Лекция 5.2. Алгоритмы поиска и сортировки

**Оглавление**

[**Введение** 2](#_Toc180575655)

[**1.** **Алгоритмы поиска** 3](#_Toc180575656)

[**Операторы членства (Membership Operators)** 3](#_Toc180575657)

[**Линейный поиск** 3](#_Toc180575658)

[**Бинарный поиск** 4](#_Toc180575659)

[**Jump Search** 6](#_Toc180575660)

[**Поиск Фибоначчи** 7](#_Toc180575661)

[**Экспоненциальный поиск** 10](#_Toc180575662)

[**2.** **Алгоритмы сортировки** 12](#_Toc180575663)

[**Bogosort** 13](#_Toc180575664)

[**Сортировка пузырьком** 14](#_Toc180575665)

[**Алгоритм** 14](#_Toc180575666)

[**Реализация** 15](#_Toc180575667)

[**Время сортировки** 15](#_Toc180575668)

[**Сортировка выборкой** 15](#_Toc180575669)

[**Алгоритм** 15](#_Toc180575670)

[**Реализация** 16](#_Toc180575671)

[**Время сортировки** 16](#_Toc180575672)

[**Пирамидальная сортировка** 16](#_Toc180575673)

[**Алгоритм** 16](#_Toc180575674)

[**Реализация** 17](#_Toc180575675)

[**Время сортировки** 18](#_Toc180575676)

[**Сортировка вставками** 18](#_Toc180575677)

[**Алгоритм** 18](#_Toc180575678)

[**Реализация** 18](#_Toc180575679)

[**Время сортировки** 18](#_Toc180575680)

[**Сортировка слиянием** 19](#_Toc180575681)

[**Алгоритм** 19](#_Toc180575682)

[**Реализация** 20](#_Toc180575683)

[**Время сортировки** 21](#_Toc180575684)

[**Быстрая сортировка (Quicksort)** 21](#_Toc180575685)

[**Алгоритм** 21](#_Toc180575686)

[**Реализация** 21](#_Toc180575687)

[**Время выполнения** 22](#_Toc180575688)

[**Заключение** 23](#_Toc180575689)

[**Список литературы** 24](#_Toc180575690)

**Алгоритмы поиска и сортировки**

**Цель:** формирование знаний в отношении алгоритмы поиска и сортировки в языке Python, их принципов работы и методов, эффективности, быстродействия и способов оптимизации, основных библиотек, реализующих эффективные алгоритмы поиска и сортировки.

**План лекции:**Введение.1. Алгоритмы поиска.  
2. Алгоритмы сортировки.  
Заключение.Список литературы.

# **Введение**

Алгоритмы поиска и сортировки являются фундаментальными концепциями в программировании. Они помогают эффективно обрабатывать и управлять данными, что является ключевым аспектом при разработке программного обеспечения. В этой статье мы рассмотрим основные алгоритмы поиска и сортировки, а также приведем примеры их реализации на Python. Понимание этих алгоритмов поможет вам лучше разбираться в том, как работают различные структуры данных и как можно оптимизировать выполнение программ.

# **Алгоритмы поиска**

## **Операторы членства (Membership Operators)**

Алгоритмы развиваются и оптимизируются в результате постоянной эволюции и необходимости находить наиболее эффективные решения для основных проблем в различных областях.

Одной из наиболее распространенных проблем в области компьютерных наук является поиск в коллекции и определение того, присутствует ли данный объект в коллекции или нет.

В Python самый простой способ поиска объекта — использовать операторы членства. Их название связано с тем, что они позволяют нам определить, является ли данный объект членом коллекции.

Эти операторы могут использоваться с любой итерируемой структурой данных в Python, включая строки, списки и кортежи:

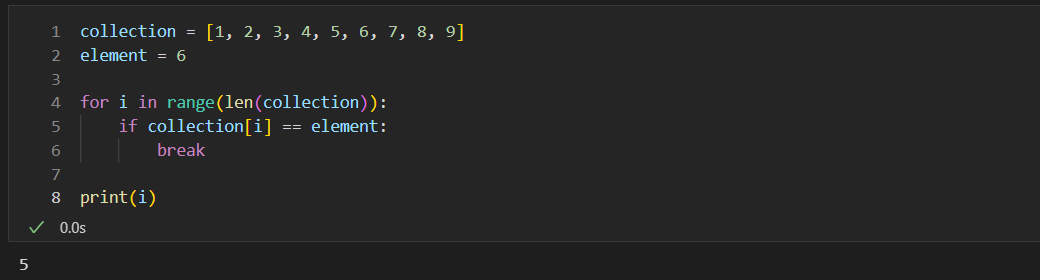
* in — возвращает True, если данный элемент присутствует в структуре данных.
* not in — возвращает True, если данный элемент не присутствует в структуре данных.

****

## **Линейный поиск**

Линейный поиск — это один из самых простых и понятных алгоритмов поиска. Мы можем думать о нем как о расширенной версии нашей собственной реализации оператора ***in*** в Python.

