**Омский государственный технический университет**

Кафедра ИВТ

Дисциплина

«**Проектирование и тестирование программного обеспечения**»

Методические указания к лабораторным работам

Омск, 2024

**Оглавление**

[Введение 4](#_Toc157343301)

[Лабораторная работа 0 9](#_Toc157343302)

[Лабораторная работа 1 10](#_Toc157343303)

[Часть 1. Разминка 10](#_Toc157343304)

[Задача 1.1 (Задача 3n + 1) 11](#_Toc157343305)

[Задача 1.2 (Сапёр) 12](#_Toc157343306)

[Задача 1.3 (LCD-дисплей) 13](#_Toc157343307)

[Задача 1.4 (Австралийское голосование) 14](#_Toc157343308)

[Часть 2. Структуры данных 15](#_Toc157343309)

[Задача 2.1 (Yahtzee) 15](#_Toc157343310)

[Задача 2.2 (Табло соревнований) 18](#_Toc157343311)

[Задача 2.3 (Числа Эрдеша) 20](#_Toc157343312)

[Задача 2.5 (Харталы) 24](#_Toc157343313)

[Задача 2.6 (Jolly Jumpers) 25](#_Toc157343314)

[Приложение 1.1 Пример 1 26](#_Toc157343315)

[Приложение 1.2 Общие структуры данных 31](#_Toc157343316)

[Лабораторная работа 2 37](#_Toc157343317)

[Часть 1. Строки 37](#_Toc157343318)

[Задача 3.1 (Дублеты) 41](#_Toc157343319)

[Задача 3.2 (Осколки файлов) 42](#_Toc157343320)

[Задача 3.3 (Дешифратор II) 43](#_Toc157343321)

[Задача 3.4 (Обычная перестановка) 44](#_Toc157343322)

[Задача 3.4 (Где Waldorf?) 45](#_Toc157343323)

[Задача 3.5 (WERTYU) 47](#_Toc157343324)

[Часть 2. Сортировка 48](#_Toc157343325)

[Задача 4.1 Сортировка Шелла 50](#_Toc157343326)

[Задача 4.2 (Такса) 52](#_Toc157343327)

[Задача 4.3 (Задача сапожника) 54](#_Toc157343328)

[Задача 4.4 (Аладушки) 55](#_Toc157343329)

[Задача 4.5 (Семья Вито) 57](#_Toc157343330)

[Приложение 2.1. Пример 58](#_Toc157343331)

[Приложение 2.2. Алгоритмы сортировки 62](#_Toc157343332)

[Приложение 2.3. Пример 2: рейтинг ухажеров 65](#_Toc157343333)

[Лабораторная работа 3 68](#_Toc157343334)

[Часть 1. Арифметика и алгебра 68](#_Toc157343335)

[Задача 5.1 (Попарно) 72](#_Toc157343336)

[Задача 5.2. (Игра в умножение) 73](#_Toc157343337)

[Задача 5.3 (Единицы) 74](#_Toc157343338)

[Задача 5.4 (Дилемма археолога) 75](#_Toc157343339)

[Задача 5.5 (Изменение порядка и сложение) 76](#_Toc157343340)

[Задача 5.6 (Начала арифметики) 77](#_Toc157343341)

[Часть 2. Комбинаторика 78](#_Toc157343342)

[Задача 6.1 (сколько чисел?) 84](#_Toc157343343)

[Задача 6.2 (Сколько частей земли?) 85](#_Toc157343344)

[Задача 6.3 (Счет) 86](#_Toc157343345)

[Задача 6.4. (Правильные выражения) 87](#_Toc157343346)

[Задача 6.5 (Монах-математик) 88](#_Toc157343347)

[Задача 6.6 (Самоописывающая последовательность) 90](#_Toc157343348)

[Задача 6.7 (Шаги) 91](#_Toc157343349)

[Приложение 3.1. Пример 92](#_Toc157343350)

[Лабораторная работа 4 98](#_Toc157343351)

[Часть 1. Поиск с возвратом (backtrack) 98](#_Toc157343352)

[Задача 7.1 (Пятнашки) 100](#_Toc157343353)

[Задача 7.2 (Шеренга) 102](#_Toc157343354)

[Задача 7.3 (Станции ТО) 103](#_Toc157343355)

[Задача 7.4 (Перетягивание каната) 104](#_Toc157343356)

[Задача 7.5 (Больший квадрат) 105](#_Toc157343357)

[Часть 2. Динамическое программирование 106](#_Toc157343358)

[Задача 8.1 (Веса и меры) 115](#_Toc157343359)

[Задача 8.2 (Умнее ли больший) 116](#_Toc157343360)

[Задача 8.3 (Однонаправленная задача коммивояжера) 117](#_Toc157343361)

[Задача 8.4 (Распил) 119](#_Toc157343362)

[Задача 8.5 (Заполнение парома) 120](#_Toc157343363)

[Задача 8.6 (Палочка По) 122](#_Toc157343364)

[Задача 8.7 (Приключения в дороге) 123](#_Toc157343365)

[Приложение 4.1. Задача восьми ферзей 125](#_Toc157343366)

[Приложение 4.2. Дополнение к примеру 8.1 129](#_Toc157343367)

[Лабораторная работа 5 133](#_Toc157343368)

[Часть 1. Обходы графов 133](#_Toc157343369)

[Задача 9.1 (Экскурсовод) 141](#_Toc157343370)

[Задача 9.2 (Лабиринт из косых) 143](#_Toc157343371)

[Задача 9.3 (Лесенки редактирования) 145](#_Toc157343372)

[Задача 9.4 (Кубики) 146](#_Toc157343373)

[Приложение 5.1. Основы работы с графами 148](#_Toc157343374)

[Часть 2. Продвинутые алгоритмы 154](#_Toc157343375)

[Задача 10.1 Ожерелье 168](#_Toc157343376)

[Задача 10.2 (Пожарное депо) 169](#_Toc157343377)

[Задача 10.3 (Большой обед) 171](#_Toc157343378)

[Задача 10.4 (Постановщик задач) 172](#_Toc157343379)

<https://blog.csdn.net/muximuxi_kgsecond/article/details/8128830>

# Введение

Целью данного курса ЛР является не научить учащихся тому, как программировать, а научить тому, как программировать лучше. Предполагается, что учащиеся знакомы с такими фундаментальны­ми понятиями, как переменные, условные выражения (к примеру, if-then-else, case), а также с основами итерации (например, for-do, while-do, repeat-until), подпрограммами и функциями.

Для написания программ, решающих поставленные в ЛР задачи, допускается использовать **любой язык программирования**. Все задачи, оформление отчетов выполняются **индивидуально**. В каждой части каждой ЛР задачи представлены в нескольких вариантах. Выбор номера задачи (варианта) осуществляется **по номеру студента** в списке группы с циклическим отсчетом (если задач 10, а учащихся 30, то задачу № 10 будут решать 10-й, 20-й, 30-й учащийся).

Внимательно изучите структуру каждой главы, **в главах имеются теоретические сведения и примеры**. Нет смысла браться за задачу, если вы не разобрали пример соответствующей главы.

Предложенные к решению задачи имеют сравнительно высокий уровень сложности, но не за счет своего объема, поэтому при написании программ для их решения возрастает ценность красивого и вдумчивого кода – важно написать не просто работающий, а обдуманный код. Поэтому при выполнении ЛР рекомендуется **соблюдать базовые рекомендации по написанию кода**:

• начинайте с готового рабочего шаблона – допустим, программа «начало» - «конец», далее делайте по 1-2 изменения (ну за исключением добавления переменных) и тестируйте работу – не тестируйте программу после 10 правок чтобы не гадать какое из них теперь не работает;

• пользуйтесь комментированием кода – добавляйте дублирующие блоки кода, изменяйте их и стирайте без сожаления если код не работает – вы легко можете вернуться к предыдущему закомментированному варианту;

• добавляйте содержательный вывод в консоль в тех местах, по которым хотите проверить, заходит в них программа при выполнении или нет (правда, более профессионально - использовать точки прерывания);

• другой вариант начала работы – сначала писать комментарии, потом саму программу.Начинайте ваши программы и процедуры с нескольких предложений, которые объясняют то, что они должны делать. Это важно, потому что если вы не можете *с легкостью* написать эти комментарии, то, вероятнее всего, вы не понимаете, что должна делать программа.

• *Документируйте каждую переменную.* Напишите одну строку комментария для каждой переменной, чтобы вы знали, что она делает. И снова, если вы не можете четко это написать, то вы не понимаете, что она тут делает. Вы будете редактировать код по крайней мере несколько циклов отладки, и вы оцените это скромное вложение в ее читабельность;

• *Используйте процедуры и функции, чтобы избежать излишнего кода.* Сокращайте объем цельного куска программы (к примеру, функции main) примерно до одного экрана. Если код перерос этот объем, стоит задуматься о выносе части вычислений в отдельный программный объект, которому вы к тому же сможете дать осмысленное имя, реализовать перегрузку операторов и т.д.

• *Внимательно читайте задачу.* Внимательно читайте каждую строку формулировки задачи. Обратите особенное внимание на описание входных и выходных дан­ных, а также на примеры входных и выходных данных, но...

• *Не предполагайте.* Чтение и понимание спецификаций - это важная часть соревновательного (и реального) программирования. Спецификации часто оставляют неописанные ловушки, в которые можно попасть.

Из того, что определенные примеры показывают какое-либо удобное свойство, еще не следует, что все тестовые данные будут этим свойством обладать. Будьте внимательны на предмет неограниченных вход­ных чисел, больших длин строк, отрицательных чисел и т. п. Любые данные, которые явно не запрещены, должны считаться разрешенными!

• *Не так быстро.* Если мы не используем экспоненциальные алгоритмы в задаче, где хватило бы полиномиальных, то **эффективность часто не является важным вопросом**. Не беспокойтесь сильно насчет эффективности, если вы не столкнулись с проблемой или не можете ее предсказать. Узнайте максимальный размер входных данных из спецификации и решите, подходит ли наиболее прямолинейный алгоритм для таких входных данных.

*• Выбор структуры данных* определяется действиями, которые вам нужны. Любой язык программирования и его популярная библиотека предоставляет стандартные структуры данных: кроме массивов могут быть полезными строки, **очереди, записи, множества** и т.д. Полезно самостоятельно подробнее ознакомиться с понятиями структур данных в используемом вами языке программирования: **сигнальными метками, словарями, очередями по приоритету** и т.д. Даже использование простого переноса данных в **структуру** может сильно облегчить понимание движения данных в программе.

Очень много времени вам может сэкономить продуманное тестирование программы. Приведем ряд советов по тестированию :

- *Проверяйте заданные входные данные*. Большинство спецификаций задач включают пример входных и выходных данных. Правильная обработка тестовых данных является необходимым, но не достаточным условием.

- *Проверяйте некорректные входные данные*. Если в описании задачи говорится, что ваша программа должна предпринимать определенные действия при нелегальных входных данных, обязательно протестируйте такие проблематичные случаи.

- *Проверяйте граничные случаи*. Многие дефекты в программах связаны с ошибками завышения или занижения на единицу. Четко проверяйте свои программы на такие условия, как пустой ввод, один объект, два объекта и значения, равные нулю.

- *Проверяйте те случаи, для которых вы знаете ответ*. Важным условием разработки грамотного тестового случая является знание правильного ответа. Ваши тесты должны работать с достаточно простыми примерами, которые вы можете решить без помощи программы. Без полного анализа желаемого поведения программы легко быть обманутым нормально выглядящими выходными данными.

- *Проверяйте большие случаи*, где вы не знаете правильного ответа. Обычно вручную можно решить только небольшие примеры. Это делает сложным оценку работы программы для большего количества входных данных. Проверьте несколько легко конструируемых случаев входных данных, таких, как случайные данные или числа от 1 до n включительно, просто для того чтобы убедиться, что программа не завершается с ошибкой и не делает ничего глупого.

- *Отображайте ваши структуры данных*. Иногда непосредственную проверку можно проще всего провести отобразив содержимое очереди или стека, чтобы увидеть, чего же не хватает. Не ленитесь писать специальные процедуры вывода для всех нетривиальных структур данных.

- *Строго проверяйте инвариантность*. Любая функция является источником ошибок и еще больше их возникает, когда после применения двух функций вы можете получать данные, строго соответствующие исходящим. Если этого не произошло – есть повод поискать ошибку.

- *Точно локализуйте ошибку* прежде чем её исправлять. Иногда вы можете предположить, что источником ошибки является вовсе не та функция, которую нужно отлаживать на самом деле.

В заключение также отметим, что все представленные задачи являются реальными задачами, используемыми студентами при подготовке к соревнованиям по олимпиадному программированию. Каждая задача имеет номера для сетевой проверки решений по адресам http://www.programming-challenges.com (вместе с большим количеством сопутствующих материалов) и http://online-judge.uva.es. Данные ресурсы предъявляют строгие требования к оформлению входного кода (и некоторые ограничения к языкам программирования), однако если вас это не остановит – вполне возможно, что вы встанете на путь достижения реальных успехов в соревновательном программировании.

**Требования к отчетам по лабораторным работам**

Каждый отчет должен иметь титульный лист, основную часть и выводы. Тексты программ допускается приводить по тексту отчета в виде текста или скриншотов (в т.ч. совмещающих вывод и текст программы), либо выносить в приложение. Тексты программ должны также содержать комментарии, в т. ч. **обязательно ФИО и группу учащегося**. **Пример** титульного листа – приложение 0 в конце данных МУ.

Основная часть отчета представляет собой последовательное описание процесса решения задачи:

**- описание задачи;**

- Ваш вариант её решения в виде **UML-диаграммы**;

- не менее 5 заготовленных наборов тестовых данных, подразумевающих **проход разных ветвей и отработку граничных условий**. Набор должен автоматически читаться программой из текстового файла (при этом не обязательно за один запуск читать все заготовленные тесты).

**- скриншот** с результатами выполнения программы для каждого набора;

**- пояснения** к алгоритму или программе (выбранный путь решения задачи, используемые программные объекты);

**Обязательны** пояснения к скриншотам.

Входные данные в тексте программы либо приглашение к вводу, а также выходные данные программы **обязательно** должны сопровождаться поясняющей их текстовой строкой.

Рекомендуемая литература по UML:

Червенчук, И. В. Моделирование объектно ориентированных систем с помощью UML : учеб. пособие / И. В. Червенчук, А. С. Грицай; ОмГТУ. - Омск: Изд-во ОмГТУ, 2020. - 105 c.

# Лабораторная работа 0

Начало работы в семестре. Задачи:

- выбрать тему РГР

- сделать предварительную блок-схему алгоритма

- сформулировать печень основных программных функций

- сформулировать перечень полезной литературы и веб-ссылок

- расписать полученные результаты в отчете

# Лабораторная работа 1

«Линейные» программы и использование структур данных

Задания

1. В соответствии со своим вариантом выберите по одной задаче из части 1 и части 2. Ознакомьтесь с рекомендациями по написанию кода, изложенными во введении, ознакомьтесь с примером решения задачи и теорией (см. приложения).

2. Напишите программы, решающие выбранные вами задачи.

3. Оформите отчет по ЛР.

## Часть 1. Разминка

Поскольку предполагается, что вы уже знакомы с основными приемами программирования, попробуйте решить следующие задачи, не требующие специальной подготовки (для удобства каждая задача приведена на отдельном листе).

### Задача 1.1 (Задача 3n + 1)

PC/UVa IDs: 110101/100

Рассмотрим следующий алгоритм генерации последовательности чисел. Начнем с целого числа n. Если n четно, то поделим на 2. Если n нечетно, то умножим на 3 и добавим 1. Будем повторять этот процесс с новым полученным n, пока n не станет равным 1. Например, для n = 22 будет сгенерирована следующая последовательность чисел:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Полагают (но это еще не доказано), что этот алгоритм сведется к n = 1 для любого целого n. По крайней мере, это предположение верно для всех целых чисел до 1 000 000.

Для данного n *длиной цикла n* будем называть число сгенерированных чисел до *и включая* 1. В примере, приведенном выше, длина цикла 22 равна 16. Для двух заданных чисел i и j вы должны определить максимальную длину цикла для всех чисел между i и j, включая обе конечные точки.

**Входные данные**

Входные данные будут состоять из серии пар целых чисел i и j, одна пара чисел в строке. Все целые числа будут меньше 1 000 000 и больше 0.

**Выходные данные**

Для каждой пары чисел i и j выведите i, j в том порядке, в каком они были введены, и после этого выведите максимальную длину цикла для всех целых чисел между i и j включая сами i и j. Эти три числа должны быть разделены одним пробелом, все три числа в одной строке, и для каждой строки входных данных должна быть одна строка выходных данных.

Пример входных данных Соответствующие выходные данные

1 10 1 10 20

100 200 100 200 125

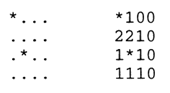
201 210 201 210 89

900 1000 900 1000 174

### Задача 1.2 (Сапёр)

РС/UVa IDs: 110102/10189

Как известно, цель игры состоит в том, чтобы найти расположение всех мин на поле размером MxN. Игра показывает вам число в клеточке, которое говорит вам, сколько мин находится в соседних с этой клеточках. Каждая клеточка имеет максимум восемь соседних. Поле 4x4 слева содержит две мины, каждая из которых представлена символом «\*». Если мы добавим к этому полю числа-подсказки по принципу, описанному выше, то получим поле, изображенное справа:



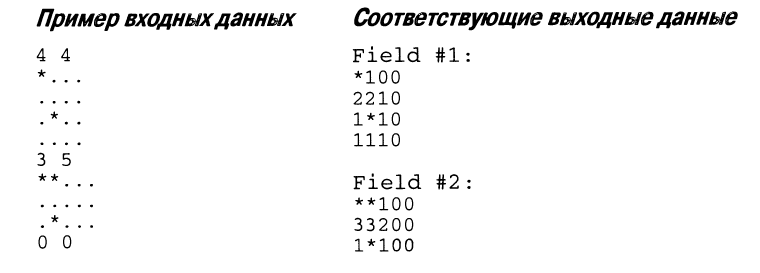
**Входные данные**

Входные данные будут состоять из произвольного числа полей. Первая строка каждого поля содержит два целых числа n и m (0 < n, m ≤ 100), которые соответствуют числу строк и столбцов поля соответственно. Каждая из последующих n строк содержит ровно m символов, представляя собой поле.

Безопасные клеточки обозначаются «.» , мины обозначаются «\*», и то и другое без кавычек. Первая строка поля, где n = m = 0, представляет собой окончание входных данных и обрабатываться не должна.

**Выходные данные**

Для каждого поля выведите сообщение Field #x:, единственное на строке, где х обозначает номер текущего поля, начиная с 1. Следующие n строк должны содержать поле с символами «.», замененными на число мин в соседних клеточках. Между выводами полей должна быть пустая строка.



### Задача 1.3 (LCD-дисплей)

РС/UVa IDs: 110104/706

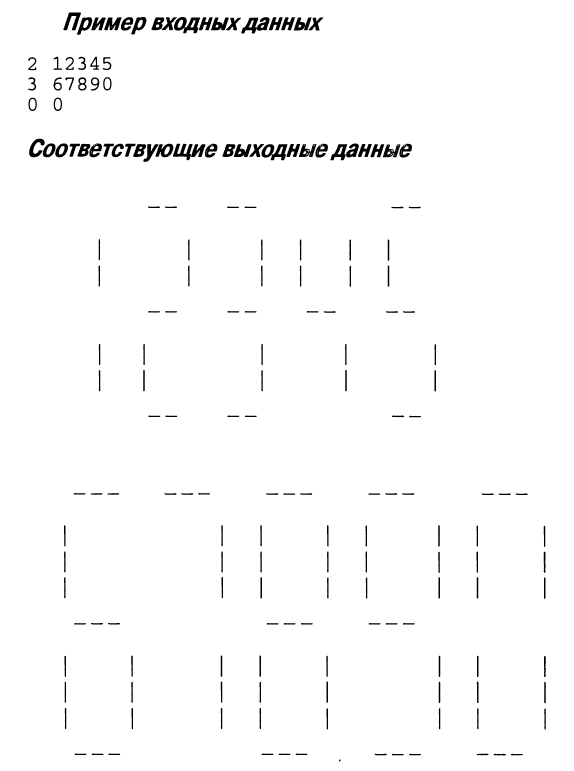
Ваш друг только недавно купил себе новый компьютер. До этого самой мощной машиной, которую он когда-либо использовал, был карманный калькулятор. Он немного расстроен, потому что LCD-дисплей его калькулятора ему правился больше, чем экран его компьютера. Чтобы обрадовать его, напишите программу, которая отображает числа в стиле LCD-дисплея.

**Входные данные**

Входные данные содержат несколько строк, по строке для каждого числа, которое нужно отобразить. Каждая строка содержит целые числа s и n, где n - это число, которое нужно отобразить (0 ≤ n ≤ 99 999 999) и s, это размер, в котором оно должно быть изображено (0 ≤ s ≤ 10). Входные данные заканчиваются строкой, состоящей из двух нулей, которую не нужно обрабатывать.

**Выходные данные**

Выведите числа указанные во входных данных в стиле LCD-дисплея, используя s «-» символов для горизонтальных сегментов и s «|» символов для вертикальных. Каждая цифра занимает ровно s + 2 колонок и 2 s + 3 строк. Не забудьте заполнить все белое пространство, занимаемое цифрами, пробелами, включая последнюю цифру. Между двумя цифрами должна быть точно одна колонка пробелов. Выводите пустую строку после каждого числа.



### Задача 1.4 (Австралийское голосование)

PC/UVa IDs: 110108/10142

Австралийские бюллетени требуют, чтобы избиратели расположили всех кандидатов в порядке предпочтения. Первоначально учитывается только первый кандидат из получившегося списка, и если один из кандидатов набрал более 50% голосов, то он считается избранным. Тем не менее, если ни один из кандидатов не набрал более 50% голосов, все кандидаты с наименьшим числом голосов выбывают. Бюллетени, засчитанные в пользу этих кандидатов, засчитываются в пользу не выбывшего кандидата, который следующим идет по порядку предпочтения. Этот процесс исключения самых слабых кандидатов и пересчет их бюллетеней в пользу следующего по порядку предпочтения, не выбывшего кандидата продолжается до тех пор, пока один из кандидатов не наберет более 50% голосов или пока у всех кандидатов не окажется одинаковое число голосов.

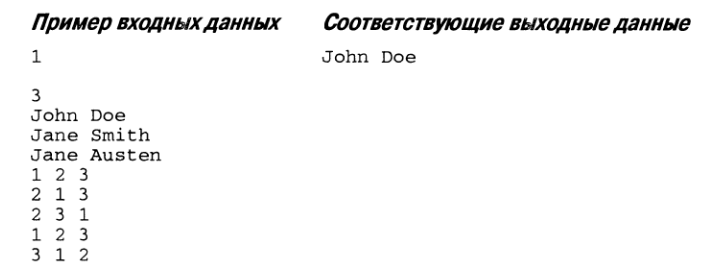
**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Первая строка каждого блока - это целое число n < 20, означающее число кандидатов. Следующие n строк содержат имена кандидатов, каждое до 80 символов длиной и содержащее любые печатаемые символы. Далее следуют до 1000 строк, каждая включает содержимое бюллетеня. Каждый бюллетень содержит числа от 1 до n в каком-либо порядке. Первое число обозначает наиболее предпочтительного кандидата; второе - второго по предпочтимости и т. д.

**Выходные данные**

Выходные данные для каждого тестового блока содержат одну строку, содержащую имя победителя, или несколько строк, содержащих имена всех кандидатов, которые набрали одинаковое число голосов. Выходные данные для каждых двух последовательных блоков должны разделяться пустой строкой.



## Часть 2. Структуры данных

Структуры данных - основа любой сложной программы. Выбор правильной структуры данных может очень сильно повлиять на сложность итоговой реализации. Выберете правильное представление данных, и вашу задачу будет легко запрограммировать. Выберете неверное представление данных, и вы можете потратить большое количество времени и кода, чтобы компенсировать ошибочность первоначального решения. Описания некоторых структур данных, которые могут вам помочь (или показать вам, как же можно было решить вашу задачу намного проще) приведены в приложении 2.

Пример к задаче части № 2 приведен в приложении 1.1.

### Задача 2.1 (Yahtzee)

РС/UVaIDs: 110208/10149

В игре Yahtzee используются пять игральных костей, которые бросаются в 13 раундах. Протокол результатов содержит 13 категорий. Счет за каждый раунд может быть записан в категорию по выбору игрока, но в каждую категорию счет можно записывать только один раз. Счет в эти 13 категорий записывается следующим образом:

• единицы - сумма всех выброшенных единиц;

• двойки - сумма всех выброшенных двоек;

• тройки - сумма всех выброшенных троек;

• четверки - сумма всех выброшенных четверок; пятерки - сумма всех выброшенных пятерок; шестерки - сумма всех выброшенных шестерок; шанс - сумма чисел на всех костях;

• три одинаковые - сумма чисел на всех костях, при условии, что как минимум три из них имеют одно и то же значение;

• четыре одинаковые - сумма чисел на всех костях, при условии, что как минимум четыре из них имеют одно и то же значение;

короткий стрит - 25 очков, при условии, что четыре кости образуют последовательность (то есть 1, 2, 3,4, или 2, 3,4, 5, или 3,4, 5, 6);

• короткий стрит - 35 очков, при условии, что все кости образуют последовательность (то есть 1, 2, 3,4, 5 или 2, 3,4, 5, 6);

• фулл-хаус - 40 очков, при условии, что на трех костях выпало одно и то же число и на оставшихся двух костях также выпало одно и то же число.

Каждая из последних шести категорий может быть засчитана за 0 очков, если необходимые условия не выполняются.

Счетом игры является сумма всех 13 категорий плюс дополнительные 35 очков, если сумма первых шести категорий 63 или больше.

Ваша задача - рассчитать максимальный счет для последовательности раундов.

**Входные данные**

Каждая строка входных данных содержит пять целых чисел от 1 до 6, представляющих собой пять костей, брошенных в каждом раунде. Для каждой игры 13 таких строк, и во входных данных может быть любое количество игр.

**Выходные данные**

Ваши выходные данные должны состоять из одной строки для каждой игры, содержащей 15 чисел: счет в каждой категории (в заданном порядке), дополнительные очки (0 или 35) и суммарный счет. Если существует несколько распределений, приводящих к одному и тому же максимальному счету, то подойдет любое.

**Пример входных данных**

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 1 1 1 1

6 6 6 6 6

6 6 6 1 1

1 1 1 2 2

1 1 1 2 3

1 2 3 4 5

1 2 3 4 6

6 1 2 6 6

1 4 5 5 5

5 5 5 5 6

4 4 4 5 6

3 13 6 3

2 2 2 4 6

**Соответствующие выходные данные**

1 2 3 4 5 0 15 0 0 0 25 35 0 0 90

3 6 9 12 15 30 21 20 26 50 25 35 40 35 327

### Задача 2.2 (Табло соревнований)

PC/UVaIDs: 110207/10258

Хотите посоревноваться в чемпионатах по программированию? Тогда вам нужно знать, как вести счет! Участники соревнований ранжируются сначала по числу решенных задач (чем больше, тем лучше), а потом по уменьшению величины штрафного времени. Если у двух и более участников совпадает и количество решенных задач, и величина штрафного времени, то они отображаются в порядке увеличения номеров команд.

Задача засчитывается решенной участником соревнований, если любая из попыток отправки этой задачи была признана верной. Штрафное время рассчитывается как число минут, прошедших до того, как было отправлено правильное решение, плюс 20 минут за каждую некорректную отправку, имевшую место до верного решения. За нерешенные задачи штрафное время не начисляется.

**Входные данные**

Входные данные начинаются со строки, содержащей одно целое положительное число, задающее число блоков, которые описаны ниже. За этой строкой следует пустая. Также пустая строка находится между двумя последовательными блоками.

Входные данные состоят из судейского списка очередности, содержащего вхождения по нескольким или по всем участникам соревнований с 1-го по 100-й, решающим задачи с 1-й по 9-ю. Каждая строка входных данных состоит из трех чисел и буквы в формате *участник-задача- время-L*, где L может быть С, I, R, U или Е. Это означает: Correct (верно), Incorrect (неверно), Clarification Request (уточняющий запрос), Unjudged (необработан) и Erroneous submission (отправка с ошибкой). Последние три случая не влияют на счет.

Строки входных данных идут в том порядке, в котором получались посылки.

**Выходные данные**

Выходные данные для каждого блока должны представлять собой табло, отсортированное по критериям, описанным выше. Каждая строка выходных данных должна содержать номер участника, количество задач, решенных участником, и полное штрафное время, полученное участником. Так как не все участники в действительности соревнуются, отображайте только тех участников, кто посылал решения.

Между выходными данными для двух последовательных блоков должна быть пустая строка.

**Пример входных данных**

1

1 2 10 I

3 1 11 С

1 2 19 R

1 2 21 С

1 1 25 С

**Соответствующие выходные данные**

1 2 66

3 1 11

### Задача 2.3 (Числа Эрдеша)

PC/UValDs: 110206/10044

Венгерский ученый Пауль Эрдеш (Paul Erdus, 1913-1966) был одним из самых известных математиков XX века. Любой математик, имевший честь быть соавтором Эрдеша, глубоко уважаем.

К сожалению, не у всех была возможность написать статью совместно с Эрдешем, лучшее, что они могли сделать, - это опубликовать статью с кем-либо, кто опубликовал научную статью в соавторстве с Эрдешем. Это дало начало так называемым числам Эрдеша. Автор, публиковавшийся совместно с Эрдешем, имеет число Эрдеша 1. Автор, не публиковавшийся с Эрдешем, но публиковавшийся совместно с кем-либо, кто имеет число Эрдеша 1, получал число Эрдеша 2 и т. д.

Ваша задача - написать программу, которая рассчитывает числа Эрдеша для данного множества статей и ученых.

**Входные данные**

Первая строка входных данных содержит число сценариев. Каждый сценарий состоит из базы данных статей и из списка имен. Он начинается со строки вида Р N, где Р и N- натуральные числа. За этой строкой следует база данных статей из Р строк, каждая из которых содержит описание одной статьи, выглядящее так:

Smith, M.N. , Martin, G. , Erdos, P. : Newtonian forms of prime factors

После Р статей идут N строк с именами. Такая строка с именем имеет следующий формат:

Martin, G.

**Выходные данные**

Для каждого сценария вы должны вывести строку, содержащую "Scenario /'" (где I - это номер сценария), и имена авторов вместе с их числами Эрдеша для всех авторов из списка имен. Авторы должны выводиться в том же порядке, в котором они были в списке имен. Число Эрдеша основано на статьях из базы данных статей этого сценария. Авторы, которые не имеют никакого отношения к Эрдешу судя по статьям данной базы данных, имеют число Эрдеша " infinity".

**Пример входных данных**

1

4 3

Smith, M.N., Martin, G., Erdos, P.: Newtonian forms of prime factors

Erdos, P., Reisig, W.: Stuttering in petri nets

Smith, M.N., Chen, X.: First order derivates in structured programming

Jablonski, Т., Hsueh, Z.: Selfstabilizing data structures

Smith, M.N.

Hsueh, Z.

Chen, X.

**Соответствующие выходные данные**

Scenario 1

Smith, M.N. 1

Hsueh, Z. infinity

Chen, X. 2

**Задача 2.4 (По порядку)**

PC/UVaIDs: 110205/10205

В Большом городе мною казино. В одном из них дилер жульничает. Она довела до совершенства несколько перетасовочных трюков; каждый трюк меняет порядок карт одним и тем же образом, когда бы он ни был использован. Простой пример - это трюк «нижней карты», при котором нижняя карта переносится на верх колоды. Используя различные комбинации известных ей трюков, нечестный дилер может расположить карты практически в любом желаемом порядке.

Вы были наняты менеджером службы безопасности, чтобы поймать этого дилера. Вам дали список всех трюков, используемых дилером, а также список визуальных подсказок, благодаря которым вы можете определить, какой трюк она использует в любой данный момент времени. Ваша задача состоит в том, чтобы предсказать порядок карт после последовательности трюков.

Стандартная игральная колода состоит из 52 карт, четыре масти по 13 значений каждая. Значения карт: 2, 3. 4, 5, б, 7, 8, 9, 10, Валет (Jack), Дама (Queen), Король (King), Туз (Асе). Названия мастей: Трефы (Clubs), Бубны (Diamonds), Червы (Hearts), Пики (Spades). Определенная карта в колоде может быть однозначно определена по ее значению и масти и обычно обозначается <значение> of <масть>. Например, «9 of Hearts» или «King of Spades». По традиции карты в новой колоде идут по мастям в алфавитном порядке, а потом по значению в порядке, описанном выше.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Каждый блок состоит из целого числа n < 100 - числа трюков, которые знает дилер. Далее следуют n наборов по 52 целых числа, каждый из которых содержит все числа от 1 до 52 в определенном порядке. Внутри каждого набора из 52 чисел, если i стоит j-м по счету, то это значит, что трюк перемещает i-ю карту в колоде на j-е место (не важно какая именно это была карта).

Далее следуют несколько строк, каждая из которых содержит целое число k от 1 до n. Оно означает, что вы увидели, как дилер делает k-й трюк, заданный во входных данных.

**Выходные данные**

Для каждого тестового блока считайте, что дилер начинает с новой колоды, карты в которой расположены в исходном порядке, описанном выше. Выведите новый порядок карт в колоде, после того как все трюки были проделаны. Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.

**Пример входных данных**

1

2

2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 3 5 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 52 51

52 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 3 0 3132 33 34 35 36 37 38 39 40 4142 43 44 45 46 47 48 49 50 51 1

1

2

**Соответствующие выходные данные**

King of Spades 2 of Clubs

4 of Clubs

5 of Clubs

6 of Clubs

7 of Clubs

8 of Clubs

9 of Clubs

10 of Clubs Jack of Clubs Queen of Clubs King of Clubs Ace of Clubs

2 of Diamonds

3 of Diamonds

4 of Diamonds

5 of Diamonds

6 of Diamonds

7 of Diamonds

8 of Diamonds

9 of Diamonds

10 of Diamonds

Jack of Diamonds Queen of Diamonds King of Diamonds Ace of Diamonds

2 of Hearts

3 of Hearts

4 of Hearts

5 of Hearts

6 of Hearts

7 of Hearts

8 of Hearts

9 of Hearts

10 of Hearts Jack of Hearts Queen of Hearts King of Hearts Ace of Hearts

2 of Spades

3 of Spades

4 of Spades

5 of Spades

6 of Spades

7 of Spades

8 of Spades

9 of Spades

10 of Spades Jack of Spades Queen of Spades Ace of Spades

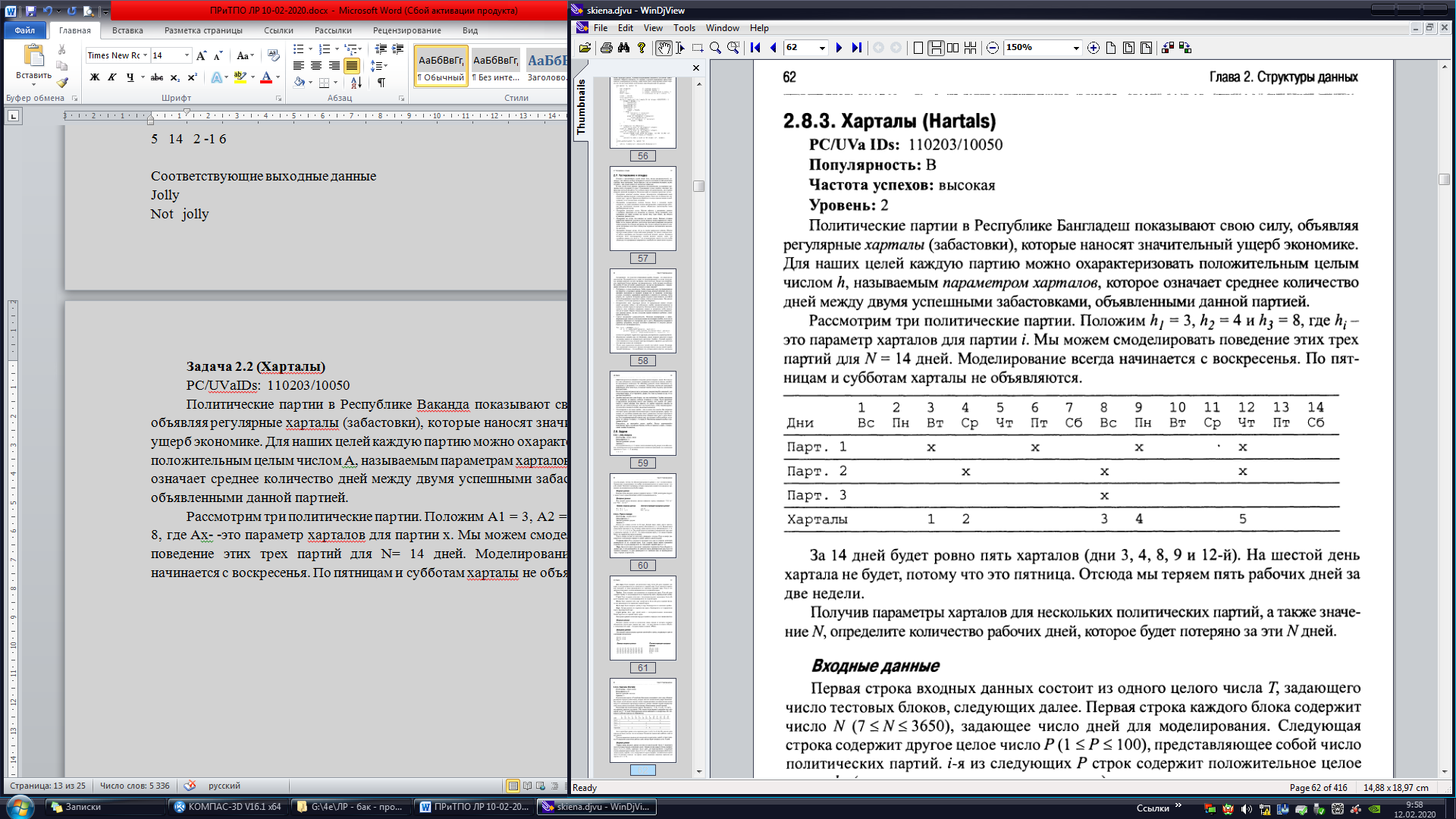
3 of Clubs

### Задача 2.5 (Харталы)

PC/UVaIDs: 110203/10050

Политические партии в Республике Ваканда показывают свою силу, объявляя регулярные харталы (забастовки), которые наносят значительный ущерб экономике. Для наших целей каждую партию можно охарактеризовать положительным целым числом А, называемым параметрам харталов, которое означает среднее количество дней между двумя успешными забастовками, объявленными данной партией.

Рассмотрим три политические партии. Положим А1 = 3, A2 = 4 и А3 = 8, где Ах- -это параметр харталов для партии х. Мы можем смоделировать поведение этих трех партий для N= 14 дней. Моделирование всегда начинается с воскресенья. По пятницам и субботам харталы не объявляются:



За 14 дней будет ровно пять харталов (дни 3, 4, 8, 9 и 12-й). На шестой день хартала не будет, потому что это пятница. Отсюда мы теряем пять рабочих дней за две недели.

Получив параметры харталов для нескольких политических партий, а также значение N, определите количество рабочих дней, которое будет потеряно за эти N дней.

