Бинарный поиск

Теперь мы перейдём к новой концепции, основанной на идеи *«разделяй и властвуй»* (или *«уменьшай (наполовину) и решай»*, по другой классификации). Смысл достаточно прост: пусть у нас есть какая-то большая задача, которую нужно решить, тогда разобьём её на две примерно равных подзадачи, решим их и после как-то объединим решения. Такой алгоритм нам уже встречался, когда мы проходили сортировку слиянием.

Ну что ж, приступим, наконец, к нашему бинарному поиску: пусть у нас есть функция f, у которой мы хотим найти корень x_0 на участке $[l;r]$, при этом мы знаем, что функция всегда отвечает условию $f(x_1) \leqslant f(x_2)$ для $l \leqslant x_1 \leqslant x_2 \leqslant r$, а также $f(l) \cdot f(r) < 0$ (аналогично условию, что $l < x < r$). Тогда мы можем поступить следующим образом: возьмём точку $m = \frac{l+r}{2}$ и вычислим $f(m)$. Теперь возможны три варианта:

1. Если окажется, что $f(m) = 0$, то корень мы точно нашли и его можно вернуть.
2. Если же $f(l) \cdot f(m) > 0$, то значит $f(l)$ и $f(m)$ — одного знака, а следовательно x_0 точно не может лежать внутри отрезка $[l;m]$, то есть мы можем присвоить $l = m$ не потеряв решений.
3. Аналогично, при $f(m) \cdot f(r) > 0$, имеем $f(m)$ и $f(r)$ — одного знака, а следовательно x_0 точно не может лежать внутри отрезка $[m;r]$, то есть мы можем присвоить $r = m$ не потеряв решений.

Во-первых, заметим, что такой алгоритм пользуется только свойством $f(l) \cdot f(r) < 0$, при этом, после одного шага это свойство сохраняется, а значит, такой алгоритм можно применять до тех пор, пока не будет достигнута нужная точность (на каждом шаге границы отрезка $[l;r]$ сужаются и можно остановиться, когда отрезок станет достаточно маленьким согласно условию задачи). А во-вторых, поймём, что на каждом шаге длина отрезка уменьшается в 2 раза, а значит, наш поиск совершит $O(\log n)$ шагов, где n — длина диапазона, в котором мы ищем.

И, в-третьих, добавим, что главное от функции — неубывание или невозрастание (далее будем называть это свойство монотонностью, хотя это и не вполне корректно), поэтому функция, отвечающая условию $f(x_1) \geqslant f(x_2)$ для $l \leqslant x_1 \leqslant x_2 \leqslant r$, тоже бы подошла.

Поиск элемента в наборе

Представим, что перед нами стоит такая задача: дан набор чисел a длины n, для которого нам нужно ответить на q запросов вида: есть ли элемент x внутри a.

Простое решение, которое можно придумать, будет работать за $O(q \cdot n)$: для каждого запроса будем перебирать все элементы и проверять, нашёлся ли запрошенный x. Но такой способ явно не оптимальный, ведь можно придумать решение с использованием множеств: сохраним набор a внутрь множества s и операции поиска будем выполнять уже внутри s, ведь там они будут работать за $O(\log n)$, а значит итоговая сложность алгоритма составит $O(q \log n)$. Но такой способ мы использовать не хотим, потому что он требует использование структуры данных, а это влечёт за собой дополнительные накладные расходы. Так что будем придумывать способ, использующий бинарный поиск.

Первое, что нам нужно — монотонная функция f, у которой мы будем искать корень. Поскольку требуется монотонность, то вполне логично будет отсортировать весь набор a (для конкретности будем сортировать по неубыванию, хотя по невозрастанию тоже можно). Теперь становится понятно, как нам нужно определить функцию f: $f(k) = a_k - x$, где x — искомое значение.

Теперь перейдём к самой реализации алгоритма. Для начала возьмём границы нашего исходного отрезка: $l = -1$ и $r = n$ (наши элементы в a имеют индексы $0, 1, \ldots, n-1$, а такие l и r как бы указывают на фиктивные элементы, которые можно считать $a_{-1} = -\infty$ и $a_n = \infty$).

