Курс лекций по олимпиадной информатике (C)

Лекции 1-8

Михаил Густокашин, 2009

Конспекты лекций подготовлены для системы дистанционной подготовки, действующей на сайте **informatics.mccme.ru**.

При нахождении ошибок или опечаток просьба сообщать по адресу

gustokashin@gmail.com

Оглавление

1 Арифметика и теория чисел **5**

1.1 Организация ввода-вывода из файла 5
1.2 Интуитивное понятие сложности алгоритма 6
1.3 Целочисленные типы данных и их использование 7
1.4 Длинные числа и операции над ними 8
1.5 Делимость и делители 11
1.6 НОД и НОК. Элементы теории остатков 11
1.7 Разложение числа на простые множители 13
1.8 Быстрое возведение в степень 14
1.9 Матрицы и операции над ними 15

2 Битовые операции и структуры данных **19**

2.1 Битовые операции 19
2.2 Стеки 21
2.3 Очереди 22
2.4 Деки 23
2.5 Кучи 24
2.6 Динамически расширяемые массивы 26
2.7 Списки 27
2.8 Сравнение динамических структур 30
2.9 Хеш-таблицы 30
2.10 Механизм запуска функций и рекурсия 33

3 Алгоритмы поиска **37**

3.1 Поиск в неупорядоченных массивах 37
3.2 Поиск порядковых статистик 38
3.3 Бинарный поиск в упорядоченных массивах 40
3.4 Бинарный поиск для монотонных функций 41
3.5 Бинарный поиск по ответу 41
3.6 Поиск по групповому признаку 44

4 Алгоритмы сортировки **47**

4.1 Сортировка пузырьком 47
4.2 Сортировка прямым выбором 48
4.3 Пирамидальная сортировка 49
4.4 Быстрая сортировка 50
4.5 Сортировка слияниями 52

4.6 Сортировка подсчетом . . . 53
4.7 Поразрядная сортировка . . . 53
4.8 Сравнение производительности сортировок . . . 56
4.9 Сканирующая прямая . . . 56

5 STL **59**
5.1 Введение . . . 59
5.2 Пара (pair) . . . 60
5.3 Стек (stack) . . . 60
5.4 Очередь (queue) . . . 61
5.5 Дек (deque) . . . 62
5.6 Очередь с приоритетами (куча, priority_queue) . . . 62
5.7 Динамически расширяемый массив (vector) . . . 62
5.8 Вектор битов (bit_vector) . . . 63
5.9 Строка (string) . . . 63
5.10 Итераторы . . . 64
5.11 Хеш-таблица (hash_set) . . . 65
5.12 Хеш-словарь (hash_map) . . . 66
5.13 Алгоритмы в STL . . . 66
5.14 Использование собственных структур . . . 68
5.15 Пример решения задачи с использованием STL . . . 69

6 Задачи на анализ таблиц **73**
6.1 Введение . . . 73
6.2 Вычисление по локальным данным . . . 74
6.3 Кратчайшие пути в лабиринте . . . 76
6.4 Система непересекающихся множеств . . . 77
6.5 Выделение связных областей . . . 79

7 Динамическое программирование с одним параметром **83**
7.1 Введение . . . 83
7.2 Оптимизация целевой функции . . . 84
7.3 Восстановление решения . . . 85
7.4 Подсчёт числа ответов . . . 88

8 Динамическое программирование с двумя параметрами **91**
8.1 Введение . . . 91
8.2 Использование нескольких подзадач . . . 91
8.3 Использование предыдущего столбца . . . 94
8.4 LR-динамика . . . 98
8.5 Динамика по профилю . . . 102

Лекция 1

Арифметика и теория чисел

Версия C от 16.11.2009

1.1 Организация ввода-вывода из файла

В большинстве олимпиад по информатике входные данные вводятся из текстового файла и выводиться должны так же в файл. Очень редко возникает необходимость считывать данные из файла дважды; организаторы стараются избегать таких задач. Мы рассмотрим метод, который позволит читать и писать данные из файлов так же, как с консоли — это позволит избежать путаницы и облегчить процесс отладки.

В лекциях мы будем рассматривать программы на языке **C**. В частности, для пишущих на **C**, рекомендуем использовать библиотеку `stdio.h` для реализации ввода-вывода вместо `iostream` и других библиотек для работы с потоками ввода-вывода. Этот выбор связан со значительно более высокой производительностью библиотеки `stdio.h`. Для пользующихся **Java** рекомендуется самостоятельно изучить функции для работы с потоками ввода-вывода, а для использующих `iostream` — изучить эту библиотеку или перейти на использование `stdio`.

Итак, перейдем к рассмотрению организации ввода-вывода. Допустим, перед нами стоит задача сложить два целых числа, не превосходящих по модулю 10000. Входные данные находятся в файле `input.txt`, вывод должен осуществляться в файл `output.txt`.

Рассмотрим решение этой задачи:

```
#include <stdio.h>

int main(void)
{
  int i, j;
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);
  scanf("%d%d", &i, &j);
  printf("%d", i+j);
  return 0;
}
```

Первая функция `freopen` перенаправляет стандартный поток ввода (`stdin`) на файл

(input.txt) для чтения (r. Полностью аналогично в следующей строке перенаправляется стандартный поток вывода.

Стоит заметить, что при использовании этого метода нет необходимости использовать fclose для закрытия файлов, т.к. стандартные потоки закрываются автоматически в конце работы программы.

Для отладки достаточно закомментировать строки, перенаправляющие ввод-вывод в файл, и тогда программа будет считывать и выводить данные на консоль. Обратите внимание, что следует не забывать убрать комментарий перед отправкой решения на проверку.

1.2 Интуитивное понятие сложности алгоритма

Оценкой эффективности реализации решения принято считать время работы программы и количество используемой памяти. Стандартные сообщения тестирующей системы о превышении максимальных лимитов обычно носят вид **TL** (Time Limit exceed, превышен лимит по времени) и **ML** (Memory Limit exceed, превышен лимит по памяти).

Эффективность программы обычно определяется, исходя из сочетания времени работы программы и используемого объема памяти, причем важность каждого из компонентов определяется в зависимости от задачи. В случае же олимпиадного программирования следует использовать еще один решающий компонент оценки эффективности — время, затраченное на написание программы. Таким образом, наиболее эффективным решением олимпиадной задачи по информатике можно считать решение, укладывающееся в **TL** и **ML**, и написанное за кратчайшее время.

Основная часть времени работы программы состоит из произведения количества итераций цикла (т.е. сколько раз были выполнены действия) на время выполнения команд в цикле. Время выполнения условно будем называть «константой» (не зависящей от входных данных), а количество итераций — «сложностью» алгоритма. Допустим, алгоритм линейный, т.е., например, на вход дается N чисел, и следует подсчитать их сумму. Тогда сложность алгоритма будет обозначаться как $O(N)$ (O-большое от N). Константа же для этой задачи может различаться в зависимости от конкретных условий. Например, для целых чисел константа будет в несколько раз меньше, чем для вещественных (т.к. целочисленные операции выполняются в несколько раз быстрее). Для квадратичного алгоритма (например, для сортировки пузырьком) сложность записывается как $O(N^2)$. Сложность может зависеть и от нескольких параметров, если они определяют количество циклов. Тогда сложность может записываться как $O(N^2+M^3)$. Если в алгоритме нет циклов, зависящих от входных данных, то говорят, что он «работает за константу», т.е. за $O(1)$.

Обычно за O-большое обозначают сложность в худшем случае. Некоторые алгоритмы работают очень по-разному в зависимости от входных данных, в этом случае мы будем отдельно указывать «сложность в среднем».

Очень часто в процессе подсчета сложности алгоритма используется функция $\log N$ («логарифм от N»). В программировании мы обычно будем использовать двоичный логарифм. Он определяется так: $2^{\log_2 N} = N$. В дальнейшем под записью $\log N$ мы будем понимать $\log_2 N$. Интересная особенность логарифма заключается в том, что он растет очень медленно при росте N. Так при $N = 8 : \log N = 3$, $N = 65536 : \log N = 16$, $N = 4294967296 : \log N = 32$. Скорость роста логарифма намного меньше, чем даже у

квадратного корня. Поэтому при больших значениях N алгоритм со сложностью $\log N$ и большой константой будет эффективнее линейного алгоритма с маленькой константой.

На этом закончим введение в теорию сложности алгоритмов.

1.3 Целочисленные типы данных и их использование

На данный момент на олимпиадах используются, в основном, 32-битные системы и компиляторы. Приведем таблицу, в которой указаны максимальные и минимальные значения для основных типов переменных:

Название	**Байт**	**Минимум**	**Максимум**
unsigned char	1	0	255
char	1	−128	127
short	2	−32768	32767
unsigned short	2	0	65535
int	4	−2147483648	2147483647
unsigned int	4	0	4294967295
int64 (long long)	8	-2^{63}	$2^{63}-1$

Следует заметить, что операции с `int64` выполняются несколько медленнее, чем с другими типами целочисленных данных.

При программировании следует стараться использовать тип `int` кроме ситуаций, когда использование меньшего типа позволит уложиться в **ML** или требуется заведомо больший тип данных.

Отметим некоторые особенности использование типа `int64` в языке **C**. Во-первых, в некоторых реализациях он обозначается как `__int64`, в некоторых других (в основном в UNIX-системах): `long long`. Кроме того, на различных платформах используется своя форматная строка для ввода-вывода `int64`. Универсальный вывод программируется с помощью макрокоманд и выглядит следующим образом:

```
#ifdef WIN32
  printf("%I64d", n);
#else
  printf("%lld", n);
#endif
```

В некоторых ситуациях не хватает даже типа `int64`. Иногда, если данные не намного превышают размер `int64`, можно воспользоваться типами переменных с плавающей точкой. Так `long double` может хранить без ошибок целое число длиной до 19 десятичных знаков. Однако следует помнить, что вещественные типы обрабатываются значительно медленнее, для них не определены операции взятия остатка и деления нацело. Несмотря на это, все же существуют ситуации, когда их использование оправдано.

Классический пример на тему выбора типа переменных выглядит так: даны два числа X и Y по модулю меньшие, чем 2^{31}, следует вывести их сумму. Участник смотрит на данные, выбирает тип `int` и получает ошибку, т.к. результат сложения может не уместиться в этот тип данных.

Если же не хватает и этих переменных, то единственный выход — использование длинной арифметики, т.е. арифметики на массивах, чему и будет посвящена следующая часть лекции.

1.4 Длинные числа и операции над ними

При рассмотрении этой темы нам потребуются знания, полученные в начальной школе. В частности, выполнение арифметических операций «в столбик».

Идея реализации длинных чисел заключается в использовании массивов, состоящих из чисел «коротких» (они будут играть роль цифр). Будем придерживаться соглашения, что числа прижаты к «правому краю», т.е. младшие разряды стоят в ячейках с большим номером.

Во-первых, определим максимальную длину длинного числа. В общем случае — это небанальная задача, и этот параметр определяется с помощью комбинаторных формул, но можно постараться подобрать заведомо удовлетворяющее нашим потребностям значение.

Удобнее всего определить один раз максимальное значение длины (количества элементов в массиве), чтобы можно было уменьшать его для более удобной отладки. Это делается с помощью следующей макроподстановки:

```
#define MAXLEN 1000
```

Мы определили максимальную длину массива в 1000 «цифр». Кроме того, удобно определить структуру для хранения числа:

```
typedef struct
{
  int val[MAXLEN+1];
  int st;
} vlong;
```

Поле `st` будет указывать на ту позицию, с которой начинается осмысленная часть числа (т.е. начинаются отличные от нуля цифры; в дальнейшем мы увидим, что разумное использование этого поля избавляет нас от огромного количества бесполезных операций). Макроподстановка `typedef` делается для того, чтобы не писать каждый раз слово `struct`. Программирующие на **C++** и **Java** могут создать класс и дальнейшие функции записывать в виде переопределенных операторов этого класса.

Вспомним, как осуществляется сложение в столбик. Мы начинаем идти с младших разрядов, складываем цифры, стоящие на данных позициях, прибавляем то, что было «в уме», новое значение «в уме» равно целой части от деления полученного результата на 10, в итоговое число на эту позицию — остаток от деления результата на 10. Так мы повторяем до тех пор, пока оба числа не кончатся (т.е. мы не дойдем до начала большего числа), в случае же, если «в уме» не ноль — мы дописываем эту цифру в начало. В реализации длинного сложения — все то же самое, только для экономии времени и места мы можем использовать, например, не систему счисления с основанием 10, а систему с основанием в 10000 (для сложения мы можем выбрать и большее значение). Для реализации этого метода и, опять же, более удобной отладки удобно задать константу с основанием системы счисления:

```
#define SYS 10000
```

Перейдем собственно к описанию функции сложения:

```
void add(vlong *op1, vlong *op2, vlong *res)
{
  vlong *mxop, *mnop;
  int i, flag=0, st;
  mxop = op1->st>op2->st?op1:op2;
  mnop = op1->st<=op2->st?op1:op2;
  st=mnop->st;
  for(i=MAXLEN; i >= mxop->st; i--) {
    res->val[i] = mxop->val[i] + mnop->val[i] + flag;
    flag = res->val[i] / SYS;
    res->val[i] %= SYS;
  }
  for(i=mxop->st-1; i >= mnop->st; i--) {
    res->val[i] = mnop->val[i] + flag;
    flag = res->val[i] / SYS;
    res->val[i] %= SYS;
  }
  if (flag) res->val[--st] = flag;
  res->st = st;
}
```

Сделаем некоторые пояснения к программе. Первым делом она выбирает большее и меньшее по количеству знаков число. Здесь опять же полная аналогия со сложением чисел в столбик: пока у нас оба числа еще не кончились, мы складываем по всем правилам, а затем переписываем начало большего числа, не забывая про значение «в уме».

Обычно большие числа инициализируются какими-либо малыми значениями, над которыми затем производятся операции, которые и приводят к росту этого числа. Т.е. короткое число надо записывать в `x.val[MAXLEN]`, а `x.st` устанавливать в `MAXLEN`.

Если `a`, `b` и `c` - переменные типа `vlong`, то вызов функции должен выглядеть как `add(&a, &b, &c)`. Это сделано для того, чтобы передавалось не само значение структуры, а указатель на нее (и не было необходимости в копировании).

Вывести значение длинного числа можно следующим образом:

```
printf("%d", c.val[c.st]);
for (i=c.st+1; i<=MAXLEN; i++)
  printf("%.4d", c.val[i]);
```

Обратите внимание на то, что первое число выводится без ведущих нулей, а каждое следующее должно выводится с ведущими нулями. При изменении базовой системы счисления (в нашем случае это 10000) надо заменять цифру в выражении `%.4d` на количество нулей в основании этой системы. Просто выводить элементы массива с помощью `%d` нельзя! Такую ошибку достаточно легко допустить и очень сложно найти.

Сложность полученного алгоритма получилась $O(N)$, где N — количество разрядов в большем числе. Если бы мы не использовали поля `st`, то нам каждый раз пришлось бы складывать все `MAXLEN` разрядов, а с учетом того, что очень часто используется не весь массив (в процессе вычисления числа растут, или мы не можем точно определить длину заранее), то сложность программирования оправдывается производительностью.

Длинное вычитание осуществляется аналогично. Используются те же идеи, что и в вычитании в столбик.

Теперь рассмотрим умножение длинного числа на короткое (коротким числом будем называть то, квадрат которого помещается в переменную элементарного типа, т.е. меньшее SYS). При этом функцию умножения на короткое будем писать сразу с прицелом на использование в умножении длинного на длинное.

```
void mul(vlong *op1, int sh, int offs, vlong *res)
{
  int i, flag=0;
  int st = op1->st;
  for(i=MAXLEN-offs+1; i <= MAXLEN; i++)
    res->val[i] = 0;
  for(i=MAXLEN; i >= st; i--) {
    res->val[i-offs] = op1->val[i] * sh + flag;
    flag = res->val[i-offs] / SYS;
    res->val[i-offs] %= SYS;
  }
  if (flag) res->val[--st-offs] = flag;
  res->st = st-offs;
}
```

Сложность алгоритма умножения длинного числа на короткое составляет $O(N)$, где N — количество разрядов в длинном числе. С использованием двух предыдущих функций умножение длинного числа на длинное реализуется очень просто.

```
void mul_long(vlong *op1, vlong *op2, vlong *res)
{
  int i;
  vlong temp;
  res->st = MAXLEN;
  res->val[MAXLEN] = 0;
  for (i=MAXLEN; i>= op1->st; i--) {
    mul(op2, op1->val[i], MAXLEN-i, &temp);
    add(res, &temp, res);
  }
}
```

Вызов функции умножения на короткое: `mul(&x, sh, 0, &res)`, где `x` — это длинное число, `sh` — короткое, `res` — результат. Вызов функции умножения длинного на длинное: `mul_long(&op1, &op2, &res)`, где `op1` и `op2` — длинные числа, а `res` — их произведение.

Инициализация значений и вывод осуществляется аналогично сложению.

Сложность умножения длинного числа на длинное получилась $O(N \times M)$, где N и M — количество разрядов в операндах. Существуют более быстрые методы умножения длинных чисел (например, метод Карацубы), однако в школьных олимпиадах их применение требуется крайне редко.

Длинное деление используется гораздо реже и реализуется аналогично делению в столбик. Длинные вещественные числа также оставим без внимания (интересующиеся могут найти способы хранения вещественных чисел в памяти компьютера и реализовать нечто подобное на массивах).

На этом закончим рассмотрение длинных чисел.

1.5 Делимость и делители

Отвлечемся от технических вещей и перейдем к программе курса математики начальной школы.

Простым числом называется натуральное число, большее 1, которое делится нацело только на себя и 1 (имеет два натуральных делителя). Составными числами, соответственно, называются все остальные натуральные числа, большие единицы.

Простые числа имеют множество полезных применений, а также и сами по себе нередко являются сутью задачи. Наиболее известное промышленное применение простых чисел — в шифровании **RSA** с открытым ключом.

В первую очередь нам необходимо уметь проверять, является ли число N простым. Т.е. нам необходимо узнать, существуют ли такие натуральные числа x, y ($1 < x, y < N$), что $x \times y = N$. Кроме того, условимся, что $x \leq y$, тогда можно сказать, что x меньше либо равен квадратному корню из числа N (если это условие не выполнено, то произведение будет заведомо больше N).

Таким образом, можно написать следующую функцию, которая будет довольно эффективно проверять число на простоту:

```
int isprime(int n)
{
  int i, j;
  if (2 == n) return 1;
  j = (int)sqrt((double)n)+1;
  for (i=2; i<=j; i++)
    if (!(n%i)) return 0;
  return 1;
}
```

Чтобы функция `sqrt` работала, необходимо подключить библиотеку `math.h`.

Пользуясь теми же соображениями, очень просто написать функцию, находящую все разложения числа на два множителя. Для этого достаточно убрать проверку на равенство N двум и в заменить `return 0` на обработку двух найденных делителей `i` и `n/i`.

Оба этих алгоритма имеют сложность $O(\sqrt{N})$.

1.6 НОД и НОК. Элементы теории остатков

Наибольшим общим делителем (НОД) двух натуральных чисел называется такое максимальное натуральное число, которое является делителем и первого и второго

числа. Наименьшим общим кратным (НОК) двух натуральных чисел называется такое минимальное натуральное число, которое делится нацело на оба этих числа.

Аналогично вводятся понятия НОД и НОК многих чисел. На практике НОД и НОК многих чисел считаются с помощью последовательно попарного подсчета НОД и НОК (т.е. считается НОД уже обработанной части и очередного числа).

Задачи, в которых необходимо подсчитать НОД или НОК возникают довольно часто, особенно для НОД. Так, например, в подсчете количества точек с целыми координатами на отрезке используется НОД x и y координат отрезка.

В младшей школе изучается алгоритм Евклида, который позволяет найти НОД двух чисел. В нем используются следующие соотношения:

1) НОД$(a, 0) = a$

2) НОД(a, b) =НОД$(a\%b, b)$ =НОД$(a, b\%a)$

Запишем функцию подсчета НОД (GCD — Greatest Common Divisor):

```
int gcd(int ap, int bp)
{
  int c, a = ap, b = bp;
  while (b != 0) {
    c = a % b;
    a = b;
    b = c;
  }
  return a;
}
```

Этот алгоритм очень прост, его сложность в худшем случае, равна $O(\log N)$, где N — большее из чисел.

НОК можно искать исходя из соотношения НОД$(a, b)\times$НОК$(a, b) = a \times b$.

Существует еще один способ поиска НОД: бинарный алгоритм Евклида. Этот способ основывается на соотношениях НОД$(2 \times a, 2 \times b) = 2\times$НОД$(a, b)$, НОД$(2 \times a, 2 \times b + 1)$ =НОД$(a, 2 \times b + 1)$, НОД$(2 \times a + 1, 2 \times b + 1)$ =НОД$(2 \times (a - b), 2 \times b + 1)$. Хотя при ипользовании этих соотношений время написания программы и собственно поиска НОД несколько возрастет (хотя по прежнему сложность алгоритма будет составлять $O(\log n)$), этот способ намного удобнее при поиске НОД длинных чисел. Действительно, в обычном алгоритме Евклида необходима операция взятия остатка от деления длинных чисел, а в бинарном — намного более простые операции длинного вычитания и деления на 2.

В олимпиадных задачах довольно часто требуется найти что-либо «по модулю» какого-либо числа, т.е. подсчитать остаток от деления результата на заданное число. Теория чисел достаточно подробно изучается в школьном курсе математики и мы не будем углубляться в нее, взяв лишь несколько практически важных результатов.

В частности, нам важны следующие свойства остатков: $(a + b)\%n = (a\%n + b\%n)\%n$ и $(a \times b)\%n = (a\%n \times b\%n)\%n$. Эти свойства часто бывают полезными, т.к. обычно в задачах, где требуется подсчитать остатки, возникают довольно большие числа, и уменьшение размерности операндов в промежуточных вычислениях намного облегчает задачу.

Также довольно часто требуется найти удовлетворяющий условию элемент какой-либо последовательности по модулю данного числа. В этом случае можно составить

«таблицу умножения» по модулю n или другую таблицу с правилами перехода — это может значительно облегчить решение задачи.

1.7 Разложение числа на простые множители

Вернемся к простым числам. Любое число можно представить в виде произведения простых чисел, причем это представление будет единственно.

Обычно, такое представление выглядит в виде одномерного массива, содержащего простые числа, и еще одного одномерного массива для непосредственного представления числа, в каждой ячейке которого содержится степень соответственного просто числа.

Например, пусть у нас есть массив простых чисел $[2, 3, 5, 7, 11, 13]$, тогда массив с представлением числа 2600 будет выглядеть как $[3, 0, 2, 0, 0, 1]$ из которого можно получить $2^3 \times 5^2 \times 13^1 = 2600$.

Такое представление неоправданно генерировать для одного числа (если это не является необходимым условием для решения задачи). Однако, если чисел много, то приведение их в такое представление и обратно может быть весьма полезным.

Например, с помощью такого представления очень просто реализовать умножение. Рассмотрим наше число 2600 и число 11858, которое представляется массивом $[1, 0, 0, 2, 2, 0]$. Чтобы получить произведение этих чисел, достаточно сложить соответствующие элементы массивов. Результатом будет массив $[4, 0, 2, 2, 2, 1]$, т.е. $2^4 \times 5^2 \times 7^2 \times 11^2 \times 13^1 = 30830800$, что совпадает с результатом умножения.

После некоторых размышлений можно также реализовать деление с остатком, но в рамках лекции мы этого делать не будем, желающие могут сами придумать алгоритм.

Из этого представления также можно получить НОД и НОК этих чисел.

Для нахождения НОД надо выбирать минимум из степеней (это логично, т.к. число X^n кратно X^k, если $n \geq k$). Для подсчета НОК надо брать максимум из степеней. Этот же метод работает для подсчета НОД и НОК произвольного количества чисел.

Для наших чисел 2600 и 11858, НОД, подсчитанный таким образом, представим массивом $[1, 0, 0, 0, 0, 0]$, т.е. равен 2, а НОК — массивом $[3, 0, 2, 2, 2, 1]$ и равен 15415400. С помощью алгоритма Евклида несложно убедиться, что результат верен.

Кроме того, у разложения числа на простые множители существуют более специфичные назначения, которые встретятся по ходу решения практических туров.

Нахождение всех простых чисел до N занимает $O(N \times \sqrt{N})$ времени, подсчет степеней для одного числа занимает около $O(N)$ времени (если степени не слишком большие), и операции умножения, нахождения НОД и НОК занимают также $O(N)$ времени. Таким образом, суммарное время составляет около $O(N \times \sqrt{N} + N \times M)$, где N — максимальное из чисел, а M — количество этих чисел.

В некоторых случаях нам необходимо разложить только одно число на простые множители. В такой ситуации наиболее эффективным методом будет модификация функции проверки числа на простоту. Действительно, если каждый раз при нахождении делителя мы будем делить разлагаемое число на него до тех пор, пока оно делится (и запоминать степень, с которой этот делитель входит в разложение числа), то мы получим все простые делители кроме, возможно, одного. Действительно, поиск делителей необходимо осуществлять до квадратного корня из числа, а в случае, если после деления осталась не единица — этот остаток также является простым делителем, причем

степень его вхождения всегда равна 1. Например, 26 представляется как $2^1 \times 13^1$, где 13 — единственный делитель, больший квадратного корня.

1.8 Быстрое возведение в степень

Довольно часто возникает задача быстрого возведения числа или другого объекта, для которого определена операция умножения, в какую-либо степень. Наивный алгоритм, когда мы просто нужное число раз умножаем число на само себя, имеет сложность $O(N)$, где N — показатель степени.

При возведении в степень число растет очень быстро (а значит, наверняка требуются операции с длинными числами). Поэтому научиться возводить число в степень быстрее, чем за $O(N)$ — достаточно важная задача.

Рассматривая степени некоторых чисел, можно догадаться о методе, которым следует пользоваться. Например, для возведения 3 в 4-ю степень нам нужно проделать 3 операции умножения. Однако, если изменить порядок действий на такой: $(3^2)^2$, то потребуется всего два умножения. Следует помнить, что при возведении числа в степени еще в какую-либо степень показатели степеней перемножаются. Именно на сокращении четных степеней основывается идея быстрого возведения в степень.

Приведем текст функции, которая возводит число a в степень n:

```
int pow(int a, int n)
{
  int b, c, k;
  k = n;
  b = 1;
  c = a;
  while (k)
    if (!(k%2)) {
      k /= 2;
      c *= c;
    } else {
      k--;
      b *= c;
    }
  return b;
}
```

Каждый раз, когда степень четная, мы возводим вспомогательную переменную в квадрат, а показатель степени делим на 2. Если число нечетное, то умножаем текущее вспомогательное число на результат и уменьшаем показатель степени на 1. Таким образом хотя бы один раз из двух у нас произойдет уменьшение показателя степени вдвое и, исходя из этого, мы получим сложность алгоритма $O(\log N)$.

Достижение такой сложности очень полезно, особенно при использовании медленных операций с длинными числами. Таким образом можно выбрать «универсальный» рецепт: использовать быстрое возведение степень везде, где показатель может превышать 100. С учетом того, что функция достаточно простая, можно использовать быстрое возведение в степень во всех случаях.

В англоязычной литературе алгоритм быстрого возведения в степень обзывают забавным словосочетанием “Russian peasant algorithm”, что переводится как «алгоритм русского крестьянина».

1.9 Матрицы и операции над ними

В рамках нашей сегодняшней лекции мы будем рассматривать только квадратные матрицы, состоящие из чисел. Матрица является, по сути, двумерным массивом, чтобы наглядно представить, что это такое, можно открыть электронную таблицу. Каждый элемент матрицы определяется именем этой матрицы и двумя числами — индексами массива.

Двумерные массивы очень часто используются в программировании для хранения данных, но сегодня мы остановимся на алгебраических свойствах матриц.

