**Модуль 5. Алгоритмы сортировки и поиска  (4 ак. ч.)**

* Понятие сложности алгоритма.
* Основные алгоритмы сортировки и поиска.
* Практикум. Поиск информации по заданному критерию.

Оглавление

[Введение 1](#_Toc146848666)

[Понятие сложности алгоритма. 1](#_Toc146848667)

[Какова сложность следующих алгоритмов? 9](#_Toc146848668)

## Введение

Алгоритмы сортировки и поиска — важный класс алгоритмов. Они могут использоваться как самостоятельно, так и в качестве основы для более сложных алгоритмов, о которых мы поговорим в следующих главах.

Мы познакомимся с различными типами алгоритмов сортировки, сравним их производительность при различных подходах к разработке; затем подробно рассмотрим несколько алгоритмов поиска. Наконец, мы разберем изученные алгоритмы на практическом примере.

К концу главы вы научитесь оценивать сильные и слабые стороны различных алгоритмов сортировки и поиска.

Такие алгоритмы — основа для большинства более сложных алгоритмов. Знакомство с ними поможет вам понять современные сложные алгоритмы. Итак, в нас ждут:

## Понятие сложности алгоритма.

Одно решение - разные алгоритмы

def palindrom\_one(string):  
 reverse = ""  
 i = len(string) - 1  
 while i >= 0:  
 reverse += string[i]  
 i -= 1  
 if string.lower() == reverse.lower():  
 return True  
 return False  
  
  
def palindrom\_two(string):  
 mid = len(string) // 2  
 j = len(string) - 1  
 for i in range(mid):  
 if string[i] != string[j]:  
 return False  
 j -= 1  
 return True  
  
  
def palindrom\_three(string):  
 s1 = list(string)  
 s2 = s1.copy()  
 s2.reverse()  
 if s1 == s2:  
 return True  
 return False

Все 3 функции делаю одно и тоже (НАГАН, А РОЗА УПАЛА)

Есть ли разница?

Все 3 работают. Задача выбрать какой мы будем использовать в нашем приложении.

В одном случае строка состоит из 5 символов, во втором из тысячи или миллиона.

Конечно, мы можем использовать возможности модуля timeit

s = 'Молоко делили ледоколом'

start\_time = timeit.default\_timer()  
for i in range(10):  
 palindrom\_one(s)  
print(timeit.default\_timer() - start\_time)  
  
start\_time = timeit.default\_timer()  
for i in range(1000):  
 palindrom\_two(s)  
print(timeit.default\_timer() - start\_time)  
  
start\_time = timeit.default\_timer()  
for i in range(1000):  
 palindrom\_three(s)  
print(timeit.default\_timer() - start\_time)

Результат:

0.0042372000170871615

0.0005130000645294785

0.001256900024600327

Результат для s = 'Молоко делили ледоколом':

0.8191749999532476

0.0005519998958334327

0.08907839993480593

И предположить, что вторая функция отработает быстрее. Чуть позже вернемся к этим функциям

Начнем с простого:

def sum\_one(n):  
 sum = 0  
 for i in range(1, n + 1):  
 sum += i  
 return sum  
  
  
def sum\_two(n):  
 sum = (n \* (n + 1)) / 2  
 return sum

Обе функции считаю сумму последовательности. Вопрос тот же самый. Какую выбрать? Почему? Попробуем это замерить.

Импортируем из модуля timeit класс Timer

from timeit import Timer

Синтаксис следующий

t = Timer("function\_name(10)", "from \_\_main\_\_ import function\_name")

в таймере формируем запрос к функции.

А с помощью timeit вызываем функцию заданное количество раз.  
  
t.timeit(number=1)

Начинаем исследовать

t = Timer("sum\_one(10000)", "from \_\_main\_\_ import sum\_one")  
print(f'{t.timeit(number=1):.6f}')  
  
t = Timer("sum\_one(100000)", "from \_\_main\_\_ import sum\_one")  
print(f'{t.timeit(number=1):.6f}')  
  
t = Timer("sum\_one(1000000)", "from \_\_main\_\_ import sum\_one")  
print(f'{t.timeit(number=1):.6f}')

Результат:

0.000432

0.004908

0.052329

При увеличение n – время увеличивается, причем линейно!

Теперь тоже самое проделаем со второй функцией

t = Timer("sum\_two(10000)", "from \_\_main\_\_ import sum\_two")  
print(f'{t.timeit(number=1):.7f}')  
  
t = Timer("sum\_two(100000)", "from \_\_main\_\_ import sum\_two")  
print(f'{t.timeit(number=1):.7f}')  
  
t = Timer("sum\_two(1000000)", "from \_\_main\_\_ import sum\_two")  
print(f'{t.timeit(number=1):.7f}')

Результат:

0.0000037

0.0000014

0.0000012

Мы видим, что функции по барабану, какого размера n! Понятно, что есть зависимость от операционной системы, от железа.

t = Timer("sum\_one(10)", "from \_\_main\_\_ import sum\_one")  
print(f'{t.timeit(number=1000):.6f}')  
  
t = Timer("sum\_two(10)", "from \_\_main\_\_ import sum\_two")  
print(f'{t.timeit(number=1000):.6f}')

Результат:

0.000555

0.000112

А на n = 100\_000, результат:

4.904088

0.000132

t = Timer("sum\_one(100\_000)", "from \_\_main\_\_ import sum\_one")  
print(f'{t.timeit(number=10):.6f}')  
  
t = Timer("sum\_two(100\_000)", "from \_\_main\_\_ import sum\_two")  
print(f'{t.timeit(number=10):.6f}')

Результат:

0.050431

0.000005

Во сколько раз разница, в 10\_000 раз! На 5 порядков

Вернемся в первой функции, которой без разницы размерность n.

Но наша задача уметь оценивать самим эффективность работы алгоритмов. Или по другому их сложность! Ручками.

Например, рассмотрим первую функцию

def sum\_one(n):  
 sum = 0 #1 -присваивание  
 for i in range(1, n + 1):  
 sum += i # n-присваиваний  
 return sum

Итого: сложность равна 1 + n

Но нам не все операции интересны. Нам интересна доминирующая часть функции, которая перекрывает все остальное.

Если представить, что у вас есть премия 100 рублей при зарплате в 1000 рублей – это неплохо. У при зарплате в 100\_000 р. вы её не заметите.

Т.е. мы учитываем только доминирующую часть. В нашем случае это n.

Порядок этой величины называют big O.

Таким образом сложность этого алгоритма О(n)

А теперь посмотрим на вторую функцию

def sum\_two(n):  
 sum = (n \* (n + 1)) / 2  
 return sum

то, мы увидим, что всегда у нас выполняется одна операция. Сложность O(1).

Гениально. Всегда нужно стремится к такой эффективности. Уходить от переборных методов. Изучать Декстра, жадные алгоритмы, динамику.

O(1) никогда не зависит от количества входных данных.

