



## Урок 7

# Алгоритмы сортировки

Сортировка пузырьком. Быстрая сортировка (Quick sort). Сортировка Шелла. Сортировка сложных структур с использованием ключа. Обратная сортировка. Сортировка с использованием функции attrgetter.

[Введение](#)

[Алгоритмы сортировки](#)

[Алгоритм сортировки выбором](#)

[Алгоритм сортировки вставками](#)

[Быстрая сортировка](#)

[Сортировка Шелла](#)

[Обратная сортировка](#)

[Сортировка с использованием функции attrgetter](#)

[Сортировка сложных структур с использованием ключа](#)

[Оценка алгоритмов сортировки](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

# Введение

На этом уроке разберем основные алгоритмы сортировки. Это одна из фундаментальных алгоритмических задач программирования.

Сортировка — это упорядочение элементов любой структуры хранения данных (массив, список, кортеж и так далее) в соответствии с конкретным критерием. Пример из жизни — построение по росту на уроке физкультуры. Самые высокие становятся на правый фланг, за ними — ребята поменьше ростом, и замыкает строй на левом фланге самый низкорослый ученик. Не важно, как зовут учеников, — в строю учитывается только один параметр — рост. А в классном журнале школьники записаны по фамилии, алфавитном порядке. Обратите внимание: природа сортируемых объектов не важна.

Обычно под алгоритмом сортировки подразумевают алгоритм упорядочивания множества элементов по возрастанию или убыванию.

Элементы с одинаковыми значениями в упорядоченной последовательности располагаются рядом друг за другом в любом порядке. Но иногда бывает полезно сохранить их первоначальный порядок.

В этих алгоритмах лишь часть данных используется в качестве ключа сортировки. Это атрибут (или несколько), по значению которого определяется порядок элементов. При написании алгоритмов сортировок массивов следует учесть, что ключ полностью или частично совпадает с данными.

Практически каждый алгоритм сортировки можно разбить на 3 части:

- сравнение, определяющее упорядоченность пары элементов;
- перестановка, меняющая местами пару элементов;
- собственно сортирующий алгоритм, который сравнивает и переставляет элементы до тех пор, пока все они не будут упорядочены.

Алгоритмы сортировки можно встретить там, где обрабатывается и хранится большой объем информации. Некоторые задачи обработки данных решаются проще, если данные заранее упорядочить.

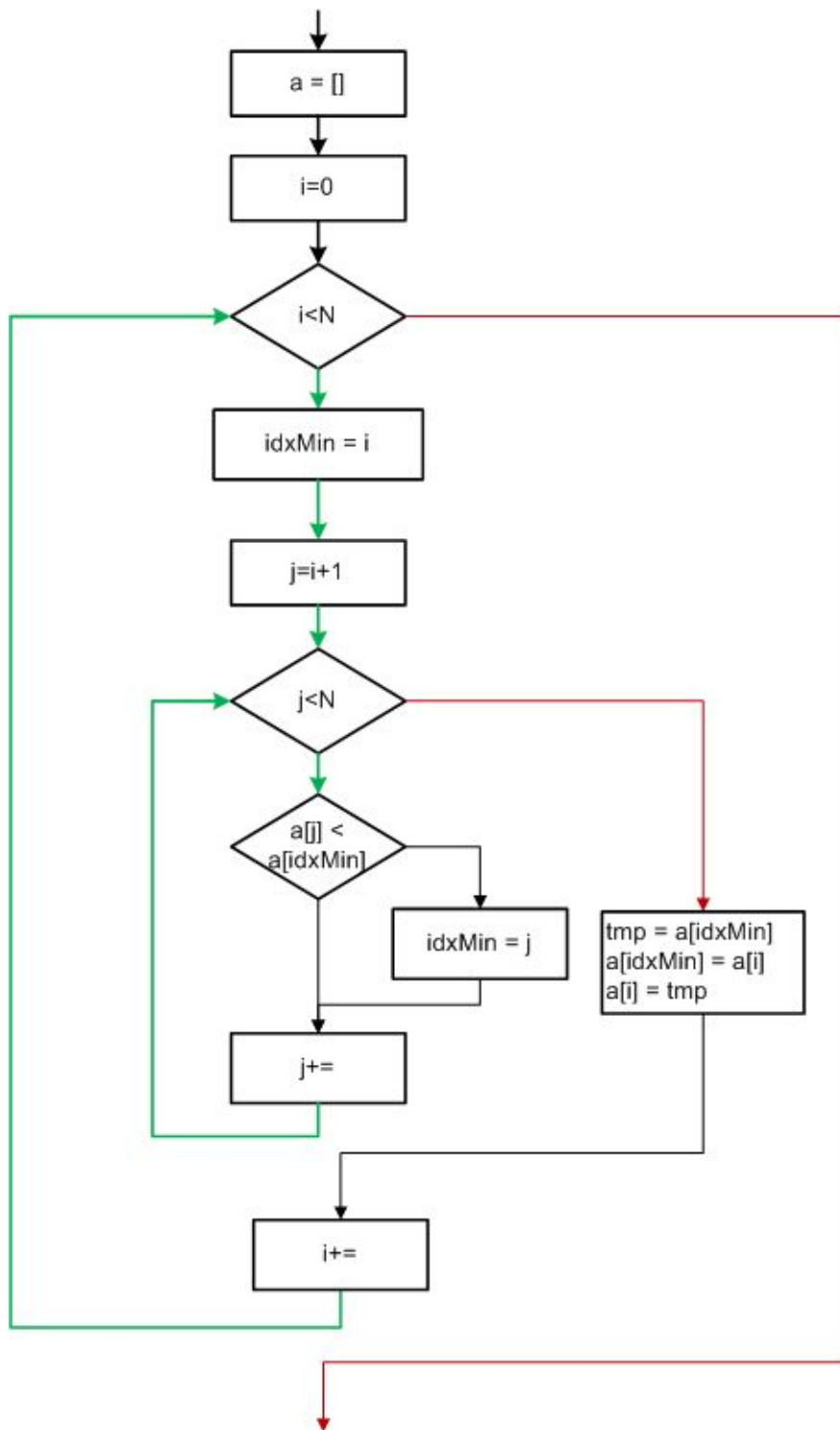
Все алгоритмы сортировки рассмотрим применительно к массивам.