Матрица является математическим объектом, и для нее, как и для чисел, определены некоторые операции, в частности, сложение и вычитание матриц. Допустим, что A, B и C — квадратные матрицы одинакового размера. Тогда матрица C, равная сумме матриц A и B определяется поэлементно, как $C[i][j] = A[i][j] + B[i][j]$. Точно так же определяется и разность матриц. Умножение матрицы на число — это просто умножение каждого элемента матрицы на это число.

По-другому происходит умножение матриц. Допустим, $C = A \times B$, тогда $C[i][j] = \sum_{k=0}^{n-1}(A[i][k] \times B[k][j])$, k изменяется от 0 до $n-1$ (n — размер матрицы, нумерация начинается с нуля). Знак $\sum$ означает сумму, т.е. $\sum_{k=0}^{4} A[k]$ — это то же самое, что $A[0] + A[1] + A[2] + A[3] + A[4]$. Обратите внимание, что $A \times B$, вообще говоря, не равно $B \times A$.

Перед написанием функции умножения матриц определим несколько макроподстановок.

```
#define MAXN 100
```

Эта макроподстановка будет определять размер матрицы. Удобно завести ее в виде константы, чтобы иметь возможность отладить программу (на отладчике очень тяжело просматривать большие двумерные массивы).

```
#define matr(a) int a[MAXN][MAXN]
```

Это можно использовать просто для удобства написания.

```
#define For(a,b) for(a=0;a<b;a++)
```

Во многих задачах используются циклы по какой-либо переменной от нуля до другой переменной. Очень удобно оформлять их в таком виде. Т.е. команда `For(i, j)` перед компиляцией заменится на `for(i=0; i<j; i++)`. Если привыкнуть к такому стилю, то он поможет экономить время, которое очень ценно на олимпиаде. Теперь перейдем непосредственно к реализации функции умножения матриц:

```
void mulmatr(matr(a), matr(b), matr(c))
{
  int i, j, k;
```

```
  For(i, MAXN)
    For(j, MAXN) {
      c[i][j] = 0;
      For(k, MAXN)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

Рассмотрим еще одно понятие, которое позволит нам применить хитрость, ускоряющее работу программы в несколько раз.

Транспонированной матрицей называется такая матрица, у которой столбцы становятся строками, а строки — столбцами. Т.е. $A^T[i][j] = A[j][i]$ (A^T - обозначение транспонированной матрицы).

В случае умножения матриц мы берем строку одной матрицы и столбец другой и осуществляем к ним последовательный доступ. В памяти компьютера многомерные массивы разворачиваются в одномерные, и доступ к строке идет довольно быстро, за счет оптимизаций компилятора при подсчете адресных выражений и кэширования (все данные лежат рядом и автоматически попадают в кэш). В случае же со столбцом адресное выражение (положение элемента массива в физической памяти компьютера) приходится вычислять заново и данные не попадают в кэш.

В нашей функции мы обращаемся к одному и тому же столбцу `MAXN` раз, за счет чего производительность резко падает. Если предварительно применить транспонирование матрицы, а потом вернуть ее обратно, то нам удастся ускорить программу в $3 - 5$ раз без использования дополнительной памяти. Измененная функция будет выглядеть так:

```
void mulmatr(matr(a), matr(b), matr(c))
{
  int i, j, k;
  For(i, MAXN)
    For(j, i) {
      k = b[i][j];
      b[i][j] = b[j][i];
      b[j][i] = k;
    }
  For(i, MAXN)
    For(j, MAXN) {
      c[i][j] = 0;
      For(k, MAXN)
        c[i][j] += a[i][k] * b[j][k];
    }
  For(i, MAXN)
    For(j, i) {
      k = b[i][j];
      b[i][j] = b[j][i];
      b[j][i] = k;
    }
}
```

В конце работы функции мы возвращаем матрицу B в ее исходное состояние. Теперь у нас есть только умножение строки на строку, которое происходит заметно быстрее. Можно было использовать и другие методы хранения столбца, например, записывая его в одномерный массив.

Кроме того, существует также операция умножения матрицы на вектор (одномерный массив). Пусть A — матрица, B - вектор из чисел, тогда в результате умножения мы получим вектор C, который будет определяться исходя из формулы $C[i] = B[i] \times \sum_{k=0}^{n-1} A[i][k]$.

Лекция 2

Битовые операции и структуры данных

Версия C от 16.11.2009

2.1 Битовые операции

Как известно, в компьютерах обычно применяется двоичное представление числа. Сейчас нас не будет интересовать представление числа в архитектуре **x86**, мы будем считать, что число записывается так, как это принято: слева стоят старшие разряды, справа — младшие.

Мы изучим следующие операции, комбинациями которых будем добиваться нужных результатов: отрицание (замена всех 0 на 1 и наоборот), или, и, исключающее или (поразрядное сложение по модулю 2). Операции записываются, соответственно, как `~`, `|`, `&`, `^`.

Кроме того, существует еще две операции, которые нам пригодятся: сдвиг влево и сдвиг вправо (с дополнением нулями справа и слева соответственно). Они обозначаются, как `<<` и `>>`.

Рассмотрим несколько примеров.

1. Установить k-ый справа бит числа n в 1. Разделим операцию на два этапа: создание числа с единственной 1 на k-ой позиции и логическое или с переменной n.

   ```
   j = 1;
   j <<= k;
   n |= j;
   ```

   В битовых операциях также можно использовать сокращенную запись операций.

2. Проверить, является ли k-ый бит переменной n единицей. Здесь все делается почти также.

   ```
   j = 1;
   j <<= k;
   j &= n;
   ```

Если j отлично от 0, то k-ый бит был 1, иначе 0.

3. Проверить, есть ли в двоичной записи числа n хотя бы один 0. Идея заключается в следующем: создадим переменную, полностью состоящую из единиц, а затем сравним с n.

```
j = 0;
j = ~j;
```

Если n и j совпадают, то ни одного нуля в записи n нет.

4. Установить k правых бит переменной n в нули. Для решения этой задачи воспользуемся применением операций сдвига вправо (при этом самые правые биты удалятся) и сдвига влево на то же количество элементов (эти позиции заполнятся нулями).

```
n >>= k;
n <<= k;
```

5. Дано n чисел, каждое из которых встречается в последовательности два или кратное двум число раз, кроме одного, которое встречается нечетное число раз. Необходимо найти это число.

   Для решения этой задачи следует воспользоваться следующими утверждениями: $x \wedge x \equiv 0$ («x исключающее или x тождественно равно нулю» или «x исключающее или x равно 0 для любого x») и $(x \wedge y) \wedge x \equiv y$. Исходя из этих соображений можно просто применить исключающее или («поксорить») все числа, и в результате останется искомое число, т.к. встречающееся четное количество раз числа сократятся.

```
k = 0;
For (i, n) {
  scanf("%d", &x);
  k ^= x;
}
```

Довольно частым применением битовых операций являются битовые булевские массивы (они занимают в 8 раз меньше памяти, чем обычные булевские массивы, т.к. для хранения значения «истина» или «ложь» достаточно одного бита).

Определим как константу максимальный размер массива. Мы будем использовать в качестве носителя тип `int`, который состоит из 32 бит (в современных компиляторах).

```
#define MAXN 1000
int bitarr[MAXN];
```

Таким образом, мы получим массив из 32000 бит.

Опишем три функции `set(n)` — установку n-го бита в 1, `unset(n)` — установку n-го бита в 0 и `get(n)`, которая будет возвращать значение n-го бита. Все эти операции мы уже рассматривали (установку бита в 0 и 1 и получение значения бита), поэтому приведем пример только одной из функций, например, `get(int n)`, а остальные несложно сделать по аналогии.

```
int get(int n)
{
  int seg, off, j = 1;
  seg = n / 32;
  off = n % 32;
  j <<= off;
  if ((bitarr[seg] & j) != 0) return 1;
  else return 0;
}
```

Здесь `seg` — индекс элемента в массиве, куда попадет данный бит, а `off` — номер бита в этом элементе.

Следует заметить, что такой метод работает медленнее, чем обычный булевский массив и должен использоваться только в случаях, если нам очень критична память или существует необходимость в какой либо специфичной проверке, например, найти хотя бы один 0 среди большого количества 1.

2.2 Стеки

Стеком называется структура данных, в которой данные, записанные первыми, извлекаются последними (FILO: First In - Last Out). Например, если мы записали в стек числа 1, 2, 3, то при последующем извлечении получим 3, 2, 1.

Удобно представить стек в виде узкого колодца или рюкзака, в который мы можем класть предмет только наверх и забирать только верхний предмет.

Мы будем реализовывать стек на одномерном массиве, а указателем на вершину стека (первый свободный элемент в массиве) в таком случае будет целочисленная переменная — индекс свободного элемента. Для стека определены две операции `push(x)` — записать в стек элемент (в нашем случае — число) и `pop()` — извлечь из стека элемент.

Размер стека, как и обычно, определим в виде константы:

`#define MAXN 1000`

Сам стек будем описывать в виде структуры.

```
typedef struct
{
  int sp;
  int val[MAXN];
} stack;
```

Мы можем создавать стеки, просто написав, например, `stack a, b;`.

При передаче параметров функции нам нужно будет указывать, с каким конкретно стеком мы хотим работать, а чтобы данные не копировались, будем передавать их по указателю.

```
void push(stack *s, int x)
{
  s->val[s->sp++] = x;
}
```

```
int pop(stack *s)
{
  return s->val[--s->sp];
}
```

В функции `push` мы записываем добавляемый элемент в первую свободную позицию, а затем увеличиваем ее номер (постинкремент). В функции pop мы уменьшаем указатель на вершину стека (предекремент), а затем возвращаем значение из последней занятой ячейки.

Для стеков созданных в статической памяти (так, как мы создавали их в примере), вызовы функций будут выглядеть как `push(&a, x); x = pop(&a);` где `x` — число, а `a` — стек. Перед этим необходимо инициализировать указатель на вершину стека нулем (`s.sp = 0`).

Сложность обеих операций над стеком составляет $O(1)$.

Если нам будут необходимы какие-то дополнительные функции работы со стеком, (например, определение пуст ли стек или количества элементов в нем), то всю необходимую информацию мы можем найти в поле `sp`. Напомним, что `sp` — текущее количество элементов в стеке.

При планировании размера стека надо учитывать не общее количество элементов, а максимальное количество одновременно находящихся в стеке элементов (хотя часто эти значения совпадают).

Стеки используются достаточно часто и в большинстве архитектур компьютеров реализованы аппаратно. Одно из применений стеков мы рассмотрим ниже.

2.3 Очереди

Очередь имеет интуитивно понятное название. Элемент, который попал в очередь раньше, выйдет из нее также раньше (т.е. элементы извлекаются в порядке поступления). По-английски очередь называется queue («кью») или FIFO (First In - First Out).

Очередь мы будем реализовывать на одномерном массиве, аналогично стеку. Тут нам придется немного отойти от аналогий с реальностью для повышения производительности. Если реализовывать очередь в программе как очередь в магазине, где люди постепенно двигаются к кассе, то извлечение элемента из очереди будет иметь сложность $O(N)$, где N - количество элементов в очереди), т.к. все элементы будет необходимо передвинуть. Гораздо удобнее в нашей ситуации перемещать «кассу», т.е. изменять указатель на начало очереди.

Однако в этом случае возникает другая проблема: если в предыдущем случае размер очереди ограничивался количеством одновременно находящихся в ней элементов, то здесь нам необходимо будет создавать очередь с размером, равным общему количеству элементов, которые в ней побывают.

Эту проблему мы решим, закольцевав очередь. Можно представить круг, где помечены две позиции — с какой уходить, и на какую становиться новому элементу. Для этого в реализации мы будем брать оба указателя по модулю $MAXN$.

Очередь также реализуем в виде структуры:

```
typedef struct
```

```
{
  int qh, qt;
  int val[MAXN];
} queue;
```

Теперь мы можем создавать очереди, просто написав `queue a, b;`/

Для очереди существует две операции: извлечь элемент из головы (head) очереди (`deq`) и добавить элемент в хвост (tail) очереди (`enq`).

```
void enq(queue *q, int x)
{
  q->val[(q->qt++)%MAXN] = x;
}

int deq(queue *q)
{
  return q->val[(q->qh++)%MAXN];
}
```

Как и в прошлый раз, необходимо передавать в функции указатель, т.е. их вызов должен выглядеть как: `enq(&a, x); x = deq(&a);` где `x` — число, а `a` — очередь. Так же, как и в случае со стеком, мы должны предварительно инициализировать оба указателя очереди нулями: `a.qt = 0; a.qh = 0;`

Признаком того, что очередь пуста или переполнилась, следует считать равенство полей `qt` и `qh`. Количество элементов в очереди определяется так: `qlen = (qt-qh)%MAXN`.

Очереди часто используются в качестве буферов и во многих устройствах реализованы аппаратно.

2.4 Деки

Деком (deque) называется структура, в которой добавление и извлечение элементов возможно с двух сторон. Т.е. это некоторая смесь стека и очереди.

Для дека можно использовать абсолютно ту же структуру, что для очереди, но функций будет уже 4 (добавление и извлечение в начало и в конец). Для такой структуры данных мы приведем просто исходный текст, все делается полностью аналогично стекам и очередям.

```
typedef struct
{
  int dh, dt;
  int val[MAXN];
} deque;

void push_front(deque *d, int x)
{
  if (d->dh < 1) d->dh += MAXN;
  d->val[(--d->dh)%MAXN] = x;
```

```
}

void push_back(deque *d, int x)
{
  d->val[(d->dt++)%MAXN] = x;
}

int pop_front(deque *d)
{
  return d->val[(d->dh++)%MAXN];
}

int pop_back(deque *d)
{
  if (d->dt < 1) d->dt += MAXN;
return d->val[(--d->dt)%MAXN];
}
```

Единственное усложнение состоит в том, что мы добавили проверку на то, чтобы указатели не становились отрицательными. Тогда определение количества элементов в деке будет выглядеть следующим образом: `dlen = a.dt>a.dh?(a.dt-a.dh)%MAXN:MAXN-(a.dh-a.dt)%`, что эквиваленно записи:

```
if (a.dt > a.dh) dlen = (a.dt-a.dh)%MAXN;
else dlen = MAXN - (a.dh-a.dt)%MAXN;
```

Здесь `a` — дек. Напомним, что перед использованием дека следует установить в ноль поля `dh` и `dt`.

2.5 Кучи

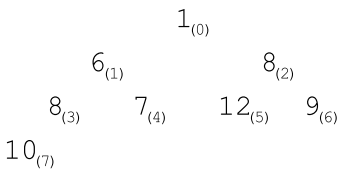


Рис. 2.1: Пример кучи

Куча (по-английски heap) — структура данных, которая может выдавать минимальный (или максимальный) элемент за $O(1)$, добавление нового элемента и удаление минимального элемента происходит за $O(\log N)$ (где N — количество элементов в куче). Другие операции над кучей не определены (хотя при необходимости могут быть введены, однако эффективность их будет невысокой, и они обязаны поддерживать свойства кучи).

Другое название кучи — очередь с приоритетами, что и отражает ее сущность.

Сразу перейдем к рассмотрению реализации кучи на одномерном массиве. Назовем элементы с индексами $i \times 2 + 1$ и $i \times 2 + 2$ потомками элемента с индексом i. Элемент i будет называться предком этих элементов. Несложно заметить, что потомки двух разных элементов не пересекаются, и каждый элемент, кроме нулевого, является чьим-либо потомком. Основное свойство кучи: каждый элемент не больше своих потомков.

Например, массив 1, 6, 8, 7, 12, 9, 10 может являться кучей (напомним, что индексация в массиве начинается с нуля).

Для хранения кучи создадим структуру, аналогичную предыдущим:

```
typedef struct
{
  int hs;
  int val[MAXN];
} heap;
```

Опишем три функции. В первую очередь напишем функцию, возвращающую наименьший элемент. Она будет очень простая, т.к. из свойства кучи следует, что минимум находится в нулевом элементе:

```
int get_min(heap *h)
{
  return h->val[0];
}
```

Перед использованием этой функции необходимо обязательно проверить, что куча не пуста!

Следующей определим функцию добавления элемента в кучу. Новый элемент будем добавлять в конец кучи, а затем обменивать его с предком, пока он меньше, чем его предок или мы не достигли нулевого индекса. При этом свойство кучи не нарушится.

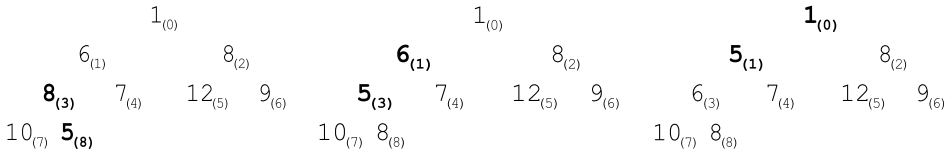


Рис. 2.2: Добавление в кучу (сравнивыемые элементы выделены)

```
void add_heap(heap *h, int x)
{
  int y, pos=h->hs, npos;
  h->val[h->hs++] = x;
  npos=(pos-1)/2;
  while (pos && h->val[pos] < h->val[npos]) {
    y=h->val[pos];
    h->val[pos]=h->val[npos];
    h->val[npos] = y;
    pos=npos;
    npos=(pos-1)/2;
  }
}
```

Как уже написано выше, сложность добавления элемента в кучу составляет $O(\log N)$ — каждый раз индекс текущего элемента уменьшается вдвое.

Кроме того, часто возникает задача удаления минимального элемента. Мы будем реализовывать это следующим образом: запишем на место нулевого элемента последний, уменьшим размер кучи на 1 и просеем нулевой элемент по куче так, чтобы сохранилось свойство кучи. Будем реализовывать просеивание следующим образом: если элемент больше, чем меньший из своих потомков, то меняем их местами и продолжаем процесс, пока выполнено условие или мы не вышли за пределы кучи. В реализации этого алгоритма применена одна хитрость, которая будет пояснена ниже.

```
void del_heap(heap *h)
{
  int minp, pos=0, y;
  h->val[0]=h->val[--h->hs];
  while (pos*2+1 < h->hs) {
    y=pos*2+1;
    minp=h->val[y]<h->val[y+1]?y:y+1;
    if (h->val[pos]>h->val[minp]) {
      y=h->val[pos];
      h->val[pos]=h->val[minp];
      h->val[minp]=y;
      pos=minp;
    } else break;
  }
}
```

На первый взгляд, единственный случай, где теоретически возможна ошибка, когда у нас имеется всего один потомок (т.е. `pos*2+2 == h->hs`). Такой вариант возможен, если количество элементов в куче четно.

В этом случае мы выбираем минимум из первого потомка и первого элемента вне кучи, что, казалось бы, является грубой ошибкой. Но мы знаем, что просеиваемое число и первое число вне кучи, равны. Это гарантирует нам, что обмена с элементом вне кучи не произойдет, а если необходим обмен с единственным потомком, то он будет выполнен корректно.

2.6 Динамически расширяемые массивы

До сих пор мы использовали статические массивы, их размер был определен заранее и не мог изменяться в процессе работы программы. Это обычный и правильный метод, отходить от которого стоит только в некоторых ситуациях. Например, когда мы имеем много массивов, которые могут быть произвольной длины, и известно только ограничение на сумму их длин или длинная арифметика (в этом случае необходимо хранить цифры в обратном порядке, прижатыми к левому краю, а не в прямом порядке, как об этом говорилось в первой лекции).

Вначале мы выделяем какое-то количество памяти под массив, а затем, по мере необходимости, расширяем массив. Если пользоваться наивным методом, т.е. увеличивать массив на 1 элемент каждый раз, когда мы вышли за текущий размер, то производительность будет очень мала. Операция расширения массива требует некоторых накладных расходов, связанных с работой ОС по перераспределению памяти, а в неудачном

случае требуется еще и копирование всего содержимого массива в новую область памяти. Это связано с тем, что массив в языке Си обязан быть непрерывным, а расширить его на существующей памяти не всегда возможно.

Мы будем реализовывать компромиссный по времени и требуемой памяти вариант, он будет выполнять $\log N$ выделений памяти (где N — количество элементов в массиве), а неиспользованной останется не больше половины памяти, выделенной под массив.

Вначале мы создадим массив некоторого начального размера, а когда количество использованных элементов будет приближаться к текущему размеру — будем увеличивать размер массива вдвое. Это и даст нам требуемую сложность в $\log N$ выделений памяти, а половина неиспользованной памяти возникнет в случае, если мы прекратили добавление элементов сразу после очередного расширения массива.

Для использования функций выделения памяти в языке **Си** мы должны подключить библиотеку `stdlib.h`.

Допустим, перед нами стоит задача считать неизвестное заранее количество чисел до конца файла. Это можно реализовать с помощью следующего фрагмента программы:

```
int now=0, size=2, i, *a;
a=(int*)malloc(sizeof(int)*size);
while (scanf("%d", &a[now++]) == 1)
  if (now >= size-1) {
    size*=2;
    a=(int*)realloc(a, sizeof(int)*size);
  }
```

В конце программы (или когда массив перестал быть нам нужным), необходимо освободить выделенную память. Это делается с помощью функции `free(a);`

В общем случае, динамически расширяемые массивы являются довольно неплохим средством хранения неизвестного заранее количества данных с относительно небольшими накладными расходами. В динамически расширяемых массивах можно делать все то же самое, что и в обычных массивах, это делает их весьма привлекательными.

Кроме того, по необходимости, размер массива можно уменьшать той же самой функцией `realloc`; можно уменьшать размер массива вдвое, если из него происходит удаление (логично делать это в случае, если `now*2 < size`). Однако, чтобы избежать слишком частого изменения размеров массива, когда он почти заполнен, можно поставить условие на уменьшение `now*4 < size`, но размер массива по-прежнему уменьшать вдвое.

2.7 Списки

Рассмотрим еще одну динамическую структуру данных, называемую связным списком. Каждый элемент списка представляет собой структуру, одно поле которой содержит информацию (ключ), а другое — ссылку на следующий элемент. Существуют также двусвязные списки, в которых хранится ссылка не только на следующий элемент, но и на предыдущий. Начинается список с указателя на элемент списка. В целом его вид можно представить следующим образом:

Один элемент списка будем описывать с помощью следующей структуры:

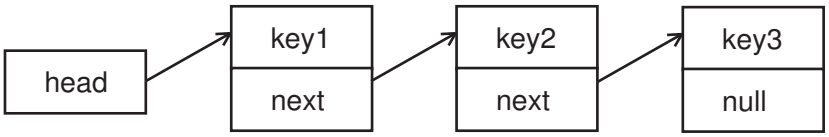


Рис. 2.3: Общий вид списка

```
typedef struct
{
  int key;
  struct _list *next;
} list;
```

Чтобы работать со списком, необходимы указатели на элемент, которые заводятся следующим образом: `list *head=NULL, *temp;` Довольно необычно происходит операция добавления в список: чтобы добиться сложности $O(1)$, добавление происходит в начало списка. Это реализуется следующим фрагментом кода:

```
  temp = (list*)malloc(sizeof(list));
  temp->key = newkey;
  temp->next = head;
  head = temp;
```

Первая строка выделяет память под новый элемент, вторая записывает новое значение ключа, остальные поясним рисунком.

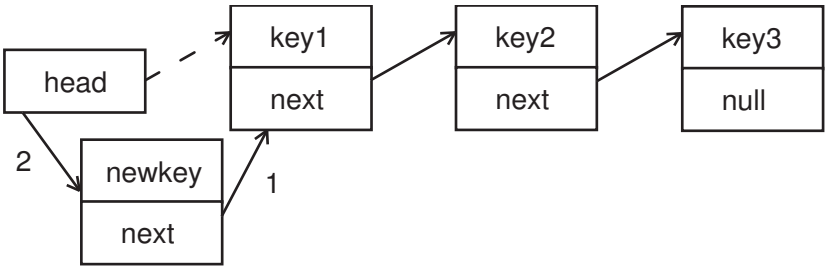


Рис. 2.4: Добавление элемента в начало списка

Аналогично осуществляется вставка после элемента, на который имеется ссылка, достаточно заменить `head` в нашем коде на его поле `next`.

Поиск элемента по ключу осуществляется за $O(N)$ — нам необходимо пройти весь список. Напишем функцию, которая возвращает указатель на элемент по его значению ключа или `NULL`, если элемента с таким ключом не существует.

```
list* find(list* head, int key)
{
  list *now=head;
  while (now != NULL) {
```

```
    if (key == now->key) break;
    now = now->next;
  }
  return now;
}
```

Однако, удаление элемента невозможно реализовать только зная ссылку на него (т.к. необходимо, чтобы список остался связным, а удаление элемента создаст в нем «дырку»). Напишем отдельную функцию удаления элемента с заданным ключом и возвращающую ссылку на новый список (без этого элемента).