Даже если мы для второго алгоритма будем иметь такую формулу:

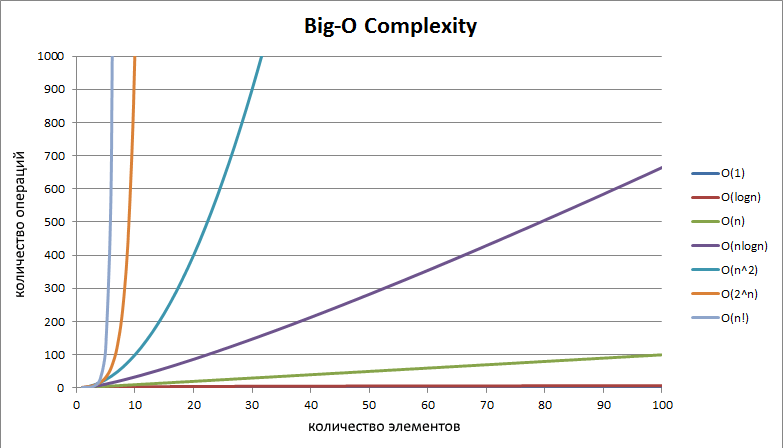
100 + n\*\*2 + 25

Учитывать ли числа 100 и 25. Конечно если n=1, то да. Но по мере увеличения n нам наплевать и мы оставляем доминирующую часть. Всё равно сложность останется O(1).

Хрестоматийно, каждый алгоритм имеет свой худший, лучший и средний случай.

К примеру, мне нужно ехать из точки А в точку B. Можно доехать без пробок за 30 мин, а с пробками 45 мин, а можно вообще не доехать, или приехать в место окончательной регистрации граждан.

Технически сложность алгоритма может быть любой, хоть n в кубе.

Но очень часто в 99% - случаем нам попадутся вот такие сложности

O(1) – константа, линия уходит вертикально вверх, t -константа. Они редки.

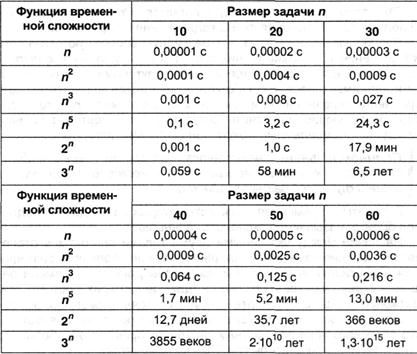
1. Константая сложность
2. Логарифмическая (логарифм n)
3. Линейная
4. Линейно-логарифмическая
5. Квадратичная, кубическая
6. Экспоненциальная (2\*\*n)
7. Факториальная

Чем ниже чем ужаснее.

Но на малых n – разницы нет. Факториал уходит в бесконечность.

Учимся дальше это дело использовать.

Ниже пример задачи, который выполняется с какой-то скоростью (временем). Мы видим, что правильный алгоритм способен на 15 порядков работать быстрее, а при малых n разницу мы не видим.



Важно уметь не только определять сложность алгоритма, но и его рассчитывать.

Посмотрим на абстрактный пример:

def test(n):  
 a = 1  
 b = 2  
 c =3   
 for i in range(n):  
 for j in range(n):  
 a = i \* i  
 b = j \* j  
 c = i \* j  
 x = 4  
 y = 5  
 for i in range(n):  
 x = a \* b + 42  
 y = c \* i  
 z = 100

Какова сложность данной функции (bigO)

3 + 3n\*\*2 + 2 + 2n + 1

1. Находим 3 присваивания
2. Далее еще 3 присваивания в цикле, который находится еще в одном цикле

А цикл в цикле – это квадрат O(n2). Значит эта часть дает 3n\*\*2

1. Далее снова 2 присваивания (+2)
2. Далее снова 2 шага в обычном цикле 2n

Таким образом размер задания

T(n) = 3 + 3n\*\*2 + 2 + 2n + 1

Это промежуточный шаг. А из него мы считаем сложность. Но по определенным правилам (математики)

3 + 3n\*\*2 + 2 + 2n + 1 = 3n\*\*2 + 2n + 6

И опять, константы можно проигнорировать, остается 3n\*\*2 + 2n

Дальше оставляем доминирующую часть: 3n\*\*2

Ну и 3 несущественна, при большом n

Таким образом получаем:

**T(n) = O(n\*\*2)**

Обратите внимание, что это всего лишь концепт.

Вы должны уметь визуально вычленять самую сложную доминирующую часть

1. count += 1 # O(1)
2. цикл for, while # O(n)
3. цикл в цикле О(n\*\*2)

Пара примеров:

def pow\_one(base, exp):  
 result = 1  
 while exp > 0:  
 result \*= base  
 exp -= 1  
 return result

**T(n) = 1 + 2n = n => O(n)**

def pow\_two(base, exp):  
 result = 1;  
 while exp > 0:  
 if exp % 2 == 0:  
 base \*= base  
 exp //= 2  
 else:  
 result \*= base  
 exp -= 1  
 return result

1 + 2 (n/2) = > O (log n)

А теперь подумайте, проанализируйте, что делают эти функции.

Вторя в 2 раза больше, но работает гораздо быстрее. Посмотрите на график.

Мини практикум:

6 заданий

## Какова сложность следующих алгоритмов?

def one(array):  
 result = 0;  
 for i in array:  
 for j in array:  
 result += i \* j  
 return result

# O(n\*\*2)  
  
def two(array):  
 result = 0  
 for i in array:  
 result += 1  
 for j in array:  
 result -= 1  
 return result  
  
# O(n)

def three(n):  
 i = 0  
 while n > 0:  
 i += 2  
 n //= 2  
 return i  
  
# O(log n)

def four(n):  
 for i in range(n):  
 j = 0  
 cnt = 50 \* n # если поменять на n \*\*2 то O(n\*\*3)  
 sum = 0  
 while j < cnt:  
 j += 1  
 sum += 1  
 return sum

# O(n\*\*2)  
  
def five(n):  
 for i in range(n):  
 while i % 2 != 0: # банальный if  
 print(i)  
 i -= 1  
 print("Done")  
  
# O(n)

def six(n):  
 for i in range(n):  
 cnt = n \* n  
 for j in range(cnt):  
 if j == 4:  
 return -1  
 print("Done")

# O(1)

Оператор return

Сколько итераций будет выполнять внутренний цикл. Всегда 4!

Поэтому константная сложность.

Задача.

Написать анаграмму. Если все символы из первой строки входят во вторую строку.

def anagram(s1: str, s2: str) -> bool:  
 pass  
  
  
anagram('abc', 'abc')  
anagram('abc', 'bca')  
anagram('abc', 'cba')  
anagram('abc', 'bac')  
anagram('abc', 'cab')

Примеры анаграмм: «пила» и «липа», «пост» и «стоп»

ТЗ

1. Написать функцию
2. Посчитать сложность алгоритма
3. Если он сложный, подумайте, как его оптимизировать.

Напоминаю, мы всегда ориентируемся на наиихудший вариант развития (наихудший алгоритм)

Когда нет других вариантов, используем технику полного перебора.