**Входные данные**

Первая строка входных данных состоит из одного целого числа Т, задающего число тестовых блоков, следующих далее. Первая строка каждого блока содержит число N (7≤ N ≤ 3650), задающее число дней для моделирования. Следующая строка содержит другое целое число Р (1 ≤ Р ≤ 100), представляющее собой число политических партий, i-я из следующих Р строк содержит положительное целое число А, (которое никогда не кратно семи), задающее параметр харталов для партии i (≤ i ≤P).

**Выходные данные**

Для каждого тестового блока выведите отдельной строкой количество потерянных рабочих дней.

**Пример входных данных**

2 14 3 3 4 8 100 4 12 15 25 40

**Соответствующие выходные данные**

5 15

### Задача 2.6 (Jolly Jumpers)

PC/UVa IDs: 110108/10142

Последовательность n > 0 целых чисел называется jolly jumper, если абсолютные значения разностей последовательных элементов принимают все возможные значения от 1 до n - 1. К примеру,

1 4 2 3

это jolly jumper, потому что абсолютные разности равны 3, 2 и 1 соответственно. Определение подразумевает, что любая последовательность из одного числа - это jolly jumper. Напишите программу, которая определяет, является ли каждая из введенных последовательностей jolly jumper.

**Входные данные**

Каждая строка входных данных содержит число n < 3000, за которым следуют n целых чисел, представляющих собой последовательность.

**Выходные данные**

Для каждой строки входных данных выведите строку, говорящую "Jolly" или "Not jolly".

**Пример входных данных**

4 1 4 2 3

5 1 4 2 -1 6

## Приложение 1.1 Пример 1

Примером ко второй части ЛР является простая карточная игра.

В детской карточной игре «Война», стандартная 52-карточная колода делится на двоих игроков (1 и 2) так, чтобы у каждого игрока оказалось по 26 карт. Игроки не смотрят на свои карты, а держат их стопкой рубашкой вверх. Цель игры - выиграть все карты.

Оба игрока играют, переворачивая их верхние карты рубашкой вниз и выкладывая их на стол. Кто открыл старшую карту, тот забирает обе карты и добавляет их (рубашкой вверх) вниз своей стопки. Старшинство карт обычное, от старшей к младшей: А, К, Q, J, T, 9, 8, 7, 6, 5, 4, 3, 2. Масти игнорируются. Далее оба игрока переворачивают следующую карту и повторяют. Игра продолжается до тех пор, пока один из игроков не выиграет все карты.

Когда вскрытые карты оказываются равными по старшинству, происходит война. Эти карты остаются на столе, а игроки снимают со стопки следующую карту и кладут ее на стол рубашкой вверх, а затем еще одну, рубашкой вниз. Чья карта, вскрытая второй, оказалась старше, тот и выигрывает войну и добавляет все шесть карт снизу своей стопки. Если новые вскрытые карты также оказались равны по старшинству, то война продолжается: каждый игрок кладет еще одну карту рубашкой вверх и одну рубашкой вниз. Война продолжается тем же манером до тех пор, пока вскрытые карты оказываются равными по старшинству. Как только вскрыты различные карты, игрок, который вскрыл старшую карту, выигрывает все карты на столе.

Если у одного из игроков заканчиваются карты в середине войны, то другой игрок автоматически выигрывает. Карты добавляются снизу стопки точно в том порядке, в котором они сдавались, точнее, первая карта игрока 1, первая карта игрока 2, вторая карта игрока 1 и т. д.

Игра в войну может затянуться надолго. Но насколько долго? Задача - написать программу, которая будет симулировать игру и сообщать число ходов.

▪ Вначале рассмотрим колоду карт саму по себе.

Какая структура лучше всего подходит для представления колоды карт? Ответ на этот вопрос зависит от того, что вы собираетесь с ними делать. Собираетесь ли вы тасовать их? Сравнивать их значения? Собираетесь ли вы искать определенное расположение карт в колоде? Действия, которые вам нужны, определяют операции структуры данных.

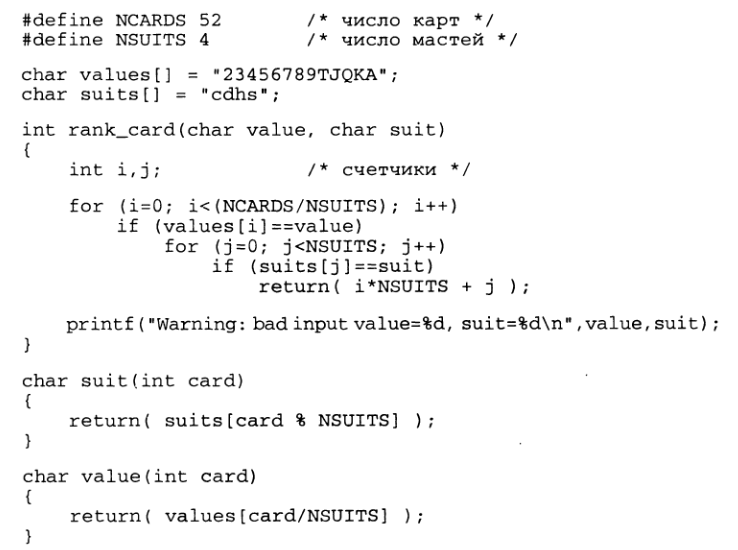
Основное действие, которое нам требуется от нашей колоды, - это сдавать карты сверху и добавлять их снизу колоды. Таким образом, естественно представить руку каждого игрока с использованием очереди FIFO.

Но есть еще и более фундаментальная проблема. Как мы будем представлять каждую карту? Карты имеют и масть (трефы, бубны, червы и пики) и значения (туз, 2-10, валет, дама и король). У нас есть возможность выбрать один из нескольких вариантов. Мы можем представлять каждую карту двумя символами или числами, которые будут означать масть и достоинство карты. В задаче про войну мы можем даже вообще не обращать внимания на масть, но такая реализация может вызвать проблемы. Что, если нам будет необходимо вывести выигравшую карту или потребуется непосредственное доказательство того, что наша реализация очереди работает правильно? Альтернативный вариант состоит в представлении каждой карты числом от 0 до 51 и задании соответствия между таким представлением и реальными картами.

Основная операция в игре «Война» - это сравнение карт по их значениям. Это сложнее сделать, используя первый вариант представления карт, потому что мы должны сравнивать по исторически сложившемуся, но случайному порядку карт. Вероятно, для этой задачи пришлось бы использовать специальную логику.

Вместо этого мы продемонстрируем подход с преобразованием данных (mapping) в качестве распространенной полезной методики профаммирования. Для любого случая, когда мы можем создать числовую ранжирующую функцию и обратную к ней деранжирующую функцию, которые применяются к определенному множеству объектов s, принадлежащих S, мы можем представить каждый объект целочисленным значением. Основное свойство состоит в том, что s = unrank(rank(s)). Таким образом, можно считать, что ранжирующая функция - это хеш-функция без конфликтов.

Как мы можем ранжировать и деранжировать игральные карты? Мы упорядочиваем карты в порядке от младшей к старшей и отмечаем, что существуют четыре карты с одинаковым значением. Умножение и деление являются ключевыми действиями при отображении карт на диапазон от 0 до 51.



Ранжирующие и деранжирующие функции легко написать для перестановок, подмножеств и большинства комбинаторных объектов. Это является распространенной методикой программирования, которая может упростить операции на многих типах данных.

▪ Строковый ввод/вывод

Для нашей программы входные данные состоят из двух строк для каждой вводимой колоды, первая строка соответствует картам игрока 1, вторая строка соответствует картам игрока 2. Вот пример карт для трех игр:

4d Ks As 4h Jh 6h Jd Qs Qh 6s 6с 2с Кс 4s Ah 3h Qd 2h 7s 9s 3c 8h Kd 7h Th Td 8d 8c 9c 7c 5d 4c Js Qc 5s Ts Jc Ad 7dKh Tc 3s 8s 2d 2s 5h6dAc 5c 9h 3d 9d 6d9d8c4sKc7c 4dTcKd3s 5h 2h Ks 5c 2s Qh 8d 7d 3d Ah Js Jd 4c Jh 6c Qc 9hQdQs 9s Ac 8h Td Jc 7s 2d 6s As 4hTs 6h2cKhTh7h5s 9c 5dAd3h8s 3c Ah As 4c 3s7d Jc 5h 8s Qc KhTd 3h 5c 9h 8c Qs 3dKs4dKd6c 6s7hQh3c Jd 2h8h7s 2c 5d7c 2dTc JhAc 9s 9c 5sQd4s Js 6dKc 2s Th 8d 9d 4h Ad 6h Ts

Считывать нечисловые данные требуется во многих задачах курса ЛР. Отметим, что у вас есть несколько способов считывания текстовых данных.

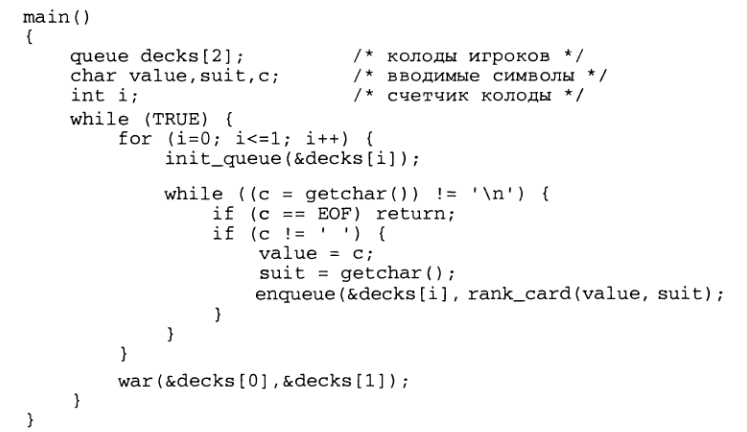
- Вы можете циклически считывать по одному символу из входного потока (то есть getchar () в С) и обрабатывать их по одному.

- Вы можете циклически скачивать форматированные лексемы (то есть scanf () в С) и обрабатывать их по необходимости.

- Вы можете считать всю строку как строковую переменную (то есть gets () в С) и после этого обработать ее путем доступа к символам или подстрокам.

- Вы можете использовать современные примитивы ввода/вывода, такие, как строковые переменные, если ваш язык их поддерживает. Конечно, вам все равно придется решить, нужны ли вам символы, строки или что-либо еще в качестве базовой единицы входных данных.

В нашей реализации войны мы воспользуемся первой возможностью из предложенных, то есть будем последовательно считывать символы и обрабатывать их один за другим. Чтобы сделать пример более иллюстративным, мы делаем явную проверку на конец строки (`\n` в С):



Обратите внимание на то, что мы представляем две колоды массивом очередей вместо двух разных переменных типа «очередь». Таким образом, мы избавляемся от необходимости повторять весь код обработки входных данных для каждой колоды.

После того как мы получили соответствующий «фундамент», разработав наши структуры данных, основная подпрограмма становится достаточно прямолинейной. Обратите внимание, что порядок, в котором выигранные карты помещаются в выигравшую их колоду, также может быть смоделирован в виде очереди, так что мы снова можем использовать наш абстрактный тип данных:



## Приложение 1.2 Общие структуры данных

К основным структурам данных традиционно относят следующие:

1. **Стеки**

Стеки и очереди - это контейнеры, из которых объекты извлекаются в зависимости от порядка их поступления и вне зависимости от их содержимого. Стеки поддерживают порядок последний вошел, первый вышел (last-in, first-out - LIFO). Очевидные операции со стеком включают в себя:

- Push(x,s) - вставить объект х на вершину стека s;

- Pop(s) - получить (и удалить из стека) верхний объект стека s;

- Initialize(s) - создать пустой стек;

- Full(s), Empty(s) - проверить, возможны ли ещё операции добавления и извлечения элемента из стека соответственно.

Важно, что для стандартных стеков и очередей не определена никакая операция поиска элементов.

Определение этих абстрактных операций позволяет создать стековый модуль для использования без знания деталей реализации. Самая простая реализация использует массив с переменной, являющейся индексом, которая обозначает вершину стека. Альтернативный вариант реализации с использованием связанных списков лучше, так как там отсутствует возможность переполнения.

Стеки хорошо моделируют «стопки» объектов, таких, как тестовые ip-адреса в тестовой локальной сети. Когда тесты сетевого приложения проведены, освобожденный ip-адрес помещается на вершину «стопки». Когда необходимо тестирование сетевого приложения, ему присваивается ip-адрес с вершины. Стек является подходящей структурой данных для решения этой задачи, так как все равно, какой адрес будет использован следующим. Таким образом, одним важным приложением стеков являются те случаи, когда порядок не важен, потому что стеки достаточно просто реализовать, а предоставляемой ими функциональности достаточно для решения всех задач.

Порядок в стеке важен при обработке любой **вложенной** структуры. Это включает в себя формулы со скобками (помещаем при «(», извлекаем при «)»), рекурсивный вызов программ (помещаем при входе в процедуру, извлекаем при выходе из процедуры) и обход графов в глубину (помещаем при обнаружении вершины, извлекаем при оставлении ее в последний раз).

**2. Очереди**: поддерживают порядок *первый вошел, первый вышел* (first-in, first-out -FIFO). Это кажется более честным, чем последний вошел, первый вышел, и именно поэтому очереди в магазинах реализованы как очереди, а не как стеки. Колоды игральных карт могут быть смоделированы очередями, так как мы сдаем карты с верха колоды, а после помешаем их под низ. Очереди FIFO будут использованы при реализации поиска в ширину в графах в последних лабораторных работах.

Очевидные операции с очередью включают в себя:

- Enqueue(x,q) - вставить объект х в хвост очереди q;

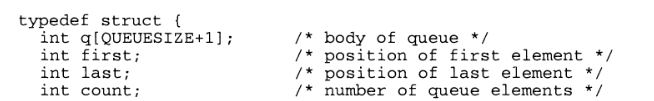
- Dequeue(q) - получить (и удалить из очереди) первый объект очереди q.

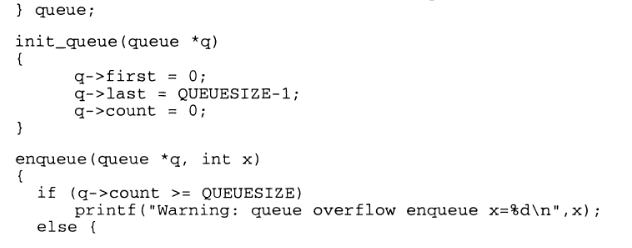
- Initialize(q), Full(q), Empty(q) - аналогичны соответствующим операциям со стеком.

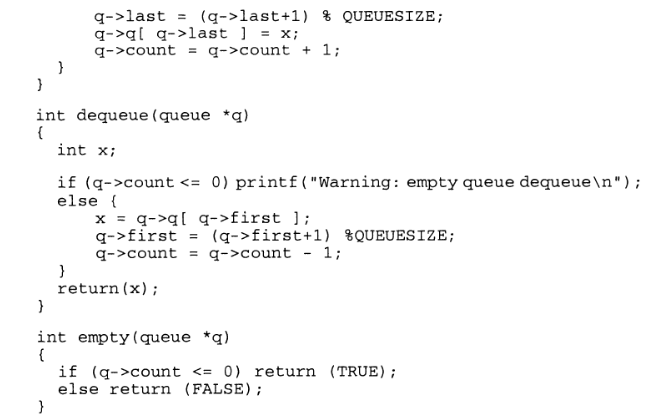
Очереди реализуются сложнее, чем стеки, потому что действие происходит на обоих концах. Простейшая реализация использует массив, вставляя новые элементы на одном конце и перемещая все оставшиеся элементы, чтобы заполнить пустое место, возникающее после каждой операции извлечения.

Тем не менее, чересчур затратно перемешать все элементы при каждой операции извлечения. Как можно реализовать лучше? Мы можем использовать индексы для первого (голова) и последнего (хвост) элемента массива/очереди и проводить все операции локально. Нет никакой причины, по которой мы должны явно очищать предварительно использованные ячейки, хотя так мы будем оставлять след из мусора позади ранее извлеченных объектов.

Кольцевые списки позволяют нам повторно использовать это пустое пространство. Обратите внимание, что указатель на голову очереди всегда позади указателя на хвост. Когда очередь полна, эти два индекса будут указывать на соседние или совпадающие элементы. Существует несколько различных способов работы с индексами для кольцевых очередей. Все они не так уж просты как может показаться. Самое простое решение отличает заполненное пространство от пустого, ведя подсчет количества элементов в очереди:







Очереди являются одной из немногих структур данных, которые легче программировать, используя связанные списки, так как в этом случае исчезает необходимость проверки условий цикличности.

**3. Словари** (Dictionaries)

Словари поддерживают извлечение по содержанию, а не по положению, как это делают стеки и очереди. Словари поддерживают три основные операции:

• Insert(x.d) - вставить объект х в словарь d;

• Delete(x.d) - удалить объект x (или объект, на который указывает х) из словаря d;

• Search(k.d) - вернуть объект с ключом k, если таковой имеется в словаре d.

В направленных структурах данных можно предложить десятки способов реализации словарей, включая отсортированные/несортированные связанные списки, отсортированные/несортированные массивы, и целый лес, полный случайных, однонаправленных (AVL) и черно-красных деревьев; не говоря обо всех вариациях хеширования.

Основным объектом анализа этих реализаций является производительность, точнее, достижение лучшего возможного компромисса между тремя основными операциями со словарем. Но на практике мы обычно хотим получить простейший путь решения проблемы, удовлетворяющий ограничениям по времени. Корректный выбор реализации зависит от того, насколько сильно меняется содержимое вашего словаря в процессе выполнения.

• Статические словари. Эти структуры строятся один раз и никогда не меняются. Таким образом, они должны поддерживать поиск, но не вставку и удаление. Правильным выбором для статического словаря обычно является массив. Единственным важным вопросом будет вопрос о том, нужно ли держать его отсортированным, чтобы использовать бинарный поиск для быстрой обработки запросов. Если у вас нет жестких временных ограничений, то, вероятнее всего, не имеет смысла использовать бинарный поиск до тех пор, пока n не превысит 100 или около того. Возможно, вы сумеете обойтись последовательным поиском до n = 1000 или более при условии, что вам не нужно будет проводить слишком много поисков. Отметим, что алгоритмы сортировки и бинарного поиска достаточно сложны для отладки.

• Полудинамические словари. Эти структуры поддерживают поиск и вставку, но не удаление. Если мы знаем верхний предел числа элементов, которые могут быть вставлены, мы можем использовать массив, иначе мы должны использовать связанные структуры. Если попробовать представить себе задачи, в которых требуется создать словарь без удобной возможности удаления элементов, примеры приходят на ум достаточно легко. Обыкновенный текстовый словарь пополняется не так уж часто, для поиска используется гораздо чаще, а слова из него удаляются разве что в странах с жесткой цензурой (допустим, добавлением «обратного» фильтра).

Хеш-таблицы являются превосходными структурами данных для словарей, в особенности если не требуется поддерживать удаление. Идея состоит в применении функции к поисковому ключу, так что мы можем определить, где объект появится в массиве, не просматривая остальные объекты. Чтобы создать таблицу разумного размера, мы должны учесть коллизии, когда два различных ключа привязаны к одной ячейке.

Два компонента хеширования - это (1) определение хеш-функции, которая привяжет ключи к целым числам в определенном диапазоне, и (2) создание массива, чей размер соответствует этому диапазону, так чтобы значение хеш-функции означало индекс.

Простая хеш-функция превращает ключ в целое число и берет значение, равное целочисленному остатку от деления этого числа на размер хеш-таблицы. Выбор простого числа в качестве размера таблицы (или, по крайней мере, отказ от выбора очевидных составных чисел, таких, как 1000) помогает избежать проблем. Строки могут быть переведены в целые числа, если использовать буквы алфавита в качестве цифр системы счисления с основанием, равным длине алфавита.

Отсутствие необходимости удаления делает открытую адресацию простым, удобным способом для разрешения конфликтов. При открытой адресации мы используем простое правило для решения того, куда положить новый объект, если желаемое место уже занято. Пусть мы всегда кладем его в следующую незанятую ячейку. При поиске данного объекта мы идем в предназначенное место и начинаем последовательный поиск. Если мы обнаруживаем пустую ячейку до того, как обнаруживаем объект, то он не существует в таблице. Удаление в схеме с открытой адресацией неприемлемо, поскольку удаление одного элемента может сломать цепочку вставок, сделав некоторые элементы недоступными. Ключ к эффективности лежит в выборе достаточно большой таблицы, в которой будет много свободного места.

• Полностью динамические словари. Хеш-таблицы также удобны для реализации полностью динамических словарей при условии, что мы используем формирование цепочки данных при разрешении конфликтов. В данном случае с каждой позицией в хеш-таблице мы связываем связанный список, так что задачи вставки, удаления и запросов сводятся к аналогичным задачам для связанных списков. Если хеш-функция работает хорошо, то m ключей будут равномерно распределены по таблице размера n, так что каждый список окажется достаточно коротким и поиск будет быстрым.

**4. Очереди по приоритету (Priority queues)**

Очереди по приоритету - это структуры данных на множествах объектов, поддерживающие три операции:

• Insert(x.p) - вставить объект х в очередь по приоритету р;

• Maximum(p) - получить объект с максимальным ключом в очереди по приоритету р;

• ExtractMax(p) - получить и удалить из очереди объект с максимальным ключом в р.

Очереди по приоритету используются для работы с расписаниями и календарями. Они решают, чья очередь следующая при моделировании аэропортов, автостоянок и схожих вещей, всех, где требуется распланировать события по часам, реагировать на какие-либо изменения данных (приоритетов).

Очереди по приоритету также используются при планировании событий в алгоритмах с линейным проецированием (sweep-line), обычных в вычислительной геометрии. Чаше всего, очереди по приоритету при этом используют для хранения точек, которые еще не встречались, упорядоченных по х-координате, и продвижения линии вперед по одному шагу.

Самая распространенная реализация очередей с приоритетом это частично упорядоченное бинарное дерево - бинарная куча (binary heap), с которым можно эффективно работать как сверху вниз, так и снизу вверх. Эти деревья очень удобны и эффективны, но имеют некоторые проблемы со временем отклика. Гораздо проще работать с отсортированным массивом, особенно если не предполагается слишком много вставок.

**5. Множества (Sets)**

Множества (подмножества) - это неупорядоченные наборы элементов, набранные из заданного универсального множества U. Структура данных множества отличается от словарей, потому что существует, как минимум, неявная необходимость задать, какие элементы из U не входят в данное подмножество.

Базовые операции над подмножествами:

• Member(x,S) - является ли объект х элементом подмножества S?

• Union(A.B) - построить множество А и В из всех элементов, которые принадлежат А или В;

• Intersection(A.B) - построить множество А п В из всех элементов, которые принадлежат и А и В;

• Insert(x.S), Delete(x.S) - вставить/удалить элемент х в/из множества S.

Для множеств большого или неограниченного универсума, очевидным решением будет представление множества словарем. Использование отсортированных словарей делает реализацию операции объединения и пересечения намного легче, по существу сводя задачу к слиянию двух отсортированных массивов. Элемент лежит в объединении, если он появился хотя бы раз в объединенном списке, и лежит в пересечении, если он появился там ровно два раза.

Для множеств, набранных из небольших, неизменных универсумов, удобным представлением будут битовые векторы: n-битный вектор или массив может представлять собой любое подмножество из n -элементного универсума. Бит i будет выставлен в 1, если i принадлежит S. Вставка и удаление элементов просто изменяют значение соответствующего бита на противоположный. Пересечение и объединение делаются с помощью операций логического «и» и логического «или» соответственно. Так как на один элемент тратится только один бит, то битовые векторы могут иметь небольшой размер для удивительно больших значений U. Например, массив из 1000 стандартных четырехбайтовых целых чисел (integer) может представлять любое подмножество 32 000 элементов.

В C++ данные структуры реализованы в библиотеке STL, в JAVA – в пакете java.util.

# Лабораторная работа 2

Строки и сортировка

**Задания**

1. В соответствии со своим вариантом выберите по одной задаче из части 1 и части 2. Ознакомьтесь с примером решения задачи и теорией (см. приложения).

2. Напишите программы, решающие выбранные вами задачи.

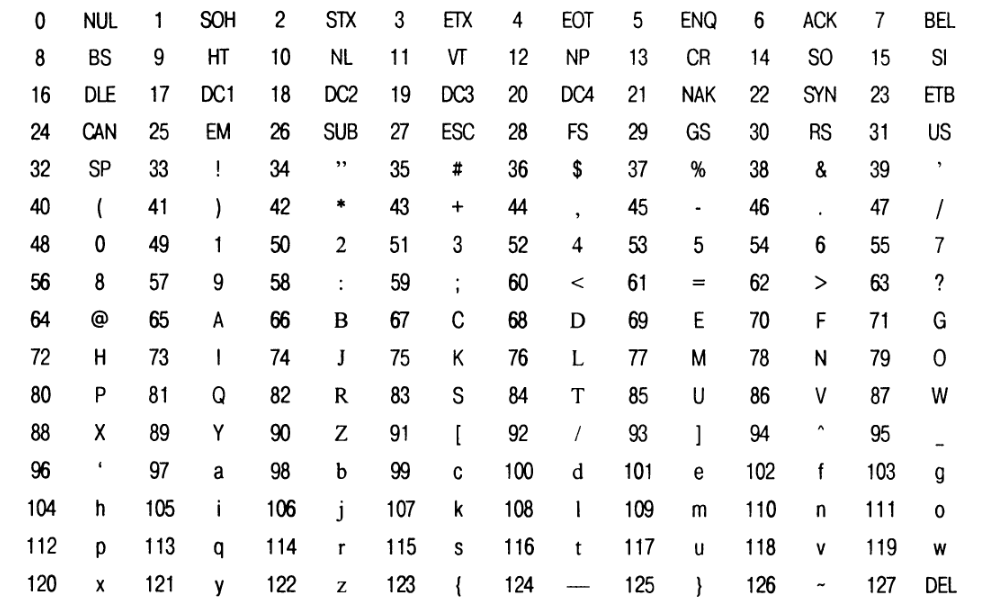
3. Оформите отчет по ЛР.

## Часть 1. Строки

Текстовые строки - это фундаментальная структура данных, чья важность постоянно возрастает. Поисковики в Интернете, такие, как Google, находят миллиарды документов практически мгновенно. Расшифровка человеческого генома даёт три миллиарда символов текста, описывающих все протеины, из которых создан человеческий организм. Множество ученых по всему миру занято поиском структур и последовательностей в этой строке.

При работе с текстовыми строками полезно немного разбираться в идейной основе кодов символов. Американский стандартный код обмена информацией (American Standard Code for Information Interchange - ASCII) - это однобайтовый код символов, для которого заданы 2 - 128 символов. Байты состоят из восьми бит; это значит, что старший бит всегда выставляется равным нулю.

ASCII-коды представлены ниже:



В каждой паре левый элемент - это десятичный (основание системы счисления - десять) номер в спецификации, а правый элемент - это привязанный к этому номеру символ. Привязка символов производилась не случайным образом. Несколько интересных свойств разработки таблицы облегчают программирование.

• У всех непечатаемых символов либо старшие три бита равны нулю, либо все семь младших битов равны единице. Из-за этого их очень легко отсеивать, прежде чем выводить мусор на дисплей, хотя почему-то очень немногие программы так делают.

• И прописные, и строчные буквы, а также цифры идут последовательно. Таким образом, мы можем проходить по всем буквам/цифрам, просто проходя от значения первого символа (скажем, "а") до значения последнего символа (скажем, "z").

Другим следствием последовательного расположения является то, что мы можем преобразовать символ (скажем, "I") в его порядковый номер в упорядоченной последовательности (восьмой, если считать, что "А" - это нулевой символ), просто вычтя первый символ ("А").

Мы можем преобразовать символ (скажем, "С") из верхнего в нижний регистр, добавив разницу между начальными символами верхнего и нижнего регистра ("С"-"А" + "а"). Аналогично символ *х* является прописной буквой, только если он лежит между "А" и "Z ".

• При заданных кодах символов, мы можем предсказать, что произойдет при простой сортировке текстовых файлов. Что из "х", "3 ", "С" считается первым в алфавитном порядке? Сортировка по алфавиту означает просто сортировку по кодам символов. Использование другой упорядоченной последовательности потребовало бы более сложных функций сравнения.

• Непечатаемые коды символов для новой строки (10) и возврата каретки (13) созданы для задания конца текстовых строк. Непоследовательное использование этих кодов является проблемой при переносе текстовых файлов между UNIX- и Windows-системами.

Более современные международные модели кодов символов, такие, как Уникод (Unicode), используют два или даже три байта на знак и теоретически могут представить любой символ любого языка на Земле. Тем не менее старый добрый ASCII остается в живых, будучи встроенным в Уникод. Когда все биты старшего порядка выставлены в 0, текст интерпретируется как последовательность однобайтовых, а не двухбайтовых символов. Таким образом, мы все еще можем применять более простую, более эффективную в смысле памяти кодировку и при этом иметь возможность использовать тысячи новых символов.

Все это привносит множество различий в управление текстом в различных языках программирования. Старые языки, такие, как Pascal, С, C++, считают, что тип char в сущности 8-битный. Таким образом, символьный тип данных выбирается для работы с любыми файлами, даже с теми, которые не считаются печатаемыми. С другой стороны, Java разрабатывался с поддержкой Уникода, так что символы являются 16-битными. Старший байт тождественно равен нулю при работе с ASCII/ISO Latin 1 текстом. Не забывайте об этой разнице, когда вы переключаетесь между языками программирования.

Строки это последовательности символов, причем порядок, очевидно, имеет значение. Важно знать, как ваш язык программирования представляет строки, потому что существует несколько различных возможностей.

• Массивы, оканчивающиеся нуль-символом. C/C++ считает строки массивами. Строка заканчивается, как только она доходит до нулевого символа " \ 0 ", то есть нулевой символ - ASCII. Если явно не заканчивать строку этим символом, то обычно она расширяется кучей непечатаемых символов. Если вы определяете строку и не хотите, чтобы возникла ошибка, то должен быть выделен массив достаточно большого размера, способный вместить строку максимально возможной длины (плюс нулевой символ). Преимуществом представления в виде массива является то, что отдельные символы доступны по индексу как элементы массива.

• Массив плюс длина. При другом подходе первая ячейка массива используется для сохранения длины строки, таким образом устраняя необходимость добавления символа конца строки. Такая реализация используется внутри Java, хотя пользователь рассматривает строки как объекты с набором операторов и методов, действующих на них.

• Связанные списки символов. Текстовые строки могут быть представлены с использованием связанных списков, но обычно этого избегают из-за высоких издержек памяти, связанных с использованием указателя длиной несколько байт для каждого однобайтового символа. Все же такое представление может быть полезно, если вам необходимо часто добавлять/удалять подстроки из тела строки.

Используемое представление имеет большое влияние на то, какие операции могут легко и эффективно производиться. Вы можете сравнить эти три структуры данных по отношению к следующим свойствам:

• Какая использует наименьшее количество памяти? Для строк какого размера?

• Какая ограничивает содержимое строк, которое возможно представить? Какая обеспечивает постоянный доступ к i-му символу?

• Какая предоставляет удобный способ проверки, что i-й символ действительно лежит внутри строки, помогая тем самым избежать ошибок нарушения границ?

• Какая предоставляет удобный способ вставки/удаления /-го символа?

• Какое представление используется, когда пользователи ограничены максимальной длиной строк 255 (например, имена файлов в Windows)?

Пример разработки программы для обработки строк представлен в приложении 2.1.

### Задача 3.1 (Дублеты)

PC/UVaIDs: 110307/10150

Дублетом называются два слова, которые отличаются ровно в одной букве (например, «booster» и «rooster», или «rooster» и «roaster», или «roaster» и «roasted»).

Вам задается словарь длиной не более 25 143 слов, состоящих из строчных букв, причем длина каждого слова не превышает 16 букв. Далее вам задается некоторое число пар слов. Для каждой пары слов найдите последовательность слов, имеющую наименьшую длину, причем первое слово последовательности должно совпадать с первым словом из заданной пары, а последнее слово последовательности - со вторым словом из пары. Каждая пара соседних слов последовательности должна быть дублетом. Например, если вам задана пара «booster» и «roasted», возможным решением является «booster», «rooster», «roaster», «roasted», при условии, что все эти слова присутствуют в словаре.

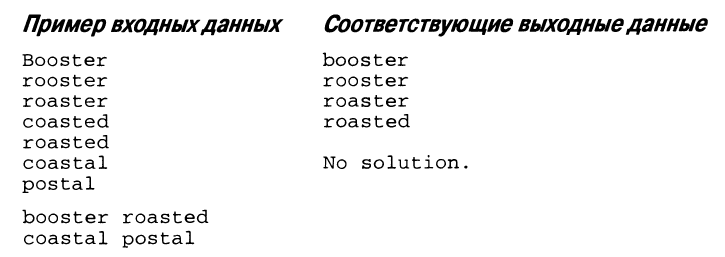
**Входные данные**

Файл входных данных содержит словарь, за которым следует некоторое число пар слов. Словарь состоит из произвольного числа слов, по одному на строку, и завершается пустой строкой. Далее идут пары слов; каждая строка содержит пару слов, разделенных пробелом.

**Выходные данные**

Для каждой введенной пары напечатайте набор строк, начинающийся с первого слова и заканчивающийся последним. Каждая пара соседних строк должна быть дублетом.

Если существует несколько вариантов минимальных решений, то подойдет любое. Если решения не существует, выведите строку "No solution". Между блоками должна быть пустая строка.



### Задача 3.2 (Осколки файлов)

PC/UValDs: 110306/10132

Ваш друг споткнулся, когда вез тележку с компьютерными файлами по лаборатории. Все файлы упали на землю и разбились. Ваш друг собрал все осколки и обратился к вам с просьбой снова собрать их вместе.

К счастью, **все файлы на тележке были одинаковы**, каждый разбился ровно на два осколка, и все осколки были найдены. К сожалению, не все файлы сломались в одном и том же месте, и осколки полностью смешались во время падения на пол.

Оригинальные бинарные осколки были транслированы в строки ASCII из единиц и нулей. Вашей задачей является написание программы, которая определяет последовательность бит, записанную в невезучем файле.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Каждый блок состоит из последовательности «файловых осколков», по одному на строку, завершающейся какой-либо меткой или пустой строкой. Каждый осколок состоит из ASCII нулей и единиц.

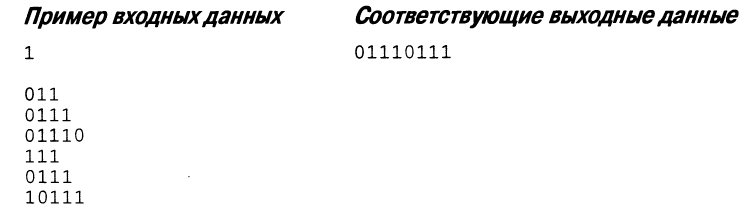
Во входных данных содержится 2N осколков, поэтому должна существовать возможность объединить эти осколки в пары так, чтобы получилось **N копий** выходной строки.

**Выходные данные**

Для каждого тестового блока выведите единственную строку, состоящую из ASCII нулей и единиц и задающую битовую строку оригинального файла. Если существующее решение не единственно, то в качестве выходных данных подойдет любое из возможных решений.

Ваш друг уверен, что на тележке было не более 144 файлов и что файлы были не длиннее 256 байт.

Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.



### Задача 3.3 (Дешифратор II)

PC/UVa IDs: 110304/850

Распространенный, но ненадежный метод шифровки текста состоит в перемене букв алфавита. Другими словами, каждая буква алфавита последовательно заменяется в тексте какой-то другой буквой. Чтобы шифровка была обратимой, никакие две буквы не заменяются одной и той же буквой.

Мощным методом криптоанализа является атака с известным открытым текстом (known plain text attack). При атаке с известным открытым текстом дешифровщик знает фразу или предложение, зашифрованное противником, и путем изучения зашифрованного текста выясняет метод кодировки.

Ваша задача - расшифровать несколько зашифрованных строк текста, полагая, что каждая строка использует один и тот же набор замещений и что одна из закодированных строк является шифровкой открытого текста *the quick brown fox jumps over the lazy dog*.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

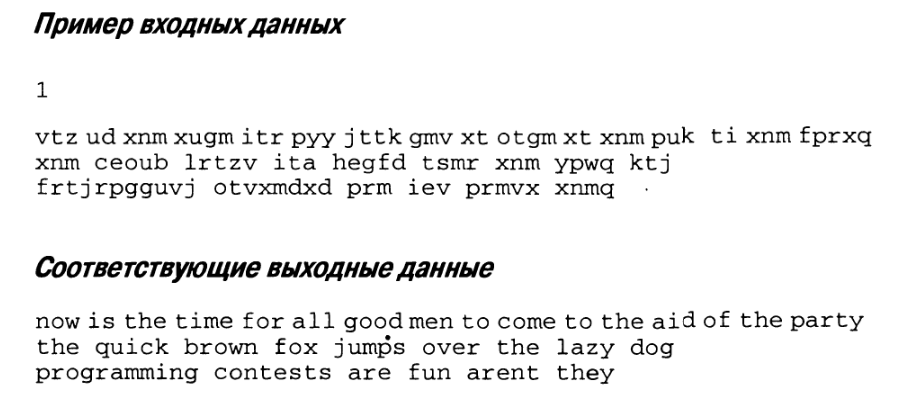
Каждый блок состоит из нескольких строк входных данных, зашифрованных способом, описанным выше. Зашифрованные строки содержат только строчные буквы и пробелы, и их длина не превышает 80 символов. Число строк входных данных не превышает 100. Номер известной вам строки не известен вам заранее.

**Выходные данные**

Для каждого тестового блока расшифруйте каждую строку и напечатайте их в стандартный вывод. Если существует несколько различных вариантов дешифровки, то подойдет любой. Если расшифровка невозможна, выведите No solution.

Выходные данные для двух последовательных блоков должны быть разделены

пустой строкой.



### Задача 3.4 (Обычная перестановка)

PC/UVa IDs: 110303/10252

Даны две строки a и b вывести строку х максимальной длины, состоящую из, букв, таких, что существует произвольная перестановка х, являющаяся подстрокой произвольной перестановки строки a и одновременно являющаяся подстрокой произвольной перестановки строки b.

**Входные данные**

Файл входных данных содержит несколько блоков, причем каждый блок состоит из двух последовательных строк. Это значит, что строки 1 и 2 - это тестовый блок 1, строки 3 и 4 - другой тестовый блок, и т. д. Каждая строка состоит из символов нижнего регистра, причем первая строка в паре обозначает а, а вторая строка - b. Максимальная длина каждой строки - 1000 символов.

**Выходные данные**

Для каждого набора входных данных выведите строку, содержащую х. Если несколько строк подходит под вышеописанные критерии, выберите первую в алфавитном порядке.

**Пример входных данных**

pretty

women

walking

down

the

street

**Соответствующие выходные данные**

e

nw

et

### Задача 3.4 (Где Waldorf?)

PC/UVa IDs: 110302/10010

По заданной сетке букв размером m х n и списку слов определить позицию в сетке, в которой находится это слово.

Слово в сетке может располагаться только по прямой непрерывной линии букв. Регистр букв значения для совпадения не имеет (то есть строчные и прописные буквы считаются одинаковыми). Слово может располагаться в любом из восьми диагональных, горизонтальных и вертикальных направлений.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Каждый блок начинается со строки, содержащей два целых числа m и n, причем 1 < m,n < 50 в десятичной системе счисления. Следующие m строк содержат ровно по n букв каждая, они представляют собой сетку букв, в которой необходимо искать слова. Буквы в сетке могут быть как прописными, так и строчными. Далее за сеткой следует строка, содержащая одно целое число k (1 < k < 20). Следующие k строк входных данных содержат список слов для поиска, одно слово в строке. Эти слова состоят только из прописных и строчных букв - никаких пробелов, дефисов или других символов, не принадлежащих стандартному алфавиту.

