С. П. Мальцев

ОЛИМПИАДНОЕ ПРОГРАММИРОВАНИЕ

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
БУРЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ ДОРЖИ БАНЗАРОВА

С. П. Мальцев

ОЛИМПИАДНОЕ ПРОГРАММИРОВАНИЕ

*Рекомендовано УМС БГУ
в качестве учебно-методического пособия для обучающихся
по направлениям подготовки 01.03.02 Прикладная математика
и информатика, 02.03.01 Математика и компьютерные науки,
02.03.03 Математическое обеспечение и администрирование
информационных систем, 09.03.03 Прикладная информатика*

Улан-Удэ
Издательство Бурятского госуниверситета
2019

УДК 004.421
ББК 22.183.49
М 215

Утверждено к печати
редакционно-издательским советом
Бурятского госуниверситета

Рецензенты

старший преподаватель кафедры информационных технологий,
заведующий лабораторией программных систем
Бурятского государственного университета
Б. В. Хабитуев

кандидат физико-математических наук, доцент, заведующий
кафедрой высшей математики и общеобразовательных дисциплин
Бурятского института инфокоммуникаций Сибирского
государственного университета телекоммуникаций и информатики
С. Г. Баргуев

Мальцев С. П.

М 215 **Олимпиадное программирование** : учебно-методическое пособие. — Улан-Удэ: Бурятского госуниверситета, 2019. — 135 с.
ISBN 978-59793-1396-2

В учебно-методическом пособии приведены некоторые алгоритмы компьютерной обработки данных и структуры, встречающиеся на олимпиадах по программированию.

Пособие предназначено для бакалавриата по направлениям подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем, 02.03.01 Математика и компьютерные науки, 01.03.02 Прикладная математика и информатика, 09.03.03 Прикладная информатика, 01.03.01 Математика.

© С. П. Мальцев, 2019
© Бурятский госуниверситет
им. Д. Банзарова, 2019

ISBN 978-59793-1396-2

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ.. 6

1. ОБ ОЛИМПИАДАХ И ОЛИМПИАДНОМ ПРОГРАММИРОВАНИИ 8

Контрольные вопросы к параграфу 10

2. СИСТЕМА АВТОМАТИЧЕСКОЙ ПРОВЕРКИ РЕШЕНИЙ EJUDGE 11

Контрольные вопросы к параграфу 15

3. ТЕОРЕТИЧЕСКИЙ МИНИМУМ. ОСНОВЫ ЯЗЫКА С++ 17

Контрольные вопросы к параграфу 39

4. ГЕНЕРАТОР ТЕСТОВ ДЛЯ НАПИСАНИЯ КОМПЛЕКТОВ ЗАДАЧ. 40

Контрольные вопросы к параграфу 44

5. ПРИМЕРЫ АЛГОРИТМОВ КОМПЬЮТЕРНОЙ ОБРАБОТКИ ДАННЫХ. 45

5.1 Алгоритм Евклида нахождения НОД (наибольшего общего делителя) 45

5.2 Решето Эратосфена.. 46

5.3 Числа Фибоначчи 47

5.4 Поиск в ширину... 50

5.5 Поиск в глубину... 54

5.6Топологическая сортировка.. 56

5.7 Алгоритм поиска компонент связности в графе 58

5.8 Поиск мостов 59

5.9 Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры для разреженных графов 61

5.10 Алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин 64

5.11 Минимальное остовное дерево. Алгоритм Крускала с системой непересекающихся множеств 66
5.12 Нахождение Эйлерова пути за O (M) 68
5.13 Алгоритм Куна нахождения наибольшего паросочетания в двудольном графе 69
5.14 Знаковая площадь треугольника и предикат "По часовой стрелке" 76
5.15 Пересечение двух отрезков 77
5.15 Пересечение окружности и прямой...................................... 80
5.16 Построение выпуклой оболочки обходом Грэхэма............... 83
5.17 Префикс-функция. Алгоритм Кнута-Морриса-Пратта............ 85
5.18 Алгоритмы хэширования в задачах на строки 89
5.19 Алгоритм Ахо-Корасик... 90
5.20 Sqrt-декомпозиция 96
5.21 Система непересекающихся множеств 99
5.22 Дерево отрезков 107
5.23 Нахождение наибольшей нулевой подматрицы 113
5.24 Числа Каталана... 117
6. ЛАБОРАТОРНЫЕ РАБОТЫ 120
ЗАКЛЮЧЕНИЕ... 125
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.. 126
ПРИЛОЖЕНИЕ... 130
Приложение №1. Примерное содержание файла statement.xml 130
Приложение №2. Пример файла statement.xml конкретной задачи 130

Приложение №3. Пример решения задачи из приложения 2. ..132
Приложение №4. Пример чекера для задачи из приложения №2 133
Приложение №5. Шаблон отчёта по лабораторной работе.......134
Приложение №6. Критерии оценки лабораторной работы 134

ПРЕДИСЛОВИЕ

Настоящее учебное издание представляет собой учебно-методическое пособие для дисциплин «Олимпиадное программирование», «Курс по программированию», «Дискретная математика и теория графов», «Спортивное программирование», «Структуры и алгоритмы компьютерной обработки данных» в рамках реализации образовательной программы высшего образования по направлениям подготовки бакалавров 02.03.03 Математическое обеспечение и администрирование информационных систем, 02.03.01 Математика и компьютерные науки, 01.03.02 Прикладная математика и информатика, 09.03.03 Прикладная информатика очной и заочной формы обучения и подготовлено в соответствии с требованиями Федерального государственного образовательного стандарта высшего образования.

Компетенции обучающегося, формируемые в результате освоения дисциплины:

Данная дисциплина способствует формированию следующих компетенций, предусмотренных ФГОС ВО 3+ по направлениям подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем, 02.03.01 Математика и компьютерные науки, 01.03.02 Прикладная математика и информатика, 09.03.03 Прикладная информатика: способность программировать приложения и создавать программные прототипы решения прикладных задач (ПК-8).

В результате освоения дисциплины студент должен:

Знать: основные идиомы разработки алгоритмов; основные структуры данных, используемые для представления типовых информационных объектов (STL); основные алгоритмы и характеристики их сложности для типовых задач, часто встречающихся и ставших «классическими» в области информатики

Уметь: доказывать корректность составленного алгоритма и оценивать основные характеристики его сложности; реализовывать алгоритмы и используемые структуры данных средствами языков программирования высокого уровня; экспериментально (с

помощью компьютера) исследовать эффективность алгоритма и программы;
Владеть: некоторыми математическими методами анализа алгоритмов; классификации алгоритмических задач по их сложности, сводимости алгоритмических задач к известным задачам определенного класса сложности.

Основной задачей настоящего учебно-методического пособия систематизация и практическая реализация знаний в рамках дисциплин «Олимпиадное программирование».

Пособие состоит из 6 глав и приложений:

Первые четыре главы включают базовые понятия олимпиадного программирования и основы языка C++.

В пятой главе идет описание некоторых алгоритмов компьютерной обработки данных, встречающихся на олимпиадах по программированию.

В шестой главе представлены лабораторные работы по алгоритмам из предыдущей главы.

В приложении приведены некоторые файлы автоматической системы проверки решений и критерии оценивания лабораторных работ.

Желаем успехов!

1. ОБ ОЛИМПИАДАХ И ОЛИМПИАДНОМ ПРОГРАММИРОВАНИИ

Олимпиада по программированию – это интеллектуальное соревнование по решению различных задач на компьютере, для решения которых требуется применить алгоритм или написать программу. Обычно участникам олимпиады выдается комплект из нескольких задач. Задача считается решенной, если участнику удалось составить программу, которая правильно работает на всех подготовленных тестах. Сами же тесты участникам не доступны, а проверка осуществляется автоматически в проверяющей системе.

Олимпиады бывают личные и командные. В командных олимпиадах обычно участвует 3 человека и им на всё время олимпиады предоставляется 1 компьютер для решения задач. Для проведения подобных соревнований используются специализированные программные турнирные системы.

Крупнейшая международная студенческая командная олимпиада по программированию — ACM International Collegiate Programming Contest. Генеральными спонсорами чемпионата выступают такие компании как Microsoft и IBM. В 2004 году в ней участвовало 3150 команд из 75 стран. Команды из России неоднократно становились победителями этого престижного соревнования.

Первая олимпиада по программированию в СССР (под названием олимпиада по информатике) прошла среди школьников и состоялась в 1988 году в Свердловске. В дальнейшем олимпиады по информатике стали частью всесоюзных (а после распада СССР — всероссийских) предметных олимпиад школьников. Также проводятся многоуровневые командные олимпиады среди школьников, по правилам аналогичным правилам международных студенческих олимпиад.

Олимпиады по информатике среди студентов СССР в масштабах всей страны не проводились. Начиная с 1996 года студенты российских вузов начали участвовать в соревнованиях, входящих в систему командного чемпионата мира по программированию среди студентов, проводимого американской ассоциацией ACM. Также энтузиастами организовывались различные внутри- и межвузовские олимпиады. Обычно эти

олимпиады проводятся при финансировании какой-либо фирмы, занимающейся разработкой программного обеспечения и заинтересованной в привлечении талантливых студентов на работу к себе.

В последнее время появился более общий термин «спортивное программирование». Состязания по спортивному программированию не связаны напрямую с системой образования, то есть в них также принимают участие и профессиональные программисты. Популярное состязание по спортивному программированию в мире — ресурс TopCoder, на котором регулярно проводят раунды (SRM), по результатам которых формируется рейтинг участников, а также ежегодный TopCoder Open. Появившийся позднее российский ресурс Codeforces также проводит регулярные раунды, по результатам которых также формируется свой рейтинг. Крупные ИТ-компании проводят регулярные и как правило личные соревнования по программированию, среди таковых — Google Code Jam, Facebook Hacker Cup, Russian Code Cup.

В Бурятском государственном университете, помимо Республиканской студенческой олимпиады между командами студентов вузов г. Улан-Удэ, также проходит подготовка к чемпионату мира ACMICPC. Так 22 сентября 2018 года проходил квалификационный тур Восточно-Сибирского четвертьфинала чемпионата ACM ICPC 2018 . В турнире приняли участие студенты БГУ, ВСГУТУ, СФУ (Красноярск), ИГУ (Иркутск), ХГУ (Абакан) и других университетов, а также школьные команды (вне зачёта).

Бурятский государственный университет представляли 31 студент Института математики и информатики (11 команд). За первое место боролись команды Иркутского государственного университета (Irkutsk SU 4: Mindtravellers) и Команда БГУ (Buryat SU 1), но на последней задаче команда БГУ уступила конкурентам. Третье место заняла команда Красноярского СФУ (ISIT SFU 3).

По результатам турнира вторая команда БГУ (Buryat SU 2) заняла 13 итоговое место с 6 решёнными задачами из 8, три команды решили 5 из 8, четыре команды БГУ решили 4 задачи из 8.

Особенностью олимпиадных задач является художественность их условия. В условиях редко ведётся речь о структурах данных и алгоритмах, приводящих к решению. Чаще

условие задачи представляет собой короткий рассказ со своим сюжетом, героями и конфликтом. Таким образом, чтобы решить олимпиадную задачу, нужно предварительно составить математическую модель событий, и уже по ней подобрать или построить подходящий алгоритм. Алгоритм может быть как одним из уже известных, так и абсолютно новым, непохожим на другие.

Контрольные вопросы к параграфу

1. Что такое олимпиадное программирование.
2. Какие олимпиады по программированию вы знаете.

2. СИСТЕМА АВТОМАТИЧЕСКОЙ ПРОВЕРКИ РЕШЕНИЙ EJUDGE

Ejudge – это система для проведения различных мероприятий, в которых необходима автоматическая проверка программ. Система может применяться для проведения олимпиад и поддержки учебных курсов. Кроме этого система распространяется под лицензией GPL, имеет многоязычный веб-интерфейс и поддерживает защищённое исполнение программ (если установлен патч к ядру Linux). Также система активно используется для проведения олимпиад в различных учебных заведениях.

Для проведения олимпиады необходимо зарегистрировать участников: лично или командно. Далее нужно дать участникам возможность читать условия задач и отправлять решения на тестирование. Перед отсылкой на тестирование участник выбирает компилятор и файл с исходным кодом решения. Далее система на сервере пытается скомпилировать решение с помощью выбранного компилятора. Если произошли ошибки, то участнику выдаётся сообщение. Если компиляция прошла успешно, то происходит непосредственно тестирование. Исполняемому файлу на вход (STDIN или файл) подаются входные данные, заранее сформированные автором задачи. На выполнение обычно ставятся ограничения по времени и по памяти. Если решение участника уложилось в лимиты и выдало ответ, то система сверяет этот ответ с авторским. Кроме того, система должна вести статистику, показывать положение участников.

Установка и настройка Ejudge подробно описана на сайте проекта, также есть пошаговые инструкции. Через пакетный менеджер дистрибутива нужно установить необходимые компиляторы, Ejudge при конфигурировании автоматически их подхватит. Так же перед установкой нужно указать пути директорий турниров, веб-сервера и так далее.

Подготовка олимпиадных задач для ejudge

Краткое руководство по подготовке задач олимпиад по программированию для использования в системе ejudge. В руководстве будут рассматриваться только задачи для олимпиад по

правилам ACM ICPC или аналогичным. Обычно в ejudge олимпиадная задача состоит из:

1. Файл условия (statement.xml)
2. Решение жюри (на каком-либо из разрешенных на соревновании языке)
3. Набор тестов - пары текстовых файлов, содержащие входные и выходные данные
4. Чекер - программа, проверяющая корректность решения участника
5. Конфигурация в настройках контеста (serve.cfg)

Условие

Условие задачи хранится в файле statement.xml. Примерное его содержание показано в приложении №1.

Параметры в теге problem, по документации ejudge, не используются. Можно записать любые значения, например, в id номер задачи (A), а в type – standard. В тегах title, description, input_format, output_format, notes текст должен быть в формате XHTML (т.е. корректность тегов строго проверяется). Любой из этих тегов может отсутствовать

1. title содержит название задачи
2. description содержит основную часть условия задачи
3. input_format содержит подробное описание формата входных данных. Для параметров должны быть указаны диапазоны значений. Также здесь рекомендуется обращать внимание участников на разделители параметров (одиночные пробелы, произвольное количество пробелов, переводы строк и т.п.)
4. output_format содержит подробное описание формата выходных данных
5. notes содержит примечания к условию

Для лучшего отображения рекомендуется абзацы текста в тегах description, input_format, output_format, notes оборачивать в теги <p>

Для вставки изображения в условие нужно поместить его файл в каталог задачи и добавить в условие

examples содержит примеры входных и выходных данных. Каждый пример описывается тегом example, тег input содержит пример входных данных,output - выходных. Данные в тегах input и output будут отображаться с сохранением форматирования, обращайте внимание на лишние пробелы. Пример statement.xml реальной задачи приведен в приложении №2.

Решение жюри

Решение задачи, предоставленное жюри должно проходить все тесты, соответствовать ограничениям по времени и памяти, а также полностью соответствовать правилам олимпиады (если правилами накладываются какие-либо особые ограничения). Также рекомендуется, чтобы решение было написано на языке, разрешенном на олимпиаде. Пример решения (для задачи, описанной выше) приведен в приложении №3.

Набор тестов

Тест - это два текстовых файла, содержащих корректные входные и выходные данные задачи. Рекомендуется, чтобы:

1. Выходные данные содержали вывод решения жюри
2. Первые тесты соответствовали тестам из приверов в условии
3. Входные данные соответствовали рекомендациям polygon:
    a. каждая строка завершается EOLN (переводом строки);
    b. не содержат символы с кодами меньше 32;
    c. не содержат начальных или конечных пробелов;
    d. не содержат два подряд идущих пробела;
    e. не содержат начальных или конечных пустых строк;
    f. файл не пустой.

Файлы тестов можно поместить в каталог tests, файлы входных данных назвать номером теста, файлы выходных данных - номером теста и расширением .a(расположение и наименование можно изменить в настройках контеста).

Чекер

Чекер - программа, проверяющая корректность решения участника, использующая три файла: входные данные, выходные данные жюри, выходные данные участника.
Проверка может закончиться с одним из результатов:

1. OK - решение принято
2. WA (Wrong Answer) - решение неправильное
3. PE (Presentation Error) - нарушенформатвывода
4. FAIL - проверка завершилась с ошибкой

Для написания чекеров обычно используется библиотека testlib, предоставляющая многие полезные методы. При использовании библиотеки, нужно ее код (testlib.h) также включить в файлы задачи.
testlib предоставляет несколько стандартных чекеров, например:

1. fcmp.cpp - Lines, doesn't ignore whitespaces
2. hcmp.cpp - Single huge integer
3. lcmp.cpp - Lines, ignore whitespaces
4. ncmp.cpp - Single or more int64, ignores whitespaces
5. rcmp4.cpp - Single or more double, max any error 1e-4
6. rcmp6.cpp - Single or more double, max any error 1e-6
7. rcmp9.cpp - Single or more double, max any error 1e-9
8. wcmp.cpp - Sequence of tokens
9. yesno.cpp - Single yes or no, case insensitive

Пример чекера для задачи, описанной выше приведен в приложении №4.

Настройки контеста

Каждой задаче соответствует раздел в настройках контеста (файл serve.cfg). Для задания общих настроек для всех задач используются “абстрактные” задачи. Примернастройкитакойзадачи:

```
[problem]
abstract
short_name = "PskovGeneric"
use_stdin
use_stdout
use_corr
xml_file = "statement.xml"
test_pat = "%02d"
corr_pat = "%02d.a"
```

```
time_limit = 5
real_time_limit = 10
max_vm_size = 256M
max_stack_size = 256M
max_file_size = 256M
check_cmd = "check"
```

Здесь определены настройки:
Имя задачи PskovGeneric
Используются стандартные потоки ввода и вывода
Условие хранится в файле statement.xml
Имя входных данных теста - его номер (не менее двух символов с ведущим нулем при необходимости)
Имя выходных данных теста - его номер (аналогично входным данным) и расширение .a
Ограничение на процессорное время выполнения 5 секунд, ограничение астрономического времени 10 секунд
Ограничение памяти 256МБ
Далее описываются настройки конкретных задач.
Пример настройки для задачи, описанной выше:

```
[problem]
super = PskovGeneric
internal_name = "rot13"
short_name = "B"
long_name = "Шифр ROT13"
```

Здесь определены настройки:
Базовая абстрактная задача PskovGeneric (описанная выше)
Внутреннее имя rot13 (в каталоге с таким именем задача будет хранится на сервере ejudge)
Краткое имя задачи B
Длинное имя задачи Шифр ROT13

Контрольные вопросы к параграфу

1. Опишите правила оформления условия для задачи в системе ejudge.
2. Опишите правила составления тестов для задачи в системе ejudge.

3. Что такое чекер, приведите пример стандартного чекера.
4. Опишите основные настройки контеста в системе ejudge.

3. ТЕОРЕТИЧЕСКИЙ МИНИМУМ. ОСНОВЫ ЯЗЫКА C++

C++ компилируемый язык программирования общего назначения, сочетает свойства как высокоуровневых, так и низкоуровневых языков программирования. В сравнении с его предшественником, языком программирования Си, наибольшее внимание уделено поддержке объектно-ориентированного и обобщённого программирования. Название «язык программирования C++» происходит от языка программирования C, в котором унарный оператор ++ обозначает инкремент переменной.

Язык программирования C++ широко используется для разработки программного обеспечения. А именно, создание разнообразных прикладных программ, разработка операционных систем, драйверов устройств, а также видео игр и многое другое. Существует несколько реализаций языка программирования C++ — как бесплатных, так и коммерческих. Их производят проекты: GNU, Microsoft и Embarcadero (Borland). Проект GNU — проект разработки свободного программного обеспечения (СПО).

Язык программирования C++ был создан в начале 1980-х годов, его создатель сотрудник фирмы Bell Laboratories — Бьёрн Страуструп. Он придумал ряд усовершенствований к языку программирования C, для собственных нужд. Т. е. изначально не планировалось создания языка программирования C++. Ранние версии языка C++, известные под именем «Си с классами», начали появляться с 1980 года. Язык C, будучи базовым языком системы UNIX, на которой работали компьютеры фирмы Bell, является быстрым, многофункциональным и переносимым. Страуструп добавил к нему возможность работы с классами и объектами, тем самым зародил предпосылки нового, основанного на синтаксисе C, языка программирования. Синтаксис C++ был основан на синтаксисе C, так как Бьёрн Страуструп стремился сохранить совместимость с языком C.

В 1983 году произошло переименование языка из «Си с классами»в «язык программирования C++». В него были добавлены новые возможности: виртуальные функции, перегрузка функций и операторов, ссылки, константы и многое другое. Его первый коммерческий выпуск состоялся в октябре 1985 года. Язык

программирования C++ является свободным, то есть никто не обладает на него правами.

Совсем не обязательно знать какой-то другой язык программирования. C++ является простым и понятным языком, все благодаря его удобному синтаксису. Конечно, для начинающих программистов, часть кода написана на C++, может быть менее понятной, чем эквивалентный код, написанный на другом языке. Это связано с тем, что C++ интенсивно использует специальные символы ({} [] * & |! и т. д.), вместо интуитивно понятных английских слов. Специальные символы просто необходимо один раз запомнить и всю жизнь пользоваться. Кроме того, упрощение интерфейса ввода/вывода в C++ по сравнению с языком Си, а также стандартная библиотека шаблонов (STL), делают обмен и манипулирование данными в программе достаточно простыми операциями. При этом язык C++ не теряет свою мощность.

Что такое объектно-ориентированное программирование (ООП)? Это модель программирования (парадигма), основная концепция которой — рассматривать каждый компонент в программировании как объект, со своими свойствами и методами. Данная парадигма является заменой структурного программирования, где акцент был сделан на процедуры.

Что такое ANSI-C++? ANSI-C++ это стандарт языка C++, который был составлен и опубликован международными организациями стандартизации ANSI/ISO. Но прежде, чем это стандарт был опубликован, C++ уже широко использовался, и поэтому существует много кода, который не соответствует стандарту ANSI-C++. Этот стандарт был опубликован в 1998 году, после, в стандарт были внесены дополнения в 2003 году.

Информация

Бит — это минимальная единица измерения объёма информации, так как она хранит одно из двух значений — 0 (False) или 1 (True). False и True в переводе на русский ложь и истина соответственно. То есть одна битовая ячейка может находиться одновременно лишь в одном состоянии из возможных двух. Напомню, два возможных состояния битовой ячейки равны — 1 и 0. Есть определённые операции, для манипуляций с битами. Эти операции называются логическими или булевыми операциями, названные в честь одного из математиков — Джорджа Буля (1815-

1864), который способствовал развитию этой области науки. Все эти операции могут быть применены к любому биту, независимо от того, какое он имеет значение — 0(нуль) или 1(единицу). Ниже приведены основные логические операции и примеры их использования.

Логическая операция И (AND). Обозначение AND: &

Логическая операция И выполняется с двумя битами, назовем их a и b. Результат выполнения логической операции И будет равен 1, если a и b равны 1, а во всех остальных (других) случаях, результат будет равен 0. Смотрим таблицу истинности логической операции and.

a(бит 1)	b(бит 2)	a(бит 1) & b(бит 2)
0	0	0
0	1	0
1	0	0
1	1	1

Логическая операция ИЛИ (OR). Обозначение OR: |

Логическая операция ИЛИ выполняется с двумя битами (a и b). Результат выполнения логической операции ИЛИ будет равен 0, если a и b равны 0 (нулю), а во всех остальных (других) случаях, результат равен 1 (единице). Смотрим таблицу истинности логической операции OR.

a(бит 1)	b(бит 2)	a(бит 1) | b(бит 2)
0	0	0
0	1	1
1	0	1
1	1	1

Логическая операция исключающее ИЛИ (XOR). Обозначение XOR: ^

Логическая операция исключающее ИЛИ выполняется с двумя битами (a и b). Результат выполнения логической операции XOR будет равен 1 (единице), если один из битов a или b равен 1 (единице), во всех остальных случаях, результат равен 0 (нулю). Смотрим таблицу истинности логической операции исключающее ИЛИ.

a(бит 1)	b(бит 2)	a(бит 1) ^ b(бит 2)
0	0	0
0	1	1
1	0	1
1	1	0

Логическая операция НЕ (not) Обозначение NOT: ~

Логическая операция НЕ выполняется с одним битом. Результат выполнения этой логической операции напрямую зависит от состояния бита. Если бит находился в нулевом состоянии, то результат выполнения NOT будет равен единице и наоборот. Смотрим таблицу истинности логической операции НЕ.

a(бит 1)	~a(отрицание бита)
0	1
1	0

Введение в язык C++

Язык C++ представляет собой набор команд, которые говорят компьютеру, что необходимо сделать. Этот набор команд, обычно называется исходный код или просто код. Командами являются или «функции» или «ключевые слова». Ключевые

слова(зарезервированные слова C/C++) являются основными строительными блоками языка. Функции являются сложными строительными блоками, так как записаны они в терминах более простых функций — вы это увидите в нашей самой первой программе, которая показана ниже. Такая структура функций напоминает содержание книги. Содержание может показывать главы книги, каждая глава в книге может иметь своё собственное содержание, состоящее из пунктов, каждый пункт может иметь свои подпункты. Хотя C++ предоставляет много общих функций и зарезервированных слов, которые вы можете использовать, все-таки возникает потребность в написании своих собственных функций.

