Алгоритм бинарного поиска

Бинарный поиск применяется для решения произвольных уравнений вида $f(x) = 0$ при условии, что известны две точки a и b такие, что $f(a) \leq 0$, $f(b) \geq 0$ и функция $f(x)$ непрерывна на интервале $[a; b]$

Идея решения очень проста. Берем точку c на середине отрезка $[a; b]$

$$c = \frac{a+b}{2}$$

и находим значение $f(c)$. Если $f(c) < 0$ то продолжаем поиск на интервале $[c, b]$, иначе продолжаем поиск на интервале $[a, c]$.

За один шаг алгоритма длина интервала уменьшается в 2 раза, поэтому после k шагов длина отрезка будет равна $\frac{b-a}{2^k}$, что позволит нам найти приближенное решение уравнения с любой заранее заданной точностью. Для оценки количества итераций можно воспользоваться приближенным равенством $2^{10} \approx 10^3$. Таким образом, за 100 итераций можно получить точность $(b-a)10^{-30}$, чего достаточно для решения практически любой задачи.

Замечание 1

Если уравнение имеет несколько решений на заданном интервале, то алгоритм найдет одно из них.

Замечание 2

Если функция убывает, то есть $f(a) \geq 0$ и $f(b) \leq 0$, то можно перейти к функции $f'(x) = -f(x)$ и выполнить алгоритм для функции $f'(x)$

Реализация

При реализации алгоритма на одном из языков программирования можно хранить интервал поиска в переменных `L` и `R`. Тогда основная часть алгоритма примет вид

```
L:=a;
R:=b;
for i:=1 to 100 do begin
   C:=(L+R)/2;
   V:=... {вычисляем f(C)}
    if V<0 then
       L:=C
    else
       R:=C;
end;
writeln(L);
```

Целочисленный бинарный поиск

Пусть задана некоторая последовательность пронумерованных элементов произвольного множества, причем номера элементов образуют непрерывный отрезок целых чисел $[a; b]$. Кроме того, определен некоторый предикат — логическая функция, позволяющая вычислить для каждого элемента множества логическое значение ноль или один, причем для всех элементов с номерами меньшими k это логическое значение всегда ноль, а для всех элементов с номерами большими или равными k — всегда один. Задача целочисленного бинарного поиска заключается в нахождении этого числа k.

Элементы, на которых выполняется двоичный поиск могут храниться в массиве, но это не обязательно. Логическая функция может быть реализована в программе в виде отдельной функции `bool␣test(int␣val)`, но может быть и просто подставлена как условие в инструкции ветвления.

Реализация алгоритма очень проста. Заведем две переменные: `left` и `right`, которые будут задавать диапазон номеров элементов $[left; right]$, в который обязательно входит мскомое число k. Будем сжимать диапазон поиска, поддерживая следующий инвариант: $test(left) = 0$ и $test(right) = 1$.

Тогда, программа примет такой вид.

```
int left=a-1;
int right=b+1;
while (left+1<right) {
  int mid=(left+right)/2;
  if (test(mid))
    right=mid;
  else
    left=mid;
}
cout<<right;
```

Ответьте самостоятельно на следующие вопросы.

1. Почему в `left` присваивается `a-1`, а не просто `a`?

2. Почему в `right` присваивается `b+1`, а не просто `b`?

3. Почему условие цикла имеет вид `left+1<right`?

4. Почему после завершения цикла ответ записан в переменной `right`, а не в `left`?

5. Будет ли программа работать если все логические значения равны нулям или все равны единицам? Что она выведет в ответ в этом случае?

Стандартные алгоритмы `low_bound(first,last,value)` и `upper_bound(first,last,value)`, примененные к итераторам произвольного доступа, также используют бинарный поиск. Легко понять, что условием в этом случае будет `*mid>=val` для `lower_bound` и `*mid>val` для `upper_bound`.

Тернарный поиск

Тернарный поиск применяется для нахождения минимума функции $f(x)$ в интервале $[a; b]$ при условии, что функция на этом интервале сначала убывает, проходит через минимум, потом возрастает.

В отличие от бинарного поиска, интервал $[a; b]$ делится на три равные части точками c_1 и c_2 по формулам $c_1 = \frac{2a+b}{3}$, $c_2 = \frac{a+2b}{3}$ Далее вычисляются значения функции в точках c_1 и c_2. Если $f(c_2) > f(c_1)$, то поиск продолжается в интервале $[a; c_2]$, иначе в интервале $[c_1; b]$.

Алгоритм работает корректно, так как если точки c_1 и c_2 находятся по разные стороны от минимума, то вне зависимости от выполнения условия минимум останется в интервале поиска. Если минимум находится справа от c_2 то функция убывает на отрезке от a до c_2 и $f(c_1) > f(c_2)$, и минимум останется в интервале поиска $[c_1; b]$.

Аналогично, если минимум находится слева от c_1 то функция возрастает на отрезке от $_1$ до b и $f(c_1) < f(c_2)$, и минимум останется в интервале поиска $[a; c_2]$.

За один шаг алгоритма длина интервала уменьшается в $\frac{2}{3}$ раза, а за два шага в $\frac{4}{9}$ раза. Поскольку $\frac{4}{9} < \frac{1}{2}$ алгоритм сходится примерно в два раза медленнее, чем простой бинарный поиск. Таким образом, в большинстве случаев для получения ответа будет достаточно 200 итераций.

Замечание 1

Алгоритм может работать некорректно, если функция на отрезке содержит более одного интревала непрерывного убывания или возрастания.

Замечание 2

Для нахождения максимума функции в заданном интервале можно перейти к функции $f'(x) = -f(x)$ и выполнить алгоритм для функции $f'(x)$

Реализация

При реализации алгоритма для минимума на одном из языков программирования можно хранить интервал поиска в переменных `L` и `R`. Тогда основная часть алгоритма примет вид

```
L:=a;
R:=b;
for i:=1 to 100 do begin
   C1:=(2*L+R)/3;
   C2:=(L+2*R)/3;
   V1:=... {вычисляем f(C1)}
   V2:=... {вычисляем f(C2)}
   if (V1>V2) then
      L:=C1
   else
      R:=C2;
end;
writeln(L);
```