## Алгоритмы сортировки

### Алгоритм сортировки выбором

Идея метода выбора — сначала найти минимальный элемент массива и поставить его на первое место. Следующий проход по массиву начинается со второго элемента, который сравнивается со всеми остальными элементами массива. Наименьший меняется с ним местами, и так повторяется  $N-1$  раз ( $N$  — это размер массива). Так во внутреннем цикле происходит поиск наименьшего элемента, запоминается его значение и место в массиве. После он меняется местами с текущим элементом массива.

Блок-схема:

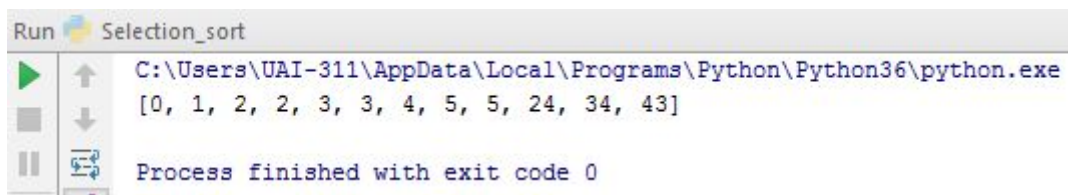


Отсюда и название алгоритма – в нем наименьший элемент выбирается поочередно.

Реализация на Python:

```
def selection_sort(array_to_sort):
    a = array_to_sort
    for i in range(len(a)):
        idx_min = i
        for j in range(i+1, len(a)):
            if a[j] < a[idx_min]:
                idx_min = j
        tmp = a[idx_min]
        a[idx_min] = a[i]
        a[i] = tmp
    return a

ary = [0,3,5,1,2,3,5,4,2,34,43,24]
print(selection_sort(ary))
```



## Алгоритм сортировки вставками

Алгоритм сортировки вставками сложно изложить словами, но его принцип считается одним из самых простых. Идея заключается в том, что при каждом проходе по массиву мы берем элемент и ищем его позицию для вставки.

