**Введение**

Олимпиадное программирование состоит из двух частей – проектирования алгоритмов и реализации алгоритмов.

Проектирование алгоритмов. По сути своей, олимпиадное программирование – это придумывание эффективных алгоритмов решения корректно поставленных вычислительных задач. Для проектирования алгоритмов необходимы навыки в решении задач и знание математики. Зачастую решение появляется в результате сочетания хорошо известных методов и новых идей.

Важную роль в олимпиадном программировании играет математика. На самом деле четких границ между проектированием алгоритмов и математикой не существует.

Реализация алгоритмов. В олимпиадном программировании решение задачи оценивается путем проверки реализованного алгоритма на ряде тестов. Поэтому придумать алгоритм недостаточно, его еще нужно корректно реализовать, для чего требуется умение программировать. Олимпиадное программирование сильно отличается от традиционной программной инженерии: программы короткие (несколько сотен строк – уже редкость), писать их нужно быстро, а сопровождение после соревнования не требуется.

В настоящее время на соревнованиях по программированию популярнее всего языки C++, Python и Java. Например, среди 3000 лучших участников Google Code Jam 2017 79% писали на C++, 16% – на Python и 8% – на Java.

На данном сайте Вы сможете ознакомиться с необходимой теорией для языка Python для успешного написания олимпиад в области информатики, а также проверить полученные знания на практике, решением предоставленных задач.

**Тема 1. Эффективность**

Во время своей работы программы используют различные структуры данных и алгоритмы, в связи с чем обладают разной эффективностью и скоростью решения задачи. Дать оценку оптимальности решения, реализованного в программе, поможет понятие вычислительной сложности алгоритмов.

1.1. Основные понятия

Вычислительная сложность (алгоритмическая сложность) - понятие, обозначающее функцию зависимости объема работы алгоритма от размера обрабатываемых данных.

Вычислительная сложность пытается ответить на центральный вопрос разработки алгоритмов: как изменится время исполнения и объем занятой памяти в зависимости от размера входных данных?. С помощью вычислительной сложности также появляется возможность классификации алгоритмов согласно их производительности.

В качестве показателей вычислительной сложности алгоритма выступают:

1. Временная сложность (время выполнения).

Временная сложность алгоритма - это функция от размера входных данных, равная количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

Временная сложность алгоритма зачастую может быть определена точно, однако в большинстве случаев искать точное ее значение бессмысленно, т.к. работа алгоритма зависит от ряда факторов, например, скорости процессора, набора его инструкций и т.д.

1. Асимптотическая сложность.

Асимптотическая сложность оценивает сложность работы алгоритма с использованием асимптотического анализа.

Алгоритм с меньшей асимптотической сложностью является более эффективным для всех входных данных.

1.2. Асимптотические нотации

Асимптотическая сложность алгоритма описывается соответствующей нотацией:

1. О-нотация, O («О»-большое): описывает верхнюю границу времени (время выполнения «не более, чем…»);
2. Омега-нотация, Ω («Омега»-большое): описывает нижнюю границу времени (время выполнения «не менее, чем…»).

Например, T(n)=O(N2) говорит о том, что алгоритм имеет квадратичное время выполнения относительно размера входных данных в качестве верхней оценки («О большое от эн квадрат»).

Каждая оценка при этом может быть:

1. наилучшая: минимальная временная оценка;
2. наихудшая: максимальная временная оценка;
3. средняя: средняя временная оценка.

При оценке, как правило, указывается наихудшая оценка.

Допустим, имеется задача поиска элемента в массиве. При полном переборе слева направо:

1. наилучшая оценка: O(1), если искомый элемент окажется в начале списка;
2. наихудшая оценка: O(N), если искомый элемент окажется в конце списка;
3. средняя оценка: O(N/2)=O(N)

1.3. Верхняя оценка и O-нотация

Наиболее часто используемой оценкой сложности алгоритма является верхняя (наихудшая) оценка, которая обычно выражается с использованием нотации O-большое.

Выделяют следующие основные категории алгоритмической сложности в O-нотации:

1. Постоянное время: O(1)

* Время выполнения не зависит от количества элементов во входном наборе данных.
* Пример: операции присваивания, сложения, взятия элемента списка по индексу и др.

1. Линейное время: O(N)

* Время выполнения пропорционально количеству элементов в коллекции.
* Пример: найти имя в телефонной книге простым перелистыванием, почистить ковер пылесосом и т.д.

1. Логарифмическое время: O(logN)

* Время выполнения пропорционально логарифму от количества элементов в коллекции.
* Пример: найти имя в телефонной книге (используя двоичный поиск).

1. Линейно-логарифмическое время: O(NlogN)

* Время выполнения больше чем, линейное, но меньше квадратичного.
* Пример: обработка N телефонных справочников двоичным поиском.

1. Квадратичное время: O(N2)

* Время выполнения пропорционально квадрату количества элементов в коллекции.
* Пример: вложенные циклы (сортировка, перебор делителей и т.д.)

1.4. Оценка сложности алгоритмов

Для оценки вычислительной сложности алгоритмов необходимо знать и учитывать сложности:

