**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:

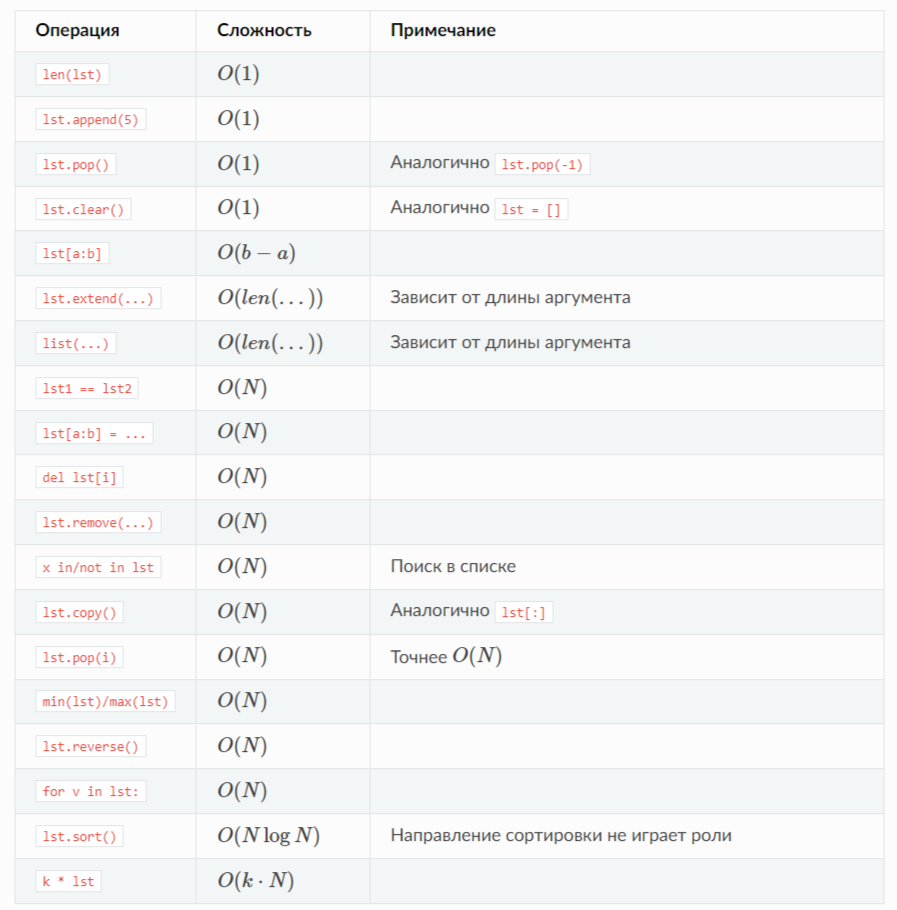
* используемых структур данных;
* совокупности различных операций.

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).



