МОСКОВСКИЕ УЧЕБНО-ТРЕНИРОВОЧНЫЕ СБОРЫ ПО ИНФОРМАТИКЕ

весна – 2006

Под редакцией В. М. Гуровица

Москва
Издательство МЦНМО
2007

УДК 519.671
ББК 22.18
М82

Московские учебно-тренировочные сборы по информатике.
М82 **Весна–2006** / Под ред. В. М. Гуровица — М.: МЦНМО, 2007. — 194 с.: ил.

ISBN ?-?????-???-?

Книга предназначена для школьников, учителей информатики, студентов и просто любителей решать задачи по программированию. В ней приведены материалы весенних Московских учебно-тренировочных сборов по информатике 2006 года: задачи практических туров, планы лекций и материалы избранных лекций и семинаров. Ко всем задачам прилагаются тесты для автоматической проверки их решений, которые можно найти на сайте `www.olympiads.ru/moscow/sbory`

УДК 519.671, ББК 22.18

Авторы статей: А. П. Лахно, Д. П. Кириенко, Д. Н. Королев, Б. О. Василевский, Ю. Г. Кудряшов, П. И. Митричев, В. А. Матюхин, А. А. Шестимеров, А. В. Фонарёв.

ISBN ?-?????-???-?

ОГЛАВЛЕНИЕ

Введение

Московские учебно-тренировочные сборы по информатике проводятся не первый год. Раньше они проводились после городской олимпиады и ставили целью в первую очередь отобрать школьников из числа призеров олимпиады на Всероссийскую олимпиаду по информатике.

Для этого организовывалось несколько (3–4) отборочных туров, а также нескольких обзорных лекций по основным алгоритмам, используемым на олимпиадах.

Весной 2006 года благодаря поддержки Департамента образования г. Москвы, Московского института открытого образования и Московского центра непрерывного математического образования удалось организовать и провести сборы в новом, расширенном формате.

С 20 марта по 2 апреля 2006 года около 40 школьников, успешно выступивших на Московской олимпиаде по информатике (как на основном туре, так и на олимпиаде для 7–9 классов, проводившейся в этом году впервые), приняли участие в сборах, прошедших на базе школы 179 Московского институа открытого образования, организованных под руководством зам. директора школы по информатизации, руководителя команды г. Москвы на Всероссийской олимпиаде школьников Д. П. Кириенко и под научным руководством победителя Всероссийской олимпиады школьников, члена жюри Всероссийской олимпиады по информатике А. В. Чернова (ВМК МГУ).

Ежедневно на сборах школьникам предлагался пятичасовой отборочный или тренировочный тур, после которого проводился разбор задач. Туры составлялись как правило из задач школьных и студенческих олимпиад разного уровня прошлых лет. После этого школьникам предлагалось на выбор прослушать одну из, как правило, трех лекций разного уровня. Первая лекция была расчитана на начинающих — школьников, только осваивающих азы языка программирования. Вторая лекция предполагала знание основных алгоритмов, третья обычно выходила далеко за рамки материала, предполагающегося известным участникам школьных олимпиад по информатике. Планы всех лекций и записи избранных лекций приводятся в соответствующих разделах этой книги.

В организации и проведении сборов также приняли активное участие:

Владимир Гуровиц, руководитель команды г. Москвы на Всероссийской олимпиаде по информатике (школа 2007, школа 218, МЦНМО),

Виктор Матюхин, председатель методической комиссии Московской олимпиады по информатике (ВМК МГУ, МЦНМО, гимназия 1543), Иван Ященко, исп. директор Московского центра непрерывного математического образования, зав. кафедрой математики и информатики Московского института открытого образования, зам. председателя оргкомитета Московской олимпиады по информатике,

Алексей Семенов, профессор, ректор Московского института открытого образования, председатель оргкомитета Московской олимпиады по информатике,

Борис Василевский, призер Всероссийской олимпиады по информатике, член Научного комитета Всероссийской олимпиады по информатике (мехмат МГУ),

Алексей Лахно, призер Всероссийской олимпиады по информатике, член Научного комитета Всероссийской олимпиады по информатике (мехмат МГУ),

Сергей Шедов, директор Мытищинской школы программистов, руководитель команды Московской области на Всероссийской олимпиаде школьников (ВМК МГУ),

Андрей Шестимеров, преподаватель Мытищинской школы программистов (ВМК МГУ),

Алексей Гусаков, призер Всероссийской олимпиады по информатике (мехмат МГУ),

Антон Фонарев, призер Всероссийской олимпиады по информатике,

Дмитрий Королев, призер Всероссийской олимпиады по информатике (МИФИ),

Ярослав Леонов (ВМК МГУ),

Вадим Антонов, призер Всероссийской олимпиады по информатике (ВМК МГУ), член Научного комитета Всероссийской олимпиады по информатике.

Александр Мамонтов (мехмат МГУ),

Маргарита Трухина (ВМК МГУ),

Роман Жуйков, призер Всероссийской олимпиады школьников по информатике (ВМК МГУ),

Алексей Тимофеев, призер Всероссийской олимпиады школьников по информатике, член Научного комитета Всероссийской олимпиады по информатике (мехмат МГУ),

Александр Шень, профессор (мехмат МГУ, НМУ).

Евгений Барский, руководитель команд МФТИ по программированию.

Многие материалы этих и последующих сборов доступны на сайте сборов `http://sbory.179.ru`, а архивы контестов — на сайте `http://www.olympiads.ru/moscow/sbory/`

Часть I

Задачи практических туров

Первый день: практический тур (стандартные задачи)

Задача A. Наименьшее общее кратное

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Найдите наименьшее общее кратное всех целых чисел от 1 до N.

Наименьшим общим кратным натуральных чисел a_1, a_2, $\ldots$, a_k называется число A, такое что A делится на a_i для всех i от 1 до k, причем A — наименьшее натуральное число, обладающее этим свойством.

Формат входного файла

Одно целое число N ($1 \leqslant N \leqslant 1000$).

Формат выходного файла

Выведите одно целое число — наименьшее общее кратное всех чисел от 1 до N.

Пример

a.in	a.out
3	6

Задача B. Сортировка

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Дана последовательность целых чисел из диапазона $[-2\,147\,483\,648, 2\,147\,483\,647]$. Вам поручается отсортировать эту последовательность и удалить из нее все повторения элементов, т. е. необходимо удалить все кроме одной копии числа в последовательности.

Формат входного файла

В первой строке входного файла находится целое число N — количество чисел в последовательности ($1 \leqslant N \leqslant 65\,536$). Последующие N строк содержат N целых чисел (по одному числу в строке).

Формат выходного файла

В выходном файле должно быть записано не более N чисел, отсортированных в порядке убывания, если N четно, и в порядке возрастания, если N нечетно. Каждое число должно встречаться в файле не более чем один раз.

Пример

b.in	b.out
6 8 8 7 3 7 7	8 7 3

Задача C. Прямоугольники

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Даны вещественные числа a, b, c, d. Выяснить, можно ли в прямоугольник со сторонами a, b целиком поместить прямоугольник со сторонами c, d.

Формат входного файла

В первой строке входного файла находятся вещественные числа a, b, c и d, разделенные одним или несколькими пробелами или символами перевода строки.

Формат выходного файла

Выведите в выходной файл слово `YES`, если второй прямоугольник можно поместить в первый, или слово `NO` в противном случае.

Пример

c.in	c.out
317 10 11 23	NO
31 10 10 31	YES

Задача D. Площадь

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Подсчитайте площадь заданного многоугольника.

Формат входного файла

В первой строке входного файла находится число N ($3 \leqslant N \leqslant 50\,000$) — количество вершин многоугольника. Последующие N строк содержат по 2 целых числа x и y ($-10\,000 \leqslant x, y \leqslant 10\,000$) — координаты вершин

Формат выходного файла

Выведите в выходной файл площадь многоугольника с точностью 600 знаков после запятой.

Пример

d.in	d.out
3 0 0 1 0 0 1	0.500000000000000...000

Выходной файл содержит 600 нулей.

Задача E. Две окружности

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

На плоскости даны две окружности. Ваша задача — найти все их общие точки.

Формат входного файла

В первой строке входного файла находится число K ($1 \leqslant K \leqslant 10\,000$) — количество пар окружностей. Каждая последующая пара строк описывает пару окружностей: в каждой строке записаны 3 целых числа x, y, r — координаты центра и радиус соответствующей окружности ($-1\,000 \leqslant x, y \leqslant 1\,000$, $0 < r \leqslant 1\,000$).

Формат выходного файла

Для каждой пары окружностей вы должны вывети одну из следующих фраз.

- «`There are no points!!!`»— если окружности не пересекаются.
- «`There are only i of them....`» — если окружности пересекаются ровно в i точках. В этом случае последующие i строк должны содержать координаты точек пересечения в формате `x y`. Точки должны быть выведены в лексикографическом порядке (сначала с меньшей координатой x, а при равных x — сначала с меньшей y). Координаты следует выводить с 6 знаками после запятой.
- «`I can't count them - too many points :(`» — если точек пересечения бесконечно много.

Все фразы должны быть выведены без кавычек. Вывод для каждой следующей пары окружностей должен быть отделен от предыдущего одной пустой строкой.

Пример

e.in	e.out
2 0 0 2 4 0 2 0 0 1 1000 1000 1	There are only 1 of them.... 2.000000 0.000000 There are no points!!!

Задача F. Игра

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

На листке записано в одну строку N $(2 \leqslant N \leqslant 100)$ целых положительных чисел. Каждое число не превышает 200. Играют двое. За каждый ход можно зачеркивать крайнее число либо слева, либо справа. Зачеркнутое число добавляется к очкам игрока.

N — четное. Необходимо вывести максимально возможную сумму очков для первого игрока при условии, что противник играет наилучшим образом.

Формат входного файла

В первой строке входного файла содержится одно целое число N $(2 \leqslant N \leqslant 100)$. В следующих N строках записан исходный ряд чисел, по одному числу в строке.

Формат выходного файла

Выходной файл должен содержать единственное число — максимально возможную сумму очков для первого игрока при наилучшей игре второго игрока.

Пример

f.in	f.out
6 4 7 2 9 5 2	18

Задача G. Рюкзак Алладина

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Попав в пещеру с сокровищами, наш Алладин не стал брать старую почерневшую лампу. Он кинулся собирать в свой рюкзак золотые монеты и драгоценные камни. Он бы, конечно, взял все, но чудес не бывает — слишком большой вес рюкзак может просто не выдержать.

Много раз он выкладывал одни вещи и на их место помещал другие, пытаясь как можно выше поднять стоимость взятых драгоценностей.

Требуется определить максимальную стоимость груза, который Алладин может поместить в свой рюкзак.

Будем считать, что в пещере имеются предметы N различных типов, количество предметов каждого типа не ограничено. Максимальный вес, который может выдержать рюкзак, равен W. Каждый предмет типа i имеет вес w_i и стоимость v_i $(i = 1, 2, \ldots, N)$.

Формат входного файла

В первой строке входного файла содержится два натуральных числа W и N — максимальный вес предметов в рюкзаке и количество типов предметов ($1 \leqslant W \leqslant 250$, $1 \leqslant N \leqslant 35$). Следующие N строк содержат по два числа w_i и v_i — вес предмета типа i и его стоимость ($1 \leqslant w_i \leqslant 250$, $1 \leqslant v_i \leqslant 250$).

Формат выходного файла

Выведите одно целое число — максимальную стоимость груза, вес которого не превышает W.

Пример

g.in	g.out
10 2 5 10 6 19	20

Задача H. Сумма

Имя входного файла: h.in
Имя выходного файла: h.out
Ограничение по времени: 10 сек
Ограничение по памяти: 64 мегабайта

Рассматриваются все разбиения натурального числа N на сумму K неотрицательных слагаемых ($1 \leqslant N \leqslant 32$, $2 \leqslant K \leqslant 32$). Суммы, отличающиеся только порядком слагаемых, считаем различными. Упорядочим все разбиения по убыванию первого слагаемого, при равных первых слагаемых — по убыванию второго слагаемого, при равных первых и вторых слагаемых — по убыванию третьего слагаемого и т. д. Пронумеруем их. Например, для $N=4$, $K=3$ все разбиения перечислены в таблице.

Номер	Слагаемые			Номер	Слагаемые		
	1-е	2-е	3-е		1-е	2-е	3-е
1	4	0	0	9	1	1	2
2	3	1	0	10	1	0	3
3	3	0	1	11	0	4	0
4	2	2	0	12	0	3	1
5	2	1	1	13	0	2	2
6	2	0	2	14	0	1	3
7	1	3	0	15	0	0	4
8	1	2	1				

Напишите программу, которая находит разбиение по номеру либо номер по разбиению.

Формат входного файла

В первой строке входного файла записана последовательность чисел. Если первое число 0, то необходимо найти разбиение по номеру, а если 1, то номер по разбиению. В первом случае далее в файле записано количество слагаемых, сумма и номер разбиения. Во втором случае далее в файле записано количество слагаемых K и затем разбиение (K неотрицательных чисел, сумма которых равна N). Все числа разделены пробелами.

Формат выходного файла

Выведите в выходной файл разбиение либо номер.

Пример

h.in	h.out
0 3 4 9	1 1 2
1 3 1 2 1	8

Задача I. Скобочки

Имя входного файла: i.in
Имя выходного файла: i.out
Ограничение по времени: 1 сек
Ограничение по памяти: 64 мегабайта

Дано целое число N ($1 \leqslant N \leqslant 10$). Выведите в алфавитном порядке все правильные скобочные последовательности длины $2N$, полагая, что символ '(' в алфавите идет раньше чем ')'.

Правильная скобочная последовательность — это либо пустая строка, либо строка вида (S), где S — правильная скобочная последовательность, либо строка вида $\mathtt{S}_1\mathtt{S}_2$, где $\mathtt{S}_1$ и $\mathtt{S}_2$ — правильные скобочные последовательности.

Формат входного файла

Входной файл содержит одно целое число N ($1 \leqslant N \leqslant 10$).

Формат выходного файла

Выведите в выходной файл в алфавитном порядке все правильные скобочные последовательности длины $2N$, по одной последовательности на строке, без пробелов.

Пример

i.in	i.out
3	((())) (()()) (())() ()(()) ()()()

Задача J. Спасение (p, q)-коня

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Путешествуя по Стране чудес, Алиса случайно наткнулась на (p, q)-коня. Зная, к чему приводят встречи в чистом поле с незнакомыми одинокими девочками, (p, q)-конь дал стрекача. Поле, по которому бежит Конь, имеет вид шахматной доски $M \times N$ клеток ($1 \leqslant N, M \leqslant 100$). Чтобы убежать, Конь должен переместиться из позиции (x_1, y_1), где он встретился с Алисой, в позицию (x_2, y_2). За один ход (p, q)-конь перемещается на p клеток в одном направлении и на q в другом (перпендикулярном). Обычный шахматный конь, например, является $(2, 1)$-конем.

Определить минимально возможное число ходов для спасения Коня.

Формат входного файла

Первая и единственная строка входного файла содержит 8 целых чисел M, N, p, q, x_1, y_1, x_2, y_2 ($1 \leqslant x_1, x_2 \leqslant M$, $1 \leqslant y_1, y_2 \leqslant N$, $0 \leqslant p \leqslant M \leqslant 100$, $0 \leqslant q \leqslant N \leqslant 100$).

Формат выходного файла

Первая строка выходного файла должна содержать целое число K — минимальное число ходов, которое потребуется Коню, чтобы убежать. В следующих $K + 1$ строках выведите последовательно координаты всех клеток спасительного маршрута. В случае, если искомого маршрута нет, выведите в выходной файл единственное число -1.

Пример

j.in	j.out
3 3 1 1 1 1 3 3	2 1 1 2 2 3 3
2 2 1 1 1 1 1 2	-1

Задача K. Кирпичи

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Имеется бесконечное количество прямоугольных кирпичей размерами $x_i \times y_i \times z_i$, каждый из которых можно ставить на любую грань (размеры каких-то двух стороны будут размерами основания, размер третьей стороны — высотой). Ваша задача — написать программу, находящую максимальную высоту башни, которую можно построить из этих кирпичей. Один кирпич может быть поставлен на другой, если размеры основания верхнего кирпича строго меньше соответствующих размеров основания нижнего.

Формат входного файла

В первой и единственной строке входного файла записано целое число N ($1 \leqslant N \leqslant 30$) — количество типов кирпичей, за которым следуют $3N$ целых чисел (N троек x_i, y_i, z_i) описывающих размеры каждого типа кирпичей ($1 \leqslant x_i, y_i, z_i \leqslant 65\,000$).

Формат выходного файла

Выведите в выходной файл единственное целое число — максимальную высоту башни.

Пример

k.in	k.out
1 10 20 30	40
2 6 8 10 5 5 5	21

Задача L. Мины

Имя входного файла:	`l.in`
Имя выходного файла:	`l.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Миротворцы ООН в одной из горячих точек планеты обезвреживали минное поле следующим образом. Имея карту, на которой каждая мина задана своими декартовыми координатами, они, обратив внимание на то, что никакие 3 мины не лежат на одной прямой, протянули специальный шнур от мины к мине так, чтобы он образовал выпуклый многоугольник минимального периметра, при этом все остальные мины оказались внутри многоугольника. Обезвредив соединенные мины, они вновь протянули шнур по тому же принципу, и опять обезвредили соединенные шнуром мины. Так продолжалось до тех пор, пока очередной шнур оказалось невозможным протянуть, руководствуясь изложенными правилами. Сколько мин осталось обезвредить и сколько раз саперам приходилось протягивать шнур?

Формат входного файла

В первой строке входного файла записано целое число N ($3 \leqslant N \leqslant 1\,000$) — количество мин. Во второй строке записано $2N$ целых чисел (N пар x_i, y_i), описывающих координаты каждой мины ($-32\,000 \leqslant x_i, y_i \leqslant 32\,000$).

Формат выходного файла

Выведите в выходной файл два целых числа через пробел — количество оставшихся мин и количество операций по натягиванию шнура.

Пример

l.in	l.out
9 0 0 0 8 6 8 6 0 1 1 1 7 5 7 5 1 3 2	1 2

Задача M. Конденсация графа

Имя входного файла:	`m.in`
Имя выходного файла:	`m.out`
Ограничение по времени:	0.5 сек
Ограничение по памяти:	64 мегабайта

Вам задан связный ориентированный граф с N вершинами и M ребрами ($1 \leqslant N \leqslant 20\,000$, $1 \leqslant M \leqslant 200\,000$). Найдите компоненты сильной связности заданного графа и топологически отсортируйте его конденсацию.

Формат входного файла

Граф задан во входном файле следующим образом: первая строка содержит числа N и M. Каждая из следующих M строк содержит описание ребра — два целых числа из диапазона от 1 до N — номера начала и конца ребра.

Формат выходного файла

На первой строке выведите число K — количество компонент сильной связности в заданном графе. На следующей строке выведите N чисел — для каждой вершины выведите номер компоненты сильной связности, которой принадлежит эта вершина. Компоненты сильной связности должны быть занумерованы таким образом, чтобы для любого ребра номер компоненты сильной связности его начала не превышал номера компоненты сильной связности его конца.

Пример

m.in	m.out
6 7 1 2 2 3 3 1 4 5 5 6 6 4 2 4	2 1 1 1 2 2 2

Задача N. Парковка

Имя входного файла:	`n.in`
Имя выходного файла:	`n.out`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Администрация города в связи с резким увеличением потока туристов решила организовать в центре города автомобильную стоянку. Стоянка имеет вид кольца с N парковочными местами. Парковочные места занумерованы от 1 до N по часовой стрелке. Автомобилист, подъезжая к стоянке, двигается вдоль нее по часовой стрелке, пока не найдет свободное парковочное место. После этого он паркует свой автомобиль на этом месте.

Вчера, после открытия, стоянку посетило M автомобилистов. Про каждого автомобилиста известно время его подъезда к стоянке, время, когда он покинул свою стоянку, а также парковочное место, около которого он подъехал к стоянке. Требуется определить для каждого автомобилиста номер парковочного места, которое он занял. Известно, что никакие два события не произошли одновременно, в частности, никакие два автомобилиста не подъезжают к парковке одновременно, и пока некоторый автомобилист ищет место для парковки, ни один другой автомобилист не подъезжает к парковке и не покидает ее.

Формат входного файла

Первая строка входного файла содержит числа N — количество парковочных мест и M — количество автомобилистов ($1 \leqslant N \leqslant 10^5$, $1 \leqslant M \leqslant N$). Следующие M строк содержат описания автомобилистов в следующем формате: t_1 — время подъезда к стоянке, t_2 — время, когда он покинул стоянку, и c — номер парковочного места, около которого он подъехал к стоянке (t_1, t_2 целые, $1 \leqslant t_1 < t_2 \leqslant 10^9$; $1 \leqslant c \leqslant N$).

Формат выходного файла

Выведите в выходной файл M чисел — для всех автомобилистов в порядке их перечисления во входном файле выведите номер занятого парковочного места.

Пример

n.in	n.out
6 6 1 9 6 2 5 6 3 7 6 4 11 4 6 10 5 8 12 4	6 1 2 4 5 1

Задача O. Паросочетание

Имя входного файла:	`o.in`
Имя выходного файла:	`o.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Граф называется двудольным, если его множество вершин V можно разбить на два непересекающихся множества вершин A и B, так чтобы концы любого ребра в этом графе находились в разных множествах. Паросочетанием в графе называется подмножество S множества ребер E этого графа, не имеющих общих вершин (для любых ребер $e_1 = (u_1, v_1)$ и $e_2 = (u_2, v_2)$, лежащих в S, выполнено $u_1 \neq u_2$, $u_1 \neq v_2$, $v_1 \neq u_2$ и $v_1 \neq v_2$).

Ваша задача — найти максимальное паросочетание в заданном двудольном графе. Максимальным называется паросочетание, состоящее из максимального количества ребер.

Формат входного файла

Первая строка входного файла содержит два целых числа N и M ($1 \leqslant N, M \leqslant 250$) — количество вершин во множествах A и B соответственно. Следующие N строк содержат описания ребер графа. В $(i+1)$-й строке содержится список вершин множества B, соединенных с i-й вершиной множества A. Список оканчивается числом 0. Нумерация вершин в множествах A и B независимая (вершины нумеруются с 1).

Формат выходного файла

Первая строка выходного файла должна содержать одно целое число L — количество ребер в максимальном паросочетании. Каждая из следующих L строк должна содержать описание одного ребра паросочетания — два целых числа u_i и v_i (номер вершины в множестве A и номер вершины в множестве B).

Пример

o.in	o.out
2 2 1 2 0 2 0	2 1 1 2 2

Задача P. Среднее расстояние

Имя входного файла:	`p.in`
Имя выходного файла:	`p.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Вам дано описание дорожной сети страны. Ваша задача — найти среднюю длину кратчайшего пути между двумя городами.

Средней длиной называется отношение суммы по всем парам городов (a, b) длин кратчайших путей $l_{a,b}$ из города a в город b к числу таких пар. Здесь a и b — различные натуральные числа в диапазоне от 1 до N, где N — общее число городов в стране. Следует учитывать только такие пары городов, между которыми есть кратчайший путь.

Формат входного файла

Сеть дорог задана во входном файле следующим образом: первая строка содержит числа N и K $(1 \leqslant N \leqslant 100,\ 1 \leqslant K \leqslant N(N-1))$, где K — количество дорог. Каждая из следующих K строк содержит описание дороги с односторонним движением — три целых числа a_i, b_i и l_i $(1 \leqslant a_i, b_i \leqslant N,\ 1 \leqslant l_i \leqslant 1\,000)$. Это означает, что имеется дорога длины l_i, которая ведет из города a_i в город b_i.

Формат выходного файла

Вы должны вывести в выходной файл единственное вещественное число — среднее расстояние между городами. Расстояние должно быть выведено с 6 знаками после десятичной точки.

Пример

p.in	p.out
6 4 1 2 7 3 4 8 4 5 1 4 3 100	25.000000

Второй день: практический тур

Задача A. Метеоритный дождь

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	0.3 сек
Ограничение по памяти:	64 мегабайта

Вам была передана информация о надвигающемся метеоритном дожде. Также у Вас есть информация о возможных координатах точек падения метеоритов в случае, если они пройдут слои атмосферы.

Вам необходимо определить, какие из метеоритов могут упасть на Ваше государство. Территория государства представляет собой выпуклый многоугольник, граница которого задана списком вершин в порядке обхода против часовой стрелки. Необходимо для множества координат точек падения определить, лежат ли они внутри государства. Точки, лежащие на границе государства, не считаются лежащими внутри.

Возможно, что среди вершин, описывающих границу многоугольника, существует более двух, находящихся на некоторой прямой.

Формат входного файла

В первой строке файла находится целое число N $(3 \leqslant N \leqslant 20\,000)$ — количество вершин, описывающих границу многоугольника.

В каждой из следующих N строк находится два числа, разделенные пробелом, описывающие координаты очередной точки границы.

В следующей строке находится целое число M $(1 \leqslant M \leqslant 20\,000)$ — количество метеоритов.

В каждой из следующих M строк находится два числа, разделенные пробелом, описывающие координаты точки, для которой необходимо проверить условие задачи.

Все координаты во входном файле — целые числа, по модулю не превосходящие 10^6.

Формат выходного файла

Файл должен содержать M строк, по одной строке на каждый метеорит. Если точка падения метеорита находится на территории Вашего государства, то строка должна содержать единственное слово `YES`, иначе — `NO`.

Пример

input.txt	output.txt
4 2 4 8 4 6 8 4 6 4 3 5 4 7 5 5 6 7	NO NO YES YES

Задача B. Джекпот

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Вы являетесь одним из организаторов популярной телевизионной игры, в которой один из участников дошел до финального конкурса. В случае победы в этом конкурсе, он должен получить огромную сумму денег. Но проблема в том, что у Вас нет достаточного количества денежных средств, чтобы выплатить призовой фонд, поэтому необходимо не допустить выигрыш участника.

В чем же заключается конкурс? Участнику предлагается несколько игровых столов, из которых он выбирает любой, а затем делает первый ход в игре, правила которой будут описаны ниже. В случае его победы в игре, он побеждает в конкурсе и получает призовой фонд. Следует заметить, что соперником участника является «мастер игры», играющий по оптимальной стратегии (то есть стратегии, позволяющей ему выиграть при любых ходах соперника, если это возможно для данной игры).

Правила игры

На игровом столе лежит некоторое количество палочек. Два игрока делают ходы по очереди. При каждом ходе игрок вытягивает некоторое число палочек. Это число должно не превышать m и лежать в интервале от 1 до $(m^2 \bmod k)+1$, где $a \bmod b$ — остаток от деления a на b, m — число палочек, оставшихся на столе, а k — некоторое число. (Эту формулу в свое время придумали Вы и очень этим горды). Игрок, вытягивающий последнюю палочку, проигрывает.

Так, например, если $k=3$, а палочек на столе 5, то вы можете вытянуть одну либо две палочки, так как $(5^2 \bmod 3)+1$ равно 2. Вытянув две палочки, вы оставите соперника с единственным вариантом хода, ибо он сможет вытянуть только одну палочку (так как $(3^2 \bmod 3)+1=1$).

У Вас есть n палочек, которые вы должны полностью распределить по игровым столам. Все игровые столы должны содержать различное число палочек, и игровых столов должно быть, по крайней мере, два, так как вы должны предоставить участнику выбор. Также вы не можете составить игру из одной палочки, так как это очевидно проигрышная для участника игра. Ну и самое главное условие, которое Вам необходимо выполнить: какой бы игровой стол ни выбрал участник, и как бы он ни играл, «мастер игры», пользующийся оптимальной стратегией, должен выиграть.

Формат входного файла

В единственной строке записаны целые числа n $(5 \leqslant n \leqslant 1000)$ и k $(2 \leqslant k \leqslant 1000)$, разделенные пробелом.

Формат выходного файла

В случае невозможности формирования выигрышного для вас набора игр выходной файл должен содержать единственную строку, в которой записано число 0. В противном случае в первой строке файла должно находится число x — количество сформированных игровых столов, а во второй строке — x различных чисел, разделенных пробелом — количество палочек на каждом из игровых столов (числа могут находиться в произвольном порядке). В случае множества вариантов ответа, вывести любой из них.

Пример

input.txt	output.txt
9 3	2 3 6
5 2	0

Задача C. Марсианские города

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

В 2004 году связь с марсоходом «BLR» («Bulb Location and Rummage»), вот уже несколько лет работавшем на Марсе, была потеряна. Долгое время ученым не удавалось установить с ним связь, и этот проект был приостановлен. Прошло совсем немного времени, и свершилась давняя мечта человечества — на красную планету ступила нога человека.

Изучая более детально поверхность, ученые находили все больше доказательств существовавшей когда-то (а, возможно, и существующей до сих пор) жизни. Однако этого было недостаточно, и вскоре силы сосредоточились на более детальном изучении того, что находилось под землей. В ходе работ одной из групп ученых, неожиданно были найдены некие подземные ходы. На первый взгляд они выглядели как обыкновенные пещеры, однако позже было выяснено, что природа вряд ли могла создать что-то подобное.

Вскоре было найдено еще множество систем, состоящих из пещер и подземных переходов. Все они обладали некоторыми общими свойствами, и ученые предположили, что это остатки марсианских городов.

Систему пещер и переходов ученые считали марсианским городом, если она обладала следующими свойствами:

1) каждый переход соединял только одну пару пещер;

2) они располагались в несколько параллельных рядов, причем в каждом из рядов было одинаковое число пещер, а i-е пещеры в рядах (если считать в некотором направлении, одинаковом для всех рядов) также образовывали ровные ряды. Ученые для удобства называли эти ряды соответственно горизонтальными и вертикальными (для каждого из городов это выбиралось произвольно и принципиального значения не имело). Таким образом, каждая пещера в марсианском городе характеризовалась своим расположением в одном вертикальном и одном горизонтальном ряду;

3) из каждой пещеры вело не более четырех переходов — они вели во все пещеры, соседние в горизонтальном и вертикальном рядах.

Вскоре в одной из таких пещер и был найден «BLR». Оказалось, что он уже многие годы ездил по марсианским городам и собирал разнообразную полезную информацию, которую ученые, конечно, хотели бы использовать, поскольку сами они еще не успели изучить все более подробно. Однако некоторые устройства марсохода были давно сломаны, и существовала возможность, что он объезжал одни и те же города по несколько раз. Поэтому, чтобы систематизировать полученную информацию, ученым нужен был способ, чтобы сравнивать описания городов по некоторым критериям и выяснять, описывают ли они один и тот же город. Было известно, что, попадая в город, он обязательно объезжал все переходы между пещерами и запоминал для каждого перехода пару пещер им соединяемых. Также важно было то, что пещеры в городах могли располагаться на различных глубинах. Эта информация и была взята за основу предложенного вскоре метода сравнения.

Итак, рассматриваются описания двух городов, содержащие одинаковое количество пещер и переходов. Каждой пещере из первого описания устанавливается некоторая соответствующая пещера во втором так, чтобы выполнялись следующие свойства:

1) для любых двух пещер из первого описания, соответствующие им во втором различны;

2) две пещеры из первого описания соединены (не соединены) переходом только тогда, когда соответствующие им пещеры во втором описании соединены (не соединены) переходом.

Если такое соответствие установлено, и соответствующие города находятся на одинаковых глубинах, то можно было бы считать описания принадлежащими одному городу. Однако, как уже упоминалось выше, не все оборудование марсохода работало стабильно, и поэтому замеры глубин в разное время могли немного отличаться. Поэтому было решено описанным ниже способом вычислять показатель близости, и считать два описания принадлежащими одному городу, если этот показатель был достаточно мал.

В качестве показателя близости брали сумму квадратов разностей глубин, на которых находились соответствующие пещеры из данных описаний.

Таким образом, в целом метод сравнения состоял в следующем: найти соответствие между пещерами из двух описаний такое, чтобы оно удовлетворяло описанным выше свойствам, и показатель близости был как можно меньше. Однако такое соответствие не всегда удавалось легко установить, поскольку пещер часто было много, и система переходов могла быть довольно запутанной. Для этого и потребовалась программа, которая для пары описаний городов определяет наименьший из возможных показателей близости.

Формат входного файла

В первой строке входного файла находятся числа N и M — число пещер и число переходов в каждом из описаний. Далее идут два блока, имеющих одинаковую структуру. Для каждого из блоков предполагается, что пещеры занумерованы целыми числами от 1 до N ($N \leqslant 10\,000$). Первые N строк в блоке содержат по одному числу — i-я строка блока содержит глубину пещеры с номером i (целое число от $-32\,767$ до $32\,767$). Каждая из следующих M строк содержит пару различных чисел — номера пещер, соединяемых переходом. Каждая пара пещер содержится в описании не более одного раза.

Формат выходного файла

В единственной строке выходного файла должно содержаться одно целое число — наименьший из возможных показателей близости либо строка `incorrect`, если хотя бы одно из описаний не может являться марсианским городом.

Третий день: практический тур

Задача A. Предложение

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Предложение состоит из слов. Словом называется последовательность, состоящая из символов с ASCII-кодами, большими 32. Слова в предложении разделяются одним или несколькими пробелами (ASCII-код 32). *Характеристикой* пары слов называется количество несовпадающих букв у этой пары, находящихся на одних и тех же позициях в каждом слове. Если длины слов различны, то считается, что слово меньшей длины дополняется справа до длины другого слова символами с ASCII-кодами, меньшими 32. Необходимо найти *характеристику предложения* — сумму характеристик всех возможных пар слов.

Формат входного файла

Во входном файле находится предложение, содержащее не больше 200 слов. Каждое слово имеет длину, меньшую 251.

Формат выходного файла

Выведите одно число — характеристику предложения.

Пример

input.txt	output.txt
`Земля земная`	`4`
`Горох зарыт зря`	`13`

Задача B. Текстовый редактор

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Пусть имеется некоторая строка $S = s_1 s_2 \ldots s_n$, где s_i — некоторый символ. Определим операцию замены всех вхождений G в S строкой $L = l_1 l_2 \ldots l_m$ следующим образом. Пусть курсор находится на первой позиции строки S. Выберем самое левое вхождение G в S, которое начинается в позиции курсора или правее, и заменим его строкой L. Затем перенесем курсор на следующий символ (может быть даже символ конца строки S) после самого правого символа вставленной строки L. Так повторяем до тех пор, пока не останется вхождений G в S справа от позиции курсора. Результат операции замены всех вхождений G в S строкой L будем обозначать через $RES(S, G, L)$.

Необходимо реализовать одну из функций текстового редактора — заменить все вхождения G в S строкой L, т. е. найти $T = RES(S, G, L)$. Необходимо также найти количество символов в строке R, которая получается из S двойным применением указанной процедуры (то есть $R = RES(T, G, L)$).

(Будем говорить, что строка $G = g_1 g_2 \ldots g_k$ входит в строку S, начиная с позиции j, если $s_{j+i-1} = g_i$ для всех $i = 1, 2, \ldots, k$. Вхождение G в S назовем самым левым, если не существует больше других вхождений G в S левее данного.)

Формат входного файла

В первой строке входного файла находится S, во второй — G, в третьей — L. Длина каждой их строк не превышает 200 символов. ASCII коды символов, входящих в эти строки, больше 31.

Формат выходного файла

В первой строке выходного файла выведите строку $T = RES(S, G, L)$. Во второй строке выведите число символов в строке $R = RES(T, G, L)$.

Пример

input.txt	output.txt
QWERTYabababababa aba ba	QWERTYbabbaba 12
ABC DEF GHI	ABC 3
textedit edit replace	textreplace 11

Задача C. Последовательность

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Известны первые k членов последовательности — a_1, a_2, ..., a_k ($0 \leqslant a_i \leqslant 9$, где $i = 1, 2, \ldots, k$). Другие члены последовательности вычисляются по следующему правилу: $a_i = \sum_{j=i-k}^{i-1} a_j$, то есть каждый следующий член равен сумме k предыдущих. Необходимо найти последние r цифр числа a_n.

Формат входного файла

В первой строке входного файла находятся 3 целых числа — k, n и r ($1 \leqslant k \leqslant 20$, $1 \leqslant n \leqslant 10^{18}$, $1 \leqslant r \leqslant 9$). В следующей строке находится k чисел — a_1, a_2, ..., a_k.

Формат выходного файла

Первой строкой выходного файла выведите r цифр числа a_n. Ведущие нули следует опустить.

Пример

input.txt	output.txt
2 5 1 1 2	8
1 100001 1 5	5

Задача D. Жажда скорости

Имя входного файла:	`nfs.in`
Имя выходного файла:	`nfs.out`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

На некотором прямолинейном участке дороги находятся N контрольных пунктов. Для каждого контрольного пункта известно X_i — расстояние в километрах от начала дороги до i-го контрольного пункта, и T_i — время в минутах, не позже которого данный контрольный пункт должен быть пройден. Гонщик может начинать гонку в любом месте на дороге. Машина едет с постоянной скоростью, равной 4 км/мин. Будем считать, что машина может развернуться моментально и на прохождение контрольного пункта гонщик затрачивает 0 минут. Гонщику сообщается число M — количество контрольных пунктов, которые нужно пройти для успешного завершения гонки. Кроме этого, требуется затратить на гонку как можно меньше времени.

Формат входного файла

В первой строке входного файла находятся 2 целых числа — N и M ($1 \leqslant M \leqslant N \leqslant 500$). В последующих N строках находятся по два целых числа — X_i, T_i ($0 \leqslant X_i, T_i \leqslant 100\,000$). Контрольные пункты перечисляются в порядке возрастания X_i. Гарантируется, что $X_i \neq X_j$ для любых $i \neq j$.

Формат выходного файла

Если успешно завершить гонку нельзя, то выведите `Impossible`.

Иначе первой строкой выведите расстояние в километрах от начала дороги до точки, в которой начинается гонка, и время в минутах — продолжительность гонки (отличающуяся от правильной не более чем на 0.001). На следующей строке выведите номера контрольных пунктов в порядке их прохождения (необходимо вывести только один возможный вариант).

Пример

nfs.in	nfs.out
3 2 10 1 14 5 18 9	18 1 3 2

nfs.in	nfs.out
3 3 0 1 4 0 8 3	4 3 2 1 3

Четвертый день: практический тур

Задача A. The Baker's tale

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	10 сек
Ограничение по памяти:	64 мегабайта

Булочник сообщил экспедиции важную информацию о методах охоты на Снарка. Однако, для него не это было главным. Он знал, что если вместо Снарка ему встретится Буджум, шансы оставить от себя хотя бы мокрое место у Булочника ничтожно малы. Узнав про это, его товарищи по экспедиции решили выяснить у Благозвона, чем отличаются породистые Снарки от не менее породистых Буджумов. После ответов из серии «Увидите… Узнаете…», Барристер, мобилизовав всё своё красноречие, смог вытащить следующую информацию. Следы, оставляемые обоими существами, представляют собой набор точек на земле, некоторые из которых соединены между собой отрезками. Любые две точки соединены не более, чем одним отрезком.

Следы Снарков могут быть самыми разными, но все они получены из «базового следа» (AB, BC, AC, AD, BD, CD) при помощи некоторого числа преобразований всего двух видов, а именно:

1. $(AX, BX, CX) \to (AQ, BP, CR, PQ, QR, PR)$;
2. $(AB, CD) \to (AC, BD)$.

Здесь буквами обозначены точки, парами букв — отрезки, соединяющие точки. При преобразовании, точки из левой части, отсутствующие в правой части преобразования, исчезают, равно как отрезки из левой части преобразования, отсутствующие в правой. Объекты, отсутствующие в левой части, но присутствующие в правой, добавляются.

Если же след не может быть получен из «базового» подобным образом, то это след Буджума. Осталось только научиться различать эти случаи, и опасность исчезнуть для Булочника резко уменьшится.

Формат входного файла

$1 \leqslant K \leqslant 1000$ — число точек, $N > 0$ — число отрезков, затем сами отрезки, каждый отрезок с новой строки. Отрезки задаются номерами точек A_i и B_i для концов каждого отрезка. Точки нумеруются натуральными числами от 1 до K.