В какой же части программы начало? Каждая программа в C++ имеет одну функцию, её называют главная или main-функция, выполнение программы начинается именно с этой функции. Из главной функции, вы также можете вызывать любые другие функции, неважно, являются ли они написанными нами, или, как упоминалось ранее, предоставляются компилятором.

Так как же получить доступ к этим Стандартным функциям? Чтобы получить доступ к стандартным функциям, которые поставляются с компилятором, необходимо подключить заголовочный файл используя препроцессорную директиву — #include. Почему это эффективно? Давайте посмотрим на примере рабочей программы:

```
1	#include<iostream>
2	usingnamespace std;
3
4	int main()
5	{
6		cout <<"Моя первая программа на C++\n";
7		cin.get();
8	}
```

Рассмотрим подробно элементы программы. #include это директива «препроцессору», которая сообщает компилятору поместить код из заголовочного файла iostream в нашу программу перед тем как создать исполняемый файл. Подключив к программе заголовочный файл вы получаете доступ к множеству различных функций, которые можете использовать в своей программе.

Например, оператору cout требуется iostream. Строка using namespace std; сообщает компилятору, что нужно использовать группу функций, которые являются частью стандартной библиотеки std. В том числе эта строка позволяет программе использовать операторы, такие как cout. Точка с запятой является частью синтаксиса C++. Она сообщает компилятору, что это конец команды. Чуть позже вы увидите, что точка с запятой используется для завершения большинства команд в C++.

Следующая важная строка программы int main(). Эта строка сообщает компилятору, что есть функция с именем main, и что функция возвращает целое число типа int. Фигурные скобки { и } сигнализируют о начале { и конце } функции. Фигурные скобки используются и в других блоках кода, но обозначают всегда одно — начало и конец блока, соответственно.

В C++ объект cout используется для отображения текста (произносится как «Си аут»). Он использует символы <<, известные как «оператор сдвига», чтобы указать, что отправляется к выводу на экран. Результатом вызова функции cout << является отображение текста на экране. Последовательность \n фактически рассматривается как единый символ, который обозначает новую строку (мы поговорим об этом позже более подробно). Символ \n перемещает курсор на экране на следующую строку. Опять же, обратите внимание на точку с запятой, её добавляют в конец, после каждого оператора C++.

Следующая команда cin.get(). Это еще один вызов функции, которая считывает данные из входного потока данных и ожидает нажатия клавиши ENTER. Эта команда сохраняет консольное окно от закрытия, до тех пор, пока не будет нажата клавиша ENTER. Это даёт вам время для того, чтобы посмотреть результат выполнения программы.

По достижении конца главной функции (закрывающая фигурная скобка), наша программа вернёт значение 0 для операционной системы. Это возвращаемое значение является важным, поскольку, проанализировав его, ОС может судить о том, успешно завершилась наша программа или нет. Возвращаемое значение 0 означает успех и возвращается автоматически (но только для типа данных int, другие функции, требуют вручную возвращать

значение), но, если бы мы хотели вернуть что-то другое, например 1, мы должны были бы сделать это вручную.

```
1	#include<iostream>
2
3	usingnamespace std;
4
5	intmain()
6	{
7		cout <<"Моя первая программа на C++\n";
8		cin.get();
9		return 0;
10	}
```

Обязательно комментируйте свои программы!

Добавляйте комментарии к коду, чтобы сделать его понятнее не только для себя, но и для других. Компилятор игнорирует комментарии при выполнении кода, что позволяет использовать любое количество комментариев, чтобы описать реальный код. Чтобы создать комментарий используйте или //, который сообщает компилятору, что остальная часть строки является комментарием или /* и затем */. Когда вы учитесь программировать, полезно иметь возможность комментировать некоторые участки кода, для того, чтобы увидеть, как изменяется результат работы программы. Подробно прочитать о технике комментирования, вы можете тут.

Что делать со всеми этими типами переменных?

Иногда это может сбить с толку — иметь несколько типов переменных, когда кажется, что некоторые типы переменных являются избыточными. Очень важно использовать правильный тип переменной, так как некоторым переменным, требуется больше памяти, чем другим. Кроме того, из-за способа хранения в памяти, числа с плавающей точкой, типы данных float и double являются «неточным», и не должны использоваться, когда необходимо сохранить точное целое значение.

Объявление переменных в C++

Чтобы объявить переменную используется синтаксис тип <имя>;. Вот некоторые примеры объявления переменных:

```
1	int num;
```

```
2       char character;
3       float num_float;
```

Допустимо объявление нескольких переменных одного и того же типа в одной строке, для этого каждая из них должна быть отделена запятой.

```
1       int x, y, z, d;
```

Если вы смотрели внимательно, вы, возможно, видели, что объявление переменной всегда сопровождается точкой с запятой. Подробнее о соглашении — «об именовании переменных», можно прочитать тут.

Распространенные ошибки при объявлении переменных в С++

Если вы попытаетесь использовать переменную, которую не объявили, ваша программа не будет скомпилирована, и вы получите сообщение об ошибке. В С++, все ключевые слова языка, все функции и все переменные чувствительны к регистру.

Использование переменных

Итак, теперь вы знаете, как объявить переменную. Вот пример программы, демонстрирующий использование переменной:

```
1       #include <iostream>
2
3       using namespace std;
4
5       int main()
6       {
7         int number;
8
9         cout << "Введитечисло: ";
10        cin >> number;
11        cin.ignore();
12        cout << "Выввели: "<< number <<"\n";
13        cin.get();
14      }
```

Давайте рассмотрим эту программу и изучим её код, строку за строкой. Ключевое слово int говорит о том, что number — целое число. Функция cin >> считывает значение в number, пользователь должен нажать ввод после введенного числа. cin.ignore () — функция, которая считывает символ и игнорирует его. Мы организовали свой ввод в программу, после ввода числа, мы

нажимаем клавишу ENTER, символ, который также передаётся в поток ввода. Нам это не нужно, поэтому мы его отбрасываем. Имейте в виду, что переменная была объявлена целого типа, если пользователь попытается ввести десятичное число, то оно будет обрезано (то есть десятичная часть числа будет игнорироваться). Попробуйте ввести десятичное число или последовательность символов, когда вы запустите пример программы, ответ будет зависеть от входного значения.

Обратите внимание, что при печати из переменной кавычки не используются. Отсутствие кавычек сообщает компилятору, что есть переменная, и, следовательно, о том, что программа должна проверять значение переменной для того, чтобы заменить имя переменной на её значение при выполнении. Несколько операторов сдвига в одной строке вполне приемлемо и вывод будет выполняться в том же порядке. Вы должны разделять строковые литералы (строки, заключенные в кавычки) и переменные, давая каждому свой оператор сдвига <<. Попытка поставить две переменные вместе с одним оператором сдвига << выдаст сообщение об ошибке. Не забудьте поставить точку с запятой. Если вы забыли про точку с запятой, компилятор выдаст вам сообщение об ошибке при попытке скомпилировать программу.

Изменение и сравнение величин

Конечно, независимо от того, какой тип данных вы используете, переменные не представляют особого интереса без возможности изменения их значения. Далее показаны некоторые операторы, используемые совместно с переменными:

* умножение,
- вычитание,
+ сложение,
/ деление,
= присвоение,
== равенство,
> больше,
< меньше.
!= неравно
>= больше или равно
<= меньше или равно

Операторы, которые выполняют математические функции, должны быть использованы справа от знака присвоения, для того, чтобы присвоить результат переменной слева.

Вот несколько примеров:

```
1	a = 4 * 6;
2	a = a + 5;
3	a == 5
```

Вы часто будете использовать == в таких конструкциях, как условные операторы и циклы.

```
1	a < 5  // Проверка, а менее пяти?
2	a > 5  // Проверка, а больше пяти?
3	a == 5 // Проверка, а равно пяти?
4	a != 5 // Проверка, а неравно пяти?
5	a >= 5 // Проверка, а больше или равно пяти?
6	a <= 5 // Проверка, а меньше или равно пяти?
```

Рекурсия

Рекурсия достаточно распространённое явление, которое встречается не только в областях науки, но и в повседневной жизни. Например, эффект Дросте, треугольник Серпинского и т. д. Самый простой вариант увидеть рекурсию – это навести Web-камеру на экран монитора компьютера, естественно, предварительно её включив. Таким образом, камера будет записывать изображение экрана компьютера, и выводить его же на этот экран, получится что-то вроде замкнутого цикла. В итоге мы будем наблюдать нечто похожее на тоннель.

В программировании рекурсия тесно связана с функциями, точнее именно благодаря функциям в программировании существует такое понятие как рекурсия или рекурсивная функция. Простыми словами, рекурсия – определение части функции (метода) через саму себя, то есть это функция, которая вызывает саму себя, непосредственно (в своём теле) или косвенно (через другую функцию). Типичными рекурсивными задачами являются задачи: нахождения n!, числа Фибоначчи. Вообще говоря, всё то, что решается итеративно можно решить рекурсивно, то есть с использованием рекурсивной функции. Всё решение сводится к решению основного или, как ещё его называют, базового случая. Существует такое понятие как шаг рекурсии или рекурсивный вызов. В случае, когда рекурсивная функция вызывается для

решения сложной задачи (не базового случая) выполняется некоторое количество рекурсивных вызовов или шагов, с целью сведения задачи к более простой. И так до тех пор, пока не получим базовое решение. Разработаем программу, в которой объявлена рекурсивная функция, вычисляющая n!

```
1	// factorial.cpp: точка входа для консольного приложения.
2
3	#include "stdafx.h"
4	#include <iostream>
5	using namespace std;
6
7	unsigned long int factorial(unsigned long int);
8	int i = 1; // инициализация глобальной переменной
9	unsigned long int result; // глобальная переменная функцией
10
11	int main(int argc, char* argv[])
12	{
13		int n; // переменная для передачи введенного числа
14		cout<< "Enter n!: ";
15		cin >> n;
16		cout<< n << "!" << "=" << factorial(n) << endl;
17		system("pause");
18		return 0;
19	}
20
21	unsigned long int factorial(unsigned long int f)
22	{
23		if (f == 1 || f == 0) // базовое или частное решение
24			return 1; // все мы знаем, что 1!=1 и 0!=1
25		cout << "Step\t" << i << endl;
26		i++; // операция инкремента шага рекурсивных вызовов
27		cout << "Result= " << result << endl;
28		result = f * factorial(f - 1); // функциявызываетсамусебя
29		return result;
30	}
```

В строках 7, 9, 21 объявлен тип данных unsigned long int, так как значение факториала возрастает очень быстро, например, уже 10! = 3 628 800. Если не хватит размера типа данных, то в

результате мы получим совсем не правильное значение. В коде объявлено больше операторов, чем нужно, для нахождения n!. Это сделано для того, чтобы, отработав, программа показала, что происходит на каждом шаге рекурсивных вызовов. Обратите внимание на выделенные строки кода, строки 23, 24, 28 — это рекурсивное решение n!. Строки 23, 24 являются базовым решением рекурсивной функции, то есть, как только значение в переменной f будет равно 1 или 0 (так как мы знаем, что 1! = 1 и 0! = 1), прекратятся рекурсивные вызовы, и начнут возвращаться значения, для каждого рекурсивного вызова. Когда вернётся значение для первого рекурсивного вызова, программа вернёт значение вычисляемого факториала. В строке 28 функция factorial() вызывает саму себя, но уже её аргумент на единицу меньше. Аргумент каждый раз уменьшается, чтобы достичь частного решения.

По результату работы программы хорошо виден каждый шаг и результат на каждом шаге равен нулю, кроме последнего рекурсивного обращения. Необходимо было вычислить пять факториал. Программа сделала четыре рекурсивных обращения, на пятом обращении был найден базовый случай. И как только программа получила решение базового случая, она порешала предыдущие шаги и вывела общий результат.

Итак, чтобы найти 5! нужно знать 4! и умножить его на 5; 4! = 4 * 3! и так далее. Переделаем программу нахождения факториала так, чтобы получить таблицу факториалов. Для этого объявим цикл for, в котором будем вызывать рекурсивную функцию.

```
1	// factorial.cpp: точка входа для консольного приложения.
2	
3	#include"stdafx.h"
4	#include<iostream>
5	usingnamespace std;
6	
7	unsignedlongint factorial(unsignedlongint);
8	unsignedlongint result;
9	
10	int main(intargc, char* argv[])
11	{
12		int n; // локальная переменная для передачи
```

```
13      введенного числа
14              cout<<"Enter n!: ";
15              cin >> n;
16              for (int k = 1; k <= n; k++)
17              {
18                      cout<< k <<"!"<<"="<< factorial(k) << endl;
19              }
20              system("pause");
21              return 0;
22      }
23
24      unsignedlongint factorial(unsignedlongintf)
25      {
26              if (f == 1 || f == 0) // базовое или частное решение
27                      return 1; // все мы знаем, что 1!=1 и 0!=1
28      //cout << "Step\t"<< i <<endl;
29              i++;
30              //cout <<"Result= "<< result << endl;
31              result = f * factorial(f - 1); // функция вызывает саму
32      себя
33              return result;
34      }
35
```

В строках 16 — 19 объявлен цикл, в котором вызывается рекурсивная функция. Всё ненужное в программе закомментировано. Запустив программу, нужно ввести значение, до которого необходимо вычислить факториалы.

Теперь видно, насколько быстро возрастает факториал, кстати говоря, уже результат 14! не правильный, это и есть последствия нехватки размера типа данных. Правильное значение 14! = 87178291200.

Рассмотрим ещё одну типичную задачу — нахождение чисел Фибоначчи, используя рекурсию. Далее приведен код рекурсивного решения такой задачи. Вводим в ком строке порядковый номер числа из ряда Фибоначчи, и программа найдёт все числа из ряда Фибоначчи порядковые номера которых меньше либо равны введённого.

```
1	// fibonacci.cpp: точка входа для консольного приложения.
2
3	#include"stdafx.h"
4	#include<iostream>
5	// библиотека для форматирования выводимой информации
6	#include<iomanip>
7	usingnamespace std;
8
9	unsignedlong fibonacci(unsignedlong);
10
11	int main(intargc, char* argv[])
12	{
13		unsignedlong entered_number;
14		cout <<"Enter number from the Fibonacci series: ";
15		cin >> entered_number;
16		for (int counter = 1; counter <= entered_number; counter++)
17			cout << setw(2) << counter <<" = "
18	<< fibonacci(counter) << endl;
19		system("pause");
20		return 0;
21	}
22
23	unsignedlong fibonacci(unsignedlongentered_number)
24	{
25		if (entered_number == 1 || entered_number == 2)
26			return (entered_number - 1);
27		return fibonacci(entered_number - 1) +
26			fibonacci(entered_number - 2);
27	}
```

В строке 6 подключена библиотека <iomanip> для того, чтобы воспользоваться функцией setw(), которая в свою очередь выравнивает первый столбец чисел, то есть номера. Как мы можем заметить, сначала числа однозначные от 1 – 9, а потом идут двузначные. Если убрать данную функцию, то произойдет сдвиг влево чисел от 1 и до 9.

Решение сводится к разбиению сложной задачи к двум более простым. Например, чтобы найти третье число из ряда Фибоначчи, необходимо сначала найти первое и второе, а потом

сложить их. Первое число является частным случаем и равно 0 (нулю), второе число также является частным случаем и равно 1. Следовательно, третье число из ряда Фибоначчи равно сумме первого и второго = 1. Приблизительно так же рассуждала запрограммированная нами рекурсивная функция поиска чисел ряда Фибоначчи.

Разработаем ещё одну рекурсивную программу, решающую классическую задачу — «Ханойская башня». Даны три стержня, на одном из которых находится стопка n-го количества дисков, причём диски имеют не одинаковый размер (диски различного диаметра) и расположены таким образом, что по мере прохождения, сверху вниз по стержню диаметр дисков постепенно увеличивается. То есть диски меньшего размера должны лежать только на дисках большего размера. Необходимо переместить эту стопку дисков с начального стержня на любой другой из двух оставшихся (чаще всего это третий стержень). Один из стержней использовать как вспомогательный. Перемещать можно только по одному диску, при этом диск большего размера никогда не должен находиться над диском меньшего размера.

Допустим необходимо переместить три диска с первого стержня на третий, значит второй стержень вспомогательный. Программу надо написать для n-го количества дисков. Так как мы решаем данную задачу рекурсивно, то для начала необходимо найти частные случаи решения. В данной задаче частный случай только один – это когда необходимо переместить всего один диск, и в этом случае даже вспомогательный стержень не нужен, но на это просто не обращаем внимания. Теперь необходимо организовать рекурсивное решение, в случае, если количество дисков больше одного. Введём некоторые обозначения, для того, чтоб не писать лишнего:

<Б> — стержень, на котором изначально находятся диски (базовый стержень);

<П> — вспомогательный или промежуточный стержень;
<Ф> — финальный стержень – стержень, на который необходимо переместить диски.

Далее, при описании алгоритма решения задачи будем использовать эти обозначения. Чтобы переместить три диска с <Б> на<Ф> нам необходимо сначала переместить два диска

с <Б> на <П> а потом переместить третий диск(самый большой) на<Ф>, так как <Ф> свободен.

Для того, чтобы переместить n дисков с <Б> на <Ф> нам необходимо сначала переместить n-1 дисков с <Б> на <П> а потом переместить n-й диск(самый большой) на <Ф>, так как <Ф> свободен. После этого необходимо переместить n-1 дисков с <П> на <Ф>, при этом использовать стержень <Б> как вспомогательный. Эти три действия и есть весь рекурсивный алгоритм. Этот же алгоритм на псевдокоде:
n-1 переместить на <П>
n переместить на <Ф>
n-1 переместить с <П> на <Ф>, при этом использовать <Б> как вспомогательный

```
1	// hanoi_tower.cpp: точка входа для консольного приложения.
2	// Программа, рекурсивно решающая задачу "Ханойская
3	башня"
4
5	#include"stdafx.h"
6	#include<iostream>
7	#include<iomanip>
8	usingnamespace std;
9
10	void tower(int, int, int, int);
11	int count = 1;
12
13	int _tmain(intargc, _TCHAR* argv[])
14	{
15		cout <<"Enter  of  numbers  of  disks: ";
16		int number;
17		cin >> number;
18		cout <<"Enter the number of basic rod: ";
19		int basic_rod;
20		cin >> basic_rod;
21		cout <<"Enter the number of final rod: ";
22		int final_rod;
23		cin >> final_rod;
24		int help_rod;
25		if (basic_rod != 2 && final_rod != 2)
```

```
26                      help_rod = 2;
27              else
28                      if (basic_rod != 1 && final_rod != 1)
29                              help_rod = 1;
30                      else
31                              if (basic_rod != 3 && final_rod != 3)
32                                      help_rod = 3;
33              tower(
34                      number,
35                      basic_rod,
36                      help_rod,
37                      final_rod);
38              system("pause");
39              return 0;
40      }
41
42      void tower(intcount_disk, intbaza, inthelp_baza, intnew_baza)
43      {
44              if (count_disk == 1) // условие завершения рекурсивных
45      вызовов
46              {
47                      cout << setw(2) << count <<") "<<baza<<" "
48                              <<"->"<<" "<<new_baza<< endl;
49                      count++;
50              }
51              else
52              {
53                      tower(count_disk - 1, baza, new_baza, help_baza);
54                      tower(1, baza, help_baza, new_baza);
55                      tower(count_disk - 1, help_baza, baza, new_baza);
56              }
57      }
```

Все эти задачи можно было решить итеративно. Возникает вопрос: “Как лучше решать, итеративно или рекурсивно?”. Недостаток рекурсии в том, что она затрачивает значительно больше компьютерных ресурсов, нежели итерация. Это выражается в большой нагрузке, как на оперативную память, так и на

процессор. Если очевидно решение той или иной задачи итеративным способом, то им и надо воспользоваться иначе, использовать рекурсию! В зависимости от решаемой задачи сложность написания программ изменяется при использовании того или иного метода решения. Но чаще задача, решённая рекурсивным методом с точки зрения читабельности кода, куда понятнее и короче.

До сих пор мы писали программы единым, функционально неделимым, кодом. Алгоритм программы находился в главной функции, причём других функций в программе не было. Мы писали маленькие программы, поэтому не было потребности в объявлении своих функций. Для написания больших программ, опыт показывает, что лучше пользоваться функциями. Программа будет состоять из отдельных фрагментов кода, под отдельным фрагментом кода понимается функция. Отдельным, потому, что работа отдельной функции не зависит от работы какой-нибудь другой. То есть алгоритм в каждой функции функционально достаточен и не зависим от других алгоритмов программы.

Однажды написав функцию, её можно будет с лёгкостью переносить в другие программы. Функция (в программировании) — это фрагмент кода или алгоритм, реализованный на каком-то языке программирования, с целью выполнения определённой последовательности операций. Итак, функции позволяют сделать программу модульной, то есть разделить программу на несколько маленьких подпрограмм (функций), которые в совокупности выполняют поставленную задачу. Еще один огромнейший плюс функций в том, что их можно многократно использовать. Данная возможность позволяет многократно использовать один раз написанный код, что в свою очередь, намного сокращает объем кода программы!

Кроме того, что в С++ предусмотрено объявление своих функций, также можно воспользоваться функциями определёнными в стандартных заголовочных файлах языка программирования С++. Чтобы воспользоваться функцией, определённой в заголовочном файле, нужно его подключить. Например, чтобы воспользоваться функцией, которая возводит некоторое число в степень, нужно подключить заголовочный файл <cmath> и в

запустить функцию pow() в теле программы. Разработаем программу, в которой запустим функцию pow().

```
1	// inc_func.cpp: определяет точку входа для консольного
2	приложения.
3
4	#include"stdafx.h"
5	#include<cmath>
6
7	int main(intargc, char* argv[])
8	{
9		float power = pow(3.14, 2);
10		return 0;
11	}
```

Подключение заголовочных файлов выполняется так, как показано в строке 5, т. е. объявляется препроцессорная директива #include, после чего внутри знаков <> пишется имя заголовочного файла. Когда подключен заголовочный файл, можно использовать функцию, что, и сделано в строке 9. Функция pow() возводит число 3.14 в квадрат и присваивает полученный результат переменной power, где pow — имя функции; числа 3.14 и 2 — аргументы функции;

Всегда после имени функции ставятся круглые скобочки, внутри которых, функции передаются аргументы, и если аргументов несколько, то они отделяются друг от друга запятыми. Аргументы нужны для того, чтобы функции передать информацию. Например, чтобы возвести число 3.14 в квадрат используя функцию pow(), нужно как-то этой функции сообщить, какое число, и в какую степень его возводить. Вот именно для этого и придуманы аргументы функций, но бывают функции, в которых аргументы не передаются, такие функции вызываются с пустыми круглыми скобочками. Итак, для того, чтобы воспользоваться функцией из стандартного заголовочного файла С++ необходимо выполнить два действия:

1. Подключить необходимый заголовочный файл;
2. Запустить нужную функцию.

Кроме вызова функций из стандартных заголовочных файлов, в языке программирования С++ предусмотрена

возможность создания собственных функций. В языке программирования C++ есть два типа функций:
Функции, которые не возвращают значений
Функции, возвращающие значение
Функции, не возвращающие значения, завершив свою работу, никакого ответа программе не дают. Рассмотрим структуру объявления таких функций.

```
1	// структура объявления функций не возвращающих значений
2	void/*имя функции*/(/*параметры функции*/)
3	{
4		// тело функции
5	}
```

Строка 2 начинается с зарезервированного слова void — это тип данных, который не может хранить какие-либо данные. Тип данных void говорит о том, что данная функция не возвращает никаких значений. void никак по-другому не используется и нужен только для того, чтобы компилятор мог определить тип функции. После зарезервированного слова void пишется имя функции. Сразу за именем функции ставятся две круглые скобочки, открывающаяся и закрывающаяся. Если нужно функции передавать какие-то данные, то внутри круглых скобочек объявляются параметры функции, они отделяются друг от друга запятыми. Строка 2 называется заголовком функции. После заголовка функции пишутся две фигурные скобочки, внутри которых находится алгоритм, называемый телом функции. Разработаем программу, в которой объявим функцию нахождения факториала, причём функция не должна возвращать значение.

```
1	// struct_func.cpp: определяет точку входа для консольного
2	приложения.
3	
4	#include"stdafx.h"
5	#include<iostream>
6	usingnamespace std;
7	
8	// объявление функции нахождения n!
9	void faktorial(intnumb)// заголовокфункции
10	{
11		int rezult = 1;
```

```
12              for (int i = 1; i <= numb; i++)
13                      rezult *= i;
14              cout<<numb<<"! = "<< rezult << endl;
15      }
16
17      int main(intargc, char* argv[])
18      {
19              int digit; // переменная для хранения значения n!
20              cout <<"Enter number: ";
21              cin >> digit;
22              faktorial(digit);// запуск функции нахождения
23      факториала
24              system("pause");
25              return 0;
26      }
```

После того, как были подключены все необходимые заголовочные файлы, можно объявлять функцию нахождения факториала. Под объявлением функции подразумевается выбор имени функции, определение параметров функции и написание алгоритма, который является телом функции. После выполнения этих действий функцию можно использовать в программе. Так как функция не должна возвращать значение, то тип возвращаемых данных должен быть void. Имя функции — faktorial, внутри круглых скобочек объявлена переменная numb типа int. Эта переменная является параметром функции faktorial(). Таким образом, все объявления в строке 8 в совокупности составляют заголовок функции. Строки 9 — 14 составляют тело функции faktorial(). Внутри тела в строке 10 объявлена переменная rezult, которая будет хранить результат нахождения n! После чего, в строках 11-12 Объявлен оператор цикла for для нахождения факториала. В строке 13 объявлен оператор cout, с помощью которого значение факториала будет печататься на экране. Теперь, когда функция объявлена можно воспользоваться ею. В строке 21 запускается функция faktorial(digit), внутри скобочек функции передаётся аргумент, т. е. значение, содержащееся в переменной digit.

