Двоичный поиск

Двоичный поиск, он же бинарный поиск или бинпоиск, он же алгоритм деления пополам или дихотомия — это целая серия алгоритмов, объединённых одной идеей. Мы их последовательно рассмотрим.

Часть I. Вещественный двоичный поиск

1. Прочность нити на разрыв. Для начала рассмотрим следующую задачу — не совсем по программированию. Пусть у нас есть, например, нитка. Мы можем подвешивать к этой нитке различные грузы, при этом если мы подвесим слишком большой груз, то нитка порвётся. Соответственно, есть некоторая пороговая масса груза m_0 такая, что от любой бóльшой массы нитка рвётся, а от любой меньше — нет. Поставим задачу определить эту пороговую массу.

(Может возникнуть вопрос: а что происходит, если масса точно равна m_0? Будем считать, что происходит что-то конкретное, что имеет место только один из следующих вариантов: либо при m_0 нитка всегда рвётся, либо всегда нет. На самом деле, это нам совершенно несущественно, как будет видно ниже.)

Будем считать, что запасы нитки у нас достаточно велики — её хватит на проведение большого количества экспериментов. Будем также считать, что нитка абсолютно невесома (не рвётся под своей тяжестью) и однородна, т.е. для любого её куска пороговая масса одна и та же — все-таки у нас олимпиады по программированию, а не по физике[1]. Также будем считать, что нам доступны грузы абсолютно любой (неотрицательной, конечно) массы, с бесконечной точностью (т.е. что мы, например, можем сделать груз массы ровно $42\pi + 137.035999074$ грамм.

Итак, постараемся определить m_0 за минимальное число экспериментов. Точнее, конечно, *абсолютно точно* определить m_0 у нас не получится, т.к. для этого надо определить бесконечное число десятичных знаков, которые могут быть в записи m_0. Поэтому выберем маленькое число ε и постараемся найти m_0 с точностью ε, т.е. найти такие l и r, что $l \leqslant m_0 \leqslant r$, но $r - l \leqslant \varepsilon$.

Алгоритм достаточно прост. Мы знаем, что если к нити не подвешивать груз (т.е. подвесить груз массы 0), то она не порвётся. С другой стороны, будем считать, что мы знаем такую массу M, при которой нить точно порвётся (например, если это обычная швейная нить, то вряд ли она выдержит груз массой 1 тонну). Соответственно, обозначим эти две величины как l и r и далее будем их корректировать. А именно, на каждом шагу алгоритма возьмём среднее арифметическое масс l и r и поставим эксперимент: порвётся нитка от такой массы или нет? Если порвётся, то корректируем границу r, иначе l:

```
l:=0;
r:=m_big;  // та самая масса M, при которой нить точно рвётся
while r-l>eps do begin    // eps --- требуемая точностью
    m:=(r+l)/2;
    if (нить рвётся от массы m) then
        r:=m
    else l:=m;
end;
```

В результате у нас *всегда* при выполнении алгоритма будет так, что от массы r нить рвётся, а от массы l — нет. В итоге после окончания цикла мы и найдём такие l и r, что $r - l \leqslant eps$, и $l \leqslant m_0 \leqslant r$.

Вернёмся к вопросу о том, что происходит с ниткой при массе, точно равной m_0, а именно поймём, почему ответ на этот вопрос нам не важен. Действительно, во-первых, крайне маловероятно, что в некоторый момент выполнения алгоритма у нас m будет точно равно m_0. Мы ведь проверяем конечное число значений, а m_0 может быть любым вещественным числом — чтобы мы попали точно в m_0, нам должно очень крупно повезти (строго говоря, вероятность этого — например, с учётом случайности отношения M/m_0 — точно равна нулю). Но во-вторых, пусть даже на очередной итерации у нас получилось $m = m_0$, и пусть нитка порвалась. Тогда у нас будет $r = m_0$, и при всех дальнейших экспериментах нитка рваться не будет, т.к. везде далее будет $m < m_0$, в итоге получим $r = m_0$ и $l \approx m_0 - \varepsilon$ — все равно m_0 мы с нужной точностью нашли. Совершенно аналогично, если бы при $m = m_0$ нитка не порвалась бы, то мы получили бы в итоге $l = m_0$ и $r \approx m_0 + \varepsilon$ — в пределах заданной точности решение то же самое.