Формат выходного файла

Слово `Snark`, если это след Снарка, либо `Boojum`, если это след Буджума.

Пример

input.txt	output.txt
2 1 1 2	Boojum
4 6 1 2 2 3 3 1 1 4 2 4 3 4	Snark

Задача B. The Beaver's Lesson

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

После того, как Бобёр и Бойня были атакованы птицей Джубджуб, для защиты от атаки Бобру надо было вслух производить арифметические действия. Так как арифметика (а особенно в экстремальной ситуации) не была сильной стороной Бобра, а иногда для построения узоров требуются некоторые расчёты, то у бобра с собой была разновидность арифмометра — «бобрифмометр».

Бобрифмометр состоит из ленты, поделенной на K ячеек (лента позаимствована Бобром из телеграфного аппарата, и изначально в каждой ячейке записана или точка, или тире) и каретки с карандашом, на другом конце которого имеется ластик. Каретка может перемещаться вправо и влево. Если каретка находится над ячейкой, то с её помощью Бобёр может поставить в данную ячейку тире, если там находится точка, и поставить в данную ячейку точку, если там находится тире.

Программа для бобрифмометра записывается на деревянной пластинке и кодируется в виде направляющих пазов.

Запись команд бобрифмометра для базовых операций:

`>` переход на одну ячейку вправо;
`<` переход на одну ячейку влево;
`v` поставить тире в ячейку, над которой находится каретка;
`x` поставить точку в ячейку, над которой находится каретка;
`s` завершить работу программы.

Кроме того, в программе могут использоваться макроопределения, которые описываются в начале программы. Описание макроопределения представляет собой последовательность команд, которой предшествует заголовок макроопределения `M<имя>`. Признаком завершения макроопределения является строка с символом `E`. В качестве имени используется любое натуральное число меньше 10 000. В описании макроопределений нельзя использовать или определять другие макросы. Макрос может быть включен в программу строкой `#<имя>`.

Например, следующая программа:

```
M4
>
v
E
>
#4
>
>
#4
s
```

приведёт к выполнению такой последовательности команд:

```
>
>
v
>
>
>
v
s
```

Программа должна обязательно заканчивать свою работу командой `s`. Это нормальное завершение работы. Программа завершает свою работу также тогда, когда не может выполнить действие: нельзя поставить тире туда, где уже стоит тире, нельзя поставить точку туда, где

уже стоит точка и нельзя сместиться за пределы ленты. В этих случаях бобрифмометр ломается. В начале работы программы бобрифмометр находится в некотором состоянии: на ленте расставлены точки и тире, а каретка находится над определенной ячейкой.

Бобру надо с помощью бобрифмометра из одного числа, записанного точками и тире на ленте, получить заданное число, написав код программы. Но проблема в том, что деревянная пластина отнюдь не бесконечна.

Напишите программу, которая составляет код для бобрифмометра, переводящий его из одного заданного состояния в другое заданное состояние (состояние — это запись на ленте + положение каретки). При этом эта программа должна завершаться нормально, а её текст вместе с макроопределениями должен содержать не более $500 + (L/2)$ строк, где L — количество символов на ленте.

После 1 000 000 операций каретка ломается и бобрифмометр можно выбрасывать, так что количество базовых операций не должно превышать 1 000 000.

Формат входного файла

В первой строчке указано количество ячеек в ленте L — натуральное число, меньшее 10 000. Затем идёт описание начального и конечного состояния бобрифмометра. Состояние описывается двумя строками. Первая строка описания содержит последовательность из L точек или тире. Вторая строка описания содержит позицию каретки. Ячейки в первой строке описания нумеруются с единицы.

Формат выходного файла

Ваша программа должна вывести программу для бобрифмометра.

Пример

input.txt	output.txt
12	M1
..........-.	<
10	<
.-..-..-..-.	v
1	<
	E
	#1
	#1
	#1
	s

Задача C. The Barrister's dream

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Барристер, проснувшись, осознал, что Благозвон звонил не по делу, после чего снова заснул... и его сон продолжился.

Дело о свинье-дезертире, которое выиграл Снарк, было воспринято как прецедент и вошло в анналы юриспруденции. В соответствии с этим, материалы дела должны быть размещены на специальном стенде, который должен быть установлен в каждом суде (для ознакомления судей, присяжных и особенно адвокатов со столь революционным прецедентом).

Тексты выступлений должны размещаться на прямоугольных листах бумаги высоты 1 на специальном стенде заданной ширины W. Выступления со стороны защиты и обвинения (показания свидетелей, реплики сторон и так далее) должны быть размещены без перекрытий по разные стороны вертикальной перегородки. Для удобства ознакомления присяжных в каждой из частей на одной горизонтали могут помещаться не более двух выступлений (иначе говоря, никакая горизонтальная линия не может пересекать более двух прямоугольников, находящихся в одной части).

Нужно выбрать положение разделительной перегородки и разместить тексты выступлений без перекрытий таким образом, чтобы реплики защиты оказались в правой части, а реплики обвинения — в левой. При этом, высота стенда должна быть минимальной (стенд обрежут внизу там, где заканчивается самое нижнее выступление) с целью экономии материала.

Формат входного файла

В первой строке задана ширина стенда, целое число W, $1 \leqslant W \leqslant 50\,000$. В следующей — количество реплик защиты N_1 и обвинения N_2, разделенные пробелом. Затем идет строка с N_1 числами — длины реплик защиты, и строка с N_2 числами — длины реплик обвинения, $0 \leqslant N_1, N_2 \leqslant 1000$.

Длины реплик являются натуральными числами, не превосходящими 10 000.

Формат выходного файла

Выведите H — минимальную высоту стенда. Если требуемое размещение невозможно, выведите 0.

Пример

input.txt	output.txt
10 3 2 5 4 4 1 1	2

Задача D. The Banker's Fate

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

После того, как Злопастный Брандашмыг похитил Банкира, Банкир предложил ему скидку и некоторую сумму выкупа. Для того, чтобы объяснить похитителю выгоду своего предложения, Банкир произвел ряд нетривиальных арифметический действий. Банкир знает, что Брандашмыг совершенно не умеет пользоваться десятичной системой счисления, тем более ему не интересно знать тонкостей британской денежной системы. Торг происходил в некоторой произвольной системе счисления, принятой на данный момент в Зазеркалье. Спустя некоторое время после инцидента с Банкиром, Брандашмыг оказался пойман и предстал перед судом. Барристер, защищавший Брандашмыга, заявил, что его подзащитному не было смысла сводить Банкира с ума, приведя в качестве вещественного доказательства выкладки Банкира. Выкладки представляли собой вычисления, сделанные Банкиром, а именно, операции сложения и умножения. Однако обвинитель заявил, что данные вычисления бессмысленны, и попросил Барристера предоставить доказательства, а именно, основание системы счисления, использованной Банкиром. Ваша задача — проверить, может ли Барристер предоставить такое основание.

Формат входного файла

В первой строке записано число N — количество тождеств (N — натуральное число от 1 до 1 000 включительно), затем в последующих N строках — тождества в формате

$$X_1 \dots X_l * Y_1 \dots Y_m = Z_1 \dots Z_k$$

или

$$X_1 \dots X_l + Y_1 \dots Y_m = Z_1 \dots Z_k,$$

где все X_i, Y_i, Z_i — цифры от 0 до 9 или латинские буквы от a до z (цифры от 10 до 35). Числа m, l, k натуральные и меньше 11. В случае, если основание восстанавливается однозначно, все числа в примерах не превосходят $2^{64} - 1$.

Формат выходного файла

Вывести основание используемой во всех данных примерах системы счисления, если такая существует и восстанавливается однозначно. Выведите `0`, если такой системы не существует, или `-1`, если система счисления не восстанавливается однозначно по данным примерам. Основание необходимо вывести в десятичной системе счисления.

Пример

input.txt	output.txt
3 2*2=4 25*25=625 10+10=20	10
1 2*2=5	0
1 2*2=4	-1

Задача E. The Vanishing

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

В играх на базе D&D (Dungeons and Dragons) параметры хода определяются с помощью набора игральных костей с разным количеством граней. На грани каждой кости нанесены натуральные числа от 1 до N, где N есть число граней данной кости. Повреждение, наносимое некоторым оружием, определяется по формуле $(M\,dN + K)$, где $M\,dN$ обозначает сумму результатов независимых бросков N-гранной кости, K — некоторое натуральное число. Например, $3\,d6 + 4$ обозначает, что 3 раза бросается стандартная игральная кость с 6 гранями, выпавшие очки суммируются, а затем к сумме прибавляется 4.

В игре Вы встретили хитрого монстра Буджум со следующими свойствами: если по нему нанести удар с повреждением ровно Q единиц, где Q — натуральное число, то Буджум будет уничтожен.

Если же будет нанесено повреждение, отличное от Q, то Буджум выживет и в свой ход сделает так, что Вы исчезнете. Однако есть и приятные новости: Вы ходите раньше Буджума.

По заданным M, N, K и Q определите, какова вероятность уничтожения Буджума данным оружием.

Формат входного файла

Параметр оружия в виде $M\,dN$+K, в следующей строке — необходимое повреждение Q. Числа M, N натуральные и не больше 30. Числа K и Q натуральные и по модулю не превосходят 1 000.

Формат выходного файла

Вероятность того, что результирующее повреждение будет равно Q, в процентах с точностью два знака после десятичной точки.

Пример

input.txt	output.txt
1d6+1 2	16.67

Задача F. Contest Must Go on!

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Перед выходом из британских территориальных вод, корабль, на котором отправились в путешествие охотники за Снарком, был остановлен для досмотра. В ходе досмотра выяснилось, что Булочник не знает своего имени, а все документы он оставил в одном из сорока двух чемоданов, которые он забыл на берегу. Поэтому, с целью выяснения личности Булочника, в порт была отправлена радиограмма. К сожалению, в тот момент, когда пришла ответная радиограмма, на месте радиста находился Бобёр, который использовал ленту для своих целей. Так что лента была приведена в частичную негодность: хотя места наличия знаков на ленте были ещё различимы, только в некоторых местах можно было разобрать, точка там или тире. Кроме того, Бобёр правильно запомнил длины последовательностей подряд идущих тире, от точки до точки, а про точки он как-то не подумал, считая их фоном узора. Восстановите радиограмму целиком или, если это невозможно, восстановите те знаки, которые восстанавливаются однозначно. Если не существует ни одной радиограммы, удовлетворяющей условиям, выведите `BEAVERROR`.

Формат входного файла

Испорченная лента на первой строке в формате $A_1 \ldots A_N$, где A_i — или точка, или тире, или 0 (обозначает нечитаемый символ). Длина ленты не превосходит 10 000 знаков. Далее, второй строкой, следует список длин последовательностей подряд идущих тире, которые запомнил Бобёр, в формате $L_1\ L_2\ \ldots\ L_K$ в порядке слева направо. Например, для радиограммы `---.--.-` список будет таким: `3 2 1`.

Формат выходного файла

Восстановленная радиограмма в формате $A_1 \ldots A_N$, где A_i есть либо точка, либо тире, либо 0 (знак не восстанавливается однозначно), или строка `BEAVERROR`.

Пример

input.txt	output.txt
`.-0.-` `2 1`	`.--.-`
`000` `2`	`0-0`
`..` `1`	`BEAVERROR`

Задача G. The Hunting

Имя входного файла: `input.txt`
Имя выходного файла: `output.txt`
Ограничение по времени: 2 сек
Ограничение по памяти: 64 мегабайта

Барристера заинтересовал один из способов поимки Снарка, который он вычитал в найденном руководстве «Snark Hunting for Dummies in 21 days». Снарка можно поймать так: «Охотники должны ночью высадиться на остров с разных сторон и сближаться до определённого момента. Затем некоторые из участников охоты бросают друг другу верёвки так, что получается замкнутый многоугольник, внутри которого и спит Снарк. После чего поимка становится делом техники, если, конечно, все смогли вести себя достаточно тихо.»

Благозвон хочет, используя этот способ, обойтись минимальным количеством верёвки: ведь её запасы на корабле ограничены, а после поимки, Снарка, возможно, придётся связывать. К тому же, существует ещё одна проблема максимальная длина, на которую можно бросить верёвку, ограничена и равна K. Вычислите минимальную длину верёвки, если заданы финальные координаты всех ловцов и координаты, можно надеяться, спящего Снарка.

Формат входного файла

В первой строке записано число участников экспедиции N — натуральное число от 3 до 100 и K — максимальная длина куска верёвки, вещественное число, большее 0 и не превосходящее 30 000. В следующей строке дана пара координат Снарка. Далее записаны N пар координат X_i, Y_i финального положения каждого из участников экспедиции, по паре координат в одной строчке.

Координаты заданы с 4 знаками после запятой и по модулю не превосходят 10 000.

Формат выходного файла

Минимальная длина требуемой верёвки с точностью два знака после запятой.

Пример

input.txt	output.txt

`3 6.00`
`1.0000 1.0000`
`0.0000 0.0000`
`0.0000 3.0000`
`4.0000 0.0000`
	`12.00`

Пятый день: практический тур

Задача A. Точки

Имя входного файла: `points.in`
Имя выходного файла: `points.out`
Ограничение по времени: 1 сек
Ограничение по памяти: 64 мегабайта

На плоскости заданы N различных точек. Требуется найти количество способов выбрать три из них, так чтобы площадь треугольника с вершинами в этих точках была целым числом. Будем считать, что три точки, лежащие на одной прямой, образуют треугольник.

Формат входного файла

Первая строка входного файла содержит число N ($1 \leqslant N \leqslant 10\,000$). Далее следуют N пар целых неотрицательных чисел, задающих координаты точек. Все координаты не превосходят 1 000.

Формат выходного файла

Выведите одно число — искомое количество троек точек.

Пример

points.in	points.out
4 0 0 0 2 2 2 2 0	4

Задача B. Робот

Имя входного файла:	`robot.in`
Имя выходного файла:	`robot.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Робот движется по полю, которое состоит из N клеток, выстроенных в ряд. На каждой из клеток находится кубик определенного цвета.

До начала движения робот находится на первой клетке поля и не держит ни одного кубика. Находясь на клетке, робот может выполнить не более одного раза каждую из следующих операций: (1) положить кубик того же цвета, который лежит на текущей клетке; (2) поднять с клетки тот кубик, который находился там сначала. После этого робот перемещается на следующую клетку или останавливается, если текущая клетка последняя в поле.

Одновременно робот может держать не более K кубиков. На момент остановки робот не должен держать ни одного кубика.

Напишите программу, которая по информации о цвете кубиков и об ограничении на количество кубиков, которое может держать робот, определяет максимальное общее количество кубиков, которое робот может перенести с места на место, двигаясь по полю.

Формат входного файла

Первая строка входного файла содержит символьную строку длины N $(1 \leqslant N \leqslant 1000)$. Строка состоит из маленьких букв латинского алфавита. Каждая буква соответствует клетке поля и определяет цвет кубика, который находится в этой клетке. Вторая строка содержит ограничение на количество кубиков, которое одновременно может держать робот K $(1 \leqslant K \leqslant 25)$.

Формат выходного файла

Единственная строка выходного файла должна содержать целое число — максимальное количество кубиков, местоположение которых робот может изменить, двигаясь по полю.

Пример

robot.in	robot.out
rgbbggrmcm 2	4

Задача C. Арифметические выражения

Имя входного файла:	`arithmetic.in`
Имя выходного файла:	`arithmetic.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Выпускная работа Васи Пупкина в школе развития связана с разработкой обучающей системы для младших школьников. Вася с жаром взялся за работу. Он решил, что в первую очередь необходимо разработать модуль, обучающий школьников вычислению арифметических выражений. Он разработал проект этой программы и понял, что одна из основных его задач — это обновление арифметических выражений, которые должны вычислять обучающиеся. Ему пришла идея разделить все арифметические выражения по уровням сложности. При этом в качестве меры сложности Вася взял длину арифметического выражения. Множество арифметических выражений заданной длины можно расположить в лексикографическом (алфавитном) порядке, занумеровать подряд, начиная с номера 1, и случайно выбирать номер арифметического выражения. Очевидно, что у Васи работы непочатый край, а Вы могли бы ему помочь. Напишите программу, реализующую идею Васи о генерации арифметических выражений по заданному порядковому номеру.

Определим арифметическое выражение четырьмя правилами:

1) x — это арифметическое выражение;

2) y — это арифметическое выражение;

3) если A и B — это арифметические выражения, то арифметическими выражениями будут: (A), (B), $A+B$, $A*B$;

4) арифметическими выражениями считаются только те, которые получены по правилам 1–3.

Определим порядок для символов, используемых в выражениях, определённых выше, следующим образом: $'x' < 'y' < '(' < ')' < '*' < '+'$.

Занумеруем все правильные записи арифметических выражений заданной длины в соответствии с лексикографическим (алфавитным) порядком.

Напишите программу, которая по номеру арифметического выражения в заданном упорядоченном множестве арифметических выражений данной длины выдаёт арифметическое выражение, соответствующее данному номеру.

Формат входного файла

В первой строке файла записаны через пробел два целых числа N и M $(1 \leqslant N \leqslant 21)$, $(1 \leqslant M \leqslant 2 \cdot 10^9)$ — длина арифметического выражения в множестве и его номер соответственно.

Формат выходного файла

В выходной файл нужно вывести арифметическое выражение, которому сопоставлен номер N в упорядоченном множестве выражений длины M. Если выражение с заданным номером не существует, то вывести 0.

Пример

arithmetic.in	arithmetic.out
3 5	y*x
3 10	(y)

Задача D. Загрузка контейнера

Имя входного файла:	container.in
Имя выходного файла:	container.out
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Контейнер, имеющий форму прямоугольного параллелепипеда, необходимо загрузить ящиками. Все ящики имеют кубическую форму и одинаковые размеры, так что ребро куба можно принять за единицу длины.

Содержимое ящиков имеет разный вес и степень хрупкости, в соответствии с которыми каждому ящику присвоен целочисленный индекс из интервала от 1 до 9.

Правила загрузки разрешают ставить несколько ящиков друг на друга в том случае, если каждый ящик несет на себе груз, суммарный индекс которого меньше собственного индекса этого ящика. Так, поставить «1 на 3 на 7» можно, потому что ящик «7» несет на себе груз

суммарного индекса 4, а ящик «3» — суммарного индекса 1. А вот поставить «1 на 2 на 3 на 8» нельзя, так как ящик «3» несет на себе груз, суммарный индекс которого равен собственному индексу.

Требуется определить, можно ли загрузить в контейнер указанный набор ящиков без нарушения правил.

Формат входного файла

Первая строка файла содержит три целых числа L, N, H, определяющих вместимость контейнера: L — длина основания, N — ширина основания и H — высота $(0 \leqslant L, N, H \leqslant 40\,000)$.

Вторая строка содержит 9 чисел, описывающих набор ящиков в следующем порядке: K_1 — число ящиков индекса 1, K_2 — число ящиков индекса 2 и т. д. до K_9 — числа ящиков индекса 9, причем $K_1 + \cdots + K_9$ не превосходит $2 \cdot 10^9$.

Формат выходного файла

В первой строке файла одно число: если загрузка возможна, то 1, в противном случае — 0.

Пример

container.in	container.out
3 4 3 15 1 0 0 0 0 0 0 0	0

Шестой день: практический тур

Задача A. Следующий!

Имя входного файла:	input.txt
Имя выходного файла:	input.out
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Отсортируем все числа 0 до N включительно по количеству единиц в двоичном представлении. Таким образом, $4 = 100_2$ идет раньше чем $3 = 11_2$, так как в двоичном представлении имеет на одну единицу меньше. В случае одинакового количества единиц раньше идет то число, которое меньше.

Пример сортировки для $N = 7$: 0, 1, 2, 4, 3, 5, 6, 7.

Даны числа N и K. Требуется найти следующее после K число в указанном выше порядке.

Формат входного файла

В первой строке входного файла содержится число N ($1 \leqslant N < 10^{100}$). Вторая строка содержит число K ($0 \leqslant K \leqslant N$).

Формат выходного файла

В выходной файл выведите следующее за K число. В случае, если K — последнее число, то выведите `-1`.

Пример

input.txt	input.out
10 4	8
12 11	-1

Задача B. Периметр

Имя входного файла:	`input.txt`
Имя выходного файла:	`input.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Дано N прямоугольников. Ширина каждого прямоугольника равна 1, а их длины — это последовательные нечетные числа без повторений, начиная с 1. Например, если $N = 5$, то длины прямоугольников — 1, 3, 5, 7, 9. Требуется определить, какой минимальный периметр P может иметь фигура, сложенная из этих прямоугольников.

Прямоугольники должны лежать параллельно осям координат и не пересекаться друг с другом. Периметр фигуры будут образовывать участки сторон прямоугольников, которые не соприкасаются со сторонами других прямоугольников.

Формат входного файла

Единственное целое число N ($1 \leqslant N \leqslant 4 \cdot 10^8$).

Формат выходного файла

Вывести одно целое число P.

Пример

input.txt	input.out
2	10

Задача C. Палиндром

Имя входного файла:	`input.txt`
Имя выходного файла:	`input.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Известно, что палиндромом называется строка, которая одинаково читается как слева направо, так и справа налево. Например, палиндромами являются строки «A», «ABA», «$ABBA$», а строки «AB», «AAB», «$ABAB$» палиндромами не являются.

Рассмотрим некоторую строку S, состоящую только из латинских букв A и B. Назовем запрещенными все строки длины n, которые состоят также только из букв A и B и содержат S в качестве подстроки. Например, если $S =$ «AB» и $n = 3$, то существует четыре запрещенных строки: «AAB», «ABA», «ABB» и «BAB». Остальные строки будет называть допустимыми.

Требуется написать программу, которая для заданной строки S длиной не более пяти символов и заданного числа n определяет количество допустимых строк длины n, которые являются палиндромами.

Формат входного файла

Первая строка входного файла содержит строку S. Длина строки S не превосходит пяти. Вторая строка содержит число n ($1 \leqslant n \leqslant 100$).

Формат выходного файла

Выведите в выходной файл одно число — количество строк длины n, которые являются палиндромами и не содержат S в качестве подстроки.

Пример

input.txt	input.out
AB 3	2

Пояснение к примеру

В приведенном примере две искомые строки — «AAA» и «BBB».

Задача D. Жадная волна

Имя входного файла:	input.txt
Имя выходного файла:	input.out
Ограничение по времени:	0,2 сек
Ограничение по памяти:	64 мегабайта

Поле размером $N \times M$ клеток заполнено целыми числами. Требуется найти на поле клетку, из которой волна, запущенная не более, чем на K итераций, покроет площадь с максимальной суммой расположенных на ней чисел.

Формат входного файла

В первой строке входного файла содержатся три целых числа N, M и K ($1 \leqslant N, M \leqslant 100$, $1 \leqslant K \leqslant N + M$). Следующие N строк содержат по M чисел, каждое из которых не превосходит 10 000 по абсолютной величине.

Формат выходного файла

Выведите четыре числа R, C, P и S, где R — номер строки, C — номер столбца, из которых следует запустить волну, P — количество итераций распространения волны, S — максимальная сумма чисел, покрытых волной. Если существует несколько вариантов ответа, то вывести любой, в котором число P минимально. $1 \leqslant P \leqslant K$.

Пример

input.txt	input.out
5 4 3 1 2 3 4 1 6 7 8 9 10 11 12 0 0 0 0 2 0 0 1	3 3 3 66

Седьмой день: практический тур

Задача A. Скобки

Имя входного файла:	brackets.in
Имя выходного файла:	brackets.out
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Правильной скобочной последовательностью (далее ПСП) называется такая последовательность, которая может быть получена с помощью следующих правил:

1. Пустая строка является ПСП.
2. Если A — ПСП, то $B = (A)$ также является ПСП.
3. Если A — ПСП, то $B = [A]$ также является ПСП.
4. Если A и B — ПСП, то $C = AB$ также является ПСП.

Длина скобочной последовательности равна количеству скобок в ней. Длина ПСП всегда является четным числом. Будем говорить, что скобочная последовательность $A = a_1 a_2 \ldots a_n$ лексикографически меньше последовательности $B = b_1 b_2 \ldots b_n$, если существует такой номер i, что $a_j = b_j$ для всех $j < i$ и $a_i < b_i$.

Известно, что '(' < ')' < '[' < ']'. Перечислим все ПСП длины 4 в лексикографическом порядке: (()), ()(), ()[], ([]), [()], [[]], []() , [][].

Ваша задача найти K-ю в лексикографическом порядке ПСП длины N. Гарантируется, что такая ПСП существует.

Формат входного файла

В первой строке находится одно целое четное число N ($1 < N \leqslant 250$). Во второй строке находится одно целое число K ($1 \leqslant K \leqslant 10^{120}$).

Формат выходного файла

Выведите в первой строке выходного файла K-ю в лексикографическом порядке ПСП длины N.

Пример

brackets.in	brackets.out
4 1	(())
4 3	()[]

brackets.in	brackets.out
4 7	[]()
6 12	()[[]]

Задача B. Калькулятор

Имя входного файла:	`calcul.in`
Имя выходного файла:	`calcul.out`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Пете нужно просуммировать N чисел на своем калькуляторе. У Петиного калькулятора есть 13 кнопок: «0», «1», «2», «3», «4», «5», «6», «7», «8», «9», «+», «=», «$». Перед началом вычислений за кнопкой «$» можно закрепить любую последовательность из цифр.

Чтобы просуммировать числа $A_1, A_2, \ldots, A_N$. Петя должен последовательно набрать эти числа на калькуляторе. После каждого (кроме последнего) числа Петя должен нажимать на кнопку «+». После последнего набранного числа Пете следует нажать кнопку «=». При наборе чисел Петя может использовать кнопку «$» (В этом случае будет введена последовательность цифр, закрепленных за кнопкой «$»). На калькуляторе можно набирать числа с ведущими нулями.

Петя не любит нажимать на кнопки. Поэтому он хочет закрепить за кнопкой «$» такую последовательность цифр, чтобы общее число нажатий на кнопки было минимальным. (Начальная инициализация кнопки «$» не учитывается в качестве нажатий). Необходимо помочь Пете найти последовательность цифр, которую следует закрепить за кнопкой «$».

Формат входного файла

В первой строке находится одно целое число — N $(1 < N \leqslant 20\,000)$. В последующих строках находятся N чисел — $A_1, A_2, \ldots, A_N$ $(0 \leqslant A_1, \ldots, A_N < 10^9)$.

Формат выходного файла

Выведите в первой строке последовательность цифр, которую следует закрепить за кнопкой «$». Второй строкой выведите минимальное число нажатий на кнопки, необходимое для суммирования $A_1, A_2, \ldots, A_N$.

Пример

calcul.in	calcul.out
2 1 23	23 4
7 100 200 300 400 500 600 700	00 21
4 1033 33 33 33	033 9

Задача C. Гном

Имя входного файла:	`dwarf.in`
Имя выходного файла:	`dwarf.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Как-то раз гному Кварку попала в руки карта сокровищ. На карте отмечено N точек, в которых может находиться клад. Все точки пронумерованы числами от 1 до N. Для каждой пары точек Кварк знает длину дороги, их соединяющей. Свои поиски Кварк начинает от точки с номером 1. Прежде чем начать свой долгий путь, хитрый гном вычеркивает точки, в которых, по его мнению, клада быть не может. Гарантируется, что точка с номером 1 никогда не бывает вычеркнута. После этого Кварк выбирает некоторый маршрут, проходящий через все оставшиеся на карте точки. Маршрут не проходит через одну и ту же точку более одного раза. Кварк может ходить только по дорогам, соединяющим невычеркнутые точки.

Кварк хочет выбрать маршрут минимальной длины. Необходимо найти такой маршрут для Кварка.

Формат входного файла

В первой строке находится одно целое число — N $(1 < N \leqslant 15)$ — количество точек, отмеченных на карте. В последующих N строках находятся расстояния между точками. В $(i+1)$-й строке находятся N целых чисел — $d_{i1}, d_{i2}, \ldots, d_{iN}$ — длины дорог от i-й точки до всех остальных. Гарантируется, что $d_{ij} = d_{ji}$, $d_{ii} = 0$ и $0 < d_{ij} < 100$. В $(N+2)$-й строке находится одно целое число Q $(1 < Q \leqslant 1\,000)$ — количество вариантов вычеркивания точек для данной карты. В последующих Q строках содержится описание вариантов вычеркивания. Описание начинается с числа C $(0 \leqslant C < N)$ — количества точек, в которых, по мнению Кварка, клада быть не может. Следующие C чисел задают номера этих точек.

Формат выходного файла

Выведите Q строк. В каждой строке выведите одно целое число — длину минимального маршрута при соответствующем варианте вычеркивания точек.

Пример

dwarf.in	dwarf.out
3 0 45 10 45 0 30 10 30 0 2 0 1 3	40 45
5 0 14 20 17 14 14 0 15 19 18 20 15 0 15 16 17 19 15 0 14 14 18 16 14 0 2 3 5 4 3 0	11 58

Задача D. Делители

Имя входного файла:	`factors.in`
Имя выходного файла:	`factors.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Необходимо составить из N $(1 \leqslant N \leqslant 6)$ цифр минимальное число, которое имеет ровно D делителей. Составленное число может иметь ведущие нули. Будем считать, что 1 имеет только один делитель, а все простые числа имеют два делителя. Гарантируется, что есть по крайней мере одна цифра, отличная от 0.

Формат входного файла

В первой строке находится два целых числа — N $(1 \leqslant N \leqslant 6)$ и D $(1 \leqslant D \leqslant 240)$. Во второй строке находится N цифр, перечисленных через пробел.

Формат выходного файла

Выведите минимальное число, составленное из данных цифр, которое имеет ровно D делителей. Ведущие нули следует опустить. Гарантируется, что ответ всегда существует.

Пример

factors.in	factors.out
4 2 1 0 7 0	17

Задача E. Пятачок

Имя входного файла:	`pig.in`
Имя выходного файла:	`pig.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Пятачок решил пойти в гости к Винни-Пуху через дремучий лес. В лес входит только одна дорога. В лесу есть опушки. На каждую опушку ведет только одна дорога, а из каждой опушки выходит ровно K дорог. Есть дороги, ведущие из опушек к домику Винни-Пуха. Чтобы запомнить свой путь, Пятачок решил его закодировать в виде числа W. Изначально $W = 0$. Как только Пятачок проходит через опушку, выбирая дорогу с номером I $(1 \leqslant I \leqslant K)$, он заменяет W на $KW + (I - 1)$. Однако Пятачок умеет считать только в P-ричной $(1 < P < 11)$ системе счисления. Необходимо по числу W в P-ричной системе счисления определить номера дорог, по которым прошел Пятачок. Известно, что первая дорога, по которой прошел Пятачок, имеет номер, отличный от 1.

Формат входного файла

В первой строке находится два целых числа — P $(2 \leqslant P \leqslant 10)$ и K $(2 \leqslant K \leqslant 100)$. Во второй строке находится число W $(1 \leqslant W_{10} \leqslant 10^4)$, записанное в P-ричной системе счисления.

Формат выходного файла

Выведите номера дорог, по которым прошел Пятачок. Эти номера нужно разделять пробелом и выводить их в десятичной системе счисления. Номера дорог выводить в порядке их прохождения.

Пример

pig.in	pig.out
10 10 1230	2 3 4 1
10 2 1230	2 1 1 2 2 1 1 2 2 2 1
8 13 265	2 1 13

Задача F. Сумма

Имя входного файла:	sum.in
Имя выходного файла:	sum.out
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Дан двумерный массив A с N строками и M столбцами. Обозначим элемент массива A, находящийся на пересечении i-й строки и j-го столбца как A_{ij}. Необходимо построить такой двумерный массив B с N строками и M столбцами, что B_{ij} равняется сумме таких A_{xy}, что $A_{ij} \leqslant A_{ji}$, $i \geqslant x$ и $j \leqslant y$.

Формат входного файла

В первой строке находится два целых числа — N $(1 \leqslant N \leqslant 50)$ и M $(1 \leqslant M \leqslant 50)$. В последующих N строках находятся по M целых чисел в каждой — соответствующие элементы массива A. Все числа по абсолютной величине не превосходят 70 000.

Формат выходного файла

Выведите N строк по M чисел в каждой: j-е число в i-й строке соответствует B_{ij}.

Пример

sum.in	sum.out
2 3 -50000 0 50000 -70000 0 70000	0 50000 50000 0 120000 70000
3 4 1 6 7 12 2 5 8 11 3 4 9 10	26 25 19 12 51 49 31 23 75 72 42 33

Восьмой день: практический тур

Задача A. Королевский пароль

Имя входного файла:	input.txt
Имя выходного файла:	output.txt
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

У Сэра Лавеласа имеется новая модель суперкомпьютера Никс, которая установлена в его семейном замке. Клавиатура у этого компьютера сконструирована как квадрат с N рядами и N клавишами в каждом ряду. Сэр Лавелас использует этот компьютер для разработки суперсекретного оружия.

Наша разведка дала задание хакеру Васе узнать детали этого проекта. Он уже получил приглашение на ежегодную вечеринку, посвящённую Дню Святого Валентина, которая традиционно проводится у Сэра Лавеласа дома.

Вася знает, что Сэр Лавелас всегда устанавливает пароль из уважения к своим предкам, которые правили королевством в средние века. Его пароль может быть набран только перемещениями по клавиатуре, которые удовлетворяют правилам движения шахматного коня.

Теперь Васе нужно оценить время для выполнения плана операции. Чтобы сделать это, ему необходимо знать число возможных паролей, которые Сэр Лавелас может установить. Напишите программу, чтобы найти это число.

Формат входного файла

Входной файл содержит два числа N и L, где $L + 1$ — длина пароля (длину пароля узнала предыдущая спецагентша Оля), $1 \leqslant N \leqslant 10\,000$, а $1 \leqslant L \leqslant 12$.

Формат выходного файла

В выходной файл выведите одно число, которое является количеством возможных паролей.

Пример

input.txt	output.txt
3 3	64

Задача B. Шестиугольник

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Правильный шестиугольник со стороной N разбит на $6N^2$ треугольников.

Ваша задача — найти число способов покрыть шестиугольник с помощью ромбиков (фигур, равных объединению двух треугольников, имеющих общую сторону).

Каждый ромбик должен покрывать в точности два треугольника, при этом все треугольники должны быть покрыты и ни один треугольник не может быть покрыт более чем одним ромбиком.

Найдите число способов покрыть шестиугольник таким образом. Например, число способов покрыть шестиугольник со стороной длины 1 равно двум.

Формат входного файла

Во входном файле записано одно число N $(1 \leqslant N \leqslant 7)$ — длина стороны шестиугольника.

Формат выходного файла

Выведите в выходной файл число способов покрыть шестиугольник ромбиками.

Пример

input.txt	output.txt
1	2
2	20

Задача C. Последовательность Фибоначчи

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Последовательность целых чисел a_1, a_2, …, a_N называется Фибоначчиевой последовательностью, если $a_i = a_{i-1} + a_{i-2}$ для всех $i = 3$, 4, …, N.

Дана последовательность целых чисел c_1, c_2, …, c_M. Вы должны найти наибольшую Фибоначчиеву подпоследовательность.

Формат входного файла

В первой строке входного файла содержится число M $(1 \leqslant M \leqslant 3\,000)$. Следующая строка содержит M целых чисел, каждое из которых по модулю не превосходит 10^9.

Формат выходного файла

В первой строке выходного файла выведите максимальную длину Фибоначчиевой подпоследовательности данной последовательности. Во второй строке выведите саму подпоследовательность.

Пример

input.txt	output.txt
10 1 1 3 -1 2 0 5 -1 -1 8	5 1 -1 0 -1 -1

Задача D. Черепашьи приколы

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Существует известная загадка для детей:

Три черепахи ползут по дороге. Одна черепаха сказала: «Две черепахи впереди меня». Другая сказала: «Две черепахи позади меня». Третья черепаха сказала: «Две черепахи впереди меня, и две черепахи позади меня». Как такое может быть?

Ответ: третья черепаха лжёт.

Итак, в этой задаче N черепах ползут вдоль дороги. Часть из них ползёт в группах, то есть они не видят членов своей группы ни впереди, ни сзади. Каждая из черепах делает утверждение вида: «a_i черепах ползёт впереди меня, и b_i черепах ползёт позади меня». Ваша задача определить минимально возможное число черепах, которые лгут.

Формализуем задачу. Пусть черепаха с i-м номером имеет координату x_i. Некоторые черепахи могут иметь одинаковые координаты. Черепаха говорит правду тогда и только тогда, когда a_i равно числу черепах с номерами j такими, что $x_j > x_i$, а b_i — числу черепах с номерами j такими, что $x_j < x_i$. В противном случае, черепаха с номером i лжет.

Формат входного файла

Первая строка входного файла содержит целое число N $(1 \leqslant N \leqslant 1\,000)$. Следующие N строк содержат пары чисел a_i, b_i $(0 \leqslant a_i, b_i \leqslant 1\,000)$.

Формат выходного файла

В выходной файл выведите целое число M — минимальное число черепах, которые должны лгать. Затем выведите M целых чисел — номера черепах, которые обманывают. Номера черепах можно выводить в любом порядке. Если существуют различные варианты множества врущих черепах, то выведите любое из них.

Пример

input.txt	output.txt
3 2 0 0 2 2 2	1 3
5 0 2 0 3 2 1 1 2 4 0	2 1 4

Задача E. Наибольшая общая возрастающая подпоследовательность

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Даны две последовательности целых чисел. Напишите программу, которая находит общую для них возрастающую подпоследовательность наибольшей длины.

Последовательность $s_1, s_2, \ldots, s_N$ длины N называется возрастающей подпоследовательностью последовательности $a_1, a_2, \ldots, a_M$ длины M, если существуют $1 \leqslant i_1 < i_2 < \cdots < i_N \leqslant M$ такие, что $a_{i_j} = s_j$ для всех $1 \leqslant j \leqslant N$, и $s_j < s_{j+1}$ для всех $1 \leqslant j < N$.

Формат входного файла

Во входном файле записаны две последовательности. Каждая последовательность описывается двумя строками следующим образом: в первой строке идет длина последовательности M $(1 \leqslant M \leqslant 500)$, во второй идут M целых чисел a_i $(-2^{31} \leqslant a_i < 2^{31})$ — члены последовательности.

Формат выходного файла

В первой строке выходного файла выведите N — длину наибольшей возрастающей подпоследовательности входных последовательностей. Во второй строке выведите саму подпоследовательность.

Пример

input.txt	output.txt
5 1 4 2 5 -12 4 -12 1 2 4	2 1 4

Девятый день: практический тур

Задача A. Приближение равносторонним треугольником

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Дан треугольник ABC. Найдите такой равносторонний треугольник $A_1B_1C_1$, чтобы $r = \max(|AA_1|, |BB_1|, |CC_1|)$ было минимально.

Формат входного файла

Три пары координат точек A, B, C.

Формат выходного файла

Три пары координат точек A_1, B_1, C_1.

Длины сторон должны отличаться не более чем в $(1 + 10^{-6})$ раз. $\max(|AA_1|, |BB_1|, |CC_1|)$ должно быть больше $r_{\min}$ не более, чем в $(1 + 10^{-6})$ раз.

Пример

input.txt	output.txt
0 0 3.0 0 0 4.0	-0.654701 -0.199359 3.154701 0.666667 0.500000 3.532692

Задача B. Минное поле

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Петя обнаружил себя стоящим на минном поле в точке A. У него есть карта минного поля, на которой указаны положения мин. Ему нужно попасть в точку B.

Для надежности он хочет пройти по такой гладкой кривой, чтобы расстояние до всех мин было больше либо равно некоторой величины R.

Найдите максимально возможное R.

Формат входного файла

Координаты точки A в первой строчке. Координаты точки B во второй строчке. Количество мин M в третьей строчке, $1 \leqslant M \leqslant 30$. В следующих M строчках координаты M мин. Все координаты — действительные числа по модулю, меньшие 100 000.

Формат выходного файла

Выведите число R с точностью до двух знаков после запятой.