Суть алгоритма заключается в том*,* чтобы перебрать массив и вернуть индекс первого вхождения элемента, когда он найден:

****

Временная сложность линейного поиска равна ***O(n)***. Это означает, что время, необходимое для выполнения, увеличивается с увеличением количества элементов в нашем входном списке.

Линейный поиск не часто используется на практике, потому что такая же эффективность может быть достигнута с помощью встроенных методов или существующих операторов. К тому же, он не такой быстрый и эффективный, как другие алгоритмы поиска.

Линейный поиск хорошо подходит для тех случаев, когда нам нужно найти первое вхождение элемента в несортированной коллекции. Это связано с тем, что он не требует сортировки коллекции перед поиском (в отличие от большинства других алгоритмов поиска).

## **Бинарный поиск**

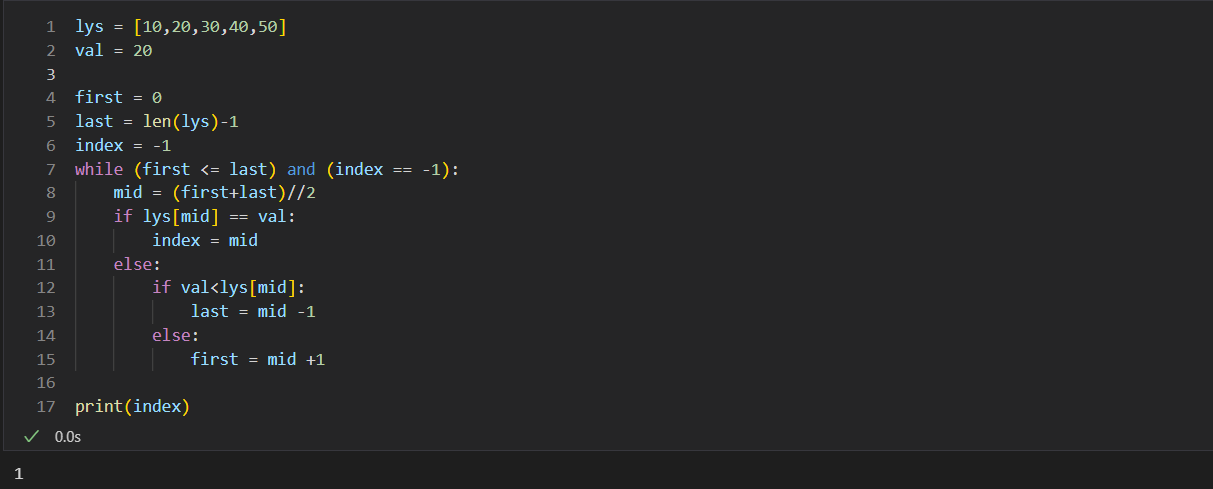
Бинарный поиск работает по принципу [«разделяй и властвуй»](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B9_%D0%B8_%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B2%D1%83%D0%B9_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма.

Предполагая, что мы ищем значение *val* в отсортированном массиве, алгоритм сравнивает *val* со значением среднего элемента массива, который мы будем называть *mid*.

* Если *mid* — это тот элемент, который мы ищем (в лучшем случае), мы возвращаем его индекс.
* Если нет, мы определяем, в какой половине массива мы будем искать *val* дальше, основываясь на том, меньше или больше значение *val* значения *mid*, и отбрасываем вторую половину массива.
* Затем мы рекурсивно или итеративно выполняем те же шаги, выбирая новое значение для *mid*, сравнивая его с *val* и отбрасывая половину массива на каждой итерации алгоритма.

Алгоритм бинарного поиска можно написать как рекурсивно, так и итеративно. [В Python рекурсия обычно медленнее](https://stackoverflow.com/questions/2651112/is-recursion-ever-faster-than-looping/2651200#2651200), потому что она требует выделения новых кадров стека.

Поскольку хороший алгоритм поиска должен быть максимально быстрым и точным, давайте рассмотрим итеративную реализацию бинарного поиска:

****

На каждой итерации алгоритм выполняет одно из следующих действий:

* Возврат индекса текущего элемента.
* Поиск в левой половине массива.
* Поиск в правой половине массива.

Мы можем выбрать только одно действие на каждой итерации. Также на каждой итерации наш массив делится на две части. Из-за этого временная сложность двоичного поиска равна ***O(log n)***.