Какова сложность алгоритма? Сложность, конечно, будем измерять в количестве экспериментов. Изначально $r - l = M$, в конце $r - l \approx \varepsilon$, каждый раз $r - l$ уменьшается в два раза, поэтому общее количество итераций будет примерно $\log_2(M/\varepsilon)$. Это не очень много; например, если $M = 1e20$, а $\varepsilon = 1e - 20$, то $\log_2(M/\varepsilon) \approx 133$.

[1] Реальные нитки неоднородны и m_0 зависит от куска. См. *Всероссийские олимпиады по физике, 1992-2001* под ред. С. М. Козела, В. П. Слободянина, часть 2, задача 10.17.

2. Общая идея. Итак, общая идея уже достаточно понятна. Пусть у нас есть некоторый критерий, определяющий, что некоторые вещественные числа являются в каком-то смысле «хорошими», а некоторые — «плохими», причём все хорошие идут до всех плохих. Нам надо определить пороговое значение, где хорошие числа становятся плохими.

Мы поступаем очень просто: берём некоторые два числа, одно из которых (l_0) гарантированно является хорошим, а другое (r_0) — плохим, и далее много раз повторяем следующую операцию: вычисляем среднее арифметическое m этих двух чисел и смотрим: хорошее оно или плохое. Если хорошее, то мы можем присвоить $l := m$, иначе $r := m$. В итоге у нас постоянно l будет хорошим, а r плохим, и на каждом шагу разность $r - l$ будет уменьшаться в два раза — за примерно $\log_2(M/\varepsilon)$ итераций (где M — начальное значение разности $r - l$) мы достигнем точности ε.

Общий алгоритм выглядит почти так же, как и приведённый выше:

```
l:=l0;
r:=r0;
while r-l>eps do begin
    m:=(r+l)/2;
    if good(m) then
        l:=m
    else r:=m;
end;
```

Здесь `good(x)` — функция, возвращающая `true`, если x — хорошее число, и `false` — если плохое. Вообще, всегда, когда пишете деление пополам, выносите проверку «хорошести» числа в отдельную функцию, чтобы не загромождать основной цикл. (Исключение см. в следующем разделе.)

Как и в примере выше, нам совершенно не важно, является ли пороговое число «хорошим» или нет.

3. Поиск корня функции. Существенный частный случай вещественного двоичного поиска (да и вообще многие вещественные двоичные поиски можно объявить частным случаем этого) — это поиск корня монотонной непрерывной функции. Итак, пусть у нас есть функция $f(x)$, которая монотонно строго возрастает, по крайней мере на интересующем нас отрезке, (т.е. для любых x_1 и x_2 таких, что $x_1 < x_2$, выполняется $f(x_1) < f(x_2)$), и непрерывна (т.е. не имеет скачков) там же. И пусть у нас есть такие два числа l и r, что $f(l) \leqslant 0$, а $f(r) \geqslant 0$.

Есть соответствующая теорема математического анализа, что в таком случае на отрезке $[l, r]$ у функции есть ноль, т.е. такое значение x_0, что $f(x_0) = 0$, причём такое значение x_0 единственно. Его легко найти двоичным поиском: просто объявляем все такие x, что $f(x) < 0$, «хорошими», а все такие x, что $f(x) > 0$, — «плохими». (Как и раньше, нам не важно, каким определить собственно то единственное x_0, для которого $f(x_0) = 0$.) И легко пишем алгоритм:

```
while r-l>eps do begin
    m:=(r+l)/2;
    if f(m)<0 then   // можно и <=
        l:=m
    else r:=m;
end;
```

Здесь в условии if'а стоит сравнение $f(m) < 0$. Это — то самое исключение, про которое я писал в предыдущем разделе: если вы ищете корень функции, то можно в поиске в if'е написать сравнение функции с нулём. Но само вычисление $f(m)$ все равно вынесите в отдельную функцию.

Аналогично можно искать и корень уравнения $f(x) = c$, если вы знаете, что $f(l) \leqslant c$, а $f(r) \geqslant c$.

4. А если функция не монотонна или не непрерывна? Если функция строго возрастает, но разрывна, то есть риск, что у неё нет корня: что в некоторый момент она может делать скачок сразу от отрицательного значения до положительного (т.е. например $f(x) < 0$ при $x \leqslant x_0$, но $f(x) > 0$ при $x > x_0$). Несложно видеть, что в итоге l и r будут ограничивать ε-окрестность этого скачка. В принципе, это вполне разумно, тем более вряд ли можно добиться чего-то лучше — из-за погрешностей при операциях с вещественными числами программно отличить разрывную функцию от просто очень быстро возрастающей вряд ли получится.

