

Урок 7. Алгоритмы сортировки

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc88744810)

[СОРТИРОВКА ВЫБОРОМ 4](#_Toc88744811)

[Описание 4](#_Toc88744812)

[Шаги 4](#_Toc88744813)

[Видео 4](#_Toc88744814)

[Программный код 5](#_Toc88744815)

[СОРТИРОВКА ВСТАВКАМИ 6](#_Toc88744816)

[Описание 6](#_Toc88744817)

[Шаги 6](#_Toc88744818)

[Видео 6](#_Toc88744819)

[Программный код 6](#_Toc88744820)

[СОРТИРОВКА ПУЗЫРЬКОМ 8](#_Toc88744821)

[Описание 8](#_Toc88744822)

[Шаги 8](#_Toc88744823)

[Видео 9](#_Toc88744824)

[Программный код 9](#_Toc88744825)

[СОРТИРОВКА ШЕЙКЕРОМ 10](#_Toc88744826)

[Описание 10](#_Toc88744827)

[Шаги 10](#_Toc88744828)

[Видео 11](#_Toc88744829)

[Программный код 11](#_Toc88744830)

[СОРТИРОВКА СЛИЯНИЕМ 12](#_Toc88744831)

[Описание 12](#_Toc88744832)

[Шаги 12](#_Toc88744833)

[Видео 12](#_Toc88744834)

[Программный код 12](#_Toc88744835)

[БЫСТРАЯ СОРТИРОВКА ХОАРА 15](#_Toc88744836)

[Описание 15](#_Toc88744837)

[Шаги 15](#_Toc88744838)

[Видео 15](#_Toc88744839)

[Программный код 15](#_Toc88744840)

[СТАНДАРТНАЯ СОРТИРОВКА 17](#_Toc88744841)

[ВЫВОДЫ 19](#_Toc88744842)



# СОРТИРОВКА ВЫБОРОМ

## Описание

Выполняется поиск минимального элемента и его рокировка с первым элементом. При этом первый элемент массива оказывается уже отсортированным, далее выполняется поиск следующего наименьшего элемент из неотсортированных и его рокировка со вторым элементом. Процесс повторяется до тех пор, пока в неотсортированном списке не останется один элемент. Левая часть отсортированных элементов увеличивается, а правая неотсортированных – уменьшается.

## Шаги

1. Найти минимальный элемент массива и поставить его на первое место.
2. Начать новый проход по массиву со второго элемента и вновь найти минимальный элемент уже из оставшихся, поставить найденный минимальный элемент на вторую позицию.
3. Повторить те же действия, но уже начиная с третьего элемента массива.

и т.д. пока все элементы массива не будут отсортированы.

## Видео

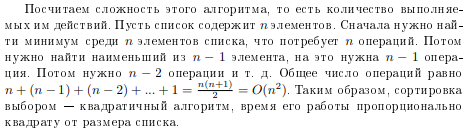
<https://www.youtube.com/watch?v=Ns4TPTC8whw>

## Программный код

**Листинг 1. Алгоритм сортировки выбором (selection\_sort.py)**

|  |
| --- |
| *"""Сортировка выбором"""* **from** random **import** randint **from** timeit **import** timeit   **def** selection\_sort(lst\_obj):  **for** i **in** range(len(lst\_obj)):  idx\_min = i  **for** j **in** range(i + 1, len(lst\_obj)):  **if** lst\_obj[j] < lst\_obj[idx\_min]:  idx\_min = j   tmp = lst\_obj[idx\_min]  lst\_obj[idx\_min] = lst\_obj[i]  lst\_obj[i] = tmp   **return** lst\_obj   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)] print(selection\_sort(orig\_list[:]))  *# замеры 10* print(  timeit(  **"selection\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"selection\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"selection\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.008928599999999995 0.5506933 44.9177326 """** |

**Сложность:**

****

# СОРТИРОВКА ВСТАВКАМИ

## Описание

Очень простой алгоритм, в основе которого проходы по массиву, на каждом из которых берется элемент и определяется его позиция для вставки. Похоже на то, что мы делаем с деньгами в своем кошельке, если хотим упорядочить банкноты. Получается массив разделяется на две части: левую (отсортированную) и правую (неотсортированную). Из правой части на каждом шаге извлекается элемент и вставляется в правильное положение левой части. Левая часть увеличивается, правая – сокращается.

## Шаги

1. Из массива последовательно берется каждый элемент;
2. Вставляется в отсортированную часть в нужную позицию (например, в начале массива).