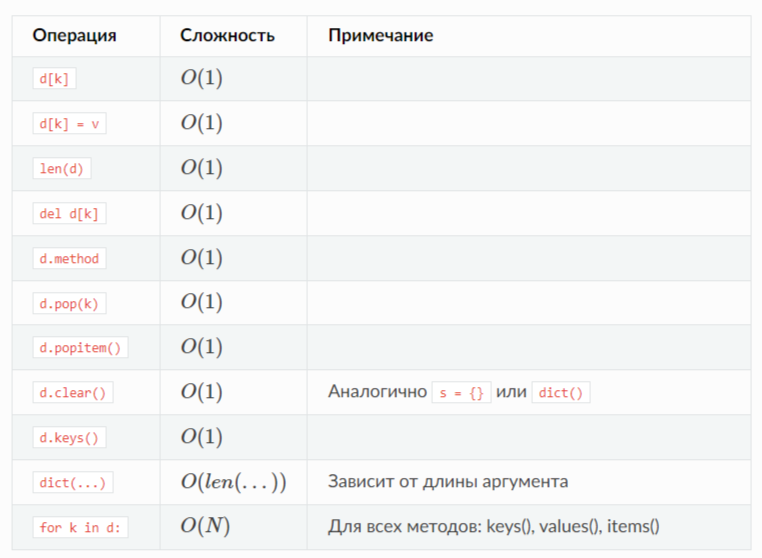
1.5.2. Множество

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



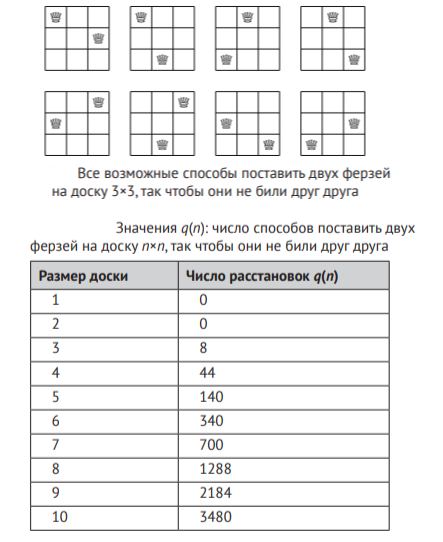
1.5.3. Словарь

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.

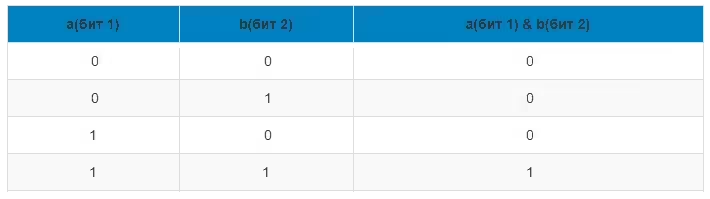
Для манипуляций с битами используют определённые операции — **логические или булевые**. Они могут применяться к любому биту, вне зависимости от того, какое у него значение — ноль или единица. Что же, давайте посмотрим на примеры использования трёх основных логических операций.

**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.

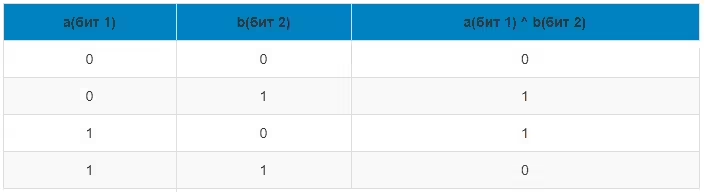


**Логическая операция XOR (исключающее ИЛИ)**

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**

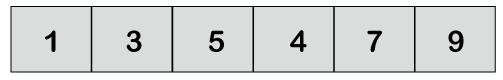
Линейный поиск – это метод поиска элементов в списке. Его еще называют последовательным поиском. Это простейший алгоритм поиска, поскольку он ищет желаемый элемент последовательно.

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

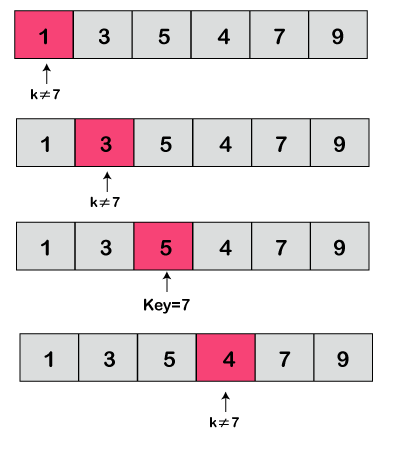
Концепция линейного поиска

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

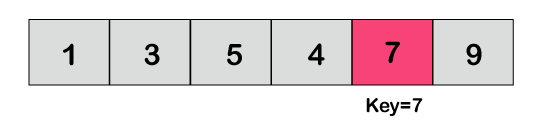
Шаг – 1: Начните поиск с первого элемента и проверьте ключ = 7 с каждым элементом списка x.



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:

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

Элементы, вышедшие из my\_stack:

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).

Пример -

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 списка**

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 –

from queue import LifoQueue

my\_stack = LifoQueue(maxsize = 5)

print(my\_stack.qsize())

my\_stack.put('x')

my\_stack.put('y')

my\_stack.put('z')

print("Стек заполнен: ", my\_stack.full())

print("Размер стека: ", my\_stack.qsize())

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

print(my\_stack.get())

print(my\_stack.get())

print(my\_stack.get())

print("\nСтек пуст: ", my\_stack.empty())

Выход:

0

0

Стек заполнен: False

Размер стека: 3

Элементы, удалённые из my\_stack

z

y

x

Стек пуст: 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.

