Лекция 5.2. Алгоритмы поиска и сортировки

**Оглавление**

[**1. Алгоритмы поиска 1**](#_Toc166829545)

[**2. Алгоритмы сортировки 4**](#_Toc166829546)

[**3. Стандартные функции поиска и сортировки 23**](#_Toc166829547)

[**Стабильность сортировки и сложные сортировки в Python 26**](#_Toc166829548)

# **Алгоритмы поиска**

**Поиск в ширину**

Поиск в ширину (англ. breadth-first search, BFS) — один из методов обхода графа. Пусть задан граф G = ( V , E ) {\displaystyle G=(V,E)} и выделена исходная вершина s {\displaystyle s}. Алгоритм поиска в ширину систематически обходит все ребра G {\displaystyle G} для «открытия» всех вершин, достижимых из s {\displaystyle s}, вычисляя при этом расстояние (минимальное количество рёбер) от s {\displaystyle s} до каждой достижимой из s {\displaystyle s} вершины. Алгоритм работает как для ориентированных, так и для неориентированных графов.[1]

Поиск в ширину имеет такое название потому, что в процессе обхода мы идём вширь, то есть перед тем как приступить к поиску вершин на расстоянии k + 1 {\displaystyle k+1}, выполняется обход вершин на расстоянии k {\displaystyle k}.

Поиск в ширину является одним из неинформированных алгоритмов поиска[2].

Работа алгоритма

Поиск в ширину работает путём последовательного просмотра отдельных уровней графа, начиная с узла-источника u {\displaystyle u}.

Рассмотрим все рёбра ( u , v ) {\displaystyle (u,v)}, выходящие из узла u {\displaystyle u}. Если очередной узел v {\displaystyle v} является целевым узлом, то поиск завершается; в противном случае узел v {\displaystyle v} добавляется в очередь. После того, как будут проверены все рёбра, выходящие из узла u {\displaystyle u}, из очереди извлекается следующий узел u {\displaystyle u}, и процесс повторяется.

**Поиск в глубину**

Поиск в глубину (англ. Depth-first search, DFS) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

Алгоритм

Пусть задан [граф](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) G = ( V , E ) , где V  — множество вершин графа, E  — множество ребер графа. Предположим, что в начальный момент времени все вершины графа окрашены в *белый* цвет. Выполним следующие действия:

1. Пройдём по всем вершинам v ∈ V .
   * Если вершина v белая, выполним для неё DFS(v).

Процедура DFS (параметр — вершина u ∈ V )

1. Перекрашиваем вершину u в *серый* цвет.
2. Для всякой вершины w , [смежной](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C_%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%BE%D0%B2_%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2) с вершиной u и окрашенной в белый цвет, [рекурсивно](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F) выполняем процедуру DFS(w)[[2]](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B2_%D0%B3%D0%BB%D1%83%D0%B1%D0%B8%D0%BD%D1%83#cite_note-2).
3. Перекрашиваем вершину u в *чёрный* цвет.

Часто используют двухцветные метки — без серого, на 1-м шаге красят сразу в чёрный цвет.

Нерекурсивные варианты

На больших графах поиск в глубину серьёзно нагружает [стек вызовов](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%BE%D0%B2). Если есть риск [переполнения стека](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D1%82%D0%B5%D0%BA%D0%B0), используют нерекурсивные варианты поиска.

**Первый вариант**, простейший, но дающий немалый объём стека — до |E|.

1. Кладём в стек первую вершину.
2. Пока стек не пуст, берём верхнюю вершину, не извлекая.
   1. Если вершина белая…
      1. Красим в серый цвет.
      2. Кладём в стек всех её белых соседок в порядке, обратном порядку обхода (если таковой важен).
   2. Если вершина серая, красим в чёрный и извлекаем.
   3. Если вершина чёрная, просто извлекаем.

Если хватает двухцветных меток…

1. Кладём в стек первую вершину.
2. Пока стек не пуст, извлекаем верхнюю вершину. Если она белая…
   1. Красим в чёрный цвет.
   2. Кладём в стек всех её белых соседок в порядке, обратном порядку обхода.

**Второй вариант**: можно представить стек вызова программно: для каждой из серых вершин в стеке будет храниться её номер u и номер текущей смежной вершины w .

Процедура DFS (параметр — вершина u ∈ V )

1. Кладём в стек пару ( u , ∅ ) . Перекрашиваем вершину u в *серый* цвет.
2. Пока стек не пуст…
   1. Берём верхнюю пару ( v , w ) , не извлекая её из стека.
   2. Находим вершину w ′ , смежную с v и следующую за w .
      1. Если таковой нет, извлекаем ( v , w ) из стека, перекрашиваем вершину v в *чёрный* цвет.
      2. В противном случае присваиваем w := w ′ , прямо в стеке.
         * Если к тому же вершина w ′ белая, кладём на стек пару ( w ′ , ∅ ) , перекрашиваем w ′ в *серый* цвет.

**Третий вариант**: можно в каждой из «серых» вершин держать текущее w и указатель на предыдущую (ту, из которой пришли).