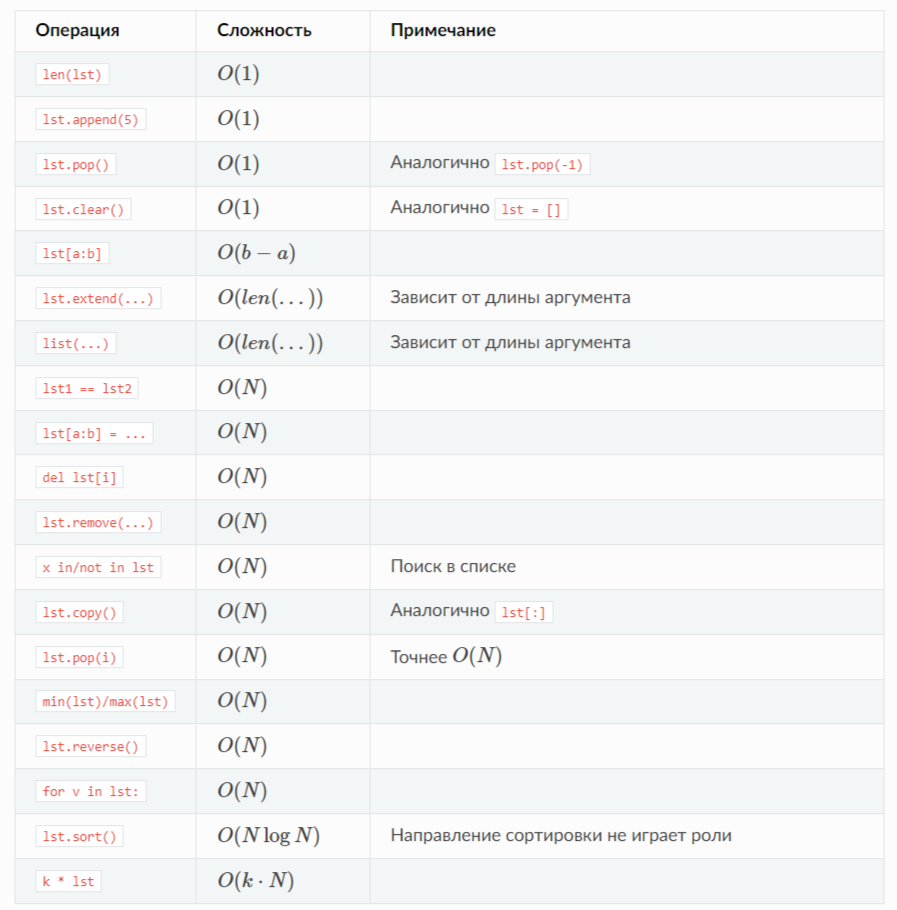
* используемых структур данных;
* совокупности различных операций.

1.5. Операции над структурами данных

В Python имеются коллекции (структуры данных), операции над которыми имеют определенную сложность.

1.5.1. Список и кортеж

Большинство операций со списком/кортежем имеют сложность O(N).



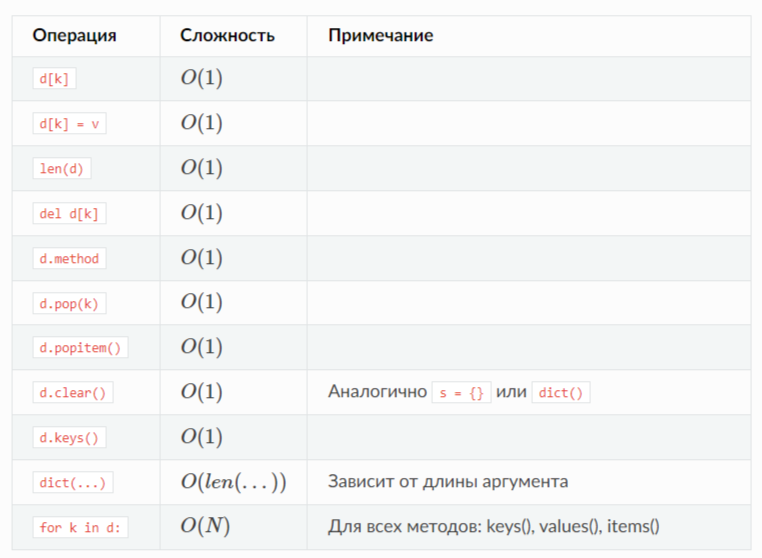
1.5.2. Множество

По сравнению со списком/кортежем множества большую часть операций выполняют со сложностью O(1).



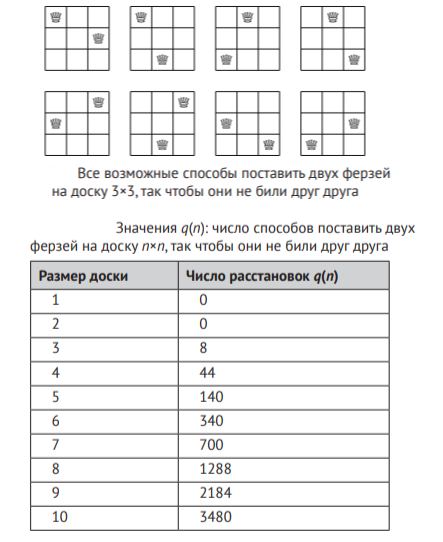
1.5.3. Словарь

Большинство операций словарей имеет сложность O(1).



1.6. Задача о двух ферзях

Разберём задачу: сколькими способами можно поставить на доску n×n двух ферзей, так чтобы они не били друг друга. На рис. показано, что на доске 3×3 это можно сделать 8 способами. Обозначим q(n) количество расстановок на доске n×n. Например, q(3) = 8, а в табл. приведены значения q(n) для 1 ≤ n ≤ 10.



Конечно, задачу можно решить по-простому: перебрать все возможные расстановки двух ферзей и подсчитать те, в которых ферзи не бьют друг друга. Временная сложность такого алгоритма O(n4 ), поскольку позицию первого ферзя можно выбрать n2 способами, а затем поставить второго ферзя n2 − 1 способами.

Поскольку число расстановок растет очень быстро, алгоритм их подсчета по одиночке будет работать слишком медленно для больших n. Следовательно, для создания эффективного алгоритма нужно придумать способ подсчета расстановок группами. Полезно отметить, что очень просто подсчитать, сколько клеток бьет один ферзь (видим это на следующем рисунке). Во-первых, он всегда бьет n − 1 клеток по горизонтали и n − 1 клеток по вертикали. Далее по обеим диагоналям ферзь бьет d − 1 клеток, где d – число клеток на диагонали. С учетом всего этого для вычисления числа клеток, в которые можно поставить второго ферзя, нужно время O(1), так что мы получаем алгоритм с временной сложностью O(n2).



К задаче можно подойти и по-другому, попытавшись найти рекуррентную формулу для подсчета расстановок, т. е. ответить на вопрос: как, зная q(n), вычислить q(n + 1)?

