Лекция 3. Бинарный поиск

Последовательный и бинарный поиск. Сложность последовательного и бинарного поиска. Поиск элементов в массиве, поиск по целым числам. Нижняя и верхняя граница искомого числа. Поиск по вещественным числам. Определение корней функции. Примеры задач.

Общее представление

Игра «Угадай число». Вам загадали число от 1 до 1000, $x \epsilon [1; 1000]$. Вы должны его отгадать, задавая вопросы, на которые можно получить ответ «да» или «нет». В худшем случае Вы последовательно называете числа: 1, 2, 3,…999. В общем случае $x \epsilon [1; n]$ и сложность такого алгоритма – $O(n)$. То есть, в худшем случае Вы зададите n вопросов, чтобы отгадать число. Идея оптимизации алгоритма заключается в делении области множества чисел, в которой производится поиск, пополам.

На рисунке изображено множество чисел $[1; 16]$. Загадано число 13. Каждый раз Вы называете число *x*, и задаете вопрос «Больше или равно *x* загаданное число?» Первый раз Вы называете число 8 – середину области поиска, и, узнав, что загаданное число больше 8, переходите в область поиска $[9; 16]$. Таким образом, область поиска уменьшилась в 2 раза. Называете число 12, и переходите в область поиска $[13; 16]$. Называете число 14, и переходите в область поиска $[13; 14]$. Наконец, вы называете число 13 и получаете ответ, что это и есть загаданное число. Вам потребовалось 4 вопроса.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Так как каждый раз область поиска уменьшалась в два раза, то в худшем случае потребуется $\log_2 n$ вопросов для отгадывания числа. Получаем логарифмическую сложность алгоритма поиска числа – $O(\log n)$. *Бинарный поиск (двоичный поиск) – алгоритм поиска в упорядоченном множестве чисел, использующий метод деления области поиска пополам и имеющий логарифмическую сложность.*

Алгоритм бинарного поиска

Применим бинарный поиск для поиска искомого числа в упорядоченном массиве. Обозначим правую границу поиска переменной *right*, левую границу – *left*. Середину области поиска – *middle*. *middle=(left+right)/2.*

```
cin>>b;              // Искомый элемент
left=-1; right=n-1; // Левая и правая граница поиска
while (right-left>1)  // Пока правая граница правее левой
{
     middle=(left+right)/2;   // Середина области поиска
     if (a[middle]>=b)
          right=middle;  // Передвигаем правую границу
     else
          left=middle;   // Иначе передвигаем левую границу
}
if (a[right]==b) cout<<right; else cout<<-1;
```

Программа выведет индекс искомого элемента, либо -1, если такого элемента в массиве нет.

После выполнения алгоритма переменная *right* будет указывать на первый элемент в массиве, который равен искомому элементу или больше искомого. Переменная *left* будет указывать на самый большой элемент, меньше искомого. Если элемента, равного искомому или больше искомого нет, то переменная *right* будет указывать на последний элемент массива.

Встроенные функции C++

В Visual Studio существует встроенная функция **binary_search**, которая проверяет, есть ли в отсортированном диапазоне элемент, равный указанному значению. Функция возвращает значение true если указанный элемент имеется в диапазоне и false – если такой элемент отсутствует. Функция требует заголовка <algorithm>.

```
#include<iostream>
#include<algorithm>
using namespace std;
int a[10]={1,2,3,4,5,6,7,8,9,10};
int main()
{
     int b;
     cin>>b;
     cout<<binary_search(a,a+10,b); //Поиск числа b в массиве a
}
```

Программа выведет 0, если элемент b найдет в массиве и 1 – если не найден.

Приведем и другие полезные функции Visual Studio. Функция **lower_bound** возвращает итератор первого элемента в отсортированном диапазоне, который равен или больше искомого элемента.

```
#include<iostream>
#include<algorithm>
using namespace std;
int a[10]={1,2,3,4,5,5,5,8,9,10};
int main()
{
     int b=5;
     cout<<*(lower_bound(a,a+10,b))<<endl;
     b=6;
     cout<<*(lower_bound(a,a+10,b));
}
```

Программа выведет значения 5, так как в массиве присутствует элемент 5 и значение 8 – значение первого элемента, большего 6.

Аналогично функции **lower_bound** работает функция **upper_bound**, которая возвращает итератор на элемент, который строго больше искомого в заданном диапазоне упорядоченных элементов.