Если функция монотонна, но убывает, а не возрастает, то решение тоже понятно: надо просто изменить знак проверки в `if`, ну или поменять местами присваивания l и r.

Если заранее неизвестно, возрастает функция или убывает (но известно, что она монотонна), то, конечно, надо ориентироваться на знаки $f(l)$ и $f(r)$. Можно заранее запомнить знак $f(l)$, но можно написать проще: просто в условии if'а написать `if f(l)*f(m)>0 then l:=m...`. Это даже легко понять: мы поддерживаем ситуацию «$f(l)$ и $f(r)$ имеют разные знаки».

Наконец, ещё один важный случай — если функция не является монотонной, но все равно изначально $f(l)$ и $f(r)$ разных знаков. Тогда аналогичный алгоритм найдёт *какой-нибудь* ноль (ну или скачок через ноль, если функция может быть разрывна). При этом, если мы заранее знаем, что $f(l) \leqslant 0$, а $f(r) \geqslant 0$, то в if'е можно писать условие $f(m) < 0$, если же мы только знаем, что $f(l)$ и $f(r)$ разных знаков, то либо надо заранее запомнить, кто какого знака, и учесть это в if'е (примерно так: `if (f(m)<0) xor flag`, здесь `flag` обозначает, что изначально было $f(l) > 0$), либо писать `f(l)*f(m)`.

Если же функция не строго монотонна, т.е. для $x_1 < x_2$ выполняется только $f(x_1) \leqslant f(x_2)$ (нестрогое! неравенство), то у функции может найтись целый отрезок, на котором она во всех точках равна нулю. В таком случае написанный выше алгоритм найдёт какую-то точку этого отрезка. (Правда, тут может показаться, что тут существенно, что мы будем делать в случае, если $f(m) = 0$. Но при надо всегда помнить, что все операции с вещественными числами неточны, поэтому, даже если $f(m)$ точно равно нулю, то в программе скорее всего получится $f(m)$ близкое к нулю, но не равное нулю — поэтому скорее всего вы просто найдёте какую-то точку этого отрезка, и ничего лучше.)

Общее всех этих случаев — что у нас изначально $f(l)$ и $f(r)$ разных знаков. Если это не так, то мы вообще не можем гарантировать наличия нуля, и метод деления пополам нам особенно не поможет. Можете подумать, какой у него будет результат, если условие разных знаков изначально не выполнено.

5. Что выводить? Действительно, что нам надо выводить как результат работы программы, т.е. как найденное значение x_0? Можно выводить и l, и r — если нам надо было найти решение с точностью ε, то оба эти значения подходят. Если вы боитесь попасть «на грань» точности, то просто сделайте в программе ε в несколько раз меньше чем требуется.

6. Решение без ε. Вместо того, чтобы гнать цикл пока $r - l > \varepsilon$, можно его выполнять, например, строго определённое количество раз. Например, можно его выполнять 100 раз — тогда гарантированно $r-l$ уменьшится в 2^{100} раз, чего очень часто бывает достаточно. Ну или сделайте цикл ещё больше раз, если 2^{100} вам мало. Это в некотором смысле обеспечивает вам наилучшую точность, которая достижима при данном количестве вычислений функции; если вместо количества итераций просто поставить отсечение по времени, то вы получите наилучшую точность, которая достижима за это время.

Кроме того, такой подход полезен, если вам надо не просто вывести ответ, а что-то с ним сделать. Иногда бывает так, что ответом на задачу является не само l, а некоторая функция от l, и изменение l ненамного приводит к достаточно сильному изменению ответа; например, если даже l отличается от нужного значения на $1e{-}5$, может оказаться, что вычисленный исходя из l ответ будет отличаться от нужного существенно сильнее, и может быть сочтён неправильным. В таком случае тоже полезно вычислить l более точно — либо задав ε поменьше, либо, что даже надёжнее, просто сделав очень много итераций.

Еще следует иметь в виду вот что. Если вы все-таки пишете деление пополам с ε, и ε у вас мало, а искомые l и r велики, то есть риск, что ε будет меньше той точности, с которой в компьютере представляются искомые l и r. Тогда в тот момент, когда разница l и r станет порядка этой точности, при дальнейших вычислениях l и r не будут меняться и деление пополам зациклится. (Например, в типе `double` хранится примено 15–17 десятичных цифр. Если в некоторый момент окажется, что r и l отличаются в последней хранимой цифре, а точность ε еще не достигнута, то $(l + r)/2$ может оказаться равно или l, или r, в результате чего поиск зациклится.)