## Видео

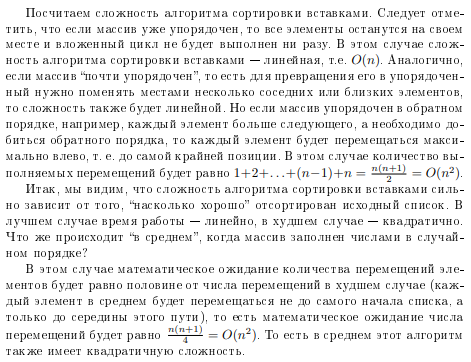
<https://www.youtube.com/watch?v=ROalU379l3U>

## Программный код

**Листинг 2. Алгоритм сортировки вставками (insertion\_sort.py)**

|  |
| --- |
| *"""Сортировка вставками"""* **from** random **import** randint **from** timeit **import** timeit   **def** insertion\_sort(lst\_obj):  **for** i **in** range(len(lst\_obj)):  v = lst\_obj[i]  j = i   **while** (lst\_obj[j-1] > v) **and** (j > 0):   lst\_obj[j] = lst\_obj[j-1]  j = j - 1   lst\_obj[j] = v  **return** lst\_obj   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"insertion\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"insertion\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"insertion\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.006171599999999999 0.5522702 52.527727 """** |

**Сложность:**



# СОРТИРОВКА ПУЗЫРЬКОМ

## Описание

В основе алгоритма многократный проход по массиву. На каждом проходе наибольшее значение из неотсортированных перемещается в конец массива (всплывает как пузырек). Указанные действия повторяются до получения полностью отсортированного массива. Последний проход будет выполнен уже по отсортированному массиву.

## Шаги

1. Идем по массиву слева направо. Если текущий элемент больше последующего, меняем их местами.
2. Повторяем замены до тех пор, пока массив оказывается полностью отсортированным.
3. В итоге элемент с самым большим значением оказывается в конце массива (всплывает, как пузырек).

## Видео

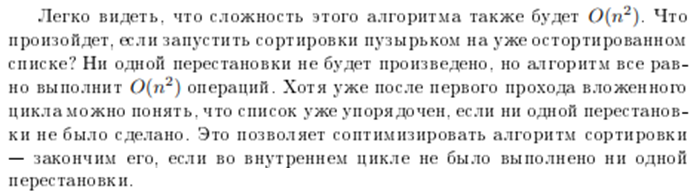
<https://www.youtube.com/watch?v=lyZQPjUT5B4>

## Программный код

**Листинг 3. Алгоритм сортировки пузырьком (bubble\_sort.py)**

|  |
| --- |
| *"""Сортировка пузырьком"""* **from** random **import** randint **from** timeit **import** timeit   **def** bubble\_sort(lst\_obj):  n = 1  **while** n < len(lst\_obj):  **for** i **in** range(len(lst\_obj)-n):  **if** lst\_obj[i] > lst\_obj[i+1]:  lst\_obj[i], lst\_obj[i+1] = lst\_obj[i+1], lst\_obj[i]  n += 1  **return** lst\_obj   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"bubble\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"bubble\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"bubble\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.009131200000000006 0.7685486 106.773521 """** |

**Сложность:**



# СОРТИРОВКА ШЕЙКЕРОМ

## Описание

Еще одно название этой сортировки – двунаправленная. Это модифицированный пузырек: проходы осуществляются не только классически (слева направо), а сперва слева направо, затем справа налево. Перестановки осуществляются аналогично пузырьку: два соседних элемента при необходимости меняются местами.

Этот алгоритм был предложен в качестве замены пузырьку, ввиду его неэффективности: после очередного прохода по массиву поиск начинается с начала и алгоритм еще раз движется от начала к концу. В основе идеи шейкера – не перескакивать к началу, а сортировать в двух направлениях. После каждого пробега начало и конец неотсортированной части будут сдвигаться.

## Шаги

1. Обход массива осуществляется в двух направлениях поочередно.
2. Диапазон сортировки постепенно сужается.
3. За один проход в конец массива «всплывает» максимальный элемент из диапазона.
4. А за следующий проход в начало массива минимальный элемент (если сортировка ведется по возрастанию).
5. Эти элемент можно больше не анализировать и таким образом диапазон сужается с двух сторон.