И получим факториальную сложность – для полного перебора. Для строки из 20 символов

20! – уйдут годы

from math import factorial  
  
print(factorial(20))

способ хрень какая-то

def anagram(s1: str, s2: str) -> bool:  
 if len(s1) != len(s2):  
 return False  
 array = list(s2)  
 pos1 = 0  
 success = True  
 found = False  
 while pos1 < len(s1) and success:  
 pos2 = 0  
 found = False  
 while pos2 < len(array) and not found:  
 if s1[pos1] == array[pos2]:  
 found = True  
 else:  
 pos2 += 1  
 if found:  
 array[pos2] = None  
 else:  
 success = False  
 pos1 += 1  
 return success

print (anagram('abc', 'abc'))  
print(anagram('abc', 'bca'))  
print(anagram('abc', 'cba'))  
print(anagram('abc', 'bac'))  
print(anagram('abc', 'cab'))

способ 2

def anagram(s1: str, s2: str) -> bool:  
 if len(s1) != len(s2):  
 return False  
 array1 = list(s1)  
 array2 = list(s2)  
 array1.sort()  
 array2.sort()  
 i = 0  
 while i < len(s1):  
 if array1[i] != array2[i]:  
 return False  
 i+=1  
 return True

Какая сложность? О(n)

def anagram(s1: str, s2: str) -> bool:  
 if len(s1) != len(s2):  
 return False  
 for i in s1:  
 if i not in s2:  
 return False  
 else:  
 return True

способ 3

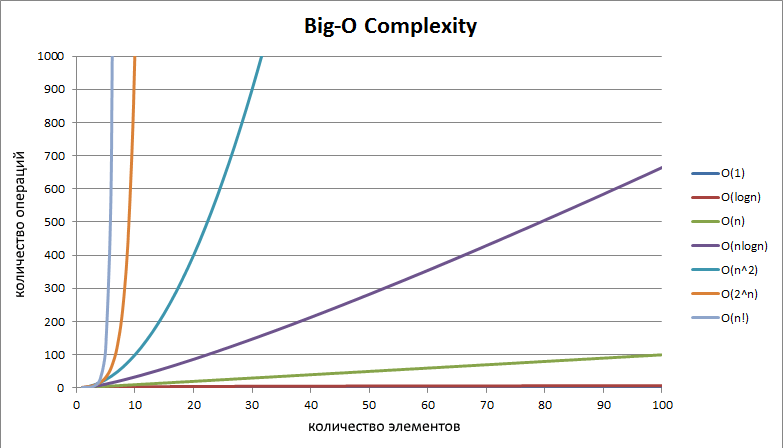
def anagram(s1: str, s2: str) -> bool:  
 if len(s1) != len(s2):  
 return False  
 array1 = list(s1)  
 array2 = list(s2)  
 array1.sort()  
 array2.sort()  
 i = 0  
 if array1==array2:  
 return True

return False

Какова сложность! О(1)?

Встроенные пайтоновские функции тоже расходуют ресурсы языка СИ.

Функция сортировка дает O(n2) или O(n log n)



Как нам найти О(n) хз. Экспериментально

Способ 4 O(n) Клсссический линейный алгоритм

def anagram(s1: str, s2: str) -> bool:  
 if len(s1) != len(s2):  
 return False  
 array1 = [x for x in range(26)]  
 array2 = [x for x in range(26)]  
 a\_pos = 97  
 for char in s1:  
 pos = ord(char) - a\_pos  
 array1[pos] = (array1[pos] or 0) + 1  
  
 for char in s2:  
 pos = ord(char) - a\_pos  
 array2[pos] = (array2[pos] or 0) + 1  
  
 for i in array1:  
 if array1[i] != array2[i]:  
 return False  
 return True

Однако! Не нужно быть фанатами! Выбираем между временем и пространством. Ресурсы памяти!!! А если алфавит китайский из 2000 симоволов. А если из миллиона.

Вернемся к началу к палиндромам!

def palindrom\_one(string):  
 reverse = ""  
 i = len(string) - 1  
 while i >= 0:  
 reverse += string[i]  
 i -= 1  
 if string.lower() == reverse.lower():  
 return True  
 return False

Какая сложность? O(n)  
  
def palindrom\_two(string):  
 mid = len(string) // 2  
 j = len(string) - 1  
 for i in range(mid):  
 if string[i] != string[j]:  
 return False  
 j -= 1  
 return True

Какая сложность? O(n / 2) или O(n)? Условно О(n)  
  
def palindrom\_three(string):  
 s1 = list(string)  
 s2 = s1.copy()  
 s2.reverse()  
 if s1 == s2:  
 return True  
 return False

Какая сложность? Copy O(n)

Что выбрать? Программист выберет 4. Все встроенные функции работают на С++.

Не нужно изобретать велосипед.

Как на СИ перевернуть строку. Лабораторная работа!

Бездумное использование чужих модулей! Тестим на время, на память, на ресурсы. Вася мог написать неэффективный алгоритм.

А он погуглил его из интернеты и затырил в свой модуль.

## Выводы¶

Асимптотический анализ — сравнение затрат времени алгоритмов, выполняющих решение некоторой задачи, при больших объемах входных данных

Сложность алгоритма — это функция, позволяющая определить, как быстро увеличивается время работы алгоритма с увеличением объёма данных

Основной оценкой роста, встречающейся в асимптотическом анализе является 'О-большое' — верхняя асимптотическая оценка роста временной функции

# Алгоритмы поиска

## Введение

Поиск информации, хранящейся в различных структурах данных, является важной частью практически каждого приложения.

Существует множество различных алгоритмов, которые можно использовать для поиска. Каждый из них имеет разные реализации и напрямую зависит от структуры данных, для которой он реализован.

Умение выбрать нужный алгоритм для конкретной задачи является ключевым навыком для разработчиков. Именно правильно подобранный алгоритм отличает быстрое, надежное и стабильное приложение от приложения, которое падает от простого запроса.

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.

Операторов членства достаточно, если нам нужно только определить, существует ли подстрока в данной строке, или пересекаются ли две строки, два списка или кортежа с точки зрения содержащихся в них объектов.

В большинстве случаев помимо определения, наличествует ли элемент в последовательности, нам нужна еще и позиция (индекс) элемента. Используя операторы членства, мы не можем получить ее.

Последовательный (линейный) поиск

'apple' in ['orange', 'apple', 'grape']  
  
't' in 'python'  
  
'q' not in 'python'

Существует множество алгоритмов поиска, которые не зависят от встроенных операторов и могут использоваться для более быстрого и/или эффективного поиска значений. Кроме того, они могут дать больше информации (например, о позиции элемента в коллекции), а не просто определить, есть ли в коллекции этот элемент.

## 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.

Обычно, то что нам нужно найти, называют ключом. В обыденной жизни ключом является ключ словаря {'key': value}.

Для списка ключом является

0 1  
['a', 'b']

Значение ‘b’

Алгоритм линейного последовательного поиска довольно прост. Функция принимает сам список и ключ (значение) который мы ищем

## Последовательный (линейный) поиск

def linear\_search(lst, key):  
 for idx, item in enumerate(lst):  
 if key == item:  
 return idx  
 return -1

Таким образом, начиная с первого элемента, от значения к значению, мы слева-направо, следуя внутреннему порядку последовательности.

Цикл for и функция enumerate позволяет получать сразу два значения.

Функция позволяет получить позицию (индекс ключа).

На каждой итерации мы тупо сравниваем.

В случае совпадения мы возвращаем индекс, по которому он находится, пока не достигнем последнего элемента.

Когда мы не нашли элемент в последовательности, то возвращаем -1. Т.е. она не содержит.

t = [1, 2, 32, 43, 54]  
print(linear\_search(t,32))

Протестим алгоритм линейного поиска.