Чтобы получить рекурсивное решение, обратим внимание на последние горизонталь и вертикаль доски n×n (на представленном рисунке). Во-первых, число расстановок, в которых на последней горизонтали и на последней вертикали нет ферзей, равно q(n − 1). Во-вторых, общее число позиций ферзя на последней горизонтали и на последней вертикали равно 2n − 1. Находящийся в этих позициях ферзь бьет 3(n − 1) клеток, поэтому для второго ферзя остается n2 − 3(n − 1) – 1 позиций. Наконец, существует (n − 1)(n − 2) расстановок, в которых оба ферзя находятся на последней горизонтали или на последней вертикали. Поскольку эти расстановки посчитаны дважды, их количество надо вычесть. Объединив все вместе, получаем рекуррентную формулу:

q(n) = q(n − 1) + (2n − 1)(n2 − 3(n − 1) − 1) − (n − 1)(n − 2) = q(n − 1) + 2(n − 1)2 (n − 2)

и вместе с ней решение сложности O(n).



Однако же существует и замкнутая формула для этой функции:

q(n) = n4/2 - 5n3/3 + 3n2/2 - n/3,

которую можно доказать по индукции, пользуясь рекуррентной формулой. А раз так, то задачу можно решить за время O(1).

**Тема 2. Битовые операции: AND, OR, NOT, XOR - таблицы значений.**

Как известно, минимальной единицей измерения информации является **бит**, который хранит одно из 2-х значений: 0 (**False**, ложь) либо 1 (**True**, истина). Таким образом, битовая ячейка может одновременно находиться лишь в одном из двух возможных состояний.

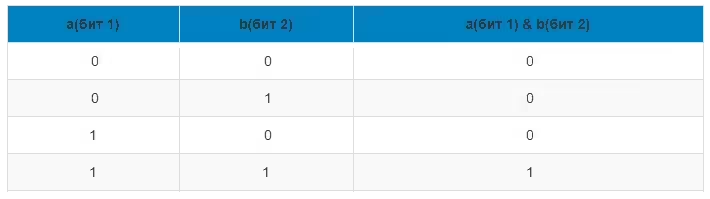
Для манипуляций с битами используют определённые операции — **логические или булевые**. Они могут применяться к любому биту, вне зависимости от того, какое у него значение — ноль или единица. Что же, давайте посмотрим на примеры использования трёх основных логических операций.

**Логическая операция AND (и)**

**AND** обозначается знаком &.

Оператор AND выполняется с 2-мя битами, возьмём, к примеру, a и b. Результат выполнения операции AND равен 1, если a и b равняются 1. В остальных случаях результат равен 0. Например, с помощью AND вы можете узнать, чётное число или нет.

Посмотрите на таблицу истинности операции AND:



**Логическая операция OR (ИЛИ)**

Обозначается знаком |.

Оператор **OR** также выполняется с 2-мя битами (a и b). Результат равен 0, если a и b равны 0, иначе он равен 1. Смотрим таблицу истинности.

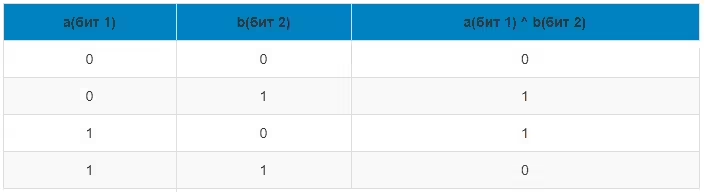


**Логическая операция XOR (исключающее ИЛИ)**

Оператор XOR обозначается ^.

**XOR** выполняется с 2-мя битами (a и b). Результат выполнения операции XOR (**исключающее ИЛИ**) равен 1, когда один из битов b или a равен 1. В остальных ситуациях результат применения оператора XOR равен 0.

Таблица истинности логической операции для XOR (исключающее ИЛИ) выглядит так:



Используя XOR (исключающее ИЛИ), вы можете поменять значения 2-х переменных одинакового типа данных, не используя временную переменную. А ещё, посредством XOR можно зашифровать текст, например:



Согласен, XOR — далеко не самый надёжный метод шифрования, но это не значит, что его нельзя сделать частью какого-либо шифровального алгоритма.

**Логическая операция NOT (НЕ)**

Это побитовое отрицание, поэтому выполняется с одним битом и обозначается ~.

Результат зависит от состояния бита. Если он в нулевом состоянии, то итог операции — единица и наоборот. Всё предельно просто.



Свойства операции XOR:

1. XOR и 0: x^0 = x
2. XOR с одинаковыми аргументами: x^x = 0
3. Коммутативность: x^y = y^x

Эти 4 логические операции следует запомнить в первую очередь, т. к. с их помощью можно получить практически любой возможный результат. Также существуют такие операции, как <<(побитовый сдвиг влево) и >>(побитовый сдвиг вправо).

**Тема 3. Проверка на простоту, разложение на множители. Наибольший общий делитель и наименьшее общее кратное: алгоритм Евклида. Признак Паскаля. Расширенный алгоритм Евклида.**

Простое число - это натуральное, целое положительное число n, которое делится только на единицу и на себя.

Пример рабочего алгоритма, определяющего число на простоту:

def IsPrime(a):

if a <= 1: return False

for d in range(2, a):

if a % d == 0:

return False

return True

Алгоритм разложения числа на простые множители:

def Prime\_fac(n):

i = 2

prime\_fac = []

while i \*\* 2 <= n:

while n % i == 0:

prime\_fac.append(i)

n = n / i

i = i + 1

if n > 1:

prime\_fac.append(n)

return prime\_fac

Наибольший общий делитель (НОД):

A, B **=** [int(x) **for** x **in** input().split()]

a, b **=** A, B

**while** b !**=** 0:

a, b **=** b, a**%**b

print (A**\***B**//**(A**\***B**//**a))

Наименьшее общее кратное (НОК):

A,B **=** [int(x) **for** x **in** input().split()]

a, b **=** A, B

**while** b!**=** 0:

a, b **=** b, a**%**b

print (A**\***B**//**a)

Алгоритм Евклида — это алгоритм, который используется для нахождения наибольшего делителя двух целых чисел.

def gcd\_rem\_division(num1, num2):

while num1 != 0 and num2 != 0:

if num1 >= num2:

num1 %= num2

else:

num2 %= num1

return num1 or num2

Треугольник Паскаля:

def PrintPasTriangle(rows):

row = [1]

for i in range(rows):

print(row)

row = [sum(x) for x in zip([0] + row, row + [0])]

Расширенный алгоритм Евклида:

def gcd\_extended(num1, num2):

if num1 == 0:

return (num2, 0, 1)

else:

div, x, y = gcd\_extended(num2 % num1, num1)

return (div, y - (num2 // num1) \* x, x)

Проверить данный алгоритм можно следующим образом:

a = gcd\_extended(426, 334)

print(f”Делитель равен {a[0]}, x = {a[1]}, y = {a[2]}”)

Делитель равен 2, x = 69, y = -88

Rак работает алгоритм?

