***https://d2xzmw6cctk25h.cloudfront.net/record/127187/attachment/e5998e6336ffec9cd4a6fe3dc83660f8.mp4***

***Алгоритмы сортировки***

Сортировка выбором

Описание:

Создавать новый список для отсортированных элементов не требуется. Им является крайняя левая часть массива. 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.

Шаги:

1. Найти минимальный элемент массива и поставить его на первое место.
2. Начать новый проход массива, со второго элемента и сравнить его со всеми оставшимися.
3. Найти наименьший и поменять его местами со вторым элементом и т.д…N-1 раз.

(N – размер массива).

Таким образом, во внутреннем цикле осуществляется поиск наименьшего элемента, запоминание его значения и позиции в массиве. После, этот элемент меняется местами с текущим элементом массива.

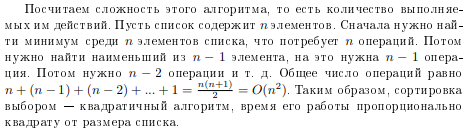
Видео:

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

**Листинг 1. Алгоритм сортировки выбором.py**

|  |
| --- |
| *"""Сортировка выбором"""* **import** timeit **import** random   **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 = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"selection\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"selection\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.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:

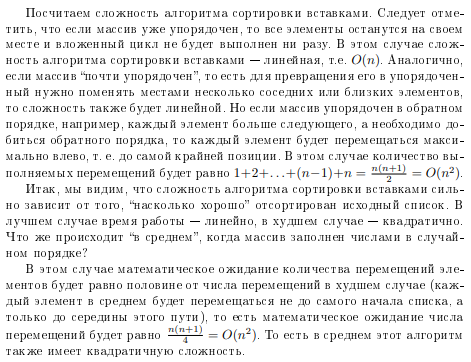
1. Из массива последовательно берется каждый элемент;
2. Вставляется в отсортированную часть (например, в начале массива).

Видео:

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

**Листинг 2. Алгоритм сортировки вставками.py**

|  |
| --- |
| *"""Сортировка вставками"""* **import** timeit **import** random   **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 = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"insertion\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"insertion\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.timeit(  **"insertion\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.006171599999999999 0.5522702 52.527727 """** |

**Сложность:**

## Сортировка пузырьком

Описание:

В основе алгоритма проход многократный проход по массиву. На каждом проходе наибольшее значение из неотсортированных перемещается в конец массива. А именно на каждом проходе выполняется последовательное сравнение пар элементов массива и, если они не соответствуют выбранному порядку (вы решили отсортировать пузырьком по возрастанию, хотя можно и по убыванию), элементы меняются местами. Если порядок двух элементов верный, то ничего не происходит. По итогам первого прохода наибольший элемент расположится в конце, т.е. всплывет, 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 предыдущего, 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:

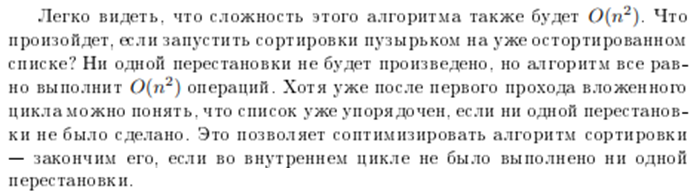
<https://www.youtube.com/watch?v=lyZQPjUT5B4>

ВНИМАНИЕ: пузырек можно оптимизировать. Нужно знать, когда сделать останов такого алгоритма. Точкой останова будет момент, в котором массив окажется полностью отсортированным. Для останова алгоритма по завершению сортировки необходимо создать переменную-флаг. Когда пара элементов меняется позициями, определяем для флага значение True. Это будет означать, что сортировку мы продолжаем. Если перестановок нет, флаг останется в исходном инициализированном значении False, т.е. массив уже будет отсортированным и алгоритм можно остановить.

**Листинг 3. Алгоритм сортировки пузырьком.py**

|  |
| --- |
| *"""Сортировка пузырьком"""* **import** timeit **import** random   **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 = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"bubble\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"bubble\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.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. При 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>

**Листинг 4. Алгоритм шейкерной сортировки.py**

|  |
| --- |
| *"""Шейкерная сортировка"""* **import** random **import** timeit  **''' разновидность пузырьковой сортировки. Отличается тем, что просмотры элементов выполняются один за другим в противоположных направлениях, при этом большие элементы стремятся к концу массива, а маленькие - к началу. '''   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 = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"cocktail\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"cocktail\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.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).

