# Алгоритмы сортировки

## Алгоритм сортировки выбором

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.

## **Алгоритм**

На практике не нужно создавать новый список для отсортированных элементов. В качестве него используется крайняя левая часть списка. Находится наименьший элемент и меняется с первым местами.

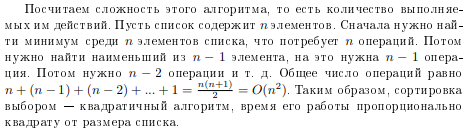
Теперь, когда нам известно, что первый элемент списка отсортирован, находим наименьший элемент из оставшихся и меняем местами со вторым. Повторяем это до тех пор, пока не останется последний элемент в списке.

**Листинг 1. task\_1.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)"**, \  setup=**"from \_\_main\_\_ import selection\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"selection\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import selection\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"selection\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import selection\_sort, orig\_list"**, number=1)) |

Сортировка выбором – это некий гибрид между пузырьковой и сортировкой вставками. Как и сортировка пузырьком, этот алгоритм проходит по массиву раз за разом, перемещая одно значение на правильную позицию. Однако, в отличие от пузырьковой сортировки, он выбирает наименьшее неотсортированное значение вместо наибольшего. Как и при сортировке вставками, упорядоченная часть массива расположена в начале, в то время как в пузырьковой сортировке она находится в конце.

**Сложность:**

****

### **Время сортировки**

Затраты времени на сортировку выборкой в среднем составляют O(n²), где n — количество элементов списка.

## 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.

Основные шаги алгоритма:

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

## **Сортировка вставками**

Как и сортировка выборкой, этот алгоритм сегментирует список на две части: отсортированную и неотсортированную. Алгоритм перебирает второй сегмент и вставляет текущий элемент в правильную позицию первого сегмента.

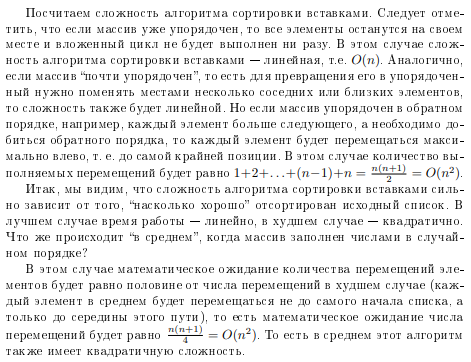
## **Алгоритм**

Предполагается, что первый элемент списка отсортирован. Переходим к следующему элементу, обозначим его х. Если х больше первого, оставляем его на своём месте. Если он меньше, копируем его на вторую позицию, а х устанавливаем, как первый элемент.

Переходя к другим элементам несортированного сегмента, перемещаем более крупные элементы в отсортированном сегменте вверх по списку, пока не встретим элемент меньше x или не дойдём до конца списка. В первом случае x помещается на правильную позицию.

**Листинг 2. task\_2.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)"**, \  setup=**"from \_\_main\_\_ import insertion\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"insertion\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import insertion\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"insertion\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import insertion\_sort, orig\_list"**, number=1)) |

**Сложность:**

### **Время сортировки**

Время сортировки вставками в среднем равно O(n²), где n — количество элементов списка.

## Алгоритм сортировки пузырьком

Сортировка пузырьком — это самый простой алгоритм сортировки. Он проходит по массиву несколько раз, на каждом этапе перемещая самое большое значение из неотсортированных в конец массива.

Суть алгоритма в том, что совершается несколько проходов по массиву. При проходе последовательно сравниваются пары элементов в массиве и в случае несоответствия выбранному порядку меняются местами. Если пары элементов находятся в верном порядке, то ничего не происходит. В результате первого прохода максимальный элемент окажется в конце, то есть всплывет словно пузырек. Затем все повторяется до того момента пока весь массив не будет отсортирован. Последний проход будет по отсортированному массиву.

## **Пузырьковая сортировка**

Этот простой алгоритм выполняет итерации по списку, сравнивая элементы попарно и меняя их местами, пока более крупные элементы не «всплывут» в начало списка, а более мелкие не останутся на «дне».

