## **Поиск в отсортированном списке**

import random

arr = random.sample(range(0, 10), 10)

print (arr)

def search\_max\_in\_list(arr):

max = arr[0]

index = 0

for i, v in enumerate(arr):

if v > max:

max = v

index = i

print (f'Максимальный элемент в списке равен {max} с ключом {index}')

search\_max\_in\_list(arr)

## **Бинарный поиск**

Двоичный, или бинарный, поиск значения в списке или массиве используется только для упорядоченных последовательностей, то есть отсортированных по возрастанию или убыванию. Заключается в определении, содержит ли массив искомое значение, а также в определение места его нахождения.

### Алгоритм

1. Находится средний элемент последовательности. Для этого первый и последний индексы связываются с переменными, а индекс среднего элемента вычисляется.
2. Значение среднего элемента сравнивается с искомым значение. В зависимости от того, больше оно или меньше значения среднего элемента, дальнейший поиск будет происходить только в левой или только в правой половинах массива. Если значение среднего элемента оказывается равным искомому, поиск завершается.
3. Иначе одна из границ исследуемой последовательности сдвигается. Если искомое значение больше значения среднего элемента, то нижняя граница сдвигается за средний элемент на один элемент справа. Если искомое значение меньше значения среднего элемента, то верхняя граница сдвигается на элемент перед средним.
4. Снова находится средний элемент теперь уже в выбранной половине. Описанный выше алгоритм повторяется для данного среза.

sorted\_list = [x for x in range(10, 30)]

print (sorted\_list)

def binary\_search(arr, key):

left = 0

right = len(arr)-1

while left <= right:

middle = (left + right)//2

if arr[middle] == key:

print (f'Индекс элемента {key} равен {middle}')

return middle

elif arr[middle] < key:

left = middle+1

elif arr[middle] > key:

right = middle-1

print ('Элемент не найден')

return False

binary\_search(sorted\_list, 29)

## **LIFO. Проверка скобок**

def is\_valid(str):

brackets = {

'(':')',

'{':'}',

'[':']'

}

stack = []

for bracket in str:

if bracket in brackets:

stack.append(bracket)

else:

if not stack:

return False

if bracket != brackets[stack.pop()]:

return False

return (len(stack) == 0)

r = is\_valid('()))')

print (r)

## 

# **ЦИКЛ В ЦИКЛЕ**

## **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».

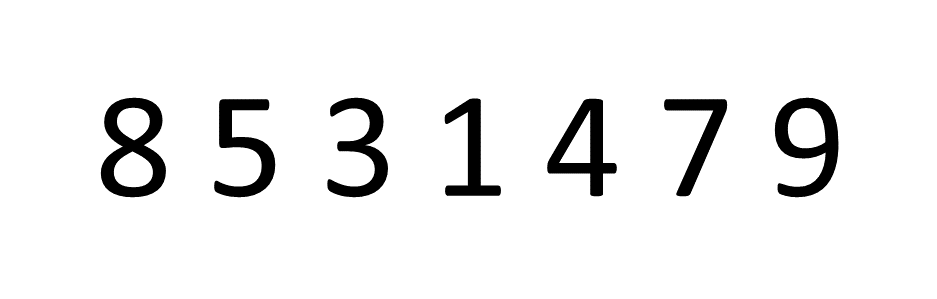
### Алгоритм

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.

### Время сортировки

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



import random

arr = random.sample(range(0, 20), 10)

def bubble\_sort(a):

N = len(a)

for i in range(N-1):

for j in range(N-i-1):

if a[j] > a[j+1]:

a[j], a[j+1] = a[j+1], a[j]

print(a)

bubble\_sort(arr)

## **Сортировка выборкой (select sort)**

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

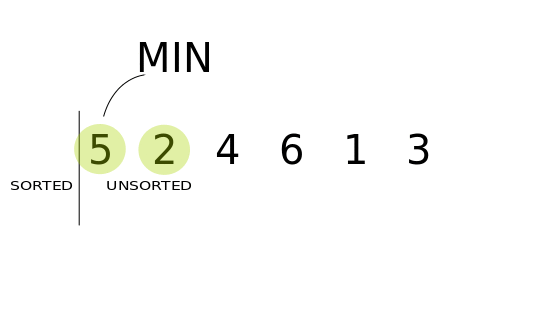
### Алгоритм

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.

