# Глава 1

# Бинарный (двоичный) поиск

Алгоритм бинарного поиска предназначен для быстрого поиска данных в упорядоченном массиве. В отличие от алгоритма линейного поиска, который в худшем случае сравнивает искомое число с каждым элементом массива, бинарный поиск даже в очень больших массивах обходится несколькими десятками сравнений.

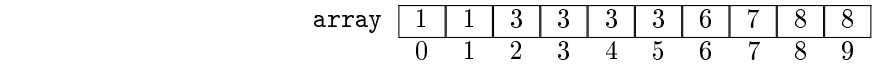
Идея этого алгоритма всем хорошо знакома по детской игре «Угадай число»: ведущий загадывает число от 1 до 100, участники называют свои предположения и получают ответы «больше» или «меньше». Если первым ходом назвать число 50, то после первого же ответа ведущего диапазон возможных чисел уменьшиться вдвое. Если ведущий скажет, что искомое число меньше 50, то на следующем шаге можно разделить пополам уже диапазон от 1 до 49, назвав число 25. На этой идее и основан рассматриваемый в этой главе алгоритм.

## 1.1 Обсуждение алгоритма

Прежде чем формулировать алгоритм и переходить к его исследованию, уточним задачу и введем обозначения.

*Дан массив* `array` *длины* `n`, *упорядоченный по неубыванию*[1] ($\mathtt{a}_0 \le \mathtt{a}_1 \le \mathtt{a}_2 \le \ldots \le \mathtt{a}_{n-1}$), *и число* `x`. *Требуется найти индекс* (*номер*) *элемента массива* (*назовем его* `index`), *равного* `x`. *Если таких элементов несколько, требуется найти индекс самого правого из них* (*такой бинарный поиск мы будем называть* правым). *Если таких элементов нет, выдать* $-1$.

[1] Когда говорят «упорядочен по возрастанию», подразумевают, что каждый следующий элемент строго больше предыдущего (последовательность возрастает). Если же хотят допустить и равные элементы, говорят, что последовательность *не убывает*, или, что массив отсортирован по *неубыванию*.




Например, для массива на рисунке и `x` = 3 алгоритм должен выдать в качестве ответа число 5 (элементы нумеруются, начиная с нуля).

## 1.2 Реализация

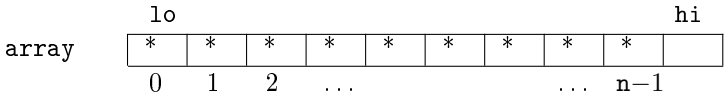
Несмотря на довольно короткий код, реализация бинпоиска содержит много тонких моментов и поначалу вызывает массу трудностей.

В каждый момент времени мы будем хранить границы интервала, в котором может находится искомый элемент, и постепенно будем сужать этот интервал. Мы будем предполагать, что выполнено следующее условие: `lo` $\leq$ `index` $<$ `hi`, где `lo` и `hi` — границы текущего интервала[2]. Обратите внимание, что левая граница может совпадать с `index`, а правая — нет. Для чего это сделано. будет ясно чуть позже. Другими словами, `index` $\in$ [`lo`, `hi`). Это условие мы будем называть *инвариантом цикла*: оно должно выполняться изначально и после каждого шага цикла.

Общая схема программы такова:

```
lo = ...
hi = ...
while (интервал еще можно уменьшить):
    mid = середина текущего интервала
    if x < a[mid]:
        hi = mid
    else:
        lo = mid
```

Теперь начнем ее уточнять. Начальные границы должны быть таковы, чтобы изначально выполнялся инвариант цикла. Для этого нужно положить `lo` = 0, `hi` = `n`. Обратите внимание, что последний элемент массива имеет номер `n` $-$ 1, но нам требуется, чтобы `hi` был изначально больше номера любого элемента массива, который может быть равен `x`. Поясним ограничения рисунком:




Обратите внимание, что изначально `hi` указывает «за пределы массива», но это не страшно, так как обращаться к элементу с индексом `hi` в программе мы не планируем. По-другому это можно представлять себе так: мы

[2]Мы не используем для обозначения границ интервала имена `left` и `right`, чтобы не возникало путаницы с левым и правым бинарным поиском.

дополнили массив еще одним элементом, равным бесконечности, т. е. числу, большему любого элемента массива.