## **Алгоритм**

Сначала сравниваются первые два элемента списка. Если первый элемент больше, они меняются местами. Если они уже в нужном порядке, оставляем их как есть. Затем переходим к следующей паре элементов, сравниваем их значения и меняем местами при необходимости. Этот процесс продолжается до последней пары элементов в списке.

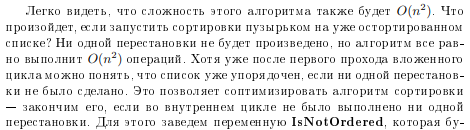
При достижении конца списка процесс повторяется заново для каждого элемента. Это крайне неэффективно, если в массиве нужно сделать, например, только один обмен. Алгоритм повторяется n² раз, даже если список уже отсортирован.

Для оптимизации алгоритма нужно знать, когда его остановить, то есть когда список отсортирован.

Чтобы остановить алгоритм по окончании сортировки, нужно ввести переменную-флаг. Когда значения меняются местами, устанавливаем флаг в значение True, чтобы повторить процесс сортировки. Если перестановок не произошло, флаг остаётся False и алгоритм останавливается.

**Листинг 3. task\_3.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)"**, \  setup=**"from \_\_main\_\_ import bubble\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"bubble\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import bubble\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"bubble\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import bubble\_sort, orig\_list"**, number=1)) |

**Сложность:**

### **Время сортировки**

Если взять самый худший случай (изначально список отсортирован по убыванию), затраты времени будут равны O(n²), где n — количество элементов списка.

## Шейкерная сортировка

Другое название – двунаправленная пузырьковая сортировка. Альтернативная версия классической пузырьковой сортировки. Модификация заключается в двунаправленности подхода: алгоритм работает не классически (слева направо), а сначала – слева направо, а потом справа налево.

При этом перестановка выполняется аналогично пузырьковому алгоритму, т.е. два соседних элемента при необходимости меняются местами. Механизм алгоритма:

Требуется упорядочить массив по возрастанию. Обозначим каждый пройденный путь от начала до конца последовательности через Wi, где i-номер пути, а обратный путь (от конца к началу) через Wj, где j – номер пути.

Тогда после выполнения Wi, один из элементов будет помещен в позицию справа, как наибольший из еще неотсортированных элементов, а после выполнения – Wj, как наименьший из неотсортированных, переместится в некоторую позицию слева. Так, например, после выполнения W1 в конце массива окажется элемент, имеющий наибольшее значение, а после – W1 в начало отправится элемент с наименьшим значением.

Почему пузырьковая сортировка не очень эффективна:

После пробега по массиву поиск перескакивает на начало и снова движется от начала к концу. Плюс шейкерной сортировки в том, чтобы не перескакивать к началу, а просто сортировать в двух направлениях. При этом после каждого пробега НАЧАЛО и КОНЕЦ неотсортированной части массива будут сдвигаться.

**Листинг 4. task\_4.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)"**, \  setup=**"from \_\_main\_\_ import cocktail\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"cocktail\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import cocktail\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"cocktail\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import cocktail\_sort, orig\_list"**, number=1)) |

Шейкерная сортировка работает немного быстрее чем пузырьковая, поскольку по массиву в нужных направлениях попеременно мигрируют и максимумы, и минимумы. Улучшения, как говорится, налицо.

## Алгоритм сортировки слиянием

**Сортировка слиянием** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *merge sort*) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи. Алгоритм был изобретён [Джоном фон Нейманом](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D0%BD_%D0%9D%D0%B5%D0%B9%D0%BC%D0%B0%D0%BD,_%D0%94%D0%B6%D0%BE%D0%BD) в [1945 году](http://ru.wikipedia.org/wiki/1945_%D0%B3%D0%BE%D0%B4).  
  
Для решения задачи сортировки эти три этапа выглядят так:

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.

