синтаксического анализа могут возникнуть две ситуации: самое длинное совпадение или найдено, или не найдено. В первом случае после α (третий компонент в тройке) неразумно добавлять символ, учитывая, что продвижение по входной последовательности происходит всегда. Во втором случае неразумно генерировать два нуля (первые два компонента тройки), впустую расходуя один целочисленный код. Проще всего обойти эти неэффективные моменты, всегда выводя не тройку, а пару в форме $\langle d, |\alpha| \rangle$ или $\langle 0, B[1] \rangle$. Этот вариант алгоритма LZ77 называется LZss, и с этого момента мы будем использовать именно его.

Для предыдущего примера синтаксический анализ алгоритма LZss будет выглядеть так:

$$\boxed{|}\text{aabbabab} \Rightarrow \langle 0, a \rangle;$$
$$\text{a}\boxed{|}\text{abbabab} \Rightarrow \langle 1, 1 \rangle;$$
$$\text{aa}\boxed{|}\text{bbabab} \Rightarrow \langle 0, b \rangle;$$
$$\text{aab}\boxed{|}\text{babab} \Rightarrow \langle 1, 1 \rangle;$$
$$\text{aabb}\boxed{|}\text{abab} \Rightarrow \langle 3, 2 \rangle;$$
$$\text{aabbab}\boxed{|}\text{ab} \Rightarrow \langle 2, 2 \rangle.$$

Во время распаковки определить доступный тип пары очень просто: считываем первое целое число, и если оно равно нулю, то это фраза из одного символа, а следующие биты кодируют его. В противном случае это скопированная фраза, расстояние до начала которой указано в считанном целом числе (больше нуля), а следующие биты кодируют ее длину.

gzip — классическая, хорошая и быстрая реализация LZ77. Ключевая задача программирования при реализации LZ77 — *быстрый поиск* в буфере опережающего просмотра B самого длинного префикса α, который начинается с W. Проверять полным перебором при линейном обратном сканировании вхождение каждого префикса из B, который начинается с W, крайне трудоемко. Следовательно, такой вариант неприемлем для сжатия длинных файлов.

К счастью, существует структура данных, позволяющая ускорить этот процесс. Самая популярная классическая реализация LZ77 — `gzip` — для определения α и поиска его предыдущего вхождения в W использует хеш-таблицу. В ней сохраняют все триграммы из конкатенации $W \times B[1, 2]$, то есть все тройки смежных символов. Триграмма выступает в качестве ключа, а ее позиция в W сохраняется в качестве *дополнительных* данных. Для повторяющихся триграмм хеш-таблица сохраняет все вхождения, отсортированные по возрастанию их позиции в S. При сдвиге W вправо после генерации пары $\langle d, \ell \rangle$ хеш-таблица обновляется путем удаления триграмм,

начинающихся с $W[1, \ell]$, и вставки триграмм, начинающихся с $B[1, \ell]$. В общей сложности происходит ℓ удалений из хеш-таблицы и ℓ вставок в нее.

Поиск α осуществляется следующим образом.

- В хеш-таблице ищется триграмма $B[1, 3]$. Если она не найдена, то `gzip` генерирует фразу $\langle 0, B[1]\rangle$ и анализ продвигается на один символ. В противном случае определяется список $\mathcal{L}$ вхождений $B[1, 3]$ в конкатенацию $W \times B[1, 2]$.
- Для каждой позиции i в $\mathcal{L}$, которая выражается как абсолютная позиция в S, алгоритм посимвольно сравнивает суффикс $S[i, n]$ с B, чтобы найти их самый длинный общий префикс α. В конце определяется разделяющая этот префикс позиция $i^* \in \mathcal{L}$.
- Если p — текущая позиция B в последовательности S, алгоритм генерирует пару $\langle p - i^*, |\alpha|\rangle$ и продвигает окно W и буфер B на позиции $|\alpha|$.

Кодирование фраз в `gzip` реализуется с помощью алгоритма Хаффмана, который применяется к двум алфавитам: один состоит из длин копий и литералов, второй — из расстояний до этих копий. Такой трюк экономит дополнительный бит при распознавании двух типов пар. Фактически $\langle d, \ell\rangle$ реализована как кодирование Хаффмана для символа c, а $\langle d, \ell\rangle$, наоборот, получена предугадыванием кодирования Хаффмана для ℓ. Литералы и длины копий кодируются одним набором символов, что позволяет декодеру после извлечения и распаковки кодового слова распознать полученный элемент. В зависимости от результата этого распознавания второй алгоритм Хаффмана декодирует или следующую пару (если декодирован символ c), или символ d (декодировано ℓ).

В `gzip` использован программный трюк, еще больше ускоряющий процесс сжатия. Это сортировка списка вхождений триграмм от самых новых к самым старым совпадениям и возможная остановка поиска α после проверки достаточного количества кандидатов. Такой подход позволяет жертвовать размером самого длинного совпадения ради скорости поиска. Для размера окна W в командной строке можно указывать параметры –1... –9, что фактически означает возможность варьировать размер W от 100 до 900 Кбайт с последующим улучшением коэффициента сжатия за счет уменьшения скорости сжатия. Как легко заметить, чем длиннее W, тем быстрее распаковка, потому что количество закодированных фраз меньше и, следовательно, декодирование алгоритмом Хаффмана порождает меньше копирований в памяти и меньше промахов кэша.

Про другие реализации можно прочитать в главе 10, где обсуждалось использование дерева суффиксов при неограниченном размере окна. Другие интересные вопросы возникают, если принять во внимание не только количество фраз, но и размер сжатого вывода (в битах). Этот размер явно зависит от количества фраз, а также от значений их целочисленных компонентов, причем эту зависимость нельзя недооценивать [2]. Если вкратце, более длинный префикс α далеко не всегда порождает более короткий сжатый файл, потому что расстояние d до его копии может оказаться очень большим, в результате чего для его кодирования потребуется много битов.

Верно и обратное: иногда лучше разделить α на две подстроки, которые можно скопировать достаточно близко, чтобы общее количество требующихся для их кодирования битов оказалось меньше, чем нужно для *d*.

13.2. LZ78

Используемое алгоритмом LZ77 скользящее окно ускоряет поиск самой длинной фразы для кодирования, но одновременно ограничивает пространство поиска и, как следствие, конечную степень сжатия. Чтобы избежать этой проблемы, сохранив скорость сжатия, в 1978 году Лемпель и Зив разработали другой алгоритм, впоследствии названный LZ78 [8]. Его основная суть заключается в *пошаговом построении явного словаря* из подмножества подстрок входной последовательности *S*. Подстроки выбираются в соответствии с простым правилом, которое будет подробно описано далее. Одновременно последовательность *S* разбивается на фразы, которые берутся из текущего словаря и кодируются по таким же схемам сжатия, как и в алгоритме LZ77.

Распознавание фраз тесно связано с обновлением словаря. Возьмем уже знакомую по алгоритму LZ77 нотацию. Пусть B — анализируемая последовательность, а $\mathcal{D}$ — словарь, в котором каждая фраза f идентифицируется с помощью целого числа $\texttt{id}(f)$. Анализ B снова сводится к поиску самого длинного префикса α, который также является фразой в словаре $\mathcal{D}$, и замены его на пару $\langle \texttt{id}(\alpha), c \rangle$, где c — символ, идущий в B после α, то есть $c = B[|\alpha| + 1]$. Затем словарь $\mathcal{D}$ обновляется добавлением новой фразы αc. Полученный словарь будет *завершенным по префиксам* (prefix-complete), так как в нем окажутся все префиксы всех фраз в $\mathcal{D}$. Размер словаря растет вместе с длиной входной последовательности. Как и в случае с алгоритмом LZ77, поток генерируемых пар можно закодировать с помощью статистического компрессора (например, алгоритма Хаффмана или арифметического кодирования) или с помощью любого кодировщика целых чисел переменной длины. Это создаст сжатый поток битов, который является конечным результатом работы алгоритма LZ78.

Декомпрессор работает очень похожим образом: он считывает из сжатого потока пару $\langle \texttt{id}, x \rangle$, определяет фразу α, соответствующую целочисленному `id` в словаре $\mathcal{D}$, генерирует подстроку αc и обновляет словарь, добавляя эту подстроку как новую фразу.

В качестве примера рассмотрим последовательность *S*, приведенную на рис. 13.1. В графе «Входные данные» в таблице показана последовательность фраз α, обнаруженная в *S* в процессе синтаксического анализа. Графа «Вывод» содержит результат замены этих фраз парами $\langle \texttt{id}(\alpha), c \rangle$. Последняя графа таблицы показывает соответствующие фразы αc, постепенно добавляемые в словарь $\mathcal{D}$, и присвоенные им идентификаторы.

Справа на рисунке находится структура данных, с помощью которой алгоритм LZ78 эффективно по времени и пространству реализует словарь $\mathcal{D}$. Это префиксное

дерево (см. главу 9), поддерживающее быструю вставку и префиксный поиск строк. Свойство завершенности по префиксам, которому удовлетворяет словарь $\mathcal{D}$, гарантирует, что префиксное дерево несжатое, то есть каждое ребро помечено одним символом. Рядом с каждым узлом u показана пара (`id`(f), c). Здесь символ c — метка входящего в узел u ребра, а f — фраза, генерируемая при спуске из корня в u. Например, узел, помеченный парой $(8, a)$, соответствует словарной строке aa с `id` = 8. Легко заметить, что по мере спуска по префиксному дереву идентификаторы увеличиваются, потому что к коротким фразам по одному прибавляются новые символы. Алгоритм кодирования прекрасно вписывается в структуру префиксного дерева. Фактически поиск самого длинного префикса B, который является словарной фразой, реализуется путем обхода дерева в соответствии с символами, из которых состоит B, и сопоставления меток ребер, пока не будет достигнут лист v. Сформированная при этом строка и будет фразой α. Новая фраза αc легко вставляется в префиксное дерево и, соответственно, в словарь $\mathcal{D}$ — достаточно добавить к v новый лист и маркировать его символом $c = B[|\alpha| + 1]$ и `id`, равным размеру словаря до момента вставки. Напоследок отмечу, что из-за завершенности словарей LZ78 по префиксам размер префиксного дерева равен *количеству* строк словаря, а не их общей длине.

Ввод	Вывод	Словарь
-	-	0: пустая строка
a	<0, a>	1: a
ab	<1, b>	2: ab
b	<0, b>	3: b
aba	<2, a>	4: aba
bb	<3, b>	5: bb
ba	<3, a>	6: ba
abab	<4, b>	7: abab
aa	<1, a>	8: aa

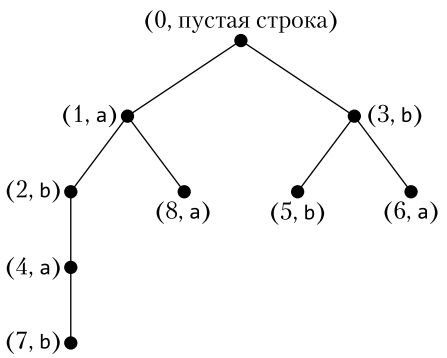


Рис. 13.1. Результат анализа алгоритмом LZ78 строки S = *aabbababbbaababaa* и несжатое префиксное дерево, построенное по соответствующему словарю

Последний вопрос заключается в том, как алгоритм LZ78 управляет большими файлами и, следовательно, большими словарями. Есть разные способы решения этой задачи. Во-первых, при достижении максимального размера словаря можно запретить ввод новых строк (этот подход выгоден при входной последовательности S с повторяющимся шаблоном подстрок). Во-вторых, заполненный словарь можно отбросить и начать новый (выгодно, когда входная последовательность S структурирована блоками, каждый с собственным повторяющимся шаблоном подстрок). Наконец, при обновлении словаря после вставки новой фразы из него удаляется одна из реже всего используемых в последнее время (это решение напоминает своего рода политику кэширования LRU).

13.3. LZW

Большую популярность набрал разработанный Уэлчем в 1984 году [6] вариант алгоритма LZ78, который получил название LZW. Его главная цель состояла в том, чтобы убрать необходимость во втором компоненте пары $\langle \texttt{id}(\alpha), c \rangle$, то есть в байте, представляющем дополнительный символ. Для достижения этой цели перед началом алгоритма все возможные односимвольные строки записываются в словарь. Это означает, что этим однобайтовым символам выделены идентификаторы фраз от 0 до 255. Синтаксический анализ последовательности S, как обычно, начинается с поиска самого длинного префикса α, который соответствует фразе в словаре $\mathcal{D}$. Следующий префикс αc из B в D не встречается, поэтому αc добавляется в словарь под следующим доступным идентификатором, а очередная искомая фраза начинается с c, а не со следующего символа, как сделано в алгоритмах LZ78 и LZ77. Фактически синтаксический анализ и обновление словаря *не согласованы*, что немного усложняет декодирование.

Предположим, что нужно обработать два последовательных идентификатора, i' и i'', и назвать соответствующие им словарные фразы α' и α''. Для перестроения словаря декодер должен из результатов чтения i' и i'' создать новую фразу f. Она имеет вид $f = \alpha'\alpha''[1]$, где $\alpha''[1]$ — первый символ фразы α''. Задача кажется простой, но это не так, потому что фраза α'' может быть недоступна в словаре $\mathcal{D}$.

Чтобы лучше понять эту концепцию, рассмотрим момент, когда компрессор сгенерировал идентификатор i' для фразы α' и вставил в словарь фразу $f = \alpha'\alpha''[1]$. Очевидно, что фраза α'' компрессору известна, потому что ему известна последовательность S, значит, он может построить f и вставить ее в словарь. Но когда следующая фраза совпадает с только что вставленной, а именно $\alpha'' = f$, начинаются проблемы, потому что декомпрессору нужно построить f как $\alpha'\alpha''[1] = \alpha' f[1]$, то есть ему требуется первый символ конструируемой фразы. Это рекурсивное определение можно решить, заметив, что $|\alpha'| \geqslant 1$, поэтому $f[1] = \alpha'[1]$. И декодер LZW может построить f как $\alpha'\alpha'[1]$, используя только фразы, доступные в текущем словаре. Затем f вставляется в $\mathcal{D}$, и таким образом LZW перестраивает словари, доступные на этапах сжатия и распаковки после чтения i'.

На рис. 13.2 показан пример кодирования *(слева)* и декодирования *(справа)* алгоритмом LZW строки S = *aabbabab*. Словарь начинается со всех возможных 256 символов в коде ASCII, поэтому идентификаторы новых фраз начинаются с 256. Вопросительный знак (?) рядом с фразами в графе «Словарь» правой таблицы указывает на их несоответствие стадии декодирования и необходимость отложить построение текущей фразы словаря до тех пор, пока не станет доступной следующая. Фактически правая таблица содержит пары строк для каждой сконструированной фразы. Все эти сочетания возможны, так как они включают доступные словарные фразы, за исключением последней, с `id` = 260. Этот `id` запускает упоминавшийся ранее особый случай. После прочтения `id` = 257 соответствующая ему фраза α' = *ab* будет доступна в словаре $\mathcal{D}$, но фразы α'' с `id` = 260 там еще нет. Тем не менее, как

объяснялось ранее, можно сделать вывод, что $f = \alpha'\alpha'[1] = aba$. Эта фраза вставляется в словарь, и словари кодировщика и декодера LZW обновляются.

Ввод	Вывод	Словарь
a	-	0-255: '\0'-'\255'
a	97 (a)	256: aa
b	97 (a)	257: ab
b	98 (b)	258: bb
ab	98 (b)	259: ba
aba	257 (ab)	260: aba
eof	260 (aba)	261: aba EOF

Ввод	Словарь	Вывод
-	0-255: '\0'-'\255'	
97	256: a?	a
97	256: aa	a
	257: a?	
98	257: ab	b
	258: b?	
98	258: bb	b
	259: b?	
257	259: ba	ab
	260: ab?	
260	261: aba	aba

Рис. 13.2. Предполагается, что словарь LZW начинается со всех отдельных символов ASCII с идентификаторами от 0 до 255. *Слева* — результат кодирования алгоритмом LZW последовательности *S* = *aabbabab*, 97 и 98 — коды ASCII для символов *a* и *b*; *справа* — результат декодирования алгоритмом LZW потока идентификаторов 97, 97, 98, 98, 257, 260

У алгоритма LZW, как и у LZ77, много популярных реализаций, в том числе формат GIF[1]. В исходном (несжатом) изображении на 1 пиксел приходится 8 бит. Соответственно, алфавит имеет размер 256, а входная последовательность *S* представляет собой поток байтов, полученных построчным чтением пикселов изображения[2]. Для представления всех возможных цветов изображения 8 бит очень мало, поэтому каждое значение на самом деле является индексом в *палитре*, записи которой представляют собой 24-битные описания фактического цвета (типичный формат RGB). В любом случае максимальное количество цветов изображения не превышает 256. Некоторые авторы [3] изучали возможность сжатия изображений GIF *с потерями*, не меняя способа представления выходных данных [5]. Основная идея довольно проста: в словаре ищется не самое длинное *точное* совпадение, а *приблизительное* совпадение. Это позволяет находить более длинные фразы, что уменьшает размер выходных данных ценой небольшого искажения изображения. Приблизительное сопоставление двух строк для цветов происходит измерением разницы с фактическими значениями RGB. Она должна быть гарантированно меньше порогового значения, чтобы не искажать исходное изображение слишком сильно.

---

[1] См.: https://ru.wikipedia.org/wiki/GIF.

[2] Формат GIF допускает чересстрочное хранение данных, подробности которого выходят за рамки краткого обсуждения, но алгоритм сжатия будет тем же самым.

13.4. Оптимальность компрессоров∞

В литературе представлено множество исследований оптимальности алгоритмов, созданных на базе LZ. Сами Лемпель и Зив продемонстрировали, что алгоритм LZ77 оптимален для определенного семейства источников (см. [7]) и что LZ78 асимптотически достигает наилучшей степени сжатия среди компрессоров с конечным числом состояний (см. [8]). Оптимальность здесь означает, что степень сжатия бесконечной сжимаемой строки, создаваемой *стационарным эргодическим источником с конечным алфавитом*, асимптотически приближается к энтропии базового источника. Позднее была получена количественная оценка *избыточности* алгоритма, которая является мерой расстояния между энтропией источника и степенью сжатия и поэтому может рассматриваться как мера того, насколько быстро алгоритм достигает энтропии источника.

Все эти параметры очень интересны, но нереалистичны, поскольку не совсем понятно, как узнать энтропию сгенерировавшего строку источника. Для обхода этой проблемы было введено понятие $\mathcal{H}_k(S)$ — *эмпирической энтропии k-го порядка* строки S. В главе 12 обсуждался случай $k = 0$, где в расчет брались частоты отдельных символов в S. Теперь же определение энтропии расширено, и в последовательности S будут рассматриваться частоты наборов из k символов. То есть во внимание начнут приниматься подпоследовательности символов, а следовательно, и *композиционная структура S*.

Рассмотрим последовательность S, составленную из алфавита $\Sigma = \{\sigma_1 \ldots \sigma_h\}$, обозначим $n\omega$ количество вхождений подстроки ω в S. Обозначение $\omega \in \Sigma^k$ указывает, что длина ω равна k. С учетом этих обозначений эмпирическая энтропия порядка k будет записываться так:

$$\mathcal{H}_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma^k} \left(\sum_{i=1}^{h} n_{\omega\sigma_i} \log\left(\frac{n_\omega}{n_{\omega\sigma_i}} \right) \right).$$

Алгоритм сжатия определяется как *близкий к оптимальному* тогда и только тогда, когда для всех k существует функция $f_k(n)$, стремящаяся к 0 при $n \to \infty$, такая, что для всех растущих последовательностей S коэффициент сжатия оцененного алгоритма не превышает $\mathcal{H}_k(S) + f_k(|S|)$. Для алгоритма LZ78 Плотник, Вайнбергер и Зив доказали это в [4]. Косараджу и Манзини отметили [1], что понятие «близкий к оптимальному» не обязательно подразумевает хороший алгоритм, ведь даже при приближении энтропии строки S к нулю он все равно может демонстрировать плохое сжатие из-за аддитивного члена $f_k(|S|)$.

Лемма 13.1. *Существуют строки, для которых степень сжатия, достигаемая с помощью алгоритма LZ78, составляет по крайней мере* $g(|S|)\mathcal{H}_0(S)$, где $\lim_{n\to\infty} g(n) = \infty$.

Доказательство. Рассмотрим строку $S = 01^{n-1}$ с энтропией $\mathcal{H}_0(S) = \Theta((\log n)/n)$. Легко увидеть, что алгоритм LZ78 разбивает строку S на $\Theta\left(\sqrt{n}\right)$ фраз. Отсюда

несложно получить $g(n)=\frac{\sqrt{n}}{\log n}$. Таким образом, по мере роста строки S энтропия $\mathcal{H}_0(S)$ снижается, но коэффициент сжатия, который дает алгоритм LZ78, уменьшается медленнее. ■

Аналогичная ситуация была связана с дополнительным битом, необходимым для каждого закодированного символа, в алгоритме Хаффмана. Этот дополнительный бит, приемлемый при высоких энтропиях, становился плохим при стремлении энтропии к 0. Чтобы обойти эти недостатки, Косараджу и Манзини ввели более строгую версию оптимальности — так называемую λ-оптимальность, применимую к любому алгоритму, коэффициент сжатия которого может быть ограничен $\lambda\mathcal{H}_k(S) + o(\mathcal{H}_k(S))$. Из предыдущей леммы легко увидеть, что алгоритм LZ78 не относится к λ-оптимальным. Существует его модифицированная версия, к которой добавлено кодирование повторов (описывается в главе 14), — 3-оптимальная для $\mathcal{H}_0$, но было показано, что для всех $k \geqslant 1$ она не является λ-оптимальной.

Теперь рассмотрим алгоритм LZ77, который с виду кажется более мощным, чем LZ78, учитывая, что его словарь богаче подстроками. Но используемый на практике вариант LZ77 с фиксированным окном сжатия оказывается хуже LZ78.

Лемма 13.2. *Алгоритм LZ77 с ограниченным скользящим окном не является грубо оптимальным.*

Доказательство. Для каждого размера скользящего окна L можно найти такую строку S, коэффициент сжатия которой превышает ее энтропию порядка k. Рассмотрим строку $(0^k1^k)^n1$ длиной $2kn + 1$ бит и выберем $k = L - 1$. Благодаря скользящему окну алгоритм LZ77 разбивает S следующим образом:

$$\underline{0}\ \underline{0^{k-1}1}\ \underline{1^{k-1}0}\ \underline{0^{k-1}1}...\underline{1^{k-1}0}\ \underline{0^{k-1}1}\ \underline{1^k}.$$

Каждая фраза имеет длину до k, разделяя входные данные на $\Theta(n)$ фраз и таким образом достигая выходного размера $\Omega(n)$.

Чтобы вычислить $\mathcal{H}_k(S)$, нужно работать со всеми подстроками строки S длиной k, которые равны $2k$: $\{0^i1^{k-i}\}_{i=1...k} \cup \{1^i0^{k-i}\}_{i=1...k}$. Теперь за всеми строками вида 0^i1^{k-i} всегда следует 1, а за всеми строками вида 1^i0^{k-i} всегда следует 0. Только за строкой 1^k $n - 1$ раз следует 0 и один раз 1. Таким образом, в рамках определения $\mathcal{H}_k(S)$ сумму по k-граммам ω можно разделить на четыре части:

$$\begin{aligned}
&\omega \in \{0^i1^{k-i}\}_{i=1...k} && \to n_{\omega 0} = 0, && n_{\omega 1} = n;\\
&\omega \in \{1^i0^{k-i}\}_{i=1...k-1} && \to n_{\omega 0} = n, && n_{\omega 1} = 0;\\
&\omega = 1^k && \to n_{\omega 0} = n - 1, && n_{\omega 1} = 1;\\
&\text{в противном случае} && \to n_{\omega 0} = 0, && n_{\omega 1} = 0.
\end{aligned}$$

Теперь легко посчитать:

$$|S|\mathcal{H}_k(S) = \log n + (n-1)\log\frac{n}{n-1} = \Theta(\log n).$$

Лемма доказана. ■

При отказе от скользящего окна алгоритм LZ77 становится приблизительно оптимальным, а в случае $\mathcal{H}_0$ также 8-оптимальным. Однако он не является λ-оптимальным для любого $k \geqslant 1$.

