

Информатика

Баранов Максим Александрович

Осень 2024

"Сегодня мы поговорим об алгоритмах сортировки — одной из ключевых тем в информатике. Сортировка используется практически в каждом аспекте компьютерных программ, от баз данных до веб-приложений. Мы рассмотрим различные методы сортировки, их преимущества, недостатки и асимптотическую сложность."

Алгоритмы (сортировки)

Алгоритм (algorithm) — это формально описанная вычислительная процедура, получающая **исходные данные** (input), называемые также входом алгоритма или его аргументом, и выдающая **результат** вычислений на выход (output).

~последовательность действий

Абстрактный тип данных — это математическая модель плюс различные операторы, определенные в рамках этой модели.

~интерфейс работы с данными

Для представления АТД используются **структуры данных**, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

~конкретная реализация АТД

На этих слайдах мы приступим к обсуждению различных алгоритмов сортировки. Сортировка — это одна из базовых операций, которую нужно уметь выполнять в программировании. Она часто является частью более сложных алгоритмов и применяется в разных задачах, от упорядочивания данных до поиска и анализа. Сегодня мы рассмотрим такие алгоритмы как сортировка выбором, пузырьком, вставками, слиянием и быстрая сортировка.

Алгоритмы (сортировки)

Хотя термины *тип данных* (или просто *тип*), *структура данных* и *абстрактный тип данных* звучат похоже, но имеют они различный смысл. В языках программирования *тип данных* переменной обозначает множество значений, которые может принимать эта переменная.

На этих слайдах мы приступим к обсуждению различных алгоритмов сортировки. Сортировка — это одна из базовых операций, которую нужно уметь выполнять в программировании. Она часто является частью более сложных алгоритмов и применяется в разных задачах, от упорядочивания данных до поиска и анализа. Сегодня мы рассмотрим такие алгоритмы как сортировка выбором, пузырьком, вставками, слиянием и быстрая сортировка.

Что читать?



Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман

Структуры ДАННЫХ и АЛГОРИТМЫ

Классическое
издание



Для глубокого понимания алгоритмов сортировки и других тем по информатике я рекомендую несколько книг и материалов. Эти ресурсы помогут вам углубить знания и понять, как применять алгоритмы на практике. Регулярное чтение профильной литературы — залог вашего успеха в изучении предмета.

Что читать?



Для глубокого понимания алгоритмов сортировки и других тем по информатике я рекомендую несколько книг и материалов. Эти ресурсы помогут вам углубить знания и понять, как применять алгоритмы на практике. Регулярное чтение профильной литературы — залог вашего успеха в изучении предмета.

Нужная математика

Разделить

$$\log_a x = \log_b x \log_a b$$

$$1 + 2 + 3 + \dots + n = (n + 1) \frac{n}{2}$$

$$q^0 + q^1 + q^2 + \dots + q^{n-1} = \frac{1 - q^n}{1 - q} \quad (q \neq 1)$$

Умножить

Алгоритмы невозможно понять без знания базовой математики, которая описывает их работу. Мы обсудим концепции, такие как сложность алгоритмов, использование Big O нотации и амортизационная стоимость. Например, когда мы получаем элемент по индексу в массиве, сложность операции составляет $O(1)$, так как это мгновенная операция. Однако, когда мы перебираем все элементы коллекции, сложность уже возрастает до $O(n)$.

Нужная математика

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0$$



Меньше

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = c \quad (c \neq 0)$$



Однаково

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \infty$$



Больше

Нужная математика

$$n! = n(n - 1)!$$

$$F_i = F_{i-1} + F_{i-2}, \quad F_0 = 0, \quad F_1 = 1$$

$$T(n) = T(n/2) + 1, \quad T(1) = 1$$



Рекуррентные формулы

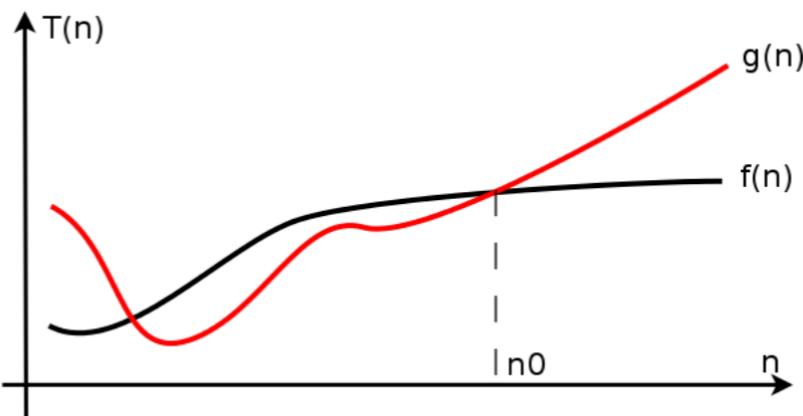
Рекуррентная формула (англ. recurrence relation) — формула вида , выражающая каждый следующий член последовательности через предыдущих членов и номер члена последовательности , вместе с заданными первыми р членами, где n — порядок рекуррентного соотношения.

Нужная математика

0-большое →

$$f(n) = O(g(n))$$

$$|f(n)| \leq C|g(n)|, \quad n \geq n_0$$



Big O нотация нужна для описания сложности алгоритмов. Для этого используется понятие времени. Тема для многих пугающая, программисты избегающие разговоров о «времени порядка N» обычное дело.

Если вы способны оценить код в терминах Big O, скорее всего вас считают «умным парнем». И скорее всего вы пройдете ваше следующее собеседование. Вас не остановит вопрос можно ли уменьшить сложность какого-нибудь куска кода до $n \log n$ против n^2 .

«О» большое и «о» малое — математические обозначения для сравнения асимптотического поведения (асимптотики) функций. Используются в различных разделах математики, но активнее всего — в математическом анализе, теории чисел и комбинаторике, а также в информатике и теории алгоритмов. Под асимптотикой понимается характер изменения функции при стремлении её аргумента к определённой точке.