Функции, возвращающие значение, по завершению своей работы возвращают определённый результат. Такие функции могут

возвращать значение любого типа данных. Структура функций, возвращающих значение будет немного отличатся от структуры функций рассмотренных ранее.

```
1	// структура объявления функций возвращающих значения
2	/*возвращаемый тип данных*//*имя функции*/(/*параметры
3	функции*/)
4	{
5		// тело функции
6		return/*возвращаемое значение*/;
7	}
```

Структура объявления функций осталась почти неизменной, за исключением двух строк. В заголовке функции сначала нужно определять возвращаемый тип данных, это может быть тип данных int, если необходимо возвратить целое число или тип данных float — для чисел с плавающей точкой. В общем, любой другой тип данных, всё зависит от того, что функция должна вернуть. Так как функция должна вернуть значение, то для этого должен быть предусмотрен специальный механизм, как в строке 5. С помощью зарезервированного слова return можно вернуть значение, по завершении работы функции. Всё, что нужно, так это указать переменную, содержащую нужное значение, или некоторое значение, после оператора return. Тип данных возвращаемого значения в строке 5 должен совпадать с типом данных в строке 2. Переделаем программу нахождения факториала так, чтобы функция faktorial() возвращала значение факториала.

```
1	// struct_func.cpp: определяет точку входа для консольного
2	приложения.
3
4	#include"stdafx.h"
5	#include<iostream>
6	usingnamespace std;
7
8	// объявление функции нахождения n!
9	int faktorial(intnumb)// заголовокфункции
10	{
11		int rezult = 1; // инициализируем переменную rezult
12	значением 1
13		for (int i = 1; i <= numb; i++) //
```

```
14      циклвычислениязначения n!
15                          rezult *= i; // накапливаем произведение в
16      переменной rezult
17              return rezult; // передаём значение факториала в
18      главную функцию
19      }
20
21      int main(intargc, char* argv[])
22      {
23              int digit; // переменная для хранения значения n!
24              cout <<"Enter number: ";
25              cin >> digit;
26              cout<< digit <<"! = "<< faktorial(digit) << endl;
27              system("pause");
28              return 0;
29      }
```

Теперь функция faktorial() имеет возвращаемый тип данных — int, так как n! — это целое число. В строке 13 объявлен оператор return, который возвращает значение, содержащееся в переменной rezult. В строке 21 выполняем запуск функции faktorial(), возвращаемое значение которой отправляем в поток вывода с помощью оператора cout. Можно было бы написать так int fakt = faktorial(digit); — переменной типа int присваиваем возвращаемое значение функции faktorial(), после чего в переменной fakt будет храниться значение n!.

Контрольные вопросы к параграфу

1. Что такое язык программирования.
2. История появления языка C++.
3. Логические операции над информацией.
4. Переменные и операторы.
5. Что такое рекурсия.

4. ГЕНЕРАТОР ТЕСТОВ ДЛЯ НАПИСАНИЯ КОМПЛЕКТОВ ЗАДАЧ.

Для наполнения сервера Ejudge олимпиадными задачами была написана небольшая программа для генерации тестов и ответов к ним. В основе этой программы лежит циклический алгоритм, каждую итерацию которого вызывается функция-генератор, принимающая в качестве параметров номер теста и потоки для ввода и вывода данных, перенаправленных на соответствующие файлы в каталоге, а также функция-решение, которая по сгенерированному тесту авторским решением находит ответ и записывает его в соответствующий файл. Разберемреализациюгенератораподробнее.

```
1	#include<iostream>
2	#include<fstream>
3	#include"testlib.h"
4	
5	#include"solver.h"
6	#include"testgen.h"
7	
8	usingnamespace std;
9	
10	typedefdoubledb;
11	
12	constint T = 100;
13	
14	ifstream in;
15	ofstream out;
16	
17	char test[100], ans[100];
18	
19	int main() {
20		for (int i = 1; i <= T; i++) {
21			cout <<"test #"<<i<<" ";
22			sprintf_s(test, "tests\\%00d.in", i);
23			sprintf_s(ans, "tests\\%00d.out", i);
24	
25			out.open(test);
```

```
26                          testgen(out, i);
27                          out.close();
28                          cout <<"ok ";
29
30                          in.open(test);
31                          out.open(ans);
32                          solver(in, out);
33                          in.close();
34                          out.close();
35                          cout <<"ok\n";
36                 }
37                 system("pause");
38        }
```

В приведенном в листинге 1 коде используются две стандартные библиотеки для работы с потоками ввода и вывода. Это библиотека iostream для работы со стандартными потоками для вывода сообщений на экран о промежуточных результатах работы генератора и библиотека fstream для работы с вводом и выводом информации в файлы. Далее подключаются два заголовочных файла solver.h и testgen.h. Это файлы с реализацией решения и генератора тестов соответственно.

Константная переменная T используется для задания количества тестов при генерации, а также для разделения тестов на блоки с разными ограничениями на входные данные. Потоки in и out нужны для открывания и записи файлов с результатами вычислений. Символьные массивы test и ans – это буферы для задания имен открываемых файлов.

Сама программа состоит из одного цикла, в котором на каждой итерации создаются два файла – тест и ответ на него. После создания файлов запускается генератор тестов, который генерирует соответствующий тест с номером i, записывает его в файл и сохраняет. На экран выводится сообщение «ok», говорящее о том, что тест с номером i создан. После этого этот же файл уже открывается как входной и тест попадает в функцию-решение. Найденный ответ записывается в файл с ответом, и на этом итерация i заканчивается. В результате мы имеем два файла с именами i.in и i.out, где номер теста заполняется слева нулями для

того, чтобы все имена файлов были трехсимвольными. Все файлы для тестирования сохраняются в подкаталоге tests каталога проекта генератора тестов.

```
1	#pragmaonce
2	#include<fstream>
3	#include"testlib.h"
4	
5	usingnamespace std;
6	
7	int N[2][11] = { { 1, 10, 100, 1000, 10000, 100000, 250000,
8	300000, 350000, 400000, 450000},
9	                                    {20, 40, 200, 5000, 25000, 400000,
10	600000, 800000, 900000, 1000000, 1000000} };
11	
12	void testgen(ofstream &out, int test) {
13		int a = N[0][test / 10]+rand()% min(N[0][test / 10], 1000);
14		int b = N[1][test / 10]+ rand()%min(N[1][test / 10],
15	1000);
16		out << a<<' '<< b;
17	}
```

Функция-генератор пишется под каждую задачу по-разному. В листинге 2 приведен пример генерации тестов на задачу про простые числа. Для теста необходимо два целых числа, а ограничения для них записаны в двумерный массив N.

```
1	#pragmaonce
2	#include<fstream>
3	#include<vector>
4	#include<algorithm>
5	#include<string>
6	#include"testlib.h"
7	
8	usingnamespace std;
9	
10	void solver(ifstream&in, ofstream&out) {
11		int a, b;
12		in>> a >> b;
13		constint N = 1e6 + 7;
```

```
14              vector<char> p(N, 1);
15              vector<int> v;
16              p[0] = 0; p[1] = 0;
17              for (int i = 0; i < N; i++) {
18                      if (p[i]) {
19                              v.push_back(i);
20                              if (i * 1ll * i < N) {
21                              for (int j = i * i; j < N; j += i){
22                                              p[j] = 0;
23                                      }
24                              }
25                      }
26              }
27              int ans = 0;
28              for (int i = 0; i < v.size(); i++) {
29                      if (v[i]>= a && v[i]<= b)
30                              ans++;
31              }
32              out<< ans;
33      }
```

В листинге 3 приведен пример реализации функции-решения, в которой для нахождения простых чисел используется алгоритм решето Эратосфена. Также, как и в случае с генератором, решение под каждую задачу пишется отдельно.

Использование вспомогательных средств
при разработке комплектов задач.

При составлении тестов для какой-либо задачи можно воспользоваться библиотекой testlib.h. Эта библиотека содержит функционал для написания генераторов тестов, валидаторов для проверки тестов на правильный формат, интеракторов для проверки решений на интерактивных задачах и чекеров для проверки на задачах с неоднозначным ответом. Первая версия библиотеки появилась в 2005 году. Является стандартом для написания тестов к задачам в России и ряде близлежащих стран.

```
1       #include"testlib.h"
2       #include<iostream>
3
```

```
4       usingnamespace std;
5
6       int main(intargc, char* argv[])
7       {
8               registerGen(argc, argv, 1);
9               int n = atoi(argv[1]);
10              cout << rnd.next(1, n) <<" ";
11              cout << rnd.next(1, n) << endl;
        }
```

В приведенном примере генератора тестов на задачу о простых числах библиотека stdlib.h не использовалась. Для получения случайных значений на тестах применилась функция rand(), которая выдает случайное числовое значение. Для того, чтобы при генерации тестов в любой момент времени и на любом компиляторе программа выдавала одинаковые результаты, нужно использовать генерацию случайных чисел из библиотеки stdlib.h.

Контрольные вопросы к параграфу

1. Основная часть генератора тестов.
2. Генератор тестов как антизадача.
3. Программа – решение.
4. Библиотека stdlib.h

5. ПРИМЕРЫ АЛГОРИТМОВ КОМПЬЮТЕРНОЙ ОБРАБОТКИ ДАННЫХ.

5.1 Алгоритм Евклида нахождения НОД (наибольшего общего делителя)

Даны два целых неотрицательных числа a и b. Требуется найти их наибольший общий делитель, т.е. наибольшее число, которое является делителем одновременно и a, и b. На английском языке "наибольший общий делитель" пишется "greatest common divisor", и распространённым его обозначением является gcd:

$$\gcd(a, b) = \max_{k=1\ldots\infty : k|a \;\&\; k|b} k$$

(здесь символом "|" обозначена делимость, т.е. "k|a" обозначает "k делит a")

Когда оно из чисел равно нулю, а другое отлично от нуля, их наибольшим общим делителем, согласно определению, будет это второе число. Когда оба числа равны нулю, результат не определён (подойдёт любое бесконечно большое число), мы положим в этом случае наибольший общий делитель равным нулю. Поэтому можно говорить о таком правиле: если одно из чисел равно нулю, то их наибольший общий делитель равен второму числу.

Алгоритм Евклида, рассмотренный ниже, решает задачу нахождения наибольшего общего делителя двух чисел a и b заO(logmin(a, b)). Данный алгоритм был впервые описан в книге Евклида "Начала" (около 300 г. до н.э.), хотя, вполне возможно, этот алгоритм имеет более раннее происхождение.

Алгоритм

Сам алгоритм чрезвычайно прост и описывается следующей формулой:

$$\gcd(a, b) = \begin{cases} a, if\ b = 0 \\ \gcd(b, a\ mod\ b), otherwise \end{cases}$$

Реализация

```
int gcd (int a, int b) {
	if (b == 0)
		return a;
	else
		return gcd (b, a % b);
```

```
}
```

Используя тернарный условный оператор C++, алгоритм можно записать ещё короче:

```
int gcd (int a, int b) {
	return b ? gcd (b, a % b) : a;
}
```

Наконец, приведём и нерекурсивную форму алгоритма:

```
int gcd (int a, int b) {
	while (b) {
		a %= b;
		swap (a, b);
	}
	return a;
}
```

5.2 Решето Эратосфена

Решето Эратосфена — это алгоритм, позволяющий найти все простые числа в отрезке [1;n] за $O(n \log \log n)$ операций.

Идея проста — запишем ряд чисел 1…n, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем делящиеся на 3, кроме самого числа 3, затем на 5, затем на 7, 11, и все остальные простые до n.

Реализация

Сразу приведём реализацию алгоритма:

```
int n;
vector<char> prime(n + 1, true);
prime[0] = prime[1] = false;
for (int i = 2; i <= n; ++i)
	if (prime[i])
		if (i * 1ll * i <= n)
			for (int j = i * i; j <= n; j += i)
				prime[j] = false;
```

Этот код сначала помечает все числа, кроме нуля и единицы, как простые, а затем начинает процесс отсеивания составных чисел. Для этого мы перебираем в цикле все числа от 2 до n, и, если текущее число i простое, то помечаем все числа, кратные ему, как составные.

При этом мы начинаем идти от i^2, поскольку все меньшие числа, кратные i, обязательно имеют простой делитель меньше i, а значит, все они уже были отсеяны раньше. (Но поскольку i^2 легко может переполнить тип int, в коде перед вторым вложенным циклом делается дополнительная проверка с использованием типа longlong.)

При такой реализации алгоритм потребляет O(n) памяти (что очевидно) и выполняет O(nloglogn)действий (это доказывается в следующем разделе).

Просеивание простыми до корня

Самый очевидный момент — что для того, чтобы найти все простые до n, достаточно выполнить просеивание только простыми, не превосходящими корня из n.

Таким образом, изменится внешний цикл алгоритма:

```
for (int i=2; i*i<=n; ++i)
```

На асимптотику такая оптимизация не влияет (действительно, повторив приведённое выше доказательство, мы получим оценку $n \ln \ln \sqrt{n} + o(n)$, что, по свойствам логарифма, асимптотически есть то же самое), хотя число операций заметно уменьшится.

Решето только по нечётным числам

Поскольку все чётные числа, кроме 2, — составные, то можно вообще не обрабатывать никак чётные числа, а оперировать только нечётными числами.

Во-первых, это позволит вдвое сократить объём требуемой памяти.

Во-вторых, это уменьшит число делаемых алгоритмом операций примерно вдвое.

5.3 Числа Фибоначчи

Определение

Последовательность Фибоначчи определяется следующим образом:

$F_0=0$,

$F_1=1$,

$F_n=F_{n-1}+F_{n-2}$.

Несколько первых её членов:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

История

Эти числа ввёл в 1202 г. Леонардо Фибоначчи (Leonardo Fibonacci) (также известный как Леонардо Пизанский (Leonardo Pisano)). Однако именно благодаря математику 19 века Люка (Lucas) название "числа Фибоначчи" стало общеупотребительным.
Впрочем, индийские математики упоминали числа этой последовательности ещё раньше: Гопала (Gopala) до 1135 г., Хемачандра (Hemachandra) — в 1150 г.

Числа Фибоначчи в природе

Сам Фибоначчи упоминал эти числа в связи с такой задачей: "Человек посадил пару кроликов в загон, окруженный со всех сторон стеной. Сколько пар кроликов за год может произвести на свет эта пара, если известно, что каждый месяц, начиная со второго, каждая пара кроликов производит на свет одну пару?". Решением этой задачи и будут числа последовательности, называемой теперь в его честь. Впрочем, описанная Фибоначчи ситуация — больше игра разума, чем реальная природа.

Индийские математики Гопала и Хемачандра упоминали числа этой последовательности в связи с количеством ритмических рисунков, образующихся в результате чередования долгих и кратких слогов в стихах или сильных и слабых долей в музыке. Число таких рисунков, имеющих в целом n долей, равно F_n.Числа Фибоначчи появляются и в работе Кеплера 1611 года, который размышлял о числах, встречающихся в природе (работа "О шестиугольных снежинках").

Интересен пример растения — тысячелистника, у которого число стеблей (а значит и цветков) всегда есть число Фибоначчи. Причина этого проста: будучи изначально с единственным стеблем, этот стебель затем делится на два, затем от главного стебля ответвляется ещё один, затем первые два стебля снова разветвляются, затем все стебли, кроме двух последних, разветвляются, и так далее. Таким образом, каждый стебель после своего появления "пропускает" одно разветвление, а затем начинает делиться на каждом уровне разветвлений, что и даёт в результате числа Фибоначчи.

Вообще говоря, у многих цветов (например, лилий) число лепестков является тем или иным числом Фибоначчи.Также в ботанике известно явление "филлотаксиса". В качестве примера

можно привести расположение семечек подсолнуха: если посмотреть сверху на их расположение, то можно увидеть одновременно две серии спиралей (как бы наложенных друг на друга): одни закручены по часовой стрелке, другие — против.

Оказывается, что число этих спиралей примерно совпадает с двумя последовательными числами Фибоначчи: 34 и 55 или 89 и 144. Аналогичные факты верны и для некоторых других цветов, а также для сосновых шишек, брокколи, ананасов, и т.д.

Для многих растений (по некоторым данным, для 90% из них) верен и такой интересный факт. Рассмотрим какой-нибудь лист, и будем спускаться от него вниз до тех пор, пока не достигнем листа, расположенного на стебле точно так же (т.е. направленного точно в ту же сторону). Попутно будем считать все листья, попадавшиеся нам (т.е. расположенные по высоте между стартовым листом и конечным), но расположенными по-другому. Нумеруя их, мы будем постепенно совершать витки вокруг стебля (поскольку листья расположены на стебле по спирали). В зависимости от того, совершать витки по часовой стрелке или против, будет получаться разное число витков. Но оказывается, что число витков, совершённых нами по часовой стрелке, число витков, совершённых против часовой стрелки, и число встреченных листьев образуют 3 последовательных числа Фибоначчи.

Впрочем, следует отметить, что есть и растения, для которых приведённые выше подсчёты дадут числа из совсем других последовательностей, поэтому нельзя сказать, что явление филлотаксиса является законом, — это скорее занимательная тенденция.

Свойства

Числа Фибоначчи обладают множеством интересных математических свойств.

Вот лишь некоторые из них:

Соотношение Кассини:

$$F_{n+1}F_{n-1} - F_n^2 = (-1^n).$$

Правило "сложения":

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$$

Из предыдущего равенства при k=n вытекает:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

Из предыдущего равенствапо индукции можно получить, чтоF_{nk} всегда кратно F_n.

Верно и обратное к предыдущему утверждение:

еслиF_m кратно F_n, то m кратно n.

НОД-равенство:

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$$

По отношению к алгоритму Евклида числа Фибоначчи обладают тем замечательным свойством, что они являются наихудшими входными данными для этого алгоритма.

Матричная формула для чисел Фибоначчи

Нетрудно доказать матричное следующее равенство:

$$(F_{n-2} F_{n-1}) * \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_{n-1} \quad F_n).$$

Но тогда, обозначая

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

получаем:

$$(F_0 \quad F_1) * P^n = (F_n \quad F_{n+1}).$$

Таким образом, для нахождения n-го числа Фибоначчи надо возвести матрицу P в степень n.

Вспоминая, что возведение матрицы в n-ую степень можно осуществить за O(logn) (алгоритм бинарного возведения в степень), получается, что n-ое число Фибоначчи можно легко вычислить за O(logn) с использованием только целочисленной арифметики.

5.4 Поиск в ширину

Поиск в ширину (обход в ширину, breadth-first search) — это один из основных алгоритмов на графах.

В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе, т.е. путь, содержащий наименьшее число рёбер.

Алгоритм работает за O(n+m), где n — число вершин, m — число рёбер.

Описание алгоритма

На вход алгоритма подаётся заданный граф (невзвешенный), и номер стартовой вершины s. Граф может быть как

ориентированным, так и неориентированным, для алгоритма это не важно.
Сам алгоритм можно понимать, как процесс "поджигания" графа: на нулевом шаге поджигаем только вершину s. На каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей; т.е. за одну итерацию алгоритма происходит расширение "кольца огня" в ширину на единицу (отсюда и название алгоритма).
Более строго это можно представить следующим образом. Создадим очередь q, в которую будут помещаться горящие вершины, а также заведём булевский массив used[], в котором для каждой вершины будем отмечать, горит она уже или нет (или иными словами, была ли она посещена).

Изначально в очередь помещается только вершина s, и used[s]=true, а для всех остальных вершин used[]=false. Затем алгоритм представляет собой цикл: пока очередь не пуста, достать из её головы одну вершину, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из просмотренных вершин ещё не горят, то поджечь их и поместить в конец очереди.
В итоге, когда очередь опустеет, обход в ширину обойдёт все достижимые из s вершины, причём до каждой дойдёт кратчайшим путём. Также можно посчитать длины кратчайших путей (для чего просто надо завести массив длин путей d[], и компактно сохранить информацию, достаточную для восстановления всех этих кратчайших путей (для этого надо завести массив "предков" p[], в котором для каждой вершины хранить номер вершины, по которой мы попали в эту вершину).
Реализация
Реализуем вышеописанный алгоритм на языке C++.
Входные данные:

```
vector<vector<int>>g; // граф
intn; // числовершин
int s; // стартовая вершина (вершины нумеруются с нуля)

// чтение графа
...
```

Сам обход:

```
queue<int>q;
q.push(s);
vector<bool> used(n);
vector<int> d(n), p(n);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
	int v = q.front();
	q.pop();
	for (size_t i = 0; i < g[v].size(); ++i) {
		int to = g[v][i];
		if (!used[to]) {
			used[to] = true;
			q.push(to);
			d[to] = d[v] + 1;
			p[to] = v;
		}
	}
}
```

Если теперь надо восстановить и вывести кратчайший путь до какой-то вершины to, это можно сделать следующим образом:

```
if (!used[to])
	cout<<"No path!";
else {
	vector<int> path;
	for (int v = to; v != -1; v = p[v])
		path.push_back(v);
	reverse(path.begin(), path.end());
	cout<<"Path: ";
	for (size_t i = 0; i < path.size(); ++i)
		cout << path[i] + 1 <<" ";
}
```

Приложения алгоритма

Поиск кратчайшего пути в невзвешенном графе.

Поиск компонент связности в графе за $O(n+m)$.

Для этого мы просто запускаем обход в ширину от каждой вершины, за исключением вершин, оставшихся посещёнными

(used=true) после предыдущих запусков. Таким образом, мы выполняем обычный запуск в ширину от каждой вершины, но не обнуляем каждый раз массив used[], за счёт чего мы каждый раз будем обходить новую компоненту связности, а суммарное время работы алгоритма составит по-прежнему O(n+m) (такие несколько запусков обхода на графе без обнуления массива usedназываются серией обходов в ширину).

Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа.

Классический пример — игра, где робот двигается по полю, при этом он может передвигать ящики, находящиеся на этом же поле, и требуется за наименьшее число ходов передвинуть ящики в требуемые позиции. Решается это обходом в ширину по графу, где состоянием (вершиной) является набор координат: координаты робота, и координаты всех коробок.

Нахождение кратчайшего пути в 0-1-графе (т.е. графе взвешенном, но с весами равными только 0 либо 1): достаточно немного модифицировать поиск в ширину: если текущее ребро нулевого веса, и происходит улучшение расстояния до какой-то вершины, то эту вершину добавляем не в конец, а в начало очереди. Нахождение кратчайшего цикла в ориентированном невзвешенном графе: производим поиск в ширину из каждой вершины; как только в процессе обхода мы пытаемся пойти из текущей вершины по какому-то ребру в уже посещённую вершину, то это означает, что мы нашли кратчайший цикл, и останавливаем обход в ширину; среди всех таких найденных циклов (по одному от каждого запуска обхода) выбираем кратчайший.

Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин (a, b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через d_a[] массив кратчайших расстояний, полученный в результате первого обхода, а через d_b[] — в результате второго обхода. Теперь для любого ребра (u, v) легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие$d_a[u]+1+d_b[u]=d_a[b]$.

Найти все вершины, лежащие на каком-либо кратчайшем пути между заданной парой вершин (a, b). Для этого надо запустить

2 поиска в ширину: из a, и из b. Обозначим через $d_a[]$ массив кратчайших расстояний, полученный в результате первого обхода, а через $d_b[]$ — в результате второго обхода. Теперь для любой вершины v легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие $d_a[u]+d_b[u]=d_a[b]$.

Найти кратчайший чётный путь в графе (т.е. путь чётной длины). Для этого надо построить вспомогательный граф, вершинами которого будут состояния (v, c), где v — номер текущей вершины, c=0…1 — текущая чётность. Любое ребро (a, b) исходного графа в этом новом графе превратится в два ребра ((u, 0), (v, 1)) и ((u, 1), (v, 0)). После этого на этом графе надо обходом в ширину найти кратчайший путь из стартовой вершины в конечную, с чётностью, равной 0.

