**Оглавление**

**[Алгоритмы сортировки на Python 2](#_Toc17939)**

[Bubble Sort (пузырьковая сортировка) 2](#_Toc31752)

[Selection Sort (сортировка выбором) 2](#_Toc18377)

[Insertion Sort (сортировка вставками) 3](#_Toc26456)

[Merge Sort (сортировка слиянием) 3](#_Toc23844)

[Quick Sort (быстрая сортировка) 4](#_Toc10118)

**[Временная сложность 7](#_Toc2429)**

**[Структуры данных 9](#_Toc4849)**

**[Классы данных 10](#_Toc20602)**

**[Спецификация Python 11](#_Toc4706)**

**[Потенциальные задачи 12](#_Toc1170)**

# **Алгоритмы сортировки на Python**

# <https://tproger.ru/translations/sorting-algorithms-in-python/> - Объяснение алгоритмов сортировки с примерами на Python

<https://visualgo.net/en/sorting> - визуализация сортировок

## **Bubble Sort (пузырьковая сортировка)**

1. Берём самый первый элемент массива и сравниваем его со вторым. Если первый больше второго — меняем их местами с первым, если нет — ничего не делаем.
2. Затем берём второй элемент массива и сравниваем его со следующим — третьим. Если второй больше третьего — меняем их местами, если нет — ничего не делаем.
3. Проходим так до предпоследнего элемента, сравниваем его с последним и ставим наибольший из них в конец массива. Всё, мы нашли самое большое число в массиве и поставили его на своё место.
4. Возвращаемся в начало алгоритма и делаем всё снова точно так же, начиная с первого и второго элемента. Только теперь даём себе задание не проверять последний элемент — мы знаем, что теперь в конце массива самый большой элемент.
5. Когда закончим очередной проход — уменьшаем значение финальной позиции, до которой проверяем, и снова начинаем сначала. Так делаем до тех пор, пока у нас не останется один элемент.

Реализация в коде:

def bubbleSort(array: List[int]):

    length = len(array)

    while length > 1:

        for i in range(length-1):

            if array[i] > array[i+1]:

                array[i], array[i+1] = array[i+1], array[i]

        length -= 1

## **Selection Sort (сортировка выбором)**

1. В неотсортированном подмассиве ищется локальный максимум (минимум).
2. Найденный максимум (минимум) меняется местами с последним (первым) элементом в подмассиве.
3. Если в массиве остались неотсортированные подмассивы — смотри пункт 1.

Реализация в коде:

def selectionSort(array: List[int]):

    length = len(array)

    while length > 1:

        max\_ = array[0]

        for i in range(length):

            if array[i] >= max\_:

                max\_ = array[i]

                j = i

        array[length-1], array[j] = array[j], array[length-1]

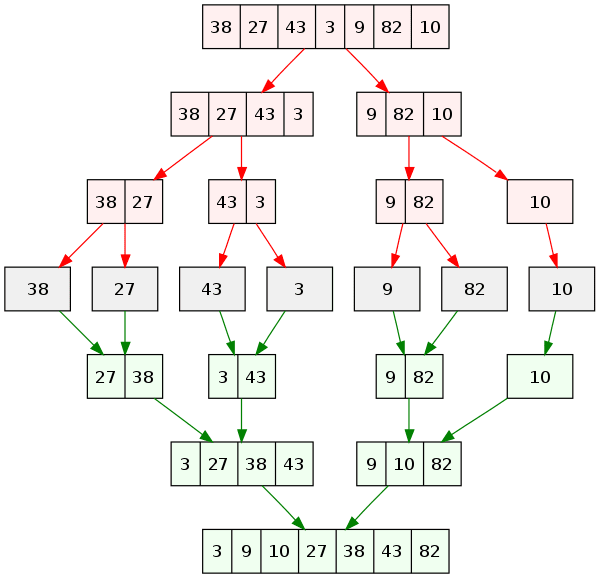
        length -= 1

## **Insertion Sort (сортировка вставками)**

Предполагается, что изначально есть готовая последовательность. На каждой итерации ее необходимо дополнить путем вставки нового числа в последовательность. Вставка производится таким образом, чтобы не нарушать упорядоченность готовой последовательности.

## **Merge Sort (сортировка слиянием)**

Интуитивный вариант — взять список, разделить его пополам, затем еще пополам и еще, пока у не окажется куча списков с длиной, равной единице. После этого нужно выстроить элементы в пары, располагая их в правильном порядке, а затем соединять эти пары вместе, образуя все большие упорядоченные группы, до тех пор, пока не получите целый отсортированный список.