Лемма 13.3. *Существуют строки, для которых коэффициент сжатия LZ77 без скользящего окна составляет по крайней мере* $g(|S|)\mathcal{H}_1(S)$, *при этом* $\lim_{n\to\infty} g(n) = \infty$.

Доказательство. Рассмотрим строку $10^k 2^{2^k} 1\,101\,10^2 1\,10^3 1 \ldots 10^k 1$ длиной $2^k + O(k^2)$ и предельное значение энтропии порядка k, равное $|S|\mathcal{H}_k(S) = k\log k + O(k)$. Строка разбивается на $k + 4$ фраз:

$$\underline{1}\,\underline{0}\,\underline{0^{k-1}2}\,\underline{2^{2^k-1}1}\,\underline{101}\,\underline{10^2 1}\ldots\underline{10^k 1}.$$

Проблема в том, что последние k фраз ссылаются на начало строки S, находящееся на расстоянии 2^k символов. Это генерирует $\Omega(k)$ длинных фраз, что дает общий размер вывода $\Omega(k^2)$ бит. ■

Итак, как и ожидалось, алгоритм LZ77 лучше, чем LZ78, но не так хорош, как хотелось бы, для $k \geqslant 1$. В следующей главе будет представлено предложенное в 1994 году преобразование Барроуза — Уилера, которое устраняет неэффективность методов на основе LZ, демонстрируя новый подход к сжатию данных, позволяющий достичь λ-оптимальности для очень малых λ одновременно для всех $k \geqslant 0$. Поэтому неудивительно, что созданный на базе этого преобразования компрессор `bzip2`, доступный в большинстве дистрибутивов операционных систем, выдает более сжатый вывод, чем `gzip`, и приобрел большую известность как в сообществе специалистов по сжатию данных, так и в других местах.

Список литературы

1. *Kosaraju S. R., Manzini G.* Compression of low entropy strings with Lempel — Ziv algorithms // Siam Journal on Computing, 29 (3): 893–911, 1999.
2. *Ferragina P., Nitto I., Venturini R.* On the bit-complexity of Lempel — Ziv compression // SIAM Journal on Computing, 42 (4): 1521–1541, 2013.
3. *Pigeon S.* An optimizing lossy generalization of LZW // Proceedings of the IEEE Data Compression Conference, 509, 2001.

4. *Plotnik E., Weinberger M., Ziv J.* Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel — Ziv algorithm // IEEE Transactions on Information Theory, 38: 16–24, 1992.
5. *Storer J. A., Szymanski T. G.* Data compression via textual substitution // Journal of the ACM, 29(4): 928–951, 1982.
6. *Welch T. A.* A technique for high-performance data compression // Computer, 17 (6): 8–19, 1984.
7. *Ziv J., Lempel A.* A universal algorithm for sequential data compression // IEEE Transactions on Information Theory, 23 (3): 337–343, 1977.
8. *Ziv J., Lempel A.* Compression of individual sequences via variable-rate coding // IEEE Transactions on Information Theory, 24 (5): 530–536, 1978.

Глава 14
СЖАТИЕ С СОРТИРОВКОЙ БЛОКОВ ДАННЫХ

> Прошли годы, и стало ясно, что Дэвид и не думал публиковать алгоритм — он был слишком занят обдумыванием новых идей.
>
> *Майк Барроуз*

В этой главе описывается разработанный Майком Барроузом и Дэвидом Уилером в исследовательском центре DEC Systems метод сжатия данных без потерь[1]. Публикация о нем появилась в техническом отчете компании [4], [9], но, так как его не приняли на тематической конференции IEEE Data Compression Conference 1994 года, как заявил Майк Барроуз в предисловии к [10][2], от дальнейших публикаций авторы решили воздержаться. К счастью, Марк Нельсон в своей статье в издании *Dr. Dobb's Journal* («Журнал доктора Добба») привлек внимание к этому методу, обеспечив его успешное распространение в научном сообществе.

Публикация идеи позволяет привлечь к решению имеющихся проблем других людей. Именно это произошло с преобразованием Барроуза — Уилера, изучением которого в 2000 году занялись множество исследователей. В результате десятилетие спустя в специальном выпуске журнала *Theoretical Computer Science* [10] группа исследователей опубликовала результаты, наработанные за это время. Майк Барроуз снова отказался публиковать оригинальный технический отчет, но написал замечательное предисловие, посвященное памяти скончавшегося в 2004 году Дэвида Уилера, заявив в конце: «Этот выпуск *Theoretical Computer Science* демонстрирует, как можно улучшить и обобщить идею, когда в нее вовлечено больше людей. Уверен, что Дэвид Уилер был бы рад видеть, что его метод вдохновил так много интересных работ».

Преобразование Барроуза — Уилера (The Burrows — Wheeler transform, BWT) предложило революционную альтернативу компрессорам на основе словаря и статистическим компрессорам. Это был новый класс подходов к сжатию данных (компрессор `bzip2` [19] или усилитель сжатия [7]), а также новый мощный класс сжатых индексов

---

[1] Майк Барроуз: «В техническом отчете, описывающем BWT, я указал 1981 год, но позже, услышав воспоминания жены Уилера Джойс, мы пришли к выводу, что это произошло в 1978-м» [10].

[2] Майк Барроуз: «Прошли годы, и стало ясно, что Дэвид и не думал публиковать алгоритм — он был слишком занят обдумыванием новых идей. В конце концов я решил заставить его: я не смог убедить его написать статью, но, если бы появился подходящий повод, я мог бы написать статью вместе с ним».

(FM-индекс [8] и множество его вариаций [16]). В этой главе будут подробно рассмотрены BWT и два других простых преобразования: перемещение к началу и кодирование длин серий, сочетание которых составляет ядро проектирования компрессоров на основе `bzip`. Такие преобразования известны как *блочно-сортирующее сжатие*. Будут упомянуты и теоретические вопросы производительности BWT, выраженные в терминах эмпирической энтропии сжимаемых данных k-го порядка. Кроме того, я обрисую основные алгоритмические вопросы, лежащие в основе проектирования первого на сегодняшний день доказуемо сжатого индекса — FM-индекса.

14.1. Преобразование Барроуза — Уилера

Преобразование Барроуза — Уилера нельзя отнести к алгоритмам сжатия, так как размера входных данных оно не меняет. Это *перестановка* входных символов (то есть преобразование без потерь), в результате которой строка лучше всего сжимается простыми алгоритмами, такими как алгоритм *перемещения к началу* (move-to-front, MTF) и алгоритм *кодирования длин серий* (run length encoding, RLE). Им посвящен раздел 14.2. Такая перестановка делает порядок символов локально однородным, что позволяет эффективно и действенно использовать сочетание MTF + RLE. В конце для сжатия выходного потока битов применяется статистическое кодирование, например алгоритм Хаффмана или арифметическое кодирование. Все эти этапы составляют основу любого из `bzip`-подобных компрессоров, которые будут рассматриваться в разделе 14.3.

Преобразование BWT состоит из пары взаимно обратных преобразований. Первое переставляет символы входной строки, а второе как по волшебству восстанавливает исходную строку. Подобная обратимость обеспечивает гарантированную распаковку входного файла.

14.1.1. Прямое преобразование

Пусть $S[1, n]$ — входная строка из n символов, взятых из *упорядоченного* алфавита Σ. К строке S добавлен специальный символ \$, не входящий в Σ, который меньше любого другого символа алфавита в соответствии с общим порядком[1].

Рассмотрим, как выполняется прямое преобразование.

1. Построим строку S\$.
2. Рассмотрим *концептуальную* матрицу $\mathcal{M}$ размером $(n + 1)(n + 1)$, строки которой включают все циклические левые сдвиги строки S\$. $\mathcal{M}$ называется *матрицей поворота* строки S[2].

---

[1] Шаг, объединяющий начальную строку и специальный символ \$, не входил в оригинальную версию алгоритма Барроуза — Уилера. Здесь он добавлен для упрощения описания.

[2] Сдвиг строки $a\alpha$ влево дает строку αa, то есть первый символ перемещается в конец исходной строки.

3. Упорядочим строки M, читая их *слева направо* в соответствии с порядком, определенным в алфавите $\Sigma \cup \{\$\}$. В результате получим матрицу $\mathcal{M}'$. Поскольку \$ меньше любого другого символа алфавита Σ, в первой строке $\mathcal{M}'$ окажется $\$S$.
4. В качестве выходных данных алгоритма установим $bw(S) = \left(\hat{L}, r\right)$, где $\hat{L}$ — строка, полученная чтением последнего столбца $\mathcal{M}'$, без символа \$, а r — позиция символа \$ в этом столбце.

Название «концептуальная матрица» в случае $\mathcal{M}$ показывает, что нужно избегать ее явного построения, которое превращает преобразование BWT в элегантный математический объект: размер $\mathcal{M}$ пропорционален квадрату длины $bw(S)$, так что даже для строки в несколько мегабайтов концептуальная матрица будет иметь размер терабайта. В разделе 14.3 я покажу, что, если воспользоваться для построения $\mathcal{M}'$ массивами суффиксов (они рассматривались в главе 10), временные и пространственные затраты будут линейно зависеть от длины входной строки S.

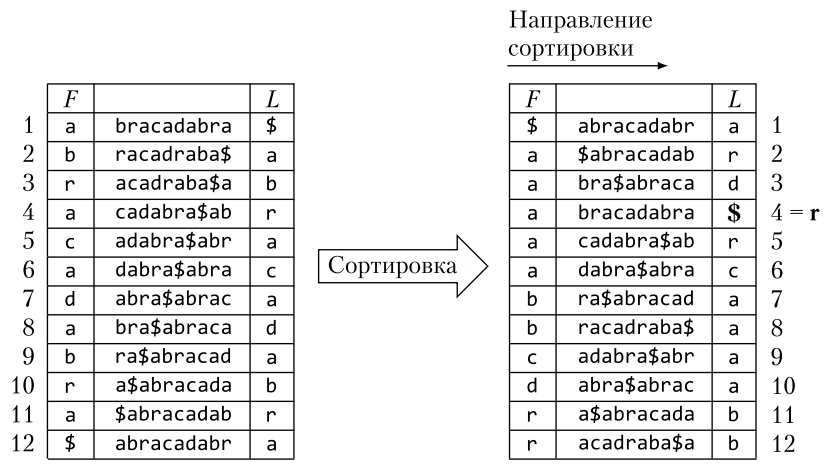
Альтернативная формулировка алгоритма, используемая реже, но упоминающаяся в [20], создает матрицу $\mathcal{M}'$ путем сортировки строк $\mathcal{M}$, читая их *справа налево* — от последнего символа каждой строки к первому. Затем сканированием первого столбца $\mathcal{M}'$ сверху вниз с пропуском символа \$ и сохранением его позиции в переменной r' формируется строка $\hat{F}$. В этом случае вывод равен $bw(S) = \left(\hat{F}, r'\right)$. Такая формулировка в некоторой степени эквивалентна предыдущей, поскольку можно формально доказать, что строки $\hat{F}$ и $\hat{L}$ демонстрируют одинаковые свойства *локальной однородности* и, следовательно, сжатия. Далее я буду называть сортировку строк $\mathcal{M}$ слева направо и выходные данные $\left(\hat{L}, r\right)$ преобразованием BWT для строки S.

Чтобы лучше понять эффективность преобразования Барроуза — Уилера, рассмотрим пример для строки S = `abracadabra`. Слева на рис. 14.1 изображена матрица $\mathcal{M}$, полученная циклическим сдвигом S, а справа — полученная сортировкой $\mathcal{M}$ матрица $\mathcal{M}'$. Символом \$ заканчивается только первый ряд $\mathcal{M}$, поэтому в первом ряду матрицы $\mathcal{M}'$ оказывается строка `$abracadabra`. Следующие три ряда $\mathcal{M}'$ займут строки, начинающиеся с `a`, затем последуют строки, начинающиеся с `b`, `c`, `d` и `r` соответственно.

Чтение первого столбца $\mathcal{M}'$, обозначенного F, дает строку `$aaaaabbcdr`, которая представляет собой упорядоченную последовательность всех символов входной строки S\$. Для получения $\hat{L}$ исключим единственное вхождение символа \$ из последнего столбца L, что даст нам $\hat{L}$ = `ardrcaaaabb`. Параметр r получит значение 4.

Этот пример иллюстрирует свойство локальной однородности, о котором я уже упоминал: последние шесть символов последнего столбца $\mathcal{M}$ образуют строку со множеством повторений `aaaabb`, легко и сильно сжимаемую двумя простыми компрессорами MTF + RLE (о них пойдет речь в разделе 14.2). Обоснованность этого утверждения будет математически подтверждена на следующих страницах, пока же отмечу только то, что такая повторяемость не случайна — она обусловлена способом сортировки строк $\mathcal{M}$ (слева направо) и тем, как пишутся тексты (слева

направо). Нужно отметить и такой положительный момент, как существование многочисленных реальных источников (называемых марковскими), генерирующих нетекстовые последовательности данных, которые преобразование Барроуза — Уилера также позволяет сделать локально однородными. Это дает возможность сильно сжимать их с помощью `bzip`-подобных компрессоров.



	F		L
1	a	bracadabra	$
2	b	racadraba$	a
3	r	acadraba$a	b
4	a	cadabra$ab	r
5	c	adabra$abr	a
6	a	dabra$abra	c
7	d	abra$abrac	a
8	a	bra$abraca	d
9	b	ra$abracad	a
10	r	a$abracada	b
11	a	$abracadab	r
12	$	abracadabr	a

F		L	
$	abracadabr	a	1
a	$abracadab	r	2
a	bra$abraca	d	3
a	bracadabra	**$**	4 = **r**
a	cadabra$ab	r	5
a	dabra$abra	c	6
b	ra$abracad	a	7
b	racadraba$	a	8
c	adabra$abr	a	9
d	abra$abrac	a	10
r	a$abracada	b	11
r	acadraba$a	b	12

Рис. 14.1. Прямое преобразование Барроуза — Уилера, примененное к строке S = abracadabra

14.1.2. Обратное преобразование

Как способ построения, так и рассмотренный пример демонстрируют перестановку $S\$$ в каждом столбце матрицы циклического сдвига $\mathcal{M}'$ (а также $\mathcal{M}$). В частности, упорядоченный по алфавиту первый столбец F = `$aaaaabbcdr` становится наиболее сжимаемым преобразованием входной строки $S\$$. К сожалению, строку F использовать как преобразование Барроуза — Уилера не получится из-за ее необратимости, ведь любой текст длиной 10, состоящий из пяти вхождений символа `a`, двух вхождений `b` и одного вхождения каждого из символов `c`, `d`, `r`, после BWT даст точно такой же столбец F. В некотором смысле преобразование Барроуза — Уилера обеспечивает для входной строки лучший вариант столбца $\mathcal{M}'$ с точки зрения его обратимости и сжимаемости.

Для более формального доказательства этого свойства используем полезную функцию, ищущую в матрице $\mathcal{M}'$ символ, предшествующий символу с заданным индексом в S.

Факт 14.1. Обозначим суффикс S (возможно, пустой), предшествующий строке i матрицы $\mathcal{M}'$, как $S[k_i, n]$ для всех $1 \leqslant i \leqslant n + 1$. Очевидно, что из-за циклического сдвига строк матрицы $\mathcal{M}'$ влево в строке i за этим суффиксом следует символ \$, а затем префикс $S[1, k_i - 1]$ (возможно, пустой).

Например, третий ряд матрицы $\mathcal{M}'$ (см. рис. 14.1) начинается с сочетания символов `abra`, за которым следует символ `$`, а затем `abracad`.

Свойство 1. В строке S символ $L[i]$ предшествует символу $F[i]$, за исключением ряда i, который заканчивается на `$`, то есть $L[i] =$ `$`. В этом случае $F[i] = S[1]$.

Доказательство. В силу факта 14.1 последний символ строки i — это $L[i] = S[k_i - 1]$, а ее первый символ — это $F[i] = S[k_i]$. Свойство доказано. ■

Интуитивно понятно, что это свойство вытекает из природы каждого ряда матриц $\mathcal{M}$ и $\mathcal{M}'$, ведь они получены циклическим сдвигом *влево* строки S`$`. Поэтому в каждом ряду за правым крайним символом (в столбце L) сразу же следует крайний левый символ из строки S (в столбце F). Следующее свойство является ключевым для проектирования обратного преобразования.

Свойство 2. Все вхождения одного символа `c` в L сохраняют тот же относительный порядок, что и в F. Это означает, что если в L этот символ находится в положении k, то и в F он окажется в положении k.

Доказательство. Обозначим $t \prec t'$ тот факт, что строка t лексикографически предшествует строке t'. Зафиксируем положение символа `c` во входной строке S. Для однократного вхождения `c` свойство очевидно, потому что единственное вхождение `c` в F отображается на единственное вхождение `c` в L. (Оба столбца являются перестановками S.)

Для более сложной ситуации, когда `c` встречается в S по крайней мере дважды, зафиксируем его положения в F, скажем $F[i]$ и $F[j]$, где $i < j$, и выберем соответствующие ряды в отсортированной матрице M'. Обозначим их $r(i)$ и $r(j)$. Обратите внимание на несколько интересных вещей.

- Ряд $r(i)$ лексикографически предшествует ряду $r(j)$. Это обусловлено порядком рядов в M' и предполагаемым неравенством $i < j$.
- В соответствии с нашим предположением оба ряда, $r(i)$ и $r(j)$, начинаются с символа `c`.
- В случае $r(i) = c\alpha$ и $r(j) = c\beta$ имеем $\alpha \prec \beta$.

Поскольку нас интересуют позиции этих двух вхождений c после отображения в L, рассмотрим ряды $r(i')$ и $r(j')$, полученные сдвигом $r(i)$ и $r(j)$ влево на один символ, — $r(i') = \alpha c$ и $r(j') = \beta c$. Этот циклический сдвиг вызывает переход первого символа $F[i]$ (соответственно, $F[j']$) в последний символ $L[i']$ (соответственно, $L[j']$). Предполагается, что $\alpha \prec \beta$, поэтому имеем $r(i') \prec r(j')$, то есть в L эта пара вхождений символа `c` сохраняет свой порядок. Сохранение порядка выполняется для каждой пары вхождений `c` в F и L, а значит, оно выполняется для всех вхождений. ■

Теперь у нас есть все математические инструменты для разработки алгоритма, который восстанавливает строку S из преобразования $bw(S) = \left(\hat{L}, r\right)$, используя следующее *LF*-отображение.

Определение 14.1. $LF[1, n + 1]$ — это массив из $n + 1$ целых чисел в диапазоне $[1, n + 1]$, такой, что $LF[i] = j$ тогда и только тогда, когда символ $L[i]$ отображается в символ $F[j]$. Таким образом, если $L[i]$ — это k-е вхождение символа `c` в L, то $F[LF[i]]$ — это k-е вхождение `c` в F.

Для символов, которые встречаются только один раз, таких как символы `$`, `c` и `d` в примере S = `abracadabra$` (см. рис. 14.1), массив LF строится довольно просто. А вот для символов `a`, `b` и `r`, которые фигурируют в строке S несколько раз, эффективное вычисление LF превращается в нетривиальный процесс. Благодаря свойству 2 это можно сделать за оптимальное время $O(n)$, как подробно описано в алгоритме 14.1. Там используется вспомогательный вектор C размером $|\Sigma| + 1$. Единица появилась из-за добавления к строке S символа `$`. Для простоты предполагается, что индексами массива C служат не целые числа, а *символы*[1].

Алгоритм 14.1. Построение *LF*-отображения из столбца *L*

```
1:  for i = 1, ..., n + 1 do
2:      C[L[i]]++;
3:  end for
4:  temp = 0, sum = 1;
5:  for i = 1, ..., |Σ| + 1 do
6:      temp = C[i];
7:      C[i] = sum;
8:      sum + = temp;
9:  end for
10: for i = 1, ..., n do
11:     LF[i] = C[L[i]];
12:     C[L[i]]]++;
13: end for
```

Первый цикл `for` алгоритма 14.1 вычисляет для каждого c параметр n_c — число вхождений этого символа в L и, таким образом, устанавливает $C[c] = n_c$ (предполагается, что изначально записи C равны нулю, а их индексы задаются рангом символов). Второй цикл `for` превращает вхождения отдельных символов в кумулятивную сумму, так что новое значение элемента $C[c]$ указывает увеличенное на единицу общее число вхождений в L символов, *меньших* чем c, а именно $C[c] = 1 + \sum_{x<c} n_x$. Поскольку для этого были введены две вспомогательные переменные `temp` и `sum`,

[1] Достаточно реализовать C как хеш-таблицу или вспомнить, что любой символ можно закодировать с помощью целого числа (код ASCII соответствует диапазону 0... 255), которое и послужит индексом в C.

общее рабочее пространство осталось равным $O(n)$. Обратите внимание на то, что после шага 7, согласно значению $C[c]$, символ `c` занимает в F первую позицию. Соответственно, перед началом последнего цикла `for` значение $C[c]$ показывает, куда именно в F перейдет первый символ `c` из L (это и есть LF-отображение для первого вхождения каждого символа алфавита). Последний цикл `for` сканирует столбец L и при каждом обнаружении символа $L[i] = c$ выполняет присваивание $LF[i] = C[c]$. Это верно, когда c встречается впервые, затем происходит приращение $C[c]$ в строке 12, и следующее вхождение `c` в L отображается на следующую позицию в F, ведь в F все начинающиеся с этого символа строки будут смежными. Таким образом, алгоритм сохраняет инвариант $LF[c] = \sum_{x<c} n_x + k$ после обработки $k - 1$ вхождений `c` в L. Легко сделать вывод, что временная сложность этого вычисления $O(n)$.

Такое LF-отображение и приведенные ранее основные свойства позволяют восстановить S из вывода $bw(S) = bw(S) = \left(\hat{L}, r\right)$. Временная и пространственная сложность такого преобразования $O(n)$. Очевидно, что сконструировать L из $bw(S)$ очень легко — достаточно вставить \$ в позицию $\hat{L}$. Алгоритм выбирает последний символ S, а именно $S[n]$, который легко идентифицируется в $L[1]$, так как первый ряд M' — это $\$S$. Затем по одному за итерацию он смещает символы строки S влево, используя два уже знакомых нам свойства. Свойство 2 позволяет сопоставить текущий символ из L (первоначально $L[1]$) с соответствующей ему копией в F, а свойство 1 — найти предшествующий этой копии символ. Для этого берется символ в конце того же ряда (то есть тот же, что и в L). Возвращающий алгоритмический фокус столбцу L двойной шаг инициирует в S сдвиги на один символ влево. За $n - 1$ итераций происходит восстановление исходной входной строки S. Псевдокод такого преобразования показан в алгоритме 14.2.

