ГЛАВА 2

О важности алгоритмов

Мы уже рассмотрели две структуры данных и увидели, как выбор подходящей может повлиять на производительность кода. Даже такие похожие на первый взгляд структуры данных, как массив и множество, могут обладать разной эффективностью.

В этой главе вы узнаете, что помимо структуры данных на эффективность кода может повлиять и выбор *алгоритма*.

Вам может показаться, что здесь скрывается какая-то сложная концепция, но это вовсе не так. Алгоритм — это просто *набор инструкций для выполнения конкретной задачи*.

Даже простой процесс вроде приготовления завтрака — это уже алгоритм, поскольку он подразумевает выполнение определенного набора шагов для решения поставленной задачи. Алгоритм приготовления завтрака (по крайней мере, в моем случае) состоит из четырех шагов.

1. Возьмите тарелку.
2. Насыпьте в тарелку хлопья.
3. Налейте в тарелку молоко.
4. Опустите в тарелку ложку.

Если мы будем четко следовать этой инструкции, то сможем насладиться завтраком.

В случае с вычислениями под алгоритмом понимается набор инструкций, данных компьютеру для выполнения конкретной задачи. Так, когда мы пишем какой-либо код, мы создаем алгоритмы — наборы инструкций, которые должен выполнять компьютер.

Мы можем описывать алгоритмы простым языком, детально излагая инструкции, которые планируем дать компьютеру. Для описания принципа работы разных алгоритмов в книге я буду использовать как обычный язык, так и код.

Иногда для выполнения одной задачи можно использовать два разных алгоритма. Мы уже сталкивались с этим в начале первой главы, когда знакомились с двумя разными способами вывода четных чисел на экран. В том случае один алгоритм предполагал выполнение в два раза большего числа шагов, чем другой.

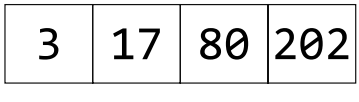
В этой главе мы познакомимся еще с двумя алгоритмами, решающими одну задачу. Только в этом случае один алгоритм будет *на несколько порядков* быстрее другого.

Но перед этим нам нужно рассмотреть новую структуру данных.

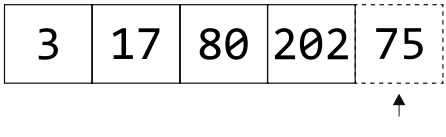
Упорядоченные массивы

Упорядоченный массив почти идентичен «классическому» из прошлой главы. Единственное отличие в том, что упорядоченные массивы требуют, чтобы значения всегда располагались — как вы уже догадались — *по порядку*. То есть новое значение нужно вставить так, чтобы в итоге все значения в массиве оставались отсортированными.

Для примера возьмем массив `[3, 17, 80, 202]`:

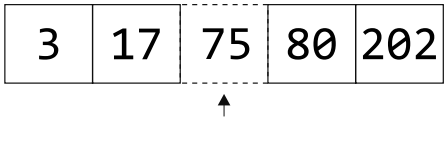


Допустим, мы хотим вставить в него число 75. Если бы этот массив был классическим, мы могли бы добавить это значение в конец:



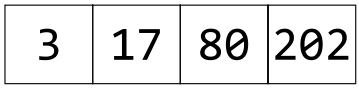
Как мы видели в главе 1, компьютер может сделать это за один шаг.

Но в случае с *упорядоченным массивом* нам придется вставить число 75 в определенную ячейку, чтобы значения располагались в порядке возрастания:

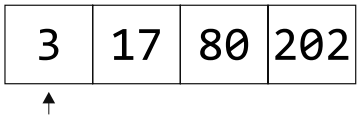


Но легче сказать, чем сделать. Компьютер не может просто вставить значение 75 в нужное место за один шаг, потому что сначала он должен *найти* подходящую позицию, а затем сдвинуть другие значения, чтобы освободить место. Рассмотрим этот процесс подробнее.

Начнем с того же упорядоченного массива.