Пример

input.txt	output.txt
0 0 3 3 2 1 1 2 2	1.41
-10 0 0 0 4 1 1 1 -1 -1 1 -1 -1	1.00

Задача C. Оптимальный путь

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

В дремучем лесу мальчик Петя хочет как можно скорее добраться из пункта A в пункт B. В лесу есть одна прямая тропинка. Скорость Пети по дремучему лесу — 5 км/ч, по тропинке — $5\sqrt{2}$ км/ч. Определите наименьшее время, за которое Петя может добраться из пункта A в пункт B.

Формат входного файла

Входной файл содержит восемь целых чисел, разделенных пробелами и переводами строки — координаты точек A и B и координаты двух различных точек тропинки:

X_A Y_A X_B Y_B

X_1 Y_1 X_2 Y_2

Все координаты по модулю не превосходят 1 000.

Формат выходного файла

Время в часах с точностью до двух знаков после запятой.

Пример

input.txt	output.txt
0 1 0 -1 -1 0 1 0	0.40

Задача D. Карта

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

На карте, которую Благозвон предложил экспедиции, не было нанесено земли, мелей, тропиков, полюсов и экватора. На ней был только океан и своеобразная случайная координатная сетка. Случайная координатная сетка представляет собой N отрезков. Концы отрезков могут совпадать.

Никакие два отрезка не пересекаются по отрезку. Благозвон сказал, что внутри каждой из частей, на которую отрезки сетки разбивают карту, находится по острову. Выясните, сколько островов представлено на заданной карте.

Формат входного файла

В первой строке записано число отрезков N — натуральное число, не превосходящее 1 000. Затем для каждого отрезка с новой строки через пробел записаны декартовы координаты концов отрезков — целые числа x_1, y_1, x_2, y_2, не превосходящие по модулю 100 000. Сама карта — прямоугольник с вершинами $(-142\,857, -142\,857)$, $(-142\,857, 142\,857)$, $(142\,857, 142\,857)$, $(142\,857, -142\,857)$.

Формат выходного файла

Количество частей, на которые эти отрезки разбили карту (включая «внешнюю» часть).

Пример

input.txt	output.txt
1 1 1 2 2	1
4 0 0 0 1 0 1 1 1 1 1 1 0 1 0 0 0	2

Второй день: практический тур для начинающих (структуры данных, волновой алгоритм)

Задача A. Стек

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Требуется реализовать операции для работы со стеком.

Формат входного файла

В первой строке входного файла записано одно натуральное число N — количество операций со стеком ($1 \leqslant N \leqslant 1\,000$).

В каждой из следующих N строк записано по одному неотрицательному целому число. Положительное число означает, что необходимо добавить это число в стек. 0 означает, что необходимо вынуть из стека последнее число и напечатать его; если стек пуст, то ничего делать не нужно.

Формат выходного файла

В выходной файл требуется вывести последовательность натуральных чисел, соответствующую инструкциям во входном файле.

Пример

a.in	a.out
5 3 2 0 0 0	2 3
5 3 0 0 0 3	3

Задача B. Скобки

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Требуется определить, является ли правильной данная последовательность круглых, квадратных и фигурных скобок

Формат входного файла

В единственной строке входного файла записано подряд N скобок ($1 \leqslant N \leqslant 255$).

Формат выходного файла

В выходной файл вывести `YES`, если данная последовательность является правильной, и `NO` в противном случае.

Пример

a.in	a.out
([])	YES
(({)})	NO

Задача C. Площадь комнаты

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Требуется вычислить площадь комнаты в квадратном лабиринте.

Формат входного файла

В первой строке входного файла записано число N — размер лабиринта ($3 \leqslant N \leqslant 10$). В следующих N строках задан лабиринт ('.' — пустая клетка, '*' — стенка). В следующей строке задано два числа — номер строки и столбца клетки, находящейся в комнате, площадь которой необходимо вычислить. Гарантируется, что эта клетка пустая, и что лабиринт окружен стенками со всех сторон. Строки нумеруются сверху вниз, столбцы — слева направо.

Формат выходного файла

В выходной файл нужно вывести единственное число — количество пустых клеток в данной комнате.

Пример

input.txt	output.txt
5 ***** **..* *.*.* *..** ***** 2 4	3

Задача D. Книжная полка

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	1 сек
Ограничение по памяти:	64 мегабайта

Виктор Александрович раскладывает свои книги на полку. Если на полке нет ни одной книги, то он просто ставит её, если есть, то ставит либо справа, либо слева от уже расставленных книг. Забирает книги он так же, то есть снимает только с правого или левого края.

Формат входного файла

В первой строке содержится число N ($1 \leqslant N \leqslant 10\,000$) — количество операций, которые провёл Виктор Александрович. Далее в N строках находится информация об операциях. Каждая операция постановки книги на полку описывается парой чисел. Первое из них (1 или 2) показывает, книга ставится с левого края или с правого соответственно, второе целое число (от 0 до 10 000) обозначает номер книги. Операции снятия книги с полки описывается одним числом — 3 или 4, с левого и правого края соответственно.

Формат выходного файла

Для каждой операции снятия книги вывести номер снимаемой книги.

Пример

d.in	d.out
5 1 1 2 2 1 3 3 4	3 2

Четвертый день: практический тур для начинающих (длинная арифметика)

Задача A. Длинное деление

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Даны два натуральных числа A и B. Требуется найти целую часть от их частного («A div B»).

Формат входного файла

Во входном файле записаны натуральные числа A и B по одному в строке ($A < 10^{100}, B \leqslant 10\,000$).

Формат выходного файла

В выходной файл выведите единственное число без ведущих нулей.

Пример

input.txt	output.txt
7 3	2

Задача B. Длинный корень

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

Формат входного файла

Во входном файле записано натуральное число A ($A \leqslant 10^{100}$).

Формат выходного файла

В выходной файл выведите максимальное натуральное число B, квадрат которого не превосходит A. Число B следует выводить без ведущих нулей.

Пример

input.txt	output.txt
17	4

Задача C. Антифакториал

Имя входного файла:	`input.txt`
Имя выходного файла:	`output.txt`
Ограничение по времени:	2 сек
Ограничение по памяти:	64 мегабайта

$N!$ (читается «эн-факториал») — это произведение всех натуральных чисел от 1 до N. Во входном файле записано значение $N!$ для некоторого натурального N. Требуется найти это N.

Формат входного файла

Во входном файле записано значение $N!$ Оно состоит не более чем из 255 цифр, и не содержит ведущих нулей.

Формат выходного файла

В выходной файл выведите искомое число N.

Пример

input.txt	output.txt
1	1
120	5

Часть II

Лекции и семинары

Планы лекций

День первый

Лекция D: Структуры данных–1 (Алексей Гусаков)

1) Списки. Односвязные и двусвязные. Добавление/удаление. Кольцевые списки.
2) Стеки. Стеки на массивах и списках. Примеры простейших задач на стеки.
3) Очереди. Очереди на списках. Очереди на массивах в виде циклического буфера.
4) Деки. Задачи на деки. Лабиринты.
5) Сортированные массивы. Бинарный поиск.
6) Хэш-таблицы.

Лекция C: Структуры данных–1 (Андрей Шестимеров)

1) Простые структуры данных: очереди и стеки; списковые структуры, представление списков в массиве; списки в динамической памяти.
2) Бинарный поиск, поиск k-го по величине в массиве.
3) Heap, Heap-sort и очередь с приоритетами.
4) Дерево отрезков: обзор; Init, Update, запросы; обобщение на многомерный случай.
5) Двойчные деревья поиска: поиск, добавление и удаление; сбалансированные деревья.

Лекция B: Геометрия (Дмитрий Королев)

1) Систематизация подходов к решению задач на вычислительную геометрию: метод границ, метод деления пополам, теорема Хэлли.
2) Разбиения плоскости: двоичное деление пространства (BSP-Trees), области Вороного, триангуляции.

3) Стереометрия: эффективное использование скалярного и векторного произведений в пространстве; еще один алгоритм решения задачи об объёме общей части выпуклых тел.

Лекция A: Суффиксные деревья. Алгоритм Мак-Крейта построения суффиксного дерева за линейное время (Борис Василевский)

1) Суффиксное дерево, наивный алгоритм.
2) Сжатое суффиксное дерево, детали реализации.
3) Суффиксные ссылки.
4) Общий шаг алгоритма Мак-Крейта.
5) Добавление следующего суффикса, «скачок по счетчику».
6) Сохранение структуры суффиксных ссылок.
7) Оценка количества действий $O(n)$.
8) Использование суффиксных деревьев: поиск подстроки в строке; обобщенное суффиксное дерево; количество различных подстрок в строке; наибольшая общая подстрока; поиск максимальных повторов.

День второй

Лекция D: Структуры данных–2 (Алексей Лахно)

1) Двоичные деревья поиска: основные определения, обход дерева и печать элементов в неубывающем порядке, поиск элемента по ключу, нахождение минимального и максимального элементов, нахождение следующего и предыдущего, добавление элемента, удаление элемента.
2) Вектор (саморасширяющийся массив): понятие вектора, оценка времени добавления элемента двумя способами.
3) Двоичная куча (Heap): понятие двоичной кучи и ее основное свойство, процедура Heapify поддержания основного свойства кучи, построение кучи, линейная оценка сложности, Heap Sort — сортировка с помощью кучи, приоритетная очередь: операции добавления и извлечения максимального элемента.

Лекция C: Структуры данных–2 (Андрей Шестимеров)

1) Декартовы деревья.
2) Хеш-таблицы: хеш-функции, методы разрешения коллизий, хеширование с открытой адресацией, двойное хеширование.
3) Системы непересекающихся множеств: операции, реализация с помощью массивов, реализация с помощью списков, лес непересекающихся множеств.

Лекция AB: LCA и RMQ за $O(1)$ (Александр Чернов)

1) Манипуляции с битами: обнуление/установка единичных битов справа и т. п., округление к ближайшей степени 2, подсчет единичных и нулевых битов в слове.
2) Постановка задачи LCA: Least common accessor в дереве, постановка задачи RMQ: Range Minimum Query, сведение задачи LCA к RMQ за $O(n)$, где n — число узлов в дереве.
3) Решение задачи RMQ с предвычислением за $O(n \log n)$ (n — размер массива) и стоимостью запроса $O(1)$ методом динамического программирования.
4) Задача ± 1 RMQ, решение задачи с предвычислением за $O(n)$ и стоимостью запроса $O(1)$. Таким образом, получаем алгоритм решения LCA с предвычислением $O(n)$ и запросом $O(1)$.
5) Сведение задачи RMQ к задаче LCA за $O(n)$. В итоге получаем RMQ с предвычислением $O(n)$ и запросом $O(1)$.

День третий

Лекция D: Длинная арифметика (Вадим Антонов)

1) Хранение длинных чисел.
2) Сравнение длинных чисел.
3) Арифметические операции над длинными числами: сложение, вычитание, умножение, деление длинного числа на короткое, деление длинного числа на длинное.

Лекция C: Базовые строковые алгоритмы (Борис Василевский)

1) Алгоритм Кнута-Морриса-Пратта: введение, обозначения, наивный алгоритм, префикс-функция, алгоритм КМП, интерпретация с использованием конечного автомата.

2) Алгоритм Ахо-Корасик: бор, сжатый бор, построение бора по набору слов, интерпретация с использованием конечного автомата, суффиксные ссылки, алгоритм Ахо-Корасик.

Лекция AB: Теорема Геделя о неполноте (Алексей Семенов)

День четвертый

Лекция D: Вычислительная геометрия на плоскости (Сергей Шедов)

1) Векторы: понятие вектора, координаты, операции над векторами, скалярное произведение векторов, векторное (косое) произведение векторов.

2) Нахождение полярного угла.

3) Типы данных для программирования задач вычислительной геометрии.

4) Взаимное расположение точек и фигур: принадлежность точки прямой, принадлежности точки лучу, принадлежность точки отрезку, расстояние от точки до прямой.

5) Определение факта пересечения двух отрезков.

6) Уравнение прямой.

7) Уравнение окружности.

8) Уравнение биссектрисы угла.

9) Уравнение касательной к окружности (геометрический метод).

10) Нахождение точек пересечения окружности и прямой (геометрический метод).

11) Нахождение точек пересечения двух окружностей (алгебраический и геометрический методы).

12) Нахождение площади простого многоугольника.

13) Определение принадлежности точки простому многоугольнику.

14) Примеры решения задач прошлых Всероссийских олимпиад рассмотренными методами: задачи «Лесной пожар» и «Раздел царства».

Лекция C: Перебор (Алексей Лахно)

1) Игровые задачи. Альфа-бета отсечение.

2) Задачи на перебор с возвратом. Метод ветвей и границ.

3) Отсечение по времени на C и Паскале.

4) Оптимизационные задачи и эвристические методы их решения: метод локальных улучшений, случайный поиск, метод отжига.

Лекция AB: Игры и стратегии (Антон Фонарев)

1) Игры на графах: выигрышные/проигрышные позиции, игры на ациклических графах, игры на циклических графах, ничья.

2) Число Спрага-Гранди: определение, теорема Спрага-Гранди, примеры (Grundy's game, Withoff's game, Euclid, Nim, Hackenbush).

3) Теория игр: антагонистические игры в нормальной форме, матричные игры, максмин и минимакс, значение игры и оптимальные стратегии, смешанные стратегии, существование решения матричной игры в смешанных стратегиях, итеративный метод решения матричных игр.

4) Позиционные игры n игроков в общем случае: существование решения конечной игры n лиц с полной информацией (центральная теорема).

День пятый

Лекция D: Динамическое программирование (Александр Мамонтов)

1) Общие понятия (что такое динамика, для чего и когда используется).

2) Принцип динамики (подзадача, разбиение задачи на подзадачи).

3) Примеры и пояснения: числа Фибоначи, задача «Гвоздики», биномальные коэффициенты, минимальный путь в таблице, наибольшая возрастающая подпоследовательность, задача «лесенки», восстановление скобок.

Лекция C: Геометрия (Сергей Шедов)

1) Повторение базовых понятий вычислительной геометрии на плоскости (векторы, скалярное и векторное (косое) произведение векторов, полярный угол, способы описания прямой и окружности на плоскости).

2) Особенности программирования задач вычислительной геометрии.

3) Определение взаимного расположения геометрических объектов и нахождение точек пересечения: расположение точки относительно прямой, луча или отрезка, взаимное расположение прямых, отрезков, лучей, взаимное расположение окружности и прямой, взаимное расположение двух окружностей.

4) Многоугольники: проверка выпуклости многоугольника, проверка принадлежности точки внутренней области простого многоугольника, вычисление площади простого многоугольника, построение выпуклой оболочки для множества из N точек плоскости, собые точки многоугольников и множеств N точек плоскости.

5) Эффективный алгоритм нахождения ближайшей пары из N точек плоскости.

6) Примеры задач всероссийских олимпиад прошлых лет на методы вычислительной геометрии.

Лекция AB: Языки и автоматы (Александр Шень)

1) Конечные автоматы: определение детерминированного конечного автомата (DFA), автомат, распознающий язык, определение и примеры регулярных языков, недетерминированный конечный автомат (NFA), детерминизация NFA.

2) Операции над автоматами: регулярность дополнения к регулярному языку, регулярность конкатенации двух регулярных языков, регулярность итерации регулярного языка.

День шестой

Лекция D: Графы - 1 (Алексей Лахно)

1) Понятие графа. Основные определения.

2) Способы представления графов.

3) Алгоритмы нахождения минимального остовного дерева: алгоритм Прима, алгоритм Краскала.

4) Поиск в глубину и его применение: общая схема, компоненты связности, топологическая сортировка, другие задачи, решаемые с помощью поиска в глубину: поиск мостов, точек сочленения, компонент сильной связности.

5) Эйлеровы циклы.

Лекция C: Графы (Алексей Тимофеев)

1) Неориентированные графы: точки сочленения, компоненты вершинной двусвязности, мосты, компоненты реберной двусвязности.

2) Ориентированные графы: топологическая сортировка, компоненты сильной связности, алгоритм Дейкстры (простая реализация, релизация с помощью кучи), алгоритм Флойда.

Лекция AB: Потоки минимальной стоимости, задача о назначениях (Антон Фонарев)

1) Потоки в сетях: определения, обозначения.

2) Задача о максимальном потоке: остаточные сети, дополняющие пути, разрезы, теорема Форда-Фалкерсона, метод Форда-Фалкерсона.

3) Реализация метода Форда-Фалкерсона: алгоритм Эдмондса-Карпа, алгоритм масштабирования.

4) Поток заданной величины минимальной стоимости: постановка задачи, согласованная декомпозиция, алгоритм нахождения потока заданной величины минимальной стоимости, потенциалы Джонсона.

5) Задача о назначениях: венгерский алгоритм, нормальное решение.

День седьмой

Семинар D: Задачи на графы (Маргарита Трухина)

Семинар C: Задачи на графы (Виктор Матюхин, Александр Чернов)

Лекция AB: Динамика по профилю (Борис Василевский)

1) Задача о замощении домино: профили, базовая линия, рекуррентное соотношение для задачи о замощении, решение задачи о замощении с использованием ДП по профилю, связь ДП по профилю и линейной алгебры, рекуррентное соотношение в ДП по профилю как умножение матрицы на вектор, быстрое возведение в степень, использование быстрого возведения матрицы в степень для существенно несимметричных задач.

2) Задача о симпатичных узорах.

3) Задача о расстановке королей: постановка задачи, эффективная нумерация профилей.

4) Задача о расстановке коней: постановка задачи, эффективное хранение разреженных матриц, графовая интерпретация ДП по профилю.

5) Быстрое ДП по профилю: изломанный профиль, правила перехода в быстром ДП по профилю для задачи о замощении домино.

День восьмой

Лекция CD: Динамическое программирование (Денис Кириенко)

1) Что такое динамика. Когда нужна, а когда не нужна динамика.

2) Основные применения динамики: подсчет количества объектов (пример: числа Каталана), нахождение минимального (максимального) решения (пример: задача «Банкомат»), позиционные игры (пример: задача «Игра с датами»).

3) Задача «Ход конем».

4) Задача «Восстановление скобок»: динамика по длине и балансу.

5) Задача «Пьяный студент»: динамика по длине и балансу.

6) Задача быстрого нахождения k-й правильной скобочной последовательности.

7) Задача рюкзака и связные задачи (выбор камней по заданной суммарной массе, задача «Копилка»).

8) Задача умножения большого числа матриц — динамика по интервалам $A(i..j)$.

9) Задача получения максимального палиндрома из данной строки (динамика по подстрокам).

10) Нахождение наибольшей общей подпоследовательности.

11) Нахождение наибольшей возрастающей подпоследовательности за время $O(n^2)$ и $O(n \log n)$.

12) Динамика от трёх и более параметров: задача покупки товаров с системой скидок.

13) Динамика «сверху вниз» (рекурсия + memorization).

Лекция B: использование библиотеки STL (Дмитрий Королев)

Динамическое программирование

Д. Кириенко

Нахождение числа сочетаний

Рассмотрим простейшую комбинаторную задачу: сколькими способами можно из данных $n \geqslant 0$ предметов выбрать некоторые k предметов ($0 \leqslant k \leqslant n$), если порядок их выбора не важен? Ответом на эту задачу является величина $C_n^k = \frac{n!}{k!(n-k)!}$, называемая *числом сочетаний из n элементов по k*. Запись $n!$ обозначает произведение $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, называемое *факториалом* числа n, при этом считается, что $0! = 1$.

Эту формулу несложно вывести. Расставить n предметов в ряд можно $n!$ способами. Рассмотрим все $n!$ расстановок и будем считать, что первые k предметов в ряду мы выбрали, а оставшиеся $n-k$ предметов — нет. Тогда каждый способ выбрать k предметов мы посчитали $k!(n-k)!$ раз, поскольку k выбранных предметов можно переставить $k!$ способами, а $n-k$ невыбранных — $(n-k)!$ способами. Поделив общее количество перестановок на количество повторов каждой выборки, получим число выборок: $C_n^k = \frac{n!}{k!(n-k)!}$.

Наивный метод

Как же вычислить значение C_n^k по данным n и k? Воспользуемся выведенной формулой. Функция `factorial(n)` возвращает факториал числа `n`, а функция `C(n,k)` возвращает значение C_n^k.

```
int factorial(int n)
{
    int f=1;
    for(int i=2;i<=n;++i)
        f=f*i;
    return f;
}

int C(int n, int k)
{
    return factorial(n)/factorial(k)/factorial(n-k);
}
```

Данный алгоритм понятен, очень быстр (его вычислительная сложность $O(n)$, поскольку вычисление значения $n!$ требует $n-1$ умножение), использует ограниченный размер дополнительной памяти. Но у

него есть существенный недостаток: он будет корректно работать только для небольших значений n и k. Дело в том, что величина $n!$ очень быстро растет с увеличением n, например, значение $13! = 6\,227\,020\,800$ превосходит максимальное возможное значение, которое можно записать в 32-разрядном целом числе, а величина

$$21! = 51\,090\,942\,171\,709\,440\,000$$

не помещается в 64-разрядном целом числе. Поэтому пользоваться данной функцией можно только для $n \leqslant 12$ при использовании 32-битной арифметики или для $n \leqslant 20$ при использовании 64-битной арифметики. При этом само значение C_n^k может быть невелико, но при проведении промежуточных вычислений факториалов может возникнуть переполнение. Например, $C_{30}^{15} = 155117520$ можно записать с использованием 32-битной арифметики, однако, значение $30!$ при этом не поместится и в 64-разрядном целом числе и вычислить значение C_{30}^{15} таким методом не удастся.

Рекурсивный метод

Проблем с переполнением можно избежать, если воспользоваться для вычисления известной формулой: $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ (при $0 < k < n$).

Действительно, выбрать из n предметов k можно C_n^k способами. Пометим один из данных n предметов. Тогда все выборки можно разбить на две группы: в которые входит помеченный предмет (их будет C_{n-1}^{k-1}) и не содержащие помеченного предмета (таких выборок будет C_{n-1}^k).

Добавив к этой формуле краевые значения: $C_n^n = C_n^0 = 1$ (выбрать все предметы или не выбрать ни одного предмета можно единственным способом), можно написать рекурсивную функцию вычисления числа сочетаний:

```
int C(int n, int k)
{
    if (k==0 || k==n)
        return 1;
    else
        return C(n-1,k-1)+C(n-1,k);
}
```

При таком решении задачи проблем с переполнением не возникнет: если значение конечного ответа не приводит к переполнению, то поскольку при его нахождении мы суммируем меньшие натуральные числа, все промежуточные значения, возвращаемые функцией `C(n,k)` тоже не будут приводить к переполнению, и такая программа, например, сможет вычислить значение C_{30}^{15}.

Но это решение крайне плохо тем, что работать оно будет очень долго, поскольку функция `C(n,k)` будет вызываться многократно для одних и тех же значений параметров `n` и `k`. Например, если мы вызовем функцию `C(30,15)`, то она вызовет функции `C(29,14)` и `C(29,15)`. Функция `C(29,14)` вызовет функции `C(28,13)` и `C(28,14)`, а `C(29,15)` вызовет функции `C(28,14)` и `C(28,15)`. Мы видим, что функция `C(28,14)` будет вызвана дважды. С увеличением глубины рекурсии количество повторяющихся вызовов функции быстро растет: функция `C(27,13)` будет вызвана три раза (дважды ее вызовет функция `C(28,14)` и еще один раз ее вызовет `C(28,13)` и т. д.

При этом каждая функция `C(n,k)` может либо вернуть значение 1, либо вернуть сумму результатов двух других рекурсивных вызовов, а, значит, любое значение, которое вернула функция `C(n,k)` получается сложением в различных комбинациях чисел 1, которыми заканчиваются все рекурсивные вызовы. Значит, при вычислении C_n^k инструкция `return 1` в приведенной программе будет выполнена ровно C_n^k раз, то есть сложность такого рекурсивного алгоритма для вычисления C_n^k не меньше, чем само значение C_n^k.

Метод динамического программирования

Итак, одна из проблем рекурсивных алгоритмов (и эта проблема возникает весьма часто, не только в рассмотренном примере) — длительная работа рекурсии за счет повторяющихся вызовов рекурсивной функции для одного и того же набора параметров. Чтобы не тратить машинное время на вычисление значений рекурсивной функции, которые мы уже когда-то вычислили, можно сохранить эти значения в массиве и вместо рекурсивного вызова функции мы будем брать это значение из массива. В задаче вычисления числа сочетаний создадим двумерный массив `B` и будем в элементе массива `B[n][k]` хранить величину C_n^k. В результате получим следующий массив (по строчкам — значения n, начиная с 0, по столбцам — значения k, начиная с 0, каждый элемент массива `B[n][k]` равен сумме двух элементов: стоящего непосредственно над ним `B[n-1][k]` и стоящего над ним слева `B[n-1][k-1]`).

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
1	6	15	20	15	6	1

Полученная числовая таблица, составленная из значений C_n^k, в которой каждый элемент равен сумме двух элементов, стоящих над ним, называется *треугольником Паскаля*.

Поскольку каждый элемент этого массива равны сумме двух элементов, стоящих в предыдущей строке, то этот массив нужно заполнять по строкам. Соответствующая функция, заполняющая массив и вычисляющая необходимое число сочетаний может быть такой: [1]

```
int C(int n, int k)
{
    int B[n+1][n+1];       // Создаем массив B из n+1 строки
    for(int i=0;i<=n;++i) // Заполняем i-ю строку массива
    {
       B[i][0]=1;         // На концах строки стоят единицы
       B[i][i]=1;
       for(int j=1;j<i;++j)
       {   // Заполняем оставшиеся элементы i-й строки
           B[i][j]=B[i-1][j-1]+B[i-1][j];
       }
    }
    return B[n][k];
}
```

Приведенный алгоритм для вычисления C_n^k требует объем памяти $O(n^2)$ и такая же его временная сложность (для заполнения одного эле-

[1] В приведенном тексте программы используется объявление `int B[n+1][k+1]` для создания двумерного массива целых чисел размера $(n+1)\times(k+1)$. Такое объявление не соответствует стандартам языков C и C++, но оно допускается в компиляторе GNU C++, поэтому, ввиду простоты такой записи, мы будем везде использовать подобное обозначение для создания динамических массивов.

Вместо такой записи можно использовать массивы фиксированного размера, например, `int B[101][101]`, если величина n не превышает 100 в решамой задаче, либо использовать динамическое распределение памяти при помощи указателей и функций `malloc` и `free` (в языке C++ используются операторы `new` и `delete`). Также в языке C++ можно использовать библиотеку STL: `vector < vector <int> > B ( n+1, vector <int> (k+1) )`.

мента массива требуется $O(1)$ операций, всего же элементов в массиве $O(n^2)$).

Подобный метод решения, когда одна задача сводится к меньшим подзадачам, вычисленные же ответы для меньших подзадач сохраняются в массиве или иной структуре данных, называется *динамическим программированием.* В рассмотренной задаче вычисления числа сочетаний использование метода динамического программирования вместо рекурсии позволяет существенно уменьшить время выполнения программы за счет некоторого увеличения объема используемой памяти.

Тем не менее, приведенный алгоритм тоже имеет небольшие недостатки. Мы вычисляем значения всех элементов последней строки, хотя нам необходим только один из них — `B[n][k]`. Аналогично, в предпоследней строке нас интересуют только два элемента — `B[n-1][k-1]` и `B[n-1][k]`, в строке, стоящей над ней — три элемента, и т. д., в то время, как мы вычисляем все элементы во всех строках. Кроме того, мы не используем почти половину созданного массива `B[n+1][n+1]`, а именно все элементы, стоящие выше главной диагонали. От этих недостатков можно избавиться, более того, можно уменьшить объем используемой памяти до $O(n)$, идея для построения такого алгоритма будет изложена в разделе «Маршруты на плоскости».

Если же в программе часто возникает необходимость вычисления числа сочетаний C_n^k для каких-то ограниченных значений n, то лучше всего создать глобальный массив для хранения всевозможных значений C_n^k для нужных нам значений n, заполнить его в начале программы, а затем сразу же брать значение числа сочетаний из этого массива, не вычисляя его каждый раз заново.

Промежуточные итоги

Составим план решения задачи методом динамического программирования.

1) Записать то, что требуется найти в задаче, как функцию от некоторого набора аргументов (числовых, строковых или еще каких-либо).

2) Свести решение задачи для данного набора параметров к решению аналогичных подзадач для других наборов параметров (как правило, с меньшими значениями). Если задача несложная, то полезно бывает выписать явное рекуррентное соотношение, задающее значение функции для данного набора параметров.

3) Задать начальные значения функции, то есть те наборы аргументов, при которых задача тривиальна и можно явно указать значение функции.

4) Создать массив (или другую структуру данных) для хранения значений функции. Как правило, если функция зависит от одного целочисленного параметра, то используется одномерный массив, для функции от двух целочисленных параметров — двумерный массив и т. д.

5) Организовать заполнение массива с начальных значений, определяя очередной элемент массива при помощи выписанного на шаге 2 рекуррентного соотношения или алгоритма.

Далее мы рассмотрим несколько типичных задач, решаемых при помощи динамического программирования.

Маршруты на плоскости

Все задачи этого раздела будут иметь общее начало.

Дана прямоугольная доска размером $n \times m$ (n строк и m столбцов). В левом верхнем углу этой доски находится шахматный король, которого необходимо переместить в правый нижний угол.

Количество маршрутов

Пусть за один ход королю разрешается передвинуться на одну клетку вниз или вправо. Необходимо определить, сколько существует различных маршрутов, ведущих из левого верхнего в правый нижний угол.

Будем считать, что положение короля задается парой чисел (a, b), где a задает номер строки, а b — номер столбца. Строки нумеруются сверху вниз от 0 до $n-1$, а столбцы — слева направо от 0 до $m-1$. Таким образом, первоначальное положение короля — клетка $(0, 0)$, а конечное — клетка $(n-1, m-1)$.

Пусть $W(a, b)$ — количество маршрутов, ведущих в клетку (a, b) из начальной клетки. Запишем рекуррентное соотношение. В клетку (a, b) можно прийти двумя способами: из клетки $(a, b-1)$, расположенной слева, и из клетки $(a-1, b)$, расположенной сверху от данной. Поэтому количество маршрутов, ведущих в клетку (a, b), равно сумме количеств маршрутов, ведущих в клетку слева и сверху от нее. Получили рекуррентное соотношение:

$$W(a, b) = W(a, b-1) + W(a-1, b).$$

Это соотношение верно при $a > 0$ и $b > 0$. Зададим начальные значения: если $a = 0$, то клетка расположена на верхнем краю доски и прийти в нее можно единственным способом — двигаясь только влево, поэтому $W(0,b)=1$ для всех b. Аналогично, $W(a,0)=1$ для всех a.

Создадим массив W для хранения значений функции, заполним первую строку и первый столбец единицами, а затем заполним все остальные элементы массива. Поскольку каждый элемент равен сумме значений, стоящих слева и сверху, заполнять массив W будем по строкам сверху вниз, а каждую строку — слева направо.

```
int W[n][m];
int i,j;
for(j=0;j<m;++j)
    W[0][j]=1;          // Первая строка заполнена единицами
for(i=1;i<n;++i)        // i меняется от 1 до n-1
{                       // Заполняем строку с номером i
    W[i][0]=1;          // Элемент в первом столбце равен 1
    for(j=1;j<m;++j)    // Заполняем остальные элементы строки
        W[i][j]=W[i][j-1]+W[i-1][j];
}
```

В результате такого заполнения получим следующий массив (пример для $n=4$, $m=5$):

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35

Легко видеть, что в результате получился треугольник Паскаля, записанный в немного другом виде, а именно, значение W[n-1][m-1] равно C_{n+m-2}^{n-1}. Ничего удивительного, ведь выписанное нами рекуррентное соотношение для количества маршрутов очень похоже на ранее выписанное соотношение для числа сочетаний.

Действительно, чтобы попасть из клетки $(0,0)$ в клетку $(n-1,m-1)$ король должен сделать $n+m-2$ хода, из которых $n-1$ ход вниз и $m-1$ ход вправо. Поэтому количество различных маршрутов, ведущих из начальной клетки в конечную равно количеству способов выбрать из общего числа $n+m-2$ ходов $n-1$ ход вниз, то есть C_{n+m-2}^{n-1} (и эта же величина равна C_{n+m-2}^{m-1} — количеству выборок $m-1$ хода вправо).

Поскольку для вычисления очередной строки массива W нам необходима только предыдущая строка, более того, для вычисления одного элемента очередной строки нам необходим только один элемент

предыдущей строки, стоящий непосредственно над ним, то мы можем обойтись одномерным массивом вместо двумерного, если будем хранить только одну строку двумерного массива, а именно, ту строку, в которой находится рассматриваемый в данный момент элемент. Получим следующую программу:

```
int W[m];               // Массив элементов одной строки
int i,j;
for(j=0;j<m;++j)
  W[j]=1;               // Самая первая строка состоит из единиц
for(i=1;i<n;++i)
{ // Теперь записываем в массив W содержимое i-й строки
    for(j=1;j<m;++j)      // Заполняем все элементы строки,
      W[j]=W[j-1]+W[j];   // кроме первого
}
```

После окончания работы этого алгоритма элемент массива W[m-1] содержит искомое число путей.

Упражнение. Напишите программу, вычисляющую значение C_n^k при помощи динамического программирования и использующую одномерный массив из $1+\min(k,n-k)$ элементов.

Как решить задачу нахождения количества маршрутов, если король может передвигаться на одну клетку вниз, вправо или по диагонали вправо-вниз? Решение будет полностью аналогичным, только реккурентная формула для количества маршрутов изменится:

$$W(a,b)=W(a,b-1)+W(a-1,b)+W(a-1,b-1).$$

Полученный в результате заполнения по такой формуле массив W будет выглядеть следующим образом:

1	1	1	1	1
1	3	5	7	9
1	5	13	25	41
1	7	25	63	129

Упражнение. Решите эту задачу с использованием одномерного массива.

Количество маршрутов с препятствиями

Пусть некоторые клетки на доске являются «запретными»: король не может ходить на них. Карта запретных клеток задана при помощи массива `Map[n][m]`: нулевое значение элемента массива означает, что данная клетка запрещена, единичное значение означает, что в клетку можно ходить. Массив `Map` считывается программой после задания значений `n` и `m`. Король может ходить только вниз или вправо.

Для решения этой задачи придется изменить рекуррентное соотношение с учетом наличия запрещенных клеток. Для запрещенной клетки количество ведущих в нее маршрутов будем считать равным 0. Получим:

$$W(a,b)=\begin{cases}W(a-1,b)+W(a,b-1), & \text{если } \texttt{Map[}a\texttt{][}b\texttt{]}=1,\\ 0, & \text{если } \texttt{Map[}a\texttt{][}b\texttt{]}=0.\end{cases}$$

Также надо учесть то, что для клеток верхней строки и левого столбца эта формула некорректна, поскольку для них не существует соседней сверху или слева клетки.

Например, если $n=4$, $m=5$ и массив `Map` задан следующим образом:

1	1	1	0	1
1	1	1	1	1
1	1	0	1	1
1	1	1	1	1

то искомый массив `W` будет таким:

1	1	1	0	0
1	2	3	3	3
1	3	0	3	6
1	4	4	7	13

Упражнение. Решите эту задачу с использованием одномерного массива. Массив `Map` при этом считывается построчно и в памяти хранится только последний считанный элемент.

Путь максимальной стоимости

Пусть каждой клетке (a,b) доски приписано некоторое число $P(a,b)$ — стоимость данной клетки. Проходя через клетку, мы получаем сумму, равную ее стоимости. Требуется определить максимально возможную сумму, которую можно собрать по всему маршруту, если разрешается передвигаться только вниз или вправо.

Будем решать данную задачу методом динамического программирования. Пусть $S(a,b)$ — максимально возможная сумма, которую можно собрать на пути из начальной клетки в клетку (a,b). Поскольку в клетки верхней строки и левого столбца существует единственно возможный маршрут из начальной клетки, то для них величина $S(a,b)$ определяется как сумма стоимостей всех клеток на пути из начальной вершины в данную (а именно, для начальной клетки $S(0,0)=P(0,0)$, для клеток левого столбца $S(a,0)=S(a-1,0)+P(a,0)$, для клеток верхней строки $S(0,b)=S(0,b-1)+P(0,b)$). Мы задали граничные значения для функции $S(a,b)$.

Теперь построим рекуррентное соотношение. Пусть (a,b) — клетка, не лежащая в первой строке или первом столбце, то есть $a>0$ и $b>0$. Тогда прийти в данную клетку мы можем либо слева, из клетки $(a,b-1)$, и тогда мы сможем набрать максимальную сумму $S(a,b-1)+P(a,b)$, либо сверху, из клетки $(a-1,b)$, тогда мы сможем набрать сумму $S(a-1,b)+P(a,b)$. Естественно, из этих двух величин необходимо выбрать наибольшую:

$$S(a,b)=P(a,b)+\max(S(a,b-1),S(a-1,b)).$$

Дальнейшая реализация алгоритма не вызывает затруднений:

```
int i,j;
S[0][0]=P[0][0];
for(j=1;j<m;++j)
    S[0][j]=S[0][j-1]+P[j];     // Заполняем первую строку
for(i=1;i<n;++i)
{                               // Заполняем i-ю строку
  S[i][0]=S[i-1][0]+P[i][0];  // Заполняем 1-й столбец
  for(j=1;j<m;++j)            // Заполняем остальные столбцы
    if ( S[i-1][j] > S[i][j-1] ) // Выбираем максимум
      S[i][j]=P[i][j]+S[i-1][j]; // из S[i-1][j] и S[i][j-1]
    else                         // и записываем в S[i][j]
      S[i][j]=P[i][j]+S[i][j-1]; // с добавлением P[i][j]
}
```

Рассмотрим пример доски при $n=4$, $m=5$. Для удобства в одной таблице укажем значения обоих массивов: до дробной черты указана стоимость данной клетки, то есть значение `P[i][j]`, после черты — максимальная стоимость маршрута, ведущего в данную клетку, то есть величина `S[i][j]`:

1 / 1	3 / 4	5 / 9	0 / 9	3 / 12
0 / 1	1 / 5	2 / 11	4 / 15	1 / 16
2 / 3	4 / 9	3 / 14	3 / 18	2 / 20
3 / 6	1 / 10	2 / 16	4 / 22	3 / 25

Итак, максимальная стоимость пути из левого верхнего в правый нижний угол в этом примере равна 25.

Как и все предыдущие, данную задачу можно решить с использованием одномерного массива S.

Как изменить данный алгоритм, чтобы он находил не только максимально возможную стоимость пути, но и сам путь? Заведем массив Prev[n][m], в котором для каждой клетки будем хранить ее предшественника в маршруте максимальной стоимости, ведущем в данную клетку: если мы пришли в клетку слева, то запишем в соответствующий элемент массива Prev значение 1, а если сверху — значение 2. Чтобы не путаться, обозначим данные значения константами:

```
const L=1; // Left
const U=2; // Upper
```

В начальную клетку запишем 0, поскольку у начальной клетки нет предшественника. В остальные клетки первой строки запишем L, а в клетки первого столбца запишем U. Остальные элементы массива Prev будем заполнять одновременно с массивом S:

```
if ( S[i-1][j] > S[i][j-1] )
{   // В клетку (i,j) приходим сверху из (i-1,j)
    S[i][j]=P[i][j]+S[i-1][j];
    Prev[i][j]=U;
}
else
{   // В клетку (i,j) приходим слева из (i,j-1)
    S[i][j]=P[i][j]+S[i][j-1];
    Prev[i][j]=L;
}
```

В результате, в приведенном выше примере мы получим следующий массив (значения массива Prev мы запишем в той же таблице после второй черты дроби, то есть в каждой клетке теперь записаны значения P[i][j] / S[i][j] / Prev[i][j]). Клетки, через которые проходит маршрут максимальной стоимости, выделены жирным шрифтом:

1 / 1 / 0	**3 / 4 / L**	**5 / 9 / L**	0 / 9 / L	3 / 12 / L
0 / 1 / U	1 / 5 / U	**2 / 11 / U**	**4 / 15 / L**	1 / 16 / L
2 / 3 / U	4 / 9 / U	3 / 14 / U	**3 / 18 / U**	2 / 20 / L
3 / 6 / U	1 / 10 / U	2 / 16 / U	**4 / 22 / U**	**3 / 25 / L**

После заполнения всех массивов мы можем восстановить путь, пройдя по нему с конца при помощи цикла, начав с конечной клетки и передвигаясь влево или вверх в зависимости от значения соответствующего элемента массива Prev:


```
i=n-1;                      // (i,j) - координаты текущей точки
j=m-1;                      // начинаем с конечной клетки
while (Prev[i][j]!=0)       // проверка, не дошли ли до начальной
{
    if(Prev[i][j]==L)
        j=j-1;              // передвигаемся влево
    else
        i=i-1;              // передвигаемся вверх
}
```

При этом мы получим путь, записанный в обратном порядке. Несложную задачу обращения этого пути и вывода его на экран оставим для самостоятельного решения.