Алгоритм 14.2. Реконструкция S из $bw(S)$

```
1: Получение столбца L из bw(S);
2: Вычисление LF[1, n + 1] из L; // по алгоритму 14.1
3: k = 1; i = n;
4: while i > 0 do
5:     S[i] = L[k];
6:     k = LF[k];
7:     i--;
8: end while
```

В качестве примера снова рассмотрим рис. 14.1, где $L[1] = S[n] =$ `a`, и выполним цикл `while` алгоритма 14.2. Согласно определению 14.1 $LF[1]$ указывает на первый ряд, начинающийся с `a`, то есть ряд 2. Эта копия символа `a` в результате LF-отображения оказывается в $F[2]$ (и действительно, $F[2] =$ `a`), соответственно, в строке S ей предшествует символ $L[2] =$ `r`. Две операции повторяются до тех пор, пока не будет восстановлена вся строка S. Если продолжить рассмотрение этого

примера, то $L[2]$ = `r` после LF-отображения в F оказывается в позиции $LF[2] = 11$ (и действительно, $F[11]$ = `r`). Фактически $L[2]$ и $F[11]$ являются первым вхождением символа `r` в столбцах L и F соответственно. После этого алгоритм рассматривает символ $L[11]$ = `b`, который в строке S предшествует символу `r`, и т. д.

Теорема 14.1. *Восстановление исходной входной строки S из ее BWT требует времени и пространства $O(n)$. Алгоритм 14.2 может вызывать по одному промаху кэша на каждый символ.*

В последнее время проблеме уменьшения количества промахов кэша, а также сокращения рабочего пространства алгоритмов, инвертирующих BWT, был посвящен ряд исследований, и в литературе есть данные о некоторых успехах (см., например, [18], [13], [14], [12]). Но пока достижения не очень значительны. Так, удалось получить небольшие константы для промахов кэша, которые увеличиваются в случае часто повторяющихся данных. Так что работы в этой области еще много!

14.2. Два простых преобразования

Рассмотрим подробно два простых алгоритма, которые использовались при проектировании компрессора `bzip2`: *перемещение к началу* (move-to-front, MTF) и *кодирование длин серий* (run-length encoding, RLE). Первый отображает символы в целые числа, а второй — последовательности одинаковых символов в пары вида ⟨символ, целое число⟩. Для полноты картины отмечу, что RLE может уменьшить длину входной последовательности с большими наборами одинаковых символов, то есть в некотором смысле он работает как компрессор. А MTF можно превратить в компрессор, преобразуя выходные целые числа соответствующими кодировщиками для символов переменной длины. Впрочем, производительность сжатия в обоих случаях очень плохая, так что без волшебного преобразования BWT просто не обойтись!

14.2.1. Преобразование MTF

В обсуждении преобразования MTF в [3] каждый символ строки S заменяется его индексом из соответствующего *динамического* списка L^{MTF}, содержащего все символы алфавита. Выходная строка S^{MTF} инициализируется пустой строкой и заполняется целыми числами из диапазона $[1, |\Sigma|]$. На каждом шаге i ищется позиция p символа $S[i]$ в $\mathcal{L}^{MTF}$ (отсчет начинается с 1). Эта позиция p добавляется в строку S^{MTF}, а список $\mathcal{L}^{MTF}$ редактируется *перемещением* символа $S[i]$ в его *начало*.

Такая обработка выгодна для столбца L преобразования $bw(S)$, потому что, как вы скоро увидите, в результате *локально однородные* подстроки L превращаются *в глобально однородную* строку L^{MTF}, состоящую преимущественно из небольших целых чисел. После этого можно применить любой компрессор целых чисел, описанный в главе 11, или, как это делает `bzip`, использовать структурные свойства L^{MTF} для последовательного преобразования RLE и, наконец, статистического кодировщика,

например алгоритма Хаффмана, арифметического кодирования или их вариантов (см. главу 12).

Преобразование MTF для строки S = bananacocco, составленной из пяти символов $\{a, b, c, n, o\}$, демонстрирует рис. 14.2. Набор индексов такой строки в списке $\mathcal{L}^{\text{MTF}}$ будет выглядеть так: {1, 2, 3, 4, 5}. Очевидно, что часто встречающиеся символы попадают в начало списка $\mathcal{L}^{\text{MTF}}$ и получают меньшие индексы в S^{MTF}. В [3] этот принцип использовался для доказательства границ сжимаемости компрессора, который применяет γ-кодирование к целым числам из S^{MTF} (см. теорему 14.3).

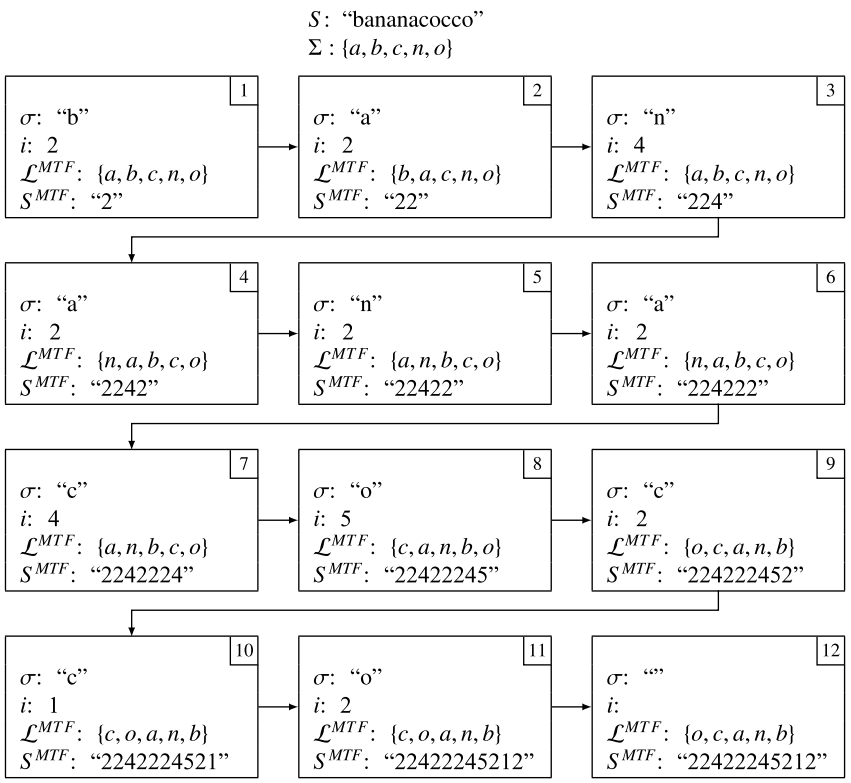


Рис. 14.2. Пример преобразования MTF для строки S = bananacocco, составленной из символов алфавита $\Sigma = \{a, b, c, n, o\}$, с набором индексов {1, 2, 3, 4, 5} из списка $\mathcal{L}^{\text{MTF}}$

В S выделяются две локальные однородные подстроки — anana и cocco с избыточностью в символах $\{a, n\}$ и $\{c, o\}$. Отображение MTF преобразует такие однородные подстроки в подстроки S^{MTF}, состоящие из небольших целых чисел. В силу локальной однородности столбца L в преобразовании $bw(S)$ список $\mathcal{L}^{\text{MTF}}$ получится полным единиц, так что применение *простого компрессора* RLE в данном случае оправданно и эффективно.

Строка S^{MTF} легко инвертируется, если начать с того же исходного списка $\mathcal{L}^{\text{MTF}}$, который использовался для преобразования MTF строки S. Следовательно, исходный список $\mathcal{L}^{\text{MTF}}$ должен быть частью заголовка сжатого файла, созданного преобразованием MTF. Пример инверсии представлен на рис. 14.3. Каждое целое число i из строки S^{MTF} алгоритм преобразует в символ, расположенный в позиции i списка $\mathcal{L}^{\text{MTF}}$, а затем перемещает этот символ в начало списка. Фактически он имитирует алгоритм преобразования, сохраняя синхронизацию обоих списков MTF.

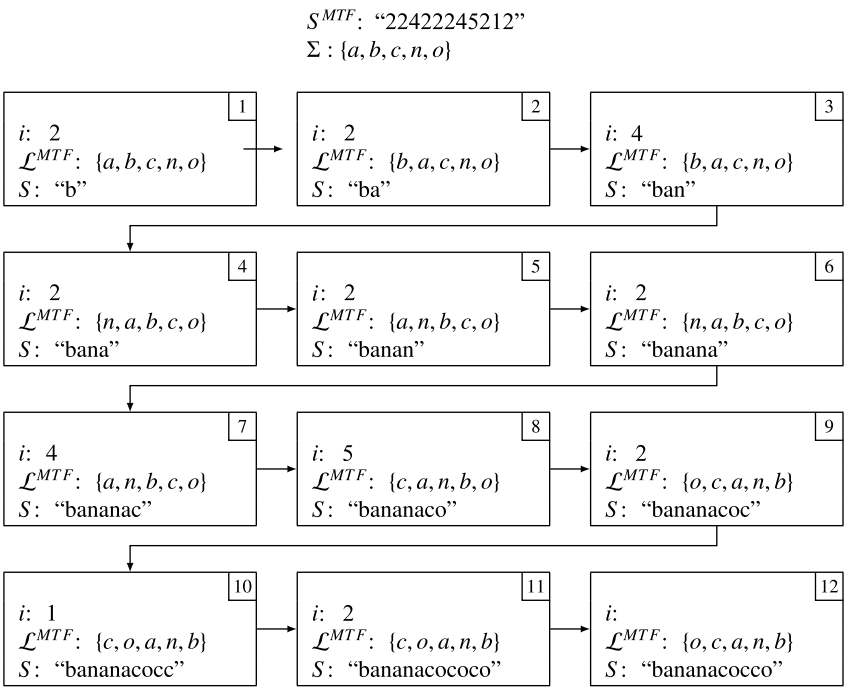


Рис. 14.3. Пример обратного преобразования MTF для строки S^{MTF} = 22422245212, начиная со списка $\mathcal{L}^{\text{MTF}}$ = {*a*, *b*, *c*, *n*, *o*}

Теорема 14.2. *Преобразование MTF строки S требует времени* $O(|S|)$ *и рабочего пространства* $O(|\Sigma|)$.

Ключевым понятием для оценки производительности сжатия MTF является так называемая *локальность ссылок*, которая ранее называлась *локально однородными подстроками*. Локальность ссылок в стоке S означает, что расстояние между последовательными вхождениями одного и того же символа мало. Например, строка bananacocco демонстрирует эту особенность в подстроках anana и cocco. Понятно, что это всего лишь приблизительное определение концепции, но пока будем придерживаться этой интуитивной формулировки.

При локальности ссылок во входной строке S MTF-компрессор (он выполняет преобразование MTF для строки S, а затем сжимает целые числа в S^{MTF}) работает лучше, чем алгоритм Хаффмана. Это может показаться удивительным, потому что, как было продемонстрировано в главе 12, алгоритм Хаффмана представляет собой *оптимальный префиксный код*. На самом деле ничего удивительного тут нет, ведь MTF-компрессор не является префиксным кодом, потому что любой символ может быть *динамически* связан с различными кодовыми словами. Обратите внимание на то, что на рис. 14.2 символу c в строке S^{MTF} соответствуют три разных числа — 4, 2, 1, а значит, и три разных кодовых слова.

Лемма 14.1. *Компрессор на базе преобразования MTF и γ-кода может быть лучше алгоритма Хаффмана за счет неограниченного множителя* $\Omega(\log n)$, *где n — длина сжимаемой строки.*

Доказательство. Возьмем строку $S = 1^n 2^n ... n^n$ длиной n^2, определенную для целочисленного алфавита размером n. Так как каждый символ встречается n раз, распределение равномерно и на один символ алгоритму Хаффмана требуется $\Theta(\log_2 n)$ бит. В этом случае общее сжатие строки S алгоритмом Хаффмана потребует $\Theta(|S| \log n) = \Theta(n^2 \log n)$ бит.

Применив к строке S преобразование MTF, получим строку $S^{\text{MTF}} = 1^n 2 1^{n-1} 3 1^{n-1} 4 ...$ Сжатие этой строки γ-кодом даст битовую последовательность длиной $O(n^2 + n \log n)$. Это связано с тем, что $\Theta(n^2)$ целых чисел, равных 1, кодируются как $\gamma(1) = 1$, занимая один бит, тогда как все остальные целые числа (их $n - 1$, а их значение не превышает n) при кодировании занимают $O(\log n)$ бит каждое. ■

Для входной строки S, не демонстрирующей локальности ссылок, например квазислучайной строки из символов алфавита Σ, MTF-компрессор работает хуже алгоритма Хаффмана, но ненамного. Теорема 14.3 формализует этот грубый и интуитивный анализ, объединяя преобразование MTF с γ-кодом. Само собой разумеется, что заменой γ-кода на δ-код или любой другой лучший универсальный компрессор для целых чисел верхнюю границу в теореме можно сделать ближе к эмпирической энтропии $\mathcal{H}_0$ (0-го порядка) строки S.

Теорема 14.3. *Пусть* n_c *— число вхождений символа c во входную строку S общей длиной n, а* $\rho_{\text{MTF}}(S)$ *— среднее число битов на символ, которое требуется компрессору, сжимающему строку* S^{MTF} *с помощью γ-кода. Поскольку* $\rho_{\text{MTF}}(S) \leqslant 2\mathcal{H}_0 + 1$, *можно заключить, что этот компрессор хуже алгоритма Хаффмана не более чем в два раза.*

Доказательство. Пусть $p_1, \ldots, p_{n_c}$ — позиции всех вхождений символа c в строку S. Очевидно, что между любыми двумя последовательными вхождениями, скажем p_{i-1} и p_i, может располагаться не более чем $p_i - p_{i-1}$ различных символов, включая сам символ c. Таким образом, максимальное значение индекса, который даст MTF-компрессор c в позиции p_i, не превышает $p_i - p_{i-1}$. При обработке позиции p_{i-1} символ c перемещается в начало списка — на позицию 1, затем при обработке каждого следующего символа он будет сдвигаться на одну позицию назад, пока не окажется на

позиции p_i. Это означает, что целое число, генерируемое для символа c в позиции p_i, не превышает $p_i - p_{i-1}$. Обработка этого целого числа γ-кодом требует пространства не более чем $|\gamma(p_i - p_{i-1})| \leqslant 2(\log_2 (p_i - p_{i-1})) + 1$ бит. Для первого вхождения c можно предположить, что $p_0 = 0$, закодировав его с использованием максимум $|\gamma(p_i)| \leqslant 2(\log_2 p_i) + 1$ бит. В целом пространство для хранения вхождений символа c в строку S (в битах) составит:

$$\leqslant \left|\gamma(p_1)\right| + \sum_{i=2}^{n_c} \left|\gamma(p_i - p_{i-1})\right| \leqslant$$
$$\leqslant 2\log_2(p_1) + 1 + \sum_{i=2}^{n_c} \left(2\log_2(p_i - p_{i-1}) + 1\right) =$$
$$= \sum_{i=1}^{n_c} \left(2\log_2(p_i - p_{i-1}) + 1\right).$$

Согласно неравенству Йенсена функцию логарифма можно вынести за пределы знака суммы и, усреднив ее аргументы, получить в результате телескопический ряд:

$$\leqslant n_c \left(2\log_2\left(\frac{1}{n_c}\left(\sum_{i=1}^{n_c}(p_i - p_{i-1})\right)\right) + 1\right) \leqslant$$
$$\leqslant n_c \left(2\log_2\left(\frac{n}{n_c}\right) + 1\right),$$

где последнее неравенство вытекает из того, что p_{n_c} — позиция последнего вхождения символа c в строку S — не может превышать длину этой строки n. Суммировав по всем символам $c \in \Sigma$ и разделив на длину строки n, так как $\rho_{\text{MTF}}(S)$ — это число битов на один символ в S, получим:

$$\rho_{\text{MTF}}(s) \leqslant 2\left(\sum_{c\in\Sigma} \frac{n_c}{n} \log_2\left(\frac{n}{n_c}\right)\right) + 1 = 2\mathcal{H}_0 + 1.$$

Этот тезис следует из того, что энтропия $\mathcal{H}_0$ ограничивает снизу среднюю длину кодового слова кода Хаффмана. ■

14.2.2. Преобразование RLE

Рассмотрим очень простое строковое преобразование, сопоставляющее каждой максимальной непрерывной подстроке l вхождений символа c пару $\langle \ell, c \rangle$. Предположим, что требуется сжать строку пикселов монохромного растрового изображения, где W соответствует белому, а B — черному цвету:

$$WWWWWWWWWWWBWWWWWWWWWWWWBBBBBWWWWWW.$$

Первый блок W можно сжать следующим образом:

$$\underbrace{WWWWWWWWWWW}_{11,\,W} BWWWWWWWWWWWWBBBBBWWWWWW.$$

Процесс продолжается до конца строки, что дает последовательность пар $\langle 11, W\rangle$, $\langle 1, B\rangle$, $\langle 12, W\rangle$, $\langle 5, B\rangle$, $\langle 6, W\rangle$. Легко заметить, что это кодирование без потерь, из которого очень просто восстанавливается исходная строка. Примечательно, что при $|\Sigma| = 2$, как в нашем примере, во входной строке будут чередующиеся максимальные наборы W и B, поэтому даже информации в виде длины набора и первого символа сжимаемой строки хватит для восстановления этой строки в исходное состояние. В рассматриваемом примере достаточно сгенерировать: W, 11, 1, 12, 5, 6.

На самом деле кодирование длин серий — это больше, чем преобразование, потому что, добавив к нему кодировщик целых чисел, можно получить простой компрессор, как было сделано для MTF. Чаще всего это используется при передаче факсов. Лист бумаги рассматривается как двоичное монохромное растровое изображение. Сначала к двум последовательным строкам пикселов применяется оператор `XOR`, после чего для каждой выходной строки выполняется преобразование RLE, а полученные целые числа сжимаются алгоритмом Хаффмана или арифметическим кодированием (напомню, что для черно-белых изображений размер алфавита равен двум). При условии, что факс отправляется на стандартной бумаге, оператор `XOR` дает строки, заполненные нулями, для которых RLE-преобразование генерирует несколько сильно сжимаемых последовательностей. Ничто не мешает поступить аналогичным образом с цветными изображениями, но после применения оператора `XOR` к смежным строкам нулей будет меньше, соответственно, в этом случае требуются более сложные подходы.

Кодирование длин серий может работать лучше или хуже алгоритма Хаффмана — все зависит от сжимаемой строки. Следующая лемма на примере строки, использовавшейся для доказательства леммы 14.1, показывает, что RLE может давать лучшие результаты, чем алгоритм Хаффмана.

Лемма 14.2. *Компрессор на базе сочетания RLE-преобразования и γ-кода может работать лучше алгоритма Хаффмана за счет неограниченного множителя* $\Omega(n)$, *где n — длина сжимаемой строки.*

Доказательство. Возьмем строку $S = 1^n 2^n \ldots n^n$ и вспомним из доказательства леммы 14.1, что для сжатия такой строки алгоритмом Хаффмана требуется пространство $\Theta(n^2 \log n)$ бит. Преобразование RLE строки S дает строку $S^{\text{RLE}} = \langle 1, n\rangle\langle 2, n\rangle\langle 3, n\rangle \ldots \langle n, n\rangle$. Применение γ-кода к целым числам строки S^{RLE} требует $O(\log n)$ бит на одну пару, то есть всего $O(n \log n)$ бит. ■

Конечно, бывают случаи, когда RLE-компрессор показывает намного худшие результаты, чем алгоритм Хаффмана, например, для строки S с очень короткими последовательностями одинаковых символов. Скажем, в любом английском тексте такие последовательности обычно имеют длину 1.

14.3. Компрессор bzip

Как я уже упоминал, основу компрессора `bzip` составляет последовательное сочетание трех преобразований — BWT, MTF и RLE, выходные данные которой подходят для сильного сжатия классическим статистическим компрессором, таким как алгоритм Хаффмана, арифметическое кодирование или один из их вариантов. Самый трудоемкий этап в этой цепочке — вычисление/инверсия BWT во время сжатия/распаковки соответственно. Это связано не только с количеством операций, которое в целом оценивается как $\Theta(n)$, но и с сильно фрагментированным шаблоном доступа к памяти, вызывающим многочисленные промахи кэша.

Хорошую работу компрессора `bzip` обеспечивает локальная однородность строки, получаемой после преобразования Барроуза — Уилера. Сейчас я обрисую общую картину и направление действий, а математические нюансы оставим до раздела 14.4. Рассмотрим входную строку S и одну из ее подстрок w, имеющую n_w вхождений. Пусть $c_1, \ldots, c_{n_w}$ — предшествующие этим вхождениям подстроки w символы. Учитывая способ построения $bw(S)$, можно прийти к выводу, что все ряды с префиксом w в M' (их n_w) будут смежными, но, возможно, перетасованными относительно их позиций в S в соответствии с символами, которые следуют за w в каждом ряду. В любом случае символы c_i, предшествующие w, являются смежными в столбце L (соответствующим образом перетасованными) и, таким образом, составляют подстроку L. В случае марковской строки S, то есть строки, в которой генерация новых символов происходит на основе предыдущих, как в лингвистических текстах, ожидается, что все символы c_i будут различными, причем это свойство сохраняется при росте длины префикса w. Благодаря именно такой однородности все последующие шаги в компрессоре `bzip` обеспечивают крайне эффективное сжатие L.

Для наглядности рассмотрим работу компрессора `bzip` со строкой S, определенной как повторенная три раза строка `mississippi`, то есть имеющей высокую степень повторяемости. Первым делом вычисляется $bw(S)$. Для экономии места я опущу детали этого вычисления, сразу показав результат, который можно проверить вручную: L = `ippp ssss ssmm miip ppii isss sssi iiii i`. Для простоты чтения все символы сгруппированы в четверки, а $r = 16$ (считая от 1). Теперь к L применяется преобразование MTF, начинающееся со списка $\mathcal{L}^{\text{MTF}} = \{i, m, p, s\}$. Как r, занимающее 4–8 байт, так и $\mathcal{L}^{\text{MTF}}$ в явном виде хранятся в заголовке сжатого файла. Преобразование MTF дает строку:

$$L^{\text{MTF}} = \texttt{1311 4111 1141 1414 1121 1411 1112 1111 1}.$$

Обратите внимание на то, что последовательности одинаковых символов порождают наборы единиц, за исключением первого символа каждой последовательности,

с которым сопоставляется целое число, представляющее его положение в L^{MTF} на момент его обработки.

Первая особенность компрессора `bzip` заключается в том, что RLE не применяется к последовательностям всех возможных символов. Вместо него используется ограниченный вариант RLE1, который сжимает только наборы единиц. Таким образом, L^{MTF} можно рассматривать как эффективный способ резервирования символов (целых чисел) 0 и 1 для двоичного кодирования длин серий, состоящих из единиц. Например, последовательность `11111` кодируется по схеме, известной как код Уилера: ее длина увеличивается на 1 (5 + 1 = 6), затем из двоичного представления 6 (`110`) удаляется первый бит, что дает двоичную последовательность `10`. Первое приращение гарантирует, что в увеличенном виде длина последовательности будет не менее 2, то есть представлена по крайней мере двумя двоичными цифрами, из которых первая обязательно 1. В этом случае удаление первого бита оставляет не менее одного бита для вывода. Расшифровать код Уилера легко — достаточно проделать эти шаги в обратном порядке.

Ключевое свойство кода Уилера состоит в том, что в выходной битовой последовательности цифр не больше, чем символов в L^{MTF}, поэтому все это можно рассматривать как предварительное сжатие, эффективность которого растет по мере увеличения длины последовательностей единиц в L^{MTF}. Для нашего примера двоичный вывод будет выглядеть так:

`RLE1` = `0314 1041 4031 4141 0210`.

Очевидно, что декодер легко распознает кодировки выходных последовательностей, поскольку они состоят из максимальных наборов нулей и единиц. Напомню, что эти числа зарезервированы для данной цели.