Сложности работы алгоритма бинарного поиска

Сложность работы алгоритма бинарного поиска – $O(\log n)$. Предположим, нам нужно найти число в диапазоне $[1; 1000]$. Каждый шаг алгоритма уменьшает область поиска в два раза: 1000 чисел → 500 чисел → 250 чисел → 125 чисел → 63 числа →32 числа →16 чисел→8 чисел→4

числа→2 числа→1 число. Нам потребовалось $\lceil \log 1000 \rceil = 10$ итераций. То есть, достаточно 10 итераций. Покажем, что меньше итераций быть не может.

Приведем доказательство при помощи протоколов работы алгоритма. Укажем все возможные числа, которые мы можем искать в диапазоне $[1; 1000]$ в первом столбце. В первой строке указываем номера вопросов, которые мы задаем при поиске числа. Предположим, нам потребовалось 9 вопросов. В каждой ячейке ставим знак «+» или «-», в зависимости от того получили вы ответ «да» или «нет» на ваш вопрос. Получаем некоторую таблицу – протокол игры. Каждая строка показывает, как было угадано число, указанное слева в строке.

Искомое число	1	2	3	4	5	6	7	8	9
1	+	-	-	-	-	-	-	-	-
2	-	+	+	+	+	+	+	+	+
3	-	-	-	-	-	+	+	+	+
….	-	+	+	+	+	+	+	+	+
….									
1000	+	-	-	+	-	-	-	-	-

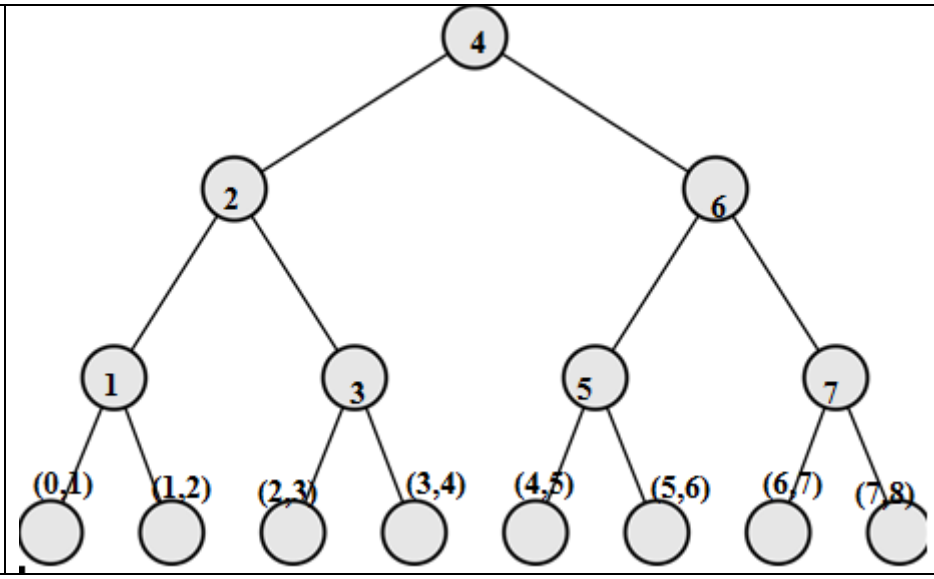
Всего получается 2^9=512 вариантов различных строк (отличающихся друг от друга). Строк в таблице 1000 и получается, что по принципу Дирихле хотя бы для одна комбинация будет соответствовать не менее, чем $\left\lceil \frac{1000}{512} \right\rceil = 2$ строкам протокола. Получается, что существует хотя бы 2 строки, в которых одинаковые комбинации знаков «+» и «-». Поскольку разные числа не могли быть угаданы алгоритмом на основе одних и тех же ответов на одни и те же вопросы, то мы получаем противоречие. То есть, для алгоритма бинарного поиска потребуется $\lceil \log n \rceil$ операций.

Работу алгоритма бинарного поиска можно изобразить в виде дерева. Корень дерева – это первое деление диапазона поиска на две части. Предположим, что массив поиска a[8]={0,1,2,3,4,5,6,7}. Присваиваем значения границ: l= 0, r=8. Корень дерева соответствует m=(l+r)/2=4. Из корня дерева выходят два ребра – одно (левое) соответствует ветке «да», другое (правое) – ветке «нет» условия `if (a[m]>b)`, где b – искомое число. Две дочерние вершины корня – это следующие переходы по границе *middle*.

Корень дерева – первая проверка
l= 0; r;=8; m=4

l= 0; r=4; m=2
l= 4; r=8; m=6

l= 0; r=2; m=1
l=2; r=4; m=3
l=4; r=6; m=5
l=6; r=8; m=7



Листья дерева – указатели на r и l после работы алгоритма. r будет указывать на искомое число, если таковое есть.