### Время сортировки

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



import random

arr = random.sample(range(0, 20), 10)

print (arr)

def selection\_sort(arr):

for i in range(len(arr)):

min = i

for j in range(i + 1, len(arr)):

if arr[j] < arr[min]:

min = j

arr[min], arr[i] = arr[i], arr[min]

print (arr)

selection\_sort(arr)

## **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.

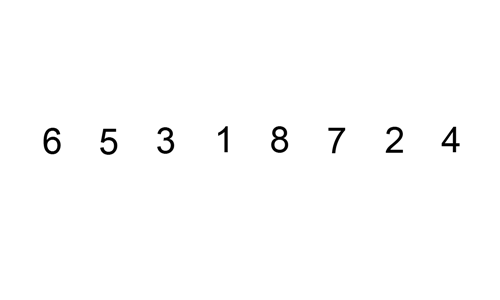
### Алгоритм

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.

### Время сортировки

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



#Сортировка вставками

import random

arr = random.sample(range(0, 20), 10)

print (arr)

def insertion\_sort(a):

N = len(a)

for i in range(1, N):

for j in range(i, 0, -1):

if a[j] < a[j-1]:

a[j], a[j-1] = a[j-1], a[j]

return a

res = insertion\_sort(arr)

print (res)

## 

# **Слияние списков и рекурсия while i<len(a) and j<len(b)**

## **Слияние отсортированных списков**

a = [x for x in range(5, 15, 4)]

b = [x for x in range(0, 10, 2)]

print (a, b, sep='\n')

def merge\_lists(a, b):

i = 0

j = 0

res = []

while i<len(a) and j<len(b):

if a[i] < b[j]:

res.append(a[i])

i += 1

else:

res.append(b[j])

j += 1

res += a[i:] + b[j:]

print(res)

merge\_lists(a, b)

## **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

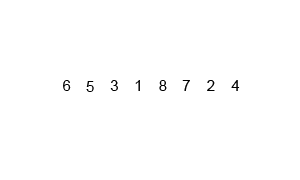
Список рекурсивно разделяется пополам, пока в итоге не получатся списки размером в один элемент. Массив из одного элемента считается упорядоченным. Соседние элементы сравниваются и соединяются вместе. Это происходит до тех пор, пока не получится полный отсортированный список.

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.

### Время сортировки

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

1. Несортированный список последовательно делится на N списков, где каждый включает один «несортированный» элемент, а N — это число элементов в оригинальном массиве.
2. Списки последовательно сливаются группами по два, создавая новые отсортированные списки до тех пор, пока не появится один финальный отсортированный список.



import random

arr = random.sample(range(0, 20), 10)

print (arr)

def merge(a,b):

i = 0

j = 0

res = []

while i < len(a) and j < len(b):

if a[i] < b[j]:

res.append(a[i])

i += 1

else:

res.append(b[j])

j += 1

res += a[i:] + b[j:]

return res

def merge\_sort(arr):

if len(arr) <= 1:

return (arr)

else:

mid = len(arr)//2

left = merge\_sort(arr[:mid])

right = merge\_sort(arr[mid:])

return merge(left, right)

print(merge\_sort(arr))

## **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.

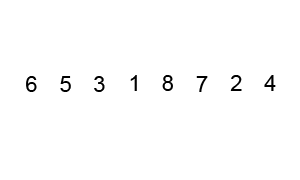
### Алгоритм

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.

### 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)*.



import random

arr = random.sample(range(0, 20), 10)

print (arr)

def quick\_sort(arr):

if len(arr) <= 1:

return (arr)

sep = arr[0]

# sep = arr[random.randint(0, len(arr)-1)]

left = [x for x in arr if x < sep]

right = [x for x in arr if x > sep]

mid = [x for x in arr if x == sep]

return quick\_sort(left) + quick\_sort(mid) + quick\_sort(right)

print(quick\_sort(arr))