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

Это рекурсивный алгоритм, который постоянно разбивает список пополам. Если список пуст или состоит из одного элемента, то он отсортирован по определению (базовый случай). Если в списке больше, чем один элемент, мы разбиваем его и рекурсивно вызываем сортировку слиянием для каждой из половин. После того, как обе они уже отсортированы, выполняется основная операция, называемая **слиянием**. Слияние - это процесс комбинирования двух меньших сортированных списков в один новый, но тоже отсортированный.

Важно отметить, что список может иметь нечётное количество элементов. Для алгоритма это не принципиально, поскольку длины будут различаться максимум на единицу.

## **Сортировка слиянием**

Этот алгоритм относится к алгоритмам «разделяй и властвуй». Он разбивает список на две части, каждую из них он разбивает ещё на две и т. д. Список разбивается пополам, пока не останутся единичные элементы.

Соседние элементы становятся отсортированными парами. Затем эти пары объединяются и сортируются с другими парами. Этот процесс продолжается до тех пор, пока не отсортируются все элементы.

## **Алгоритм**

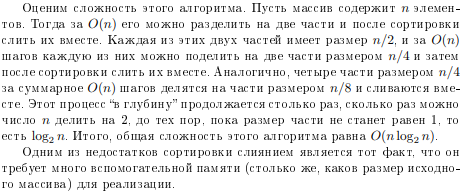
Список рекурсивно разделяется пополам, пока в итоге не получатся списки размером в один элемент. Массив из одного элемента считается упорядоченным. Соседние элементы сравниваются и соединяются вместе. Это происходит до тех пор, пока не получится полный отсортированный список.

Сортировка осуществляется путём сравнения наименьших элементов каждого подмассива. Первые элементы каждого подмассива сравниваются первыми. Наименьший элемент перемещается в результирующий массив. Счётчики результирующего массива и подмассива, откуда был взят элемент, увеличиваются на 1.

**Листинг 5. task\_5.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)"**, \  setup=**"from \_\_main\_\_ import merge\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"merge\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import merge\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"merge\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import merge\_sort, orig\_list"**, number=1)) |

## **Сложность:**



Обратите внимание, что функция merge\_sort(), в отличие от предыдущих алгоритмов, возвращает новый список, а не сортирует существующий. Поэтому такая сортировка требует больше памяти для создания нового списка того же размера, что и входной список.

### **Время сортировки**

В среднем время сортировки слиянием составляет O(n log n).

## Алгоритм быстрой сортировки

Во многом идея быстрой сортировки такая же, как у алгоритма сортировки слиянием. Выберем некоторый элемент q, называемый барьерным элементом. Разобьем массив на две части, переупорядочив его элементы. В первой части соберем элементы, меньшие или равные q, а во второй части — большие или равные q. Теперь достаточно отсортировать обе части, после чего выполнить их конкатенацию безо всякого дополнительного слияния.

## **Быстрая сортировка**

Этот алгоритм также относится к алгоритмам «разделяй и властвуй». Его используют чаще других алгоритмов, описанных в этой статье. При правильной конфигурации он чрезвычайно эффективен и не требует дополнительной памяти, в отличие от сортировки слиянием. Массив разделяется на две части по разные стороны от опорного элемента. В процессе сортировки элементы меньше опорного помещаются перед ним, а равные или большие —позади.

## **Алгоритм**

Быстрая сортировка начинается с разбиения списка и выбора одного из элементов в качестве опорного. А всё остальное передвигаем так, чтобы этот элемент встал на своё место. Все элементы меньше него перемещаются влево, а равные и большие элементы перемещаются вправо.