Глубина дерева – количество уровней в нем. Глубина дерева определяет сложность алгоритма – показывает сколько итераций совершил алгоритм бинарного поиска. Количество листьев дерева определяется как 2^n, где n – номер уровня дерева (уровень вершины-корня нулевой). Вернемся к вопросу сложности алгоритма бинарного поиска. Покажем при помощи дерева работы алгоритма, что меньше, чем 10 итераций для поиска искомого числа среди 1024

чисел, алгоритм сделать не можем. Действительно, если итераций будет выполнено 9, то листьев в дереве получим 512, то есть все наши числа найдены быть не могут. Для работы алгоритма бинарного поиска потребуется $\lceil \log n \rceil$ операций.

Поиск корня уравнения алгоритмом бинарного поиска

Алгоритм бинарного поиска удобно применять для определения корней уравнения (так называемый вещественный бинарный поиск).

Задача. Найдите такое число x, что $x^2 + \sqrt{x} = C$, с точностью не менее 6 знаков после точки.

Входные данные

В единственной строке содержится вещественное число $1.0 \le C \le 10^{10}$.

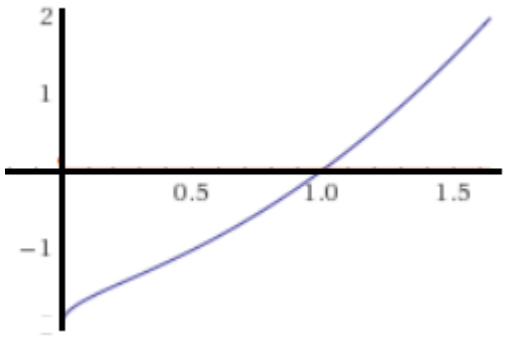
Выходные данные

Выведите одно число — искомый x.

Примеры

Входные данные	Выходные данные
2.0000000000	1.000000000
Входные данные	Выходные данные
18.0000000000	4.000000000

На рисунке представлен график функции $y = x^2 + \sqrt{x} - 2$, соответствующей первому входному тесту. Заметим, что функция является возрастающей функцией на всей области определения, как сумма двух возрастающих функций. $y(0) \le 0$, $y(C) \ge 0$. Значит, функция имеет на промежутке $[0; C]$ единственный корень. Поиск корня будем производить с точностью $\varepsilon = 10^{-10}$. Будем уменьшать диапазон поиска, сдвигая правую или левую границу поиска, пока не достигнем заданной точности. Программа приведена ниже.




```
#include<iostream>
#include<float.h>
#include<cmath>
#include<math.h>
using namespace std;
long double f(long double x){
    return x*x+sqrt(x);
}
int main()
{
    freopen("input.txt","r",stdin);
    freopen("output.txt","w",stdout);
    long double c,eps=1e-10,left=0,right=1e15,middle;
    cin>>c;
    while(fabs(right-left)>eps){
        middle=(left+right)/2.0;
        if (f(middle)-c<0)
            left=middle;
        else
            right=middle;
    }
```

```
    cout <<fixed;
    cout.precision(7);
    cout<<right;
    return 0;
}
```

В приведенной программе окончание поиска происходит в том случае, когда рассматриваемый отрезок станет меньше заданной погрешности. Примерное количество итераций алгоритма будет равно $log\left(\frac{R-L}{\varepsilon}\right)$. В рассматриваемом примере смещение границ поиска происходило на основе сравнения значения функции с нулем: f(middle)-con<0. Если нет информации относительно монотонности функции, но точно известно, что на интервале поиска она имеет единственный корень, то разумнее будет проверять наличие разных знаков функции на границах новой области поиска: в случае $f(l) * f(m) \leq 0$ передвигать правую границу (переходим в левый промежуток), а в случае $f(r) * f(m) \leq 0$ передвигать левую границу (переходим в правый промежуток).

Решение задач. Бинарный поиск по ответу

Задачи, в которых требуется найти какое-либо значение, часто могут быть решены при помощи алгоритма бинарного поиска. В этом случае говорят о бинарном поиске по ответу. Для решения таких задач необходимо определить исходную область поиска: левую и правую границу поиска. Область поиска как раз и представляет собой упорядоченное множество потенциальных ответов, среди которых нужно выбрать тот, который удовлетворяет условию задачи. Важно предварительно сформулировать условие перехода в левую или правую половину области поиска на каждом шаге алгоритма. Для этого, как правило, происходит вычисление значения данной в задаче характеристики и переход в левую или правую область поиска в зависимости от того, больше или меньше эта характеристика, чем та, которая требуется для ответа. Рассмотрим пример.