Можно обойтись и без дополнительного массива предшественников Prev. Если имеется заполненный массив S, то предшественника каждой клетки мы можем легко определить, сравнив значения элемента массива S для ее левого и правого соседа и выбрав ту клетку, для которой это значение наибольшее.

```
i=n-1;
j=m-1;
while ( i>0 && j>0 )
{
    if ( S[i][j-1] > S[i-1][j] )
       j=j-1;
    else
       i=i-1;
}
```

Этот алгоритм заканчивается, когда мы выйдем на левый или верхний край доски (цикл while продолжается, пока i>0 и j>0, то есть клетка не лежит на верхнем или левом краю доски), после чего нужно будет двигаться по краю влево или вверх, пока не дойдем до начальной клетки. Это можно реализовать двумя циклами (при этом из этих двух циклов выполняться будет только один, поскольку после выполнения предыдущего фрагмента программы значение одной из переменных i или j будет равно 0):

```
while(i>0)
{   // Движение вверх, пока возможно
    i=i-1;
}
while(j>0)
```

```
{   // Движение влево, пока возможно
    j=j-1;
}
```

Таким образом, если мы хотим восстановить путь наибольшей стоимости, необходимо либо полностью сохранять в памяти значения всего массива `S`, либо хранить в памяти предшественника для каждой клетки. Для восстановления пути потребуется память порядка $O(nm)$. Сложность алгоритма нахождения пути максимальной стоимости также имеет порядок $O(nm)$ и, очевидно, не может быть уменьшена, поскольку для решения задачи необходимо использование каждого из nm элементов данного массива `P`.

Числа Каталана

Рассмотрим произвольное арифметическое выражение. Теперь сотрем в этом выражении всё кроме скобок. Получившуюся последовательность из скобок будем называть «правильной скобочной последовательностью». Любая правильная скобочная последовательность состоит из равного числа открывающихся и закрывающихся скобок. Но этого условия недостаточно: например, скобочная последовательность «(())» является правильной, а последовательность «)()(» — нет.

Можно дать и точное определение правильной скобочной последовательности.

1) Пустая последовательность (то есть не содержащая ни одной скобки) является правильной скобочной последовательностью.

2) Если «A» — правильная скобочная последовательность, то «(A)» (последовательность A, взятая в скобки) — правильная скобочная последовательность.

3) Если «A» и «B» — правильные скобочные последовательности, то «AB» (подряд записанные последовательности A и B) — правильная скобочная последовательность.

Обозначим количество правильных скобочных последовательностей длины $2n$ (то есть содержащих n открывающихся и n закрывающихся скобок) через C_n и решим задачу нахождения C_n по заданной величине n.

Для небольших n значения C_n несложно вычислить полным перебором. $\mathrm{C}_0 = 1$ (правильная скобочная последовательность длины 0 ровно одна — пустая), $\mathrm{C}_1 = 1$ (последовательность «()»), $\mathrm{C}_2 = 2$ (последовательности «(())» и «()()»), $\mathrm{C}_3 = 5$ («((()))», «(()())», «(())()», «()(())», «()()()»).

Запишем рекуррентную формулу для C_n. Пусть X — произвольная правильная скобочная последовательность длины $2n$. Она начинается с открывающейся скобки. Найдем парную ей закрывающуюся скобку и представим последовательность X в виде:

$$X = (A)B,$$

где A и B — тоже правильные скобочные последовательности. Если длина последовательности A равна $2k$, то последовательность A можно составить C_k способами. Тогда длина последовательности B равна $2(n-k-1)$ и последовательность B можно составить C_{n-k+1} способами. Комбинация любого способа составить последовательность A с любым способом составить последовательность B даст новую последовательность X, а величина k может меняться от 0 до $n-1$. Получили рекуррентное соотношение:

$$\mathrm{C}_n = \mathrm{C}_0\mathrm{C}_{n-1} + \mathrm{C}_1\mathrm{C}_{n-2} + \mathrm{C}_2\mathrm{C}_{n-3} + \ldots + \mathrm{C}_{n-2}\mathrm{C}_1 + \mathrm{C}_{n-1}\mathrm{C}_0.$$

Напишем функцию, вычисляющую значение C_n по данному n:

```
int Catalan(int n)
{
    int C[n+1];   // Создаем массив для хранения C[m]
    C[0]=1;
    for (int m=1; m<=n; ++m) // Вычисляем C[m] для m=1..n
    {
        C[m]=0;
        for (int k=0; k<m; ++k)
            C[m]+=C[k]*C[m-1-k];
    }
    return C[n];
}
```

Мы назвали функцию `Catalan`, поскольку значения C_n называются *числами Каталана* в честь бельгийского математика XIX века Евгения Шарля Каталана. Начало ряда чисел Каталана выглядит следующим образом:

n	0	1	2	3	4	5	6	7	8	9	10
C_n	1	1	2	5	14	42	132	429	1 430	4 862	16 796

Для чисел Каталана хорошо известна и нерекуррентная формула:

$$\mathrm{C}_n = \frac{\mathrm{C}_{2n}^n}{n+1} = \frac{(2n)!}{n!(n+1)!}.$$

Более подробно про правильные скобочные последовательности а также элементарное доказательство данной формулы можно прочесть в [1, 2.6–7].

Задача «Банкомат»

Рассмотрим следующую задачу. В обороте находятся банкноты k различных номиналов: a_1, a_2, …, a_k рублей. Банкомат должен выдать сумму в N рублей при помощи минимального количества банкнот или сообщить, что запрашиваемую сумму выдать нельзя. Будем считать, что запасы банкнот каждого номинала неограничены.

Рассмотрим такой алгоритм: будем выдавать банкноты наибольшего номинала, пока это возможно, затем переходим к следующему номиналу. Например, если имеются банкноты в 10, 50, 100, 500, 1000 рублей, то при $N = 740$ рублей такой алгоритм выдаст банкноты в 500, 100, 100, 10, 10, 10, 10 рублей. Подобные алгоритмы называют «жадными», поскольку каждый раз при принятии решения выбирается тот вариант, который кажется наилучшим в данной ситуации (чтобы использовать наименьшее число банкнот каждый раз выбирается наибольшая из возможных банкнот).

Но для решения данной задачи в общем случае жадный алгоритм оказывается неприменимым. Например, если есть банкноты номиналом в 10, 60 и 100 рублей, то при $N = 120$ жадный алгоритм выдаст три банкноты: $100 + 10 + 10$, хотя есть способ, использующий две банкноты: $60 + 60$. А если номиналов банкнот только два: 60 и 100 рублей, то жадный алгоритм вообще не сможет найти решения.

Но эту задачу можно решить при помощи метода динамического программирования. Пусть $F(n)$ — минимальное количество банкнот, которым можно заплатить сумму в n рублей. Очевидно, что $F(0) = 0$, $F(a_1) = F(a_2) = \ldots = F(a_k) = 1$. Если некоторую сумму n невозможно выдать, будем считать, что $F(n) = \infty$ (бесконечность).

Выведем рекуррентную формулу для $F(n)$, считая, что значения $F(0)$, $F(1)$, …, $F(n-1)$ уже вычислены. Как можно выдать сумму n? Мы можем выдать сумму $n - a_1$, а потом добавить одну банкноту номиналом a_1. Тогда нам понадобится $F(n - a_1) + 1$ банкнота. Можем выдать сумму $n - a_2$ и добавить одну банкноту номиналом a_2, для такого способа понадобится $F(n - a_2) + 1$ банкнота и т. д. Из всевозможных способов выберем наилучший, то есть:

$$F(n) = \min(F(n - a_1), F(n - a_2), \ldots, F(n - a_k)) + 1.$$

Теперь заведем массив `F[n+1]`, который будем последовательно заполнять значениями выписанного рекуррентного соотношения. Будем

предполагать, что количество номиналов банкнот хранится в переменной `int k`, а сами номиналы хранятся в массиве `int a[k]`.

```
const int INF=1000000000;// Значение константы "бесконечность"
int F[n+1];
F[0]=0;
int m, i;
for(m=1;m<=n;++i)  // заполняем массив A
{                  // m - сумма, которую нужно выдать
  F[m]=INF;        // помечаем, что сумму i выдать нельзя
  for(i=0;i<k;++i) // перебираем все номиналы банкнот
  {
    if(m>=a[i] && F[m-a[i]]+1<F[m])
      F[m] = F[m-a[i]]+1;// изменяем значение F[m], если нашли
  }                      // лучший способ выдать сумму m
}
```

После окончания этого алгоритма в элементе `F[n]` будет храниться минимальное количество банкнот, необходимых, чтобы выдать сумму `n`. Как теперь вывести представление суммы `n` при помощи `F(n)` банкнот? Опять рассмотрим все номиналы банкнот и значения $n - a_1$, $n - a_2$, …, $n - a_k$. Если для какого-то i окажется, что $F(n - a_i) = F(n) - 1$, значит, мы можем выдать банкноту в a_i рублей и после этого свести задачу к выдаче суммы $n - a_i$, и так будем продолжать этот процесс, пока величина выдаваемой суммы не станет равна 0:

```
if (F[n]==INF)
    cout<<"Требуемую сумму выдать невозможно"<<endl;
else
{
    while(n>0)
    {
        for(i=0;i<k;++i)
        {
            if(F[n-a[i]]==F[n]-1)
            {
                cout<<a[i]<<" ";
                break;
            }
        }
    }
}
```

Задача о рюкзаке

Грабитель, проникший в банк, обнаружил в сейфе k золотых слитков, массами $w_1, w_2, \ldots, w_k$ килограмм. При этом грабитель может унести не более W килограмм. Определите набор слитков, который должен взять грабитель, чтобы унести как можно больше золота.

Эта задача является частным случаем задачи об укладке рюкзака. Сформулируем ее в общем случае.

Дано k предметов, i-й предмет имеет массу $w_i > 0$ и стоимость $p_i > 0$. Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины W (вместимость рюкзака), а суммарная стоимость была максимальна. Другими словами, нужно определить набор бинарных величин $(b_1, b_2, \ldots, b_k)$, такой, что

$$b_1w_1 + b_2w_2 + \ldots + b_kw_k \leqslant W,$$

а величина $b_1p_1 + b_2p_2 + \ldots + b_kp_k$ — максимальная. Величина b_i равна 1, если i-й предмет включается в набор, и равна 0 в противном случае.

Задача укладки рюкзака очень сложна. Если перебирать всевозможные подмножества данного набора из k предметов, то получится решение сложности не менее чем $O(2^k)$. В настоящее время неизвестен (и, скорее всего, вообще не существует) алгоритм решения этой задачи, сложность которого является многочленом от k.

Мы рассмотрим решение данной задачи для случая, когда все входные данные — целочисленные, сложность которого будет $O(kW)$.

Рассмотрим следующую функцию. Пусть $A(s, n)$ есть максимальная стоимости предметов, которые можно уложить в рюкзак максимальной вместимости n, если можно использовать только первые s предметов из заданных k.

Зададим краевые значения функции $A(s, n)$.

Если $s = 0$, то $A(0, n) = 0$ для всех n (ни один предмет нельзя брать, поэтому максимальная стоимость равна 0).

Если $n = 0$, то $A(s, 0) = 0$ для всех s (можно брать любые из первых s предметов, но вместимость рюкзака равна 0).

Теперь составим рекуррентное соотношение в общем случае. Необходимо из предметов с номерами $1, \ldots, s$ составить рюкзак максимальной стоимости, чей вес не превышает n. При этом возможно два случая: когда в максимальный рюкзак включен предмет с номером s и когда предмет s не попал в максимальный рюкзак.

Если предмет s не попал в максимальный рюкзак массы n, то максимальный рюкзак будет составлен только из предметов с номерами $1, \ldots, s-1$, следовательно, $A(s, n) = A(s-1, n)$.

Если же в максимальный рюкзак включен предмет s, то масса оставшихся предметов не превышает $n - w_s$, а от добавления предмета s общая стоимость рюкзака увеличивается на p_s. Значит, $A(s, n) = A(s-1, n-w_s) + p_s$. Теперь из двух возможных вариантов составить рюкзак массы, не превосходящей n, из предметов $1, \ldots, s$ нужно выбрать наилучший:

$$A(s, n) = \max\left(A(s-1, n), A(s-1, n-w_s) + p_s\right).$$

Теперь составим программу. Будем считать, что веса предметов хранятся в массиве `w[1], ..., w[k]`, а их стоимости в массиве `p[1], ..., p[k]`. Значения функции $A(s, n)$, где $0 \leqslant s \leqslant k$, $0 \leqslant n \leqslant W$, будем хранить в массиве `A[k+1][W+1]`.

```
int A[k+1][W+1];
for(n=0;n<=W;++n)        // Заполняем нулевую строчку
    A[0][n]=0;
for(s=1;s<=k;++s)        // s - максимальный номер предмета,
{                        // который можно использовать
    for(n=0;n<=W;++n)    // n - вместимости рюкзака
    {
        A[s][n]=A[s][n-1];
        if ( n>=w[s] && ( A[s-1][n-w[s]]+p[s] > A[s][n]) )
            A[s][n] = A[s-1][n-w[s]]+p[s];
    }
}
```

В результате исполнения такого алгоритма в элементе массива `A[k][W]` будет записан ответ на поставленную задачу. Легко видеть, что сложность этого алгоритма, равно как и объем используемой им памяти, являются величиной $O(kW)$.

Рассмотрим пример работы этого алгоритма. Пусть максимальная вместимость рюкзака $W = 15$, количество предметов $k = 5$, их стоимости и массы таковы:
$w_1 = 6$, $p_1 = 5$, $w_2 = 4$, $p_2 = 3$, $w_3 = 3$, $p_3 = 1$,
$w_4 = 2$, $p_4 = 3$, $w_5 = 5$, $p_5 = 6$.

В приведенной ниже таблице указаны значения заполненного массива `A[k+1][W+1]`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$s=0$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$s=1$	0	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5
$s=2$	0	0	0	0	3	3	5	5	5	5	8	8	8	8	8	8
$s=3$	0	0	0	1	3	3	5	5	5	6	8	8	8	9	9	9
$s=4$	0	0	3	3	3	4	6	6	8	8	8	9	11	11	11	12
$s=5$	0	0	3	3	3	6	6	9	9	9	10	12	12	14	14	14

Первая строка массива соответствует значениям $A(0,n)$. Поскольку ни одного предмета брать нельзя, то строка заполнена нулями: из пустого множества предметов можно составить рюкзак нулевой массы.

Вторая строка массива соответствует значению $s=1$, то есть рюкзак можно составлять только из первого предмета. Вес этого предмета $w_1=6$, а его стоимость $p_1=5$. Поэтому при $n<6$ мы не можем включить этот предмет в рюкзак и значение $A(1,n)$ равно 0 при $n<6$. Если $n \geqslant w_1$, то мы можем включить первый предмет в рюкзак, а поскольку других предметов нет, то $A(1,n)=5$ (так как $p_1=5$).

Рассмотрим третью строку массива, соответствующую двум предметам ($s=2$). Добавляется второй предмет, более легкий и менее ценный, чем первый ($w_2=4$, $p_2=3$). Поэтому $A(2,n)=0$ при $n<4$ (ни один предмет взять нельзя), $A(2,n)=3$ при $n=4$ и $n=5$ (в рюкзак включается предмет номер 2 ценности 3), $A(2,n)=5$ при $6 \leqslant n \leqslant 9$ (при данном n выгоднее в рюкзак включить предмет 1, поскольку его ценность выше) и, наконец, $A(2,n)=8$ при $n \geqslant 10$ (при данной вместимости рюкзака можно взять оба предмета).

Аналогично заполняются остальные строки массива, при заполнении элемента $A(s,n)$ рассматривается две возможности: включать или не включать предмет с номером s.

Как теперь вывести на экран тот набор предметов, который входит в максимальный рюкзак? Сравним значение `A[k][W]` со значением `A[k-1][W]`. Если они равны, то максимальный рюкзак можно составить без использования предмета с номером `k`. Если не равны, то предмет с номером `k` обязательно входит в максимальный рюкзак. В любом случае, задача печати рюкзака сводится к задаче печати рюкзака для меньшего числа предметов. Напишем это в виде рекурсивной функции `Print(int s, int n)`, которая по параметрам `s` и `n` печатает номера предметов, входящих в максимальный рюкзак массой не более `n` и составленный из предметов 1, …, `s`:

```
void Print(int s, int n)
{
  if (A[s][n]==0) // максимальный рюкзак для параметров (s,n)
```



```
    return;       // имеет нулевую ценность,
                  // поэтому ничего не выводим
  else if (A[s-1][n] == A[s][n])
    Print(s-1,n);  // можно составить рюкзак без предмета s
  else
  {
    Print(s-1,n-w[s]); // Предмет s должен обязательно
    cout<<s<<endl;     // войти в рюкзак
  }
}
```

Для печати искомого рюкзака необходимо вызвать функцию с параметрами `(k,W)`.

В приведенном примере для печати максимального рюкзака вызовем функцию `Print(5,15)`. Поскольку $A(5,15)=14$, а $A(4,15)=12$ (с использованием только первых 4 предметов мы можем собрать рюкзак максимальной стоимости 12, а с использованием всех 5 предметов — стоимости 14), предмет номер 5 обязательно входит в рюкзак. Далее рассмотрим $A(4,10)$ (общая вместимость рюкзака уменьшилась на вес включенного предмета). Поскольку $A(4,10)=A(3,10)=A(2,10)=8$, то мы можем исключить из рассмотрения предметы номер 4 и 3 — можно собрать рюкзак вместимости 10 и стоимости 8 только из первых двух предметов. Для этого необходимо включить оба этих предмета. Таким образом, оптимальный рюкзак будет состоять из предметов 1, 2, 5, его масса будет равна $6+4+5=15$, а стоимость — $5+3+6=14$.

Можно составить рюкзак и по-другому. Поскольку вес предмета 4 равен двум, его стоимость p_4 равна трём, а $A(4,10)=A(3,8)+p_4$, то мы можем включить в наш рюкзак предмет 4. Теперь рассмотрим $A(3,8)=5$ — как составить рюкзак массы не более 8 и стоимости 5 из первых трех предметов. Поскольку $A(3,8)=A(2,8)=A(1,8)=5$, то мы исключаем из рассмотрения предметы 3 и 2, но включаем предмет 1. Получим рюкзак из предметов 1, 4, 5, его масса будет $6+2+5=13$ и стоимость также равна $5+3+6=14$. Поскольку стоимость обоих полученных рюкзаков получилась одинаковой, а масса в каждом случае не превосходит максимально допустимого значения $W=15$, то оба решения подходят.

Литература

1. *А. Шень.* Программирование: теоремы и задачи. М: МЦНМО, 2004.
2. *Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.* Алгоритмы: построение и анализ. М: Издательский дом «Вильямс», 2005.

Динамическое программирование по профилю

Б. Василевский

К большинству олимпиадных задач ограничения (по времени, по памяти) жюри подбирает по принципу «как можно больше». То есть чтобы любые разумные реализации правильного решения проходили, а всё остальное — нет.

Когда встречается задача с маленькими ограничениями (например, до 10), это означает, что либо автор намеренно сбивает Вас с правильного пути, либо действительно эта задача решается каким-то (оптимизированным) перебором.

Динамическое программирование по профилю — одна из таких оптимизаций. Часто в таких задачах дело происходит на прямоугольной таблице, одна из размерностей которой достаточно мала (не более 10). Требуется проверить существование, посчитать количество способов, стоимость и т. д. (как в обычном динамическом программировании). Асимптотика алгоритма, основанного на этой идее, является экспоненциальной только по одной размерности, а по второй — линейная или даже лучше.

Некоторые обозначения и определения

Матрицей размера $n \times m$ называется прямоугольная таблица $n \times m$, составленная из чисел.

Обычно матрицы обозначают заглавными латинскими буквами. Элемент i-й строки j-го столбца матрицы A обозначают через a_{ij} или $a[i, j]$ (соответствующая маленькая латинская буква с индексами).

Произведением двух матриц $\underset{n\times m}{A}$ *и* $\underset{m\times k}{B}$ называется матрица такая $\underset{n\times k}{C}$, что

$$c_{ij} = \sum_{t=1}^{m} a_{it} b_{tj}. \tag{1}$$

В этом случае пишут: $C = AB$.

D в степени k (D^k), при условии, что D — квадратная (т. е. $n = m$) определяется следующим образом:

- $D^0 = E$ (единичная матрица), где

$$E = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}.$$

  На диагонали у нее стоят единицы, в остальных клетках — нули. Она не зря называется единичной — при умножении на нее матрица не изменяется: $AE = EA = A$.

- $D^i = D^{i-1}D, \ i > 0$.

В приводимом коде будет использоваться функция `bit(x,i)`, возвращающая единицу или ноль — i-й бит в двоичной записи числа x (нумерация битов с нуля).

```
// возвращает i-й бит числа x, нумерация с нуля
function bit(x, i : integer) : integer;
begin
    if (i < 0) then bit := 0 else
        if (x and (1 shl i) = 0) then bit := 0 else bit := 1;
end;
```

Задача о замощении домино

Чтобы понять, что такое динамика по профилю, будем рассматривать разные задачи и разбирать их решения с помощью этого приема.

Для начала рассмотрим известную задачу: дана таблица $n \times m$, нужно найти количество способов полностью замостить ее неперекрывающимися костяшками домино (прямоугольниками 2×1 и 1×2). Считаем $n, m \leqslant 10$.

Заметим, что в процессе замощения каждая клетка таблицы будет иметь одно из двух состояний: покрыта какой-нибудь доминошкой или нет. Чтобы запомнить состояние клеток одного столбца, достаточно одной переменной P типа `integer`. Положим i-й бит P равным 1, если i-я сверху клетка данного столбца занята, 0 — если свободна. Будем говорить в таком случае, что P — битовая карта нашего столбца.

Теперь дадим определение базовой линии и профиля.

Базовой линией будем называть вертикальную прямую, проходящую через узлы таблицы.

Можно занумеровать все базовые линии, по порядку слева направо, начиная с нуля. Таким образом, базовая линия с номером i — это прямая, отсекающая первые i столбцов от всех остальных (если такие имеются).

Отныне через b_i будем обозначать базовую линию с номером i.

Столбцы занумеруем так: слева от b_i будет столбец с номером i. Другими словами, столбец с номером i находится между b_{i-1} и b_i.

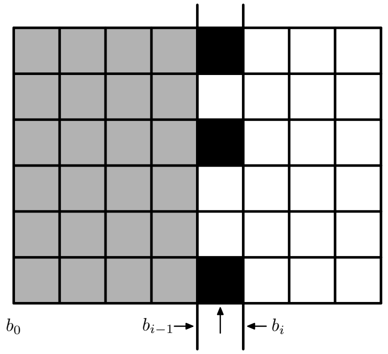


Рис. 1. Профиль будет таким: $100101_2 = 1 + 4 + 32 = 37$ (заняты первая + третья + шестая клетки).

Профилем для базовой линии с номером i (b_i) будем называть битовую карту для столбца с номером i при следующих дополнительных условиях:

1) все клетки слева от b_{i-1} уже покрыты;

2) в i-м столбце нет вертикальных доминошек;

3) считается, что справа от b_i нет покрытых клеток.

Первое условие возникает от желания считать количество способов постепенно, сначала рассматривая первый столбец, потом второй при условии, что первый уже заполнили и т. д. Второе и третье вводятся для того, чтобы один и тот же способ покрытия не был посчитан более одного раза. Точный смысл этих условий будет раскрыт ниже.

Для каждого профиля p_1 базовой линии b_i определим множество профилей p_2 базовой линии b_{i+1}, которые могут быть из него получены. На таблицу, соответствующую p_1 (то есть все клетки слева от b_{i-1} полностью покрыты, в i-м столбце клетки отмечены согласно p_1), можно класть доминошки только двух типов:

1) горизонтальные доминошки, которые пересекают b_i (то есть делятся ей пополам);

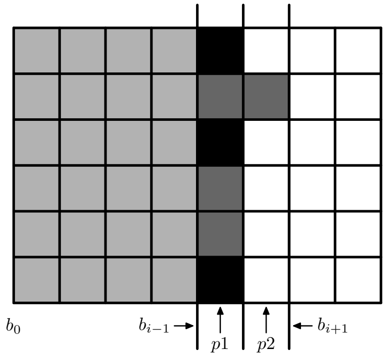


Рис. 2. $p_1 = 37$, $p_2 = 2$; нетрудно заметить по рисунку, что из p_1 можно получить p_2. Таким образом, $d[37, 2] = 1$.

2) вертикальные доминошки, которые лежат слева от b_i.

Также должны быть выполнены естественные дополнительные условия:

1) новые доминошки не должны перекрываться друг с другом;

2) они должны покрывать только незанятые клетки;

3) каждая клетка i-го столбца должна быть покрыта.

Битовая карта столбца $i + 1$ и будет возможным профилем p_2.

Очевидно, что полученный таким образом профиль p_2 действительно удовлетворяет всем условиям, накладываемым на профиль.

Пусть $d[p_1, p_2]$ – количество способов из профиля p_1 (для b_i) получить p_2 (для b_{i+1}). Очевидно, для данной задачи про доминошки это число может быть только единицей или нулем. Различные задачи будут отличаться в основном только значениями $d[p_1, p_2]$.

Заметим, что всего профилей 2^n: от $00\ldots0_2 = 0$ до $11\ldots1_2 = 2^n - 1$. Поэтому в данном случае матрица D будет иметь размер $2^n \times 2^n$.

Пусть теперь $a[i, p]$ — количество способов таким образом расположить доминошки, что p — профиль для b_i (таким образом, все клетки левее b_{i-1} покрыты).

Напишем рекуррентное соотношение.

- Начальные значения ($i = 1$):
  $a[1, 0] = 1$;
  $a[1, p_2] = 0,\ p_2 = 1,\ \ldots,\ 2^n - 1$

- Общая формула ($i > 1$):

$$a[i, p_1] = \sum_{p_2=0}^{2^n-1} a[i-1, p_2]\, d[t, p_1] \tag{2}$$

Заметим, что для базовой линии номер 1 существует единственный профиль (то есть битовая карта, удовлетворяющая условиям профиля) — карта незаполненного столбца.

Ответ на вопрос задачи будет записан в $a[m+1, 0]$. Ошибкой бы было считать правильным ответом число $a[m, 2^n - 1]$, так как в этом случае не учитывается возможность класть вертикальные доминошки в последнем столбце (см. второй пример).

Обсудим «странные» условия на доминошки при получении одного профиля из другого. Казалось бы, забыт еще один тип доминошек, которые могут участвовать при формировании нового профиля, а именно полностью лежащие в столбце $i+1$. Дело в том, что если разрешить их, то некоторые способы замощения будут считаться более одного раза. Например, пусть $n = 2, m = 2$. Тогда $d'[0][3] = 2$, так как можно положить либо две вертикальные доминошки, либо две горизонтальные. Аналогично, $d'[3][3] = 1$ (можно положить одну вертикальную). В итоге

$$D' = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}, \qquad A' = \begin{pmatrix} 1 & 1 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{pmatrix}$$

Имеем неправильный ответ 3 (можно посчитать вручную, что на самом деле ответ 2).

Напротив, если следовать данному правилу получения из одного профиля другого, то можно убедиться в верности вычислений.

Упражнение. *Доказать, что* $a[i, p]$ *вычисляется правильно.*

Вот какими будут D и A если $n = 2, m = 4$:

$$D = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & 1 & 2 & 3 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 3 \end{pmatrix}$$

Таким образом, замостить доминошками таблицу 2×4 можно 5 способами, а таблицу 2×2 — двумя. Если смотреть $a[m, 2^n - 1]$, то получим 2 и 1. Очевидно, что таблицу 2×2 можно замостить двумя, а не одним способом: пропущен вариант, когда кладутся две вертикальные доминошки. В случае 2×4 пропущены три замощения — все случаи, когда последний столбец покрыт вертикальной доминошкой.

Существует два способа для вычисления D. Первый заключается в том, чтобы для каждой пары профилей p_1 и p_2 проверять, можно ли из p_1 получить p_2 описанным способом.

При втором способе для каждого профиля p_1 пытаемся его замостить, кладя при этом только домоношки разрешенных двух типов. Для всех профилей p_2 (и только для них), которые при этом получались в следующем столбце, положим $d[p1, p2] = 1$. В большинстве случаев этот способ более экономичный, так что логично использовать именно его.

Ниже приведен код рекурсивной процедуры, которая заполняет строку $d[p]$, то есть находит все профили, которые можно получить из p:

```
procedure go(n, profile, len : integer);
begin
    // n - из условия задачи
    // profile - текущий профиль
    // len - длина profile

    if (len = n) then begin
        // как только profile получился длины n, выходим
        d[p][profile] := 1;
        exit;
    end;
    if (bit(p, len) = 0) then begin
        // текущая ячейка в p (с номером len + 1) не занята
        go(p, profile + (1 shl len), len + 1);
        // положили горизонтальную доминошку

        if (len < n - 1) then
            if (bit(p, len + 1) = 0) then begin
                // не занята еще и следующая ячейка
                go(p, profile, len + 2);
                // положили вертикальную доминошку
            end;
    end else begin
        go(p, profile, len + 1);
        // текущая ячейка занята, ничего положить не можем
    end;
end;
```

```
procedure determine_D;
var p : integer;
begin
  for p := 0 to (1 shl n) - 1 do
    go(p, 0, 0); // запускать надо именно с такими параметрами
end;
```

Алгоритм вычисления D и A работает за

$O(2^{2n})$ (вычисление D) $+O(2^{2n}m)$ (вычисление A) $= O(2^{2n}m)$.

Задача о симпатичных узорах

Рассмотрим еще одну задачу с прямоугольной таблицей.

Дана таблица $n \times m$, каждая клетка которой может быть окрашена в один из двух цветов: белый или черный. Симпатичным узором называется такая раскраска, при которой не существует квадрата 2×2, в котором все клетки одного цвета. Даны n и m. Требуется найти количество симпатичных узоров для соответствующей таблицы.

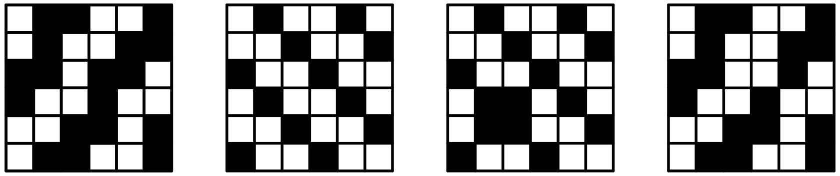


Рис. 3. Первые два узора симпатичные; у третьего и четвертого есть полностью черный и, соответственно, белый квадратик 2×2.

Будем считать профилем для b_i битовую карту i-го столбца (единицей будет кодировать черную клетку, а нулем — белую). При этом узор, заключенный между нулевой и i-й базовыми линиями, является симпатичным.

Из профиля p_1 для b_i можно получить p_2 для b_{i+1}, если и только если можно так закрасить $(i+1)$-й столбец, что его битовая карта будет соответствовать p_2, и между b_{i-1} и b_{i+1} не будет полностью черных либо белых квадратиков 2×2.

Сколькими же способами из одного профиля можно получить другой? Понятно, что закрасить нужным образом либо можно, либо нельзя (так как раскрашивать можно единственным образом — так, как закодировано в p_2). Таким образом, $d[p_1, p_2] \in \{0, 1\}$.

Вычислять D можно либо проверяя на «совместимость» (то есть наличие одноцветных квадратов 2×2) все пары профилей, либо генерируя все допустимые профили для данного. Ниже приведен код, реализующий первую идею:

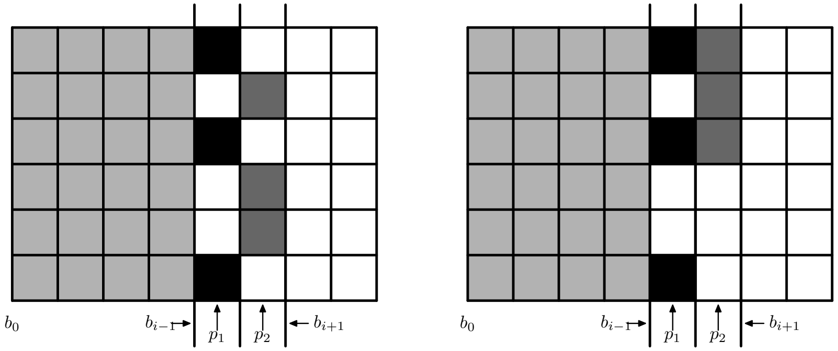


Рис. 4. На левом рисунке $p_1 = 1 + 4 + 32 = 37$, $p_2 = 2 + 8 + 16 = 26$; так как между b_{i-1} и b_{i+1} не встречаются одноцветные квадратики 2×2, то p_2 может быть получен из p_1. На рисунке справа p_1 также равно 37, а $p_2 = 1 + 2 + 4 = 7$.

```
// можно ли из p1 получить p2
function can(p1, p2 : integer) : boolean;
var i : integer;
    b : arrray[1..4] of byte;
begin
    for i := 0 to n - 2 do begin
        b[1] := bit(p1, i);
        b[2] := bit(p1, i + 1);
        b[3] := bit(p2, i);
        b[4] := bit(p2, i + 1);

        if (b[1] = 1) and (b[2] = 1) and (b[3] = 1)
                                and (b[4] = 1) then begin
            // квадрат в строках i и i + 1 черный
            can := false;
            exit;
        end;

        if (b[1] = 0) and (b[2] = 0) and (b[3] = 0)
                                and (b[4] = 0) then begin
            // квадрат в строках i и i + 1 белый
            can := false;
            exit;
        end;
    end;
    can := true;
```

```
end;

procedure determine_D;
var p1, p2 : integer;
begin
    for p1 := 0 to (1 shl n) - 1 do
        for p2 := 0 to (1 shl n) - 1 do
            if can(p1, p2) then d[p1, p2] :=1
            else d[p1, p2] := 0;
end;
```

После того, как вычислена матрица D, остается просто применить формулу (2) (так как расуждения на этом этапе не изменяются).

Связь ДП по профилю и линейной алгебры

Рекуррентное соотношение (2) будет встречаться нам не только в задаче о замощении или симпатичном узоре, но и во многих других задачах, решаемых динамикой по профилю. Поэтому логично, что существует несколько способов вычисления A, используя уже вычисленную D (а не только наивно по (2)). В этом пункте мы рассмотрим способ, основанный на возведении в степень матрицы:

1) $a[i]$ можно считать матрицей 1×2^n;

2) D — матрица $2^n \times 2^n$;

3) $a[i] = a[i-1]D$. Если расписать эту формулу по определению произведения, то получится в точности (2).

Следуя определению степени матрицы, получаем

$$a[m] = a[0]D^m \tag{3}$$

Вспомним, как возвести действительное число a в натуральную степень b за $O(\log b)$ (считаем, что два числа перемножаются за $O(1)$). Представим b в двоичной системе счисления: $b = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_k}$, где $i_1 < i_2 < \ldots < i_k$. Тогда $k = O(\log b)$. Заметим, что a^{2^i} получается из $a^{2^{(i-1)}}$ возведением последнего в квадрат. Таким образом, за $O(k)$ можно вычислить все a^{p_t}, $p_t = 2^{i_t}$, $t = 1, \ldots, k$. Перемножить их за линейное время тоже не представляет труда.

Логично предположить, что аналогичный алгоритм сгодится и для квадратных матриц. Единственное нетривиальное утверждение — $A^{2^i} = (A^{2^{i-1}})^2$, ведь по определению $A^{2t} = \underbrace{A(A(\ldots A)\ldots)}_{2t}$, а мы хотим

приравнять его к $(A^t)(A^t)$. Его истинность следует из ассоциативности умножения матриц $(AB)C = A(BC)$. Само свойство можно доказать непосредственно, раскрыв скобки в обеих частях равенства.

Приведем код процедуры возведения в степень (функция `mul` перемножает две квадратные матрицы размера `w` $\times$ `w`):

```
function mul(a, b : tmatr) : tmatr;
var res : tmatr;
    i, j, t : integer;
begin
    for i := 1 to w do begin
        for j := 1 to w do begin
            res[i][j] := 0;
            for t := 1 to w do begin
                res[i][j] := res[i][j] + a[i][t]*b[t][j];
            end;
        end;
    end;
    mul := res;
end;

function power(a : tmatr; b : integer) : tmatr;
var i, j : integer;
    res, tmp : tmatr;
begin
    res := E; // единичная матрица
    tmp := a;
    while (b > 0) do begin
        if (b mod 2 = 1) then res := mul(res, tmp);
        b := b div 2;
        tmp := mul(tmp, tmp);
    end;
    power := res;
end;
```

Как уже говорилось, будет сделано $O(\log b)$ перемножений. В данном случае, на каждое перемножение тратится n^3 операций (где n — размерность матрицы). Так что этот алгоритм будет работать за $O(n^3 \log b)$.

Вернемся к (3). Матрицу D мы умеем вычислять за $O((2^n)^2 n) = O(4^n n)$ (как в рассмотренных задачах). Вектор $a[m]$ сумеем найти за $O((2^n)^3 \log b) = O(8^n \log m)$. В итоге получаем асимптотику $O(8^n \log m)$. При больших m (например, 10^{100}) этот способ вычисления A несравнимо лучше наивного.

Задача о расстановке королей

Дана шахматная доска $n \times m$ и число k. Нужно посчитать количество способов размещения на этой доске k королей так, чтобы они не били друг друга.

Профилем опять будет битовая карта столбца слева от базовой линии. В данном случае удобно запоминать расположение королей. Таким образом, единица будет означать наличие короля на соответствующей позиции. При переходе от одного профиля к другому ставим королей справа от b_i так, чтобы они не били друг друга и предшествующих им.

Заметим, что снова $d_{ij} \in \{0, 1\}$. Отличие этой задачи от предыдущих заключается в следующем. Количество «настоящих» профилей сильно отличается от 2^n: если в позиции j есть король, то в позициях $j-1$ и $j+1$ его заведомо нет. То есть, в двоичной записи профиля не должно встречатся двух подряд идущих единиц. Пусть $f(n)$ — количество возможных профилей длины n.

Напишем для $f(n)$ рекуррентную формулу. Количество профилей, у которых на n-м месте стоит 1, равно $f(n-2)$, так как на $(n-1)$-м не может стоять 0, на остальные ячейки ограничений нет. Если же на n-м месте стоит 0, то количество будет равно $f(n-1)$. Тогда $f(n) = f(n-1) + f(n-2)$; $f(1) = 2, f(2) = 3$. Получили, что $f(n)$ — $(n+2)$-е число Фибоначчи. Известно, что $f(n) < (1,62)^{n+2}$. При $n = 10$ имеем $f(n) = 144$ против количества битовых карт $1\,024$, уменьшение более чем в 7 раз!

Использовать это наблюдение можно по-разному.