**Выходные данные**

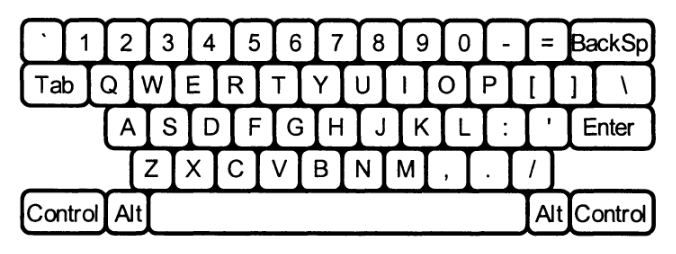
Для каждого слова в тестовом блоке выведите два целых числа, представляющих собой его положение в сетке. Эти числа должны быть разделены одним пробелом. Первое число в паре - это строка, где расположена первая буква данного слова (1 соответствует самой верхней строке, т соответствует самой нижней строке). Второе число в паре представляет собой столбец, где расположена первая буква данного слова (1 соответствует самому левому столбцу, п самому правому). Если в сетке данное слово встречается более одного раза, выведите расположение самого верхнего варианта (то есть тот случай, в котором первая буква расположена максимально близко к верху сетки). Если под это условие подходит два и более слова, выведите самый левый из этих случаев. Все слова встречаются в сетке по крайней мере один раз.

Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.



### Задача 3.5 (WERTYU)

PC/UVa IDs: 110301/10082



Обычная ошибка при наборе состоит в том, что вы помещаете ваши руки на клавиатуру на один ряд правее верной позиции. Тогда "Q" печатается как "W", "J" печатается как "К", и т. д. Ваша задача состоит в расшифровке сообщения, набранного таким образом.

**Входные данные**

Входные данные состоят из нескольких строк текста. Каждая строка может содержать цифры, пробелы, прописные буквы (кроме "Q", "A", "Z") и знаки препинания, показанные выше [кроме обратной кавычки (')]. Клавиши, обозначенные словами [Tab, BackSp, Control и т. д.], не представлены во входных данных.

**Выходные данные**

Вы должны заменить каждую букву и знак пунктуации тем, который находится непосредственно слева от него на клавиатуре QWERTY, изображенной выше. Пробелы во входных данных должны повторяться в выходных.

**Пример входных данных**

О S, GOMR YPSFU/

**Соответствующие выходные данные**

I AM FINE TODAY.

## Часть 2. Сортировка

Сортировка является одной из наиболее фундаментальных алгоритмических задач в теории вычислительных машин и систем по двум различным причинам. Во-первых, сортировка - это полезная операция, которая эффективно решает многие задачи, с которыми встречается каждый программист. Как только вы поймете, что ваша задача - это определенный случай сортировки, надлежащее использование библиотечных функций быстро решит проблему.

Во-вторых, были разработаны буквально десятки различных алгоритмов сортировки, каждый из которых основывается на определенной хитрой идее или наблюдении. Большинство примеров разработки алгоритмов ведет к интересным алгоритмам, включающим «разделяй и властвуй», рандомизацию, инкрементную вставку и продвинутые структуры данных. Из свойств этих алгоритмов следует множество интересных задач по программированию/математике.

Мы рассмотрим основные приложения сортировки, так же как и теорию, на которой основаны наиболее важные алгоритмы. Разумеется, популярные языки программирования предлагают на выбор различные библиотеки функций для сортировки данных.

Ключом к пониманию сортировки является понимание того, как она может быть использована для решения многих важных задач программирования.

• Проверка уникальности. Как мы можем проверить, все ли элементы данного набора объектов S являются различными? Отсортируем их либо в возрастающем, либо в убывающем порядке, так что любые повторяющиеся объекты будут следовать друг за другом. После этого один проход по всем элементам с проверкой равенства S[i] = S[i+1] для любого 1 < I < n решает поставленную задачу.

• Удаление повторяющихся элементов. Как мы можем удалить все копии, кроме одной, любого из повторяющихся элементов в S? Сортировка и чистка снова решают задачу. Обратите внимание, что чистку проще всего производить, используя два индекса - back, указывающий на последний элемент в очищенной части массива, и i, указывающий на следующий элемент, который нужно рассмотреть. Если S [back] < > S [i], увеличиваем back и копируем S[i] в S[back].

• Распределение приоритетов событий. Предположим, что у нас имеется список работ, которые необходимо сделать, и для каждой определен свой собственный срок сдачи. Сортировка объектов по времени сдачи (или по аналогичному критерию) расположит работы в том порядке, в котором их необходимо делать. Очереди по приоритетам удобны для работы с календарями и расписаниями, когда имеются операции вставки и удаления, но сортировка удобна в том случае, когда набор событий не меняется в процессе выполнения.

• Медиана/Выбор. Предположим, что мы хотим найти k-й по величине объект в S из n элементов. После сортировки объектов в порядке возрастания нужный нам будет находиться в ячейке S[k], либо, к примеру, в S[n/2]. В определенных случаях этот подход может быть использован для нахождения (слегка неэффективным образом) наименьшего, наибольшего и медианного элемента.

• Расчет частоты. Какой элемент чаще всего встречается в S, то есть является модой? После сортировки линейный проход позволяет нам посчитать число раз, которое встречается каждый элемент, достаточно оптимальным способом.

• Восстановление первоначального порядка. Как мы можем восстановить первоначальное расположение набора объектов, после того как мы переставили их для некоторых целей? Добавим дополнительное поле к записи данных объекта, такое, что для i-й записи это поле равняется i. Сохранив это поле во время всех перестановок, мы сможем отсортировать по нему в любой момент когда нам потребуется восстановить первоначальный порядок.

• Создание пересечения/объединения. Как мы можем рассчитать пересечение или объединение двух контейнеров? Если они оба отсортированы, мы можем объединить их если будем выбирать наименьший из двух ведущих элементов, помещать его в новое множество, если хотим, а затем удалять из соответствующего списка.

• Поиск необходимой пары. Как мы можем проверить, существуют ли два целых числа х, у в множестве S таких, что х + у = z для какого-то заданного z? Вместо того чтобы перебирать все возможные пары, отсортируем числа в порядке возрастания. С ростом S[i], при увеличении i, его возможный партнер j, такой что S[j] = z - S[i], уменьшается. Таким образом, уменьшая j соответствующим образом при увеличении i, мы получаем быстрое решение.

• Эффективный поиск. Как мы можем эффективно проверить, принадлежит ли элемент s множеству S? Конечно, упорядочивание множества с целью применения эффективного бинарного поиска - это, наверное, наиболее стандартное приложение сортировки. Но не забывайте и остальные описанные выше.

Примеры алгоритмов сортировки и практической задачи приведены в приложениях 2 и 3.

### Задача 4.1 Сортировка Шелла

PC/UValDs: 110407/10152

Король Йертл хочет перегруппировать свой трон из черепах так, чтобы его самые знатные дворяне и ближайшие советники оказались ближе к вершине. Для изменения порядка черепах в груде доступна лишь одна операция: черепаха может уползти со своего места в груде и заползти по остальным черепахам наверх.

Вам задается начальный порядок груды черепах и желаемый порядок той же груды, ваша задача состоит в том, чтобы найти минимальную последовательность операций, которые преобразуют заданную груду в желаемую.

**Входные данные**

Первая строка входных данных состоит из одного целого числа К, задающего количество тестовых блоков. Каждый тестовый блок состоит из целого числа W, задающего число черепах в груде. Следующие n строк содержат начальный порядок груды черепах. Каждая строка содержит имя черепахи, начиная с черепахи, находящейся на верху груды, и заканчивая черепахой, находящейся в самом низу. Имя каждой черепахи уникально, причем каждое имя состоит не более чем из 80 символов, которые выбираются из символьного множества, состоящего из буквенно-цифровых символов, символа пробела и точки («.»).

Следующие n строк входных данных задают желаемый порядок груды, снова перечисляя имена черепах сверху вниз. Каждый тестовый блок состоит ровно из  
2n + 1 строк. Число черепах (n) не превышает 200.

**Выходные данные**

Для каждого тестового блока выходные данные состоят из последовательности имен черепах - по одному на строку, означающих порядок, в котором черепахи должны уползать со своего места и заползать наверх. Эта последовательность операций должна преобразовывать начальный порядок в желаемый и иметь минимальную длину. Если подходят несколько различных решений, вы можете привести любое.

После каждого тестового блока должна быть пустая строка.

**Пример входных данных**

2

3

Yertle

Duke of Earl

Sir Lancelot

Duke of Earl

Yertle

Sir Lancelot

9

Yertle

Duke of Earl

Sir Lancelot

Elizabeth Windsor

Michael Eisner

Richard M. Nixon

Mr. Rogers

Ford Perfect

Mack

Yertle

Richard M. Nixon

Sir Lancelot

Duke of Earl

Elizabeth Windsor

Michael Eisner

Mr. Rogers

Ford Perfect

Mack

**Соответствующие выходные данные**

Duke of Earl

Sir Lancelot Richard M. Nixon Yertle

### Задача 4.2 (Такса)

РС/UVaIDs: 110406/10138

У платного шоссе структура оплаты проезда работает следующим образом: проезд по дороге стоит определенную сумму за каждый километр проезда в зависимости от времени суток, в котором началась поездка. Камеры на каждом въезде и выезде фиксируют номера всех въезжающих и выезжающих машин. Каждый календарный месяц владельцу машины отправляется счет за каждый километр проезда (по таксе, определяемой временем суток), плюс один доллар за поездку, плюс два доллара за предоставление счета. Ваша работа состоит в том, чтобы подготовить счет за один месяц по данным фотографиям номерных знаков автомобилей.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Каждый тестовый блок делится на две части: структура оплаты и время проезда. Структура оплаты состоит из строки, содержащей 24 неотрицательных целых числа, обозначающих плату (центов/км) с 00:00 до 00:59, плату с 1:00 до 01:59 и т. д. для каждого часа в сутках. Каждая запись проезда включает номер автомобиля (до 20 буквенно-цифровых символов), время и дату (mm: dd: hh:mm), слово enter или exit и расположение входа или выхода (**в км, считая от одного конца шоссе**). Все даты будут принадлежать одному месяцу. Каждой записи типа «enter» соответствует следующая (хронологически) запись для этого же автомобиля, при условии, что это запись типа «exit». Не имеющие пары записи типа «enter» или «exit» игнорируются. Вы можете считать, что для одного автомобиля никакие две записи не имеют одно и то же время. Время записывается в 24-часовом формате. Записей не более 1000.

**Выходные данные**

Для каждого тестового блока выведите строку для каждого автомобиля, содержащую номер автомобиля и общую сумму счета, в алфавитном порядке по номеру автомобиля. Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.

**Пример входных данных**

1

10 10 10 10 10 10 20 20 20 15 15 15 15 15 15 15 20 30 20 15 15 10 10 10 ABCD123 01:01:06:01 enter 17

765DEF 01:01:07:00 exit 95

ABCD123 01:01:08:03 exit 95

765DEF 01:01:05:59 enter 17

**Соответствующие выходные данные**

765DEF $10.80

ABCD123 $18.60

### Задача 4.3 (Задача сапожника)

PC/UValDs: 110405/10026

У сапожника имеется N заказов от покупателей, которые он должен выполнить. Сапожник может заниматься в день только одним заказом, и заказы обычно требуют на выполнение несколько дней. Для i-го заказа целое число Ti  
(1 ≤ Ti ≤ 1000) означает число дней, необходимых сапожнику для завершения заказа.

Но за популярность нужно платить. За каждый день задержки перед тем, как он приступит к работе над i-м заказом, сапожник согласился платить штраф в размере Si (1 ≤ Si ≤ 1000) центов в день. Помогите сапожнику, написав программу, находящую последовательность работ, ведущую к минимальному штрафу.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Первая строка каждого блока содержит целое число, задающее число заказов N, причем 1 ≤ N ≤ 1000. i-я последующая строка содержит время завершения Ti и ежедневный штраф Si для i-го заказа.

**Выходные данные**

Для каждого тестового блока ваша программа должна вывести последовательность заказов, ведущую к минимальному штрафу. Каждый заказ представляется своей позицией во входных данных. Все целые числа должны находиться на одной строке выходных данных и каждая пара чисел должна быть разделена одним пробелом. Если возможны несколько решений, выведите первое в лексикографическом порядке.

Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.

**Пример входных данных**

1

4

3 4

1 1000

2 2

5 5

**Соответствующие выходные данные**

2 13 4

### Задача 4.4 (Аладушки)

PC/UVaIDs: 110402/120

Приготовить идеальную стопку оладий - это хитрое дело, потому что, как вы ни старайтесь, все оладьи в стопке имеют разные диаметры. Тем не менее, для аккуратности вы можете упорядочить стопку по размеру так, чтобы каждая оладья была меньше всех оладий, находящихся под ней. Размер оладьи определяется ее диаметром.

Сортировка стопки производится серией «переворотов» оладий. Переворот состоит в том, что вы помещаете лопатку между двумя оладьями в стопке и переворачиваете (меняете порядок на обратный) все оладьи на лопатке (реверсируете подстопку). Переворот задается позицией оладьи, находящейся внизу подстопки, которую нужно перевернуть по отношению ко всей стопке. Позиция нижней оладьи 1, тогда как для стопки из n оладий позиция верхней оладьи = n.

Стопка определяется заданием диаметра каждой оладьи в стопке в порядке следования. Например, рассмотрим три стопки оладий, причем в левой стопке оладья 8 является самой верхней:

8 7 2

4 6 5

6 4 8

7 8 4

5 5 6

2 2 7

От левой стопки к средней можно перейти путем flip(3). От средней стоики к правой можно перейти путем команды flip(l).

**Входные данные**

Входные данные состоят из последовательности стопок оладий. Число оладий в каждой стопке лежит между 1 и 30, и каждая оладья имеет целочисленный диаметр, лежащий в пределах от 1 до 100. Входные данные завершаются символом конца файла. Каждая стопка задается одной строкой входных данных, при этом верхняя оладья в стопке идет первой, нижняя последней и все оладьи разделены пробелами.

**Выходные данные**

Для каждой стопки оладий ваша программа должна повторять оригинальную стопку одной строкой, за которой должна следовать последовательность переворотов, упорядочивающая стопку оладий так, что самая большая оладья находится внизу стопки, а самая маленькая наверху. Последовательность переворотов для каждой стопки должна завершаться 0, указывающим на то, что больше переворотов не нужно.

**Пример входных данных**

1 2 3 4 5

5 4 3 2 1

5 1 2 3 4

**Соответствующие выходные данные**

1 2 3 4 5 0

5 4 3 2 1 1 0

5 1 2 3 4 1 2 0

### Задача 4.5 (Семья Вито)

PC/UVaIDs: 110401/10041

Знаменитый гангстер Вито Дедстоун переезжает в Нью-Йорк. У него там очень большая семья, и все живут на Лямафия-авеню. Так как он собирается навещать всех своих родственников очень часто, он хочет найти дом рядом с ними.

На самом деле Вито хочет свести к минимуму совокупное расстояние до всех своих родственников, и он шантажирует вас с целью заставить написать программу, которая решит его проблему.

Входные данные

Входные данные состоят из нескольких тестовых блоков. Первая строка содержит число тестовых блоков.

В каждом тестовом блоке вам будет задано целое число родственников r

(0 < r < 500) и номера домов (также целые числа) s1, s2..... si.....sr, в которых они

проживают. Несколько родственников могут жить в одном доме.

**Выходные данные**

Для каждого тестового блока ваша программа должна вывести минимальную сумму расстояний от оптимального дома Вито до домов всех его родственников. Расстояние между двумя домами с номерами si, и sj вычисляется по формуле  
dij = | Si—Sj|.

**Пример входных данных**

2

2 2 4

3 2 4 6

**Соответствующие выходные данные**

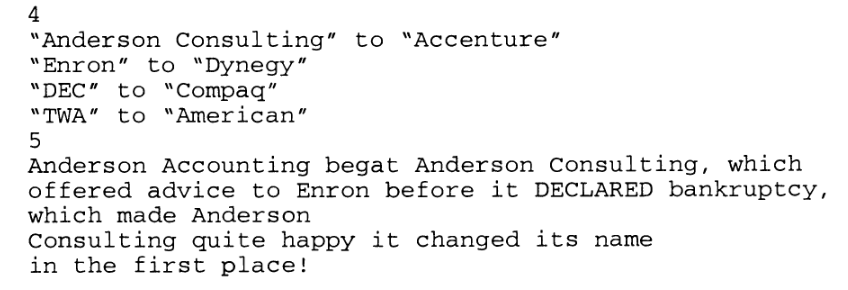
2 4

## Приложение 2.1. Пример

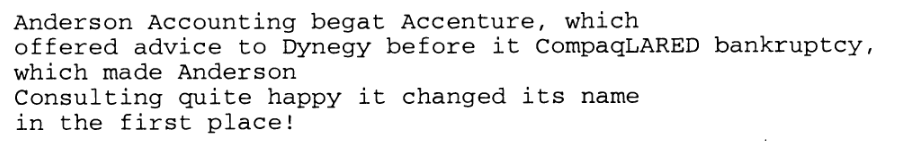
Корпоративная смена имен происходит все чаще, так как компании объединяются, покупают друг друга, пытаются скрыться от дурной славы или даже поднимают курс акций – в начале 2000-х секретом успеха было добавить .com к имени компании.

Из-за этих изменений при прочтении старых документов сложно установить текущее имя компании. Ваша компания, Digiscam (ранее Algorist Technologies), предложила вам разработать программу, которая обслуживает базу данных корпоративных смен имен и производит необходимые замены в старых документах, чтобы поддерживать их на уровне современных требований.

Входными данными для вашей программы является файл с заданным числом смен имен, за которым следует заданное число строк текста для исправления. Только точные совпадения строк должны заменяться. Смен имен будет не более 100, и длина каждой строки текста не превышает 1000 символов. Пример входных данных:



Что должно быть преобразовано в:

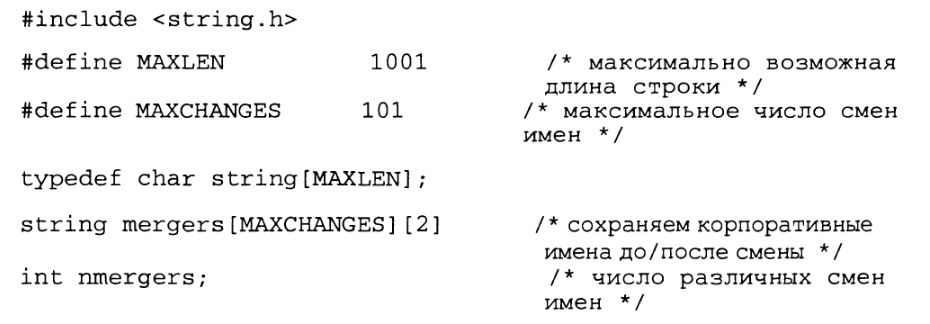


В спецификации не указано, что вы должны соблюдать разделители между словами (такие как пробел). Поэтому преобразование DECLARED в CompaqLARED это именно то, что нужно было сделать согласно ТЗ.

*Решение начинается здесь*

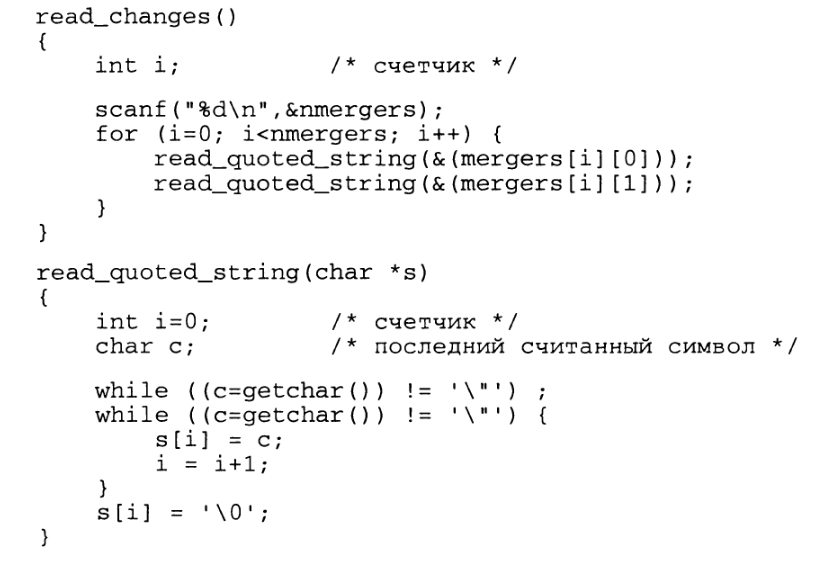
Какие типы операций со строками нам необходимы для решения этой задачи? Нам нужно уметь читать строки и сохранять их, искать необходимые вхождения в строках, изменять их и, наконец, выводить их.

Обратите внимание на то, что файл входных данных разбит на две части. Первая часть, словарь смены имен, должна быть полностью считана и обработана до начала преобразования текста. Объявление важных структур данных:



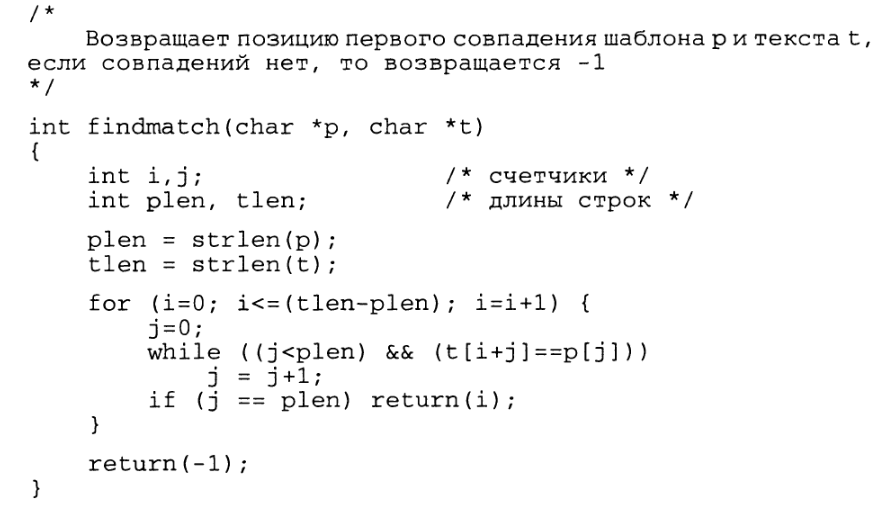
Мы представляем словарь двумерным массивом строк. У нас нет необходимости сортировать ключи в каком-то определенном порядке, так как мы будем просматривать все для каждой строки текста.

Чтение списка имен компаний в какой-то степени осложнено тем фактом, что нам нужно анализировать строку, чтобы извлечь данные между кавычками. Хитрость состоит в том, что нужно игнорировать текст до первой кавычки и забирать его до второй кавычки.



*Поиск шаблонов*

Простейший алгоритм определения присутствия шаблонной строки р в тексте / совмещает начало шаблонной строки с каждым символом текста и проверяет, совпадает ли каждый символ шаблонной строки с соответствующим ему символом текста.



Обратите внимание, что эта подпрограмма ищет только точные совпадения с шаблоном. Если буква является прописной в шаблоне, а в тексте она таковой не является, то совпадения нет. Что более важно: если имя компании разбито между строками (смотрите пример входных данных), то никакого совпадения обнаружено не будет. Такие поиски можно проводить, изменив сравнение текста/шаблона t [ i+j ] = = р [ j ] на что-либо более интересное. Подобная методика может применяться с использованием джокеров - символов, которые совпадают с любым.

Простейший алгоритм может в худшем случае потребовать О(|р||t|) времени. Можете ли вы придумать пример шаблона и текста произвольной длины, который на самом деле потребовал бы столько времени, без единого совпадения с шаблоном? Обычный прямолинейный поиск будет много более эффективен, так как мы продвигаемся дальше по тексту сразу же после того, как обнаружили первое несовпадение. Существуют и более сложные алгоритмы с линейной зависимостью времени работы от длины входных данных. Вероятнее всего, эти алгоритмы реализованы в библиотеке для работы со строками вашего языка программирования.

*Управление строками*

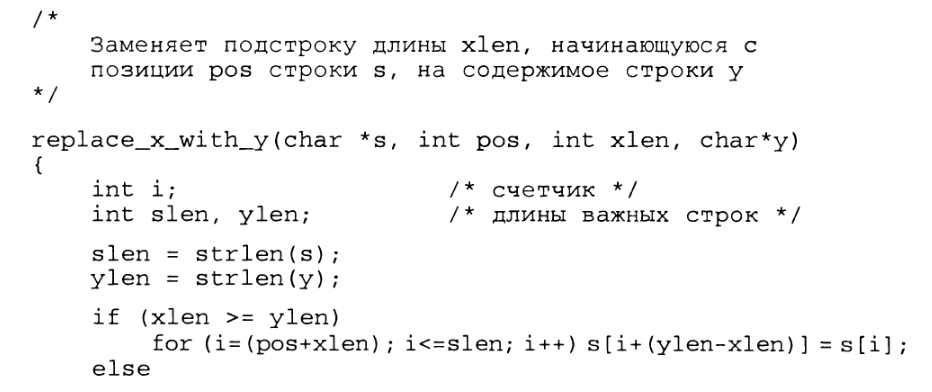
Для управления строками требуется точно знать, какое представление строк вы или ваш язык программирования использует. Здесь мы полагаем, что строки представляются последовательностью однобайтовых символов в массиве, заканчивающемся нуль-символом окончания, как принято в С.

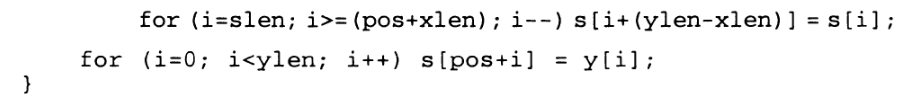
Рассмотрение строк как массивов делает многие операции сравнительно легкими.

• Вычисление длины строки. Просматриваем символы, входящие в строку, и добавляем к счетчику по единичке за каждый, пока не дойдем до нуль-символа.

• Копирование строки. Если ваш язык программирования не поддерживает копирование массивов за одну операцию, вы должны явно пройти циклом по своей строке и скопировать символы по одному. Не забывайте зарезервировать достаточно места для новой копии и не теряйте нуль-символ!

• Запись строки в обратном порядке. Простейшей реализацией этого будет скопировать строку справа налево во второй массив. Правая конечная точка определяется исходя из длины строки. Не забудьте завершить новую строку символом конца строки! Запись строки в обратном порядке может делаться путем перестановки символов, если вы хотите уничтожить первоначальную строку. В качестве примера мы реализовали подпрограмму, заменяющую подстроку в заданной позиции другой строкой. Она нам потребуется для нашей программы корпоративного слияния. Нетривиальной частью является перестановка остальных символов строки так, чтобы поместилась новая строка. Если замещающая подстрока длиннее оригинальной, нам нужно отодвинуть оставшиеся символы так, чтобы не произошло наложения. Если замещающая подстрока короче, нам нужно подвинуть оставшиеся символы так, чтобы закрыть образовавшееся пустое место.

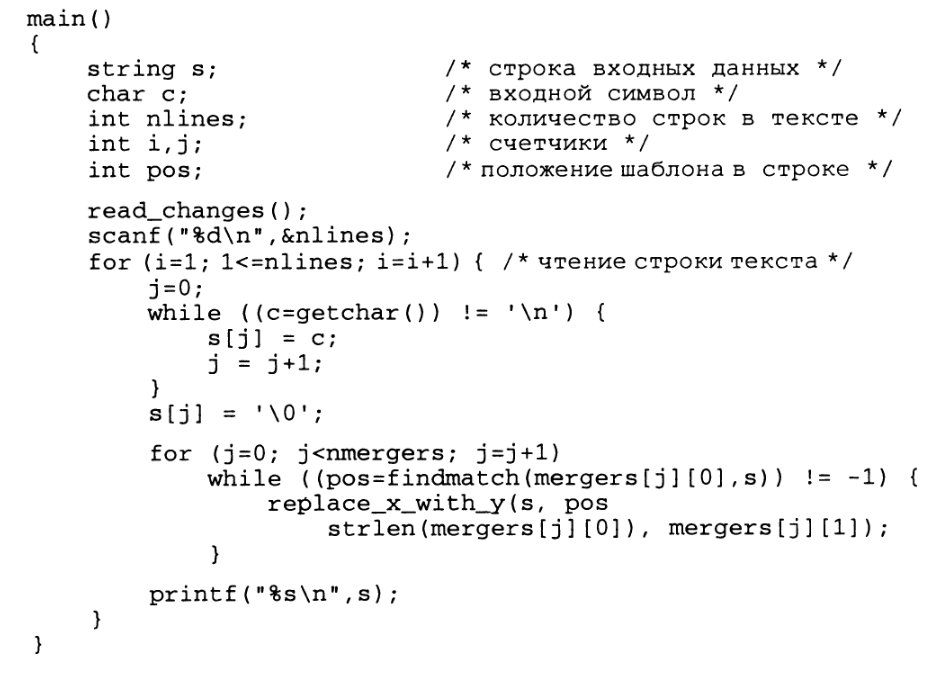




Очевидный альтернативный вариант реализации состоит в компоновке новой строки во временном буфере и последующей перезаписи всех символов строки s содержимым буфера.

*Завершение программы*

После того как созданы все необходимые подпрограммы, оставшаяся часть программы становится довольно простой:



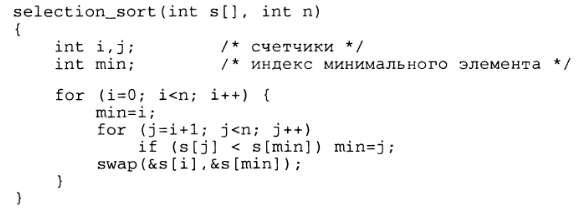
## Приложение 2.2. Алгоритмы сортировки

Вполне возможно, что вы видели десяток или даже больше алгоритмов сортировки данных. Вы помните пузырьковую сортировку, сортировку методом вставок, сортировку методом выбора, пирамидальную сортировку, сортировку слиянием, быструю сортировку, поразрядную сортировку, распределяющую сортировку, сортировку Шелла, симметричный обход деревьев и сортировочные сети? Вероятнее всего, вам надоело читать уже на середине списка. Кому нужно столько способов сделать одно и то же, особенно при условии, что существует библиотека функций сортировки, поставляемая вместе с вашим языком программирования?

Настоящей причиной для изучения алгоритмов является то, что идеи, стоящие за ними, стоят за алгоритмами для **многих других** задач. Если вы поймете, что пирамидальная сортировка на самом деле основана на структурах данных, быстрая сортировка на самом деле основана на рандомизации, сортировка слиянием на самом деле основана на принципе «разделяй и властвуй», то у вас появится большой диапазон алгоритмических инструментов для последующего использования.

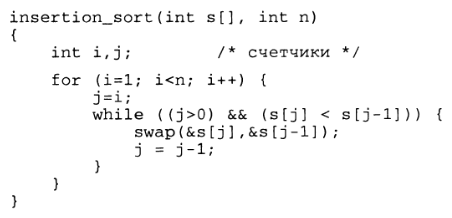
Мы рассмотрим несколько достаточно поучительных алгоритмов. Не забудьте познакомиться с полезными свойствами (такими, как минимизация перемещения данных), приведенными для каждого алгоритма.

• Сортировка методом выбора. Этот алгоритм разбивает входной массив на отсортированную и несортированную части, на каждой итерации находит наименьший элемент, имеющийся в несортированной части, и перемешает его в конец отсортированной области.



Сортировка методом выбора производит множество сравнений, но достаточно эффективна, если все, что нас интересует, - это число перемещений данных. Алгоритмом производится всего за n - 1 перестановок в наихудшем случае; считайте сортировку «обратной перестановкой цепочки минимумов» (reversed permutation). Также на ее примере можно пронаблюдать эффективность продвинутых структур данных. Использование очереди по приоритетам для несортированной части массива неожиданно превращает О(n2) сортировку методом выбора в O(n lg n)-пирамидальную сортировку.

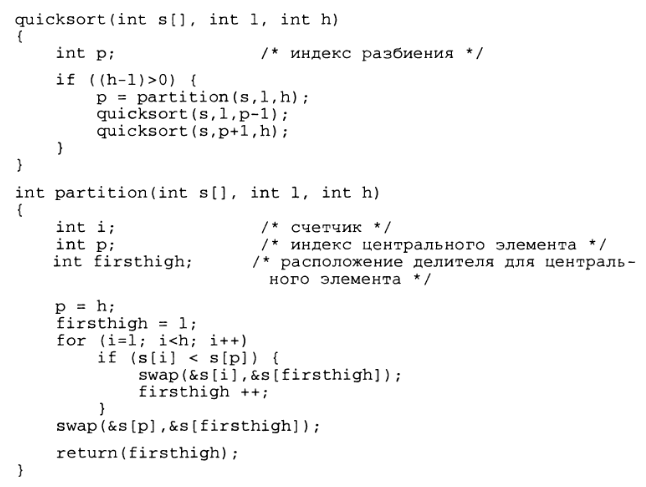
• Сортировка методом вставки. Этот алгоритм также использует отсортированную и несортированную части массива. На каждой итерации очередной несортированный элемент помещается на соответствующую позицию в отсортированной области.



Сортировка методом выбора достаточно важна как алгоритм, сводящий к минимуму количество перемещаемых данных. *Инверсией* в перестановке р называется пара элементов, стоящих не по порядку, то есть i, j такие, что i <j, хотя p[i] > p[j]. Каждая перестановка в методе вставки удаляет только одну инверсию, в противном случае ни один элемент не перемещается, так что число перестановок равняется числу инверсий. Так как в почти упорядоченной перестановке содержится очень мало инверсий, сортировка методом вставки может быть весьма эффективной для таких данных.

• Быстрая сортировка. Этот алгоритм сводит задачу сортировки одного большого массива к задаче сортировки двух меньших массивов, добавляя шаг разбиения. Разбиение делит массив на элементы меньшие центрального/делящего элемента и на элементы строго большие центрального/делящего элемента. Так как никакому элементу больше не понадобится покидать свою область, каждый подмассив может сортироваться отдельно. Для облегчения сортировки подмассивов в аргументы quicksort включены индексы первого (1) и последнего (h) элемента подмассива.

Быстрая сортировка интересна по нескольким причинам. При правильной реализации это самый быстрый алгоритм сортировки «в памяти». Алгоритм является прекрасной иллюстрацией возможностей рекурсии. Алгоритм partition удобен для многих задач сам по себе. Например, как вы разделите массив, содержащий только 0 и 1, на две части, каждая из которых состоит только из одного символа?

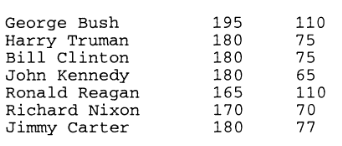


## Приложение 2.3. Пример 2: рейтинг ухажеров

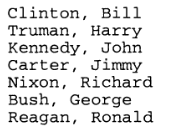
У красотки Полли нет недостатка в прекрасно воспитанных ухажерах. Напротив, самой большой ее проблемой является отслеживание самых лучших из них. Она достаточно умна, чтобы понять, что программа, ранжирующая мужчин от наиболее к наименее желаемому, упростит ее жизнь. Также она достаточно настойчива, чтобы упросить вас написать эту программу.

Полли очень любит танцевать, и она считает, что оптимальный рост ее партнера составляет1 180 сантиметров. Поэтому первое ее требование состоит в том, чтобы найти кого-либо, чей рост близок, насколько это возможно, к этой величине; будут они чуть выше или чуть ниже, не имеет значения. Среди всех кандидатов одного роста ей нужен кто-либо, чей вес близок, насколько это возможно, к 75 килограммам, но не превышает этой величины. Если все кандидаты одного роста весят больше, то она выберет самого легкого. Если у двух или более людей все эти характеристики совпадают, то отсортируйте их по фамилии, а после, если это необходимо, по имени.

Полли нужно видеть только имена отсортированных кандидатов, так что входной файл:



приведет к следующим выходным данным:



**Решение**

Суть этой проблемы состоит в достаточно сложной сортировке по критериям, заданным для нескольких полей. Существует как минимум два различных способа, которыми мы можем это сделать. В первом способе мы производим несколько проходов сортировки, сортируя сначала по наименее важному ключу, затем по следующему но важности ключу и т. д., пока мы не проведем последнюю сортировку по самому важному ключу.

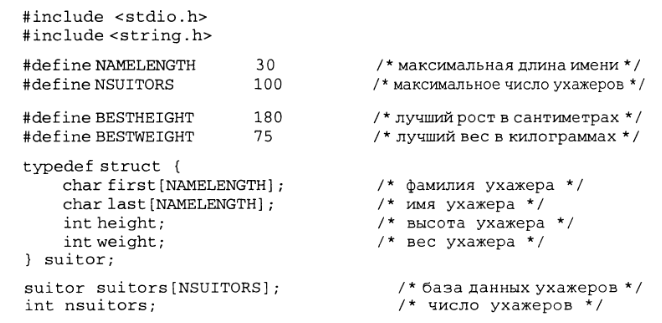
Почему нужно сортировать именно в таком порядке? Второстепенные ключи используются только для того, чтобы разрешить равенство в сортировке по главному ключу. При условии, что наш алгоритм устойчив, то есть сохраняет относительный порядок равных ключей, наша обработка второстепенных ключей остается незатронутой, если это важно для конечного ответа.

Не все алгоритмы сортировки устойчивы; более того, самые быстрые неустойчивы. Внимательно ознакомьтесь с документацией, прежде чем предполагать устойчивость любой функции сортировки.

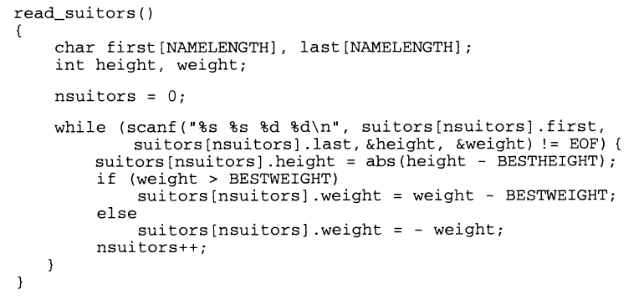
При другом подходе, который предпочтем мы, все ключи свертываются в одну сложную функцию сравнения. При выборе такого способа проще всего воспользоваться преимуществами библиотечной подпрограммы сортировки

Когда есть возможность, используйте встроенные в ваш язык программирования библиотеки сортировки/поиска.

Для решения проблемы Полли со свиданиями мы хотим сделать шаг сортировки по нескольким критериям настолько простым, насколько это возможно. Сначала мы должны разобраться с базовыми структурами данных.