# **Алгоритмы сортировки**

Алгоритм сортировки — это алгоритм для упорядочивания элементов в списке. В случае, когда элемент в списке имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

**Свойства и типы**

* **Устойчивость** — устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами.
* **Естественность поведения** — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.
* **Использование операции сравнения.** Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоёмкость *худшего случая* для этих алгоритмов составляет *O*( n ⋅ log ⁡ n ), но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещё одним важным свойством алгоритма является его сфера применения. Здесь основных типов упорядочения два:

* **Внутренняя сортировка** оперирует массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте без дополнительных затрат.
  + В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.
* **Внешняя сортировка** оперирует запоминающими устройствами большого объёма, но не с произвольным доступом, а последовательным (упорядочение файлов), то есть в данный момент «виден» только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным во внешней памяти производится намного медленнее, чем операции с оперативной памятью.
  + Доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим.
  + Объём данных не позволяет им разместиться в ОЗУ.

Также алгоритмы классифицируются по:

* потребности в дополнительной памяти или её отсутствию
* потребности в знаниях о структуре данных, выходящих за рамки операции сравнения, или отсутствию таковой

Алгоритмы сортировки можно разделить на следующие категории:

* Устойчивой сортировки
* Неустойчивой сортировки
* Непрактичные
* Не основаны на сравнениях

Логичнее всего начать с категории, которая используется только в академических целях: непрактичные алгоритмы. К ним можно отнести bogosort, сортировку перестановкой и гравитационную сортировку. Рассмотрим bogosort.

**Bogosort** (от *амер. комп. жарг.* bogus — неработоспособный, нефункциональный, бесполезный) — неэффективный алгоритм сортировки, используемый только в образовательных целях и противопоставляемый другим, более реалистичным алгоритмам. Bogosort является частным случаем алгоритма Лас-Вегас.

Существуют две версии этого алгоритма: детерминированная версия, которая перебирает все перестановки до тех пор, пока не будет получен отсортированный массив, и случайная версия, которая случайным образом переставляет свои входные данные.

Если этот алгоритм использовать для сортировки колоды карт, то сначала в нём нужно проверить, лежат ли все карты по порядку, и если не лежат, то случайным образом перемешать её, проверить лежат ли теперь все карты по порядку, и повторять процесс, пока не отсортируется колода.

При работе 4-ядерного процессора на частоте 2,4 ГГц (9,6 млрд операций в секунду):

|  |  |
| --- | --- |
| Кол-во элементов | Среднее время |
| 10 | 0,0037 с |
| 11 | 0,045 с |
| 12 | 0,59 с |
| 13 | 8,4 с |
| 14 | 2,1 мин |
| 15 | 33,6 мин |
| 16 | 9,7 ч |
| 17 | 7,29 сут |
| 18 | 139 сут |
| 19 | 7,6 лет |
| 20 | 160 лет |

Таким образом, колода в 32 карты будет сортироваться этим компьютером в среднем 2,7⋅1019 лет.

Данная категория предназначена исключительно для образовательных целей, в связи с чем переходим к более практичным и используемым алгоритмам: устойчивой и неустойчивой сортировки, основанным на сравнении элементов в коллекции.

**Сортировка пузырьком**

Сортировка пузырько́м (англ. bubble sort), сортиро́вка простыми обменами, метод сортировки обменами — один из алгоритмов сортировки. По сравнению с другими алгоритмами считается простейшим для понимания и реализации. Эффективен для массивов небольшого размера. n {\displaystyle n} — размер массива, количество элементов массива. Сложность алгоритма: O {\displaystyle O} ( n 2 ) {\displaystyle (n^{2})}.

Алгоритм считается учебным, на практике (вне учебной литературы) не применяется (на практике применяются более эффективные (совершенные) алгоритмы). Лежит в основе некоторых более эффективных алгоритмов, например, алгоритма шейкерной сортировки, алгоритма пирамидальной сортировки и алгоритма быстрой сортировки.

Выполняется некоторое количество проходов по массиву — начиная от начала массива, перебираются пары соседних элементов массива. Если 1-й элемент пары больше 2-го, элементы переставляются (выполняется обмен).

Пары элементов массива перебираются (проходы по массиву повторяются) либо ( n − 1 ) {\displaystyle (n-1)} раз, либо до тех пор, пока на очередном проходе не обнаружится, что более не требуется выполнять перестановки (обмены) (массив отсортирован).

При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива (как бы «всплывает» до нужной позиции, как пузырёк в воде — откуда и название алгоритма).

Пример работы алгоритма

Используя алгоритм сортировки пузырьком, отсортируем по возрастанию массив чисел, равный «5 1 4 2 8». Выделены те элементы, которые сравниваются на текущем этапе.

Первый проход:

(5 1 4 2 8) (1 5 4 2 8), Здесь алгоритм сравнивает два первых элемента и меняет их местами.

(1 5 4 2 8) (1 4 5 2 8), Меняет местами, так как 5 > 4 {\displaystyle 5>4}

(1 4 5 2 8) (1 4 2 5 8), Меняет местами, так как 5 > 2 {\displaystyle 5>2}

(1 4 2 5 8) (1 4 2 5 8), Теперь, ввиду того, что элементы стоят на своих местах ( 8 > 5 {\displaystyle 8>5}), алгоритм не меняет их местами.