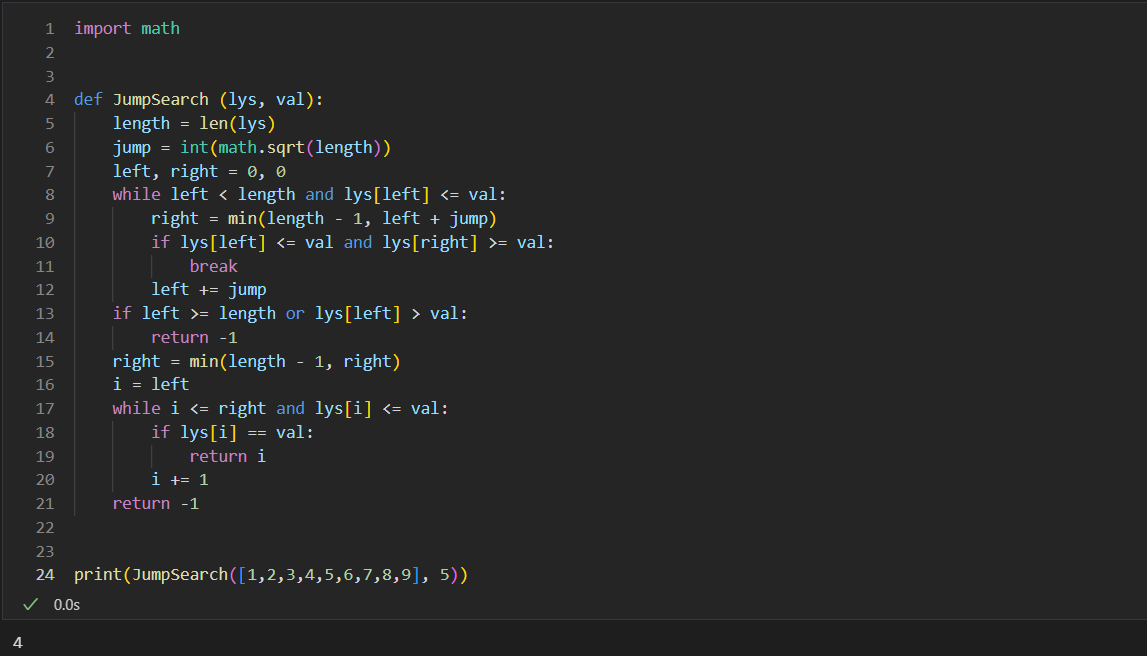
Одним из недостатков бинарного поиска является то, что если в массиве имеется несколько вхождений элемента, он возвращает индекс не первого элемента, а ближайшего к середине.

Бинарный поиск довольно часто используется на практике, потому что он эффективен и быстр по сравнению с линейным поиском. Однако у него есть некоторые недостатки, такие как зависимость от оператора **//**. Существует много других алгоритмов поиска, работающих по принципу «разделяй и властвуй», которые являются производными от бинарного поиска. Некоторые из них мы рассмотрим далее.

## **Jump Search**

Jump Search похож на бинарный поиск тем, что он также работает с отсортированным массивом и использует аналогичный подход «разделяй и властвуй» для поиска по нему.

Его можно классифицировать как усовершенствованный алгоритм линейного поиска, поскольку он зависит от линейного поиска для выполнения фактического сравнения при поиске значения.



В заданном отсортированном массиве мы ищем не постепенно по элементам массива, а скачкообразно. Если у нас есть размер прыжка, то наш алгоритм будет рассматривать элементы входного списка *lys* в следующем порядке: *lys[0], lys[0+jump], lys[0+2jump], lys[0+3jump]* и так далее.

С каждым прыжком мы сохраняем предыдущее значение и его индекс. Когда мы находим множество значений (блок), где *lys[i] < element < lys[i + jump]*, мы выполняем линейный поиск с *lys[i]* в качестве самого левого элемента и *lys[i + jump]* в качестве самого правого элемента в нашем множестве:

Поскольку это сложный алгоритм, давайте рассмотрим пошаговое вычисление:

* Jump search сначала определит размер прыжка путем вычисления *math.sqrt(len(lys))*. Поскольку у нас 9 элементов, размер прыжка будет √9 = 3.
* Далее мы вычисляем значение переменной *right*. Оно рассчитывается как минимум из двух значений: длины массива минус 1 и значения *left + jump*, которое в нашем случае будет 0 + 3 = 3. Поскольку 3 меньше 8, мы используем 3 в качестве значения переменной *right*.
* Теперь проверим, находится ли наш искомый элемент 5 между *lys[0] и lys[3]*. Поскольку 5 не находится между 1 и 4, мы идем дальше.
* Затем мы снова делаем расчеты и проверяем, находится ли наш искомый элемент между *lys[3] и lys[6]*, где 6 — это *3 + jump*. Поскольку 5 находится между 4 и 7, мы выполняем линейный поиск по элементам между *lys[3] и lys[6]* и возвращаем индекс нашего элемента:

Временная сложность jump search равна ***O(√n)***, где ***√n*** — размер прыжка, а ***n*** — длина списка. Таким образом, с точки зрения эффективности jump search находится между алгоритмами линейного и бинарного поиска.

Единственное наиболее важное преимущество jump search по сравнению с бинарным поиском заключается в том, что он не опирается на оператор деления (***/***).