```
list* del(list *head, int key)
{
  list *now=head, *prev;
  if (key == head->key) {
    head = head->next;
    free(now);
  } else {
    prev = now;
    now = now->next;
    if (key == now->key) {
      prev->next = now->next;
      free(now);
    }
  }
  return head;
}
```

Здесь мы отдельно рассматриваем случай, когда необходимо удалить первый элемент списка, т.к. он не имеет предыдущего. В противном случае мы делаем удаление согласно рисунку:

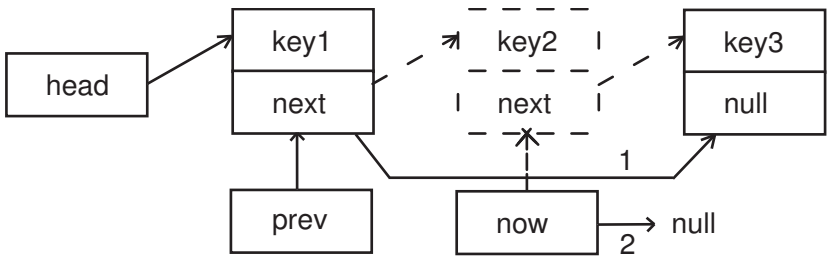


Рис. 2.5: Удаление элемента из списка

Над списком можно определить множество различных функций (например, объединение двух списков, удаление элементов, обладающих каким-либо признаком и т.д.), но они все базируются на изложенных выше идеях.

2.8 Сравнение динамических структур

Мы изучили две структуры в динамической памяти: динамически расширяемый массив (вообще говоря, массив изменяемого размера, т.к. он может и уменьшаться) и списки. Во-первых, оценим требования к памяти. Динамически расширяемый массив может иметь такое же количество пустых элементов, как и непустых, т.е. его требование к памяти, в худшем случае, записывается как `2*N*sizeof(key)`, для списка это требование записывается в общем случае как `N*(sizeof(key)+sizeof(*key))`. Здесь `N` — количество элементов в структуре, `sizeof(key)` — размер ключа, `sizeof(*key)` — размер указателя (обычно 4 байта). Так для ключа типа `int` (4 байта) худший случай динамически расширяемого массива совпадает с обычным случаем списка.

Производительность по времени работы измерялась в тиках (`GetTickCount`) на 10^7 элементах:

	Создание	**Поиск**	**Удаление**	**Доступ**
Массив	422	31	78	0
Список	4688	63	63	63

В обоих случаях структуры заполнялись последовательными числами от 0 до $10^7 - 1$, поиск, удаление и доступ по индексу осуществлялись к элементу с номером $10^7/2$. Для заполнения, поиска и удаления элемента в списке использовались приведенные выше функции, при удалении элемента в середине массива осуществлялся сдвиг конца массива («дырка» не образовывалась).

Оставим значения в таблице без комментариев, и в дальнейшем будем использовать списки в задачах, где критична производительность (большое количество элементов в списках) только в случае крайней необходимости. Крайняя необходимость возникает, когда происходит много вставок (или удалений) в средину и, особенно, если позиция для вставки (удаления) не сильно отличается от текущего положения.

2.9 Хеш-таблицы

Допустим, перед нами стоит следующая задача: нам дается множество ключей (уникальных значений) и требуется уметь быстро проверять, входит ли ключ в наше множество. При этом множество ключей может изменяться, т.е. ключи могут добавляться и исключаться из множества.

Сейчас мы будем рассматривать все на числовых примерах. Допустим, нам дан набор целых чисел от 1 до 10000, а затем идет серия запросов вида «есть ли число X в множестве?». Мы можем создать булевский массив, в котором будем помечать, встречалось данное число или нет. При этом сложность одного запроса будет $O(1)$.

Если же чисел больше, то мы можем попробовать использовать для хранения признака наличия числа во множестве не 1 байт, а 1 бит, пользуясь битовыми функциями. Это даст нам возможность увеличить максимальный размер числа в 8 раз. Но и этого может не хватить. Например, если числа изменяются от 0 до 2^{31}, то такую таблицу невозможно разместить в памяти, которая дается нашему решению.

Эта задача имеет решение в некоторых частных случаях. Например, пусть максимальный размер множества равен 1000, а каждый элемент может быть в пределах от 0

до 2^{31}. Если мы будем создавать таблицу размером 2^{31} элементов, то будет использована меньше ее одной миллионной части.

Будем решать такой класс задач (когда количество чисел намного меньше максимального значения) с помощью так называемых хеш-таблиц.

Введем понятие хеш-функции, как функции, отображающей множество ключей (в нашем случае чисел от 0 до 2^{31}) в меньшее множество ключей (соизмеримое с максимальным количеством элементов — в нашем случае с 1000). Хеш-функция должна обладать двумя основными свойствами: быть быстрой и равномерно генерировать ключи (т.е. чтобы одному и тому же ключу в малом множестве соответствовало примерно равное количество ключей в большом множестве). Иногда от хеш-функции требуют неустойчивости (т.е. чтобы при близких значениях ключей большого множества она генерировала сильно отличающиеся ключи малого множества).

Размер таблицы (вообще говоря, одномерного массива) для эффективной работы должен быть больше, чем удвоенное количество элементов малого множества. Обозначим размер таблицы за N.

Будем использовать в качестве хеш-функции операцию взятия остатка от деления числа большого множества на `N` (`X % N`). Это достаточно хорошая функция: считается относительно быстро и распределена равномерно. Итак, мы считаем остаток от деления нашего числа `X` на `N` и записываем `X` в ячейку с индексом `X % N`. Затем, при проверке числа `Y`, мы просто смотрим на ячейку `Y % N` и, если число `Y` находится там, то возвращаем признак наличия. Сложность опять получается $O(1)$.

Однако возникает проблема — несколько чисел могут иметь одинаковый остаток от деления на N. Такая ситуация называется «коллизией». Существует несколько способов разрешения коллизий, мы рассмотрим способ со списками, каждый из которых соответствует одному значению остатка (одной ячейке хеш-таблицы).

Допустим, размер нашей хеш-таблицы равен 8 и в нее были внесены элементы $6, 19, 27, 11, 16, 22$. Тогда она будет выглядеть как на рисунке.

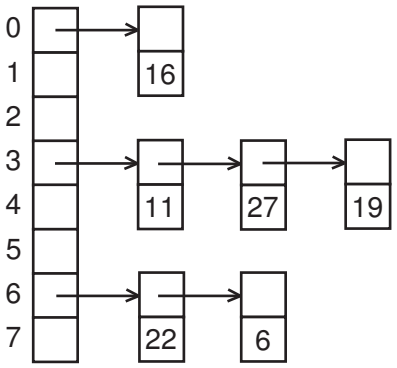


Рис. 2.6: Пример хеш-таблицы

Использование списков в данном случае уместно, т.к. количество элементов в каждом списке будет небольшим. Можно использовать и динамически расширяемые массивы, если значений в хеш-таблице достаточно много. Теперь для добавления элемента нам надо подсчитать его хеш-функцию и поместить в соответствующий этому значению список. Для проверки принадлежности элемента множеству нам также надо посчитать его хеш-функцию и попытаться найти его в нашем списке.

В среднем такая хеш-таблица будет работать за $O(1)$, т.к. коллизии будут встречаться редко. Ее можно «завалить» по времени только в случае, если точно знать размер таблицы, а при использовании набора тестов для проверки задачи это невозможно. Поэтому следует выбирать произвольный размер, больший $2 \times N$, где N — максимальное количество элементов, одновременно находящихся в хеш-таблице.

Функции работы с хеш-таблицей можно реализовать так:

```
typedef struct {
```

```
  struct -node *next;
  int data;
} node;

node **hashTable;
int hashTableSize;

int hash(int data) {
  return (data % hashTableSize);
}

node *insertnode(int data) {
  node *p, *p0;
  int bucket;
  bucket = hash(data);
  p = (node*) malloc(sizeof(node));
  p0 = hashTable[bucket];
  hashTable[bucket] = p;
  p->next = p0;
  p->data = data;
  return p;
}

void deletenode(int data) {
  node *p0, *p;
  int bucket;
  p0 = 0;
  bucket = hash(data);
  p = hashTable[bucket];
  while (p && (p->data != data)) {
    p0 = p;
    p = p->next;
  }
  if (!p) return;
  if (p0) p0->next = p->next;
  else hashTable[bucket] = p->next;
  free (p);
}

node *findnode (int data) {
  node *p;
  p = hashTable[hash(data)];
  while (p && (p->data != data)) p = p->next;
  return p;
}
```

Перед работой необходимо создать хеш-таблицу в динамической памяти и устано-

вить ее размер. Например:

```
hashTableSize = 20001;
hashTable = (node**) malloc(hashTableSize*sizeof(node*));
```

Кроме того, хеш-функции могут использоваться и в других ситуациях, например при сравнении сложных объектов. Мы заранее считаем хеш-функцию от каждого объекта, а затем, при сравнении, в первую очередь сравниваем значения хеш-функция и проводим сравнение для объектов полностью только в случае совпадения хешей.

Например, простейший способ подсчитать хеш-функцию от строки — сложить коды всех символов, входящих в строку. Такая хеш-функция будет генерировать одинаковые значения для строк, которые содержат одинаковые буквы, независимо от порядка.

Существуют и другие методы подсчета хеш-функции от строки, например подсчет полинома по какому-либо модулю.

Хеш-функция от строки обычно должна хорошо пересчитываться при удалении первого символа строки и добавлении нового символа — это позволяет использовать такое хеширование при поиске подстроки в строке (метод Рабина-Карпа).

В принципе, хеш-функция может быть введена для любых объектов.

2.10 Механизм запуска функций и рекурсия

Рассмотрим механизм запуска функции в программе.

При вызове функции необходимо:

- сохранить текущие значения регистров процессора
- запомнить «точку возврата», т.е. то место, куда мы должны вернуться после выполнения функции (вообще говоря, это тоже специальные регистры процессора)
- передать параметры в функцию
- выделить место под локальные переменные. Сам код функции хранится отдельно в оперативной памяти

Все эти данные помещаются в стек. Таким образом, он имеет следующий вид:

Свободная память
Локальные переменные функции
Параметры функции
Значения регистров процессора и адрес точки возврата
Данные, которые были в стеке до вызова функции

Как видно, для вызова функции требуется достаточно много накладных расходов, поэтому короткие функции следует оформлять в виде макросов или делать их inline (встраиваемыми), что является, по сути, тем же макросом в «осовремененном» виде.

Перейдем теперь к рекурсивным вызовам функции. Условимся разделять «функцию» (машинный код команд функции) и «экземпляр функции» (совокупность содержимого области стека для этой функции вместе с машинным кодом).

Рассмотрим для примера следующую простую рекурсивную функцию (слева заданы номера строк):

```
1  void rec(int n) {
2    if  (n > 0)
3      rec(п-1);
4    printf("%d  ",  n);
5  }
6
7  int  main(void) {
8    rec (2) ;
9    return 0;
10 }
```

Первый вызов функции произойдет на строке 8. Содержимое стека при этом будет таким (содержимое активного экземпляра выделено).

Параметр функции n = 2
Возврат в функцию main после строки 8

Затем происходит следующий вызов функции из экземпляра 1 в строке 3. Содержимое стека будет выглядеть так:

Параметр функции n = 1
Возврат в первый экземпляр функции rec после строки 3
Параметр функции n = 2
Возврат в функцию main после строки 8

Затем еще один вызов и создание нового экземпляра:

Параметр функции n = 0
Возврат во второй экземпляр функции rec после строки 3
Параметр функции n = 1
Возврат в первый экземпляр функции rec после строки 3
Параметр функции n = 2
Возврат в функцию main после строки 8

Поскольку параметр стал равен 0, то следующего вызова не произойдет. Будет выведено значение параметра (т.е. число 0) и произойдет выход из функции (переход к точке возврата и удаление из стека всех данных этого экземпляра функции). Т.е. стек снова примет следующий вид:

Параметр функции n = 1
Возврат в первый экземпляр функции rec после строки 3
Параметр функции n = 2
Возврат в функцию main после строки 8

После третьей строки второго экземпляра функции идет только вывод параметра (т.е. выводится число 1). Больше никаких действий в этом экземпляре не происходит, а значит происходит возврат в первый экземпляр функции после третьей строки и стек имеет следующий вид:

Параметр функции n = 2
Возврат в функцию main после строки 8

Первый экземпляр функции после третьей строки выводит свой параметр (2) и заканчивает работу — происходит возврат в функцию main, которая также заканчивает свою работу.

Вывод программы будет 0 1 2.

Лекция 3

Алгоритмы поиска

Версия C от 16.11.2009

3.1 Поиск в неупорядоченных массивах

Самым простым вариантом поиска можно считать поиск элемента в одномерном неупорядоченном массиве. Сформулируем задачу следующим образом: дан одномерный неупорядоченный массив, состоящий из целых чисел, и необходимо проверить, содержится ли данное число в этом массиве.

Пусть массив называется a и состоит из n элементов, а искомое число равно k. Тогда код, осуществляющий поиск, можно записать так:

```
int j = -1;
for (i=0; i < n; i++)
  if (a[i] ==  k) j = i;
```

В случае если число k ни разу не встречалось в массиве, j будет равно -1. Приведенная выше функция будет искать последнее вхождение числа k в массиве a. Если нам необходимо искать первое вхождение, то вместо присваивания $j = i$ следует добавить оператор `break;` (в этом случае искомый индекс будет храниться в переменной i).

И в том и в другом случае алгоритм будет иметь сложность $O(N)$.

На этом примере можно рассмотреть «барьерный» метод, который может быть полезен в очень многих задачах. Для использования барьерного метода наш массив должен иметь один дополнительный элемент (т.е. его длина должна быть не меньше, чем $n + 1$ элемент). Отметим, что таким способом можно искать только первое вхождение элемента:

```
a[n+1] = k;
for (i=0; a[i] != k; i++);
```

Если элемент k встречается в массиве, то его индекс будет находиться в переменной i, если же такой элемент в массиве не встречается, то i будет равно $n + 1$.

Рассмотрим отдельно задачу поиска минимума и максимума в массиве. Так же как и при поиске вхождения элемента, будем искать не само значения минимума или максимума, а индекс минимального (максимального) элемента. Это избавит нас от многих проблем и позволит совершать меньшее количество ошибок при программировании. Поиск минимального элемента в массиве a будет выглядеть следующим образом:

```
imin = 0;
For (i, n)
  if (a[i] < a[imin]) imin = i;
```

Индекс минимального элемента будет храниться в переменной $imin$, а сам минимум равен $a[imin]$. Минимум и максимум следует обязательно искать по индексу, а не по значению. Например, если мы будем пытаться хранить непосредственно значение минимума или максимума, то можем легко ошибиться с начальной инициализацией. Например, для массива вещественных чисел определить значения, которыми изначально следует инициализировать минимум и максимум.

Теперь рассмотрим задачу поиска минимума и максимума одновременно. Можно реализовать такой поиск аналогично:

```
imin = 0;
imax = 0;
for (i=1; i<n; i++)
{
  if (a[i] < a[imin]) imin = i;
  if (a[i] > a[imax]) imax = i;
}
```

Такая реализация требует $2 \times N - 2$ сравнения. Но эту задачу можно решить и за меньшее количество сравнений. Разобьем все элементы на пары, и будем искать в каждой паре минимум и максимум ($N/2$ сравнений), затем минимум будем искать только среди минимальных элементов пар, а максимум — среди максимальных. Общее количество сравнений будет около $3 \times N/2$ (проблема возникает, когда количество элементов нечетное — один из элементов остается без пары). Точно это можно записать как $\lceil 3 \times N/2 \rceil - 2$, где $\lceil\ \rceil$ — округление до большего целого.

Рассмотрим еще один способ поиска максимума. После разбиения элементов на пары будем продолжать этот процесс, аналогично турниру «на вылет». Т.е. заново разобьем максимальные элементы из пар на пары и снова найдем максимум и т.д. Для поиска максимального элемента будет по-прежнему требовать $N - 1$ операция сравнения, но сам максимальный элемент будет участвовать только в $\log N$ сравнениях. И одно из этих сравнений обязательно будет со вторым по величине элементом. Таким образом, для поиска второго по величине элемента будет требоваться $\lceil \log N \rceil - 1$ сравнение (при условии, что все сравнения для максимального элемента проведены).

Обычно такие методы используются в особых случаях, когда это непосредственно требуется в решении задачи. Для общего случая подходят более простые методы, где количество сравнений не играет такой важной роли.

Однако и этот метод может быть полезен при поиске «порядковых статистик» массива. k-ой порядковой статистикой массива называется k-ый по счету элемент этого массива (т.е. если массив отсортировать по неубыванию, то k-ая порядковая статистика — это элемент, стоящий на k-ой позиции).

3.2 Поиск порядковых статистик

Как известно, существуют методы сортировки массива за $O(N \log N)$. Для поиска k-ой порядковой статистики можно отсортировать массив и вывести k-ый элемент. Но

в этом случае мы совершаем множество лишних действий, ведь с помощью сортировки мы найдем все порядковые статистики, а не только k-ую.

Приведенный ниже алгоритм работает за $O(N)$. Существует алгоритм поиска i-ой порядковой статистики за $O(N)$ в худшем случае, но он тяжел в реализации и имеет большую константу.

Схема алгоритма имеет следующий вид. Пусть k — номер искомой порядковой статистики, l (это маленькая латинская L) и r — текущие левая и правая границы области массива a, в которой мы ищем k-ую статистику. Если `l == r`, то область поиска ограничена одним элементом, т.е. k-ая порядковая статистика равна `a[r]`.

На каждом шаге будем выбирать число `s = (l + r) / 2`. Вообще говоря, мы можем выбирать произвольное число из интервала $[l, r]$, но генерация случайного числа занимает достаточно много времени, поэтому лучше использовать какое-либо фиксированное число. Расположим элементы массива интервала $[l, r]$ так, чтобы сначала шли все элементы, не превосходящие `a[s]`, а затем все остальные. Первый элемент второй группы обозначим за j. Тогда если $k \leq j$, то будем продолжать поиск с неизменным значением l и $r = j$ (в левой части массива), иначе будем осуществлять поиск при $l = i$ и неизмененным r (в правой части массива).

Сначала запишем функцию, осуществляющую такой поиск, а затем приведем пример и необходимые пояснения:

```
int search(int *a, int k, int l, int r)
{
  int m, i=l, j=r, tmp;
  if (l == r) return a[r];
  m = a[(l + r) / 2];
  while (i < j) {
    while (a[i] < m) i++;
    while (a[j] > m) j--;
    if (i < j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++; j--;
    }
  }
  if (k < j) return search(a, k, l, j);
  if (i < r) return search(a, k, i, r);
  return a[k];
}
```

Вызывать функцию следует так: `search(a, k, 0, n-1)`, где a — массив, k — номер статистики, которую надо получить, а n — количество элементов в массиве. После выполнения этой функции искомый элемент будет находится на своем месте, т.е. ответ на задачу будет содержаться в элементе $a[k]$.

Перестановку элементов мы осуществляем следующим образом: находим первый «неправильный» элемент слева, затем первый неправильный элемент «справа», а в случае, если указатели не сошлись, осуществляем обмен этих ячеек массива.

Приведем пример для массива $\{2, 7, 8, 6, 0, 4, 1, 9, 3, 5\}$ и $k = 9$:

1. $\{2, 7, 8, 6, 0, 4, 1, 9, 3, 5\}, l = 0, r = 9, m = 0$
2. $\{0, 7, 8, 6, 2, 4, 1, 9, 3, 5\}, l = 1, r = 9, m = 4$
3. $\{0, 3, 1, 4, 2, 6, 8, 9, 7, 5\}, l = 5, r = 9, m = 9$
4. $\{0, 3, 1, 4, 2, 6, 8, 5, 7, 9\}, l = 9, r = 9, m = 9$

На поиск максимального элемента нам потребовалось 4 вызова функции `search`.

Доказательство сложности $O(N)$ опирается на суммирование ряда, в котором i-ый элемент равен $N/2^i$. Методы доказательства сходимости рядов изучаются в школьном или университетском курсе математического анализа.

Кроме того, что поиск k-ой порядковой статистики ставит k-ый элемент на свое место, существует еще одно его полезное применение. А именно, с помощью поиска k-ой порядковой статистики можно выделить k наименьших чисел массива — они будут находится в элементах с индексами от 0 до k, но не будут упорядочены.

3.3 Бинарный поиск в упорядоченных массивах

Под упорядоченным массивом будем понимать массив, упорядоченный по неубыванию, т.е. $a[1] \leq a[2] \leq \ldots \leq a[N]$.

У нас имеется заданная своими границами область поиска. Мы выбираем ее середину, и, если искомый элемент меньше, чем средний, то поиск осуществляется в левой части, иначе — в правой. Действительно, если искомый элемент меньше среднего, то и меньше всех элементов, которые находятся правее среднего, а значит, их сразу можно исключить из рассмотрения. Аналогично для случая, когда искомый элемент больше среднего.

Код, осуществляющий бинарный поиск в упорядоченном массиве выглядит так:

```
while (l<r) {
  m=(l+r)/2;
  if (a[m]<k) l=m+1;
  else r=m;
}
if (a[r]==k) printf("%d", r);
else printf("-1");
```

Перед выполнением этого кода следует присвоить переменным l и r значения 0 и $n - 1$ соответственно. В случае если элемент не найдем, эта программа выводит -1.

Сложность алгоритма бинарного поиска составляет $O(\log N)$, где N — количество элементов в массиве.

3.4 Бинарный поиск для монотонных функций

Бинарный поиск может использоваться не только для поиска элементов в массиве, но и для поиска корней уравнений и значений монотонных (возрастающих или убывающих) функций. Напомним, что функция называется возрастающей, если $\forall x_1, x_2 : x_1 > x_2 \Rightarrow f(x_1) > f(x_2)$ (для любых x_1 и x_2, если $x_1 > x_2$, то $f(x_1)$ также больше $f(x_2)$).

Действительно, так же как и в массиве, мы можем исключить из рассмотрения половину текущей области, если нам заведомо известно, что там не существует решения. В случае же, если функция не монотонна, то воспользоваться бинарным поиском нельзя, т.к. он может выдавать неправильный ответ, либо находить не все ответы.

Для примера рассмотрим задачу поиска кубического корня. Кубическим корнем из числа x (обозначается $\sqrt[3]{x}$) называется такое число y, что $y^3 = x$.

Сформулируем задачу так: для данного вещественного числа x ($x \geq 1$) найти кубический корень с точностью не менее 5 знаков после точки.

Функция при $x \geq 1$, ограничена сверху числом x, а снизу — единицей. Таким образом, за нижнюю границу мы выбираем 1, за верхнюю — само число x. После этого делим текущий отрезок пополам, возводим середину в куб и если куб больше x, то заменяем верхнюю грань, иначе — нижнюю.

Код будет выглядеть следующим образом:

```
r = x;
l = 1;
while (fabs(l-r)>eps) {
  m=(l+r)/2;
  if (m*m*m<x) l=m;
  else r=m;
}
```

Для того чтобы пользоваться функцией `fabs`, необходимо подключить библиотеку `math.h`.

3.5 Бинарный поиск по ответу

Во многих задачах в качестве ответа необходимо вывести какое-либо число. При этом достаточно легко сказать, больше ли это число, чем нужно, или меньше, несмотря на то, что вычисление самого ответа может быть довольно трудоемкой операцией. В таком случае мы можем выбрать число заведомо меньшее ответа и число заведомо большее ответа, а правильное решение искать бинарным поиском.

Для примера рассмотрим решение задач нескольких прошедших олимпиад.

Очень легкая задача

Московская олимпиада по информатике 2006-2007

Сегодня утром жюри решило добавить в вариант олимпиады еще одну, Очень Легкую Задачу. Ответственный секретарь Оргкомитета напечатал ее условие в одном экземпляре, и теперь ему нужно до начала олимпиады успеть сделать еще N копий. В его распоряжении имеются два ксерокса, один из которых копирует лист за секунд, а другой — за y. (Разрешается использовать как один ксерокс, так и оба одновременно.

Можно копировать не только с оригинала, но и с копии.) Помогите ему выяснить, какое минимальное время для этого потребуется.

Формат входных данных

Во входном файле записаны три натуральных числа N, x и y, разделенные пробелом ($1 \le N \le 2 \times 10^8$, $1 \le x, y \le 10$).

Формат выходных данных

Выведите одно число — минимальное время в секундах, необходимое для получения N копий.

Примеры

Входные данные	Выходные данные
4 1 1	3
5 1 2	4

Существует конструктивное решение этой задачи (формула), которую можно вывести и, при желании, доказать. Однако очень легко реализовать решение этой задачи с помощью бинарного поиска.

Первую страницу мы копируем за $min(x, y)$ секунд и, затем, рассматриваем решение уже для $N - 1$ страницы.

Пусть l - минимальное время, r - максимальное. Минимум нам необходимо потратить 0 секунд, максимум, например $(N - 1) \times x$ секунд (страницы делаются полностью на одном ксероксе). Считаем среднее значение и смотрим, сколько полных страниц можно напечатать за это время, используя оба ксерокса. Если количество страниц меньше $N - 1$, то мы меняем нижнюю границу, иначе — верхнюю.

```
int main(void)
{
  int n, x, y, i, j, l, r, now;
  double speed;
  scanf("%d%d%d", &n, &i, &j);
  x=i<j?i:j;
  y=i>j?i:j;
  l=0;
  r = (n-1)*y;
  while (l != r) {
    now = (l+r)/2;
    j = now / x + now / y;
    if (j < n-1) l = now+1;
    else r = now;
  }
  printf("%d", r+x);
  return 0;
}
```

Автобус

13 Украинская олимпиада

Служебный автобус совершает один рейс по установленному маршруту и в случае наличия свободных мест подбирает рабочих, которые ожидают на остановках, и отвозит их на завод. Автобус также может ждать на остановке рабочих, которые еще не пришли.

Известно время прихода каждого рабочего на свою остановку и время проезда автобуса от каждой остановки до следующей. Автобус приходит на первую остановку в нулевой момент времени. Продолжительность посадки рабочих в автобус считается нулевой.

Задание: Написать программу, которая определит минимальное время, за которое автобус привезет максимально возможное количество рабочих.

Формат входных данных

Входной текстовый файл в первой строке содержит количество остановок N и количество мест в автобусе M. Каждая i-я строчка из последующих N строчек содержит целое число — время движения от остановки i к остановке $i+1$ ($N+1$-я остановка — завод), количество рабочих K, которые придут на i-ю остановку, и время прихода каждого рабочего на эту остановку в порядке прихода ($1 \leq M \leq 2000, 1 \leq N, K \leq 200000$).

Формат выходных данных

Единственная строка выходного текстового файла должен содержать минимальное время, необходимое для перевозки максимального количества рабочих.

Примеры

Входные данные	Выходные данные
3 5 1 2 0 1 1 1 2 1 4 0 2 3 4	4

Сначала определим, что такое максимально возможное количество рабочих. Если общее количество рабочих больше вместимости автобуса, то это — объем автобуса, если же рабочих меньше чем вместимость автобуса — то это количество всех рабочих (в этом случае вместимости автобуса уместно присвоить значение, равное количеству людей).

Когда мы считываем данные, следует определить время прихода последнего человека (т.е. то время, когда уже все люди будут на остановках) — это будет максимум в бинарном поиске. Минимум будет равен нулю. Если автобус должен задержаться перед остановкой, то он должен сделать это перед первой остановкой (действительно, если он подъедет к первой остановке, заберет людей, а потом будет ждать у второй остановки, то в это время на первую могут придти еще люди, а если ждать перед первой, то люди со второй никуда не денутся). Минимум и максимум у нас есть. Теперь берем задержку, равную $x = (min + max)/2$. С помощью процедуры, которая будет описана ниже, вычисляем, сколько людей успеет придти до момента x на первую остановку, для второй остановки будет задержка, равная $x + a[1]$, где $a[1]$ — время следования от первой остановки до второй, для третьей задержка — $x + a[1] + a[2]$ и т.д. Если количество севших в автобус на всех остановках больше либо равно вместимости автобуса, то надо заменить x на $(min + x)/2$, если остались места в автобусе то $x = (x + max)/2$. Условие выхода будет такое: если при некоторой задержке x автобус заполнен, а при задержке $(x-1)$ автобус не полон, то ответ x.

Теперь второй бинарный поиск. Тот самый, который определяет, сколько людей успеет придти на определенную остановку до определенного момента. Здесь максимумом дихотомии будет количество людей на остановке, а минимумом — ноль. Выбираем среднего человека — если его время прихода меньше, чем задержка, то $x = (x+max)/2$, если он не успеет придти, то $x = (min + x)/2$. Здесь условие выхода такое: если человек успевает придти на остановку, а следующий за ним нет — то ответом будет номер человека. Отдельно нужно обрабатывать случай, если на автобус сядут все люди с остановки.

3.6 Поиск по групповому признаку

В прошлой части лекции мы рассматривали задачи, в которых для данного можно получить ответ «больше» или «меньше». Теперь рассмотрим задачи, в которых для некоторого подмножества всех элементов можно получить ответ, например, на вопрос «содержится ли искомый элемент в данном подмножестве?».

Для примера рассмотрим задачу поиска одного радиоактивного шарика среди 8 шариков. При этом мы можем измерить радиоактивность некоторой группы шариков и определить, содержится ли радиоактивный шарик в этой группе. Необходимо минимизировать количество измерений для худшего случая.

Представим номера шариков в виде двоичных чисел:

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

Мы можем найти радиоактивный шарик за три измерения. При этом в первом измерении будут участвовать шарики, содержащие 1 в первом разряде, во втором — шарики с 1 во втором разряде и т.д. Результаты измерений будем записывать так: если в группе содержится радиоактивный шарик, то запишем 1, в соответствующий номеру измерения разряд, в противном случае запишем 0.

Полученное двоичное число будет однозначно определять номер радиоактивного шарика.

В общем случае для поиска 1 шарика среди N шариков необходимо $\lceil \log_2(N) \rceil$ измерений.

Похожее решение имеет следующая задача: среди 9 монет необходимо найти одну фальшивую, пользуясь чашечными весами, если известно, что фальшивая монета весит больше настоящей. Здесь для каждого взвешивания возможно три результата: перевесила левая чашка, перевесила правая и весы уравновешены.

Закодируем номера монет в троичной системе счисления:

0	1	2	3	4	5	6	7	8
00	01	02	10	11	12	20	21	22

Задачу можно решить за два взвешивания, при этом 1 будет означать, что монету нужно положить на левую чашу весов, 2 — на правую чашу, а 0 — что монета не участвует во взвешивании.

Запишем результаты каждого взвешивания в соответствующий разряд (1 — перевесила левая чаша, 2 — правая, 0 — весы уравновешены). Полученный результат однозначно определяет номер фальшивой монеты. В общем случае для поиска ответа необходимо $\lceil \log_3(N) \rceil$ взвешиваний.

Теперь рассмотрим задачу поиска фальшивой монеты среди 12 с помощью чашечных весов, при условии, что неизвестно, тяжелее ли фальшивая монета или легче.

Первое логичное условие, которое мы опускали в предыдущих задачах, состоит в том, что количество монет на различных чашах весов должно быть одинаковым.

Интуитивное решение состоит в том, чтобы разделить монеты на 4 части — 2 части отложить и положить по одной части на каждую из чаш весов, если весы уравновешены, то фальшивая монета находится в отложенной части, иначе — среди монет, положенных на чаши. Будем продолжать данный процесс для части, содержащей фальшивую монету, и получим оценку $\lceil \log_2(N) \rceil$ (фактически, задача свелась к задаче о радиоактивном

шарике). При этом мы никак не учитываем результаты предыдущих взвешиваний, которые также могут нести полезную информацию.

Введем следующую систему кодирования: 0 означает, что монета не участвует во взвешивании, 1 — что участвует (при этом, если она уже участвовала во взвешиваниях, то монета остается на той же чаше), 2 — что монета участвует во взвешивании (при этом она обязательно участвовала в одном из предыдущих взвешиваний и в текущем взвешивании лежит на противоположной чаше весов). Из этого условия следует, что ни в каком коде 2 не может предшествовать 1. Запишем все варианты, выписывая, для удобства, коды в столбик:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	1	0	1	1	1	0	1	1	1	1	1
0	0	1	0	1	0	1	1	1	0	2	1	2	2
0	1	0	0	1	1	0	1	2	2	0	2	1	2

Чтобы было возможно провести измерения, необходимо, чтобы в каждой строке было четное число элементов (иначе количество монет на чашах будет различным). Для этого вычеркнем из таблицы, например, 0 и 7 столбцы (как видно, 0 столбец не меняет четность, и задача может быть решена и для 13 монет).

В первом взвешивании возьмем 8 монет с 1 в первом разряде и положим их на чаши весов по 4. На 2 взвешивании уберем 3 монеты (у которых 0 во втором разряде), переложим 3 монеты на противоположную чашу весов и заполним оставшиеся места теми монетами, у которых 1 во втором разряде возникает впервые. Пользуясь теми же соображениями, проведем второе взвешивание.

Результаты взвешиваний будем записывать следующим образом: если чаши уравновешены, то записываем в разряд, соответствующий измерению, 0. Если одна чаша весов перевесила впервые или в ту же сторону, как и в предыдущем взвешивании (когда чаши не были уравновешены), то записываем 1. Если же весы перевесили в противоположную предыдущему неуравновешенному взвешиванию сторону, то запишем 2. В результате получим код фальшивой монеты (действительно, если фальшивая монета не участвовала во взвешивании, то весы уравновешены, если была на одной и той же чаше — получим единицы, а если на разных — двойки). В общем случае количество взвешиваний будет равно $\lceil log_3(2 \times N + 1) \rceil$, т.е. для $N \geq 9$ это решение более эффективно, чем интуитивное.

В общем случае, в решении задач, где можно определить некий признак для группы элементов или для нескольких групп, мы должны стремиться разбить элементы на как можно более похожие по количеству группы (это необходимо для минимизации количества измерений в худшем случае). Каждому элементу следует ставить в соответствие код фиксированной длины и находить метод отображения результатов измерений в соответствующий код.

Лекция 4

Алгоритмы сортировки

Версия C от 16.11.2009

4.1 Сортировка пузырьком

Условимся считать массив отсортированным, если элементы расположены в порядке неубывания (т.е. каждый элемент не меньше предыдущего). Все примеры будем рассматривать на типе `int`, однако он может быть заменен любым другим сравнимым типом данных.

Рассмотрение методов сортировки начнем с сортировки пузырьком (BubbleSort).

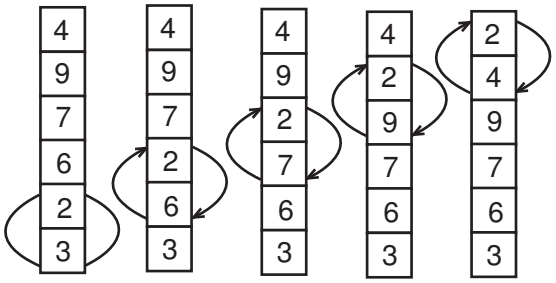


Рис. 4.1: Нулевой проход. Сравниваемые пары и обмены выделены

Это один из простейших методов сортировки, который обычно входит в школьный курс программирования. Название метода отражает его сущность: на каждом шаге самый «легкий» элемент поднимается до своего места («всплывает»). Для этого мы просматриваем все элементы снизу вверх, берем пару соседних элементов и, в случае, если они стоят неправильно, меняем их местами.

Вместо поднятия самого «легкого» элемента можно «топить» самый «тяжелый».

Т.к. за каждый шаг на свое место встает ровно 1 элемент (самый «легкий» из оставшихся), то нам потребуется выполнить N шагов.

Текст функции сортировки можно записать так:

```
void bubble_sort(int a[], int n)
{
  int i, j, k;
  For(i, n)
  for (j=n-1; j>i; j--)
    if (a[j-1] > a[j]) {
      k = a[j-1];
      a[j-1] = a[j];
      a[j] = k;
    }
```

```
}
```

Напомним, что здесь мы использовали макроподстановку

```
#define For(a,b) for(a=0; a<b; a++)
```

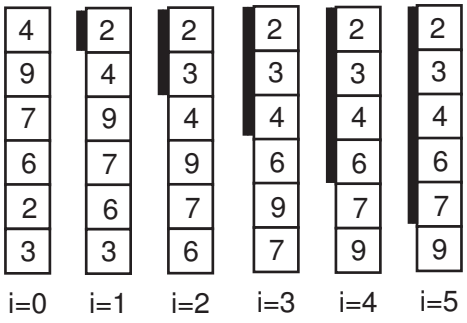


Рис. 4.2: Номер прохода. Отсортированная часть выделена полосой

которая избавляет нас от необходимости писать длинные команды и уменьшает количество возможных опечаток и ошибок.

Алгоритм не использует дополнительной памяти, т.е. все действия осуществляются на одном и том же массиве.

Сложность алгоритма сортировки пузырьком составляет $O(N2)$, количество операций сравнения: $N \times (N-1)/2$. Это очень плохая сложность, но алгоритм имеет два плюса.

Во-первых, он легко реализуется, а значит, может и должен применяться в тех случаях, когда требуется однократная сортировка массива. При этом размер массива не должен быть больше 10000, т.к. иначе алгоритм сортировки пузырьком не будет укладываться в отведенное время.

Во-вторых, сортировка пузырьком использует только сравнения и перестановки соседних элементов, а значит, может использоваться в тех задачах, где явно разрешен только такой обмен и для сортировки, например, списков.

Существуют разнообразные «оптимизации» сортировки пузырьком, которые усложняют (а нередко и увеличивают время работы алгоритма), но не приносят выгоды ни в плане сложности, ни в плане быстродействия.

На этом плюсы сортировки пузырьком заканчиваются. В дальнейшем мы еще более сузим область применения сортировки пузырьком.

4.2 Сортировка прямым выбором

Рассмотрим еще один квадратичный алгоритм, который, однако, является оптимальным по количеству присваиваний и может быть использован, когда по условию задачи необходимо явно минимизировать количество присваиваний.

Суть метода заключается в следующем: мы будем выбирать минимальный элемент в оставшейся части массива и приписывать его к уже отсортированной части. Повторив эти действия N раз, мы получим отсортированный массив.

```
void select_sort(int a[], int n)
{
  int i, j, k;
  For (i,n) {
    k=i;
    for(j=i+1; j<n; j++)
      if (a[j]<a[k]) k=j;
```

```
    j=a[k]; a[k]=a[i]; a[i]=j;
  }
}
```

Количество сравнений составляет $O(N^2)$, а количество присваиваний всего $O(N)$. В целом это плохой метод, и он должен быть использован только в случаях, когда явно необходимо минимизировать количество присваиваний.

4.3 Пирамидальная сортировка

Начнем рассмотрение эффективных алгоритмов сортировки (работающих за $O(N \log N)$) с пирамидальной сортировки, в которой используются знакомые нам идеи кучи.

Мы будем выбирать из кучи самый большой элемент, и записывать его в начало уже отсортированной части массива (сортировка выбором в обратном порядке). Т.е. отсортированный массив будет строиться от конца к началу. Такие ухищрения необходимы, чтобы не было необходимости в дополнительной памяти и для ускорения работы алгоритма — куча будет располагаться в начале массива, а отсортированная часть будет находиться после кучи.

Напомним свойство кучи максимумов: элементы с индексами $i+1$ и $i+2$ не больше, чем элемент с индексом i (естественно, если $i+1$ и $i+2$ лежат в пределах кучи). Пусть n — размер кучи, тогда вторая половина массива (элементы от $n/2+1$ до n) удовлетворяют свойству кучи. Для остальных элементов вызовем функцию «проталкивания» по куче, начиная с $n/2$ до 0.

```
void down_heap(int a[], int k, int n)
{
  int  temp=a[k];
  while (k*2+1 < n) {
    y=k*2+1;
    if (y < n-1 && a[y] < a[y+1]) y++;
    if (temp >= a[y]) break;
    a[k]=a[y];
    k=y;
  }
  a[k]=temp;
}
```

Эта функция получает указатель на массив, номер элемента, который необходимо протолкнуть и размер кучи. У нее есть небольшие отличия от обычных функций работы с кучей. Номер минимального предка хранится в переменной y, если необходимость в обменах закончена, то мы выходим из цикла и записываем просеянную переменную на предназначенное ей место.

Сама сортировка будет состоять из создания кучи из массива и N переносов элементов с вершины кучи с последующим восстановлением свойства кучи:

```
void heap_sort(int a[], int n) {
  int i, temp;
```

```
  for(i=n/2-1; i >= 0; i--) down_heap(a, i, n);
  for(i=n-1; i > 0; i--) {
    temp=a[i]; a[i]=a[0]; a[0]=temp;
    down_heap(a, 0, i);
  }
}
```

Всего в процессе работы алгоритма будет выполнено $3 \times N/2 - 2$ вызова функции `down_heap`, каждый из которых занимает $O(\log N)$. Таким образом, мы и получаем искомую сложность в $O(N \log N)$, не используя при этом дополнительной памяти. Количество присваиваний также составляет $O(N \log N)$.

Пирамидальную сортировку следует осуществлять, если из условия задачи понятно, что единственной разрешенной операцией является «проталкивание» элемента по куче, либо в случае отсутствия дополнительной памяти.

4.4 Быстрая сортировка

Мы уже рассматривали идеи, которые используются в быстрой сортировке, при поиске порядковых статистик. Точно так же, как и в том алгоритме, мы выбираем некий опорный элемент и все числа, меньшие его перемещаем в левую часть массива, а все числа большие его — в правую часть. Затем вызываем функцию сортировки для каждой из этих частей.

Таким образом, наша функция сортировки должна принимать указатель на массив и две переменные, обозначающие левую и правую границу сортируемой области.

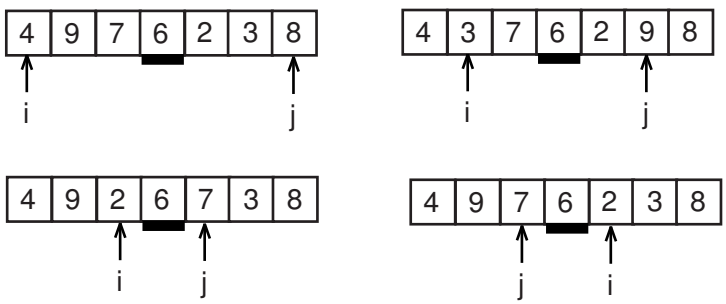


Рис. 4.3: Первый проход быстрой сортировки

Остановимся более подробно на выборе опорного элемента. В некоторых книгах рекомендуется выбирать случайный элемент между левой и правой границей. Хотя теоретически это красиво и правильно, но на практике следует учитывать, что функция генерации случайного числа достаточно медленная и такой метод заметно ухудшает производительность алгоритма в среднем.

Наиболее часто используется середина области, т.е. элемент с индексом $(l + r)/2$. При таком подходе используются быстрые операции сложения и деления на два, и в целом он работает достаточно неплохо. Однако в некоторых задачах, где сутью является исключительно сортировка, хитрое жюри специально подбирает тесты так, чтобы «завалить» стандартную быструю сортировку с выбором опорного элемента из середины. Стоит заметить, что это очень редкая ситуация, но все же стоит знать, что можно

выбирать произвольный элемент с индексом m так, чтобы выполнялось неравенство $l \le m \le r$. Чтобы это условие выполнялось, достаточно выбрать произвольные два числа x и y и выбирать m исходя из следующего соотношения: $m = (x \times l + y \times r)/(x+y)$. В целом такой метод будет незначительно проигрывать выбору среднего элемента, т.к. требует двух дополнительных умножений.

Приведем текст функции быстрой сортировки с выбором среднего элемента в качестве опорного:

```
void quick_sort(int a[], int left, int right)
{
  int i = left, j = right, temp, p;
  p = a[(left+right)/2];
  do {
    while (a[i] < p) i++;
    while (a[j] > p) j--;
    if (i <= j) {
      temp = a[i]; a[i] = a[j]; a[j] = temp;
      i++; j--;
    }
  } while (i <= j);
  if (j > left) quick_sort(a, left, j);
  if (i < right) quick_sort(a, i, right);
}
```

Чтобы воспользоваться быстрой сортировкой, необходимо передать в функцию левую и правую границы сортируемого массива (т.е., например, вызов для массива `a` будет выглядеть как `quick_sort(a, 0, n-1)`.

Алгоритм быстрой сортировки в среднем использует $O(N \log N)$ сравнений и $O(N \log N)$ присваиваний (на практике даже меньше) и использует $O(\log N)$ дополнительной памяти (стек для вызова рекурсивных функций). В худшем случае алгоритм имеет сложность $O(N^2)$ и использует $O(N)$ дополнительной памяти, однако вероятность возникновения худшего случая крайне мала: на каждом шаге вероятность худшего случая равна $2/N$, где N — текущее количество элементов.

Рассмотрим возможные оптимизации метода быстрой сортировки.

Во-первых, при вызове рекурсивной функции возникают накладные расходы на хранение локальных переменных (которые нам не особо нужны при рекурсивных вызовах) и другой служебной информацией. Таким образом, при замене рекурсии стеком мы получим небольшой прирост производительности и небольшое снижение требуемого объема дополнительной памяти.

Во-вторых, как мы знаем, вызов функции — достаточно накладная операция, а для небольших массивов быстрая сортировка работает не очень хорошо. Поэтому, если при вызове функции сортировки в массиве находится меньше, чем K элементов, разумно использовать какой-либо нерекурсивный метод, например, сортировку вставками или выбором. Число K при этом выбирается в районе 20, конкретные значения подбираются опытным путем. Такая модификация может дать до 15% прироста производительности.

Быструю сортировку можно использовать и для двусвязных списков (т.к. в ней осуществляется только последовательный доступ с начала и с конца), но в этом случае

возникают проблемы с выбором опорного элемента — его приходится брать первым или последним в сортируемой области. Эту проблема можно решить неким набором псевдослучайных перестановок элементов списка, тогда даже если данные были подобраны специально, эффект нейтрализуется.

4.5 Сортировка слияниями

Сортировка слияниями также основывается на идее, которая уже была нами затронута при рассмотрении алгоритма поиска двух максимальных элементов. В этом алгоритме мы сначала разобьем элементы на пары и упорядочим их внутри пары. Затем из двух пар создадим упорядоченные четверки и т.д.

3	7	8	2	4	6	1	5	Последовательности длины 1
3 7		2 8		4 6		1 5		Слияние до упорядоченных пар
2 3 7 8				1 4 5 6				Слияние пар в упорядоченные четверки
1 2 3 4 5 6 7 8								Слияние пар в упорядоченные четверки

Интерес представляет сам процесс слияния: для каждой из половинок мы устанавливаем указатели на начало, смотрим, в какой из частей элемент по указателю меньше, записываем этот элемент в новый массив и перемещаем соответствующий указатель.

Опишем функцию слияния следующим образом:

```
void merge(int a[], int b[], int c, int d, int e)
{
  int p1=c, p2=d, pres=c;
  while (p1 < d && p2 < e)
    if (a[p1] < a[p2])
      b[pres++] = a[p1++];
    else
      b[pres++] = a[p2++];
  while (p1 < d)
    b[pres++]=a[p1++];
  while (p2 < e)
    b[pres++]=a[p2++];
}
```

Здесь a — исходный массив, b — массив результата, c и d — указатели на начало первой и второй части соответственно, e — указатель на конец второй части.

Далее опишем довольно хитрую нерекурсивную функцию сортировки слиянием:

```
void merge_sort(int a[], int n)
{
  int *temp, *a2=a, *b=(int*)malloc(n*sizeof(int)), *b2;
  int c, k = 1, d, e;
  b2=b;
  while (k <= 2*n) {
    for (c=0; c<n; c+=k*2) {
```

```
      d=c+k<n?c+k:n;
      e=c+2*k<n?c+2*k:n;
      merge(a2, b, c, d, e);
    }
    temp = a2; a2 = b; b = temp;
    k *= 2;
  }
  for (c=0; c<n; c++)
    a[c] = a2[c];
  free(b2);
}
```

Рекурсивная реализация сортировки слияними несколько проще, но обладает меньшей эффективностью и требует $O(\log N)$ дополнительной памяти.

Алгоритм имеет сложность $O(N \log N)$ и требует $O(N)$ дополнительной памяти.

В оригинале этот алгоритм был придуман для сортировки данных во внешней памяти (данные были расположены в файлах) и требует только последовательного доступа. Этот алгоритм применим для сортировки односвязных списков.

4.6 Сортировка подсчетом

Это сортировка может использоваться только для дискретных данных. Допустим, у нас есть числа от 0 до 99, которые нам следует отсортировать. Заведем массив размером в 100 элементов, в котором будем запоминать, сколько раз встречалось каждое число (т.е. при появлении числа будем увеличивать элемент вспомогательного массива с индексом, равным этому числу, на 1). Затем просто пройдем по всем числам от 0 до 99 и выведем каждое столько раз, сколько оно встречалось. Сортировка реализуется следующим образом:

```
for (i=0; i<MAXV; i++)
  c[i] = 0;
for (i=0; i<n; i++)
  c[a[i]]++;
k=0;
for (i=0; i<MAXV; i++)
  for (j=0; j<c[i]; j++)
    a[k++]=i;
```

Здесь $MAXV$ — максимальное значение, которое может встречаться (т.е. все числа массива должны лежать в пределах от 0 до $MAXV - 1$).

Алгоритм использует $O(MAXV)$ дополнительной памяти и имеет сложность $O(N + MAXV)$. Его применение дает отличный результат, если $MAXV$ намного меньше, чем количество элементов в массиве.

4.7 Поразрядная сортировка

Алгоритм сортировки подсчетом чрезвычайно привлекателен своей высокой производительностью, но она ухудшается при возрастании $MAXV$, также резко возрастают

требования к дополнительной памяти. Фактически, невозможно осуществить сортировку подсчетом для переменных типа `unsigned int` ($MAXV$ при этом равно 2^{32}).

В качестве развития идеи сортировки подсчетом рассмотрим поразрядную сортировку. Сначала отсортируем числа по последнему разряду (единиц). Затем повторим то же самое для второго и последующих разрядов, пользуюсь каким либо устойчивым алгоритмом сортировки (т.е. если числа с одинаковым значением в сортируемом разряде шли в одном порядке, то в отсортированной последовательности они будут идти в том же порядке).

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	837	657	839

Для примера приведем таблицу, в первом столбце которой расположены исходные данные, а в последующих — результат сортировки по разрядам.

Для самой сортировки будем использовать сортировку подсчетом. После этого будем переделывать полученную таблицу так, чтобы для каждого возможного значения разряда сохранялась позиция, начиная с которой идут числа с таким значением в соответствующем разряде (т.е. сколько элементов имеют меньшее значение в этом разряде). Назовем этот массив c.

После этого будем проходить по всему исходному массиву, смотреть на текущее значение разряда (i), записывать текущее число во вспомогательный массив (b) на позицию $c[i]$, а затем увеличивать $c[i]$ (чтобы новое число с таким же значением текущего разряда не легло поверх уже записанного).

Пусть количество знаков в числе равно k, а количество возможных значений равно m (система счисления, использованная при записи числа). Тогда количество присваиваний, производимое алгоритмом, будет равно $O(k \times N + k \times m)$, а количество дополнительной памяти — $O(N + k \times m)$.

Приведем эффективную реализацию поразрядной сортировки для беззнаковых 4-байтных чисел (`unsigned int`). Мы будем использовать 4 разряда, каждый из которых равен байту (система счисления с основанием 256). Эта реализация использует несколько хитростей, которые будут пояснены ниже.

```
void radix_sort(unsigned int a[], int n)
{
  unsigned int *t, *a2=a;
  unsigned int *b=(unsigned int*) malloc(n*sizeof(int));
  unsigned int *b2;
  unsigned char *bp;
  int i, j, npos, temp;
  int c[256][4];
  b2 = b;
  memset(c, 0, sizeof(int)*256*4);
  bp = (unsigned char*) a;
  For(i,n)
    For (j,4) {
      c[*bp][j]++;
      bp++;
    }
  For(j,4) {
```

```
    npos = 0;
    For(i,256) {
      temp = c[i][j];
      c[i][j] = npos;
      npos += temp;
    }
  }
  For(j,4) {
    bp = (unsigned char*) a2 + j;
    For(i,n) {
      b[c[*bp][j]++] = a2[i];
      bp += 4;
    }
    t = a2; a2 = b; b = t;
  }
  free(b2);
}
```

Функция `memset` используется для заполнения заданной области памяти нулями (обнуление массива), она находится в библиотеке `string.h`. Всю таблицу сдвигов (c) мы будем строить заранее для всех 4 разрядов. Для эффективно доступа к отдельным байтам мы будем использовать указатель bp типа `unsigned char *` (тип `char` как раз занимает 1 байт и может трактоваться как число). Затем мы формируем модифицированную таблицу и проводим собственно функцию расстановки чисел по всем 4 разрядам.

Внимательный читатель заметит в приведенной функции несколько мест, которые на первый взгляд кажутся ошибочными. Хотя в текстовом описании мы и говорили, что следует сортировать, начиная с последних разрядов, в реализации мы начинаем с первых байтов. Это объясняется тем, что в архитектуре **x86** числа хранятся в «перевернутом» виде — это было сделано для совместимости с младшими моделями.

Второе возможное место для ошибки — массив a не ставится в соответствие указателю отсортированного массива и не осуществляется копирование отсортированных элементов в него в конце работы функции. Это связано с четным количеством разрядов, так что результат в итоге и так оказывается в массиве a.

Вообще говоря, далеко не обязательно так сильно связывать поразрядную сортировку с аппаратными средствами. Более того, основное удобство поразрядной сортировки состоит в там, что с ее помощью можно сортировать сложные структуры, такие как даты, строки (массивы) и другие структуры со многими полями.

4.8 Сравнение производительности сортировок

Название	Сравнений	Присваиваний	Память	Время
Пузырьком	$O(N^2)$	$O(N^2)$	0	$---$
Выбором	$O(N^2)$	$O(N)$	0	$---$
Пирамидальная	$O(N \log N)$	$O(N \log N)$	0	16437
Быстрая	$O(N \log N)$	$O(N \log N)$	$O(\log N)$	5618
Слиянием	$O(N \log N)$	$O(N \log N)$	$O(N)$	7669
Подсчетом	0	$O(N)$	$O(MAXV)$	322
Поразрядная	0	$O(kN + km)$	$O(N + km)$	1784

Тестирование проводилось на 10^7 случайных чисел, которые не превышали 10^5. Для квадратичных алгоритмов тестирование не проводилось (поскольку это заняло бы очень большое время). При тестировании использовались приведенные выше функции. В таблице k — количество «цифр», а m — количество значений, которое может принимать каждая цифра.

Быстрая сортировка в худшем случае имеет $O(N^2)$ сравнений и присваиваний.

Как видно из таблицы, поразрядная сортировка опережает все остальные на «обычных» данных, но для практического применения больше подходит быстрая сортировка, т.к. разница уменьшается по мере уменьшения количества сортируемых элементов, а само по себе считывание 10^7 элементов занимает время намного большее, чем разница во времени между этими сортировками.

При решении той или иной задачи следует выбирать нужный тип сортировки исходя из таблицы и специфических условий применимости (минимизация количества сравнений или присваиваний, сортировка списков или последовательный доступ к данным).

Еще одна практическая рекомендация заключается в следующем: при сортировке сложных структур (например, строк) следует не производить обмены, а просто переставлять указатели (строка может состоять из нескольких тысяч символов, а указатель занимает 4 байта и процесс сортировки может ускориться во много раз).

4.9 Сканирующая прямая

Очень часто в олимпиадных задачах возникает необходимость обработать некоторые события по порядку. В этом случае события упорядочиваются по оси «времени» и сканирующая прямая движется вдоль этой прямой, переходя от события к следующему. Рассмотрим, например, такую классическую задачу: на прямой задано N отрезков двумя числами: координатами начала и конца отрезка. Требуется определить, какое максимальное количество отрезков покрывает некоторую точку.

Будем двигаться по «событиям» (в нашем случае событием считается достижение точки, которая является началом или конца отрезка) слева направо. Если очередная точка — начало отрезка, то прибавим к счетчику начавшихся, но не кончившихся отрезков единицу. Если же очередная точка — конец отрезка, то вычтем из счетчика единицу. В случае, если в одной точке происходит несколько событий, то будем учитывать сначала начала отрезков, а затем их концы: в нашей задаче точка считается покрытой отрезком, если она покрыта хотя бы одной его точкой, а значит в случае если в некоторой точке предыдущий отрезок кончается, а новый — начинается, то необходимо учесть, что точка покрыта двумя отрезками. В других задачах упорядочивания

одновременных событий определяется из условия.

Для реализации такого решения, необходимо «слить» координаты начал и концов отрезков в один массив, при этом дополнив их признаком того, является ли точка началом или концом отрезка. Затем этот массив сортируется в первую очередь по координате, а в случае их совпадения — по признаку начала или конца. После этого необходим цикл, который проходит по всем упорядоченным точкам и прибавляет единицу к счетчику в случае если очередная точка имеет признак начала и вычитает единицу если очередная точка является концом отрезка. Достигнутый во время этого прохода максимум и будет являться ответом на задачу.

Задача становится интереснее, если кроме самого максимального количества отрезков, которыми покрыта точка, необходимо вывести и номера этих отрезков.

Рассмотрим решение «в лоб». Список начавшихся, но не закончившихся, отрезков легко поддерживать в виде булевского массива. Однако при нахождении очередного максимума надо совершить проход по этому массиву чтобы сохранить ответ. Один проход по такому массиву займет $O(N)$ времени, ответ также может обновляться порядка N раз (когда все отрезки последовательно вложены друг в друга). Таким образом, сложность такого решения составит $O(N^2)$, что слишком много для большинства задач такого рода.

Более разумно сделать двухпроходное решение. Первый проход будет решать исходную задачу — искать максимальное количество отрезков, покрывающих точку. Второй проход будет незначительно отличаться от первого, а именно: когда значение счетчика начавшихся, но не закончившихся отрезков станет равно максимальному значению, следует прекратить дальнейший поиск и пройти по булевскому массиву, в котором помечены эти отрезки и вывести их. Сложность такого решения составит $O(N)$ (первый и второй проход займут порядка N операций, проход по массиву открытых отрезков также займет $O(N)$ операций и будет выполнен один раз.

Лекция 5

STL

Версия C от 21.11.2009

5.1 Введение

STL расшифровывается как **Standard Template Library**. По-русски это звучит как «Стандартная библиотека шаблонов».

Механизм шаблонов и библиотека **STL** относятся к языку **C++**, поэтому мы не будем сильно углубляться в синтаксис и часть моментов оставим без подробного описания. Программы, использующие **STL** можно компилировать только компилятором языка **C++** (например, **g++**)!

Вкратце шаблоны можно характеризовать как функции, для которых задается тип входных параметров. Например, **QuickSort** одинаково хорошо может сортировать и целые, и вещественные числа, а также другие типы данных. Однако без использования шаблонов нам пришлось бы создавать свою функцию для каждого из типов данных, при этом очень часто функции различались бы только в заголовке и типах локальных переменных. Использование шаблона значительно ускоряет разработку больших проектов и библиотек, но их непосредственное использование на олимпиаде неоправданно, т.к. значительно проще сделать copy-paste и внести необходимые изменения в новую функцию.

STL предоставляет собой набор шаблонных функций, которые реализуют функциональность многих структур данных и некоторых алгоритмов. В задачах, где требуется выполнить чисто технические действия (а такие действия почти всегда бывают в олимпиадных задачах) уместно использовать **STL**, т.к. это ускоряет процесс написания программы. Тем не менее, это не означает, что можно забыть методы сортировки и реализацию структур данных, т.к. часто решение задачи основывается на каком-либо алгоритме, но не может быть реализовано с помощью библиотечной функции (например, наложены какие-либо ограничения, которые невозможно или очень сложно отразить в библиотеке).

Таким образом, знание **STL** может избавить нас от рутинной работы, а это не только приятно, но и полезно.

Книги, посвященные **STL**, нередко содержат несколько сотен страниц, поэтому, естественно, в лекции будет охвачена далеко не вся функциональность библиотеки. В этой лекции мы будем рассматривать возможности **STL** только для уже изученных

структур данных и алгоритмов, в дальнейшем будем указывать возможности **STL** по мере изучения новых структур данных и алгоритмов.

Для того чтобы **STL** работал, в начале программы (после всех `#include`) необходимо написать `using namespace std;` Эта команда указывает, что надо пользоваться пространством имен `std`. В нашем курсе суть команды не важна, главное — нам необходимо знать, что это накладывает на нас ограничения - запрещено использовать некоторые слова, которые становятся «ключевыми» (например: `stack`, `queue`, `vector`, `sort`, `count`, ...).

5.2 Пара (pair)

Пара, фактически является шаблонной структурой, которая содержит два поля (возможно, разных типов), называются они `first` и `second`.

Для того чтобы использовать `pair`, необходимо подключить библиотеку `<utility>` (обратите внимание, что `.h` писать не надо).

Пусть, например, мы хотим создать пару из целого и вещественного числа. Тогда ее создание будет выглядеть следующим образом:

```
pair <int, double> p;
```

Теперь мы можем обращаться к `p` точно так же, как к обычной структуре, например, так:

```
p.first = 5; p.second = 3.1415;
```

Пары одинакового типа можно присваивать друг другу. Пары используются в ассоциативных контейнерах (о них будет сказано позже). Поля пары могут быть не только элементарного, но и составного типа, например, опять же парой. Для примера, приведем реализацию структуры, хранящей дату с использованием пар (год, месяц, день):

```
pair <int, pair <int, int> > date1, date2;
```

Обратите внимание, что `>` разделены пробелом, если не разделять их, то эта запись будет интерпретироваться как оператор сдвига вправо `>>`, что приведет к ошибке при компиляции.

Обращение к полям будет выглядеть так:

```
int year = date1.first;
int month = date1.second.first;
int day = date1.second.second;
```

Не очень удобный формат, но можно придумать макросы, которые облегчат нам доступ к полям.

Крайне полезное свойство состоит в том, что пары можно сравнивать. При этом сравнение идет слева направо. Т.о. для нашей даты сначала сравнятся годы, при равных годах сравнятся месяцы и т.д. Это очень удобно использовать при сортировке. На этом закончим рассмотрение пар и перейдем к более осмысленным структурам данных.

5.3 Стек (stack)

Стек уже знакомая нам структура данных. Удобство использование стека **STL** состоит в том, что у нас нет необходимости заранее задавать размер стека, однако за это

приходится расплачиваться накладными расходами. Фактически, **STL** реализует стек на структуре, которая работает аналогично расширяемому массиву из второй лекции ($\log N$ выделений памяти и двукратное увеличение требуемой памяти).

Чтобы использовать стек, необходимо подключить библиотеку `<stack>`. Приведем пример программы, а затем разберемся со всеми использованными функциями:

```
#include <stack>
#include <stdio.h>

using namespace std;

int main()
{
  stack <int> S;
  S.push(8);
  S.push(7);
  int x = S.size(); //x==2
  while (!S.empty()) {
    printf("%d ", S.top());
    S.pop();
  }
  return 0;
}
```

Вывод этой программы будет 78 (как и положено стеку).

Вначале мы создаем стек для целых чисел `stack <int>`, при этом он пуст. Для стека определены следующие функции:

`void push(<type>)` — добавление элемента в стек.

`void pop()` — удаляет элемент с вершины стека.

`<type> top()` — возвращает элемент с вершины стека.

`unsigned int size()` — определяет размер стека (количество элементов).

`bool empty()` — возвращает истину, если стек пуст.

Эти функции являются методами класса. Для тех, кто не знает **C++** следует понимать их аналогично полям структуры (синтаксис обращения к методам класса такой же, как к полям структуры).

5.4 Очередь (queue)

Для работы с очередью необходимо подключить библиотеку `<queue>`. Очередь также называется `queue`. Чтобы создать очередь из вещественных чисел следует наисать `queue <double> q;`

У очереди имеются следующие методы:

`void push(<type>)` — добавление элемента в очередь.

`void pop()` — удаляет элемент из головы очереди.

`<type*> front()` — возвращает ссылку на элемент из головы очереди.

`<type*> back()` — возвращает ссылку на элемент из хвоста очереди.

`unsigned int size()` — определяет размер очереди (количество элементов).

`bool empty()` — возвращает истину, если очередь пуста.

5.5 Дек (deque)

Для работы с деком необходимо подключить библиотеку `<deque>`. Чтобы создать дек из символов следует наисать `deque <char> d;`

Очереди и стеки в **STL** реализуются над деком (т.к. дек позволяет реализовывать функциональность и того и другого).

Некоторые методы дека:

`void push_back(<type>)` — добавление элемента в конец дека.

`void push_front<type>)` — добавление элемента в начало дека.

`void pop_back()` — удаляет элемент с конца дека.

`void pop_front()` — удаляет элемент с начала дека.

`<type*> front()` — возвращает ссылку на элемент из головы дека.

`<type*> back()` — возвращает ссылку на элемент из хвоста дека.

`unsigned int size()` — определяет размер дека (количество элементов).

`bool empty()` — возвращает истину, если дек пуст.

5.6 Очередь с приоритетами (куча, priority_queue)

Для работы с кучей (очередью с приоритетами) необходимо подключить библиотеку `<queue>`. Куча называется `priority_queue`. Чтобы создать кучу из вещественных чисел следует наисать `priority_queue <double> h;`

Некоторые методы кучи:

`void push(<type>)` — добавление элемента в кучу.

`void pop()` — удаляет наибольший элемент из кучи.

`<type> top()` — возвращает наибольший элемент в куче.

`unsigned int size()` — определяет размер кучи (количество элементов).

`bool empty()` — возвращает истину, если куча пуста.

5.7 Динамически расширяемый массив (vector)

`vector` в **STL** ведет себя точно также, как динамически расширяемый массив из второй лекции. Для того чтобы `vector` работал необходимо подключить библиотеку `<vector>`.

Мы можем создавать пустой вектор так: `vector <int> v;`

Кроме того, мы можем создать вектор с заранее заданным начальным количеством элементов: `vector <int> v(10);`

Обратите внимание, что запись `vector <int> v[10];` создаст массив из 10 пустых векторов.

Для добавления элементов в вектор следует использовать уже знакомый нам `push_back`. Доступ к элементам вектора можно осуществлять так же, как и в обычном массиве, задавая индекс в квадратных скобках.

Кроме того, в векторе существуют и некоторые другие полезные методы:

`void push_back(<type>)` — добавление элемента в конец вектора. При необходимости (вектор заполнен) он расширяется вдвое.

`unsigned int size()` — определяет размер вектора (количество элементов).

`unsigned int capacity()` — возвращает максимальное количество элементов, которое может поместится в вектор без расширения.

`void resize()` — изменяет размер вектора. Эта операция аналогична `realloc`.

`void clear()` — очищает содержимое вектора (размер вектора становится равным нулю).

5.8 Вектор битов (bit_vector)

В библиотеке `<vector>` также существует специальный тип для `vector <bool>` (логические переменные, принимающие значения `true` и `false`) — `bit_vector`. Этот тип не просто создает вектор из булевских переменных (каждая из которых занимает 1 байт), а оптимизирует расположение так, что в один байт помещается 8 `bool`-переменных. Работа его аналогична битовому массиву, но мы лишаемся возможности оперировать целыми наборами битов в рамках одной переменной (что часто бывает удобно и является необходимым этапом решения задачи). Таким образом, область применения `bit_vector` весьма ограничена. Для него существуют операции `resize`, обращение по индексу и некоторые другие. Проверьте, что ваш компилятор поддерживает `bit_vector` (этот тип доступен далеко не во всех компиляторах).

5.9 Строка (string)

Класс `string` является расширением класса `vector <char>` и содержит некоторые специфичные для строк методы. `string` также умеет динамически расширяться, однако следует помнить, что это требует некоторых накладных расходов (как и в динамически расширяемом массиве).

В принципе, значительная часть функциональности работы со строками реализована в библиотеке `<string.h>` (функции `strcmp`, `strcat`, `strcpy`, `strlen` и др.), но использование `string`-объектов часто бывает удобнее.

Для использования `string` необходимо подключить библиотеку `<string>`. Сам объект string создается так: `string s1, s2;`

Для `string`-объектов определена операция сложения, т.е. можно написать `s1 + s2` и результатом этой операции будет конкатенация строк (строка `s2` будет дописана в конец строки `s1`). Также определена операция присваивания, которая создает копию строки (а не присваивает указатель, как в случае `char[]`). Кроме того, для `string` переопределены операции `<<` и `>>`, которые позволяют выводить и вводить `string` с помощью потоков ввода-вывода. Строки можно сравнивать между собой, пользуясь обычными операциями сравнения (`<`, `>` и т.д.)

`string` можно присваивать обычным С-массивам типа `char`, массивы могут участвовать в операциях сложения, сравнения и т.п. (для них неявно строится `string`-обертка)

Объект `string` поддерживает всю функциональность объекта `vector` (т.е. методы, применимые к `vector` могут быть также применены к `string` с тем же синтаксисом). Рассмотрим некоторые специфичные для `string` методы:

`char* c_str()` — возвращает указатель на массив `char`, который содержит С-строку. Этим методом удобно пользоваться, если используется вывод библиотеки `<stdio.h>` и в некоторых других ситуациях.

`unsigned int length()` — возвращает длину строки. Предпочтительнее использовать этот метод, а не `size()`, т.к. реализация строки может быть не только на векторе, и в таком случае функция `length` будет работать быстрее.

`unsigned int find(string substr)` — ищет подстроку `substr` в строке. Возвращает индекс, с которого начинается первое вхождение подстроки в строку.

`unsigned int find(string substr, unsigned int from)` — ищет подстроку `substr` в строке начиная с позиции `from`. Это удобно использовать для поиска всех данных подстрок.

`unsigned int rfind(string substr)` — ищет подстроку `substr` в строке. Возвращает индекс, с которого начинается последнее вхождение подстроки в строку. Фактически, то же самое что и `find`, но мы начинаем искать вхождения с конца.

`unsigned int rfind(string substr, unsigned int from)` — ищет последнюю подстроку `substr` в строке, индекс начала которой меньше `from`.

`unsigned int find_first_of(string substr)` — ищет первое вхождение какого-либо символа из `substr` в строке. Существует также аналог с параметром `from`, функция `find_last_of`, которая делает то же самое, но с конца строки, а также функция `find_first_not_of`, она ищет первый символ в строке, который не входит в `substr` (также существует `find_last_not_of`). Использование этих функций позволяет, например, разбить строку на токены (слова), передавая в качестве `substr` строку, содержащую знаки препинания и пробелы, например “ ,.\t\n)”. Это часто бывает полезно, когда необходио разбить текст на отдельные слова (встроенных средств для этого, к сожалению, нет, и приходится реализовывать это вручную).

`string substr(unsigned int n)` — создает новую строку, содержащую первые n символов строки.

`string substr(unsigned int k, unsigned int n)` — создает новую строку, содержащую n символов строки, начиная с k-го.

Также существует много других методов и их вариаций, подробнее о которых можно прочитать в соответствующем разделе помощи. Отметим методы `replace` (удаление подстрок) и `insert` (вставка подстрок), которые часто бывают полезны, но обилие их вариаций не позволяет описать их в рамках одной лекции.

5.10 Итераторы

Итератор — универсальный способ доступа к элементам контейнера. Итератор можно представить себе как указатель, для которого определено несколько операций. В частности, полезными для нас будут операции разыменовывания `(*it)` — доступ к данным по этому «указателю» (можно также использовать `->` для обращения к полям), инкремента (`++`) — переход к следующему объекту в контейнере и сравнение итераторов (`==` и `!=`). На самом деле, операций для итераторов несколько больше, но мы не будем их использовать.

Итераторы делятся на 3 типа: произвольного доступа (мы не будем их использовать, т.к. они либо не поддерживаются контейнером, либо могут быть заменены на обращение по индексу), последовательного и обратного доступа. Итератор последовательного доступа удобен для последовательной обработки элементов контейнера, а итератор обратного доступа осуществляет доступ к элементам контейнера в обратном порядке.

Для любого контейнера определены два итератора: начала и конца. Допустим, мы

хотим вывести все числа из вектора (`vector <int> v`) последовательно. Соответствующий код будет выглядеть так:

```
vector <int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
  printf("%d ", *it);
```

Чтобы создать итератор по какому-либо контейнеру, необходимо сначала указать тип контейнера (`vector <int>` в нашем случае), затем написать `::iterator` и указать имя итератора.

Чтобы итератор указывал на первый элемент контейнера, необходимо присваивание `it = v.begin()`. Метод `begin()` возвращает указатель на первый элемент любого контейнера. Мы будем идти до последнего элемента. Это реализуется с помощью `it != v.end()`. `end()` указывает на следующий после последнего элемент контейнера. Переход к следующему элементу осуществляется с помощью операции инкремента `it++`. При выводе мы просто разыменовываем итератор и получаем его содержимое.

Теперь выведем наш вектор в обратном направлении. Это делается очень похоже:

```
vector <int>::reverse_iterator it;
for (it = v.rbegin(); it != v.rend(); it++)
  printf("%d ", *it);
```

Обратите внимание, что при создании итератора у нас появилось `reverse_`, а при указании начала и конца вектора в начале появилась буква `r`.

Остальные свойства итераторов будем изучать при рассмотрении новых контейнеров и алгоритмов.

5.11 Хеш-таблица (hash_set)

Хеш-таблица — уже знакомая нам структура, которая позволяет добавлять, удалять и проверять наличие элемента во множестве за $O(1)$. В этом разделе мы будем рассматривать хеш-таблицы только от элементарных типов (об использовании `hash_set` для других типов можно прочитать в файле помощи по языку).

Для того чтобы `hash_set` работал, необходимо подключить библиотеку `<hash_set>`. В некоторых компиляторах `hash_set` находится не в пространстве имен `std`, а в пространстве `stdext`. Если ваш компилятор относится к такому типу, то следует вначале писать `using namespace stdext;`

Допустим, мы хотим создать хеш-таблицу для целых чисел. Это запишется так `hash_set <int> h;`

Для нас будут полезны следующие методы:

`void insert(<type>)` — добавление элемента в хеш-таблицу.

`hash_set <type>::iterator find(<type>)` — возвращает итератор, указывающий на элемент, который передается в качестве параметра. Если такого элемента не существует, то результат будет равен значению метода `end()` для соответствующей хеш-таблицы. Изменять значение по этому итератор нельзя!

`void erase(hash_set <type>::iterator)` — удаляет значение, на которое указывает итератор, из хеш-таблицы.

`void clear()` — очищает содержимое хеш-таблицы.

Интересующиеся могут самостоятельно изучить класс `hash_multiset`, в котором может быть несколько одинаковых элементов.

5.12 Хеш-словарь (hash_map)

Хеш-словарь очень похож на хеш-таблицу, но принимает в качестве значения пару, при этом хеширование идет только по первому элементу пары. Например, мы можем хранить пару ICQ UIN и имя пользователя. При этом первый элемент в паре должен быть уникальным.

Для того чтобы `hash_map` работал, необходимо подключить библиотеку `<hash_map>`. `hash_map` также может находиться в пространстве имен `stdext`.

Для нашего примера мы можем создать такой хеш-словарь: `hash_map <int, string> hm;`

Методы у `hash_map` очень похожи на методы `hash_set`. Поэтому просто приведем прмер:

```
hash_map <int, string> hm;
hash_map <int, string>::iterator it;
hm.insert(pair <int, string> (204883678, "gu"));
hm.insert(pair <int, string> (666666666, "unknown"));
it = hm.find(666666666);
printf("%s\n", it->second.c_str());
```

У `hash_map` есть приятная и удобная дополнительная функциональность, а именно оператор `[]` (оператор доступа по индексу). Его использование может выглядеть следующим образом:

```
int i = 204883678;
printf("%s", hm[i].c_str());
```

Добавлять элементы в хеш-словарь также можно с помощью оператора `[]`. Т.е. например, мы можем написать следующую конструкцию:

`hm[123456789] = "somebody";` и этот элемент будет добавлен в словарь.

5.13 Алгоритмы в STL

В библиотеке `<algorithm>` представлен достаточно большой (и для нас зачастую бесполезный) набор алгоритмов. Все алгоритмы работают с итераторами на контейнерах, для корректной работы необходимо использование пространства имен `std`. Нам, в первую очередь, интересны алгоритмы над контейнером `vector` (в этом разделе будем пользоваться `vector <int> v`). На самом деле, мы можем пользоваться обычным массивом, например `int v[100]`. Тогда вместо `v.begin()` следует подставлять `v`, а вместо `v.end()` — `v+n`, где `n` — количество элементов в массиве (не размер массива, а количество осмысленных элементов). Это возможно благодаря тому, что обычный указатель может интерпретироваться как итератор.

При использовании алгоритмов **STL** на векторе следует, по возможности, избегать лишних накладных расходов, связанных с выделениями памяти. В олимпиадных задачах почти всегда либо известно максимально возможное количество элементов (тогда мы можем конструировать вектор константного размера в начале программы), либо количество элементов задается во входных данных (тогда мы можем конструировать вектор после прочтения переменной, задающей количество элементов). В таком случае у нас не будет использоваться лишняя память, а накладные расходы будут очень малы (сравнимы с выделением памяти с помощью `malloc`).

Мы будем рассматривать не все алгоритмы и далеко не все способы работы с ними. В конце этой части будет приведен список других более или менее полезных алгоритмов. Все многообразие алгоритмов и способов работы с ними описано в разделах помощи среды разработки или в `man`. Таким образом, нам достаточно лишь понимать смысл и помнить (хотя бы примерно) название алгоритма.

Начнем с алгоритмов сортировки. В **STL** представлено два алгоритма сортировки: `sort` — осуществляющий быструю сортировку и `stable_sort`, который сохраняет относительный порядок следования одинаковых элементов.

Синтаксис вызова функции сортировки будет такой: `sort(v.begin(), v.end());`

Если данные необходимо отсортировать в обратном порядке, то следует использовать обратный итератор: `sort(v.rbegin(), v.rend());`

Можно сортировать не только массивы целиком, но и их непрерывные части, задавая нужные итераторы (обычно при этом используются итераторы произвольного доступа).

Следующий интересный алгоритм создает кучу максимумов из массива. Его вызов выглядит так: `make_heap(v.begin(), v.end());` Фактически, это то же самое, что первая часть HeapSort (построение кучи). Сложность построения кучи $O(N \log N)$.

После того, как у нас появился heap (он будет находиться в том же самом векторе `v`), мы можем проделать с ним некоторые действия. Например, получить их этой кучи отсортированный массив. Для этого надо вызвать функцию `sort_heap(v.begin(), v.end());` Вектор `v` будет отсортирован. Это то же самое, что второй этап HeapSort.

Кроме того, мы можем добавлять элементы к вектору, являющемуся кучей (вообще говоря, для куч лучше использовать `priority_queue`). Для этого нужно добавить элемент `x` с помощью `v.push_back(x)`, а затем вызвать `push_heap(v.begin(), v.end());`

Чтобы получить значение максимального элемента, нам достаточно разыменовать итератор `v.begin()`. Для удаления максимального элемента из кучи используется функция `pop_heap(v.begin(), v.end());`

Почти все алгоритмы вызываются аналогично — указанием начального и конечного итераторов. Так, например, полезными могут оказаться следующие алгоритмы: `reverse` — записывает последовательность «задом наперед», `random_shuffle` — переставляет элементы в случайном порядке, `is_sorted` и `is_heap` проверяющие, соответственно, является ли массив отсортированным или кучей.

Для поиска порядковой статистики можно воспользоваться функцией `nth_element(v.begin(), v` где `k` — номер искомой порядковой статистики. В среднем поиск имеет линейную сложность (он полностью аналогичен рассмотренному нами в лекции по поиску методу).

Также полезной может оказаться функция бинарного поиска в массиве. При этом массив должен быть упорядочен по невозрастанию. Она вызывается так: `binary_search(v.begin(),` где `i` — искомый элемент.

5.14 Использование собственных структур

До сих пор мы использовали **STL** для элементарных типов или других сравнимых типов. Как же использовать мощь **STL** для своих типов данных (структур, массивов и т.п.)?

Программирующие на **C++** могут сделать это очень просто — создать класс, в котором определить `operator<`. Например, мы можем использовать класс `myint` для кучи минимумов:

```
class myint
{
public:
  int num;
  bool operator<(myint &other)
  {
    if (num > other.num) return true;
    return false;
  }
};
```

Будьте аккуратны, ведь теперь во всех случаях числа типа `myint` будут сравниваться «неправильно» (т.е. «меньше» будет означать, что на самом деле «не меньше»)!

Незнакомые с **C++** могут сделать это не менее просто. Для этого надо описать функцию, принимающую на вход две структуры и возвращающую `true`, если первая «меньше» второй и `false` в противном случае. После этого во всех вызовах **STL** функций следует указывать имя этой функции (без скобок и параметров).

Приведем пример программы, в которой создается набор структур, и они сортируются только по первому полю (довольно частая задача):

```
#include <algorithm>

using namespace std;

typedef struct
{
  int f, s;
} tfs;

bool ls(tfs a, tfs b)
{
  if (a.f < b.f) return true;
  else return false;
}

int main() {
  tfs b[10];
  for (int i=9; i>=0; i--) {
    b[i].f = 10-i;
```

```
    b[i].s = i*i;
  }
  sort(b, b+10, ls);
  return 0;
}
```

Точно так же, переопределяя функцию сравнения, можно добиться построения кучи минимумов, сортировки масива в обратном порядке и т.п.

В конце лекции приведем ссылку на хорошую документацию по **STL**:
`http://www.sgi.com/tech/stl/table_of_contents.html`

5.15 Пример решения задачи с использованием STL

Рассмотрим задачу с Московской олимпиады 2006 года.

Тупики

На вокзале есть K тупиков, куда прибывают электрички. Этот вокзал является их конечной станцией, поэтому электрички, прибыв, некоторое время стоят на вокзале, а потом отправляются в новый рейс (в ту сторону, откуда прибыли).

Дано расписание движения электричек, в котором для каждой электрички указано время ее прибытия, а также время отправления в следующий рейс. Электрички в расписании упорядочены по времени прибытия. Поскольку вокзал — конечная станция, то электричка может стоять на нем довольно долго, в частности, электричка, которая прибывает раньше другой, отправляться обратно может значительно позднее.

Тупики пронумерованы числами от 1 до K. Когда электричка прибывает, ее ставят в свободный тупик с минимальным номером. При этом если электричка из какого-то тупика отправилась в момент времени X, то электричку, которая прибывает в момент времени X, в этот тупик ставить нельзя, а электричку, прибывающую в момент $X+1$ — можно.

Напишите программу, которая по данному расписанию для каждой электрички определит номер тупика, куда прибудет эта электричка.

Формат входных данных

Сначала вводятся число K — количество тупиков и число N — количество электропоездов ($1 \leqslant K \leqslant 500000, 1 \leqslant N \leqslant 500000$). Далее следуют N строк, в каждой из которых записано по 2 числа: время прибытия и время отправления электрички. Время задается натуральным числом, не превышающим 10^9. Никакие две электрички не прибывают в одно и то же время, но при этом несколько электричек могут отправляться в одно и то же время. Также возможно, что какая-нибудь электричка (или даже несколько) отправляются в момент прибытия какой-нибудь другой электрички. Время отправления каждой электрички строго больше времени ее прибытия.

Все электрички упорядочены по времени прибытия. Считается, что в нулевой момент времени все тупики на вокзале свободны.

Формат выходных данных

Выведите N чисел — по одному для каждой электрички: номер тупика, куда прибудет соответствующая электричка. Если тупиков не достаточно для того, чтобы организовать движение электричек согласно расписанию, выведите два числа: первое должно равняться 0 (нулю), а второе содержать номер первой из электричек, которая не сможет прибыть на вокзал.

Примеры

Входные данные	Выходные данные
1 1 2 5	1
1 2 2 5 5 6	0 2
2 3 1 3 2 6 4 5	1 2 1

Приведем разбор задачи, который облегчит понимание решения:

Нам необходимо упорядочить события (приезды и отъезды электричек). Сортировать необходимо по времени события (ключ, по которому строится куча); номеру электрички, с которой произошло событие; и признаку события (приезд или отъезд). В случае, если приезд и отъезд случаются одновременно, сначала должны стоять отъезжающие электрички (в этой задаче удобно прибавить ко времени отъезда единицу).

Кроме того, заведем кучу (минимумов) свободных тупиков. Сначала добавим туда все тупики.

Затем пойдем по событиям. Если событие — приезд электрички, то необходимо взять из кучи свободных тупиков минимальный тупик и сохранить для этой электрички номер тупика, в который она встанет (это удобно делать в массиве, где индекс равен номеру электрички). Если же куча свободных тупиков пуста, то сразу выводим, что расставить электрички нельзя.

Если событие — отъезд электрички, то смотрим, какой тупик она занимала и добавляем этот тупик в кучу свободных тупиков.

Для вывода ответа необходимо просто вывести содержимое массива, в котором сохранялись номера тупиков при приезде электрички.

Решение этой задачи с использованием STL может выглядеть так:

```
#include <stdio.h>
#include <queue>
#include <algorithm>

typedef struct {
  int num, time, etype;
} rail_event;

using namespace std;

#define MAXN 500010

bool ls (rail_event a, rail_event b)
{
  if (a.time < b.time)
    return true;
  else if ((a.time == b.time) && (a.etype < b.etype))
```