**Тема 8. Теория графов. Определения графа, основных элементов графов. Типы графов. Деревья: число вершин и висячих вершин. Представления графов в памяти: с помощью матрицы смежности и списка ребер. DFS и BFS. Алгоритм. Восстановление пути. Алгоритм Дейкстры.**

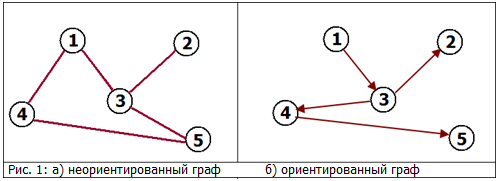
**Теория** **графов** – **это** область дискретной математики, особенностью которой является геометрический подход к изучению объектов и связей между ними. Объекты называются вершинами **графа**, связи между парами объектов – **ребрами**.

**В математике, Граф** — это абстрактное представление множества объектов и связей между ними. Графом называют пару (V, E) где V это множество вершин, а E множество пар, каждая из которых представляет собой связь (эти пары называют рёбрами).

**Вершина** - точка в графе, отдельный объект, для топологической модели графа не имеет значения координата вершины, её расположение, цвет, вкус, размер; однако при решении некоторых задачах вершины могут раскрашиваться в разные цвета или сохранять числовые значения.

**Ребро** - неупорядоченная пара двух вершин, которые связаны друг с другом. Эти вершины называются концевыми точками или концами ребра.

Граф может быть ориентированным или неориентированным. В ориентированном графе, связи являются направленными (то есть пары в E являются упорядоченными, например пары (a, b) и (b, a) это две разные связи). В свою очередь в неориентированном графе, связи ненаправленные, и поэтому если существует связь (a, b) то значит что существует связь (b, a).



*Пример:*

Неориентированный граф: Соседство (в жизни). Если (1) сосед (3), то (3) сосед (1).  
Ориентированный граф: Ссылки. Сайт (1) может ссылаться на сайт (3), но совсем не обязательно (хотя возможно) что сайт (3) ссылается сайт (1).

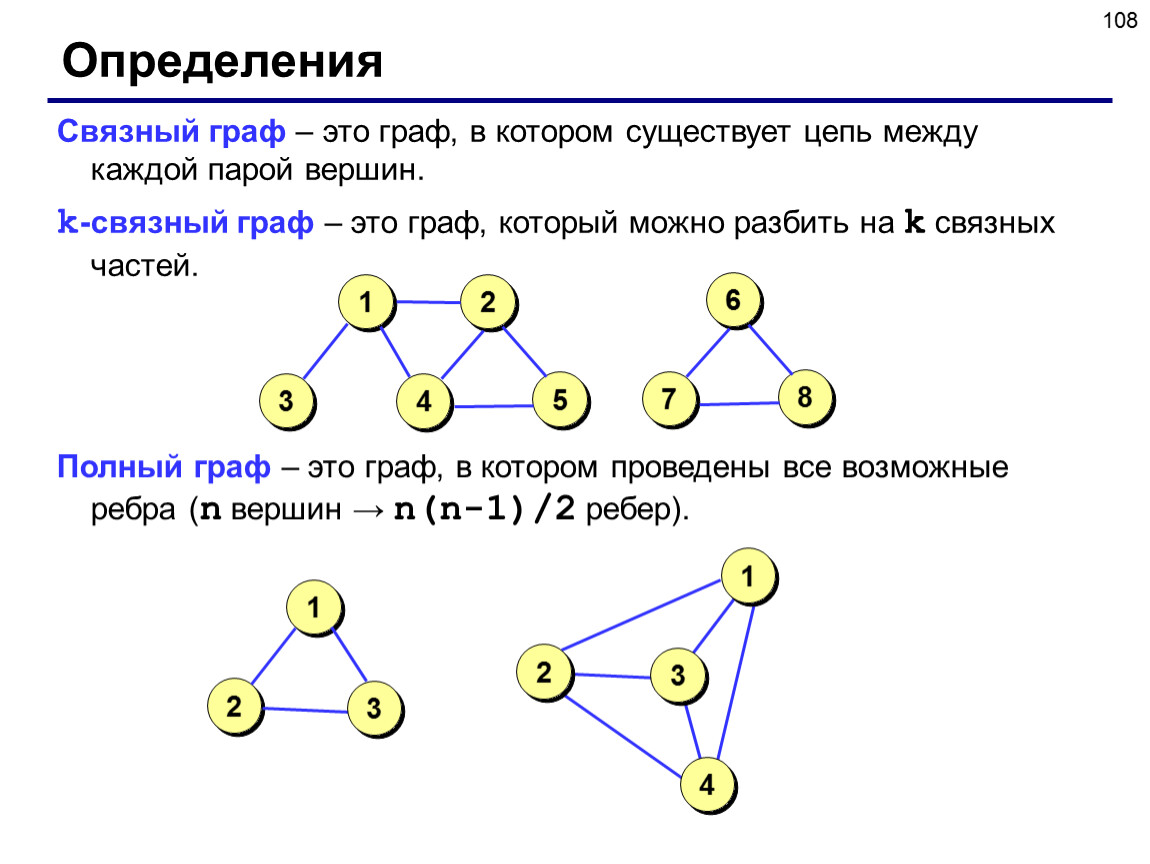
Степень вершины может быть входящая и исходящая (для неориентированных графов входящая степень равна исходящей).