Нужная математика

$$f(n) = O(g(n))$$

Свойства



$$\beta g(n) = O(g(n))$$

$$f(n) + g(n) = O(\max[f(n), g(n)])$$

Нужная математика

- `const nums = [1,2,3,4,5];`
 - `const firstNumber = nums[0];`
-
- `const nums = [1,2,3,4,5];`
 - `let sum = 0;`
 - `for(let num of nums){`
 - `sum += num;`
 - `}`

1. 1.

Возьмем массив из 5 чисел:

Допустим надо получить первый элемент. Используем для этого индекс:

Тут приведен алгоритм $O(1)$

На сколько это сложный алгоритм? Можно сказать: «совсем не сложный — просто берем первый элемент массива». Это верно, но корректнее описывать сложность через количество операций, выполняемых для достижения результата, в зависимости от ввода (*операций на ввод*).

Другими словами: насколько возрастет кол-во операций при увеличении кол-ва входных параметров.

В нашем примере входных параметров 5, потому что в массиве 5 элементов. Для получения результата нужно выполнить одну операцию (взять элемент по индексу). Сколько операций потребуется если элементов массива будет 100? Или 1000? Или 100 000? Все равно нужна только одна операция.

Т.е.: «одна операция для всех возможных входных данных» — $O(1)$.

$O(1)$ можно прочитать как «сложность порядка 1» (order 1), или «алгоритм выполняется за постоянное/константное время» (constant time).

Вы уже догадались что $O(1)$ алгоритмы самые эффективные.

СУММА

2.

Опять зададимся вопросом: сколько операций на ввод нам потребуется? Здесь нужно перебрать все элементы, т.е. операция на каждый элемент. Чем больше массив, тем больше операций.

Используя Big O нотацию: $O(n)$, или «сложность порядка n (order n)». Так же такой тип алгоритмов называют «линейными» или что алгоритм «линейно масштабируется».

Анализ

```
const sumContiguousArray = function(ary){
    //get the last item
    const lastItem = ary[ary.length - 1];
    //Gauss's trick
    return lastItem * (lastItem + 1) / 2;
}
const nums = [1,2,3,4,5];
const sumOfArray = sumContiguousArray(nums);
```

Можем ли мы сделать суммирование более эффективным? В общем случае нет. А если мы знаем, что массив гарантированно начинается с 1, отсортирован и не имеет пропусков? Тогда можно применить формулу $S = n(n+1)/2$ (где n последний элемент массива):

Такой алгоритм гораздо эффективнее $O(n)$, более того он выполняется за «постоянное/константное время», т.е. это $O(1)$.

Фактически операций не одна: нужно получить длину массива, получить последний элемент, выполнить умножение и деление. Разве это не $O(3)$ или что-нибудь такое? В Big O нотации фактическое кол-во шагов не важно, важно что алгоритм выполняется за константное время.

Алгоритмы с константным временем это всегда $O(1)$. Тоже и с линейными алгоритмами, фактически операций может быть $O(n+5)$, в Big O нотации это $O(n)$.

Не самые лучшие решения: $O(n^2)$

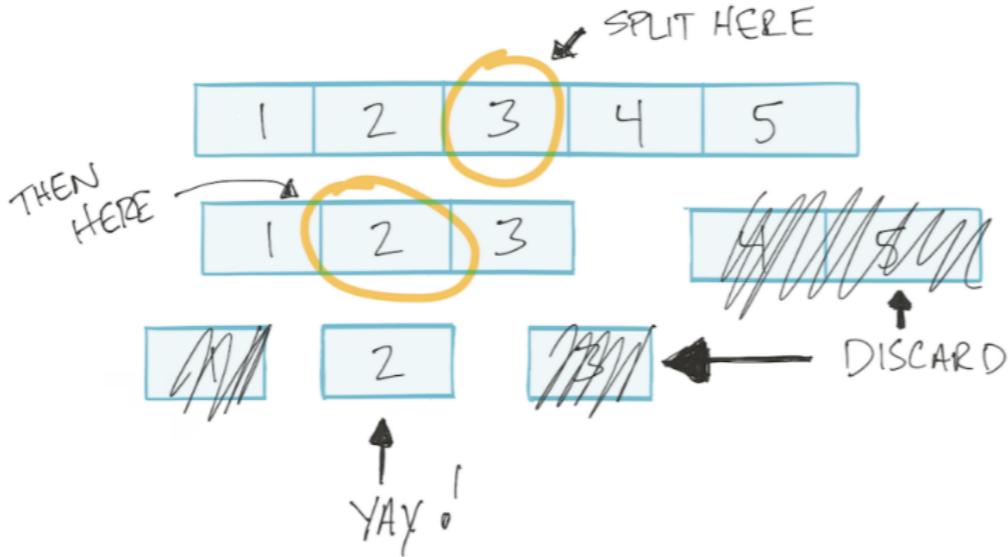
```
const hasDuplicates = function (num) {
    //loop the list, our O(n) op
    for (let i = 0; i < nums.length; i++) {
        const thisNum = nums[i];
        //loop the list again, the O(n^2) op
        for (let j = 0; j < nums.length; j++) {
            //make sure we're not checking same number
            if (j !== i) {
                const otherNum = nums[j];
                //if there's an equal value, return
                if (otherNum === thisNum) return true;
            }
        }
    }
    //if we're here, no dups
    return false;
}
const nums = [1, 2, 3, 4, 5, 5];
hasDuplicates(nums); //true
```

Давайте напишем функцию которая проверяет массив на наличие дублей. Решение с вложенным циклом:

Мы уже знаем что итерирование массива это $O(n)$. У нас есть вложенный цикл, для каждого элемента мы еще раз итерируем — т.е. $O(n^2)$ или «сложность порядка n квадрат».

Алгоритмы с вложенными циклами по той же коллекции всегда $O(n^2)$.

