



Центр дополнительного образования детей

# Дистантное обучение

117630, Москва, ул. акад. Челомея, д. 8б, тел./факс 936-3104

Курс «Олимпиадные задачи по программированию».

Преподаватель: Михаил Сергеевич Густокашин

## Лекция 3

### Алгоритмы поиска в олимпиадных задачах.

#### Оглавление:

Поиск в неупорядоченных массивах. ....	2
Поиск порядковых статистик.....	3
Бинарный поиск в упорядоченных массивах. ....	4
Бинарный поиск для монотонных функций.....	5
Бинарный поиск по ответу.....	5

© Михаил Густокашин, 2007

[msg@list.ru](mailto:msg@list.ru)

<http://g6prog.narod.ru>

<http://desc.ru>

## 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 такой элемент в массиве не встречается, то *i* будет равно *n*+1.

Рассмотрим отдельно задачу поиска минимума и максимума в массиве. Также 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  $2N-2$  сравнения. 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  $3N/2$  (проблема возникает, когда количество элементов нечетное – один из элементов остается без пары). Точно это можно записать как  $3\lceil N/2 \rceil - 2$ , где  $\lceil \cdot \rceil$  - 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).

### 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, меньшие  $a[s]$ , 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  $l = j+1$  и неизменным  $r$ .

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

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

```

        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);
    else return search(a, k, j+1, r);
}

```

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.

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.

### Бинарный поиск в упорядоченных массивах.

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

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.

## 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  $\sqrt[3]{x}$  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`.

## 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

Максимальное время работы на одном тесте:	0.2 секунды
Максимальный объем используемой памяти:	64 мегабайта
Имя входного файла:	ain.txt
Имя выходного файла:	aout.txt

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  $x$  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 ( $1 \leq N \leq 2 \cdot 10^8$ ,  $1 \leq x, y \leq 10$ ).

### 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  $(N-1) \cdot x$  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;
    freopen("ain.txt", "r", stdin);
    freopen("aout.txt", "w", stdout);
    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;
}
```

Следующая задача была на 13 Украинской олимпиаде под названием «Автобус»:

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 BUS, которая определит минимальное время, за которое автобус привезет максимально возможное количество рабочих.

Входные данные: Входной текстовый файл BUS.DAT в первой строке содержит количество остановок  $N$  и количество мест в автобусе  $M$ . Каждая  $i$ -я строка 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 ( $1 \leq M \leq 2000$ ,  $1 \leq N, K \leq 200000$ ).

Пример входных данных.

```
3 5
1 2 0 1
1 1 2
1 4 0 2 3 4
```

Выходные данные: Единственная строка выходного текстового файла BUS.SOL должен содержать минимальное время, необходимое для перевозки максимального количества рабочих.

Пример выходных данных.

```
4
```

Решение задачи выглядит так:

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).

Задачу надо решать с помощью двоичного поиска (бинарного поиска, дихотомии). А лучше с помощью двух двоичных поисков.

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 двоичном 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 прийти 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 прийти 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 вторая дихотомия. Та самая, которая определяет, сколько людей успеет прийти 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 прийти, то  $x = (\min + x) / 2$ . Здесь условие выхода такое: если человек успевает прийти на остановку, а следующий за ним нет - то ответом будет номер человека. Отдельно нужно обрабатывать случай, если на автобус сядут все люди с остановки.