5.5 Поиск в глубину

Это один из основных алгоритмов на графах.
В результате поиска в глубину находится лексикографически первый путь в графе.
Алгоритм работает за O (N+M).
Применения алгоритма
Поиск любого пути в графе.
Поиск лексикографически первого пути в графе.
Проверка, является ли одна вершина дерева предком другой:
В начале и конце итерации поиска в глубину будет запоминать "время" захода и выхода в каждой вершине. Теперь за O(1) можно найти ответ: вершина i является предком вершины j тогда и только тогда, когда starti < startj и endi > endj.
Задача LCA (наименьший общий предок).
Топологическаясортировка:
Запускаем серию поисков в глубину, чтобы обойти все вершины графа. Отсортируем вершины по времени выхода по убыванию - это и будет ответом.
Проверка графа на ацикличность и нахождение цикла
Поиск компонент сильной связности:
Сначала делаем топологическую сортировку, потом транспонируем граф и проводим снова серию поисков в глубину в порядке,

определяемом топологической сортировкой. Каждое дерево поиска - сильносвязная компонента.

Поиск мостов:

Сначала превращаем граф в ориентированный, делая серию поисков в глубину, и ориентируя каждое ребро так, как мы пытались по нему пройти. Затем находим сильносвязные компоненты. Мостами являются те рёбра, концы которых принадлежат разным сильносвязным компонентам.

Реализация

```
vector< vector<int>> g; // граф
	intn; // числовершин

	vector<int> color; // цвет вершины (0, 1, или 2)

	vector<int> time_in, time_out; // "времена" захода и выхода из вершины
	int dfs_timer = 0; // "таймер" для определения времён

	void dfs(intv) {
		time_in[v] = dfs_timer++;
		color[v] = 1;
		for (vector<int>::iterator i = g[v].begin(); i != g[v].end(); ++i)
			if (color[*i] == 0)
				dfs(*i);
		color[v] = 2;
		time_out[v] = dfs_timer++;
	}
```

Это наиболее общий код. Во многих случаях времена захода и выхода из вершины не важны, так же как и не важны цвета вершин (но тогда надо будет ввести аналогичный по смыслу булевский массив used). Вот наиболее простая реализация:

```
vector<vector<int>>g; // граф
intn; // числовершин
vector<char> used;
void dfs(intv) {
	used[v] = true;
	for (vector<int>::iterator i = g[v].begin(); i != g[v].end(); ++i)
```

```
            if (!used[*i])
                dfs(*i);
}
```

5.6Топологическая сортировка

Дан ориентированный граф с n вершинами и m рёбрами. Требуется перенумеровать его вершины таким образом, чтобы каждое рёбро вело из вершины с меньшим номером в вершину с большим.

Иными словами, требуется найти перестановку вершин (топологический порядок), соответствующую порядку, задаваемому всеми рёбрами графа.

Топологическая сортировка может быть не единственной (например, если граф — пустой; или если есть три такие вершины a, b, c, что из a есть пути в b и в c, но ни из b в c, ни из c в b добраться нельзя).

Топологической сортировки может не существовать вовсе — если граф содержит циклы (поскольку при этом возникает противоречие: есть путь и из одной вершины в другую, и наоборот).

Распространённая задача на топологическую сортировку — следующая. Есть n переменных, значения которых нам неизвестны. Известно лишь про некоторые пары переменных, что одна переменная меньше другой. Требуется проверить, не противоречивы ли эти неравенства, и если нет, выдать переменные в порядке их возрастания (если решений несколько — выдать любое). Легко заметить, что это в точности и есть задача о поиске топологической сортировки в графе из n вершин.

Алгоритм

Для решения воспользуемся обходом в глубину.

Предположим, что граф ацикличен, т.е. решение существует. Что делает обход в глубину? При запуске из какой-то вершины v он пытается запуститься вдоль всех рёбер, исходящих из v. Вдоль тех рёбер, концы которых уже были посещены ранее, он не проходит, а вдоль всех остальных — проходит и вызывает себя от их концов.

Таким образом, к моменту выхода из вызова dfs(v) все вершины, достижимые из v как непосредственно (по одному ребру), так и

косвенно (по пути) — все такие вершины уже посещены обходом. Следовательно, если мы будем в момент выхода из dfs(v) добавлять нашу вершину в начало некоего списка, то в конце концов в этом списке получится топологическая сортировка.

Эти объяснения можно представить и в несколько ином свете, с помощью понятия "времени выхода" обхода в глубину. Время выхода для каждой вершины v — это момент времени, в который закончил работать вызов dfs(v) обхода в глубину от неё (времена выхода можно занумеровать от 1 до n). Легко понять, что при обходе в глубину время выхода из какой-либо вершины vвсегда больше, чем время выхода из всех вершин, достижимых из неё (т.к. они были посещены либо до вызова dfs(v), либо во время него).

Таким образом, искомая топологическая сортировка — это сортировка в порядке убывания времён выхода.

Реализация

Приведём реализацию, предполагающую, что граф ацикличен, т.е. искомая топологическая сортировка существует. При необходимости проверку графа на ацикличность легко вставить в обход в глубину, как описано в статье по обходу в глубину.

```
int n; // числовершин
vector<int> g[MAXN]; // граф
bool used[MAXN];
vector<int> ans;

void dfs(intv) {
	used[v] = true;
	for (size_t i = 0; i < g[v].size(); ++i) {
		int to = g[v][i];
		if (!used[to])
			dfs(to);
	}
	ans.push_back(v);
}

void topological_sort() {
	for (int i = 0; i < n; ++i)
		used[i] = false;
	ans.clear();
```

```
	for (int i = 0; i < n; ++i)
		if (!used[i])
			dfs(i);
	reverse(ans.begin(), ans.end());
}
```

Здесь константе MAXN следует задать значение, равное максимально возможному числу вершин в графе.
Основная функция решения — это topological_sort, она инициализирует пометки обхода в глубину, запускает его, и ответ в итоге получается в векторе ans.

5.7 Алгоритм поиска компонент связности в графе

Дан неориентированный граф G с n вершинами и m рёбрами. Требуется найти в нём все компоненты связности, т.е. разбить вершины графа на несколько групп так, что внутри одной группы можно дойти от одной вершины до любой другой, а между разными группами — пути не существует.
Алгоритм решения
Для решения можно воспользоваться как обходом в глубину, так и обходом в ширину.
Фактически, мы будем производить серию обходов: сначала запустим обход из первой вершины, и все вершины, которые он при этом обошёл — образуют первую компоненту связности. Затем найдём первую из оставшихся вершин, которые ещё не были посещены, и запустим обход из неё, найдя тем самым вторую компоненту связности. И так далее, пока все вершины не станут помеченными.

Итоговая асимптотика составит O(n+m): в самом деле, такой алгоритм не будет запускаться от одной и той же вершины дважды, а, значит, каждое ребро будет просмотрено ровно два раза (с одного конца и с другого конца).
Реализация
Для реализации чуть более удобным является обход в глубину:

```
int n;
vector<int> g[MAXN];
bool used[MAXN];
vector<int> comp;
```

```
void dfs(intv) {
	used[v] = true;
	comp.push_back(v);
	for (size_t i = 0; i < g[v].size(); ++i) {
		int to = g[v][i];
		if (!used[to])
			dfs(to);
	}
}

void find_comps() {
	for (int i = 0; i < n; ++i)
		used[i] = false;
	for (int i = 0; i < n; ++i)
		if (!used[i]) {
			comp.clear();
			dfs(i);

			cout<<"Component:";
			for (size_t j = 0; j < comp.size(); ++j)
				cout<<' '<< comp[j];
			cout << endl;
		}
}
```

Основная функция для вызова — find_comps(), она находит и выводит компоненты связности графа.

Мы считаем, что граф задан списками смежности, т.е. g[i] содержит список вершин, в которые есть рёбра из вершины i. Константе MAXN следует задать значение, равное максимально возможному количеству вершин в графе.

Вектор comp содержит список вершин в текущей компоненте связности.

5.8 Поиск мостов

Пусть дан неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным (или,

точнее, увеличивает число компонент связности). Требуется найти все мосты в заданном графе.
Неформально эта задача ставится следующим образом: требуется найти на заданной карте дорог все "важные" дороги, т.е. такие дороги, что удаление любой из них приведёт к исчезновению пути между какой-то парой городов.

Ниже мы опишем алгоритм, основанный на поиске в глубину, и работающий за время $O(n+m)$, где n — количество вершин, m — рёбер в графе.
Заметим, что на сайте также описан онлайновый алгоритм поиска мостов — в отличие от описанного здесь алгоритма, онлайновый алгоритм умеет поддерживать все мосты графа в изменяющемся графе (имеются в виду добавления новых рёбер).
Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через root. Заметим следующий факт (который несложно доказать):
Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to, кроме как спуск по ребру (v, to) дерева обхода в глубину.)

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.
Итак, пусть tin(v) — это время захода поиска в глубину в вершину v. Теперь введём массив fup[v], который и позволит нам отвечать на вышеописанные запросы. Время fup[v] равно минимуму из времени захода в саму вершину tin[v], времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений fup[to] для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = min\begin{cases} tin[v], \\ tin[p], for\ all\ (v,p) - back\ edge \\ fup[to], for\ all\ (v,to) - tree\ edge \end{cases}$$

(здесь "back edge" — обратное ребро, "tree edge" — ребро дерева)
Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] \leq tin[v]$. (Если $fup[to] = tin[v]$, то это означает, что найдётся обратное ребро, приходящее точно в v; если же $fup[to] < tin[v]$, то это означает наличие обратного ребра в какого-либо предка вершины v.)
Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] > tin[v]$, то это ребро является мостом; в противном случае оно мостом не является.

5.9 Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры для разреженных графов

Алгоритм

Сложность алгоритма Дейкстры складывается из двух основных операций: время нахождения вершины с наименьшей величиной расстояния d[v], и время совершения релаксации, т.е. время изменения величины d[to].
При простейшей реализации эти операции потребуют соответственно O(n) и O(1) времени. Учитывая, что первая операция всего выполняется O(n) раз, а вторая — O(m), получаем асимптотику простейшей реализации алгоритма Дейкстры:$O(n^2+m)$.
Понятно, что эта асимптотика является оптимальной для плотных графов, т.е. когда $m \approx n^2$. Чем более разрежен граф (т.е. чем меньше m по сравнению с максимальным количество рёбер n^2), тем менее оптимальной становится эта оценка, и по вине первого слагаемого. Таким образом, надо улучшать время выполнения операций первого типа, не сильно ухудшая при этом время выполнения операций второго типа.
Для этого надо использовать различные вспомогательные структуры данных. Наиболее привлекательными

являются Фибоначчиевы кучи, которые позволяют производить операцию первого вида за O(logn), а второго — за O(1). Поэтому при использовании Фибоначчиевых куч время работы алгоритма Дейкстры составит $O(n \log n + m)$, что является практически теоретическим минимумом для алгоритма поиска кратчайшего пути. Кстати говоря, эта оценка является оптимальной для алгоритмов, основанных на алгоритме Дейкстры, т.е. Фибоначчиевы кучи являются оптимальными с этой точки зрения (это утверждение об оптимальности на самом деле основано на невозможности существования такой "идеальной" структуры данных — если бы она существовала, то можно было бы выполнять сортировку за линейное время, что, как известно, в общем случае невозможно; впрочем, интересно, что существует алгоритм Торупа (Thorup), который ищет кратчайший путь с оптимальной, линейной, асимптотикой, но основан он на совсем другой идее, чем алгоритм Дейкстры, поэтому никакого противоречия здесь нет). Однако, Фибоначчиевы кучи довольно сложны в реализации (и, надо отметить, имеют немалую константу, скрытую в асимптотике).
В качестве компромисса можно использовать структуры данных, позволяющие выполнять оба типа операций (фактически, это извлечение минимума и обновление элемента) за $O(logn)$. Тогда время работы алгоритма Дейкстры составит:

$$O(n \log n + m \log n) = O(m \log n)$$

В качестве такой структуры данных программистам на C++ удобно взять стандартный контейнер set или priority_queue. Первый основан на красно-чёрном дереве, второй — на бинарной куче. Поэтому priority_queue имеет меньшую константу, скрытую в асимпотике, однако у него есть и недостаток: он не поддерживает операцию удаления элемента, из-за чего приходится делать "обходной манёвр", который фактически приводит к замене в асимптотике $\log n$ на $\log m$ (с точки зрения асимптотики это на самом деле ничего не меняет, но скрытую константу увеличивает).

Реализацияс контейнером set. Поскольку в контейнере нам надо хранить вершины, упорядоченные по их величинам d[], то удобно в контейнер помещать пары: первый элемент пары — расстояние, а второй — номер вершины. В результате в set будут храниться пары, автоматически упорядоченные по расстояниям, что нам и нужно.

```
constint INF = 1000000000;

int main() {
	int n;
	... чтение n ...
		vector< vector < pair<int, int>>> g(n);
	... чтение графа ...
		int s = ...; // стартовая вершина

	vector<int> d(n, INF), p(n);
	d[s] = 0;
	set< pair<int, int>> q;
	q.insert(make_pair(d[s], s));
	while (!q.empty()) {
		int v = q.begin()->second;
		q.erase(q.begin());

		for (size_t j = 0; j < g[v].size(); ++j) {
			int to = g[v][j].first,
				len = g[v][j].second;
			if (d[v] + len < d[to]) {
				q.erase(make_pair(d[to], to));
				d[to] = d[v] + len;
				p[to] = v;
				q.insert(make_pair(d[to], to));
			}
		}
	}
}
```

В отличие от обычного алгоритма Дейкстры, становится ненужным массив u[]. Его роль, как и функцию нахождения вершины с наименьшим расстоянием, выполняет set. Изначально в него помещаем стартовую вершину s с её расстоянием. Основной цикл алгоритма выполняется, пока в очереди есть хоть одна вершина. Из очереди извлекается вершина с наименьшим расстоянием, и затем из неё выполняются релаксации. Перед выполнением каждой успешной релаксации мы сначала удаляем из set старую пару, а

затем, после выполнения релаксации, добавляем обратно новую пару (с новым расстоянием d[to]).

5.10 Алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин

Дан ориентированный или неориентированный взвешенный граф G с n вершинами. Требуется найти значения всех величин d_{ij} — длины кратчайшего пути из вершины i в вершину j.
Предполагается, что граф не содержит циклов отрицательного веса (тогда ответа между некоторыми парами вершин может просто не существовать — он будет бесконечно маленьким).
Этот алгоритм был одновременно опубликован в статьях Роберта Флойда (Robert Floyd) и Стивена Уоршелла (Варшалла) (Stephen Warshall) в 1962 г., по имени которых этот алгоритм и называется в настоящее время. Впрочем, в 1959 г. Бернард Рой (Bernard Roy) опубликовал практически такой же алгоритм, но его публикация осталась незамеченной.
Описание алгоритма
Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на фазы.
Перед k-ой фазой (k=1…n) считается, что в матрице расстояний d[][] сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества $\{1, 2, \ldots, k-1\}$ (вершины графа мы нумеруем, начиная с единицы).
Иными словами, перед k-ой фазой величина d[i][j] равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).
Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний d[][] записать матрицу смежности графа: d[i][j]=g[i][j] — стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность" ∞. Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k-ой фазе, и хотим пересчитать матрицу d[][] таким образом, чтобы она соответствовала требованиям уже для k+1-ой фазы. Зафиксируем какие-то вершины i и j. У нас возникает два принципиально разных случая:

Кратчайший путь из вершины i в вершину j, которому разрешено дополнительно проходить через вершины $\{1, 2, \ldots, k\}$, совпадает с кратчайшим путём, которому разрешено проходить через вершины множества $\{1, 2, \ldots, k-1\}$.

В этом случае величина d[i][j] не изменится при переходе с k-ой на k+1-ую фазу.

"Новый" кратчайший путь стал лучше "старого" пути.

Это означает, что "новый" кратчайший путь проходит через вершину k. Сразу отметим, что мы не потеряем общности, рассматривая далее только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая $i \Rightarrow k$, а другая — $k \Rightarrow j$), то каждая из этих половинок уже не заходит в вершину k. Но тогда получается, что длина каждой из этих половинок была посчитана ещё на k-1-ой фазе или ещё раньше, и нам достаточно взять просто сумму d[i][k]+d[k][j], она и даст длину "нового" кратчайшего пути.

Объединяя эти два случая, получаем, что на k-ой фазе требуется пересчитать длины кратчайших путей между всеми парами вершин i и j следующим образом:

new_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);

Таким образом, вся работа, которую требуется произвести на k-ой фазе — это перебрать все пары вершин и пересчитать длину кратчайшего пути между ними. В результате после выполнения n-ой фазы в матрице расстояний d[i][j] будет записана длина кратчайшего пути между i и j, либо ∞, если пути между этими вершинами не существует.

Последнее замечание, которое следует сделать, — то, что можно не создавать отдельную матрицу new_d[][] для временной матрицы кратчайших путей на k-ой фазе: все изменения можно делать сразу в матрице d[][]. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить

тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.
Асимптотика алгоритма, очевидно, составляет $O(n^3)$.
Реализация
На вход программе подаётся граф, заданный в виде матрицы смежности — двумерного массива d[][] размера n*n, в котором каждый элемент задаёт длину ребра между соответствующими вершинами.
Требуется, чтобы выполнялось d[i][i]==0 для любых i.

```
for (int k = 0; k < n; ++k)
	for (int i = 0; i < n; ++i)
		for (int j = 0; j < n; ++j)
			d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

Предполагается, что если между двумя какими-то вершинами нет ребра, то в матрице смежности было записано какое-то большое число (достаточно большое, чтобы оно было больше длины любого пути в этом графе); тогда это ребро всегда будет невыгодно брать, и алгоритм сработает правильно.
Правда, если не принять специальных мер, то при наличии в графе рёбер отрицательного веса, в результирующей матрице могут появиться числа вида ∞-1, ∞-2, и т.д., которые, конечно, по-прежнему означают, что между соответствующими вершинами вообще нет пути. Поэтому при наличии в графе отрицательных рёбер алгоритм Флойда лучше написать так, чтобы он не выполнял переходы из тех состояний, в которых уже стоит "нет пути":

```
for (int k = 0; k < n; ++k)
	for (int i = 0; i < n; ++i)
		for (int j = 0; j < n; ++j)
			if (d[i][k] < INF && d[k][j] < INF)
				d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

5.11 Минимальное остовное дерево. Алгоритм Крускала с системой непересекающихся множеств

Будет рассмотрена реализация с использованием структуры данных "система непересекающихся множеств" (DSU), которая позволит достигнуть асимптотики O (M log N).
Описание

Так же, как и в простой версии алгоритма Крускала, отсортируем все рёбра по неубыванию веса. Затем поместим каждую вершину в своё дерево (т.е. своё множество) с помощью вызова функции DSU MakeSet - на это уйдёт в сумме O (N). Перебираем все рёбра (в порядке сортировки) и для каждого ребра за O (1) определяем, принадлежат ли его концы разным деревьям (с помощью двух вызовов FindSet за O (1)). Наконец, объединение двух деревьев будет осуществляться вызовом Union - также за O (1). Итого мы получаем асимптотику O (M log N + N + M) = O (M log N).

Реализация

Для уменьшения объёма кода реализуем все операции не в виде отдельных функций, а прямо в коде алгоритма Крускала.

Здесь будет использоваться рандомизированная версия DSU.

```
vector<int> p(n);

int dsu_get(intv) {
	return (v == p[v]) ? v : (p[v] = dsu_get(p[v]));
}

void dsu_unite(int a, int b) {
	a = dsu_get(a);
	b = dsu_get(b);
	if (rand() & 1)
		swap(a, b);
	if (a != b)
		p[a] = b;
}

... вфункции main() : ...

	int m;
vector< pair <int, pair<int, int>>> g; // вес - вершина 1 - вершина 2
... чтениеграфа ...

	int cost = 0;
vector< pair<int, int>> res;
```

```
sort(g.begin(), g.end());
p.resize(n);
for (int i = 0; i < n; ++i)
	p[i] = i;
for (int i = 0; i < m; ++i) {
	int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
	if (dsu_get(a) != dsu_get(b)) {
		cost += l;
		res.push_back(g[i].second);
		dsu_unite(a, b);
	}
}
```

5.12 Нахождение Эйлерова пути за O (M)

Эйлеров путь - это путь в графе, проходящий через все его рёбра. Эйлеров цикл - это эйлеров путь, являющийся циклом.
Задача заключается в том, чтобы найти эйлеров путь в неориентированном мультиграфе с петлями.
Алгоритм

Сначала проверим, существует ли эйлеров путь. Затем найдём все простые циклы и объединим их в один - это и будет эйлеровым циклом. Если граф таков, что эйлеров путь не является циклом, то, добавим недостающее ребро, найдём эйлеров цикл, потом удалим лишнее ребро.

Чтобы проверить, существует ли эйлеров путь, нужно воспользоваться следующей теоремой. Эйлеров цикл существует тогда и только тогда, когда степени всех вершин чётны. Эйлеров путь существует тогда и только тогда, когда количество вершин с нечётными степенями равно двум (или нулю, в случае существования эйлерова цикла).
Кроме того, конечно, граф должен быть достаточно связным (т.е. если удалить из него все изолированные вершины, то должен получиться связный граф).
Искать все циклы и объединять их будем одной рекурсивной процедурой:
procedure FindEulerPath (V)

1. перебрать все рёбра, выходящие из вершины V;

каждое такое ребро удаляем из графа, и
вызываем FindEulerPath из второго конца этого ребра;
2. добавляем вершину V в ответ.

Сложность этого алгоритма, очевидно, является линейной относительно числа рёбер.

Но этот же алгоритм мы можем записать в нерекурсивном варианте:

```
stack St;
в St кладём любую вершину (стартовая вершина);
пока St не пустой
        пусть V - значение на вершине St;
        если степень(V) = 0, то
                добавляем V к ответу;
                снимаем V с вершины St;
        иначе
                находим любое ребро, выходящее из V;
                удаляем его из графа;
                второй конец этого ребра кладём в St;
```

Несложно проверить эквивалентность этих двух форм алгоритма. Однако вторая форма, очевидно, быстрее работает, причём кода будет не больше.

Задача о домино

Приведём здесь классическую задачу на эйлеров цикл - задачу о домино.

Имеется N доминошек, как известно, на двух концах доминошки записано по одному числу (обычно от 1 до 6, но в нашем случае не важно). Требуется выложить все доминошки в ряд так, чтобы у любых двух соседних доминошек числа, записанные на их общей стороне, совпадали. Доминошки разрешается переворачивать.

Переформулируем задачу. Пусть числа, записанные на донимошках, - вершины графа, а доминошки - рёбра этого графа (каждая доминошка с числами (a,b) - это ребра (a,b) и (b,a)). Тогда наша задача сводится к задаче нахождения эйлерова пути в этом графе.

5.13 Алгоритм Куна нахождения наибольшего паросочетания в двудольном графе

ан двудольный граф G, содержащий n вершин и m рёбер. Требуется найти наибольшее паросочетание, т.е. выбрать как можно больше рёбер, чтобы ни одно выбранное ребро не имело общей вершины ни с каким другим выбранным ребром.

Описание алгоритма

Необходимые определения

Паросочетанием M называется набор попарно несмежных рёбер графа (иными словами, любой вершине графа должно быть инцидентно не более одного ребра из множества M). Мощностью паросочетания назовём количество рёбер в нём. Наибольшим (или максимальным) паросочетанием назовём паросочетание, мощность которого максимальна среди всех возможных паросочетаний в данном графе. Все те вершины, у которых есть смежное ребро из паросочетания (т.е. которые имеют степень ровно один в подграфе, образованном M), назовём насыщенными этим паросочетанием.

Цепью длины k назовём некоторый простой путь (т.е. не содержащий повторяющихся вершин или рёбер), содержащий k рёбер.

Чередующейся цепью (в двудольном графе, относительно некоторого паросочетания) назовём цепь, в которой рёбра поочередно принадлежат/не принадлежат паросочетанию.

Увеличивающей цепью (в двудольном графе, относительно некоторого паросочетания) назовём чередующуюся цепь, у которой начальная и конечная вершины не принадлежат паросочетанию.

Теорема Бержа

Формулировка. Паросочетание является максимальным тогда и только тогда, когда не существует увеличивающих относительно него цепей.

Доказательство необходимости. Покажем, что если паросочетание M максимально, то не существует увеличивающей относительно него цепи. Доказательство это будет конструктивным: мы покажем, как увеличить с помощью этой увеличивающей цепи P мощность паросочетания M на единицу.

Для этого выполним так называемое чередование паросочетания вдоль цепи P. Мы помним, что по определению первое ребро цепи P не принадлежит паросочетанию, второе — принадлежит, третье — снова не принадлежит, четвёртое — принадлежит, и т.д. Давайте поменяем состояние всех рёбер вдоль цепи P: те рёбра,

которые не входили в паросочетание (первое, третье и т.д. до последнего) включим в паросочетание, а рёбра, которые раньше входили в паросочетание (второе, четвёртое и т.д. до предпоследнего) — удалим из него.
Понятно, что мощность паросочетания при этом увеличилась на единицу (потому что было добавлено на одно ребро больше, чем удалено). Осталось проверить, что мы построили корректное паросочетание, т.е. что никакая вершина графа не имеет сразу двух смежных рёбер из этого паросочетания. Для всех вершин чередующей цепи P, кроме первой и последней, это следует из самого алгоритма чередования: сначала мы у каждой такой вершины удалили смежное ребро, потом добавили. Для первой и последней вершины цепи P также ничего не могло нарушиться, поскольку до чередования они должны были быть ненасыщенными. Наконец, для всех остальных вершин, — не входящих в цепь P, — очевидно, ничего не поменялось. Таким образом, мы в самом деле построили паросочетание, и на единицу большей мощности, чем старое, что и завершает доказательство необходимости.