Далее нам нужно считать входные данные. Обратите внимание, что мы не сохраняем действительный рост и вес ухажера! Критерии ранжирования Полли для роста и веса достаточно неудобны и основываются на том, как эти величины соотносятся с эталонным ростом/весом вместо обычного линейного упорядочивания (то есть в порядке убывания или возрастания). Вместо простого сохранения мы изменяем каждый рост и вес так, чтобы величины были линейно упорядочены по привлекательности:

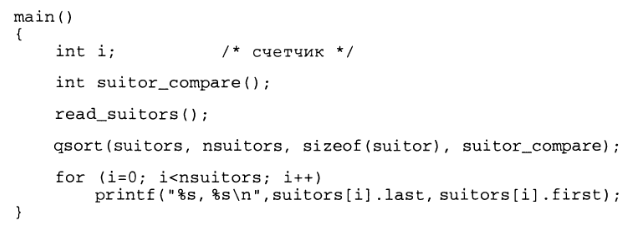


Наконец, обратите внимание, что мы прочитали имя и фамилию как лексемы, вместо того чтобы считывать их посимвольно.

Центральная функция сравнения принимает в качестве входных данных двух ухажеров a и b и решает, а лучше, b лучше или они одинаковы по привлекательности. Чтобы удовлетворить требованиям qsort, мы должны возвращать -1, 1 и 0 в этих трех случаях соответственно. Это реализовано в следующей функции сравнения:



После того как мы разобрались с подпрограммами сравнения и считывания входных данных, все, что остается, - это основная программа, которая на самом деле просто вызывает qsort и формирует выходные данные.



# Лабораторная работа 3

Арифметика и алгебра,

Задания

1. В соответствии со своим вариантом выберите по одной задаче из части 1 и части 2. Ознакомьтесь с рекомендациями по написанию кода, изложенными во введении, ознакомьтесь с примером решения задачи и теорией (см. приложения).

2. Напишите программы, решающие выбранные вами задачи.

3. Оформите отчет по ЛР.

## Часть 1. Арифметика и алгебра

Первые компьютеры создавались для ускорения вычислений, ученые-теоретики вычислительной техники также написали важные работы по чистой математике.

В любом языке программирования есть целочисленный тип данных, поддерживающий четыре основных арифметических действия: сложение, вычитание, умножение и деление. Эти действия обычно привязаны практически напрямую к аппаратным арифметическим инструкциям, так что диапазон целых чисел зависит от процессора.

Если рассматривать 32-битный компьютер, стандартный для него целочисленный тип данных поддерживает целые числа примерно в диапазоне  
± 231 = ± 2 147 483 648. Таким образом, на обычных машинах, используя стандартные целые числа, мы можем спокойно считать до миллиарда или около того.

Большинство языков программирования поддерживают long- или даже long long-целочисленный тип данных, которые часто задают 64-битные или даже 128-битные целые числа. Так как 2 = 9 223 372 036 854 775 808, то мы говорим о числах, на несколько порядков превышающих триллион. Это очень много, настолько много, что просто сосчитать до него со скоростью современного компьютера потребует времени намного больше, чем вам захочется ждать. Так что обычно его оказывается более чем достаточно, если речь не идет о математических исследованиях или о задачах соревнований по программированию.

Стандартные 32-битные целые числа обычно представляются четырьмя последовательными байтами, а 64-битные целые числа - массивом из восьми байт. Это неэффективно при сохранении большого числа не таких уж больших чисел. Например, компьютерные изображения часто представляются матрицей однобайтовых цветов (то есть 256 градаций серого) в целях эффективного использования места.

Положительные целые числа представляются в положительном двоичном виде. Для отрицательных чисел обычно используется более сложное представление, такое, как двоично дополняемое, которое облегчает вычисления на аппаратном уровне ценой большей запутанности.

Величина чисел, представляемых числами с плавающей запятой, может быть невероятно большой, особенно при использовании чисел с плавающей запятой с удвоенной точностью. Но, тем не менее, не забывайте, что эта величина является следствием представления числа в экспоненциальной форме записи, то есть в виде а х 2с. На *а* и на *с* отведено конечное число битов, поэтому точность ограничена. Не думайте, что float позволяет считать до очень больших чисел через весь диапазон включая дробный. В этих целях лучше использовать целые и длинные целые числа.

Для представления действительно огромных целых чисел требуется «сшивать» цифры вместе. Двумя возможными представлениями являются:

• *Массивы цифр*. Самым простым представлением для длинных целых чисел является массив цифр, в котором начальный элемент массива соответствует наименее значащей цифре. Использование счетчика, содержащего количество цифр в числе, может увеличить эффективность путем упрощения операций, не влияющих на выход.

• *Связанные списки цифр*. Динамические структуры необходимы в том случае, если мы действительно собираемся использовать вычисления с произвольной точностью, то есть если нет верхнего предела длины чисел. Обратите внимание, что целые числа длиной 100 000 цифр являются достаточно длинными по любым меркам, но могут быть представлены с использованием массивов длиной всего 100 000 байт каждый. На современных машинах такой размер - мелочь.

В этом разделе мы реализуем основные арифметические действия для представления массивом цифр. Динамическое выделение памяти может дать иллюзию способности получить неограниченное количество памяти по необходимости. Тем не менее, связанные структуры могут очень неэкономно расходовать память, так как часть каждого узла состоит из ссылок на другие узлы.

Что динамическое выделение памяти на самом деле дает, так это свободу использовать память там, где это нужно. Если вы хотите создать большой массив высокоточных целых чисел, несколько из которых будут большими, а остальные маленькими, тогда вам будет гораздо удобнее использовать представление в виде списка цифр, так как вы не можете себе позволить выделить огромное количество памяти для всех элементов.

Внутреннее представление чисел с плавающей запятой меняется в зависимости от компьютера, языка программирования и компилятора. Из-за этого работа с ними является большой проблемой.

Существует IEEE-стандарт для работы с числами с плавающей запятой, причем число компаний, следующих ему, постоянно увеличивается, но в любом случае могут появиться проблемы при вычислениях с очень высокой точностью. Числа с плавающей запятой представляются в экспоненциальной записи, то есть а • 2с, причем и на мантиссу а, и на порядок степени с выделено ограниченное число битов. Действия над двумя числами, у которых показатель степени сильно различается, часто приводят к ошибкам переполнения или обнуления, так как битов, отведенных на мантиссу, недостаточно для сохранения результата.

Эти проблемы являются источниками многих ошибок округления. Самая важная проблема связана с проверкой на эквивалентность двух вещественных чисел, так как младшие биты мантиссы содержат достаточное количество мусора, чтобы сделать эту проверку бессмысленной. Поэтому никогда не проверяйте, является ли число с плавающей запятой равным нулю или другому вещественному числу. Вместо этого проверяйте, лежит ли ваше число в диапазоне ±е от того числа, с которым хотите сравнивать.

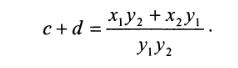
Во многих задачах вам потребуется вывести ответ с определенным количеством цифр после запятой. Тут мы должны разобраться с разницей между округлением (rounding) и усечением (truncating). Примером усечения является функция floor, преобразующая вещественное число в целое путем отбрасывания дробной части. Округление используется для более точного расчета последнего разряда числа. Для округления числа X до k десятичных цифр используйте формулу

round(X, k) = floor(10kX + (1/2)) / 10k.

Для отображения определенного числа цифр, когда это требуется, используйте функцию форматного вывода.

Вам также известны формулы для работы с дробями:

• Сложение. Перед сложением мы должны привести дроби к общему знаменателю, так что



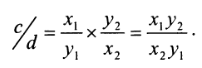
• Вычитание. Аналогично вычитанию, так как c-d = с + -\ х d, получаем



• Умножение. Так как умножение - это повторяющееся сложение, то легко показать, что



• Деление. Деление дробей эквивалентно умножению делимого на перевернутый делитель, так что



Прямая реализация этих операций ведет к значительной опасности переполнения. Важно сокращать дроби до их простейшей формы, то есть заменять 2/4 на 1/2. Ключ к этому лежит в нахождении наибольшего общего делителя числителя и знаменателя, то есть наибольшего целого числа, на которое делится без остатка и числитель и знаменатель.

Нахождение наибольшего общего делителя методом подбора или полным перебором может затребовать очень много ресурсов. Тем не менее, известный алгоритм Евклида для нахождения НОД эффективен и прост для реализации.

### Задача 5.1 (Попарно)

PC/UValDs: 110508/10202

Любой набор из n целых чисел образует n(n - 1)/2 сумм если сложить все возможные пары. Ваша задача состоит в том, чтобы найти n целых чисел по заданному набору сумм.

**Входные данные**

Каждая строка входных данных содержит n, за которым следуют n(n- 1)/2 целых чисел, разделенных пробелами, причем 2 < n < 10.

**Выходные данные**

Для каждой строки входных данных выведите одну строку, содержащую n целых чисел в неубывающем порядке таких, что входные числа - это попарные суммы этих n чисел. Если существует более одного решения, то подойдет любое. Если решения не существует, выведите "Impossibleeee".

**Пример входных данных**

3 1269 1160 1663

3 1 1 1

5 226 223 225 224 227 229 228 226 225 227

5 216 210 204 212 220 214 222 208 216 210

5 -1 0 -1 -2 1 0 -1 1 0 -1

5 79950 79936 79942 79962 79954 79972 79960 79968 79924 79932

**Соответствующие выходные данные**

383 777 886

Impossibleeee

111 112 113 114 115

101 103 107 109 113

-1 -1 0 0 1

39953 39971 39979 39983 39989

### Задача 5.2. (Игра в умножение)

PC/UVa IDs: 110505/847

Стэн (Stan) и Олли (Ollie ) играют в умножения, умножая целое число р на одно из чисел от 2 до 9. Стэн всегда начинает с р = 1, умножает, затем Олли умножает получившееся число, затем Стэн и т. д. До начала игры они выбирают целое число 1 < n < 4 294 967 295, и выигрывает тот, кто первым достигнет р ≥ n.

**Входные данные**

Каждая строка входных данных содержит одно целое число n.

**Выходные данные**

Для каждой строки входных данных выведите одну из строк - Stan wins или Ollie wins считая, что они оба играют идеально.

**Пример входных данных**

162

17

34012226

**Соответствующие выходные данные**

Stan wins

Ollie wins

Stan wins

### Задача 5.3 (Единицы)

PC/UVa IDs: 110504/10127

Для любого заданного целого числа 0 < n < 10 000, не кратного 2 и 5, существует число, кратное n, такое, что в десятичной записи оно является последовательностью единиц. Сколько цифр в наименьшем таком числе? Обратите внимание, что само число определять не обязательно.

**Пример входных данных**

3

7

9901

**Соответствующие выходные данные**

3

6

12

### Задача 5.4 (Дилемма археолога)

PC/UVa IDs: 110503/701

Археолог, ищущий доказательства того, что инопланетяне в прошлом прилетали на Землю, наткнулся на частично уничтоженную стену, содержащую странные последовательности чисел. Левая часть строк, содержащих эти цифры, всегда не повреждена, а правая, к сожалению, часто отсутствует из-за эрозии камня. Тем не менее, он заметил, что все числа, у которых сохранились все цифры, являются степенями 2, так что у него появилась очевидная гипотеза, что все числа являются степенями 2. Чтобы удостовериться в этом, он выбрал несколько чисел, для которых очевидно, что число разборчивых цифр строго меньше, чем число потерянных, и попросил вас найти минимальную степень 2 (если такая вообще существует), чьи первые цифры совпадают с теми, которые он выбрал.

Таким образом, вы должны написать программу, которая для заданного числа определяет минимальную степень Е (если она существует) такую, что первые цифры 2Е совпадают с заданным числом (не забывайте, что потеряно более половины цифр).

**Входные данные**

Каждая строка входных данных содержит положительное число N, не превышающее 2 147 483 648.

**Выходные данные**

Для каждого из этих целых чисел выведите наименьшее положительное целое число Е такое, что первые цифры 2Е в точности совпадают с цифрами N, или, если таковой степени не существует, выведите "no power of 2".

**Пример входных данных**

1

2

10

**Соответствующие выходные данные**

7

8

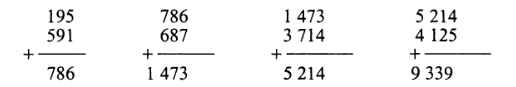
20

### Задача 5.5 (Изменение порядка и сложение)

PC/UValDs: 110502/10018,

Функция *изменения порядка и сложения* начинает с числа, меняет порядок его цифр на противоположный и складывает получившееся число с начальным. Если сумма не является палиндромом (то есть не дает одно и то же число прочитанная слева направо и справа налево), мы повторяем эту процедуру до тех пор, пока она им не станет.

Например, если мы начнем с числа 195, то получим 9339 в качестве итогового палиндрома после четвертого сложения:



Этот метод сходится к палиндрому за несколько итераций практически для всех целых чисел. Но существуют интересные исключения. 196 является первым числом, для которого не было обнаружено палиндрома. Но тем не менее не было доказано, что такого палиндрома не существует.

Вы должны написать программу, которая для заданного числа выдает итоговый палиндром (если он существует) и число итераций/сложений, которые потребовались, чтобы его найти.

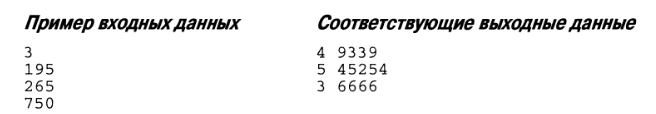
Вы можете считать, что все числа, задаваемые в качестве тестовых данных, сойдутся к ответу за менее чем 1000 итераций (сложений) и приведут к итоговому палиндрому, не превышающему 4 294 967 295.

**Входные данные**

Первая строка содержит целое число N (0< N < 100), задающее число тестовых случаев, а каждая из следующих N строк содержит одно целое число Р чей палиндром вы должны найти.

**Выходные данные**

Для каждого из N целых чисел выведите строку, содержащую минимальное число итераций, необходимых для нахождения палиндрома, один пробел и затем собственно итоговый палиндром.



### Задача 5.6 (Начала арифметики)

PC/UVaI Ds: 110501/10035

Детей учат складывать многоразрядные числа справа налево, по одной цифре за один раз. Многие из детей считают операцию «переноса», когда 1 переносится в следующий разряд, достаточно сложной. Ваша работа состоит в том, чтобы сосчитать число операций переноса для каждого набора задач на сложение, чтобы учителя могли оценить их сложность.

**Входные данные**

Каждая строка входных данных содержит два беззнаковых целых числа, каждое длиной не более 10 цифр. Последняя строка входных данных содержит «0 0».

**Выходные данные**

Для каждой строки входных данных, за исключением последней, рассчитайте число операций переноса, возникающих при сложении двух чисел, и выведите его в формате, показанном ниже.

**Пример входных данных**

123 456

555 555

123 594

0 0

**Соответствующие выходные данные**

No carry

3 carry

1 carry

## Часть 2. Комбинаторика

Комбинаторика - это математика счета. Существует несколько основных счетных задач, которые регулярно появляются в вычислительной технике и программировании.

Задачи по комбинаторике замечательны тем, что они опираются на смекалку и проницательность. Как только вы посмотрите на задачу под правильным углом, ответ неожиданно становится очевидным. Это делает их идеальными для соревнований но программированию - удачно подмеченный факт может устранить необходимость написания сложной программы, генерирующей и считающей все решения, и заменить это все одним вызовом простой формулы. Иногда это ведет к решениям «с отключенным компьютером». Если итоговые вычисления рассматриваются только для небольших целых чисел или фактически идентичны для всех входных данных, то можно рассчитать все возможные решения, используя, скажем, карманный калькулятор, и потом написать программу, выдающую ответы по требованию.

Приведем небольшой пример.

Рассмотрим ряд чисел (сверху записаны их номера, снизу – значения):

0 1 2 3 4 5 6 7

0 1 3 7 15 31 63 127

Можете ли вы угадать решение? Вы должны заметить, что кажется, что результат удваивается, впрочем, судя по формуле, в этом нет ничего удивительного. Но полного совпадения с 2n нет. Рассматривая вариации этой функции, вы должны прийти к гипотезе, что Тn = 2n - 1. Чтобы довести работу до конца, мы должны доказать это предположение, используя три шага:

1. Доказываем, что это верно для базиса: Т0 = 20 - 1 = 0.

2. Теперь полагаем, что это верно для Тn-1

3. Используем предположение, чтобы завершить доказательство:

Tn = 2Tn-1 + 1 = 2n-1.

**Базовые методики счета**

Здесь мы рассмотрим определенные основные правила счета и формулы, которые вы, возможно, видели, но уже забыли. В частности, существует три основных правила счета, из которых получаются многие формулы. Важно понять, какое конкретное правило нужно применять в вашей задаче.

• *Правило умножения*. Утверждает, что если существует |А| вариантов из множества А и |В| вариантов из множества В, то тогда существует |А| х |В| комбинаций одного варианта из множества А и одного из множества В.

Например, пусть у вас есть 5 рубашек и 4 брюк. Тогда у вас есть 5 х 4 = 20 различных вариантов костюмов на завтра.

• *Правило сложения*. Утверждает, что если существует |А| вариантов из множества A, и |В| вариантов из множества В, то тогда существует |А| + |В| вариантов, что случится А или В при условии, что элементы А и В различны. Например, если у вас есть 5 рубашек и 4 штанов и в прачечной повредили одну из вещей, то тогда существует 9 возможно поврежденных вещей.

• *Формула включений-исключений*. Правило сложения является специальным случаем более общей формулы, когда два множества могут пересекаться, точнее:



Например, пусть А - это набор расцветок моих рубашек, а В - это расцветки моих штанов. Используя формулу включений-исключений, я могу посчитать общее число расцветок, если я знаю, какая одежда совпадает по цвету и наоборот. Причина, по которой это срабатывает, состоит в том, что при сложении множеств мы два раза считаем определенные варианты, а конкретнее те, которые входят и в то и в другое множество. Формула включений-исключений естественным образом обобщается на три множества и более:

Проблема двойного счета - это скользкий аспект комбинаторики, который может затруднить решение задач через включение-исключение. Другой мощной методикой является установление биекции. Биекция - это взаимно-однозначное соответствие между элементами одного множества и другого. Если у вас есть такое соответствие, то при подсчете размера одного множества вы автоматически получаете размер другого.

Например, если мы подсчитаем количество брюк, которые надеты на студентах в аудитории, и будем считать, что на всех студентах надеты брюки, то мы получим число людей в аудитории. Это работает, потому что существует однозначное соответствие между брюками и людьми, которое исчезнет, если мы будем рассматривать носки или разрешим студентам носить в университете шорты.

Для использования биекций нам нужен набор множеств, которые мы умеем считать, тогда мы сможем привязывать к ним другие объекты. Основные комбинаторные объекты, которые вы должны знать, перечислены ниже. Полезно примерно понимать, с какой скоростью растет число объектов, чтобы знать, когда полный перебор перестает подходить нам в качестве наилучшего решения.

• *Перестановки*. Набор n упорядоченных объектов, в котором каждый объект встречается ровно один раз, называется перестановкой. Всего существует



различных перестановок. Например, 3! = 6 перестановок трех объектов: 123, 132, 213, 231, 312 и 321. Для n = 10 имеем n = 3 628 800, так что мы, вероятно, начнём приближаться к пределу возможностей полного перебора.

• *Подмножества*. Произвольная выборка элементов из n возможных объектов называется подмножеством. Для n объектов существует 2n различных подмножеств. Таким образом, существует 23=8 подмножеств трех объектов, а конкретно: 1, 2, 3, 12, 13, 23, 123 и пустое множество (**никогда не забывайте про пустое множество!**).Для n = 20 имеем 2n = 1 048 576.

• *Размещения с повторениями* (Strings). Последовательность символов, набираемая с возможностью повторения, называется размещением с повторениями.

Существует mn различных последовательностей из n объектов m различных видов. 27 размещений длиной 3 для набора 123: 111, 112, 113, 121, 122, 123, 131, 132, 133, 211, 212, 213, 221, 222, 223, 231, 232, 233, 311, 312, 313, 321, 322, 323, 331, 332 и 333. Число двоичных размещений длины n равняется числу подмножеств n объектов (почему?), и число возможных вариантов растет еще быстрее с увеличением m.

**Рекуррентные соотношения** значительно облегчают подсчет ряда рекурсивно заданных структур. К рекурсивно заданным структурам относятся деревья, списки, правильно построенные формулы и алгоритмы «разделяй и властвуй», так что они часто встречаются на пути ученых, занимающихся алгоритмами.

Что такое рекуррентное соотношение? Это равенство, которое определено само через себя. Почему они удобны? Потому что многие натуральные функции легко выражаются рекуррентно. Рекуррентно можно представить любой полином, включая линейную функцию:

an = an-1 + 1, a1 = 1 это запись an = n.

Любая степенная функция может быть задана рекуррентно:

an = 2 an-1, a1 = 2 это запись an = 2n.

И наконец, некоторые необычные, но интересные функции, которые не так-то просто представить в обычной записи, могут быть заданы рекуррентно:

an = nan-1, a1 = 1 это запись an = n!

Таким образом, рекуррентные соотношения являются весьма универсальным способом представления функций. Часто в качестве решения счетной задачи мы получаем именно рекуррентные соотношения. Разрешение такого соотношения и получение краткой аналитической формулы нередко относится к области искусства, но, как мы увидим, компьютерные программы могут легко вычислять значение заданного рекуррентного соотношения, даже если аналитической формы не существует.

Одним из самых важных понятий в комбинаторике является понятие **биномиальных коэффициентов**, чисел, обозначаемых (nk) и задающих число способов, которыми можно выбрать k предметов из n возможных. Что можно рассчитать с их помощью?

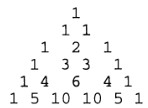
• Комиссии. Сколькими способами можно сформировать комиссию, состоящую из k членов, из n человек? По определению, (nk).

• Пути через сетку. Сколькими способами можно пройти из верхнего левого угла сетки размером n х m до правого нижнего, если идти можно только вниз или вправо? Каждый путь должен содержать m + n шагов, n вниз и m вправо. Каждый путь с различным набором шагов вниз является уникальным, так что всего существует (n+mn) таких наборов/путей.

• Коэффициенты (a + b)n. Заметим, что: (a + b)3 = 1a3 + 3a2b + 3ab2 - 1b3.

Какой коэффициент стоит перед akbn-k? Очевидно, что (nk), так как он равен числу возможных способов получения k a-одночленов из n возможных.

• Треугольник Паскаля. Каждое число является суммой двух чисел, стоящих прямо над ним:



Но почему это заинтересовало Паскаля и должно заинтересовать вас? Дело в том, что, пользуясь таким построением, можно найти биномиальные коэффициенты.

(n + 1)-й ряд треугольника задает значения (ni) для 0 ≤ i ≤ n. Изящество треугольника состоит в том, что благодаря ему можно заметить некоторые интересные соотношения, например, что сумма всех коэффициентов (n+1) ряда равняется 2n.

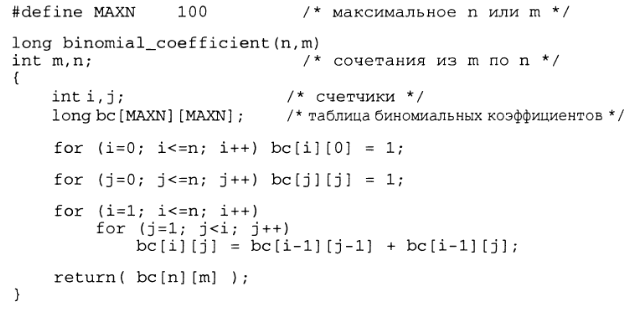
Как можно рассчитать биномиальные коэффициенты?

Во-первых, (nk) = n! / ( (n-k)! k !), так что теоретически вы можете посчитать их напрямую через факториалы. Тем не менее, у этого метода есть серьезный недостаток. Промежуточные вычисления легко могут вызвать арифметическое переполнение, тогда как итоговый коэффициент не выходит за пределы целочисленного типа данных.

Более надежный путь вычисления биномиальных коэффициентов состоит в использовании рекуррентного соотношения, неявно заложенного в треугольнике Паскаля, а именно:

(nk) = (n-1k-1) + (n-1k).

Лучше всего рассчитать такую рекуррентную последовательность, построив таблицу всех возможных величин необходимого размера. Ознакомьтесь с функцией, приведенной ниже:



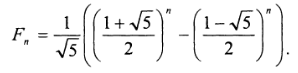
Программы, вычисляющие рекуррентные последовательности, являются основой алгоритмической методики «динамического программирования».

Существует несколько других счетных последовательностей, периодически появляющихся в приложениях и легко вычисляемых с использованием рекуррентных соотношений. Мудрый комбинаторщик никогда о них не забывает.

**Числа Фибоначчи**. Задаваемые рекуррентным соотношением Fn= Fn-1 + Fn-2 и начальными значениями F0 = 0 и F1 = 1, они периодически встречаются, вероятно, потому что это простейшее рекуррентное соотношение. Первые несколько значений: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55... .

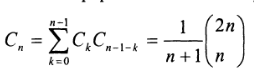
Для чисел Фибоначчи существует поразительное число математических тождеств.

Аналитическая форма:



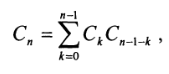
может дать очень важное для практики следствие. Так как абсолютное значение  лежит между 0 и 1, то возведение его в любую степень оставляет число в этом интервале. Таким образом, первое слагаемое в формуле- величина значительно большая и может быть использовано для оценки Fn с точностью до плюс или минус 1.

**Числа Каталана**. Рекуррентная формула и соответствующая ей аналитическая форма:



определяют числа Каталана, которые возникают в удивительном числе комбинаторных задач. Несколько первых чисел: 2, 5, 14, 42, 132, 429, 1430,..., причем С0 = 1.

Сколькими способами можно построить правильную формулу из n наборов левых и правых скобок? Например, для n - 3 это можно сделать пятью способами: ((( ))), ()(()), (())(), (()()) и ()()(). Самой левой открывающей скобке *l* соответствует определенная закрывающая скобка *r*, которая должна разбивать формулу на две правильные части, часть между *l* и *r* и часть справа от *r* . Если левая часть содержит k пар, то правая должна содержать n – k – 1 пар, так как *l, r* представляют собой одну пару Обе эти подформулы должны быть правильными, что ведет к рекуррентной формуле



и мы получаем числа Каталана.

Также отметим **числа Эйлера**, **числа Стирлинга** и т.д. Если вам нужно проверить, не являются ли Ваши числа какой-либо известной последовательностью – вам поможет справочник Sloane’s Handbook of Integer Sequences.

Наиболее мощным инструментом решения задач по рекуррентным соотношениям является рекурсия.

### Задача 6.1 (сколько чисел?)

PC/UVaIDs: 110601/10183

Напомним определение чисел Фибоначчи:

f1 = 1

f2 = 2

fn = fn-1 + fn-2 (n ≥ 3).

Для двух заданных чисел а и b определите, сколько чисел Фибоначчи лежит в диапазоне [а, b].

**Входные данные**

Входные данные содержат несколько тестовых блоков. Каждый тестовый блок состоит из двух неотрицательных целых чисел а и b. Входные данные завершаются блоком a = b = 0. В других случаях а ≤ b ≤ 10100 .

**Выходные данные**

Для каждого тестового блока выведите одну строку, содержащую количество чисел Фибоначчи fn, таких, что a ≤ fn ≤ b.

**Пример входных данных**

10 100

1234567890 9876543210

0 0

**Соответствующие выходные данные**

5

4

### Задача 6.2 (Сколько частей земли?)

PC/UVaIDs: 110602/10213

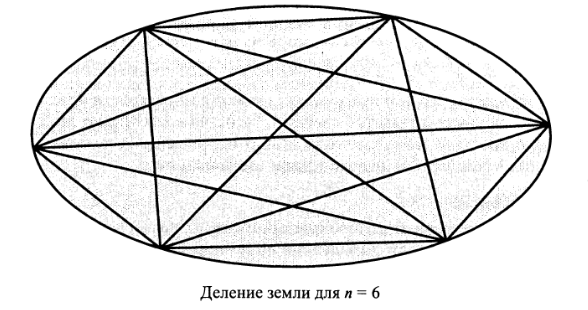
Вам дается кусок земли в виде эллипса и предлагается выбрать n произвольных точек на ее границе. После этого вы соединяете прямыми линиями каждую точку со всеми остальными, образуя n(n - 1)/2 соединений. Какое максимальное число кусков земли вы можете получить?

**Входные данные**

Первая строка входного файла содержит одно целое число s (0 < s < 3500), задающее число экземпляров входных данных. Следующие s строк задают s экземпляров входных данных, причем каждая строка содержит только одно целое число n (0 ≤ n ≤ 231).

**Выходные данные**

Для каждого экземпляра входных данных выведите на отдельной строке максимально возможное число кусков земли, задаваемых n точками.



**Пример входных данных**

4

1

2

3

4

**Соответствующие выходные данные**

1

2

4

8

### Задача 6.3 (Счет)

PC/UVaIDs: 110603/10198

Маленький Мумба знает, как считать до миллиона африканских долларов, но писать числа он еще только учится. Он уже научился писать цифры 1, 2, 3 и 4. Правда, он ещё не знает, что 4 отличается от 1, так что он думает, что 4 - это просто другой способ написания 1.

Ему нравится простая игра, которую он придумал: он пишет числа из четырех известных ему цифр и складывает их значения. Например:

132 = 1 + 3 + 2 = 6

112314 = 1 + 1 + 2 + 3 + 1 + 1 = 9 (не забывайте, что Мумба считает, что 4 = 1).

Мумбе интересно, сколько таких чисел с суммой равной числу n он может составить. Для n = 2 он может составить 5 чисел: 11 , 14 , 41 , 44 и 2. Тем не менее, он не может найти это количество для n больших 2 и поэтому просит вас помочь ему.

**Входные данные**

Входные данные состоят из произвольного числа целых чисел n таких, что  
1 ≤ n ≤ 1000.

**Выходные данные**

Для каждого считанного целого числа выведите строку, содержащую одно целое число, показывающее, сколько чисел с суммой цифр равной n может составить Мумба.

**Пример входных данных**

2

3

**Соответствующие выходные данные**

5

13

### Задача 6.4. (Правильные выражения)

PC/UVaIDs: 110604/10157

Пусть Х - это множество правильно построенных скобочных выражений. Элементами Х являются строки, состоящие только из символов «(» и «)», причем они определяются следующим образом.

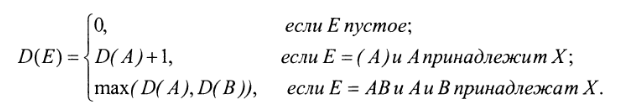
• Пустая строка принадлежит X.

• Если А принадлежит X, то (А) также принадлежит Х.

• Если А и В принадлежат X, то их конкатенация АВ также принадлежит X.

Например, строки ()(())() и (()(())) это правильно построенные скобочные выражения и поэтому они принадлежат множеству X. Выражения (()))(() и ())(() построены неправильно и не принадлежат множеству X.

*Длиной* правильно построенного скобочного выражения Е называется число одиночных скобок (символов) в Е. *Вложенность* D(E) множества Е определяется следующим образом:



Например, длина ()(())() равна 8, а вложенность 2. Напишите программу, которая считывает n и d и рассчитывает число правильно построенных скобочных выражений длины n и вложенности d.

**Входные данные**

Входные данные состоят из пар целых чисел n и d, причем на одной строке находится максимум одна пара чисел и 2 ≤ n ≤ 300, 1 ≤ d ≤ 150. Входные данные могут содержать пустые строки, которые не нужно рассматривать.

**Выходные данные**

Для каждой пары чисел во входных данных выведите строку, содержащую одно целое число - число правильно построенных скобочных выражений длины n и вложенности d.

**Пример входных данных Соответствующие выходные данные**

6 2 3

300 150 1

**Замечание.** Для длины 6 и вложенности 2 три правильно построенных скобочных выражения: ( () ) (), () ( () ) и ( () () ).

### Задача 6.5 (Монах-математик)

PC/UVaIDs: 110606/10254

Древнюю историю, связанную с задачей о ханойских башнях, знают все. Более поздняя легенда рассказывает о том, что когда брамины узнали, сколько времени у них займет перемещение **64 дисков** с одного колышка на другой, они решили найти более быстрое решение и воспользоваться им.



Один из монахов сказал своим товарищам, что они могут произвести это перемещение за день если будут работать со скоростью один диск в секунду и будут использовать дополнительный колышек. Он предложил следующее решение.

• Сначала перемещаем верхние диски (***k*** верхних дисков) на один из свободных колышков.

• Далее перемещаем оставшиеся n - k дисков (в общем случае n дисков), используя стандартное решение с тремя колышками.

• И наконец, перемещаем верхние k дисков на нужное место, используя четыре колышка.

Он вычислил значение k, сводящее к минимуму количество перемещений, и обнаружил, что хватит 18 433 перемещения. Таким образом, они могут потратить всего 5 часов 7 минут и 13 секунд, используя эту схему, вместо 500 000 миллионов лет, которые потребовались бы без дополнительного колышка!

Попробуйте повторить рассуждения мудрого монаха и вычислить число перемещений при использовании четырех колышков при условии, что за раз можно перемещать только один диск и на колышек диск можно класть только так, чтобы под ним не было дисков меньшего размера. Вычислите k, сводящее к минимуму число перемещений при таком способе решения.

**Входные данные**

Входной файл содержит несколько строк входных данных. Каждая строка содержит одно целое число 0 ≤ N ≤ 10 000, задающее число переносимых дисков.

**Выходные данные**

Для каждой строки входных данных выведите одну строку, содержащую целое число, равное числу перемещений, необходимых для переноса N дисков на конечный колышек.

**Пример входных данных**

1

2

28

64

**Соответствующие выходные данные**

1

3

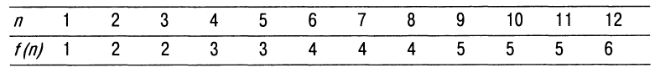
769

18433

### Задача 6.6 (Самоописывающая последовательность)

PC/UValDs: 110607/10049

Самоописывающая последовательность Соломона Голомба - это неубывающая последовательность положительных целых чисел, оперирующая понятиями «число-экземпляр» и «число экземпляров». Начало последовательности выглядит так:



Рассчитайте значение последовательности по заданному числу шагов n.

**Входные данные**

Входные данные могут содержать несколько тестовых блоков. Каждый тестовый блок занимает отдельную строку и состоит из одного целого числа n  
(1 ≤ n ≤ 2 000 000 000).

**Выходные данные**

Для каждого тестового блока входных данных выведите строку, содержащую значение функции *f*(*n*).

**Пример входных данных**

100

9999

123456

1000000000

**Соответствующие выходные данные**

21

356

1684

438744

### Задача 6.7 (Шаги)

PC/UVaIDs: 110608/846

Рассмотрим процесс пошагового перехода от целого числа х к целому числу у по целочисленным точкам числовой прямой. Длина каждого шага должна быть неотрицательной и может быть на единицу больше, равной или на единицу меньшей, чем длина предыдущего шага.

Каково минимальное число шагов, необходимое, чтобы добраться из х в y? Длина и первого и последнего шага должна быть 1.

**Входные данные**

Входные данные начинаются со строки, содержащей число тестовых блоков n. Каждый из последующих тестовых блоков состоит из строки, содержащей два целых числа: 0 ≤ х ≤ у <231.

**Выходные данные**

Для каждого тестового блока выведите строку, содержащую минимальное число шагов, которые необходимы, чтобы добраться из х в у.

**Пример входных данных**

3

45 48

45 49

45 50

**Соответствующие выходные данные**

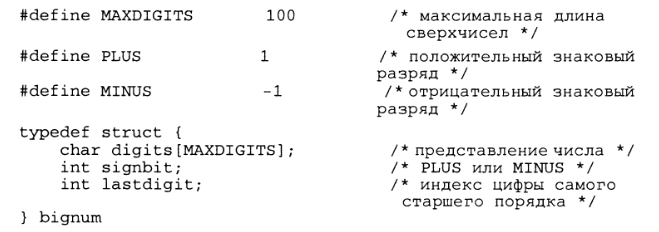
3

3

4

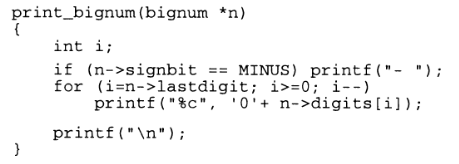
## Приложение 3.1. Пример

Наша структура данных для сверхчисел выглядит так:



Обратите внимание, что каждая цифра (0-9) представляется однобайтовым символом. Хотя работать с такими числами немного сложнее, экономия памяти позволяет нам уменьшить чувство вины за то, что мы не используем связанные структуры. Использование 1 и -1 в качестве возможных значений signbit окажется удобным, так как мы сможем перемножить знаковые разряды и получить правильный ответ.

Обратите внимание, что нет никакой причины, но которой мы должны проводить паши вычисления в десятичной системе счисления. На самом деле, использование большего основания системы счисления позволяет увеличить эффективность, уменьшая число цифр, необходимых для представления каждого числа. И все же десятичная система счисления облегчает преобразование из/в представление, которое удобно выводить (правда не менее удобна система счисления с основанием, например, 10 000):



Для простоты маши функции будут игнорировать возможность переполнения.

Первые алгоритмы, узнанные нами в школе, были связаны с четырьмя стандартными арифметическими действиями: сложением, вычитанием, умножением и делением. Мы учились с ними работать, не всегда понимая теорию, лежащую в их основе.

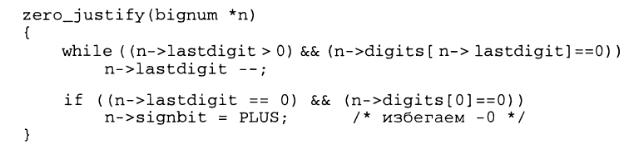
Сейчас мы рассмотрим эти действия из курса начальной школы, делая акцент на понимании того, как они работают, и на том, как можно объяснить их компьютеру. Для всех четырех действий примем следующие обозначения для аргументов с = а **х** b, где **х** - это +, -, \* или /.

• Сложение. Сложение двух целых чисел производится справа налево, при этом любой остаток переносится в следующий разряд. Наличие отрицательных чисел все усложняет, превращая сложение в вычитание. Проще всего это обработать, выделив в специальный случай.

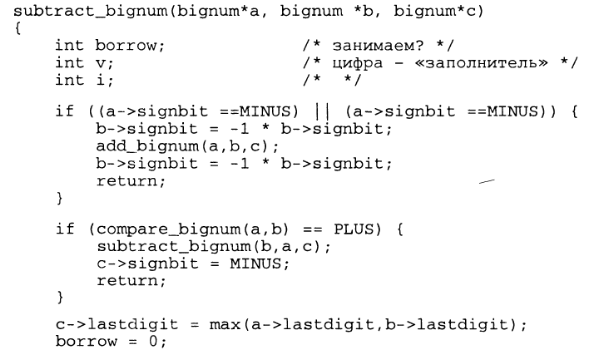


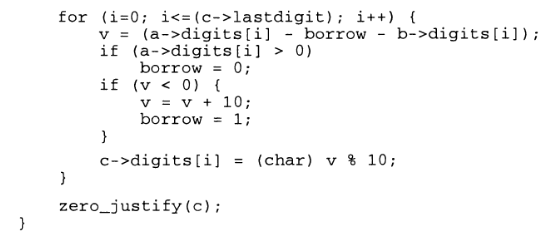
Код содержит в себе ряд интересных вещей. Работа со знаковым разрядом не относится к числу тривиальных. Мы сводим определенные случаи к вычитанию, меняя знак чисел на минус и/или переставляя порядок операций, но при этом изменение знаков происходит первым.

Собственно сложение реализуется достаточно просто и упрощено инициализацией всех цифр старших порядков в 0 и тем, что мы рассматриваем последний перенос как специальный случай сложения цифр. Операция zero\_justify изменяет lastdigit так, чтобы не считались лишние нули, стоящие впереди. Эту функцию можно спокойно вызывать после каждого действия, в особенности из-за того, что в ней поправляется случай - 0.