Реализация в коде:

def mergeSort (nums: List[int]):

      length = len(array)

      # Если список не пустой и не единичный

      if length > 1:

        # Разделяем список на 2 части

        center = len(nums) // 2

        left = nums[:center]

        right = nums[center:]

        # Сортируем отдельно обе половины списка

        mergeSort(left)

        mergeSort(right)

        # Теперь нужно отслеживать три индекса - i, j и k:

        # i — индекс в списке left,

        # j — индекс в списке right,

        # k — индекс в исходном списке array, в который в конечном итоге нужно

        # вставить все числа по порядку.

        i = j = k = 0

        # Если число из списка left меньше, чем число из списка right, мы

        # вставляем его в nums на позицию k, после чего увеличиваем индекс i на

        # единицу. Если число из списка right меньше или равно числу из списка

        # left, тогда оно отправляется в nums, а мы увеличиваем на единицу индекс

        # j. Наконец, после добавления любого из чисел в список nums, мы

        # увеличиваем на единицу индекс k.

        while i < len(left) and j < len(right):

          if left[i] < right[j]:

              nums[k] = left[i]

              i+=1

          else:

              nums[k] = right[j]

              j+=1

          k+=1

        # Цикл для перебора остатков элементов в списке

        while i < len(left):

            nums[k] = left[i]

            i+=1

            k+=1

        while j < len(right):

            nums[k] = right[j]

            j+=1

            k+=1

## **Quick Sort (быстрая сортировка)**

Алгоритм:

1. Выбрать значения для 3 переменных: опорный индекс (pivot), начало интервала (begin), конец интервала (end).

Для pivot можно выбрать начало, середину или конец массив. В качестве значений по умолчанию begin=0 и end=None.

Допустим, в качестве pivot выбрано начало.

1. Переменной end на начальном шаге присваиваем значение len(array)-1.
2. Проверяем, был ли пройден весь массив. Для этого необходимо сравнить begin >= end. Если да, то вернуть None. Если нет, то начать первую итерацию алгоритма.
3. Выбираем опорный элемент (array[begin]).
4. Сравниваем значения на отрезке от begin+1 до end+1 (опорный элемент (array[begin]) не входит в данный отрезок).

Если array[begin] >= array[i], то мы увеличиваем pivot на 1 и меняем местами array[i] и array[pivot], чтобы толкнуть меньший элемент к началу.

1. По завершению прохода цикла создаем новый опорный элемент, меняя array[pivot] и array[begin] местами.
2. Возвращаем значение pivot.
3. Начинаем вторую итерацию алгоритма. Для этого рекурсивно вызываем функцию quick\_sort(). В качестве входных значений: array, begin, pivot-1 (после pivot-1 все значения уже отсортированы и располагаются справа). Повторяем шаги 3-7.
4. В третьей итерации в качестве входных значений: array, pivot+1, end.

Реализация в коде:

def partition(array: List[int], begin: int, end: int) -> int:

    # Опорный индекс делаем равным начальному. Данный индекс отвечает за число,

    # которое будет отсортировано в конец

    pivot = begin

    # Рассматриваем обрезанный кусок массива, в который НЕ включен опорный

    # элемент array[begin]. Отсортированный кусок остается справа в конце.

    for i in range(begin + 1, end + 1):

        # Смотрим, является ли число в обрезанном массиве меньше

        # или равным опорному элементу

        if array[i] <= array[begin]:

            # Если да, то мы увеличиваем индекс опорного элемента

            pivot += 1

            # Меняем местами значения под текущим индексом и увеличинным опорным

            # индексом, т. е. постепенно

            # отодвигаем большее число в конец

            array[i], array[pivot] = array[pivot], array[i]

    # Завершаем цикл, создаем новый опорный элемент: меняем array[pivot] и

    # array[begin] местами

    array[pivot], array[begin] = array[begin], array[pivot]

    # Вернули индекс нового опорного элемента

    return pivot

# Реализация рекурсивной части

def quickSort\_short(array: List[int], begin=0, end=None) -> List[int]:

    # Начальная инициализация

    if end is None:

        end = len(array) - 1

    def \_quicksort(array: List[int], begin: int, end: int) -> Any:

        # Если мы перебрали весь массив, т. е. начальный индекс больше или равен

        # концу сортировки, то выходим

        if begin >= end:

            return

        # Просматриваем весь массив

        pivot = partition(array, begin, end)

        # Вместо конца массива стоит опорный индекс, поскольку мы закинули самое

        # большое число в конец - массив там отсортирован

        \_quicksort(array, begin, pivot - 1)

        # Во время прохода по массиву меньшие числа могут застрять в середине.

        # Поэтому мы смещаем начало на единицу

        \_quicksort(array, pivot + 1, end)

    return \_quicksort(array, begin, end)

Другая реализация предлагает использовать метод разделяй и властвуй. В этом методе мы так же избираем опорный элемент. Предположим, что он находится в начале массива или pivot=array[0].