Доказательство достаточности. Докажем, что если относительно некоторого паросочетания M нет увеличивающих путей, то оно — максимально.
Доказательство проведём от противного. Пусть есть паросочетание M’, имеющее бОльшую мощность, чем M. Рассмотрим симметрическую разность Q этих двух паросочетаний, т.е. оставим все рёбра, входящие в M или в M’, но не в оба одновременно.

Понятно, что множество рёбер Q — уже наверняка не паросочетание. Рассмотрим, какой вид это множество рёбер имеет; для удобства будем рассматривать его как граф. В этом графе каждая вершина, очевидно, имеет степень не выше 2 (потому что каждая вершина может иметь максимум два смежных ребра — из одного паросочетания и из другого). Легко понять, что тогда этот граф состоит только из циклов или путей, причём ни те, ни другие не пересекаются друг с другом.

Теперь заметим, что и пути в этом графе Q могут быть не любыми, а только чётной длины. В самом деле, в любом пути в графе Q рёбра чередуются: после ребра из M идёт ребро из M’, и наоборот. Теперь, если мы рассмотрим какой-то путь нечётной

длины в графе Q, то получится, что в исходном графе G это будет увеличивающей цепью либо для паросочетания M, либо для M'. Но этого быть не могло, потому что в случае паросочетания M это противоречит с условием, а в случае M' — с его максимальностью (ведь мы уже доказали необходимость теоремы, из которой следует, что при существовании увеличивающей цепи паросочетание не может быть максимальным).

Докажем теперь аналогичное утверждение и для циклов: все циклы в графе Q могут иметь только чётную длину. Это доказать совсем просто: понятно, что в цикле рёбра также должны чередоваться (принадлежать по очереди то M, то M'), но это условие не может выполниться в цикле нечётной длины — в нём обязательно найдутся два соседних ребра из одного паросочетания, что противоречит определению паросочетания.

Таким образом, все пути и циклы графа$Q = M \oplus M'$ имеют чётную длину. Следовательно, граф Q содержит равное количество рёбер из M и из M'. Но, учитывая, что в Qсодержатся все рёбра M и M', за исключением их общих рёбер, то отсюда следует, что мощность M и M' совпадают. Мы пришли к противоречию: по предположению паросочетание M было не максимальным, значит, теорема доказана.

Алгоритм Куна

Алгоритм Куна — непосредственное применение теоремы Бержа. Его можно кратко описать так: сначала возьмём пустое паросочетание, а потом — пока в графе удаётся найти увеличивающую цепь, — будем выполнять чередование паросочетания вдоль этой цепи, и повторять процесс поиска увеличивающей цепи. Как только такую цепь найти не удалось — процесс останавливаем, — текущее паросочетание и есть максимальное.

Осталось детализировать способ нахождения увеличивающих цепей. Алгоритм Куна — просто ищет любую из таких цепей с помощью обхода в глубину или в ширину. Алгоритм Куна просматривает все вершины графа по очереди, запуская из каждой обход, пытающийся найти увеличивающую цепь, начинающуюся в этой вершине.

Удобнее описывать этот алгоритм, считая, что граф уже разбит на две доли (хотя на самом деле алгоритм можно

реализовать и так, чтобы ему не давался на вход граф, явно разбитый на две доли).

Алгоритм просматривает все вершины v первой доли графа:$v=1\ldots n_1$. Если текущая вершина v уже насыщена текущим паросочетанием (т.е. уже выбрано какое-то смежное ей ребро), то эту вершину пропускаем. Иначе — алгоритм пытается насытить эту вершину, для чего запускается поиск увеличивающей цепи, начинающейся с этой вершины.

Поиск увеличивающей цепи осуществляется с помощью специального обхода в глубину или ширину (обычно в целях простоты реализации используют именно обход в глубину). Изначально обход в глубину стоит в текущей ненасыщенной вершине v первой доли. Просматриваем все рёбра из этой вершины, пусть текущее ребро — это ребро (v, to). Если вершина to ещё не насыщена паросочетанием, то, значит, мы смогли найти увеличивающую цепь: она состоит из единственного ребра (v, to); в таком случае просто включаем это ребро в паросочетание и прекращаем поиск увеличивающей цепи из вершины v. Иначе, — если to уже насыщена каким-то ребром (p, to), то попытаемся пройти вдоль этого ребра: тем самым мы попробуем найти увеличивающую цепь, проходящую через рёбра (v, to), (to, p). Для этого просто перейдём в нашем обходе в вершину p — теперь мы уже пробуем найти увеличивающую цепь из этой вершины.

Можно понять, что в результате этот обход, запущенный из вершины v, либо найдёт увеличивающую цепь, и тем самым насытит вершину v, либо же такой увеличивающей цепи не найдёт (и, следовательно, эта вершина v уже не сможет стать насыщенной).

После того, как все вершины$v=1\ldots n_1$ будут просмотрены, текущее паросочетание будет максимальным.

Время работы

Итак, алгоритм Куна можно представить как серию из n запусков обхода в глубину/ширину на всём графе. Следовательно, всего этот алгоритм исполняется за время O(nm), что в худшем случае есть $O(n^3)$.

Однако эту оценку можно немного улучшить. Оказывается, для алгоритма Куна важно то, какая доля выбрана за первую, а какая — за вторую. В самом деле, в описанной выше реализации запуски обхода в глубину/ширину происходят только из вершин

первой доли, поэтому весь алгоритм исполняется за время $O(n_1 m)$, где n_1 — число вершин первой доли. В худшем случае это составляет $O(n_1^2 n_2)$ (где n_2 — число вершин второй доли). Отсюда видно, что выгоднее, когда первая доля содержит меньшее число вершин, нежели вторая. На очень несбалансированных графах (когда n_1 и n_2 сильно отличаются) это выливается в значительную разницу времён работы.

Реализация

Приведём здесь реализацию вышеописанного алгоритма, основанную на обходе в глубину, и принимающей двудольный граф в виде явно разбитого на две доли графа. Эта реализация весьма лаконична, и, возможно, её стоит запомнить именно в таком виде.

Здесь n — число вершин в первой доле, k — во второй доле, g[v] — список рёбер из вершины v первой доли (т.е. список номеров вершин, в которые ведут эти рёбра из v). Вершины в обеих долях занумерованы независимо, т.е. первая доля — с номерами 1…n, вторая — с номерами 1…k.

Дальше идут два вспомогательных массива: mt и used. Первый — mt — содержит в себе информацию о текущем паросочетании. Для удобства программирования, информация эта содержится только для вершин второй доли: mt[i] — это номер вершины первой доли, связанной ребром с вершиной i второй доли (или -1, если никакого ребра паросочетания из i не выходит). Второй массив — used — обычный массив "посещённостей" вершин в обходе в глубину (он нужен, просто чтобы обход в глубину не заходил в одну вершину дважды).

Функцияtry_kuhn — и есть обход в глубину. Она возвращает true, если ей удалось найти увеличивающую цепь из вершины v, при этом считается, что эта функция уже произвела чередование паросочетания вдоль найденной цепи.

Внутри функции просматриваются все рёбра, исходящие из вершины v первой доли, и затем проверяется: если это ребро ведёт в ненасыщенную вершину to, либо если эта вершина to насыщена, но удаётся найти увеличивающую цепь рекурсивным запуском из mt[to], то мы говорим, что мы нашли увеличивающую цепь, и перед возвратом из функции с результатом true производим чередование в текущем ребре: перенаправляем ребро, смежное с to, в вершину v.

В основной программе сначала указывается, что текущее паросочетание — пустое (список mt заполняется числами -1). Затем перебирается вершина v первой доли, и из неё запускается обход в глубину try_kuhn, предварительно обнулив массив used.

Стоит заметить, что размер паросочетания легко получить как число вызовов try_kuhn в основной программе, вернувших результат true. Само искомое максимальное паросочетание содержится в массиве mt.

```
int n, k;
vector< vector<int>> g;
vector<int> mt;
vector<char> used;

bool try_kuhn(intv) {
	if (used[v])  returnfalse;
	used[v] = true;
	for (size_t i = 0; i < g[v].size(); ++i) {
		int to = g[v][i];
		if (mt[to] == -1 || try_kuhn(mt[to])) {
			mt[to] = v;
			returntrue;
		}
	}
	returnfalse;
}
int main() {
	... чтениеграфа ...

		mt.assign(k, -1);
	for (int v = 0; v < n; ++v) {
		used.assign(n, false);
		try_kuhn(v);
	}

	for (int i = 0; i < k; ++i)
		if (mt[i] != -1)
			printf("%d %d\n", mt[i] + 1, i + 1);
}
```

Ещё раз повторим, что алгоритм Куна легко реализовать и так, чтобы он работал на графах, про которые известно, что они двудольные, но явное их разбиение на две доли не найдено. В этом случае придётся отказаться от удобного разбиения на две доли, и всю информацию хранить для всех вершин графа. Для этого массив списков g теперь задаётся не только для вершин первой доли, а для всех вершин графа (понятно, теперь вершины обеих долей занумерованы в общей нумерации — от 1 до n). Массивы mt и used теперь также определены для вершин обеих долей, и, соответственно, их нужно поддерживать в этом состоянии.

5.14 Знаковая площадь треугольника и предикат "По часовой стрелке"

Определение

Пусть даны три точки p_1, p_2, p_3. Найдём значение знаковой площади S треугольника $p_1p_2p_3$, т.е. площади этого треугольника, взятой со знаком плюс или минус в зависимости от типа поворота, образуемого точками p_1, p_2, p_3: против часовой стрелки или по ней соответственно.

Понятно, что, если мы научимся вычислять такую знаковую ("ориентированную") площадь, то сможем и находить обычную площадь любого треугольника, а также сможем проверять, по часовой стрелке или против направлена какая-либо тройка точек.

Вычисление

Воспользуемся понятием косого (псевдоскалярного) произведения векторов. Оно как раз равно удвоенной знаковой площади треугольника:

$$a \wedge b = |a||b| sin\angle(a,b) = 2S,$$

где угол $\angle(a,b)$ берётся ориентированным, т.е. это угол вращения между этими векторами против часовой стрелки.

(Модуль косого произведения двух векторов равен модулю векторного произведения их.)

Косое произведение вычисляется как величина определителя, составленного из координат точек:

$$2S = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}.$$

Раскрывая определитель, можно получить такую формулу:

$$2S = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2).$$

Можно сгруппировать третье слагаемое с первыми двумя, избавившись от одного умножения:

$$2S = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1).$$

Последнюю формулу удобно записывать и запоминать в матричном виде, как следующий определитель:

$$2S = \begin{vmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{vmatrix}.$$

Реализация

Функция, вычисляющая удвоенную знаковую площадь треугольника:

```
int triangle_area_2 (int x1, int y1, int x2, int y2, int x3, int y3) {
	return (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
}
```

Функция, возвращающая обычную площадь треугольника:

```
double triangle_area (int x1, int y1, int x2, int y2, int x3, int y3) {
	return abs (triangle_area_2 (x1, y1, x2, y2, x3, y3)) / 2.0;
}
```

Функция, проверяющая, образует ли указанная тройка точек поворот по часовой стрелке:

```
bool clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
	return triangle_area_2 (x1, y1, x2, y2, x3, y3) < 0;
}
```

Функция, проверяющая, образует ли указанная тройка точек поворот против часовой стрелки:

```
bool counter_clockwise (int x1, int y1, int x2, int y2, int x3, int y3) {
	return triangle_area_2 (x1, y1, x2, y2, x3, y3) > 0;
}
```

5.15 Пересечение двух отрезков

Даны два отрезка AB и CD (они могут вырождаться в точки). Требуется найти их пересечение: оно может быть пустым (если отрезки не пересекаются), может быть одной точкой, и может быть целым отрезком (если отрезки накладываются друг на друга).

Алгоритм

Работать с отрезками будем как с прямыми: построим по двум отрезкам уравнения их прямых, проверим, не параллельны ли прямые. Если прямые не параллельны, то всё просто: находим их точку пересечения и проверяем, что она принадлежит обоим отрезкам (для этого достаточно проверить, что точка принадлежит каждому отрезку в проекции на ось X и на ось Y по отдельности). В итоге в этом случае ответом будет либо "пусто", либо единственная найденная точка.

Более сложный случай — если прямые оказались параллельными (сюда же относится случай, когда один или оба отрезка выродились в точки). В этом случае надо проверить, что оба отрезка лежат на одной прямой (или, в случае, когда они оба вырождены в точку — что эта точка совпадает). Если это не так, то ответ — "пусто". Если это так, то ответ — это пересечение двух отрезков, лежащих на одной прямой, что реализуется достаточно просто — надо взять максимум из левых концов и минимум из правых концов.

В самом начале алгоритма напишем так называемую "проверку на bounding box" — во-первых, она необходима для случая, когда два отрезка лежат на одной прямой, а во-вторых, она, как легковесная проверка, позволяет алгоритму работать в среднем быстрее на случайных тестах.

Реализация

Приведём здесь полную реализацию, включая все вспомогательные функции по работе с точками и прямыми.

Главной здесь является функция intersect, которая пересекает два переданных ей отрезка, и если они пересекаются хотя бы по одной точке, то возвращает true, а в аргументах left и right возвращает начало и конец отрезка-ответа (в частности, когда ответ — это единственная точка, возвращаемые начало и конец будут совпадать).

```
constdouble EPS = 1E-9;

structpt {
	double x, y;

	booloperator< (constpt&p) const {
```

```
		return x <p.x - EPS || abs(x - p.x) < EPS && y <p.y -
EPS;
	}
};

structline {
	double a, b, c;

	line() { }
	line(ptp, ptq) {
		a = p.y - q.y;
		b = q.x - p.x;
		c = -a * p.x - b * p.y;
		norm();
	}

	void norm() {
		double z = sqrt(a*a + b * b);
		if (abs(z) > EPS)
			a /= z, b /= z, c /= z;
	}

	double dist(ptp) const {
		return a * p.x + b * p.y + c;
	}
};

#definedet(a,b,c,d)  (a*d-b*c)

inlinebool betw(double l, double r, double x) {
	return min(l, r) <= x + EPS && x <= max(l, r) + EPS;
}

inlinebool intersect_1d(double a, double b, double c, double d) {
	if (a > b)  swap(a, b);
	if (c > d)  swap(c, d);
	return max(a, c) <= min(b, d) + EPS;
}
```

```
bool intersect(pt a, pt b, pt c, pt d, pt & left, pt & right) {
	if (!intersect_1d(a.x, b.x, c.x, d.x) || !intersect_1d(a.y, b.y, c.y,
d.y))
		returnfalse;
	line m(a, b);
	line n(c, d);
	double zn = det(m.a, m.b, n.a, n.b);
	if (abs(zn) < EPS) {
		if (abs(m.dist(c)) > EPS || abs(n.dist(a)) > EPS)
			returnfalse;
		if (b < a)  swap(a, b);
		if (d < c)  swap(c, d);
		left = max(a, c);
		right = min(b, d);
		returntrue;
	}
	else {
		left.x = right.x = -det(m.c, m.b, n.c, n.b) / zn;
		left.y = right.y = -det(m.a, m.c, n.a, n.c) / zn;
		return betw(a.x, b.x, left.x)
			&&betw(a.y, b.y, left.y)
			&&betw(c.x, d.x, left.x)
			&&betw(c.y, d.y, left.y);
	}
}
```

5.15 Пересечение окружности и прямой

Дана окружность (координатами своего центра и радиусом) и прямая (своим уравнением). Требуется найти точки их пересечения (одна, две, либо ни одной).

Решение

Вместо формального решения системы двух уравнений подойдём к задаче с геометрической стороны (причём, за счёт этого мы получим более точное решение с точки зрения численной устойчивости).

Предположим, не теряя общности, что центр окружности находится в начале координат (если это не так, то перенесём его туда, исправив соответствующе константу C в уравнении прямой). Т.е. имеем окружность с центром в (0,0) радиуса r и прямую с уравнением Ax + By + C = 0.

Сначала найдём ближайшую к центру точку прямой - точку с некоторыми координатами (x0,y0). Во-первых, эта точка должна находиться на таком расстоянии от начала координат:

$$\frac{|C|}{sqrt(A2+B2)}$$

Во-вторых, поскольку вектор (A,B) перпендикулярен прямой, то координаты этой точки должны быть пропорциональны координатам этого вектора. Учитывая, что расстояние от начала координат до искомой точки нам известно, нам нужно просто нормировать вектор (A,B) к этой длине, и мы получаем:

$$x0 = -\frac{AC}{A2+B2}$$

$$y0 = -\frac{BC}{A2+B2}$$

(здесь неочевидны только знаки 'минус', но эти формулы легко проверить подстановкой в уравнение прямой - должен получиться ноль)

Зная ближайшую к центру окружности точку, мы уже можем определить, сколько точек будет содержать ответ, и даже дать ответ, если этих точек 0 или 1.

Действительно, если расстояние от (x0, y0) до начала координат (а его мы уже выразили формулой - см. выше) больше радиуса, то ответ - ноль точек. Если это расстояние равно радиусу, то ответом будет одна точка - (x0,y0). А вот в оставшемся случае точек будет две, и их координаты нам предстоит найти.

Итак, мы знаем, что точка (x0, y0) лежит внутри круга. Искомые точки (ax,ay) и (bx,by), помимо того что должны принадлежать прямой, должны лежать на одном и том же расстоянии d от точки (x0, y0), причём это расстояние легко найти:

$$d = sqrt(r2 - \frac{C2}{A2+B2})$$

Заметим, что вектор (-B,A) коллинеарен прямой, а потому искомые точки (ax,ay) и (bx,by) можно получить, прибавив к точке (x0,y0) вектор (-B,A), нормированный к длине d (мы получим одну искомую точку), и вычтя этот же вектор (получим вторую искомую точку).

Окончательное решение такое:

$$mult = sqrt(\frac{d2}{A2 + B2})$$

$$ax = x0 + B\ mult$$

$$ay = y0 - A\ mult$$

$$bx = x0 - B\ mult$$

$$by = y0 + A\ mult$$

Если бы мы решали эту задачу чисто алгебраически, то скорее всего получили бы решение в другом виде, которое даёт бОльшую погрешность. Поэтому "геометрический" метод, описанный здесь, помимо наглядности, ещё и более точен.

Реализация

Как и было указано в начале описания, предполагается, что окружность расположена в начале координат.

Поэтому входные параметры - это радиус окружности и коэффициенты A,B,C уравнения прямой.

```
double r, a, b, c; // входные данные

double x0 = -a * c / (a*a + b * b), y0 = -b * c / (a*a + b * b);
if (c*c > r*r*(a*a + b * b) + EPS)
	puts("no points");
elseif (abs(c*c - r * r*(a*a + b * b)) < EPS) {
	puts("1 point");
	cout<< x0 <<' '<< y0 <<'\n';
}
else {
	double d = r * r - c * c / (a*a + b * b);
	double mult = sqrt(d / (a*a + b * b));
	double ax, ay, bx, by;
	ax = x0 + b * mult;
	bx = x0 - b * mult;
	ay = y0 - a * mult;
	by = y0 + a * mult;
```

```
        puts("2 points");
        cout<< ax <<' '<< ay <<'\n'<< bx <<' '<< by <<'\n';
}
```

5.16 Построение выпуклой оболочки обходом Грэхэма

Даны N точек на плоскости. Построить их выпуклую оболочку, т.е. наименьший выпуклый многоугольник, содержащий все эти точки.

Мы рассмотрим метод Грэхэма (Graham) (предложен в 1972 г.) с улучшениями Эндрю (Andrew) (1979 г.). С его помощью можно построить выпуклую оболочку за время O (N log N) с использованием только операций сравнения, сложения и умножения. Алгоритм является асимптотически оптимальным (доказано, что не существует алгоритма с лучшей асимптотикой), хотя в некоторых задачах он неприемлем (в случае параллельной обработки или при online-обработке).

Описание

Алгоритм. Найдём самую левую и самую правую точки A и B (если таких точек несколько, то возьмём самую нижнюю среди левых, и самую верхнюю среди правых). Понятно, что и A, и B обязательно попадут в выпуклую оболочку. Далее, проведём через них прямую AB, разделив множество всех точек на верхнее и нижнее подмножества S1 и S2 (точки, лежащие на прямой, можно отнести к любому множеству - они всё равно не войдут в оболочку). Точки A и B отнесём к обоим множествам. Теперь построим для S1 верхнюю оболочку, а для S2 - нижнюю оболочку, и объединим их, получив ответ. Чтобы получить, скажем, верхнюю оболочку, нужно отсортировать все точки по абсциссе, затем пройтись по всем точкам, рассматривая на каждом шаге кроме самой точки две предыдущие точки, вошедшие в оболочку. Если текущая тройка точек образует не правый поворот (что легко проверить с помощью Ориентированной площади), то ближайшего соседа нужно удалить из оболочки. В конце концов, останутся только точки, входящие в выпуклую оболочку.

Итак, алгоритм заключается в сортировке всех точек по абсциссе и двух (в худшем случае) обходах всех точек, т.е. требуемая асимптотика O (N log N) достигнута.

Реализация

```
structpt {
	double x, y;
};

bool cmp(pta, ptb) {
	return a.x < b.x || a.x == b.x && a.y < b.y;
}

bool cw(pt a, pt b, pt c) {
	return a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y) < 0;
}

bool ccw(pt a, pt b, pt c) {
	return a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y) > 0;
}

void convex_hull(vector<pt>& a) {
	if (a.size() == 1)  return;
	sort(a.begin(), a.end(), &cmp);
	pt p1 = a[0], p2 = a.back();
	vector<pt> up, down;
	up.push_back(p1);
	down.push_back(p1);
	for (size_t i = 1; i < a.size(); ++i) {
		if (i == a.size() - 1 || cw(p1, a[i], p2)) {
			while (up.size() >= 2 && !cw(up[up.size() - 2],
up[up.size() - 1], a[i]))
				up.pop_back();
			up.push_back(a[i]);
		}
		if (i == a.size() - 1 || ccw(p1, a[i], p2)) {
			while (down.size() >= 2 &&
!ccw(down[down.size() - 2], down[down.size() - 1], a[i]))
				down.pop_back();
			down.push_back(a[i]);
		}
	}
```

```
        a.clear();
        for (size_t i = 0; i < up.size(); ++i)
                a.push_back(up[i]);
        for (size_t i = down.size() - 2; i > 0; --i)
                a.push_back(down[i]);
}
```

5.17 Префикс-функция. Алгоритм Кнута-Морриса-Пратта

Префикс-функция. Определение

Дана строка$s = [0 \dots n-1]$. Требуется вычислить для неё префикс-функцию, т.е. массив чисел$\pi[0 \dots n-1]$, где $\pi[n]$ определяется следующим образом: это такая наибольшая длина наибольшего собственного суффикса подстроки $s[1 \dots n]$, совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$ полагается равным нулю.

Математически определение префикс-функции можно записать следующим образом:

$$\pi[i] = \max\{k : s[0 \dots k-1] = s[i-k+1 \dots i]\}.$$

Например, для строки "abcabcd" префикс-функция равна: [0,0,0,1,2,3,0], что означает:

у строки "a" нет нетривиального префикса, совпадающего с суффиксом;

у строки "ab" нет нетривиального префикса, совпадающего с суффиксом;

у строки "abc" нет нетривиального префикса, совпадающего с суффиксом;

у строки "abca" префикс длины 1 совпадает с суффиксом;

у строки "abcab" префикс длины 2 совпадает с суффиксом;

у строки "abcabc" префикс длины 3 совпадает с суффиксом;

у строки "abcabcd" нет нетривиального префикса, совпадающего с суффиксом.

Другой пример — для строки "aabaaab" она равна: [0,1,0,1,2,2,3].

Тривиальный алгоритм

Непосредственно следуя определению, можно написать такой алгоритм вычисления префикс-функции:

```
vector<int> prefix_function(string s) {
	int n = (int)s.length();
	vector<int> pi(n);
	for (int i = 0; i < n; ++i)
		for (int k = 0; k <= i; ++k)
			if (s.substr(0, k) == s.substr(i - k + 1, k))
				pi[i] = k;
	return pi;
}
```

Как нетрудно заметить, работать он будет за $O(n^3)$, что слишком медленно.

Эффективный алгоритм

Этот алгоритм был разработан Кнутом (Knuth) и Праттом (Pratt) и независимо от них Моррисом (Morris) в 1977 г. (как основной элемент для алгоритма поиска подстроки в строке).

Первая оптимизация

Первое важное замечание — что значение $\pi[i+1]$ не более чем на единицу превосходит значение $\pi[i]$ для любого i.