```
    return true;
  else return false;
}

int answer[MAXN];
priority_queue <int, vector <int>, greater <int> > bays;
vector <rail_event> events;

int main()
{
  int n, k, i, j, arrival, departure;
  rail_event revent;
  scanf("%d%d", &k, &n);
  for (i=1; i<=k; i++)
    bays.push(i); // все тупики свободны
  for (i=1; i<=n; i++) {
    scanf("%d%d", &arrival, &departure);
    revent.num = i;
    revent.time = arrival;
    revent.etype = 1;
    events.push_back(revent);
    revent.num = i;
    revent.time = departure + 1;
    revent.etype = 0;
    events.push_back(revent);
  }
  sort(events.begin(), events.end(), ls);
  for (i=0; i<events.size(); i++) {
    revent = events[i];
    if (revent.etype == 1) { // прибытие
      if (bays.empty()) {
        printf("0 %d", revent.num);
        return 0;
      } else {
        answer[revent.num] = bays.top();
        bays.pop();
      }
    } else  // отбытие
      bays.push(answer[revent.num]);
  }
  for (i=1; i<=n; i++)
    printf("%d\n", answer[i]);
  return 0;
}
```

Лекция 6

Задачи на анализ таблиц

Версия C от 24.11.2009

6.1 Введение

В олимпиадных задачах довольно часто встречаются задачи, где входные данные заданы в виде таблицы. Под таблицей в данном случае понимается двумерный массив, но приведённые алгоритмы также подходят для трёх и более мерных массивов.

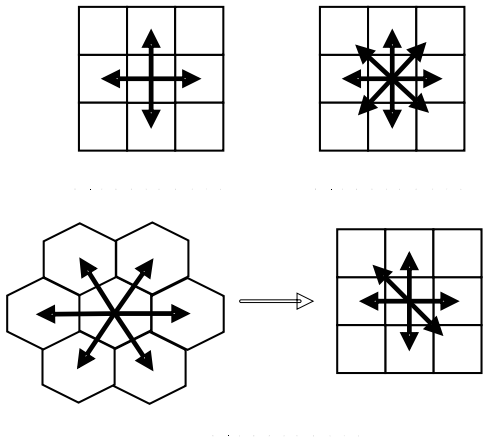


Рис. 6.1: Примеры связности

Введём понятие связности, т.е. определим, какие клетки являются «соседними». Если клетки считаются соседними, когда у них существует общая сторона, то это называется «4-связностью». Если клетки называются соседними, когда у них существует хотя бы одна общая точка, то это «8-связность». Также существуют более сложные способы определения соседей, например, если каждая ячейка представляет собой правильный шестиугольник или треугольник (6- и 3-связность). Иногда «соседи» могут определяться и совсем по другим параметрам, не по общим точкам или сторонам. Например, соседними могут называться клетки, на которые можно перейти ходом шахматного коня.

Почти все типы связности могут быть реализованы на квадратной матрице.

Аналогично можно определить тип связности для трёх и более мерных массивов.

Напомним о ещё одной особенности хранения массивов, которая заключается в том, что в памяти компьютера все многомерные массивы разворачиваются в одномерные. Этот факт мы уже использовали для ускорения работы функции умножения матриц, также его можно использовать для ускорения работы функций работы с таблицами. В общем случае метод выглядит так: если нам необходимо пройти по всем элементам таблицы, то наиболее внешний цикл должен идти по более раннему индексу. Для двумерной таблицы это выглядит так:

```
for (i=0; i<n; i++)
```

```
for (k=0; k<n; k++)
 // какие-то действия с table[i][k]
```

В данной лекции мы рассмотрим несколько характерных задач, которые возникают при обработке таблиц.

6.2 Вычисление по локальным данным

Существует широкий класс задач, где по таблице необходимо подсчитать какие-либо значения для каждой ячейки. Чаще всего значения вычисляются только по данным из текущей и соседних ячеек.

Одна из часто возникающих проблем при решении задач с таблицами — нехватка памяти, для хранения всей таблицы. Таким образом, задачи, которые должны решаться только с использованием локальных данных, обладают тремя характерными признаками:

1. требуется вычисление параметра для каждой ячейки (т.е. ответ также должен быть таблицей или в качестве ответа должна выводиться ячейка, содержащая, например, максимум или минимум),

2. если таблица квадратная размером N, то алгоритмы со сложностью большей, чем $O(N^2)$, не проходят по времени,

3. вся таблица (N^2 ячеек) не укладывается в памяти.

Существует множество самых разнообразных задач на вычисление значений в ячейках по локальным данным, но можно выделить несколько приёмов, позволяющих облегчить их решение.

Первый приём — «барьерный» метод. Суть его заключается в том, что мы «окружаем» таблицу рамкой, содержащей какие-либо допустимые или специальные значения. Этот метод предназначен для того, чтобы избежать обработки частных случаев, связанных с выходом за границы массива. При этом ширина рамки зависит от того, сколько соседей мы используем для подсчёта значения в данной ячейке (чаще всего рамка имеет ширину всего в одну клетку). Рамка никак не должна влиять на решение (т.е. если, например, мы суммируем значения соседей, то рамка должна быть нулевой и т.п.).

Следующий приём является не совсем спортивным. На большинстве олимпиад ввод и вывод осуществляется с помощью файлов. Иногда бывает очень удобно просматривать таблицу несколько раз и накапливать некоторые значения. При этом если таблица не помещается в памяти, мы можем закрывать входной файл и открывать его заново (при этом порядок обхода таблицы изменить нельзя — только так, как она даётся во входных данных). Однако, во многих тестирующих системах программе передаётся уже открытый поток ввода, а следовательно такой приём применить нельзя.

Также во многих задачах требуется знать часть уже вычисленных значений соседей, и исходя из этих данных получать ответ. Здесь довольно важен порядок обхода - обычно он делается сверху вниз построчно, а начинают обход, чаще всего, с какой-либо угловой ячейки.

Рассмотрим примеры задач, где требуется вычисление значений по локальным данным.

Программистика

Всероссийская олимпиада 2002

В Перми становится популярной игра «Программистика».

Для игры требуются плоские квадратные фишки 4-х видов, представляющие собой поле 3×3 с вырезанной центральной клеткой. В остальных клетках каждой фишки записаны числа от 1 до 8. Все виды фишек показаны на рисунке. Количество фишек каждого вида не ограничено.

1	2	3
8		4
7	6	5

7	8	1
6		2
5	4	3

5	6	7
4		8
3	2	1

3	4	5
2		6
1	8	7

Игра проводится на поле размером $N \times N$. Первоначально все клетки поля заполнены единицами.

В начале игры Магистр несколько раз случайным образом помещает произвольные фишки на игровое поле так, что фишка попадает на поле целиком, а ее центральная клетка совпадает с одной из клеток поля. После помещения очередной фишки все числа в восьми клетках игрового поля, которые перекрывает фишка, умножаются на соответствующие числа в клетках фишки, и результаты становятся новыми значениями этих клеток игрового поля.

Таким образом, после окончания процесса размещения фишек игровое поле оказывается заполненным полученными произведениями. Далее Магистр передаёт получившееся поле игроку, которому необходимо установить для каждой клетки поля, сколько раз Магистр в неё помещал центральные клетки фишек.

Требуется для каждого входного файла, содержащего полученное Магистром поле, сформировать соответствующий ему выходной файл, в N строках которого содержится по N чисел, показывающих, сколько раз в соответствующую клетку помещались центральные клетки фишек.

Рассмотрим верхнюю левую клетку игрового поля $(0, 0)$ — обозначим число, записанное в ней, за X. На неё могут оказать влияние только карточки с центром в ячейке $(1, 1)$, причём только своим левым верхним углом (т.е. числами 1, 7, 5 и 3 соответственно). Заметим, что числа 3, 5 и 7 — взаимно просты (их НОД равен 1), а это означает, что разложение числа X на множители 3, 5 и 7 однозначно. Таким образом, мы можем определить количество карточек 2, 3 и 4 типа в клетке $(1, 1)$ как максимальные степени 7, 5 и 3 на которые делится число X. При этом следует не забывать изменять и другие клетки, которые накрывает карточка с центром в ячейке $(1, 1)$.

Чтобы окончательно вычислить ответ в ячейке $(1, 1)$ следует также определить число карточек 1 типа. Заметим, что ячейка $(1, 0)$ (обозначим ее содержимое за Y) изменяется только под воздействием карточек с центром в $(1, 1)$ (умножается на чётное число) и карточек с центром в $(2, 1)$ (умножается на нечётное число). После удаления карточек 2, 3 и 4 типа с центром в $(1, 1)$ умножать на чётное число эту ячейку может только карточка 1 типа с центром в $(1, 1)$ — карточки в $(2, 1)$ чётность изменить не могут. Таким образом, количество карточек 1 типа с центром в $(1, 1)$ определяется как максимальная степень 2, на которую делится число Y (при этом, опять же, следует не забывать пересчитывать значения во всех ячейках, накрываемой данной карточкой).

Тот же метод можно использовать для последовательного вычисления ячеек, двигаясь во внешнем цикле по строкам, а во внутреннем — по столбцам. Предыдущая часть будет уже полностью вычислена и не окажет никакого влияния.

6.3 Кратчайшие пути в лабиринте

Ещё один большой класс задач, где данные задаются таблицей, являются задачи на поиск путей в лабиринте. При этом обычно лабиринт состоит из проходимых клеток, стен (непроходимых клеток), начальной и конечной позиции.

Рассмотрим метод поиска кратчайшего пути, который называется «волновым алгоритмом» или обходом в ширину. Такое название он получил за то, что движение происходит «волной» от начальной позиции (т.е. равномерно во все стороны) и как только волна доходит до конечной позиции, мы прекращаем работу алгоритма и восстанавливаем ответ.

Для использования алгоритма нам понадобится массив $N \times N$ для хранения длины пути в каждой ячейке (на самом деле, во многих случаях можно воспользоваться тем же массивом, в котором задаётся сам лабиринт). Кроме того, нам понадобится очередь, в которой будут храниться координаты клеток текущего шага «волны».

Поместим в начальную клетку 0 (т.е. мы можем добраться в неё за 0 шагов), проходимые клетки будем помечать, например, -1, а непроходимые -2. Занесём в очередь координаты начальной клетки и начнём выполнение алгоритма.

Алгоритм состоит в следующем: пока очередь не пуста, берём клетку из ее начала и рассматриваем ее соседей. Если соседняя клетка помечена как проходимая и ещё не посещённая (т.е. в ней находится -1), то записываем в неё число на 1 большее, чем в текущей клетке (нам понадобился ещё один шаг) и заносим ее в очередь.

0		6	7	8
1		5		9
2	3	4		8
3		5	6	7

Собственно, это и есть весь алгоритм. В конечной клетке будет записано число шагов на пути до неё от начала. Если же нам необходимо восстановить ещё и путь, то необходимо перемещаться из конечной клетки в любую соседнюю, в которой записан номер на 1 меньший, чем в текущей.

Т.е. путь будет восстановлен в обратном порядке.

Возможный вид таблицы и очереди после работы алгоритма (строка N — номер шага, в самой очереди не используется, т.к. номер шага хранится в таблице и приведена для пояснения):

X	0	0	0	0	1	2	2	2	3	2	4	3	4	4	4
Y	0	1	2	3	2	2	3	1	3	0	3	0	2	0	1
N	0	1	2	3	3	4	5	5	6	6	7	7	8	8	9

В этом алгоритме мы не учитывали, что у граничных клеток нет части соседей. Этот частный случай легко обойти, если создать вокруг лабиринта барьер из непроходимых клеток. В некоторых задачах, наоборот, можно обходить лабиринт снаружи — тогда барьер должен быть проходимым.

Ещё один вариант задач с лабиринтами — когда у нас нет непроходимых клеток, но есть тонкие стенки между клетками (у одной клетки максимум 4 окружающие ее стенки). В таком случае нам нужно для каждой клетки помнить, какие у неё есть стенки. Удобнее всего делать это с помощью 4 битов — каждый бит отвечает за одну стенку:

Обычно такого рода лабиринты задаются списком стен. Чтобы получить готовое к употреблению описание лабиринта необходимо: во-первых «обнести» весь лабиринт стенками (установить для каждой граничной ячейки соответствующий бит, а для угловых — 2 бита), а во-вторых — расставить стенки (изменить по одному биту в двух

ячейках, которые разделяет данная стенка). Аналогично можно реализовать и любые другие лабиринты (с 6, 8-связностью и даже более сложные) — необходимо просто увеличить количество бит для описания стен.

Теперь на каждом шаге волнового алгоритма нам надо проверять, что соседняя ячейка не занята и две ячейки не разделяет стенка (смотреть соответствующий соседней ячейке бит). В остальной части алгоритм останется без изменений. При восстановлении пути также надо следить за тем, чтобы не ходить через стены.

Существует класс задач, в которых необходимо найти не кратчайший путь от одной клетки до остальных, а наоборот — кратчайшие пути от многих клеток до одной. Здесь можно воспользоваться инвертированием, т.е. найти пути от одной клетки до всех одним обходом в ширину, а потом просто развернуть пути в обратную сторону. Аналогично можно поступить и в случае если существует много начальных клеток и много конечных — волновые алгоритмы следует запускать из тех клеток, количества которых меньше (т.е. если начальных клеток меньше — запускаем волны оттуда и наоборот в противном случае), затем для каждой клетки следует выбрать минимум.

6.4 Система непересекающихся множеств

Перед рассмотрением следующего алгоритма работы с таблицами необходимо отвлечься и освоить одну несложную структуру данных — систему непересекающихся множеств.

В олимпиадных задачах довольно часто требуется разбить набор объектов на непересекающиеся множества (т.е. каждый объект может лежать только в одном множестве, но в одном множестве может находиться несколько объектов). Пусть объекты пронумерованы от 0 до $N-1$. В качестве идентификатора множества будем использовать также числа от 0 до $N-1$. При инициализации, обычно, все множества состоят из одного элемента, т.е. объект с номером i лежит во множестве с номером i.

Для системы непересекающихся множеств определены две операции: `fset(x)`, возвращающая номер множества, в котором лежит элемент и операция `sunion(x, y)`, объединяющая множества, содержащие элементы x и y в одно. Первая операция используется, обычно, для проверки того, лежат ли два элемента в одном множестве или в разных.

Эффективность различных структур будем оценивать как количество операций, необходимое для объединения N множеств в одно в наихудшем для данной структуры случае.

Самым простым способом реализации является одномерный массив, где индекс задаёт номер объекта, а значение - номер множества, в котором этот объект находится. Проверка будет просто возвращать значение из запрошенной ячейки, а объединение должно проходить по всему массиву и заменять все числа на (или наоборот). В худшем случае каждый раз мы будем добавлять по одному элементу, таким образом, нам потребуется N проходов по массиву и сложность составит $O(N^2)$.

Рассмотрим более эффективную реализацию, где кроме массива будет храниться также список элементов множества (для списка мы будем хранить указатели на начало и конец). Этот способ требует $O(N)$ дополнительной памяти. При проверке мы будем также возвращать значение из массива, а при модификации проходить по одному из списков и менять значения в массиве, а потом прикреплять этот список к концу

другого. Если мы будем делать это бездумно, то сложность также составит $O(N^2)$ — это случай, когда каждый раз длинный список будет прикрепляться к концу списка длины 1. Если ввести для каждого списка такое поле, как длина списка (оно легко пересчитывается при объединении) и приписывать каждый раз более короткий список, то для последовательного объединения всех списков длины 1 сложность получится $O(N)$. Однако, если каждый раз длины списков будут равны (худший для нас случай), мы получаем сложность $O(N \log N)$. На каждом шаге будут объединяться все пары списков равной длины, каждый раз длина списка будет увеличиваться вдвое, следовательно, общее количество шагов будет составлять $\log N$.

Рассмотрим также реализацию на стягивающихся корневых деревьях. Для каждого множества выберем элемент, который назовём «представителем». Представитель указывает сам на себя, и сначала все элементы указывают сами на себя. Для каждого дерева введём такое понятие, как высота, которое сродни длине списка и более «низкое» дерево следует прикреплять к более высокому, а в случае, если высоты были равны, высота результирующего дерева увеличится на единицу. Требования к дополнительной памяти составят $O(N)$.

Функция поиска должна идти наверх, пока не дойдёт до представителя, а функция объединения вызывает две функции поиска, а затем уже объединяет деревья. Казалось бы, такой способ не даёт никаких особых преимуществ перед списками, однако можно модифицировать функцию поиска так, чтобы при проходе по всем элементам пути до представителя она, получив указатель на представителя, для всех элементов на пути устанавливала указатель непосредственно на представителя. Это требует ещё $O(\log N)$ дополнительной памяти (эта оценка сильно завышена). Приведём реализацию системы непересекающихся множеств:

```
typedef struct {
  int rank, p;
} syst;

void init(syst *a, int n) {
  int i;
  for (i = 0; i<n; i++) {
    a[i].p = i;
    a[i].rank = 0;
  }
}

int fset(syst *a, int x) {
  if (x != a[x].p)
    a[x].p = fset(a, a[x].p);
  return a[x].p;
}
void sunion(syst *a, int x, int y) {
  if (a[fset(a, x)].rank < a[fset(a, y)].rank)
    a[fset(a, x)].p = a[fset(a, y)].p;
  else {
    a[fset(a, y)].p = a[fset(a, x)].p;
```

```
    if (a[fset(a, x)].rank == a[fset(a, y)].rank)
      a[fset(a, x)].rank++;
  }
}
```

Сложность объединения всех множеств в одно практически линейна. Оставим без доказательства этот факт и точную оценку, однако интуитивно понятно, почему операция `fset` будет выполняться с каждым разом все быстрее.

6.5 Выделение связных областей

На олимпиадах достаточно часто предлагаются задачи, в которых необходимо выделить в таблице связные области (т.е. каждой группе соприкасающихся «закрашенных» клеток присвоить уникальный номер) и модификации таких задач. Существует несколько способов решения данной задачи, которые различаются по сложности реализации и требованиям к памяти.

Простейший способ — «обход в глубину», когда мы используем рекурсивную функцию, которая помечает клетку, как уже просмотренную и вызывает себя для всех ещё не просмотренных закрашенных соседей. Пишется такая функция очень просто, но в худшем случае (когда закрашено все поле), количество рекурсивных вызовов будет N^2, что очень много. В целом алгоритм выделения связных областей обходом в глубины выглядит так: мы проходим всю таблицу и если встречаем закрашенную не помеченную клетку, то вызываем для неё рекурсивную функцию с новым номером (который будет номером данной области). Сложность этого алгоритма будет составлять $O(N^2)$.

Следующий способ — использование обхода в ширину, модификации волнового алгоритма. Отличается от предыдущего он только тем, что вместо вызова рекурсивной функции, новая клетка добавляется в очередь. При этом помечать клетку следует вместе с помещением в очередь (чтобы не возникало ситуаций, когда одна и та же клетка попала в очередь 2 или более раз). Требования по дополнительной памяти (для хранения очереди) у этого алгоритма будут $O(N)$.

Рассмотрим ещё одну реализацию поиска связных областей с помощью последовательного сканирования. В этом алгоритме мы будем просматривать таблицу сверху вниз, на каждом шаге нам известна уже размеченная строка (на первом шаге — строка, состоящая из не закрашенных клеток, барьер) и по ней мы будем размечать следующую, ещё не размеченную строку. Процесс разметки происходит следующим образом: мы идём по уже размеченной строке, если нам попалась клетка, относящаяся к какой-то области (X), то помечаем все не размеченные клетки под ней в обе стороны, как относящиеся к той же области. Если в процессе разметки клеток нижней строки над какой-либо клеткой оказалась клетка,

0	1	0	0	2	0	3	0	0	0
0	−1	−1	−1	−1	0	−1	0	−1	0
Шаг 1: нижнаяя строка не размечена									
0	1	0	0	2	0	3	0	0	0
0	*1*	*1*	*1*	*1*	0	−1	0	−1	0
Шаг 2: размечаем область 1 в новой строке. Области 1 и 2 объединяются									
0	1	0	0	2	0	3	0	0	0
0	1	1	1	1	0	*3*	0	−1	0
Шаг 3: размечаем область 3 в новой строке									
0	1	0	0	2	0	3	0	0	0
0	1	1	1	1	0	3	0	*4*	0
Шаг 4: неразмеченная ячейка попадает в новую область 4									

помеченная, как относящаяся к другой области (Y), то области X и Y следует объединить в одну (эта ситуация возникает, например, если мы размечаем перевёрнутую букву П — в верхних строках у нас будут две разные области, а затем они объединяться перемычкой). После того, как мы выполнили проход по верхней строке, в нижней могут остаться ещё не размеченные клетки — для них следует создать новые области и разметить их. При этом смежные клетки следует помещать в одну область.

Для реализации областей (проверки принадлежности одной области и объединения областей) удобно использовать систему непересекающихся множеств, реализованную с помощью стягивающихся корневых деревьев.

```
int main(void)
{
  syst regions[MAXN*MAXN/2];
  int table[MAXN+2][MAXN+2];
  int i, j, k, n;
  int nowreg=1;
  scanf("%d", &n);
//Инициализация системы множеств
  init(regions, n*n/2);
//Создание барьера
  for (i=0; i<n+1; i++)
    table[i][0] = table[i][n+1] = table[0][i] = table[n+1][i] = 0;
//Считывание и преобразование входных данных
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++) {
      scanf("%d", &table[i][j]);
      if (table[i][j] == 1) table[i][j] = -1;
    }
//Обход таблицы
  for (i=0; i < n+1; i++) {
    for (j=0; j<n+1; j++) {
      if (table[i][j] > 0 && table[i+1][j] == -1) {
        k=j;
//Размечаем нижнюю строку налево
        while (table[i+1][k] == -1) {
          table[i+1][k] = fset(regions, table[i][j]);
          if (table[i][k] > 0 && fset(regions, table[i][k])
          != table[i+1][k])
            sunion(regions, table[i][k], table[i+1][k]);
          k++;
        }
        k=j-1;
//... и направо
        while (table[i+1][k] == -1) {
          table[i+1][k] = fset(regions, table[i][j]);
          if (table[i][k] > 0 && fset(regions, table[i][k])
          != table[i+1][k])
```

```
            sunion(regions, table[i][k], table[i+1][k]);
          k--;
        }
      }
    }
//Размечаем новые области
    for (j=0; j<n+1; j++)
      if (table[i+1][j] == -1) {
        k = j;
        while (table[i+1][k] == -1)
        table[i+1][k++] = nowreg;
        nowreg++;
      }
    }
//Вывод результатов разметки
  for (i=1; i <= n; i++) {
    for (j=1; j <= n; j++)
      if (table[i][j] != 0)
        printf("%d ", fset(regions, table[i][j]));
      else
        printf("0 ");
      printf("\n");
  }
  return 0;
}
```

Приведённая выше программа основывается на использовании системы непересекающихся множеств.

Отметим, что на самом деле, нам не нужно хранить всю таблицу для использования метода сканирования — достаточно только 2 строк. Однако это нельзя использовать даже для непосредственного вывода результатов (опять же пример с перевёрнутой П). Пользоваться тем, что мы можем не хранить всю таблицу можно только в задачах, где необходимо подсчитать некоторые параметры связных областей (например, количество элементов в максимальном и минимальном множестве), но не при непосредственном выводе разметки.

Максимальное количество связных множеств на таблице $N \times N$ составляет $\lceil N/2 \times N/2 \rceil$, но при использовании сканирования нам важны только множества в двух строках, т.е. достаточно поддерживать список свободных номеров множеств и использовать только $2 \times \lceil N/2 \rceil$ дополнительной памяти.

Метод сканирования можно также использовать и для решения других задач, которые, на первый взгляд, должны решаться другим методом. В частности, можно проверять, связаны ли две клетки (т.е. между ними существует путь в лабиринте). Для этого достаточно проверить, что они лежат в одной и той же связной области. Метод сканирования применим там, где невозможно поместить в память всю таблицу — в таком случае следует искать его модификацию, решающую конкретную задачу.

Лекция 7

Динамическое программирование с одним параметром

Версия C от 30.11.2009

7.1 Введение

Динамическим программированием называется метод, который позволяет решать «переборные» задачи, опираясь на уже решённые подзадачи меньшего размера. При этом необходимо, чтобы все решения можно было запомнить в таблице (т.е. данные, посчитанные однажды, не пересчитываются).

Идеи динамического программирования очень похожи на идеи метода доказательства по индукции, известного из школьной программы математики. Сформулируем необходимые требования:

1. Все решения подзадач должны запоминаться в таблице.
2. Существует известные решения для задачи с малой размерностью (аналогично проверке для минимального параметра в методе математической индукции).
3. Существует способ выразить решение задачи через подзадачи (возможно, отличные от исходной задачи) меньшей размерности. Это больше всего напоминает рекуррентное соотношение в математике.

Требование меньшей размерности означает, что значение параметра (или одного из параметров, если их несколько) в подзадаче должно быть меньше, чем значение параметра в задаче и в итоге последовательность подзадач должна сходиться к известным решениям (нулевому параметру, например). Если параметры подзадач «уходят в бесконечность» или возникает круговая зависимость — это означает, что метод динамического программирования неприменим. В некоторых случаях разумно подсчитать сумму всех параметров и требовать, чтобы на каждом шаге эта сумма уменьшалась (однако это не всегда возможно).

Задачи динамического программирования делятся на два типа: оптимизация целевой функции и подсчёт количества вариантов решения. В первом случае нам необходимо выбрать лучшее среди всех решений подзадач, во втором — просуммировать все количества решений подзадач.

В случае если ищется решение, иногда бывает необходимо подсчитать не только сам ответ, но и восстановить решение полностью (записать, какие действия выполнялись на каждом шаге).

7.2 Оптимизация целевой функции

К задачам оптимизации целевой функции для заданного числа подзадач относятся те задачи, в которых решение можно найти исходя из знания решений нескольких предыдущих задач. Будем рассматривать примеры исходя из приведённой выше схемы.

Покупка билетов

Московская олимпиада 2003

За билетами на премьеру нового мюзикла выстроилась очередь из N человек, каждый из которых хочет купить 1 билет. На всю очередь работала только одна касса, поэтому продажа билетов шла очень медленно, приводя «постояльцев» очереди в отчаяние. Самые сообразительные быстро заметили, что, как правило, несколько билетов в одни руки кассир продаёт быстрее, чем когда эти же билеты продаются по одному. Поэтому они предложили нескольким подряд стоящим людям отдавать деньги первому из них, чтобы он купил билеты на всех.

Однако для борьбы со спекулянтами кассир продавала не более 3-х билетов в одни руки, поэтому договориться таким образом между собой могли лишь 2 или 3 подряд стоящих человека.

Известно, что на продажу i-му человеку из очереди одного билета кассир тратит A_i секунд, на продажу двух билетов — B_i секунд, трех билетов — C_i секунд. Напишите программу, которая подсчитает минимальное время, за которое могли быть обслужены все покупатели.

Обратите внимание, что билеты на группу объединившихся людей всегда покупает первый из них. Также никто в целях ускорения не покупает лишних билетов (то есть билетов, которые никому не нужны).

Во входном файле записано сначала число N — количество покупателей в очереди ($1 \le N \le 5000$). Далее идет N троек натуральных чисел A_i, B_i, C_i. Каждое из этих чисел не превышает 3600. Люди в очереди нумеруются начиная от кассы.

В выходной файл выведите одно число — минимальное время в секундах, за которое могли быть обслужены все покупатели.

Будем искать решение по приведённой выше схеме.

Создадим одномерный массив (X), длиной 5000, в i-ой ячейке которого будем запоминать время, необходимое для покупки билетов всеми людьми до i-го включительно.

Далее — начальные данные. Создадим в массиве барьер — трёх фиктивных людей с номерами 0, -1 и -2; в соответствующие ячейки заполним нулями (время покупки билетов фиктивными людьми — 0 секунд). Заполним соответствующие ячейки массивов A, B и C «бесконечностью», которой в данном случае может быть любое число больше 3600 (фиктивные люди покупают билеты очень медленно, это необходимо, чтобы они никогда не участвовали в решении).

Теперь запишем оптимизирующую функцию. Пусть заполнены все ячейки до K-ой, не включая ее. Нам необходимо выразить $X[K]$ через уже решённые подзадачи. Для каждого человека (если считать, что за ним очереди нет) существует всего 3 варианта приобрести билет: либо все до него покупают билет, и он берет только для себя; либо все,

кроме предыдущего покупают себе билета, а предыдущий берет два билета; либо все, кроме двух предыдущих покупают себе билеты, а стоящий на позиции $K-2$ покупает 3 билета. Т.к. требуется посчитать минимальное время, то формула будет выглядеть следующим образом:

$X[K] = min(X[K-1] + A_k, X[K-2] + B_{k-1}, X[K-3] + C_{k-2})$

Таким образом, для решения нам необходимо инициализировать массив X фиктивными людьми, считать массивы A, B и C, и прогнать цикл по K от 1 до N. Ответ будет содержаться в ячейке $X[N]$.

Сложность такого решения линейна.

Рассмотрим задачи, где для получения ответа требуются все уже решённые задачи меньшего размера.

Для примера возьмём классическую задачу динамического программирования: задачу о наибольшей возрастающей подпоследовательности.

Условие ее такого:

Дана последовательность из N чисел. Требуется найти длину наибольшей возрастающей подпоследовательность. Элементы в подпоследовательности не обязательно идут подряд (т.е. часть элементов исходной последовательности можно «выбросить») и каждый элемент подпоследовательности должен быть больше предыдущих.

Будем действовать по уже выработанной схеме.

Создадим одномерный массив (X) длины N, в K-ой ячейке которого будем хранить максимальную длину возрастающей подпоследовательности, последним элементом которой является элемент с номером K.

В данной задаче не будем создавать барьер, а подберём такую оптимизирующую функцию, которая сможет обойтись без него.

Сама оптимизирующая функция будет выглядеть следующим образом: среди всех предыдущих чисел, меньших текущего, выберем то, для которого длина наибольшей возрастающей подпоследовательности максимальна. Если таких чисел не нашлось, то в $X[K]$ запишем 1 (подпоследовательность состоит из одного числа).

$X[K] = \max_{i=0,k-1}(X[i]|X[K] > X[i]) + 1$

Как уже было описано выше, в случае, если таких чисел не нашлось, функция max должна возвращать 0.

Чтобы получить ответ, необходимо просмотреть все элементы массива X и выбрать среди них максимальный.

Данное решение верно, т.к. мы перебираем все варианты (но не пересчитываем каждый раз решение подзадачи).

Сложность такого решения будет $O(N^2)$.

7.3 Восстановление решения

В предыдущих задачах мы только находили ответ, но не восстанавливали самого решения. Например, в задаче о покупке билетов могло бы требоваться указать, сколько билетов купил какой человек. А в задаче о поиске наибольшей возрастающей подпоследовательности — указать элементы этой подпоследовательности.

Обычно восстановление проводится задом наперёд, т.е. от ответа к начальной позиции и, при необходимости, переворачивается. Для восстановления используются либо

циклы, которые «перепрыгивают» некоторые элементы, двигаясь от конца к началу или рекурсивные функции.

Так, в задаче о наибольшей возрастающей подпоследовательности для того, чтобы восстановить ее необходимо кроме длины хранить ещё и номер предыдущего элемента (а для первого элемента в подпоследовательности, когда длина равна 1, некий специальный признак, например, число -1). Затем, после того, как мы найдём элемент с ответом, получим решение, если будем двигаться, начиная с его номера и переходя на предыдущий до тех пор, пока не достигли -1. Ответ будет получен в обратном порядке, существует два варианта — записать во вспомогательный массив и вывести его с конца, либо использовать рекурсивную функцию перехода на предыдущий и выводить ответ на выходе из функции. Стоит отметить, что рекурсия работает медленнее и занимает больше памяти.

Рассмотрим, в качестве примера, ещё одну классическую задачу динамического программирования: дискретную задачу о рюкзаке.

Дан набор, состоящий из N неделимых предметов (т.е. разделять каждый предмет на части нельзя), для каждого из которого известен его вес M_i и стоимость S_i. Максимальная грузоподъёмности рюкзака — L. Требуется унести в рюкзаке предметы, суммарная стоимость которых максимальна.

Для решения этой задачи создадим массив (X) длины L, K-я ячейка которого представляет собой структуру и хранит:

1. Максимальную суммарную стоимость вещей на данный момент (поле sum). Общий вес вещей K

2. Какая вещь была добавлена последней (поле num).

Инициализируем массив следующим образом: в $X[0]$ (нулевой суммарный вес) запишем фиктивную вещь с номером -1 и суммарной стоимостью 0. Во все остальные ячейки запишем в качестве номера вещи -2 — признак того, что набор такого веса составить невозможно.

Сделаем динамику «по вещам», т.е. будем искать решение задачи для J первых вещей исходя из знания оптимального решения для $J-1$ первой вещи. Для 0 вещей наше утверждение выполняется. На каждом шаге будем пытаться добавить очередную вещь ко всем уже составленным наборам. Если набор суммарным весом, равным сумме веса очередной вещи и веса дополняющего ее до текущего веса набора, не существовал или его стоимость была меньше, чем новая, то заменяем его.