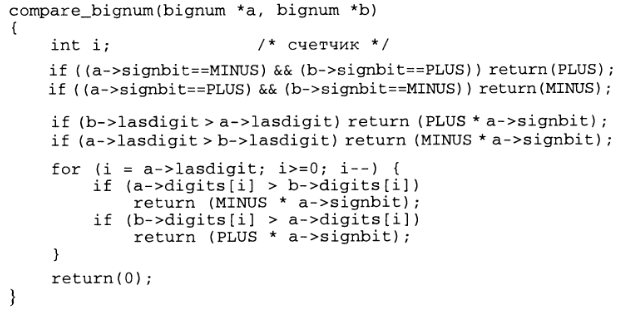


• Вычитание. Вычитание сложнее, чем сложение, потому, что при вычитании требуется занимать из соседних разрядов. Чтобы занятие не привело к ошибке, проще всего сделать так, чтобы уменьшаемое было всегда больше вычитаемого.



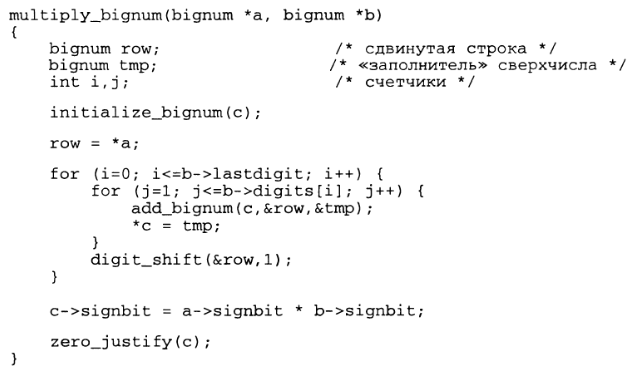


• Сравнение. Чтобы решить, какое из двух чисел больше, требуется операция сравнения. Сравнение идет от старшего разряда к младшему, начиная со знакового разряда.

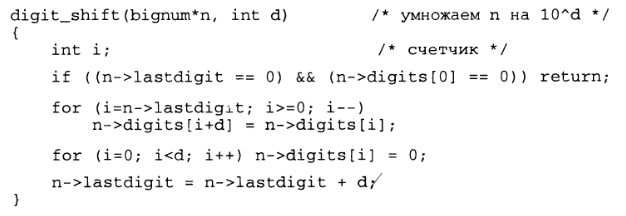


• Умножение. Умножение кажется более продвинутым действием, чем сложение и вычитание. Такая развитая цивилизация, как римляне, имела проблемы с умножением, хотя числа римлян и выглядят впечатляюще.

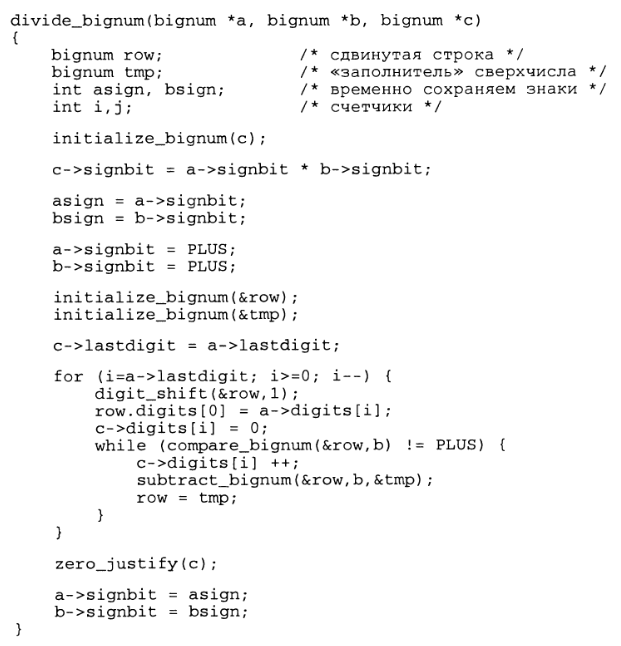
Проблема римлян была в том, что они не использовали основание системы счисления. Конечно, мы можем рассматривать умножение как многократное сложение и решить задачу так, но получится очень и очень медленно. Возведение в квадрат 999 999 требует около миллиона операций, но при этом легко делается вручную, используя метод умножения строки на строку («столбиком»), который мы знаем со школы.



Каждое действие требует сдвига первого числа на один разряд вправо и последующего прибавления сдвинутого первого числа к сумме d раз, где d- это соответствующая цифра второго числа. Мы могли сделать что-то более хитрое, чем повторяющееся прибавление, но так как цикл не может быть пройден более девяти раз на цифру, то любая экономия времени будет незначительной. Сдвиг числа на один разряд вправо эквивалентен умножению его на основание системы счисления, равной 10 для десятичных чисел.



• Деление. Хотя длинное деление - это действие, которого боятся школьники и разработчики компьютеров, оно тоже может быть обработано более простым основным циклом, чем кажется поначалу. Деление повторяющимися вычитаниями опять работает чересчур медленно с длинными числами, но простой цикл сдвига остатка влево, включая следующую цифру, и вычитания нужное число раз делителя намного проще, чем «угадывание» каждой цифры, как нас учили в школе.



Эта подпрограмма производит целочисленное деление и отбрасывает остаток. Если вы хотите посчитать остаток от деления а на b, то вы всегда можете посчитать a-b(a/b). Более хитрые методы рассматриваются при обсуждении арифметических операций над абсолютными значениями чисел.

Знак частного и остатка в случае, когда делимое и делитель имеют разные знаки, не определен, так что не удивляйтесь, если ответ будет зависеть от языка программирования.

• Возведение в степень. Возведение в степень - это повторяемое умножение, так что тут возникают те же проблемы с производительностью, что и при многократном сложении длинных чисел. Хитрость состоит в том, чтобы заметить, что

an = an div 2 • a n div 2 • a n mod 2,

так что можно обойтись, используя только логарифмическое число умножений.

# Лабораторная работа 4

Поиск с возвратом и динамическое программирование

Задания

1. В соответствии со своим вариантом выберите по одной задаче из части 1 и части 2. Ознакомьтесь с рекомендациями по написанию кода, изложенными во введении, ознакомьтесь с примерами решения и теорией.

2. Напишите программы, решающие выбранные вами задачи.

3. Оформите отчет по ЛР.

## Часть 1. Поиск с возвратом (backtrack)

Современные компьютеры обладают настолько высоким быстродействием, что иногда решение задачи полным перебором становится вполне эффективным способом. Например, временами, чтобы посчитать число объектов множества, проще напрямую сконструировать его и не использовать сложные комбинаторные формулы.

Тактовая частота современного процессора в среднем равна 1 гигагерцу, что эквивалентно миллиарду операций в секунду. Чтобы сделать что-то интересное, потребуется несколько сотен инструкций и более. Таким образом, вы можете считать, что за одну секунду на современной машине вы успеете перебрать несколько миллионов объектов.

Важно понимать, как это много (или мало) - миллион. Один миллион перестановок соответствует всем перестановкам 10 или 11 объектов, но не более. Один миллион подмножеств соответствует всем комбинациям примерно 20 объектов и не более. Для решения задач значительно большего размера требуется аккуратно отсекать все ненужные ветви поиска и внимательно следить за тем, чтобы мы перебирали только необходимые объекты.

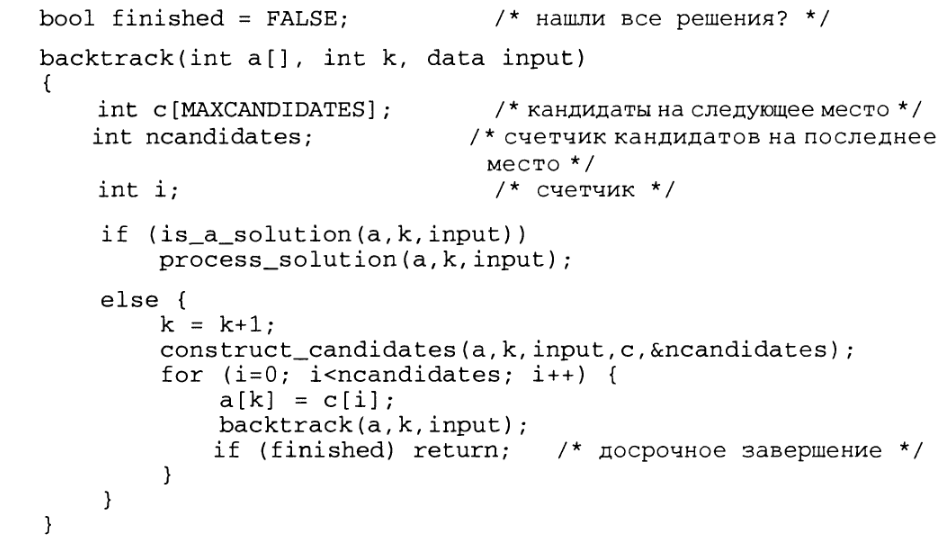
Мы рассмотрим алгоритмы поиска с возвратом для полного перебора, а также различные методики его сокращения, позволяющие использовать эти алгоритмы по максимуму.

**Поиск с возвратом** - это метод систематического перебора всех возможных конфигураций поискового пространства. Это алгоритм/методика общего характера, которая требует подстройки для каждого конкретного случая.

В общем случае мы будем формулировать решение в виде вектора а = (а1, а2, ..., an), где каждый элемент а выбирается из конечного упорядоченного множества S. Такой вектор может задавать перестановку, при этом аi задает i-й элемент перестановки. Или такой вектор может задавать заданное подмножество множества S, причем в этом случае ai в данном конкретном векторе - это истина в том и только в том случае, если i-й элемент универсума принадлежит S. Такой вектор может даже задавать последовательность ходов в игре или путь по графу, и в этом случае аi содержит i-е событие последовательности.

На каждом шаге алгоритма перебора с возвратом мы начинаем с заданного частичного решения, скажем, а = (а1, а2,..., аk) и пытаемся расширить его, добавляя еще один элемент в конец вектора. После этого мы должны проверить, является ли то, что мы имеем на данный момент, решением; если так, то мы должны вывести его, сосчитать, или сделать с ним все, что мы хотим. Если же нет, то нужно проверить, можем ли мы расширить получившееся частное решение до полного. Если да, то повторяем все вышеописанное и продолжаем. Если нет, то удаляем последний элемент а и подставляем на это место следующий возможный вариант, если он существует.

Простой вариант кода приведен ниже. Глобальный флаг finished добавлен для возможности досрочного завершения, которая может быть использована по необходимости в любой конкретной задаче если возможно приближенное решение, либо требуется завершение программы без прерывания.



В этом алгоритме три подпрограммы зависят от конкретного применения.

*- is\_a\_solution (a, k, input).* Эта булева функция проверяет, являются ли первые к элементов вектора а полным решением данной задачи. Последний аргумент, input, позволяет нам передавать в подпрограмму информацию общего характера. Мы можем использовать его для передачи n, размера искомого решения. Это имеет смысл при построении всех перестановок размера n или подмножеств n элементов, но при построении объектов переменного размера, таких, как последовательности ходов игры, может стать ненужным. При таком использовании алгоритма последний аргумент можно игнорировать.

*- construct\_candidates (a, k, input, с, ncandidates).* Эта подпрограмма заполняет массив с полным набором всех возможных кандидатов на k-е место в массиве а при условии, что заданы первые k-1 элементов. Число кандидатов возвращается с помощью ncandidates. И снова input может использоваться для передачи вспомогательной информации, такой, как желаемый размер решения.

*- process\_solution (a, k, input).* Эта подпрограмма учитывает, выводит или еще как-то обрабатывает полное решение после того, как оно построено. Обратите внимание, что вспомогательный аргумент input здесь не нужен, так как k задает число элементов в решении.

При поиске с возвратом полнота решения гарантируется полным перебором вариантов. Эффективность гарантируется тем, что ни одно возможное решение не проверяется дважды.

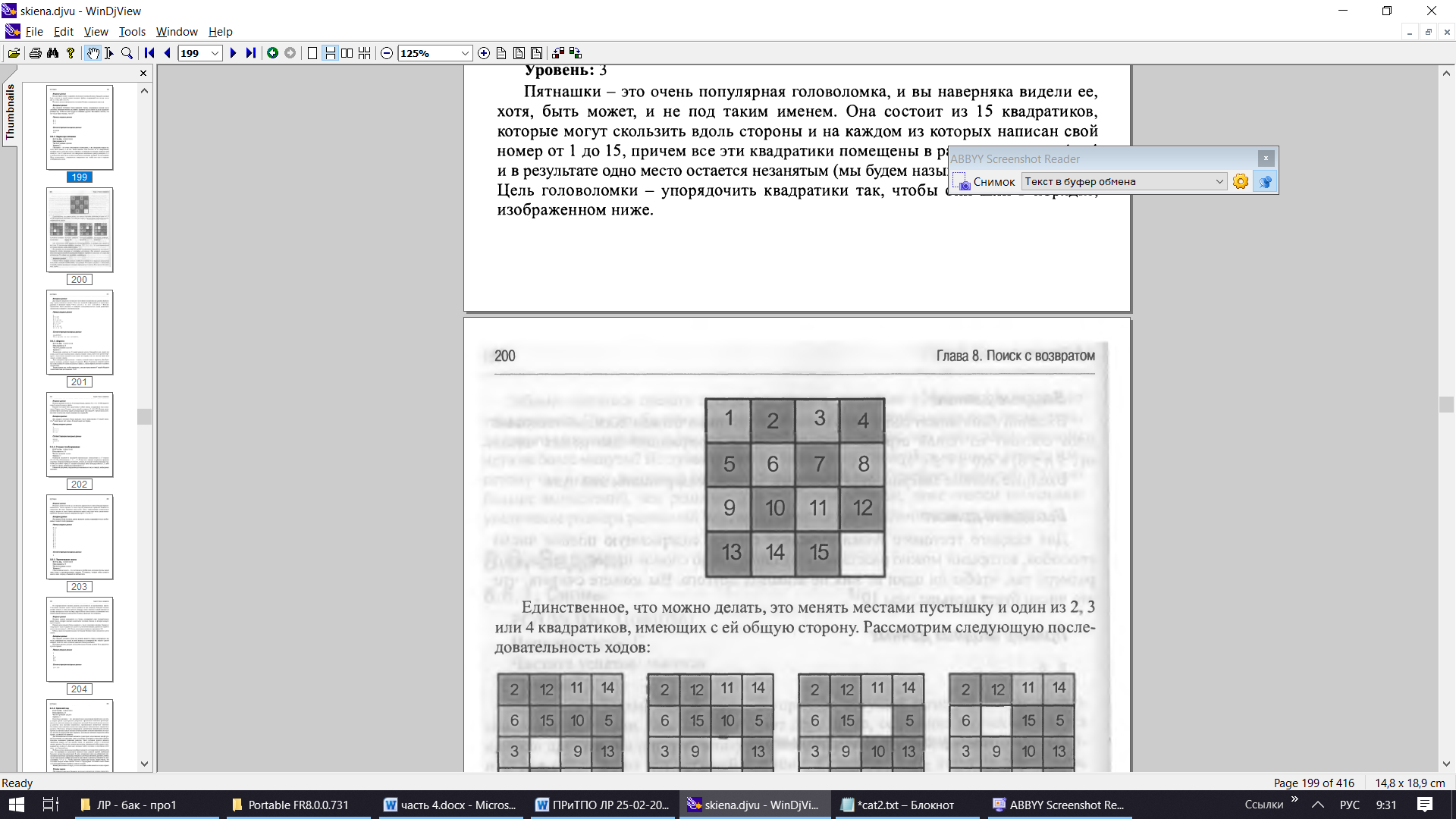
Обратите внимание на то, как легко реализуется поиск с возвратом с помощью рекурсии. Так как память под массив новых кандидатов с выделяется заново при каждом рекурсивном вызове функции, то подмножества еще не рассмотренных кандидатов на каждое место не будут пересекаться друг с другом. Мы увидим, что в графах при поиске в глубину используется практически тот же самый рекурсивный алгоритм. Перебор с возвратом можно рассматривать как поиск в глубину в неявном графе.

Пример решения тестовой задачи приведен в приложении 1.

### Задача 7.1 (Пятнашки)

PC/UVaIDs: 110802/10181

Пятнашки - это очень популярная головоломка, и вы наверняка видели ее. Она состоит из 15 квадратиков, которые могут скользить вдоль стороны и на каждом из которых написан свой номер от 1 до 15, причем все эти квадратики помещены в рамку размером 4x4 и в результате одно место остается незанятым (мы будем называть его пустышкой). Цель головоломки - упорядочить квадратики так, чтобы они шли в порядке, изображенном ниже.



Примечание: упростится ли поиск если допустить несколько верных решений (с пустышкой в любом углу и единицей в противоположном)? Вы можете реализовать этот вариант задачи. Единственное, что можно делать, это менять местами пустышку и один из 2, 3 или 4 квадратиков, имеющих с ей общую сторону.

Мы обозначаем ходы исходя из соседа пустышки, с которым она меняется местами. Разрешенными являются значения "R", "L", "U", "D" для перемещений пустышки вправо, влево, вверх и вниз.

Вы должны по начальному состоянию головоломки определить последовательность шагов, ведущую к итоговому состоянию. Вы ограничены 50 ходами для решения головоломки (достаточное ли это число для решения любого начального состояния? Опишите это в отчете).

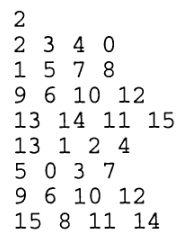
**Входные данные**

Первая строка входных данных содержит целое число и, задающее количество начальных позиций головоломки. Следующие An строк содержат п начальных позиций, причем на каждую позицию приходится 4 строки. Пустышка обозначается нулем.

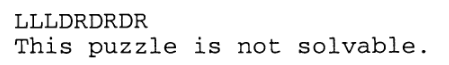
**Выходные данные**

Для каждого введенного начального состояния головоломки вы должны вывести одну строку выходных данных. Если для заданной конфигурации не существует решения, то выведите строку "This puzzle is not solvable". Если же головоломка имеет решение, то выведите последовательность ходов, решающих головоломку в формате, описанном выше.

**Пример входных данных**



**Соответствующие выходные данные**



### Задача 7.2 (Шеренга)

РС/UVaIDs: 110803/10128

Рассмотрим шеренгу из N людей разного роста. Каждый из них видит все слева, если он или она выше всех людей, стоящих слева; иначе поле зрения перекрыто. Аналогично каждый из них видит все справа, если он или она выше всех людей, стоящих справа.

Было совершено преступление - человек, стоящий слева от шеренги, убил бумерангом человека, стоящего справа от шеренги. Ровно Р человек в шеренге видели все слева и ровно R человек видели все справа, и, таким образом, они могут служить свидетелями.

Чтобы определить достоверность показаний защита наняла вас, чтобы определить, сколько перестановок N людей обладают таким свойством для заданных Р и R.

**Входные данные**

Входные данные состоят из Т тестовых блоков, причем Т (1 ≤ Т ≤ 10 000) задается первой строкой входного файла.

Каждый тестовый блок представляет собой строку, содержащую три целых числа. Первое число N задает число людей в шеренге (1 ≤ N ≤ 13). Второе число соответствует количеству людей, которые видят все слева (Р). Третье число соответствует количеству людей, видящих все справа (R).

**Выходные данные**

Для каждого тестового блока выведите число перестановок N людей таких, что Р людей видят все слева и R людей видят все справа.

**Пример входных данных**

3

10 4 4

11 3 1

3 1 2

**Соответствующие выходные данные**

90720

1026576

1

### Задача 7.3 (Станции ТО)

PC/UVaIDs: 110804/10160

Компания занимается продажей персональных компьютеров в N городах  
(3 <N< 35), обозначаемых 1, 2, ..., N. М пар этих городов соединены прямыми дорогами. Компания решила построить несколько станций техобслуживания так, чтобы для любого города X станция находилась либо непосредственно в X, либо в каком-то городе, напрямую соединенном с Х,

Напишите программу, определяющую минимальное число станций, необходимых компании.

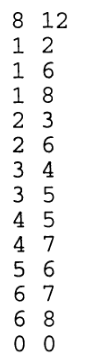
**Входные данные**

Входные данные состоят из нескольких вариантов условия. Каждый вариант начинается с числа городов N и числа пар М, разделенных пробелом. Каждая из следующих М строк содержит пару целых чисел, определяющих соединенные города, причем на одну строку приходится ровно одна пара чисел, разделенных пробелом. Входные данные завершаются при N = 0 и М = 0.

**Выходные данные**

Для каждого блока тестовых данных выведите строку, содержащую число необходимых станций техобслуживания.

**Пример входных данных**



**Соответствующие выходные данные**

2

### Задача 7.4 (Перетягивание каната)

PC/UVaIDs: 110805/10032

Перетягивание каната - это состязание в грубой силе, когда две группы людей тянут канат в противоположные стороны. Та команда, которая сумела утянуть канат в свою сторону, объявляется победителем.

На корпоративном пикнике решили посостязаться в перетягивании каната. Участников пикника нужно честно разбить на две команды. Каждый человек должен попасть в одну или другую команду, число человек в одной команде не должно превышать число человек в другой более чем на одного, и суммарные веса людей каждой команды должны быть близки, насколько это возможно.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка.

Первая строка каждого блока содержит n, число участников пикника. Каждая из следующих n строк содержит вес одного из участников пикника, причем вес задается целым числом между 1 и 450. Число участников пикника не превышает 100.

Между двумя последовательными тестовыми блоками также находится пустая строка.

**Выходные данные**

Для каждого тестового блока вы должны вывести строку, содержащую два числа: суммарный вес людей в одной команде и суммарный вес людей в другой команде. Если эти числа отличны, выведите сначала меньшее.

Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.

**Пример входных данных**

1

3

100

90

200

**Соответствующие выходные данные**

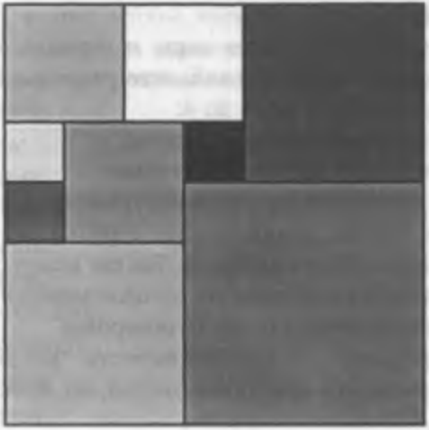
190 200

### Задача 7.5 (Больший квадрат)

PC/UVaIDs: 110808/10270

У Томми есть много бумажных квадратиков. Длина их стороны (размер) изменяется от 1 до N - 1, и у него есть неограниченное число квадратов любого размера. Но ему очень хочется получить больший квадрат - квадрат размера N.

Он может получить такой квадрат, построив его из уже имеющихся квадратов. Например, квадрат размера 7 может быть построен из 9 меньших квадратов:



Внутри квадрата не должно быть пустого места, меньшие квадраты не должны выходить за пределы большего и не должны перекрываться. Кроме того, Томи хочет истратить минимально возможное число квадратов. Поможете?

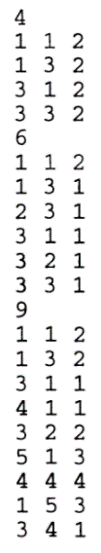
**Входные данные**

Первая строка входных данных содержит одно целое число Т, задающее число тестовых блоков. Каждый тестовый блок состоит из одного целого числа N, причем  
2 ≤ N ≤ 50.

**Выходные данные**

Для каждого тестового блока выведите строку, содержащую одно число К, задающее минимальное количество квадратов, из которых можно построить квадрат заданного размера. Далее должны идти К строк, каждая из которых должна содержать три целых числа х,у и *l*, задающие координаты левого верхнего угла (1 ≤ х,у ≤ N) и длину стороны соответствующего квадрата.

**Пример входных данных Соответствующие выходные данные**

3

4

3

7

## Часть 2. Динамическое программирование

Обычно несложно написать программу, которая найдет подходящее решение, но, чтобы программа всегда возвращала самое лучшее решение, вам нужно глубоко понимать задачу.

Динамическое программирование - это очень мощный и широко распространенный инструмент для решения задач по оптимизации структур, упорядоченных слева направо, таких как символьные строки. Он несложно реализуется, если в нем разобраться.

Чтобы динамическое программирование не казалось шаманством нужно рассмотреть достаточное количество примеров – вы можете почитать о вычислении биномиальных коэффициентов или алгоритме Флойда для вычисления кратчайших путей.

Ниже приведены две тестовые задачи. Первая - это классический пример динамического программирования, который можно найти в любом учебнике. Вторая демонстрирует более специфическое применение динамического программирования - разработку алгоритмов.

Во многих задачах требуется найти лучшее решение при заданных ограничениях. Мы уже сталкивались с подобным. Например, в задачах на перебор с возвратом часто требуется найти наибольшую, наименьшую или наиболее выигрышную конфигурацию. При переборе с возвратом перебираются все возможные решения и из них выбирается наилучшее, поэтому итоговое решение должно быть правильным. Но такой подход возможен лишь для задач небольшого размера.

Алгоритм называется *жадным*, если в каждой точке принятия решения он выбирает наилучший локальный вариант. Например, естественно было бы пытаться найти кратчайший путь из х в у выйдя из х и в каждой точке ветвления выбирая ребро минимальной стоимости. Естественно, но неверно! Более того, если корректность жадного алгоритма не доказана, то с большой вероятностью он приведет к неверному результату.

Как же быть? Динамическое программирование предлагает разрабатывать гибкие алгоритмы, систематически перебирающие все возможные варианты (что обеспечивает правомерность), но сохраняющие при этом промежуточные результаты, чтобы избежать повторения одних и тех же вычислений (что обеспечивает эффективность).

Алгоритмы динамического программирования определяются рекурсивными алгоритмами/функциями, решающими целую проблему через решения более мелких проблем.

Чтобы обеспечить эффективность такого алгоритма, нам требуется сохранять достаточно информации, позволяющей избежать повторения уже сделанных вычислений. Почему поиск в глубину эффективен? Потому что мы маркируем найденные вершины, чтобы не посещать их снова. Почему грубый перебор с возвратом требует больших вычислительных мощностей? Потому что он перебирает все возможные пути/решения, вместо того чтобы рассматривать только те варианты, которые мы не видели раньше.

Динамическое программирование - это методика *эффективной реализации рекурсивных алгоритмов через сохранение промежуточных результатов*. Хитрость в том, чтобы понять, что простой рекурсивный алгоритм рассчитывает одни и те же подзадачи снова, снова и снова. Но если так, то, чтобы повысить эффективность алгоритма, мы можем сохранять полученные результаты в таблице, а не считать все заново. Только после того как у вас есть правильно работающий алгоритм, имеет смысл пытаться его ускорить через матрицу решений.

Примеры 8.1 и 8.2 демонстрируют один из способов реализации динамического программирования. Вам также рекомендуется самостоятельно ознакомиться с такими методами решения задач как метод оптимизации Ньютона, метод деформируемого многогранника и другие методы численной оптимизации.

**Пример 8.1: Стоимость редактирования**

Задача нахождения шаблонов в строках текста имеет очень большое значение, и ранее мы рассматривали соответствующие алгоритмы. Но там мы ограничились исследованием точного совпадения строк, то есть поиском позиций, где шаблонная строка s в точности содержится в тестовой строке р. На практике все часто совсем не так просто. Возможность ошибок при написании теста или шаблона лишает нас возможности жесткой проверки на идентичность. Эволюционные изменения входных данных также легко могут привести к тому, что мы будем искать устаревшие шаблоны.

Если нам предстоит работать с неточным совпадением, то первым делом нам нужно задать функцию стоимости, определяющую, насколько различаются две строки. При измерении расстояния сводится к минимуму стоимость изменений, которые необходимо сделать, чтобы преобразовать одну строку к другой. Существует три естественных типа изменений.

• Замещение. Замена одного символа шаблона *s* на отличный от него символ текста *t*, к примеру, переход от «shot» к «spot».

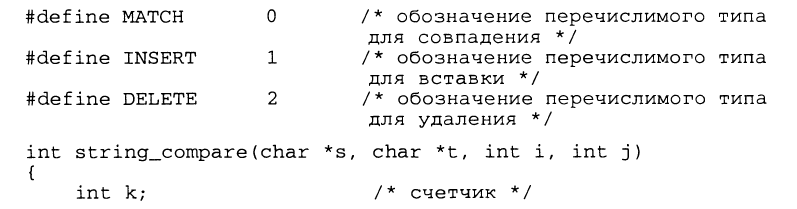
• Вставка. Вставка одного символа в шаблон s так, чтобы он совпал с текстом,

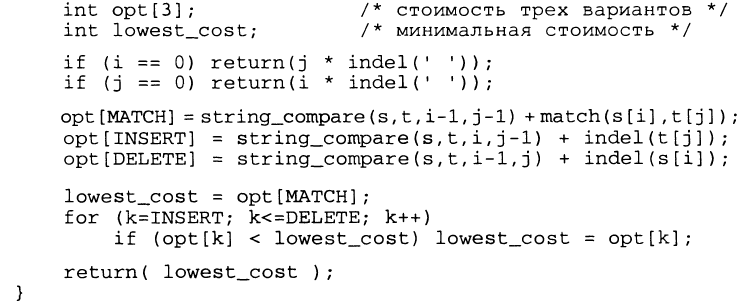
к примеру, переход от «gogle» к «google».

• Удаление. Удаление одного символа из шаблона s так, чтобы он совпадал с текстом t, к примеру, переход от «hour» к «our».

Чтобы мы могли говорить об идентичности строк, мы должны задать стоимость каждой операции преобразования строк. Если считать стоимость каждой операции равной единице, мы опишем стоимость редактирования (edit distance) между двумя строками (другие варианты стоимости также могут привести к интересным результатам).

Но как мы можем найти стоимость редактирования? Мы можем разработать рекурсивный алгоритм на основании того наблюдения, что последний символ строки при последней операции редактирования должен быть либо вставлен, либо заменен, либо удален. Отбрасывая символы, использованные в последней операции, мы приходим к паре меньших строк. Пусть i и j - это последние символы значащих префиксов s и t соответственно. После последней операции могут получиться три пары меньших строк, соответствующие замене, вставке и удалению. Если бы мы знали стоимость редактирования трех пар меньших строк, мы могли бы выбрать вариант, ведущий к решению наименьшей стоимости. Мы можем найти их стоимости, используя рекурсию:

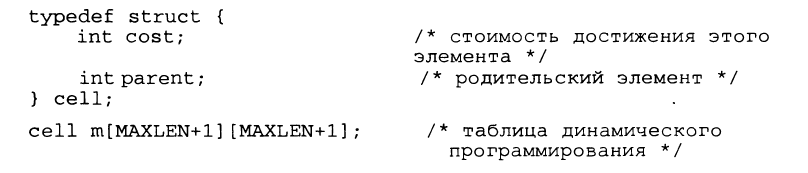




Программа работает абсолютно верно, но медленно. Почему же алгоритм работает так медленно? Ему требуется экспоненциальное время, потому что он вычисляет одни и те же величины снова, снова и снова. На каждом символе строки рекурсия разделяется на три ветви, что значит, что скорость разрастания, по меньшей мере, 3n. На самом деле, скорость еще больше, так как большинство вызовов уменьшает только один из двух индексов, а не оба одновременно.

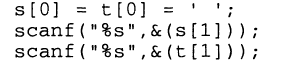
Как сделать этот алгоритм пригодным для практического использования? Важно заметить, что большинство рекурсивных вызовов заново считают то, что уже было вычислено ранее. Откуда это можно понять? Число возможных уникальных рекурсивных вызовов - |s| x |t|, так как именно столько различных пар (i, j) могут передаваться в качестве параметров вызова. Сохранив в таблице значения для каждой такой пары, мы сможем избежать повторного вычисления уже известных величин.

Ниже приводится реализация этого алгоритма на основе динамического программирования (с использованием таблиц). Таблица реализуется двумерной матрицей m, в которой каждая из |s| x |t| ячеек содержит стоимость соответствующей подоперации и указатель на родительский элемент, с помощью которого можно понять, как мы пришли к этой ячейке.

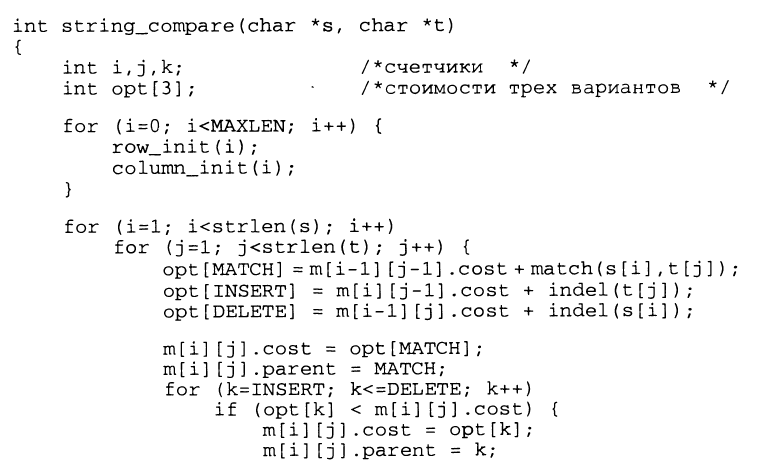


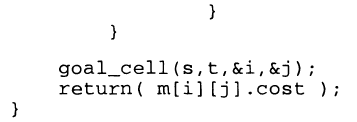
Между версией, основанной на динамическом программировании, и версией, основанной на рекурсии, существуют три отличия. Во-первых, промежуточные значения получаются через обращение к таблице, а не через рекурсивные вызовы. Во-вторых, поле parent обновляется для каждой ячейки, что позволит нам позже восстановить последовательность редактирования. В-третьих, благодаря функции goal\_cell() мы можем сделать больше, чем просто вывести m[| s |] [| t |].cost. Это позволит нам применять подпрограмму для более широкого класса задач.

Обратите внимание, что в дальнейших подпрограммах мы придерживаемся определенных нестандартных соглашений по строкам и индексам. В частности, мы полагаем, что каждая строка начинается с дополнительного пробела, так что первый значимый символ строки s находится в s [1]:



Зачем это делать? Так мы сможем использовать одинаковые индексы и для строк, и для матрицы m. Не забывайте, что в крайнем ряду и крайней строке матрицы m мы должны хранить граничные значения, соответствующие пустому префиксу. С другой стороны, мы могли не трогать входные строки, а просто соответствующим образом подправить индексы.





Обратите внимание на порядок, в котором заполняются элементы матрицы. Чтобы определить значение элемента (i,y), нам требуется знать значения трех элементов, а именно (i-1,j-1), (i,j-1) и (i-1,j). В принципе подойдет любой порядок заполнения, обладающий этим свойством, например использованный нами порядок заполнения по строкам.

Ниже приводится пример работы программы, в котором за пять шагов «thou shalt not» преобразуется в «you should not» (показаны значения стоимости и родительские элементы).

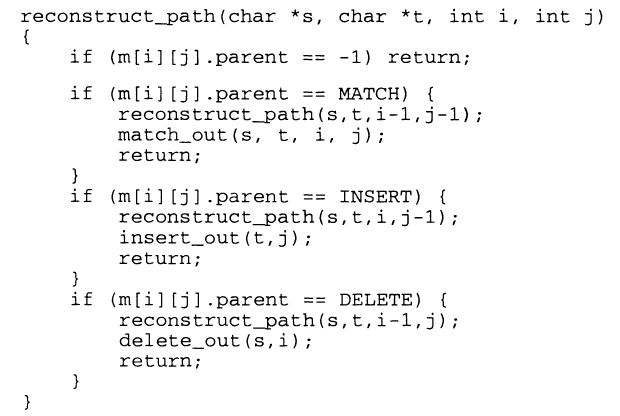


Реализация на основе динамического программирования, приведенная выше, возвращает стоимость оптимального решения, но не само решение. Замечательно, когда ты знаешь, что от «thou shalt not» к «you should not» можно перейти за пять шагов, но как именно выглядят эти шаги?

Возможные решения данной задачи описываются путями через матрицу динамического программирования. Путь начинается исходной конфигурацией (пара пустых строк (0,0)) и заканчивается итоговым конечным состоянием (пара полных строк (|s|,|t|)). Ключ к построению решения лежит в восстановлении операций, использованных на каждом шаге пути, приведшего к оптимальному решению. Эти операции сохранялись в поле parent каждого элемента массива.

Восстановление проводится на основании указателя parent. Мы начинаем с итогового элемента и идем назад, пока не дойдем до начальной конфигурации. Поле parent элемента m [ i, j ] позволяет нам узнать тип преобразования для (i,j) - MATCH, INSERT или DELETE. Последовательность преобразований перехода от «thou shalt not» к «you should not»: DSMMMMMISMSMMMM, то есть мы удаляем первую «t», заменяем «h» на «у», следующие пять символов совпадают, поэтому мы их не трогаем, затем мы вставляем «о», заменяем «а» на «u» и заменяем «t» на «d».

Если начинать с итогового элемента, то последовательность операций будет восстановлена в обратном порядке. Правда, умное использование рекурсии может сделать все за нас.



Во многих задачах, включая поиск стоимости редактирования, переход можно восстановить без явного использования массива последних преобразований (построения полной матрицы как приведено на рисунке выше). Можно просто идти назад и смотреть, какое из преобразований могло привести к текущему элементу так, чтобы в итоге получилась заданная стоимость.

**Пример 8.2. Оптимизация лифта**

Я работаю в очень высоком здании с очень медленным лифтом. Особенно меня раздражает, когда люди нажимают кнопки нескольких соседних этажей (скажем, 13, 14 и 15-го), а я еду с нижнего этажа на верхний. Моя поездка наверх прерывается трижды, по разу на каждом из этажей. Было бы гораздо вежливее со стороны этих людей, если бы они нажали только 14-й, а люди с 13-го и 15-го этажей прошли бы по одному этажу пешком. В любом случае кто-то из них мог бы пройтись.

Наша задача состоит в написании программы оптимизации лифта. Желаемый этаж каждого из пассажиров задается в начале поездки. Далее лифт должен решить, на каких этажах он будет останавливаться. Число остановок лифта не должно превышать k, при этом этажи нужно выбрать так, чтобы свести к минимуму полное число этажей, которое нужно пройти людям вверх или вниз. Можно считать, что лифт знает, сколько человек выходит на каждом этаже.

Мы полагаем, что стоимость подъема на один лестничный пролет равняется стоимости спуска; не забывайте, что у этих людей есть возможность поупражняться. Если однозначного решения не существует, руководство предлагает отдавать предпочтение тем случаям, когда лифт останавливается на минимально возможном этаже, так как на это требуется меньше электричества. Обратите внимание, что лифт вовсе не обязан останавливаться на этажах, указанных пассажирами. Если пассажиры указали 27 и 29 этажи, то лифт вместо этого может остановиться на 28.

Это пример обычной задачи на программирование/алгоритмы, которая прекрасно решается с помощью динамического программирования. Как мы до этого додумались и что делать дальше?