7. Выбор l и r. Как уже многократно говорилось, надо выбрать l и r так, чтобы l было «хорошим», а r — «плохим» (в случае с функцией — чтобы l и r были разных знаков). В общем случае это нетривиальная задача, в каждом конкретном случае надо думать особо.

Бывает так, что границы заданы довольно естественно просто по смыслу области определения функции $f(x)$; например, если мы решаем методом деления пополам уравнение $\operatorname{tg} x = a$ при некотором a (конечно, можно было бы просто вычислить арктангенс без всякого двоичного поиска, но для примера рассмотрим решение методом деления пополам), то мы можем взять $l = -\pi/2$, $r = \pi/2$ — обратите внимание, кстати, что нам все равно не придётся вычислять $f(l)$ и $f(r)$, поэтому не страшно, что функция в этих точках обращается в бесконечность.

Бывает так, что границы легко найти; например, если мы решаем уравнение $x^2 = a$ при некотором $a > 1$ (аналогично, можно было бы извлечь корень, но для примера поговорим про деление пополам), то понятно, что нас устраивают $l = 0$, $r = a$.

Но важно также понимать, что совершенно не страшно завысить начальные значения l и/или r даже на несколько порядков величины — все равно деление пополам очень быстро сходится. Например, пусть вы знаете, что $f(0) \leqslant 0$ (и поэтому берете $l = 0$), но вам сложно оценить, при каком r получится $f(r) \geqslant 0$. Бывает так, что вы понимаете, что в вашей задаче ответ не может быть больше, например, $1e20$ (вообще, это должна быть какая-то нетривиальная задача, чтобы ответ был больше $1e20$, если все входные данные не очень велики), то берите $r = 1e20$. Или даже $1e100$. Это все не очень страшно, вы просто добавите себе пару сотен лишних итераций; если функция вычисляется быстро, то сотня лишних итераций не страшна. Правда, в этом случае вам надо быть абсолютно уверенным, что при достаточно больших аргументах функция все-таки становится положительной, а то вдруг она вообще всегда отрицательна...

Часть II. Целочисленный двоичный поиск

Теперь перейдём к целочисленному бинарному поиску — поиску, в котором нам требуется найти некоторое целое число. Как ни странно, этот вариант оказывается сложнее вещественного поиска.

1. Опять порог разрыва нити. Для начала вернёмся к нашей задаче про прочность однородной невесомой нити. Но пусть теперь мы не можем подвесить к ней произвольный груз, а только груз массой в целое число грамм.

Прежде чем обсуждать, как решить эту задачу, обсудим, а *что же, собственно, мы хотим получить*? Бессмысленно теперь спрашивать критическую массу, т.к. она, вообще говоря, может быть вещественной. Но понятно, что у нас до некоторой массы (m_*) *включительно* нить рваться не будет, а вот начиная с массы ($m_* + 1$) *включительно* и выше нить рваться будет. Поэтому нас будут интересовать именно две такие *соседние* массы l и r (соседние в том смысле, что $r = l + 1$), что при массе l нить ещё не рвётся, а вот при массе r нить уже рвётся.

Заметьте, что постановка вопроса абсолютно симметрична относительно l и r: нельзя сказать, что правильный ответ l, но не r; точно также нельзя сказать, что правильный ответ r, а не l. Мы можем ставить два симметричных вопроса: «при какой максимальной массе нить ещё не рвётся» (и ответ будет l) или «при какой минимальной массе нить уже рвётся» (и ответ будет r). Они оба симметричны, поэтому бессмысленно говорить, что какой-то из них более правильный или более логичный, поэтому и бессмысленно говорить, что l или r более правильный или логичный ответ. Поэтому мы будем искать именно такую *пару* чисел l и r, а не какое-то одно число.

Как это делать? Кажется достаточно просто:

```
l:=0;
r:=m_big;  // какая-нибудь масса M, при которой нить точно рвётся
while ??? do begin
    m:=(r+l) div 2;
    if (нить рвётся от массы m) then
        r:=m
    else l:=m;
end;
```

Но что написать в условие? Как мы уже обсудили выше, мы хотим найти два соседних числа l и r. Поэтому цикл будем выполнять пока $r - l > 1$ (или, что то же самое $r > l + 1$), т.е. пока они ещё не стали соседними. В результате у нас всегда поддерживается состояние «от массы l нить не рвётся, а от массы r нить рвётся».