1) Будем рассматривать только настоящие профили (при вычислении D и A), используя рекурсию:

```
procedure go(len, profile : integer);
begin
    // profile - текущий профиль
    // len = (длина profile) + 1

    if (len > n) then begin
        // как только profile получился длины n, выходим
        writeln(profile, ' - настоящий профиль');
        exit;
    end;

    go(len + 1, profile*2);
    // приписать ноль мы всегда можем
```



```
    if (profile mod 2 = 0) then
                go(len + 1, profile*2 + 1);
    // приписать единицу можно только если рядом ноль
end;

procedure print_all_true_profiles;
begin
  go(1, 0); //запускать надо именно с такими параметрами
end;
```

2) Занумеруем все настоящие профили так, чтобы быстро получать по номеру профиль. Тогда чтобы перебрать все настоящие профили, нужно будет пустить цикл по номеру с 1 до их количества $f(n)$, и каждый раз находить профиль по текущему номеру. Например, можно брать номера в лексикографически упорядоченном списке профилей (чтобы сказать, какой из двух профилей больше лексикографически, достаточно сравнить их как числа).

3) Пусть p — настоящий ненулевой профиль. Если заменить в нем произвольную единичку (в двоичной записи) на нолик, то получится тоже настоящий профиль. Другими словами, если убрать короля, то ничего плохого не будет. Аналогично, если в p заменить 0 на 1, чтобы не возникло двух подряд идущих единиц, результатом будет настоящий профиль.

   На этом основан еще один способ получения всех настоящих профилей: «поиском в ширину». Из настоящих профилей, в котоых ровно i королей, получаем всевозможные настоящие профили из $i+1$ королей.

```
var use : array[0..(1 shl n) - 1] of byte;
   // use[p] = 1, если p - в очереди
   q : array[1..(1 shl n)] of integer; // очередь
   r : integer; // итоговое количество настоящих профилей

procedure determine_all_true_profiles;
var i, l, x, y : integer;
begin
    l := 0;
    r := 1;
    for i := 1 to (1 shl n) - 1 do use[i] := 0;
    use[0] := 1;
```

```
    q[1] := 0;
    while (l < r) do begin
        l := l + 1;
        x := q[l];
        // теперь попробуем подобавлять единички
        for i := 0 to n - 1 do
            if (bit(x,i) = 0) and (bit(x, i - 1) = 0)
                       and (bit(x, i + 1) = 0) then begin
                y := x + (1 shl i);
                if (use[y] = 0) then begin
                    r := r + 1;
                    q[r] := y;
                end;
            end;
    end;
end;
```

После завершения работы процедуры в очереди q будут содержаться все настоящие профили.

В итоге при использовании нашего наблюдения получаем асимптотику $O(m(f(n))^2)$

Задача о расстановке коней

На этот раз на шахматной доске $n \times n$ нужно подсчитать количество способов расставить k коней так, чтобы они не били друг друга.

Мы уже привыкли, что профиль — битовая карта столбца левее b_i. Но здесь этой информации мало, так как кони могут прыгать сразу через два столбца. Поэтому профиль должен описывать два соседних столбца.

Для двух профилей $pr_1 = \langle p_1, p_2\rangle$ (p_1 — *битовая карта первого столбца,* p_2 — *второго) и* $pr_2 = \langle p'_1, p'_2\rangle$ *положим* $d[pr_1, pr_2] = 1$ *если и только если*

а) $p_2 = p'_1$;

б) кони не бьют друг друга.

В противном случае $d[pr_1, pr_2] = 0$.

Таким образом, переходов на порядок меньше, чем количество профилей. Всего профилей 4^n, а из каждого профиля получается менее 2^n новых за счет совпадения p_2 и p'_1. Другими словами, матрица D сильно разрежена (имееет много нулевых элементов).

Поэтому в задаче о расстановки коней можно добиться значительного ускорения (по сравнению с шаблонным алгоритмом), если хранить D в виде «списков смежности»: $D'[pr_1]$ — список всех таких pr_2, из которых можно получить pr_1, то есть $d[pr_2][pr_1] = 1$. В этом случае формула (2) будет выглядеть так:

$$a[i, p] = \sum_{t \in D'[p]} a[i-1, t]\, d[t, p].$$

Суммарное время вычисления $a[i]$ равно количеству всевозможных переходов, то есть количество ненулевых элементов в D. В нашем случае (задаче о конях) их не более $4^n \times 2^n = 8^n$, то есть время работы алгоритма с такой оптимизацией будет $O(8^n m)$ — это в 2^n раз меньше, чем время стандартного алгоритма.

ДП по изломанному профилю

Другое название этому методу — «быстрая динамика по профилю». Идея в том, чтобы добиться как можно меньшего числа переходов (от одного профиля к другому).

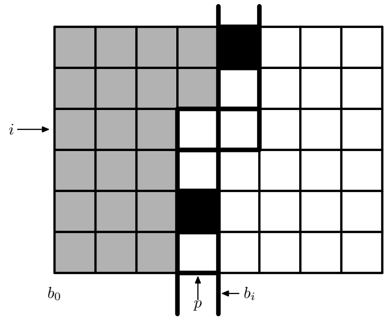


Рис. 5. Изображение профиля $\langle 33, 2\rangle$ ($33 = 2^0 + 2^5$).

Еще раз используем в качестве примера задачу о замощении. Базовая линия теперь будет ломаной: при прохождении через i-ю горизонталь сверху вниз, она переходит на предыдущую вертикаль и спускается до низу (см. рис. 5).

Профилем будет пара $\langle p, i\rangle$, в p будет информация о $n+1$ маленьком квадратике слева от базовой линии, имеющем с ней общие точки; i обозначает номер горизонтали, на которой произошел излом. Квадратики профиля будут нумероваться сверху вниз, так что угловой будет иметь номер $i+1$. Горизонтали будем нумеровать нуля, так что i пробегает значения $0..n-1$.

Для двух профилей $pr_1 = \langle p_1, i_1 \rangle$ *и* $pr_2 = \langle p_2, i_2 \rangle$ *положим* $d[pr_1][pr_2] = 1$ *если и только если:*

а) *если* $i < n - 1$, *то* $i_1 + 1 = i_2$; *иначе* $i_2 = 0$;

б) *можно так положить доминошку, накрывающую* $(i+1)$*-й квадратик, что после этого в* p_2 *будет храниться в точности информация о соответствующих квадратиках.*

Проще говоря, доминошку можно класть только двумя способами — как показано на рисунках (на $(i+1)$-й квадратик можно положить не более одной вертикальной и горизонтальной доминошки). То, что потом получается после сдвига вниз излома, и будет новым профилем. Заметим, что если $(i+1)$-я клетка занята, то доминошку уже не надо класть, и $\langle p, i \rangle$ логично отождествить с $\langle p, i+1 \rangle$ («$i+1$» пишется условно, нужно всегда иметь в виду возможность $i = n - 1$).

Легко заметить, что количество профилей увеличилось в $2n$ раз (добавилось число от 1 до n и еще один бит). Но зато количество переходов резко сократолсь с 2^n до двух!

В нижеприведенном куске кода для профиля $\langle p, i \rangle$ выводятся все переходы из него (напомним, что нумерация горизонталей начинается с нуля и $i = 0..n-1$):

```
procedure print_all_links(p, i : integer);
begin
    if (bit(p, i + 1) = 0) then begin
        if (i = n - 1) then begin
            writeln('<', (p - (2 shl i)) shl 1, ', ', 0, '>');
        end else begin
            writeln('<', p - (2 shl i), ', ', i + 1, '>');
        end;
    end else begin
        if (bit(p, i) = 0) then begin
            if (i = n - 1) then begin
                writeln('<', p shl 1, ', ', 0, '>');
            end else begin  writeln('<', p + (1 shl i), ', ',
                            (i + 1) mod n, '>');         end;
        end;
        if (i < n - 1) and (bit(p , i + 2) = 0) then begin
            writeln('<', p + (4 shl i), ', ', i + 1, '>');
        end;
    end;
end;
```


При такой реализации существует немало профилей только с одним переходом (например, у которых $(i+1)$-й бит равен единице).

Отождествим все профили с один переходом с теми, кто их них получается. Это будет выглядеть так: пусть pr_2 (и только он) получается из pr_1, который, в свою очередь, получается из pr_0. Тогда имеются такие соотношения: $d[pr_0, pr_1] = 1$, $d[pr_1, pr_2] = 1$. Отождествить pr_1 и pr_2 — это, по сути, заменить эти два соотношение на одно, то есть теперь $d[pr_0, pr_1] = 0$ и $d[pr_1, pr_2] = 0$, но $d[pr_0, pr_2] = 1$, и так далее.

Таким образом, возможно сокращение профилей не менее чем вдвое. Дальнейшие оптимизации мы оставляем читателю.

В итоге получаем асимптотику $2^n n$ (количество переходов, то есть время на вычисление $a[i]$) умножить на m равно $O(2^n nm)$. Она значительно лучше всего, что мы получали до сих пор, и это серьезный повод использовать изломанный профиль вместо обычного.

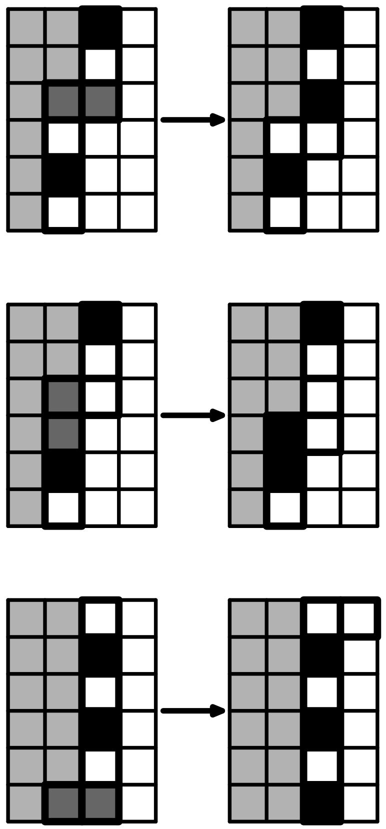


Рис. 6. Возможные переходы.

Задачи, на которых можно потренироваться

1) Сайт `http://acm.sgu.ru`, задачи 131, 132, 197, 223, 225.

2) Московская олимпиада по информатике, 2004 год, заочный тур, задача J («Узор»), `http://www.olympiads.ru/moscow/2004/zaoch/problems.shtml`

.

Декартовы деревья: пример и реализация двоичного дерева поиска

А. Шестимеров

Часто при решении задач необходимо осуществлять поиск данных в каких-то наборах. Пусть у нас есть некоторое множество, и нам требуется проверить, есть ли в этом множестве элемент, добавить элемент в множество, удалить элемент из него. Кроме того, можно предложить много других запросов, например, поиск K-го по величине элемента.

Если в множестве N элементов, то при использовании линейных структур, можно получить сложность порядка $O(N)$ операций для поиска данных и $O(1)$ для добавления (когда мы используем обычный массив) или $O(\log N)$ для поиска и $O(N)$ для добавления (например, если отсортировать элементы, то найти объект можно двоичным поиском) Но для k запросов на поиск и добавление сложность работы будет порядка $O(kN)$.

Мы построим структуру данных, которая, которая позволяет производить операции поиска, добавления и удаления элемента за $O(\log N)$.

В качестве такой структуры данных можно использовать двоичное дерево поиска. Тогда найти элемент можно будет за $O(h)$, где h — глубина (высота) дерева.

Таким образом, чтобы построить оптимальную структуру, необходимо уменьшать глубину дерева. Минимальная глубина — $[\log N] + 1$ — у идеально сбалансированного дерева (у которого для каждой вершины число элементов в левом и правом поддеревьях различаются максимум на 1). Но при добавлении нового элемента, для того чтобы глубина дерева была логарифмом от числа вершин, необходимо сильно перестраивать дерево, а это, к сожалению, очень сложная задача.

Поэтому используют деревья, которые хранят в вершинах дополнительную информацию о дереве. Среди них AVL или сбалансированные деревья, красно-черные деревья и др.

В сбалансированном дереве у каждой вершины высота левого поддерева h_L и высота правого поддерева h_r отличается не более чем на 1 ($|h_L - h_r| \leqslant 1$). Разность высот поддеревьев называют фактором сбалансированности внутреннего узла. Высота поддерева и, соответственно, разность высот левого и правого сына и являются той дополнительной информацией, которая используется для поддержки высоты дерева. Для AVL-деревьев высота дерева превосходит высоту идеально сбалансированного не более чем на 45%.

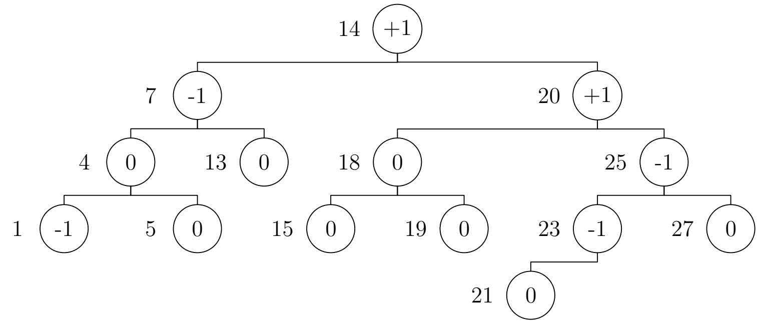


Рис. 1. АВЛ-дерево. В вершинах записан фактор балансировки вершины: 0 — если высоты левого и правого поддеревьев равны, +1 — если правое поддерево выше, −1 — если левое поддерево выше. Слева от вершин записаны ключи элементов.

Для того, чтобы поддерживать высоту дерева, при добавлении нового элемента производится «поворот» или «вращение». То есть для любого множества хранимых элементов следующие два дерева являются деревьями поиска:

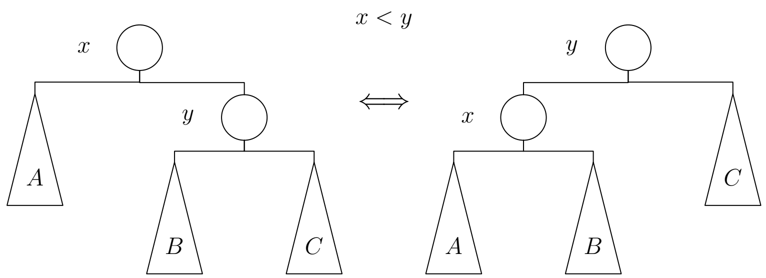


Рис. 2. Общий вид поворота для деревьев.

Тогда при нарушение какого-либо свойства одного из типов деревьев, например, сбалансированности для AVL-деревьев, с помощью поворота оно может быть восстановлено (рис. 3).

Понятно, что проведение одного поворота проходит за $O(1)$, а общее число вращений в сбалансированном дереве пропорционально высоте.

Но для сбалансированных деревьев при поддержании правила разности высот одного типа поворота недостаточно (рис. 4). Необходимо проводить дополнительные модификации или производить так называемый «большой поворот». Итак, для создания этой структуры только для добавления элементов нам понадобится реализовать 4 процедуры (левый и правый малые и большие повороты). Эти процедуры получа-

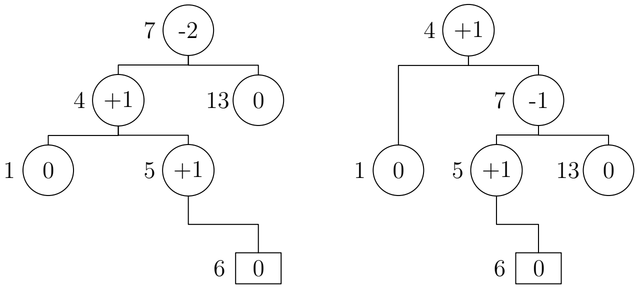


Рис. 3. При добавлении узла с ключом 6 произошла разбалансировка (левое дерево), которая восстановилась при вращении (правое дерево).

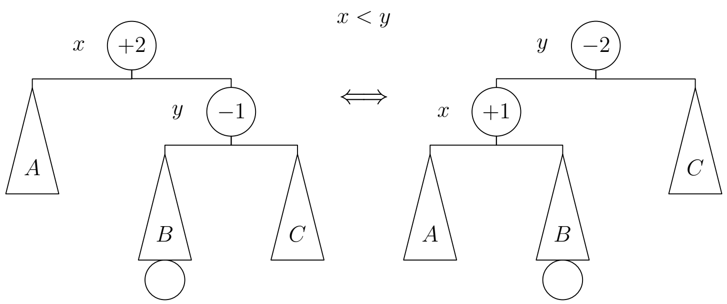


Рис. 4. Случай, когда поворот (для АВЛ-деревьев его называют *малым*) не приводит к балансировке дерева.

ются достаточно большими (более 150 строк кода) и в них легко допустить ошибку.

Хотелось бы найти более простой способ поддержания «небольшой» глубины дерева. Один из самых простых в практической реализации способов, при котором так же хранится дополнительная информация, был предложен Дж. Вуиллемином в 1978 году и основывается на объединении двух структур: двоичного дерева поиска и двоичной кучи (*дуча* или *treap*). Для того, чтобы получить такую структуру, каждому элементу множества назначают некоторый уникальный приоритет. Дуча — это двоичное дерево, в котором для ключей элемента выполняется свойство двоичного дерева поиска, а для приоритетов — правила кучи.

Будем считать, что мы храним в нашей структур пары (x, y), где x — ключ, а y — приоритет элемента. Тогда, если (x, y) — вершина дерева, а (x_L, y_L) и (x_R, y_R) — левый и правый потомки, соответственно, то должны соблюдаться:

1) $x_L < x < x_R$ — правило дерева поиска;

2) $y < y_L$, $y < y_R$ — правило кучи.

Когда у нас есть ключ, по которому производится поиск, и приоритет каждого элемента, мы получаем набор пар (x_i, y_i). Если по ключам x_i построить двоичное дерево поиска, то его глубина будет зависеть от порядка добавления вершин в дерево. Однако, если использовать приоритет, то построенное дерево всегда будет одинаковым.

Действительно, корень дерева обладает минимальным приоритетом, значит, корень дерева определяется однозначно (приоритет у каждой вершины уникальный). По ключу вершины все множество разбивается на два подмножества — те элементы, ключ которых меньше ключа вершины, и те, ключ которых больше. Эти два подмножества образуют два поддерева, которые тоже строятся однозначно.

Представим пары (x_i, y_i) как точки на координатной (декартовой) плоскости. Тогда вершиной дерева будет точка с максимальным y_i, то есть самая верхняя. Из левой части плоскости относительно корня будет строится левое поддерево, а из правой — соответственно правое.

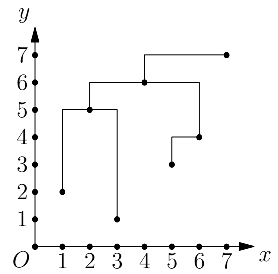


Рис. 5. Декартово дерево для точек $(3, 1)$, $(1, 2)$, $(5, 3)$, $(6, 4)$, $(2, 5)$, $(4, 6)$, $(7, 7)$.

Именно из-за этой интерпретации такую структуру данных называют декартовым деревом. Декартово дерево из набора вершин с фиксированными ключами и приоритетами строится однозначно, то есть не зависит от способа построения). Следовательно, сложность запроса на поиск не зависит от способа добавления элементов. Поэтому, ниже будет предложено два варианта (одинаковых по вычислительной сложности) добавления вершин в дерево.

Перед тем как, приступать к поиску, добавлению и удалению вершин, определим, как хранится дерево в памяти:

```
PNode = TNode;
TNode = record
    x,y:integer; // ключ узла и приортет
    l,r:PNode;
end;
```

Поиск элемента осуществляется как в обычном двоичном дереве:

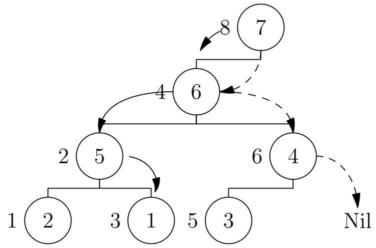


Рис. 6. Поиск элемента с ключом 3 (успешный) и элемента с ключом 7 (неуспешный).

```
function Search(node: PNode; key:integer): PNode;
begin
  if node = nil then Search := node
  else
    if node.x = key then Search := node
    else
      if node.x>key then Search := Search(node.l)
      else
        Search := Search(node.r);
end;
```

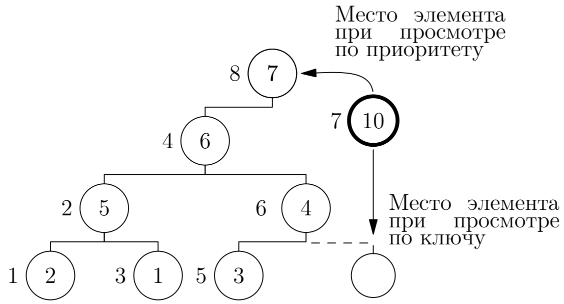


Рис. 7. Добавление в дерево узла (7,10).

Добавление элемента уже отличается от других вариантов построения деревьев. Можно выделить два основных способа построения дучи. Способы различаются только тем, как добавляется элемент в уже построенное корректное дерево — сначала по правилу двоичного дерева или сначала по правилу кучи. Рассмотрим пример — пусть у нас уже есть корректное дерево (рис. 7), в которое мы хотим добавить элемент $(7, 10)$:

Первый способ

Добавим ключ как в обычное двоичное дерево:

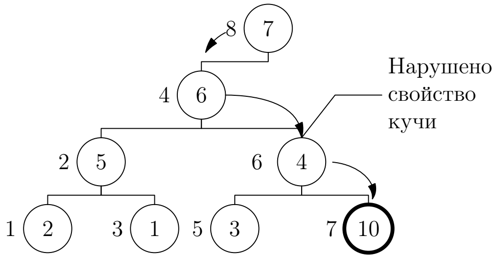


Рис. 8. Добавление в дерево узла (7,10).

```
procedure Insert(var node : PNode; key, priority : integer);
begin
  if node = nil then begin
    new(node);
    node.x := key;
    node.y := priority;
  end else if key<node.x then begin
    insert(node.l, key, priority);
    ...
  end else if key>node.x then begin
    insert(node.r, key, priority);
    ...
  end;
end;
```

В получившемся дереве *может* нарушиться всего одно свойство дучи — свойство кучи для той вершины, к которой подсоединили новую. В этом случае, после левого или правого поворота свойство кучи для этой вершины будет восстановлено.

При повороте общего вида нарушенное свойство кучи исправится:

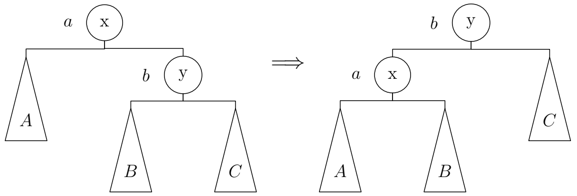


Рис. 9. Левый поворот: $a < b$, $x > A$, $y > B$, $y > c$, следовательно, после поворота получим $y > x > C$; так как до добавления y свойство кучи было выполнено, то $x > B$, то есть поддерево с вершиной y будет дучей.

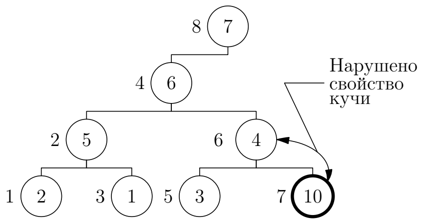


Рис. 10. Нужен левый поворот в узле $(6, 4)$.

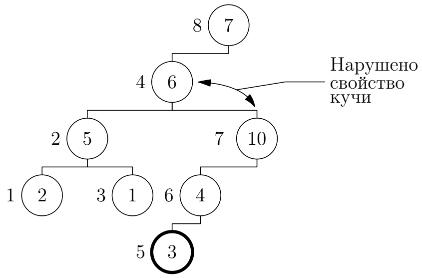


Рис. 11. Нужен левый поворот в узле $(4, 6)$.

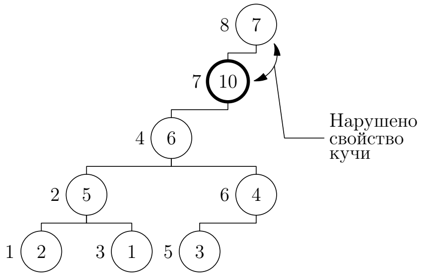


Рис. 12. Нужен правый поворот в узле $(8, 7)$.

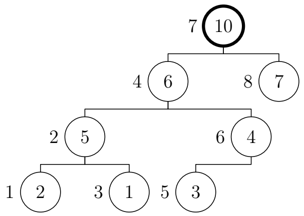


Рис. 13. Построена правильная дуча.

```
procedure left_rotate
    (var p : PNode);
var
  t:PNode;
begin
  t := p.r;
  p.r := p.r.l;
  t.l := p;
  p := t;
end;
```

```
procedure right_rotate
    (var p : PNode);
var
  t:PNode;
begin
  t := p.l;
  p.l := p.l.r;
  t.r := p;
  p := t;
end;
```


Если заменить многоточие в процедуре добавления на

```
if node.l.y>priority then right_rotate(node);
```

для добавления в левого сына и

```
if node.r.y>priority then right_left(node);
```

для добавления в правого сына, получится полная процедура добавления в дерево.

Число операций для поиска места вставки вершины и число поворотов пропорционально глубине дерева.

Второй способ

Находим место в куче, куда должна попасть добавляемая вершина:

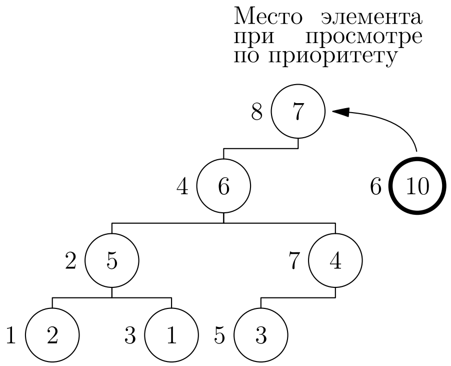


Рис. 14. Добавление в дерево узла (6,10).

```
procedure Insert(var node : PNode; key, priority : integer);
begin
  if node = nil then begin
    new(node);
    node.x := key;
    node.y := priority;
  end else if priority>node.y then begin
    // нашли то место, где должны быть вершина
  end else if key>node.x then
    insert(node.r,key,priority);
  else if key<node.x then
    insert(node.y,key,priority);
end;
```

Теперь надо что-то сделать с поддеревом, корнем которого является та вершина, на место которой должна встать добавляемая (в нашем примере это поддерево с корнем в $(8, 7)$). Все вершины, которые меньше ключа добавляемой, должны войти в левое поддерево, которые больше — в правое поддерево для вершины, которую добавляем (рис. 15).

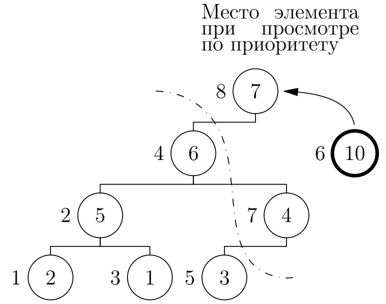


Рис. 15. Узлы разделяются на те, которые попадут в левое и правое поддеревья.

Процедуру разделения дерева проще определить рекурсивно: пустое дерево разделяется на два пустых. В общем случае разделение производится, как показано на рис. 16

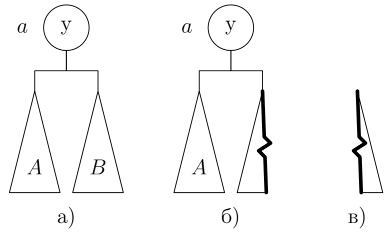


Рис. 16. а) Разбиваем дерево на два по ключу Q $(Q > y)$; б) левое поддерево; в) правое поддерево.

Здесь правое поддерево вершины (a, y) так же разделяется по ключу Q. Точно так же, происходит разделение для случая $Q < y$.

```
procedure Split
   (var node : PNode; key : integer; var left, right : PNode);
begin
  if node = nil then begin
    left := nil;
    right:= nil;
  end begin
    if key>node.x then begin
      split(node.r, x, node.r, right);
```

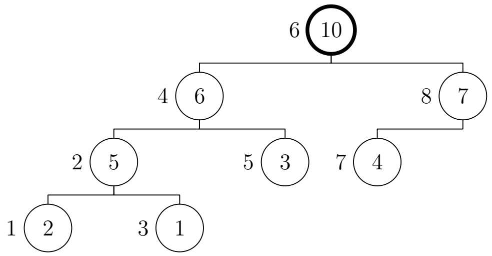



Рис. 17. Разделенные поддеревья становятся левыми и правыми поддеревьями для новой вершины.

```
      left := node;
    end else begin
      split(node.l, x, left, node.l);
      right := node;
    end;
  end;
end;
```

Когда в процедуре добавления мы нашли место в куче, создаем новую вершину и присоединяем два полученых поддерева к ней:

```
          new(tmp);
          split(node, key, tmp.l, tmp.r);
          tmp.x := key;
          tmp.y := priority;
          node := tmp;
```

Удаление элемента так же можно провести двумя способами: первый — уменьшить приоритет вершины таким образом, чтобы она стала листом (при изменении приоритета нарушится свойство кучи, которое восстанавливается поворотами), и затем удалить как лист. Сложность при этом пропорциональна глубине дерева. Второй способ — удаляем вершину сразу, при этом получаем два поддерева, которые надо объединить в одно:

```
function merge(const left, right:PNode):PNode;
begin
  if right = nil then
    merge := left
  else if left = nil then
    merge := right
  else begin
```

```
    // если приоритет левого больше, то он
    // становится корнем
    if left.y > right.y then begin
      left.r := merge(left.r, right);
      merge := left;
    // корнем становится правый, его левый
    // потомок получается объединением левого
    // поддерева и текущего левого потомка
    end else begin
      right.l := merge(right.l, left);
      merge := right;
    end;
  end;
end;
```

Теперь оценим эффективность структуры. Сложность всех операций пропорциональна глубине дерева. Понятно, что можно выбрать приоритеты таким образом, чтобы глубина дерева была равна числу вершин (например, добавляемым вершинам приоритет задается в порядке убывания). Но это худший случай. И если задавать приоритеты случайным образом, то ожидаемая глубина любой вершины и среднее время выполнения операций поиска, добавления и удаления элементов будет $O(\log N)$.

Пусть x_k — k-й по величине элемент, и $A_k^i = 1$, если x_i — предшественник x_k в дереве. Тогда глубина вершины x_k равна $\sum\limits_{i=1}^{n} A_k^i$, и средняя глубина вершины x_k равна $\sum\limits_{i=1}^{n} P(A_k^i = 1)$, где $P(A_k^i = 1)$ — вероятность того, что x_i — предшественник x_k.

Пусть $X(i,k)$ — множество вершин $\{x_i, \ldots, x_k\}$ при $i < k$ и $\{x_k, \ldots, x_i\}$, при $i > k$. Тогда верно следующее утверждение: x_i — предок x_k $(i \neq k)$ тогда и только тогда, когда x_i имеет максимальный приоритет среди $X(i,k)$.

Действительно, если x_i — корень, то утверждение верно, если x_k — корень, то x_i — не имеет максимального приоритета.

Если корень — вершина x_j из $X(i,k)$, то x_i и x_k лежат в разных поддеревьях и x_j — имеет максимальный приоритет.

Если корень x_j — не из $X(i,k)$, то вершины x_i и x_k лежат в одном поддереве и дальше рассматриваем то же утверждение для него.

При выборе приоритетов случайным образом вероятность того, что каждая из вершин принадлежит множеству $X(i,k)$, одинаковая и равна $\dfrac{1}{|k-i|+1}$.

То есть

$$P(A_k^i = 1) = \begin{cases} 0, & \text{если } k = i, \\ \frac{1}{k-i+1}, & \text{если } k > i, \\ \frac{1}{i-k+1}, & \text{если } k < i. \end{cases}$$

При этом, средняя глубина

$$\sum_{i=1}^{n} P(A_k^i = 1) = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} =$$

$$= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k} \frac{1}{j} < \ln k + \ln(n-k) - 2 < 2\ln n.$$

Значит, среднее время операций добавления, удаления и поиска в декартовых деревьях со случайными приоритетами равно $O(\log N)$.

В заключение хочется отметить, что для добавления и удаления элементов у нас получилось два эквивалентных способа. Так, вставка элементов может производиться разделением дерева на два. С другой стороны, если взять второй способ добавления (поворотами), то разделение дерева проще всего сделать, добавив фиктивную вершину с максимальным приоритетом и нужным нам ключом. Эта вершина при поворотах «всплывет» в корень дерева, и её левые и правые поддеревья будут искомыми. Каждый из этих способов прост в написании, и в итоге получается эффективная реализация двоичного дерева поиска.

Дерево Фенвика

А. Лахно

На практике часто возникают задачи, в которых приходится работать с динамически изменяющимися данными. Дерево Фенвика является структурой данных, позволяющей достаточно эффективно выдавать ответы на запросы о частичных суммах на различных отрезках массива, элементы которого могут меняться в процессе работы.

Пусть задан массив A из N чисел: $A_0, A_1, \ldots, A_{N-1}$. Дерево Фенвика — это структура данных, позволяющая выполнять две операции:

- $rsq(i, j)$ — выдать сумму элементов массива A с i-го по j-й включительно (rsq является сокращением от range sum query);
- $update(k, d)$ — прибавить к k-му элементу массива A некоторое число d.

При простейшей реализации $rsq(i, j)$ работает за время $O(j-i+1)$, что в общем случае составляет порядка $O(N)$, а $update(k, d)$ — за $O(1)$. Для ответа на запрос $rsq(i, j)$ достаточно в цикле просуммировать указанный отрезок массива, а при выполнении операции $update(k, d)$ — изменить k-й элемент A_k.

Заметим однако, что при большом количестве запросов вида $rsq(i, j)$ простейшая реализация не слишком эффективна, так как в общем случае ответ на каждый запрос требует времени, зависящего линейно от длины массива.

Дерево Фенвика позволяет существенно сократить это время, поскольку выполняет каждую из указанных операций за время $O(\log N)$.

Описание структуры

На практике дерево Фенвика представляет собой массив B из N чисел: $B_0, B_1, \ldots, B_{N-1}$, в k-м элементе которого хранится сумма элементов массива A с $f(k)$-го по k-й: $B_k = \sum\limits_{i=f(k)}^{k} A_i$, где $f(k) = k \,\&(k+1)$. Под & имеется ввиду операция побитового И.

$$\begin{array}{r} \& \begin{array}{r} k \\ k{+}1 \end{array} \\ \hline f(k) \end{array} \qquad \begin{array}{r} \& \begin{array}{r} \ldots 0111\ldots 1 \\ \ldots 1000\ldots 0 \end{array} \\ \hline \ldots 0000\ldots 0 \end{array}$$

Рис. 1.

Рассмотрим двоичное представление числа k (рис. 1): пусть в нем сначала идет произвольная последовательность 0 и 1, потом 0 и дальше некоторое количество единиц, в том

числе нулевое. $f(k)$ получается из k заменой всех этих подряд идущих единиц в младших разрядах нулями. Если же в младшем разряде числа k стоит 0, то $f(k) = k$. Так, например, для $k = 11_{10} = 1011_2$ получаем $f(k) = 1000_2 = 8_{10}$, а для $k = 10_{10} = 1010_2$ получаем $f(k) = 1010_2 = 10_{10}$. Отметим важное свойство функции f: $0 \leqslant f(k) \leqslant k$ для любого $k \geqslant 0$.

Диаграмма на рис. 2 показывает, что будет храниться в массиве B (дереве Фенвика) при $N = 8$: клетка i-й строки k-го столбца заштрихована, если A_i входит в частичную сумму B_k, то есть $f(k) \leqslant i \leqslant k$.

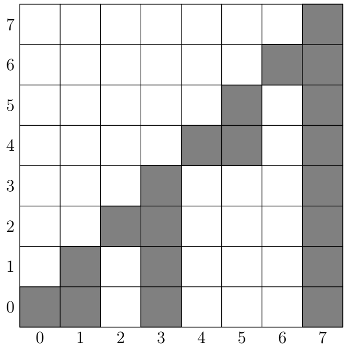


Рис. 2.

Рассмотрим, как с помощью дерева Фенвика реализуются операции ответа на запрос $rsq(i, j)$ о частичной сумме элементов массива A с i-го по j-й и запрос $update(k, d)$ прибавления к k-му элементу массива A числа d.

Ответ на запрос о частичной сумме элементов массива $\boldsymbol{A} - \boldsymbol{rsq(i, j)}$

Заметим, что $rsq(i, j) = rsq(0, j) - rsq(0, i-1)$ (для $i = 0$ положим $rsq(0, -1) = 0$). Таким образом, для ответа на запрос $rsq(i, j)$ достаточно научиться отвечать на запрос вида $rsq(0, k)$, который для краткости обозначим $rsq(k)$.

Ответ на запрос $rsq(k)$ вычисляется следующим образом:

$$rsq(k) = B_k + B_{f(k)-1} + B_{f(f(k)-1)-1} + \ldots + B_{f(f(\ldots f(k)-1\ldots)-1)-1} \tag{1}$$

Перепишем эту формулу в виде

$$rsq(k) = B_{k_0} + B_{k_1} + B_{k_2} + \ldots + B_{k_n}$$

— суммирование начинается с B_k, а индекс каждого следующего слагаемого получается из индекса предыдущего по формуле: $k_{i+1} = f(k_i) - 1 = (k_i \,\&(k_i + 1)) - 1$, где $k_0 = k$. Суммирование заканчивается, когда k_i становится меньше 0, то есть $k_{m+1} = f(k_m) - 1 < 0$. Такой момент наступает, поскольку $k_{i+1} = f(k_i) - 1 \leqslant k_i - 1 < k_i$, то есть каждый следующий индекс строго меньше предыдущего.

Покажем, почему указанная сумма (1) действительно будет являться ответом на запрос $rsq(k)$, то есть $B_{k_0} + B_{k_1} + B_{k_2} + \ldots + B_{k_n} = \sum\limits_{i=0}^{k} A_i$.

Действительно,

$$B_{k_0} + B_{k_1} + B_{k_2} + \ldots + B_{k_n} =$$

$$= \sum_{i=f(k_0)}^{k_0} A_i + \sum_{i=f(k_1)}^{k_1} A_i + \sum_{i=f(k_2)}^{k_2} A_i + \ldots + \sum_{i=f(k_m)}^{k_m} A_i =$$

$$= \sum_{i=f(k_0)}^{k_0} A_i + \sum_{i=f(k_1)}^{f(k_0)-1} A_i + \sum_{i=f(k_2)}^{f(k_1)-1} A_i + \ldots + \sum_{i=f(k_m)}^{f(k_{m-1})-1} A_i =$$

$$= \sum_{i=f(k_m)}^{k_0} A_i.$$

Для последнего индекса k_m верно: $f(k_m) - 1 < 0$, откуда в силу неотрицательности $f(k_m)$ следует $f(k_m) = 0$, то есть $\sum\limits_{i=f(k_m)}^{k_0} A_i = \sum\limits_{i=0}^{k} A_i$. Таким образом, указанная сумма (1) и будет суммой всех элементов массива A с 0-го по k-й.

Например, для $k = 14$ по формуле (1) имеем:

$$rsq(14) = B_{14} + B_{13} + B_{11} + B_7 =$$

$$= \sum_{i=14}^{14} A_i + \sum_{i=12}^{13} A_i + \sum_{i=8}^{11} A_i + \sum_{i=0}^{7} A_i = \sum_{i=0}^{14} A_i.$$

Ниже приводится текст подпрограммы, реализующий ответ на запрос $rsq(k)$:

```
 1: Function rsq(k : LongInt) : LongInt;
 2: var
 3:   res : LongInt;
 4: begin
 5:   res := 0;
 6:   while (k >= 0) do begin
 7:     inc(res, B[k]);
 8:     k := k and (k + 1) - 1;
 9:   end;
10:   rsq := res;
11: end;
```

Для оценки времени работы данной подпрограммы необходимо оценить количество итераций цикла `while` в строках 6–9, то есть количество слагаемых m в сумме (1). Для этого рассмотрим двоичное представление числа k и то, как оно преобразуется при вычислении номера следующего индекса суммы по предыдущему: $k_{i+1} = f(k_i) - 1$, где $k_0 = k$.

k_0	14	1110_2
$f(k_0)$	14	1110_2
k_1	13	1101_2
$f(k_1)$	12	1100_2
k_2	11	1001_2
$f(k_2)$	8	1000_2
k_3	7	0111_2
$f(k_3)$	0	000_2

Рис. 3.