Также для удобства понимания работы с индексами договоримся, что у нас всегда будут выполняться условия $a_l < x \leqslant a_r$. Тогда наш алгоритм будет выглядеть следующим образом:

```
while (r - l > 1) {                 // можно закончить поиск, когда l и r стали соседними
    int m = (r + l) / 2;            // вычисляем середину
    (a[m] < x ? l : r) = m;         // тернарным оператором очень удобно двигать границы :)
}
```

Понятно, что согласно изначальному условию если ответ и содержится, то только в элементе a_r, поэтому если $r = n$ или $a_r > x$, то элемент не нашёлся, а иначе $a_r = x$ и это первый элемент равный x, и поэтому такой бинарный поиск называется *«lower bound»*.

Можно было договориться об инварианте $a_l \leqslant x < a_r$, и тогда если x не нашёлся бы в случаях $l = -1$ и $a_l < x$, а иначе $a_l = x$ было бы последним подходящим элементом. Если бы наш алгоритм возвращал индекс r, то такой бинарный поиск назывался бы *«upper bound»*

И понятно, что алгоритм, решающих исходную задачу работал бы за $O(n \log n + q \log n)$, где первое слагаемое возникает из-за сортировки, а второе — ответы на запросы.

Встроенный бинарный поиск

Разумеется, в C++ уже есть встроенные функции, которые ищут значения внутри отсортированных контейнеров. Они называются `lower_bound`, `upper_bound` (откуда же взялись такие названия :)) и `binary_search`, использовать их можно так:

```
vector<int> p1 = {2, 3, 5, 7, 11};
int p2[] = {2, 3, 5, 7, 7, 11, 13};
upper_bound(p1.begin(), p1.end(), 8) - p1.begin();      // 4, так как 8 < p1[4] = 11
lower_bound(p2, p2 + 7, 7) - p2;                        // 3, так как 7 <= p2[3] = 7
binary_search(p2, p2 + 7, 17);                          // false, так как не нашлось
```

Только перед этим необходимо подключить нужный файл:

```
#include <algorithm>                // содержит lower_bound, upper_bound и binary_search
```

И, кончено же, встроенные алгоритмы бинарного поиска работают за $O(\log n)$[1].

Бинарный поиск по ответу

Есть целый набор задач, в которых относительно легко понять, является ли какое-то число ответом, при этом выписать явную формулы для вычисления этого числа достаточно сложно. Именно в таких задачах используется *«бинарный поиск по ответу»*.

Пусть у нас есть $n > 0$ дипломов шириной w и высотой h, при этом дипломы нельзя поворачивать и накладывать друг на друга. Нужно найти минимальное целое x такое, что все дипломы войдут на квадратный стенд $x \times x$, не вылезая за его пределы.

Во-первых заметим, что в этой задаче присутствует монотонная функция: если x является ответом, то для всех $x' > x$ стенд $x' \times x'$ также является ответом, а для всех $x' < x$ стенд $x' \times x'$ ответом точно не является. Так что приняв $l = 0$ и $r = n \cdot \max(w, h)$ мы сможем сделать бинарный поиск по ответу (стенда 0×0 точно не хватит, а вот $(n \cdot \max(w, h)) \times (n \cdot \max(w, h))$ — точно хватит).

Теперь остаётся понять, является ли какое-то число m ответом. А сделать это действительно не сложно, ведь на стенд размером $m \times m$ войдёт $\left\lfloor \frac{m}{w} \right\rfloor$ дипломов по горизонтали и $\left\lfloor \frac{m}{h} \right\rfloor$ по вертикали

[1] Это справедливо для контейнеров, в которых есть доступ к произвольному элементу; для множеств и отображений нужно использовать методы самих контейнеров, которые мы изучили раньше: `s.lower_bound(x);`.