В конце результат работы RLE1 сжимается статистическим компрессором, который работает с алфавитом, состоящим из целых чисел в диапазоне $[1, |\Sigma| + 1]$. Более подробную информацию можно найти на странице компрессора `bzip2` [19][1], особенно относительно статистического кодирования и проверки эффективности сжатия на эталонном тесте Squash[2], который дает доступ к множеству компрессоров, наборов данных и машин для максимально широкого сравнения.

Непосредственный вывод из сказанного о практической производительности компрессоров, с которыми вы познакомились на этих страницах, будет таким: компрессоры на основе LZ быстрее всего выполняют сжатие и декомпрессию из-за их алгоритмической структуры, благоприятной для кэш-памяти. Еще они

---

[1] Там позиции отсчитываются от 0, соответственно, компрессор RLE работает с наборами 0, а все остальные числа увеличиваются на 1, чтобы символы 0 и 1 по-прежнему были зарезервированы для кодирования длин серий.

[2] См.: https://quixdb.github.io/squash-benchmark/.

позволяют достичь очень интересных коэффициентов сжатия. Компрессоры же на основе BWT довольно медленные, что вызвано алгоритмической структурой этого преобразования, но они дают значительные коэффициенты сжатия. Поскольку вычисление $bw(S)$ требует больших затрат, его реализации делят входной файл на блоки, а затем *по очереди* преобразуют их. В случае компрессоров на основе словаря размер блока влияет на компромиссное соотношение между коэффициентом сжатия и скоростью сжатия, но при увеличении длины блоков скорость декомпрессии снижается. Текущая реализация `bzip2` позволяет указывать размер сжимаемого блока с помощью параметров командной строки –1... –9, что соответствует блокам размером 100... 900 Кбайт.

Остается проблема прямого преобразования Барроуза — Уилера. Явно построить матрицу вращения $\mathcal{M}$, а тем более ее отсортированную версию $\mathcal{M}'$ нельзя, потому что для последовательности длиной n это потребует рабочего пространства $\Theta(n^2)$. Вот почему большинство компрессоров на основе BWT применяют трюки, позволяющие обойтись без построения этих матриц. Один из таких трюков включает использование массивов суффиксов, которые были описаны в главе 10, где подробно рассматривались алгоритмы их эффективного построения. Преобразование BWT задействует один из этих алгоритмов[1]. Публикации на тему BWT подстегнули интерес к литературе о проблеме построения массива суффиксов (см., например, [1], [15], [17]).

Чтобы лучше понять связь между массивами суффиксов и BWT, рассмотрим пример. Вычислим массив суффиксов SA = [12, 11, 8, 1, 4, 6, 2, 5, 7, 10, 3] для строки S = `abracadabra$`. Эти структуры данных показаны на рис. 14.4. Первые четыре столбца содержат суффиксы строки S и ее массив суффиксов SA. В пятом столбце находится соответствующая упорядоченная матрица вращения M', а ее последний столбец — L. Из-за символа-ограничителя $ сортировка суффиксов эквивалентна сортировке строк матрицы M. Вы можете самостоятельно проверить, что приведенная далее формула связывает SA с L:

$$L[i] = \begin{cases} S[SA[i]-1] \text{ при } SA[i] \neq 1; \\ \$ \text{ в противном случае.} \end{cases}$$

Известно, что в строке S каждый символ $L[i]$ предшествует символу $F[i]$ (см. свойство 1). Символ $F[i]$ стоит первым в ряду i матрицы M', то есть это первый символ суффикса, начинающегося с $SA[i]$. Поэтому можно заключить, что $L[i]$ равен символу

[1] М. Барроуз: «...я заручился его помощью, чтобы эффективно реализовать этап сортировки, для чего требовалось учитывать как постоянные множители, так и асимптотическое поведение. Мы пробовали разные подходы, только часть из которых попала в статью, но поставленные цели были достигнуты: мы показали, что алгоритм можно сделать достаточно быстрым для того, чтобы его можно было использовать на современных машинах...» [10].

строки S, который предшествует позиции $SA[i]$. Особым случаем является ряд, соответствующий строке S и, следовательно, находящийся перед первым суффиксом, для которого предшествующим символом будет \$. Таким образом, при наличии массива суффиксов строки S вывод строки L по этой формуле будет происходить за линейное время. Мы доказали следующую теорему.

Теорема 14.4. *Для входной строки S преобразование $bw(S)$ имеет такую же временную сложность и сложность ввода-вывода, как и построение массива суффиксов. Общие затраты на преобразование $bw(S)$ с использованием алгоритма DC3 из главы 10 оптимальны в нескольких моделях вычислений. В частности, они составляют $O(n)$ в RAM-модели и $O(\texttt{Sort}(n))$ в двухуровневой модели памяти, где $\texttt{Sort}(n)$ — это оценка количества операций ввода-вывода при сортировке n атомарных элементов.*

Суффиксы	Индекс	Отсортированные суффиксы	Индекс	$\mathcal{M}'$	L
abracadabra\$	1	\$	12	\$abracadabra	a
bracadabra\$	2	a\$	11	a\$abracadabr	r
racadabra\$	3	abra\$	8	abra\$abracad	d
acadabra\$	4	abracadabra\$	1	abracadabra\$	\$
cadabra\$	5	acadabra\$	4	acadabra\$abr	r
adabra\$	6	adabra\$	6	adabra\$abrac	c
dabra\$	7	bra\$	9	bra\$abracada	a
abra\$	8	bracadabra\$	2	bracadabra\$a	a
bra\$	9	cadabra\$	5	cadabra\$abra	a
ra\$	10	dabra\$	7	dabra\$abraca	a
a\$	11	ra\$	10	ra\$abracadab	b
\$	12	racadabra\$	3	racadabra\$ab	b

Рис. 14.4. Сравнение массива суффиксов и отсортированной матрицы вращения $\mathcal{M}'$ для строки S = abracadabra \$. В столбец L копируется последний символ каждого ряда матрицы $\mathcal{M}'$

14.4. Повышение эффективности сжатия∞

Для начала вспомним, что эмпирическая энтропия порядка k — это мера неопределенности (или информации), связанная со строкой S, составленной из символов алфавита $\Sigma = \{\sigma_1 \ldots \sigma_h\}$. Пусть $n\omega$ — количество вхождений подстроки ω в S. Обозначение $\omega \in \Sigma^k$ указывает, что длина ω равна k. В разделе 13.4 была получена формула:

$$\mathcal{H}_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma^k} \left(\sum_{i=1}^{h} n_{\omega\sigma_i} \log\left(\frac{n_\omega}{n_{\omega\sigma_i}} \right) \right).$$

При $k = 0$ речь идет о классической эмпирической энтропии нулевого порядка, которая вычисляется относительно частот отдельных символов в S, то есть без какого-либо контекста длиной k. Очевидно, что $\mathcal{H}_k(S) \leqslant \mathcal{H}_0(S)$, но может быть и намного меньше, и с ростом $|S|$ и k это значение сходится к энтропии генерирующего строку S источника.

Эта формула подсказывает способ разработки компрессора, который достигает энтропии $\mathcal{H}_k(S)$, причем начиная с компрессора, который достигает $\mathcal{H}_0(S)$ своей входной строки, например арифметического кодирования или алгоритма Хаффмана. Алгоритм такого типа называется *усилителем сжатия*, потому что позволяет сдвигать границу производительности сжатия с $\mathcal{H}_0$ до $\mathcal{H}_k$. Как ни странно, такой переход обеспечивается преобразованием Барроуза — Уилера [7].

Чтобы проиллюстрировать эту мощную инновационную идею, рассмотрим обобщенный статистический компрессор нулевого порядка C_0, производительность которого на один символ строки S ограничена величиной $\mathcal{H}_0(S) + f(|S|)$ бит. Отмечу, что функция $f(|S|) = 2/|S|$ достигается при арифметическом кодировании, а для алгоритма Хаффмана она составляет $f(|S|) = 1$ (см. главу 12).

Чтобы превратить C_0 в эффективный компрессор k-го порядка C_k, поступим следующим образом:

- выполним для входной строки S преобразование Барроуза — Уилера $bw(S)$;
- для всех возможных подстрок ω строки S разобьем столбец L, сформировав подстроки L_ω, каждая из которых образована последними символами строк с префиксом ω;
- сожмем каждую подстроку L_ω компрессором C_0 и выполним конкатенацию выходных битовых последовательностей, отсортировав их в соответствии с алфавитным порядком ω (или, что эквивалентно, по вхождению L_ω в L).

Легко заметить, что L_ω — подстрока L, ведь в матрице $\mathcal{M}'$ строки с префиксом ω являются смежными. При наличии для строки S массива `lcp` разбиение L занимает линейное время (см. главу 10) и, соответственно, не влияет на эффективность конечного компрессора C_k. Для производительности сжатия в битах на символ легко вывести границу:

$$\frac{1}{|S|}\sum_{\omega\in\Sigma^k}|L_\omega|\left(\mathcal{H}_0\left(L_\omega\right)+f\left(|L_\omega|\right)\right)=\mathcal{H}_k\left(S\right)+O\left(|\Sigma|^k\right),$$

применив определение $\mathcal{H}_k(S)$ к суммированию $\mathcal{H}_0(L_\omega)$ и воспользовавшись тем, что для алгоритма Хаффмана и арифметического кодирования $f(|L_\omega|) \leqslant 1$. Очевидно, что чем эффективнее компрессор нулевого порядка, тем ближе он к $\mathcal{H}_0$ и тем меньше $f(|L_\omega|)$, что делает аддитивный член $O(|\Sigma|^k)$ пренебрежимо малым. В [7] показано, что фиксировать k не нужно, поскольку существует усилитель сжатия, за оптимальное время $O(|S|)$ определяющий раздел L и обеспечивающий более сильное сжатие,

чем достигается с помощью C_k для $k \geqslant 0$. Это элегантный и не слишком сложный алгоритм, но для экономии места я не буду приводить его здесь, а отошлю заинтересованного читателя к этой статье.

14.5. Сжатое индексирование∞

Я уже подчеркивал биективное соответствие между строками матрицы вращения $\mathcal{M}$ и суффиксами строки S, а также связь между строкой L и полученным на базе S массивом суффиксов (см. рис. 14.4). Это основа конструкции FM-индекса, который стал первым сжатым полнотекстовым индексом с эффективным поиском подстрок и занятостью пространства, ограниченной сверху эмпирической энтропией k-го порядка индексированной строки. Этот индекс можно рассматривать как *сжатую версию* массива суффиксов или сжатый `bzip`-файл *с функцией поиска*. Я не буду углубляться в детали реализации FM-индекса, а просто расскажу о его особенностях, сосредоточившись на основных алгоритмических идеях. Узнать дополнительные детали можно в исходной статье [8] и обзоре [16].

Для простоты изложения опишу три основные операции.

- `Count`(P) возвращает *диапазон рядов* [*first*, *last*] матрицы $\mathcal{M}'$ (и, соответственно, суффиксов в массиве суффиксов), префикс которых — это строка $P[1, p]$. Значение (*last* – *first* + 1) учитывает количество этих вхождений шаблона.
- `Locate`(P) возвращает список всех позиций в индексированной строке S, в которых встречается P (возможно, они не упорядочены).
- `Extract`(i, j) возвращает подстроку $S[i, j]$ из ее сжатого представления в FM-индексе.

Например, на рис. 14.4 у шаблона P = `ab` всего два вхождения, при этом *first* = 3 и *last* = 4. Как легко увидеть, эти два ряда соответствуют двум суффиксам, $S[1, 12]$ и $S[8, 12]$, которые имеют префикс P.

В функции `Count`(P) извлечение рядов *first* и *last* реализовано не с помощью бинарного поиска, как в массивах суффиксов, а особым методом поиска в столбце L. Еще используются массив C, который хранит в $C[c]$ позицию первого вхождения символа c в столбце F (см. шаг 7 алгоритма 14.1), и дополнительная структура данных, эффективно поддерживающая очень простую функцию `Rank`(c, k). Эта функция сообщает количество вхождений символа c в префикс строки $L[1, k]$. Отмечу, что массив C мал в том смысле, что его размер пропорционален количеству символов в алфавите. Строка L и структура данных, реализующая для этой строки функцию `Rank`(c, k), даже при хранении в сжатом виде способны эффективно поддерживать извлечение $L[i]$ и подсчет количества вхождений символа c. В литературе предлагается множество решений последней задачи (см., например, классические результаты в [8], [11], [2], [16]). Я упомяну некоторые из них, возможно, на момент написания этих строк уже не самые лучшие, ведь исследования в этой области идут очень активно.

Теорема 14.5. *Пусть $S[1, n]$ — строка, составленная из символов алфавита Σ, а L — ее BWT.*

- *Для $|\Sigma| = O(\texttt{polylog}(n))$ существует структура данных, поддерживающая запросы* `Rank` *к L за время $O(1)$ и требующая пространство $nH_k(S) + o(n)$ бит для любого $k = o(\log_{|\Sigma|} n)$, которая извлекает любой символ L за то же время.*
- *Для обобщенного алфавита Σ существует структура данных, поддерживающая запросы* `Rank` *к L за время $O(\log\log|\Sigma|)$ и требующая пространство $nH_k(S) + o(n\log|\Sigma|)$ бит для любого $k = o(\log_{|\Sigma|} n)$, которая извлекает любой символ L за то же время.*

Это означает, что функция `Rank` может быть реализована за постоянное или почти постоянное время в пространстве, очень близком к энтропии k-го порядка индексируемой строки S. Массиву C требуется пространство всего $O(|\Sigma|)$, что пренебрежимо мало для реальных алфавитов. То есть этот ансамбль структур данных действительно очень компактен.

Осталось показать, как этот ансамбль позволяет реализовать функцию `Count`(P). Псевдокод такой реализации показан в алгоритме 14.3, который называют алгоритмом *обратного поиска*. Он делает p итераций, пронумерованных от p до 1. На итерации i сохраняется следующий инвариант: *параметр first указывает на первый ряд отсортированной матрицы вращения $\mathcal{M}'$ с префиксом $P[i, p]$, а параметр last — на последнюю строку $\mathcal{M}'$ с префиксом того же суффикса $P[i, p]$*. Он изначально корректен в силу построения. $C[c]$ — первый ряд матрицы $\mathcal{M}'$, начинающийся с c, а $C[c + 1] - 1$ — последний ряд $\mathcal{M}'$, начинающийся с c (напомню, что нумерация строк начинается с 1). В качестве примера возьмем $P =$ `ab` и матрицу $\mathcal{M}'$, приведенную на рис. 14.4. Сначала $p = 2$, $P[2] = b$ и $C[b] = 7$ (учтены одно вхождение в строку S символа \$ и пять вхождений символа a), при этом $C[b + 1] = C[c] = 9$ (учтены еще два вхождения b в строку S). То есть перед началом обратного поиска $[7, 8]$ это корректный диапазон рядов, префикс которых представляет собой один символ $P[2] = b$.

Алгоритм 14.3. Подсчет вхождений шаблона $P[1, p]$ в S

```
1: i = p, c = P[p];
2: first = C[c], last = C[c + 1] − 1;
3: while (first ⩽ last and i > 1) do
4:     c = P[i − 1];
5:     first = C[c] + Rank(c, first − 1);
6:     last = C[c] + Rank(c, last) − 1;
7:     i = i − 1;
8: end while
9: return (first, last)
```

На каждой итерации алгоритм 14.3 находит диапазон рядов [*first*, *last*] с префиксом $P[i, p]$, а затем определяет новый диапазон [*first*, *last*] с префиксом $P[i-1, p] = P[i-1] \times P[i, p]$, который на один символ длиннее ранее обработанного шаблона суффикса. Это работает следующим образом. Сначала с помощью функции `Rank` определяются первое и последнее вхождения символа $c = P[i-1]$ в подстроку $L[first, last]$, в частности, $\texttt{Rank}(c, first-1)$ подсчитывает вхождения c до позиции *first* в L, а $\texttt{Rank}(c, last)$ — вхождения c до позиции *last* в L. Эти два значения позволяют определить, какие вхождения c включены в $L[first, last]$, и использовать их для вычисления *LF*-отображения первого и последнего вхождений c в этом диапазоне. Равенства, фигурирующие в шагах 5 и 6 алгоритма 14.3, взяты из свойства 2 и определения 14.1. Их эффективная с точки зрения временных и пространственных затрат реализация выполняется с помощью сжатой структуры данных для $\texttt{Rank}(c, k)$ (см. теорему 14.5).

Формальное доказательство того, что это отображение фактически извлекает новый диапазон строк [*first*, *last*] с префиксом $P[i-1, p]$, можно найти в оригинальной публикации [8]. А мы проверим корректность этого утверждения на примере. Снова обратимся к рис. 14.4 и шаблону $P = \texttt{ab}$. В этом случае диапазон рядов матрицы $\mathcal{M}'$ с префиксом в виде последнего символа шаблона $P[2] = \texttt{b}$ будет [7, 8] (напомню, что P обрабатывается в обратном порядке, отсюда и название — *обратный поиск*). Для предыдущего символа шаблона $P[1] = \texttt{a}$ алгоритм вычисляет $\texttt{Rank}(a, first-1) = \texttt{Rank}(a, 6) = 1$ и $\texttt{Rank}(a, last) = \texttt{Rank}(a, 8) = 3$, так как $L[1, first-1]$ содержит одно вхождение a, а $L[1, last]$ — три вхождения a. То есть новый диапазон определяется следующим образом: $first = C[a] + \texttt{Rank}(a, 6) = 2 + 1 = 3$, а $last = C[a] + \texttt{Rank}(a, 8) - 1 = 2 + 3 - 1 = 4$, что дает непрерывный диапазон [3, 4] строк с префиксом вида $P = \texttt{ab}$.

После заключительной итерации ($i = 1$) параметры *first* и *last* будут разграничивать ряды $\mathcal{M}'$, содержащие все суффиксы с префиксом P. Очевидно, что при $last < first$ шаблон P в индексированной строке S отсутствует. Сказанное подытоживает следующая теорема.

Теорема 14.6. *Для строки* $S[1, n]$, *составленной из символов алфавита* Σ, *существует сжатый индекс, которому для поддержки операции* $\texttt{Count}(P[1, p])$ *требуется время* $O(pt_{\texttt{rank}})$, *где* $t_{\texttt{rank}}$ *— это временные затраты на одну операцию* `Rank` *над преобразованием bw*(S). *Пространственные затраты ограничены* $n\mathcal{H}_k(S) + o(n \log |\Sigma|)$ *битами для любого* $k = o(\log_{|\Sigma|} n)$.

Что интересно, подключение сжатой структуры данных из теоремы 14.5, дает реализацию $\texttt{Count}(P)$ за оптимальное время $O(p)$ и сжатие пространства. Но такое решение несколько неэффективно по количеству операций ввода-вывода, поскольку на каждой итерации из-за требуемых алгоритмом 14.3 перемещений по L и операций `Rank` возможны до $\Theta(p)$ промахов кэша. В литературе описан ряд попыток сделать FM-индексы независимыми от размера кэша или оптимизировать их с учетом кэш-памяти, но найти столь же элегантное решение пока не получилось.

Теперь опишем реализацию Locate(P). Для фиксированного параметра μ из $\mathcal{M}'$ выбираются строки i, которые соответствуют суффиксам индексированной строки S. Позиции этих суффиксов в S описываются формулой $pos(i) = 1 + j\mu$, где $j = 0, 1, 2...$ Каждая пара $\langle i, \text{pos}(i)\rangle$ в явном виде хранится в структуре данных $\mathcal{P}$, которая поддерживает запросы членства за постоянное время (на первом компоненте ряда). Если строка S выбрана и, соответственно, индексирована в структуре данных $\mathcal{P}$, в ней можно быстро найти позицию $pos(r)$ для индекса ряда r. В противном случае алгоритм вычисляет $h = LFt(r)$, где t — итерация, на которой ряд h выбирается и индексируется в $\mathcal{P}$. В этом случае $pos(r) = pos(h) + t$, потому что каждое вычисление LF сдвигает нас по строке S на позицию назад. Метод выборки гарантирует, что для поиска ряда в структуре данных $\mathcal{P}$ требуется не более μ итераций, таким образом, вхождения occ шаблона P обнаруживаются с помощью $O(\mu\, occ)$ запросов к структуре данных Rank.

Теорема 14.7. *Для строки $S[1, n]$, составленной из символов алфавита* Σ, *существует сжатый индекс, которому для поддержки операции* Locate(P) *требуются время $O(\mu\, occ)$ и пространство $O(n/\mu \log n)$ битов при условии доступности диапазона рядов* [*first*, *last*] *с префиксом P.*

Если зафиксировать $\mu = \log 1 + \varepsilon n$, то решение для одного вхождения потребует полилогарифмического времени и потенциально сублинейного пространства (в битах) в индексированной строке длиной n. Возможны различные компромиссы, и сейчас литература изобилует данными об асимптотических улучшениях и экспериментальных исследованиях этих сжатых индексов.

Неудивительно, что операцию Count(P) можно адаптировать для реализации последней базовой операции, поддерживаемой FM-индексом, — Extract(i, j). Пусть r — ряд матрицы M' с префиксом $S[j, n]$, а значение r после выполнения Count(P) известно. Алгоритм выполняет присваивание $S[j] = F[r]$, а затем запускает цикл, который движется по строке S назад от $S[j - 1]$ (потому что элемент $S[j]$ был найден через массив F), развертывая LF-отображение, реализованное с помощью структуры данных Rank, следующим образом: $S[j - 1 - t] = L[LFt[r]]$, где $t = 0, 1... j - i - 1$. Он останавливается после $j - i$ шагов, когда достигнут элемент $S[i]$. Примерно такой подход использовался в инверсии BWT, с той разницей, что массив LF недоступен в явном виде — его записи генерируются на лету посредством вычислений Rank. Это все еще гарантирует эффективный по времени доступ к массиву LF и сжатое пространство (благодаря теореме 14.5).

Учитывая привлекательные асимптотические характеристики и структурные свойства FM-индекса, несколько авторов провели обширный набор экспериментов, чтобы изучить его поведение на практике [5][1]. Эксперименты показали, что

[1] Ознакомиться с библиотекой структур данных Succinct можно по адресу https://github.com/simongog/sdsl-lite.

FM-индекс компактен (занимаемое им пространство обычно не так уж далеко от достигаемого компрессором `bzip`), быстро считает количество вхождений шаблона (тратя на один символ несколько микросекунд), а затраты на их извлечение, когда их немного, вполне разумны — около 100 тыс. вхождений/с. Кроме того, FM-индекс позволяет жертвовать занимаемым пространством ради времени поиска, выбирая объем хранимой в нем вспомогательной информации, то есть правильно устанавливая параметр μ и несколько других параметров, возникающих при реализации `Rank`. В результате FM-индекс объединяет сжатие и полнотекстовое индексирование. Он, подобно компрессору `bzip`, инкапсулирует сжатую версию исходного файла, доступную с помощью функции `Extract`, и одновременно, подобно суффиксным деревьям и массивам суффиксов, позволяет искать произвольные шаблоны с помощью функций `Count` и `Locate`. Для работы достаточно просматривать небольшую часть сжатого файла, избегая его полной распаковки.