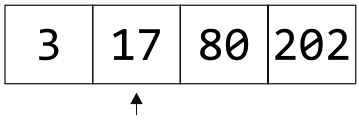
Шаг 1: проверяем значение с индексом 0, чтобы определить, где должно располагаться значение, которое мы хотим вставить (75): слева или справа от него:



Поскольку 75 больше 3, мы знаем, что новое значение должно быть вставлено где-то справа от него. Но мы еще не знаем, в какую именно ячейку, поэтому нам нужно проверить следующее значение.

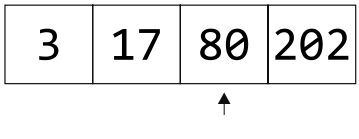
Мы назовем этот шаг *сравнением*, ведь мы сравниваем вставляемое значение с числом, присутствующим в упорядоченном массиве.

Шаг 2: проверяем значение в следующей ячейке:



Значение 75 больше 17, поэтому нам нужно двигаться дальше.

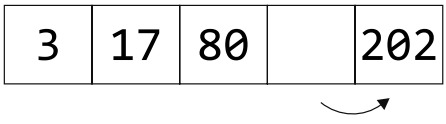
Шаг 3: проверяем значение в следующей ячейке:



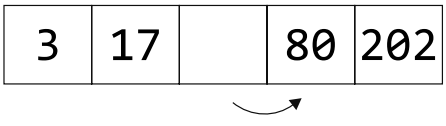
Значение 80 больше 75, поэтому мы приходим к выводу, что значение 75 нужно вставить слева от 80, чтобы сохранить порядок элементов массива. Чтобы

освободить место для вставки значения 75, нам нужно сдвинуть остальные данные.

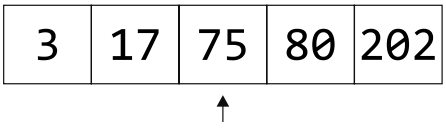
Шаг 4: сдвигаем последнее значение вправо:



Шаг 5: сдвигаем предпоследнее значение вправо:



Шаг 6: помещаем значение 75 в нужную ячейку:



Как видите, перед помещением значения в упорядоченный массив всегда нужно проводить поиск, чтобы определить правильное место вставки. Это одно из различий в производительности между классическим массивом и упорядоченным.

В этом примере мы видим, что изначально в массиве было четыре элемента, а на вставку ушло шесть шагов. Так, в случае с упорядоченным массивом из N элементов количество шагов для вставки значения равно $N + 2$.

Интересно, что это количество шагов остается одинаковым вне зависимости от того, в какую именно позицию массива помещается новое значение. При вставке значения в начало мы выполняем меньше сравнений и больше сдвигов, а ближе к концу — больше сравнений, но меньше сдвигов. Меньше всего шагов нужно для вставки нового значения в самый конец, так как эта операция не подразумевает никаких сдвигов. В этом случае мы выполняем N шагов для сравнения нового значения со всеми существующими и один для самой вставки, что в общей сложности дает $N + 1$ шагов.

Хотя упорядоченный массив проигрывает классическому в эффективности, если речь идет о вставке, он многократно превосходит его, когда дело доходит до поиска.

Поиск в упорядоченном массиве

В прошлой главе я описал процесс поиска определенного значения в классическом массиве: мы проверяем каждую ячейку по одной — слева направо, — пока не найдем искомое значение. Там же я отметил, что этот процесс называется линейным поиском. Посмотрим, чем отличается линейный поиск при работе с классическим и упорядоченным массивами.

Допустим, у нас есть обычный массив `[17, 3, 75, 202, 80]`. Если бы мы искали значение 22, которого здесь нет, нам пришлось бы проверить каждый элемент, потому что оно может быть где угодно. Учитывая это, мы можем остановить поиск до достижения конца массива, только если нам удастся найти нужное значение до этого момента.

Но в случае с упорядоченным массивом мы можем остановить поиск раньше, даже если искомого значения в массиве нет. Допустим, мы ищем число 22 в упорядоченном массиве `[3, 17, 75, 80, 202]`. Мы можем остановить поиск, как только дойдем до 75, поскольку 22 никак не может быть справа от него.