«Сложность порядка $\log n$ »: $O(\log n)$



В примере выше, вложенный цикл, сам по себе (если не учитывать что он вложенный) имеет сложность $O(n)$, т.к. это перебор элементов массива. Этот цикл заканчивается как только будет найден нужный элемент, т.е. фактически не обязательно будут перебраны все элементы. Но в Big O нотации всегда рассматривается худший вариант — искомый элемент может быть самым последним.

Здесь вложенный цикл используется для поиска заданного элемента в массиве. Поиск элемента в массиве, при определенных условиях, можно оптимизировать — сделать лучше чем линейная $O(n)$.

Пускай массив будет отсортирован. Тогда мы сможем использовать алгоритм «бинарный поиск»: делим массив на две половины, отбрасываем не нужную, оставшуюся опять делим на две части и так пока не найдем нужное значение. Такой тип алгоритмов называется «разделяй и властвуй» Divide and Conquer.

Улучшим $O(n^2)$ до $O(n \log n)$

```
const nums = [1, 2, 3, 4, 5];
const searchFor = function (items, num) {
    //use binary search!
    //if found, return the number. Otherwise...
    //return null. We'll do this in a later chapter.
}
const hasDuplicates = function (nums) {
    for (let num of nums) {
        //let's go through the list again and have a look
        //at all the other numbers so we can compare
        if (searchFor(nums, num)) {
            return true;
        }
    }
    //only arrive here if there are no dups
    return false;
}
```

Вернемся к задачке проверки массива на дубли. Мы перебирали все элементы массива и для каждого элемента еще раз делали перебор. Делали $O(n)$ внутри $O(n)$, т.е. $O(n^2)$ или $O(n^2)$.

Мы можем заменить вложенный цикл на бинарный поиск*. Т.е. у нас остается перебор всех элементов $O(n)$, внутри делаем $O(\log n)$. Получается $O(n * \log n)$, или $O(n \log n)$.

Использовать бинарный поиск для проверки массива на дубли — плохое решение. Здесь лишь показывается как в терминах Big O оценить сложность алгоритма показанного в листинге кода выше. Хороший алгоритм или плохой — для данной заметки не важно, важна наглядность.

Мышление в терминах Big O

- Получение элемента коллекции это $O(1)$. Будь то получение по индексу в массиве, или по ключу в словаре в нотации Big O это будет $O(1)$;
- Перебор коллекции это $O(n)$;
- Вложенные циклы по той же коллекции это $O(n^2)$;
- Разделяй и властвуй (Divide and Conquer) всегда $O(\log n)$;
- Итерации которые используют Divide and Conquer это $O(n \log n)$;

Задача 1

Чтобы зарегистрироваться на сайте надо ввести свои логин и пароль. Не может быть двух пользователей с одинаковыми логинами. Поэтому сервер должен убедиться, что имя пользователя уникально. Напишите программу, которая проверяет нового пользователя в списке зарегистрированных.

Рассмотрим задачу регистрации пользователей на сайте. Мы должны убедиться, что имя пользователя уникально. Как это сделать? Мы можем использовать алгоритм поиска, который проверит, есть ли уже такой пользователь в базе данных. Например, можно применить хеширование для быстрого поиска и минимизации временных затрат.

```
1 # Линейный поиск
2 def check(login, users):
3     for x in users:
4         if login == x:
5             return True
6     return False
7
8
9 users = ['megatron', 'superman', 'ira_dota', 'masha18', 'ivan']
10
11 result_1 = check('ira_dota', users)
12 result_2 = check('kirill', users)
13
14 print(f'ira_dota: {result_1}, kirill: {result_2}')
```

```
1 # Бинарный поиск
2 def check(login, users):
3     low = 0
4     high = len(users)-1
5     while low <= high:
6         m = low + (high-low)//2 # эквивалентно (high+low)//2
7         if users[m] == login:
8             return True
9         elif users[m] < login:
10            low = m+1
11        else:
12            high = m-1
13    return False
14
15
16 users = ['megatron', 'superman', 'ira_dota', 'masha18', 'ivan']
17
18 users_sorted = sorted(users)
19
20 result_1 = check('ira_dota', users_sorted)
21 result_2 = check('kirill', users_sorted)
22
23 print(f'ira_dota: {result_1}, kirill: {result_2}')
```

В общем случае можно возвращать позицию элемента, а не просто True или False. Для этого надо немного изменить код!

Сравнение алгоритмов

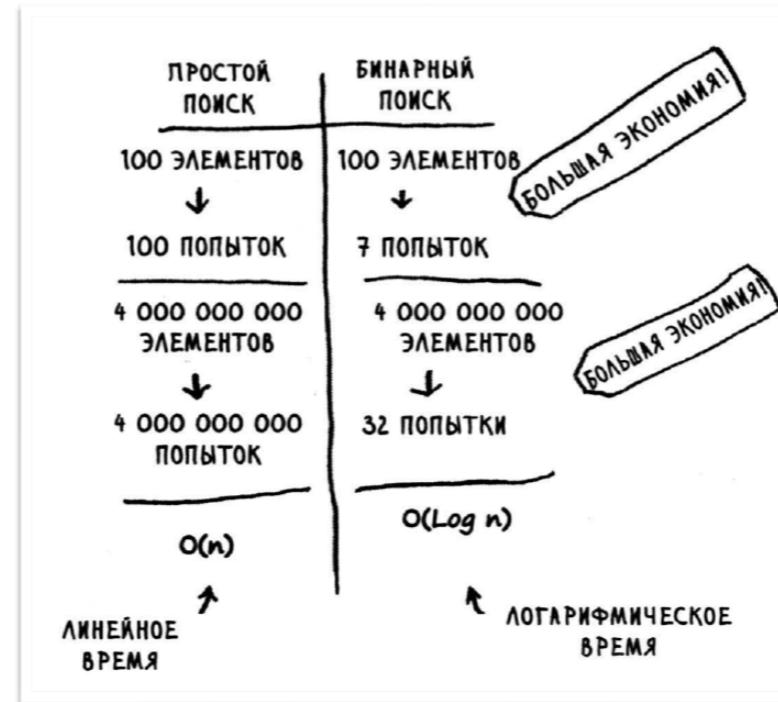
Время работы?