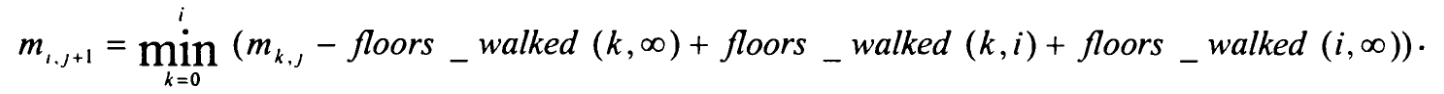
Вспомните, что алгоритмы динамического программирования основываются на рекурсивных алгоритмах. Решение о том, где лучшего всего сделать k-ю остановку, зависит от стоимости всех возможных решений для k-1 остановки. Если вы сможете сказать мне стоимость лучшего из частных решений, то я смогу принять верное решение насчет последней остановки.

Эффективные алгоритмы динамического программирования нередко требуют упорядоченных входных данных. Нам важен тот факт, что требования пассажиров можно упорядочить от наименьшего этажа к наибольшему. К примеру, пусть лифт сначала остановился на этаже F1, а затем на этаже F2. Вторая остановка может не иметь никакого значения для пассажира, который хочет попасть на этаж F1 или ниже. Это значит, что задачу можно разбить на части. Если мне нужно добавить третью остановку F3 выше F2, то для выбора ее позиции мне не нужно ничего знать о F1.

Мы почувствовали, что задача имеет отношение к динамическому программированию. Как выглядит алгоритм? Для начала нам нужно определить функцию стоимости для частных решений, которая позволит нам принимать более важные решения. Что вы скажете о такой:

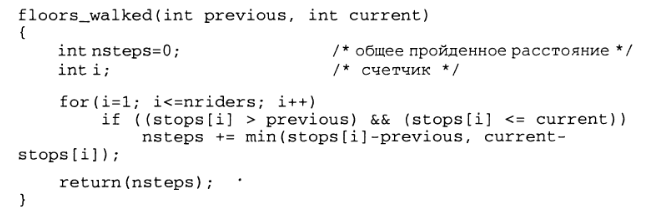
Пусть m [i] [j] задает минимальную стоимость обслуживания всех пассажиров, если делается ровно j остановок, последняя из которых делается на этаже i.

Может ли эта функция помочь нам разместить (j + 1)-ю остановку так, чтобы общая стоимость уменьшилась? Да. (j+1)-я остановка по определению должна располагаться выше j-й остановки на этаже i. Значит, новая остановка будет интересовать только тех пассажиров, которые хотят выйти выше i-го этажа. Чтобы понять, как это может нам помочь, мы должны правильно разделить пассажиров между новой остановкой и i на основании того, к какой остановке ближе они находятся. Эта идея приводит к заданию следующей рекуррентной последовательности:

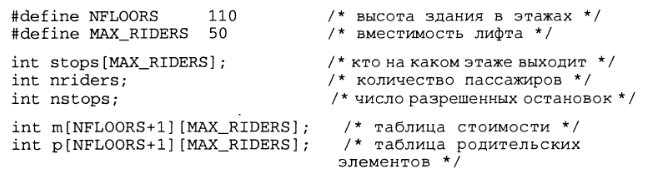


Что она значит? Если последняя остановка совершается на этаже i, то предыдущая должна быть на этаже k < i. Какова стоимость решения в таком случае? Мы должны вычесть из mk,j, стоимость обслуживания всех пассажиров выше k (то есть floors\_walked(k, ∞)) и заменить ее (предположительно) уменьшенной стоимостью добавления остановки на этаже i (то есть floors\_walked(k, i) + floors\_walked(i, ∞)).

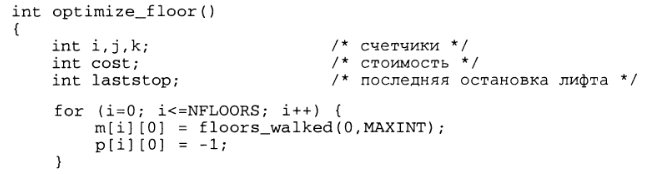
Основной здесь является функция floors\_walked (а, b). Она подсчитывает полное число этажей, проходимых пассажирами, которые направляются на этажи, лежащие между двумя последовательными остановками а и b. Каждый такой пассажир идет к своему этажу кратчайшим путем:

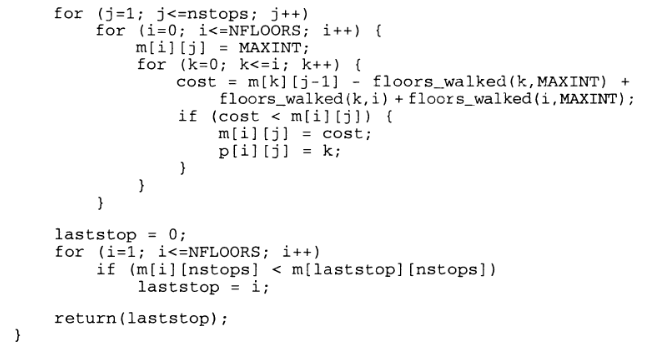


Как только вы ухватили основную идею, реализация алгоритма становится очевидной. Мы создаем глобальные матрицы для хранения таблиц динамического программирования. В одной из таблиц хранятся стоимости, в другой - родительские элементы:

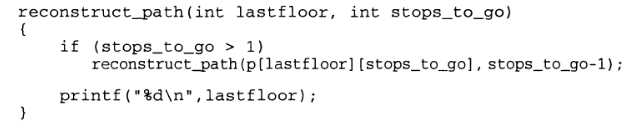


Функция оптимизации - это непосредственная реализация рекуррентного соотношения. Особое внимание уделяется такому расположению циклов, чтобы все величины были подсчитаны к тому моменту, когда они могут понадобиться.





Наконец, нам нужно восстановить решение. Идея та же, что и в предыдущих примерах: нужно идти по указателям на родительские элементы в обратную сторону.



Будем считать нижний этаж нулевым. Для случая, когда по одному пассажиру хочет выйти на каждом этаже от 1-го до 10-го, и разрешена всего одна остановка, получаем, что лучше всего остановиться на 7-м этаже, суммарная стоимость - 18 лестничных пролетов (пассажиры, направляющиеся на этажи 1, 2 и 3-й, должны с нулевого этажа идти пешком). Если разрешено две остановки, то лучше всего остановиться на этажах 3-м и 8-м, стоимость - 11, а лучшие три остановки - это 3,6 и 9-й этажи, приводящие к суммарной стоимости в 7 пролетов.

### Задача 8.1 (Веса и меры)

РС/UVaIDs: 111 103/10154

Черепашка по имени Мак очень боится, что его панцирь может сломаться. Поэтому он попросил вас дать совет, как поставить черепашек друг на друга, чтобы построить трон Йертла Черепахи. Каждая из 5607 черепах, призванных Йертлом, обладает своей силой и весом. Ваша задача состоит в том, чтобы составить из них башню максимальной высоты.

**Входные данные**

Входные данные состоят из нескольких строк, каждая из которых содержит два целых числа, разделенные одним или более пробелом. Эти числа задают вес и силу черепахи. Вес черепахи измеряется в граммах. Сила, которая также измеряется в граммах - это максимальный вес, который способна выдержать черепаха (включая свой собственный). Таким образом, черепаха, весящая 300 грамм и имеющая силу 1000 грамм, может держать 700 грамм на спине. Число черепах не превышает 5607.

**Выходные данные**

Выходные данные должны состоять из одного целого числа, равного максимальному числу черепах, которые можно поставить стопкой так, чтобы сила ни одной из них не была превышена.

**Пример входных данных**

300 1000 1000 1200 200 600 100 101

**Соответствующие выходные данные**

3

### Задача 8.2 (Умнее ли больший)

PC/UVaIDs: 111101/10131

Некоторые люди считают, что чем слон больше, тем он умнее. Чтобы опровергнуть это утверждение, вы хотите проанализировать набор слонов и расположить наибольшее из возможных подмножеств этого набора в последовательность слонов с увеличивающимся весом, но уменьшающимся IQ.

**Входные данные**

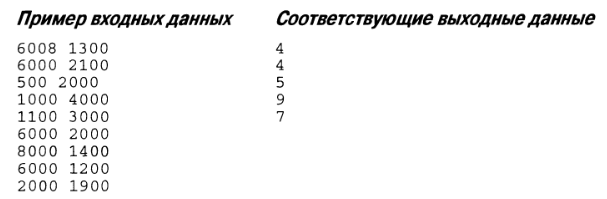
Входные данные состоят из данных о наборе слонов, по одному слону на строку, и завершаются меткой конца файла. Данные о каждом конкретном слоне состоят из двух целых чисел: первое - это вес слона в килограммах, второе - его IQ. Оба числа лежат в пределах от 1 до 10 000. Во входных данных содержится информация не более чем о 1000 слонов. У двух слонов могут быть одинаковыми: вес, IQ или и вес, и IQ.

**Выходные данные**

Первая строка выходных данных должна содержать целое число n - длину найденной последовательности слонов. Каждая из оставшихся n строк должна содержать единственное целое число, задающее слона. Обозначим числа на i-й строке входных данных как W[i] и S[i]. Если ваша последовательность состоит из n слонов a[1], а[2],..., а[n], то должны выполняться условия:



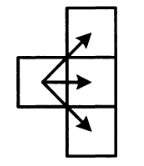
Все неравенства строгие: веса должны строго возрастать, a IQ должны строго уменьшаться. Программа может вывести любой верный ответ из возможных для заданных входных данных.



### Задача 8.3 (Однонаправленная задача коммивояжера)

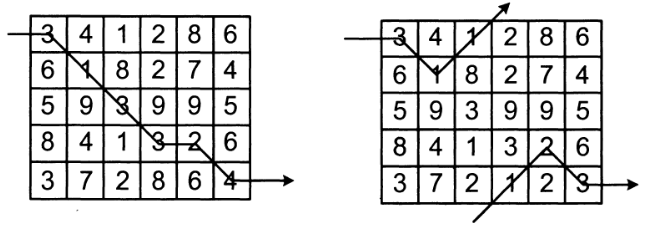
РС/UVaIDs: 111104/116

Для заданной матрицы размером m х n вы должны написать программу, которая вычисляет путь с минимальным весом с левого края матрицы до правого. Путь начинается в любой строке первого столбца и состоит из последовательности шагов, обрывающихся в столбце n. Каждый шаг состоит в переходе из столбца i в столбец i + 1 в соседнюю (по горизонтали или диагонали) ячейку. Первая и последняя строки (строки 1 и m) матрицы считаются соседними; то есть матрица «сворачивается», образуя горизонтальный цилиндр. Разрешенные шаги показаны ниже.



Вес пути рассчитывается как сумма целых чисел, записанных в каждой из n посещенных ячеек.

Пути с минимальным весом через две немного отличные матрицы размером 5x6 показаны ниже. Значения матриц отличаются только в нижнем ряду. Путь, показанный для правой матрицы, использует соседство первого и последнего рядов.



**Входные данные**

Входные данные состоят из последовательности описаний матриц. Каждая матрица задается строкой с количеством строк и столбцов, которые обозначаются m и n соответственно. Далее следуют m х n целых чисел, идущих по строкам; то есть первые n целых чисел составляют первую строку матрицы, следующие n целых чисел составляют вторую строку и т. д. Целые числа, стоящие на одной строке, отделены друг от друга одним или более пробелами. Обратите внимание: целые числа не обязаны быть положительными. Во входном файле могут быть заданы одна или более матриц. Входные данные завершаются маркером конца файла.

Для каждого описания число рядов составляет от 1 до 10 включительно; число столбцов - от 1 до 100 включительно. Вес любого пути не превышает целого числа, для хранения которого потребуется больше 30 бит.

**Выходные данные**

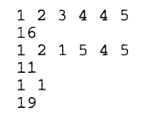
Для каждого описания матрицы требуется вывести две строки. Первая строка задает путь минимальной стоимости, а вторая - собственно стоимость этого пути. Путь состоит из последовательности n целых чисел (разделенных одним или более пробелами), задающих ряды, из которых состоит путь минимальной стоимости. Если путей минимальной стоимости больше одного, то должен быть выведен лексикографически меньший путь.

**Пример входных данных**





**Соответствующие выходные данные**



### Задача 8.4 (Распил)

PC/UValDs: 111105/10003

Вам нужно распилить деревянный брус на несколько кусков. Самая удобная компания Analog Cutting Machinery (ACM) берет плату за пилку в зависимости от размера бруса, который нужно распилить.

Легко понять, что различные заказы приводят к различным ценам. Например, рассмотрим брус длиной 10 м, который необходимо распилить на расстоянии 2, 4 и 7 м считая от одного конца. Это можно сделать несколькими способами. Можно распилить сначала на отметке 2 м, потом 4 и потом 7 м. Это приведет к стоимости: 10 + 8 + 6 = 24, потому что сначала длина бруса, который пилили, была 10 м, затем она стала 8 м, и наконец, 6 м. А можно распилить иначе: сначала на отметке 4 м, затем 2 и затем 7 м. Это приведет к стоимости: 10 + 4 + 6 = 20, что для нас лучше.

Ваш начальник требует, чтобы вы написали программу, которая находит минимальную стоимость распила для любого бруса заданного размера.

**Входные данные**

Входные данные состоят из нескольких тестовых блоков. Первая строка каждого тестового блока содержит положительное число А, задающее длину бруса, который нужно распилить. Вы можете считать, что А < 1000. Следующая строка содержит число Б (Б < 50) распилов, которые нужно сделать.

Следующая строка содержит С положительных чисел Сi (0 < Сi < A ), задающих места, в которых необходимо сделать распилы, в строго возрастающем порядке. Входной блок с A = 0 задает окончание входных данных.

**Выходные данные**

Выведите стоимость пилки с минимальной ценой в формате, приведенном ниже.

**Пример входных данных**

100

3

25 50 75

10

4

4 5 7 8

0

**Соответствующие выходные данные**

The minimum cutting price is 200.

The minimum cutting price is 22.

### Задача 8.5 (Заполнение парома)

PC/UValDs: 111106/10261

Паромы используются для перевозки машин через реки или другие водные препятствия. Обычно паромы имеют достаточно большую ширину, поэтому машины на них паркуют в две линии.

Имеется одна очередь из машин, заезжающих на паром, и оператор направляет машины из очереди на правый или левый борт парома так, чтобы нагрузка была сбалансирована. Все машины в очереди имеют разную длину, которую оператор заранее узнает, осматривая очередь. На основании своих наблюдений и длины парома оператор решает, на какую сторону должны заезжать машины, чтобы загрузить паром максимальным числом машин из очереди. Напишите программу, которая будет указывать оператору, какую машину на какую сторону отправлять так, чтобы поместилось максимальное число машин.

**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка.

Первая строка каждого тестового блока содержит одно целое число от 1 до 100 -длину парома (в метрах). Для каждой машины в очереди имеется своя строка во входных данных, задающая длину машины в сантиметрах - целое число от 100 до 3000 включительно. Последняя строка входных данных содержит 0. Машины должны загружаться по порядку, а также с учетом того факта, что суммарная длина машин но каждому борту не должна превышать длину парома. Должно быть загружено как можно больше машин, начиная с первой и далее по порядку до тех пор, пока вы не сможете загрузить очередную машину.

Между двумя последовательными блоками входных данных находится пустая строка.

**Выходные данные**

Для каждого тестового блока первая строка выходных данных должна содержать число машин, которые могут быть загружены на паром. Для каждой машины, которая может быть загружена на паром, в порядке их следования во входных данных выведите строку, содержащую "port", если машина должна быть направлена на левую сторону и "starboard", если машина должна быть отправлена на правую сторону. Если под приведенные условия подходит более одного варианта расположения машин, подойдет любой.

Выходные данные для двух последовательных тестовых блоков должны быть разделены пустой строкой.

**Пример входных данных**

1

50

2500

3000

1000

1000

1500

700

800

0

**Соответствующие выходные данные**

6

port

starboard

starboard

starboard

port

port

### Задача 8.6 (Палочка По)

PC/UValDs: 111107/10271

В Китае для еды используют две палочки, но панда По – Воин Дракона, поэтому он использует набор из трех палочек, двух обычных и еще одной дополнительной; это длинная палочка, на которую он насаживает особенно большие куски еды. Длины двух коротких палочек должны совпадать как можно с большей точностью, а длина дополнительной неважна до тех пор, пока она самая длинная. Для набора палочек с длинами А, В, С (А ≤ В ≤ С) функция (А - В)2 определяет «ошибку» набора.

По пригласил *К* гостей на вечеринку по случаю дня Воина Дракона, и на ней он хочет представить свой способ использования палочек. Ему нужно приготовить *К* + 8 наборов палочек (для себя, тигрицы, богомола, змеи, журавля, обезьяны, мастера Шифу, мастера Угвея и К других гостей). Но все палочки, которые у него есть, имеют разную длину! Зная эти длины, он должен составить К + 8 наборов так, чтобы суммарная «ошибка» всех наборов была минимальна.

**Входные данные**

Первая строка входных данных содержит одно целое число Т, задающее число тестовых блоков (1 ≤ Т ≤ 20). Каждый тестовый блок начинается с двух целых чисел К и N (0 ≤ К ≤ 1000, 3К + 24 ≤ N ≤ 5000), задающих число гостей и число палочек. Далее в неубывающем порядке идут N положительных целых чисел Li, которые задают длину палочек (1 ≤ Li ≤ 32 000).

**Выходные данные**

Для каждого тестового блока входных данных выведите строку, содержащую минимальную суммарную «ошибку» всех наборов.

**Пример входных данных**

1

1 40

1 8 10 16 19 22 27 33 36 40 47 52 56 61 63 71 72 75 81

81 84 88 96 98 103 110 113 118 124 128 129 134 134

139  148 157 157 160 162 164

**Соответствующие выходные данные**

23

Замечание. Возможный вариант составления девяти наборов для этого примера: (8,10, 16), (19,22,27), (61,63,75), (71,72,88), (81,81,84), (96,98, 103), (128,129,148), (134, 134, 139) и (157, 157, 160).

### Задача 8.7 (Приключения в дороге)

PC/UValDs: 111108/10201

Вы собираетесь арендовать грузовик для бездорожья, чтобы вернуться из Ухрюпинска в большой город. Но цены на бензин тут так высоки, что вы решили узнать, сколько топлива потребуется скормить по дороге этому прожорливому зверю.

Грузовику требуется один литр бензина на каждый пройденный километр. Он оснащен 200-литровым баком. Когда вы арендуете грузовик в Ухрюпинске, бак заполнен наполовину. Когда вы будете его возвращать в городе, бак также должен быть заполнен по крайней мере наполовину, или компания «Ухрюпинск-комфорт», которая сдала вам автомобиль в аренду, сдерет с вас еще больше денег. Ваша цепь - потратить на горючее как можно меньше денег, но при этом не оказаться посреди дороги без бензина.

**Входные данные**

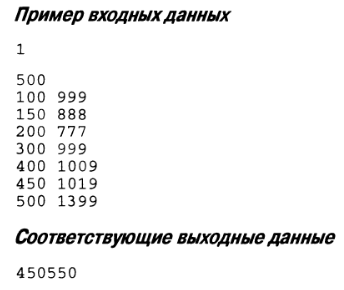
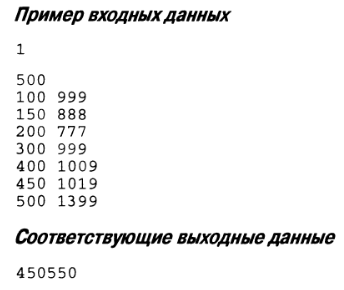
Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков, за которой следует пустая строка.

Каждый тестовый блок содержит только целые числа. Первое число - это расстояние от Ухрюпинска до города, не превышающее 10 000. Далее идет набор максимум из 100 описаний заправочных станций, задающих все заправки вдоль вашего пути в неубывающем по расстоянию порядке. Каждое описание состоит из расстояния в километрах от Ухрюпинска до заправочной станции и цены одного литра горючего на этой заправочной станции, в десятках крышек. Цена не превышает 2000 крышек за литр. Между двумя последовательными блоками входных данных находится пустая строка.

**Выходные данные**

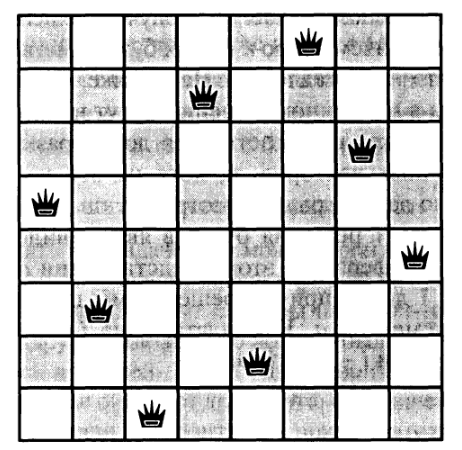
Для каждого тестового блока выведите минимальное количество денег, которое вы должны истратить на горючее в дороге от Ухрюпинска до города. Если не существует возможности добраться до Ухрюпинска, удовлетворив условиям, приведенным выше, то выведите "Donna try liv Uhryupinsk".

Выходные данные для двух последовательных тестовых блоков должны быть разделены пустой строкой.



## Приложение 4.1. Задача восьми ферзей

Задача восьми ферзей - это классическая головоломка, в которой требуется расставить восемь ферзей на шахматной доске размером 8x8 так, чтобы они не били друг друга. Это значит, что ни на одной вертикали, горизонтали или диагонали не могут находиться два ферзя одновременно, что показано на рис. 8.1. В течение многих лет эта задача изучалась многими известными математиками, включая Гаусса, а также огромным числом не таких известных людей, изучавших основы программирования.



В задаче нет ничего, что запрещало бы рассмотрение чисел больших восьми. В задаче «N ферзей» спрашивается, сколькими способами можно разместить n ферзей на шахматной доске размером n х n так, чтобы ни один из них не атаковал другого. Даже для среднего n количество решений настолько велико, что их становится неинтересно выводить. Но для каких значений n мы можем найти их число за умеренное количество времени?

Чтобы понять суть задачи такого типа, обычно требуется построить решения для небольших случаев вручную. Очевидно, что для n = 2 решений не существует, так как второй ферзь будет атаковать первого по вертикали, горизонтали или диагонали. Для n = 3 вариантов уже больше, но методом проб и ошибок вы должны понять, что и в этом случае решений не существует. Мы попробуем построить решение для n = 4 - нетривиального случая наименьшего размера.

Для использования backtrack-поиска нам нужно аккуратно выбрать самый краткий и эффективный способ представления наших решений в виде вектора. Как можно представить решение задачи «n ферзей» и насколько большим может быть это представление?

Наиболее прямолинейным представлением было бы эмулировать наш генератор подмножеств и использовать вектор решений, в котором ai принимало бы значение истина в том и только в том случае, если на i-й клетке стоит ферзь. Для этого каждой клетке требуется присвоить уникальное имя от 1 до n. Набор возможных решений содержит для i-й клетки истину, если ни один из ранее поставленных ферзей не бьет эту клетку, и ложь - в противном случае. Мы получим решение после того, как будут заполнены все n клеток, причем ровно n из них будут иметь значение истина.

Подойдет ли нам это представление? Не похоже, что оно будет кратким, так как почти все элементы в найденном решении будут иметь значение ложь. Также из этого следует его высокая стоимость. Для доски размером 8x8 существует 264 = 1.84 х 1019 различных векторов, и хотя не все из них будут полностью построены, на это число даже страшно смотреть.

А что, если i-и элемент решения будет в явном виде содержать клетку, на которой располагается i-и ферзь? В этом представлении аi будет целым числом, имеющим значение от 1 до n2, причем решение мы получим в том случае, если заполним первые n элементов а. Кандидатами на i-е место будут все клетки, которые не бьются первыми i-1 ферзями.

Лучше ли это представление предыдущего? При использовании этого представления для доски размером 8x8 существует «всего» 648 =2.81 х 1014 векторов. Мы существенно снизили сложность задачи, но нам еще далеко до поискового пространства порядка 106 на котором наши возможности уже подходят к своему пределу. Чтобы поиск с возвратом заработал, нам придется отбрасывать или *отсекать* большую часть вариантов еще до того, как они построены.

Экспоненциальный рост поисковых пространств с увеличением числа вариантов носит название ***комбинаторного взрыва***. Таким образом, даже задачи среднего размера быстро переходят ту грань, за которой они уже не могут быть решены за разумное количество времени. Чтобы использовать алгоритм поиска с возвратом для решения интересных задач, мы должны сужать поисковое пространство, отсекая каждую ветвь поиска в тот момент, когда мы понимаем, что она не может привести к правильному решению.

Термин *отсекать* в данном случае подходит как нельзя больше. Рекурсивные вызовы backtrack определяют дерево. Когда мы отсекаем лишние ветви, понимая, что на самом деле множество кандидатов на место является пустым, мы уходим от неконтролируемого роста.

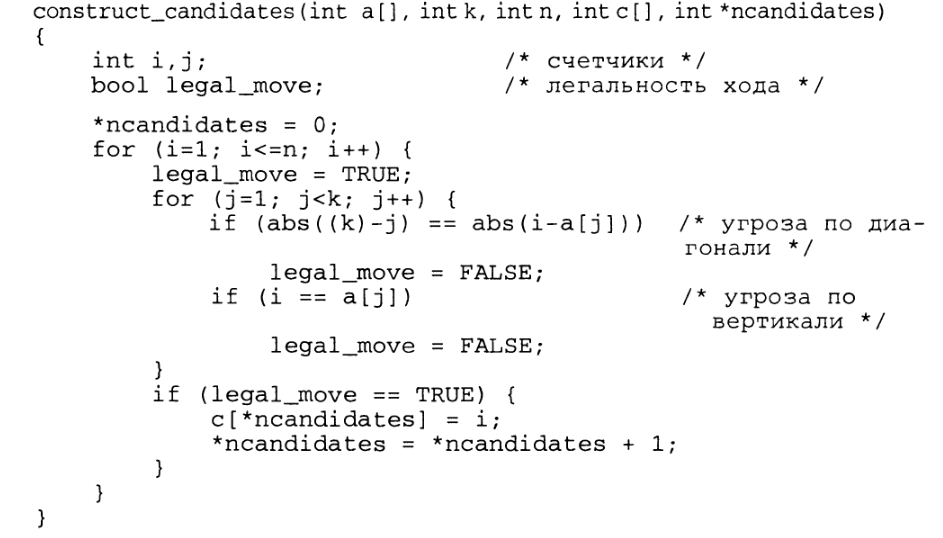
Так как же мы можем использовать поиск с отсечением вариантов для представления позиции, приведенного выше? Во-первых, мы можем избавиться от симметричных случаев. В случаях, рассматриваемых нами ранее, нет никакой разницы между ферзем, находящимся на первом месте вектора (а1), и ферзем, находящимся во второй позиции (а2). Если все так и оставить, то каждое решение будет построено 8! = 40 320 раз. Это можно легко исправить, если рассматривать только те случаи, в которых ферзь в аi находится на клетке с большим номером, чем ферзь в аi-1.

Это простое изменение уменьшит количество вариантов до (648) = 4.426 x 109.

Если еще поразмышлять над задачей, то можно прийти и к лучшему представлению. Обратите внимание, что в решении задачи n ферзей на одну горизонталь должен приходиться ровно один ферзь. Почему? Если бы на какой-то горизонтали не было ферзей, тогда на какой-то другой обязательно должно было бы быть два ферзя, чтобы в сумме у нас получилось n ферзей. Но на одной горизонтали не может быть два ферзя, потому что в этом случае они бьют друг друга. Ограничение кандидатов для i-го ферзя до 8 клеток i-и горизонтали приводит нас к поисковому пространству размером 88= 1.677 х 107, что является хоть и большой, но уже доступной величиной.

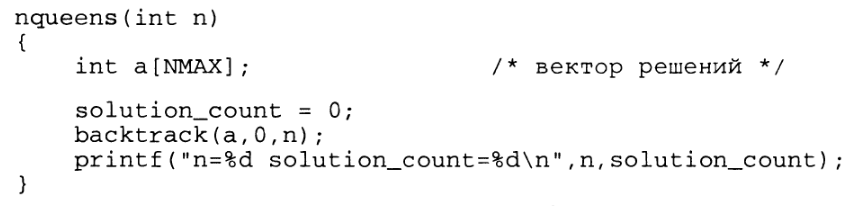
Но мы можем еще больше упростить задачу. Так как никакие два ферзя не могут находиться на одной вертикали, то мы получаем, что n вертикалей конечного решения должны образовывать перестановку n. Избегая повторяющихся элементов, мы уменьшаем наше поисковое пространство до 8! =40 320, что представляет совсем небольшую работу для любой более-менее быстрой машины.

Самой важной подпрограммой является построение кандидатов. Мы повторно проверяем, бьется ли k-я клетка данной горизонтали одним из ранее поставленных ферзей. Если да, то идем дальше, но если же нет, то мы включаем ее в качестве возможного кандидата.



Оставшиеся подпрограммы достаточно просты, так как нам нужно только посчитать решения, а не выводить их.





Итоги вычислений:

n=l solution\_count=l

n=2 solution\_count=0

n=3 solution\_count=0

n=4 solution\_count=2

n=5 solution\_count=10

n=6 solution\_count=4

n=7 solution\_count=40

n=8 solution\_count=92

n=9 solution\_count=352

n=10 solution\_count=724

n=11 solution\_count=2 680

n=12 solution\_count=14 200

n=13 solution\_count=73 712

n=14 solution count=365 596

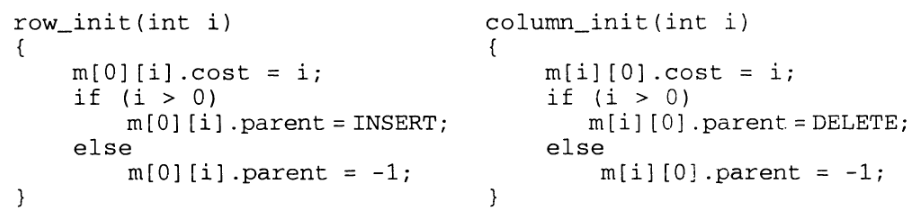
Более эффективные программы могли бы продвинуться чуть дальше. Можно увеличить быстродействие при построении кандидатов, если обрывать внутренний цикл for сразу же как булевская переменная примет значение «ложь». Еще больше времени можно сэкономить, если использовать дальнейшее отсечение вариантов. К текущей реализации мы возвращаемся, как только на k-й горизонтали нет легальных ходов. Но если на какой-то последующей горизонтали (скажем, (k + 2)-й) нет легальных ходов, то все, что мы делаем на k-й, лишено смысла. Чем раньше мы это поймем, тем лучше.

Мы могли бы попробовать извлечь больше выгоды из симметрии. Поворот любого решения на 90° дает нам другое решение, так же как и симметрия относительно центра доски. Рассматривая только одно решение из каждого класса эквивалентности, мы могли бы в значительной степени понизить вычислительную нагрузку.

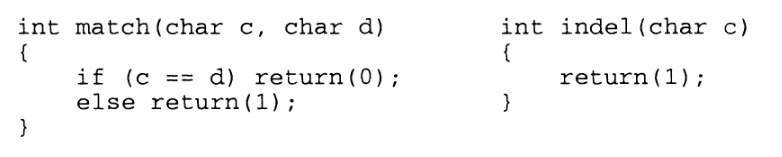
## Приложение 4.2. Дополнение к примеру 8.1

В процедурах оптимизации и восстановления пути использовались некоторые функции, которые мы до этого не определили. Их можно разбить по четырем категориям.

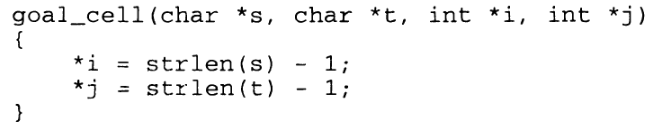
• Инициализация таблиц. Функции row\_init () и column\_init () инициализируют нулевую строку и столбец таблицы динамического программирования соответственно. В задаче о расстоянии редактирования ячейки (i, 0) и (0, i) соответствуют сравнению строк длины i со строками нулевой длины. Для преобразования требуется ровно i вставок/удалений, так что определение этих функций очевидно.



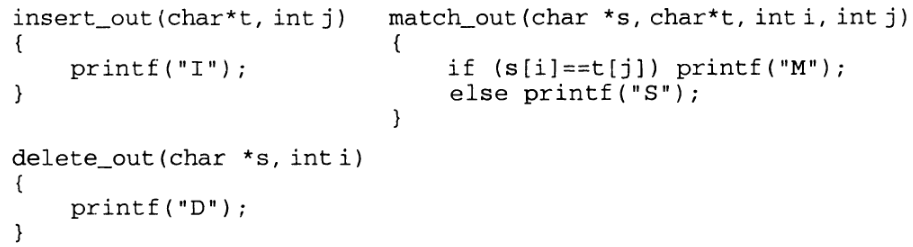
• Расчет стоимости. Функции match (с, d) и indel (с) определяют стоимость для преобразования символа с в символ d и для вставки/удаления символа с. Для простейшего варианта стоимость редактирования match должна возвращать 0, если символы идентичны, и 1 иначе, тогда как indel должна возвращать 1 вне зависимости от аргумента. Но, безусловно, можно использовать и более точные функции стоимости, которые, например, будут более мягко относиться к заменам, расположенным рядом для стандартных раскладок клавиатуры, или к таким, которые звучат или выглядят похоже.



• Обнаружение конечной ячейки. Функция goal\_cell возвращает индексы ячейки, задающей конечную точку решения. Для стоимости редактирования она определяется исходя из длины двух входных строк. Тем не менее другие приложения, как мы увидим ниже, могут не иметь строго определенной конечной точки.



• Возвращение назад. Функции match\_out, insert\_out и delete\_out отвечают за определенные действия для каждой операции редактирования при восстановлении пути. Для алгоритма стоимости редактирования такими действиями могут быть (в зависимости от требований к приложению) выведение совершенной на этом шаге операции редактирования или символа, задействованного на этом шаге.

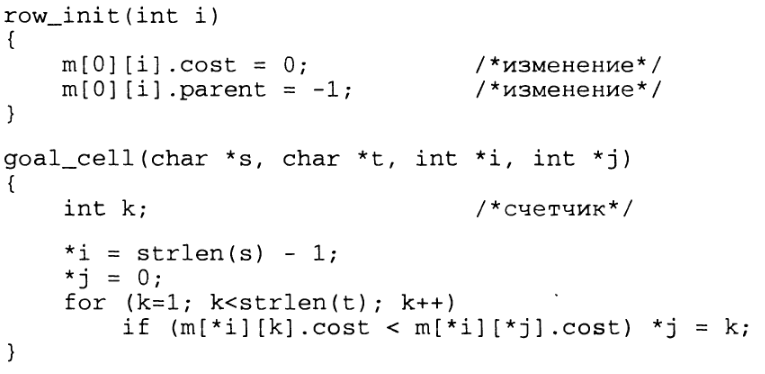


Для нашего случая вычисления стоимости редактирования все эти функции достаточно просты. Тем не менее мы должны признать, что корректная работа с граничными условиями и индексами ячеек весьма сложна. Хотя алгоритмы динамического программирования достаточно просто разрабатываются, как только вы поняли методику, понимание всех тонкостей требует аккуратных расчетов и полной проверки.

Как видно, для такой простой программы приходится разрабатывать достаточно сложную инфраструктуру. Тем не менее, внеся минимальные изменения в те функции, которые мы получили, можно решить несколько важных задач.

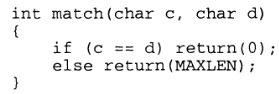
• Совпадение подстрок. Предположим, мы хотим найти позицию, где короткая строка s лучше всего совпадает с длинным текстом t. Скажем, мы ищем слово «Skiena» с учетом всех его неправильных написаний (Skienna, Skena, Skina и т. д.). Такой поиск с помощью наших обычных функций будет обладать малой чувствительностью, так как в каждой стоимости редактирования решающую роль будет играть стоимость удаления основной части текста.

Нам нужен такой вариант поиска расстояния редактирования, в котором стоимость совпадения не зависит от того, где это совпадение начинается, так чтобы совпадение в центре текста ничем не отличалось от совпадения в начале. Аналогично конечным состоянием не обязательно является конец обеих строк, это просто самое дешевое место, в котором достигается совпадение подстроки. Изменив две соответствующие функции, получим подходящее решение.



• Наибольшая общая подпоследовательность. Часто нам нужно найти наибольшую разрозненную строку символов, которая включается в каждое из двух слов. Наибольшая общая подпоследовательность (НОП) для слов «democrat» и «republican» - это еса.

Обычная подпоследовательность задается всеми совпадениями символов по прямой редактирования. Чтобы число таких совпадений было максимальным, мы должны отказаться от замены несовпадающих символов. Это достигается путем изменения функции расчета стоимости для совпадения:



На самом деле, вполне достаточно сделать стоимость замены большей, чем суммарная стоимость удаления и вставки, чтобы замена потеряла всякий смысл как операция редактирования.

• Наибольшая монотонная подпоследовательность. Числовая последовательность называется монотонно возрастающей, если i-й элемент, не меньше (i-l)-гo. В задаче о наибольшей монотонной подпоследовательности требуется удалить минимальное число символов из входной строки S так, чтобы оставшаяся строка представляла собой монотонно возрастающую подпоследовательность. К примеру, наибольшая возрастающая подпоследовательность для «243517698» это «23568».

Фактически наша задача сводится к задаче о наибольшей обшей подпоследовательности, если составить вторую строку из элементов S, отсортированных в порядке возрастания. Любая общая подпоследовательность этих двух строк будет (1) представлять символы в таком порядке, в каком они идут в S, и (2) использовать только символы, идущие в порядке возрастания в упорядоченной строке, значит, нам нужна просто самая длинная такая подпоследовательность. Конечно, такой подход можно легко модифицировать для случая убывающей подпоследовательности просто изменив направление сортировки.

Как вы могли заметить, наша простая функция для расстояния редактирования легко позволяет проделывать множество вещей. Трудность лишь в том, чтобы заметить, что ваша задача - это частный случай неточного совпадения строк.

# Лабораторная работа 5

Графовые алгоритмы

Задания

1. В соответствии со своим вариантом выберите по одной задаче из части 1 и части 2. Ознакомьтесь с рекомендациями по написанию кода, изложенными во введении, ознакомьтесь с теорией, приложениями, примерами кода.

2. Напишите программы, решающие полученные вами задачи.

3. Оформите отчет по ЛР.

## Часть 1. Обходы графов

В теории алгоритмов графы - это одна из универсальных тем, абстрактное представление, которым описывается организация транспортных систем, электрических цепей, взаимодействий людей и сетей телекоммуникации. То, что столь многие различные структуры могут быть смоделированы с использованием одного формализма, дает образованному программисту большое преимущество.

В первой части мы будем рассматривать только те задачи, которые требуют элементарного знания графовых алгоритмов, точнее, соответствующего использования графовых структур данных и алгоритмов обхода.

**Теория**

***Граф*** G = (V, Е) определяется множеством вершин V и множеством ребер Е, состоящим из упорядоченных или неупорядоченных пар из V. При моделировании дорожной сети вершины могут представлять города или перекрестки, некоторые из которых соединены дорогами/ребрами. При анализе исходного кода компьютерной программы вершины могут представлять строки кода, и если строка *у* может быть выполнена после строки *х*, то строки *х* и *у* соединяются ребром. При анализе социальных сетей вершины обычно обозначают людей, а ребра соединяют пары связанных личностей.

Графы обладают несколькими фундаментальными свойствами, влияющими на выбор структуры для представления графа и на алгоритмы, которые могут быть применены к графу. Первым шагом в каждой задаче на графы является определение особенностей графа, с которым предстоит работать.