Сделаем это на примере для $k = 14$ (рис. 3). Количество слагаемых равно количеству единичных битов в числе $f(k)$ плюс один. Переход к каждому следующему индексу «выбивает» ровно по одному единичному биту из числа $f(k_i)$. А поскольку в худшем случае количество единичных бит в $f(k)$ может быть порядка $O(\log N)$, то и ответ на запрос $rsq(k)$ в худшем случае требует порядка $O(\log N)$ времени.

Изменение k-го элемента массива A — $update(k, d)$

При изменении k-го элемента массива A необходимо соответствующим образом изменить элементы массива B, в определении которых частичные суммы содержат элемент A_k. То есть для выполнения операции $update(k, d)$ (прибавления к k-му элементу массива A числа d) нужно прибавить d к тем элементам B_j массива B, для которых $f(j) \leqslant k \leqslant j$.

Утверждается, что все такие j, и только они, являются элементами последовательности $k_0, k_1, k_2, \ldots, k_m$, где $k_0 = k$, $k_{i+1} = k_i \mid (k_i + 1)$, $k_{m+1} = k_m \mid (k_m + 1) > N - 1$. Под $\mid$ имеется в виду операция побитового ИЛИ.

Сначала покажем, что указанная последовательность строго возрастает, и ее длина составляет порядка $O(\log N)$ в худшем случае. Для этого рассмотрим двоичное представление числа k_i (рис. 4). Число k_{i+1} получается из k_i заменой младшего нулевого бита на единичный. Таким образом, $k_{i+1} > k_i$, а длина последовательности m заведомо не превосходит количества бит в двоичном представлении числа N.

$$\begin{array}{rr} & k_i \\ \mid & k_i{+}1 \\ \hline & k_{i+1} \end{array} \qquad \begin{array}{rr} & \ldots 0111 \ldots 1 \\ \mid & \ldots 1000 \ldots 0 \\ \hline & \ldots 1111 \ldots 1 \end{array}$$

Рис. 4.

На рис. 5 приведен пример указанной последовательности для $N = 93$ и $k = 73$.

$N - 1$	92	1011100_2
$k_0 = k$	73	1001001_2
k_1	75	1001011_2
$k_2 = k_m$	79	1001111_2
k_3	95	1011111_2

Рис. 5.

Покажем, почему элементы последовательности являются индексами тех и только тех элементов массива B, которые нужно изменить при операции $update(k, d)$.

j	$X0\ldots$
k	$Y1\ldots$

Рис. 6.

Поскольку при увеличении номера элемента последовательности, количество единичных битов на конце k_i возрастает, то $f(k_i)$ убывает. Учитывая еще и то, что сама последовательность k_i возрастает, получаем:

$$f(k_m) < \ldots < f(k_2) < f(k_1) < f(k_0) \leqslant k = k_0 < k_1 < k_2 < \ldots < k_m$$

Таким образом, все элементы нашей последовательности являются индексами тех элементов массива B, которые надо изменить.

Покажем, что других элементов, которые надо изменить, нет. Предположим, что есть некоторый индекс j, не лежащий в нашей последовательности, такой что $f(j) \leqslant k \leqslant j$.

Рассмотрим младший бит, в котором у j стоит 0, а у k стоит 1 — такой существует, поскольку j не встречается в последовательности k_i. Обозначим числа, образованные более старшими битами чисел j и k, через X и Y соответственно. Если $X > Y$, то $j \geqslant f(j) \geqslant X0\ldots0_2 > k$, а стало быть, частичная сумма B_j не содержит элемента A_k. Если же $X \leqslant Y$, то $k > j \geqslant f(j)$ и опять же частичная сумма B_j не содержит элемента A_k. Значит, индексы всех элементов массива B, которые надо изменить присутствуют в последовательности k_i.

Ниже приводится текст подпрограммы, реализующий операцию изменения `update(k, d)`:

```
1: Procedure update(k, d : LongInt);
2: begin
3:   while (k < N) do begin
4:     inc(B[k], d);
5:     k := k or (k + 1);
6:   end;
7: end;
```

В силу того, что длина последовательности k_i составляет $O(\log N)$ в худшем случае, количество итераций цикла `while` в строках 3–6, а значит, и время работы процедуры `update(k, d)`, составляет $O(\log N)$.

Построение дерева Фенвика

Для построения дерева Фенвика по заданному массиву A достаточно изначально взять нулевой массив B из N элементов и N раз вызвать операцию изменения элемента *update*: $update(0, A_0)$, $update(1, A_1)$, …, $update(N-1, A_N - 1)$. Таким образом, построение дерева Фенвика требует $O(N \log N)$ времени и $O(N)$ памяти.

Обобщение на двумерный случай

Оказывается, что дерево Фенвика достаточно легко обобщается на двумерный (а вообще говоря, и на k-мерный) случай.

Пусть задан двумерный массив A размера $M \times N$, элементами которого являются числа A_{ij}, $0 \leqslant i \leqslant M-1$, $0 \leqslant j \leqslant N-1$. Двумерное дерево Фенвика — это структура данных, позволяющая выполнять две операции:

- $rsq(x1, y1, x2, y2)$ — выдать сумму $\sum\limits_{\substack{x1 \leqslant x \leqslant x2,\\ y1 \leqslant y \leqslant y2}} A_{xy}$ элементов A_{xy} на прямоугольнике $x1 \leqslant x \leqslant x2$, $y1 \leqslant y \leqslant y2$;
- $update(x, y, d)$ — прибавить к элементу A_{xy} некоторое число d.

Двумерное дерево Фенвика представляет собой двумерный массив B размера $M \times N$, элементами которого являются числа B_{xy}, $0 \leqslant x \leqslant M-1$, $0 \leqslant y \leqslant N-1$. $B_{xy} = \sum\limits_{\substack{f(x) \leqslant i \leqslant x,\\ f(y) \leqslant j \leqslant y}} A_{ij}$.

По аналогии с одномерным случаем заметим (рис. 7), что

$$rsq(x1, y1, x2, y2) = rsq(0, 0, x2, y2) - rsq(0, 0, x1-1, y2) - \\ - rsq(0, 0, x2, y1-1) + rsq(0, 0, x1-1, y1-1).$$

Таким образом, для ответа на запрос $rsq(x1, y1, x2, y2)$ достаточно научиться отвечать на запросы вида $rsq(0, 0, x, y)$, которые для краткости обозначим $rsq(x, y)$.

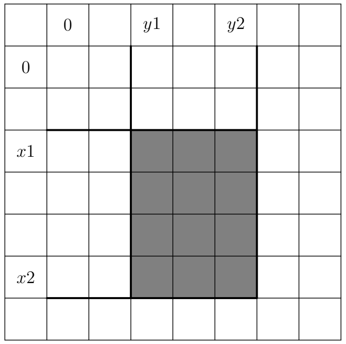


Рис. 7.

В остальном, реализация операций $rsq(x, y)$ и $update(x, y, d)$ делается аналогично одномерному случаю из соображений независимости по каждому направлению.

Ниже приводится текст подпрограмм, реализующих операции $rsq(x, y)$ и $update(x, y, d)$:

```
Function rsq(x, y : LongInt) : LongInt;
var
  i,res : LongInt;
begin
  res := 0;
  while (x >= 0) do begin
    i := y;
    while (i >= 0) do begin
```

```
      inc(res, B[x, i]);
      i := i and (i + 1) - 1;
    end;
    x := x and (x + 1) - 1;
  end;
  rsq := res;
end;
Procedure update(x, y, d : LongInt);
var
  i : LongInt;
begin
  while (x < M) do begin
    i := y;
    while (i < N) do begin
      inc(B[x, i], d);
      i := i or (i + 1);
    end;
    x := x or (x + 1);
  end;
end;
```

По соображениям, аналогичным тем, что приводились в одномерном случае, времена работы $rsq(x, y)$ и $update(x, y, d)$ составляют $O((\log M)(\log N))$ и $O((\log M)(\log N))$ соответственно.

Для построения двумерного дерева Фенвика достаточно взять изначально нулевой массив B и $M \times N$ раз применить операцию *update*: $update(x, y, A_{xy})$, $0 \leqslant x \leqslant M - 1$, $0 \leqslant y \leqslant N - 1$. Построение двумерного дерева Фенвика требует $O(MN(\log M)(\log N))$ времени и $O(MN)$ памяти.

Пример использования

Разберем такую задачу.

Папа Карло нашел длинную палку в сарае и решил сделать из нее вешалку для ключей. Для этого он по всей длине забивает в палку гвоздики в разных местах. Папа Карло время от времени считает, сколько он уже забил гвоздиков на разных участках палки.

Каждое действие Папы Карло — это либо забивание очередного гвоздика в точке с заданной координатой X, *либо подсчет числа забитых гвоздиков на отрезке* $[X, Y]$. *Все координаты неотрицательные и не превосходят* 10^9, *количество действий не превосходит* 10^5.

Буратино, узнав, в чем дело, решил помочь Папе Карло. Буратино подробно записал все действия Папы Карло в процессе изготовления вешалки и теперь хочет проверить расчеты по числу гвоздиков на разных интервалах, сделанные во время работы. Помогите ему в этом.

Требуется выдать результаты всех расчетов по числу гвоздиков на разных интервалах.

Решение. Если для каждой координаты хранить количество забитых в точку с этой координатой гвоздиков, то подсчет числа гвоздиков на отрезке $[X, Y]$ является запросом $rsq(X, Y)$ к соответствующему дереву Фенвика.

Однако при ограничениях на координаты до 10^9 такой подход не проходит ни по требуемой памяти, ни по времени работы.

Решением проблемы в данном случае является так называемый «метод сжатия координат». Суть его состоит в следующем: отсортируем по координате все точки, встречающиеся во входном файле (их количество не превосходит $2 \cdot 10^5$). Тут надо учитывать как точки, в которые забиваются гвоздики, так и концы отрезков, на которых производится подсчет числа гвоздиков.

Перенумеруем точки в соответствии с их позицией в отсортированном массиве. При этом точкам с одинаковыми координатами присвоим одинаковые номера.

После этого, заменив координаты всех точек их номерами, задачу можно решать с помощью дерева Фенвика, как это было предложено вначале, поскольку все номера не превосходят $2 \cdot 10^5$.

Литература

[1] *P. M. Fenwick*, A new data structure for cumulative frequency tables. Software — Practice and Experience 24, 3 (1994), 327-336, 1994.

Игры и стратегии

А. Фонарёв

Статья предназначена для широкого круга читателей, интересующихся программированием и желающих познакомиться с таким замечательным предметом, как теория игр. Автор не претендует на обзор этой молодой ветви математики, а лишь разберет некоторые конкретные примеры доступные большинству старшеклассинков.

Раздел 1 расскажет об играх на графах (к которым относятся, например, шахматы). Будет изложен алгоритм, дающий одному из игроков оптимальную стратегию поведения.

Для понимания материала раздела 3 читателю потребуются базовые знания математического анализа и линейной алгебры. Конечная цель — доказать, что всякая матричная игра имеет решение в смешанных стратегиях.

Раздел 2 находится между описанными частями как в прямом, так и в переносном смысле слова. Он расскажет о функции Шпрага-Гранди, имеющей не только теоретическое, но и прикладное значение.

При написании данного текста основное внимание уделялось именно сути вопроса. Автор просит извинить его за отсутствие строгости в некоторых местах (иногда приходится делать выбор между доступностью языка и формализмом). И, конечно же, настоятельно рекомендуется решать все встречающиеся в тексте задачи.

1. Игры на графах

Игры на ациклических графах

Рассмотрим конечный ориентированный граф без петель и кратных ребер, исходящая степень некоторых из вершин которого равна 0. Пусть в одной из вершин находится фишка. Двое по очереди перемещают ее по дугам (то есть из вершины a можно пойти в вершину b, если и только если существует ребро ab). Проигрывает тот из игроков, кто не может сделать хода. Такую игру назовем игрой на конечном графе или просто *игрой на графе.*

Задача 1. *Представим, что ничья для одного из игроков равносильна проигрышу. Покажите, что в таком случае шашки и шахматы являются играми на графах.*

Данный класс игр хорош тем, что он не слишком узок и, в то же время, легко описывается математическим языком. Вопрос: есть ли у

одного из игроков выигрышная стратегия, иначе говоря, может ли он гарантировать себе победу вне зависимости от действий соперника. Эту задачу мы и будем решать, ограничившись для начала случаем ациклического графа.

Во-первых, в отсутствии циклов игра заведомо конечна. Во-вторых, оказывается, что у одного из игроков имеется стратегия оптимального поведения.

Напомним, что топологической сортировкой вершин графа называется такая их нумерация, что начало любого ребра имеет больший номер, чем его конец.

Задача 2. *Докажите, что граф можно топологически отсортировать тогда и только тогда, когда он не содержит циклов* (*в том числе петель*).

Назовем вершину *выигрышной*, если, находясь в ней, первый игрок может гарантировать себе победу, *проигрышной* — если это может сделать второй. Топологически отсортируем граф и докажем по индукции (параметр — номер вершины после сортировки), что каждую его позицию можно так классифицировать.

Вершины с исходящей степенью 0, естественно, проигрышные; из первой ни одно ребро вести не может, поэтому база индукции выполнена. Рассмотрим очередную непомеченную вершину. Пусть концы всех исходящих из нее ребрер выигрышные. Это значит, что после нашего хода противник может всегда гарантировать себе победу, следовательно, текущая вершина проигрышная. Наоборот, пусть существует ход в проигрышную позицию, тогда, сделав его, мы поставим соперника в безвыходное положение. Получаем, что рассматриваемая вершина выигрышная. Итак, мы только что предъявили способ определения выигрышности/проигрышности позиций.

Описанный выше метод хорош тем, что он может быть эффективно запрограммирован. Скажем, что алгоритм работает за время $O(f)$, если он всегда выполняет не более, чем Cf действий, где C — константа, а f — некоторая функция, зависящая от входных данных[2]. Пусть n и m — число вершин и ребер графа соответственно, v_i — исходящая степень вершины с номером i. Топологическая сортировка графа эффективно выполняется за время $O(n+m)$. На обработку всех вершин уходит $O(v_1+\ldots+v_n)$ действий. Исходя из того, что $\sum_{i=1}^{n} v_i = m$, получаем конечную оценку — $O(n+m)$.

[2]В большинстве случаев лишь от размера входных данных.

Случай произвольного графа

Попробуем обобщить построенный алгоритм на графы с циклами. Будем пробовать помечать вершины последовательно, пользуясь наблюдениями предыдущего раздела.

1) Если из вершины все ребра ведут в выигрышные, то она проигрышная.

2) Если из вершины существует ход в проигрышную, то она выигрышная.

Задача 3. *Покажите, что не любая классификация вершин графа, удовлетворяющая данным условиям, удовлетворяет здравому смыслу.*

Заведем очередь, в которую изначально добавим все вершины с исходящей степенью 0. На каждом шаге алгоритма будем извлекать очередную вершину. Пусть она оказалась проигрышной, тогда все непомеченные, из которых в нее ведут ребра, можно смело объявлять выигрышными и добавлять в очередь. Второй случай чуть сложнее. Требуется быстро определять вершины с первым свойством. Эта проблема решается так: заведем для каждой вершины счетчик c_i, где будет записано количество выигрышных позиций, в которые можно пойти из данной. Рассматривая очередную выигрышную вершину, пробежимся по всем ситуациям, которые могут ей предшествовать и увеличим их счетчики. Если случилось так, что c_i стал равен исходящей степени вершины, пометим ее как проигрышную и добавим в очередь. Алгоритм завершается, когда последняя окажется пустой. Заметим, что сложность осталась той же — $O(n+m)$.

Задача 4. *Дайте разумное определение ничейной вершины и докажите, что именно они остались неклассифицированными в результате выполнения алгоритма.*

Задача 5. *Модифицируйте алгоритм так, чтобы он либо позволял выиграть за минимальное число ходов, либо максимально оттянуть поражение.*

2. Функция Шпрага-Гранди

Функция Шпрага-Гранди была независимо введена математиками *Sprague* и *Grundy* в 1935 и 1939 годах соответственно. Все дальнейшие формулировки сделаны для игр на ациклических графах и немного беднее, чем могли бы быть. Это сделано специально, ибо данный раздел носит в большей мере прикладное значение, чем теоретическое.

Пусть V и E — множества вершин и ребер графа соответственно, $\mathbb{Z}_+$ — множество целых неотрицательных чисел. За $\deg v$ обозначим исходящую степень вершины v. Функция Шпрага-Гранди $Gr\colon V \to \mathbb{Z}_+$ определяется так: если $\deg v = 0$ то $Gr(v) = 0$, иначе $Gr(v) = \min\{\mathbb{Z}_+ \setminus S\}$, где S — множество чисел, состоящее из значений функции в позициях, в которые можно попасть из v за один ход, то есть $S = \{Gr(u) \mid vu \in E\}$.

Задача 6. *Посчитайте функцию Шпрага-Гранди для нескольких графов и определите выигрышность/проигрышность каждой позиции.*

Если Вы честно проделали предыдущее упражнение, то должны были сделать следующее наблюдение: позиция проигрышная тогда и только тогда, когда $Gr(v) = 0$.

Задача 7. *Докажите предыдущее утверждение (например, предъявив выигрышную стратегию одного из игроков).*

Функция Шпрага-Гранди обрадает одним интересным свойством.

▷ **Определение 1.** Игра G называется *суммой* игр $G = G_1 + \ldots + G_n$, если на каждом ходу игрок выбирает ровно одну из игр G_i и делает в ней ход[3].

Задача 8. *Покажите, что сумма игр на ациклических графах — также игра на ациклическом графе.*

Теорема 1. *Пусть* $G = G_1 + \ldots + G_n$, p_i *— текущая позиция в игре* G_i. *Тогда если* $p = p_1 + \ldots + p_n$ *— позиция в суммарной игре, то* $Gr(p) = Gr(p_1) \oplus \ldots \oplus Gr(p_n)$, *где* $\oplus$ *— побитовое исключающее или.*

Доказательство. Так как G является игрой на конечном ациклическом графе, все позиции можно топологически упорядочить. Проведем доказательство по индукции (параметр — номер вершины после сортировки). База выполнена, так как для первой вершины $Gr(p) = 0 = 0 \oplus \ldots \oplus 0$. Пусть p_i — текущая позиция в игре G_i, $a_i = Gr(p_i)$, $A = a_1 \oplus \ldots \oplus a_n$. Чтобы был выполнен индукционный переход, докажем два факта.

1) Позиции со значениями Gr, меньшими, чем A, достижимы за один ход.

2) Не существует хода в позицию со значением A.

[3] Здесь ход уже в смысле выбранной игры.

Начнем со второго. По построению игры G изменится не более одного a_i. Из определения функции Шпрага-Гранди следует, что оно не могло остаться прежним, а, значит, изменится и значение $a_1 \oplus \ldots \oplus a_n$. Пусть $0 \leqslant B < A$. Рассмотрим старший двоичный разряд из тех, в которых A и B различаются. Так как $B < A$, у A он равен единице, а у B — нулю. Первое означает, что найдется a_i, в котором он также ненулевой. Проверку того, что искомый ход можно выполнить именно в игре G_i, оставляем в качестве упражнения читателю. □

3. Матричные игры

Основные понятия

Теория игр — раздел математики, в котором исследуются модели принятия решений в условиях столкновения интересов сторон. С этого момента от читателя потребуются базовые знания линейной алгебры и анализа, хотя изложение будет максимально доступным.

Определение 2. Система

$$\Game = (X, Y, K),$$

где X и Y — непустые множества, а $K : X \times Y \to \mathbb{R}$, называется *антагонистической игрой в нормальной форме.*

Элементы $x \in X$ и $y \in Y$ называются *стратегиями* игроков 1 и 2 соответственно, элемент $(x, y) \in X \times Y$ — позицией в игре $\Game$, а K — функцией выигрыша первого игрока. Выигрыш второго игрока в ситуации (x, y) положим равным $-K(x, y)$. В подобных случаях говорят, что $\Game$ — *игра с нулевой суммой.*

Описанная модель интерпретируется так: двое одновременно и независимо выбирают стратегии из множеств X и Y соответственно, причем первый получает выигрыш равный $K(x, y)$[4].

Определение 3. Антагонистическая игра, в которой оба игрока имеют лишь конечное число стратегий, называется *матричной.*

Введем несколько удобных обозначений для матричных игр. Пусть $m = |X|$, $n = |Y|$, тогда без ограничения общности можно отождествлять X с множеством $M = \{1, 2, \ldots, m\}$, Y — с $N = \{1, 2, \ldots, n\}$, а функция выигрыша удобно запишется матрицей $A = \{\alpha_{ij}\}$ размера $m \times n$, где $\alpha_{ij} = K(x_i, y_j)$.

Рассмотрим антагонистическую игру $\Game = (X, Y, K)$. Какой выигрыш может гарантировать себе первый игрок? Пусть он выбрал стратегию $x_0 \in X$. В предположении, что второй игрок действует лучшим для себя образом, выигрыш первого будет сколь угодно близок (сверху) к

[4] Принято считать, что отрицательный выигрыш равносилен проигрышу.

$\inf_{y\in Y} K(x_0, y)$. Переходя ко всему множеству стратегий X, получаем оценку

$$\underline{v} = \sup_{x\in X} \inf_{y\in Y} K(x, y),$$

которую назовем *нижним значением игры.* Итак, первый игрок может обеспечить себе выигрыш, сколь угодно близкий к $\underline{v}$. Если внешний экстремум достигается, $\underline{v}$ также называют *максимином*, а стратегию $x^* \in X$, такую что $\inf_{y\in Y} K(x^*, y) = \underline{v}$ — *стратегией максимина.*

Если рассмотреть ту же игру со стороны второго игрока, аналогично определяются *верхнее значение игры*

$$\overline{v} = \inf_{y\in Y} \sup_{x\in X} K(x, y),$$

минимакс и *стратегия минимакса.*

Лемма 1. *В антагонистической игре $\underline{v} \leqslant \overline{v}$.*

Доказательство. Рассмотрим произвольные $x \in X$ и $y \in Y$, тогда

$$K(x, y) \leqslant \sup_{x\in X} K(x, y).$$

Переходя к точной нижней грани по y, получаем

$$\inf_{y\in Y} K(x, y) \leqslant \inf_{y\in Y} \sup_{x\in X} K(x, y).$$

Теперь в правой части стоит константа, а $x \in X$ в левой части выбиралось произвольно, откуда

$$\sup_{x\in X} \inf_{y\in Y} K(x, y) \leqslant \inf_{y\in Y} \sup_{x\in X} K(x, y).$$

Что и требовалось доказать. □

Определение 4. Позиция $(x^*, y^*) \in X \times Y$ в игре $\Game = (X, Y, K)$ называется *ситуацией равновесия* или *седловой точкой*, если ни одному из игроков не выгодно от нее отклоняться. Иными словами, для любых $x \in X$ и $y \in Y$ выполнено

$$K(x, y^*) \leqslant K(x^*, y^*) \leqslant K(x^*, y).$$

Найти ситуации равновесия и означает решить игру в случае оптимального поведения игроков. Множество седловых точек в игре $\Game$ обозначим за $\mathrm{Z}(\Game) \subseteq X \times Y$.

Лемма 2. *Пусть (x_1^*, y_1^*) и (x_2^*, y_2^*) — ситуации равновесия в антагонистической игре $\Game$, тогда*

1) $K(x_1^*, y_1^*) = K(x_1^*, y_2^*) = K(x_2^*, y_1^*) = K(x_2^*, y_2^*)$;

2) $(x_1^*, y_2^*) \in \mathrm{Z}(\Game)$, $(x_2^*, y_1^*) \in \mathrm{Z}(\Game)$.

Доказательство. Из определения седловой точки для любых $x \in X$ и $y \in Y$ имеем

$$K(x, y_1^*) \leqslant K(x_1^*, y_1^*) \leqslant K(x_1^*, y),$$
$$K(x, y_2^*) \leqslant K(x_2^*, y_2^*) \leqslant K(x_2^*, y).$$

Отсюда

$$K(x_2^*, y_1^*) \leqslant K(x_1^*, y_1^*) \leqslant K(x_1^*, y_2^*) \leqslant K(x_2^*, y_2^*) \leqslant K(x_2^*, y_1^*),$$

что и доказывает первую часть леммы. При произвольном выборе $x \in X$ и $y \in Y$

$$K(x, y_1^*) \leqslant K(x_1^*, y_1^*) = K(x_1^*, y_2^*) = K(x_2^*, y_2^*) \leqslant K(x_2^*, y).$$

Аналогично доказывается, что $(x_2^*, y_1^*) \in \mathrm{Z}(\Game)$. □

Пусть X^* и Y^* — проекции $\mathrm{Z}(\Game)$ на X и Y соответственно[5], тогда по доказанной лемме $\mathrm{Z}(\Game) = X^* \times Y^*$.

▷ **Определение 5.** Множество X^* (Y^*) называется *множеством оптимальных стратегий* первого (второго) игрока.

▷ **Определение 6.** Пусть (x^*, y^*) — ситуация равновесия в антагонистической игре $\Game$, тогда $v_{\Game} = K(x^*, y^*)$ называется *значением игры* $\Game$.

Корректность последнего определения следует из леммы 2. Если ясно, о какой игре идет речь, вместо $v_{\Game}$ будем писать просто v. Теперь мы готовы доказать некоторый содержательный факт.

Теорема 2. *Для того, чтобы в игре $G = (X, Y, K)$ существовала ситуация равновесия, необходимо и достаточно, чтобы достигались и были равны*

$$\underline{v} = \max_{x \in X} \inf_{y \in Y} K(x, y) = \min_{y \in Y} \sup_{x \in X} K(x, y) = \overline{v}.$$

Доказательство. Необходимость. Пусть (x^*, y^*) — точка равновесия. Имеем:

$$\inf_{y \in Y} \sup_{x \in X} K(x, y) \leqslant \sup_{x \in X} K(x, y^*) \leqslant K(x^*, y^*),$$
$$\sup_{x \in X} \inf_{y \in Y} K(x, y) \geqslant \inf_{y \in Y} K(x^*, y^*) \geqslant K(x^*, y^*).$$

[5] Проекцией множества $K \subset X_1 \times X_2$ на множество X_i, $i = 1, 2$, называется образ K при отображении $(x_1, x_2) \mapsto x_i$.

Откуда видно, что

$$\overline{v} = \inf_{y \in Y} \sup_{x \in X} K(x, y) \geqslant \sup_{x \in X} \inf_{y \in Y} K(x, y) = \underline{v}.$$

Обратное неравенство составляет утверждение леммы 2. Следовательно,

$$\underline{v} = K(x^*, y^*) = \overline{v},$$

причем внешние экстремумы достигаются в точках x^* и y^* соответственно.

Достаточность. Пусть $\underline{v} = \overline{v}$, максимум и минимум достигаются в точках $x^* \in X$ и $y^* \in Y$ соответственно.

$$\underline{v} = \sup_{x \in X} \inf_{y \in Y} K(x, y) = \inf_{y \in Y} K(x^*, y) \leqslant K(x^*, y^*),$$
$$\overline{v} = \inf_{y \in Y} \sup_{x \in X} K(x, y) = \sup_{x \in X} K(x, y^*) \geqslant K(x^*, y^*).$$

Из этих неравенств и предположения $\underline{v} = \overline{v}$ следует, что

$$K(x, y^*) \leqslant K(x^*, y^*) \leqslant K(x^*, y)$$

при любых $x \in X$ и $y \in Y$. Иными словами, $(x^*, y^*) \in \mathrm{Z}(\Game)$. Что и требовалось доказать. □

Перед тем, как перейти к новому этапу, сформулируем лемму, доказательство которой оставляем читателю, как упражнение.

Лемма 3 (О масштабе)**.** *Пусть $\Game = (X, Y, K)$ и $\tilde{\Game} = (\tilde{X}, \tilde{Y}, \tilde{K})$ — антагонистические игры, причем $\tilde{K} = \alpha K + \beta$, тогда*

$$\mathrm{Z}(\tilde{\Game}) = \mathrm{Z}(\Game), \ \tilde{v} = \alpha v + \beta$$

(*если значение игры существует*).

Смешанное расширение игры

К сожалению, равновесие в матричной игре существует не всегда. Например, для случая

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

$\underline{v} = -1 \neq 1 = \overline{v}$, то есть седловых точек нет.

Чтобы справиться с возникшей трудностью, дадим следующее определение.

▷ **Определение 7.** Игра $\tilde{\Game} = (\tilde{X}, \tilde{Y}, \tilde{K})$ называется *смешанным расширением игры* $\Game = (X, Y, K)$, если $\tilde{X}$ $(\tilde{Y})$ — множество случайных величин со значениями в X (Y), а $\tilde{K}$ — математическое ожидание случайной величины $K(\tilde{x}, \tilde{y})$ (где $\tilde{x} \in \tilde{X}$ и $\tilde{y} \in \tilde{Y}$ фиксированы).

В последнем случае мы отождествляем стратегии игроков (случайные величины со значениями в конечном множестве) с их вероятностными распределениями.

$$\tilde{X} = \{p \in \mathbb{R}^m : p_i \geqslant 0, \sum p_i = 1\}, \qquad \tilde{Y} = \{q \in \mathbb{R}^n : q_j \geqslant 0, \sum q_j = 1\}.$$

За этим определением стоит очень естественная ситуация: представим, что разыгрывается большое число партий, тогда игроки могут выбирать каждую из своих стратегий с некоторой вероятностью (возможно, единичной или нулевой). Говорить о каком-либо определенном выигрыше уже не приходится, зато можно посчитать средний результат. Это и есть математическое ожидание.

Выясняется, что смешанного расширения уже достаточно, чтобы в любой игре найти седловую точку.

Теорема 3. *Смешанное расширение матричной игры имеет ситуацию равновесия.*

Доказательство. Без ограничения общности будем считать, что A — матрица, состоящая из строго положительных элементов (следствие леммы о масштабе).

Рассмотрим задачу линейного программирования

$$\begin{gathered} xA \geqslant u, \\ x \geqslant 0, \\ xw \longrightarrow \min \end{gathered}$$

и двойственную к ней

$$\begin{gathered} Ay \leqslant w, \\ y \geqslant 0, \\ uy \longrightarrow \max, \end{gathered}$$

где u — вектор-строка размера n, состоящая из одних единиц, а w — вектор-столбец размера m, также состоящий из одних единиц.

Из строгой положительности матрицы A следует, что задачи имеют решения. Пусть это x_0 и y_0. По теореме о слабой двойственности $x_0 w = u y_0$; обозначим эту величину за Θ. Определим $\tilde{x} = x_0/\Theta$, $\tilde{y} = y_0/\Theta$.

Пусть x — некоторая смешанная стратегия первого игрока. Тогда

$$1 = xw \geqslant xA\tilde{y} \quad \Rightarrow \quad 1/\Theta \geqslant xA\tilde{y}.$$

Аналогично $\tilde{x}Ay \geqslant 1/\Theta$. С другой стороны

$$\Theta = x_0 w \leqslant x_0(Ay_0) = (x_0 A) y_0 \leqslant u y_0 = \Theta.$$

Значит, $1/\Theta = \tilde{x}A\tilde{y}$ и $xA\tilde{y} \leqslant \tilde{x}A\tilde{y} \leqslant \tilde{x}Ay$. Получаем, что $(\tilde{x}, \tilde{y})$ — седловая точка игры. □

Задача 9. *Покажите, что любая игра на конечном графе сводится к матричной.*

Задача 10. *Для игр на ациклических графах дайте определение стратегии и седловой точки.*

Задача 11. *Придумайте алгоритм решения матричной игры размера $2 \times n$ в смешанных стратегиях со сложностью а) $O(n^4)$; б)* $O(n \log n)$.*

Введение в STL

Д. Королёв

В этой статье будет дан обзор библиотеки STL с самого начального уровня, и до продвинутой, весьма полезной в ряде задач, функциональности.

Контейнеры STL

Проще всего начать знакомство с STL со стандартных типов для хранения данных — контейнеров.

Каждый раз, когда в программе возникает необходимость оперировать множеством элементов, в дело вступают контейнеры. Контейнер — это практическая реализация функциональности некоторой структуры данных. В языке C (не в C++) существовал только один встроенный тип контейнера: массив. Сам по себе массив имеет ряд недостатков: к примеру, размер динамически выделенного массива невозможно определить на этапе выполнения.

Однако основной причиной для более внимательного ознакомления с контейнерами STL является отнюдь не вышеперечисленные недостатки массива. Истинная причина кроется несколько глубже. Дело в том, что в реальном мире структура данных, информацию о которых необходимо хранить, далеко не всегда удачно представима в виде массива. В большинстве случаев требуется контейнер несколько иной функциональности.

К примеру, нам может потребоваться структура данных «множество строк», поддерживающая следующие функции:
— добавить строку к множеству;
— удалить строку из множества;
— определить, присутствует ли в рассматриваемом множестве данная строка;
— узнать количество различных строк в рассматриваемом множестве;
— просмотреть всю структуру данных, «пробежав» все присутствующие строки.

Конечно, легко запрограммировать тривиальную реализацию функциональность подобной структуры данных на базе обычного массива. Но такая реализация будет крайне неэффективной. Для достижения приемлемой производительности имеет смысл реализовать хэш-таблицу или сбалансированное дерево, но задумайтесь: разве реализация подобной структуры данных (хэш либо дерево) зависит от типа хранимых объектов? Если мы потом захотим использовать ту же структуру не

для строк, а, скажем, для точек на плоскости — какую часть кода придётся переписывать заново?

Реализация подобных структур данных на чистом C оставляла программисту два пути.

1) Жёсткое решение (Hard-Coded тип данных). При этом изменение типа данных приводило к необходимости внести большое число изменений в самых различных частях кода.

2) По возможности сделать обработчики структуры данных независимыми от используемого типа данных. Иными словами, использовать тип `void*` везде, где это возможно.

По какому бы пути реализации структуры данных в виде контейнера вы не пошли, скорее всего, никто другой ваш код понять, и тем более модифицировать, будет не в состоянии. В лучшем случае другие люди смогут им просто пользоваться. Именно для таких ситуаций существуют *стандарты* — чтобы программисты могли говорить друг с другом на одном и том же формальном языке.

Шаблоны (Templates) в C++ предоставляют замечательную возможность реализовать контейнер один раз, формализовать его внешние интерфейсы, дать асимптотические оценки времени выполнения каждой из операций, а после этого просто пользоваться подобным контейнером с любым типом данных. Можете быть уверены: разработчики стандарта C++ так и поступили. В первой части курса мы на практике познакомимся с основными концепциями, положенными в основу контейнеров C++.

Первые шаги

Для того, чтобы в программе можно было пользоваться функциональностью STL, следует подключить соответствующие заголовочные файлы. Большинство контейнеров описываются в заголовочном файле, имя которого совпадает с типом контейнера. Расширения у заголовочных файлов при этом отсутствует. Например, если вы планируете использовать в своей программе стандартный контейнер `stack`, то в программу следует добавить следующую директиву:

```
#include <stack>
```

Типы контейнеров, а также алгоритмы, функторы, да и вообще вся библиотека STL определены не в глобальном пространстве имён. Ввиду большого количества зарезервированных слов в STL, все они вынесены в отдельное пространство имён, получившее название `std`.

Существует несколько способов пользоваться ключевыми словами из пространства имён `std`. Универсального, стопроцентно пригодного на все случаи жизни решения искать не следует, у каждого метода есть свои преимущества и недостатки. Перечислим их.

1) Добавить строку

   ```
   using namespace std;
   ```

   непосредственно после всех подключаемых (`#include`) файлов.

   Это решение плохо тем, что в STL есть ряд достаточно простых зарезервированных слов (например, `swap`, `set`, `find`, `count`), и подобное подключение пространства имён `std` потребует от программиста аккуратности в выборе названий для своих классов и функций. Тем не менее, если вы поставили себе целью овладеть STL в совершенстве, автор рекомендует использовать подключение пространства имён `std` именно в таком формате.

2) Подключить желаемые модули STL отдельными директивами `using`:

   ```
   using std::stack;
   using std::find;
   ```

3) Использовать директиву `typedef` для каждого используемого типа:

   ```
   typedef std::vector<int> VI;
   ```

4) Писать префикс `std::` перед каждым STL типом данных либо алгоритмом.

Кроме того, не следует забывать про «особенность» парсеров компиляторов C++ в части описания шаблонов. Как вы помните, параметры шаблонов перечисляются в угловых скобках. При этом, две подряд идущих угловых скобки большинство трансляторов (совершенно справедливо) воспринимают как операцию «<<» или «>>». Поэтому, при наличии в коде вложенных STL конструкций следует не забывать оставлять между ними пробельный символ. Пример:

```
// Нехорошо: две закрывающих угловых скобки подряд
vector<vector<int>> WrongDefinition;

// OK
vector< vector<int> > RightDefinition;
```


Забегая вперёд, следует заметить, что один из наиболее удачных способов борьбы с подобными ошибками — использование `typedef`. Повторяя предыдущий пример:

```
typedef vector<int> vi;
typedef vector<vi> vvi;
```

Линейный контейнер `vector`

Простейшим контейнером STL является `vector`. Это всего лишь обычный (C-like) массив с расширенной функциональностью. Контейнер `vector` — единственный в STL обратно-совместимый с чистым C контейнер. Это означает, что `vector` по сути дела и является обычным динамическим массивом, но с рядом дополнительных функций.

Знакомство с `vector` начнём с примеров:

```
#include <vector>

using namespace std;

vector<int> v(10);

for(int i = 0; i < 10; i++) {
   v[i] = (i+1)*(i+1);
}

for(int i = 9; i > 0; i--) {
   v[i] -= v[i-1];
}
```

Далее в этом тексте директивы `#include` и `using` будут опускаться.

Мы видим, что при описании контейнера типа `vector` сразу после ключевого слова `vector` в угловых скобках — в качестве шаблонного параметра — указывается тип данных.

Когда в программе встречается описание следующего вида

```
vector<int> v;
```

на самом деле создаётся пустой вектор. Будьте внимательны с конструкциями вроде

```
vector<int> v[10];
```

В данном коде переменная `v` описывается как массив из десяти векторов целых чисел, каждый из которых изначально пуст. В большинстве случаев программист имеет в виду не это. Для размещения в памяти вектора ненулевого начального размера следует использовать конструктор, т. е. писать круглые скобки вместо квадратных.

Вектор всегда может сказать свой текущий размер:

```
int elements_count = v.size();
```

Здесь следует сделать два замечания. Первое заключается в том, что согласно стандарту STL все методы, возвращающие размер контейнера, имеют беззнаковый тип. Это чревато не только надоедливыми предупреждениями о приведении и сравнении знакового и беззнакового типов, но и более серьёзными проблемами. Поэтому, в экстремальном программировании, автор рекомендует использовать простой макрос, который возвращает тип контейнера в «обычном» знаковом целом типе.

```
#define sz(c) int((c).size())
```

По ходу примеров, приводимых по тексту далее, эта конструкция использоваться не будет, дабы не вводить читателя в заблуждение.

Второе замечание состоит в следующем: конструкция вида `c.size() == 0` является признаком дурного тона в STL. Если следует определить, не пуст ли контейнер, следует использовать специальный метод `empty()`, определённый для каждого контейнера.

```
// Подобного кода следует избегать.
bool is_nonempty_notgood = (Container.size() >= 0);

// OK
bool is_nonempty_ok = ! Container.empty();
```

Дело в том, что не любой контейнер может узнать свой размер за $O(1)$. А для того, чтобы узнать, к примеру, есть ли хотя бы один элемент в двусвязном списке, определённо не имеет смысла пробегаться по всем его элементам и подсчитывать их количество, не так ли?

Также с векторами широко используется функция `push_back(x)`. Метод `push_back(x)` добавляет элемент в конец вектора, расширяя его на один элемент. Поясним это на примере:

```
vector<int> v;
for(int i = 1; i < 1000000; i *= 2) {
    v.push_back(i);
}
int elements_count = v.size();
// Здесь, кстати, мы получим предупреждение
// о signed/unsigned mismatch
```


При использовании метода `push_back(x)` не следует беспокоиться за лишние операции выделения памяти. Конечно же, вектор не будет расширяться на один элемент при вызове `push_back(x)`. Если о чём и следует беспокоиться при использовании `push_back(x)` — так это об объёме используемой памяти. В большинстве реализаций STL использование `push_back(x)` может привести к тому, что занятый вектором объём памяти будет в два раза превосходить реальную потребность. О методах работы с памятью для векторов и о способах эффективного их использования мы ещё поговорим ниже.