Память?

Количество инструкций?

Скорость роста времени или памяти?

Асимптотическая сложность



Амортизационная стоимость

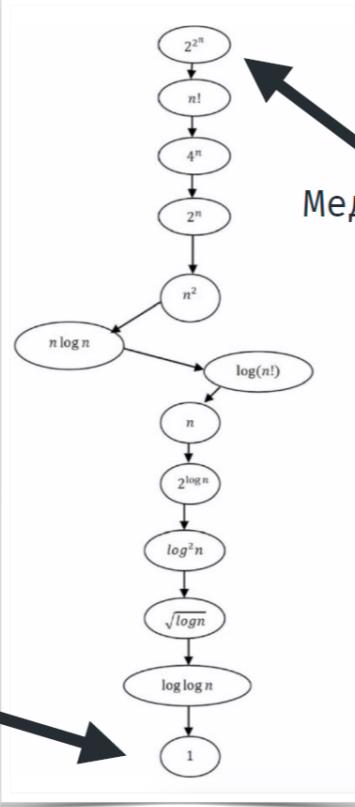
$$\tilde{T}(n) = \frac{T(n)}{n}$$



Средняя стоимость **одной** из n операций в наихудшем случае

Быстрый алгоритм!

Медленный алгоритм...





Задача 2

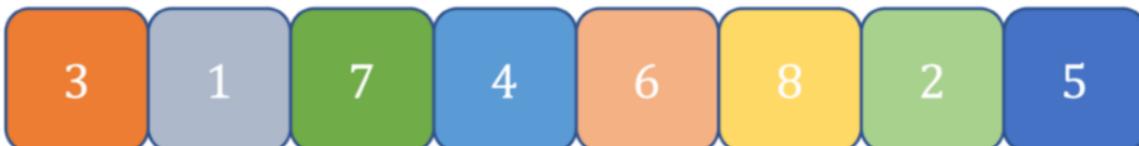
В системе регистрации был сбой и в списке идентификаторов пользователей есть повторяющиеся. Напишите программу, которая проверяет повторы.

Второй пример — это ситуация с регистрацией, где произошел сбой, и появились повторяющиеся идентификаторы пользователей. Мы должны написать программу, которая найдет эти дублирующиеся значения. Для решения этой задачи могут быть полезны алгоритмы сортировки или хеш-таблицы.

Шпаргалка

- Бинарный поиск работает намного быстрее простого.
- Время выполнения $O(\log n)$ быстрее $O(n)$, а с увеличением размера списка, в котором ищется значение, оно становится намного быстрее.
- Скорость алгоритмов **не** измеряется в секундах.
- Время выполнения алгоритма описывается *ростом* количества операций.
- Время выполнения алгоритмов выражается как «**О-большое**».