На каждом шагу расстояние между l и r уменьшается... Уменьшается ли? Понятно, что оно не может уменьшаться до бесконечности: и l и r целые числа. Поэтому ясно, что в некоторый момент окажется, что $m = l$ или $m = r$ и расстояние перестанет уменьшаться. Но также несложно видеть, что **если** $r - l > 1$**, то** $l < m < r$, причём неравенства строгие. Т.е. *в этой реализации* бинарного поиска всегда внутри цикла будет $l < m < r$ со строгими неравенствами, и поэтому расстояние будет уменьшаться.

Но это очень важный момент. Если бы мы в какой-нибудь другой задаче написали бы цикл с другим условием

```
// НЕПРАВИЛЬНЫЙ КОД!
...
while r>l do begin
    m:=(r+l) div 2;
    if ... then
        r:=m
    else l:=m;
end;
```

то программа очень легко могла бы зациклиться. Действительно, в какой-то момент окажется $l = r - 1$ (например, $l = 4$, $r = 5$). Тогда получится $m = l$ (в примере $m = 9 \text{ div } 2 = 4 = l$), выполнится вторая ветка if'а, и в результате ни l, ни r не изменятся (останется $l = 4$, $r = 5$). Цикл выполнится ещё раз с тем же результатом, и так далее.

Это есть первая ловушка, в которую вы можете попасть, когда пишете деление пополам: когда l и r сходятся очень близко, в очередной момент может оказаться, что за одну итерацию ни l, ни r не изменились, и программа зацикливается. Всегда помните про это, если будете выдумывать свой способ написания бинарного поиска (хотя это стоит делать только в совсем особенных случаях); в частности, никогда не пишете деление пополам с условием `while r>l` (или `while r<>l` или ему эквивалентным).

Итак, правильное решение задачи о целочисленном пределе прочности нити следующее:

```
l:=0;
r:=m_big;  // какая-нибудь масса M, при которой нить точно рвётся
while r-l>1 do begin
    m:=(r+l) div 2;
    if (нить рвётся от массы m) then
        r:=m
    else l:=m;
end;
```

Это действительно корректное решение, оно действительно работает. За какое время? Несложно видеть, что за $O(\log M)$ — каждая итерация уменьшает разницу $r - l$ примерно в два раза.

2. Общий случай. Аналогично вещественному двоичному поиску, тут тоже можно сформулировать алгоритм в общем случае. Итак, пусть у нас все целые числа разделены на две категории: «хорошие» и «плохие», при этом все хорошие идут до всех плохих, и мы знаем два числа: l_0 — хорошее, и r_0 — плохое.

Нам надо найти границу между хорошими и плохими числами, т.е. такое хорошее число l и такое плохое число r, что $r - l = 1$. (Как и выше, постановка задачи симметрична относительно l и r.) Решение понятно:

```
l:=l0;
r:=r0;
while r-l>1 do begin
    m:=(r+l) div 2;
    if good(m) then
        l:=m
    else r:=m;
end;
```

Это можно называть *поиском скачка монотонной логической функции*. В том смысле, что у вас есть функция `good`, она логическая, т.е. возвращает значение логического (`boolean`) типа, и она монотонная, т.е. если $i < j$, то $good(i) \geqslant good(j)$ (мы считаем, что `true` > `false`; функция получается «убывающей», но могла бы быть и возрастающей, что соответствовало бы тому, что сначала идут плохие числа, а потом хорошие, и потребовалось бы просто поменять местами l и r в ветках if'а). И нам надо найти её скачок, т.е. два таких соседних числа l и r, что $good(l) = true$, а $good(r) = false$.

Обратите внимание на ещё один важный момент. Приведённая выше программа никогда не будет вызывать функцию `good` с аргументами l_0 или r_0; важны только значения для промежуточных аргументов. Проще говоря, не важно, являются ли l_0 и r_0 хорошими или плохими числами — главное, чтобы между ними все хорошие шли до всех плохих. Фактически, мы мысленно подразумеваем, что l_0 хорошее, а r_0 плохое, но никогда это не проверяем. (Аналогично замечанию про тангенс выше в вещественном поиске.) Это нам будет важно в дальнейшем.