Сначала проверяется, равно ли первое число нулю, если это так, то второе число является делителем, а коэффициенты равны 0 и 1, так как «num1 \* x + num2 \* y = y» в том случае, если y = 1, а левое произведение равно нулю.

Функция возвращает три числа: делитель, коэффициент x и коэффициент y. Для её реализации используется рекурсия, делитель получается тем же образом, что и в классическом рекурсивным алгоритме, а коэффициенты рекурсивно вычисляются по формулам:

x = y — (num2 // num1) \* x

y = x

**Тема 4. Линейный поиск**

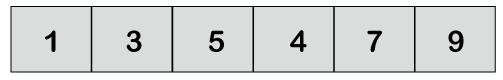
Линейный поиск – это метод поиска элементов в списке. Его еще называют последовательным поиском. Это простейший алгоритм поиска, поскольку он ищет желаемый элемент последовательно.

Он сравнивает каждый элемент со значением, которое мы ищем. Если оба совпадают, элемент найден, и алгоритм возвращает позицию индекса ключа.

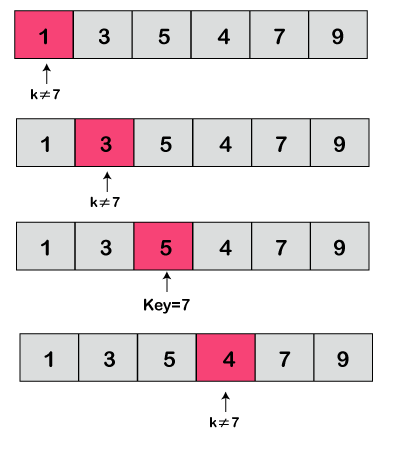
Концепция линейного поиска

Давайте разберёмся в следующих шагах, чтобы найти элемент key = 7 в данном списке.

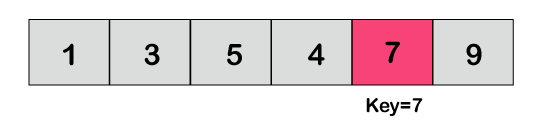
Шаг – 1: Начните поиск с первого элемента и проверьте ключ = 7 с каждым элементом списка x.



Шаг – 2: Если элемент найден, вернуть индексную позицию ключа.



Шаг – 3: Если элемент не найден, возвращаемого элемента нет.



Алгоритм решения данной задачи:

def linear\_Search(list1, n, key):

for i in range(0, n):

if (list1[i] == key):

return i

return -1

list1 = [1 ,3, 5, 4, 7, 9]

key = 7

n = len(list1)

res = linear\_Search(list1, n, key)

If res == -1:

print("Элемент не найден")

else:

print("Элемент найден по индексу ", res)

**Тема 5. Квадратичные сортировки (выбор, вставки, пузырек). Сортировка подсчётом. Быстрая сортировка Хоара.**

**Сортировка выбором**

Одна из наиболее часто возникающих в программировании задач — задача о сортировке элементов массива (списка). Постановка задачи — дан список элементов A, которые можно сравнивать (например, чисел, строк, кортежей и т. д.). Необходимо переставить элементы списка местами так, чтобы было выполнено условие A[i]<=A[i+1] для всех пар соседних элементов. Например, если был дан список [4, 1, 2, 4, 2, 3], то отсортированный список будет иметь вид [1, 2, 2, 3, 4, 4]. Такой порядок сортировки называется сортировкой по неубыванию элементов (но чаще используют не вполне точный термин «сортировка в порядке возрастания»). Если заменить условие на A[i] >= A[i+1], то получится сортировка в порядке невозрастания (убывания). Для сортировки списков придумано много различных алгоритмов.

Один из наиболее простых алгоритмов — сортировка выбором. Идея алгоритма следующая. Сначала выберем в списке наименьший элемент и поставим его на место с индексом 0 в списке (в начало списка). Потом среди всех оставшихся элементов выберем наименьший и поставим его на место с индексом 1. Затем выберем наименьший среди элементов элементов, начиная с третьего, и поставим его на место c индексом 2 и т. д.

Таким образом, в этой сортировке два вложенных цикла. Внешний цикл осуществляется по переменной i начиная с 0. При этом все элементы списка до элемента с индексом i (то есть A[:i]) есть наименьшие элементы списка, упорядоченные по неубыванию.

Теперь выберем среди элементов списка A[i:] элемент с наименьшим значением и поменяем его местами с элементом с индексом i.

def SelectionSort(A):

for i in range(0, len(A) - 1):

# Среди элементов A[i:] выбираем наименьший

# Сохраняем его индекс в переменной min\_idx

min\_idx = i

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

if A[j] < A[min\_idx]:

min\_idx = j

# Теперь поставим A[min\_idx] на место A[i]

A[i], A[min\_idx] = A[min\_idx], A[i]

Можно модифицировать алгоритм — не сохранять индекс наименьшего из просмотренных элементов, а при просмотре элементов в срезе A[i:] обменивать очередной элемент A[j] местами с A[i], если A[j]<A[i]:

def SelectionSort(A):

for i in range(0, len(A) - 1):

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

if A[j] < A[i]:

A[i], A[j] = A[j], A[i]

Посчитаем сложность этого алгоритма. Пусть список содержит n элементов. Сначала нужно найти минимум среди n элементов списка, что потребует n операций. Потом нужно найти наименьший из n-1 элемента, на это нужно n-1 операция. Потом нужно n-2 операции и т. д. Общее число операций равно

n + (n - 1) + (n - 2) + ... + 1 = n(n + 1)/2 = O(n2)

Таким образом, сортировка выбором — квадратичный алгоритм, время его работы пропорционально квадрату от размера списка.