Пример массива неупорядоченных целочисленных данных

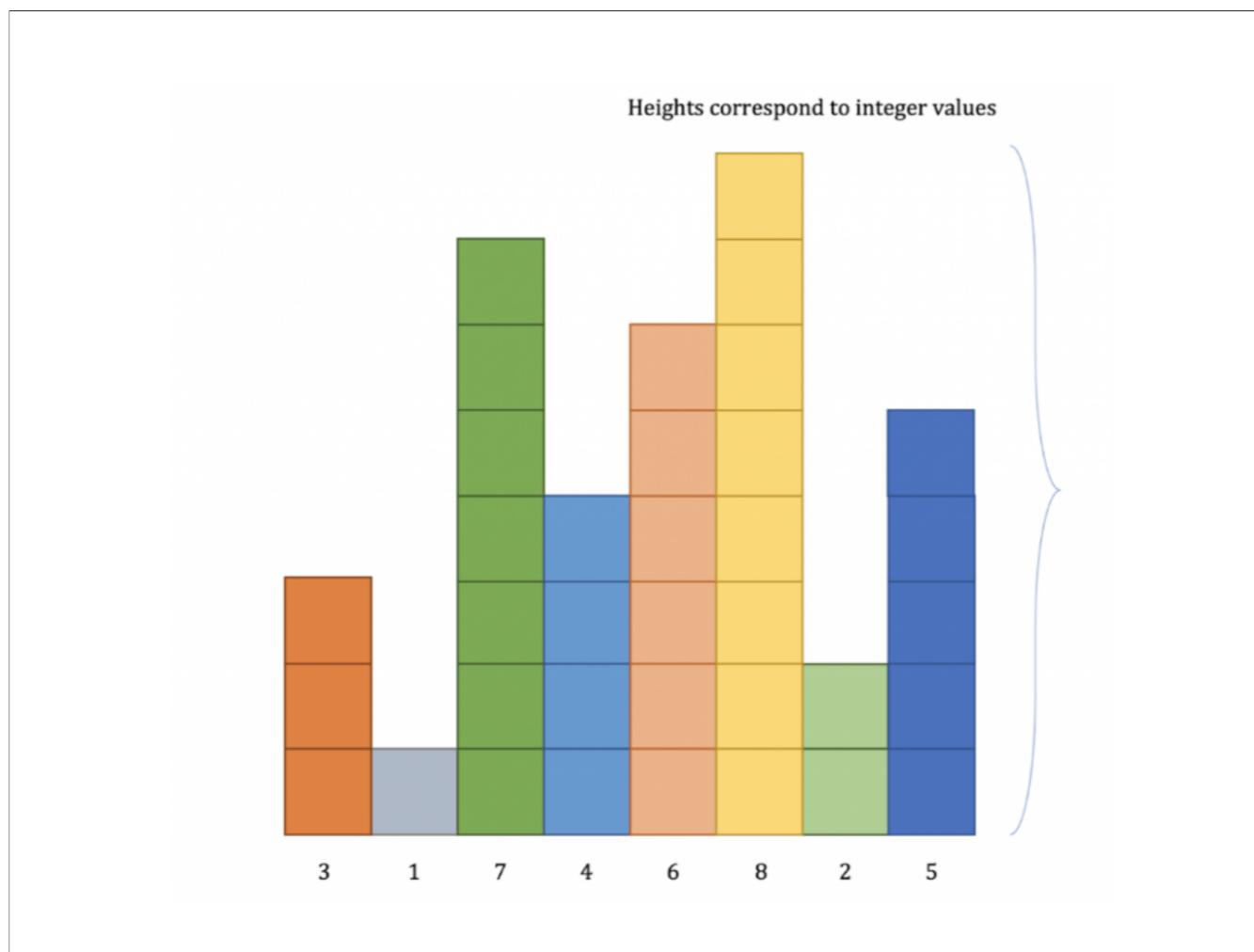


Array of Integer Data: n = 8

Сортировка массивов часто используется в программировании, чтобы помочь понять данные и выполнить поиск. Поэтому скорость сортировки больших объемов информации крайне важна для функциональных проектов и оптимизации времени работы. Есть много алгоритмов для упорядочения объектов. В статье вы посмотрите на реализацию и визуализацию пяти популярных алгоритмов сортировки. Код написан на Python, а графический интерфейс построен на Tkinter.

Эти 5 алгоритмов включают:

- Сортировка выбором
- Сортировка пузырьком
- Сортировка вставками
- Сортировка слиянием
- Быстрая сортировка quicksort



Давайте представим числовые данные с помощью столбцов. Высота столбца i равна значению элемента массива i . У них всех одинаковая ширина. Такое изображение списка числовых значений с Рис. 1 в виде столбцов приведено на Рис. 2.

Сортировка числовых данных по убыванию или по возрастанию требует соответствующей перестановки значений. Для анимации каждого алгоритма с данными, требуется обновлять диаграмму после замены отдельных элементов.

Сортировка выбором

```
def selection_sort(self, unsorted, n):

    # итерируемся по массиву
    for i in range(0, n):

        # инициализируемся первым значением
        current_min = unsorted[i]

        # инициализируем минимальный индекс
        min_index = i

        # итерируемся по оставшимся элементам массива
        for j in range(i, n):

            # проверяем, если j-тое значение меньше текущего минимального
            if unsorted[j] < current_min:

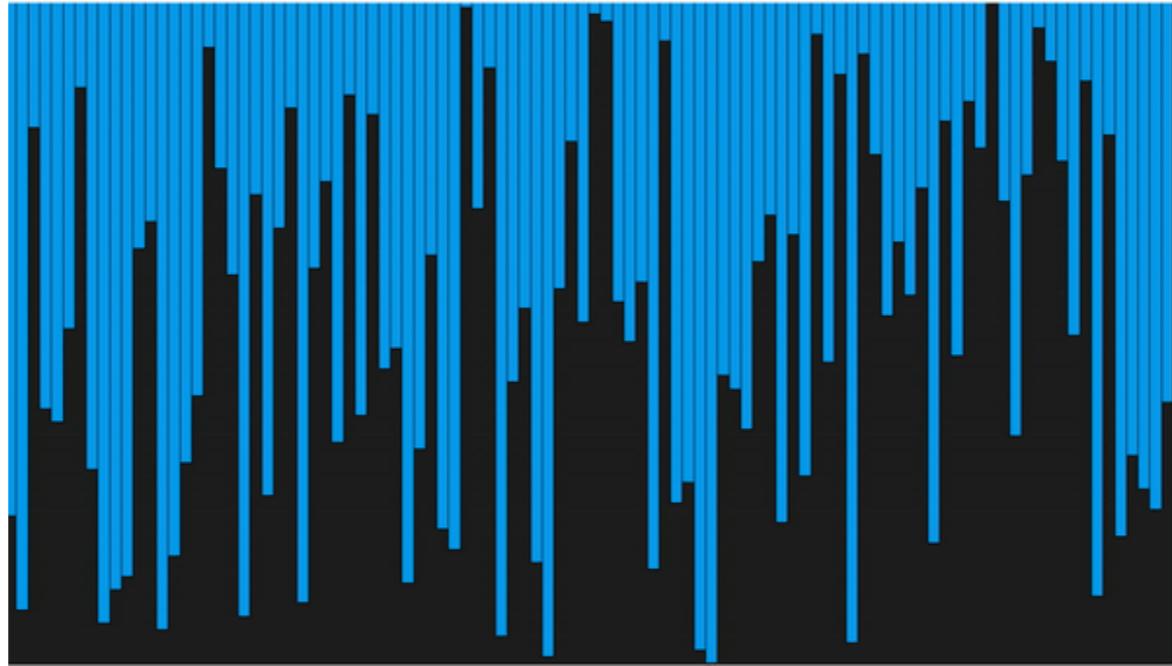
                # обновляем минимальные значение и индекс
                current_min = unsorted[j]
                min_index = j

        # меняем i-тое и j-тое значения
        swap(unsorted, i, min_index)
```

Алгоритмы сортировки выбором и пузырьком — это простые, но не самые эффективные методы сортировки. Оба алгоритма имеют сложность $O(n^2)$, что делает их неэффективными для больших наборов данных. Сортировка выбором находит наименьший элемент и меняет его местами с первым, тогда как пузырьковая сортировка многократно сравнивает соседние элементы и меняет их местами.

Ниже — реализация сортировки выбором на Python с подробными комментариями.

Внешний цикл производит итерации по всей длине несортированного массива. В это время внутренний цикл ищет минимальное значение в оставшейся части набора данных. Затем происходит единая замена, переставляющая элемент под номером i и элемент min_index .



Метод сортировки выбором обычно включает параметры:

- Временная сложность = $O(n^2)$.

n^2 итераций очевидны из двух вложенных циклов.

- Пространственная сложность = $O(1)$.

Как упоминалось выше, сортировка происходит в том же массиве, поэтому использование памяти не зависит от данных обработки.