(такие скобочки — это округление вниз к ближайшему целому). А значит итоговая функция, по которой совершается поиск является $f(m) = \left\lfloor \frac{m}{w} \right\rfloor \cdot \left\lfloor \frac{m}{h} \right\rfloor$, и если $f(m) < n$, то стенд слишком маленький, значит нужно сделать $l = m$; а иначе $f(m) \geqslant n$, а значит стенд достаточного размера, поэтому делаем $r = m$. Итоговый ответ окажется в границе r.

Понятно, что сложность такого решения будет $O(\log(n \cdot \max(w, h)))$, ведь мы ищем среди $n \cdot \max(w, h)$ потенциальных ответов. Также понятно, что при любом бинарном поиске по ответу нам нужно будет делать $O(\log t)$ операций поиска, где t — размер диапазона, в котором мы ищем ответ.

Вещественный бинарный поиск

Теперь рассмотрим ещё одну задачу, чтобы понять широту применимости бинарного поиска. Пусть нам нужно вычислить квадратный корень числа n с точностью 10^{-6}, при этом использовать встроенные функции корня (и возведения в степень) запрещается.

Здесь достаточно очевидно, что бинарный поиск у нас будет по функции $f(m) = m^2$. Если оказалось, что $f(m) = n$, то m — искомое число; если же $f(m) < n$, то такого m ещё мало, поэтому нужно сделать $l = m$; а иначе $f(m) > n$ и делаем $r = m$.

Единственным отличием от предыдущих задач будет то, что l и r должны быть вещественными числами, ведь используя целые числа мы не сможем достичь нужной точности. Останавливать же поиск мы можем когда разность $r - l$ станет слишком маленькой, например меньше 10^{-7}. Но поскольку операции с вещественными числами имеют погрешности, то заданная разность может не достигаться и может получиться бесконечный цикл. Поэтому иногда делают бинарный поиск с заданным числом итераций (например 120, потому что $\log_2 10^{36} \approx 120$, а диапазона 10^{36} почти всегда хватает, ведь положительные числа в задачах почти всегда лежат в диапазоне $[10^{-18}; 10^{18}]$).

Такой бинарный поиск называется *«вещественным»* (потому что работает с вещественными числами), его сложность, как обычно, пропорциональна $O(\log t)$, где t — размер диапазона. Также приятным свойством вещественного бинарного поиска является факт, что после окончания поиска обе границы (l и r) оказываются достаточно близки к ответу, поэтому ответом можно считать любое из чисел.

Тернарный поиск

Теперь рассмотрим алгоритм, очень похожий на бинарный поиск — тернарный поиск. Его суть заключается в следующем: пусть у нас есть функция $f(x)$, у которой ровно один минимум x_0 (с максимумом аналогично) на отрезке $[l; r]$, разбивающий функцию на две монотонных части $[l; x_0]$ и $[x_0; r]$.

Тогда на каждом шаге мы можем делить текущий отрезок на три части: $m_1 = l + \frac{r-l}{3} = \frac{2l+r}{3}$ и $m_2 = r - \frac{r-l}{3} = \frac{l+2r}{3}$. И если оказалось, что $f(m_1) \leqslant f(m_2)$, то на участке $[m_2; r]$ точно нет ответа, поэтому можем присвоить $r = m_2$. Аналогично при $f(m_1) \geqslant f(m_2)$, на участке $[l; m_1]$ точно нет ответа, поэтому можем присвоить $l = m_1$.

Заканчивать такой поиск нужно, когда отрезок станет достаточно маленьким или после достаточного числа итераций. Сложность тернарного поиск $O(\log t)$, где t — длина диапазона (на самом деле понятно, что тернарного поиск основание логарифма 1.5, а бинарного — 2, но эта константа в асимптотике опускается, поэтому у поисков одинаковые сложности).

Применять тернарный поиск можно, например, в задачах по геометрии, когда вам не хочется выводить явную формулу. Ещё стоит отметить, что достаточно часто можно обойтись без тернарного поиска, заменив его на бинарный поиск по производной.