```
for (j=0; j<N; j++) { //цикл по вещам
  for (i=L-M[j]; i>=0; i--) //цикл по весам
    if (X[i].num != -1)
//если набор веса i (дополнительный) существует
      if ((X[i+M[j]].num == -2) || (X[i+M[j]].sum < S[j] + X[i].sum)) {
//если набор не существовал или был хуже нового, то заменяем
        X[i+M[j]].sum = S[j] + X[i].sum;
        X[i+M[j]].num = j;
      }
}
```

Особенность этого метода — проход с конца массива к началу. Это необходимо, чтобы одна и та же вещь не вошла в набор два раза. Проиллюстрируем на примере. Пусть у нас имеются вещи со следующими парами вес-стоимость (1, 3), (3, 2), (4, 6), (2, 2) и рюкзак грузоподъёмностью 6.

Вес	0	1	2	3	4	5	6
Шаг 1	−1 0	*1 3*	−2 0	−2 0	−2 0	−2 0	−2 0
Шаг 2	−1 0	1 3	−2 0	*2 2*	*2 5*	−2 0	−2 0
Шаг 3	−1 0	1 3	−2 0	2 2	*3 6*	*3 9*	−2 0
Шаг 4	−1 0	1 3	*4 2*	*4 5*	3 6	**3** **9**	*4 8*

Первое число в столбце — номер вещи, второе — суммарный вес. Курсивом помечена обновлённая информация, жирным шрифтом — те ячейки, где проводилось сравнение, но замена не произошла (результат не был улучшен).

Для того, чтобы получить ответ, нам необходимо найти в массиве элемент с максимальным полем sum. Восстановление начнём с этого поля, двигаясь назад. А именно — запомним или выведем поле num той ячейки, где находимся сейчас. Это означает, что вещь с номером num была положена последней и чтобы восстановить предыдущую часть необходимо отнять ее вес и продолжать это до тех пор, пока не дойдём до фиктивной вещи с номером -1.

Разложение числа

Московская олимпиада 2006

Учительница математики попросила школьников составить арифметическое выражение, так чтобы его значение было равно данному числу N, и записать его в тетради. В выражении могут быть использованы натуральные числа, не превосходящие K, операции сложения и умножения, а также скобки. Петя очень не любит писать, и хочет придумать выражение, содержащее как можно меньше символов. Напишите программу, которая поможет ему в этом.

В первой строке входного файла записаны два натуральных числа: N ($1 \le N \le 10000$) — значение выражения и K ($1 \le K \le 10000$) — наибольшее число, которое разрешается использовать в выражении.

В единственной строке выходного файла выведите выражение с данным значением, записывающееся наименьшим возможным количеством символов.

Если решений несколько, выведите любое из них.

При подсчёте длины выражения учитываются все символы: цифры, знаки операций, скобки.

Заметим, что для решения представления числа i существует три варианта: требуемое число может быть записано непосредственно ($i \le K$, требуемое количество символов — количество цифр в числе), число может быть представлено в виде суммы или произведения двух меньших чисел. Будем считать множитель и просто запись числа одним и тем же (в этом случае скобки не требуются). В случае если мы используем произведение двух чисел, хотя бы одно из которых представляется суммой, то это число должно быть окружено скобками.

Перед тем, как восстанавливать выражение, научимся считать его длину. Для этого нам потребуется два вспомогательных массива, один (m) из которых будет содержать длину выражения этого числа как множителя (непосредственно число или произведение), а второй (s) — длину выражения этого числа через сумму чисел.

Вначале заполним массивы «бесконечностями» (в нашем случае это может быть число $n+2$, когда число представляется в виде суммы единиц). Затем заполним элементы

массива m с индексами от 1 до k длиной числа (для этого есть несколько способов: подсчёт длины строки, вычисление десятичного логарифма или просто запоминание).

После этого наступает черёд непосредственно динамического программирования. Будем пытаться для каждого числа (i) из диапазона от $k+1$ до n найти наилучшее представление его в виде произведения и в виде суммы. Для представления в виде суммы нам необходимо перебрать все пары слагаемых j и p, таких, что $j \le p$ и $j+p=i$, и выбрать среди них наилучшую. При этом для каждого из чисел j и p необходимо выбирать минимум из представления в виде суммы и виде произведения. Сами эти числа необходимо запомнить, т.е., например, записать в специальные массивы $s1[n]=j$ и $s2[n]=p$. Эта информация сильно облегчит восстановление решения.

Кроме того, для числа i нужно найти наилучшее представление в виде произведения. Для этого будем перебирать все числа j, начиная с 2 и пока $j^2 \le i$. Для каждого j будем находить дополнительный множитель p ($p=i/j$), такой, что $j \times p = i$ (т.е. число i делится на j нацело). Естественно, что для чисел j и p также нужно выбирать наилучшее из представлений. Однако в случае произведения возникает небольшое отличие: если сомножитель представляется в виде суммы, необходимо окружить его скобками. Таким образом, к представлению сомножителя (каждого из чисел p и j) в виде суммы, необходимо прибавлять 2 (т.е. сравнивать между собой $s[j]+2$ и $m[j]$). Лучшее разбиение на множители также нужно запоминать ($m1[n]=j$ и $m2[n]=p$).

После построения таких массивов мы уже знаем длину ответа ($min(s[n],m[n])$) и нам необходимо восстановить его. Для восстановления воспользуемся рекурсивной функцией, принимающей на вход два параметра: число, которое нужно восстановить и признак того, является ли восстанавливаемое число сомножителем (это необходимо для правильной расстановки скобок). При первом запуске число не будет являться сомножителем (ответ не надо окружать скобками). Сама функция должна рассматривать три основных случая: если восстанавливаемое число не превышает k, то просто выводим это число; если наше число наилучшим образом представимо произведением, то вызываем восстановление для первого сомножителя, печатаем знак «*» и вызываем функцию печати второго сомножителя (в этом случае в функции должны передаваться признаки того, что восстанавливаемое число является сомножителем); в случае же если число наилучшим образом представимо в виде суммы (здесь также два случая: нужны ли скобки или нет; если нужны, то прибавляем к длине 2 и ставим эти скобки вокруг выражения), то вызываем восстановление для первого операнда, ставим между ними знак «+» и восстанавливаем второй операнд (в этом случае признак сомножителя передавать не надо — скобки не нужны).

Существует и другой вариант решения, при котором представление сразу накапливается в виде строки (`string`), а в конце работы просто выводится.

При решении этой задачи необходимо писать достаточно аккуратную реализацию, т.к. ее асимптотическая сложность $O(N^2)$, что в нашем случае равно 10^8. Поэтому некоторая сложность восстановления оправдана уверенностью в производительности такого метода (по сравнению с операциями над динамически выделяемыми строками).

7.4 Подсчёт числа ответов

Задачи подсчёта числа ответов практически не отличаются от задач поиска ответа. Однако вместо функции, выбирающей оптимальное решение, здесь используется сложе-

ние или умножение, в зависимости от реализации. Следует отметить, что в большинстве такого рода задач для хранения количества вариантов требуются длинная арифметика, поэтому перед реализацией стоит оценить возможную длину максимального ответа, для того, чтобы получить требуемую длину числа. Операция сложения двух чисел, максимальное из которых равно N, требует $O(\log N)$ времени (т.е. пропорционально количеству знаков), а длинное умножение $O(\log^2 N)$.

Рассмотрим классическую математическую задачу о числах Фибоначчи, известную ещё в Средневековье. В оригинале это задача о кроликах, которые, как и положено кроликам, плодятся и через некоторое время умирают. Задача состоит в том, чтобы определить количество кроликов через некоторый промежуток времени.

Если переформулировать ее в математических терминах, то получим рекуррентное соотношение $F_i = F_{i-1} + F_{i-2}$ и $F_0 = 1$, $F_1 = 1$. Задача состоит в том, чтобы найти F_N по заданному N.

Для вычисления N-го числа Фибоначчи существует формула, однако она непригодна для компьютерного использования.

Оценим скорость роста чисел Фибоначчи. Заметим, что числа возрастают и $2 \times Fi \geq F_{i+1}$. Таким образом, можно сделать оценку $F_N \leq 2^N$. Отсюда получим, что операция сложения двух длинных чисел Фибоначчи будет занимать $O(\log N)$.

Если пользоваться формулой, приведённой в определении (т.е. просто складывать последовательно пары подряд идущих чисел для получения следующего), то общая сложность с учётом расходов на длинную арифметику получится $O(N \log N)$.

Стоит заметить, что при этом вовсе не обязательно хранить N длинных чисел, достаточно только 3: для двух предыдущих чисел и результата сложения.

Т.к. задачи, требующие подсчёта чисел Фибоначчи, встречаются достаточно часто, постараемся найти более быстрый способ их подсчёта.

Рассмотрим матрицу $\left(\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right)$. Верхний левый элемент в ней соответствует первому числу Фибоначчи. Возведём эту матрицу в квадрат: $\left(\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right)^2 = \left(\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right)$, теперь в верхнем левом углу матрицы стоит уже второе число Фибоначчи. Аналогично для 3-го числа: $\left(\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right) \times \left(\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right) = \left(\begin{smallmatrix}3&2\\2&1\end{smallmatrix}\right)$. Вообще говоря, выполняется следующее соотношение:

$$\begin{pmatrix}1&1\\1&0\end{pmatrix} \times \begin{pmatrix}F_{i-1}&F_{i-2}\\F_{i-2}&F_{i-3}\end{pmatrix} = \begin{pmatrix}F_{i-1}+F_{i-2}&F_{i-1}\\F_{i-1}&F_{i-2}\end{pmatrix} = \begin{pmatrix}1&1\\1&0\end{pmatrix}$$

Т.к. за начальную матрицу мы принимаем также матрицу, состоящую из первых чисел Фибоначчи, то можно получить следующее утверждение:

$$\begin{pmatrix}1&1\\1&0\end{pmatrix}^N = \begin{pmatrix}F_N&F_{N-1}\\F_{N-1}&F_{N-2}\end{pmatrix}$$

Если просто возводить матрицу в степень, то мы получим сложность $O(N \log^2 N)$. Член $log^2 N$ возник здесь из-за необходимости длинного умножения.

Однако мы знаем способ быстрого возведения в степень для чисел, который также применим и для матриц. Т.о. мы можем добиться сложности $O(\log^3 N)$.

На практике скорость работы выглядит так (усреднённые значения по 5 экспериментам):

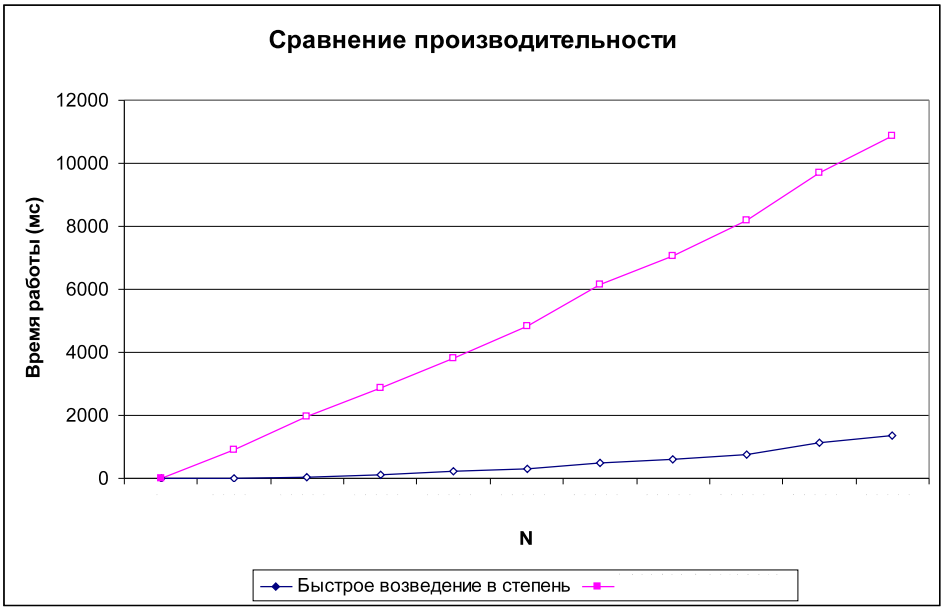


Рис. 7.1: Сравнение производительности

Лекция 8

Динамическое программирование с двумя параметрами

Версия C от 10.12.2009

8.1 Введение

Динамическое программирование с двумя или более параметрами почти не отличается от динамического программирования с одним параметром: в нем также встречаются задачи, в которых требуется подсчёт количества решений и нахождение оптимального решения. Также задачи можно классифицировать по количеству используемых решений меньших подзадач: задачи, использующие одну или несколько меньших подзадач; задачи, использующие все задачи меньшего размера (динамика по подмножествам) и частный случай — LR-динамика (динамика по подстрокам).

Приёмы динамического программирования будут рассматриваться на примерах реальных олимпиадных и классических задачах.

8.2 Использование нескольких подзадач

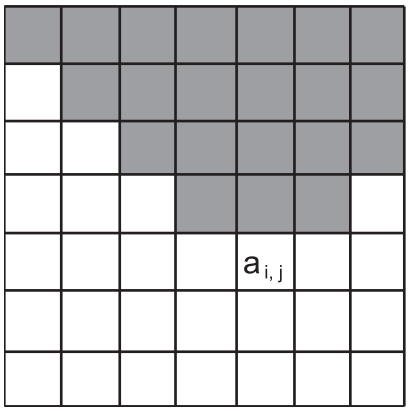
Аналогично одномерной динамике, существуют задачи использующие лишь несколько подзадач меньшего размера. Рассмотрим такие задачи на примерах.

Таблица

4-й этап Всероссийской олимпиады 2008, запад

Рассмотрим прямоугольную таблицу размером $n \times m$. Занумеруем строки таблицы числами от 1 до n, а столбцы — числами от 1 до m. Будем такую таблицу последовательно заполнять числами следующим образом.

Обозначим через a_{ij} число, стоящее на пересечении i-ой строки и j-ого столбца. Первая строка таблицы заполняется заданными числами — a_{11}, a_{12}, ..., a_{1m}. Затем заполняются строки с номерами от 2 до n. Число a_{ij} вычисляется как сумма всех чисел таблицы, находящихся в «треугольнике» над элементом a_{ij}. Все вычисления при этом выполняются по модулю r.



Более точно, значение a_{ij} вычисляется по следующей формуле:

$$a_{ij} = \left(\sum_{k=1}^{i-1} \sum_{\substack{i-k \\ 1 \leqslant t \leqslant m}}^{i+k} a_{i-k,t} \right) mod \ \ r$$

Например, если таблица состоит из трёх строк и четырёх столбцов, и первая строка состоит из чисел $2, 3, 4, 5$, а $r = 40$ то для этих исходных данных таблица будет выглядеть следующим образом (взятие по модулю показано только там, где оно приводит к изменению числа):

2	3	4	5
$5 = 2 + 3$	$9 = 2 + 3 + 4$	$12 = 3 + 4 + 5$	$9 = 4 + 5$
$23 = 2 + 3 + 4 + 5 + 9$	$0 = (2 + 3 + 4 + 5 + 5 + 9 + 12) \bmod 40 = 40 \bmod 40$	$4 = (2 + 3 + 4 + 5 + 9 + 12 + 9) \bmod 40 = 44 \bmod 40$	$33 = 3 + 4 + 5 + 12 + 9$

Требуется написать программу, которая по заданной первой строке таблицы (a_{11}, a_{12}, …, a_{1m}), вычисляет последнюю строку, как описано выше.

Формат входных данных

Первая строка входного файла содержит числа n, m и r ($2 \leqslant n, m \leqslant 2000, 2 \leqslant r \leqslant 10^9$) — число строк и столбцов таблицы соответственно, а так же число, по модулю которого надо посчитать ответ. Следующая строка содержит m целых чисел — первую строку таблицы: a_{11}, a_{12}, …, a_{1m}. Все a_{1i} неотрицательны и не превосходят 10^9.

Формат выходных данных

В первой строке выходного файла необходимо вывести m чисел — последнюю строку таблицы: a_{n1}, a_{n2}, ..., a_{nm}.

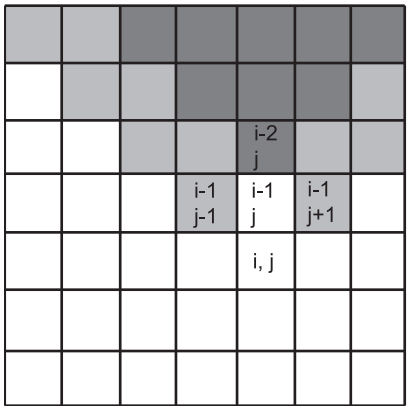
Примеры

Входные данные	Выходные данные
2 3 10 1 2 3	3 6 5
3 3 10 1 1 1	8 0 8
3 4 40 2 3 4 5	23 0 4 33

Разбор

Из условия задачи следует, что таблицу необходимо анализировать, двигаясь сверху-вниз. При этом очерёдность анализа элементов в строке не важна, т.к. никакого влияния на вычисляемую ячейку другие ячейки той же строки не оказывают.

База динамики (первая строка) нам явно задана. Рассмотрим два варианта вычисления значения в ячейке с использованием уже вычисленных данных.



В первом варианте нам достаточно завести массив B в котором мы будем накапливать ответы для ячеек. Очевидно, первая строка массива B совпадает с первой строкой массива A. Рассмотрим, как можно вычислить значение ячейки $B[i, j]$. Треугольник с вершиной в точке (i, j) состоит из треугольников с вершинами в точках $(i-1, j-1)$, $(i-1, j+1)$ а также из точек $(i-1, j)$ и (i, j). В то же время, треугольник с вершиной в точке $(i-2, j)$ перекрывается дважды, что приводит к двукратному суммированию значений в этом треугольнике. Избежать этого очень просто — достаточно один раз вычесть уже подсчитанную для этого треугольника сумму. Таким образом, в итоге получим следующую формулу $B[i][j] = B[i-1][j-1] + B[i-1][j+1] - B[i-2][j] + A[i-1][j] + A[i][j]$. Пользуясь этой формулой легко подсчитать массив, содержащий ответ.

Такой вариант также можно автоматизировать по используемой памяти: достаточно закольцевать по первому измерению массив A (при этом нам достаточно хранить всего две строки: текущую и предыдущую), а также закольцевать массив B (здесь понадобится три строки: текущая и две предыдущих). Пример закольцовывания массива приведён в разделе, посвящённом реализации очередей.

Для удобства программирования можно создать виртуальную «нулевую» строку в массиве B и заполнить ее нулями (создать барьер) — это избавит от необходимости рассматривать вторую строку как частный случай. Также разумно использовать барьеры из нулей слева и справа от таблицы.

Второй способ решения этой задачи также опирается на идеи динамического программирования, но использует решения подзадач, отличных от исходных. А именно, можно создать два дополнительных массива L и R, таких, что ячейка $L[i][j]$ будет содержать сумму всех элементов на диагонали, проведённой от точки (i, j) вверх и влево (для массива R диагональ проводится вверх и вправо). Первая строка массивов L и R будет совпадать с первой строкой массива A. Таким образом, формулы пересчёта будут

выглядеть следующим образом:

- $L[i][j] = L[i-1][j-1] + A[i][j]$
- $R[i][j] = R[i-1][j+1] + A[i][j]$
- $B[i][j] = B[i-1][j] + L[i-1][j-1] + R[i-1][j+1]$

При реализации этого метода барьеры слева и справа также будут полезны, а все массивы можно закольцевать по первому измерению всего в две строки.

Оба способа будут иметь сложность $O(n \times m)$ т.к. каждый элемент таблицы просматривается один раз.

Рассмотрим ещё одну задачу, при решении которой используется несколько подзадач меньшей размерности. Это задача о наибольшей общей подпоследовательности (НОП) двух последовательностей.

	1	*2*	*3*	*4*	*5*
2	0	**1**	1	1	1
3	0	1	**2**	2	2
2	0	**1**	2	2	2
4	0	1	2	**3**	3
5	0	1	3	3	**4**

Рис. 8.1: Пример таблицы для последовательностей $\{1, 2, 3, 4, 5\}$ и $\{2, 3, 2, 4, 5\}$

Если из некоторой последовательности $\{a\}$ вычеркнуть часть ее элементов, то получившаяся последовательность будет называться подпоследовательностью последовательности $\{a\}$. Пусть заданы две последовательности $\{a\}$ и $\{b\}$. Их наибольшей общей подпоследовательностью $\{c\}$ называется самая длинная подпоследовательность $\{a\}$, которая также является подпоследовательностью $\{b\}$.

Базой динамики в этом случае будет выступать тот факт, что НОП двух пустых последовательностей является также пустая последовательность. Теперь необходимо найти способ получать решение задачи, через меньшие подзадачи. Рассмотрим первые n элементов последовательности $\{a\}$ и первые m элементов последовательности $\{b\}$ и попытаемся найти НОП этих последовательностей. Допустим, что для всех $p < n$ и $q < m$ мы научились находить НОП. Тогда переход можно сформулировать так:

- Если $a_n = b_m$, то НОП(a, b) = НОП$(a-1, b-1)$ + 1
- Если $a_n \neq b_m$, то НОП$(a, b) = \max($НОП$(a-1, b,\ a, b-1))$

Результаты вычислений можно хранить в двумерной таблице (первый индекс — параметр n, второй — m) и заполнять ее двигаясь сначала по строкам, а затем по столбцам. В случае, если не ставится задача восстановления ответа, можно закольцевать эту таблицу и использовать лишь две строки. В случае, если необходимо восстановление ответа, необходимо хранить всю таблицу и перемещаться из текущей ячейки (n, m) либо в ячейку $(n-1, m-1)$ если $a_n = b_m$, либо в ячейку $(n-1, m)$ или $(n, m-1)$ в зависимости от того, в какой из ячеек значение совпадает со значением в (n, m). Ответ будет восстановлен, как обычно, в обратном порядке.

8.3 Использование предыдущего столбца

Если в случае одномерной динамики при мы могли использовать только один или несколько предыдущих элементов (или все предыдущие элементы), то в двух и более

мерной динамике часто возникают задачи, в которых используется только предыдущий столбец (т.е. для подсчёта решения задачи необходимо знать решения всех подзадач, у которых один из параметров на единицу меньше, чем у данной). Рассмотрим этот случай также на примере реальной задачи:

Маскарад

Московская олимпиада 10-11 2005

По случаю введения больших новогодних каникул устраивается великий праздничный бал-маскарад. До праздника остались считанные дни, поэтому срочно нужны костюмы для участников. Для пошивки костюмов требуется L метров ткани. Ткань продаётся в N магазинах, в которых предоставляются скидки оптовым покупателям. В магазинах можно купить только целое число метров ткани. Реклама магазина номер i гласит «Мы с радостью продадим Вам метр ткани за P_i бурлей, однако если Вы купите не менее R_i метров, то получите прекрасную скидку — каждый купленный метр обойдётся Вам всего в Q_i бурлей». Чтобы воплотить в жизнь лозунг «экономика страны должна быть экономной», правительство решило потратить на закупку ткани для костюмов минимальное количество бурлей из государственной казны. При этом ткани можно купить больше, чем нужно, если так окажется дешевле. Ответственный за покупку ткани позвонил в каждый магазин и узнал, что:

1. реклама каждого магазина содержит правдивую информацию о ценах и скидках;
2. магазин номер i готов продать ему не более F_i метров ткани.

Ответственный за покупку очень устал от проделанной работы и поэтому поставленную перед ним задачу «закупить ткань за минимальные деньги» переложил на своих помощников. Напишите программу, которая определит, сколько ткани нужно купить в каждом из магазинов так, чтобы суммарные затраты были минимальны.

Формат входных данных

В первой строке входного файла записаны два целых числа N и L ($1 \leqslant N \leqslant 100, 0 \leqslant L \leqslant 100$). В каждой из последующих N строк находится описание магазина номер i — 4 целых числа P_i, R_i, Q_i, F_i ($1 \leqslant Q_i \leqslant P_i \leqslant 1000, 1 \leqslant R_i \leqslant 100, 0 \leqslant F_i \leqslant 100$).

Формат выходных данных

Первая строка выходного файла должна содержать единственное число — минимальное необходимое количество бурлей.

Во второй строке выведите N чисел, разделённых пробелами, где i-ое число определяет количество метров ткани, которое нужно купить в i-ом магазине. Если в i-ом магазине ткань покупаться не будет, то на i-ом месте должно стоять число 0. Если вариантов покупки несколько, выведите любой из них.

Если ткани в магазинах недостаточно для пошивки костюмов, выходной файл должен содержать единственное число -1.

Примеры

Входные данные	Выходные данные
2 14 7 9 6 10 7 8 6 10	88 10 4
1 20 1 1 1 1	-1

Разбор

Будем решать эту задачу «динамикой по магазинам». В качестве базы динамики можно взять простой и понятный факт: покупка любого количества ткани в пустом наборе магазинов обойдётся в бесконечное число бурлей (в качестве бесконечности можно взять любое число, большее 100 000 (максимальное количество метров умноженное на максимальную цену), но такое, чтобы сумма двух бесконечностей не приводила к переполнению типа.

При реализации решения полезно написать функцию, которая по номеру магазина и количеству метров, которые необходимо купить в этом магазине, будет определять стоимость покупки. Обозначим эту функцию за $f(s, m)$, где s — номер магазина, а m — количество метров.

Теперь необходимо находить решение с помощью уже решённых задач. Пусть мы научились покупать любое количество метров ткани в первых $i-1$ магазине наилучшим образом. К нашему набору добавляется i-ый магазин и необходимо для каждого количества метров определить, сколько будет стоить их покупка в i магазинах (т.е. для каждого из i первых магазинов необходимо определить, сколько метров ткани в каком магазине покупать). Будем перебирать всевозможные количества метров ткани, которые надо купить. Пусть на данный момент нам необходимо научится покупать j метров ткани в i первых магазинах, при этом цены покупки k метров ткани в наборе из $i-1$ магазине известны для всех k и составляют $C[i-1][k]$.

Купить j метров ткани в i магазинах можно $j+1$ способом: можно купить 0 метров ткани в наборе из $i-1$ первого магазина, а все j метров — в магазине с номером i; можно купить один метр ткани в наборе из $i-1$ магазина, а $j-1$ — в магазине с номером i и т.д. Поскольку нас интересует минимальная цена покупки, то формула будет следующей:

$$C[i][j] = \min_{k=0\ldots j}(f(i,k) + C[i-1][j-k])$$

В этой задаче требуется восстановление решение. Для этого можно завести ещё один массив такого же как C размера и в каждой его ячейке хранить количество метров, купленное в этом магазине для заданных i и j.

Казалось бы, задача решена. Но в процессе разбора мы оставили некоторую неопределённость в вопросе того, какое максимальное количество метров необходимо купить. Первое приходящее в голову решение: не покупать больше чем L метров неправильно. Ведь может оказаться, что нам нужно купить 10 метров по 20 бурлей (итого 200), но если оптом купить 100 по оптовой цене в 1 бурль, то цена составит всего 100 бурлей (оставшиеся метры можно выкинуть). Поскольку каждый магазин продаёт не более 100 метров ткани, то в качестве максимального количества покупаемой ткани можно взять $L+100$. В таком случае для окончательного решения необходимо выбрать минимальное стоимость покупки K метров ткани, где $L \leqslant K \leqslant L+100$. В остальном решение не изменится. Его сложность будет составлять $O(N \times (L+100)^2)$.

Рассмотрим ещё один, более изощрённый пример использования подзадач из «предыдущего столбца».

Еловая аллея

Московская заочная олимпиада 2003

Мэр города Урюпинска решил посадить на главной аллее города, которая проходит с запада на восток, голубые ели. Причём сажать ели можно не во всех местах, а только на специально оставленных при асфальтировании аллеи клумбах.

Как оказалось, голубые ели бывают M различных сортов. Для ели каждого сорта известна максимальная длина ее тени в течение дня в западном и в восточном направлении (W_i и E_i соответственно). При этом известно, что ели растут гораздо лучше, если в течение дня они не оказываются в тени других елей.

Координатная ось направлена вдоль аллеи с запада на восток.

По заданным координатам клумб вычислите максимальное число елей, которое можно посадить, соблюдая условие о том, что никакая ель не должна попадать в тень от другой ели.

Формат входных данных

Во входном файле записано сначала натуральное число M — количество сортов елей ($1 \leqslant M \leqslant 100$). Затем идёт M пар чисел W_i, E_i, описывающих максимальную длину тени в западном и восточном направлении в течение дня для каждого сорта ели (числа W_i, E_i — целые, из диапазона от 0 до 30000). Далее идёт натуральное число N — количество клумб, в которых можно сажать ели ($1 \leqslant N \leqslant 100$). Далее идёт N чисел, задающих координаты клумб (координаты — целые числа, по модулю не превышающие 30000). Клумбы перечислены с запада на восток (в порядке возрастания их координат).

Если на клумбе с координатой X мы посадили ель, максимальная тень которой в восточном направлении равна E, то все клумбы с координатами от $X + 1$ до $X + E\check{}1$ попадают в тень от этой ели, а клумба с координатами $X + E$ — уже нет. Аналогично для тени в западном направлении.

Формат выходных данных

В выходной файл выведите сначала число A — максимальное количество елей, которые удастся посадить, а затем A пар чисел, описывающих ели. Первое число каждой пары задаёт номер клумбы, в которую садится ель. Второе число определяет номер сорта этой ели.

Пример

Входные данные	Выходные данные
3 1000 3 1 200 128 256 3 1 2 4	2 1 1 3 2

Разбор

В этой задаче потребуется динамика с двумя параметрами. Одним из них будет количество высаженных ёлок, а вторым — последняя занятая клумба. В самой таблице будем хранить сорт ели, посаженной в эту последнюю занятую клумбу.

Подумаем, какая база динамики здесь может пригодится. Путём несложных рассуждений можно понять, что первая клумба всегда должна быть занята, её пропуск может только ухудшить решение. При этом, поскольку слева от самой левой клумбы ничего не растёт, то никаких ограничений на тень, отбрасываемую на запад ёлкой, посаженной в первую клумбу, не накладывается. Логично, что необходимо минимизировать тень, отбрасываемую этой ёлкой на восток. Таким образом, базой динамики будет следующее: для количества ёлок 1 и последней занятой клумбой 1 в клумбу номер 1 необходимо

сажать ёлку, которая отбрасывает наименьшую тень на восток:

$$D[1][1] = K, \text{где } E_k \leqslant \min_{i=0\ldots M-1} E_i$$

Все остальные первого столбца (где количество высаженных ёлок превышает 1, номер последней занятой клумбы) можно заполнить признаком невозможности такой конфигурации — числом -1, например.

База динамики известна, теперь необходимо сформулировать общее правило перехода к большему решению. Для каждого количества занятых клумб, для каждого номера последней занятой клумбы будем выбирать сорт ели, который можно в неё посадить и который отбрасывает наименьшую тень на восток (сорт, отбрасывающий большую тень на восток может только ухудшить наше решение, накрыв лишние клумбы, но не улучшить его). Будем последовательно рассматривать соотношения, которые должны выполняться. В данный момент мы сажаем ёлку в клумбу номер i и общее количество ёлок должно стать j (логично, что $j \leqslant i$). Будем перебирать все предыдущие клумбы $prev = 1 \ldots i-1$. Для каждой из предыдущих клумб необходимо выполнить некоторые проверки и действия. Естественно, что $D[j-1][prev] \neq -1$, иначе предыдущей конфигурации не существовало. Далее необходимо проверить, что предыдущая ёлка не накрывает своей тенью текущую клумбу, т.е. $X[prev] + E[D[j-1][prev]] \leqslant X[i]$. Если эти условия выполнены, то в клумбу с номером i при условии, что последняя предыдущая занятая клумба имеет номер $prev$ теоретически можно посадить ель (i-я клумба не накрывается тенью от предыдущей ели). Будем перебирать все сорта елей $p = 0 \ldots M$. Для каждого из сортов необходимо проверить, что он, будучи высаженным в клумбу номер i не накрывает своей тенью предыдущую клумбу (с номером $prev$). Эта проверка осуществляется следующим образом: $X[prev] \leqslant X[i] - W[p]$. Если условие выполнено, то такой сорт ели в эту клумбу посадить можно. Среди всех подходящих сортов выберем и запишем в ячейку $D[j][i]$ номер этого сорта. Для восстановления решения удобно содержать массив такого же размера, в соответствующую ячейку которого записывать значение $A[j][i] = prev$, при котором был достигнут наилучший результат.

Чтобы найти ответ, необходимо пройти по таблице и найти такое $D[j][i] \neq -1$, в котором j (количество высаженных ёлок) максимально. Выведем это число. Запомним, какой сорт ели был высажен в клумбу с номером i и восстановим полное решение. Поскольку одна ель уже была высажена, то оставшееся количество уменьшилось на 1. Номер предыдущей занятой клумбы нам известен — это $A[j][i]$, сорт ели для предыдущей клумбы легко определить: $D[j-1][A[j][i]]$. Затем проделаем те же действия для ячейки $D[j-1][A[j][i]]$. Уменьшая количество высаженных ёлок на 1, мы восстановим полное решение.

Сложность решения этой задачи составит $O(N^3 \times M)$ (необходим проход по всем ячейкам таблицы за N^2, для каждой из ячеек мы перебираем всевозможные предыдущие клумбы за N и всевозможные сорта за M).

8.4 LR-динамика

В предыдущих разделах мы рассматривали динамику, где либо использовалось некоторое фиксированное количество предыдущих подзадач, либо все задачи размерностью от 1 до $N-1$. В двумерном случае также возможен вариант, когда задача задаётся не

одним параметром N, в двумя параметрами L и R — левой и правой границей подряд идущих значений, для которых мы ищем решение. Также такой метод называется «динамика по подстрокам». Обычно, в таких задачах за базу динамики берётся диагональ матрицы, т.е. подстроки, состоящие из одного значения (при $L = R$). Принципы динамического программирования остаются те же самые, тем не менее, рассмотрим на примерах.

Скобки

Московская командная олимпиада 2004

Назовём строку S правильной скобочной последовательностью, если она состоит только из символов '{', '}', '[', ']', '(', ')' и выполнено хотя бы одно из следующих трёх условий:

1. S — пустая строка;
2. S можно представить в виде $S = S_1 + S_2 + S_3 + ... + S_N$ ($N > 1$), где S_i — непустые правильные скобочные последовательности, а знак "+"обозначает конкатенацию (приписывание) строк;
3. S можно представить в виде $S =' \{'+C+'\}'$ или $S =' [' + C + ']'$ или $S =' (' + C + ')'$, где C является правильной скобочной последовательностью.

Дана строка, состоящая только из символов '{', '}', '[', ']', '(', ')'. Требуется определить, какое минимальное количество символов надо вставить в эту строку для того, чтобы она стала правильной скобочной последовательностью.

Формат входных данных

В первой строке входного файла записана строка, состоящая только из символов '{','}', '[',']', '(',')'. Длина строки не превосходит 100 символов.

Формат выходных данных

Вывести в первую строку выходного файла единственное неотрицательное целое число — ответ на поставленную задачу.

Примеры

Входные данные	Выходные данные
{(})	2
([{}])	0

Разбор

Задача решается методом LR-динамики. А именно, в ячейке таблицы $D[L, R]$ будем хранить сколько скобок необходимо добавить, чтобы последовательность символов с L по R стала правильной скобочной последовательностью.

Для облегчения программирования введём базу динамики следующим образом:

1. Все последовательности, состоящие из одной скобки, требуют добавления одной скобки (т.е. главную диагональ матрицы заполняем 1).
2. Последовательности, у которых правая граница меньше левой, требуют добавления 0 скобок.

Теперь необходимо научится решать задачу для параметров L и R считая, что все меньшие подзадачи уже решены. В LR-динамике размер задачи определяется как разность между правой и левой границей (в нашем случае это количество символов).

Задача решается очень легко. Если на месте L стоит открывающая скобка, а на месте R — соответствующая ей закрывающая, то $D[L,R] = D[L+1, R-1]$. Это соотношение объясняется тем, что на интервале $L+1, R-1$ у нас уже получена правильная скобочная последовательность, а окружённая парой скобок она также остаётся правильной и добавлений не требует.

В случае же если на местах L и R не стоит пара соответствующих скобок, то решение определяется следующим образом:

$$D[L,R] = \min_{K=L\ldots R-1}(D[L,K] + D[K+1,R])$$

Эта формула обозначает, что мы разбиваем последовательность от L до R на две части всеми возможными способами и выбираем среди всех таких разбиений наилучшее. В частности, последовательность '[](' будет разбита на две последовательности из 2 символов, каждая из которых требует 0 добавленных скобок, и результат для неё также будет равен 0.

Метод разбиения на 2 части применяется практически во всех задачах на LR-динамику, а отличаются они лишь начальной инициализацией и специфичными условиями, при которых можно находить лучшее решение.

Также в этой задаче удобно использовать программистский приём, называемый «ленивой динамикой» или «рекурсией с меморизацией». Действительно, если в предыдущих задачах порядок решения подзадач определялся и реализовывался легко, то в LR-динамике необходимо решать подзадачи, двигаясь по диагоналям матрицы, что уже неприятно. А встречаются задачи, в которых порядок обхода ещё более запутанный. В таких ситуациях удобно пользоваться рекурсивной функцией, которая будет похожа на перебор всех вариантов, но будет запоминать однажды посчитанное решение. В неё же можно загнать и ограничения. Идея ее следующая: сначала заполним всю матрицу, которая необходима для подсчёта, признаком отсутствия решения (например, числом -1). Обращаться к этой матрице при ее подсчёте можно только с помощью рекурсивной функции, т.е. чтобы обратится к элементу $D[L][R]$ необходимо каждый раз вызывать функцию $f(L,R)$. Функция, в свою очередь, будет проверять, подсчитано ли это значение и если да, то возвращать его. Иначе необходимо подсчитать значение, сохранить его в матрице и опять же вернуть. Для это задачи функция может выглядеть так (мы рассмотрим пример только для круглых скобок, остальные несложно добавить):