Список литературы

1. *Adjeroh D., Bell T., Mukherjee A.* The Burrows — Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching. Springer, 2008.
2. *Barbay J., He M., Munro J. I., Rao S. S.* Succinct indexes for strings, binary relations and multi-labeled trees // Proceedings of 18th ACM — SIAM Symposium on Discrete Algorithms (SODA), 680–689, 2007.
3. *Bentley J. L., Sleator D. D., Tarjan R. E., Wei V. K.* A locally adaptive data compression scheme // Communication of the ACM, 29 (4): 320–330, 1986.
4. *Burrows M., Wheeler D. J.* A Block-Sorting Lossless Data Compression Algorithm // Technical Report 124, Digital Systems Research Center (SRC), 1994.
5. *Nelson M. R.* Data compression with the Burrows — Wheeler transform // Dr. Dobb's Journal of Software Tools, 21 (9): 46–50, 1996.
6. *Ferragina P., Gonzalez R., Navarro G., Venturini R.* Compressed text indexes: From theory to practice // ACM Journal on Experimental Algorithmics, 13: article 12, 2009.
7. *Ferragina P., Giancarlo R., Manzini G., Sciortino M.* Compression boosting in optimal linear time // Journal of the ACM, 52 (4): 688–713, 2005.
8. *Ferragina P., Manzini G.* Indexing compressed texts // Journal of the ACM, 52 (4): 552–581, 2005.
9. *Ferragina P., Manzini G.* Burrows — Wheeler transform // *Kao M.-Y.* Encyclopedia of Algorithms. Springer, 2008.
10. *Ferragina P., Manzini G., Muthukrishnan S., coeditors.* Theoretical Computer Science: Special Issue on the Burrows // Wheeler Transform, 387 (3), 2007.
11. *Ferragina P., Manzini G., M kinen V., Navarro G.* Compressed representations of sequences and full-text indexes // ACM Transactions on Algorithms, 3: article 20, 2007.

12. *Karkkainen J., Kempa D., Puglisi S. J.* Slashing the time for BWT inversion // Proceedings of the IEEE Data Compression Conference, 99–108, 2012.
13. *Karkkainen J., Puglisi S. J.* Medium-space algorithms for inverse BWT // Proceedings of the 18th European Symposium on Algorithms (ESA), Lecture Notes in Computer Science, Springer, 6346, 451–462, 2010.
14. *Karkkainen J., Puglisi S. J.* Cache-friendly Burrows — Wheeler inversion // Proceedings of the 1st International Conference on Data Compression, Communication and Processing (CCP), 38–42, 2011.
15. *Manzini G., Ferragina P.* Engineering a lightweight suffix array construction algorithm // Algorithmica, 40 (1): 33–50, 2004.
16. *Navarro G., Mäkinen V.* Compressed full-text indexes // ACM Computing Surveys, 39 (1): article 2, 2007.
17. *Puglisi S. J., Smyth W. F., Turpin A.* A taxonomy of suffix array construction algorithms // Proceedings of the Prague Stringology Conference, 1–30, 2005.
18. *Seward J.* Space-time tradeoffs in the inverse B – W transform // Proceedings of the IEEE Data Compression Conference, 439–448, 2001.
19. *Seward J.* bzip2. https://www.sourceware.org/bzip2.
20. *Witten I. H., Moffat A., Bell T. C.* Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, second edition, 1999.

Глава 15
КОМПАКТНЫЕ СТРУКТУРЫ ДАННЫХ

Когда Клод Шеннон встречает Дональда Кнута...

В предыдущей главе вы познакомились со сжатой версией суффиксных массивов — FM-индексом. В современной литературе предлагается множество сжатых решений для большинства, если не для всех классических структур данных: массивов, деревьев и графов [4]. В последней главе мне хотелось бы дать общее представление о новых подходах к проектированию структур данных и обсудить те из них, которые кажутся наиболее значимыми и плодотворными с дидактической точки зрения. Побочным эффектом этого обсуждения станет введение парадигмы, называемой *программированием без указателей*. Это отказ от явного использования указателей и, следовательно, целочисленных смещений в 4–8 байт для индексации произвольных элементов, таких как строки, узлы или ребра. Вместо них применяются *компактные* структуры данных, построенные на соответствующих двоичных массивах, которые включают в себя указатели, некоторые операции над ними и многое другое.

По крайней мере, с теоретической точки зрения программирование без указателей считается жизнеспособной современной альтернативой для представления классических структур данных на основе указателей в сжатом пространстве, не меняющей их асимптотическую сложность. Так как эта книга посвящена проектированию алгоритмов, следует упомянуть, что для программирования без указателей попрежнему требуются квалифицированные инженеры — разработчики алгоритмов, способные определить, насколько эффективен такой подход, и использовать его наиболее плодотворно для создания приложений, работающих с большими данными.

15.1. Компактное представление двоичных массивов

Рассмотрим следующий эталонный пример. Дан словарь $\mathcal{D}$ из n строк общей длиной m. Нужно отобразить его в одну строку $T[1, m]$ (без разделителей между соседними строками) с поддержкой двух запросов: `Access_string`(i), извлекающего из T строку i, и `Which_string`(x), запрашивающего начальную позицию строки в T, включая символ $T[x]$.

Обычно эта задача решается с помощью *массива указателей* $A[1, n]$ на строки словаря $\mathcal{D}$, который реализуется с помощью их смещений в $T[1, m]$, таким образом, требует пространство $\Theta(n \log m)$ бит. В этом случае запрос `Access_string`(i) возвращает $A[i]$, а запрос `Which_string`(x) сводится к поиску предшественника x

в A. С помощью бинарного поиска первая операция выполняется за время $O(1)$, а вторая — за $O(\log n)$.

Альтернативный подход заключается в использовании компактных представлений для смещений в A. Я опишу два варианта. В первом смещения A реализованы через двоичный массив $B[1, m]$, где $B[i] = 1$ тогда и только тогда, когда $T[i]$ является первым символом словарной строки, обогащенной сжатой структурой данных, которая поддерживает некоторые базовые (но полезные) операции над битами B. Во втором варианте эксплуатируется тот факт, что смещения A представляют собой увеличивающиеся целые числа, поэтому для их индексации в сжатой форме можно использовать код Элиаса — Фано, с которым вы познакомились в главе 11.

Для первого решения нужно учесть, что запрос `Access_string`(i) ищет в двоичном массиве B бит с номером i, установленный в 1. А вот запрос `Which_string`(x) ищет первый установленный в 1 бит слева от $B[x]$ включительно. Это эквивалентно подсчету количества k единиц в массиве $B[1, x]$ с последующим переходом к k-му биту, установленному в 1. Первая операция называется `Select`(i), а вторая, выполняющая подсчет, — `Rank`(x). Обе они реализуются сканированием массива B, но в худшем случае это требует изрядных затрат. Далее я покажу вам структуру данных, позволяющую выполнить обе эти операции за постоянное время, которой в дополнение к B требуется пространство $o(m)$ битов. Благодаря такой пространственной сложности это решение называется *кратким* (из-за явного хранения B) и в случае $n = o(m/\log m)$ занимает более компактное пространство, чем решение на основе указателя.

Для второго решения применяются алгоритмические свойства кода Элиаса — Фано, который запросом `Access`(i) извлекает $A[i]$ за постоянное время, а запросом `Rank`(x) — за логарифмическое время. Занимаемое пространство ограничено сверху $O(n \log(m/n))$ битами (см. главу 11). Я покажу, что эта пространственная граница связана с энтропией массива B, поэтому для всех значений m и n такое решение называется *сжатым* и с точки зрения занимаемого пространства является лучшим, чем решение на основе указателя.

15.1.1. Краткое решение с помощью функций Rank и Select

Для достижения поставленных целей вводятся два примитива и соответствующие им структуры данных, называемые `Rank` и `Select`.

Определение 15.1. Пусть $B[1, m]$ — двоичный массив.

- Примитив `Rank` индекса i в массиве B относительно бита $b \in \{0, 1\}$ представляет собой число битов b, встречающихся в диапазоне $B[1, i]$. Формально $\texttt{Rank}_1(i) = \sum_{j=1}^{i} B[j]$. Обратите внимание на то, что $\texttt{Rank}_0(i)$ вычисляется за постоянное время как $i - \texttt{Rank}_1(i)$.
- Примитив `Select` индекса i в массиве B относительно бита $b \in \{0, 1\}$ возвращает *позицию* i-го вхождения бита b в массив B и обозначается $\texttt{Select}_b(i)$. В отличие от примитива `Rank` невозможно за постоянное время вычислить $\texttt{Select}_0$ и $\texttt{Select}_1$, то есть для каждого примитива нужны соответствующие структуры данных.

Рассмотрим следующий двоичный массив:

	1	2	3	4	5	6	7	8	9	10	11
$B =$	0	0	1	0	1	0	0	1	0	1	0

и примитив $\texttt{Rank}_1(6)$, который запрашивает количество единиц, встречающихся в первых шести позициях массива B. В следующем массиве выделена та часть, в которой нужно произвести *подсчет*, а результат равен $\texttt{Rank}_1(6) = 2$:

	1	2	3	4	5	6	7	8	9	10	11
$B =$	0	0	**1**	0	**1**	0	0	1	0	1	0

.

Теперь рассмотрим примитив $\texttt{Select}_1(3)$, который запрашивает позицию третьего вхождения бита 1 в массиве B. Ответ на него — $\texttt{Select}_1(3) = 8$. В массиве выделена возвращаемая позиция, а еще указан ранг всех единичных битов, предшествующих запрошенному.

	1	2	3	4	5	6	7	8	9	10	11
$B =$	0	0	1	0	1	0	0	**1**	0	1	0
			1		2			3			

.

Реализация структуры данных Rank. *Краткая* структура данных, поддерживающая операцию `Rank`, состоит из *трех уровней*. На первом происходит логическое разбиение двоичного массива B на *блоки большего размера* Z. Для каждого из них сохраняется метаинформация, необходимая для поддержки операции `Rank`. На втором уровне каждый большой блок логически разбивается на маленькие блоки размером z и для каждого сохраняется еще какая-то метаинформация. Третий уровень состоит из таблицы прямого доступа, индексированной маленькими блоками и запрашиваемыми позициями. Далее будет доказано, что метаинформация, хранящаяся на первых двух уровнях, и таблица третьего уровня сжаты в пространстве и совокупно занимают $o(m)$ бит. Для простоты предположим, что z делит Z таким образом, что *большой блок* номер i равен $B[Z(i-1)+1, Zi]$, а j-й *малый блок* внутри i-го большого блока равен $B[Z(i-1)+z(j-1)+1, Z(i-1)+zj]$, где $i, j \geqslant 1$.

Метаинформация, связанная с i-м большим блоком, состоит из количества единиц в префиксе массива B, предшествующего этому большому блоку. Она называется *абсолютным рангом* и обозначается r_i. Метаинформация, связанная с j-м малым блоком внутри i-го большого блока, состоит из количества единиц в префиксе большого блока, предшествующего этому малому блоку. Она называется *относительным рангом* и обозначается $r_{i,j}$. Наглядно поясняет эти понятия рис. 15.1. В частности, если r_i по определению является числом единиц, предшествующих (до начала B) выделенному большому блоку, то $r_{i+1} = r_i + 4$, потому что в этом блоке четыре единицы. Что касается метаинформации малых блоков, обратите внимание на то, что $r_{i,1} = 0$, — это показано для полноты картины. Два других относительных ранга составляют $r_{i,2} = 2$, поскольку второму малому блоку в выделенном большом блоке предшествуют две единицы, и $r_{i,3} = 3$, так как третьему

малому блоку в выделенном большом блоке предшествуют три единицы (это показано в масштабированном массиве).

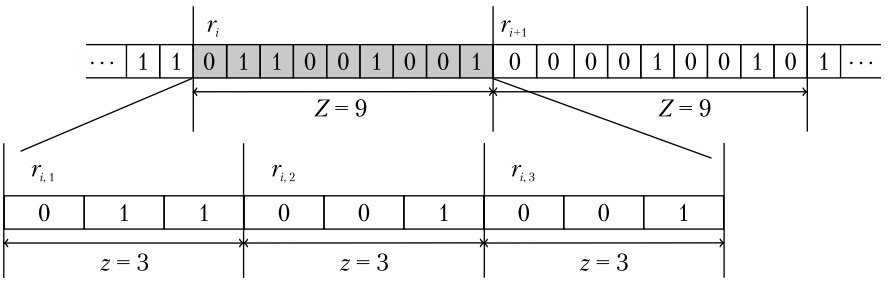


Рис. 15.1. Пример метаинформации больших блоков размером $Z = 9$ и малых блоков размером $z = 3$ в контексте реализации операции Rank. Здесь r_i — количество единиц от начала двоичного массива B до первой записи выделенного большого блока (исключенного), поэтому $r_{i+1} = r_i + 4$. Для относительных рангов, связанных с малыми блоками, $r_{i,1} = 0$, $r_{i,2} = 2$, $r_{i,3} = 3$

Занятость пространства всеми абсолютными рангами можно вычислить, умножив количество больших блоков на пространство, необходимое для хранения одного абсолютного ранга, а именно $O(m/Z \log m)$ бит, ведь каждый абсолютный ранг меньше размера m массива B. По тому же принципу пространство, необходимое для хранения всех относительных рангов, составляет $O(m/z \log Z)$, ведь каждый относительный ранг меньше размера Z больших блоков, благодаря чему для его хранения достаточно $O(\log Z)$ бит.

Пусть $Z = (\log m)2$ и $z = 1/2 \log m$. В этом случае совокупное занятое пространство составит:

$$= O\left(\frac{m}{Z}\log m + \frac{m}{z}\log Z\right) =$$

$$= O\left(\frac{m}{\log^2 m}\log m + \frac{m}{(1/2)\log m}\log\left(\log^2 m\right)\right) =$$

$$= O\left(\frac{m}{\log m} + \frac{m}{\log m}\log\log m\right) =$$

$$= O\left(\frac{m\log\log m}{\log m}\right) = o(m).$$

Легко убедиться, что получить Rank позиций за постоянное время можно в конце каждого блока — как маленького, так и большого, читая абсолютный ранг большого блока, следующего за запрашиваемой позицией, или суммируя абсолютный ранг большого блока, охватывающего запрашиваемую позицию, и относительный ранг малого блока, следующего за запрашиваемой позицией.

Итак, мы знаем, как ответить на запрос `Rank` по последней позиции каждого блока — и маленького, и большого. Но как быть с запросами позиций внутри *малых блоков*? Первое решение базируется исключительно на метаинформации, предоставляемой абсолютными и относительными рангами. Предположим, запрос $\texttt{Rank}_1(x)$ сформулирован для произвольной позиции x в массиве B, возможно, находящейся внутри малого блока. Для упрощения объяснения пусть $B[x]$ входит в j-й малый блок, который, в свою очередь, находится внутри i-го большого блока, обозначенного $B_{i,j}$. Ответ на $\texttt{Rank}_1(x)$ вычисляется как $ri + ri,j + \texttt{Count1}[B_{i,j}, x]$, где последний член подсчитывает количество единиц в малом блоке $B_{i,j}$ до бита $B[x]$ включительно. Обратите внимание на то, что индексы i и j вычисляются как $i = 1 + \left\lfloor \frac{x-1}{Z} \right\rfloor$ и $j = 1 + \left\lfloor \frac{(x-1) \bmod Z}{z} \right\rfloor$, где $r_1 = 0, r_{i,1} = 0$. Напомню, что индексы отсчитываются от 1. С учетом этой информации извлечение первых двух величин занимает постоянное время, тогда как на операцию $\texttt{Count}_1[B_{i,j}, x]$, выполняемую сканированием $B_{i,j}$, в худшем случае затрачивается время $O(z) = O(\log m)$.

Когда z умещается в одно слово памяти, операцию $\texttt{Count}_1[B_{i,j}, x]$ можно реализовать за постоянное время с помощью примитивов, манипулирующих отдельными битами, такими как функция `std::pop_count`[1]. Для z, состоящих из небольшого количества слов памяти, можно развернуть операции SIMD (single instruction, multiple data — «одиночный поток команд, множественный поток данных») и по-прежнему сделать все очень быстро. Для более длинных z оптимальная теоретическая граница постоянного времени все еще достижима, но только если включить третью часть метаданных, состоящую из таблицы R (рис. 15.2), которая содержит ответы для всех возможных конфигураций малых блоков и запрошенных позиций.

R	Возм.		
Блок	**1**	**2**	**3**
000	0	0	0
001	0	0	1
010	0	1	1
011	0	1	2
100	1	1	1
101	1	1	2
110	1	2	2
111	1	2	3

Рис. 15.2. Таблица поиска R предвычисленных рангов для всех возможных малых блоков размером $z = 3$ бита. Здесь $R[b, o]$ обозначает ранг элемента в относительной позиции o внутри блока двоичной конфигурации b

[1] https://en.cppreference.com/w/cpp/numeric/popcount.

Для вычисления $\texttt{Count}_1[B_{i,j}, x]$ нужно обратиться к соответствующей записи в R, а именно $R[B_{i,j}, o]$, где $o = 1 + ((x-1) \bmod z)$ — смещение бита $B[x]$ в малом блоке $B_{i,j}$. Такая процедура требует трех доступов к памяти (один для r_i, один для $r_{i,j}$, и один для $R[B_{i,j}, o]$) и двух сложений, то есть занимает время $O(1)$. Удивительно, но хранение таблицы R требует далеко не таких больших затрат, как может показаться, учитывая, что $z = (1/2)\log m$. Фактически R состоит из $2z$ рядов и z столбцов, а каждая запись может быть представлена в $O(\log z)$ битах, потому что она подсчитывает количество единиц в малом блоке. Таким образом, совокупно для R необходимо пространство размером $2^z z \log z = O\left(2^{\log\sqrt{m}}(\log m)(\log\log m)\right) = o(m)$ бит. Последний столбец таблицы R хранит информацию, также доступную в $r_{i,j}$, поэтому его можно отбросить как избыточный, даже если это не изменит асимптотическую занятость пространства.

Теорема 15.1. *Структуре данных* `Rank` *требуется пространство* $o(m)$ *бит, то есть она асимптотически сублинейна по отношению к размеру двоичного массива* $B[1, m]$. *Алгоритм* `Rank` *в худшем случае работает постоянное время и обращается к массиву* B *только для чтения.*

Рассмотрим процесс вычисления $\texttt{Rank}_1(x)$ на примере, приведенном на рис. 15.1. Имеющийся на нем элемент $x = 17$ на рис. 15.3 отмечен стрелкой. Алгоритм разделен на три этапа для извлечения трех слагаемых формулы $r_i + r_{i,j} + R[B_{i,j}, o]$. Напомню, что $i = 1 + \left\lfloor \frac{x-1}{Z} \right\rfloor$, $j = 1 + \left\lfloor \frac{(x-1) \bmod Z}{z} \right\rfloor$, а $o = 1 + ((x-1) \bmod z)$.

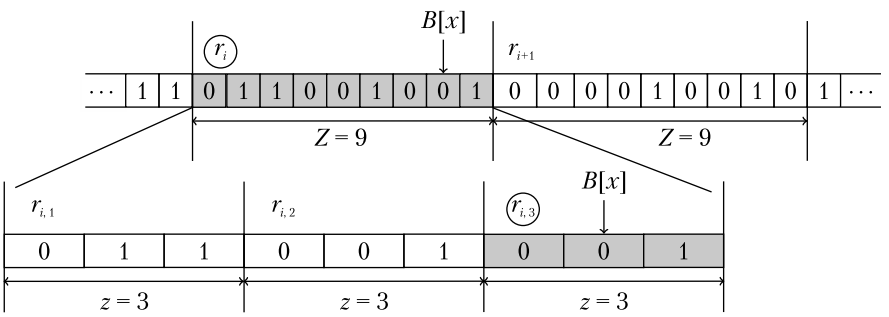


Рис. 15.3. Графическое пояснение вычисления Rank. Абсолютные и относительные ранги обведены кружками

1. Ищем большой i-й блок, включающий в себя $B[26]$, а именно $i = 1 + \left\lfloor \frac{17-1}{9} \right\rfloor = 2$, что дает абсолютный ранг r_2.
2. Ищем малый j-й блок, включающий в себя $B[26]$, а именно $j = 1 + \left\lfloor \frac{(17-1) \bmod 9}{3} \right\rfloor = 3$, что дает относительный ранг $r_{2,3}$.

3. Находим смещение $B[x]$ внутри малого блока $B_{2,3}$, а именно $o = 1 + ((17 - 1) \bmod 3) = 2$, что дает доступ к записи $R[001, 2]$, которая в рассматриваемом примере равна нулю (см. рис. 15.2).
4. В результате получаем $\texttt{Rank}_1(17) = r_2 + r_{2,3} + R[001, 2]$.

Реализация Select. В отличие от структуры данных `Rank`, у которой двоичный массив B разбивается на большие и малые блоки фиксированной длины, при выполнении `Select` основой для разбиения становится количество битов, установленных в 1.

Пусть $K = \log^2 m$, а переменная Z обозначает размер в битах *больших блоков*, которые содержат K бит, установленных в 1. Очевидно, что при переходе от одного большого блока к другому Z будет меняться, но для простоты я обозначил это значение одной буквой и буду давать пояснения в тексте. По определению $Z \geqslant K$, поэтому для хранения всех начальных позиций больших блоков требуется $O(m/K \log m) = o(m)$ бит. Поскольку большие блоки содержат ровно K бит, установленных в 1, простая арифметическая операция может указать, в каком большом блоке находится та единица, которую ищет операция $\texttt{Select}_1(i)$.

Другими словами, для поиска $\texttt{Select}_1(i)$ нужно рассмотреть содержимое большого блока. Сканировать его непроизводительно, поэтому спроектируем второй уровень структуры данных `Select`, разбив большие блоки на *маленькие*, размер которых также будет зависеть от количества единиц. При этом будем различать разреженные и плотные большие блоки. Большой блок называется *разреженным*, если в нем мало единиц относительно его размера. В количественном выражении мало означает, что $Z > K^2$. В противном случае блок называется *плотным* и определяется неравенством $Z \leqslant K^2$.

В разреженном большом блоке можно явно хранить позиции его битов, установленных в 1, так как это займет не слишком много места. Требуемое пространство $O\left(\frac{m}{K^2} K \log m\right) = O\left(\frac{m}{\log^2 m}\right) = o(m)$ бит. Эта формула вытекает из того, что длина разреженного большого блока $Z > K^2$ и $K = \log^2 m$.

В случае плотного большого блока ($Z \leqslant K^2$) нужно действовать рекурсивно, разбивая его на *малые блоки* по $k = (\log \log m)^2$ бит, установленных в 1. Пусть z — длина *малого блока*. Как и в случае с большими блоками, z может быть разной, но для упрощения записи нижние индексы я опущу. Хранение всех начальных позиций малых блоков относительно начала плотного большого блока, в который они входят, требует пространства $O\left(\frac{m}{k} \log K^2\right) = O\left(\frac{m}{(\log \log m)^2} \log(\log^4 m)\right) = o(m)$ бит. Длина малого блока в данном случае не менее k, а длина охватывающего его плотного большого блока не превышает K^2.

Чтобы отслеживать в малых блоках позиции битов, установленных в 1, повторим процедуру для больших блоков, введя различие между *разреженными* и *плотными* малыми блоками в зависимости от того, больше или меньше длина z малого блока, чем $k^2 = (\log \log m)^4$ бит. Позиции единиц разреженного малого блока ($z > k^2$) можно

хранить в явном виде, но теперь относительно начала окружающего большого блока, занимая пространство $O\left(\frac{m}{k^2}k\log K^2\right)=O\left(\frac{m}{(\log\log m)^2}\log\left(\log^4 m\right)\right)=o(m)$ бит. Здесь длина разреженного малого блока $z > k^2$, соответственно, количество блоков составляет $O(m/k^2)$. Более того, z короче плотного большого блока, длина которого $Z \leqslant K^2$. Осталось сохранить единицы, встречающиеся в плотных малых блоках внутри плотных больших блоков. Повторим то, что делалось для третьего уровня структуры данных `Rank`, предварительно вычислив таблицу T всех ответов на запрос $\texttt{Select}_1$ в плотном малом блоке. Размер этой таблицы $z \leqslant k^2 = (\log\log m)^4$, то есть она занимает пространство $O(z \times 2^z \log z) = o(m)$ бит.