Второй проход:

(1 4 2 5 8) (1 4 2 5 8)

(1 4 2 5 8) (1 2 4 5 8), Меняет местами, так как 4 > 2 {\displaystyle 4>2}

(1 2 4 5 8) (1 2 4 5 8)

Теперь массив полностью отсортирован, но алгоритму это неизвестно. Необходимо сделать полный проход, чтобы определить, что перестановок (обменов) элементов не было.

Третий проход:

(1 2 4 5 8) (1 2 4 5 8)

(1 2 4 5 8) (1 2 4 5 8)

Теперь массив отсортирован и алгоритм может быть завершён.

**Сортировка перемешиванием, или Шейкерная сортировка**

Сортировка перемешиванием, или Шейкерная сортировка, или двунаправленная (англ. Cocktail sort) — разновидность пузырьковой сортировки. Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства.

Во-первых, если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения.

Во-вторых, при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Эти две идеи приводят к следующим модификациям в методе пузырьковой сортировки. Границы рабочей части массива (то есть части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

Лучший случай для этой сортировки — отсортированный массив ( O ( n ) {\displaystyle O(n)}), худший — отсортированный в обратном порядке ( O ( n 2 ) {\displaystyle O(n^{2})}).

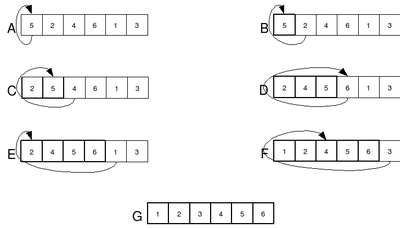
Наименьшее число сравнений в алгоритме Шейкер-сортировки C = N − 1 {\displaystyle C=N-1}. Это соответствует единственному проходу по упорядоченному массиву (лучший случай)

|  |
| --- |
| Трассировка программы: |
| 3 1 5 8 1 0 4 6 6 7 |
| 3 1 5 8 0 1 4 6 6 7 |
| 3 1 5 0 8 1 4 6 6 7 |
| 3 1 0 5 8 1 4 6 6 7 |
| 3 0 1 5 8 1 4 6 6 7 |
| 0 3 1 5 8 1 4 6 6 7 Left=1 |
| 0 1 3 5 8 1 4 6 6 7 |
| 0 1 3 5 1 8 4 6 6 7 |
| 0 1 3 5 1 4 8 6 6 7 |
| 0 1 3 5 1 4 6 8 6 7 |
| 0 1 3 5 1 4 6 6 8 7 |
| 0 1 3 5 1 4 6 6 7 8 Right=10 |
| 0 1 3 1 5 4 6 6 7 8 |
| 0 1 1 3 5 4 6 6 7 8 Left=3 |
| 0 1 1 3 4 5 6 6 7 8 |

Образно алгоритм можно описать так: на каждом шаге основного цикла рассматривается массив a[Left]÷a[Right], после выполнения двух внутренних циклов минимальный и максимальный элемент в исходном массиве перетекают к краям, минимальный в — a[Left], максимальный — в a[Right]. Пусть максимальный элемент имеет индекс k, тогда массив можно изобразить так: a[Left], a[1],..,a[k-1],A[k], a[k+1],..,a[Right];После сравнения A[k] с a[k+1] значение A[k] перейдет в k+1-ую ячейку, после сравнения k+1-й c k+2-й — в k+2-eю, и так далее, пока он не сместится в крайне правое положение с индексом Right. Аналогично для минимального. После выполнения цикла по всем подмассивам он отсортируется.

**Сортировка вставками**

Сортировка вставками (англ. Insertion sort) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов[1]. Вычислительная сложность — O ( n 2 ) {\displaystyle O(n^{2})}.

[](https://commons.wikimedia.org/wiki/File:Dsa_ins_sort.png?uselang=ru)Пример сортировки вставками

На вход алгоритма подаётся последовательность n чисел: a 1 , a 2 , . . . , a n . Сортируемые числа также называют *ключами*. Входная последовательность на практике представляется в виде массива с n элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности a 1 ′ , a 2 ′ , . . . , a n ′ , чтобы выполнялось следующее соотношение a 1 ′ ⩽ a 2 ′ ⩽ . . . ⩽ a n ′ .

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части. Проблема с долгим сдвигом массива вправо решается при помощи смены указателей.

**Сортировка слиянием**

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». 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 в 1945 году.

Подробный алгоритм сортировки

Действие алгоритма на примере сортировки случайных точек.

Для решения задачи сортировки эти три этапа выглядят так:

Сортируемый массив разбивается на две части примерно одинакового размера;

Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;

Два упорядоченных массива половинного размера соединяются в один.

1.1. — 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

3.1. Соединение двух упорядоченных массивов в один.

Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два уже отсортированных по возрастанию подмассива. Тогда:

3.2. Слияние двух подмассивов в третий результирующий массив.

На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.

3.3. «Прицепление» остатка.

Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

**Сортировка с помощью двоичного дерева**

Сортировка с помощью двоичного дерева (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, англ. tree sort) — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Данная сортировка является оптимальной при получении данных путём непосредственного чтения из потока (например, файла, сокета или консоли).

Алгоритм

* Построение двоичного дерева.
* Сборка результирующего массива путём обхода узлов в необходимом порядке следования ключей.