3. Что же является ответом? В вещественном двоичном поиске l и r различались несущественно — разница между ними была меньше ε, и поэтому было все равно, какое из чисел выводить. Но в целочисленном двоичном поиске l и r различаются существенно, и поэтому вопрос о том, что из них считать ответом, нетривиален.

Но, как я уже неоднократно писал выше, с точки зрения бинарного поиска l и r равнозначны. Поэтому решение о том, что считать ответом, зависит от той задачи, в которой вы решили применить бинарный поиск. Могут быть задачи, где ответом будет наибольшее хорошее число — тогда ответ будет l (например, если бы в задаче про нитку стоял бы вопрос «какой максимальный целочисленный вес выдерживает нить?»). Могут быть задачи, где ответом будет наименьшее плохое число — тогда ответ будет r. Могут быть задачи, где ответ вычисляется как-нибудь ещё более сложно, и т.д.

Главное — что бинарный поиск вам нашёл границу «хороших» и «плохих» чисел, а что делать с этим дальше — уже ваше дело, зависит от задачи.

Часть III. Поиск элемента в отсортированном массиве

1. Постановка задачи. Очень важный частный случай бинарного поиска — это поиск заданного элемента в отсортированном массиве. В простейшей постановке задача звучит так. Вам дан массив a, и гарантируется, что он отсортирован по неубыванию: $a[i] \leqslant a[j]$ если $i < j$. Кроме того, вам дано число x, и от вас требуется найти такой индекс i, что $a[i] = x$, или сообщить, что такого индекса нет.

Нередко, когда говорят о бинпоиске, имеют в виду именно эту задачу, но написать программу двоичного поиска элемента в отсортированном массиве, не учитывая то, что говорилось выше, — очень сложно[2].

Но с учётом того, что мы уже знаем, написать эту программу становится очень легко. Надо только определить, какие числа мы будем считать «хорошими», а какие — «плохими». Давайте, например, определим так: «хорошими» мы будем считать такие числа i, что $a[i] < x$, а «плохими» — такие, что $a[i] \geqslant x$. (Обратите внимание, что хорошими и плохими мы называем *индексы* массива, а не сами значения массива.) В результате у нас все хорошие значения будут идти до плохих значений, и можно применить деление пополам.

Заметим, что, в отличие от вещественного двоичного поиска, здесь довольно важно, к какому варианту отнести ситуацию точного равенства $a[i] = x$, т.к. такие элементы вполне могут существовать, и, более того, их может быть несколько. Пока поступим так, как написано выше: будем считать такие индексы плохими; подробнее обсудим ниже.

Только чему взять равным l_0 и r_0? Вспомним, что бинарному поиску не важно, хорошие или плохие числа l_0 и r_0 — важны только промежуточные числа. Поэтому — внимание! — можно взять $l_0 = 0$, а $r_0 = N + 1$, если элементы в массиве у нас занумерованы от 1 до N.

[2] Д. Кнут утверждает, что, хотя первый раз двоичный поиск был опубликован в 1946 году, первая реализация двоичного поиска *без ошибок* появилась только в 1962 году. Есть ещё один известный эксперимент, что только 10% программистов могут написать двоичный поиск без багов. См. подробнее `http://habrahabr.ru/post/91605/` и указанные там ссылки.

Т.е. мы берём l_0 перед первым элементом массива, а r_0 — после последнего. Можно мысленно считать, что перед первым элементом массива у нас идёт бесконечно большое отрицательное число (которое меньше всех других и гарантированно меньше x), а после последнего — бесконечно большое положительное число. Мы все равно никогда не будем реально проверять, чему равно $a[l_0]$ или $a[r_0]$.

Итоговый код получается следующий:

```
l:=0;
r:=n+1;
while r-l>1 do begin
    m:=(r+l) div 2;
    if a[m]<x then
        l:=m
    else r:=m;
end;
```

2. А что является тут ответом? Напомним постановку задачи: надо найти такой индекс i, что $a[i] = x$, либо сообщить, что такого нет. Как это сделать?

Вспомним определение хороших и плохих чисел: $a[l] < x$, а $a[r] \geqslant x$ всегда. Поэтому в конце, когда $r = l + 1$, это значит, что мы нашли два числа подряд такие, что одно меньше x, а второе — больше или равно x.

Тогда понятно, что если $a[r] = x$, то ответ — r, иначе такого индекса нет. Правда, есть сложность: может оказаться $r = r_0 = N + 1$ (если x больше всех элементов массива, см. также ниже). Это надо не забыть и явно проверить, чтобы не получить выход за пределы массива.