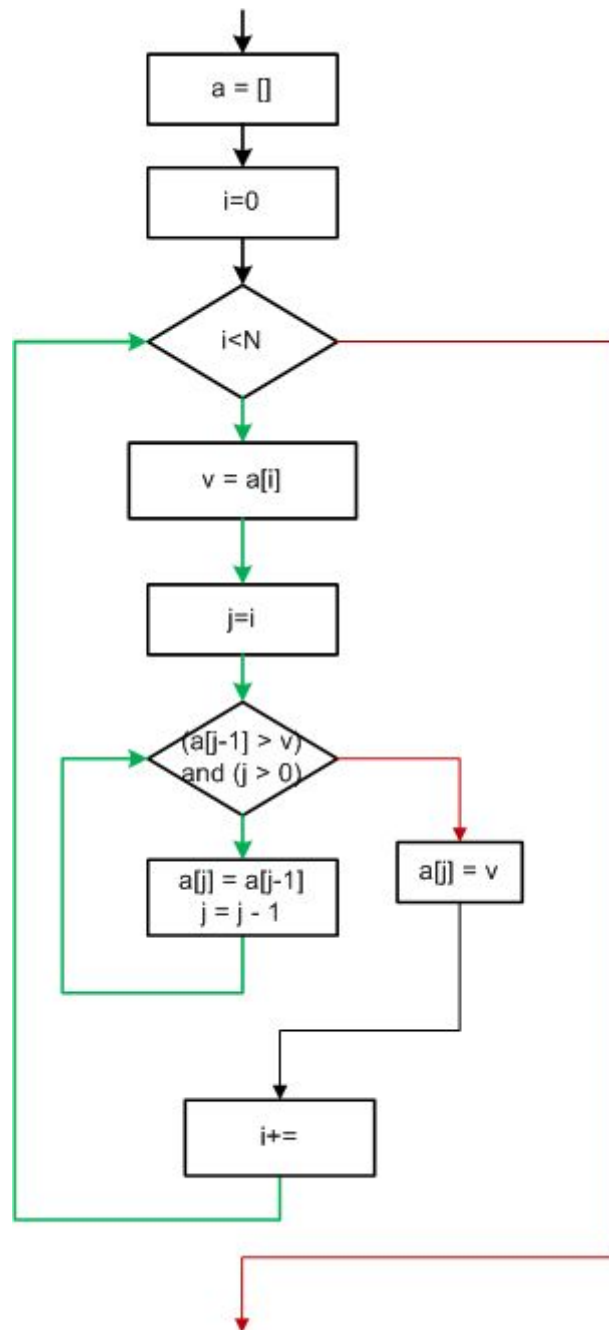
Каждый из нас не раз применял принцип этого алгоритма, когда сортировал купюры в кошельке. Берем 100 рублей и смотрим: остальные банкноты – по 50 и 500 рублей. Между ними и вставляем сотенную.

Основной (внешний) цикл для прохода по массиву начинается не с 0-го элемента, а с 1-го, потому что элемент до первого будет нашей отсортированной последовательностью. Уже относительно этого элемента с индексом 0 мы будем вставлять все остальные.

Основные шаги алгоритма:

- из массива последовательно берется каждый элемент;
- вставляется в отсортированную часть (например, в начале массива).

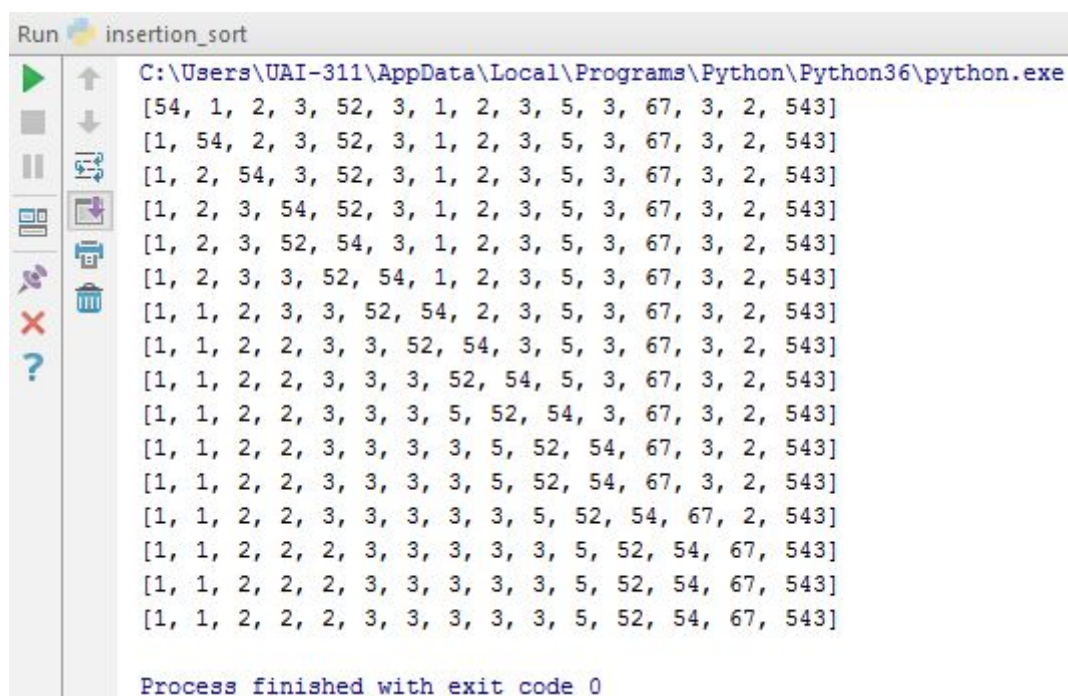
Блок-схема:



Программная реализация:

```
def insertion_sort(array_to_sort):
    a = array_to_sort
    for i in range(len(a)):
        v = a[i]
        j = i
        while (a[j-1] > v) and (j > 0):
            a[j] = a[j-1]
            j = j - 1
        a[j] = v
        print(a)
    return a

ary = [54,1,2,3,52,3,1,2,3,5,3,67,3,2,543]
print(insertion_sort(ary))
```



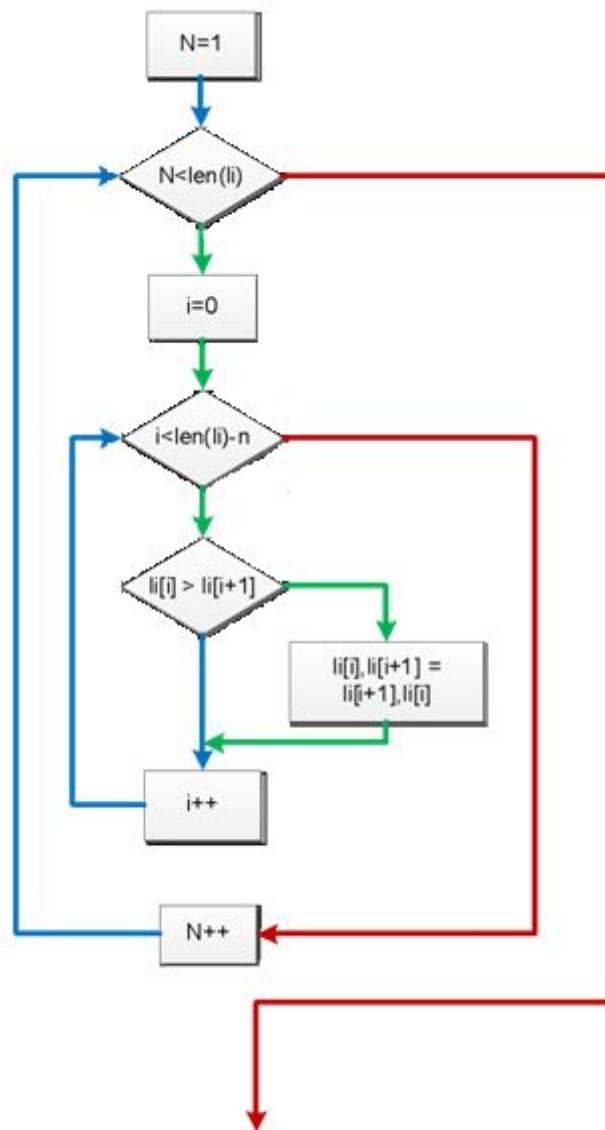
```
Run insertion_sort
C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe
[54, 1, 2, 3, 52, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 54, 2, 3, 52, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 2, 54, 3, 52, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 2, 3, 54, 52, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 2, 3, 52, 54, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 2, 3, 3, 52, 54, 1, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 1, 2, 3, 3, 52, 54, 2, 3, 5, 3, 67, 3, 2, 543]
[1, 1, 2, 2, 3, 3, 52, 54, 3, 5, 3, 67, 3, 2, 543]
[1, 1, 2, 2, 3, 3, 3, 52, 54, 5, 3, 67, 3, 2, 543]
[1, 1, 2, 2, 3, 3, 3, 5, 52, 54, 3, 67, 3, 2, 543]
[1, 1, 2, 2, 3, 3, 3, 3, 5, 52, 54, 67, 3, 2, 543]
[1, 1, 2, 2, 3, 3, 3, 3, 3, 5, 52, 54, 67, 2, 543]
[1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 5, 52, 54, 67, 543]
[1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 5, 52, 54, 67, 543]
[1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 5, 52, 54, 67, 543]
```

## Алгоритм сортировки пузырьком

Это наиболее распространенный, и, пожалуй, самый простой метод сортировки. Название закрепилось по аналогии с движением воздушного пузырька, который выталкивается вверх водой.

Идея в том, чтобы по очереди ставить элементы на то место, которое они должны занимать в отсортированном массиве. Массив из  $N$  элементов сортируется в  $N-1$  этапов. На первом этапе на последнее место ставится самый большой элемент, и к нему больше не обращаются. На втором этапе на предпоследнее место помещается второй по величине элемент массива и так далее. На последнем этапе на позицию первого элемента становится второй снизу элемент массива. После этого весь массив оказывается упорядоченным. Каждый этап состоит из последовательных сравнений соседних элементов, начиная с пары с индексами  $0 - 1$  и заканчивая  $(N-1) - N$ . Если  $(N-1)$  оказывается больше  $N$ , эта пара элементов меняется местами. Таким образом, на  $N$  месте оказывается наибольший элемент из просмотренных.