Эффективность

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка. Соответственно, для n объектов сложность будет составлять, что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать, что может привести к общей сложности порядка.

При физическом развёртывании древовидной структуры в памяти требуется не менее чем ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

**Сортировка выбором**

Сортировка выбором (англ. selection sort) — алгоритм сортировки. Может быть как устойчивый, так и неустойчивый. На массиве из n {\displaystyle n} элементов имеет время выполнения в худшем, среднем и лучшем случае O ( n 2 ) {\displaystyle O(n^{2})}, предполагая что сравнения делаются за постоянное время.

Алгоритм без дополнительного выделения памяти

Шаги алгоритма:

* Находим номер минимального значения в текущем списке.
* Производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции).
* Теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

Сравнение с другими алгоритмами сортировки

Так как после каждого прохода по внутреннему циклу делается только один обмен, то общее число обменов равно n − 1 {\displaystyle n-1}, что в n / 2 {\displaystyle n/2} раз меньше, чем в сортировке пузырьком.

Число проходов по внутреннему циклу равно n − 1 {\displaystyle n-1} даже в случае сортировки частично или полностью отсортированного массива.

* Число сравнений в теле цикла равно ( n − 1 ) ∗ n / 2 {\displaystyle (n-1)\*n/2}.
* Число сравнений в заголовках циклов ( n − 1 ) ∗ n / 2 {\displaystyle (n-1)\*n/2}.
* Число сравнений перед операцией обмена n − 1 {\displaystyle n-1}.
* Суммарное число сравнений n 2 − 1 {\displaystyle n^{2}-1}.
* Число обменов n − 1 {\displaystyle n-1}.

Время сортировки 10000 коротких целых чисел на одном и том же программно-аппаратном комплексе сортировкой выбором составило ≈ 40 секунд, а ещё более улучшенной сортировкой пузырьком ≈ 30 секунд.

Пирамидальная сортировка сильно улучшает базовый алгоритм, используя структуру данных «куча» для ускорения нахождения и удаления минимального элемента. Существует также двунаправленный вариант сортировки методом выбора, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

**Сортировка расчёской**

Сортировка расчёской (англ. comb sort) — это довольно[уточнить] упрощённый алгоритм сортировки, изначально спроектированный Влодзимежем Добосевичем в 1980 г. Позднее он был переоткрыт и популяризован в статье Стивена Лэйси и Ричарда Бокса в журнале Byte Magazine в апреле 1991 г. Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея — устранить черепах, или маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (кролики, большие значения в начале списка, не представляют проблемы для сортировки пузырьком).

В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица (сортировка Шелла также основана на этой идее, но она является модификацией сортировки вставками, а не сортировки пузырьком).

В «пузырьке», «шейкере» и «чёт-нечете» при переборе массива сравниваются соседние элементы. Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, который может быть, например, 1.25. Сначала расстояние между элементами максимально, то есть равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы как и в сортировке пузырьком, но такая итерация одна.

Последним рассмотрим один из самых эффективных алгоритмов сортировки – быстрая сортировка.

**Быстрая сортировка**

Быстрая сортировка, сортировка Хоара (англ. quicksort), часто называемая qsort (по имени в стандартной библиотеке языка Си) — алгоритм сортировки, разработанный английским информатиком Тони Хоаром во время его работы в МГУ в 1960 году.

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O ( n log ⁡ n ) {\displaystyle O(n\log n)} обменов при упорядочении n {\displaystyle n} элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы (таким образом улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов).

Общая идея алгоритма состоит в следующем:

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже).
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие»[2].
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения (см. ниже).

Хоар разработал этот метод применительно к машинному переводу; словарь хранился на магнитной ленте, и сортировка слов обрабатываемого текста позволяла получить их переводы за один прогон ленты, без перемотки её назад. Алгоритм был придуман Хоаром во время его пребывания в Советском Союзе, где он обучался в Московском университете компьютерному переводу и занимался разработкой русско-английского разговорника.

Алгоритм

Общий механизм сортировки

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

* Выбрать элемент из массива. Назовём его опорным.
* Разбиение: перераспределение элементов в массиве таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.
* Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

В наиболее общем виде алгоритм на псевдокоде (где A — сортируемый массив, а low и high — соответственно, нижняя и верхняя границы сортируемого участка этого массива) выглядит следующим образом:

algorithm quicksort(A, low, high) is

if low < high then

p:= partition(A, low, high)

quicksort(A, low, p - 1)

quicksort(A, p, high)

Здесь предполагается, что массив A передаётся по ссылке, то есть сортировка происходит «на том же месте», а неописанная функция partition возвращает индекс опорного элемента.

Для выбора опорного элемента и операции разбиения существуют разные подходы, влияющие на производительность алгоритма.

Возможна также следующая реализация быстрой сортировки:

algorithm quicksort(A) is

if A is empty

return A

pivot:= A.pop() (извлечь последний или первый элемент из массива)

lA:= A.filter(where e <= pivot) (создать массив с элементами меньше опорного)

rA := A.filter(where e > pivot) (создать массив с элементами больше опорного)

return quicksort(lA) + [pivot] + quicksort(rA) (вернуть массив состоящий из отсортированной левой части, опорного и отсортированной правой части).