В большинстве процессоров [использование оператора деления является дорогостоящим по сравнению с другими основными арифметическими операциями](http://ithare.com/wp-content/uploads/part101_infographics_v08.png) (сложение, вычитание и умножение), поскольку реализация алгоритма деления является итеративной.

Стоимость сама по себе очень мала, но когда количество искомых элементов очень велико, а количество необходимых операций деления растет, стоимость может постепенно увеличиваться. Поэтому jump search лучше бинарного поиска, когда в системе имеется большое количество элементов: там даже небольшое увеличение скорости имеет значение.

Чтобы ускорить jump search, мы могли бы использовать бинарный поиск или какой-нибудь другой алгоритм для поиска в блоке вместо использования гораздо более медленного линейного поиска.

## **Поиск Фибоначчи**

Поиск Фибоначчи — это еще один алгоритм «разделяй и властвуй», который имеет сходство как с бинарным поиском, так и с jump search. Он получил свое название потому, что использует [числа Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8) для вычисления размера блока или диапазона поиска на каждом шаге.

Числа Фибоначчи — это последовательность чисел 0, 1, 1, 2, 3, 5, 8, 13, 21 …, где каждый элемент является суммой двух предыдущих чисел.

Алгоритм работает с тремя числами Фибоначчи одновременно. Давайте назовем эти три числа *fibM, fibM\_minus\_1 и fibM\_minus\_2*. Где *fibM\_minus\_1 и fibM\_minus\_2* — это два числа, предшествующих *fibM* в последовательности:

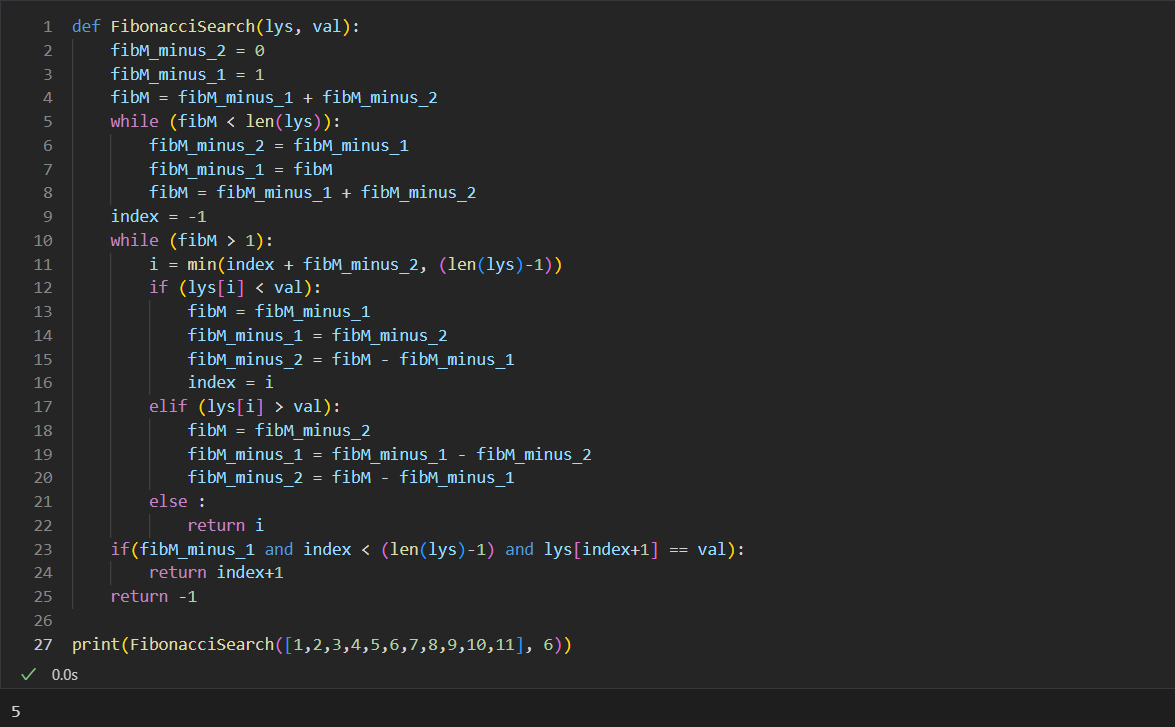
*fibM = fibM\_minus\_1 + fibM\_minus\_2*

Мы инициализируем значения 0, 1, 1 или первые три числа в последовательности Фибоначчи. Это поможет нам избежать IndexError в случае, когда наш массив *lys* содержит очень маленькое количество элементов.

Затем мы выбираем наименьшее число последовательности Фибоначчи, которое больше или равно числу элементов в нашем массиве *lys*, в качестве значения *fibM*. А два числа Фибоначчи непосредственно перед ним — в качестве значений *fibM\_minus\_1* и *fibM\_minus\_2*. Пока в массиве есть элементы и значение *fibM* больше единицы, мы:

* Сравниваем *val* со значением блока в диапазоне до *fibM\_minus\_2* и возвращаем индекс элемента, если он совпадает.
* Если значение больше, чем элемент, который мы в данный момент просматриваем, мы перемещаем значения *fibM*, *fibM\_minus\_1* и *fibM\_minus\_2* на два шага вниз в последовательности Фибоначчи и меняем индекс на индекс элемента.
* Если значение меньше, чем элемент, который мы в данный момент просматриваем, мы перемещаем значения *fibM*, *fibM\_minus\_1* и *fibM\_minus\_2* на один шаг вниз в последовательности Фибоначчи.