Вот так выглядит реализация линейного поиска в упорядоченном массиве на языке Ruby:

```
def linear_search(array, search_value)

  # Перебираем все элементы массива:
  array.each_with_index do |element, index|

    # Если находим искомое значение, возвращаем его индекс:
    if element == search_value
      return index

    # Если мы обнаружили элемент, значение которого превышает искомое,
    # можем выйти из цикла раньше:
    elsif element > search_value
      break
    end
  end

  # Если искомое значение в массиве не обнаружено, возвращаем nil:
  return nil

end
```

Этот метод принимает два аргумента: `array` — упорядоченный массив, где мы осуществляем поиск, и `search_value` — искомое значение.

Вот как можно использовать эту функцию для поиска значения `22` в массиве из нашего примера:

```
p linear_search([3, 17, 75, 80, 202], 22)
```

Как видите, метод `linear_search` перебирает все элементы массива в поисках значения `search_value`. Процесс останавливается, как только значение обрабатываемого элемента превышает `search_value`, поскольку мы знаем, что после этого нужного значения в массиве точно не будет.

Учитывая все это, мы можем прийти к выводу, что в определенных ситуациях линейный поиск в упорядоченном массиве может занять меньше шагов, чем в классическом. При этом, если искомое значение превышает значения остальных элементов массива или вообще отсутствует, нам все равно придется проверить каждую ячейку.

Итак, на первый взгляд, стандартные и упорядоченные массивы не сильно различаются в плане эффективности, по крайней мере, в худших сценариях. Для обоих типов, содержащих N элементов, на линейный поиск может уйти до N шагов.

Но далее мы познакомимся с мощным алгоритмом, многократно превосходящим линейный поиск.

До сих пор мы думали, что единственный способ нахождения значения в упорядоченном массиве — линейный поиск. Но это всего лишь *один из возможных алгоритмов* поиска значения, а не *единственный*.

Большое преимущество упорядоченного массива по сравнению с классическим в том, что первый позволяет использовать алгоритм *бинарного (двоичного) поиска,* который работает *гораздо* быстрее линейного.