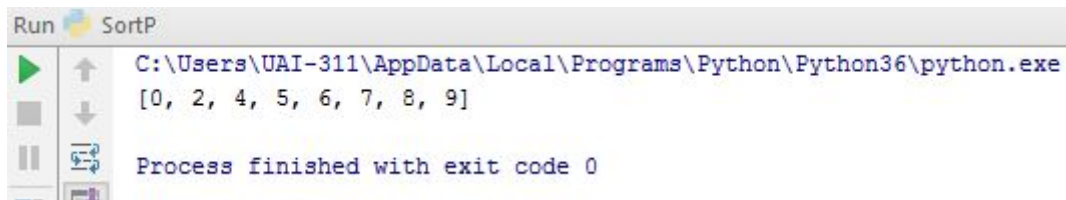
Блок-схема:



Программная реализация:

```
li = [5,2,7,4,0,9,8,6]
n = 1
while n < len(li):
    for i in range(len(li)-n):
        if li[i] > li[i+1]:
            li[i],li[i+1] = li[i+1],li[i]
    n += 1
print(li)
```

Результат работы программы:



Возможно, цикл выдвигания наибольшего числа не выполнит ни одной перестановки. Это будет означать, что массив уже отсортирован. Но алгоритм не учитывает этот факт и продолжает работу, устанавливая самые большие элементы в конец анализируемого подмассива – хотя они уже стоят на своих местах. Поэтому необходимо ввести одно дополнение: если за время прохождения внутреннего цикла не было произведено перестановок, то алгоритм сортировки завершается.

## Быстрая сортировка

В результирующем упорядоченном массиве каждый элемент делит массив на две части: стоящие слева от него элементы – не больше его, а стоящие справа – не меньше его.

Как получить этот массив? Сначала случайным образом выбираем элемент массива, который будет корневым. Именно относительно него будет производиться сортировка оставшихся элементов. Слева от корневого элемента поставим те, что меньше его, в справа – больше. В итоге корневой элемент оказывается на своем месте. Далее по аналогии сортируем левую и правую части массива.

Этот алгоритм можно реализовать как рекурсивным перебором, так и с помощью итерационного подхода. Наиболее распространен первый вариант. Рассмотрим его.

### Шаг рекурсии

1. Выбираем ключевой индекс и разделяем по нему массив на две части. Это можно делать разными способами, но в данной реализации используем случайное число. Распространенный метод выбора – это взять целое от среднего значения суммы индексов всех элементов;
2. Перемещаем все элементы больше ключевого в правую часть массива, а меньше ключевого – в левую. Теперь ключевой элемент находится в правильной позиции;
3. Повторяем первые два шага, пока массив не будет полностью отсортирован.

Рассмотрим работу алгоритма на графическом примере. Дан следующий массив:



3	7	4	4	6	5	8
---	---	---	---	---	---	---

Случайным образом выбираем ключевой элемент:

3	7	4	4	6	5	8
---	---	---	---	---	---	---

Получаем, что ключевым будет элемент под индексом 4, с соответствующим значением 6. Затем перемещаем остальные элементы массива относительно корневого по правилу: слева элементы меньше корневого, справа больше.

3	5	4	4	6	7	8
---	---	---	---	---	---	---

Рекурсивно вызываем метод быстрой сортировки на каждой из частей.

Определяем ключевые элементы: в левой части – это пятерка. При перемещении значений она изменит свой индекс. **Важно именно ключевое значение, а не его индекс.**

3	5	4	4	6	7	8
3	4	4	5	6	7	8

Снова применяем быструю сортировку:

3	4	4	5	6	7	8
3	4	4	5	6	7	8

И еще раз:

3	4	4	5	6	7	8
3	4	4	5	6	7	8

Осталось одно неотсортированное значение. Поскольку мы знаем, что все остальное уже отсортировано, алгоритм завершает работу.