## Видео

<https://www.youtube.com/watch?v=ahi0pDTAjps>

## Программный код

**Листинг 4. Алгоритм шейкерной сортировки (shaker\_sort.py)**

|  |
| --- |
| *"""Шейкерная сортировка"""* **from** random **import** randint **from** timeit **import** timeit   **def** cocktail\_sort(lst\_obj):  left = 0  right = len(lst\_obj) - 1  **while** left <= right:  **for** i **in** range(left, right):  **if** lst\_obj[i] > lst\_obj[i+1]:  lst\_obj[i], lst\_obj[i+1] = lst\_obj[i+1], lst\_obj[i]  right -= 1  **for** i **in** range(right, left, -1):  **if** lst\_obj[i-1] > lst\_obj[i]:  lst\_obj[i], lst\_obj[i-1] = lst\_obj[i-1], lst\_obj[i]  left += 1  **return** lst\_obj   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"cocktail\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"cocktail\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"cocktail\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.009604599999999991 0.7762804999999999 97.97526020000001 """** |

# СОРТИРОВКА СЛИЯНИЕМ

## Описание

Рекурсивный алгоритм, постоянно разбивающий массив пополам. Если количество элементов в массиве равно 1, значит он уже отсортирован (это базовый случай). Если в массиве находится более одного элемент, такой массив разбивается и для каждой части рекурсивно вызывается слияние каждой части. Когда обе части отсортированы, выполняется непосредственно слияние, т.е. процесс комбинирования двух меньших сортированных списков в один новый, тоже отсортированный. Отметим, что в списке может быть и нечетное число элементов, для алгоритма это не принципиальный момент, поскольку длины могут отличаться не более чем на единицу.

## Шаги

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.
4. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

## Видео

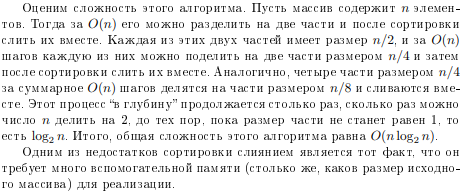
<https://www.youtube.com/watch?v=XaqR3G_NVoo>

## Программный код

**Листинг 5. Алгоритм сортировки слиянием (merge\_sort.py)**

|  |
| --- |
| *"""Сортировка слиянием"""* **from** random **import** randint **from** timeit **import** timeit   **def** merge(left\_lst, right\_lst):  *"""Выполняет слияние подсписков"""* sorted\_lst = []  left\_lst\_index = right\_lst\_index = 0   left\_lst\_length, right\_lst\_length = len(left\_lst), len(right\_lst)   **for** \_ **in** range(left\_lst\_length + right\_lst\_length):  **if** left\_lst\_index < left\_lst\_length **and** \  right\_lst\_index < right\_lst\_length:  *# Сравниваем первые элементы в начале каждого списка  # Если первый элемент левого подсписка меньше,  # добавляем его в отсортированный массив* **if** left\_lst[left\_lst\_index] <= right\_lst[right\_lst\_index]:  sorted\_lst.append(left\_lst[left\_lst\_index])  left\_lst\_index += 1  *# Если первый элемент правого подсписка меньше,  # добавляем его в отсортированный массив* **else**:  sorted\_lst.append(right\_lst[right\_lst\_index])  right\_lst\_index += 1   *# Если достигнут конец левого списка,  # элементы правого списка добавляем в конец результирующего списка* **elif** left\_lst\_index == left\_lst\_length:  sorted\_lst.append(right\_lst[right\_lst\_index])  right\_lst\_index += 1  *# Если достигнут конец правого списка,  # элементы левого списка добавляем в отсортированный массив* **elif** right\_lst\_index == right\_lst\_length:  sorted\_lst.append(left\_lst[left\_lst\_index])  left\_lst\_index += 1  **return** sorted\_lst   **def** merge\_sort(nums):  **if** len(nums) <= 1: *# Базовый случай* **return** nums   mid = len(nums) // 2 *# Ищем середину списка   # Сортируем и объединяем подсписки* left\_list = merge\_sort(nums[:mid])  right\_list = merge\_sort(nums[mid:])   *# Объединяем отсортированные списки в результирующий* **return** merge(left\_list, right\_list)   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"merge\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"merge\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"merge\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.019539102000000003 0.29186973099999997 3.8153470499999997 """** |

**Сложность:**



ВНИМАНИЕ: в представленном алгоритме, в отличие от предыдущих, возвращает новый массив, а не сортирует существующий. Поэтому в данном случае потребуется память для создания нового массива того же размера, и исходный массив.

# БЫСТРАЯ СОРТИРОВКА ХОАРА

## Описание

Во многом схожа с сортировкой слиянием. В основе выбор некоторого элемента, называемого опорным или барьерным. По этому элементу массив разбивается на две части. В первую перемещаются элементы, меньшие или равные опорному. Во вторую – большие или равные опорному. Далее необходимо отсортировать обе части и выполнить их конкатенацию.

## Шаги

1. В массиве случайным образом определяется опорный элемент.
2. Выполняется процедура разбиения массива, перемещающая все элементы, меньшие опорного влево от него, большие – вправо, равные – в третий подмассив.
3. Для двух первых подмассивов рекурсивно повторяется эта же процедура, если в каждом подмассиве не более двух элементов.