Задача «Дипломы» (Региональный этап Всероссийской олимпиады школьников по информатике 2009 - 2010 учебного года)

К окончанию школы у Пети накопилось *n* дипломов, причём, все они имели одинаковые размеры : *w* — в ширину и *h* — в высоту. Петя решил украсить свою комнату, повесив на одну из стен свои дипломы. Он решил купить специальную доску, чтобы прикрепить её к стене, а к ней — дипломы. Петя хочет, чтобы доска была квадратной. Каждый диплом должен быть размещён строго в прямоугольнике размером *w* на *h*. Дипломы запрещается поворачивать на 90 градусов. Прямоугольники, соответствующие различным дипломам, не должны иметь общих внутренних точек. Требуется написать программу , которая вычислит минимальный размер стороны доски, которая потребуется Пете для размещения всех своих дипломов.

```
l = 0; r= (w + h) * n + 1;
while (r  - l > 1)
{
     x := (r + l) / 2;
     if ((x / w) * (x / h) >= n)
          r= x;
     else
          l= x;
}
cout>>r;
```

Для решения задачи предварительно была выбрана область поиска: минимальный размер квадрата – 0, максимальный размер квадрата (с избытком) таков, что квадрат превращается в

линию длины *(w + h) * n*. На каждой итерации алгоритма мы вычисляем *x* – середину области поиска. Это потенциальная длина требуемого квадрата. Такой квадрат проверяется на то, вместится ли в него больше, чем нужно дипломов или нужное количество дипломов. Если это так, то передвигаем правую границу поиска, то есть будем уменьшать искомую длину квадрата (ведь вместилось больше, чем нужно дипломов). И повторяем поиск до тех пор, пока левая и правая граница поиска не окажутся рядом.

Заметим, что навык решения задач методом бинарного поиска по ответу пригодится вам почти на каждой серьезной олимпиаде.

Задание

1. Ниже приведена программа на языке C++ .

```
#include<iostream>
using namespace std;
const int n=20;
int a[20]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int binary_search(int b){
     int left=-1, right=n-1, middle;
     while(right-left>1) {
          middle=(right+left)/2;
          if (a[middle]>=b)
               right=middle;
          else
               left=middle;
     }
     return right;
}
int main()
{
     int b;
     cin >>b;
     cout<<binary_search(b);
     return 0;
}
```

а) укажите значения переменных *left* и *right* после работы алгоритма бинарного поиска, если в начале программы был задан массив int a[20]={1,2,3,4,5,6,7,8,8,8,8,12,13,14,15,16}. b=8.

б) int a[20]={1,2,3,4,5,6,7,8,8,8,8,12,13,14,15,16}. b=18.

в) int a[20]={1,2,3,4,5,6,7,8,8,8,8,12,13,14,15,16}. b=14.

г) сколько итераций выполнит алгоритм в худшем случае для поиска заданного числа?

2. Сколько нужно вопросов, чтобы наверняка отгадать (целое) число от 1 до 10^{100}, используя двоичный поиск? **Ответ. 333**

3. Сколько вопросов понадобится для отгадывания числа от 1 до 1000, если в качестве первого вопроса обязательно надо спросить «делится ли число на 3»? **Ответ. 11**

4. Реализуйте самостоятельно функции lower_bound, upper_bound.

5. Сколько потребуется шагов алгоритма бинарного поиска, чтобы начиная с отрезка длины r-l=2 дойти до отрезка длины меньше 10^{-10}? **Ответ**. 34

6. Напишите рекурсивный алгоритм бинарного поиска. Какой алгоритм более эффективен – рекурсивный или нерекурсивный? Дайте объяснение вашего ответа.

Литература

А.Шень. Программирование: теоремы и задачи. – М.: МЦНМО, 2004. Второе издание, исправленное и дополненное.

Курс сайта www.stepic.org «Введение в теоретическую информатику». Александр Шень.

Курс «Олимпиадные задачи по программированию». М.С. Густокашин. Электронный ресурс. http://prog.mathbaby.ru/dokuwiki/lib/exe/fetch.php?media=2015-9m:9m-binsearch-lection.pdf

Ворожцов А.В., Винокуров Н.А. Лекции «Алгоритмы: построение, анализ и реализация на языке программирования Си». – М.: Издательство МФТИ, 2007.