Действительно, в противном случае, если бы $\pi[i+1] > \pi[i]+1$, то рассмотрим этот суффикс, оканчивающийся в позиции i+1 и имеющий длину $\pi[i+1]$ — удалив из него последний символ, мы получим суффикс, оканчивающийся в позиции i и имеющий длину $\pi[i+1]-1$, что лучше $\pi[i]$, т.е. пришли к противоречию. Иллюстрация этого противоречия (в этом примере $\pi[i-1]$ должно быть равно 3):

$\pi[i-1] = 2\pi[i-1] = 2$

$$\underbrace{\overbrace{s_0\ s_1}\ s_2\ s_3}\ \dots\ \underbrace{s_{i-3}\ \overbrace{s_{i-2}\ s_{i-1}}\ s_i}$$

$\pi[i] = 4\pi[i] = 4$

(на этой схеме верхние фигурные скобки обозначают две одинаковые подстроки длины 2, нижние фигурные скобки — две одинаковые подстроки длины 4)

Таким образом, при переходе к следующей позиции очередной элемент префикс-функции мог либо увеличиться на единицу, либо не измениться, либо уменьшиться на какую-либо величину. Уже этот факт позволяет нам снизить асимптотику до O(n^2) — поскольку за один шаг значение могло вырасти

максимум на единицу, то суммарно для всей строки могло произойти максимум n увеличений на единицу, и, как следствие (т.к. значение никогда не могло стать меньше нуля), максимум n уменьшений. В итоге получится O(n) сравнений строк, т.е. мы уже достигли асимптотики O(n[2]).

Вторая оптимизация

Пойдём дальше — избавимся от явных сравнений подстрок. Для этого постараемся максимально использовать информацию, вычисленную на предыдущих шагах.

Итак, пусть мы вычислили значение префикс-функции $\pi[i]$ для некоторого i. Теперь, если$s[i+1] = s[\pi[i]]$, то мы можем с уверенностью сказать, что$\pi[i+1] = \pi[i] + 1$, это иллюстрирует схема:

$$\pi[i] s_3 = s_{i+1} \pi[i] s_3 = s_{i+1}$$

$$s_0\ s_1\ s_2\ s_3\ \ \dots\ \ s_{i-2}\ s_{i-1}\ s_i\ s_{i+1}$$

$$\pi[i+1] = \pi[i] + 1 \pi[i+1] = \pi[i] + 1$$

(на этой схеме снова одинаковые фигурные скобки обозначают одинаковые подстроки)

Пусть теперь, наоборот, оказалось, что $s[i+1] \neq s[\pi[i]]$. Тогда нам надо попытаться попробовать подстроку меньшей длины. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине $j < \pi[i]$, что по-прежнему выполняется префикс-свойство в позиции i, т.е.s[0…j-1]=s[i-j+1…i]:

$$\pi[i] \pi[i]$$

$$s_0\ s_1\ s_2\ s_3\ \ \dots\ \ s_{i-3} s_{i-2}\ s_{i-1}\ s_i\ s_{i+1}$$

$$jj$$

Действительно, когда мы найдём такую длину j, то нам будет снова достаточно сравнить символы s[i+1] иs[j] — если они совпадут, то можно утверждать, что$\pi[i+1] = j + 1$. Иначе нам надо будет снова найти меньшее (следующее по величине) значение j, для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения j кончатся — это

происходит, когда j=0. В этом случае, если s[i+1]=s[0], то $\pi[i+1]=1$, иначе $\pi[i+1]=0$.

Итак, общая схема алгоритма у нас уже есть, нерешённым остался только вопрос об эффективном нахождении таких длин j. Поставим этот вопрос формально: по текущей длине j и позиции j (для которых выполняется префикс-свойство, т.е.s[0…j-1]=s[i-j+1…i]) требуется найти наибольшее k<j, для которого по-прежнему выполняется префикс-свойство:

$$jj$$

$$\underbrace{\overbrace{s_0\ s_1}\ s_2\ s_3}\quad \dots \quad \overbrace{s_{i-3} s_{i-2}\ \underbrace{s_{i-1}\ s_i}}\ s_{i+1}$$

$$kk$$

После столь подробного описания уже практически напрашивается, что это значение k есть не что иное, как значение префикс-функции $\pi[j-1]$, которое уже было вычислено нами ранее (вычитание единицы появляется из-за 0-индексации строк). Таким образом, находить эти длины k мы можем за O(1) каждую.

Итоговый алгоритм

Итак, мы окончательно построили алгоритм, который не содержит явных сравнений строк и выполняет O(n) действий.

Приведём здесь итоговую схему алгоритма:

Считать значения префикс-функции $\pi[i]$ будем по очереди: от i=1 к i=n-1 (значение $\pi[0]$ просто присвоим равным нулю).

Для подсчёта текущего значения $\pi[i]$ мы заводим переменную j, обозначающую длину текущего рассматриваемого образца. Изначально $j=\pi[i-1]$.

Тестируем образец длины j, для чего сравниваем символы s[i] и s[j]. Если они совпадают — то полагаем $\pi[i]=j+1$ и переходим к следующему индексу i+1. Если же символы отличаются, то уменьшаем длину j, полагая её равной $\pi[j-1]$, и повторяем этот шаг алгоритма с начала.

Если мы дошли до длины j=0 и так и не нашли совпадения, то останавливаем процесс перебора образцов и полагаем $\pi[i]=0$ и переходим к следующему индексу i+1.

Реализация

Алгоритм в итоге получился удивительно простым и лаконичным:

```
vector<int> prefix_function(string s) {
	int n = (int)s.length();
	vector<int> pi(n);
	for (int i = 1; i < n; ++i) {
		int j = pi[i - 1];
		while (j > 0 && s[i] != s[j])
			j = pi[j - 1];
		if (s[i] == s[j])  ++j;
		pi[i] = j;
	}
	return pi;
}
```

Как нетрудно заметить, этот алгоритм является онлайновым алгоритмом, т.е. он обрабатывает данные по ходу поступления — можно, например, считывать строку по одному символу и сразу обрабатывать этот символ, находя ответ для очередной позиции. Алгоритм требует хранения самой строки и предыдущих вычисленных значений префикс-функции, однако, как нетрудно заметить, если нам заранее известно максимальное значение, которое может принимать префикс-функция на всей строке, то достаточно будет хранить лишь на единицу большее количество первых символов строки и значений префикс-функции.

5.18 Алгоритмы хэширования в задачах на строки

Алгоритмы хэширования строк помогают решить очень много задач. Но у них есть большой недостаток: что чаще всего они не 100%-ны, поскольку есть множество строк, хэши которых совпадают. Другое дело, что в большинстве задач на это можно не обращать внимания, поскольку вероятность совпадения хэшей всё-таки очень мала.

Определение хэша и его вычисление

Один из лучших способов определить хэш-функцию от строки S следующий:

h(S) = S[0] + S[1] * P + S[2] * P^2 + S[3] * P^3 + ... + S[N] * P^N

где P - некоторое число.
Разумно выбирать для P простое число, примерно равное количеству символов во входном алфавите. Например, если строки предполаются состоящими только из маленьких латинских букв, то хорошим выбором будет P = 31. Если буквы могут быть и заглавными, и маленькими, то, например, можно P = 53.
Во всех кусках кода в этой статье будет использоваться P = 31.
Само значение хэша желательно хранить в самом большом числовом типе - int64, он же long long. Очевидно, что при длине строки порядка 20 символов уже будет происходить переполнение значение. Ключевой момент - что мы не обращаем внимание на эти переполнения, как бы беря хэш по модулю 2^64.
Пример вычисления хэша, если допустимы только маленькие латинские буквы:

```
constintp = 31;
longlong hash = 0, p_pow = 1;
for (size_t i = 0; i < s.length(); ++i)
{
	// желательно отнимать 'a' от кода буквы
	// единицу прибавляем, чтобы у строки вида 'aaaaa' хэш был
ненулевой
	hash += (s[i] - 'a' + 1) * p_pow;
	p_pow *= p;
}
```

В большинстве задач имеет смысл сначала вычислить все нужные степени P в каком-либо массиве.

5.19 Алгоритм Ахо-Корасик

Пусть дан набор строк в алфавите размера k суммарной длины m. Алгоритм Ахо-Корасик строит для этого набора строк структуру данных "бор", а затем по этому бору строит автомат, всё за O(m) времени и O(mk) памяти. Полученный автомат уже может использоваться в различных задачах, простейшая из которых — это нахождение всех вхождений каждой строки из данного набора в некоторый текст за линейное время.

Данный алгоритм был предложен канадским учёным Альфредом Ахо (Alfred Vaino Aho) и учёным Маргарет Корасик (Margaret John Corasick) в 1975 г.

Бор. Построение бора

Формально, бор — это дерево с корнем в некоторой вершине Root, причём каждое ребро дерево подписано некоторой буквой. Если мы рассмотрим список рёбер, выходящих из данной вершины (кроме ребра, ведущего в предка), то все рёбра должны иметь разные метки.

Рассмотрим в боре любой путь из корня; выпишем подряд метки рёбер этого пути. В результате мы получим некоторую строку, которая соответствует этому пути. Если же мы рассмотрим любую вершину бора, то ей поставим в соответствие строку, соответствующую пути из корня до этой вершины.

Каждая вершина бора также имеет флаг leaf, который равен true, если в этой вершине оканчивается какая-либо строка из данного набора.

Соответственно, построить бор по данному набору строк — значит построить такой бор, что каждой leaf-вершине будет соответствовать какая-либо строка из набора, и, наоборот, каждой строке из набора будет соответствовать какая-то leaf-вершина.

Опишем теперь, как построить бор по заданному набору строк за линейное время относительно их суммарной длины.

Введём структуру, соответствующую вершинам бора:

```
struct vertex {
	int next[K];
	bool leaf;
};

vertex t[NMAX+1];
intsz;
```

Т.е. мы будем хранить бор в виде массива t (количество элементов в массиве - это sz) структур vertex. Структура vertex содержит флаг leaf, и рёбра в виде массива next[], где next[i] — указатель на вершину, в которую ведёт ребро по символу i, или -1, если такого ребра нет.

Вначале бор состоит только из одной вершины — корня (договоримся, что корень всегда имеет в массиве t индекс 0). Поэтому инициализация бора такова:

```
memset (t[0].next, 255, sizeoft[0].next);
sz = 1;
```

Теперь реализуем функцию, которая будет добавлять в бор заданную строку s. Реализация крайне проста: мы встаём в корень бора, смотрим, есть ли из корня переход по букве s[0]: если переход есть, то просто переходим по нему в другую вершину, иначе создаём новую вершину и добавляем переход в эту вершину по букве s[0]. Затем мы, стоя в какой-то вершине, повторяем процесс для буквы s[1], и т.д. После окончания процесса помечаем последнюю посещённуювершинуфлагом leaf=true.

```
void add_string(const string &s) {
	int v = 0;
	for (size_t i = 0; i < s.length(); ++i) {
		char c = s[i] - 'a'; // в зависимости от алфавита
		if (t[v].next[c] == -1) {
			memset(t[sz].next, 255, sizeof t[sz].next);
			t[v].next[c] = sz++;
		}
		v = t[v].next[c];
	}
	t[v].leaf = true;
}
```

Линейное время работы, а также линейное количество вершин в боре очевидны. Поскольку на каждую вершину приходится O(k) памяти, то использование памяти есть O(nk).

Потребление памяти можно уменьшить до линейного (O(n)), но за счёт увеличения асимптотики работы до O(nlogk). Для этого достаточно хранить переходы next не массивом, а отображением map<char, int>.

Построение автомата

Пусть мы построили бор для заданного набора строк. Посмотрим на него теперь немного с другой стороны. Если мы рассмотрим любую вершину, то строка, которая соответствует ей, является префиксом одной или нескольких строк из набора; т.е.

каждую вершину бора можно понимать как позицию в одной или нескольких строках из набора.

Фактически, вершины бора можно понимать как состояния конечного детерминированного автомата. Находясь в каком-либо состоянии, мы под воздействием какой-то входной буквы переходим в другое состояние — т.е. в другую позицию в наборе строк. Например, если в боре находится только строка “abc” и мы стоим в состоянии 2 (которому соответствует строка “ab”), то под воздействием буквы “c” мы перейдём в состояние 3.

Т.е. мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние.

Более строго, пусть мы находимся в состоянии p, которому соответствует некоторая строка t, и хотим выполнить переход по символу c. Если в боре из вершины p есть переход по букве c, то мы просто переходим по этому ребру и попадаем в вершину, которой соответствует строка tc. Если же такого ребра нет, то мы должны найти состояние, соответствующее наидлиннейшему собственному суффиксу строки t (наидлиннейшему из имеющихся в боре), и попытаться выполнить переход по букве c из него.

Например, пусть бор построен по строкам “ab” и “bc”, и мы под воздействием строки “ab” перешли в некоторое состояние, являющееся листом. Тогда под воздействием буквы “c” мы вынуждены перейти в состояние, соответствующее строке “b”, и только оттуда выполнить переход по букве “c”.

Суффиксная ссылка для каждой вершины p — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине p. Единственный особый случай — корень бора; для удобства суффиксную ссылку из него проведём в себя же. Теперь мы можем переформулировать утверждение по поводу переходов в автомате так: пока из текущей вершины бора нет перехода по соответствующей букве (или пока мы не придём в корень бора), мы должны переходить по суффиксной ссылке.

Таким образом, мы свели задачу построения автомата к задаче нахождения суффиксных ссылок для всех вершин бора. Однако

строить эти суффиксные ссылки мы будем, как ни странно, наоборот, с помощью построенных в автомате переходов.
Заметим, что если мы хотим узнать суффиксную ссылку для некоторой вершины v, то мы можем перейти в предка p текущей вершины (пусть c — буква, по которой из p есть переход в v), затем перейти по его суффиксной ссылке, а затем из неё выполнить переход в автомате по букве c.

Таким образом, задача нахождения перехода свелась к задаче нахождения суффиксной ссылки, а задача нахождения суффиксной ссылки — к задаче нахождения суффиксной ссылки и перехода, но уже для более близких к корню вершин. Мы получили рекурсивную зависимость, но не бесконечную, и, более того, разрешить которую можно за линейное время.

Перейдём теперь к реализации. Заметим, что нам теперь понадобится для каждой вершины хранить её предка p, а также символ pch, по которому из предка есть переход в нашу вершину. Также в каждой вершине будем хранить intlink — суффиксная ссылка (или -1, если она ещё не вычислена), и массив intgo[k] — переходы в автомате по каждому из символов (опять же, если элемент массива равен -1, то он ещё не вычислен). Приведём теперь полную реализацию всех необходимых функций:

```
structvertex {
	intnext[K];
	bool leaf;
	int p;
	char pch;
	int link;
	int go[K];
};

vertex t[NMAX + 1];
int sz;

void init() {
	t[0].p = t[0].link = -1;
	memset(t[0].next, 255, sizeof t[0].next);
	memset(t[0].go, 255, sizeof t[0].go);
	sz = 1;
```

```
}

void add_string(const string & s) {
	int v = 0;
	for (size_t i = 0; i < s.length(); ++i) {
		char c = s[i] - 'a';
		if (t[v].next[c] == -1) {
			memset(t[sz].next, 255, sizeof t[sz].next);
			memset(t[sz].go, 255, sizeof t[sz].go);
			t[sz].link = -1;
			t[sz].p = v;
			t[sz].pch = c;
			t[v].next[c] = sz++;
		}
		v = t[v].next[c];
	}
	t[v].leaf = true;
}

int go(int v, char c);

int get_link(intv) {
	if (t[v].link == -1)
		if (v == 0 || t[v].p == 0)
			t[v].link = 0;
		else
			t[v].link = go(get_link(t[v].p), t[v].pch);
	return t[v].link;
}

int go(int v, char c) {
	if (t[v].go[c] == -1)
		if (t[v].next[c] != -1)
			t[v].go[c] = t[v].next[c];
		else
			t[v].go[c] = v == 0 ? 0 : go(get_link(v), c);
	return t[v].go[c];
}
```

Нетрудно понять, что, за счёт запоминания найденных суффиксных ссылок и переходов, суммарное время нахождения всех суффиксных ссылок и переходов будет линейным.

5.20 Sqrt-декомпозиция

Sqrt-декомпозиция — это метод, или структура данных, которая позволяет выполнять некоторые типичные операции (суммирование элементов подмассива, нахождение минимума/максимума и т.д.) за $O(\sqrt{n})$, что значительно быстрее, чем O(n) для тривиального алгоритма.

Сначала мы опишем структуру данных для одного из простейших применений этой идеи, затем покажем, как обобщать её для решения некоторых других задач, и, наконец, рассмотрим несколько иное применение этой идеи: разбиение входных запросов на sqrt-блоки.

Структура данных на основе sqrt-декомпозиции

Поставим задачу. Дан массива[0…n-1]. Требуется реализовать такую структуру данных, которая сможет находить сумму элементов a[l…r] для произвольных l и r за $O(\sqrt{n})$ операций.

Описание

Основная идея sqrt-декомпозиции заключается в том, что сделаем следующий предпосчёт: разделим массив a на блоки длины примерно $\sqrt{n}$, и в каждом блоке i заранее предпосчитаем сумму b[i] элементов в нём.

Можно считать, что длина одного блока и количество блоков равны одному и тому же числу — корню из n, округлённому вверх:

S=$\lceil\sqrt{n}\rceil$

тогда массив a[] разбивается на блоки примерно таким образом:

$$\underbrace{a[0]\ a[1]\ \dots\ a[s-1]}_{b[0]}\quad \underbrace{a[s]\ a[s+1]\ \dots\ a[2*s-1]}_{b[1]}\quad \dots\quad \underbrace{a[(s-1)*s]\ \dots\ a[n]}_{b[s-1]}$$

Хотя последний блок может содержать меньше, чем s, элементов (если n не делится на s), — это не принципиально.

Таким образом, для каждого блока k мы знаем сумму на нём b[k]:

$$b[k] = \sum_{i=k*s}^{\min(n-1,(k+1)*s-1)} a[i]$$

Итак, пусть эти значения b_k предварительно подсчитаны (для этого надо, очевидно, O(n) операций). Что они могут дать при вычислении ответа на очередной запрос (l, r)? Заметим, что если отрезок [l;r] длинный, то в нём будут содержаться несколько блоков целиком, и на такие блоки мы можем узнать сумму на них за одну операцию. В итоге от всего отрезка [l;r] останется лишь два блока, попадающие в него лишь частично, и на этих кусках нам придётся произвести суммирование тривиальным алгоритмом.

Иллюстрация (здесь через k обозначен номер блока, в котором лежит l, а через p — номер блока, в котором лежит r):

sum=?

… a[l] … a[(k+1)*s-1] a[(k+1)*s] … a[(k+2)*s-1] … a[(p-1)*s] … a[p*s-1] a[p*s] … a_r …

b[k+1] b[p]

На этом рисунке видно, что для того чтобы посчитать сумму в отрезке [l…r], надо просуммировать элементы только в двух "хвостах": [l…(k+1)*s-1] и[p*s…r], и просуммировать значения b[i] во всех блоках, начиная с k+1 и заканчивая p-1:

$$\sum_{i=l}^{r} a[i] = \sum_{i=l}^{(k+1)*s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p*s}^{r} a[i]$$

(примечание: эта формула неверна, когда k=p: в таком случае некоторые элементы будут просуммированы дважды; в этом случае надо просто просуммировать элементы с l по r)

Тем самым мы экононим значительное количество операций. Действительно, размер каждого из "хвостов", очевидно, не превосходит длины блока s, и количество блоков также не превосходит s. Поскольку s мы выбирали $\approx \sqrt{n}$, то всего для вычисления суммы на отрезке [l…r] нам понадобится лишь $O(\sqrt{n})$ операций.

Реализация

Приведём сначала простейшую реализацию:

```
// входныеданные
int n;
vector<int> a(n);

// предпосчёт
int len = (int)sqrt(n + .0) + 1; // и размер блока, и количество блоков
```

```
vector<int> b(len);
for (int i = 0; i < n; ++i)
	b[i / len] += a[i];

// ответ на запросы
for(;;) {
	int l, r; // считываем входные данные - очередной запрос
	int sum = 0;
	for (int i = l; i <= r; )
		if (i % len == 0 && i + len - 1 <= r) {
			// если i указывает на начало блока, целиком
лежащего в [l;r]
			sum += b[i / len];
			i += len;
		}
		else {
			sum += a[i];
			++i;
		}
}
```

Недостатком этой реализации является то, что в ней неоправданно много операций деления (которые, как известно, выполняются значительно медленнее других операций). Вместо этого можно посчитать номера блоков c_l и c_r, в которых лежат границы l и r соответственно, и затем сделать цикл по блокам с c_l+1 по c_r+1, отдельно обработав "хвосты" в блоках c_l и c_r. Кроме того, при такой реализации случай c_l=c становится особым и требует отдельной обработки:

```
int sum = 0;
int c_l = l / len, c_r = r / len;
if (c_l == c_r)
	for (int i = l; i <= r; ++i)
		sum += a[i];
else {
	for (int i = l, end = (c_l + 1)*len - 1; i <= end; ++i)
		sum += a[i];
	for (int i = c_l + 1; i <= c_r - 1; ++i)
		sum += b[i];
```

```
        for (int i = c_r * len; i <= r; ++i)
                sum += a[i];
}
```

5.21 Система непересекающихся множеств

Эта структура данных предоставляет следующие возможности. Изначально имеется несколько элементов, каждый из которых находится в отдельном (своём собственном) множестве. За одну операцию можно объединить два каких-либо множества, а также можно запросить, в каком множестве сейчас находится указанный элемент. Также, в классическом варианте, вводится ещё одна операция — создание нового элемента, который помещается в отдельное множество.

Таким образом, базовый интерфейс данной структуры данных состоит всего из трёх операций:
make_set(x) — добавляет новый элемент x, помещая его в новое множество, состоящее из одного него.
union_sets(x,y) — объединяет два указанных множества (множество, в котором находится элемент x, и множество, в котором находится элемент y).
find_set(x) — возвращает, в каком множестве находится указанный элемент x. На самом деле при этом возвращается один из элементов множества (называемый представителем или лидером (в англоязычной литературе "leader")). Этот представитель выбирается в каждом множестве самой структурой данных (и может меняться с течением времени, а именно, после вызовов union_sets()).

Например, если вызов find_set() для каких-то двух элементов вернул одно и то же значение, то это означает, что эти элементы находятся в одном и том же множестве, а в противном случае — в разных множествах.

Описываемая ниже структура данных позволяет делать каждую из этих операций почти за O(1) в среднем (более подробно об асимптотике см. ниже после описания алгоритма).
Также в одном из подразделов статьи описан альтернативный вариант реализации DSU, позволяющий добиться асимптотики O(logn) в среднем на один запрос при $m \leq n$; а при $m \gg n$ (т.е. m значительно больше n) — и вовсе времени O(1) в

среднем на запрос (см. "Хранение DSU в виде явного списка множеств").

Построение эффективной структуры данных

Определимся сначала, в каком виде мы будем хранить всю информацию.

Множества элементов мы будем хранить в виде деревьев: одно дерево соответствует одному множеству. Корень дерева — это представитель (лидер) множества.

При реализации это означает, что мы заводим массив parent, в котором для каждого элемента мы храним ссылку на его предка в дерева. Для корней деревьев будем считать, что их предок — они сами (т.е. ссылка зацикливается в этом месте).

Наивная реализация

Мы уже можем написать первую реализацию системы непересекающихся множеств. Она будет довольно неэффективной, но затем мы улучшим её с помощью двух приёмов, получив в итоге почти константное время работы.

Итак, вся информация о множествах элементов хранится у нас с помощью массива parent.

Чтобы создать новый элемент (операция make_set(v)), мы просто создаём дерево с корнем в вершине v, отмечая, что её предок — это она сама.

Чтобы объединить два множества (операция union_sets(a,b)), мы сначала найдём лидеров множества, в котором находится a, и множества, в котором находится b. Если лидеры совпали, то ничего не делаем — это значит, что множества и так уже были объединены. В противном случае можно просто указать, что предок вершины b равен a (или наоборот) — тем самым присоединив одно дерево к другому.

Наконец, реализация операции поиска лидера (find_set(v)) проста: мы поднимаемся по предкам от вершины v, пока не дойдём до корня, т.е. пока ссылка на предка не ведёт в себя. Эту операцию удобнее реализовать рекурсивно (особенно это будет удобно позже, в связи с добавляемыми оптимизациями).

```
void make_set(intv) {
	parent[v] = v;
}
```

```
int find_set(int v) {
	if (v == parent[v])
		return v;
	return find_set(parent[v]);
}

void union_sets(int a, int b) {
	a = find_set(a);
	b = find_set(b);
	if (a != b)
		parent[b] = a;
}
```

Впрочем, такая реализация системы непересекающихся множеств весьма неэффективна. Легко построить пример, когда после нескольких объединений множеств получится ситуация, что множество — это дерево, выродившееся в длинную цепочку. В результате каждый вызов find_set() будет работать на таком тесте за время порядка глубины дерева, т.е. за O(n).