На практике она не используется, а служит лишь в образовательных целях, так как использует дополнительную память, чего можно избежать.

Выбор опорного элемента

В ранних реализациях, как правило, опорным выбирался первый элемент, что снижало производительность на отсортированных массивах. Для улучшения эффективности может выбираться средний, случайный элемент или (для больших массивов) медиана первого, среднего и последнего элементов.[3] Медиана всей последовательности является оптимальным опорным элементом, но её вычисление слишком трудоёмко для использования в сортировке.

Выбор опорного элемента по медиане трёх для разбиения Ломуто:

mid := (low + high) / 2

if A[mid] < A[low]

swap A[low] with A[mid]

if A[high] < A[low]

swap A[low] with A[high]

if A[high] < A[mid]

swap A[high] with A[mid]

pivot:= A[mid]

Разбиение Ломуто

Данный алгоритм разбиения был предложен Нико Ломуто[4] и популяризован в книгах Бентли (Programming Pearls) и Кормена (Введение в алгоритмы).[5] В данном примере опорным выбирается последний элемент. Алгоритм хранит индекс в переменной i. Каждый раз, когда находится элемент, меньше или равный опорному, индекс увеличивается, и элемент вставляется перед опорным. После разбиения опорный элемент окажется в позиции i - на границе между двумя подмножествами. Хоть эта схема разбиения проще и компактнее, чем схема Хоара, она менее эффективна и используется в обучающих материалах. Сложность данной быстрой сортировки возрастает до O(n2), когда массив уже отсортирован или все его элементы равны. Существуют различные методы оптимизации данной сортировки: алгоритмы выбора опорного элемента, использование сортировки вставками на маленьких массивах. В данном примере сортируются элементы массива A от low до high (включительно)[5]:

algorithm partition(A, low, high) is

pivot := A[high]

i := low - 1

for j := low to high do

if A[j] ≤ pivot then

i := i + 1

if i ≠ j then

swap A[i] with A[j]

swap A[i + 1] with A[high]

return i + 1

Сортировка всего массива может быть выполнена с помощью выполнения quicksort(A, 1, length(A)).

Достоинства и недостатки

Достоинства:

* Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
* Алгоритм очень короткий: запомнив основные моменты, его легко написать «из головы», невелика константа при n log ⁡ n {\displaystyle n\log n}.
* С модификациями требует лишь O ( log ⁡ n ) {\displaystyle O(\log n)} дополнительной памяти в виде стека (неулучшенный рекурсивный алгоритм — в худшем случае O ( n ) {\displaystyle O(n)} стека).
* Хорошо сочетается с механизмами кэширования и виртуальной памяти.
* Допускает естественное распараллеливание (сортировка выделенных подмассивов в параллельно выполняющихся подпроцессах).
* Допускает эффективную модификацию для сортировки по нескольким ключам (в частности — алгоритм Седжвика для сортировки строк): благодаря тому, что в процессе разделения автоматически выделяется отрезок элементов, равных опорному, этот отрезок можно сразу же сортировать по следующему ключу.
* Работает на связных списках и других структурах с последовательным доступом, допускающих эффективный проход как от начала к концу, так и от конца к началу.

Недостатки:

* Сильно деградирует по скорости (до O ( n 2 ) {\displaystyle O(n^{2})}) в худшем или близком к нему случае, что может случиться при неудачных входных данных.
* Прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека, так как в худшем случае ей может потребоваться сделать O ( n ) {\displaystyle O(n)} вложенных рекурсивных вызовов.
* Неустойчив.

Улучшения

Улучшения алгоритма направлены, в основном, на устранение или смягчение вышеупомянутых недостатков, вследствие чего все их можно разделить на три группы: придание алгоритму устойчивости, устранение деградации производительности специальным выбором опорного элемента, и защита от переполнения стека вызовов из-за большой глубины рекурсии при неудачных входных данных.