3. Левый и правый двоичные поиски. Из написанного выше несложно видеть, что, если искомое число в массиве есть, то мы не просто его найдём, но найдём *самое левое* (т.е. с наименьшим индексом) его вхождение.

А если мы хотим найти *самое правое*? Это тоже довольно легко: просто надо i считать хорошим числом, если $a[i] = x$, т.е. поменять строгое на нестрогое неравенство и наоборот в определении хороших и плохих чисел. Индекс будем считать хорошим, если $a[i] \leqslant x$, и плохим наоборот. Получаем следующее решение:

```
l:=0;
r:=n+1;
while r-l>1 do begin
    m:=(r+l) div 2;
    if a[m]<=x then    // отличие только в этой строчке!
        l:=m
    else r:=m;
end;
```

(Чтобы определить ответ, проверять теперь надо, конечно, $a[l]$: если он равен x, то искомый индекс — l, иначе число x в массиве отсутствует. Тут также придётся особо проверить случай $l = 0$.)

Эти два варианта двоичного поиска называются *левым* и *правым* двоичным поиском.

4. Бинарный поиск как поиск места вставки. Давайте ещё обсудим более подробно, что же именно происходит, если нужный элемент в массиве не найден. Тогда мы находим два таких соседних индекса l и r, что $a[l] < x$, а $a[r] > x$. Это можно определить так: мы находим то место, куда надо было бы вставить значение x, если бы мы хотели вставить его в массив, сохранив отсортированность — а именно, его надо вставить между элементами l и r.

В частности, может оказаться, что $l = 0$ и $r = 1$ — это значит, что x меньше всех элементов массива. Может оказаться, что $l = n$, $r = n + 1$ — т.е. x больше всех элементов массива. Но утверждение, что мы нашли место, куда надо было бы вставить x, верно во всех случаях, и это зачастую оказывается полезно.

В частности, обратите внимание, что, например, правый поиск работал бы, даже если бы мы взяли $l_0 = 1$, но тогда мы не смогли бы отличить ситуацию «x надо вставить перед первым элементом» и «x надо вставить сразу после первого элемента». Аналогично, левый поиск работал бы, если бы мы взяли $r_0 = n$, но мы бы не отличили случай «x надо вставить после всех элементов» и «x надо вставить перед последним».

Терминология поиска места, куда надо вставить x, также достаточно просто работает и в случае, когда x найдено. Для левого поиска получается l равно последней позиции перед первым вхождением x, а r — первому вхождению x. Таким образом, левый поиск показывает, куда надо было бы вставить число x, чтобы сохранить упорядоченность, причём если такие элементы в массиве уже есть, то надо вставить перед первым таким элементом. Аналогично, правый поиск ищет, куда надо вставить число x, чтобы сохранить упорядоченность, причём если такие элементы в массиве уже есть, то он пытается вставить после последнего из них.

5. Ошибки в целочисленном бинарном поиске. Выше приведён очень простой и надёжный код поиска элемента в отсортированном массиве. В принципе, есть много разных других способов реализации, но многие из них сложнее или менее надёжные; не случайно эта задача считается весьма сложной.

Упомяну несколько вариантов кода, которые могут показаться разумными, но которые тем не менее имеют те или иные недостатки или вообще не работают.

Во-первых, может появиться желание во внутреннем if'е разобрать случай точного попадания в x: если $a[m] = x$, то прервать работу. Это имеет три недостатка. Во-первых, теперь теряются преимущества левого и правого поисков; вы никогда не можете быть уверены, какое вхождение вы найдёте, если их несколько. Во-вторых, может возникнуть желание писать цикл с условием `while l<>r` или даже `while l<=r` («пока ещё остаются нерассмотренные элементы»), но тогда есть, как указывалось выше, риск зацикливания. В-третьих, код становится в полтора раза сложнее из-за лишнего условия.

Ещё стандартный подход — взять изначально $l = 1$, $r = n$. Это имеет два недостатка. Во-первых, вы не сможете отделить случаи «x меньше всех элементов массива» и «x надо вставить между первым и вторым элементами», и аналогично не сможете отличить случаи «x больше всех элементов массива» и «x надо вставить между последние и предпоследним элементами». Более того, если x все-таки нашёлся, то он может быть как в элементе l, так и в элементе r, поэтому после основного цикла поиска потребуется ещё одна проверка.