Сортировка пузырьком

```
def bubble_sort(self, unsorted, n):
    """ алгоритм сортировки пузырьком """

    # итерируемся по неотсорт. массиву до предпоследнего элемента
    for i in range(0, n - 1):

        # проставляем условия флага для финального списка
        swapped = False

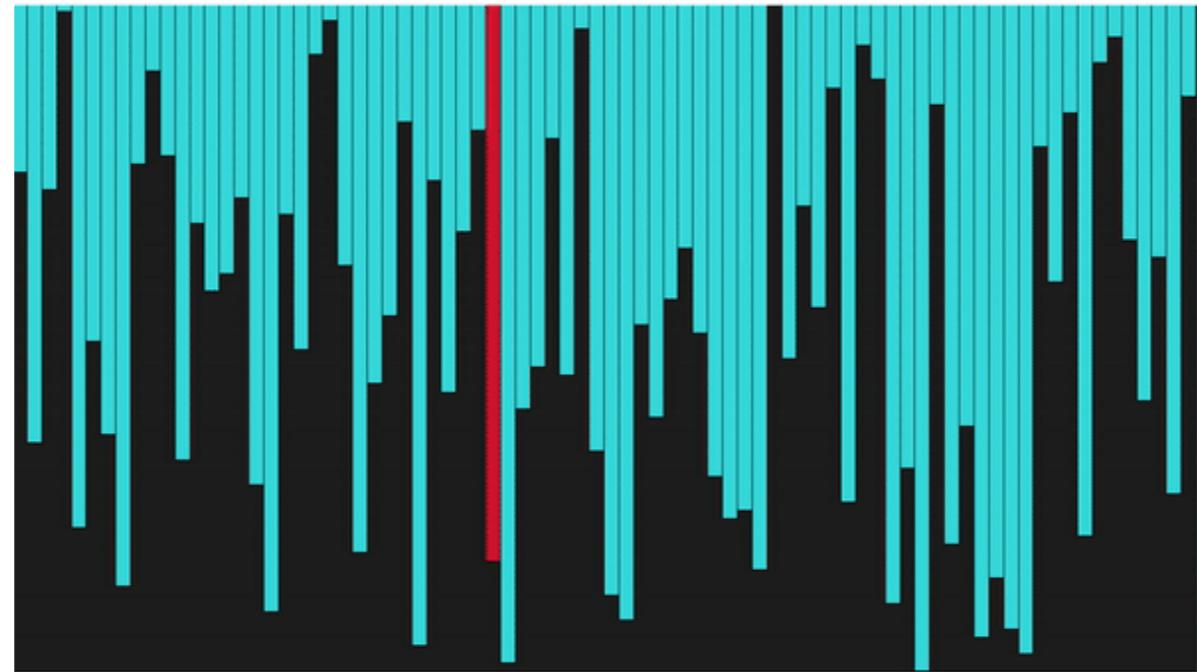
        # итерируемся по оставшимся неотсортированным объектам
        for j in range(0, n - 1 - i):

            # сравниваем соседние элементы
            if unsorted[j].value > unsorted[j + 1].value:

                # меняем элементы местами
                swap(unsorted, j, j + 1)
                swapped = True

            # завершаем алгоритм, если смены не произошло
            if not swapped:
                break
```

Несколько раз проходит по списку, сравнивая соседние элементы. Элементы меняются местами в зависимости от условия сортировки. Ниже показана реализация сортировки выбором на Python с комментариями.



Обратите внимание, что из всех алгоритмов у сортировки пузырьком наихудший случай и среднее значение:

• Временная сложность = $O(n^2)$.

Внутренний цикл работает не менее n раз. Поэтому операция займет не менее n^2 времени

• Пространственная сложность = $O(1)$.

Дополнительная память не используется, потому что происходит обмен элементов в исходном массиве

Большие значения всплывают, как пузырьки, в верхнюю часть списка по мере выполнения программы, как показано на Рис. 4.

Сортировка вставками

```
def insertion_sort(unsorted, n):
    """ сортировка вставками """

    # итерация по неотсортированным массивам
    for i in range(1, n):

        # получаем значение элемента
        val = unsorted[i].value

        # записываем в hole индекс i
        hole = i

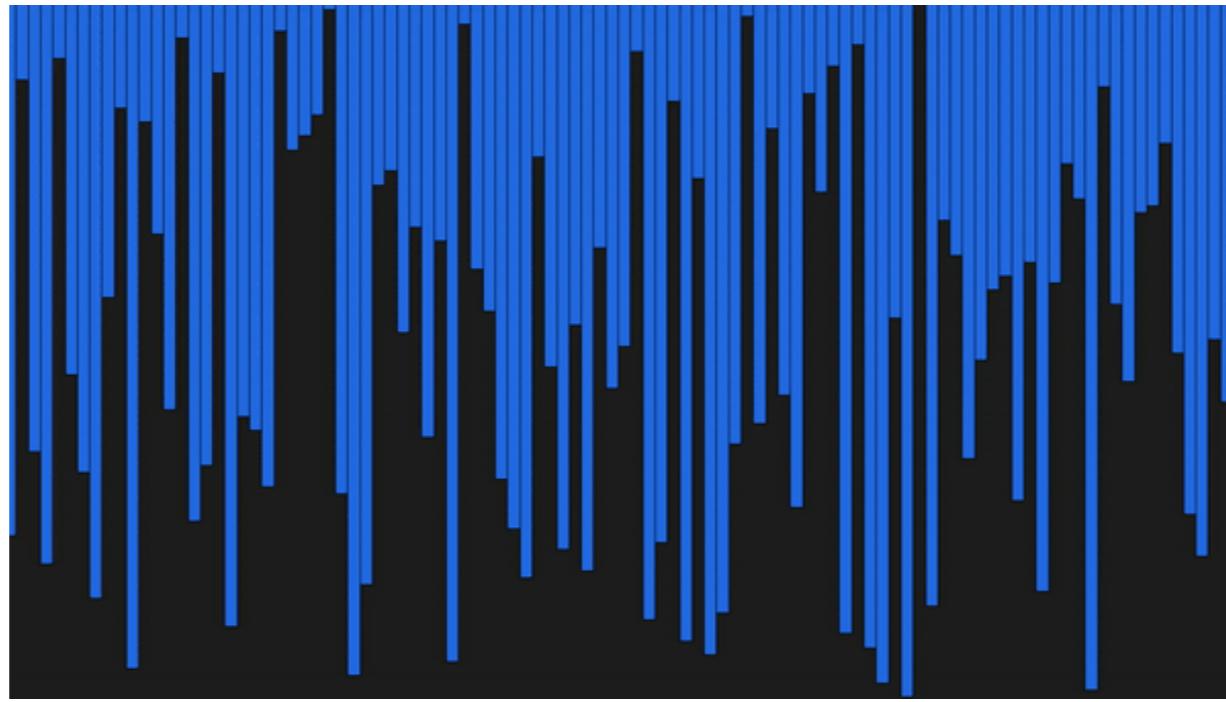
        # проходим по массиву в обратную сторону, пока не найдём элемент больше текущего
        while hole > 0 and unsorted[hole - 1].value > val:

            # переставляем элементы местами , чтобы получить правильную позицию
            unsorted[hole].value = unsorted[hole - 1].value

            # делаем шаг назад
            hole -= 1

        # вставляем значение на верную позицию
        unsorted[hole].value = val
```