Чтобы лучше понять, как работает операция $\texttt{Select}_1$, обратимся к рис. 15.4. Пусть $K = 3$ и $k = 2$. Обратите внимание на то, что выделенный большой блок является плотным. Количество установленных в 1 битов в нем $K = 3$ при $Z = 7 < 9 = K^2$. А следующий за ним большой блок разреженный, поскольку при таком же количестве битов, установленных в 1, его размер $Z = 10 > 9 = K^2$. Более того, первый малый блок выделенного большого блока плотный, так как он содержит $k = 2$ бита, установленных в 1, при размере $z = 3 < 4 = k^2$, в то время как второй малый блок разрежен и фактически содержит меньше k бит, установленных в 1, поскольку k не является делителем K. Согласно описанию структуры данных для примитива `Select` позиции единиц разреженного большого блока и разреженного малого блока могут храниться явно: первые — как абсолютные значения, вторые — как значения относительно начала охватывающего их большого блока. В качестве альтернативы первый малый блок выделенного большого блока, если он плотный, вносит вклад в построение таблицы T.

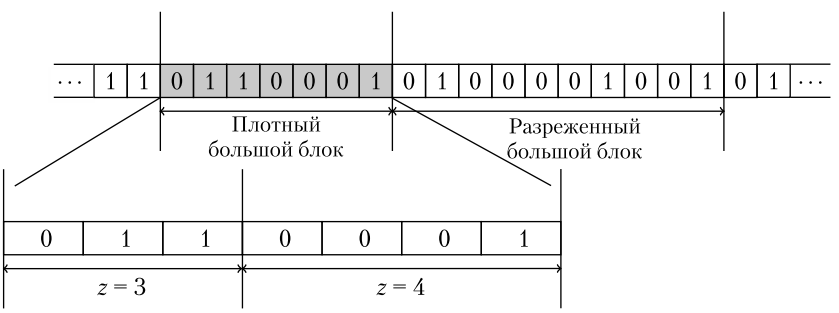


Рис. 15.4. Реализация Select. В этом примере $K = 3$ и $k = 2$. Комментарии в тексте

Вот как выглядит реализация операции $\texttt{Select}_1(i)$, которая использует трехуровневую структуру данных.

1. Вычисляем индекс $j = 1 + \left\lfloor \frac{i-1}{K} \right\rfloor$ *большого блока*, включая i-й бит, установленный в 1, из двоичного массива B. Обозначим этот большой блок B_j.

2. Для разреженного блока B_j структура данных сохраняет результат $\texttt{Select}_1(i)$ в явном виде, и на этом процесс завершается. В противном случае структура данных сохраняет начальную позицию (плотного) большого блока B_j — назовем ее s_j.
3. Превращаем примитив $\texttt{Select}_1(i)$ внутри массива B в *относительный* примитив $\texttt{Select}_1(i')$ внутри блока B_j, вычислив $i' = 1 + (i - 1 \bmod \mathrm{K})$.
4. Вычисляем индекс $j' = 1 + \left\lfloor \frac{i'-1}{k} \right\rfloor$ *малого блока*, включая i'-й бит блока B_j, установленный в 1.
5. Структура данных сохранила начальную позицию $B_{j,j'}$ относительно начала блока B_j — назовем ее $s'_{j'}$.
6. На этом этапе для разреженного блока $B_{j,j'}$ структура данных сохраняет позицию i'-го бита, установленного в 1, в B_j относительно начала этого большого блока. Прибавляя эту относительную позицию к s_j, мы получаем ответ на запрос $\texttt{Select}_1(i)$. При наличии доступа к предварительно вычисленной таблице T с двоичной конфигурацией $B_{j,j'}$ и значением $1 + (i - 1 \bmod k^2)$ для получения ответа на запрос $\texttt{Select}_1(i)$ нужно прибавить к извлеченной записи T сумму $s_j + s'_{j'}$.

Длина плотных небольших блоков очень мала (они короче, чем $(\log\log m)^4$, что для применяемых на практике значений m дает совсем маленькое значение), поэтому, как и в случае операции `Rank`, можно прибегнуть к сканированию, отказавшись от предварительно вычисленной таблицы T. Таким образом, доказана следующая теорема.

Теорема 15.2. *Пространство, занимаемое структурой данных* $\texttt{Select}_1$, *составляет* $o(m)$ бит, *то есть оно растет медленнее, чем линейно, относительно размера двоичного массива* $B[1, m]$. *Алгоритм* $\texttt{Select}_1$ *выполняется за постоянное время в худшем случае и обращается к массиву B только в режиме чтения. Те же ограничения по времени и пространству справедливы для* $\texttt{Select}_0$.

15.1.2. Компактное решение с использованием кодирования Элиаса — Фано

Подход, который я сейчас опишу, может быть напрямую применен к массиву смещенных указателей $A[1, n]$ на строки словаря D или их характеристическому двоичному вектору $B[1, m]$. Фактически последний можно преобразовать в первый, взяв из B позиции битов, установленных в 1. Это даст возрастающую последовательность положительных целых чисел, то есть допустимый ввод для кода Элиаса — Фано.

В качестве примера рассмотрим двоичный массив:

	1	2	3	4	5	6	7	8	9	10	11
$B =$	0	1	1	0	0	0	0	1	0	1	0

,

который нужно превратить в массив:

$$A = \begin{array}{|c|c|c|c|} \hline 2 & 3 & 8 & 10 \\ \hline \end{array}\ .$$

Как говорилось в главе 11, сжатие массива A с помощью кода Элиаса — Фано дает два массива:

$$L = \begin{array}{|c|c|c|c|} \hline 10 & 11 & 00 & 10 \\ \hline \end{array}$$

и

$$H = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ \hline \end{array}\ ,$$

определенные по $n = 4$ элементам (отсюда четыре единицы в массиве H), где размер пространства $u = 11$, размер слова $b = \lceil \log_2 11 \rceil = 4$, количество битов для наименее значимой части $\ell = \left\lceil \log \frac{u}{n} \right\rceil = \left\lceil \log \frac{11}{4} \right\rceil = 2$ и количество битов для наиболее значимой части $h = b - \ell = 2$.

Как упоминалось в главе 11, наиболее интересное свойство кода Элиаса — Фано — это возможность дополнить массив H структурами данных и алгоритмами для эффективной поддержки следующих операций:

- `Access`(i), которая для индекса $1 \leqslant i \leqslant n$, возвращает $A[i]$;
- `NextGEQ`(x), которая для целого числа $0 \leqslant x < u$ возвращает минимальный элемент, подходящий под условие $A[i] \geqslant x$.

Для массива A можно легко вычислить значение $\texttt{Select}_1(B, i)$ с помощью операции `Access`(i) и значение $\texttt{Rank}_1(B, i)$ с помощью операции `NextGEQ`($i + 1$) – 1. При этом для упрощения объяснения предполагается, что $A[n + 1] = \infty$ и `NextGEQ` возвращает позицию i элемента $A[i] \geqslant x$, а не его значение. Теперь у нас есть все, чтобы обсудить реализацию этих двух операций с использованием терминологии главы 11 и дополнением в виде массива H, который опирается на структуру данных, поддерживающую примитив $\texttt{Select}_1$.

Для вычисления `Access`(i) необходимо объединить старшие и младшие биты $A[i]$ из массивов L и H соответственно. Наименее значимые биты $A[i]$ легко извлекаются при наличии доступа к i-му блоку из ℓ битов двоичной последовательности L. Извлечь h старших бит $A[i]$ немного сложнее — в массиве H нужно определить ссылающуюся на эти биты отрицательную унарную последовательность. Алгоритм знает только входное i, значение $A[i]$ неизвестно, а именно отрицательная унарная последовательность включает i-й бит, установленный в `1` в H. Позицию этого бита, считая от 1, дает операция $\texttt{Select}_1(i, H)$. Остается вычесть i и получить количество предшествующих этой позиции нулей. В соответствии со свойствами кода Элиаса — Фано оно эквивалентно конфигурации корзины с нужным битом `1`. Представив эту конфигурацию с помощью h бит, мы получаем самую значимую часть $A[i]$. Таким

образом, временная сложность этого алгоритма совпадает со сложностью примитива `Select`, а согласно теореме 15.2, она постоянна. Дополнительная пространственная сложность, необходимая для поддержки примитива `Select` для массива H, составляет $o(|H|) = o(n)$ бит, то есть она сублинейна по n — числу битов, установленных в 1, а не по размеру m массива B.

Теперь выполним операцию `Access`(3), которая должна вернуть $A[3] = 8$. Получить $\ell = 2$ младших бита $A[3]$ можно из третьей пары битов в L, которые равны $L[3] =$ `00`. Извлечение оставшихся $h = b - \ell = 2$ старших бит выполняется вычислением $\texttt{Select}_1(3, H) - 3 = 5 - 3 = 2$ и последующим кодированием конфигурации этой корзины в $h = 2$ битах, то есть `10`. Объединение двух битовых последовательностей дает нужный результат: $A[3] =$ `10` $\times$ `00` $= (\texttt{1000})_2 = 8$.

Теперь рассмотрим реализацию другой операции — `NextGEQ`(x). Алгоритмическая идея заключается в идентификации корзины целого числа x с учетом h старших бит и определении результата операции `NextGEQ`(x) в ходе просмотра находящихся в этой корзине целых чисел массива A. Обозначим h старших битов числа x как v_h. Элементы массива A в контейнере v_h ищут путем просмотра его отрицательной унарной последовательности в массиве H. При $v_h > 0$ эта последовательность начинается с бита в позиции $p = \texttt{Select}_0(v_h) + 1$, в противном случае $p = 0$. А заканчивается она в позиции $q = \texttt{Select}_0(v_h + 1)$.

Итак, $H[p, q] = 1^{q-p-1}0$ — это отрицательная унарная последовательность, а $q - p$ — количество целых чисел в массиве A, у которых h старших бит равны vh. Если бит $H[p] = 0$, корзина пуста ($q - p = 0$) и ни одно целое число в A не имеет тех же h старших бит, что и x. Соответственно, операция `NextGEQ`(x) должна вернуть первый элемент следующей непустой корзины, у которой h старших бит наверняка превышают vh. Этот элемент соответствует первой `1` справа от $H[p]$. Его позиция не имеет значения, достаточно знать его ранг i в массиве A, а затем выполнить операцию `Access`(i).

Ранг i равен $p - vh$, поскольку vh — это количество битов, установленных в `0` в $H[1, p]$. В противном случае элементы корзины имеют те же самые h старших бит, что и x, следовательно, элемент, отвечающий запросу `NextGEQ`(x), соответствует или биту `1` в $H[p, q]$, или первой `1` справа от $H[q]$. Чтобы различить эти две ситуации, нужно найти для $i = p - v_h \ldots q - v_h$ первое значение, возвращаемое операцией `Access`(i), которое больше или равно x. Количество элементов в корзине не превышает $2^\ell = \Theta(u/n)$ (точнее, их не больше, чем $\min\{n, 2l\}$), поэтому их сканирование займет время $O(\min\{n, u/n\})$ в худшем случае. Этот процесс можно ускорить, выполнив по элементам двоичный поиск за время $O(\min\{\log n, \log (u/n)\})$. Дополнительная пространственная сложность для поддержки примитива `Select` над массивом H составляет $o(|H|) = o(n)$ бит.

Теперь выполним операцию `NextGEQ`(9), которая должна вернуть $A[4] = 10$. Так как $9 =$ `1001`, $\ell = 2$ и $h = 2$, получаем $p = \texttt{Select}_0(10_2) + 1 = \texttt{Select}_0(2) + 1 = 4 + + 1 = 5$ и $q = \texttt{Select}_0(2 + 1) = 7$. Для $H[5] = 1$ корзина непуста и ее элементы в $H[5, 6]$ имеют те же h старших цифр, что и `10`. Сканирование массива A между

позициями $i = 5 - 2 = 3$ и $i = 7 - 2 = 5$ (не включительно) возвращает значение 10, ведь $A[3, 4] = [8, 10]$. В качестве еще одного примера рассмотрим `NextGEQ`(4), где 4 = `0100`, представленное в $b = 4$ битах. Вычислим $p = \texttt{Select}_0(\texttt{01}) + 1 = \texttt{Select}_0(1) + 1 = 3 + 1 = 4$. При $H[4] = 0$ соответствующая корзина пуста и требуется искать первый элемент следующей непустой корзины. Это элемент с рангом $i = p - v_h = 4 - 1 = 3$ в массиве A, а именно $A[3] = 8$, который извлекается операцией `Access`(3).

Теорема 15.3. *Для массива $A[1, n]$ возрастающих положительных целых чисел в диапазоне $[0, m)$ существует компактный индекс, занимающий пространство — $2n + n\left\lceil \log_2 \frac{m}{n} \right\rceil + o(n)$ бит и поддерживающий извлечение его элементов (операцию* `Access`*) за постоянное время в худшем случае, а извлечение целого числа, большего или равного заданному (операцию* `NextGEQ`*), — за время $O(\log(m/n))$ в худшем случае.*

Для двоичного массива B, интерпретируемого как характеристический двоичный вектор массива A возрастающих положительных целых чисел, эта теорема формулируется следующим образом.

Теорема 15.4. *Для двоичного массива $B[1, m]$ с n позициями, равными 1, существует компактный индекс, занимающий $2n + n\left\lceil \log_2 \frac{m}{n} \right\rceil + o(n)$ бит, и поддерживается выполнение операции $\texttt{Rank}_1(i)$ за время $O(\log(m/n))$, а операции $\texttt{Select}_1(i)$ — за время $O(1)$ в худшем случае.*

Обратите внимание на то, что с ростом n число дополнительных битов на элемент, а именно $2 + \left\lceil \log_2 \frac{m}{n} \right\rceil$, уменьшается и в итоге сходится к 2 (плюс члены малого порядка). И наоборот, при уменьшении n число дополнительных битов становится равным $\Theta(\log m)$, как для классических указателей. В любом случае предлагаемое решение не хуже решения на основе указателя, а при плотном B может оказаться намного лучше. Наконец, стоит отметить, что член $n \log(m/n)$ относится к информационно-теоретической нижней границе кодирования n элементов в m позициях, заданной $\left\lceil \log \binom{m}{n} \right\rceil$ битами. Этот последний член можно переписать как $\left\lceil \log \binom{m}{n} \right\rceil = n \log(em/n) - O(\log n) - \Theta(n^2/m)$, что связано с энтропией нулевого порядка $\mathcal{H}_0$ битовой строки длиной m с n битами, установленными в 1. Фактически $\left\lceil \log \binom{m}{n} \right\rceil = m\mathcal{H}_0 - O(\log m)$ (см. подраздел 2.3.1 в [4]). Следовательно, можно сделать вывод, что пространственная сложность предлагаемого подхода близка к оптимальной, за исключением дополнительных двух бит на элемент, при условии, что дополнительная информация о распределении единиц недоступна и поэтому не может быть использована.

15.2. Компактное представление деревьев

Рассмотрим проблему хранения дерева в сжатом виде с сохранением возможности эффективного выполнения некоторых операций над его структурой.

15.2.1. Двоичные деревья

Классический подход к представлению двоичного дерева состоит в том, что на узел, внутренний или листовой, есть два указателя, благодаря которым навигационные операции к *левым* или *правым* дочерним элементам могут быть реализованы за постоянное время. При отсутствии дочернего элемента или достижении листа указатели принимают значение `NULL`. Такое представление дерева занимает $\Theta(n \log n)$ битов, где n — общее количество узлов и листьев. Для более сложных запросов, таких как *запросы к родителям* или *запросы размера поддерева*, требуется дополнительное пространство. В первом случае это дополнительные $\Theta(n \log n)$ бит для хранения указателя на родителя узла, во втором случае потребуются еще $\Theta(n \log n)$ битов для хранения размера поддерева.

Вопрос в том, необходимо ли такое пространство для поддержки навигационных операций за постоянное время или его можно уменьшить. Известно, что нижняя граница сложности хранения двоичного дерева составляет $2n$ бит, потому что асимптотически существует 2^{2n} различных двоичных деревьев из n узлов[1], таким образом, это количество битов нужно, чтобы отличать одно дерево от другого. В этом разделе я опишу блестящую идею Гая Якобсона, датируемую 1989 годом [3]. Она обеспечивает указанную нижнюю границу использования памяти (с учетом членов низшего порядка) путем преобразования навигационных запросов по двоичным деревьям с постоянным временем выполнения в операциях `Rank` и `Select` на полученных из этих деревьев двоичных массивах. Эти операции тоже выполняются за постоянное время. Беспроигрышная ситуация.

Пример бинарного дерева T из $n = 8$ узлов, включая листья, демонстрирует рис. 15.5. Преобразование за три этапа генерирует из него бинарный массив B.

1. **Развертка.** Дополнить все небинарные узлы и листья в T специальными узлами, называемыми *фиктивными* листьями, сформировав расширенное двоичное дерево $\hat{T}$.
2. **Однобитовые метки.** Фиктивные листья в $\hat{T}$ помечаются битом 0, а узлы, которые присутствовали в исходном дереве T, — битом 1.

---

[1] Можно исследовать бинарное дерево, преобразуя его в сбалансированную последовательность открывающих и закрывающих скобок. Открывающая скобка появляется при посещении узла в процессе прямого обхода дерева, а закрывающая — после полного обхода этого узла и его поддерева. После этого каждое бинарное дерево из n узлов будет идентифицироваться последовательностью из n пар скобок. Эти конфигурации $C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1} \approx \frac{4^{n-1}}{(n-1)^{3/2}\sqrt{\pi}}$, называемые *числами Каталана*, асимптотически равны $\Theta\left(2^{2n}\right)$.

3. **Сериализация.** Обойти дерево $\hat{T}$ по уровням слева направо и записать в массив B обнаруженные в процессе обхода бинарные метки.

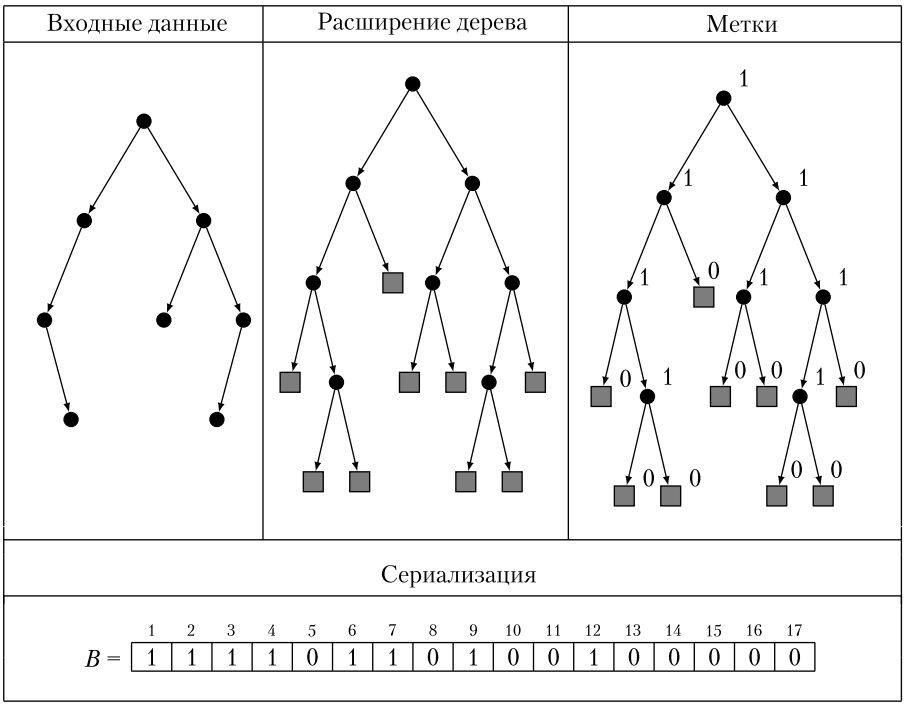


Рис. 15.5. Построение двоичного массива B из двоичного дерева T. *Фиктивные листья* представлены серыми квадратами. Входное двоичное дерево состоит из $n = 8$ узлов, включая его листья. Первый шаг создает расширенное двоичное дерево $\hat{T}$ из $2n + 1 = 17$ общих узлов, включая фиктивные листья. Затем фиктивные листья помечаются 0, а исходные узлы — 1. После чего помеченные узлы $\hat{T}$ сериализуются в двоичный массив B из 17 бит

Легко показать, что расширенное бинарное дерево $\hat{T}$ состоит из $2n + 1$ узлов, включая фиктивные листья, таким образом, выходной двоичный массив B состоит из $2n + 1$ бит. Вы можете самостоятельно доказать это методом индукции.

Для навигационных запросов мы *логически* маркируем (то есть метки не хранятся, а используются для выполнения алгоритмов) каждый узел расширенного дерева $\hat{T}$ двумя целыми числами, которые для удобства обсуждения и иллюстрации обозначены жирным и обычным шрифтом. **Жирные** метки представляют собой целые числа от 1 до n, связанные с узлами входного дерева T. Они соответствуют нумерации T при *поиске в ширину* (breadth-first search, BFS). Обычные метки — это целые числа от 1 до $2n + 1$, прикрепленные к узлам расширенного дерева $\hat{T}$. Они соответствуют нумерации $\hat{T}$ при поиске в ширину. Эта маркировка узлов дерева $\hat{T}$ естественным образом отображается на маркировку битов B. Это тоже *логическая*

маркировка (на рис. 15.6 она показана только для иллюстрации), так как эти числа нигде не сохраняются.

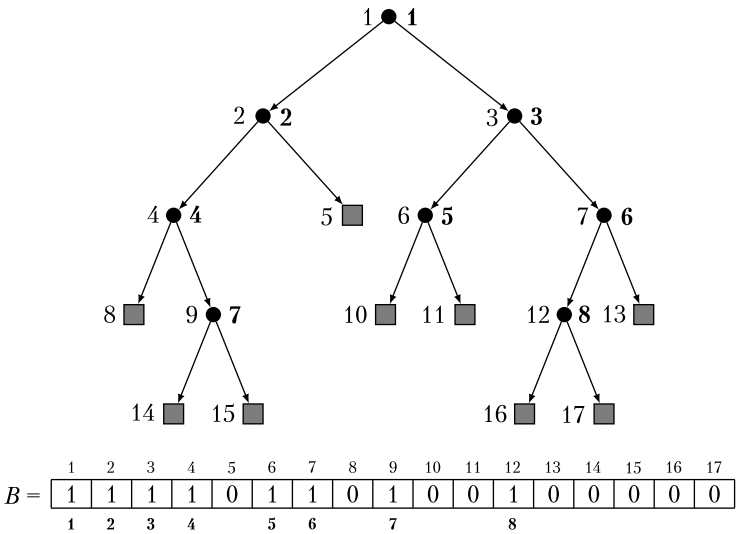


Рис. 15.6. Логическая маркировка расширенного дерева $\hat{T}$ и соответствующего двоичного массива B. Индексы над B относятся к простой маркировке (то есть BFS-нумерации) узлов $\hat{T}$, а внизу жирным шрифтом обозначены индексы узлов T (тоже BFS-нумерация)

Теперь нужно обогатить двоичный массив $B[1, 2n + 1]$ двумя структурами данных, которые поддерживают операции $\texttt{Rank}_1$ и $\texttt{Select}_1$, и, согласно теореме 15.2, добавить $o(n)$ бит к занимаемому массивом B пространству. Ключевой инструмент для реализации навигационных операций над T — биекция от жирных меток к простым и, наоборот, благодаря операциям `Select` и `Rank` соответственно. Точнее, простую метку j можно получить из жирной метки $\mathbf{i}$ узла дерева $\hat{T}$ как результат операции $\texttt{Select}_1(\mathbf{i})$ и, наоборот, жирную метку $\mathbf{i}$ можно получить из простой метки j узла в $\hat{T}$ как результат операции $\texttt{Rank}_1(j)$. Первое вытекает из того факта, что единицы назначаются узлам входного дерева T, которое затем включается в $\hat{T}$, и операция $\texttt{Select}_1(i)$ дает положение рассматриваемого узла в массиве B, перескакивая через нули, появившиеся из-за добавленных фиктивных листьев. Обратное вычисление, то есть $\mathbf{i} = \texttt{Rank}_1(j)$, возможно благодаря комплементарности операций `Rank` и `Select`. Например, на рис. 15.6 крайний правый узел дерева $\hat{T}$ на втором уровне имеет простую метку $j = 7$ и жирную метку $\mathbf{i} = \mathbf{6}$. Вы можете самостоятельно проверить на массиве B, что $7 = \texttt{Select}_1(\mathbf{6})$ и, наоборот, $\mathbf{6} = \texttt{Rank}_1(7)$.