• ***Ориентированный или неориентированный***. Граф *G* = (*V*, *Е*) называется *неориентированным*, если из того, что ребро (*х*, *у*) принадлежит *Е*, следует, что  
(*у*, *х*) также принадлежит *Е*. Если это не так, то граф называется ориентированным. Дорожная сеть между городами обычно неориентированная, так как по любой большой дороге можно ездить в обе стороны. Уличная сеть внутри городов практически всегда ориентированная, так как существуют *хотя бы несколько* улиц с односторонним движением. Графы программ обычно ориентированные, так как выполнение идет построчно и его направление меняется только в местах ветвления. Большинство графов, представляющих теоретический интерес, неориентированные.

• ***Взвешенный или невзвешенный***. Граф называется взвешенным, если каждому ребру (или вершине) G присвоено *численное значение* или вес. Обычными весами ребер графа дорожной сети, в зависимости от приложения, могут быть расстояние, время поездки или максимальная пропускная способность дороги между х и у. В невзвешенных графах между различными ребрами и вершинами нет никакой разницы в стоимости.

Разница между взвешенными и невзвешенными графами становится особенно очевидной при поиске кратчайшего пути между двумя вершинами. В невзвешенном графе кратчайший путь должен состоять из минимального числа ребер, и он может быть найден путем поиска в ширину, обсуждаемого ниже. Поиск кратчайшего пути во взвешенном графе требует использования более сложных алгоритмов, обсуждаемых в следующей части.

• ***Циклический или ациклический***. Граф, не содержащий циклов, называется ациклический. Связные неориентированные ациклические графы называются деревьями. Деревья - это простейшие интересные графы, причем они являются внутренне рекурсивными структурами, так как рассечение любого ребра приводит к появлению двух меньших деревьев.

Ориентированные ациклические графы называются DAG (Directed Acyclic Graph). Они естественным образом возникают в задачах планирования, где ребро (*х*,*у*) означает, что событие у должно произойти позже события *х*. Операция, называемая топологической сортировкой, упорядочивает вершины DAG с учетом этих ограничений предшествования. Обычно топологическая сортировка - это первый шаг в любом алгоритме, связанном с DAG.

• ***Простой или не простой***. Определенные типы ребер усложняют работу с графами. Петлей называется ребро (х, х) для которого используется только одна вершина. Ребро (х, у) называется мультиребром, если оно встречается в графе несколько раз.

При реализации графовых алгоритмов обеим этим структурам требуется уделять особое внимание. Поэтому граф, в котором таких структур нет, называется простым.

*•* ***Привязанный или топологический***. Граф называется привязанным (embedded), если его вершинам и ребрам было присвоено определенное геометрическое положение. Таким образом, любое изображение графа - это привязка, которая может влиять, а может и не влиять на алгоритм.

Иногда структура графа полностью определяется геометрией его привязки. Например, если нам задан набор точек на плоскости и нам нужно найти обход минимальной стоимости, при котором мы посещаем все вершины графа (задача о коммивояжере), то топологией, лежащей в основе, будет полный граф, в котором каждая вершина соединена со всеми остальными. Веса обычно определяются евклидовым расстоянием между двумя точками.

Другой пример топологии на основе геометрии проявляется в сетках точек. Во многих задачах, определенных на сетке размером *n* х *m*, требуется перемещаться между соседними точками, так что ребра неявным образом появляются из геометрии.

*•* ***Явный или неявный***. Графы не обязательно строятся явным образом, а потом обходятся. Нередко они строятся в процессе использования. Хорошим примером будет перебор с возвратом. Вершинами неявного поискового графа являются состояния поискового вектора, а ребра соединяют пары состояний, которые могут быть получены непосредственно друг из друга. Нередко проще работать с неявным графом, чем строить его явно для последующего анализа.

• ***Помеченный или непомеченный***. В помеченном графе каждой вершине присвоено уникальное имя или идентификатор, который отличает ее от остальных вершин. В непомеченных графах никаких различий такого рода не делается.

В большинстве графов, возникающих при решении конкретных задач, метки возникают естественным образом (например, имена городов в транспортной сети). Обычной задачей, возникающей в графах, является проверка на изоморфизм, определяющая, совпадают ли топологические структуры двух графов, если отбросить все метки. Такие задачи обычно решаются поиском с возвратом, когда мы присваиваем каждой вершине каждого графа определенные метки так, чтобы итоговые структуры получились идентичными.

**Структуры данных для графов**

Графы можно представить несколькими способами. Ниже мы обсудим четыре полезных варианта. Мы полагаем, что граф *G* = (*V*,*Е*) содержит *n* вершин и *m* ребер.

**• *Матрица смежности***. Мы можем представить *G* используя матрицу *М* размером *n* х *m*, в которой элемент *M* [*i*, *j*] равен, скажем, 1, если (*i, j*) является ребром *G*, и 0 иначе. Это позволяет быстро отвечать на такие вопросы, как: «Принадлежит ли (*i*, *j*) *G* ?» - и позволяет быстро обновлять граф в случае вставки и удаления. Тем не менее, для графов со многими вершинами и небольшим количеством ребер мы займем чересчур много лишнего места.

Рассмотрим граф, представляющий уличную карту Манхэттена в Нью-Йорке. Каждое пересечение двух улиц будет вершиной графа, и соседние перекрестки будут соединяться ребрами. Насколько велик такой граф? В основе Манхэттен 15 авеню, каждое из которых пересекает примерно 200 улиц. Это дает нам примерно 3000 вершин и 6000 ребер, так как каждая вершина соседствует еще с четырьмя вершинами и каждое ребро является общим для двух вершин. Такое небольшое количество данных можно легко и удобно хранить, но у матрицы смежности будет 3000 х 3000 = 9 000 000 ячеек, и почти все они будут пустыми!

**• *Списки смежных вершин через списки***. Более эффективным способом представления разреженных графов являются связанные списки, в которых хранятся инцидентные соседи каждой вершины. Для работы со списками смежных вершин требуются указатели, но в этом нет ничего страшного, если у вас есть опыт работы со связанными структурами.

При работе со списками смежности становится сложнее ответить на вопрос о принадлежности данного ребра (*i*, *j*) графу, так как нам нужно просматривать соответствующий список, чтобы найти подходящее ребро. Тем не менее, нередко совсем несложно построить алгоритмы работы с графами, не прибегающие к таким запросам. Обычно мы проходим по всем ребрам графа за один заход, используя обход в ширину или в глубину, и обновляем ребро в момент его посещения.

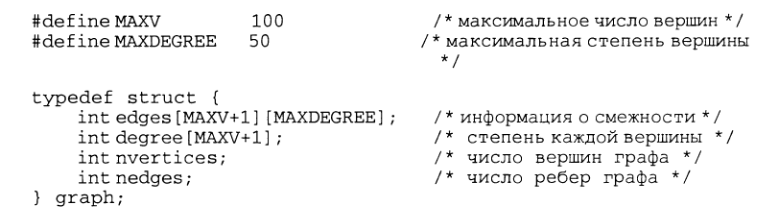
**• *Списки смежных вершин через матрицы*.** Списки смежности также можно реализовать через матрицы, избегая, таким образом, работы с указателями. Мы можем представить список массивом (или, что эквивалентно, строкой в матрице), введя счетчик *k* числа элементов и помещая их в первые *k* элементов массива. Теперь мы можем последовательно обойти элементы от первого к последнему, просто увеличивая счетчик цикла, а не путешествуя по указателям. На первый взгляд кажется, что эта структура данных объединила в себе худшие черты матриц смежности (большой размер) и списков смежности (необходимость поиска ребер). Тем не менее, существуют и плюсы. Во-первых, эту структуру данных проще всего запрограммировать, особенно, если речь идет о статичных графах, неизменных после построения. Во-вторых, проблему с размером можно решить, если выделять строки для каждой вершины динамически и делать их соответствующего размера.

Чтобы доказать нашу точку зрения, мы будем использовать эту структуру данных во всех последующих примерах.

**• *Таблица ребер*.** Еще более простой структурой данных будет массив или связанный список ребер. Работая с ней, сложнее отвечать на такие вопросы, как: «Какие вершины являются соседними для *x*?» - но она прекрасно подходит для определенных простых процедур, таких, как алгоритм Крускала (Kruskal) остовного дерева минимального веса.

Как было сказано выше, мы будем использовать списки смежности через матрицы в качестве наших основных графовых структур. Совсем несложно преобразовать эти подпрограммы для работы с честными, реализованными через указатели списками смежности.

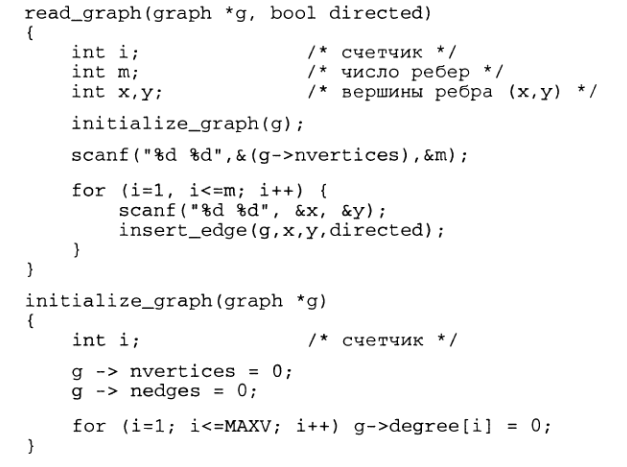
Графы будут представляться следующим типом данных. Для каждого графа мы храним счетчик числа вершин и присваиваем каждой вершине номер от 1 до nvertices. Мы храним ребра в массиве размером MAXV x MAXDEGREE, так что каждая вершина может быть соседней для MAXDEGREE других. Установив MAXDEGREE равной MAXV, мы сможем описать любой простой граф, но для графов низкого порядка это трата места.

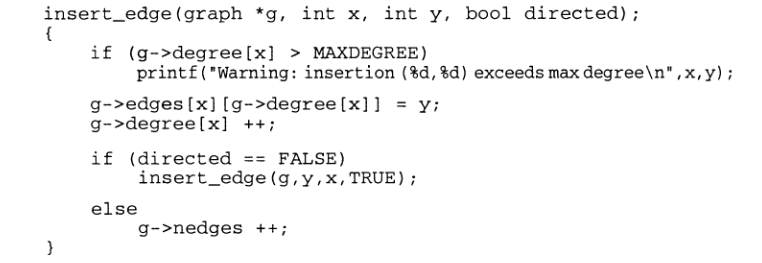


Ориентированное ребро (*х, у*) мы задаем целым числом *у* в списке смежности *х*, который расположен в подмассиве graph->edges [x]. Поле degree хранит количество входов для данной вершины. Неориентированное ребро (*х*, *у*) отмечается дважды в любой графовой структуре, основанной на смежности - один раз как *у* в списке *х* и второй раз как *х* в списке *у*.

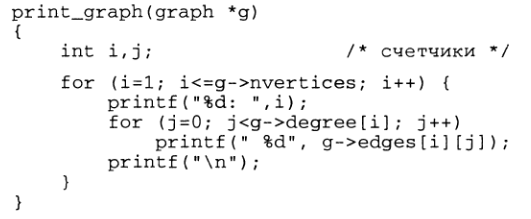
Чтобы продемонстрировать использование этой структуры данных, мы покажем, как граф считывается из файла. Обычное представление графа состоит из первой строки, задающей число вершин и ребер графа, за которой следует список ребер -по две вершины на строку.

Самой важной подпрограммой является insert\_edge. Мы передаем в нее булевский параметр directed, определяющий, нужно ли нам вставлять две копии ребра или только одну. Обратите внимание на использование рекурсии для решения поставленной задачи:





Теперь вывод соответствующего графа свелся к вложенным циклам:



**Обход графа**

Базовой операцией в любом графовом алгоритме является полный и систематический обход графа. Мы хотим посетить каждую вершину и каждое ребро ровно один раз в строго определенном порядке. Существует два основных алгоритма обхода: поиск в ширину (breadth-first search - BFS) и поиск в глубину (depth-first search - DFS). Для определенных задач нет никакой разницы, какой из них вы будете использовать, но в некоторых случаях выбор становится жизненно важным.

Обе процедуры обхода графа используют одну фундаментальную идею, а именно: мы должны помечать вершины, которые уже видели, чтобы не пытаться посетить их снова. Иначе мы можем потеряться в лабиринте и не суметь из него выбраться. BFS и DFS различаются только порядком, в котором они рассматривают вершины.

Поиск в ширину следует использовать в том случае, если (1) нам не важен порядок, в котором мы обходим вершины и ребра графа, то есть нас устроит любой, или если (2) нам нужно найти кратчайший путь в невзвешенном графе.

В приложении 5.1 вы можете посмотреть, как данная реализация используется для обхода в ширину, в глубину, обнаружения циклов и связных компонентов.

**Топологическая сортировка**

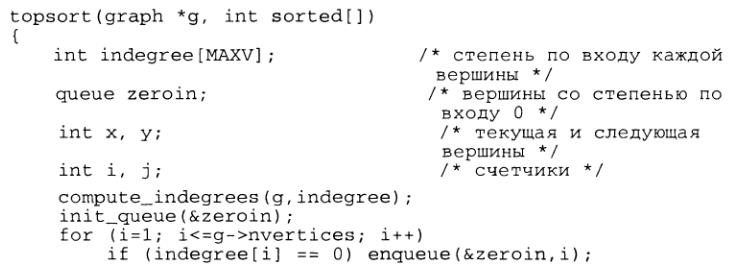
Топологическая сортировка - это фундаментальная операция на ориентированных ациклических графах (DAG). С ее помощью вершины располагаются в таком порядке, что все направленные ребра идут «слева направо» (от вершины с меньшим порядковым номером к вершине с большим). Очевидно, что такой расстановки не существует, если граф содержит ориентированные циклы, так как нельзя идти по прямой вправо и вернуться туда, откуда вы начали.

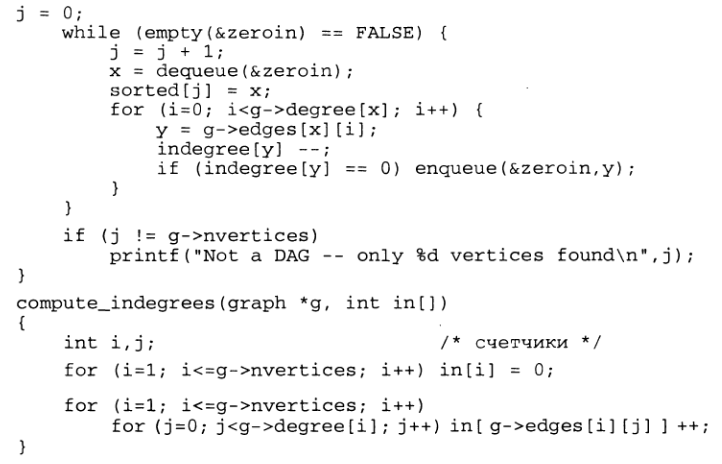
Важность топологической сортировки состоит в том, что благодаря ей вы можете обрабатывать каждую вершину до обработки ее наследников. Пусть ребра представляют собой ограничения предшествования так, что ребро (*х, у*) означает, что работа *х* должна быть выполнена до работы *у*. Тогда топологическая сортировка определяет правильное расписание. Кроме того, для заданного DAG может существовать множество таких расстановок.

Но существуют и другие приложения. Пусть мы ищем самый короткий (или самый длинный) путь из *х* в *у* в DAG. Определенно, никакая вершина, идущая после *у* в топологическом порядке, не может принадлежать такому пути, потому что не существует способа вернуться обратно к у. Мы можем обрабатывать соответствующим образом все вершины слева направо в топологическом порядке, рассматривая влияние исходящих из них ребер, и знать, что мы рассмотрим все, что нам нужно, и ничего лишнего.

Топологическая сортировка эффективно реализуется с использованием варианта поиска в глубину. Тем не менее, более прямолинейный алгоритм основан на анализе входящих ребер всех вершин DAG. Если в вершину не заходит ни одно ребро, то есть ее степень по входу равна 0, то мы можем спокойно ставить ее первой в топологическом порядке. Удаление опирающихся на нее ребер может привести к появлению новых вершин со степенью по входу равной 0. Этот процесс можно продолжать до тех пор, пока все вершины не будут расставлены по порядку; если такого не случится, то граф содержал цикл и не являлся DAG.

Рассмотрим следующую реализацию.





Нужно отметить несколько вещей. Поскольку поле degree записей графовой структуры данных содержит степень по выходу вершин, то сначала мы вычисляем степени по входу каждой вершины DAG. Эти величины совпадают для неориентированных графов, но могут различаться для ориентированных.

Для хранения вершин со степенью по входу 0 подойдет любой контейнер, так как порядок никоим образом не влияет на корректность решения. Обрабатывая вершины в разном порядке, можно получить разные топологические сортировки.

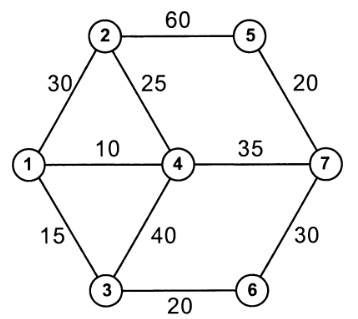
И наконец обратите внимание, что эта реализация не удаляет ребра из графа на самом деле. Вполне достаточно рассмотреть их влияние на степень по входу и обойти их, а не удалять.

### Задача 9.1 (Экскурсовод)

РС/UVa IDs: 110903/10099

М-р Ж. работает экскурсоводом в Республике Бангладеш. Его текущее задание состоит в том, чтобы показать группе туристов удаленный город. Как и во всех странах, определенные пары городов соединены дорогами с двусторонним движением. В каждой паре между соединенными городами и только между ними действует междугороднее автобусное сообщение, использующее дорогу, соединяющую их напрямую. В каждой автобусной службе есть определенный предел количества перевозимых пассажиров. У м-ра Ж. есть карта, на которой показаны города и соединяющие их дороги, а также пассажирские лимиты соответствующих автобусных служб.

Он не всегда может перевезти всех туристов в место назначения за одну поездку. Например, рассмотрим следующую карту дорог для семи городов, где дороги представляются ребрами, а число, написанное на каждом ребре, задает пассажирский лимит соответствующей автобусной службы.



М-ру Ж. потребуется минимум пять поездок, чтобы перевезти 99 туристов из города 1 в город 7, так как ему нужно сопровождать автобус с каждой группой. Наилучшим маршрутом будет 1 - 2 - 4 - 7.

Помогите м-ру Ж. найти маршрут, чтобы перевезти всех туристов в требуемый город за минимальное число поездок.

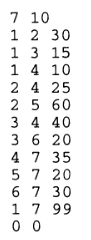
**Входные данные**

Входные данные будут состоять из одного и более тестовых блоков. Первая строка каждого блока содержит два целых числа: N (N ≤ 100) и R, задающие соответственно число городов и число дорожных сегментов. Каждая из следующих R строк содержит три целых числа (С1, С2 и Р), где C1 и С2 - это номера городов и Р (Р > 1) -это максимальное число пассажиров, которые могут перевозиться между двумя городами. Номера городов - это положительные целые числа от 1 до N.  
(R + 1)-я строка содержит три целых числа (S, D и T), задающих, соответственно начальный город, конечный город и число туристов, которых необходимо перевезти. Входные данные завершаются при N и R равных нулю.

**Выходные данные**

Для каждого введенного тестового блока сначала выведите номер сценария, а затем на отдельной строке минимальное число необходимых поездок. После каждого тестового блока выведите пустую строку.

**Пример входных данных**



**Соответствующие выходные данные**

Scenario #1

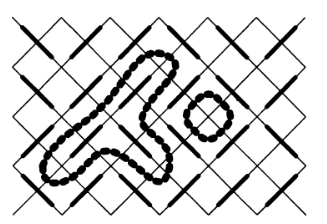
Minimum Number of Trips = 5

**Дополнительный вопрос:** как вы будете обрабатывать ситуацию, в которой структура графа введена ошибочно и решение задачи невозможно?

### Задача 9.2 (Лабиринт из косых)

РС/UVa IDs: 110904/705

Заполняя прямоугольник левыми (/) и правыми (\) косыми чертами (слешами), можно строить небольшие лабиринты. Вот пример:



Как можно заметить, пути в лабиринте не могут разветвляться, так что во всем лабиринте могут содержаться только (1) циклические пути и (2) пути, которые где-то заходят и где-то выходят. Нас интересуют только циклы. В нашем примере их ровно два.

Ваша задача состоит в написании программы, которая подсчитывает число циклов и находит длину самого длинного. Длина определяется как количество маленьких квадратов, из которых состоит цикл (те, границы которых на картинке нарисованы серым цветом). В нашем примере протяженность длинного цикла равна 16, а короткого 4.

**Входные данные**

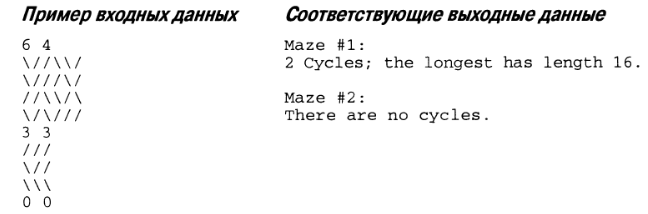
Входные данные состоят из нескольких описаний лабиринтов. Каждое описание начинается со строки, содержащей два целых числа *w* и *h* (1 ≤ *w*, *h* ≤ 75), задающих ширину и высоту лабиринта. Следующие *h* строк задают лабиринт, каждая из них содержит по *w* символов; причем все эти символы являются или «/» или «\».

Входные данные завершаются тестовым блоком, начинающимся с w = h = 0. Этот блок обрабатывать не нужно.

**Выходные данные**

Для каждого лабиринта выведите строку "Maze #n:", где n - это номер лабиринта. Далее выведите строку "k Cycles; the longest has length L. ", где k - это число циклов в лабиринте, а L - это протяженность самого длинного из них. Если лабиринт является ациклическим, выведите "No cycles . ".

После каждого тестового блока выведите пустую строку.



### Задача 9.3 (Лесенки редактирования)

PC/UVa IDs: 110905/10029

Назовем ступенькой редактирования (edit step) такое преобразование слова *x* в слово *y*, что слова *х* и *у* принадлежат словарю и слово *х* может быть преобразовано в слово *у* путем добавления, удаления или изменения одной буквы. Преобразования dig в dog и dog в do являются ступеньками редактирования. Лесенка ступенек редактирования (edit step ladder) - это отсортированная в описанном порядке последовательность слов. Для заданного словаря вы должны найти самую большую лесенку ступенек редактирования.

**Входные данные**

Входные данные состоят из словаря: набора слов в нижнем регистре в лексикографическом порядке по одному слову на строку. Длина каждого слова не превышает 16 букв, и в словаре не более 25 000 слов.

**Выходные данные**

Выходные данные состоят из одного целого числа, количества слов в самой большой лесенке ступенек редактирования.

**Пример входных данных**

cat

dig dog

fig

fin

fine

fog

log

wine

**Соответствующие выходные данные**

5

**Примечание**: при тестировании вы можете легко найти в интернете реальные словари нужного объема, исследуйте их с помощью разработанной программы.

### Задача 9.4 (Кубики)

PC/UVa IDs: 110906/10051

У вас есть N цветных кубиков разного веса. Каждая грань кубика покрашена в свой цвет. Вам нужно построить максимально высокую башню, подчиняющуюся тем ограничениям, что (1) мы не можем класть тяжелый кубик на легкий и (2) цвет нижней грани каждого кубика (за исключением самого нижнего) должен совпадать с цветом верхней грани кубика под ним.

**Входные данные**

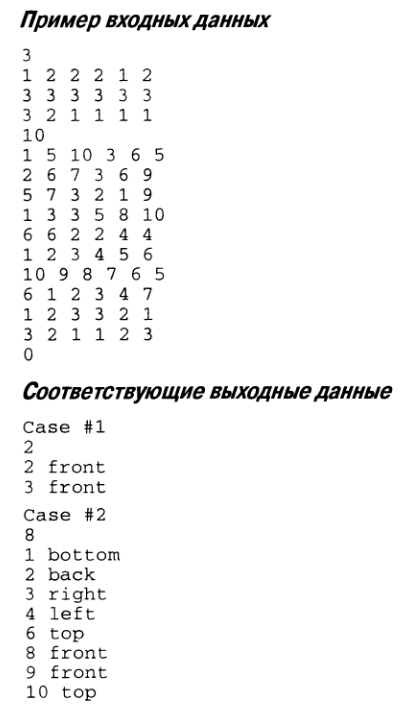
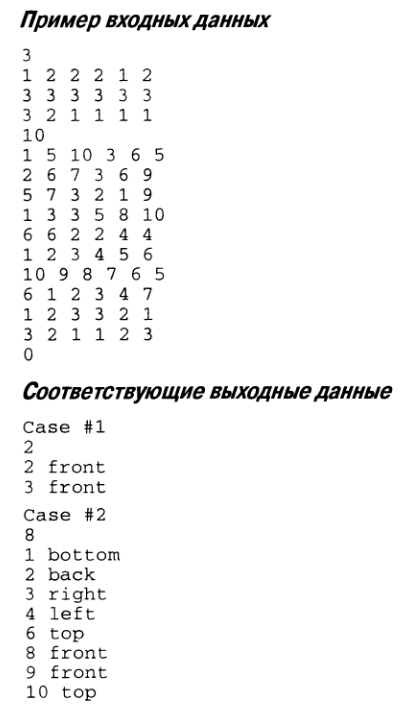
Входные данные могут содержать несколько тестовых блоков. Первая строка каждого тестового блока содержит целое число N(1 ≤ *N* ≤ 500), задающее число данных вам кубиков, *i*-я строка из следующих *N* содержит описание *i*-го кубика. Кубик описывается цветами его граней в следующем порядке: передняя, задняя, левая, правая, верхняя и нижняя. Для удобства оттенки цветов описываются целыми числами от 1 до 100. Кубики вводятся в порядке их утяжеления, то есть кубик 1 самый легкий, а кубик N самый тяжелый.

Входные данные завершаются при равном 0.

**Выходные данные**

Для каждого блока начните с вывода строки, содержащей номер тестового блока, как показано в примере выходных данных. На следующей строке выведите число кубиков в самой высокой возможной башне. Далее выведите описания кубиков в башне, по одному описанию на строку. Каждое описание содержит порядковый номер этого кубика во входных данных, один пробел, а затем строку-идентификатор (front, back, left, right, top, bottom) верхней храни кубика в башне. Может существовать несколько решений, подойдет любое.

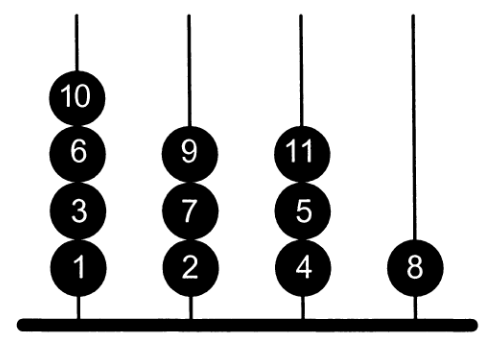
Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.



#### Задача 9.5 (Магнитные башни)

PC/UVa IDs: 110908/10276

Существует много интересных вариаций задачи о Ханойских башнях. В этой версии у нас имеется n колышков и шарики, содержащие все числа от 1, 2, 3,..., ∞. Если сумма чисел на двух шариках не является полным квадратом (то есть с2 для некоторого целого с), то они будут отталкивать друг друга с такой силой, что никогда не смогут соприкоснуться.



Игрок помещает шарики на колышки по одному, в порядке увеличения номеров (то есть сначала шарик 1, затем шарик 2, затем шарик 3 и т. д.). Игра заканчивается, когда у игрока нет хода, не приводящего к отталкиванию шариков.

Цель игры - поместить как можно больше шариков на колышки. На рисунке выше приведен наилучший возможный результат для четырех колышков.

**Входные данные**

Первая строка содержит одно целое число T, задающее число тестовых блоков (1 ≤ Т ≤ 50). Каждый тестовый блок содержит одно целое число N  
(1 ≤ N ≤ 50), задающее число доступных колышков.

**Выходные данные**

Для каждого тестового блока выведите строку, содержащую целое число, равное максимальному количеству шаров, которые возможно поместить. Если поместить можно бесконечное количество шаров, то выведите " -1".

**Пример входных данных** **Соответствующие выходные данные**

2 11

4 337

25

## Приложение 5.1. Основы работы с графами

**Поиск в ширину**

В нашей реализации поиска в ширину, bfs, мы используем два булевских массива, чтобы хранить информацию о каждой вершине графа. Мы говорим, что вершина открыта (discovered) когда первый раз ее посещаем. Вершина считается обработанной (processed) после того, как мы обошли все ребра, опирающиеся на эту вершину. Таким образом, статус каждой вершины изменяется от неоткрытой к открытой, а затем к обработанной. Эту информацию можно хранить с помощью одной переменной перечислимого типа данных, но вместо этого мы решили использовать две булевские переменные.

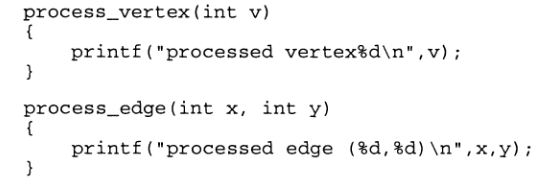
Как только вершина открыта, она помещается в очередь. Мы обрабатываем вершины в порядке «первый вошел, первый ушел», то обрабатываются первыми старшие вершины, то есть именно те, что ближе всего к корневому узлу.



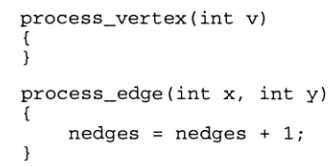
**Использование обхода**

Точное поведение bfs зависит от функций process\_vertex () и process\_edge (). Путем задания этих функций мы можем легко определить, что должно происходить при единственном посещении каждого ребра и каждой вершины.

Задав эти функции так:



мы напечатаем каждую вершину и каждое ребро ровно один раз. А задав их так:



мы найдем точное число ребер. Во многих задачах требуется предпринимать различные действия при нахождении ребер и вершин. Эти функции позволяют нам легко подстраиваться под требования конкретной задачи.

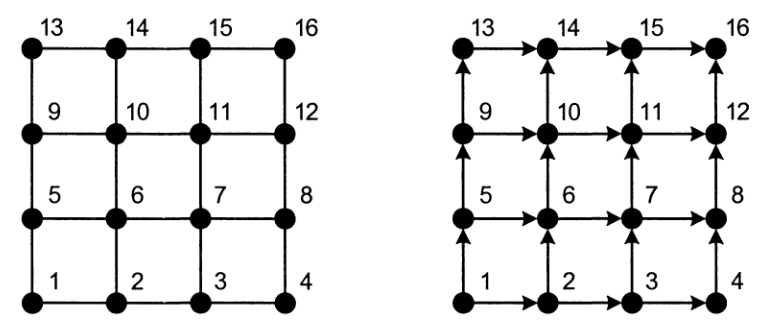
Еще одна возможность подстройки содержится в булевском предикате valid\_edge, позволяющем нам игнорировать существование определенных ребер во время нашего обхода графа.

**Нахождение путей**

Массив parent, заполняемый внутри bfs(), может оказаться весьма полезным при поиске интересующих нас путей по графу. Вершина, являющаяся предком вершины *i*, задается как parent[i]. Так как во время обхода мы открываем все вершины, то, за исключением корня, все узлы имеют предка. Предковое отношение задает открываемое дерево, причем начальный поисковой узел является корнем этого дерева.

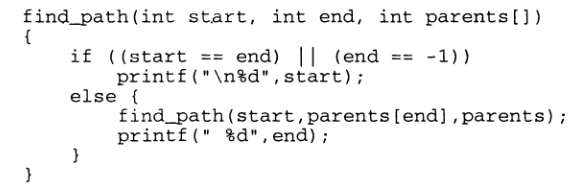
Так как вершины открываются в порядке увеличения расстояния от корня, это дерево обладает очень важным свойством. Уникальный путь по дереву от корня к любому узлу *х* принадлежащему *V* содержит минимально возможное число ребер (или, что то же самое, промежуточных вершин) для любого пути от корневого узла до *х* в графе *G*.

Мы можем воспроизвести этот путь по цепи предков, начиная с *х* и заканчивая корнем. Обратите внимание, что нам нужно работать в обратном направлении. Мы не можем найти путь от корня до *х*, так как такое направление не совпадает с направлением указателей на предков. Вместо этого мы должны искать путь от *х* к корню.



Неориентированный граф-сетка размера 4x4 (слева) и DAG, ребра которого идут по направлению к вершинам с большим номером (справа)

Так как в этом случае мы получаем путь, обратный желаемому, мы можем или (1) сохранить его, а затем, используя стек, явно поменять направление на противоположное, или (2) позволить рекурсии сделать все без нашего участия, как показано в следующей подпрограмме.



Для примера нашего сетчатого графа, приведенного на рисунке, алгоритм выдал следующую последовательность родительских отношений.



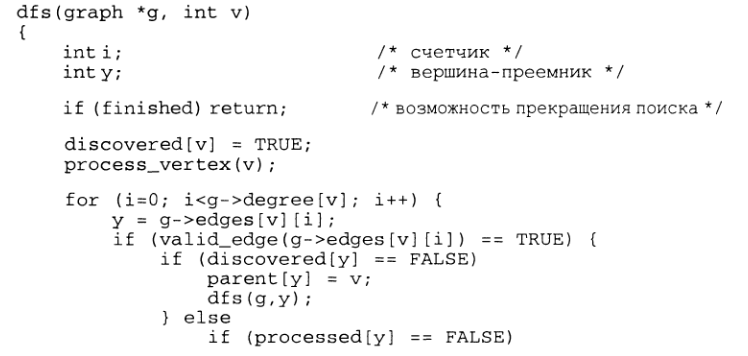
Исходя из родительских отношений, самым коротким путем из левого нижнего угла в правый верхний является {1, 2, 3, 4, 8, 12, 16}. Конечно, кратчайший путь не единствен, число таких путей в таком графе подсчитывалось в предыдущих ЛР.

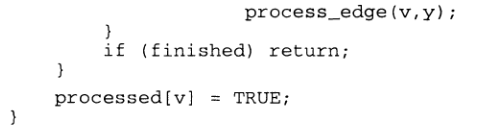
При использовании поиска в ширину для нахождения кратчайшего пути от *х* до *у* следует помнить две вещи: во-первых, дерево кратчайших путей можно использовать только в том случае, если корневым узлом BFS был задан *х*. Во-вторых, BFS дает кратчайший путь только для невзвешенного графа. Алгоритмы поиска кратчайшего пути во взвешенных графах будут рассмотрены в части 5.2.

**Обход графа в глубину**

В основе поиска в глубину лежит та же идея, что и в переборе с возвратом. Оба алгоритма производят полный перебор всех возможностей, продвигаясь, пока это возможно, и возвращаясь, когда не остается неисследованных вариантов дальнейшего продвижения. Оба алгоритма проще рассматривать как рекурсивные алгоритмы.

Поиск в глубину можно рассматривать как поиск в ширину с очередью, замененной стеком. Изящность реализации dfs через рекурсию состоит в том, что исчезает необходимость явной реализации стека.





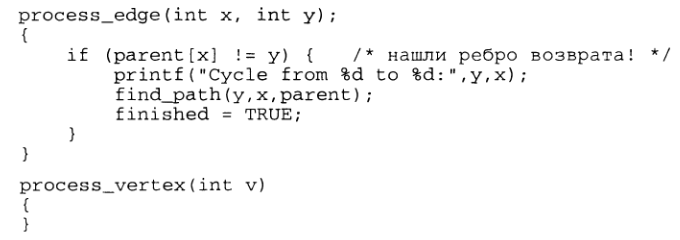
Корневые деревья - это особые случаи графов (ориентированные, ациклические, степень по входу не превышает 1, причем порядок определяется по исходящим ребрам каждого узла). Левое поддерево-корень-правое поддерево (in-order), корень-левое поддерево-правое поддерево (pre-order) и левое поддерево-правое поддерево-корень (post-order) - обходы, по сути, являются DFS, различающимися только в том, как они используют порядок исходящих ребер, и в том, когда они обрабатывают вершины.

**Обнаружение циклов**

Поиск в глубину разбивает все ребра неориентированного графа на две группы, ребра дерева (tree edge) и ребра возврата (back edge). К ребрам дерева относятся те ребра, которые сохраняются в массиве parent, то есть ребра, открывающие новые вершины. Ребрами возврата называются ребра, указывающие «обратно в дерево», то есть их конец является предком ранее обнаруженной вершины.

То, что все ребра попадают в одну из этих двух категорий, является занимательным свойством поиска в глубину. Почему ребро обязательно должно идти к предку? В DFS все узлы, достижимые из данной вершины *v*, исследуются прежде, чем мы закончим обход из *v*, так что иные топологии невозможны для неориентированных графов. В случае ориентированных графов итоговая картина получается в чем-то более сложной, но все равно высокоструктурироваиной.

Ребра возврата играют решающую роль в обнаружении циклов в неориентированных графах. Если ребер возврата не существует, то все ребра являются ребрами дерева и никаких циклов нет. Но любое ребро возврата, идущее от *х* к предку *у*, создает цикл из пути в дереве от *у* к *х*. Такой цикл легко обнаружить, используя dfs.



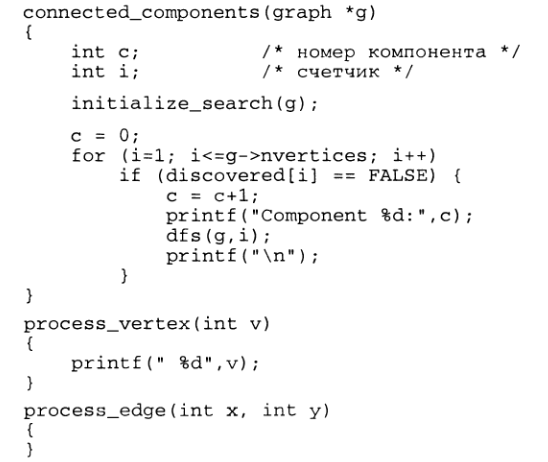
Мы используем флаг finished для прекращения работы после обнаружения первого цикла в нашем сеточном графе 4 х 4, а именно 3 4 8 7, причем ребром возврата является (7,3).

**Связные компоненты**

Связным компонентом неориентированного графа называется максимальное множество вершин таких, что существует путь между любой парой вершин. Они являются различными, не соединенными «кусками» графа.

Большое количество с виду сложных задач сводится к нахождению или подсчету связных компонент. Например, проверка разрешимости таких головоломок, как кубик Рубика или «пятнашки», на самом деле является проверкой связности графа разрешенных конфигураций.

Связные компоненты легко могут быть найдены с помощью поиска в глубину или поиска в ширину, так как порядок вершин не важен. Обычно мы начинаем с первой вершины. Все, что мы откроем во время этого поиска, должно являться частью одного и того же связного компонента. Далее мы повторяем поиск, начиная с одной из неоткрытых вершин (если таковые существуют), чтобы обнаружить следующий компонент, и так далее до тех пор, пока не будут обнаружены все вершины.



## Часть 2. Продвинутые алгоритмы

Различные варианты представления графов и алгоритмы обхода из раздела 5.1 являются «кирпичиками» для любого вычисления, основанного на графах. В этой части мы рассмотрим более продвинутую теорию графов и более продвинутые алгоритмы. Примеры реализации алгоритмов не вынесены в приложение поскольку они имеют большое практическое значение и настоятельно рекомендуется с ними ознакомиться.