**Сортировка вставками**

Сортировка вставками использует похожий инвариант: первый элементы списка, то есть срез A[:i] уже отсортирован. По-иному устроен алгоритм добавления i-го элемента к уже отсортированной части. Здесь берется элемент A[i] и добавляется к уже отсортированной части списка. Например, пусть i = 5 и срез A[:i] = [1, 4, 6, 8, 8], а значение A[i] == 5. Тогда элемент A[i] == 5 нужно поставить после элемента A[ 1] == 4, а все элементы, которые больше 5 сдвинуть вправо на 1. Получится cрез A[:i + 1] = [1, 4, 5, 6, 8, 8]. Таким образом, при вставке элемента A[i] в срез A[:i] так, чтобы в результате получился упорядоченный срез, все элементы, которые больше A[i] будут двигаться вправо на одну позицию. А в освободившуюся позицию и будет вставлен элемент A[i].

При этом значение A[i] нужно сохранить в переменной, т. к. на место элемента A[i], возможно, будет записан элемент A[i – 1].

Получаем следующий алгоритм:

def InsertionSort(A):

for i in range(1, len(A)):

# В new\_elem сохранили значение A[i]

new\_elem = A[i]

# Начиная с элемента A[i - 1]

j = i - 1

# все элементы, которые больше new\_elem

while j >= 0 and A[j] > new\_elem:

# сдвигаем вправо на 1

A[j + 1] = A[j]

j -= 1

# На свободное место записываем new\_elem

A[j + 1] = new\_elem

Посчитаем сложность алгоритма сортировки вставками. Следует отметить, что если массив уже упорядочен, то все элементы останутся на своем месте и вложенный цикл не будет выполнен ни разу. В этом случае сложность алгоритма сортировки вставками — линейная, т. е. . Аналогично, если массив «почти упорядочен», то есть для превращения его в упорядоченный нужно поменять местами несколько соседних или близких элементов, то сложность также будет линейной.

Но если массив упорядочен в обратном порядке, например, каждый элемент больше следующего, а необходимо добиться обратного порядка, то каждый элемент будет перемещаться максимально влево, т. е. до самой крайней позиции. В этом случае количество выполняемых перемещений будет равно

1 + 2 + ... + (n - 1) + n = n(n + 1)/2 = O(n2)

Итак, мы видим, что сложность алгоритма сортировки вставками сильно зависит от того, «насколько хорошо» отсортирован исходный список. В лучшем случае время работы — линейно, в худшем случае — квадратично. Что же происходит «в среднем», когда массив заполнен числами в случайном порядке?

В этом случае математическое ожидание количества перемещений элементов будет равно половине от числа перемещений в худшем случае (каждый элемент в среднем будет перемещаться не до самого начала списка, а только до середины этого пути), то есть математическое ожидание числа перемещений будет равно n(n + 1)/4 = O(n2). То есть в среднем этот алгоритм также имеет квадратичную сложность.

**Сортировка пузырьком**

Алгоритм сортировки пузырьком построен на простой идее. Пусть есть два соседних элемента, которые стоят в неправильном порядке, то есть A[i] > A[i + 1]. Поменяем их местами. Оказывается, такой операции уже достаточно, чтобы отсортировать массив — достаточно повторять такую операцию до тех пор, пока есть соседние неправильно упорядоченные элементы. Но необходимо еще организовать процесс так, чтобы он завершился.

Пройдем по всему списку слева направо. Если есть два неправильно упорядоченных элемента — переставим их. В результате самый большой элемент списка «всплывет» в его конец — станет последним элементом. Повторим этот проход еще раз — второй по величине элемент списка «всплывет» в конец, остановившись перед наибольшим элементов. За следующий проход мы можем установить на место третий элемент и т. д. При этом последние, уже «всплывшие» наибольшие элементы не нужно затрагивать алгоритмом сортировки. Получим следующий алгоритм:

def BubbleSort(A):

for j in range(len(A) - 1, 0, -1):

for i in range(0, j):

if A[i] > A[i + 1]:

A[i], A[i + 1] = A[i + 1], A[i]

Легко видеть, что сложность этого алгоритма также будет O(n2).

Что произойдет, если запустить сортировки пузырьком на уже остортированном списке? Ни одной перестановки не будет произведено, но алгоритм все равно выполнит O(n2) операций. Хотя уже после первого прохода вложенного цикла можно понять, что список уже упорядочен, если ни одной перестановки не было сделано. Это позволяет соптимизировать алгоритм сортировки — закончим его, если во внутреннем цикле не было выполнено ни одной перестановки. Для этого заведем переменную IsNotOrdered, которая будет равна True, если список не упорядочен. Перед проходом внутреннего цикла мы будем устанавливать IsNotOrdered=False (априори считаем, что список уже упорядочен), но если обнаруживается пара неупорядоченных элементов, то выполняется присваивание IsNotOrdered=True. Внешний цикл выполняется, пока переменная IsNotOrdered принимает значение True.

def BubbleSort(A):

j = len(A) - 1

IsNotOrdered = True

while IsNotOrdered:

IsNotOrdered = False

for i in range(0, j):

if A[i] > A[i + 1]:

A[i], A[i + 1] = A[i + 1], A[i]

IsNotOrdered = True

j -= 1

Такой алгоритм также будет работать за линейное время на почти упорядоченных массивах (довольно скоро массив упорядочится и внешний цикл закончится).

Есть различные алгоритмы, построенные на основе пузырьковой сортировке. Например, в шейкерной сортировке внутренний цикл проходится поочередно слева направо, затем справа налево. А в сортировке Шелла элементы переставляются не соседние элементы, а элементы, отстоящие на большее расстояние, что позволяет быстрее перемещать элементы по списку, выполняя сразу несколько шагов.\

**Сортировка подсчётом**

Алгоритм

Подсчитываем, сколько раз в массиве встречается каждое значение, и заполняем массив подсчитанными элементами в соответствующих количествах. Счётчики для всего диапазона чисел создаются заранее (изначально равны нулю).

Сложность сортировки по времени

* Худшая O(n + k)
* Средняя O(n + k)
* Лучшая O(n)

Шаги к правильному решению

Создадим функцию counting\_sort, которая принимает на вход список и переменную largest.

Внутри функции создадим список c нулями и длиной largest+1.

Для каждого элемента входного списка увеличиваем значение элемента таким образом c[alist[i]] = c[alist[i]] + 1.

Теперь список с содержит частоту каждого элемента входного списка.