```
int f(int L, int R)
{
  int K, now;
  if (D[L][R] == -1) // значение еще не подсчитано
    if (R < L) D[L][R] = 0; // база динамики №1
    else if (R == L) D[L][R] = 1; // база динамики №2
    else
      if ((S[L] == '(') && (S[R] == ')')) D[L][R] = f(L+1, R-1);
      else { // первая и последняя скобки не образуют пары
        D[L][R] = R-L+1; // в худшем случае к каждой добавим
```

```
        for (K = L; K < R; K++) { // разбиваем на две части
          now = f(L, K) + f(K+1, R); // ищем стоимость
          if (now < D[L][R]) D[L][R] = now; // если надо меняем
        }
      }
  return D[L][R]; // возвращаем последнее значение
}
```

Теперь для того чтобы найти решение задачи достаточно считать входную строку S, заполнить матрицу D признаками неподсчитанности решения и вывести значение функции $f(0, N-1)$, где N — длина входной строки S. Все остальное сделается «само» и о порядке вычисления подзадач беспокоится не следует.

Реализовывать ленивую динамику надо всегда исходя из приведённой выше схемы: это позволит писать ее коротко и правильно. Ленивая динамика применима во всех задачах динамического программирования и, местами, даже более понятна, но злоупотреблять ей не стоит, т.к. она вызывает накладные расходы как по памяти (забивается стек), так и по времени (запуск функции — не самая быстрая операция).

Рассмотрим ещё одну, немного нестандартную, простую задачу на LR-динамику:

Максимальный подпалиндром

Командный чемпионат школьников Санкт-Петербурга 1999

Палиндромом называется строка, которая одинаково читается как слева направо, так и справа налево. Подпалиндромом данной строки называется последовательность символов из данной строки, не обязательно идущих подряд, являющаяся палиндромом. Например, `HELOLEH` является подпалиндромом строки `HTEOLFEOLEH`. Напишите программу, находящую в данной строке подпалиндром максимальной длины.

Формат входных данных

Во входном файле находится строка длиной не более 100 символов, состоящая из заглавных букв латинского алфавита.

Формат выходных данных

Выведите на первой строке выходного файла длину максимального подпалиндрома, а на второй строке сам максимальный подпалиндром. Если таких подпалиндромов несколько, то ваша программа должна вывести любой из них.

Пример

Входные данные	Выходные данные
HTEOLFEOLEH	7 HELOLEH

Разбор

Отличие этой задачи от стандартной LR-динамики состоит в том, что разбиения на две части не предусматривается. При решении задачи будем пользоваться стандартными средствами: заведём матрицу D где первый индекс будет означать левую границу подстроки, для которой мы ищем максимальный подпалиндром, а второй — правую.

База динамики будет следующей: последовательность из 0 или отрицательного количества символов (когда L больше чем R) имеет максимальный подпалиндром длины 0. Последовательность из 1 символа даёт подпалиндром длины 1 (поскольку строка длины 1 всегда является палиндромом).

Переход к большей задаче осуществляется следующим образом:

- Если $S[L] = S[R]$, то $D[L, R] = D[L + 1, R - 1] + 2$
- Если $S[L] \neq S[R]$, то $D[L, R] = max(D[L, R - 1], D[L + 1, R])$

При реализации удобно использовать ленивую динамику.

8.5 Динамика по профилю

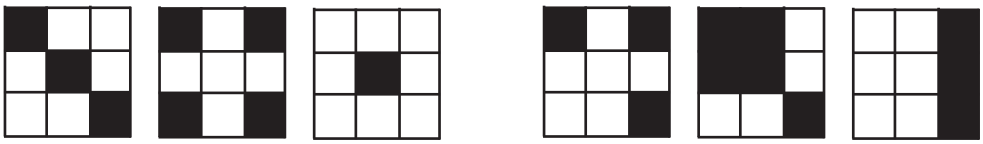
Динамическим программированием по профилю называется такой вид динамического программирования, когда подзадача меньшего размера оканчивается некоторой конфигурацией (профилем), которую можно закодировать числом. К подзадаче добавляется ещё один элемент с некоторой конфигурацией и он становится профилем новой подзадачи. Кроме того, можно составить таблицу допустимых переходов от одного профиля к другому; составить набор допустимых начальных профилей и решать задачу, увеличивая количество элементов в подзадаче. Обычно задачи на динамику по профилю требуют подсчёта числа решений. Обратимся, как обычно при изучении динамического программирования, к примерам.

Симпатичные узоры

Всероссийская командная олимпиада школьников по программированию 2000

Компания BrokenTiles планирует заняться выкладыванием во дворах у состоятельных клиентов узор из чёрных и белых плиток, каждая из которых имеет размер 1×1 метр. Известно, что дворы всех состоятельных людей имеют наиболее модную на сегодня форму прямоугольника $M \times N$ метров.

Однако при составлении финансового плана у директора этой организации появилось целых две серьёзных проблемы: во первых, каждый новый клиент очевидно захочет, чтобы узор, выложенный у него во дворе, отличался от узоров всех остальных клиентов этой фирмы, а во вторых, этот узор должен быть симпатичным. Как показало исследование, узор является симпатичным, если в нем нигде не встречается квадрата 2×2 метра, полностью покрытого плитками одного цвета. На рисунке 1 показаны примеры различных симпатичных узоров, а на рисунке 2 — несимпатичных.



Для составления финансового плана директору необходимо узнать, сколько клиентов он сможет обслужить, прежде чем симпатичные узоры данного размера закончатся. Помогите ему!

Формат входных данных

В первой строке входных данных содержатся два положительных целых числа, разделённых пробелом: M и N ($1 \leqslant M \times N \leqslant 30$).

Формат выходных данных

Выведите единственное число — количество различных симпатичных узоров, которые можно выложить во дворе размера $M \times N$. Узоры, получающиеся друг из друга сдвигом, поворотом или отражением, считаются различными.

Примеры

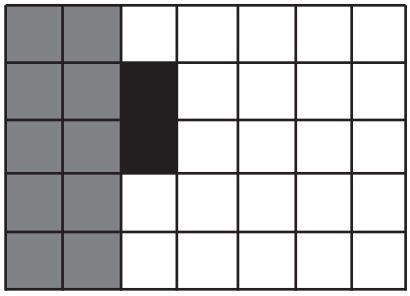
Входные данные	Выходные данные
2 2	14
3 3	322

Разбор

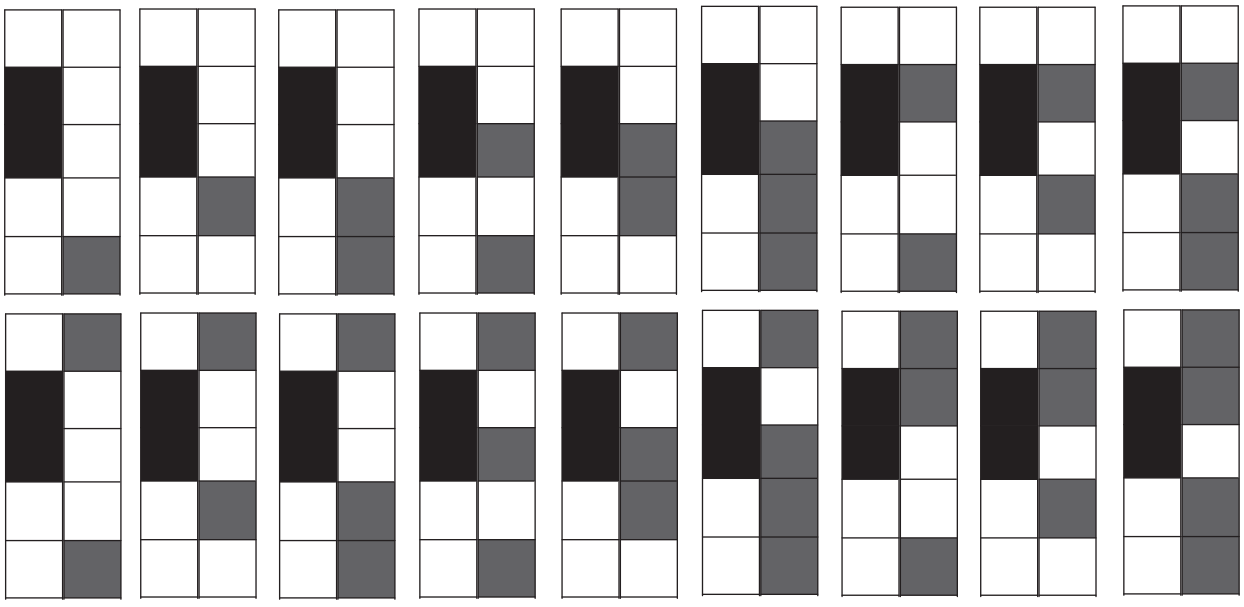
Заметим, что каждая клетка таблицы может иметь два состояния: быть белой (0) или чёрной (1). Из ограничений к задаче можно понять, что длина минимальной из сторон не превысит 5, будем называть минимальную из сторон столбцом.

Если в таблице всего один столбец, то он может быть заполнен произвольным образом, т.е. ответом будет являться число M^2.

Каждый столбец кодируется 5 битами и может иметь 2^5 состояний. Будем называть число, которым кодируется столбец *профилем*. Будем считать, что все столбцы с номерами от 1 до $K-1$ уже полностью корректно заполнены и столбец с номером $K-1$ имеет профиль равный числу M.



На рисунке последний столбец имеет профиль 01100_2 или 12_{10}. Покажем совместимые с ним профили, которые при расположении столбцов рядом не образуют квадрат 2×2 одного цвета. Такими профилями будут профили 00001_2, 00010_2, 00011_2, 00101_2, 00110_2, 00111_2, 01001_2, 01010_2, 01011_2, 10001_2, 10010_2, 10011_2, 10101_2, 10110_2, 10111_2, 11001_2, 11010_2 и 11011_2.



Таким образом, если имеется корректная таблица из $K-1$ столбца, оканчивающаяся профилем 01100_2, то из неё можно получить таблицы длиной K и имеющие в качестве профиля одно из 18 совместимых с 01100_2 чисел. При этом для каждой длины таблицы и её профиля можно хранить количество вариантов составить такую корректную таблицу, а при каждом переходе суммировать для таблицы длины K с профилем P все значения для таблиц длины $K-1$ и профилем, совместимым с P. В виде формулы это можно записать так:

$$D[K][P] = \sum_{L=0}^{2^M} D[K-1][L] \times g(P,L)$$

Где K — длина таблицы, P — интересующий нас профиль, M — высота столбца, L — профиль предыдущего столбца, а функция $g(P,L)$ возвращает 1 в случае, если столбцы P и L совместимы и 0 в противном случае. База динамики, как уже говорилось выше, состоит в следующем: при $K=1$ для любого профиля количество вариантов раскраски клеток равно единице. Получать решение следует двигаясь сначала по количеству столбцов, для каждого из столбцов перебирать профиль и для каждого профиля, в свою очередь, перебирать все предыдущие профили.

Функцию $g(P,L)$ можно заменить предподсчитанной матрицей, чтобы не вызывать проверку на совместимость профилей много раз. При этом считать совместимость профилей удобно с использованием битовых операций.

Сложность такого решения составит $O(N \times (2^M)^2)$. Ответ гарантированно не превысит 2^30 степени, т.к. всего, даже без ограничений, всю таблицу можно раскрасить 2^30 способами.

Рассмотрим ещё один метод решения задач на динамику по профилю.

Симпатичные узоры возвращаются

Зимние сборы к международной олимпиаде 2003

Со времён написания условия предыдущей задачи многое изменилось. Симпатичные узоры стали очень популярны по всему миру, поэтому люди готовы содержать очень большой участок земли, лишь бы иметь на ней узор, не встречающийся больше нигде.

Теперь компания BrokenTiles является ведущим производителем симпатичных узоров в мире! Для составления плана исполнительному директору Васе по-прежнему необходимо знать, сколько клиентов могут рассчитывать на узор данных размеров.

Так как масштабы буквально мировые, $N \leqslant 10^{100}$. Однако Вася не любит большие числа, поэтому просит выдать ответ по модулю P.

Формат входных данных

В первой строке входных данных содержатся три положительных целых числа, разделённых пробелом: N, M и P ($1N \leqslant 10^{100}, 1 \leqslant M \leqslant 5, 1 \leqslant P \leqslant 10\,000$).

Формат выходных данных

Выведите единственное число — количество различных симпатичных узоров, которые можно выложить во дворе размера $M \times N$ по модулю P. Узоры, получающиеся друг из друга сдвигом, поворотом или отражением, считаются различными.

Примеры

Входные данные	Выходные данные
2 2 5	4
3 3 23	0

Разбор

Эта задача очень похожа на предыдущую, но та решалась за линейное от N время, что наверняка вызовет превышение ограничения на время работы программы, при $N = 10^{100}$.

Тем не менее, задача решаемая. Достаточно вспомнить, что мы умеем считать числа Фибоначчи за $O(\log N)$ и воспользоваться похожим методом. Пусть у нас есть таблица совместимости профилей G размером $2^M \times 2^M$. Приведём пример такой таблицы для $M = 2$:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Единица на пересечении i-ой строки и j-го столбца обозначает, что переход от профиля i к профилю j возможен. Если мы просуммируем значения всех ячеек этой матрицы, то получим количество вариантов раскраски прямоугольника 2×2. Вспомним, как делался переход от одного профиля к другому:

$$D[K][P] = \sum_{L=0}^{2^M} D[K-1][L] \times G[P][L]$$

Заметим, что для некоторого профиля мы перебираем все другие профили и суммируем количество вариантов в случае, если они совместимы (если они несовместимы, то тоже суммируем, но нули). Это чрезвычайно похоже на умножение матриц. Действительно, возьмём матрицу G и возведём ее в квадрат. При этом идейно произойдёт следующее: мы будем идти от профиля i в профиль j через всевозможные профили, т.е. мы переберём все k и, в случае, если возможны переходы из i в k и из k в j, то увеличим количество вариантов попасть из i в j за два шага на 1. Таким образом, сумма элементов матрицы G^2 будет равна ответу для задачи при $N = 3$. Аналогично можно делать переход не через один, а через несколько шагов. Т.е. сумма элементов матрицы G^{N-1} будет равна ответу на задачу длины N. Для ускорения можно воспользоваться быстрым возведением в степень (оно применимо для любых объектов, допускающих умножение), оно и другие необходимые операции описаны в лекции, посвящённой арифметике и теории чисел.