Мы хотим уменьшать полуинтервал [lo, hi) до тех пор, пока в нем не останется ровно один элемент. Для этого hi должно быть на 1 больше lo (в этом случае в полуинтервал входит только элемент с индексом lo). Цикл будет выполняться, пока lo != hi − 1.

Перейдем к определению середины отрезка. Известная из математики формула (lo + hi)/2 не всегда даст нам целый результат: в случае, если lo и hi разной четности, его придется округлить, воспользовавшись целочисленным делением. Общая формула примет вид: (lo + hi)//2.

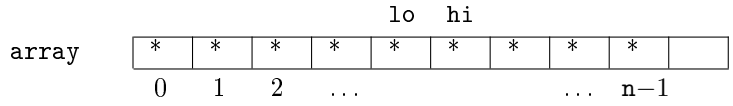
Таким образом, код бинарного поиска будет иметь следующий вид:

```
lo = 0
hi = n
while lo != hi - 1:
    mid = (lo + hi) // 2
    if x < a[mid]:
        hi = mid
    else:
        lo = mid
```

Остановимся подробнее на операторе if. Если искомый элемент меньше «среднего», то он лежит *строго левее* mid. Таким образом, присваивая hi = mid мы пользуемся тем, что в нашем инварианте x строго меньше hi. В противном случае x ≥ a[mid], и присваивая lo = mid мы оставляем x возможность находиться на месте с номером mid.

Обратите внимание, что мы не проверяем, равен ли x среднему элементу (это было бы пустой тратой времени, поскольку вероятность попасть в большом массиве на данный элемент очень мала), а лишь уменьшаем промежуток, на котором стоит искать x.

После окончания цикла мы получим такую картинку:




Осталось определить, есть ли в массиве элемент, равный x. Поскольку index ∈ [lo, hi) = [lo, lo+1), если x присутствует в массиве, он равен a[lo]. Допишем программу:

```
if a[lo] == x:
    return lo
else:
    return -1
```

## 1.3 Анализ кода

Программа написана, но осталась еще масса вопросов, которые внимательный читатель уже должен был себе задать.

*Почему программа не зациклится?* Иными словами, почему цикл `while` рано или поздно закончит свою работу. Разделим этот вопрос на две части. Во-первых, давайте поймем, почему мы не проскочим нужный момент, то есть почему hi не станет меньше либо равным `lo`. Пусть на некотором шаге hi еще больше, чем `lo` + 1. Докажем, что на следующем шаге `hi` не может стать меньше `lo` + 1. Для этого достаточно заметить, что

1) `mid` лежит левее `hi` (ведь оно не больше чем среднее арифметическое `hi` и `lo`);

2) `mid` лежит правее `lo` (если `lo` и `hi` одной четности, то $\texttt{mid} = (\texttt{lo} + \texttt{hi})/2 > (\texttt{lo} + \texttt{lo})/2 = \texttt{lo}$; если же они имеют разную четность, то $\texttt{mid} = (\texttt{lo} + \texttt{hi} - 1)/2 > \texttt{lo} + (\texttt{lo} + 1) - 1/2 = \texttt{lo}$, поскольку в данном случае `hi` больше, чем `lo` + 1.

Таким образом, и после присваивания $\texttt{lo} = \texttt{mid}$, и после присваивания $\texttt{hi} = \texttt{mid}$ значение `lo` будет меньше значения `hi`.

Во-вторых, нужно понять, почему промежуток рано или поздно уменьшится до длины 1. Для этого достаточно заметить, что поскольку `mid` лежит строго между `lo` и `hi`, то на каждом шаге промежуток уменьшается хотя бы на единицу.

Итак, мы доказали, что промежуток рано или поздно станет сколько угодно малым, и при этом «не проскочит» длину 1. В этот момент цикл и закончит свою работу.

*Если элементов, равных* `x`*, несколько, то почему наш код найдет именно самый правый?* Мы уже знаем, что если в массиве есть элемент, равный `x`, то наш алгоритм уменьшит промежуток до одного элемента, и этот элемент будет равен `x`. Рассмотрим самый правый элемент, равный `x`. Докажем, что он всегда будет оставаться в нашем промежутке. Действительно, если mid окажется правее него, то мы передвинем `hi` в `mid`, если `mid` окажется левее него, то мы передвинем `lo` в `mid`, а если `mid` укажет на наш элемент, то мы передвинем `lo` в `mid`, и тем самым элемент не покинет полуинтервала $[\texttt{lo}, \texttt{hi})$.