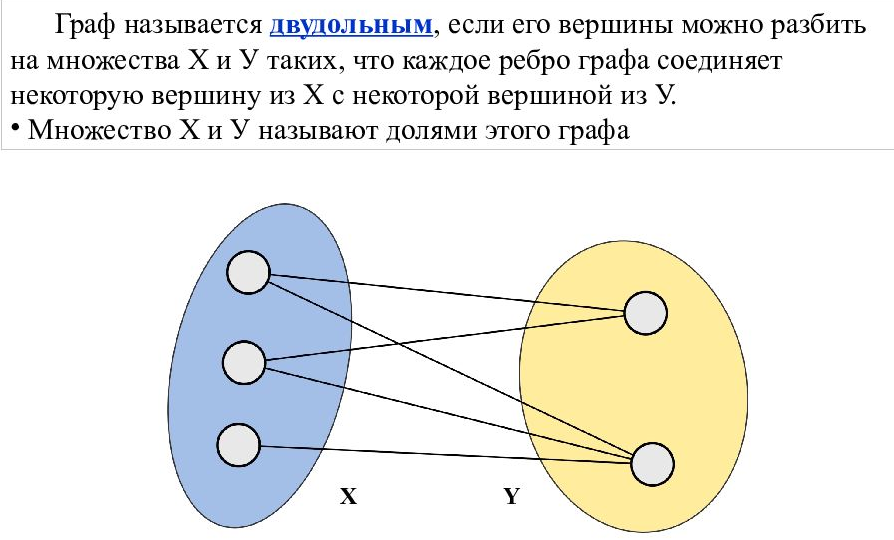
**Входящая степень вершины v** - это количество ребер вида (i, v), то есть количество ребер которые «входят» в v.

**Исходящая степень вершины v** - это количество ребер вида (v , i), то есть количество ребер которые «выходят» из v.

**Путь в графе -** это конечная последовательность вершин, в которой каждые две вершины идущие подряд соединены ребром. Путь может быть ориентированным или неориентированным в зависимости от графа. На рис 1.а, путем является например последовательность [(1), (4), (5)] на рис 1.б, [(1), (3), (4), (5)].

У графов есть ещё много разных свойств (например они могут быть связными, двудольными, полными).





**Циклом** называется замкнутый путь, не проходящий дважды через одну и ту же вершину.

**Деревом** называется связный граф, не содержащий циклов.

Таким образом, в дереве невозможно, перемещаясь по ребрам и не проходя по одному ребру два или более раз, вернуться в исходную вершину.

**Простым путём** называется путь, в котором никакое ребро не встречается дважды.

Легко проверить, что дерево — это граф, в котором любые две вершины соединены ровно одним простым путем. Если выкинуть любое ребро из дерева, то граф станет несвязным. Таким образом:

**Дерево** — минимальный по числу рёбер связный граф.

**Висячей вершиной** называется вершина, из которой выходит ровно одно ребро.

Подграф называется **остовным деревом**, если он является деревом и множество его вершин совпадает с множеством вершин исходного графа.

Теоремы:

1. В дереве с более чем одной вершиной есть висячая вершина.
2. В дереве число вершин на 1 больше числа рёбер.
3. У любого связного графа есть остовное дерево.

**Представление графов**

Существует два способа представления графа, в виде списков смежности и в виде матрицы смежности. Оба способа подходят для представления ориентированных и неориентированных графов.

**Матрица смежности**

Этот способ является удобным для представления плотных графов, в которых количество рёбер (|E|) примерно равно количеству вершин в квадрате (|V|2).