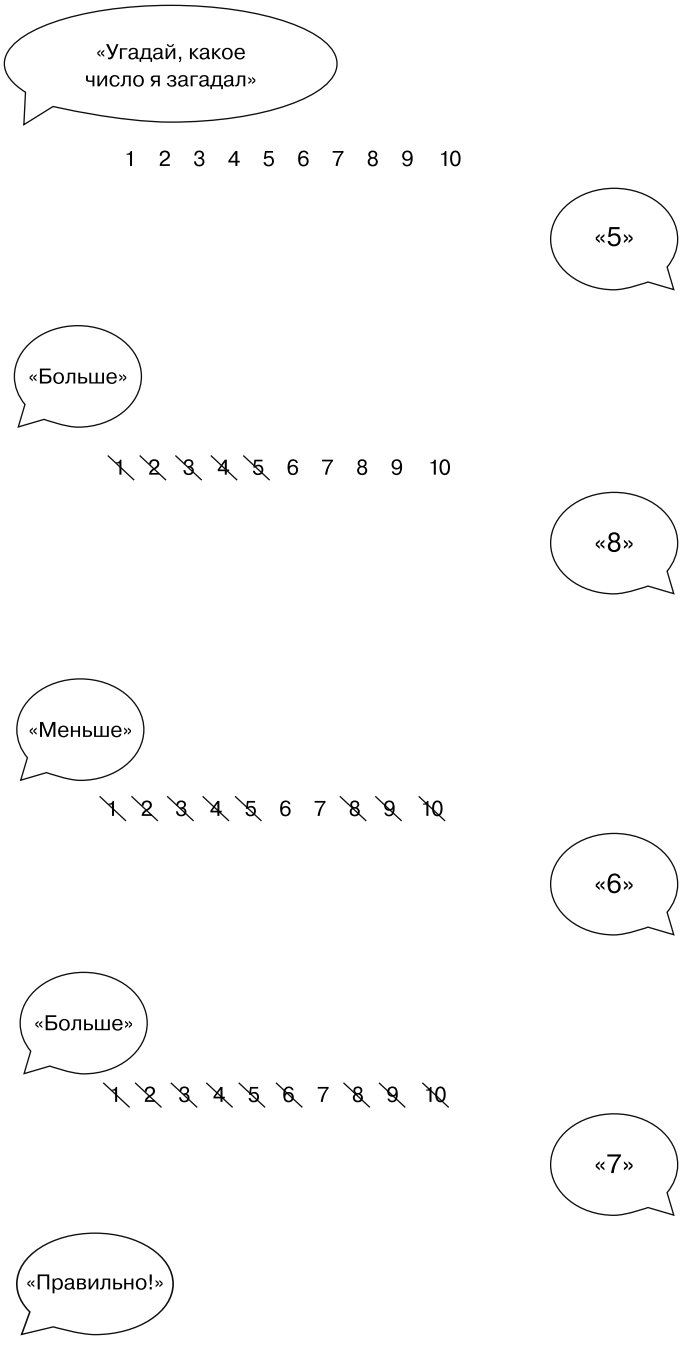
Бинарный поиск

Вы наверняка в детстве играли в такую игру: один загадывает число от 1 до 100, а другой пытается его угадать. Каждый раз, когда угадывающий называет число, ему сообщается, больше оно, чем загаданное, или нет.

Принцип этой игры понятен. Скорее всего, вы бы не начали с единицы, а вместо этого первым делом назвали бы число, которое находится ровно посередине. Почему? Потому что, выбрав 50, вне зависимости от полученной подсказки вы сразу исключаете половину возможных чисел!

Если вы называете 50, а вам говорят, что загаданное число больше, вы выбираете 75, исключая половину *оставшихся* чисел. Если затем вам скажут, что загаданное число меньше 75, вы выберете 62 или 63 и будете продолжать выбирать среднее значение, каждый раз исключая половину оставшихся чисел.

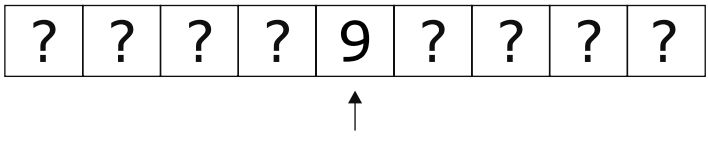
Представьте процесс угадывания числа от 1 до 10. Именно так выглядит принцип работы двоичного (или бинарного) поиска.


«Угадай, какое число я загадал»
1 2 3 4 5 6 7 8 9 10
«5»
«Больше»
1 2 3 4 5 6 7 8 9 10
«8»
«Меньше»
1 2 3 4 5 6 7 8 9 10
«6»
«Больше»
1 2 3 4 5 6 7 8 9 10
«7»
«Правильно!»

Посмотрим, как осуществляется бинарный поиск в упорядоченном массиве. Итак, у нас есть упорядоченный массив из, скажем, девяти элементов. Компьютер не знает, какое значение в каждой из ячеек, поэтому массив можно изобразить так:

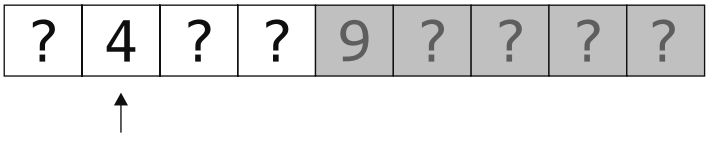
Допустим, мы хотим найти значение 7. Вот как будет работать алгоритм двоичного поиска.