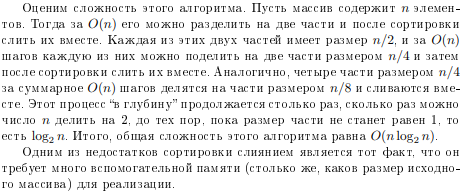
Видео:

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

**Листинг 5. Алгоритм сортировки слиянием.py**

|  |
| --- |
| *"""Сортировка слиянием"""* **import** timeit **import** random   **def** merge\_sort(lst\_obj):  **if** len(lst\_obj) > 1:  center = len(lst\_obj) // 2  left = lst\_obj[:center]  right = lst\_obj[center:]   merge\_sort(left)  merge\_sort(right)   *# перестали делить  # выполняем слияние* i, j, k = 0, 0, 0   **while** i < len(left) **and** j < len(right):  **if** left[i] < right[j]:  lst\_obj[k] = left[i]  i += 1  **else**:  lst\_obj[k] = right[j]  j += 1  k += 1   **while** i < len(left):  lst\_obj[k] = left[i]  i += 1  k += 1   **while** j < len(right):  lst\_obj[k] = right[j]  j += 1  k += 1  **return** lst\_obj   orig\_list = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"merge\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"merge\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.timeit(  **"merge\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.017428799999999994 0.27980170000000004 3.89777 """** |

## **Сложность:**



ВНИМАНИЕ: в представленном алгоритме, в отличие от предыдущих, возвращает новый массив, а не сортирует существующий. Поэтому в данному случае потребуется больше памяти 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:

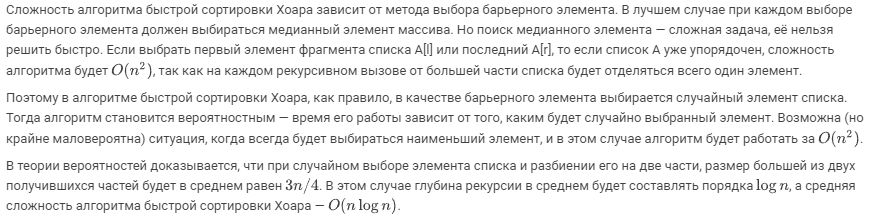
1. В массиве случайным образом определяется опорный элемент.
2. Выполняется процедура разбиения массива, перемещающая все элементы, меньшие опорного влево от него, большие – вправо, равные – в третий подмассив.
3. Для двух первых подмассивов рекурсивно повторяется эта же процедура, если в каждом подмассиве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

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

**Листинг 6. Алгоритм быстрой сортировки.py**

|  |
| --- |
| *"""Быстрая сортировка"""* **import** timeit **import** random   **def** my\_calc(lst\_obj):  **def** quick\_sort(lst\_obj):  **if** len(lst\_obj) <= 1:  **return** lst\_obj  **else**:  q = random.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 = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"my\_calc(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"my\_calc(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.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[[1]](https://ru.wikipedia.org/wiki/Timsort#cite_note-1) сортировки в [Python](https://ru.wikipedia.org/wiki/Python" \o "Python), [OpenJDK](https://ru.wikipedia.org/wiki/OpenJDK" \o "OpenJDK) 7[[2]](https://ru.wikipedia.org/wiki/Timsort#cite_note-2) и реализован в [Android](https://ru.wikipedia.org/wiki/Android" \o "Android) JDK 1.5[[3]](https://ru.wikipedia.org/wiki/Timsort#cite_note-3). 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 1.py**

|  |
| --- |
| *"""Стандартная сортировка"""* **import** timeit **import** random  **""" 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 = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  **""" 0.0007880000000000109 0.005651299999999998 0.10248260000000001 """** |

**Листинг 8. Стандартная сортировка 2.py**

|  |
| --- |
| *"""Стандартная сортировка"""* **import** timeit **import** random   **def** reverse\_sort(lst\_obj):  lst\_obj.sort()  **return** orig\_list   orig\_list = [random.randint(-100, 100) **for** \_ **in** range(10)]  *# замеры 10* print(  timeit.timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(  timeit.timeit(  **"reverse\_sort(orig\_list[:])"**,  globals=globals(),  number=1000))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(  timeit.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 дополнительное место для результирующего массива.
* Сортировка вставками выполняет меньше сравнений, чем сортировка выборкой и в реальности должна быть производительнее, но в данном эксперименте она выполняется немного медленней. Сортировка вставками делает гораздо больше обменов элементами. Если эти обмены занимают намного больше времени, чем сравнение самих элементов, то такой результат вполне закономерен.
* Вы познакомились с шестью 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.