Есть ещё один вариант, который долгое время считался «совсем правильным». Идея состоит в том, чтобы поддерживать l и r так, чтобы искомое число x, если оно есть в массиве, лежало бы в *полуинтервале* индексов $[l, r)$, т.е. что искомый индекс i удовлетворяет условию $l \leqslant i < r$. (Аналогично можно требовать полуинтервал $(l, r]$). Это довольно хороший подход, код получается в точности таким же, как указано выше, только с другой инициализацией l — можно взять $l = 1$ (для $(l, r]$ можно взять $r = n$). Получается правый поиск (для $(l, r]$ — левый), и единственная проблема — невозможно отличить случай «x меньше всех элементов массива» и «x надо вставить между первым и вторым элементами» (для $(l, r]$ — два симметричных случая на другом конце массива).

Кстати, в бинарном поиске, даже написанном выше, есть ещё одна проблема (от неё даже не так давно страдали библиотечные функции двоичного поиска). При вычислении $(l+r)$ div 2 может произойти целочисленное переполнение — если изначально r было очень близко к максимальному числу, которое можно сохранить в вашем целочисленном типе. В реальных олимпиадных задачах это встречается весьма редко, и решается обычно просто — просто используйте бóльший тип данных. Но есть и вариант без использовать большего типа данных — можно просто написать $l + (r - l)$ div 2.

Часть IV. Деление пополам по ответу

Деление пополам по ответу — это важный способ применения деления пополам. Фактически, это применение приведённых выше кодов с функцией *good*, только в ситуации, когда значение функции *good* вычисляется сложным образом.

Рассмотрим классический пример. Есть N прямоугольных листов бумаги («дипломов») одинакового размера $w \times h$. Можно купить квадратную доску размера $L \times L$, повесить её на стену так, чтобы одна сторона была горизонтальной, а другая вертикальной, и на эту доску повесить эти дипломы так, чтобы они не перекрывались. При этом дипломы тоже надо повесить не поворачивая: сторона w должна быть горизонтальной, а сторона h — вертикальной. Какой минимальный размер доски (L) требуется, чтобы повесить все дипломы?

Понятно, что дипломы надо вешать на доску один вплотную к другому начиная с угла — так, что они будут образовывать решётку с шагом по горизонтали w, а по вертикали — h. Предположим, что мы выбрали некоторый размер доски L. Сколько максимум дипломов можно на неё повесить? Несложно видеть, что ответом будет $(L \text{ div } h) \cdot (L \text{ div } w)$.

Итак, мы научились решать задачу, в некотором смысле обратную данной: по размеру доски мы научились определять количество дипломов. Но нам надо решить обратную задачу: по количеству дипломов найти размер доски.

Понятно, что чем больше будет размер доски, тем больше будет дипломов, и наоборот. Поэтому мы можем применить бинарный поиск. А именно, мы знаем, что доска размера 0 нам точно не подходит. Доска некоторого большого размера (например, $Nw + Nh$) нам точно подходит. Объявим все размеры досок, которые нам подходят, «хорошими», а все размеры, которые нам не подходят — «плохими». Ясно, что все плохие числа идут до хороших. Поэтому делением пополам мы можем найти границу — два соседних числа, одно из которых плохое, а другое — хорошее. Далее очевидно, что это хорошее число и будет ответом:

```
function good(x:integer):boolean;
var nn:integer;
begin
nn:=(x div w)*(x div h); // столько дипломов можно повесить на доску размера x
result:=nn>=n; // если это >= чем общее число дипломов, то да
end;
```

```
...

l:=0;
r:=n*w+n*h;
while r-l>1 do begin
    m:=(r+l) div 2;
    if good(m) then
        r:=m
    else l:=m;
end;
writeln(r);
```

То есть мы просто задаём конкретную реализацию функции `good`, которая будет определять, может ли быть наше число ответом.

Это и называется делением пополам по ответу. Вы пишете функцию, которая проверяет, может ли быть некоторое число ответом. И вы доказываете, что все ответы идут после всех «не-ответов». Поэтому вы объявляете все ответы «хорошими», «не-ответы» — плохими, и запускаете деление пополам для поиска границы.

Аналогично можно писать деление пополам по ответу и в случаях, когда ответ является вещественным числом.

(Вообще, мы фактически вернулись к тому же, с чего начинали: если вы вспомните задачу о разрыве нитки, которую мы обсуждали вначале, то фактически там мы и реализовывали деление пополам по ответу.)