В данном представлении мы заполняем матрицу размером |V| x |V| следующим образом:

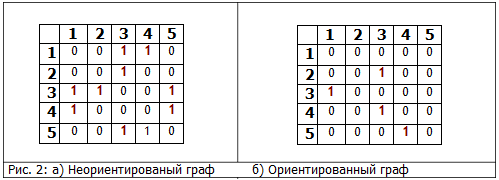
A[i][j] = 1 (Если существует ребро из i в j)

A[i][j] = 0 (Иначе)

Данный способ подходит для ориентированных и неориентированных графов. Для неориентированных графов матрица A является симметричной (то есть A[i][j] == A[j][i], т.к. если существует ребро между i и j, то оно является и ребром из i в j, и ребром из j в i). Благодаря этому свойству можно сократить почти в два раза использование памяти, храня элементы только в верхней части матрицы, над главной диагональю)

Понятно что с помощью данного способа представления, можно быстро проверить есть ли ребро между вершинами v и u, просто посмотрев в ячейку A[v][u].

С другой стороны этот способ очень громоздкий, так как требует O (|V|2) памяти для хранения матрицы.

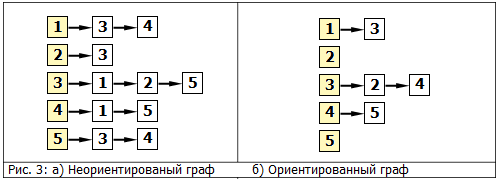


**Списки смежности**

Данный способ представления больше подходит для разреженных графов, то есть графов у которых количество рёбер гораздо меньше чем количество вершин в квадрате (|E| << |V|2).

В данном представлении используется массив Adj содержащий |V| списков. В каждом списке Adj[v] содержатся все вершины u, так что между v и u есть ребро. Память требуемая для представления равна O (|E| + |V|) что является лучшим показателем чем матрица смежности для разреженных графов.

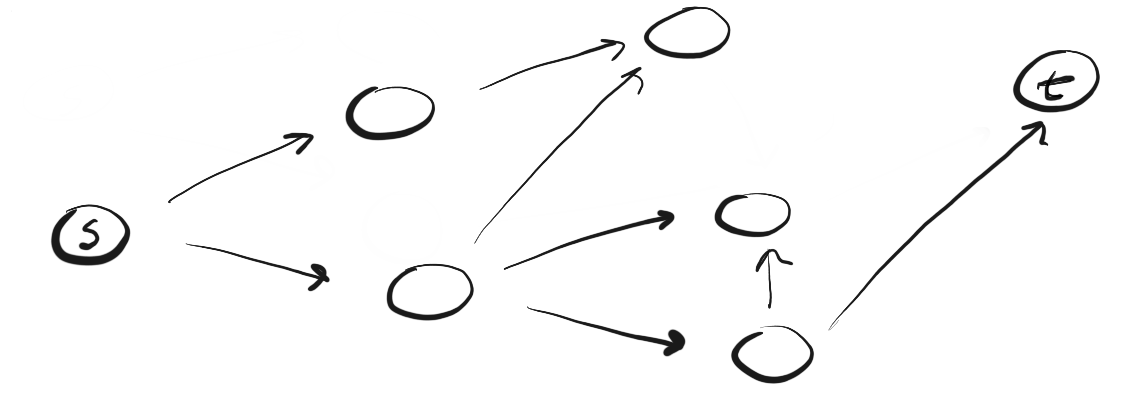
Главный недостаток этого способа представления в том, что нет быстрого способа проверить существует ли ребро (u, v).