Давайте посмотрим на реализацию этого алгоритма на Python:



Давайте посмотрим на пошаговый процесс поиска:

* Присваиваем переменной *fibM* наименьшее число Фибоначчи, которое больше или равно длине списка. В данном случае наименьшее число Фибоначчи, отвечающее нашим требованиям, равно 13.
* Значения присваиваются следующим образом:

*fibM = 13*

*fibM\_minus\_1 = 8*

*fibM\_minus\_2 = 5*

*index = -1*

* Далее мы проверяем элемент *lys[4]*, где 4 — это минимум из двух значений — *index + fibM\_minus\_2 (-1+5)* и длина массива минус 1 (11-1). Поскольку значение *lys[4]* равно 5, что меньше искомого значения, мы перемещаем числа Фибоначчи на один шаг вниз в последовательности, получая следующие значения:

*fibM = 8*

*fibM\_minus\_1 = 5*

*fibM\_minus\_2 = 3*

*index = 4*

* Далее мы проверяем элемент *lys[7]*, где 7 — это минимум из двух значений: *index + fibM\_minus\_2 (4 + 3)* и длина массива минус 1 (11-1). Поскольку значение *lys[7]* равно 8, что больше искомого значения, мы перемещаем числа Фибоначчи на два шага вниз в последовательности, получая следующие значения:

*fibM = 3*

*fibM\_minus\_1 = 2*

*fibM\_minus\_2 = 1*

*index = 4*

* Затем мы проверяем элемент *lys[5]*, где 5 — это минимум из двух значений: *index + fibM\_minus\_2 (4+1)* и длина массива минус 1 (11-1) . Значение *lys[5]* равно 6, и это наше искомое значение!

Временная сложность поиска Фибоначчи равна ***O(log n)***. Она такая же, как и у бинарного поиска. Это означает, что алгоритм в большинстве случаев работает быстрее, чем линейный поиск и jump search.

Поиск Фибоначчи можно использовать, когда у нас очень большое количество искомых элементов и мы хотим уменьшить неэффективность, связанную с использованием алгоритма, основанного на операторе деления.

Дополнительным преимуществом использования поиска Фибоначчи является то, что он может вместить входные массивы, которые слишком велики для хранения в кэше процессора или ОЗУ, потому что он ищет элементы с увеличивающимся шагом, а не с фиксированным.

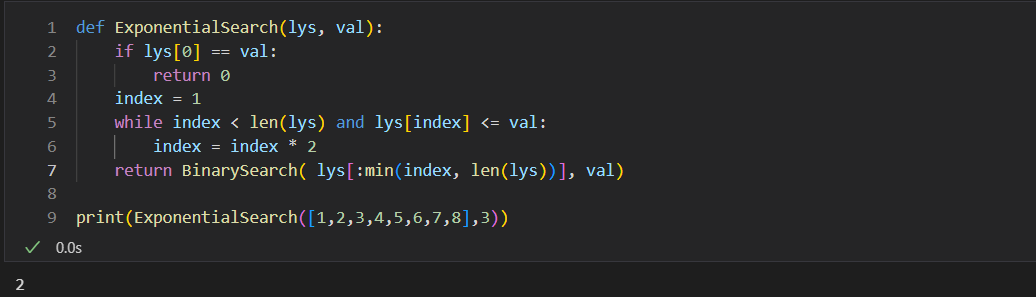
## **Экспоненциальный поиск**

Экспоненциальный поиск — это еще один алгоритм поиска, который может быть достаточно легко реализован на Python, по сравнению с jump search и поиском Фибоначчи, которые немного сложны. Он также известен под названиями galloping search, doubling search и Struzik search.

Экспоненциальный поиск зависит от бинарного поиска для выполнения окончательного сравнения значений. Алгоритм работает следующим образом:

* Определяется диапазон, в котором, скорее всего, будет находиться искомый элемент.
* В этом диапазоне используется двоичный поиск для нахождения индекса элемента.

Реализация алгоритма экспоненциального поиска на Python:



Рассмотрим работу алгоритма пошагово.