Теперь все готово к реализации трех навигационных операций, упомянутых в начале этого раздела. Обратите внимание на то, что в расширенном бинарном дереве $\hat{T}$ у *внутреннего* узла с жирным номером $\mathbf{x}$ есть левый потомок с простым номером $2x$ и правый потомок с простым номером $2x + 1$. Это правило нумерации двоичных

куч, примененное к *полному* бинарному дереву $\hat{T}$. Вы легко можете в этом удостовериться, глядя на рис. 15.6. Кроме того, чтобы узнать, является ли узел в дереве $\hat{T}$ внутренним (исходным) узлом или фиктивным листом, достаточно проверить, какое значение имеет соответствующий бит $B[i]$ — 1 или 0, где i — обычная метка этого узла. У узла с обычной меткой 7 и жирной меткой **6** левый потомок имеет обычную метку $2 \times \mathbf{6} = 12$, а правый потомок — обычную метку $2 \times \mathbf{6} + 1 = 13$, более того, это внутренний узел дерева $\hat{T}$ и его бит $B[7] = 1$. Его левый потомок также является внутренним узлом, так как его бит $B[12] = 1$, тогда как правый потомок — это фиктивный лист, ведь его бит $B[13] = 0$.

Теперь у нас есть все алгоритмические ингредиенты для реализации трех основных навигационных операций над узлами входного дерева T за постоянное время.

1. **Левый потомок (left_child).** Воспользуемся приведенной ранее формулой для узла входного дерева T, помеченного жирным номером $\mathbf{x}$, и сначала вычислим обычную маркировку его левого дочернего элемента, то есть $2 \times \mathbf{x}$, а затем преобразуем ее в соответствующую жирную маркировку, вычислив $\texttt{left\_child}(\mathbf{x}) = \texttt{Rank}_1(2\mathbf{x})$.
2. **Правый потомок (right_child).** Аналогичным способом получается жирная метка для правого дочернего элемента — $\texttt{right\_child}(\mathbf{x}) = \texttt{Rank}_1(2\mathbf{x} + 1)$.
3. **Предок (parent).** Инвертировав процесс вычисления правого и левого дочерних элементов, получим жирную метку родителя — $\texttt{parent}(\mathbf{x}) = \lfloor \texttt{Select}_1(x)/2 \rfloor$.

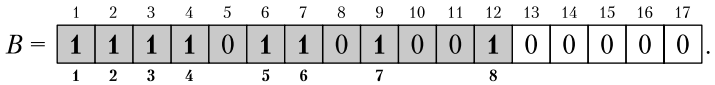
Как было отмечено ранее, алгоритм должен проверить, $B[2\mathbf{x}] = 0$ или $B[2\mathbf{x} + 1] = 0$. В этих случаях извлеченный потомок равен `NULL`. Предок имеет значение `NULL`, когда запрос выполняется над корнем дерева T, отсюда и узел с жирной меткой **1**. Таким образом, доказано следующее.

Теорема 15.5 *Двоичное дерево из n узлов можно представить в* $2n + 1 + o(n)$ *битах и с поддержкой запросов* `parent`, `left_child` и `right_child` *за постоянное время.*

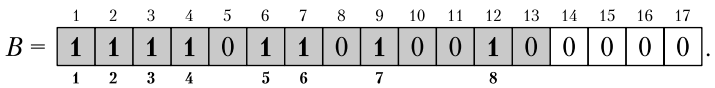
Обратите внимание на то, что это *краткое* представление достигает той же производительности запроса, что и представление на основе указателя, но с улучшением в $\Theta(\log n)$ раз по занимаемому пространству. На самом деле для перемещения вниз по дереву T нужно построить только структуру данных для $\texttt{Rank}_1$ — структура данных $\texttt{Select}_1$ требуется лишь для вычислений, связанных с узлами-предками. Наконец, отмечу, что такое краткое представление дерева можно обогатить вспомогательными данными, прикрепленными к узлам T, — достаточно сохранить массив $A[1, n]$, содержащий в элементах $A[i]$ вспомогательную информацию, которая связана с узлом с жирной маркировкой $\mathbf{i}$. Подобным образом можно управлять вспомогательной информацией, связанной с ребрами T, используя в качестве их индексного дескриптора целевой узел.

В заключение рассмотрим три описанные ранее навигационные операции на примере двоичного дерева, показанного на рис. 15.6. Вычислим левого потомка узла с жирной меткой **6** в дереве $\hat{T}$: $\texttt{left\_child}(\mathbf{6}) = \texttt{Rank}_1(2 \times \mathbf{6}) = \texttt{Rank}_1(12) = 8$. Такой

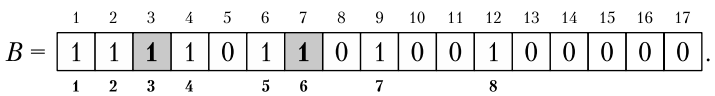
потомок существует, потому что $B[12] = 1$. Графически вычисление выглядит следующим образом:



Теперь вычислим правого потомка того же узла: `right_child(6)` = $\texttt{Rank}_1(2 \times 6 + 1) = \texttt{Rank}_1(13) = 6$. Такого потомка не существует, потому что $B[13] = 0$. Графически вычисление выглядит следующим образом:



Наконец, вычислим предка этого узла: $\texttt{parent}(6) = \lfloor \texttt{Select}_1(6)/2 \rfloor = \lfloor 7/2 \rfloor = 3$. Мы можем сделать вывод, что предок существует, потому что его индекс отличается от 0. Графически вычисление выглядит следующим образом:



15.2.2. Деревья произвольной формы

Рассмотрим другую реализацию деревьев, достаточно мощную для того, чтобы управлять упорядоченными деревьями произвольной степени и выполнять больший набор запросов за постоянное время, например получать сведения о предке, первом потомке (слева), следующем потомке (справа) и степени узла. Свойство упорядочения дает возможность ссылаться на первого и следующего за ним потомка. Решение называется LOUDS, что означает **L**evel **O**rder **U**nary **D**egree **S**equence — последовательность унарных степеней по уровням дерева. Оно базируется на интуитивном представлении о том, что дерево однозначно определяется своей последовательностью степеней, записанной в порядке BFS, и работает следующим образом.

1. **Развертка.** Добавляем фиктивный корневой узел степени 1.
2. **Метка со степенью узла.** Каждому узлу дерева назначаем метку с его степенью.
3. **Сериализация.** Обходим дерево по уровням слева направо (порядок BFS), кодируя последовательности степеней узлов в унарном виде и сохраняя их в двоичном массиве B.

Пример работы метода LOUDS для дерева со степенью не выше трех представлен на рис. 15.7. Рисунок позволяет сначала вывести границу для занимаемого этим представлением дерева пространства, то есть массива B, а затем получить два свойства массива B, лежащих в основе реализации четырех навигационных операций, поддерживаемых LOUDS.

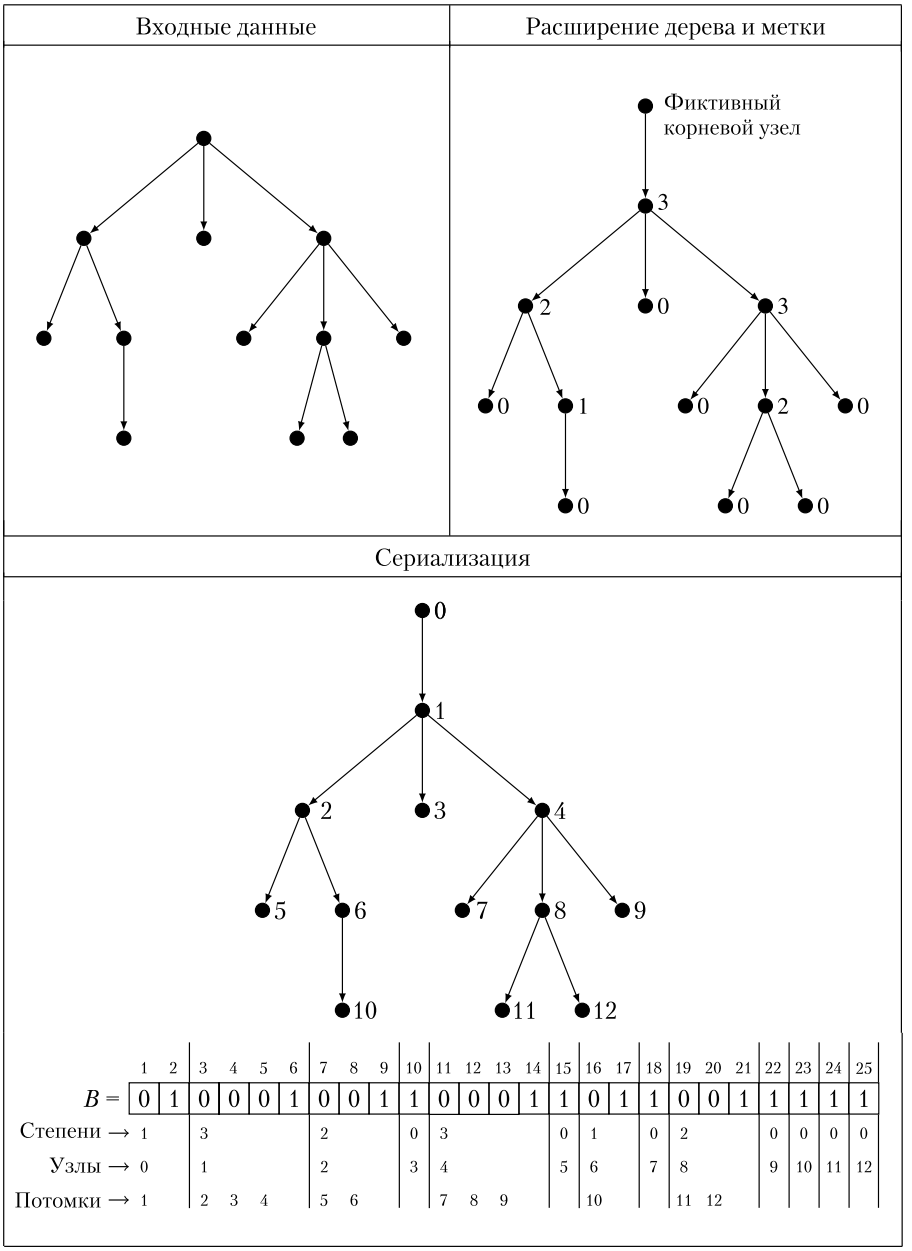


Рис. 15.7. Применение метода LOUDS к дереву, расположенному *вверху слева*. Вверху справа на том же дереве показан результат расширения и присваивания меток. Справа от каждого узла указана его степень. Результат сериализации показан в дереве *внизу*, снабженном нумерацией BFS для каждого узла. Ниже него приведен двоичный массив *B*, состоящий из последовательности унарных представлений степеней для каждого узла

Для наглядности на рис. 15.7 над B указаны индексы массива, а под ним — три списка. Список «Степени» содержит степени узлов, закодированные в унарных числах в массиве B, список «Узлы» содержит соответствующую нумерацию BFS узлов в дереве T, а список «Потомки» представляет для каждой записи со значением 0 в массиве B дочерний (-е) элемент (-ы) соответствующего узла из списка «Узлы».

Методу LOUDS требуется $2n + 1$ бит для кодирования структуры дерева, как и представлению Якобсона из предыдущего раздела. Доказательство осуществляется простым подсчетом: количество нулевых битов в B равно n, потому что унарное кодирование связывает нулевой бит с каждым потомком узла в расширенном дереве и, таким образом, со всеми его узлами, кроме фиктивного корня. Количество единичных битов в B равно $n + 1$, потому что каждый узел, включая фиктивный корень, генерирует унарное кодирование, которое заканчивается 1.

Из нумерации BFS узлов дерева и упорядоченной по BFS сериализации унарного кодирования их степеней нетрудно интуитивно понять корректность следующих двух свойств, связывающих биты массива B и узлы дерева.

Свойство 1. Узел номер k согласно обходу BFS соответствует k-му нулевому биту в массиве B.

Свойство 2. Потомки k-го узла соответствуют максимальной последовательности нулей, следующей за k-м единичным битом в массиве B.

Эти два свойства позволяют легко реализовать четыре навигационные операции за постоянное время. Напомню, что нумерация BFS узла совпадает с положением в массиве B соответствующего нулевого бита. Все вычисления будут рассматриваться на примере рис. 15.7.

Степень (deg). Степень узла с номером BFS x вычисляется по формуле $\texttt{deg}(x) = \texttt{Select}_1(x+1) - (\texttt{Select}_1(x) + 1)$ (здесь используется свойство 2). Например, $\texttt{deg}(4) = \texttt{Select}_1(5) - (\texttt{Select}_1(4) + 1) = 14 - (10 + 1) = 3$, поскольку:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$B =$	0	1	0	0	0	**1**	0	0	**1**	**1**	0	0	0	**1**	1	0	1	1	0	0	1	1	1	1	1

Предок (parent). Для узла с номером BFS x нумерация BFS его родителя вычисляется по формуле $\texttt{parent}(x) = \texttt{Rank}_1(\texttt{Select}_0(x)) = \texttt{Select}_0(x) - x$ (здесь сначала используется свойство 1, а затем свойство 2), например, $\texttt{parent}(5) = \texttt{Select}_0(5) - 5 = 7 - 5 = 2$, поскольку:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$B =$	**0**	1	**0**	**0**	**0**	1	**0**	0	1	1	0	0	0	1	1	0	1	1	0	0	1	1	1	1	1

Первый потомок (first_child). Из процедуры вычисления $\texttt{deg}(x)$ мы знаем, что степень узла с номером BFS x расположена между позициями $\texttt{Select}_1(x) + 1$ и $\texttt{Select}_1(x+1)$.

Поэтому при $\texttt{deg}(x) = 0$ возвращается значение `NULL`. В противном случае происходит переход к первому 0 этой последовательности унарной степени. В массиве B он находится в позиции $\texttt{Select}_1(x) + 1$. Нумерация BFS соответствующего узла возвращается с помощью операции $\texttt{Rank}_1$ (здесь используется свойство 2). Обратите внимание на то, что $\texttt{Rank}_0(\texttt{Select}_1(x) + 1) = \texttt{Select}_1(x) + 1 - x$. Таким образом, доказано, что:

$$\texttt{first\_child}(x) = \begin{cases} \texttt{NULL}, \text{ если } \texttt{deg}(x) = 0; \\ \texttt{Select}_1(x) + 1 - x \text{ в противном случае.} \end{cases}$$

Для примера найдем `first_child`(9). Так как $\texttt{deg}(9) = \texttt{Select}_1(9 + 1) - \texttt{Select}_1(9) - 1 = 22 - 21 - 1 = 0$, ответ `NULL`:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$B =$	0	**1**	0	0	0	**1**	0	0	**1**	**1**	0	0	0	**1**	**1**	0	**1**	**1**	0	0	**1**	**1**	1	1	1

А вот для `first_child`(8), так как $\texttt{deg}(8) = \texttt{Select}_1(8 + 1) - \texttt{Select}_1(8) - 1 = 21 - 18 - 1 = 2$, первый дочерний элемент узла, чья нумерация BFS равна 8, существует, и его нумерация BFS может быть вычислена с помощью $\texttt{first\_child}(8) = \texttt{Select}_1(8) + 1 - 8 = 18 + 1 - 8 = 11$:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$B =$	0	**1**	0	0	0	**1**	0	0	**1**	**1**	0	0	0	**1**	**1**	0	**1**	**1**	0	0	**1**	1	1	1	1

Следующий потомок (next_sibling). Для узла с нумерацией BFS x вычислим $y = \texttt{Select}_0(x)$ как соответствующий ему нулевой бит в массиве B (согласно свойству 1). Таким образом, если следующий бит $B[y + 1]$ равен 0, узел следующего потомка получает следующий номер BFS, то есть $x + 1$, в противном случае запрошенного узла не существует, то есть:

$$\texttt{next\_sibling}(x) = \begin{cases} x + 1, \text{ если } B\left[\texttt{Select\_0}(x) + 1\right] = 0; \\ \texttt{NULL} \text{ в противном случае.} \end{cases}$$

В качестве примера найдем `next_sibling`(12). Сначала вычислим $y = \texttt{Select}_0(12) = 20$, и так как $B[y + 1] = B[21] = 1$, получим ответ `NULL`, ведь следующего дочернего элемента не существует:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$B =$	**0**	1	**0**	**0**	**0**	1	**0**	**0**	1	1	**0**	**0**	**0**	1	1	**0**	1	1	**0**	**0**	1	1	1	1	1

Для определения `next_sibling`(11) вычислим $y = \texttt{Select}_0(11) = 19$, и так как $B[19 + 1] = B[20] = 0$, запрошенный элемент имеет нумерацию BFS 11 + 1 = 12:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$B =$	**0**	1	**0**	**0**	**0**	1	**0**	**0**	1	1	**0**	**0**	**0**	1	1	**0**	1	1	**0**	0	1	1	1	1	1

15.3. Компактное представление графов

Мы наконец-то пришли к последней теме книги, вероятно, одной из самых сложных из всех исследованных в последние годы из-за появления графовых баз данных, социальных сетей и графов знаний. Нас заваливают данными, и по большей части они связаны, что можно плодотворно смоделировать с помощью такой структуры данных, как *граф*. Этот раздел посвящен попыткам разработки компактных представлений графов, которые эффективно поддерживают некоторые варианты обхода. Если с узлами или ребрами графа связана дополнительная информация — так называемые *маркированные графы*, то для работы с ними используются дополнительные структуры данных, как было с маркированными деревьями, обсуждавшимися в предыдущем разделе.

Рассмотрим три ортогональных подхода к компактному представлению графов. В одном код Элиаса — Фано используется для компактной индексации возрастающих целочисленных последовательностей, описывающих списки смежности узлов. Второй задействует особенности веб-графов для получения сильно сжимаемых двоичных последовательностей. В третьем применяется сложный подход к компактной индексации, основанный на k^2-деревьях и регулярном разложении двоичной матрицы смежности на двоичные подматрицы.

Начнем с обозначений и терминологии. Входной граф обозначается $G = (V, E)$, где V — множество из n узлов графа, а E — множество из m ребер графа. Для наглядности предполагается, что узлы идентифицируются с помощью положительных целых чисел от 1 до n. Символ E перегрузим для обозначения также матрицы смежности графа, то есть $E[u, v] = 1$ тогда и только тогда, когда (u, v) является ребром G. Для заданного узла $u \in V$ узлы v, соединенные с u ребром графа (при $E[u, v] = 1$), называются *смежными узлами u*. Сортировка *в порядке возрастания* их целочисленных меток дает *список смежности u*. Классическое и наивное представление G состоит из кодирования узлов в списках смежности. Для этого каждому узлу требуется пространство $\Theta(\log n)$ бит, таким образом, хранение всего графа займет $\Theta(m \log n)$ бит.

Пример представления графа через его списки смежности представлен на рис. 15.8. Рисунок наглядно показывает, как использовать код Элиаса — Фано для сжатия списков смежности G и эффективного доступа к ним в связи с их расположением по возрастанию. Это уже обсуждалось в главе 11. Такой подход хорошо обоснован в случаях, когда невозможно доказать специальные свойства распределения целых чисел (идентификаторов узлов) в списках смежности. В этом случае пространственная сложность оценивается как $O(m(2 + \log n^2/m))$ бит, ведь у нас есть m бит, установленных в 1 для n^2 возможных записей E.

Далее рассмотрим особые графы, для которых можно использовать *специально разработанные* подходы к компактному представлению, добиваясь дополнительной экономии пространства по сравнению с кодировкой Элиаса — Фано.

Узел	Полустепень исхода	Дополнительные узлы
…	…	…
15	9	13, 15, 16, 17, 18, 19, 23, 24, 203
16	10	15, 16, 17, 22, 23, 24, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
…	…	…

Рис. 15.8. Наивное представление графов с помощью списков смежности

15.3.1. Графы, моделирующие структуру Интернета

Веб-графом называется направленный граф, в котором роль узлов играют веб-страницы, а ребра представляют собой гиперссылки между ними. Веб-страницы идентифицируются с помощью их URL-адресов, которые можно преобразовать в целые числа (nodeID), а их *реверсированные* URL-адреса упорядочиваются по алфавиту[1].

Как показано в [1], веб-графы удовлетворяют двум интересным свойствам: *локальности* и *сходства*. Первое означает, что большинство исходящих со страницы ссылок ведут на страницы, расположенные на одном хосте, а согласно второму две страницы одного и того же хоста имеют много общих исходящих ссылок. Интерпретируя эти два свойства в терминах nodeID, можно сделать вывод, что в соответствии со свойством *локальности* узел u часто указывает на узлы v, такие что $|u - v|$ мало. Часто мала и разница между узлами из одного списка смежности, потому что они могут исходить с одного хоста. Свойство *сходства* позволяет сделать вывод, что при маленьком $|u - v|$ списки смежности узлов u и v имеют много общих элементов.

Таким образом, свойство *локальности* предполагает, что для упорядоченных реверсированных URL-адресов списки смежности узлов с одного и того же хоста находятся близко друг к другу и показывают наличие *кластеров* целых чисел, возможно образуя *непрерывные возрастающие последовательности*. Как обсуждалось в главе 11, код Элиаса — Фано использует такой тип распределения целых чисел не самым лучшим образом. Куда лучше справляются простые γ- или δ-коды, а еще лучше с точки зрения занимаемого пространства — интерполяционный код, у которого, однако, нет эффективного доступа к отдельным элементам сжатых списков. Поэтому, когда не требуются резервные копии графа, лучше всего выбрать γ- и δ-коды, учитывая их интересный компромисс между хранением и доступом. Если не доказано особых свойств возрастающей последовательности nodeID, их можно применять в алгоритмах в качестве стратегии *экранирующего* сжатия.

---

[1] Это означает, что компоненты строки URL www.corriere.it/esteri/page.html записываются в обратном порядке — it.corriere.www/esteri/page.html, а затем сортируются по алфавиту. Такой подход позволяет сгруппировать URL-адреса, исходящие из одного хоста.

Свойство *сходства* дает возможность хранить веб-графы более эффективно [1]. Основная идея заключается в представлении списка смежности $L[x]$ узла x как модифицированной версии некоторого предыдущего, но близкого списка $L[y]$, называемого *списком ссылок*. Разность $x - y = r > 0$ называется *числом ссылок*. Равенство $r = 0$ обозначает, что $L[x]$ кодируется как есть, без сжатия ссылок относительно любого предыдущего $L[y]$. В этом случае для обеспечения различий между последовательными идентификаторами узлов можно использовать целочисленный кодер, такой как γ- или δ-коды. Решающее значение имеет выбор r, который осуществляется в пределах окна размером W. Чем больше W, тем выше коэффициент сжатия, но тем медленнее происходит сжатие и тем больше памяти этот процесс потребляет. Фактически значение W влияет на количество списков ссылок, которые перебирает алгоритм в поиске варианта, обеспечивающего наилучшее сжатие $L[x]$. Список $L[y]$ может быть, в свою очередь, сжат как модифицированная версия некоторого предыдущего списка $L[z]$, где $z < y$ и $y - z < W$ и т. д., тем самым создавая *цепочки ссылок* произвольной длины. В данном случае это влияет на эффективность декомпрессии. Чтобы можно было жертвовать занятостью пространства ради эффективности декомпрессии, авторы [1] ввели параметр R, который называется *максимальным количеством ссылок* и ограничивает набор списков ссылок в окне W теми, которые не создают цепочек длиннее R. Малое значение R, скорее всего, приведет к худшему сжатию, но к более короткому (случайному) времени доступа[1].