Сортировка вставками работает быстрее, чем предыдущие алгоритмы, при малых объемах данных. Она имеет сложность $O(n^2)$ в худшем случае, но в лучшем случае может достичь $O(n)$, если данные уже частично отсортированы. Этот алгоритм эффективен для небольших массивов или в случаях, когда массив почти отсортирован.



У сортировки вставками есть наихудший случай:

- Временная сложность = **O(n²)**. Как внешний цикл for, так и внутренний while работают приблизительно n раз
- Пространственная сложность = **O(1)**. Операции проводятся с исходным массивом. Таким образом, дополнительной памяти не требуется.

Сортировка слиянием

```
def divide(self, unsorted, lower, upper):
    """ рекурсивная функция для разделения массива на два подмассива для сортировки """

    # с помощью рекурсии достигнут базовый случай
    if upper <= lower:
        return

    # получаем среднее значение для разделения
    mid = (lower + upper) // 2

    # делим массив посередине
    divide(unsorted, lower, mid)
    divide(unsorted, mid + 1, upper)

    # склеиваем отсортированные массивы
    merge(unsorted, lower, mid, mid + 1, upper)
```

Сортировка слиянием — это более сложный алгоритм с эффективной временной сложностью $O(n \log n)$. Она основана на принципе "разделяй и властвуй", разбивая массив на более мелкие части, сортируя их и затем сливая обратно в один отсортированный массив.

Другими словами, задача раскладывается на более мелкие аналогичные подзадачи до тех пор, пока не будет решен базовый случай.

Несортированный массив разделяется до тех пор, пока не выделится базовый случай отдельных элементов. Затем происходит сравнение между временными массивами, перемещающимися обратно вверх по стеку рекурсии.

```
def merge(unsorted, l_lower, l_upper, r_lower, r_upper):
    """ merging two sorted arrays to one sorted array """

    # извлекаем левый и правый индексы
    i, j = l_lower, r_lower

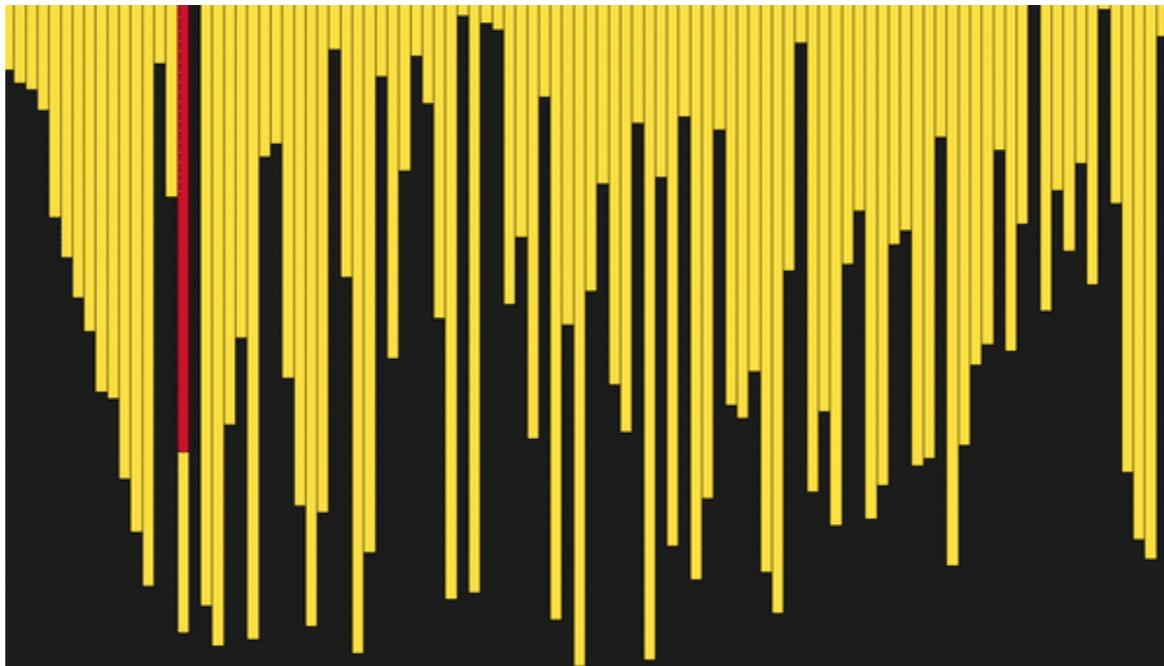
    # инициализируем временный массив
    temp = []

    # проходим по индексам
    while i <= l_upper and j <= r_upper:

        # определяем, какое значение следующим вставить во временный массив
        if unsorted[i].value <= unsorted[j].value:
            temp.append(unsorted[i])
            i += 1
        else:
            temp.append(unsorted[j])
            j += 1

    # одно из условий выше заканчивается первым
    # поэтому обрабатываем незаконченный случай
    while i <= l_upper:
        temp.append(unsorted[i])
        i += 1
    while j <= r_upper:
        temp.append(unsorted[j])
        j += 1
```

```
# присваиваем значения из временного массива
for y, k in enumerate(range(l_lower, r_upper + 1)):
    unsorted[k] = temp[y]
```



У сортировки слиянием есть:

- Временная сложность = $O(n \log(n))$. У алгоритмов сортировки «разделяй и властвуй» такая времененная сложность. Эта сложность – наихудший сценарий для подобных алгоритмов.
- Пространственная сложность = $O(n)$, если выделение памяти увеличивается не быстрее константы kN , т.е. кратной размеру набора данных.