Далее мы должны проанализировать этот алгоритм.

Маленький, не означает быстрый.

Что нужно взять за шаг? Количество сравнений.

Учитываем, что список неупорядоченный. Элементы размещены случайны\\

Вероятность найти нужный нам ключ примерно одинакова. Если в списке n-чисел, то нам потребуется n-сравнений, чтобы убедится, что там нет нужного элемента.

А если этот элемент есть – какие варианты. Если элемент находится на 1-позиции – нам потребуется всего одно сравнение.

В середине n/2 – сравнений.

По сути получаем линейную зависимость

Элемент + - +/-

-----------------------------

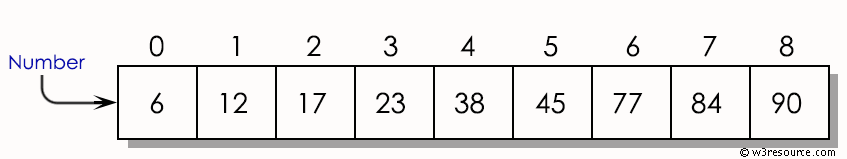
Есть 1 n n/2

Нет n n n

Если элемент присутствует – наилучший случай – это константа, наихудший – это n, средний n/2. При отсутствии элемента – всегда будет наихудшая n.

Для больших n – n/2 стремится к n. Аппроксимируем до сложности O(n).

А если список будет отсортирован,



Линейный поиск в отсортированном списке выглядит теперь вот так:

def ordered\_linear\_search(lst, key):  
 for idx, item in enumerate(lst):  
 if item == key:  
 return idx  
 elif item > key:  
 return -1  
 return -1

Казалось бы нам нужно тоже самое количество сравнений для поиска.

Но обратите внимание, в случае если элемент у нас отсутствует, у нас появляется некий профит. Какой ?

Мы сравниваем последовательно, до какого-то числа.

В этот момент у нас есть информация. Например, мы ищем число 14.

Когда мы дошли до числа 17, мы видим, что 17 явно больше 14. Смысла искать дальше те до конца. elif item > key: return -1

Анализ

Элемент + - +/-

-----------------------------

Есть 1 n n/2

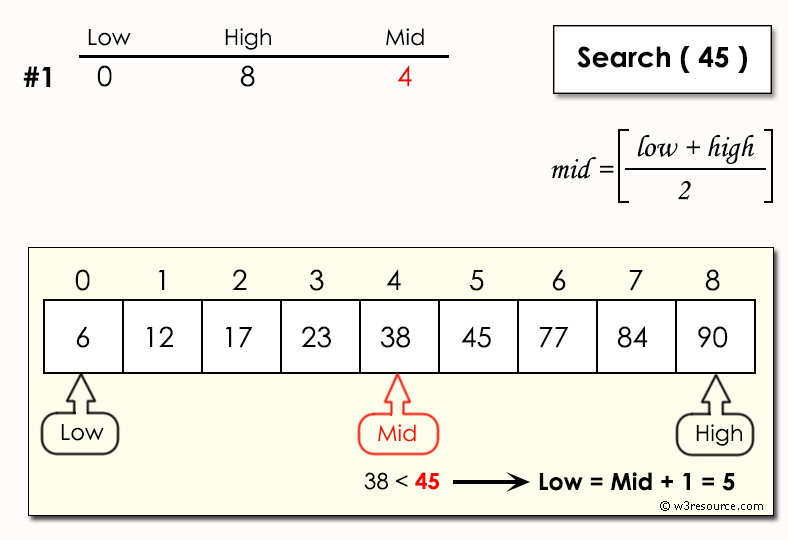
Нет n n n/2

Показывает, что усредненная сложность везде будет n/2. Чуть лучше!

Но не идеально. Можем перейти к бинарному (или двоичном) поиску! Но в отсортированном списке.

## Бинарный поиск

Бинарный поиск работает по принципу «разделяй и властвуй». Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма.

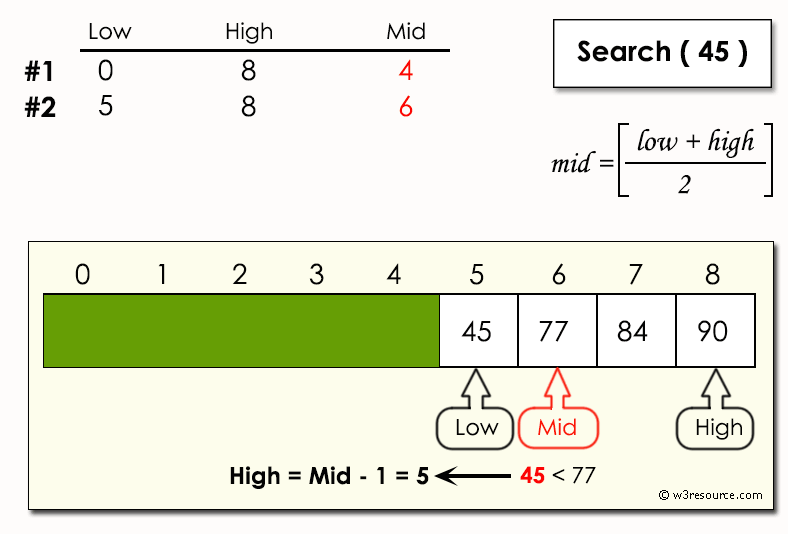


Идея с следующем. Начинаем искать с середины.

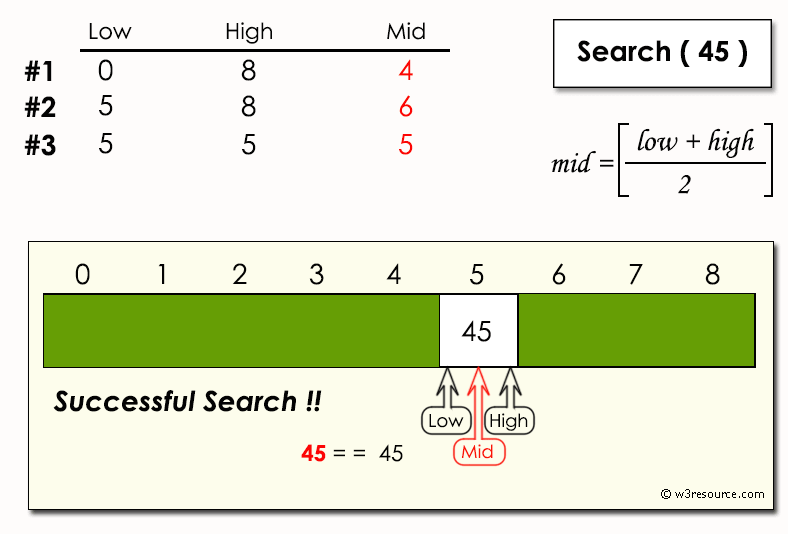
Получаем список, режим его пополам, сравниваем среднее число (Mid).

Если совпало с ключем, то возвращаем его, если нет, то мы смело можем исключить сразу половину (левую или правую) списка.

Т.е. если середина списка 38, а мы ищем 45, которое расположено в правой части списка, то левую половину, включая число 38, мы можем больше не рассматривать (исключить, удалить). Т.к. 45 > 38.