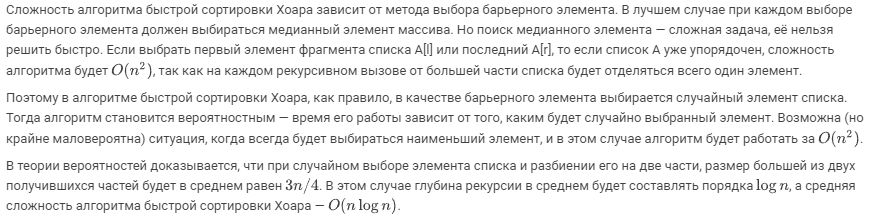
Простая реализация быстрой сортировки Хоара выглядит так:

**Листинг 6. task\_6.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)"**, \  setup=**"from \_\_main\_\_ import my\_calc, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"my\_calc(orig\_list)"**, \  setup=**"from \_\_main\_\_ import my\_calc, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)] *# замеры 1000* print(timeit.timeit(**"my\_calc(orig\_list)"**, \  setup=**"from \_\_main\_\_ import my\_calc, orig\_list"**, number=1)) |

В данном примере в списке L собираются элементы, меньшие q, в списке R — большие q, а в списке M — равные q. Разделение на три списка, а не на два используется для того, чтобы алгоритм не зацикливался, например, в случае, когда в списке остались только равные элементы. Барьерный элемент q выбирается случайным образом из списка при помощи функции choice из модуля random.

## **Сложность:**



### **Время выполнения**

В среднем время выполнения быстрой сортировки составляет O(n log n).

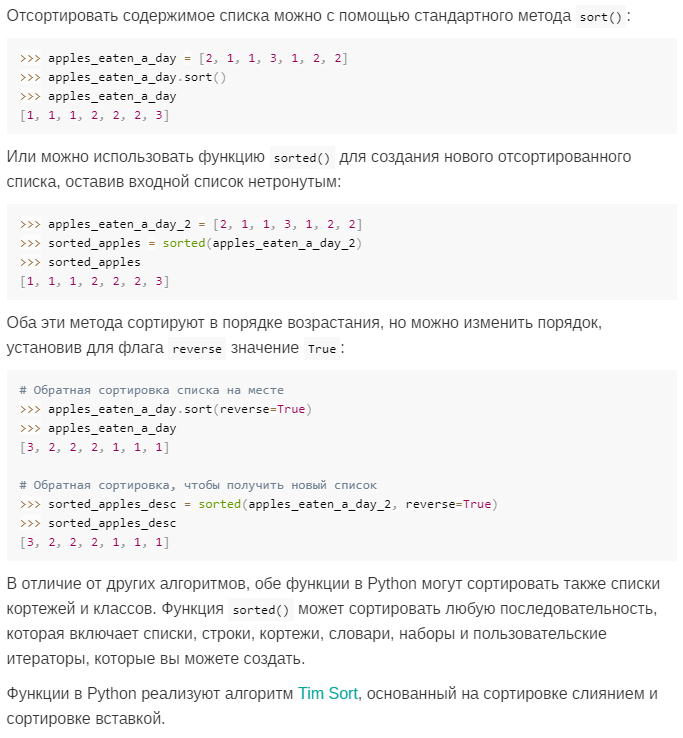
Обратите внимание, что алгоритм быстрой сортировки будет работать медленно, если опорный элемент равен наименьшему или наибольшему элементам списка. При таких условиях, в отличие от сортировок кучей и слиянием, обе из которых имеют в худшем случае время сортировки O(n log n), быстрая сортировка в худшем случае будет выполняться O(n²).

## 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)[[en]](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)). 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 6. task\_6.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)"**, \  setup=**"from \_\_main\_\_ import reverse\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"reverse\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import reverse\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"reverse\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import reverse\_sort, orig\_list"**, number=1)) |

**Листинг 7. task\_7.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)"**, \  setup=**"from \_\_main\_\_ import reverse\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(100)]  *# замеры 100* print(timeit.timeit(**"reverse\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import reverse\_sort, orig\_list"**, number=1))  orig\_list = [random.randint(-100, 100) **for** \_ **in** range(1000)]  *# замеры 1000* print(timeit.timeit(**"reverse\_sort(orig\_list)"**, \  setup=**"from \_\_main\_\_ import reverse\_sort, orig\_list"**, number=1)) |

## 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.