Каждому элементу от 1 до length(c) -1 добавляем к значению текущего элемента значение предыдущего c[i] = c[i] + c[i - 1].

Создадим result список с таким же размером, как и входной список.

Создадим цикл, который итерируется по списку в обратном порядке.

В каждой итерации цикла установим result[c[x]] = x, а затем уменьшим c[x] на 1.

def counting\_sort(alist, largest):

c = [0]\*(largest + 1)

for i in range(len(alist)):

c[alist[i]] = c[alist[i]] + 1

c[0] = c[0] - 1

for i in range(1, largest + 1):

c[i] = c[i] + c[i - 1]

result = [None]\*len(alist)

for x in reversed(alist):

result[c[x]] = x

c[x] = c[x] - 1

return result

alist = input('Введите список неотрицательных чисел: ').split()

alist = [int(x) for x in alist]

k = max(alist)

sorted\_list = counting\_sort(alist, k)

print('Отсортированный список: ', end='')

print(sorted\_list)

**Быстрая сортировка Хоара.**

Этот алгоритм, чаще называемый просто «быстрая сортировка» (англ. Quicksort) придуман английским ученым Чарльзом Хоаром в 1960 году.

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

Простая реализация быстрой сортировки Хоара выглядит так:

def QuickSort(A):

if len(A) <= 1:

return A

else:

q = random.choice(A)

L = []

M = []

R = []

for elem in A:

if elem < q:

L.append(elem)

elif elem > q:

R.append(elem)

else:

M.append(elem)

return QuickSort(L) + M + QuickSort(R)

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

Тот же алгоритм можно записать еще проще в «функциональном» стиле:

def QuickSort(A):

if len(A) <= 1:

return A

else:

q = random.choice(A)

L = [elem for elem in A if elem < q]

M = [q] \* A.count(q)

R = [elem for elem in A if elem > q]

return QuickSort(L) + M + QuickSort(R)

Однако, такая реализация алгоритма, как и сортировка слиянием, требует O(n) дополнительной памяти. Возможна реализация алгоритма сортировки слиянием без использования дополнительной памяти:

def QuickSort(A, l, r):

if l >= r:

return

else:

q = random.choice(A[l:r + 1])

i = l

j = r

while i <= j:

while A[i] < q:

i += 1

while A[j] > q:

j -= 1

if i <= j:

A[i], A[j] = A[j], A[i]

i += 1

j -= 1

QuickSort(A, l, j)

QuickSort(A, i, r)

Эта реализация не возвращает никакого значения, а сортирует элементы списка A «на месте», то есть модифицирует переданный список A. Два дополнительный параметра l и r указывают на номер первого и последнего элемента того фрагмента списка, который нужно отсортировать (включая эти элементы), то есть элемент сортирует срез A[l:r+1]. Для сортировки всего списка A необходимо вызвать QuickSort(A, 0, len(A) – 1).

Если l >= r, то ничего сортировать не нужно, срез пустой или содержит один элемент. Иначе случайным образом выбирается барьерный элемент. Далее заводятся два указателя i = l и j = r. Затем элементы списка переставляются так, чтобы элементы, которые меньше или равны q оказывались слева от указателя i, а те, которые больше или равны q оказывались справа от j. После окончания распределения рекурсивно сортируются две получившиеся части списка.

Данная реализация не требует дополнительной памяти (за исключением памяти, необходимой для организации рекурсии).

Асимптотика алгоритма

Сложность алгоритма быстрой сортировки Хоара зависит от метода выбора барьерного элемента. В лучшем случае при каждом выборе барьерного элемента должен выбираться медианный элемент массива. Но поиск медианного элемента — сложная задача, её нельзя решить быстро. Если выбрать первый элемент фрагмента списка A[l] или последний A[r], то если список A уже упорядочен, сложность алгоритма будет O(n2), так как на каждом рекурсивном вызове от большей части списка будет отделяться всего один элемент.

Поэтому в алгоритме быстрой сортировки Хоара, как правило, в качестве барьерного элемента выбирается случайный элемент списка. Тогда алгоритм становится вероятностным — время его работы зависит от того, каким будет случайно выбранный элемент. Возможна (но крайне маловероятна) ситуация, когда всегда будет выбираться наименьший элемент, и в этом случае алгоритм будет работать за O(n2).

В теории вероятностей доказывается, чти при случайном выборе элемента списка и разбиении его на две части, размер большей из двух получившихся частей будет в среднем равен 3**π** / 4. В этом случае глубина рекурсии в среднем будет составлять порядка log n, а средняя сложность алгоритма быстрой сортировки Хоара — O(n log n).

**Тема 6. Рекурсия: НОД, Ханойские башни, перебор с возвратом.**

**Наибольший общий делитель** можно определить также рекурсивным способом, давайте разберём решение данной задачи.

Описание задачи

Программа принимает на вход два числа и находит наибольший общий делитель (НОД) с использованием рекурсии.

Решение задачи

1. Принимаются два числа, которые сохраняются в отдельные переменные.
2. Передаем оба числа в рекурсивную функцию в качестве аргумента.
3. В качестве базового условия рекурсии принимаем равенство нулю второго числа (второго аргумента функции). В этом случае результатом работы функции является первое число (первый аргумент функции).
4. В противном случае снова рекурсивно вызываем эту функцию и в качестве первого аргумента передаем ей второй аргумент из предыдущего вызова функции, а в качестве второго — остаток от деления первого аргумента на второй аргумент.
5. Когда функция завершит свою работу, ее результатам будет первый аргумент из последнего вызова этой функции. Он и будет наибольшим общим делителем (НОД).
6. Выводим результат на экран.
7. Конец.

Исходный код

Ниже дан исходный код, который осуществляет нахождение наибольшего общего делителя (НОД) с использованием рекурсии. Результаты работы программы также даны ниже.

def gcd(a, b):

if (b == 0):

return a

else:

return gcd(b, a % b)

a = int(input("Введите первое число:"))

b = int(input("Введите второе число:"))

GCD = gcd(a, b)

print("НОД: ")

print(GCD)

Объяснение работы программы

1. Пользователь вводит два числа и они записываются в переменные a и b.
2. Затем эти два числа передаются в качестве аргументов в рекурсивную функцию gcd().
3. В качестве базового условия рекурсии принимаем равенство 0 второго аргумента функции: b == 0. В этом случае результатом работы функции будет первый аргумент a.
4. В противном случае снова рекурсивно вызываем нашу функцию следующим образом: gcd(b, a % b). То есть, в качестве первого аргумента передаем ей второй аргумент из предыдущего вызова функции (b), а в качестве второго —остаток от деления a на b.
5. Когда функция завершит свою работу, ее результатам будет первый аргумент из последнего вызова этой функции. Он и будет наибольшим общим делителем (НОД).
6. Выводим результат работы функции на экран.