## Видео

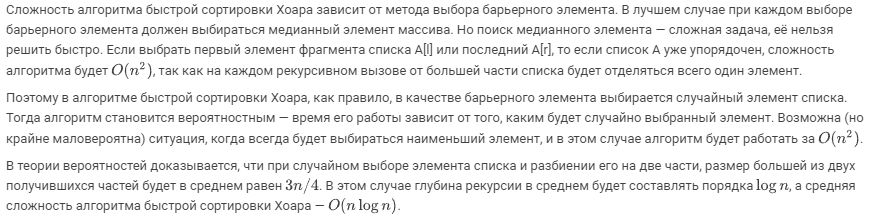
<https://www.youtube.com/watch?v=ywWBy6J5gz8>

## Программный код

**Листинг 6. Алгоритм быстрой сортировки (hoare\_quick\_sort.py)**

|  |
| --- |
| *"""Быстрая сортировка"""* **from** random **import** randint, choice **from** timeit **import** timeit   *# по сути будет одно направление* **def** my\_calc(lst\_obj):  **def** quick\_sort(lst\_obj):  **if** len(lst\_obj) <= 1:  **return** lst\_obj  **else**:  q = choice(lst\_obj)  L = []  M = []  R = []  **for** elem **in** lst\_obj:  **if** elem < q:  L.append(elem)  **elif** elem > q:  R.append(elem)  **else**:  M.append(elem)  **return** quick\_sort(L) + M + quick\_sort(R)  **return** quick\_sort   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"my\_calc(orig\_list[:])"**,  globals=globals(),  number=1000))   orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"my\_calc(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"my\_calc(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.00031299999999999384 0.0006287000000000098 0.0020312999999999998 """** |

**Сложность:**



# СТАНДАРТНАЯ СОРТИРОВКА

Гибридный [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), сочетающий сортировку вставками и сортировку слиянием, опубликованный в 2002 году [Тимом Петерсом](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D0%B5%D1%82%D0%B5%D1%80%D1%81,_%D0%A2%D0%B8%D0%BC&action=edit&redlink=1). В настоящее время Timsort является стандартным алгоритмом сортировки в [Python](https://ru.wikipedia.org/wiki/Python). Основная идея алгоритма в том, что в реальном мире сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. На таких данных Timsort существенно быстрее многих алгоритмов.

Основная идея алгоритма

* По специальному алгоритму входной массив разделяется на подмассивы.
* Каждый подмассив сортируется [сортировкой вставками](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%BE%D0%B9).
* Отсортированные подмассивы собираются в единый массив с помощью модифицированной [сортировки слиянием](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%81%D0%BB%D0%B8%D1%8F%D0%BD%D0%B8%D0%B5%D0%BC).

**Листинг 7. Стандартная сортировка (sorted\_sort.py)**

|  |
| --- |
| *"""Стандартная сортировка"""* **from** random **import** randint **from** timeit **import** timeit  **""" Внутри Python использует Timsort – гибридный алгоритм сортировки,  сочетающий сортировку вставками и сортировку слиянием.  Смысл в том, что в реальном мире часто встречаются частично  отсортированные данные, на которых Timsort работает ощутимо  быстрее прочих алгоритмов сортировки. Сложность по времени:  O(n log n) в худшем случае и O(n) – в лучшем.   list.sort() - Сортирует лист, но возвращает None sorted(list) - Сортирует лист и возвращает его """   def** reverse\_sort(lst\_obj):  ordered\_list = sorted(lst\_obj)  **return** ordered\_list   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.0007880000000000109 0.005651299999999998 0.10248260000000001 """** |

**Листинг 8. Стандартная сортировка (sort\_sort.py)**

|  |
| --- |
| *"""Стандартная сортировка"""* **from** random **import** randint **from** timeit **import** timeit   **def** reverse\_sort(lst\_obj):  lst\_obj.sort()  **return** orig\_list   orig\_list = [randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.00039260000000000683 0.004685000000000009 0.0959738 """** |

# ВЫВОДЫ

* Пузырьковая сортировка — самый медленный из всех алгоритмов. Возможно, он будет полезен как введение в тему алгоритмов сортировки, но не подходит для практического использования.
* Быстрая сортировка хорошо оправдывает своё название, почти в два раза быстрее, чем сортировка слиянием, и не требуется дополнительная память для результирующего массива.
* Вы познакомились с семью различными алгоритмами сортировок и их реализациями на Python. Масштаб сравнения и количество перестановок, которые выполняет алгоритм вместе со средой выполнения кода, будут определяющими факторами в производительности. В реальных приложениях Python рекомендуется использовать встроенные функции сортировки, поскольку они реализованы именно для удобства разработчика.