Теория графов - это раздел математики, изучающий свойства графов. Благодаря ему у нас есть язык для описания графов. Многие задачи легко решаются, если суметь распознать лежащее в их основе понятие теории графов, а затем использовать классические алгоритмы для решения переформулированной задачи.

Мы начнем с обзора основ теории графов, а затем рассмотрим важные алгоритмы поиска таких вещей, как минимальные остовные деревья, кратчайшие пути и максимальные потоки. Мы наметим в общих чертах важные алгоритмы, которые относительно несложно запрограммировать, используя инструменты, разработанные в предыдущем разделе (см. приложения к нему).

Графы состоят из вершин и ребер. Простейшим свойством вершины является ее ***степень***, то есть число соединений с ней (к примеру, при добавлении петли степень вершины увеличивается на 2).

Степени вершин обладают несколькими важными свойствами. Сумма степеней всех вершин любого неориентированного графа в два раза больше числа его ребер, так как любое ребро дает вклад в степень каждой из смежных вершин. Следствием этого является то, что любой граф содержит четное число вершин нечетной степени. Для ориентированных графов аналогичное утверждение состоит в том, что сумма всех степеней по входу равняется сумме всех степеней по исходу. Четность степеней вершин играет важную роль в обнаружении эйлеровых циклов, которые мы рассмотрим далее.

***Деревьями*** называются неориентированные графы, не содержащие циклов. Степени вершин важны для анализа деревьев. Листом дерева называется вершина степени 1. Любое дерево, содержащее n вершин, содержит n -1 ребро, так что любое нетривиальное дерево содержит, как минимум, два листа. Важно, что удаление листа означает получение меньшего, «подстриженного» дерева, а не его разбиение.

***Корневым деревом*** называется ориентированный граф, в котором степень по входу каждого узла, за исключением корневого, равна 1. Степень по выходу листьев равна 0. Бинарным деревом называется корневое дерево, степень по выходу каждой вершины которого равна либо 0, либо 2. Как минимум половина вершин бинарного дерева должна быть листьями.

***Остовным деревом*** графа G = (V,E) называется подмножество ребер Е` из Е такое, что Е` является деревом на V. Иначе говоря, не каждый граф является деревом, но для в практических задачах часто приходится строить дерево поверх имеющегося графа, еще и удовлетворяя определенным условиям, самое очевидное из которых (указанное в определении остовного графа) – включение в наше дерево всех имеющихся вершин.

Для любого связного графа остовные деревья существуют; родительское отношение, получаемое нами при открытии вершин в процессе поиска в глубину или поиска в ширину, вполне подходит для создания одного из них. Остовное дерево минимального веса является весьма важным для взвешенных графов.

Граф называется ***связным***, если существует неориентированный путь между любой нарой его вершин. Существование остовного дерева доказывает связность графа. Алгоритм поиска связных компонентов, основанный на поиске в глубину, обсуждался ранее.

Тем не менее нужно знать и о других видах связности. Вершинной (реберной) связностью называют минимальное число вершин (ребер), которое нужно удалить, чтобы граф стал несвязным. Наиболее интересным частным случаем является наличие единственного слабого звена. Единственная вершина, чье удаление ведет к несвязности графа, называется точкой сочленения; если в графе таких точек нет, то граф называется двусвязным. Единственное ребро, чье удаление ведет к несвязности графа, называется перешейком (bridge); если в графе таких ребер нет, то он называется двусвязным по ребрам (edge-biconnected).

Простая задача - «в лоб» проверить граф на наличие точек сочленения и перешейков. Для каждой вершины/ребра удаляем ее/его из графа и проверяем, остался ли граф связным (не забудьте вернуть на место вершину/ребро перед следующим удалением).

При работе с ориентированными графами мы нередко сталкиваемся с сильно ***связанными компонентами***, то есть с такими частями графа, что в каждой существует ориентированный путь между любой парой вершин. Дорожные сети должны быть сильно связаны, иначе будут существовать места, куда вы сможете приехать, но не сможете оттуда вернуться домой, не нарушив знаки одностороннего движения.

Мы можем ***распознать*** сильно связанные компоненты в графе, основываясь на следующей идее. С помощью поиска в глубину несложно найти ориентированный цикл - любое ребро возврата и путь к нему в DFS-дереве задает такой цикл. Все вершины этого цикла должны принадлежать одному и тому же сильно связанному компоненту. Таким образом, мы можем стянуть (сжать) вершины этого цикла в одну точку, представляющую компонент, и затем повторить этот процесс. Эта последовательность сжатий заканчивается тогда, когда не остается ни одного ориентированного цикла и каждая вершина представляет собой один сильно связанный компонент.

Если связанный граф не является деревом, он обязательно содержит циклы. Особенно интересными являются циклы, включающие в себя все вершины или ребра графа.

***Эйлеровым циклом*** называется такое путешествие по графу, при котором **каждое ребро** посещается ровно один раз. Детская задача об изображении геометрической фигуры не отрывая карандаша от бумаги является примером нахождения эйлерова цикла (или пути), вершины которого являются пересечениями линий рисунка, а ребра линиями, которые нужно изобразить. В идеале маршрут почтальона должен являться эйлеровым циклом, чтобы он мог посетить каждую улицу (ребро) единожды до того, как вернуться домой. Строго говоря, эйлеровы циклы - это цепи, а не циклы, так как они могут посещать каждую вершину более одного раза.

Если неориентированный граф является связным и все его вершины имеют четную степень, то он содержит эйлеров цикл. Почему? Цепь должна заходить и исходить из каждой вершины на ее пути, из чего следует, что степени всех вершин должны быть четными. В этой идее содержится также способ нахождения эйлерова цикла. Мы можем найти простой цикл в графе используя алгоритм, основанный на DFS, который обсуждался выше. Удалив ребра этого цикла, получим вершины с четной степенью. После того как мы разбили все ребра на несовместные по ребрам циклы, мы можем их объединять по общим вершинам, чтобы построить эйлеров цикл.

В случае ориентированных графов важным условием является то, что все вершины должны иметь степень по выходу равную степени по входу. Удаление любого цикла сохраняет это свойство, так что в ориентированных графах эйлеровы циклы можно строить аналогичным способом. Эйлеровой цепью называется такое путешествие по графу, при котором каждое ребро посещается ровно один раз и которое может заканчиваться не там, где оно начиналось. При таком определении условие четности может быть нарушено ровно для двух вершин, одна из которых должна быть начальной вершиной, а вторая конечной.

***Гамильтоновым*** называется цикл, проходящий но **каждой вершине** графа ровно один раз. В задаче о коммивояжере требуется на взвешенном графе найти кратчайшее такое путешествие. Задача на эйлеров цикл в G = (V, Е) может быть сведена к задаче на гамильтонов граф построением графа G' = (V', Е') такого, что каждая вершина из V' соответствует ребру из Е и Е' содержит ребра, соединяющие все соседние пары ребер из G.

К сожалению, для решения задач на гамильтоновы графы не существует эффективного алгоритма. Таким образом, при необходимости решения у вас есть две возможности. Если граф достаточно маленький, то можно использовать поиск с возвратом. Каждый гамильтонов цикл описывается перестановкой вершин. Мы возвращаемся, когда не существует ребра, соединяющего последнюю рассмотренную вершину с непосешенной. Если граф слишком велик для такого подхода, мы должны попробовать переформулировать задачу, возможно, как задачу на эйлеров цикл на другом графе.

***Планарным*** (плоским) называется граф, который можно изобразить на плоскости так, чтобы никакие два его ребра не пересекались. Планарными являются многие графы, с которыми мы встречаемся. Любое дерево планарно (можете ли вы объяснить, как построить изображение заданного дерева без самопересечений?). Любая дорожная сеть, не содержащая бетонных/стальных мостов, должна быть планарной. Структура смежности выпуклого многогранника также должна являться планарным графом.

Планарные графы обладают несколькими важными свойствами. Во-первых, существует тесная взаимосвязь между числом вершин *n*, ребер *m* и граней *f* любого планарного графа. Теорема Эйлера утверждает, что *n* - *m* + *f* = 2. Дерево содержит n - 1 ребро, так что любое планарное изображение дерева содержит ровно одну грань, а именно лицевую грань. Любая привязка куба (8 вершин и 12 ребер) должна содержать 6 граней, это может подтвердить любой человек, игравший в кости.

Для проверки на планарность и нахождения привязки без самопересечений существуют эффективные алгоритмы, но они все труднореализуемы. Тем не менее, теорема Эйлера позволяет достаточно просто определить, что определенный граф не является планарным. Любой планарный граф содержит не более 3n - 6 ребер для n > 2. Из этого ограничения следует, что максимальная степень вершины, которую может содержать планарный граф, равняется 5, и удаление этой вершины оставляет меньший планарный граф, обладающий тем же свойством. Проверка того, является ли данный рисунок планарной привязкой, аналогична проверке того, пересекаются ли отрезки из заданного набора.

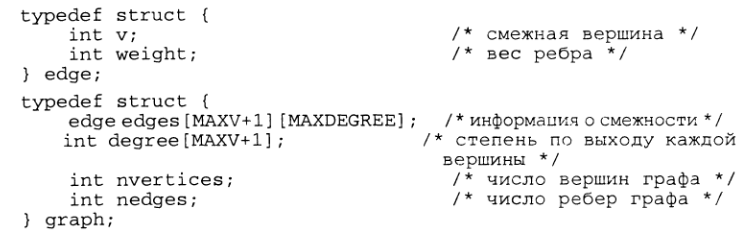
*Задача на минимальное остовное дерево*

Для реберно-взвешенных графов нас особенно интересует остовное дерево минимального веса, то есть остовное дерево, сумма весов ребер которого минимальна.

Остовные деревья минимального веса решают задачи по соединению множества точек (представляющих города, перекрестки или что-либо другое) минимальным количеством дорожного полотна, кабеля или труб. Любое дерево - это связный граф с минимальным числом ребер, а остовное дерево минимального веса - это связный граф с минимальным реберным весом.

Двумя основными алгоритмами для вычисления остовного дерева минимального веса являются алгоритм Крускала (Kruskal) и алгоритм Прима (Prim). Здесь мы рассмотрим алгоритм Прима, потому что его чуть проще запрограммировать, а также, потому что из него можно получить алгоритм Дейкстры (Dijkstra) поиска кратчайшего пути, внеся минимальные изменения.

Сначала нам нужно обобщить графовую структуру данных из предыдущего раздела так, чтобы она поддерживала реберно-взвешеиные графы. Ранее реберная составляющая содержала только вторую точку заданного ребра. Теперь нам нужно заменить ее структурой, позволяющей сохранять взвешенные ребра.



Также нам нужно соответствующим образом обновить различные алгоритмы обхода и инициализации. Это несложное задание.

В алгоритме Прима остовное дерево минимальною веса наращивается ступенчато, начиная с заданной вершины. Жадный алгоритм обеспечивает корректность: мы всегда добавляем ребро минимального веса, соединяющее вершину, принадлежащую дереву, и вершину, лежащую вне него.

При простейшей реализации этого алгоритма каждой вершине присваивается булевская переменная, обозначающая принадлежность этой вершины к дереву (массив intree в нижеследующем коде), а затем на каждом шаге ищется ребро с минимальным весом и ровно одной intree-вершиной.

Наша реализация немного умнее. Для каждой вершины дерева она ищет свое ребро минимальной стоимости, соединяющее ее с вершиной, еще не принадлежащей дереву, и добавляет его. После каждой вставки мы должны обновлять минимальные стоимости для всех вершин. Тем не менее, так как новая вершина - это единственное изменение в дереве, все возможные изменения в весах происходят за счет исходящих из нее ребер.



Остовное дерево минимальной стоимости или его стоимость можно воспроизвести двумя различными способами. Проще всего было бы добавить в эту процедуру выражения для вывода ребер по мере нахождения или суммирования их весов в специальной переменной для последующего использования. С другой стороны, так как топология дерева сохраняется в массиве parent, то он, в сумме с оригинальным графом, выдаст вам всё про остовное дерево минимального веса.

Этот алгоритм нахождения минимального остовного дерева обладает несколькими интересными свойствами, помогающими решить множество связанных задач.

- Остовное дерево с максимальным весом. Пусть мы наняли зловредную телефонную компанию соединить группу домов вместе и оплата услуг компании пропорциональна количеству израсходованного кабеля. Естественно, что они захотят построить остовное дерево настолько дорогое, насколько это возможно. Остовное дерево максимального веса для любого графа можно найти, просто поменяв знаки весов всех его ребер и запустив после этого алгоритм Прима. Самое «отрицательное» дерево в «отрицательном» графе является остовным деревом максимального веса в оригинале.

Большинство графовых алгоритмов совсем не так легко приспосабливается к отрицательным числам. Более того, алгоритмы поиска кратчайшею пути плохо работают с отрицательными числами, и они определенно не найдут путь максимальной длины, если использовать эту методику.

- Остовное дерево с минимальным произведением весов. Пусть мы хотим построить остовное дерево с минимальным произведением весов ребер. При этом мы считаем, что все веса положительны. Так как lg(a x b) = lg(a) + lg(b), то минимальное остовное дерево графа, в котором вес каждого ребра заменен его логарифмом - это остовное дерево с минимальным произведением весов.

- Наименее критичное остовное дерево. Иногда нам нужно найти остовное дерево с наименьшим возможным весом самого тяжелого ребра. На самом деле, остовное дерево минимального веса обладает таким свойством. Это следует из справедливости алгоритма Крускала. Такие деревья могут оказаться полезными, если реберный вес задает стоимость, пропускную способность или мощность. Менее эффективный, но более простой способ состоит в удалении всех «тяжелых» ребер из графа и проверки получившегося графа на связность. Проверки такого рода можно проводить, используя BFS/DFS.

Остовное дерево минимального веса уникально, если все m реберных весов отличны друг от друга. Если нет, то остовное дерево минимального веса, получаемое нами, зависит от порядка, в котором алгоритм Прима выбирает один из двух равнозначных случаев.

*Кратчайшие пути*

Задача нахождения кратчайшего пути встречалась нам в предыдущих разделах. Она может быть решена даже применением сортировки, и более красиво и логично - поиском в ширину.

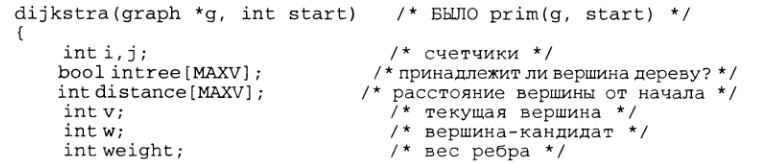
Но BFS не подходит для поиска кратчайшего пути во взвешенном графе, потому что кратчайший взвешенный путь между а и b необязательно содержит минимальное число ребер. У всех нас есть свои маршруты для поездок/прогулок, содержащие больше поворотов, чем самая простая дорога, но на которые тратится меньше времени из-за того что мы не попадаем в пробки, не стоим на светофорах и т. д.

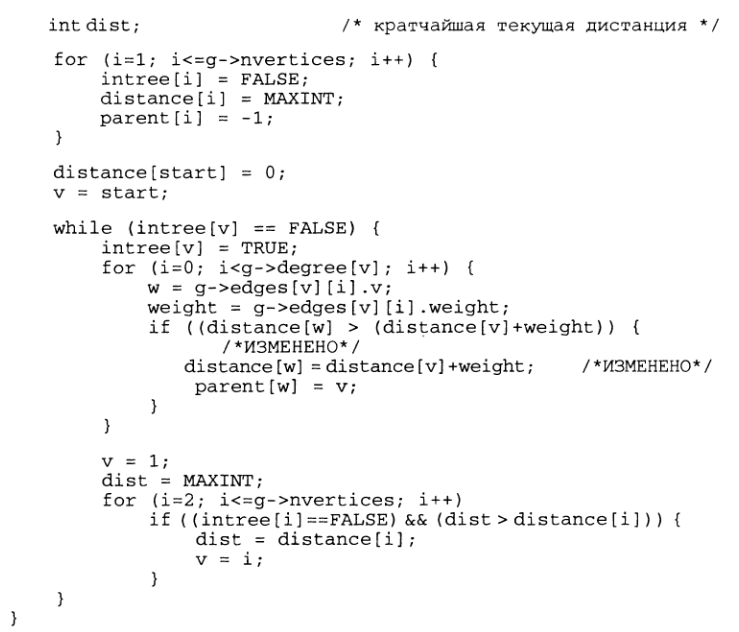
Для поиска кратчайшего пути в реберно-взвешенных графах или графах со взвешенными вершинами предпочтительно использовать ***алгоритм Дейкстры***. Для заданной вершины s он находит кратчайший путь от s до всех остальных вершин, включая желаемую вершину t.

Основная идея весьма схожа с алгоритмом Прима. На каждой итерации мы собираемся добавлять ровно одну вершину к дереву вершин, для которых мы знаем кратчайший путь до s. Точно так же как и в алгоритме Прима, мы будем отслеживать для всех вершин лучший путь, известный на данный момент, и добавлять их в порядке увеличения стоимости.

Разница между алгоритмами Прима и Дейкстры в том, как они оценивают желательность каждой вершины, не входящей в дерево. В задаче нахождения минимального остовного дерева все, что нас интересует - это вес следующего потенциального ребра дерева. Для нахождения кратчайшего пути нам нужно выбрать вершину, ближайшую (в смысле наименьшего путевого расстояния) к началу. Получаем функцию от веса нового ребра и от расстояния от начала смежной вершины дерева.

На самом деле, это не повлечет практически никаких изменений. Реализация алгоритма Дейкстры получается из реализации алгоритма Прима изменением трех строчек, одна из которых это просто название функции:





Как с помощью dijkstra найти длину кратчайшего пути от start до заданной вершины t? Она в точности равна значению distance[t]. Как мы можем восстановить собственно путь? Следуя указателям parent, начиная с *t* и до тех пор, пока не дойдем до start (или до -1, если пути не существует).

В отличие от алгоритма Прима, алгоритм Дейкстры работает только на графах с неотрицательными значениями реберных весов. Причина в том, что иначе мы можем посреди алгоритма обнаружить настолько отрицательный вес, что он изменит кратчайший путь от s до какой-то вершины, которая уже находится в дереве.

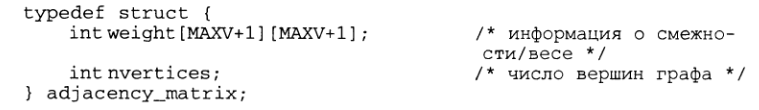
В большинстве прикладных задач отрицательные ребра не возникают, что делает наши рассуждения чистой теорией. Алгоритм Флойда (Floyd), обсуждаемый ниже, работает верно для всех графов, не содержащих циклов отрицательной стоимости, полностью искажающих понятие кратчайшего пути.

*Кратчайшие пути между всеми парами вершин*

Многим приложениям требуется знать длины кратчайших путей между всеми парами вершин заданного графа. Например, пусть вы хотите найти «центральную» вершину, сводящую к минимуму наибольшее или среднее расстояние до всех остальных вершин. Или пусть вам нужно найти диаметр графа - кратчайший путь между всеми парами вершин наибольшей длины. Он может соответствовать максимальному времени, которое требуется письму или сетевому пакету для доставки между двумя произвольными пунктами.

Мы могли бы решить эту задачу, вызвав алгоритм Дейкстры для каждой из n возможных начальных вершин. Но алгоритм Флойда для построения кратчайших путей между всеми парами вершин предлагает гораздо более изящный способ построения матрицы расстояний на основании начальной матрицы весов.

Алгоритм Флойда проще всего реализовать для матрицы смежности, которая в данном случае не является расточительством, так как нам все равно нужно сохранять все n2 расстояний. Наш тип adjacency\_matrix выделяет место для матрицы максимально возможного размера и следит за количеством вершин в графе.



Для нашей реализации матрицы смежности очень важно определить, как мы будем обозначать ребра, которых в графе нет. Для невзвешенных графов обычно используют следующее соглашение: ребра графа обозначаются 1, а отсутствие ребра - 0. Но тогда доя взвешенных графов мы получаем совершенно неверное представление, так как отсутствие ребра дает бесплатный переход от одной вершины к другой. Вместо этого для отсутствующего ребра нам нужно использовать значение MAXINT. Таким образом, мы можем и проверить наличие ребра в графе, и игнорировать отсутствующие ребра при поиске кратчайших путей при условии, что диаметр графа меньше MAXINT.

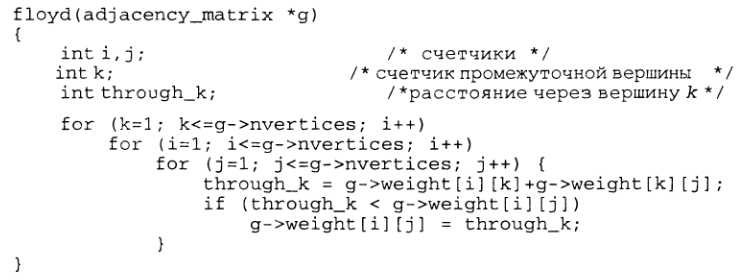


Все это достаточно просто. Как мы можем найти кратчайшие пути в такой матрице? Алгоритм Флойда начинает с нумерации вершин от 1 до n, но эти числа используются для упорядочивания вершин, а не их маркировки.

Алгоритм состоит из n итераций, где на *k*-й итерации в качестве промежуточных шагов при составлении пути между *х* и *у* могут использоваться только *k* первых вершин. При *k* = 0 промежуточные вершины использоваться не могут, так что все возможные кратчайшие пути состоят из начальных ребер графа. Таким образом, начальная матрица кратчайших путей между всеми парами вершин эквивалентна начальной матрице смежности. На каждой итерации нам становится доступно все большее число возможных кратчайших путей. Добавление вершины *k* в качестве промежуточной влияет только на те кратчайшие пути, которые проходят через *k*, так что



Справедливость этого утверждения не очевидна, впрочем, справедливость и изящность реализации очевидны и без этого:



Результат работы алгоритма Флойда в его текущей реализации не позволяет воссоздавать собственно кратчайший путь между любой парой вершин. Если вам нужен сам путь, используйте алгоритм Дейкстры. Тем не менее стоит отметить, что большинству задач на все пары вершин требуется именно итоговая матрица расстояний. Именно для таких задач и был разработан алгоритм Флойда.

Другое важное приложение алгоритма Флойда - это нахождение ***транзитивного замыкания*** ориентированного графа. При анализе ориентированного графа нас часто интересует, какие вершины достижимы из заданного узла.

Например, рассмотрим *граф шантажа*, определенный на множестве из n людей, в котором ребро *(i,j)* существует, если у *i* имеется достаточно значимая информация, чтобы заставить *j* сделать то, что необходимо *i*. Вы хотите нанять одного из этих *n* людей в качестве вашего личного представителя. Кто из них обладает большей властью в смысле возможностей шантажа?

Простейшим ответом была бы вершина максимальной степени, но на самом деле лучший представитель - это тот, у кого есть цепочки шантажа к максимальному количеству людей. Возможно, Стив может шантажировать только Мигеля, но если Мигель может шантажировать всех остальных, то Стив - это тот человек, который вам нужен.

Вершины, достижимые из любого заданного узла, можно найти с помощью поиска в ширину или в глубину. Но полную картину можно получить при решении задачи на кратчайшие пути между всеми парами вершин. Если в результате работы алгоритма Флойда кратчайший путь из *i* в *j* остался равным MAXINT, то можно считать, что ориентированного пути из *i* в *j* не существует. Любая вершина, такая, что длина кратчайшего пути до нее меньше MAXINT, должна быть доступна как в смысле теории графов, так и в смысле шантажа.

*Потоки и паросочетания*

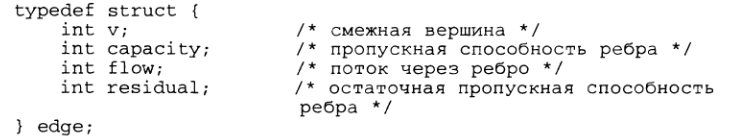
Любой реберно-взвешенный граф можно рассматривать как сеть труб, причем вес ребра *(i,j)* задает пропускную способность трубы. Пропускную способность можно рассматривать как функцию площади поперечного сечения трубы - широкая труба может пропустить 10 единиц потока за определенное количество времени, тогда как труба с меньшей площадью поперечного сечения - всего 5 единиц. Для заданного взвешенного графа G и двух вершин s и t в задаче потока в сети спрашивается, какой максимальный поток может быть отправлен из s в t с учетом максимальной пропускной способности каждой трубы.

Хотя задача потока в сети представляет интерес сама по себе, она важна прежде всего как средство для решения других важных задач на графы. ***Паросочетанием*** в графе G = (V,E) называется такое подмножество ребер Е' из Е, что никакие два ребра не опираются на одну и ту же вершину. Таким образом, паросочетания задают такие пары вершин, что каждая вершина принадлежит максимум одной паре.

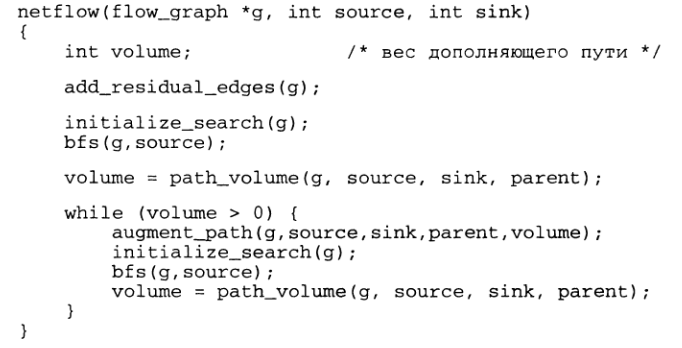
Граф G называется ***двудольным***, если его вершины можно разделить на два множества, назовем их L и R, таких, что для любого ребра одна его вершина принадлежит L, а другая R. Многие естественные графы являются двудольными. Например, пусть некоторые вершины обозначают работы, которые необходимо выполнить, а оставшиеся вершины - людей, которые могут их сделать. Существование ребра (j,p) обозначает, что человек р может сделать работу j. Естественным толкованием паросочетаний будет назначение на работу.

Максимальное паросочетание в двудольном графе можно найти, используя поток в сети. Создаем ***исток*** s, который ребрами веса 1 соединен со всеми вершинами в L. Создаем ***сток*** t, который соединен ребрами веса 1 со всеми вершинами в R. Присваиваем каждому ребру двудольного графа G вес 1. Теперь максимальный поток из s в t определяет максимальное паросочетание в G. Мы всегда можем найти поток равный паросочетанию, взяв ребра паросочетания и ребра, соединяющие их со стоком и истоком. Кроме того, большего потока существовать не может. Каким образом мы сможем пропустить более одной единицы потока через какую-либо вершину?

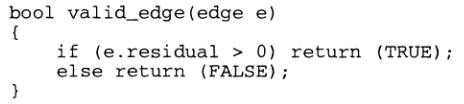
Для расчета потока в сети проще всего реализовать алгоритм дополняющего пути Форда - Фалкерсона (Ford - Fulkerson). Для каждого ребра нам нужно отслеживать ноток через него, а также остаточную пропускную способность. Тогда мы должны изменить нашу структуру edge, добавив новые поля.



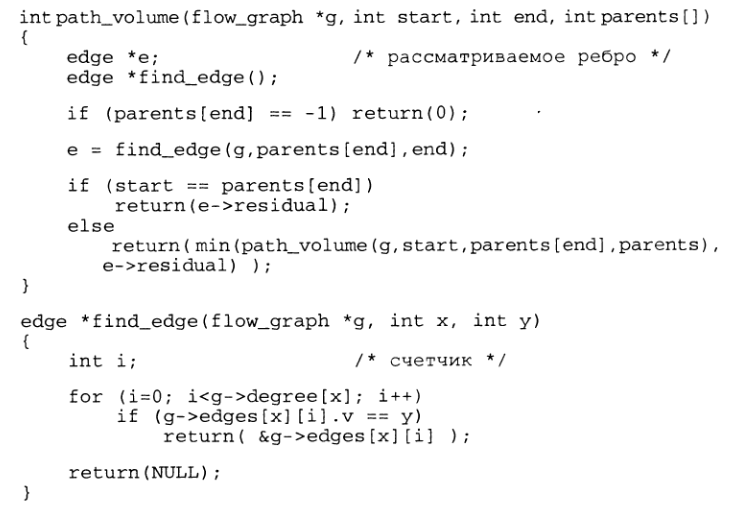
Мы ищем любой путь от истока до стока, который увеличивает общий поток, и прибавляем его к сумме. Мы достигнем оптимального потока, когда не будет существовать ни одного такого дополняющего пути.



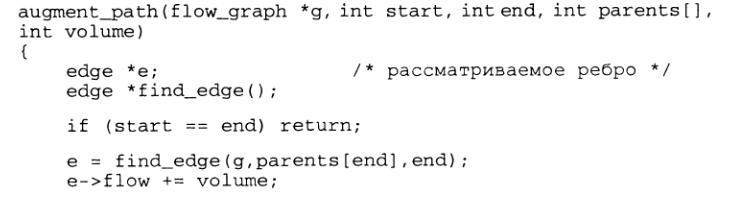
Любой дополнительный путь от истока до стока увеличивает поток, так что мы можем использовать bfs для поиска такого пути в соответствующем графе. Мы можем идти вдоль тех ребер сети, чья пропускная способность использована не полностью, или, говоря другими словами, остаточный поток этих ребер должен быть положительным. На основании этого утверждения bfs будет различать насыщаемые и ненасыщаемые ребра.

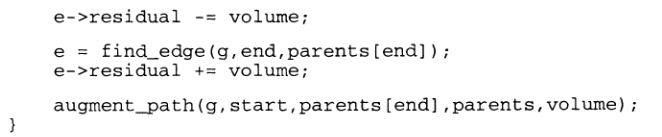


Дополнение пути переносит максимальный возможный поток из остаточной пропускной способности в положительный поток. Объем, который мы можем передать, ограничивается ребром пути с наименьшим значением остаточной пропускной способности, также как и движение по улицам города ограничивается наиболее загруженной точкой.



Отправка дополнительной единицы потока по ориентированному ребру (i,j) уменьшает остаточную пропускную способность ребра (i,j), но увеличивает остаточную пропускную способность ребра (j, i). Таким образом, при дополнении пути требуется смотреть и прямые и обратные ребра для каждого звена пути.





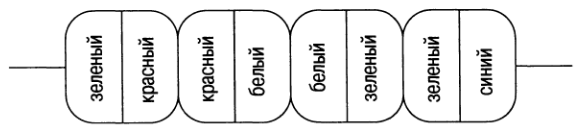
Для инициализации потокового графа требуется создать ориентированные потоковые ребра (i,j) и (j, i) для каждого ребра сети *е* = (i,j). Все начальные потоки считаются равными 0. Начальный остаточный поток (i,j) выставляется равным пропускной способности *е*, а начальный остаточный поток (j, i) выставляется равным 0.

Сетевые потоки - это продвинутая методика программирования, и для того, чтобы определить, можно ли решить конкретную задачу сетевыми потоками, требуется опыт.

### Задача 10.1 Ожерелье

РС/UVa IDs: 111002/10054

У Мэри есть красивое ожерелье, сделанное из цветных бусин. Любые две соседние бусины в ожерелье имеют один и тот же цвет в точке касания, как показано ниже.



Но, ах! Ожерелье порвалось, и бусины разлетелись по всему полу. Мэри постаралась собрать их все, но она не уверена, что у нее это получилось. Тогда она попросила вас помочь. Она хочет знать, можно ли собрать ожерелье по тому же принципу, что и старое, из тех бусин, что у нее есть. И если можно, то как?

**Входные данные**

Первая строка входных данных содержит целое число *Т*, задающее число тестовых блоков. Первая строка каждого тестового блока содержит целое число N (5 ≤ N ≤ 1000), задающее количество бусин, найденных Мэри. Каждая из следующих N строк содержит два целых числа, задающих цвета бусины. Цвета задаются целыми числами от 1 до 50.

**Выходные данные**

Для каждого тестового блока выведите его номер как показано в примере выходных данных. Если восстановить ожерелье невозможно, выведите строку, содержащую "impossibleee". В противном случае выведите N строк, каждая из которых содержит описание одной бусины. Для любого i (1 ≤ i ≤ N- 1) второе число строки i должно совпадать с первым числом строки i + 1. Кроме того, второе число строки N должно равняться первому числу строки 1 (должна выполняться собираемость ожерелья). Если возможны несколько решений, то подойдет любое.

Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.

|  |  |
| --- | --- |
| **Пример входных данных** | **Выходные данные**  CASE #1  Impossibleee  CASE #2  2 1  1 3  3 4  4 2  2 2 |

### Задача 10.2 (Пожарное депо)

PC/UVa IDs: 111003/10278

Город обслуживается несколькими пожарными депо. От жителей стали поступать жалобы, что расстояние между некоторыми домами и ближайшим пожарным депо чересчур велико, так что придется построить новое. Вам нужно выбрать местоположение нового депо так, чтобы уменьшить расстояние от домов жалующихся жителей до ближайшею депо.

В городе не более 500 перекрестков, соединенных отрезками дорог различной длины. Любой перекресток является пересечением не более 20 отрезков дорог. Мы полагаем, что и дома и пожарные депо располагаются на перекрестках. Более того, мы считаем, что на каждом перекрестке есть хотя бы один дом. Также на одном перекрестке могут находиться несколько пожарных депо.

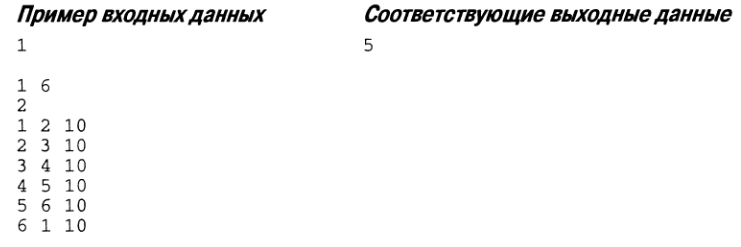
**Входные данные**

Входные данные начинаются со строки, содержащей одно положительное целое число, которое означает количество тестовых блоков. За ней следует пустая строка. Между двумя последовательными тестовыми блоками также находится пустая строка.

Первая строка каждого тестового блока содержит два положительных целых числа: количество существующих пожарных депо f (f ≤ 100) и количество перекрестков i (i ≤ 500). Перекрестки пронумерованы последовательно от 1 до i. Далее следуют f строк, каждая из которых содержит номер перекрестка, на котором расположено пожарное депо. Затем следуют строки, каждая из которых содержит три положительных целых числа: номер перекрестка, номер другого перекрестка и длину соединяющего их отрезка дороги. Движение по всем отрезкам двустороннее (по крайней мере, для пожарных машин), между любой парой перекрестков существует маршрут.

**Выходные данные**

Для каждого тестового блока выведите наименьший номер перекрестка, на котором нужно построить пожарное депо так, чтобы свести к минимуму максимальное расстояние от каждого перекрестка до ближайшего к нему пожарного депо. Выходные данные для двух последовательных блоков должны быть разделены пустой строкой.



### Задача 10.3 (Большой обед)

PC/UVa IDs: 111007/10249

Все команды, участвовавшие в ACM World Finals, приглашаются на большой банкет, который устраивается после церемонии награждения. Чтобы члены разных команд общались как можно больше, за одним столом не разрешается сидеть двум членам одной и той же команды.

Определите по заданному числу членов каждой команды (включая участников, тренеров, запасных и гостей) и числу мест за каждым столом, можно ли рассадить команды так, как описано выше. Если это возможно, то выведите один из возможных вариантов рассадки. Если существует несколько вариантов решения, то подойдет любой.

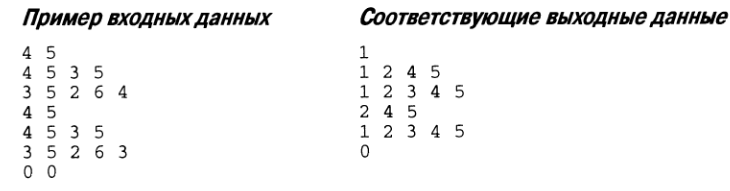
**Входные данные**

Входной файл может состоять из нескольких тестовых блоков. Первая строка каждого блока содержит два целых числа, 1 ≤ М ≤ 70 и 1 ≤ N ≤ 50, задающих число команд и столов соответственно. Вторая строка каждого тестового блока содержит М чисел: i-е число mi, показывает количество членов команды номер i. Число членов любой команды не превышает 100 человек. Третья строка содержит N чисел, причем j-е число nj, 2 ≤ nj ≤ 100, означает число мест за j-м столом.

Входные данные заканчиваются тестовым блоком, содержащим нули в качестве М и N.

**Выходные данные**

Для каждого тестового блока выведите строку, содержащую 1 или 0, указывающую на то, существует ли корректная рассадка членов команд. В случае положительного ответа выведите еще М строк, i-я из которых содержит номера столов (от 1 до N) для членов команды номер /.



### Задача 10.4 (Постановщик задач)

PC/UVa lDs: 111008/10092

В этом году так много студентов решили принять участие в региональных этапах олимпиад по программированию, что в жюри решили составить отбраковочные тесты для того, чтобы выявить наиболее многообещающих студентов. В этот тест могут входить до 100 задач из 20 категорий. Вам предложили набрать задачи для этого теста.

Поначалу работа казалась легкой, так как вам сказали, что дадут набор примерно из 1000 задач, поделенный на соответствующие категории. После того как вы получили задачи, оказалось, что их авторы нередко относили одну задачу к нескольким категориям. Так как ни одну из задач нельзя использовать в тесте более одного раза и число задач каждой категории установлено заранее, то выбрать задачи для теста - не такая уж и легкая работа.

**Входные данные**

Входной файл может содержать несколько тестовых блоков, каждый из которых начинается со строки, содержащей два целых числа, nk и np, где nk - это количество категорий, а np - число задач в данном наборе. Число категорий лежит в диапазоне от 2 до 20, а число задач в наборе не превышает 1000.

Вторая строка содержит nk положительных целых чисел, причем i-е число задает количество задач, которые должны быть отнесены к категории i (1 ≤ I ≤ nk). Вы можете считать, что сумма этих чисел не превышает 100.

j-я из следующих np строк содержит информацию о категориях j-й задачи набора. Каждое такое описание категорий задачи начинается с положительного целого числа, задающего количество категорий, к которым может быть отнесена задача. За этим числом следуют сами номера категорий.

Входные данные завершаются тестовым блоком, который содержит нули в качестве nk и np.

**Выходные данные**

Для каждого тестового блока выведите строку, сообщающую, можно ли набрать задачи из набора, следуя заданным ограничениям. Для положительного ответа используйте 1, для отрицательного - 0.

Если задачи набрать можно, выведите еще nk строк так, чтобы i-я строка содержала номера задач, которые могут быть включены в категорию i. Номера задач должны быть положительными целыми числами, не превышающими np, и два последовательных номера должны быть разделены одним пробелом. Подойдет любая правильная выборка задач.

|  |  |
| --- | --- |
|  |  |

*Приложение 0 – Пример титульного листа отчета по ЛР*

Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Омский государственный технический университет»

Кафедра «Информатика и вычислительная техника»

Отчёт по лабораторной работе № \_\_\_\_

по дисциплине

«Проектирование и тестирование программного обеспечения»

|  |  |
| --- | --- |
|  | Выполнил  Студент гр. БМТХ-181  Серый И.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подп., дата)  Проверил  Старший преподаватель каф. ИВТ  Звонов А.О. \_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подп., дата) |

Омск, 2024