Программная реализация:

```
import random

def qsort_inplace(lst, start=0, end=None):
    """
    Отсортировать список методом быстрой сортировки
    """
    def subpart(lst, start, end, pivot_index):
        lst[start], lst[pivot_index] = lst[pivot_index], lst[start]
        pivot = lst[start]
        x = start + 1
        y = start + 1

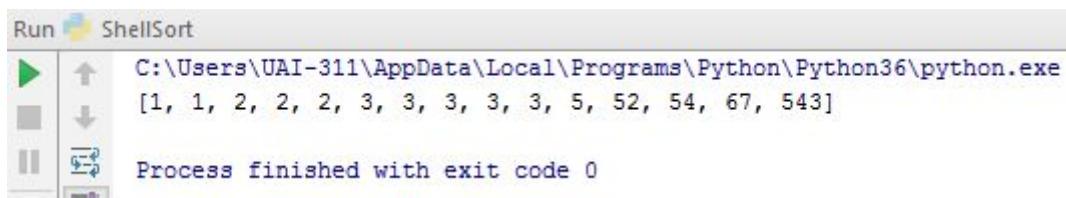
        while y <= end:
            if lst[y] <= pivot:
                lst[y], lst[x] = lst[x], lst[y]
                x += 1
            y += 1

        lst[start], lst[x - 1] = lst[x - 1], lst[start]
        return x - 1

    if end is None:
        end = len(lst) - 1
    if end - start < 1:
        return

    pivot_index = random.randint(start, end)
    x = subpart(lst, start, end, pivot_index)
    qsort_inplace(lst, start, x - 1)
    qsort_inplace(lst, x + 1, end)

ary = [54, 1, 2, 3, 52, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
print(qsort_inplace(ary))
```



## Сортировка Шелла

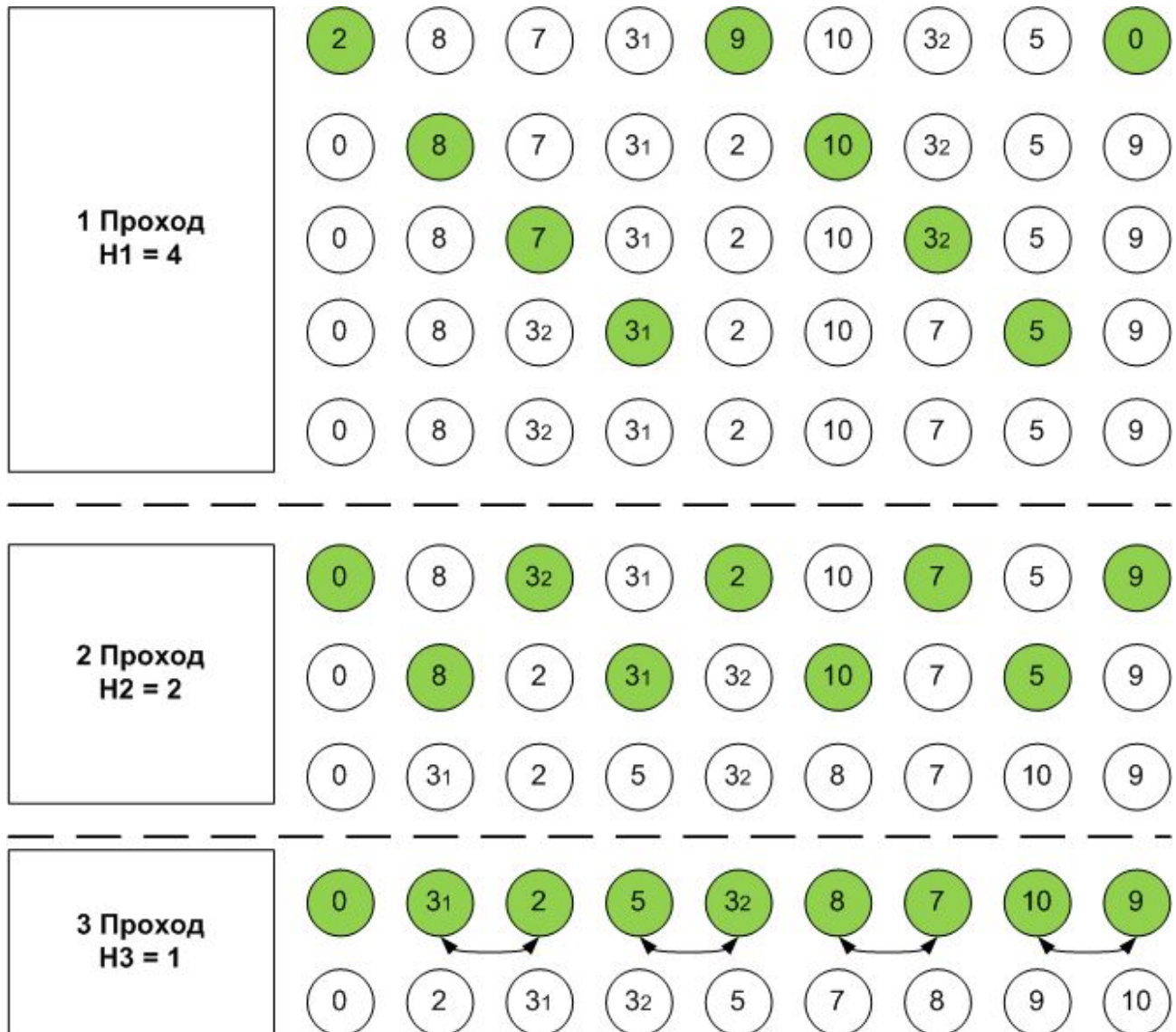
Идея алгоритма – в разделении массива на группы и в сравнениях. Группы определяются по расстоянию между элементами.