И тоже самое повторяем с оставшейся частью списка, делим пополам, проверяем, удаляем половину списка. Т.о. с третьей попытки мы находим нужный ключ в искомой коллекции.



Реализация на Пайтон.

def binary\_search(lst, key):  
 first = 0  
 last = len(lst) - 1  
  
 while first <= last:  
 mid = (first + last) // 2  
 if lst[mid] == key:  
 return mid  
 else:  
 if key < lst[mid]:  
 last = mid - 1  
 else:  
 first = mid + 1  
 return -1

Известная вещь. Игра угадай число. Как за меньшее число попыток угадать число.

То же использует в себе бинарный (двоичный) поиск.

Для анализа этого алгоритма строим табличку. И получаем сложность O(log n).

Логарифмическая сложность, т.к. каждый раз делим список пополам.

Элемент + - +/-

---------------------------------------

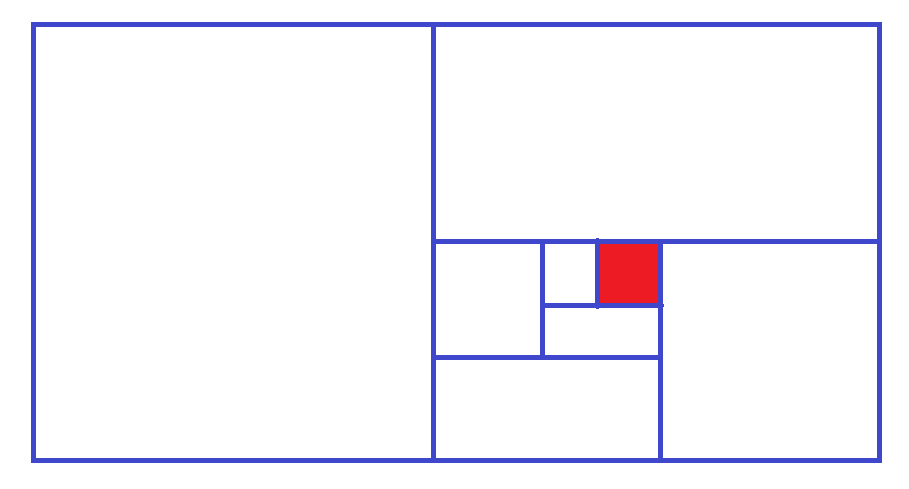
Есть 1 log n log n

Нет log n log n log n

Мы вторглись на территорию стратегии «Разделяй и властвуй».

Хрестоматийная задача – поле, которое нужно разбить на квадраты.

Найти минимальный квадрат



Вообще мы можем в данной ситуации обойтись без цикла. Хотя наш алгоритм имеет хороший O(log n).

Но это в следующем разделе нашего модуля. Использование рекурсии

## Выводы по алгоритмам поиска

Поиск является одной из важнейших процедур обработки структурированной информации

Асимптотическая сложность алгоритма линейного поиска — O(n)

Асимптотическая сложность алгоритма бинарного поиска для упорядоченных списков — O(log n)

## Бинарный поиск с использование рекурсии

def binary\_search(lst, key):  
 if len(lst) == 0:  
 return -1  
 else:  
 mid = len(lst) // 2  
 if lst[mid] == key:  
 return mid  
 else:  
 if key < lst[mid]:  
 return binary\_search(lst[:mid], key)  
 else:   
 return binary\_search(lst[mid+1:], key)

Что мы видим? Нет цикл. Он отсутствует.

Но у нас дважды присутствует вызов этой же функции

return binary\_search(lst[:mid], key)

рекурсивная функция вызывает сама себя.

Проверяем середину, если ключ меньше изменяем диапазон списка срезом до половинки, или от середины до конца.

Пример простой рекурсии

def question(message):  
 if input(message + ': ').lower() == 'всегда':  
 return  
 else:  
 print('Подумайте...')  
 question(message)  
  
question('Ваше кредо?')  
print("Молодец!")

Если неправильный ввод, то здесь мы снова вызываем эту функцию.

Есть функциональные языки, в которых нет циклов while, for и т.д.

Все зацикливания построены на рекурсиях.

Второй пример:

Алгоритм Евклида

def gcd(a, b):  
 if b == 0:  
 return a  
 return gcd(b, a%b)  
  
  
  
print (gcd(102,68))

Наибольшее целое число, на которое делится а и в (102, 68).

Базовым случаем (выходом из рекурсии) является базовое условие

if b == 0:  
 return a

и это условия обязательно должно наступить. Рано или поздно a%b станет равным 0.

Иначе бесконечный цикл, бесконечная рекурсия.

Недостатки рекурсии. Она опасна!

Два классических примера!

Так ли хороша рекурсия?

def factorial1(n): # 5 = 1 \* 2 \* 3 \* 4 \* 5  
 result = 1  
 for i in range(1, n+1):   
 result \*= i  
 return result

def factorial2(n): #5  
 if n == 0:  
 return 1  
 return n \* factorial2(n - 1) #5 \* F(4) \* F(3) \* F(2) \* F(1)

Какая лучше?

Померяем время?

from timeit import Timer  
t1 = Timer("factorial1(10)", "from \_\_main\_\_ import factorial1")  
print("one ", t1.timeit(number=1), "milliseconds")  
t1 = Timer("factorial2(10)", "from \_\_main\_\_ import factorial2")  
print("two ", t1.timeit(), "milliseconds")

Результат:

one 8.499999239575118e-06 milliseconds

two 0.987192699998559 milliseconds

Результат для n = 100:

one 1.2700002116616815e-05 milliseconds

two 12.742952200002037 milliseconds

Глубина рекурсии

import sys

sys.getrecursionlimit()

1000

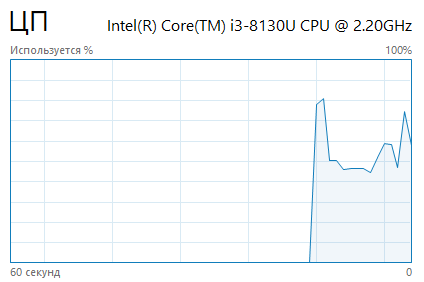
Читы:

sys.setrecursionlimit(10\*\*4)

t1 = Timer("factorial2(100000)", "from \_\_main\_\_ import factorial2")  
print("two ", t1.timeit(), "milliseconds")

[Previous line repeated 997 more times]

RecursionError: maximum recursion depth exceeded



Стек вызова функции факториал

def factorial(3):  
 if 3 == 0: return 1  
 return 3 \* factorial(3 - 1)  
  
 def factorial(2):  
 if 2 == 0: return 1  
 return 2 \* factorial(2 - 1)  
  
 def factorial(1):  
 if 1 == 0: return 1  
 return 1 \* factorial(1 - 1)  
  
 def factorial(0):  
 if 0 == 0: return 1

Пример

Задан алгоритм вычисления функции *F*(*n*), где *n*  — натуральное число:

*F*(*n*)  =  7, при *n* < 7;

*F*(*n*)  =  2*n* + *F*(*n* − 1), если *n* ≥ 7.