Шаг 1: начинаем поиск со средней ячейки. Мы можем обратиться к ней сразу, поскольку для вычисления ее индекса нам достаточно разделить длину массива на 2. Итак, проверяем значение в этой ячейке:



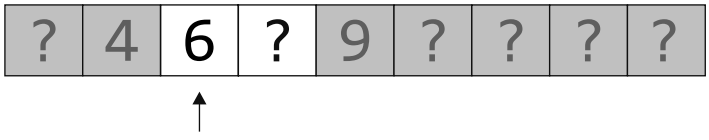
Обнаруженное значение равно 9, поэтому мы можем предположить, что число 7 находится где-то слева от него. Так мы успешно исключили половину ячеек массива — все ячейки справа от той, что содержит 9 (и ее саму):

Шаг 2: проверяем среднюю из оставшихся ячеек. Таких здесь две, и мы произвольно выбираем левую:

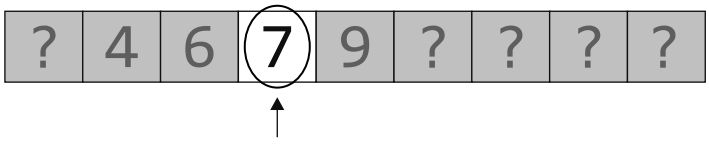


В ней мы видим число 4, поэтому искомое значение 7 должно находиться где-то справа от нее. Мы исключаем ячейку с числом 4 и ту, что слева от нее:

Шаг 3: у нас осталось две ячейки для проверки, и мы произвольно выбираем левую:



Шаг 4: проверяем последнюю ячейку (если это не 7, значит, такого значения в упорядоченном массиве вообще нет).



Мы нашли число 7 за четыре шага. Хотя в этом примере столько же шагов ушло бы и на линейный поиск, чуть позже вы сможете оценить истинную мощь двоичного.

Обратите внимание, что алгоритм двоичного поиска можно применить только к упорядоченному массиву. В классическом массиве значения могут располагаться в любом порядке, так что мы никогда не узнаем, где искать нужное значение — слева или справа от выбранной наугад ячейки. В этом и есть одно из преимуществ упорядоченных массивов: они позволяют осуществить бинарный поиск.

Программная реализация

Вот как выглядит реализация бинарного поиска на языке Ruby:

```
def binary_search(array, search_value)

  # Сначала определяем нижнюю и верхнюю границы диапазона, в котором
  # может находиться искомое значение. Изначально нижняя граница - это
  # первое значение массива, а верхняя - последнее:

  lower_bound = 0
  upper_bound = array.length - 1

  # Запускаем цикл, где последовательно проверяем средние значения диапазона:

  while lower_bound <= upper_bound do

    # Находим среднее значение между верхней и нижней границами:
```

```
    # (нам не нужно беспокоиться о том, что результат будет дробным,
    # так как в Ruby результат деления целых чисел всегда
    # округляется в меньшую сторону до ближайшего целого числа)

    midpoint = (upper_bound + lower_bound) / 2

    # Проверяем значение в средней ячейке:

    value_at_midpoint = array[midpoint]

    # Если это значение совпадает с искомым, поиск завершается.
    # Если нет, меняем нижнюю или верхнюю границу в зависимости от того,
    # превышает ли искомое значение то, которое мы обнаружили, или нет:

    if search_value == value_at_midpoint
      return midpoint
    elsif search_value < value_at_midpoint
      upper_bound = midpoint - 1
    elsif search_value > value_at_midpoint
      lower_bound = midpoint + 1
    end
  end

  # Если мы сузили границы до такой степени, что между ними не осталось
  # ячеек, значит, искомого значения в этом массиве нет:

  return nil
end
```

Разберем этот код. Как и метод `linear_search`, функция `binary_search` принимает в качестве аргументов массив `array` и искомое значение `search_value`.

Вот пример вызова этого метода:

```
p binary_search([3, 17, 75, 80, 202], 22)
```

Сначала он задает диапазон индексов ячеек, в которых может быть обнаружено значение `search_value`:

```
lower_bound = 0
upper_bound = array.length - 1
```

Поскольку в самом начале искомое значение `search_value` может быть обнаружено в любом месте массива, для нижней границы `lower_bound` мы задаем первый индекс, а для верхней `upper_bound` — последний.