Тогда далее нам необходимо создать три списка:

* значения меньше pivot;
* значения равные pivot;
* значения больше pivot.

Мы будем каждый такой список делить еще на 3 и далее до тех пор, пока длина списка не станет равной 1.

Реализация в коде:

# Через разбитие элементов на 3 части

def quickSort\_long(lst: List[int]) -> List[int]:

  # Условие выхода из рекурсии

  if len(lst) <= 1:

    return lst

  # Опорный элемент

  pivot = lst[0]

  left = list(filter(lambda x: x < pivot, lst))

  center = [x for x in lst if x == pivot]

  right = list(filter(lambda x: x > pivot, lst))

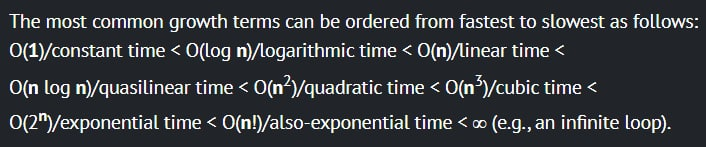
  return quickSort\_long(left) + center + quickSort\_long(right)

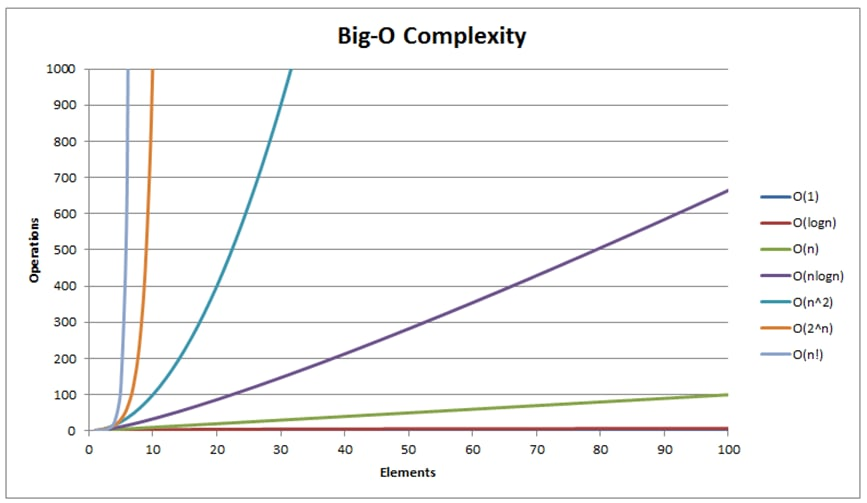
**Другие алгоритмы на Python**

<https://github.com/algorithmica-org/ru/blob/master/binary-search.md> - бинарный поиск в Гитхабе

# **Временная сложность**

<https://proglib.io/p/slozhnost-algoritmov-i-operaciy-na-primere-python-2020-11-03> - анализ сложности с примерами программ на Python





Типы временных сложностей и что за них можно сделать:

**O(1)**

* **O(log n)**
* **O(n)**
* **O(n log n)**
* **O(n2)**
* **O(2n)**

**O(n!)**

<https://pythonist.ru/chto-takoe-vremennaya-slozhnost-algoritma/>

# **Структуры данных**

Линейные структуры данных – массивы, стеки, связанные списки, хэш-таблицы и очереди.

Нелинейные структуры данных – деревья, графы, множества.

<https://bestprogrammer.ru/programmirovanie-i-razrabotka/8-struktur-dannyh-python>

# **Классы данных**

**<https://python-scripts.com/data-class>**

# **Спецификация Python**

**Общее**

<https://nuancesprog.ru/p/4448/> - лайфхаки в Python

<http://pythonicway.com/python-functinal-programming> - информация о **lambda**, **map**, **filter**, **reduce**, **zip (кратко)**

**map**

**reduce**

**filter**

**lambda**

В Python можно определить функцию, не давая ей имени. Такая функция называется анонимной. Лямбда-функции могут иметь любое количество аргументов, но только одно выражение.

**zip**

**Генераторы**

<https://pythonist.ru/generatory-v-python/>

**Декораторы**

**Итераторы**

# Потенциальные задачи

<https://proglib.io/p/15-questions-for-programmers> (15 шт)

<https://cmsmagazine.ru/journal/items-80-problems-with-it-interviews/> (80 шт)

<https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%BE%D0%BA%D1%81_%D0%9C%D0%BE%D0%BD%D1%82%D0%B8_%D0%A5%D0%BE%D0%BB%D0%BB%D0%B0>