* Проверяем, соответствует ли первый элемент списка искомому значению: поскольку *lys[0]* равен 1, а мы ищем 3, мы устанавливаем индекс равным 1 и двигаемся дальше.
* Перебираем все элементы в списке, и пока элемент с текущим индексом меньше или равен нашему значению, умножаем значение индекса на 2:
  1. *index = 1, lys[1]* равно 2, что меньше 3, поэтому значение *index* умножается на 2 и переменной *index* присваивается значение 2.
  2. *index = 2, lys[2]* равно 3, что равно 3, поэтому значение *index* умножается на 2 и переменной *index* присваивается значение 4.
  3. *index = 4, lys[4]* равно 5, что больше 3. Условие выполнения цикла больше не соблюдается и цикл завершает свою работу.
* Затем выполняется бинарный поиск в полученном диапазоне ([срезе](https://pythonist.ru/metody-spiskov-python/)) *lys[:4]*. В Python это означает, что подсписок будет содержать все элементы до 4-го элемента, поэтому мы фактически применяем бинарный поиск.

Экспоненциальный поиск выполняется за время ***O(log i)***, где ***i*** — индекс искомого элемента. В худшем случае временная сложность равна ***O(log n)***, когда искомый элемент — это последний элемент в массиве (***n*** — это длина массива).

Экспоненциальный поиск работает лучше, чем бинарный, когда искомый элемент находится ближе к началу массива. На практике мы используем экспоненциальный поиск, поскольку это один из наиболее эффективных алгоритмов поиска в [неограниченных или бесконечных массивах](https://stackoverflow.com/questions/21510201/what-is-an-unbounded-array).

# **Алгоритмы сортировки**

Сортировка — это когда мы упорядочиваем элементы в итерируемом объекте по возрастанию, убыванию или другому критерию.

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 сортировки:**

* **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**

**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**

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**

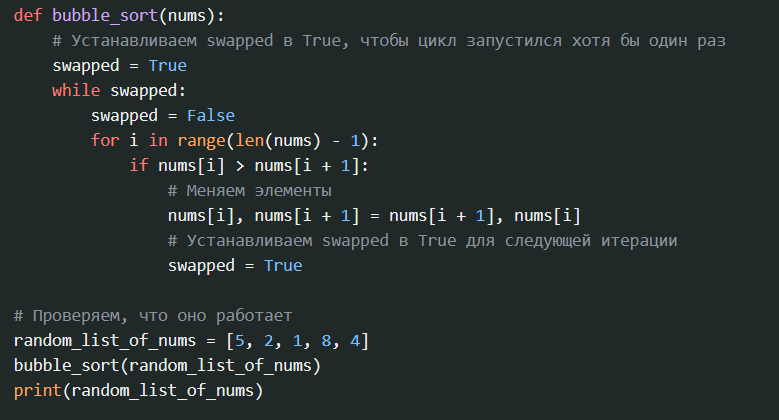
Сначала сравниваются первые два элемента списка. Если первый элемент больше, они меняются местами. Если они уже в нужном порядке, оставляем их как есть. Затем переходим к следующей паре элементов, сравниваем их значения и меняем местами при необходимости. Этот процесс продолжается до последней пары элементов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

### **Реализация**

 Алгоритм работает в цикле while и прерывается, когда элементы ни разу не меняются местами. Вначале присваиваем swapped значение True, чтобы алгоритм запустился хотя бы один раз.

### **Время сортировки**

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

## **Сортировка выборкой**

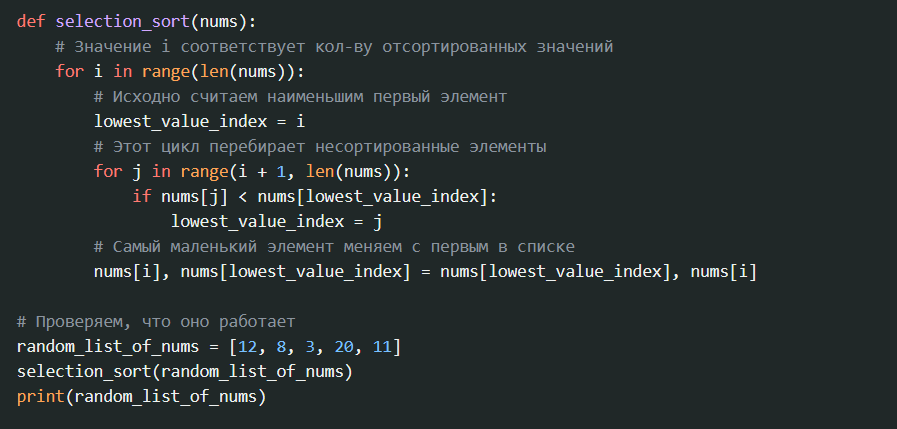
Этот алгоритм сегментирует список на две части: отсортированную и неотсортированную. Наименьший элемент удаляется из второго списка и добавляется в первый.

### **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.

### **Реализация**



По мере увеличения значения i нужно проверять меньше элементов.

### **Время сортировки**

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

## **Пирамидальная сортировка**

Также известна как сортировка кучей. Этот популярный алгоритм, как и сортировки вставками или выборкой, сегментирует список на две части: отсортированную и неотсортированную. Алгоритм преобразует второй сегмент списка в структуру данных «куча» (heap), чтобы можно было эффективно определить самый большой элемент.