Сам процесс поиска происходит внутри цикла `while`:

```
while lower_bound <= upper_bound do
```

Этот цикл выполняется, пока у нас остается диапазон ячеек, в которых может находиться искомое значение `search_value`. Как мы увидим далее, в процессе

поиска наш алгоритм будет последовательно сужать этот диапазон, пока между нижней и верхней границами не останется ячеек (`lower_bound <= upper_bound`). Если за это время искомое значение не будет найдено, мы придем к выводу, что `search_value` в массиве нет.

В цикле код проверяет значение `midpoint`, которое находится в середине диапазона:

```
midpoint = (upper_bound + lower_bound) / 2
value_at_midpoint = array[midpoint]
```

`value_at_midpoint` — это элемент в середине диапазона.

Если `value_at_midpoint` совпадает с искомым значением `search_value`, мы добились цели и можем возвратить индекс соответствующего элемента:

```
if search_value == value_at_midpoint
  return midpoint
```

Если `search_value` меньше, чем `value_at_midpoint`, значит, нужное значение следует искать левее. Так мы можем сузить диапазон поиска, выбрав в качестве `upper_bound` индекс слева от средней точки `midpoint`, поскольку `search_value` не может находиться справа от нее:

```
elsif search_value < value_at_midpoint
  upper_bound = midpoint - 1
```

И наоборот, если `search_value` больше, чем `value_at_midpoint`, значит, нужное значение следует искать справа от средней точки `midpoint`, поэтому мы увеличиваем значение `lower_bound`:

```
elsif search_value > value_at_midpoint
  lower_bound = midpoint + 1
```

Когда диапазон сужается до 0 элементов, мы возвращаем значение `nil`. В этом случае мы точно знаем, что `search_value` в массиве нет.

Сравнение алгоритмов бинарного и линейного поиска

При работе с небольшими упорядоченными массивами алгоритм двоичного поиска не сильно превосходит в эффективности алгоритм линейного поиска. Но посмотрим, что происходит с большими массивами.

Вот максимальное число шагов для выполнения каждого типа поиска в массиве со 100 значениями:

- линейный поиск — 100 шагов;
- двоичный поиск — 7 шагов.

При линейном поиске, если искомое значение находится в последней ячейке или превышает значение в ней, нам придется проверить каждый элемент. В случае с массивом в 100 значений на это уйдет 100 шагов.

Но при использовании двоичного поиска каждое предположение, которое мы делаем, исключает половину ячеек, которые нам иначе пришлось бы проверять. Первое предположение позволяет избавиться сразу от 50 ячеек.

Если мы посмотрим на это с другой стороны, то увидим закономерность.

В случае с массивом из трех элементов двоичный поиск потребовал бы максимум двух шагов.

Если мы удвоим количество ячеек в массиве (и для простоты добавим еще одну, чтобы их общее количество было нечетным), у нас получится семь ячеек. Двоичный поиск в таком массиве потребует максимум трех шагов.

Если мы снова удвоим число ячеек (и добавим еще одну), так чтобы упорядоченный массив содержал 15 элементов, то максимальное количество шагов для выполнения двоичного поиска будет равно четырем.

Закономерность следующая: каждый раз, когда мы удваиваем размер упорядоченного массива, количество шагов для выполнения двоичного поиска увеличивается на единицу. Это логично, так как каждый шаг поиска исключает половину элементов.

Все это говорит о необычайной эффективности такого алгоритма: каждый раз, когда мы удваиваем объем данных, в алгоритм двоичного поиска добавляется *всего один шаг*.

Сравним его с линейным поиском. Если бы в массиве было три элемента, нам потребовалось бы выполнить до трех шагов. В случае с массивом из семи элементов — до семи, а из 100 — до 100. Итак, при линейном поиске *число шагов равно количеству элементов*. Поэтому каждый раз, когда мы удваиваем размер массива, мы *удваиваем* и число шагов, которые нужно выполнить. В случае же с двоичным поиском удвоение размера массива добавляет *лишь один шаг*.