Это весьма далеко от той асимптотики, которую мы собирались получить (константное время работы). Поэтому рассмотрим две оптимизации, которые позволят (даже применённые по отдельности) значительно ускорить работу.

Эвристика сжатия пути

Эта эвристика предназначена для ускорения работы find_set().

Она заключается в том, что когда после вызова find_set(v) мы найдём искомого лидера p множества, то запомним, что у вершины v и всех пройденных по пути вершин — именно этот лидер p. Проще всего это сделать, перенаправив их parent[] на эту вершину p.

Таким образом, у массива предков parent[] смысл несколько меняется: теперь это сжатый массив предков, т.е. для каждой вершины там может храниться не непосредственный предок, а предок предка, предок предка предка, и т.д.

С другой стороны, понятно, что нельзя сделать, чтобы эти указатели parent всегда указывали на лидера: иначе при выполнении операции union_sets() пришлось бы обновлять лидеров у O(n) элементов.

Таким образом, к массиву parent[] следует подходить именно как к массиву предков, возможно, частично сжатому.

Новая реализация операции find_set() выглядит следующим образом:

```
int find_set(int v) {
	if (v == parent[v])
		return v;
	return parent[v] = find_set(parent[v]);
}
```

Такая простая реализация делает всё, что задумывалось: сначала путём рекурсивных вызовов находится лидера множества, а затем, в процессе раскрутки стека, этот лидер присваивается ссылкам parent для всех пройденных элементов.

Реализовать эту операцию можно и не рекурсивно, но тогда придётся осуществлять два прохода по дереву: первый найдёт искомого лидера, второй — проставит его всем вершинам пути. Впрочем, на практике не рекурсивная реализация не даёт существенного выигрыша.

Оценка асимптотики при применении эвристики сжатия пути

Покажем, что применение одной эвристики сжатия пути позволяет достичь логарифмическую асимптотику: $O(\log n)$ на один запрос в среднем.

Заметим, что, поскольку операция union_sets() представляет из себя два вызова операции find_set() и ещё O(1) операций, то мы можем сосредоточиться в доказательстве только на оценку времени работы O(m) операций find_set().

Назовём весом w[v] вершины v число потомков этой вершины (включая её саму). Веса вершин, очевидно, могут только увеличиваться в процессе работы алгоритма.

Назовём размахом ребра (a, b) разность весов концов этого ребра: |w[a]-w[b]| (очевидно, у вершины-предка вес всегда больше, чем у вершины-потомка). Можно заметить, что размах какого-либо фиксированного ребра (a,b) может только увеличиваться в процессе работы алгоритма.

Кроме того, разобьём рёбра на классы: будем говорить, что ребро имеет класс k, если его размах принадлежит отрезку$[2^k; 2^{k+1} - 1]$. Таким образом, класс ребра — это число от 0 до |logn|.

Зафиксируем теперь произвольную вершину x и будем следить, как меняется ребро в её предка: сначала оно отсутствует (пока вершина x является лидером), затем проводится ребро из x в какую-то вершину (когда множество с вершиной x присоединяется к другому множеству), и затем может меняться при сжатии путей в процессе вызовов find_path. Понятно, что нас интересует асимптотика только последнего случая (при сжатии путей): все остальные случаи требуют O(1) времени на один запрос.

Рассмотрим работу некоторого вызова операции find_set: он проходит в дереве вдоль некоторого пути, стирая все рёбра этого пути и перенаправляя их в лидера.

Рассмотрим этот путь и исключим из рассмотрения последнее ребро каждого класса (т.е. не более чем по одному ребру из класса0,1,…$\lceil \log n \rceil$). Тем самым мы исключили $O(\log n)$ рёбер из каждого запроса.

Рассмотрим теперь все остальные рёбра этого пути. Для каждого такого ребра, если оно имеет класс k, получается, что в этом пути есть ещё одно ребро класса k (иначе мы были бы обязаны исключить текущее ребро, как единственного представителя класса k). Таким образом, после сжатия пути это ребро заменится на ребро класса как минимум k+1. Учитывая, что уменьшаться вес ребра не может, мы получаем, что для каждой вершины, затронутой запросом find_path, ребро в её предка либо было исключено, либо строго увеличило свой класс.

Отсюда мы окончательно получаем асимптотику работы m запросов: $O((n+m)\log n)$, что (при $m \geq n$) означает логарифмическое время работы на один запрос в среднем.

Эвристика объединения по рангу

Рассмотрим здесь другую эвристику, которая сама по себе способна ускорить время работы алгоритма, а в сочетании с эвристикой сжатия путей и вовсе способна достигнуть практически константного времени работы на один запрос в среднем.

Эта эвристика заключается в небольшом изменении работы union_sets: если в наивной реализации то, какое дерево будет присоединено к какому, определяется случайно, то теперь мы будем это делать на основе рангов.

Есть два варианта ранговой эвристики: в одном варианте рангом дерева называется количество вершин в нём, в другом — глубина

дерева (точнее, верхняя граница на глубину дерева, поскольку при совместном применении эвристики сжатия путей реальная глубина дерева может уменьшаться).

В обоих вариантах суть эвристики одна и та же: при выполнении union_sets() будем присоединять дерево с меньшим рангом к дереву с большим рангом.

Приведём реализацию ранговой эвристики на основе размеров деревьев:

```
void make_set(intv) {
	parent[v] = v;
	size[v] = 1;
}

void union_sets(int a, int b) {
	a = find_set(a);
	b = find_set(b);
	if (a != b) {
		if (size[a] < size[b])
			swap(a, b);
		parent[b] = a;
		size[a] += size[b];
	}
}
```

Приведём реализацию ранговой эвристики на основе глубины деревьев:

```
void make_set(intv) {
	parent[v] = v;
	rank[v] = 0;
}

void union_sets(int a, int b) {
	a = find_set(a);
	b = find_set(b);
	if (a != b) {
		if (rank[a] < rank[b])
			swap(a, b);
		parent[b] = a;
		if (rank[a] == rank[b])
```

```
                    ++rank[a];
        }
}
```

Оба варианта ранговой эвристики являются эквивалентными с точки зрения асимптотики, поэтому на практике можно применять любую из них.

Оценка асимптотики при применении ранговой эвристики

Покажем, что асимптотика работы системы непересекающихся множеств при использовании только ранговой эвристики, без эвристики сжатия путей, будет логарифмической на один запрос в среднем: $O(\log n)$.

Здесь мы покажем, что при любом из двух вариантов ранговой эвристики глубина каждого дерева будет величиной $O(\log n)$., что автоматически будет означать логарифмическую асимптотику для запроса find_set, и, следовательно, запроса union_sets.

Рассмотрим ранговую эвристику по глубине дерева. Покажем, что если ранг дерева равен k, то это дерево содержит как минимум 2^k вершин (отсюда будет автоматически следовать, что ранг, а, значит, и глубина дерева, есть величина $O(\log n)$.). Доказывать будем по индукции: для k=0 это очевидно. При сжатии путей глубина может только уменьшиться. Ранг дерева увеличивается с k-1 до k, когда к нему присоединяется дерево ранга k-1; применяя к этим двум деревьям размера k-1 предположение индукции, получаем, что новое дерево ранга k действительно будет иметь как минимум 2^k вершин, что и требовалось доказать.

Рассмотрим теперь ранговую эвристику по размерам деревьев. Покажем, что если размер дерева равен k, то его высота не более $\lfloor \log k \rfloor$. Доказывать будем по индукции: для k=1 утверждение верно. При сжатии путей глубина может только уменьшиться, поэтому сжатие путей ничего не нарушает. Пусть теперь объединяются два дерева размеров k_1 и k_2; тогда по предположению индукции их высоты меньше либо равны, соответственно, $\lfloor \log k_1 \rfloor$ и $\lfloor \log k_2 \rfloor$. Не теряя общности, считаем, что первое дерево — большее ($k1 \leq k2$), поэтому после объединения глубина получившегося дерева из k_1+k_2вершин станет равна:

$$h = \max(\lfloor \log k_1 \rfloor, 1 + \lfloor \log k_2 \rfloor).$$

Чтобы завершить доказательство, надо показать, что:

?

$$h \le \lfloor \log(k_1 + k_2) \rfloor,$$

?

$$2^h = \max(2^{\lfloor \log k_1 \rfloor}, 2^{\lfloor \log k_2 \rfloor}) \le 2^{\lfloor \log(k_1 + k_2) \rfloor},$$

что есть почти очевидное неравенство, поскольку $k_1 \le k_1 + k_2$ и$2k_2 \le k_1 + k_2$.

Объединение эвристик: сжатие пути плюс ранговая эвристика

Как уже упоминалось выше, совместное применение этих эвристик даёт особенно наилучший результат, в итоге достигая практически константного времени работы.

Мы не будем приводить здесь доказательства асимптотики, поскольку оно весьма объёмно (см., например, Кормен, Лейзерсон, Ривест, Штайн "Алгоритмы. Построение и анализ"). Впервые это доказательство было проведено Тарьяном (1975 г.).

Окончательный результат таков: при совместном применении эвристик сжатия пути и объединения по рангу время работы на один запрос получается $O(\alpha(n))$ в среднем, где $\alpha(n)$ — обратная функция Аккермана, которая растёт очень медленно, настолько медленно, что для всех разумных ограничений n она не превосходит 4 (примерно для $n \le 10^{600}$).

Именно поэтому про асимптотику работы системы непересекающихся множеств уместно говорить "почти константное время работы".

Приведём здесь итоговую реализацию системы непересекающихся множеств, реализующую обе указанные эвристики (используется ранговая эвристика относительно глубин деревьев):

```
void make_set(intv) {
	parent[v] = v;
	rank[v] = 0;
}

int find_set(int v) {
	if (v == parent[v])
		return v;
	return parent[v] = find_set(parent[v]);
}
```

```
void union_sets(int a, int b) {
	a = find_set(a);
	b = find_set(b);
	if (a != b) {
		if (rank[a] < rank[b])
			swap(a, b);
		parent[b] = a;
		if (rank[a] == rank[b])
			++rank[a];
	}
}
```

5.22 Дерево отрезков

Дерево отрезков — это структура данных, которая позволяет эффективно (т.е. за асимптотику $O(\log n)$) реализовать операции следующего вида: нахождение суммы/минимума элементов массива в заданном отрезке (a[l…r], где l и r поступают на вход алгоритма), при этом дополнительно возможно изменение элементов массива: как изменение значения одного элемента, так и изменение элементов на целом подотрезке массива (т.е. разрешается присвоить всем элементам a[l…r] какое-либо значение, либо прибавить ко всем элементам массива какое-либо число).
Вообще, дерево отрезков — очень гибкая структура, и число задач, решаемых ей, теоретически неограниченно. Помимо приведённых выше видов операций с деревьями отрезков, также возможны и гораздо более сложные операции (см. раздел "Усложнённые версии дерева отрезков"). В частности, дерево отрезков легко обобщается на большие размерности: например, для решения задачи о поиске суммы/минимума в некотором подпрямоугольнике данной матрицы (правда, уже только за время $O(log^2 k)$).

Важной особенностью деревьев отрезков является то, что они потребляют линейный объём памяти: стандартному дереву отрезков требуется порядка 4n элементов памяти для работы над массивом размера n.

Описание дерева отрезков в базовом варианте

Для начала рассмотрим простейший случай дерева отрезков — дерево отрезков для сумм. Если ставить задачу формально, то у

нас есть массив a[0…n-1], и наше дерево отрезков должно уметь находить сумму элементов с l-го по r-ый (это запрос суммы), а также обрабатывать изменение значения одного указанного элемента массива, т.е. фактически реагировать на присвоение a[i]=x (это запрос модификации). Ещё раз повторимся, дерево отрезков должно обрабатывать оба этих запроса за время O(logn).

Структура дерева отрезков

Итак, что же представляет из себя дерево отрезков?

Подсчитаем и запомним где-нибудь сумму элементов всего массива, т.е. отрезка a[0…n-1]. Также посчитаем сумму на двух половинках этого массива: a[0…n/2] и a[n/2+1 … n-1]. Каждую из этих двух половинок в свою очередь разобьём пополам, посчитаем и сохраним сумму на них, потом снова разобьём пополам, и так далее, пока текущий отрезок не достигнет длины 1. Иными словами, мы стартуем с отрезка [0, n-1] и каждый раз делим текущий отрезок надвое (если он ещё не стал отрезком единичной длины), вызывая затем эту же процедуру от обеих половинок; для каждого такого отрезка мы храним сумму чисел на нём.

Можно говорить, что эти отрезки, на которых мы считали сумму, образуют дерево: корень этого дерева — отрезок [0, n-1], а каждая вершина имеет ровно двух сыновей (кроме вершин-листьев, у которых отрезок имеет длину 1). Отсюда и происходит название — "дерево отрезков" (хотя при реализации обычно никакого дерева явно не строится, но об этом ниже в разделе реализации).

Итак, мы описали структуру дерева отрезков. Сразу заметим, что оно имеет линейный размер, а именно, содержит менее 2n вершин. Понять это можно следующим образом: первый уровень дерева отрезков содержит одну вершину (отрезок [0…n-1]), второй уровень — в худшем случае две вершины, на третьем уровне в худшем случае будет четыре вершины, и так далее, пока число вершин не достигнет n. Таким образом, число вершин в худшем случае оценивается суммой n+n/2+n/4+n/8+…+1<2n.

Стоит отметить, что при n, отличных от степеней двойки, не все уровни дерева отрезков будут полностью заполнены. Например, при n=3 левый сын корня есть отрезок [0…1], имеющий двух потомков, в то время как правый сын корня — отрезок [2…2],

являющийся листом. Никаких особых сложностей при реализации это не составляет, но тем не менее это надо иметь в виду.

Высота дерева отрезков есть величина O(logn) — например, потому что длина отрезка в корне дерева равна n, а при переходе на один уровень вниз длина отрезков уменьшается примерно вдвое.

Построение

Процесс построения дерева отрезков по заданному массиву a можно делать эффективно следующим образом, снизу вверх: сначала запишем значения элементов a[i] в соответствующие листья дерева, затем на основе них посчитаем значения для вершин предыдущего уровня как сумму значений в двух листьях, затем аналогичным образом посчитаем значения для ещё одного уровня, и т.д. Удобно описывать эту операцию рекурсивно: мы запускаем процедуру построения от корня дерева отрезков, а сама процедура построения, если её вызвали не от листа, вызывает себя от каждого из двух сыновей и суммирует вычисленные значения, а если её вызвали от листа — то просто записывает в себя значение этого элемента массива.

Асимптотика построения дерева отрезков составит, таким образом, O(n).

Запрос суммы

Рассмотрим теперь запрос суммы. На вход поступают два числа l и r, и мы должны за время O(logn) посчитать сумму чисел на отрезке a[l…r].

Для этого мы будем спускаться по построенному дереву отрезков, используя для подсчёта ответа посчитанные ранее суммы на каждой вершине дерева. Изначально мы встаём в корень дерева отрезков. Посмотрим, в какие из двух его сыновей попадает отрезок запроса [l…r] (напомним, что сыновья корня дерева отрезков — это отрезки [0…n/2] и [n/2+1…n-1]). Возможны два варианта: что отрезок [l…r] попадает только в одного сына корня, и что, наоборот, отрезок пересекается с обоими сыновьями.

Первый случай прост: просто перейдём в того сына, в котором лежит наш отрезок-запрос, и применим описываемый здесь алгоритм к текущей вершине.

Во втором же случае нам не остаётся других вариантов, кроме как перейти сначала в левого сына и посчитать ответ на запрос в нём, а затем — перейти в правого сына, посчитать в нём

ответ и прибавить к нашему ответу. Иными словами, если левый сын представлял отрезок $[l_1 \ldots r_1]$, а правый — отрезок $[l_2 \ldots r_2]$ (заметим, что $l_2=r_1+1$), то мы перейдём в левого сына с запросом $[l_1 \ldots r_1]$, а в правого — с запросом $[l_2, r_2]$.

Итак, обработка запроса суммы представляет собой рекурсивную функцию, которая всякий раз вызывает себя либо от левого сына, либо от правого (не изменяя границы запроса в обоих случаях), либо от обоих сразу (при этом деля наш запрос на два соответствующих подзапроса). Однако рекурсивные вызовы будем делать не всегда: если текущий запрос совпал с границами отрезка в текущей вершине дерева отрезков, то в качестве ответа будем возвращать предвычисленное значение суммы на этом отрезке, записанное в дереве отрезков.

Иными словами, вычисление запроса представляет собой спуск по дереву отрезков, который распространяется по всем нужным ветвям дерева, и для быстрой работы использующий уже посчитанные суммы по каждому отрезку в дереве отрезков.

Почему же асимптотика этого алгоритма будет $O(\log n)$? Для этого посмотрим на каждом уровне дерева отрезков, сколько максимум отрезков могла посетить наша рекурсивная функция при обработке какого-либо запроса. Утверждается, что таких отрезков не могло быть более четырёх; тогда, учитывая оценку $O(\log n)$ для высоты дерева, мы и получаем нужную асимптотику времени работы алгоритма.

Покажем, что эта оценка о четырёх отрезках верна. В самом деле, на нулевом уровне дерева запросом затрагивается единственная вершина — корень дерева. Дальше на первом уровне рекурсивный вызов в худшем случае разбивается на два рекурсивных вызова, но важно здесь то, что запросы в этих двух вызовах будут соседствовать, т.е. число l’’запроса во втором рекурсивном вызове будет на единицу больше числа r’ запроса в первом рекурсивном вызове. Отсюда следует, что на следующем уровне каждый из этих двух вызовов мог породить ещё по два рекурсивных вызова, но в таком случае половина этих запросов отработает нерекурсивно, взяв нужное значение из вершины дерева отрезков. Таким образом, всякий раз у нас будет не более двух реально работающих ветвей рекурсии (можно сказать, что одна ветвь приближается к левой границе запроса, а вторая ветвь — к

правой), а всего число затронутых отрезков не могло превысить высоты дерева отрезков, умноженной на четыре, т.е. оно есть число O(logn).

В завершение можно привести и такое понимание работы запроса суммы: входной отрезок [l…r] разбивается на несколько подотрезков, ответ на каждом из которых уже подсчитан и сохранён в дереве. Если делать это разбиение правильным образом, то благодаря структуре дерева отрезков число необходимых подотрезков всегда будет O(logn), что и даёт эффективность работы дерева отрезков.

Запрос обновления

Напомним, что запрос обновления получает на вход индекс i и значение x, и перестраивает дерево отрезков таким образом, чтобы оно соответствовало новому значению a[i]=x. Этот запрос должен также выполняться за время O(logn).

Это более простой запрос по сравнению с запросом подсчёта суммы. Дело в том, что элемент a[i] участвует только в относительно небольшом числе вершин дерева отрезков: а именно, в O(logn) вершинах — по одной с каждого уровня.

Тогда понятно, что запрос обновления можно реализовать как рекурсивную функцию: ей передаётся текущая вершина дерева отрезков, и эта функция выполняет рекурсивный вызов от одного из двух своих сыновей (от того, который содержит позицию i в своём отрезке), а после этого — пересчитывает значение суммы в текущей вершине точно таким же образом, как мы это делали при построении дерева отрезков (т.е. как сумма значений по обоим сыновьям текущей вершины).

Реализация

Основной реализационный момент — это то, как хранить дерево отрезков в памяти. В целях простоты мы не будем хранить дерево в явном виде, а воспользуемся таким трюком: скажем, что корень дерева имеет номер 1, его сыновья — номера 2 и 3, их сыновья — номера с 4 по 7, и так далее. Легко понять корректность следующей формулы: если вершина имеет номер i, то пусть её левый сын — это вершина с номером 2i, а правый — с номером 2i+1.

Такой приём значительно упрощает программирование дерева отрезков, — теперь нам не нужно хранить в памяти

структуру дерева отрезков, а только лишь завести какой-либо массив для сумм на каждом отрезке дерева отрезков.
Стоит только отметить, что размер этого массива при такой нумерации надо ставить не 2n, а 4n. Дело в том, что такая нумерация не идеально работает в случае, когда n не является степенью двойки — тогда появляются пропущенные номера, которым не соответствуют никакие вершины дерева (фактически, нумерация ведёт себя подобно тому, как если бы n округлили бы вверх до ближайшей степени двойки). Это не создаёт никаких сложностей при реализации, однако приводит к тому, что размер массива надо увеличивать до 4n.

Итак, дерево отрезков мы храним просто в виде массива t[], размера вчетверо больше размера n входных данных:

```
int n, t[4*MAXN];
```

Процедура построения дерева отрезков по заданному массиву a[] выглядит следующим образом: это рекурсивная функция, ей передаётся сам массив a[], номер v текущей вершины дерева, и границы tl и tr отрезка, соответствующего текущей вершине дерева. Из основной программы вызывать эту функцию следует с параметрами v=1,tl=0, tr=n-1.

```
void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = a[tl];
	else {
		int tm = (tl + tr) / 2;
		build (a, v*2, tl, tm);
		build (a, v*2+1, tm+1, tr);
		t[v] = t[v*2] + t[v*2+1];
	}
}
```

Далее, функция для запроса суммы представляет из себя также рекурсивную функцию, которой таким же образом передаётся информация о текущей вершине дерева (т.е. числа v, tl, tr, которым в основной программе следует передавать значения 1, 0, n-1 соответственно), а помимо этого — также границы l и r текущего запроса. В целях упрощения кода эта фукнция всегда делает по два рекурсивных вызова, даже если на самом деле нужен один — просто лишнему рекурсивному вызову

передастся запрос, у которого l>r, что легко отсекается дополнительной проверкой в самом начале функции.

```
int sum (int v, int tl, int tr, int l, int r) {
	if (l > r)
		return 0;
	if (l == tl && r == tr)
		return t[v];
	int tm = (tl + tr) / 2;
	return sum (v*2, tl, tm, l, min(r,tm))
		+ sum (v*2+1, tm+1, tr, max(l,tm+1), r);
}
```

Наконец, запрос модификации. Ему точно так же передаётся информация о текущей вершине дерева отрезков, а дополнительно указывается индекс меняющегося элемента, а также его новое значение.

```
void update (int v, int tl, int tr, int pos, int new_val) {
	if (tl == tr)
		t[v] = new_val;
	else {
		int tm = (tl + tr) / 2;
		if (pos <= tm)
			update (v*2, tl, tm, pos, new_val);
		else
			update (v*2+1, tm+1, tr, pos, new_val);
		t[v] = t[v*2] + t[v*2+1];
	}
}
```

Стоит отметить, что функцию update легко сделать нерекурсивной, поскольку рекурсия в ней хвостовая, т.е. разветвлений никогда не происходит: один вызов может породить только один рекурсивный вызов. При нерекурсивной реализации скорость работы может вырасти в несколько раз.

Из других оптимизаций стоит упомянуть, что умножения и деления на два стоит заменить битовыми операциями — это также немного улучшает производительность дерева отрезков.

5.23 Нахождение наибольшей нулевой подматрицы

Дана матрица a размером n*m. Требуется найти в ней такую подматрицу, состоящую только из нулей, и среди всех таких — имеющую наибольшую площадь (подматрица — это прямоугольная область матрицы).

Тривиальный алгоритм, — перебирающий искомую подматрицу, — даже при самой хорошей реализации будет работать $O(n^2m^2)$. Ниже описывается алгоритм, работающий за O(nm), т.е. за линейное относительно размеров матрицы время.

Алгоритм

Для устранения неоднозначностей сразу заметим, что n равно числу строк матрицы a, соответственно, m — это число столбцов. Элементы матрицы будем нумеровать в 0-индексации, т.е. в обозначении a[i][j] индексы i и j пробегают диапазоныi=0 …n-1, j=0…m -1.

Шаг 1: Вспомогательная динамика

Сначала посчитаем следующую вспомогательную динамику: d[i][j] — ближайшая сверху единица для элемента a[i][j]. Формально говоря, d[i][j] равно наибольшему номеру строки (среди строк диапазоне от -1 до i), в которой в j-ом столбце стоит единица. В частности, если такой строки нет, то d[i][j] полагается равным -1 (это можно понимать как то, что вся матрица как будто ограничена снаружи единицами).

Эту динамику легко считать двигаясь по матрице сверху вниз: пусть мы стоит в i-ой строке, и известно значение динамики для предыдущей строки. Тогда достаточно скопировать эти значения в динамику для текущей строки, изменив только те элементы, в которых в матрице стоят единицы. Понятно, что тогда даже не требуется хранить всю прямоугольную матрицу динамики, а достаточно только одного массива размера m:

```
vector<int> d (m, -1);
for (int i=0; i<n; ++i) {
	for (int j=0; j<m; ++j)
		if (a[i][j] == 1)
			d[j] = i;

	// вычислили d для i-ой строки, можем здесь использовать эти значения
}
```

Шаг 2: Решение задачи

Уже сейчас мы можем решить задачу за $O(nm^2)$ — просто перебирать в текущей строке номер левого и правого столбцов искомой подматрицы, и с помощью динамики d[][]вычислять за O(1) верхнюю границу нулевой подматрицы. Однако можно пойти дальше и значительно улучшить асимптотику решения.