Чему равно значение функции *F*(2024) − *F*(2022)?

import sys  
sys.setrecursionlimit(10\*\*6)  
def F(n):  
 if n < 7:  
 return 7  
 else:  
 return 2 \* n + F(n - 1)  
print(F(2024) - F(2022))

К функциональным языкам относятся Haskell, F#, OCaml, ELM, серия языков Lisp, а также Erlang и его потомок Elixir. Иногда сюда же относят Scala и Nemerle, хотя эти языки дают возможность программировать и в функциональном, и в императивном стилях. Они старые и сейчас применяются не так часто, как большинство современных.

## Хвостовая рекурсия

Когда не происходит проблем с памятью

def factorial3(n, acc = 1):  
 if n <= 1:  
 return acc  
 return factorial3(n - 1, n \* acc)  
  
  
  
f = factorial3(5)  
print(f)  
  
  
t1 = Timer("factorial3(100)", "from \_\_main\_\_ import factorial3")  
print("three ", t1.timeit(), "milliseconds")

В хвостовой рекурсии возвращаемое значение является его вызовом

return factorial3(n - 1, n \* acc)

Было

def factorial2(n): #5  
 if n == 0:  
 return 1  
 return n \* factorial2(n - 1)

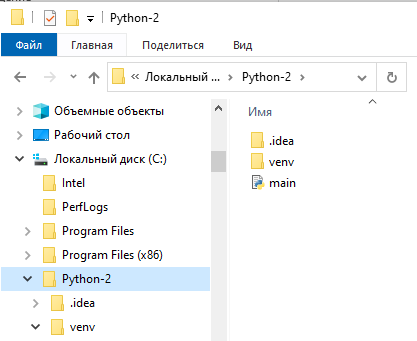
Если, что-то, что можно сделать без рекурсии с помощью цикла, надо ж\делать с помощью цикла.

Рекурсию можно делать в том случае, если неизвестно сложность алгоритма.

Пример операционная система, проводник.

Это дерево отрисовывается с помощью рекурсии. Вложенность неизвестна.

Тоже самое архиваторы, zip, rar



Последний пример:

import sys  
sys.setrecursionlimit(10\*\*6)  
  
def power1(a, n):  
 result = 1  
 for i in range(n):  
 result \*= a  
 return result

Сложность O(n)

def power2(a, n):  
 if n == 0:  
 return 1  
 return a \* power2(a, n - 1)

Сложность O(n) но проблема с памятью  
  
def power3(a, n):  
 return (1 if n == 0  
 else power3(a \* a, n // 2) if n % 2 == 0  
 else a \* power3(a, n - 1))

Сложность O(log n) логарифмическая сложность. Но проблема с памятью, хуже чем у второго.

Куча памяти уходит на создание функции и т.п.

t1 = Timer("power1(2, 10000)", "from \_\_main\_\_ import power1")  
print("one ", t1.timeit(number=100), "milliseconds")  
t1 = Timer("power2(2, 10000)", "from \_\_main\_\_ import power2")  
print("two ", t1.timeit(number=100), "milliseconds")  
t1 = Timer("power3(2, 10000)", "from \_\_main\_\_ import power3")  
print("three ", t1.timeit(number=100), "milliseconds")

результат

one 0.3392762000003131 milliseconds

two 0.47605189999740105 milliseconds

three 0.0039453000063076615 milliseconds

## Мини практикум

#### Кстати, какова сложность следующих алгоритмов?

def fib(n):  
 if n <= 2:  
 return 1  
 else:  
 return fib(n - 1) + fib(n - 2)

# O(2\*n)

x = fib(9)  
print(x)  
  
  
*# нарисовать дерево  
# fib(5)  
# fib(3) + fib(2)  
# 0 1 1 2 3 5 8 13 21 34*

! Реализация без рекурсии

def fib\_list(n):  
 f = [0, 1, 1]  
 for i in range(3, n + 1):  
 f.append(f[i - 1] + f[i - 2])  
 return f[-1]  
  
  
print(fib\_list(9))

! Рекурсия без списка

def fib\_range(n):  
 a, b = 0, 1   
 for \_ in range(n):  
 a, b = b, a + b  
 return a  
  
  
print(fib\_range(9))

# O(n)

def add(n):  
 if n < 3:  
 return n  
 if n % 3 == 0:  
 return add(n - 1) + add(n - 2) + add(n - 3)  
 return n

# O(1) с возможностью деградации O(n)

def F(n):  
 if n == 1:  
 return 0  
 if n <= 3:  
 return 1  
 else:  
 return F(n - 3) + F(n - 2) + F(n - 1)  
  
  
print(F(9))

## Выводы

Рекурсивный алгоритм должен вызывать сам себя

Все рекурсивные алгоритмы должны иметь базовый случай

Рекурсивное решение может быть очень ресурсоёмким

Во многих случаях рекурсию можно заменять итерированием

# Алгоритмы сортировки. Алгоритмы поиска.

Практическое применение. Для начала рассмотрим некоторые алгоритмы сортировки

В эпоху больших данных необходимы современные алгоритмы, чтобы эффективно сортировать и быстро находить элементы в сложных структурах. Выбор стратегии сортировки и поиска зависит от размера и типа данных. Хотя конечный результат будет одинаковым для различных алгоритмов, для эффективного решения реальной проблемы нужно подобрать наиболее подходящий вариант.

С пайтоном все отлично, взяли версию интерпретатора с python.org. Установили на винде, на маке, на линухе или убунте.

И во всех интепретаторах +- используется один и тот же алгоритм сортировки.

Тиме Петерсона.

Но с JS беда. Беда, связанная с браузерами. Интерпетатор то встроен в браузер. А браузеры конкурируют активно между собой не только интерфейсом и юзабилити, но и подкапотными движками, обрабатывающими код JS.

Mozilla: SpiderMonkey, Rhino, Tamarin

Google: V8

В данной части представлены следующие алгоритмы сортировки:

z сортировка пузырьком (bubble sort);

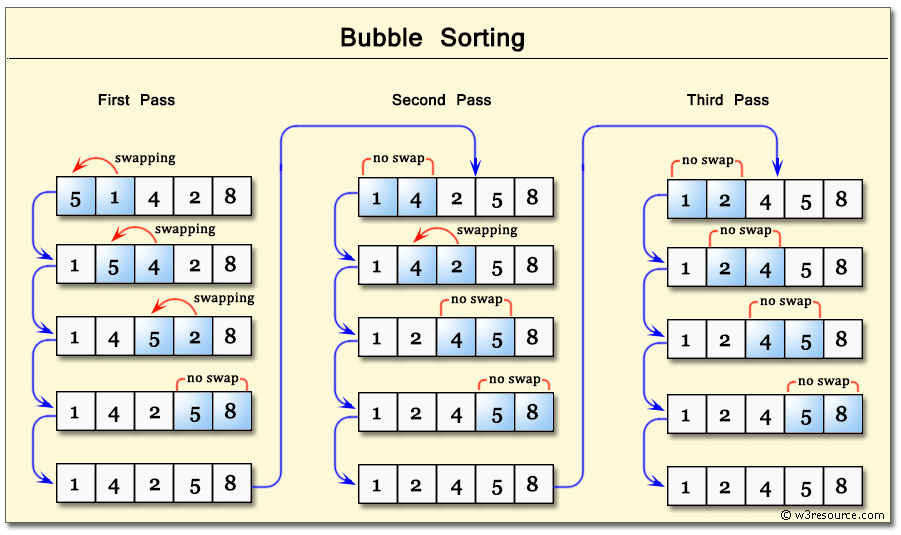
z сортировка вставками (insertion sort);

z сортировка слиянием (merge sort);

z сортировка Шелла (Shell sort); z сортировка выбором (selection sort)

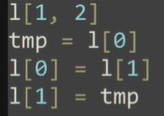
## Сортировка пузырьком

Известный, но никому ненужный – пузырьковый способ сортировки



В нем хорошо показывать работу сортировки. А тем более в Pt есть мощный способ обмена значение переменных a,b = b,a

В СИ мы бы написал:



В основе сортировки пузырьком лежит ряд итераций, называемых проходами (passes).