* Проблема неустойчивости решается путём расширения ключа исходным индексом элемента в массиве. В случае равенства основных ключей сравнение производится по индексу, исключая, таким образом, возможность изменения взаимного положения равных элементов. Эта модификация не бесплатна — она требует дополнительно O(n) памяти и одного полного прохода по массиву для сохранения исходных индексов.
* Деградация по скорости в случае неудачного набора входных данных решается по двум разным направлениям: снижение вероятности возникновения худшего случая путём специального выбора опорного элемента и применение различных технических приёмов, обеспечивающих устойчивую работу на неудачных входных данных. Для первого направления:
* Выбор среднего элемента. Устраняет деградацию для предварительно отсортированных данных, но оставляет возможность случайного появления или намеренного подбора «плохого» массива.
* Выбор медианы из трёх элементов: первого, среднего и последнего. Снижает вероятность возникновения худшего случая, по сравнению с выбором среднего элемента.
* Случайный выбор. Вероятность случайного возникновения худшего случая становится исчезающе малой, а намеренный подбор — практически неосуществимым. Ожидаемое время выполнения алгоритма сортировки составляет O(n log n).
* Недостаток всех усложнённых методов выбора опорного элемента — дополнительные накладные расходы; впрочем, они не так велики.
* Во избежание отказа программы из-за большой глубины рекурсии могут применяться следующие методы:
* При достижении нежелательной глубины рекурсии переходить на сортировку другими методами, не требующими рекурсии. Примером такого подхода является алгоритм Introsort или некоторые реализации быстрой сортировки в библиотеке STL. Можно заметить, что алгоритм очень хорошо подходит для такого рода модификаций, так как на каждом этапе позволяет выделить непрерывный отрезок исходного массива, предназначенный для сортировки, и то, каким методом будет отсортирован этот отрезок, никак не влияет на обработку остальных частей массива.
* Модификация алгоритма, устраняющая одну ветвь рекурсии: вместо того, чтобы после разделения массива вызывать рекурсивно процедуру разделения для обоих найденных подмассивов, рекурсивный вызов делается только для меньшего подмассива, а больший обрабатывается в цикле в пределах этого же вызова процедуры. С точки зрения эффективности в среднем случае разницы практически нет: накладные расходы на дополнительный рекурсивный вызов и на организацию сравнения длин подмассивов и цикла — примерно одного порядка. Зато глубина рекурсии ни при каких обстоятельствах не превысит log 2 ⁡ n {\displaystyle \log \_{2}n}, а в худшем случае вырожденного разделения она вообще будет не более 2 — вся обработка пройдёт в цикле первого уровня рекурсии. Применение этого метода не спасёт от катастрофического падения производительности, но переполнения стека не будет.
* Разбивать массив не на две, а на три части[9].

# **Стандартные функции поиска и сортировки**

Сортировка в Python выполняется с помощью sorted() и list.sort(). Разбираем на примерах, как это работает.

Сортировка в Python выполняется функцией sorted(), если это итерируемые объекты, и методом list.sort(), если это список. Рассмотрим подробнее, как это работало в старых версиях и как работает сейчас.

Примечание Вы читаете улучшенную версию некогда выпущенной нами статьи.

[Разработка на Python с нуля: роадмап программиста](https://tproger.ru/articles/python-roadmap" \t "_blank)

[tproger.ru](https://tproger.ru/articles/python-roadmap" \t "_blank)

#### Основы сортировки

Так как отсортировать список Python? Для сортировки по возрастанию достаточно вызвать функцию сортировки Python sorted(), которая вернёт новый отсортированный список:

>>> sorted([5, 2, 3, 1, 4])

[1, 2, 3, 4, 5]

Для сортировки списка Python также можно использовать метод списков list.sort(), который изменяет исходный список (и возвращает None во избежание путаницы). Обычно Python sort list не так удобен, как использование sorted(), но если вам не нужен исходный список, то так будет немного эффективнее:

>>> a = [5, 2, 3, 1, 4]

>>> a.sort()

>>> a

[1, 2, 3, 4, 5]

Прим.перев. В Python вернуть None и не вернуть ничего — одно и то же.

Ещё одно отличие заключается в том, что метод list.sort() определён только для списков, в то время как функция sorted Python работает со всеми итерируемыми объектами. Грубо говоря, функция sort Python сортирует список и сохраняет его в отсортированном виде, в то время как функция sorted Питон создаёт новый отсортированный список без изменения исходного.

>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})

[1, 2, 3, 4, 5]

Прим.перев. При итерировании по словарю Python возвращает его ключи. Если вам нужны их значения или пары «ключ-значение», используйте методы dict.values() и dict.items() соответственно.

Рассмотрим основные функции сортировки Python.

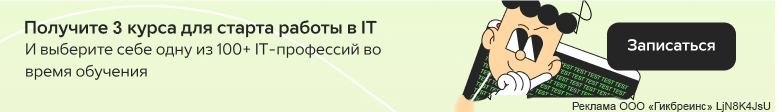
#### Функции-ключи

С версии Python 2.4 у list.sort() и sorted() появился параметр key для указания функции, которая будет вызываться на каждом элементе до сравнения. Вот регистронезависимое сравнение строк:

>>> sorted("This is a test string from Andrew".split(), key=str.lower)

['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

Значение key должно быть функцией, принимающей один аргумент и возвращающей ключ для сортировки. Работает быстро, потому что функция-ключ вызывается один раз для каждого элемента.



Часто можно встретить код, где сложный объект сортируется по одному из его индексов. Например:

>>> student\_tuples = [

('john', 'A', 15),

('jane', 'B', 12),

('dave', 'B', 10),

]

>>> sorted(student\_tuples, key=lambda student: student[2]) # сортируем по возрасту

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

Тот же метод работает для объектов с именованными атрибутами:

>>> class Student:

def \_\_init\_\_(self, name, grade, age):

self.name = name

self.grade = grade

self.age = age

def \_\_repr\_\_(self):

return repr((self.name, self.grade, self.age))

def weighted\_grade(self):

return 'CBA'.index(self.grade) / self.age

>>> student\_objects = [

Student('john', 'A', 15),

Student('jane', 'B', 12),

Student('dave', 'B', 10),

]

>>> sorted(student\_objects, key=lambda student: student.age) # сортируем по возрасту

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]​

#### Функции модуля operator

Показанные выше примеры функций-ключей встречаются настолько часто, что Python предлагает удобные функции, чтобы сделать всё проще и быстрее. Модуль [operator](https://docs.python.org/3/library/operator.html#module-operator) содержит функции itemgetter(), attrgetter() и, начиная с Python 2.6, methodcaller(). С ними всё ещё проще:

>>> from operator import itemgetter, attrgetter, methodcaller

>>> sorted(student\_tuples, key=itemgetter(2))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

>>> sorted(student\_objects, key=attrgetter('age'))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

Функции operator дают возможность использовать множественные уровни сортировки массива Python. Отсортируем учеников сначала по оценке, а затем по возрасту:

>>> sorted(student\_tuples, key=itemgetter(1, 2))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

>>> sorted(student\_objects, key=attrgetter('grade', 'age'))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

Используем функцию methodcaller() для сортировки учеников по взвешенной оценке:

>>> [(student.name, student.weighted\_grade()) for student in student\_objects]

[('john', 0.13333333333333333), ('jane', 0.08333333333333333), ('dave', 0.1)]

>>> sorted(student\_objects, key=methodcaller('weighted\_grade'))

[('jane', 'B', 12), ('dave', 'B', 10), ('john', 'A', 15)]

#### Сортировка по возрастанию и сортировка по убыванию в Python

list.sort() и sorted() есть параметр reverse, принимающий boolean-значение. Он нужен для обозначения сортировки по убыванию. Отсортируем учеников по убыванию возраста:

>>> sorted(student\_tuples, key=itemgetter(2), reverse=True)

[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

>>> sorted(student\_objects, key=attrgetter('age'), reverse=True)

[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

## Стабильность сортировки и сложные сортировки в Python

Начиная с версии Python 2.2, сортировки гарантированно стабильны: если у нескольких записей есть одинаковые ключи, их порядок останется прежним. Пример:

>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]

>>> sorted(data, key=itemgetter(0))

[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]

Обратите внимание, что две записи с 'blue' сохранили начальный порядок. Это свойство позволяет составлять сложные сортировки путём постепенных сортировок. Далее мы сортируем данные учеников сначала по возрасту в порядке возрастания, а затем по оценкам в убывающем порядке, чтобы получить данные, отсортированные в первую очередь по оценке и во вторую — по возрасту:

>>> s = sorted(student\_objects, key=attrgetter('age')) # сортируем по вторичному ключу

>>> sorted(s, key=attrgetter('grade'), reverse=True) # по первичному

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

Алгоритмы сортировки Python вроде [Timsort](https://ru.wikipedia.org/wiki/Timsort) проводят множественные сортировки так эффективно, потому что может извлечь пользу из любого порядка, уже присутствующего в наборе данных.

#### Декорируем-сортируем-раздекорируем

1. Сначала исходный список пополняется новыми значениями, контролирующими порядок сортировки.
2. Затем новый список сортируется.
3. После этого добавленные значения убираются, и в итоге остаётся отсортированный список, содержащий только исходные элементы.

Вот так можно отсортировать данные учеников по оценке:

>>> decorated = [(student.grade, i, student) for i, student in enumerate(student\_objects)]

>>> decorated.sort()

>>> [student for grade, i, student in decorated] # раздекорируем

[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

Это работает из-за того, что кортежи сравниваются лексикографически, сравниваются первые элементы, а если они совпадают, то сравниваются вторые и так далее.

Не всегда обязательно включать индекс в декорируемый список, но у него есть преимущества:

1. Сортировка стабильна — если у двух элементов одинаковый ключ, то их порядок не изменится.
2. У исходных элементов не обязательно должна быть возможность сравнения, так как порядок декорированных кортежей будет определяться максимум по первым двум элементам. Например, исходный список может содержать комплексные числа, которые нельзя сравнивать напрямую.

Ещё эта идиома называется [преобразованием Шварца](https://ru.wikipedia.org/wiki/Преобразование_Шварца) в честь Рэндела Шварца, который популяризировал её среди Perl-программистов.

Для больших списков и версий Python ниже 2.4, «декорируем-сортируем-раздекорируем» будет оптимальным способом сортировки. Для версий 2.4+ ту же функциональность предоставляют функции-ключи.

#### Использование параметра cmp

Все версии Python 2.x поддерживали параметр cmp для обработки пользовательских функций сравнения. В Python 3.0 от этого параметра полностью избавились. В Python 2.x в sort() можно было передать функцию, которая использовалась бы для сравнения элементов. Она должна принимать два аргумента и возвращать отрицательное значение для случая «меньше чем», положительное — для «больше чем» и ноль, если они равны:

>>> def numeric\_compare(x, y):

return x - y

>>> sorted([5, 2, 4, 1, 3], cmp=numeric\_compare)

[1, 2, 3, 4, 5]

Можно сравнивать в обратном порядке:

>>> def reverse\_numeric(x, y):

return y - x

>>> sorted([5, 2, 4, 1, 3], cmp=reverse\_numeric)

[5, 4, 3, 2, 1]

При портировании кода с версии 2.x на 3.x может возникнуть ситуация, когда нужно преобразовать пользовательскую функцию для сравнения в функцию-ключ. Следующая обёртка упрощает эту задачу по Python:

def cmp\_to\_key(mycmp):