Быстрая сортировка

```
def quick_sort(self, unsorted, start, end):
    """ быстрая сортировка """

    # останавливаемся, когда индекс слева достиг или превысил индекс справа
    if start >= end:
        return

    # определяем позицию следующего пивота
    i_pivot = partition(unsorted, start, end - 1)

    # рекурсивный вызов левой части
    quick_sort(unsorted, start, i_pivot)

    # рекурсивный вызов правой части
    quick_sort(unsorted, i_pivot + 1, end)
```

Наконец, быстрая сортировка, пожалуй, один из самых популярных алгоритмов, используется во многих библиотеках и языках программирования. Она имеет временную сложность $O(n \log n)$ в среднем случае, но может достигать $O(n^2)$ в худшем случае, если разделение происходит неудачно.

Быстрая сортировка примерно в два-три раза быстрее основных конкурентов, сортировки слиянием и пирамидальной сортировки. Ее часто реализуют рекурсивно, как на примере ниже.

Значения pivot лежат в основе алгоритма быстрой сортировки. По существу, pivot опорные значения. Постановка pivot означает, что объекты слева всегда меньше, а элементы справа больше, чем pivot.

```
def partition(self, unsorted, start, end):
    """ arrange (left array < pivot) and (right array > pivot) """

    # выбираем значение pivot как последний элемент неотсортированного сегмента
    pivot = unsorted[end]

    # назначаем на pivot значение левого индекса
    i_pivot = start

    # проходим от начала до конца текущего сегмента
    for i in range(start, end):

        # сравниваем текущее значение со значением pivot
        if unsorted[i].value <= pivot.value:

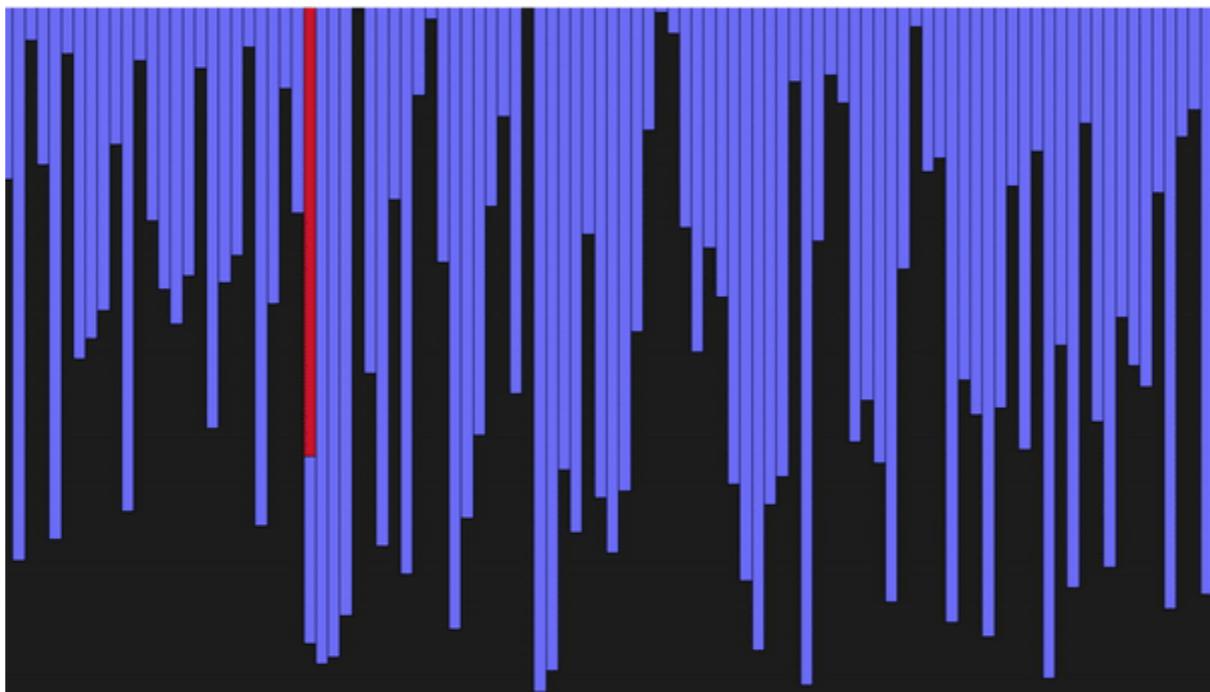
            # меняем местами текущее значение и значение pivot
            swap(unsorted, i, i_pivot)

            # увеличиваем значение пивота
            i_pivot += 1

    # ставим пивот в правильную позицию, заменив со значением слева
    swap(unsorted, i_pivot, end)

    # возвращаем следующее значение пивота
    return i_pivot
```

Рекурсивно разбивая массив, выбирая точки pivot и назначая их в правильном месте, получаем окончательно отсортированный массив.
У быстрой сортировки есть *среднее значение*:



Временная сложность = **$O(n \log n)$** . Как и сортировка слиянием, данная нотация определяется по принципу разделяй и властвуй или быстрой сортировки. Наихудший сценарий – $O(n^2)$. Однако это происходит, только когда элементы массива правильно восходят или нисходят.