Для списка размера N нужно совершить N — 1 проходов. Рассмотрим подробно первую итерацию: проход 1. Цель первого прохода — вывести наибольшее значение в конец списка. По мере выполнения алгоритма оно будет постепенно перемещаться вправо по списку.

В процессе сортировки значения соседних элементов сравниваются между собой попарно. Если в паре большее значение находится слева, происходит перестановка (обмен). Это продолжается до тех пор, пока мы не дойдем до конца списка. Работа алгоритма показана на следующей схеме (рис. 3.2)

Реализация

def bubble\_sort(lst):  
 for cnt in range(len(lst) - 1, 0, -1):  
 for i in range(cnt):  
 if lst[i] > lst[i + 1]:  
 lst[i], lst[i + 1] = lst[i + 1], lst[i]  
 return lst  
  
  
l = [5, 4, 3, 3, 2, 1, 4] # а лучше рандомно (0-99)  
a = bubble\_sort(l)  
print(a)

Сложность O(n\*\*2)

Но пузырьковая сортировки имеет свои преимущества. Если список уже отсорированный, мы можем дальше не ходить его сортировать.

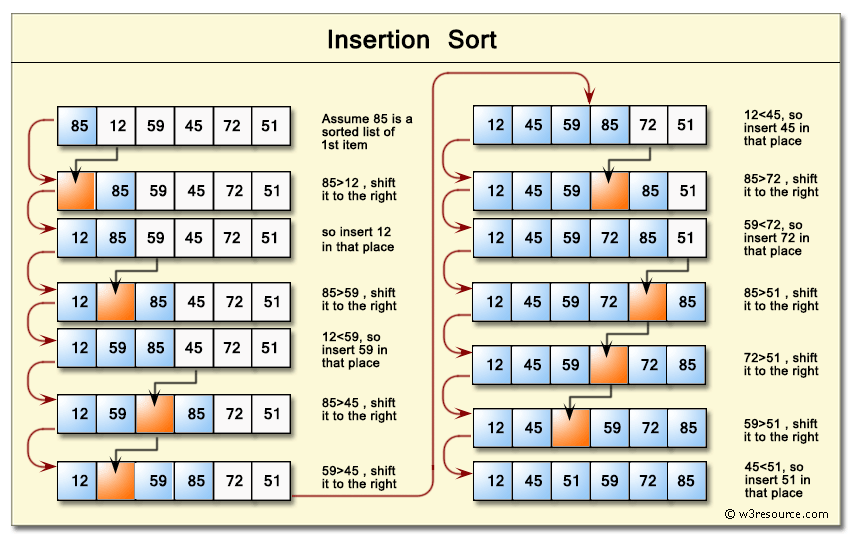
Нужен некий флаг. И счетчик. Реализуем.

def bubble\_sort(lst):  
 n = 0  
 for cnt in range(len(lst) - 1, 0, -1):  
 F = True  
 n+=1  
 for i in range(cnt):  
 if lst[i] > lst[i + 1]:  
 F = False  
 lst[i], lst[i + 1] = lst[i + 1], lst[i]  
 if F:  
 print ('ok', n)  
 break  
 return lst  
  
  
lst = [random.randint(0,100) for \_ in range(10)]  
lst = bubble\_sort(lst)  
print(lst)

Сортировка пузырьком (bubble sort) — это самый простой и медленный алгоритм сортировки. Он спроектирован так, что наибольшее значение перемещается вправо по списку на каждой итерации цикла.

При наихудшем сценарии производительность этого алгоритма равна O(n2 ), поэтому его следует использовать только для небольших наборов данных. Логика

## Следующая сортировка – Вставками



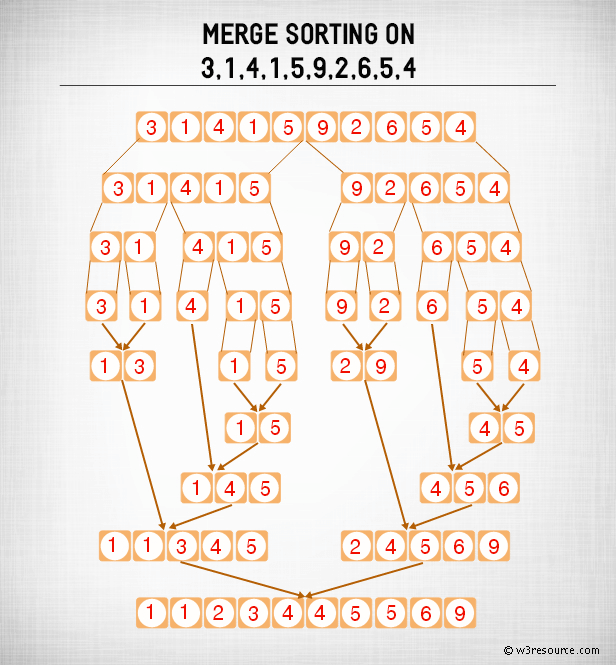
В алгоритме происходит обмен со сдвигом

def insertion\_sort(lst):  
 for i in range(1, len(lst)):  
 current = lst[i]  
 pos = i  
 while pos > 0 and lst[pos - 1] > current:  
 lst[pos] = lst[pos - 1]  
 pos = pos - 1  
 lst[pos] = current  
   
 return lst

Сложность O(n\*\*2)

Память! Операция сдвига быстрее операции обмена. За счет этого операция происходит быстрее.

## Сортировка слиянием



Сортировка слиянием — это эффективный алгоритм сортировки, который основан на разделении и слиянии массива или списка элементов. Он применяет принцип «разделяй и властвуй», разбивая задачу на более простые подзадачи, решение которых затем объединяется в один отсортированный список.

Принцип работы:

Для начала, если размер массива или списка меньше или равен 1, то он считается отсортированным.

Иначе, массив разделяется на две примерно равные части.

Каждая из этих двух частей сортируется рекурсивно с помощью сортировки слиянием.