'Перевести cmp=функция в key=функция'

class K(object):

def \_\_init\_\_(self, obj, \*args):

self.obj = obj

def \_\_lt\_\_(self, other):

return mycmp(self.obj, other.obj) < 0

def \_\_gt\_\_(self, other):

return mycmp(self.obj, other.obj) > 0

def \_\_eq\_\_(self, other):

return mycmp(self.obj, other.obj) == 0

def \_\_le\_\_(self, other):

return mycmp(self.obj, other.obj) <= 0

def \_\_ge\_\_(self, other):

return mycmp(self.obj, other.obj) >= 0

def \_\_ne\_\_(self, other):

return mycmp(self.obj, other.obj) != 0

return K

Чтобы произвести преобразование, оберните старую функцию:

>>> sorted([5, 2, 4, 1, 3], key=cmp\_to\_key(reverse\_numeric))

[5, 4, 3, 2, 1]

В Python 2.7 функция cmp\_to\_key() была добавлена в модуль functools.

#### Поддержание порядка сортировки

В стандартной библиотеке Python нет модулей, аналогичных типам данных C++ вроде set и map. Python делегирует эти задачи сторонним библиотекам, доступным в [Python Package Index](https://pypi.org): они используют различные методы для сохранения типов list, dict и set в отсортированном порядке. Поддержание порядка с помощью специальной структуры данных может помочь избежать очень медленного поведения (квадратичного времени выполнения) при наивном подходе с редактированием и постоянной пересортировкой данных. Вот некоторые из модулей, реализующих эти типы данных:

* [SortedContainers](http://www.grantjenks.com/docs/sortedcontainers/) — реализация сортированных типов list, dict и set на чистом Python, по скорости не уступает реализациям на C. Тестирование включает 100% покрытие кода и многие часы стресс-тестирования. В документации можно найти полный справочник по API, [сравнение производительности](http://www.grantjenks.com/docs/sortedcontainers/performance.html) и руководства по внесению своего вклада.
* [rbtree](https://pypi.python.org/pypi/rbtree) — быстрая реализация на C для типов dict и set. Реализация использует структуру данных, известную как красно-чёрное дерево.
* [treap](https://pypi.python.org/pypi/treap) — сортированный dict. В реализации используется Декартово дерево, а производительность улучшена с помощью Cython.
* [bintrees](https://pypi.python.org/pypi/bintrees) — несколько реализаций типов dict и set на основе деревьев на C. Самые быстрые основаны на АВЛ и красно-чёрных деревьях. Расширяет общепринятый API для предоставления операций множеств для словарей.
* [banyan](https://pypi.python.org/pypi/Banyan) — быстрая реализация dict и set на C.
* [skiplistcollections](https://pypi.python.org/pypi/skiplistcollections) — реализация на чистом Python, основанная на списках с пропусками, предлагает ограниченный API для типов dict и set.
* [blist](https://pypi.python.org/pypi/blist) — предоставляет сортированные типы list, dict и set, основанные на типе данных «blist», реализация на Б-деревьях. Написано на Python и C.

Прочее

Для сортировки с учётом языка используйте locale.strxfrm() в качестве ключевой функции или locale.strcoll() в качестве функции сравнения. Параметр reverse всё ещё сохраняет стабильность сортировки. Этот эффект можно сымитировать без параметра, использовав встроенную функцию reversed() дважды:

>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]

>>> assert sorted(data, reverse=True) == list(reversed(sorted(reversed(data))))

Чтобы создать стандартный порядок сортировки для класса, просто добавьте реализацию соответствующих методов сравнения:

>>> Student.\_\_eq\_\_ = lambda self, other: self.age == other.age

>>> Student.\_\_ne\_\_ = lambda self, other: self.age != other.age

>>> Student.\_\_lt\_\_ = lambda self, other: self.age < other.age

>>> Student.\_\_le\_\_ = lambda self, other: self.age <= other.age

>>> Student.\_\_gt\_\_ = lambda self, other: self.age > other.age

>>> Student.\_\_ge\_\_ = lambda self, other: self.age >= other.age

>>> sorted(student\_objects)

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

Для типов, сравнение которых работает обычным образом, рекомендуется определять все 6 операторов. Декоратор классов functools.total\_ordering упрощает их реализацию. Функциям-ключам не нужен доступ к внутренним данным сортируемых объектов. Они также могут осуществлять доступ к внешним ресурсам. Например, если оценки ученика хранятся в словаре, их можно использовать для сортировки отдельного списка с именами учеников:

>>> students = ['dave', 'john', 'jane']

>>> newgrades = {'john': 'F', 'jane':'A', 'dave': 'C'}

>>> sorted(students, key=newgrades.\_\_getitem\_\_)

['jane', 'dave', 'john']

# **Список литературы**

1. https://tproger.ru/translations/python-sorting

2. https://habr.com/ru/companies/kts/articles/727528/

3. https://prog-cpp.ru/algorithm-sort/

4. https://prog-cpp.ru/search-serial/

5. https://prog-cpp.ru/search-index/

6. https://prog-cpp.ru/search-binary/

7. https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC\_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B

8. https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%82%D0%B5%D0%B3%D0%BE%D1%80%D0%B8%D1%8F:%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%8B\_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0\_%D0%BD%D0%B0\_%D0%B3%D1%80%D0%B0%D1%84%D0%B0%D1%85