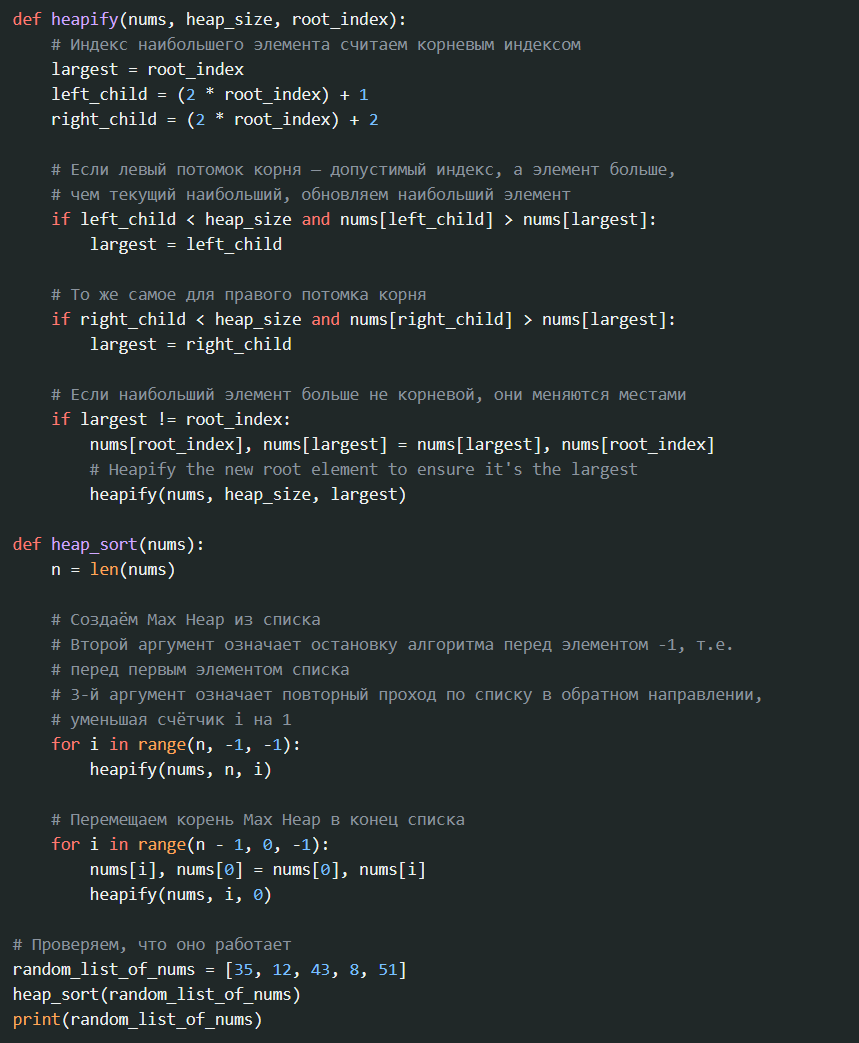
### **Алгоритм**

Сначала преобразуем список в Max Heap — бинарное дерево, где самый большой элемент является вершиной дерева. Затем помещаем этот элемент в конец списка. После перестраиваем Max Heap и снова помещаем новый наибольший элемент уже перед последним элементом в списке.

Этот процесс построения кучи повторяется, пока все вершины дерева не будут удалены.

### **Реализация**

Создадим вспомогательную функцию *heapify*() для реализации этого алгоритма:



### **Время сортировки**

В среднем время сортировки кучей составляет O(n log n), что уже значительно быстрее предыдущих алгоритмов.

## **Сортировка вставками**

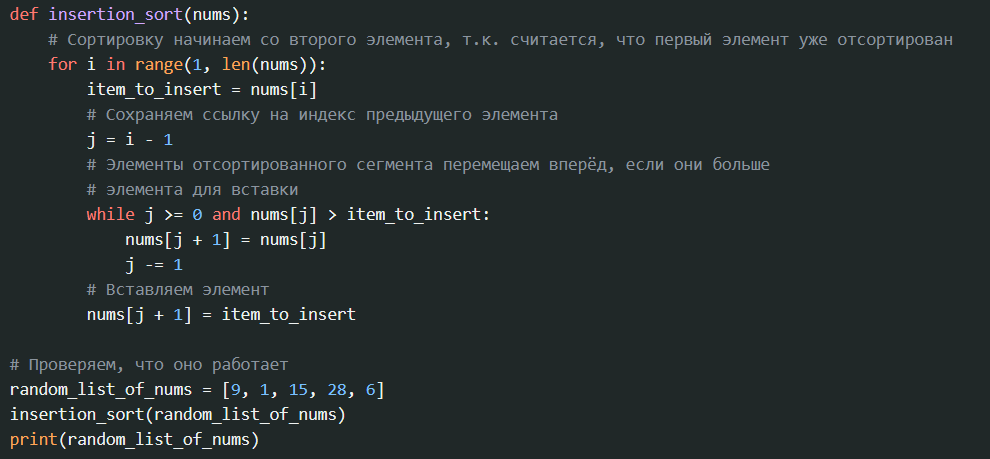
Как и сортировка выборкой, этот алгоритм сегментирует список на две части: отсортированную и неотсортированную. Алгоритм перебирает второй сегмент и вставляет текущий элемент в правильную позицию первого сегмента.