*Какова сложность приведенного алгоритма?* Поскольку на каждом шаге цикла мы уменьшаем рассматриваемый интервал примерно вдвое, а его исходная длина равна `n`, сложность представленного алгоритма равна $\log_2 \texttt{n}$. Чтобы понять, много это или мало, сравним его сложность со сложностью линейного поиска:

| `n` | 1 000 | 1 000 000 | 1 000 000 000 |
|---|---|---|---|
| линейный поиск | 1 000 операций | 1 000 000 операций | 1 000 000 000 операций |
| бинарный поиск | 10 операций | 20 операций | 30 операций |

## 1.4 FAQ

1. *Можно ли после цикла использовать* `mid` *вместо* `lo`?

Нет, нельзя. Эту ошибку допускают очень часто, поэтому остановимся на ней подробней. Во-первых, если в последней итерации цикла условие в операторе `if` оказалось истинным, то mid окажется равным `hi`, а не `lo`. Во-вторых, если массив состоит из одного числа, то цикл не выполнится ни разу, и значение `mid` будет неопределено.

2. *Если числа* `x` *нет в массиве, то что означает условие* `index` $\in$ $[$`lo`, `hi`$)$?

Можно было бы ответить формально: этот инвариант цикла обозначает, что если число `x` есть в массиве, то оно обязательно находится между `a`[`lo`] и `a`[`hi`] (возможно, совпадая с `a`[`lo`]). Но давайте задумаемся: а какой смысл в действительности имеют `lo` и `hi`, если искомого числа нет в массиве. Заметим, что независимо от присутствия x мы присваиваем в цикле `lo` такое значение, что `a`[`lo`] $\leq$ `x`, аналогично `a`[`hi`] всегда больше `x` (если считать, что в `a`[`n`] записана «бесконечность»). Таким образом, может показаться. что после выполнения цикла в случае, если `x` не найден, это значение заключено между `a`[`lo`] и `a`[`hi`] = `a`[`lo` + 1]. К сожалению, это не всегда так: в только что приведенном доказательстве по индукции мы пропустили базу, которая в данном случае как раз и не выполняется. Действительно, если `x` < `a`[0], то `x` всегда будет меньше `a`[`lo`]. В следующем разделе мы подправим наш алгоритм так, чтобы он в этой ситуации работал как ожидается.

3. *Так почему же все-таки полуинтервал*?

Давайте посмотрим, что получится, если мы заменим полуинтервал на отрезок. Пусть `high` = `hi` − 1. Тогда `index` $\in$ $[$`lo`, `high`$]$, и программа примет следующий вид:

```
lo = 0
high = n - 1
while lo != high:
    mid = (lo + high - 1) // 2
    if x < a[mid]:
        high = mid + 1
    else:
        lo = mid
```

Как видим, код получился менее симметричным и изящным, в нем стало легче допустить ошибку, хотя по смыслу он полностью эквивалентен исходной версии.

## 1.5 Левый бинпоиск

Давайте вспомним, почему написанный нами бинпоиск ищет самый правый `x`. Если на некотором шаге алгоритма оказывается, что `a`[`mid`] = `x`, все

элементы левее x отбрасываются. Поэтому гарантированно в нашем промежутке остается только самый правый x.

Давайте попробуем по аналогии написать «левый» бинпоиск. Начнем с ключевого места — с условного оператора. Теперь нам нужно в случае равенства отбрасывать всё, что правее:

```
if x <= a[mid]:
    hi = mid
else:
    lo = mid
```

Соответственно изменится и инвариант цикла: теперь `index` $\in$ (`lo`, `hi`], а значит, нужно изменить и начальные значения: `lo` $= -1$, `hi` $=$ `n` $- 1$.

## 1.6 Вставка в упорядоченный массив

Изменим исходную задачу: теперь нам требуется найти место в массиве `a`, на которое можно вставить число `x` так, чтобы массив остался упорядоченным. При этом элемент, на место которого вставляется `x`, и все элементы следом за ним сдвигаются на один вправо.

Например, если требуется вставить число 5 в массив $[1, 2, 2, 4, 7, 9]$, то результатом будет массив $[1, 2, 2, 4, \mathbf{5}, 7, 9]$. Возникает вопрос: куда вставлять `x`, если равный ему элемент уже есть в массиве (и, возможно, не один). Давайте считать, что нас интересует самое правое место, куда мы можем поставить `x`, то есть позиция, следующая за самым правым из элементов, равных `x`. Например в массив [1, 2, 5, 5, 5, 7, 9] число 5 нужно вставить заместо числа 7 (подвинув 7 и 9 на 1 вправо).