Теперь, чтобы применить к списку смежности $L[x]$ дифференциальное сжатие относительно его списка ссылок $L[y]$, строится двоичная последовательность $B[x]$ из $|L[y]|$ бит, каждый из которых указывает, является ли соответствующий элемент $L[y]$ также элементом $L[x]$. Если это так, соответствующий бит установлен в 1, в противном случае — в 0. Последовательность битов $B[x]$ называется *списком копий* $L[y]$. Из свойства *сходства* и малого значения r следует, что $L[x] \cap L[y]$ велико, поэтому в последовательности $B[x]$ многие элементы $L[x]$ представлены всего одним битом, установленным в 1. Более того, ожидается малый размер разности $L[x] \backslash L[y]$, и ее узлы, называемые *дополнительными узлами*, хранятся явно или сжимаются с помощью какого-нибудь целочисленного кодировщика, такого как γ- или δ-коды. Восстановление исходного списка смежности $L[x]$ выполняется слиянием списка дополнительных узлов со списком смежности $L[y]$, ограниченным элементами, установленными в 1 в $B[x]$. Пример дифференциального сжатия показан на рис. 15.9.

Внимательный читатель мог заметить, что списки копирования представляют собой чередующуюся последовательность максимальных наборов единиц или нулей. Это

---

[1] Теоретически обоснованный процесс управления списками и цепочками ссылок с хорошими коэффициентами сжатия наряду с быстрым и гибким доступом к сжатым спискам смежности (без полной распаковки всего графа) можно найти в описании нового представления графов Zuckerli [5]. Это масштабируемая система сжатия, предназначенная для больших реальных графов, опробованная на миллиардах узлов и ребер. В отличие от WebGraph здесь применяются передовые методы сжатия и новые эвристические алгоритмы графов, позволяющие добиться сокращения пространства до 30 % с использованием ресурсов для распаковки, сопоставимых с WebGraph.

означает, что их можно закодировать по следующей схеме: сохраняется начальный бит списка копирования, а затем длина каждого набора кодируется с помощью подходящего целочисленного компрессора. Тип набора (единицы или нули) сохранять не нужно, потому что наборы чередуются, а тип первого из них известен, так как его битовое значение хранится в начале сжатой последовательности битов. Полученная двоичная последовательность называется *блоком копирования*. Дополнительная экономия пространства достигается удалением из блоков копирования закодированной длины последнего набора, потому что ее можно узнать из другой доступной информации, например $|L[y]|$, хранящейся в `Outd`, а также из длины других наборов, доступных в блоках копирования. Познакомиться с другими приемами компактного кодирования можно в оригинальной статье [1].

Узел	Полустепень исхода	Ссылки	Список копий	Дополнительные узлы
…	…	…	…	…
15	9	0	-	13, 15, 16, 17, 18, 19, 23, 24, 203
16	10	1	011100110	22, 316, 317, 3041
17	0		-	-
18	5	3	111100000	50
…	…	…	…	…

Рис. 15.9. Представление списков смежности, приведенных на рис. 15.8, с помощью *списков копирования*. Узел 15 не подвергается сжатию ссылок (его поле «Ссылки» равно 0), тогда как у узлов 16 и 18 списки смежности сжаты относительно списка ссылок $L[15]$

На рис. 15.10 видно, что списку смежности $L[18]$ соответствует список копирования, начинающийся с набора единиц (информация в поле «Первый бит»). Этот набор имеет длину 4 (потому что первое закодированное целое число равно 4). А затем идет набор нулей длиной $|L[15]| - 4 = 9 - 4 = 5$. В этом легко убедиться, посмотрев на список копирования $L[18]$ на рис. 15.9. Для полноты рассмотрим список смежности $L[16]$: его блоки копирования равны 5, первый из них представляет собой набор нулей (информация в поле «Первый бит»), а последний несохраненный — тоже набор нулей (из-за чередования наборов 4 + 1) длиной $|L[15]| - 8 = 9 - 8 = 1$.

Узел	Полустепень исхода	Ссылки	Первый бит	Блоки копирования	Дополнительные узлы
…	…	…	…	…	…
15	9	0	-	-	13, 15, 16, 17, 18, 19, 23, 24, 203
16	10	1	0	1, 3, 2, 2	22, 316, 317, 3041
17	0	-	-	-	-
18	5	3	1	4	50
…	…	…	…	…	…

Рис. 15.10. Представление списков смежности с помощью блоков копирования. Для наглядности длины максимальных наборов единиц или нулей разделены запятыми

Эксперименты показали, что дополнительные узлы часто образуют возрастающие наборы последовательных целых чисел, которые называются *интервалами*. К ним можно применять два типа сжатия: интервал, длина которого превышает пороговое значение, кодируется через свой левый экстремум и свою длину, уже имеющую компактный вид. В противном случае целые числа интервала можно кодировать, принимая во внимание предыдущие интервалы или nodeIDs. Подробно эти варианты кодирования рассматривались в оригинальной публикации [1], где также показано, что веб-графы допускают сжатие до трех бит на ребро.

15.3.2. Обобщенные графы

В окончательной схеме сжатия для рассматриваемых здесь графов используются разреженность их матриц смежности и форма кластеризации их единичных элементов, которые обычно применяются для получения сжатого представления легко обходимых обобщенных графов [2, 4]. Предлагаемая схема базируется на k^2-арном дереве, известном также как k^2-дерево, основу которого составляет матрица смежности E входного графа.

Пусть граф состоит из n узлов и m ребер, соответственно, его матрица смежности E имеет размер $n \times n$, из которых только m элементов установлены в 1. Предполагается, что $n = k^h$. Если это не так, матрица дополняется в нижней правой части, пока ее ширина не достигнет наименьшей степени k, большей чем n. В этом случае матрица E будет состоять из $n^2 = k^{2h}$ двоичных элементов.

Исходная матрица E *логически присваивается* корню k^2-дерева. Затем ее разбивают ровно на k^2 квадратных подматриц $E_{1,1} \ldots E_{k,k}$, каждая из которых *логически присваивается* дочернему элементу корня, который, таким образом, получает k^2 дочерних элементов. Если соответствующая подматрица содержит хотя бы один бит, установленный в 1, дочерний элемент помечается как 1. В противном случае он помечается как 0. Узлы, помеченные нулем, соответствуют листьям k^2-дерева, тогда как узлы, помеченные единицей, рекурсивно разлагаются на k^2 дополнительных подматриц, которые становятся дочерними элементами разложенных узлов. Процесс останавливается, когда размер подматрицы достигает 1 (рис. 15.11).

При такой стратегии разложения высота k^2-дерева составляет $h = \Theta(\log kn)$, а количество его листьев не превышает n^2. Сбалансированным и полным k^2-дерево называется, когда матрица E заполнена единицами. Чем выше разреженность и кластеризованность единиц в E, тем меньше k^2-дерево, которое тем не менее содержит столько же путей длиной h, сколько единиц в матрице E, то есть m. Таким образом, сверху общее количество узлов ограничено как mhk^2, ведь на таких путях у любого узла будет k^2 потомков[1]. Очевидно, что чем больше k, тем менее глубоким

[1] Можно улучшить верхнюю границу числа узлов, если вспомнить, что на вершине k^2-дерева многие узлы являются общими для m путей. Уточненная формула для верхней границы будет выглядеть так: $mk^2\left(\log_{k^2}\frac{n^2}{m}+O(1)\right)$ бит.

получится дерево, но тем выше будет его коэффициент ветвления. Этот компромисс, обусловленный составом матрицы E, влияет на объем занимаемой памяти и производительность навигационных операций в k^2-дереве.

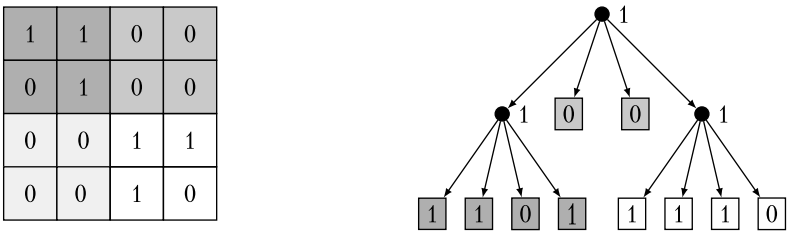


Рис. 15.11. Пример k^2-дерева для матрицы смежности 4 × 4, здесь $k = 2$. Высота дерева 2, его листья представлены квадратами, а внутренние узлы — точками. Внутренние узлы помечены значением 1, а листья — значением 0, если они соответствуют *нулевым* подматрицам, или значением 1, если соответствуют подматрице с одним битом, установленным в 1

Существование в графе ребра (u, v) легко проверить, перемещаясь по k^2-дереву. Нужно перейти от корня к дочернему элементу $E_{i,j}$ такому, что $i = \lceil (uk)/n \rceil$, $j = \lceil (vk)/n \rceil$, и задать новые u и v как $1 + ((u - 1) \bmod (n/k))$ и $1 + ((v - 1) \bmod (n/k))$ соответственно. Это вычисление рекурсивно повторяется (при этом новое n будет иметь значение n/k) для текущего посещаемого узла, пока не будет достигнут лист. Метка этого листа покажет, существует ли запрашиваемое ребро. Временная сложность в худшем случае составит $O(h)$.

Извлечь ряд или столбец сжатой матрицы немного сложнее, но это позволяет перемещаться по графу через его *прямые* и *обратные* ребра. Такой функционал не поддерживается в представлениях графа, описанных в предыдущих разделах, за исключением случаев, когда матрица смежности дополнена ее *транспонированной* версией, что требует в два раза большего пространства. Идея, послужившая основой *получения ряда* (столбец извлекается аналогичным образом) с помощью k^2-дерева, расширяет уже знакомую процедуру извлечения одной записи матрицы. Отличие в следующем: так как ряд может охватывать несколько узлов на разных уровнях, необходимо обойти много путей, после чего требуется заново объединить ответы, поступающие со всех них, то есть от всех достигнутых листьев.

Извлечение ряда 15 матрицы смежности графа с $n = 16$ узлами и $k = 2$ демонстрирует рис. 15.12. Нулевые подматрицы отделены жирными сегментами, и мы визуализируем исследуемую алгоритмом часть матрицы смежности. Обратите внимание на то, что первая половина запрашиваемого ряда доступна в дочернем элементе $E_{2,1}$, потому что она полностью нулевая. Для извлечения других записей необходимо пройти по путям на разной глубине, то есть ссылаться на подматрицы разного размера. На рисунке параметр p_i обозначает ряд, запрашиваемый на каждом рекурсивном уровне, начиная с $p_0 = 15$.

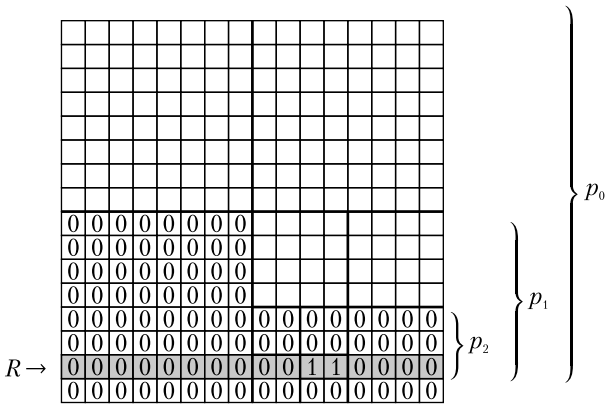


Рис. 15.12. Пример с $n = 16$ и $k = 2$. В этом случае $p_0 = 15$, $p_1 = 7$, $p_2 = 3$, $p_3 = 1$, $p_4 = 1$

Подсчитывая потомков в порядке BFS как {1, 2, 3, 4}, легко заметить, что на первом уровне посещаются два потомка (две подматрицы) корня — третий (равен нулю) и четвертый. На втором уровне посещаются два дочерних элемента четвертого потомка, которые снова имеют номера 3 и 4, причем четвертый равен нулю. На третьем уровне снова посещаются два потомка элементов с предыдущего уровня — третьего и четвертого, причем первый потомок равен нулю. Наконец, посещаются два потомка последнего элемента, имеющие метку 1.

k^2-деревья эффективно хранить с помощью представления LOUDS (см. подраздел 15.2.2), которое использует два массива: T — битовый массив для хранения меток всех внутренних узлов k^2-дерева, сериализованных в порядке BFS, и L — еще один битовый массив, хранящий метки самого нижнего уровня, упорядоченные справа налево. Для массива T создаются структуры данных $\texttt{Rank}_1$ и $\texttt{Select}_1$ для поддержки эффективной навигации по дереву. Массив L не индексируется, потому что у листьев отсутствуют потомки, то есть навигация прекращается.

В примере, приведенном на рис. 15.11, используются вот такие массивы T и L:

	1	2	3	4
$T =$	1	0	0	1

и

	1	2	3	4	5	6	7	8
$L =$	1	1	0	1	1	1	1	0

.

При наличии записи $T[i] = 1$, соответствующей внутреннему узлу k^2-дерева, позиция (согласно нумерации BFS) его j-го потомка (как внутреннего узла) определяется по формуле $\texttt{Rank}_1(T, i)k^2 + j$. Дело в том, что нужно учесть все k^2 потомков узлов слева (согласно нумерации BFS) от $T[i]$, а затем добавить запрошенное смещение потомка. Для асимптотической временной и пространственной производительности решения

на основе k^2-дерева наихудшая временная сложность извлечения из матрицы смежности ряда, полного единиц, составляет $O(n)$, ведь операция $\texttt{Rank}_1$ выполняется за постоянное время, а на уровне l в патологическом случае может потребоваться исследовать все k^l подматриц, покрывающих запрашиваемый ряд, пока не будет достигнут последний уровень с подматрицами размером 1×1, установленными в 1. В среднем можно доказать, что временная сложность оценивается как $O\left(\sqrt{m}\right)$[2]. Это означает, что чем более разрежена матрица, а соответственно, и граф, тем быстрее происходит декодирование одного ряда, а значит, и извлечение списка смежности узла индексированного графа тоже происходит быстрее. Более того, так как общий размер k^2-дерева оценивается как ограниченный сверху значением $mhk^2 = mk^2\log_k n$, это будет число битов (до членов низшего порядка), принимаемое компактным решением с использованием структур данных `Rank` и `Select`, построенных для двоичных массивов T и L. Более подробную информацию можно найти в литературе, например в [2], [4].

Список литературы

1. *Boldi P., Vigna S.* The Webgraph framework I: Compression techniques // Proceedings of the 13th International Conference on World Wide Web (WWW), 595–02, 2004.
2. *Brisaboa N. R., Ladra S., Navarro G.* k^2-trees for compact web graph representation // Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE), 8–30, 2009.
3. *Jacobson G.* Space-efficient static trees and graphs // Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS), 549–554, 1989.
4. *Navarro G.* Compact Data Structures: A Practical Approach. Cambridge University Press, 2016.
5. *Versari L., Comsa I.-M., Conte A., Grossi R.* Zuckerli: A new compressed representation for graphs // IEEE Access, 8, 219 233–219 243, 2020.

ЗАКЛЮЧЕНИЕ

> В конечном счете из наших исследований мы извлекаем только то, что будем применять на практике.
>
> *Иоганн Вольфганг фон Гёте*

Дочитав эту книгу, вы можете спросить: «Что дальше?» Я надеюсь, предыдущие 15 глав, несмотря на то что они лишь слегка коснулись алгоритмики, повысили ваш интерес к этой области и положительно повлияли на ваш академический или профессиональный подход к решению задач. Более того, я надеюсь, что после прочтения этих страниц вы согласитесь с фразой, которую я написал в предисловии: «Программирование по-прежнему остается искусством, но, чтобы выразить свои идеи максимально красиво, вам потребуются хорошие инструменты».

Как бы то ни было, хотелось бы рассказать об алгоритмических инструментах и вычислительных инфраструктурах, которые разработчикам алгоритмов придется изучать, проектировать и применять на практике в ближайшие годы.

Я подробно рассказывал про эволюцию компьютерной памяти, ее возросшую сложность, тип и количество. Все это побудило ввести *упрощенную* двухуровневую модель памяти, которая позволила довольно просто анализировать производительность предлагаемых алгоритмических решений, получая гораздо лучшее приближение, чем обеспечивает классическая RAM-модель. Но в ближайшие годы алгоритмы, оптимизированные по этой модели вычислений, вероятно, станут менее эффективными из-за эволюции инфраструктуры информационно-коммуникационных технологий и сложных задач, создаваемых новыми рабочими нагрузками с интенсивным использованием данных. Фактически в эпоху цифровой экономики данные стали основным ресурсом. Компании успешно справятся с цифровой трансформацией, если смогут разработать ориентированные на клиента и управляемые данными цифровые сервисы, которые все чаще предлагают ответы в реальном времени с помощью платформы цифровой инфраструктуры, сосредоточенной на больших данных. Нельзя забывать и о настойчивых просьбах исследователей, инженеров и аналитиков оцифровывать все большие объемы данных и предоставлять все больше вычислительных мощностей, чтобы ускорить получение результатов в ряде приложений. Это происходит в сферах автономного вождения, биологических и медицинских наук, энергетики, экономики и финансов, а также в передовых научных и инженерных исследованиях, и это лишь некоторые из примеров.

Само собой разумеется, что важную роль в продвижении этих инноваций продолжат играть устройства хранения данных, как ключевой компонент инфраструктуры высокопроизводительных вычислений. В частности, физически распределенная,

глобальная разделяемая память будет становиться все более важной для того, чтобы справляться с интенсивными рабочими нагрузками, особенно такими, как графическая аналитика, требующая нешаблонного доступа к памяти. По мере повсеместного распространения облачных архитектур следующего поколения приложениям будет требоваться все больше возможностей для перемещения вычислительных рабочих процессов через несколько контейнеров, каждый из которых динамически обеспечивается соответствующими аппаратными и программными ресурсами, возможно обогащенными некоторым встроенным интеллектом. Но, несмотря на впечатляющий прогресс в развертывании передовых инфраструктур для вычислений и хранения данных, который произошел в последние годы, в ближайшем будущем их наверняка станет еще больше. И понятно, что существующих аппаратных решений будет недостаточно для гарантированного доступа к ресурсам хранения и вычислений *в любом месте и в любой момент*, когда они потребуются.

В результате возрастет важность проектирования алгоритмов и структур данных, ведь именно они могут гарантировать достижения, далеко выходящие за рамки предсказанных законом Мура[1]. Но достижения возможны только при условии, что проектировщики алгоритмов обогатят свой инструментарий знаниями и навыками, которые могут базироваться на прочитанном в этой книге, добавив к этому методы и приемы из областей искусственного интеллекта (ИИ) и машинного обучения (МО), оптимизации и криптографии. Мотивацией тут должны служить те же устремления, которые в недавнем прошлом побудили проектировщиков и инженеров ввести в свои алгоритмические решения концепции из различных областей: области баз данных (генерация структур данных, эффективных с точки зрения ввода-вывода), биоинформатики (генерация поисковых систем генома), а совсем недавно еще и из сферы теории информации (разработка сжатых индексов, рассмотренных в главе 15).

Неудивительно, что в последние годы возник всплеск интереса, во-первых, к структурам данных и алгоритмам на базе ИИ/МО *(алгоритмы с предсказаниями)*, которые позволяют использовать распределение данных; во-вторых, к аппаратному обеспечению GPU/TPU, дающему возможность повышать производительность и проводить оптимизацию по *нескольким критериям*, а не только по времени и/или пространству; в-третьих, к работе с облачными вычислениями и хранилищами, предлагающими благодаря все более сложным криптографическим методам *устойчивость* к вызванной злыми намерениями (по внутренним и внешним причинам) утечке различных типов информации. В последнем сценарии ключевая проблема сводится к поддержке безопасных и эффективных операций поиска и добычи конфиденциальных данных. Эти данные шифруются и индексируются таким образом, чтобы избежать утечки шаблонов доступа к запросам, ответов на запросы и, конечно же, базовых индексированных данных. Для реализации в реальном времени *защищенных конфиденциальных вычислений* специалисты ищут новые эффективные способы объединения криптографических методов и методов работы со структурами

[1] Закон Мура (1965) прогнозировал, что количество транзисторов в микрочипе будет удваиваться примерно каждые 18 месяцев (см. https://ru.wikipedia.org/wiki/Закон_Мура).

данных, которые не только гарантируют безопасность с теоретической точки зрения, но и дают хорошую производительность на практике. Напоследок нужно упомянуть, что ИИ/МО и оптимизация по набору критериев также будут играть решающую роль в области сжатия данных, где цель состоит в повышении производительности инфраструктур HPC путем автоматического, динамического и эффективного определения наилучших подходов к сжатию, позволяющих работать в условиях ограниченных вычислительных ресурсов, которыми располагают современные приложения. Потребуются новые схемы кодирования, которые будут эксплуатировать не только классические повторения во входных данных, но и новые формы *закономерностей*. Эти закономерности не улавливаются существующими компрессорами, но могут быть идентифицированы с помощью правильно спроектированных и обученных моделей МО. Новые схемы кодирования потребуются и для сжатия *структурированных данных*, таких как матрицы и маркированные графы, которые генерируются приложениями ИИ/МО, графами знаний и базами данных графов. Это нужно, чтобы в конечном счете они смогли поддерживать арифметические операции/запросы непосредственно в своих сжатых версиях.

Без сомнения, в ближайшие годы разработчиков алгоритмов и инженеров ждет множество сложнейших задач.

Read IT Club

Комьюнити рецензентов
и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.

присоединиться к нам

Дэнис Ротман

RAG И ГЕНЕРАТИВНЫЙ ИИ. СОЗДАЕМ СОБСТВЕННЫЕ RAG-ПАЙПЛАЙНЫ С ПОМОЩЬЮ LLAMAINDEX, DEEP LAKE И PINECON

В книге описываются приемы создания эффективных больших языковых моделей, систем компьютерного зрения и генеративного ИИ, показывающих высокую производительность при относительно невысоких затратах. В ней приводится подробное исследование технологии RAG, а также подходов к проектированию мультимодальных пайплайнов ИИ и управлению ими. Связывая вывод с исходными документами, RAG повышает точность и контекстную релевантность результатов, предлагая динамический подход к управлению большими объемами информации.

Узнайте, как построить инфраструктуру RAG, попутно разобравшись с векторными хранилищами, фрагментацией, индексацией и ранжированием. Познакомьтесь с методами оптимизации производительности и приемами более глубокого изучения данных, включая использование адаптивного RAG и обратной связи от человека для уточнения поиска, тонкую настройку RAG, реализацию динамических RAG для поддержки принятия решений в реальном времени и визуализацию сложных данных с помощью графов знаний. Вы также увидите, как на практике объединить такие фреймворки, как LlamaIndex и Deep Lake, векторные базы данных наподобие Pinecone и Chroma и модели, предлагаемые компаниями Hugging Face и OpenAI. Приобретите навыки внедрения интеллектуальных решений, что повысит вашу конкурентоспособность в различных областях: от продакшна до обслуживания клиентов в любом проекте.

SPRINT book