Алгоритм сортировки:

1. Берем расстояние, равное  $N/2$ , где  $N$  – это размер массива;
2. Каждая группа включает в себя два элемента, расположенных друг от друга на расстоянии  $N/2$ ;

3. Элементы сравниваются между собой, и при необходимости меняются местами;
4. Меняем расстояние: сокращаем на 2. Количество групп уменьшается;
5. Расстояние с каждым проходом сокращается;
6. Проходы по массиву завершаются при  $N=1$ .

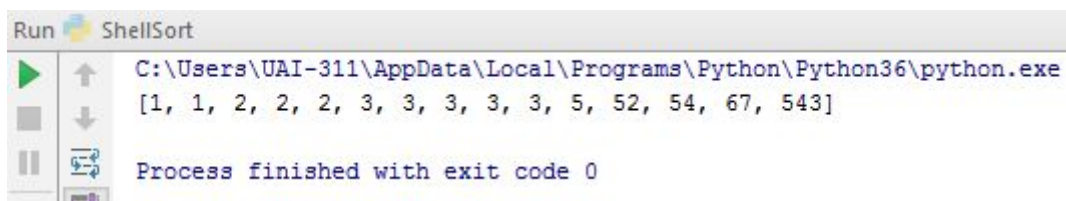
Схема прохода методом Шелла:



Программная реализация:

```
def shellsort(a):
    def new_increment(a):
        i = int(len(a) / 2)
        yield i
        while i != 1:
            if i == 2:
                i = 1
            else:
                i = int(round(i/2.2))
            yield i
    for increment in new_increment(a):
        for i in range(increment, len(a)):
            for j in range(i, increment-1, -increment):
                if a[j - increment] < a[j]:
                    break
                a[j], a[j - increment] = a[j - increment], a[j]
    return a

ary = [54, 1, 2, 3, 52, 3, 1, 2, 3, 5, 3, 67, 3, 2, 543]
print(shellsort(ary))
```



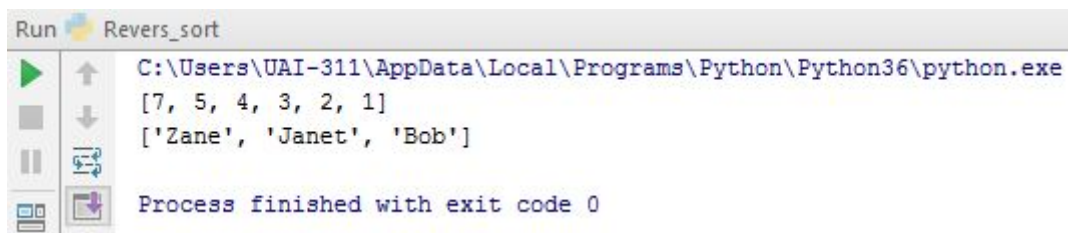
## Обратная сортировка

Функция **sorted()** намного упрощает сортировку в обратном порядке. Она принимает опциональный параметр **reverse**, который действует по строгой логике.

Программная реализация:

```
data = [3, 2, 5, 4, 7, 1]
a = sorted(data, reverse=True)
print(a) # [7, 5, 4, 3, 2, 1]

data = ('Zane', 'Bob', 'Janet')
b = sorted(data, reverse=True)
print(b) # ['Zane', 'Janet', 'Bob']
```



## Сортировка с использованием функции attrgetter

Сортировка с использованием функции **attrgetter** упорядочивает данные по определенному признаку.

```
from operator import attrgetter
```

Мы определяем имя атрибута для выборки, после чего **attrgetter** генерирует функцию, которая принимает объект и возвращает определенный атрибут из этого объекта.

```
from operator import attrgetter

getName = attrgetter('name')
result = getName(jack)
print(result) # 'jack'
```

Таким образом, **attrgetter(name)** возвращает функцию, которая ведет себя так же, как определенная ранее нашей функцией **byName\_key()**:

```
result = sorted(people, key = attrgetter('name'))
print(result) # [<name: Adam, age: 43>, <name: Becky, age: 11>, <name: Jack, age: 19>]
```

Функция **attrgetter(age)** возвращает функцию, которая ведет себя так же, как определенная ранее нашей функцией **byAge\_key()**:

```
result = sorted(people, key = attrgetter('age'))
print(result) # [<name: Becky, age: 11>, <name: Jack, age: 19>, <name: Adam, age: 43>]
```

## Сортировка сложных структур с использованием ключа

Функция **sorted()** будет принимать ключ в качестве параметра. Сам он должен быть функцией, принимающей один параметр. Затем ключ используется функцией **sorted()**, чтобы определить значение для дальнейшей сортировки.