Заметим, что рассмотренный нами изначально правый бинпоиск почти удовлетворяет требованиям. Если `x` присутствует в массиве, то ответом будет `lo` $+ 1 =$ `hi`. Если же `x` в массиве нет, то `a`[`lo`] $<$ `x` $<$ `a`[`hi`], то есть `hi` также является ответом. Но мы помним, что последнее неверно, если `x` $<$ `a`[0]: в этом случае инвариант цикла не выполняется изначально. Но это легко исправить, положив изначально `lo` $= -1$ (и оставив `hi` $=$ `n`). Можно проверить, что все остальные наши рассуждения относительно алгоритма остаются верными. Теперь если `x` окажется меньше всех элементов массива, то после цикла `lo` будет равно $-1$, а `hi` — нулю, и ответ задачи будет снова лежать в `hi`.

Обратите внимание: если `x` больше, чем каждый элемент массива, то ответом будет `hi` $=$ `n`, то есть номер пока не существующего элемента массива.

Возникает вопрос: нельзя ли в исходном алгоритме бинпоиска положить `lo` $= -1$ и получить универсальный код для решения обеих задач? К сожалению, нет. Дело в том, что после окончания цикла мы сравниваем `a`[`lo`] с `x`, чтобы определить, есть ли `x` в массиве. Но при начальном значении `lo` $= -1$ сравнение с `a`[$-1$] вызовет ошибку.

Приведем целиком получившийся код:

```
lo = -1
hi = n
while lo != hi - 1:
    mid = (lo + hi) // 2
    if x < a[mid]:
        hi = mid
    else:
        lo = mid

return hi
```

## 1.7 Количество элементов массива, равных данному числу

Научимся теперь находить не позицию одного элемента, равного `x`, а количество таких элементов в массиве. Казалось бы, если мы умеем находить самый правый такой элемент, можно идти от него влево и считать, сколько элементов равны ему. Но это решение в худшем случае имеет сложность порядка `n` (если все или почти все элементы массива равны `x`).

Научимся решать эту задачу алгоритмом сложности порядка $\log n$.

**Первый способ**. Найдем с помощью левого и правого бинпоисков позицию самого левого и самого правого `x` в массиве (назовем их `left_index` и `right_index`). Тогда ответ в задаче можно вычислить по формуле $\texttt{right\_index} - \texttt{left\_index} + 1$ (отдельно нужно проверить, что `x` есть в массиве). Но этот способ потребует написать два разных кода для левого и правого бинпоиск.

**Второй способ.** Предположим, что элементы массива — целые числа. Будем использовать только правый бинпоиск. Найдем с его помощью сначала число `x` (обознчим его позицию за `index`), а затем число $\texttt{x} - 1$ (его позицию сохраним в переменной `index_1`. Заметим, что `index_1` — самое правое число, меньшее `x`, следовательно, все элементы, равные `x` занимают места между `index_1` (не включительно) и `index` (включительно). Их количество равно $\texttt{index} - \texttt{index\_1}$ (если в массиве вообще есть элементы, равные `x`).

## 1.8 Применение

Возникает естественный вопрос: откуда на практике возьмется отсортированный массив? Реальные данные крайне редко бывают заранее упорядоченными. А если предварительно воспользоваться сортировкой, то в общем случае сложность алгоритма возрастет до $\texttt{n} \log \texttt{n}$ (минимально возможная сложность сортировки для произвольных данных), что уже больше, чем сложность линейного поиска.

Всё дело в том, что обычно запрос на поиск в массиве данных требуется выполнять многократно. Например, если нам нужно выполнить $\mathtt{k}$ запросов, то суммарная сложность при использовании линейного поиска будет равна $\mathtt{nk}$, а при использовании сортировки + бинарного поиска — $\mathtt{n}\log\mathtt{n} + \mathtt{k}\log\mathtt{n}$, так что при даже не очень больших $\mathtt{k}$ уже выгоднее использовать бинарный поиск.

Стоит отметить, что есть и более эффективные структуры для хранения данных с целью поиска, чем упорядоченный массив. Например, использование хеш-таблиц позволяет выполнять запросы на поиск элемента в среднем за константу операций. но бинарный поиск можно использовать не только для поиска в массиве, но и, например, для «быстрого подбора» ответа в задаче. Об этом будет расказано ниже.