Ясно, что искомая нулевая подматрица ограничена со всех четырёх сторон какими-то единичками (либо границами поля), — которые и мешают ей увеличиться в размерах и улучшить ответ. Поэтому, утверждается, мы не пропустим ответ, если будем действовать следующим образом: сначала переберём номер i нижней строки нулевой подматрицы, затем переберём, в каком столбце j мы будем упирать вверх нулевую подматрицу. Пользуясь значением d[i][j], мы сразу получаем номер верхней строки нулевой подматрицы. Осталось теперь определить оптимальные левую и правую границы нулевой подматрицы, — т.е. максимально раздвинуть эту подматрицу влево и вправо от j-го столбца.

Что значит раздвинуть максимально влево? Это значит найти такой индекс k_1, для которого будет$d[i][k_1]>d[i][j]$, и при этом k_1 — ближайший такой слева для индекса j. Понятно, что тогда k_1+1 даёт номер левого столбца искомой нулевой подматрицы. Если такого индекса вообще нет, то положить $k_1=-1$ (это означает, что мы смогли расширить текущую нулевую подматрицу влево до упора — до границы всей матрицы а).

Симметрично можно определить индекс k_2 для правой границы: это ближайший справа от j индекс такой, что$d[i][k_2]>d[i][j]$ (либо m, если такого индекса нет).

Итак, индексы k_1 и k_2, если мы научимся эффективно их искать, дадут нам всю необходимую информацию о текущей нулевой подматрице. В частности, её площадь будет равна $(i-d[i][j])*(k_2-k_1-1)$.

Как же искать эти индексы k_1 и k_2 эффективно при фиксированных i и j? Нас удовлетворит только асимптотика O(1), хотя бы в среднем.

Добиться такой асимптотики можно с помощью стека (stack) следующим образом. Научимся сначала искать индекс k_1, и сохранять его значение для каждого индекса j внутри текущей строки i в динамике $d_1[i][j]$. Для этого будем просматривать все

столбцы j слева направо, и заведём такой стек, в котором всегда будут лежать только те столбцы, в которых значение динамики d[][] строго больше d[i][j]. Понятно, что при переходе от столбца j к следующему столбцу j+1 требуется обновить содержимое этого стека. Утверждается, что требуется сначала положить в стек столбец j (поскольку для него стек "хороший"), а затем, пока на вершине стека лежит неподходящий элемент (т.е. у которого значение$d \leq d[i][j+1]$), — доставать этот элемент. Легко понять, что удалять из стека достаточно только из его вершины, и ни из каких других его мест (потому что стек будет содержать возрастающую по d последовательность столбцов).

Значениеd_1[i][j] для каждого j будет равно значению, лежащему в этот момент на вершине стека.

Ясно, что поскольку добавлений в стек на каждой строчке i происходит ровно m штук, то и удалений также не могло быть больше, поэтому в сумме асимптотика будет линейной.

Динамика d_2[][] для нахождения индексов k_2 считается аналогично, только надо просматривать столбцы справа налево.

Также следует отметить, что этот алгоритм потребляет O(m) памяти (не считая входные данные — матрицу a[][]).

Реализация

Эта реализация вышеописанного алгоритма считывает размеры матрицы, затем саму матрицу (как последовательность чисел, разделённых пробелами или переводами строк), и затем выводит ответ — размер наибольшей нулевой подматрицы.

Легко улучшить эту реализацию, чтобы она также выводила саму нулевую подматрицу: для этого надо при каждом изменении ans запоминать также номера строк и столбцов подматрицы (ими будут соответственно d[j]+1, i, d1[j]+1,d2[j]+1).

```
int n, m;
cin>> n >> m;
vector< vector<int>> a (n, vector<int> (m));
for (int i=0; i<n; ++i)
	for (int j=0; j<m; ++j)
		cin>> a[i][j];

int ans = 0;
```

```
vector<int> d (m, -1), d1 (m), d2 (m);
stack<int> st;
for (int i=0; i<n; ++i) {
	for (int j=0; j<m; ++j)
		if (a[i][j] == 1)
			d[j] = i;
	while (!st.empty()) st.pop();
	for (int j=0; j<m; ++j) {
		while (!st.empty() && d[st.top()] <= d[j])  st.pop();
		d1[j] = st.empty() ? -1 : st.top();
		st.push (j);
	}
	while (!st.empty()) st.pop();
	for (int j=m-1; j>=0; --j) {
		while (!st.empty() && d[st.top()] <= d[j])  st.pop();
		d2[j] = st.empty() ? m : st.top();
		st.push (j);
	}
	for (int j=0; j<m; ++j)
		ans = max (ans, (i - d[j]) * (d2[j] - d1[j] - 1));
}

cout << ans;
```

5.24 Числа Каталана

Числа Каталана — числовая последовательность, встречающаяся в удивительном числе комбинаторных задач.
Эта последовательность названа в честь бельгийского математика Каталана (Catalan), жившего в 19 веке, хотя на самом деле она была известна ещё Эйлеру (Euler), жившему за век до Каталана.
Последовательность
Первые несколько чисел Каталана C_n (начиная с нулевого):

1, 1, 2, 5, 14, 42, 132, 429, 1430, …

Числа Каталана встречаются в большом количестве задач комбинаторики. n-ое число Каталана — это:
Количество корректных скобочных последовательностей, состоящих из n открывающих и n закрывающих скобок.

Количество корневых бинарных деревьев с n+1 листьями (вершины не пронумерованы).
Количество способов полностью разделить скобками n+1 множитель.
Количество триангуляций выпуклого n+2-угольника (т.е. количество разбиений многоугольника непересекающимися диагоналями на треугольники).
Количество способов соединить 2n точек на окружности n непересекающимися хордами.
Количество неизоморфных полных бинарных деревьев с n внутренними вершинами (т.е. имеющими хотя бы одного сына).

Количество монотонных путей из точки (0,0) в точку (n,n) в квадратной решётке размером n*n, не поднимающихся над главной диагональю.
Количество перестановок длины n, которые можно отсортировать стеком (можно показать, что перестановка является сортируемой стеком тогда и только тогда, когда нет таких индексов $i<j<k$, что $a_k<a_i<a_j$).
Количество непрерывных разбиений множества из n элементов (т.е. разбиений на непрерывные блоки).
Количество способов покрыть лесенку 1…n с помощью n прямоугольников (имеется в виду фигура, состоящая из n столбцов, i-ый из которых имеет высоту i).
Вычисление
Имеется две формулы для чисел Каталана: рекуррентная и аналитическая. Поскольку мы считаем, что все приведённые выше задачи эквивалентны, то для доказательства формул мы будем выбирать ту задачу, с помощью которой это сделать проще всего.
Рекуррентная формула

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

Рекуррентную формулу легко вывести из задачи о правильных скобочных последовательностях.
Самой левой открывающей скобке 1 соответствует определённая закрывающая скобка r, которая разбивает формулу две части, каждая из которых в свою очередь является правильной скобочной последовательностью. Поэтому, если мы обозначим k=r-l-1, то для

любого фиксированного r будет ровно $C_k C_{n-1-k}$ способов. Суммируя это по всем допустимым k, мы и получаем рекуррентную зависимость на C_n.

Аналитическая формула

$$C_n = \frac{1}{n+1} C_{2\,n}^{n}$$

(здесь через C_n^k обозначен, как обычно, биномиальный коэффициент).

Эту формулу проще всего вывести из задачи о монотонных путях. Общее количество монотонных путей в решётке размером n*n равно C_{2n}^{n}. Теперь посчитаем количество монотонных путей, пересекающих диагональ. Рассмотрим какой-либо из таких путей, и найдём первое ребро, которое стоит выше диагонали. Отразим относительно диагонали весь путь, идущий после этого ребра. В результате получим монотонный путь в решётке (n+1)(n-1). Но, с другой стороны, любой монотонный путь в решётке (n+1)(n-1) обязательно пересекает диагональ, следовательно, он получен как раз таким способом из какого-либо (причём единственного) монотонного пути, пересекающего диагональ, в решётке n*n. Монотонных путей в решётке (n+1)(n-1) имеется $C_n = \frac{1}{n+1} C_{2\,n}^{n}$. В результате получаем формулу:

$$C_n = C_{2n}^{n} - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^{n}.$$

6. ЛАБОРАТОРНЫЕ РАБОТЫ

Шаблон оформления лабораторной работы и критерии её оценивания приведены в приложениях №5 и №6 соответственно.

Лабораторная работа №1
Реализация алгоритма Евклида

1. По описанию алгоритма из пункта 1параграфа 5 составить программу для нахождения наибольшего общего делителя двух натуральных чисел.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №2
Реализация алгоритма решето Эратосфена

1. По описанию алгоритма из пункта 2параграфа 5 составить программу для нахождения простых чисел.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №3
Реализация алгоритма по поиску чисел Фибоначчи.

1. По описанию алгоритма из пункта 3параграфа 5 составить программу для нахождения чисел Фибоначчи
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №4
Реализация алгоритма поиска в ширину

1. По описанию алгоритма из пункта 4параграфа 5 составить программу для обхода графа в ширину.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №5
Реализация алгоритма поиска в глубину

1. По описанию алгоритма из пункта 5параграфа 5 составить программу для обхода графа в глубину.

2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №6

Реализация алгоритма топологической сортировки

1. По описанию алгоритма из пункта 6параграфа 5 составить программу для нахождения топологической сортировки ориентированного ациклического графа.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №7

Реализация алгоритма поиска компонент связности

1. По описанию алгоритма из пункта 7параграфа 5 составить программу для нахождения компонент связности графа.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №8

Реализация алгоритма поиска мостов в графе

1. По описанию алгоритма из пункта 8параграфа 5 составить программу для нахождения мостов в графе.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №9

Реализация алгоритма Дейкстры

1. По описанию алгоритма из пункта 9параграфа 5 составить программу для нахождения кратчайших путей от заданной вершины до всех остальных.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №10

Реализация алгоритма Флойда-Уоршелла

1. По описанию алгоритма из пункта 10 параграфа 5 составить программу для нахождения кратчайших путей между всеми парами вершин в графе.

2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №11

Реализация алгоритма нахождения минимального остовного дерева

1. По описанию алгоритма из пункта 11параграфа 5 составить программу для нахождения минимального остовного дерева алгоритмом Крускала.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №12

Реализация алгоритма нахождения Эйлерова пути.

1. По описанию алгоритма из пункта 12параграфа 5 составить программу для нахождения Эйлерова пути в графе.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №13

Реализация алгоритма Куна.

1. По описанию алгоритма из пункта 13параграфа 5 составить программу для нахождения наибольшего паросочетания в двудольном графе.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №14

Реализация алгоритма нахождения знаковой площади треугольника

1. По описанию алгоритма из пункта 14параграфа 5 составить программу для нахождения знаковой площади треугольника.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №15

Реализация алгоритма нахождения пересечения двух отрезков

1. По описанию алгоритма из пункта 15 параграфа 5 составить программу для нахождения точки пересечения двух отрезков.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №16

Реализация алгоритма Грэхэма

1. По описанию алгоритма из пункта 16параграфа 5 составить программу для нахождения выпуклой оболочки множества точек.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №17

Реализация алгоритма Кнута-Морриса-Пратта

1. По описанию алгоритма из пункта 17параграфа 5 составить программу для нахождения функции КМП для произвольной строки.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №18

Реализация алгоритма хэширования

1. По описанию алгоритма из пункта 18параграфа 5 составить программу для хэширования строк.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №19

Реализация алгоритма Ахо-Корасик

1. По описанию алгоритма из пункта 19параграфа 5 составить программу для построения бора по набору строк.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №20

Реализация структуры данныхsqrt-декомпозиции

1. По описанию алгоритма из пункта 20параграфа 5 реализовать структуру sqrt-декомпозиции.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №21

Реализация структуры данныхDSU

1. По описанию алгоритма из пункта 21параграфа 5 составить программу для построения системы непересекающихся множеств.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №22

Реализация структуры данныхдерево отрезков

1. По описанию алгоритма из пункта 22параграфа 5 составить программу для построения дерева отрезков.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №23

Реализация алгоритма нахождения наибольшей нулевой подматрицы

1. По описанию алгоритма из пункта 23параграфа 5 составить программу для нахождения наибольшей нулевой подматрицы.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

Лабораторная работа №24

Реализация алгоритма поиска чисел Каталана

1. По описанию алгоритма из пункта 24параграфа 5 составить программу для нахождения чисел Каталана.
2. Для данной задачи подготовить набор входных данных.
3. Сделать отчёт о результатах проделанной работы.

ЗАКЛЮЧЕНИЕ

Итак, в пособии мы рассмотрели некоторые алгоритмы компьютерной обработки данных, структуры данных и задачи, которые ими решаются. Разобрали процесс создания комплектов задач для проведения олимпиад.

Подготовка к олимпиадам – это непрерывный процесс, и это относится не только к программированию. Для успешных выступлений нужно поддерживать спортивную форму, изо дня в день узнавать что-то новое.

Пособие написано для студентов Бурятского Государственного университета, готовящихся к выступлению на олимпиадах по программированию от внутривузовского до международного уровней.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Алексеев А.В., Беляев С.Н. Подготовка школьников к олимпиадам по информатике с использованием веб-сайта: учебно-методическое пособие для учащихся 7-11 классов. – Ханты-Мансийск: РИО ИРО, 2008. – 284 с.
2. Алексеев В.Е., Таланов В.А. Графы и алгоритмы. Структуры данных. Модели вычислений. – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. – 320 с. – (Серия «Основы информационных технологий»)
3. Андреева Е.В., Босова Л.Л., Фалина И.Н. Математические основы информатики. Элективный курс: Учебное пособие. – М.: БИНОМ. Лаборатория Знаний, 2007. – 312 с.
4. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы.
– М.: Издательский дом «Вильямс», 2000. – 384 с.
5. Босова Л.Л., Босова А.Ю., Коломенская Ю.Г. Занимательные задачи по информатике. – М.: БИНОМ. Лаборатория знаний. 2007. – 119 с.
6. Великович Л.С., Цветкова М.С. Программирование для начинающих. – М.: БИНОМ. Лаборатория знаний. 2007. – 287 с.
7. Вирт Н. Алгоритмы и структуры данных. Пер. с англ. М.: Мир, 1989. – 360 с.
8. Волчёнков С.Г., Корнилов П.А., Белов Ю.А. и др. Ярославские олимпиады по информатике. Сборник задач с решениями. – М.: БИНОМ. Лаборатория знаний. 2010. – 405 с.
9. Гасфилд Дэн. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И.В.Романовского. – СПб.: Невский Диалект; БХВ Петербург, 2003. – 654 с.
10. Долинский М.С. Алгоритмизация и программирование на Turbo Pascal: от простых до олимпиадных задач: Учебное пособие. – СПб.: Питер Принт, 2004. – 240 с.

11. Задачи по программированию /С.М. Окулов, Т.В. Ашихмина, Н.А. Бушмелева и др.; Под ред. С.М. Окулова. – М.: БИНОМ. Лаборатория знаний, 2006. – 820 с.
12. Златопольский Д. М. Программирование: типовые задачи, алгоритмы, методы. – М.: БИНОМ. Лаборатория знаний, 2007. – 223 с.
13. Иванов С.Ю., Кирюхин В.М., Окулов С. М. Методика анализа сложных задач по информатике: от простого к сложному // Информатика и образование. 2006. №10. С. 21 – 32.
14. Кирюхин В.М. Всероссийская олимпиада школьников по информатике. М.: АПК и ППРО, 2005. –212 с.
15. Кирюхин В.М. Информатика. Всероссийские олимпиады. Выпуск 1. – М.: Просвещение, 2008. – 220 с. – (Пять колец).
16. Кирюхин В.М. Информатика. Всероссийские олимпиады. Выпуск 2. – М.: Просвещение, 2009. – 222 с. – (Пять колец).
17. Кирюхин В.М. Информатика. Всероссийские олимпиады. Выпуск 3. – М.: Просвещение, 2011. – 222 с. – (Пять колец). (Планируется к выпуску в конце 2010 года).
18. Кирюхин В.М. Информатика. Международные олимпиады. Выпуск 1. – М.: Просвещение, 2009. – 239 с. – (Пять колец).
19. Кирюхин В.М., Лапунов А.В., Окулов С.М. Задачи по информатике. Международные олимпиады 1989-1996 гг. – М.: ABF, 1996. – 272 с.
20. Кирюхин В.М., Окулов С. М. Методика анализа сложных задач по информатике // Информатика и образование. 2006. №4. С. 42 – 54.
21. Кирюхин В.М., Окулов С. М. Методика анализа сложных задач по информатике // Информатика и образование. 2006. №5. С. 29 – 41.
22. Кирюхин В.М., Окулов С. М. Методика решения задач по информатике. Международные олимпиады. – М.: БИНОМ. Лаборатория знаний, 2007. – 600 с.

23. Кирюхин В.М., Цветкова М.С. Всероссийская олимпиада школьников по информатике в 2006 году. – М.: АПК и ППРО, 2006. – 152 с.
24. Кирюхин В.М., Цветкова М.С. Методическое обеспечение олимпиадной информатики в школе. Сборник трудов XVII конференции-выставки «Информационные технологии в образовании». Часть IV. – М.: «БИТ про», 2007.
25. Кнут Д. Искусство программирования для ЭВМ. Т. 1-3. – М., СПб., Киев: Вильямс, 2000.
26. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 1999. – 960с.
27. Кристофидес Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 432 с.
28. Меньшиков Ф.В. Олимпиадные задачи по программированию. – СПб.: Питер, 2006. – 315 с.
29. Московские олимпиады по информатике. 2002 – 2009. / Под ред. Е.В. Андреевой, В.М. Гуровица и В.А. Матюхина. – М.: МЦНМО, 2009. – 414 с.
30. Нижегородские городские олимпиады школьников по информатике. / Под ред. В.Д. Лелюха. – Нижний Новгород: ИПФ РАН, 2010. – 130 с.
31. Никулин Е.А. Компьютерная геометрия и алгоритмы машинной графики. – СПб.: БХВ-Петербург, 2003. – 560 с.
32. Окулов С.М. Основы программирования. – М.: БИНОМ. Лаборатория знаний, 2005. – 440 с.
33. Окулов С.М. Программирование в алгоритмах. – М.: БИНОМ. Лаборатория знаний. 2002. – 341 с.
34. Окулов С.М. Дискретная математика. Теория и практика решения задач по информатике: учебное пособие. – М.: БИНОМ. Лаборатория знаний. 2008. – 422 с.
35. Окулов С.М. Алгоритмы обработки строк: учебное пособие. – М.: БИНОМ. Лаборатория знаний, 2009. – 255 с.
36. Окулов С.М., Пестов А.А. 100 задач по информатике. – Киров: Изд-во ВГПУ, 2000. – 272 с.

37. Окулов С.М., Лялин А.В. Ханойские башни. – М.: БИНОМ. Лаборатория знаний. 2008. – 245 с. (Развитие интеллекта школьников).
38. Просветов Г.И. Дискретная математика: задачи и решения: учебное пособие. – М.: БИНОМ. Лаборатория знаний. 2008. – 222 с.
39. Пупышев В.В. 128 задач по началам программирования. – М.: БИНОМ. Лаборатория знаний. 2009. – 167 с.
40. Скиена С.С., Ревилла М.А. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. – М.: Кудиц-образ, 2005. – 416 с.
41. Столяр С.Е., Владыкин А.А.. Информатика. Представление данных и алгоритмы. – СПб.: Невский Диалект; М.: БИНОМ. Лаборатория знаний. 2007. –382 с.
42. Сулейманов Р.Р. Организация внеклассной работы в школьном клубе программистов: методическое пособие. – М.: БИНОМ. Лаборатория знаний. 2010. – 255 с.
43. Цветкова М.С. Система развивающего обучения как основа олимпиадного движения. Сборник трудов XVII конференции-выставки «Информационные технологии в образовании». Часть IV. – М.: «БИТ про», 2007.
44. Шестаков А.П. Задачи на длинную арифметику // Информатика и образование. 1999. № 8. С. 28-33.
45. Шень А. Программирование: теоремы и задачи. – М.:МЦНМО, 1995. – 264 с.

ПРИЛОЖЕНИЕ

Приложение №1. Примерное содержание файла statement.xml

```
<?xmlversion="1.0" encoding="utf-8" ?>
<problem
package="ru.codemore.contest"
	id="A"
	type="standard">
<statement language="ru_RU">
<title>...</title>
<description>
<p>...</p>
</description>
<input_format>
<p>...</p>
</input_format>
<output_format>
<p>...</p>
</output_format>
<notes>
<p>...</p>
</notes>
</statement>
<examples>
<example>
<input>...</input>
<output>...</output>
</example>
</examples>
</problem>
```

Приложение №2. Пример файла statement.xml конкретной задачи

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<problem
	package="ru.pskovedu.contest"
	id="rot13"
	type="standard">
<statement language="ru_RU">
<title>Шифр ROT13</title>
<description>
<p>
		Алгоритм шифрования ROT13 является вариацией шифра Цезаря.
</p>
<blockquote>
		Применение алгоритма ROT13 к части текста требует простой замены каждого
		буквенного символа на соответствующий ему со сдвигом на 13 позиций в алфавите.
		A становится N, B становится O, и т. д. до M, которое становится Z, а затем
		последовательно применяются буквы из начала алфавита: N становится A,
		O становится B, и так далее до Z, которая становится M. Затронуты лишь те буквы,
		которые используются в английском алфавите; цифры, символы, пробелы и все остальные
		символы остаются без изменений.
</blockquote>
<a
href="https://ru.wikipedia.org/wiki/ROT13">https://ru.wikipedia.org/wiki/ROT13</a>
</description>
<input_format>
<p>
		В первой строке дано целое число N (1&le;N&le;100) - длина строки.
</p>
<p>
		Во второй - строка состоящая из N символов - заглавных и строчных латинских букв,
```

```
             пробелов и знаков препинания ",.!?-" (не включая
кавычки), зашифрованная по алгоритму ROT13.
</p>
<p>
             Гарантируется, что строка не начинается и не
заканчивается пробелом.
</p>
</input_format>
<output_format>
<p>
             Расшифрованная строка.
</p>
</output_format>
</statement>
<examples>
<example>
<input>
13
Uryyb, jbeyq!
</input>
<output>Hello, world!</output>
</example>
</examples>
</problem>
```

Приложение №3. Пример решения задачи из приложения 2.

```
package rot13;
import java.util.Scanner;
public class Solution {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        in.nextLine();
        String s = in.nextLine();
        System.out.println(rot13(s));
    }
    public static String rot13(String s) {
```

```
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isAlphabetic(c)) {
                if (Character.isUpperCase(c)) {
                    c = (char)((c - 'A' + 13) % 26 + 'A');
                } else {
                    c = (char)((c - 'a' + 13) % 26 + 'a');
                }
            }
            sb.append(c);
        }
        return sb.toString();
}
}
```

Приложение №4. Пример чекера для задачи из приложения №2(используется стандартный fcmp.cpp).

```
#include "testlib.h"
#include <string>
#include <vector>
#include <sstream>

using namespace std;

int main(int argc, char * argv[])
{
    setName("compare files as sequence of lines");
    registerTestlibCmd(argc, argv);

    std::string strAnswer;

    int n = 0;
    while (!ans.eof())
    {
        std::string j = ans.readString();

        if (j == "" && ans.eof())
```

```
      break;

    strAnswer = j;
    std::string p = ouf.readString();

    n++;

    if (j != p)
      quitf(_wa, "%d%s lines differ - expected: '%s', found: '%s'", n,
englishEnding(n).c_str(), compress(j).c_str(), compress(p).c_str());
  }

  if (n == 1)
    quitf(_ok, "single line: '%s'", compress(strAnswer).c_str());

  quitf(_ok, "%d lines", n);
}
```

Приложение №5. Шаблон отчёта по лабораторной работе

Лабораторная работа № X

1. Поставленная задача, описание и метод решения.
2. Параметры входных данных и ожидаемый ответ на них.
3. Реализация алгоритма
4. Результаты запуска программы на наборах тестов
5. Вывод

Приложение №6. Критерии оценки лабораторной работы

Оценка 5 - правильно и без ошибок реализован требуемый алгоритм, на правильно сгенерированных тестах программа выдала ожидаемый правильный ответ, корректно и полно оформлен вывод;

Оценка 4 - правильно или с незначительными ошибками реализован алгоритм, на сгенерированных тестах получен правильный ответ, не в полной мере сделаны выводы;

Оценка 3 - программа написана с незначительными ошибками, ответы на тесты правильные, вывод не сделан или сделан не верно;

Оценка 2 - работа не выполнена, или сделана неправильно.

Учебное издание

Мальцев Станислав Петрович

ОЛИМПИАДНОЕ ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

Свидетельство о государственной аккредитации
№2670 от 11 августа 2017 г.

Формат 60х84 1/16.
Усл. печ. л. 7,9. Уч.-изд. л. 4,85. Заказ 136.

Издательство Бурятского госуниверситета
670000, г. Улан-Удэ, ул. Смолина, 24а
riobsu@gmail.com

Улан-Удэ
2019