Результаты работы программы

Пример 1:

Введите первое число:5

Введите второе число:15

НОД:

5

Пример 2:

Введите первое число:30

Введите второе число:12

НОД:

6

**Ханойские башни**

Одна из популярных головоломок XIX века.

Даны три стержня, на один из которых нанизаны восемь колец, причём кольца отличаются размером и лежат меньшее на большем. Задача состоит в том, чтобы перенести пирамиду из восьми колец за наименьшее число ходов на другой стержень. За один раз разрешается переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее.

Рекурсивный метод.

Предположим, что нам известен алгоритм перемещения n-1 дисков с одного стержня на второй, тогда функция play() для перемещения n дисков со стержня surse на стержень receiver должна будет:

* переместить n-1 диск со стержня surse на стержень storage
* последний диск n переместить со стержня surse на стержень receiver
* переместить n-1 диск со стержня storage на стержень receiver.

Программа, решающая головоломку Ханойская башня.

n = 5 # Количество дисков

def play(n, sourse, receiver, storage):

"""

Функция перемещения n дисков в головоломке Ханойская башня.

Аргументы функции:

Первый аргумент n - integer (целое число), количество дисков в пирамиде.

sourse, receiver, storage - любого типа.

Второй аргумент - sourse, стержень с которого перекладываем диски.

Третий аргумент - receiver, стержень на который перекладываем диски.

Четвёртый аргумент - storage, стержень на который перекладываем n-1 дисков

для временного хранения в процессе общей работы.

"""

if n <= 0: return

play(n-1, sourse, storage, receiver)

print("Диск ", n, " : ", sourse, " --> ", receiver)

play(n-1, storage, receiver, sourse)

play (n, 'a', 'b', 'c')

Результат работы программы решающей головоломку Ханойская башня с пятью дисками.

Диск 1 : a --> b

Диск 2 : a --> c

Диск 1 : b --> c

Диск 3 : a --> b

Диск 1 : c --> a

Диск 2 : c --> b

Диск 1 : a --> b

Диск 4 : a --> c

Диск 1 : b --> c

Диск 2 : b --> a

Диск 1 : c --> a

Диск 3 : b --> c

Диск 1 : a --> b

Диск 2 : a --> c

Диск 1 : b --> c

Диск 5 : a --> b

Диск 1 : c --> a

Диск 2 : c --> b

Диск 1 : a --> b

Диск 3 : c --> a

Диск 1 : b --> c

Диск 2 : b --> a

Диск 1 : c --> a

Диск 4 : c --> b

Диск 1 : a --> b

Диск 2 : a --> c

Диск 1 : b --> c

Диск 3 : a --> b

Диск 1 : c --> a

Диск 2 : c --> b

Диск 1 : a --> b

Не трудно заметить, что минимальное число ходов, необходимое для решения головоломки, равно (2n − 1), где n — число дисков.

**Перебор с возвратом.**

Есть такие задачи, для решения которых приходится организовывать полный перебор возможных вариантов. Перебор с возвратом (backtracking) – это общий метод упорядоченного перебора. Перебор с возвратом особенно удобен для решения задач, требующих проверки потенциально большого, но конечного числа решений. В настоящем разделе описывается схема рекурсивного перебора с возвратом. В самом общем случае мы полагаем, что решение можно записать как вектор (массив переменной длины) B = (b1,b2,…,bn), удовлетворяющий заданным условиям и ограничениям, или как множество таких векторов. При этом в одних задачах размерность решения (число n) может быть известна заранее, а в других заранее не определена. Метод перебора с возвратом основан на том, что при поиске решения многократно делается попытка расширить текущее частичное решение, то есть его продолжить. Если расширение невозможно, то происходит возврат к предыдущему более короткому частичному решению, и делается попытка его расширить другим возможным способом. В качестве исходного частичного решения мы выбираем пустой вектор, который будем обозначать ( ). На основе заданных ограничений выясняем, какие элементы являются кандидатами в b1; обозначим это подмножество через S1. В качестве b1 выбираем первый элемент из S1 и получаем частичное решение (b1). В общем случае, по частичному решению (b1,b2,…,bk-1) на основе ограничений задачи строится Sk, из которого выбираются кандидаты для расширения частичного решения (b1,b2,…,bk-1) до (b1,b2,…,bk-1,bk). Если Sk={}, то есть частичное решение (b1,b2,…,bk-1) не может быть расширено, мы возвращаемся и выбираем новый элемент bk-1. Если новый элемент bk-1 выбрать нельзя, мы возвращаемся еще дальше и выбираем новый элемент bk-2 и т. д

Одним из примеров перебора с возвратом может быть такая задача:

В алфавите языке племени «тумба-юмба» четыре буквы: «K», «L», «M» и «N». Нужно вывести на экран все возможные слова, состоящие из n букв (n>1), в которых вторая буква «K». Подсчитайте количество таких слов.

Решением будет являться алгоритм:

n = int(input('N > '))

alphabet = ['K', 'L', 'M', 'N']

res = ['KK', 'LK', 'MK', 'NK']

def extend(d: dict, alphabet: dict, l: int) -> dict:

if len(d[0]) >= 1:

return d

return extend([item + a for item in d for a in alphabet if type(a) == type('a')], alphabet, 1)

res = extend(res, alphabet, n)

print(res)

print('Всего слов', len(res))

**Тема 7. Стек. Стек функций.**

Стек в Python – это линейная структура данных, в которой данные расположены объектами друг над другом. Он хранит данные в режиме LIFO (Last in First Out). Данные хранятся в том же порядке, в каком на кухне тарелки располагаются одна над другой. Мы всегда выбираем последнюю тарелку из стопки тарелок. В стеке новый элемент вставляется с одного конца, и элемент может быть удален только с этого конца.