Если хочется изменить размер вектора, используется метод `resize(n)`:

```
vector<int> v(20);

for(int i = 0; i < 20; i++) {
    v[i] = i+1;
}

v.resize(25);

for(int i = 20; i < 25; i++) {
    v[i] = i*2;
}
```

После вызова метода `resize(n)` вектор будет содержать ровно `n` элементов. Если параметр `n` меньше, чем размер вектора до вызова `resize(n)`, то вектор уменьшится и «лишние» элементы будут удалены. Если же `n` больше, чем размер вектора, то вектор увеличит свой размер и заполнит появившиеся элементы нулями. Если хранимым типом данных является более сложный объект, нежели стандартный тип данных новые элементы будут инициализированы конструктором по умолчанию.

Важно помнить, что если использовать `push_back(x)` после `resize(n)`, то элементы будут добавлены после области, выделенной `resize(n)`, а не в неё. В нижеприведённом примере, после выполнения данного фрагмента кода, размер вектора будет равен 30, а не 25.

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
    v.push_back(i*2); // Запись производится в элементы
```

```
                    // [25..30), а не [20..25) !
}
```

Для очистки вектора (как и любого другого контейнера STL) предназначен метод `clear()`. После вызова `clear()` контейнер окажется пустым, т. е. будет содержать ноль элементов. Будьте аккуратны: `clear()` не обнуляет все элементы контейнера, но освобождает весь контейнер целиком.

Существует много способов инициализировать вектор при создании.

Вектор можно создать как копию другого вектора:

```
vector<int> v1;
...
vector<int> v2 = v1;
vector<int> v3(v1);
```

Если вы хорошо знакомы с C++, то понимаете, что `v2` и `v3` инициализируются абсолютно идентично.

Как мы уже говорили, можно создать вектор желаемого размера:

```
vector<int> Data(1000);
```

В данном примере `Data` будет содержать тысячу нулей. Не забывайте об использовании круглых, а не квадратных скобок.

Для того, чтобы проинициализировать все элементы вектора при создании значениями по умолчанию, следует передать конструктору второй параметр:

```
vector<string> names(20, "Unknown");
```

Как вы, конечно, помните, вектор может содержать любой тип данных, а не только `int`. `string` — это стандартный контейнер для строки в STL, о нём мы поговорим чуть позже.

Также очень важно бывает создать многомерный массив. Сделать это с использованием векторов можно при помощи следующей конструкции:

```
vector< vector<int> > Matrix;
// Помните о лишних пробелах между угловыми скобками!
```

Сейчас вам должно быть понятно, как указать размер матрицы при создании:

```
int N, M;
...
vector< vector<int> > Matrix(N, vector<int>(M, -1));
```


Вышеприведённая конструкция создаёт матрицу с `N` строками и `M` столбцами. Изначально матрица будет заполнена значениями -1.

При использовании векторов следует помнить об одной очень важной особенности работы с памятью в STL. Основное правило здесь можно сформулировать таким образом: контейнеры STL всегда копируются при любых попытках передать их в качестве параметра.

Таким образом, если вы передаёте вектор из миллиона элементов функции, описанной следующим образом:

```
void some_function(vector<int> v) {
   // Старайтесь никогда так не делать
   ...
}
```

то весь миллион элементов будет скопирован в другой, временный, вектор, который будет освобождён при выходе их функции `some_function`. Если эта функция вызывается в цикле, о производительности программы можно забыть сразу.

Если вы не хотите, чтобы контейнер создавал клон себя каждый раз при вызове функции, используйте передачу параметра по ссылке. Хорошим тоном считается использование при этом модификатора `const`, если функция не намерена изменять содержимое контейнера.

```
void some_function(const vector<int>& v) {
   // OK
   ...
}
```

Если содержимое контейнера может измениться по ходу работы функции, то модификатор `const` писать не следует:

```
int modify_vector(vector<int>& v) {
   // Так держать
   v[0]++;
}
```

Правило копирования данных применимо ко всем контейнерам STL без исключения.

Также следует отметить, что если объекты, хранимые в контейнере, имеют некоторый физический смысл и/или связаны с другими объектами, следует задуматься об использовании не контейнера из объектов, а контейнера из указателей на объекты. Хорошим критерием здесь может служить следующее правило: всегда ли можно создать новый экземпляр моего класса при помощи конструктора по умолчанию? В случае любых сомнений при ответе на этот вопрос не следует «разбрасываться» объектами. Следует пользоваться контейнерами из указателей. Хорошим решением при этом будет использование контейнеров, содержащих умные указатели (smart pointers), но и поддержание отдельного массива (вектора) созданных объектов — тоже достаточно грамотное решение.

Часто используется функция `vector::reserve(n)`. Как уже было сказано, вектор не выделяет по одному новому элементу в памяти на каждый вызов `push_back()`. Вместо этого, при вызове `push_back()`, вектор выделяет больше памяти, чем реально требуется. В большинстве реализаций при необходимости выделить лишнюю память, `vector` увеличивает объём выделенной памяти в два раза. На практике это бывает не очень удобно. Наиболее простой способ обойти эту проблему заключается в использовании метода `reserve`. Вызов метода `vector::reserve(size_t n)` выделяет дополнительную память в будущее пользование `vector`. Параметр `n` имеет следующий смысл: вектор должен выделить столько памяти, чтобы вплоть до размера в `n` элементов дополнительных операций выделения памяти не потребовалось.

Рассмотрим следующий пример. Пусть имеется `vector` из 1 000 элементов, и пусть объём выделенной им памяти составляет 1 024 элемента. Мы собираемся добавить в него 50 элементов при помощи метода `push_back()`. Если вектор расширяется в два раза, его размер в памяти по завершении этой операции будет составлять 2 048 элементов, т. е. почти в два раза больше, чем это реально необходимо. Однако, если перед серией вызовов метода `v.push_back(x)` добавить вызов

```
v.reserve(1050);
```

то память будет использоваться эффективно. Если вы активно используете `push_back()`, то `reserve()` — ваш друг.

Итак, мы умеем создавать вектора, добавлять в них данные и работать с этими данными. Хочется двигаться дальше: например, научиться манипулировать блоками данных внутри самого вектора. Однако, прежде чем мы перейдём к рассмотрению соответствующего инструментария STL, познакомимся с другими аспектами программирования в STL.

Пары объектов (`pair`)

В качестве введения к данной лекции поговорим о «парах» объектов в STL — `std::pair`.

Пара — это просто шаблонная структура, которая содержит два поля, возможно, различных типов. Поля имеют названия `first` и `second`. В максимально краткой форме прототип пары может выглядеть следующим образом:


```
template<typename T1, typename T2> struct pair {
   T1 first;
   T2 second;
};
```

К примеру, `pair<int,int>` есть пара двух целых чисел. Более сложный пример: `pair<string, pair<int,int> >` — строка плюс два целых числа. Использовать подобную пару можно, например, так:

```
pair<string, pair<int,int> > P;
string s = P.first; // Строка
int x = P.second.first; // Первое целое
int y = P.second.second; // Второе целое
```

Основной причиной к использованию `pair` является то, что объекты `pair` можно сравнивать. Поэтому, при всей кажущейся простоте, пары активно используются как внутри библиотеки STL, так и программистами в своих целях.

Сравнение пар предоставляет широкие возможности для экстремального программирования. Массив пар можно упорядочить, при этом упорядочивание будет производиться по полям в порядке описания пар слева направо.

Например, необходимо упорядочить целочисленные точки на плоскости по полярному углу. Одним из простых решений является поместить все точки в структуру вида

```
vector< pair<double, pair<int,int> >
```

где `double` — полярный угол точки, а `pair<int,int>` — её координаты. После этого один вызов стандартной функции сортировки приведёт к тому, точки будут упорядочены по полярному углу.

Также пары активно используются в ассоциативных контейнерах, но об этом мы поговорим позднее.

Итераторы

Пришло время поговорить об итераторах. В максимально общем смысле, итераторы — это универсальный способ доступа к данным в STL. Однако автору представляется совершенно необходимым, чтобы программист, использующий STL, хорошо понимал необходимость в итераторах.

Рассмотрим следующую задачу. Дан массив `A` длины `N`. Необходимо изменить порядок следования элементов в нём на обратный («развернуть массив на месте»).

Начнём издалека, с решения на чистом C.

```
void reverse_array_simple(int *A, int N) {
   // индексы элементов, которые мы
   // на данном шаге меняем местами
   int first = 0, last = N-1;
   while(first < last) { // пока есть что переставлять
      // swap(a,b) --- стандартная функция STL
      swap(A[first], A[last]);
      first++; // смещаем левый индекс вправо
      last--; // а правый влево
   }
}
```

Пока ничего сложного в этом коде нет. Его легко переписать, заменив индексы на указатели:

```
void reverse_array(int *A, int N) {
   int *first = A, *last = A+N-1;
   while(first < last) {
      swap(*first, *last);
      first++;
      last--;
   }
}
```

Рассмотрим основной цикл данной функции. Какие операции над указателями он выполняет? Всего лишь следующие:

* сравнение указателей (`first < last`),

* разыменование указателей (`*first, *last`),

* инкремент и декремент указателя (`first++, last--`)

Теперь представим, что, решив эту задачу, нам необходимо развернуть на месте двусвязный список или его часть.

Первый вариант функции, в котором использовались индексы в массиве, работать, конечно, не будет. Если даже написать функцию обращения к элементам списка по индексу, о производительности и/или экономии памяти можно забыть.

Обратим теперь внимание на то, что второй вариант функции, который использует указатели, может работать с любыми объектами, которые обеспечивают функциональность указателя. А именно: сравнение, разыменование, инкремент/декремент. В языке C уже есть удобный, привычный многим и проверенный временем синтаксис для непрямого обращения к данным: нам лишь осталось подставить вместо указателя объект, который умеет делать то же самое.

Именно этот подход используется в STL. Конечно, для контейнера типа `vector` итератор — это почти то же самое, что и указатель. Но для более сложных структур данных, например, для красно-чёрных деревьев, универсальный интерфейс просто необходим.

Итак, какую функциональность должен обеспечивать итератор? Примерно ту же, что и обычный указатель:

* разыменование (`int x = *it`);

* инкремент/декремент (`it1++, it2--`);

* сравнение (об этом речь пойдёт позже; пока просто скажем, что это операции `==` и `<`)

* добавление константы (`it += 20` — сдвинуться на 20 элементов вперёд);

* расстояние между итераторами (`int dist = it2-it1`);

Язык C++ предоставляет необходимые средства для создания произвольного объекта, который сможет вести себя именно таким образом.

Итак, какую функциональность должен обеспечивать итератор для двусвязного списка, чтобы наша функция `reverse_array` смогла функционировать? Более узкую, чем указатель, а именно: разыменование, инкремент/декремент, сравнение.

Следует привести более подробное пояснение. Конечно, не для каждого типа контейнера в итераторе возможно эффективно реализовать ту или иную функцию. Строго говоря, базисными являются для итератора только следующие операции:

* разыменование (`*it`);

* инкремент (`++`);

* сравнение (`==`).

В нашем примере с двусвязным списком, мы также предполагаем, что для его итератора определена операция `--`. Конечно, о добавлении константы, о сравнении двух итераторов на «больше/меньше» и, тем более, о вычислении разности между итераторами в двусвязном списке не может быть и речи.

В отличие от обычных указателей, итераторы могут также нести много другой полезной нагрузки. В качестве основных примеров, не вдаваясь в подробности, следует отметить проверку выхода за границы массива (Range Checking) и статистику использования контейнера (Profiling).

Основное преимущество итераторов, бесспорно, заключается в том, что они помогают систематизировать код и повышают коэффициент его повторного использования. Один раз реализовав некоторый алгоритм, использующий итераторы, его можно использовать с любым типом контейнера. Можете быть уверены: разработчики STL так и сделали, поэтому большую часть алгоритмов писать не придётся. С другой стороны, если вам необходимо реализовать свой тип контейнера, реализуйте ещё и механизм итераторов — и широкий спектр алгоритмов, как стандартных, так и авторских, будет сразу доступен.

На самом деле, не всем итераторам необходимо поддерживать всю функциональность. В STL пошли по следующему пути: итератор поддерживает те операции, которые он может выполнить за $O(1)$, т. е. независимо от размеров контейнера и параметров. Это означает, к примеру, что для итераторов по двусвязному списку, операции сравнения (<, >) и арифметические операции (`it += offset` или `shift = it2-it1`) не применимы. Действительно, время работы этих операций зависит как от размеров контейнера, так и от параметров. С другой стороны, операции сравнения (`it1 == it2`, `it1 != it2`), и инкремента/декремента (`it1++`, `it2--`), конечно, допустимы.

В свете вышесказанного, итераторы подразделяются по типам:
— random access iterator: итератор произвольного доступа; умеет всё, что умеет делать указатель, и даже немного больше
— normal iterator: то же, что итератор произвольного доступа, но не поддерживает арифметические операции <, >, +=, -=, -
— forward iterator: то же, что normal iterator, но не поддерживает декремент. Пример: односвязный список.

Ввиду того, что итераторы списка нельзя сравнивать при помощи `operator <`, код функции обращения списка следует модифицировать:

```
template<typename T> void reverse_list(T *first, T *last) {
   if(first != last) {
   // точнее, параноик написал бы if(!(first == last))
      while(true) {
         swap(*first, *last);
         first++;
         if(first == last) {
            break;
         }
         last--;
         if(first == last) {
            break;
         }
      }
   }
}
```


Пришло время вернуться к STL. Каждый контейнер в STL имеет собственный тип итератора, и даже часто не один. Программисту об этом знать не обязательно. Программисту лишь надо помнить, что для любого контейнера определены методы `begin()` и `end()`, которые возвращают итераторы начала и конца, соответственно.

Однако, в отличие от вышеприведённого примера, `end()` возвращает итератор, указывающий не на последний элемент контейнера, а на непосредственно следующий за ним элемент. Это часто бывает удобно. Например, для любого контейнера `c` разность `(c.end() - c.begin())` всегда равна `c.size()`, если, конечно, итераторы данного типа контейнера поддерживают арифметические операции. А `(c.begin() == c.end())` тождественно равно `c.empty()` — для любых типов контейнеров.

Таким образом, STL-совместимая версия функции `reverse_list` приведена ниже.

```
template<typename T> void
       reverse_list_stl_compliant(T *begin, T *end) {
   // Сначала мы должны уменьшить end,
   // но только для непустого диапозона
   if(begin != end)
   {
      end--;
      if(begin != end) {
         while(true) {
            swap(*begin, *end);
            begin++;
            if(begin == end) {
               break;
            }
            end--;
            if(begin == end) {
               break;
            }
         }
      }
   }
```

```
}
```

Теперь эта функция полностью соответствует функции `std::reverse(iterator begin, iterator end)`, определённой в модуле `algorithm`. В качестве упражнения хотелось бы порекомендовать читателю прочитать и разобрать код функции `std::reverse` какой-либо из стандартных реализаций STL — например, SGI или Boost.

Каждый STL контейнер имеет так называемый интервальный конструктор: конструктор, который в качестве параметров принимает два итератора, который задают интервал объектов, тип которых приводим к типу контейнера. Это будет продемонстрировано в следующих примерах.

```
vector<int> v;
...
vector<int> v2(v);

// интервальный конструктор, v3 == v2
vector<int> v3(v.begin(), v.end());

int data[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 };
vector<int> primes(data,data+(sizeof(data)/sizeof(data[0])));
```

Последняя строка инициализирует `vector<int> primes` содержимым массива `data` при помощи интервального конструктора.

Более сложные примеры:

```
vector<int> v;
...
vector<int> v2(v.begin(), v.begin() + (v.size()/2));
```

Создаваемый вектор `v2` будет содержать первую половину вектора `v`.

Особо следует выделить тот факт, что в качестве итератора алгоритмам STL можно передавать объекты произвольной природы — необходимо лишь, чтобы они поддерживали соответствующий функционал. Разберём на примере функции `reverse`:

```
int data[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
reverse(data+2, data+6);
// интервал { 5, 7, 9, 11 } переходит в { 11, 9, 7, 5 };
```

Кроме того, каждый контейнер имеет также так называемые обратные итераторы — итераторы, служащие для обхода контейнера в обратном порядке. Обратные итераторы возвращаются методами `rbegin()`/`rend()`:


```
vector<int> v;
vector<int> v2(v.rbegin()+(v.size()/2), v.rend());
```

Вектор `v2` содержит первую половину `v`, в порядке от середины к началу.

Чтобы создать объект типа итератор, следует указать его тип. Тип итератора получается приписыванием к типу контейнера `::iterator`, `::const_iterator`, `::reverse_iterator` или `::const_reverse_iterator`. Смысл этих типов уже должен быть понятен слушателю.

Таким образом, содержимое вектора можно просмотреть следующим образом:

```
vector<int> v;

for(vector<int>::iterator it = v.begin(); it!=v.end(); it++) {
  *it = (*it) * (*it); // возводим каждый элемент в квадрат
}
```

В условии цикла строго рекомендуется использовать `operator !=`, а не `operator <`.

Алгоритмы STL

Теперь мы можем осуществить краткое введение в стандартные алгоритмы STL. Большая часть алгоритмов STL построена по единому принципу. Алгоритм получает на вход пару итераторов — интервал. Если алгоритм осуществлял поиск элемента, то будет возвращен либо итератор, указывающий на соответствующий элемент, либо конец интервала. Конец интервала уже не указывает ни на какой элемент, что очень удобно.

`find(iterator begin, iterator end, some_typy value)` осуществляет поиск элемента в интервале. `find(...)` возвращает итератор, указывающий на первый найденный элемент, либо `end`, если подходящих элементов найдено не было.

```
vector<int> v;
for(int i = 1; i <= 100; i++) {
   v.push_back(i*i);
}

if(find(v.begin(), v.end(), 49) != v.end()) {
   // число 49 найдено в списке квадратов натуральных чисел,
   // не превосходящих 100
}
```

```
else {
   // число 49 не найдено
}
```

Чтобы получить значение, итератор необходимо разыменовать. Это не актуально для алгоритма `find(...)`, но будет активно использоваться в дальнейшем.

Если контейнер поддерживает итераторы произвольного доступа, то можно найти индекс найденного элемента. Для это из значения, которое вернул алгоритм, нужно вычесть начало интервала:

```
int i = find(v.begin(), v.end(), 49) - v.begin();
if(i < v.size()) {
   // 49 найдено
   assert(v[i] == 49);
}
```

Напомним, что для использования стандартных алгоритмов STL следует подключать модуль `algorithm`.

Алгоритмы `min_element` и `max_element` в пояснениях не нуждаются:

```
int data[5] = { 1, 5, 2, 4, 3 };
vector<int> X(data, data+5);

// Значение
int v1 = *max_element(X.begin(), X.end());

// Индекс
int i1 = min_element(X.begin(), X.end()) - X.begin());

int v2 = *max_element(data, data+5);
// Либо же просто в массиве
int i3 = min_element(data, data+5) - data;
```

В рамках введения в экстремальное программирование следует присмотреться к следующему макросу:

```
#define all(c) c.begin(),c.end()
```

(скобки вокруг правой части здесь ставить ни в коем случае нельзя!)

Также часто используется алгоритм `sort(iterator begin, iterator end)`.

```
vector<int> X;
...
```



```
sort(X.begin(), X.end()); // Стандартный вызов
sort(all(X)); // Почувствуйте разницу
sort(X.rbegin(), X.rend()); // Сортировка в обратном порядке
```

Алгоритм сортировки активно использует арифметику на указателях, он может работать только на итераторах произвольного доступа. Отсортировать двусвязный список — нетривиальная алгоритмическая задача. На практике в такой ситуации проще сделать из него `vector`, а затем осуществить обратное преобразование. В этом помогут интервальные конструкторы:

```
deque<int> q;
...
// Упорядочим элементы в q по неубыванию
vector<int> v(all(q));
// то же, что vector<int> v(q.begin(), q.end());
sort(all(v));
q.assign(all(v));
// то же, что q = deque<int>(all(v));
```

Проблемы с компиляцией программ под STL и их красивое решение в GNU C++

Следует сделать важное замечание о компиляции программ в STL. Дело в том, что STL распространяется в исходных кодах (это существенно влияет на производительность результирующего кода), поэтому зачастую строка с ошибкой будет указывать не на ваш код, а на внутренности STL. При этом строка с описанием ошибки будет занимать несколько сотен символов, что не сразу проливает свет на истинную причину её возникновения. Разберём только один простой пример.

Пусть мы передаём `vector<int>` как const reference параметр (как это и следует делать) в некую функцию. А в этой функции мы работаем с этим массивом посредством итераторов:

```
void f(const vector<int>& v) {
    for(
    vector<int>::iterator it = v.begin();
        // ну что же здесь не так
        ...
        ...
}
```

В этом коротком участке кода есть ошибка. Вы можете её обнаружить?

Дело в том, что из немодифицируемого (const) объекта мы пытаемся получить модифицируемый (non-const) итератор при помощи функции begin(). Подобное преобразование является недопустимым. Корректный код выглядит следующим образом:

```
void f(const vector<int>& v) {
    int r = 0;
    // используется const_iterator
    for(vector<int>::const_iterator it = v.begin();
                              it != v.end(); it++) {
        r += (*it) * (*it);
    }
    return r;
}
```

В свете всего вышесказанного, имеет смысл знать про одну очень интересную функцию компилятора GNU C++. Заметим, что данная функция не входит в стандарт C++, поэтому в большинстве других компиляторов её нет.

Итак, «оператор» typeof(x). На этапе компиляции он заменяется на тип выражения в скобках. Пример:

```
typeof(a+b) x = (a+b);
```

Переменная x будет иметь тип, соответствующий типу выражения a+b. Напомним, что typeof(c.size()) — unsigned int для всех контейнеров STL. Но наибольшую пользу можно извлечь из typeof в следующем контексте:

```
#define tr(container, iterator) \
  for(typeof(container.begin()) iterator=container.begin(); \
  it != container.end(); it++)
```

Данный макрос — сокращение от traverse — будет работать даже для самого сложного типа контейнера, независимо от того, каким образом этот контейнер к моменту использования определён. Для const-объектов он породит const_iterator'ы:

```
void f(const vector<int>& v) {
   int r = 0;
   tr(v, it) {
      r += (*it)*(*it);
   }
   return r;
}
```


Подобные ухищрения не столь важны при работе с векторами, но они совершенно незаменимы при работа с более сложными контейнерами. Это чувствуется особенно хорошо, когда код принимает примерно следующий вид:

```
void operate(const map< set< pair<int,int> >,
  vector< pair< double, pair<int,int> > > > &object) {
   for( ... ) // MAMMA MIA!
}
```

Хотя, например, такая операция, как подсчитать сумму всех double из правой части map с использованием макроса tr выполняется сравнительно просто:

```
void operate(const map< set< pair<int,int> >,
  vector< pair< double, pair<int,int> > > > &c) {
   double result = 0;
      tr(c, it) {
         tr(it->second, it2) {
            result += it2->first;
         }
      }
   return result;
}
```

Манипуляции с данными в векторах

Добавление данных в vector выполняется при помощи метода insert. У insert есть несколько форм.

```
vector<int> v;
...
// Добавить элемент со значением 42
// непосредственно после первого элемента вектора
v.insert(v.begin() + 1, 42);
```

При выполнении такой операции:

— вектор расширится на один элемент;

— все элементы вектора, начиная со второго (индекс 1) по последний, сместятся вправо, освобождая место для нового элемента;;

— новый элемент со значением 42 займёт своё место.

Если необходимо добавить много элементов в середину вектора, логично выполнить один сдвиг сразу на несколько элементов. Для этого используется интервальная форма insert: insert(iterator1 where, iterator2 what_begin, iterator2 what_end). При этом типы итераторов iterator1 и iterator2 могут не совпадать — insert может, к примеру, вставить в vector содержимое контейнера set.

```
vector<int> v;
vector<int> v2;
...
// вставить содержимое v2 в обратном порядке в середину v
v.insert(v.begin() + (v.size()/2), v2.rbegin(), v2.rend());
```

Для удаления элементов из vector используется метод erase. У erase также есть две формы:

```
erase(iterator what);
erase(iterator from, iterator to);
```

Технология методов insert/erase так или иначе используется во всех типах контейнеров STL.

Контейнер для строк: string

В STL существует специальный контейнер для работы со строками: string. Этот контейнер не очень сильно отличается от vector<char>. Различия, в основном, сосредоточены в функциях для манипулирования строками и в политике работы с памятью. Для того, чтобы узнать длину строки, принято использовать string::length(), а не vector::size().

У строк есть метод substr для быстрого получения подстроки в виде отдельной строки. Этот метод принимает в качестве параметров только индексы, и никаких итераторов:

```
string s = "hello";
string
   s1 = s.substr(0, 3), // "hel"
   s2 = s.substr(1, 3), // "ell"
   s3 = s.substr(0, s.length()-1), "hell"
   s4 = s.substr(1); // "ello"
```

Как и в случае с vector::size(), string::length() возвращает беззнаковый тип. Поэтому будьте аккуратны с конструкциями вроде (s.length()-1): для пустой строки s результат, скорее всего, не оправдает ваших ожиданий.

Также бывает удобным использовать метод string::find(). Он возвращает индекс начала первого вхождения символа или строки.

Кроме string::find, бывает удобно использовать метод string::find_first_of. Если передать в качестве параметра

find_first_of один символ, вызов find_first_of будет полностью аналогичен вызову find. Однако, если передать в качестве параметра find_first_of строку, метод вернёт индекс первого вхождения любого символа из данной строки:

```
string s = "Hello, World!";
int index = s.find_first_of(' '); // поиск первого пробела
int index2 = s.find_first_of(", !"); // поиск первого пробела,
                                     //знака восклицания или запятой
```

Особенно find_first_of удобно использовать для того, чтобы разделить строку на множество строк. Это актуально, потому как встроенных функций для string splitting в STL, к сожалению, нет.

```
// разбить строку на вектор строк,
// используя пробелы и запятые как разделители
vector<string> v;
size_t i;
while((i = s.find_first_of(", ")) != string::npos) {
    // кусок строки до первого разделителя
    v.push_back(s.substr(0, i));
    // вырезать вместе с разделителем
    s = s.substr(i+1);
}
v.push_back(s); // дописать остаток строки
```

Строки можно складывать с помощью оператора +:

```
string s = "hello";
string
    s1 = s.substr(0, 3), // "hel"
    s2 = s + s1; // " helloell"
```

Строки можно складывать с помощью оператора + и с символами, но нужно быть аккуратными с константными строками. Запись "jdklasj" — это переменная типа const char*. В большинстве случаев, но не всегда, объекты типа const char* по мере необходимости приводятся к типу string напрямую. Этого не происходит лишь в одном случае: складывать две константных строки нельзя. Для этого нужно привести первую из них к типу string напрямую. Впрочем, если обе строки — действительно константные, можно просто опустить знак +.

У string нет конструктора от одного char. Для того, чтобы создать строку, состоящую из одного символа, можно либо добавить этот символ к пустой строке (string() + 'A'), либо использовать инициализацию в стиле vector: string(1, 'B'). Соответственно, таким же образом создаются строки, состоящие из фиксированного числа одинаковых символов.

В общем случае лучше работать с типом string, чем с const char*. Иначе может так случиться, что компилятор подумает совсем не то, что вы имели в виду:

```
string s = "word", s2;

s = 'S' + s + '!'; // "Sword!"

s2 = 'A' + " and " + 'B';
//таких конструкций следует избегать. фактически здесь мы
//к числу типа char прибавили указатель, а потом еще раз число

s2 = 'X' + string(" and ") + 'Y';
// "X and Y" --- это корректно

string s3wrong = "p " + " and q "; // не компилируется

string s3 = "p" " and q";  // так можно,
//но всегда проще написать string("p") +  ' ' + "and q";
```

Также строки можно сравнивать с помощью операторов сравнения. Таким образом, vector<string> можно упорядочивать стандартными методами. Строки сравниваются в лексикографическом порядке.

Множество элементов: set

Мы подошли к двум наиболее интересным с точки зрения изучения STL контейнерам: set и map. С которым из них стоит познакомиться в первую очередь — вопрос, не имеющий однозначного ответа. Мнение автора заключается в том, что при академическом подходе к изучению STL, в первую очередь следует познакомиться с set, как с более простым контейнером из рассматриваемой пары. Всё, что можно сделать с set, можно сделать и с map, обратное же утверждение не всегда истинно. С алгоритмической точки зрения map является логическим продолжением set, в то время как многие программисты-практики зачастую смутно понимают назначение контейнера set, и всегда используют map , что менее элегантно и часто более сложно для понимания сторонними людьми.

Контейнер set, как уже было упомянуто, содержит множество элементов. Строго говоря, set обеспечивает следующую функциональность:

— добавить элемент в рассматриваемое множество, при этом исключая возможность появления дублей;
— удалить элемент из множества;
— узнать количество (различных) элементов в контейнере;
— проверить, присутствует ли в контейнере некоторый элемент.

Об алгоритмической эффективности контейнера set мы поговорим позже, вначале познакомимся с его интерфейсом.

```
set<int> s;

for(int i = 1; i <= 100; i++) {
   s.insert(i); // добавим сто первых натуральных чисел
}

s.insert(42); // ничего не произойдёт ---
   // элемент 42 уже присутствует в множестве

for(int i = 2; i <= 100; i += 2) {
   s.remove(i); // удалим чётные числа
}

// set::size() имеет тип unsigned int
int N = int(s.size()); // N будет равно 50
```

У set нет метода push_back(). Это неудивительно: ведь такого понятия, как порядок элементов или индекс элемента, в set не существует, поэтому слово «back» здесь никак не применимо.

А раз уж у set нет понятия «индекс элемента», единственный способ просмотреть данные, содержащиеся в set, заключается в использовании итераторов:

```
set<int> S;
...
// вычисление суммы элементов множества S
int r = 0;
for(set<int>::const_iterator it = S.begin();
                             it != S.end(); it++) {
   r += (*it);
}
```

Если вы пользуетесь GNU C++, то Traversing Macros будет весьма кстати. Показательный пример:

```
set< pair<string, pair< int, vector<int> > > SS;
```

```
...
int total = 0;
tr(SS, it) {
   total += it->second.first;
}
```

Обратите внимание на синтаксис `it->second.first`. Ввиду того, что `it` является итератором, перед использованием его необходимо разыменовать. «Верным» синтаксисом было бы `(*it).second.first`. Однако, в C++ есть негласное правило, что если при описании некоторого объекта есть возможность обеспечить тождественное равенство конструкций `(*it).` и `it->`, то это следует сделать, дабы не вводить пользователей в заблуждение. Разработчики STL, конечно, позаботились об этом в случае с итераторами.

Основным преимуществом `set` перед `vector` является, несомненно, быстродействие. В основном это быстродействие проявляется при выполнении операции поиска. (При добавлении операция поиска также неявно присутствует, потому как дубли в `set` не допускаются). Однако, с операцией поиска в `set`/`map` есть существенный нюанс.

Нюанс заключается в том, что вместо глобального алгоритма `std::find(...)` следует использовать метод `set set::find(...)`.

Это не означает, что `std::find(...)` не будет работать с `set`. Дело в том, что `std::find(...)` ничего не знает о типе контейнера, с которым он работает. Принцип работы `std::find(...)` крайне прост: он просматривает все элементы до тех пор, пока либо не будет найден искомый элемент, либо не будет достигнут конец интервала. Основное преимущество `set` перед `vector` заключается в использовании нелинейной структуры данных, что существенно снижает алгоритмическую сложность операции поиска; использование же `std::find(...)` анулирует все старания разработчиков STL.

Метод `set::find(...)` имеет всего один аргумент. Возращаемое им значение либо указывает на найденный элемент, либо равно итератору `end()` для данного экземпляра контейнера.

```
set<int> s;
...
if(s.find(42) != s.end()) {
// 42 присутствует
}
else {
// 42 не присутствует
}
```


Кроме `find(...)` существует также операция `count(...)`, которую следует вызывать как метод `set::count(x)`, а не как алгоритм `std::count(begin, end, x)`. Ясно, что `set::count(x)` может вернуть только 0 или 1. Некоторые программисты считают, что вышеприведённый код лучше выглядит, если использовать `count(x)` вместо `find(x)`:

```
if(s.count(42) != 0) {
   ...
}
```

Или даже

```
if(s.count(42)) {
   ...
}
```

Мнение автора заключается в том, что подобный код вводит читателя в заблуждение: сам смысл операции `count()` несовместим со случаями, когда элемент либо присутствует, либо нет. Если же вам представляется слишком длинным каждый раз писать `"[некоторая форма find]"!= container.end()`, сделайте следующие макросы:

```
#define present_member(container, element) \
         (find(all(container),element) != container.end())
#define present_global(container, element) \
         (container.find(element) != container.end())
```

Здесь `all(c)` означает `c.begin(),c.end()`

Более того, в соответствии с положением стандарта, которое называется «конкретизация шаблонов», можно написать следующий код:

```
template<typename T, typename T2> bool present(const T& c,
                                               const T2& obj) {
    return find(c.begin(), c.end(),
              (T::element_type)(obj)) != c.end();
}

template<typename T, typename T2> bool
                    present(const set<T>& c, const T2& obj) {
    return c.find((T::element_type)(obj)) != c.end();
}
```

При работе с контейнером типа `set` `present(container, element)` вызовет метод `set::find(element)`, в других случаях — `std::find(container.begin(), container.end(), element)`.

Для удаления элемента из `set` необходимо вызвать метод `erase(...)`, передав ему один элемент — элемент, который следует удалить, либо итератор, указывающий на удаляемый элемент.

```
set<int> s;
...
s.insert(54);
...
s.erase(29);
s.erase(s.find(57));
```

Как и полагается `erase(...)`, `set::erase(...)` имеет интервальную форму.

```
set<int> s;
...
set<int>::iterator it1, it2;
it1 = s.find(10);
it2 = s.find(100);
// Будет работать, если как 10, так и 100 присутствуют в множестве
if(...) {
    s.erase(it1, it2); // при таком вызове будут удалены
    // все элементы от 10 до 100 не включительно
}
else {
    // сдвинем it2 на один элемент вперёд
    // set::iterator является normal iterator
    // операция += не определена для итераторов set'a,
    //но ++ и -- допускаются
    it2++;
    s.erase(it1, it2); // а при таком --- от 10 до 100 включительно
    // приведённый код будет работать, даже если 100 был
    // последним элементом, входящим в set
}
```

Также, как и полагается контейнерам STL, у `set` есть интервальный конструктор:

```
int data[5] = { 5, 1, 4, 2, 3 };
set<int> S(data, data+5);
```

Кстати, данная функция `set` предоставляет эффективную возможность избавиться от дубликатов в `vector`:

```
vector<int> v;
```



```
...
set<int> s(all(v));
vector<int> v2(all(s));
```

Теперь `v2` содержит те же элементы, что и `v`, но без дубликатов. Приятной особенностью также является тот факт, что элементы `v2` упорядочены по возрастанию, но об этом мы поговорим позже.

В `set` можно хранить элементы любого типа, которые можно упорядочить. Об этом мы тоже поговорим позже.

Ассоциативный контейнер `map`

Теперь мы можем перейти от `set` к `map`. «Введение в `map` для чайников» могло бы выглядеть следующим образом:

```
map<string, int> M;
M["One"] = 1;
M["Two"] = 2;
M["Many"] = 7;

int x = M["One"] + M["Two"];

if(M.find("Five") != M.end()) {
   M.erase("Five");
}
```

Очень просто, не так ли?

На самом деле, `map` очень похож на `set`, за исключением того, что вместо элементов `map` хранит пары элементов `<ключ, значение>`. Поиск при этом осуществляется только по ключу. Крайне приятно наличие оператора обращения по «индексу» (`operator []`).

Для того, чтобы просмотреть содержимое `map`, необходимо использовать итераторы. Удобнее всего это делать при помощи нашего макроса `tr`. Следует помнить, что итератор указывает не на элемент `key`, а на `pair<key, value>`:

```
map<string, int> M;
...
M["one"] = 1;
M["two"] = 2;
M["google"] = 1e100;
...
// найдём сумму всех значений --- т.е. всех правых частей
// пар <string, int>
int r = 0;
```

```
tr(M, it) {
   r += it->second;
   // (*it).first == [string], (*it).second == [int]
}
```

Как и в случае с `set`, элементы `map` хранятся упорядоченными по ключу. Поэтому не следует при работе с `map::iterator` модифицировать `it->first`: если вы нарушите правила упорядочивания элементов в `map`, за последствия никто отвечать не возьмётся.

В остальном контейнер `map` по интерфейсу практически эквивалентен контейнеру `set`.

Также важно помнить, что `operator []` при обращении к несуществующему элементу в `map` создаст его. Новый элемент при этом будет инициализирован нулём (либо конструктором по умолчанию, если это не тривиальный тип данных). Данная особенность `map` может быть удобной, потому как выполнять операции с элементами можно не задумываясь об их присутствии в `map`. Существенным моментом является то, что `operator []` не является константным (то есть может изменить объект, для которого вызван), поэтому им нельзя пользоваться, если `map` передан как const reference. Используйте `map::find(element)`:

```
void f(const map<string, int>& M) {
   if(M["the meaning"] == 42) {
     // Так нельзя! M передан как const reference
   }
   if(M.find("the meaning") != M.end() &&
                  M.find("the meaning")->second == 42) {
      // А можно именно так
      cout << "Don't Panic!" << endl;
   }
}
```

Литература и ссылки

1. *Scott Meyers*, Effective STL. Addison Wesley Professional, 2001.
2. http://www.sgi.com
3. http://msdn.microsoft.com

Теория графов: определения и задачи

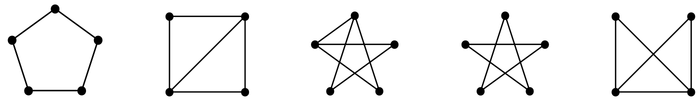
Ю. Кудряшов, П. Митричев

▷ **Определение 1.** *Графом* [6] называется пара $\Gamma = (V, E)$ из конечного множества *вершин* V и множества *рёбер* E, элементами которого являются (неупорядоченные) пары вершин графа Γ.

Граф можно представлять себе как множество точек, некоторые пары которых соединены линиями.

▷ **Определение 2.** Графы Γ_1 и Γ_2 называются *изоморфными*, если существует такая биекция $f\colon V(\Gamma_1) \to V(\Gamma_2)$, что вершины A и B графа Γ_1 соединены ребром тогда и только тогда, когда вершины $f(A)$ и $f(B)$ соединены ребром в графе Γ_2.

Задача 1. Какие из следующих пяти графов изоморфны?



Задача 2. Нарисуйте все неизоморфные друг другу графы с не более чем четырьмя вершинами.

Задача 3. Нарисуйте граф, вершинами которого являются натуральные числа от 1 до 15, а рёбрами соединены числа, одно из которых делится на другое.

Задача 4. а) Постройте граф с пятью вершинами, в котором нет ни трёх попарно соединённых, ни трёх попарно несоединённых вершин.

б) Докажите, что в каждой компании из шести человек найдутся либо три попарно знакомых, либо три попарно незнакомых человека.

Задача 5. Пусть в некоторой компании среди любых трёх человек найдутся два друга. Обязательно ли эту компанию можно разбить на две группы, так что всякие два человека из одной группы — друзья?

Задача 6. Найти наибольшее возможное количество рёбер в графе с n вершинами, если известно, что среди произвольных а) трёх, б*) четырёх его вершин есть две, не соединённые ребром.

▷ **Определение 3.** *Степенью вершины* A называется число выходящих из неё рёбер. Обозначение: $\deg A$.

[6] Точнее, *неориентированным графом без петель и кратных рёбер.*

Задача 7. Укажите степени всех вершин графов из задач 1, 2 и 3.

Задача 8. Докажите, что в графе с более чем одной вершиной есть две вершины одинаковой степени.

Задача 9. Докажите, что сумма степеней вершин произвольного графа равна удвоенному количеству его рёбер.

▷ **Определение 4.** *Путём* в графе называется конечная последовательность вершин (не обязательно различных), в которой всякие две соседние вершины соединены ребром. Путь называется *проходящим по данному ребру*, если это ребро соединяет некоторую пару соседних вершин пути. *Циклом* называется путь, в котором первая и последняя вершины совпадают. *Длиной пути* называется число рёбер, по которым проходит этот путь.

▷ **Определение 5.** Граф Γ называется *гамильтоновым*, если в нём существует путь, содержащий каждую вершину ровно один раз.

Задача 10. Докажите, что графы додекаэдра и икосаэдра гамильтоновы.

▷ **Определение 6.** Граф называется *связным*, если для любых двух его вершин существует путь, начинающийся в первой из них и заканчивающийся во второй.

Задача 11. Какие из графов задач 1, 2 и 3 связны?

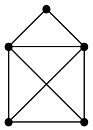
▷ **Определение 7.** Связный граф называется *деревом*, если в нём не сущесвует цикла, все рёбра которого различны.

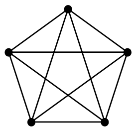
Задача 12. Докажите, что в любом дереве есть вершина степени 1.

Задача 13. Докажите, что в дереве число вершин на 1 больше числа рёбер.

▷ **Определение 8.** Граф называется *эйлеровым*, если в нём существует цикл, проходящий по каждому ребру ровно один раз.

Задача 14. Доказать, что следующие графы эйлеровы:





Задача 15. Докажите, что граф эйлеров тогда и только тогда, когда он связен и степень каждой его вершины чётна.

Задача 16*. В турнире без ничьих участвовало n команд. Каждая команда сыграла с каждой ровно по одному разу. Докажите, что можно так занумеровать команды числами $1, \ldots, n$, что $(i+1)$-я команда выиграла у i-й (для произвольного $i = 1, \ldots, n-1$).

Задача 17*. (*теорема Рамсея*) а) Докажите, что для произвольных натуральных m, n существует натуральное k такое, что в произвольном графе с k вершинами найдется либо m попарно соединенных ребрами вершин, либо n попарно несоединенных. Наименьшее такое k обозначается $R(m, n)$.

б) Найдите $R(3, 4)$.

Задача 18*. Докажите, что если для произвольных двух различных вершин A и B графа с n вершинами $\deg A + \deg B \geqslant n$, то этот граф гамильтонов (см. задачу 10).

▷ **Определение 9.** Назовём *расстоянием* между вершинами связного графа наименьшую длину пути, соединяющего эти вершины. *Диаметром графа* называется наибольшее расстояние между его вершинами.

▷ **Определение 10.** Граф называется *регулярным* графом *валентности* k, если степень каждой его вершины равна k.

▷ **Определение 11.** *Графом Мура* называется регулярный граф валентности k диаметра не больше 2, число вершин которого равно $k^2 + 1$.

Задача 19*. а) Докажите, что в регулярном графе валентности k и диаметра 2 не может быть больше, чем $k^2 + 1$ вершина.

б) Приведите примеры графов Мура при $k = 1, 2, 3$.

в) Существует ли граф Мура при $k = 7$?

г) Существует ли граф Мура при $k = 57$?

д) Докажите, что ни при каких других значениях k не существует графов Мура.

Задача 20*. Дан правильный 50-угольник. В одной из его вершин стоит доктор Фауст. У него есть три возможности: 1) бесплатно перейти в диаметрально противоположную точку; 2) заплатив Мефистофелю 1 рубль 05 копеек, перейти на соседнюю вершину против часовой стрелки; 3) получив от Мефистофеля 1 рубль 05 копеек перейти на соседнюю вершину по часовой стрелке. Известно, что доктор Фауст побывал в каждой вершине (хотя бы один раз). Докажите, что на каком-то отрезке пути кто-то кому-то заплатил не меньше 25 рублей.

Задача 21. В турнире по олимпийской системе участвовали n команд. Сколько всего было сыграно матчей?

Задача 22. Докажите, что граф с n вершинами, степень каждой из которых не менее $\frac{n-1}{2}$, связен.

Задача 23. В связном графе все вершины имеют степень 100. Докажите, что после удаления любого из рёбер он остаётся связным.

Задача 24. Докажите, что из любого связного графа можно выкинуть вершину и выходящие из неё рёбра, так, чтобы он остался связным.

Задача 25*. В кубической коробке $n \times n \times n$ лежали n^3 единичных кубиков. Кубики высыпали, каждый просверлили по диагонали, затем все плотно нанизали на нить и связали в кольцо (соединили вершину первого кубика с вершиной последнего). При каких n получившееся «ожерелье» можно убрать обратно в коробку?

Задача 26*. Все 28 Петиных одноклассников имеют по различному числу друзей в этом классе. Сколько из них дружат с Петей? А если одноклассников n?

Задача 27*. (Теорема Кэли) В графе с n вершинами каждая вершина соединена с каждой ребром (такой граф называется *полным*). Докажите, что существует ровно n^{n-2} способов выкинуть несколько рёбер так, чтобы оставшийся граф был деревом.

▷ **Определение 12.** *Ориентированным графом* называется граф, на рёбрах которого поставлены стрелки. Его рёбра называются *дугами*. Более формально, ориентированный граф — это пара $\Gamma = (V, E)$ из конечного множества вершин V и множества дуг E, элементами которого являются упорядоченные пары вершин графа Γ.

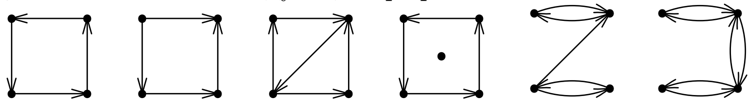
Заметим, что у нас в ориентированном графе разрешаются дуги из вершины в себя саму (*петли*), несколько дуг из одной вершины в другую (*кратные дуги*), «встречные» дуги (из A в B и из B в A).

То, что раньше называлось графом, мы теперь будем называть *неориентированным графом*.

Задача 28. Дайте (формальные!) определения *пути* и *цикла* в ориентированном графе.

▷ **Определение 13.** Ориентированный граф называется *сильно связным*, если для любых двух его вершин существует путь как из первой во вторую, так и из второй в первую.

Задача 29. Какие из следующих графов сильно связны?



▷ **Определение 14.** Ориентированный граф называется *связным*, если он окажется связным неориентированным графом после того, как мы сотрём стрелки с его дуг.

Задача 30. а) Приведите пример связного, но не сильно связного ориентированного графа.

б) Приведите пример связного ориентированного графа, в котором для некоторых двух вершин A и B нет пути ни из A в B, ни из B в A.

Задача 31. Докажите, что на рёбрах связного неориентированного графа можно так расставить стрелки, чтобы из одной из вершин существовали пути во все остальные.

Задача 32. Можно ли так расставить на рёбрах полного неориентированного графа стрелки, чтобы в полученном ориентированном графе не было циклов?

Задача 33. В полном неориентированном графе на рёбрах как-то расставили стрелки. Докажите, что найдётся вершина, из которой существуют пути во все остальные.

Задача 34*. В полном неориентированном графе с не менее чем тремя вершинами на рёбрах как-то расставили стрелки. Докажите, что можно заменить не более одной дуги на противоположную так, чтобы полученный граф стал сильно связным.

▷ **Определение 15.** Количество дуг, входящих в вершину, называется *входной полустепенью* (или *полустепенью захода*) этой вершины. Количество выходящих дуг называется *выходной полустепенью* (или *полустепенью исхода*).

Задача 35. Найдите входные и выходные полустепени каждой вершины для всех графов из задачи 29.

Задача 36. Что можно сказать о сумме всех входных полустепеней и сумме всех выходных полустепеней одного и того же ориентированного графа?

Задача 37. Сформулируйте и докажите критерий эйлеровости ориентированного графа.

Задача 38. (Цикл де Брюина) Для того, чтобы открыть кодовый замок (с кнопками от 0 до 9), необходимо набрать код из четырёх цифр, причём не важно, что было нажато до набора правильного кода. За какое наименьшее количество нажатий его можно гарантированно открыть?

Задача 39. 20 школьников решали 20 задач. Каждый решил ровно две задачи, и каждую задачу решили ровно двое. Доказать, что можно устроить разбор задач так, чтобы каждый рассказал одну решённую им задачу.

▷ **Определение 16.** Граф называется *двудольным*, если его вершины можно разбить на две группы (называемые *долями*) так, чтобы все рёбра (или дуги) были между различными долями.

Паросочетанием называется такой набор рёбер графа, что каждая вершина графа является концом не более одного ребра из набора. Паросочетание называется *совершенным*, если каждая вершина является концом ровно одного ребра паросочетания.

Раскраска вершин графа называется *правильной*, если никакие две вершины одного цвета на соединены ребром. Граф называется *k-дольным*, если правильная раскраска его вершин возможна k цветами и не менее.

Задача 40. Какие графы из задач 1, 2 и 3 являются двудольными? А сколько доль в остальных?

Задача 41. Равносильна ли двудольность неориентированного графа отсутствию циклов нечётной длины?

Задача 42. Любое ли дерево двудольно?

Задача 43*. (Теорема Холла) В некой компании n юношей. При каждом $k = 1, 2, \ldots, n$ для любых k юношей в этой компании найдется не менее k девушек, знакомых хотя бы с одним из рассматриваемых k юношей. Можно ли сосватать всех юношей за знакомых девушек? Является ли это условие необходимым?

Иными словами, верно ли, что в двудольном неориентированном графе (с n вершинами в первой доле) существует паросочетание размера n тогда и только тогда, когда для каждого набора из k вершин первой доли с ними соединены хотя бы k вершин второй доли?

Задача 44*. (Обобщение задачи 39) Докажите, что в любом регулярном двудольном неориентированном графе есть совершенное паросочетание.

Задача 45*. Верно ли, что при любой правильной раскраске k-дольного неориентированного графа в k цветов найдётся путь из k разноцветных вершин?

▷ **Определение 17.** Граф называется *планарным*, если его можно нарисовать на плоскости, изобразив вершины точками, а рёбра (или дуги) — непересекающимися кривыми. Граф, который нарисован на плоскости указанным выше образом, называется *плоским.* Части, на которые плоский граф делит плоскость (включая внешнюю часть) называются его *гранями.*

Задача 46. Какие графы из задач 1, 2 и 3 являются планарными?

Задача 47. (Формула Эйлера) Пусть в неориентированном плоском связном графе B вершин, P рёбер и $\varGamma$ граней. Тогда $B + \varGamma - P = 2$.

Задача 48. Чему равно $B + \varGamma - P$ для несвязного неориентированного плоского графа?

Задача 49. Пусть B, P и $\varGamma$ — количества вершин, рёбер и граней многогранника соответственно. Чему равно $B + \varGamma - P$?

Задача 50. а) Докажите, что в плоском неориентированном графе $2P \geqslant 3\varGamma$.

б) Докажите, что в плоском двудольном неориентированном графе $P \geqslant 2\varGamma$.

Задача 51. Докажите, что следующие графы не планарны:

а) Полный неориентированный граф с 5 вершинами. Этот граф обозначается K_5.

б) Двудольный неориентированный граф с 3 вершинами в первой доле и 3 вершинами во второй доле, причём каждая вершина первой доли соединена с каждой вершиной второй (такой граф называется *полным двудольным*). Этот граф обозначается $K_{3,3}$.

в) Произвольный неориентированный граф, у которого степени всех вершин не меньше шести.

▷ **Определение 18.** *Подграфом* данного графа называется граф, который получается из данного выкидыванием некоторых вершин и рёбер (дуг).

Два неориентированных графа называются *гомеоморфными*, если один можно получить из другого следующими операциями: взять ребро и добавить посередине этого ребра вершину; взять вершину степени 2 и заменить её и выходящие из неё рёбра на одно ребро (заметим, что эти две операции взаимно обратны).

Задача 52*. (Теорема Понтрягина-Куратовского) Докажите, что неориентированный граф планарен тогда и только тогда, когда у него нет подграфа, гомеоморфного K_5 или $K_{3,3}$.

Алгоритмы на графах (семинар)

В. Матюхин

Семинар — это форма теоретического занятия, когда школьникам предлагаются различные задачи на доказательство каких-либо свойств разобранных алгоритмов, на их модификацию и т. д. Все это позволяет лучше понять как сами алгоритмы, так и области их применения и особенности реализации.

Предлагаемый семинар по графам разбит на несколько разделов. В каждом разделе сначала приводятся задачи, а затем обсуждаются их решения. Будет полезно, если перед тем, как читать разборы задач, читатель некоторое время подумает над задачами и попытается их решить самостоятельно.

Предполагается, что читателю знакомы алгоритмы обхода графа в ширину и глубину, алгоритмы поиска кратчайших путей (Дейкстры, Флойда, Форда–Белмана) и построения каркасов минимального веса (Краскала и Прима).

Раздел 1. Обход в глубину и в ширину

В задачах этого раздела предполагается, что граф связный и неориентированный.

1. Докажите, что процедура обхода *в глубину* обойдет все вершины графа (т. е. не может оказаться так, что мы не побывали в некоторой вершине).

2. Докажите, что процедура обхода *в ширину* обойдет все вершины графа.

3. Докажите, что в результате обхода (как в глубину, так и в ширину) те ребра, по которым мы проходим, образуют остовное дерево.

4. В результате обхода в глубину получается дерево. Можно рассмотреть это дерево как корневое (корнем является вершина, из которой мы начали делать обход). Есть ребра исходного графа, которые в это дерево не попали. Докажите, что все такие ребра идут из более глубокой вершины какой-либо ветви построенного дерева в менее глубокую вершину той же ветви, и не бывает ребер, соединяющих вершины, принадлежащие разным ветвям.

5. С помощью алгоритма обхода в ширину можно искать кратчайшие пути в невзвешенном графе. Обоснуйте, почему найденные пути являются кратчайшими.

Обсуждение задач раздела 1

Задача 1 очень проста и интуитивно понятна. Давайте все-таки попробуем этот факт доказать строго. Предположим, что утверждение не верно. Т. е. в какую-то вершину мы в процессе обхода не попали (пусть это вершина B). Тогда, так как граф связен, то из исходной вершины A (из которой мы запускаем алгоритм обхода) существует путь в вершину B. Рассмотрим этот путь. В вершине A наш обход побывал, а в вершине B — нет. Тогда в этом пути есть две соседние (т. е. соединенные ребром) вершины (v_1 и v_2) такие, что в первой из них обход побывал, а во второй — нет. Вот тут-то мы и получаем противоречие. Раз обход побывал в вершине v_1, то, обрабатывая эту вершину, мы перебирали всех ее соседей. В том числе и вершину v_2. А так как мы не были в вершине v_2 ранее (вспомните, по нашему предположению, в вершине v_2 мы вообще не были!), то мы должны были пойти в вершину v_2. А значит, в вершине v_2 мы бы все-таки побывали.

Решение **задачи 2** полностью аналогично.

Задача 3. Давайте чуть-чуть переформулируем утверждение, которое нужно доказать. Во-первых, как мы уже доказали выше, мы побываем во всех вершинах, а значит, ребра, по которым мы пройдем в процессе обхода, образуют связный граф. Таким образом, нам осталось доказать, что в этом графе не будет циклов. Это можно сделать по-разному. Например, предположить, что цикл есть и доказать, что тогда в процессе обхода в какую-то вершину мы приходили дважды, получив тем самым противоречие. Но есть очень изящный способ.

Заметим, что каждый раз, когда мы проходим по ребру, мы добавляем новую вершину в список пройденных. Всего в процессе обхода мы добавим к списку пройденных все вершины графа (кроме начальной — она считается пройденной изначально). А значит, мы добавим $N-1$ вершину (если считать, что всего в графе N вершин). Следовательно, мы пройдем по $N-1$ ребру. Осталось вспомнить, что связный граф из N вершин, в котором $N-1$ ребро — дерево.

Задача 4. Для решения этой задачи нужно хорошо понимать, как устроен алгоритм обхода в глубину. Предположим, что ребро, соединяющее две разные ветви дерева, есть (пусть его концы — вершины u и v). Пусть, для определенности, обход в глубину сначала побывал в вершине u, а затем — в вершине v. Причем, поскольку эти вершины по нашему предположению принадлежат разным поддеревьям, то в вершину v обход попал, когда полностью закончил рассмотрение вершины u. Тогда, перебирая в вершине u всех ее соседей, обход в глубину увидел бы вершину v (ведь они соединены ребром) и должен был бы пойти в v

(ведь мы там еще не были!), но не пошел (по нашему предположению). Противоречие.

Задача 5. Изначально мы побывали только в начальной вершине (до нее расстояние 0). Затем мы добавили всех ее соседей (расстояние до них равно 1). Затем — их соседей (вершины на расстоянии 2) и т. д. Почему для всех вершин найденное расстояние будет кратчайшим? Действительно, пусть до некоторой вершины мы нашли не кратчайшее расстояние. Рассмотрим все такие вершины и выберем наиболее близкую к начальной вершине (назовем эту вершину v). Пусть реальное расстояние до нее равно x (соответственно, когда мы нашли расстояние, оно оказалось больше, чем x). Тогда у этой вершины есть сосед (обозначим его u), до которого расстояние $x-1$, причем это расстояние найдено правильно. Тогда, рассматривая эту вершину, мы из нее пойдем в вершину v (так как в вершине v мы еще не были) и запишем в вершину v число x. Противоречие. Почему, когда мы просматриваем u, вершина v еще не просмотрена? Потому что, как отмечено вначале, мы сначала просмотрим все вершины с расстоянием 0, потом — с расстоянием 1, 2, 3 и т. д. (подумайте, почему). А, значит, когда мы будем просматривать вершину u, расстояние до которой равно $x-1$, вершину, из которой мы пришли в v по нашему предположению мы еще не просмотрим (так как, раз в v мы по предположению запишем число больше x, то расстояние до вершины, из которой мы в нее придем, будет x или больше).

Раздел 2. Алгоритмы Дейкстры, Флойда и Форда-Белмана

В этом разделе все графы предполагаются ориентированными (рассматривать неориентированные графы с отрицательными весами ребер неинтересно).

1. Как с помощью алгоритма Флойда проверить, есть ли в графе циклы отрицательного веса? Как это же проверить с помощью алгоритма Форда-Белмана?

2. Как для двух вершин a и b установить, существует ли *кратчайший* путь из a в b?

3. Постройте пример графа (с отрицательными весами ребер), в котором путь, найденный алгоритмом Дейкстры, не будет кратчайшим.

4. Можно ли модифицировать алгоритм Дейкстры, чтобы искать длины путей в графах с отрицательными весами ребер, следующим образом? Найдем сначала самое маленькое ребро в графе (пусть оно имеет вес $-a$). Теперь прибавим к весу каждого ребра число $(a+1)$. После этого все ребра будут иметь положительный вес. Теперь с помощью алгоритма Дейкстры найдем кратчайший путь в новом графе, и будем считать, что он же является кратчайшим в исходном.

5. Рассмотрим следующую задачу. Пусть каждому ребру приписан вес. Пусть стоимость пути вычисляется как произведение весов ребер пути. Можно ли применять для подсчета кратчайшего пути в этом случае алгоритмы Дейкстры, Флойда, Форда-Белмана: а) если веса всех ребер положительны? б) если веса ребер неотрицательны? в) если веса ребер произвольные?

Обсуждение задач раздела 2

Первые две задачи этого раздела являются скорее контрольными вопросами или упражнениями, в то время как задачи 3, 4 и 5 требуют довольно серьезных размышлений.

Задача 1. Начнем с алгоритма Флойда. Что такое цикл отрицательного веса? Это путь из вершины в саму себя. А значит, нужно лишь просмотреть диагональ построенной алгоритмом Флойда матрицы и проверить, есть ли на ней отрицательные числа (подумайте, почему если цикл отрицательного веса существует, он будет найден алгоритмом Флойда).

С алгоритмом Форда-Белмана чуть сложнее. Во-первых, если окажется, что цикл отрицательного веса недостижим из той вершины, которая является начальной для алгоритма, то мы никак не узнаем о его существовании. Если же цикл отрицательного веса есть, то узнать об этом довольно просто. Напомним, что алгоритм Форда-Белмана состоит из $N-1$ шага, на каждом из которых мы перебираем все ребра и обновляем те значения расстояний до вершин, которые удается улучшить. Сделаем еще один аналогичный (N-й) шаг. Оказывается, что если на этом шаге меняется хотя бы одно из значений, то в графе есть цикл отрицательного веса, если нет — то циклов отрицательного веса, достижимых из начальной вершины, нет. Действительно, если в графе нет цикла отрицательного веса, то любой кратчайший путь из начальной вершины куда-либо состоит не более, чем из $N-1$ ребра, и, следовательно, будет найден за $N-1$ шаг алгоритма. При дальнейших шагах значения меняться уже не будут. Если же в графе есть цикл отрицательного веса, то нам выгодно «бегать» по этому циклу сколь угодно долго, и от этого расстояния до каких-то вершин будут уменьшаться. А значит, такое уменьшение какого-либо из значений произойдет и на N-м шаге (ведь если на каком-то шаге ни одно значение не меняется, то, из-за того, что шаги делаются одинаково, дальше уже ничего никогда меняться не будет).

Задача 2. Формулировка задачи кажется довольно странной. В то же время, задача отнюдь не так тривиальна, как кажется на первый взгляд. Итак, что же нужно проверить, чтобы с уверенностью уметь сказать, существует ли кратчайший путь между двумя вершинами или нет? Первое, что нужно проверить, это существует ли вообще путь из u в v. Если пути нет, то, естественно, нет и кратчайшего пути.

Как же может случиться, что путь есть, а кратчайшего пути нет? Такое бывает, когда могут быть пути сколь угодно маленькой длины (т. е. в графе есть цикл отрицательного веса). Тогда какой бы путь мы ни приняли за кратчайший, всегда можно найти путь еще меньшей длины.

Оказывается, впрочем, что просто проверить граф на наличие циклов отрицательного веса недостаточно. Может так случиться, что в графе есть цикл отрицательного веса, и в то же время кратчайший путь из u в v все-таки существует (остановитесь на минуту и подумайте, как такое возможно).

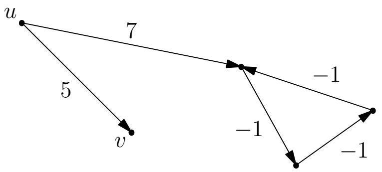
Итак, чтобы можно было найти путь из u в v сколь угодно маленькой длины, должны выполняться следующие условия:

— в графе существует цикл отрицательного веса;

— этот цикл достижим из вершины u;

— из этого цикла достижима вершина v.

Действительно, пусть в графе есть цикл отрицательного веса. Но если из него нельзя попасть в вершину v (см. рис.), то для уменьшения пути из u в v этот цикл использовать не удастся, и кратчайший путь из u в v вполне может существовать.

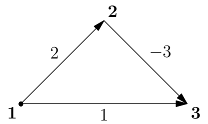


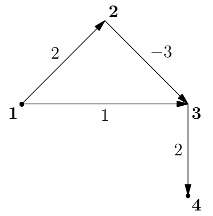
Теперь давайте сформулируем, как все-таки проверить существование кратчайшего пути из u в v, если мы пользуемся алгоритмом Флойда. Алгоритмом Флойда построим матрицу расстояний между всеми парами вершин. Дальше проверяем:

— Если пути из u в v не существует, то кратчайшего пути также не существует.

— Если удается найти такую вершину w (просто перебираем все вершины, и для каждой из них проверяем условие), что существует путь из u в w, путь из w в w имеет отрицательный вес (что соответствует циклу отрицательного веса, проходящему через w) и существует путь из w в v, то путь из u в v может иметь сколь угодно маленький вес.

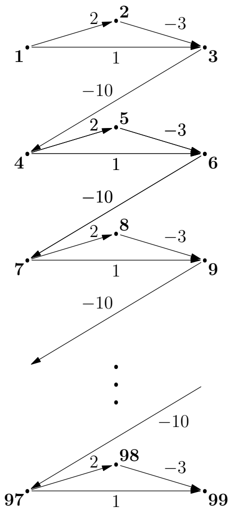
Если вершины w с описанным свойством нет, то кратчайший путь существует.





Задача 3. Как правило, первый пример, который удается найти в этой задаче, выглядит примерно так, как показано на рисунке слева. Однако здесь можно предложить модифицировать алгоритм так, чтобы значения в том числе и в тех вершинах, из которых мы уже «ходили» тоже обновлялись. Впрочем, тогда нетрудно модифицировать и предложенный пример, добавив к нему еще одно ребро (см. рис. справа). Очевидно, что на первом шаге мы найдем расстояния до вершин 2 (расстояние 2) и 3 (расстояние 1). Далее мы будем ходить из вершины 1, и найдем расстояние до вершины 4 (оно равно $1 + 2 = 3$). Далее мы будем ходить из вершины 2, и обновим значение в вершине 3 (записав туда $2 + (-3) = -1$). Но, так как из 3-й вершины мы уже ходили, то значение в вершине 4 так и останется равно 3, и путь длины 1 (через вершины 2 и 3) мы не найдем.

Здесь хочется сформулировать еще одну **задачу 3′**. Давайте модифицируем алгоритм следующим образом. Если мы обновляем значение в вершине, из которой мы уже ходили, мы снимаем пометку о том, что мы из нее ходили и теперь (когда до нее дойдет очередь) мы будем ходить из этой вершины еще раз. В этом случае в приведенном примере алгоритм будет давать правильный ответ, как и в других случаях. Что же получается, мы придумали модификацию алгоритма Дейкстры для графов с отрицательными весами ребер?



И да, и нет.

Рассмотрим пример,изображенный на рисунке (например, пусть граф будет состоять из 99 вершин, хотя, понятно, что его можно продлить и дальше).

Рассмотрим, как будет работать алгоритм Дейкстры. Сначала он пойдет из вершины 1, затем — из 3, затем — из 4, из 6, из 7, из 9, …, из 97, из 99. Далее алгоритм сделает ход из вершины 98, затем снова из 99. Далее алгоритм будет ходить из вершины 95 (в ней среди вершин, из которых мы еще не ходили, на данный момент будет записано самое маленькое число). Далее (поскольку это обновит значение в вершине 96), мы будем ходить из 96 — мы обновим значение в 97, далее будем ходить из 97, из 99, из 98. Далее мы будем ходить из вершины 92, от этого обновится 93, пойдем из 93, дойдем до самого низа, потом обновим нижний треугольник, опять пойдем из вершины 95, в результате чего опять обновим нижний треугольник. И так далее. Каждый раз, ходя из вершины какого-либо треугольника, мы будем доходить до низу, обновлять нижний треугольник, далее ходить из предпоследнего треугольника и снова обновлять нижний треугольник. После чего ходить из третьего снизу треугольника, опять доходить до низу, опять обновлять второй снизу и опять доходить до низу и т. д. Нетрудно заметить, что этот алгоритм будет иметь уже не полиномиальную, а экспоненциальную сложность (докажите это!)

Задача 4. Задача довольно интересна тем, что этот метод талантливые школьники часто придумывают сами. Кажется, от того, что мы все ребра графа увеличили на одну и ту же величину, ничего принципиально не изменилось, и кратчайший путь должен остаться кратчайшим путем. Оказывается, что это все-таки не так. Пусть мы увеличили длины всех ребер на величину b. Тогда длина пути, состоящего из одного ребра, увеличится на b, пути, состоящего из двух ребер — на $2b$, из трех — на $3b$. За счет этого кратчайший путь в исходном графе может оказаться не самым коротким в модифицированном. Например, пусть в графе было три ребра: ребро $(1, 2)$ веса 1, ребро $(1, 3)$ веса 5, ребро $(3, 2)$ веса -50. Тогда кратчайшим путем из 1 в 2 является путь через вершину 3 (его вес равен -45). Однако если веса всех ребер увеличить, как предлагается, на 51, то ребро $(1, 2)$ будет весить 52, ребро $(1, 3)$ будет весить 56, ребро $(3, 2)$ будет весить 1. Тогда вес пути напрямую будет 52, а вес пути через вершину 3 — 57. То есть кратчайшим в этом графе будет прямой путь.

Задача 5. Пункт а. Пусть веса всех ребер положительны. Припишем каждому ребру новый вес — логарифм по некоторому (одинаковому для всех) основанию от исходного веса ребра. Для определенности возьмем логарифм по основанию 2. Заметим теперь, что если у нас есть путь и мы просуммируем новые веса ребер вдоль него, то чтобы получить вес пути в исходной задаче, нам достаточно 2 возвести в получившуюся степень. Действительно, пусть путь проходит через три ребра, имеющие веса a, b, c. Тогда вес этого пути равен abc. Но эту же величину можно записать как $2^{\log a+\log b+\log c}$. Заметим, что в силу монотонности, чем больше сумма логарифмов, тем больше и произведение. Т. е. кратчайший путь в новом смысле является кратчайшим путем в исходной задаче.

Тем самым, для решения исходной задачи мы можем заменить веса ребер их логарифмами, а правило вычисления стоимости пути — на стандартное правило суммирования весов ребер. А дальше можем применить стандартные алгоритмы: Флойда, Форда-Белмана, а если все исходные веса не меньше 1, то и алгоритм Дейкстры.

Пункт б. Если исходные веса ребер могут быть как положительны, так и 0, то взять логарифмы от ребер веса 0 не получится. При этом реально можно их считать минус бесконечностью (очень маленьким числом). А можно подойти и по-другому. Сначала попробовать найти путь, проходящий через нулевое ребро (его вес будет равен 0, веса всех остальных путей заведомо не меньше). А если найти путь с нулевым ребром не получилось, то можно из графа удалить все ребра веса 0, после чего получится задача из пункта а.

Кстати, подумайте, как найти какой-нибудь путь, проходящий через нулевое ребро.

Пункт в мы оставим читателю для самостоятельного исследования.

Раздел 3. Ориентированные графы

1. Докажите, что если ориентированный граф не содержит ориентированных циклов, то топологическая сортировка его вершин возможна. Иначе говоря, вершины графа можно пронумеровать натуральными числами от 1 до N так, чтобы все дуги (дугами принято называть ребра ориентированного графа) шли из вершин с меньшим номером в вершины с большим.

2. Пусть в ориентированном графе для любых двух вершин i и j есть либо дуга из i в j, либо из j в i (такой граф называется турниром). Докажите, что в этом графе можно построить гамильтонову цепь (т. е. путь, проходящий через каждую вершину ровно один раз). Иначе говоря, если N команд сыграли полный турнир (каждая с каждой) без ничьих (результаты можно изобразить графом, проведя дуги от выигравшей команды к проигравшей), то эти команды можно выстроить в шеренгу так, что каждая команда проиграла своему левому соседу в шеренге, и выиграла у правого.

3. Граф задан матрицей смежности. Требуется за $O(N)$ найти вершину, в которую входят ребра из всех вершин и из которой не выходит ни одного ребра, либо установить, что такой вершины нет.

Обсуждение задач раздела 3

Задача 1. Утверждение. *Если в графе нет ориентированных циклов, то существует вершина, в которую не входит ни одна дуга.* Действительно, если в каждую вершину входит хотя бы одна дуга, то можно сделать следующее. Встанем в какой-нибудь вершине, и пойдем в любую вершину, из которой в данную вершину есть дуга (т. е. мы идем в направлении, обратном направлению дуг), оттуда — опять же в вершину, из которой есть дуга в эту вершину и т. д. Так как в каждую вершину входит хотя бы одна дуга, то процесс бесконечен. Так как вершин в графе N, то рано ли поздно мы окажемся в вершине, в которой мы уже были. А тогда в графе есть цикл (мы прошли по нему в обратном направлении). Утверждение доказано.

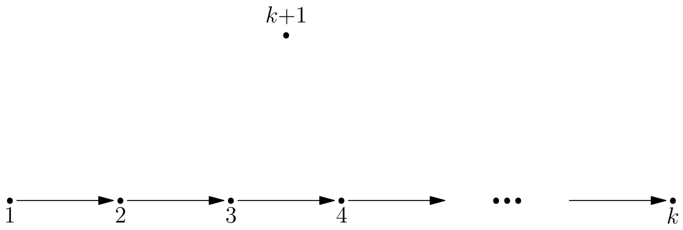
Теперь рассмотрим такой алгоритм. Найдем вершину, в которую ничего не входит. Поставим в нее число 1, и удалим эту вершину, а также все дуги, которые из нее выходили. Оставшийся граф также не будет содержать ориентированных циклов (откуда бы им взяться?), а значит, в нем будет вершина, в которую не входит ни одна дуга (в исходном графе в эту вершину могло не входить ни одной дуги, а могла входить дуга из вершины, которую мы нашли первой). Поставим в эту вершину число 2 и опять же удалим ее. Продолжим процесс. На каждом шаге нам удастся находить вершину, в которую не входит ни одной дуги, и ставить туда очередное число. Тем самым мы пронумеруем все вершины. Почему эта нумерация будет правильной? Потому что мы ставили в вершину очередное число, когда в нее не вело ни одной дуги, а это значит, что к этому моменту все вершины, из которых в исходном графе есть дуги в данную вершину, были уже из графа удалены, т.е. получили свои номера, и эти номера были меньше номера данной вершины.

Задача 2. Воспользуемся методом математической индукции.

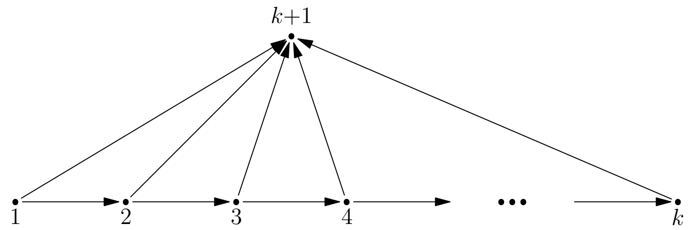
Давайте сначала оставим только две вершины. Между ними есть или дуга от первой ко второй, или от второй к первой. Таким образом, для двух вершин существует путь, проходящий через обе эти вершины.

Давайте предположим, что мы уже доказали, что для K вершин утверждение задачи верно. Докажем теперь, что оно верно для $K+1$ вершины.

Удалим вершину $K+1$ и все дуги, выходящие из нее или входящие в нее. Останется граф с тем же свойством на K вершинах. Для этого графа можно построить путь, проходящий через все K вершин (по предположению индукции). Давайте пронумеруем эти K вершин номерами от 1 до K в соответствии с их номером в пути. Теперь рассмотрим $(K+1)$-ю вершину:



Если из вершины $K+1$ идет дуга в вершину 1, то искомым является путь $(K+1, 1, 2, \ldots, K)$. Если же, наоборот, дуга идет из 1 в $K+1$, то рассмотрим, в какую сторону направлена дуга между вершинами 2 и $K+1$. Если от $K+1$ в 2, то искомым является путь $(1, K+1, 2, 3, \ldots, K)$, если же от 2 в $K+1$, то рассмотрим дугу между 3 и $K+1$. И так далее, если в какой-то момент нам встретится ситуация, что из вершины i идет дуга в $K+1$, а из $K+1$ — в $i+1$, то искомый путь будет существовать: $(1, 2, \ldots, i, K+1, i+1, i+2, \ldots, K)$. Поскольку мы рассматриваем ситуацию, когда из вершины 1 идет дуга в $K+1$ (обратную ситуацию мы рассмотрели в самом начале), то единственная возможная плохая для нас ситуация, когда такого нет, когда из всех вершин идут дуги в $K+1$ (см. рис. ниже) В том числе и из последней, K-й вершины. Но тогда искомый путь все равно существует: это путь $(1, 2, \ldots, K, K+1)$.



Задача 3. Давайте договоримся, что в матрице смежности в i-й строке будут стоять единички в тех столбцах, которые соответствуют вершинам, в которые из вершины i идут дуги.

Будем считать, что на главной диагонали матрицы стоят 0.

Давайте сформулируем в терминах матрицы смежности, что же нам нужно найти. Нам нужно найти некоторую i-ю строчку, в которой стоят все 0, а в i-м столбце стоят все 1 (за исключением диагонального элемента). В качестве упражнения докажите, что в графе не может быть двух вершин с таким свойством.

Рассмотрим такой алгоритм. Встанем в первую строку, и пойдем по ней начиная с первого элемента. Как только нам встретится 1 (пусть это произошло в j-м столбце) мы перейдем на j- ю строчку и продолжим ее проверку с $j+1$-го элемента. Если нам еще встретится 1, мы снова перейдем на соответствующую строчку и продолжим ее проверку. Как только мы дойдем до конца (N-го элемента строки), так можно утверждать, что данная вершина является единственным кандидатом. Нам останется проверить целиком строчку, в которой мы оказались и соответствующий ей столбец. Если они удовлетворяют условию, то мы нашли то, что искали, если нет, то того, что мы ищем — нет (при этом если при проверке в строке окажутся единицы, то переходить на другие строки уже не надо).

	1	**2**	**3**	**4**	**5**
1	0	0	1	1	1
2	0	0	0	1	0
3	1	0	0	1	1
4	0	0	0	0	0
5	0	1	0	1	0

Заметьте, что первый проход по строке (возможно, с переходом на строчки вниз) займет N операций, еще N операций потребуется для проверки строки, которую мы найдем и еще N — для проверки столбца. Тем самым, за $3N$ проверок мы решим нашу задачу. В качестве самостоятельного упражнения докажите, почему этот алгоритм всегда найдет искомую вершину, почему он правильно устанавливает, когда ее нет.

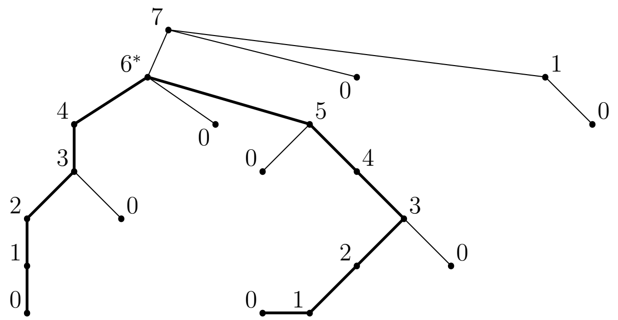
Раздел 4. Каркасы и деревья

1. Докажите, что алгоритм Краскала строит именно минимальный каркас.

2. Докажите, что алгоритм Прима строит минимальный каркас.

3. В дереве требуется найти две наиболее удаленные друг от друга вершины. Можно ли это сделать за $O(N)$? Если да, то каким образом и как для этого должно быть представлено дерево в памяти?

Задачи 1 и 2 описаны в литературе.



Задача 3. Подвесим дерево за какую-нибудь вершину неединичной степени. Тогда рассмотрим, как будет устроен самый длинный путь — он будет от некоторого листа подниматься до некоторой вершины, после чего опускаться до другого листа. В нем есть некоторая самая высокая вершина (будем называть ее «переломной»), и он распадается на два подпути, каждый из которых идет снизу вверх, и эти пути в своих поддеревьях являются самыми длинными (см. рис.).

Давайте из вершины, за которую мы довесили дерево, запустим процедуру обхода в глубину. Пусть эта процедура для каждой вершины возвращает длину самого длинного пути от этой вершины до некоторого листа (на рисунке около каждой вершины написано значение, которое вычислить эта процедура). Понятно, что процедура обхода в глубину это значение может легко вычислять (как максимум из аналогичных значений для поддеревьев плюс 1).

Помимо этого, обход в глубину будет вычислять длину самого длинного пути, в которой данная вершина является «переломной» (и запоминать наибольшее найденное значение в некоторой глобальной переменной). Нужно найти два поддерева с наибольшими длинами наибольших путей, просуммировать эти длины и добавить к ним 2 (ребра от данной вершины до вершин, где начинаются эти деревья). На приведенном рисунке «переломной» для самого длинного пути будет вершина, помеченная звездочкой, а длина самого длинного пути равна $4 + 5 + 2$.

Сложность алгоритма равна сложности обхода в глубину и при хранении графа списком ребер, выходящих из каждой вершины, достигает $O(N + M)$, а в дереве, как известно, $M = N - 1$, и, тем самым, сложность равна $O(N)$.