Опишем класс **Person** с такими атрибутами, как имя и возраст:

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return "'{}' {}".format(self.name, self.age)
```

Создаем список людей:

```
jack = Person('Jack', 19)
adam = Person('Adam', 43)
becky = Person('Becky', 11)
people = [jack, adam, becky]
```

Сама по себе функция **sorted()** не знает, что делать со списком людей:

```
jack = Person('Jack', 19)
adam = Person('Adam', 43)
becky = Person('Becky', 11)
people = [jack, adam, becky]

a = sorted(people)
print(a) # [<name: Jack, age: 19>, <name: Adam, age: 43>, <name: Becky, age: 11>]
```

Мы можем непосредственно сообщить функции **sorted()**, по какому атрибуту сортировать список, указав используемый ключ. Определим ключ в следующем примере:

```
def byName_key(person):
    return person.name
```

Функция ключа будет принимать один аргумент и выдавать значение, на котором базируется сортировка. Функция **sorted()** должна вызвать функцию определения ключа и использовать значение выдачи при сортировке списка.

```
a = sorted(people, key = byName_key)
print(a) # [<name: Adam, age: 43>, <name: Becky, age: 11>, <name: Jack, age: 19>]
```

**Важно:**

- все вызовы производим по ссылкам;
- **sorted()** будет использовать функцию **key()**, вызывая ее в каждом элементе итерируемой.

## Оценка алгоритмов сортировки

При всем разнообразии алгоритмов так и не определен один универсальный, подходящий для всех вариантов сортировки.

Зная характеристики входных данных, можно выбрать метод, работающий оптимально. Для этого необходимо знать параметры, по которым будет производиться оценка алгоритмов:

- Время сортировки – основной параметр, характеризующий быстроту алгоритма;
- Память – ряд алгоритмов сортировки требуют выделения дополнительной памяти под временное хранение данных;
- Устойчивость – параметр, который отвечает за то, что сортировка не изменит взаимного расположения равных элементов;
- Естественность поведения – параметр, указывающий на эффективность метода при обработке уже отсортированных (или частично отсортированных) данных.

Рассмотрим классификацию алгоритмов сортировки по сфере применения:

- Внутренняя сортировка – это алгоритм, который в процессе упорядочивания данных использует только оперативную память компьютера. Применяется во всех случаях, за исключением однопроходного считывания данных и однопроходной записи отсортированных данных;
- Внешняя сортировка – это алгоритм, который при упорядочивании данных использует внешнюю память (как правило, жесткие диски). Разработана для обработки больших списков данных, которые не помещаются в оперативную память.

Внутренняя сортировка является базовой для любого алгоритма внешней сортировки. Отдельные части массива данных сортируются в оперативной памяти и с помощью специального алгоритма сцепляются в один массив, упорядоченный по ключу.

Внутренняя сортировка значительно эффективнее внешней, так как на обращение к оперативной памяти затрачивается намного меньше времени, чем к носителям.

## Практическое задание

1. Отсортируйте по убыванию методом пузырька одномерный целочисленный массив, заданный случайными числами на промежутке  $[-100; 100]$ . Выведите на экран исходный и отсортированный массивы. Сортировка должна быть реализована в виде функции. По возможности доработайте алгоритм (сделайте его умнее).
2. Отсортируйте по возрастанию методом слияния одномерный вещественный массив, заданный случайными числами на промежутке  $[0; 50]$ . Выведите на экран исходный и отсортированный массивы.
3. Массив размером  $2m + 1$ , где  $m$  – натуральное число, заполнен случайным образом. Найдите в массиве медиану. Медианой называется элемент ряда, делящий его на две равные части: в одной находятся элементы, которые не меньше медианы, в другой – не больше медианы. Задачу можно решить без сортировки исходного массива. Но если это слишком сложно, то используйте метод сортировки, который не рассматривался на уроках.

## Используемая литература

1. <https://www.python.org>
2. <https://habrahabr.ru/post/133996/>
3. Марк Лутц. Изучаем Python, 4-е издание.