Затем, отсортированные части массива сливаются в одну отсортированную последовательность.

def merge\_sort(lst):

print (“Разбивка”, lst)   
 if len(lst) > 1:  
 mid = len(lst) // 2  
 left = lst[:mid]  
 right = lst[mid:]  
  
 merge\_sort(left)  
 merge\_sort(right)  
  
 i = 0  
 j = 0  
 k = 0  
  
 while i < len(left) and j < len(right):  
 if left[i] < right[j]:  
 lst[k] = left[i]  
 i += 1  
 else:  
 lst[k] = right[j]  
 j += 1  
 k += 1  
  
 while i < len(left):  
 lst[k] = left[i]  
 i += 1  
 k += 1  
  
 while j < len(right):  
 lst[k] = right[j]  
 j += 1  
 k += 1

print (“Слияние”, lst)  
 return lst  
  
lst = [random.randint(0, 100) for \_ in range(10)]  
lst = merge\_sort(lst)  
print(lst)

(n log n)

Наиболее важная часть алгоритма — это процесс слияния. Для этого можно создать временный список или массив, в который будет добавляться наименьший элемент из каждой отсортированной части массива до тех пор, пока все элементы не будут добавлены

Эта реализация функции `merge\_sort()` принимает массив или список `arr` и рекурсивно разделяет его на меньшие части, пока размер массива не станет небольшим. Затем выполняется слияние отдельных частей с помощью функции `merge()`, которая объединяет их в отсортированный результат.

Визуализация

<https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

<https://algorithm-visualizer.org/divide-and-conquer/merge-sort>

<https://algorithm-visualizer.org/branch-and-bound/binary-search>

## Тим-сортировка

Тим-сортировка: особенности алгоритма и возможности применения в Python

Тим-сортировка является модификацией сортировки слиянием и представляет собой несколько модифицированный алгоритм, разработанный Тимом Петерсоном в 1972 году.

Принцип работы тим-сортировки основывается на использовании сортировки слиянием для объединения упорядоченных блоков данных различных размеров.

Какой алгоритм сортировки чаще всего используется в Python?

В Python чаще всего используется алгоритм сортировки под названием "TimSort". TimSort является гибридным алгоритмом, который комбинирует сортировку вставками и сортировку слиянием. Он был разработан для обработки разных типов данных, включая списки с дубликатами и предварительно отсортированные списки.

Каким образом алгоритм TimSort комбинирует сортировку вставками и сортировку слиянием?

Алгоритм TimSort сначала разбивает входные данные на маленькие подмассивы и сортирует их с помощью сортировки вставками. Затем TimSort объединяет эти отсортированные подмассивы с помощью сортировки слиянием. Этот процесс повторяется, пока все данные не будут отсортированы.

Наихудшим случаем для алгоритма TimSort является ситуация, когда данные располагаются в обратном порядке или содержат только небольшое количество упорядоченных блоков. В таких случаях TimSort работает медленнее других алгоритмов сортировки, таких как QuickSort.

Источник: https://fsnslnr.su/faq/kakoi-algoritm-sortirovki-ispolzuetsya-v-python

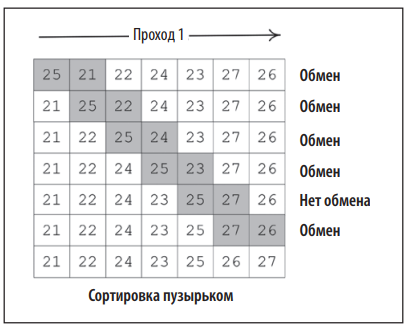
Основные особенности алгоритма:

Подходит для сортировки больших объемов данных, так как эффективность алгоритма не зависит от размера массива.

Использует принцип разбиения массива на блоки, которые затем объединяются при помощи сортировки слиянием.

Позволяет эффективно сортировать данные, расположенные на внешних носителях (например, на жестком диске) благодаря минимизации количества операций чтения и записи.

* Основные алгоритмы сортировки и поиска.



Посмотрим на видео и реализуем данный алгоритм

def createList(n=1):  
 M = [randint(1, 1000) for i in range(10 \*\* n)]  
 return M  
  
  
def BubbleSort():  
 lastElem = len(M) - 1  
 for j in range(lastElem-1):  
 for i in range(lastElem):  
 if M[i] > M[i + 1]:  
 M[i + 1], M[i] = M[i], M[i + 1]  
*# def simpleSort():  
# return M.sort()*M = createList()  
print(M)  
  
dt = timeit(BubbleSort)  
print (dt)  
  
print(M)

С помощью функции timeit замерим скорость сортировка (миллион раз)

[725, 550, 774, 857, 622, 698, 907, 85, 856, 483]

6.4817347999996855

[85, 483, 550, 622, 698, 725, 774, 856, 857, 907]

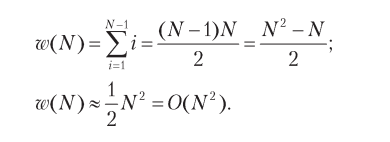
Сортировка вставками Основная идея сортировки вставками заключается в том, что на каждой итерации мы удаляем элемент из имеющейся у нас структуры данных, а затем вставляем его в нужную позицию. Именно поэтому алгоритм называется сортировкой вставками (insertion sort). На первой итерации мы сортируем два элемента данных. Затем мы расширяем выборку: берем третий элемент и находим для него позицию согласно его значению. Алгоритм выполняется до тех пор, пока все элементы не будут перемещены в правильное положение. Данный процесс показан на следующей диаграмме (рис. 3.5). Алгоритм сортировки вставками на Python выглядит так



Обратите внимание, что в основном цикле мы проходим по всему списку. В каждой итерации двумя соседними элементами являются list[j] (текущий элемент) и list[i] (следующий элемент). В выражениях list [j] > element\_next> и j >= 0 мы сравниваем текущий элемент со следующим.

Рассмотрим производительность алгоритма сортировки вставками.

Из описания алгоритма очевидно, что если структура данных уже отсортирована, он выполняется очень быстро. Фактически в этом случае сортировка имеет линейное время выполнения, то есть O(n). При наихудшем сценарии каждый внутренний цикл перемещает все элементы в списке. Если внутренний цикл мы обозначим i, наихудшая производительность алгоритма сортировки вставками определяется так:



Как правило, сортировка вставкой используется в работе с небольшими структурами данных. Для больших структур этот алгоритм не рекомендуется, поскольку обладает квадратичной средней производительно

def InsertionSort():  
 for i in range(1, len(M)):  
 j = i - 1  
 elem\_next = M[i]  
 while M[j] > elem\_next and j >= 0:  
 M[j + 1] = M[j]  
 j = j - 1  
 M[j + 1] = elem\_next  
 *# return M*M = createList()  
start = time.time()  
InsertionSort()  
end = time.time()  
print(end-start)

Сортировка слиянием Мы изучили два алгоритма сортировки: пузырьком и вставками. Производительность обоих будет лучше, если данные уже частично отсортированы. Третий алгоритм, с которым мы познакомимся, — алгоритм сортировки слиянием (merge sort), разработанный в 1940 году Джоном фон Нейманом. Отличительной чертой этого алгоритма является тот факт, что его производительность не зависит от упорядоченности входных данных. Подобно MapReduce и другим алгоритмам обработки больших данных, в его основе лежит стратегия «разделяй и властвуй». На этапе разделения алгоритм рекурсивно разбивает данные на две части до тех пор, пока размер данных не станет меньше определенного порогового значения. На этапе слияния алгоритм объединяет данные, пока мы не получим окончательный результат. Логика этого алгоритма объясняется на следующей диаграмме

Алгоритм сортировки — это алгоритм для упорядочивания элементов в какой-либо структуре

У каждого алгоритма есть свои преимущества и недостатки

Важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи

Асимптотическая сложность алгоритмов сортировки вставками — O(n^2)

Асимптотическая сложность алгоритма сортировки слиянием — O(n log n), но требует дополнительного расхода памяти