### **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.

### **Реализация**



### **Время сортировки**

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

## **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.

### **Реализация**



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).

## **Быстрая сортировка (Quicksort)**

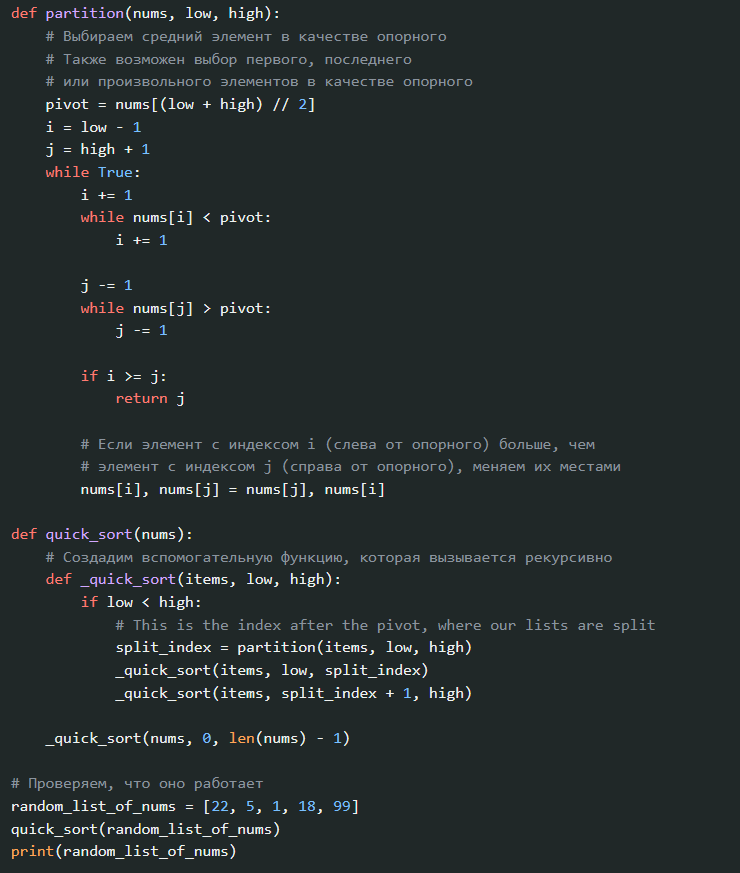
Этот алгоритм также относится к алгоритмам «разделяй и властвуй». Его используют чаще других алгоритмов, описанных в этой статье. При правильной конфигурации он чрезвычайно эффективен и не требует дополнительной памяти, в отличие от сортировки слиянием. Массив разделяется на две части по разные стороны 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.

### **Реализация**

Существует много вариаций данного метода. Способ разбиения массива, рассмотренный здесь, соответствует схеме Хоара (создателя данного алгоритма).



### **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).

# **Заключение**

Существует множество возможных способов поиска элемента в коллекции. В этой статье мы обсудили несколько алгоритмов поиска и их реализации на Python.

Выбор используемого алгоритма зависит от данных, с которыми вы будете работать. Это ваш входной массив, который мы называли *lys* во всех наших реализациях.

* Если вы хотите выполнить поиск в несортированном массиве или найти первое вхождение искомой переменной, то лучшим вариантом будет линейный поиск.
* Если вы хотите выполнить поиск в отсортированном массиве, есть много вариантов, из которых самый простой и быстрый — это бинарный поиск.
* Если у вас есть отсортированный массив, в котором вы хотите выполнить поиск без использования оператора деления, вы можете использовать либо jump search, либо поиск Фибоначчи.
* Если вы знаете, что искомый элемент, скорее всего, находится ближе к началу массива, вы можете использовать экспоненциальный поиск.
* Если ваш отсортированный массив равномерно распределен, то самым быстрым и эффективным будет интерполяционный поиск.

Если вы не уверены, какой алгоритм использовать для отсортированного массива, просто протестируйте каждый из них при помощи библиотеки time и выберите тот, который лучше всего работает с вашим dataset’ом.

Также вы познакомились с семью различными алгоритмами сортировок и их реализациями на Python. Масштаб сравнения и количество перестановок, которые выполняет алгоритм вместе со средой выполнения кода, будут определяющими факторами в производительности. В реальных приложениях Python рекомендуется использовать встроенные функции сортировки, поскольку они реализованы именно для удобства разработчика.

# **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

9. https://pythonist.ru/algoritmy-poiska-na-python/

10. https://tproger.ru/translations/sorting-algorithms-in-python