Простым примером стека является функция «Отменить» в редакторе. Функция отмены работает с последним выполненным нами событием. Мы можем выполнять две операции в стеке – PUSH и POP. Операция PUSH – это когда мы добавляем элемент, а операция POP – когда мы удаляем элемент.

Методы стека

Python предоставляет следующие методы, которые обычно используются со стеком:

* empty() – возвращает истину, если стек пуст. Временная сложность O (1).
* size() – возвращает длину. Временная сложность O (1).
* top() – этот метод возвращает адрес последнего элемента. Временная сложность O (1).
* push(g) – метод добавляет элемент ‘g’ в конец стека – временная сложность составляет O (1).
* pop() – удаляет самый верхний элемент. Временная сложность O (1).

Реализация

Python предлагает различные способы реализации стека. В этом разделе мы обсудим реализацию с использованием Python и его модуля. Мы можем реализовать стек на Python следующими способами:

* список;
* dequeu;
* LifeQueue.

Реализация с использованием списка

Список Python можно использовать как стек. Он использует метод append() для вставки элементов, где стек использует метод push(). В списке также есть метод pop() для удаления последнего элемента.

Но в списке есть недостаток: он становится медленнее по мере роста. Новый элемент в нем хранится рядом с другим. Если список растет и выходит за пределы блока памяти, Python выделяет некоторую память. Вот почему список становится медленным. Разберем на примере –

my\_stack = []

my\_stack.append('x')

my\_stack.append('y')

my\_stack.append('z')

print(my\_stack)

print('\nЭлементы, вышедшие из my\_stack:')

print(my\_stack.pop())

print(my\_stack.pop())

print(my\_stack.pop())

print('\nmy\_stack после удаления элементов:')

print(my\_stack)

Выход:

Элементы, вышедшие из my\_stack:

['x', 'y', 'z']

:

z

y

x

my\_stack после удаления элементов:

[]

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/Stack.py", line 21, in <module>

print(my\_stack.pop())

IndexError: pop from empty list

</module>

Объяснение:

В приведенном выше коде мы определили пустой список. Мы вставляли элементы один за другим, используя метод append(), похожий на метод push(). Мы также удалили элементы с помощью метода pop(), который возвращает последний элемент списка.

С использованием collection.deque

Модуль коллекции предоставляет класс deque, который используется для создания стеков Python. Deque произносится как «колода», что означает «двусторонняя очередь». Двухсторонняя очередь может быть предпочтительнее списка, поскольку она выполняет операции добавления и извлечения быстрее, чем список. Временная сложность – O (1), где список занимает O (n).

from collections import deque

my\_stack = deque()

my\_stack.append('a')

my\_stack.append('b')

my\_stack.append('c')

print('Инициализация my\_stack:')

print(my\_stack)

print('\nЭлементы, удалённые из my\_stack:')

print(my\_stack.pop())

print(my\_stack.pop())

print(my\_stack.pop())

print('\nmy\_stack после удаления элементов:')

print(my\_stack)

Выход:

Инициализация my\_stack:

deque(['a', 'b', 'c'])

Элементы, удалённые из my\_stack::

c

b

a

my\_stack после удаления элементов:

deque([])

Объяснение:

Приведенный выше код почти аналогичен предыдущему примеру. Однако единственная разница в том, что мы импортировали двухстороннюю очередь из модуля коллекции.

Сравнение Deque и список

Список хранит элементы рядом друг с другом и использует непрерывную память. Это наиболее эффективно для нескольких операций, таких как индексация в списке. Например, получение list1 [2] выполняется быстро, поскольку Python знает точное положение определенного элемента. Непрерывная память также позволяет хорошо работать со списками.

Список требует больше времени, чтобы добавить некоторые объекты, чем другие. Если блок непрерывной памяти заполнен, он получит другой блок, который может занять гораздо больше времени, чем обычная функция append().

Модуль LifeQueue

Модуль очереди имеет очередь LIFO, которая совпадает со стеком. Обычно очередь использует метод put() для добавления данных и метод () для получения данных.

Ниже приведены несколько методов, доступных в очереди.

empty() – если очередь пуста, возвращает true; в противном случае вернется false.

maxsize() – этот метод используется для установки максимального количества элементов, разрешенных в очереди.

get() – возвращает и удаляет элемент из очереди. Очередь может быть пустой; он ждет, пока элемент не станет доступным.

full() – возвращает True, если очередь заполнена. По умолчанию очередь определяется как maxsize = 0. В этом случае он не вернет True.

put(item) – добавляет элемент в очередь; если очередь заполнена, ожидает, пока освободится место.

put\_nowait(item) – добавляет элемент в очередь без задержки.

qsize() – возвращает размер очереди.

Пример –

# Implementing stack using the queue module

from queue import LifoQueue

# Initializing a my\_stack stack with maxsize

my\_stack = LifoQueue(maxsize = 5)

# qsize() display the number of elements

# in the my\_stack

print(my\_stack.qsize())

# put() function is used to push

# element in the my\_stack

my\_stack.put('x')

my\_stack.put('y')

my\_stack.put('z')

print("Stack is Full: ", my\_stack.full())

print("Size of Stack: ", my\_stack.qsize())

# To pop the element we used get() function

# from my\_stack in

# LIFO order

print('\nElements poped from the my\_stack')

print(my\_stack.get())

print(my\_stack.get())

print(my\_stack.get())

print("\nStack is Empty: ", my\_stack.empty())

Выход:

0

0

Stack is Full: False

Size of Stack: 3

Elements poped from the my\_stack

z

y

x

Stack is Empty: True

Объяснение:

Выше мы импортировали модуль очереди, который является LIFOqueue. Он работает так же, как стек, но этот модуль включает некоторые дополнительные функции, упомянутые выше. Мы определили стек с максимальным размером, что означает, что он может содержать максимум пять значений в нем.

Начальный размер массива равен нулю; мы поместили три элемента в стек с помощью метода put(). Теперь мы снова проверили, пуст ли стек и его размер. Мы имеем три элемента. Выдвинем элемент с помощью метода get(). Сначала удаляется последний добавленный элемент. После удаления всех элементов получаем пустой стек.

Источник: https://pythonpip.ru/examples/stek-python