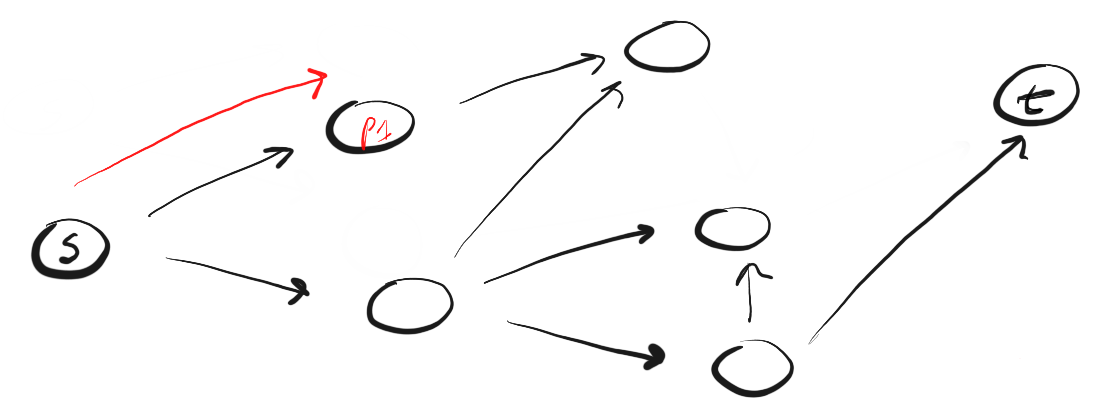
**Обход графа**

**Обход графа** — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами.  
Двумя основными алгоритмами обхода графа являются поиск в глубину (Depth-First Search, DFS) и поиск в ширину (Breadth-First Search, BFS).  
Несмотря на то, что оба алгоритма используются для обхода графа, они имеют некоторые отличия.

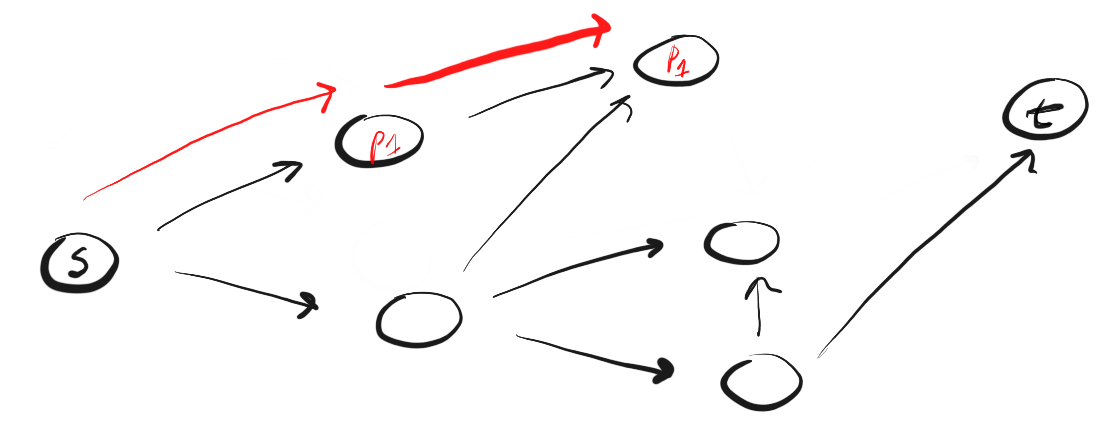
**Поиск в глубину**  
DFS следует концепции «погружайся глубже, головой вперед» («go deep, head first»). Идея заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.  
  
Рассмотрим пример. Предположим, что у нас есть ориентированный граф, который выглядит так:



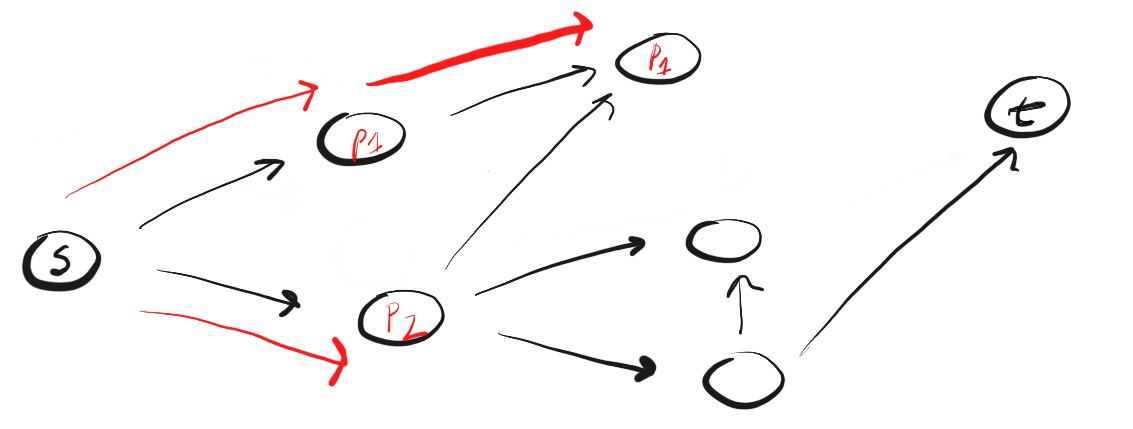
Мы находимся в точке «s» и нам нужно найти вершину «t». Применяя DFS, мы исследуем один из возможных путей, двигаемся по нему до конца и, если не обнаружили t, возвращаемся и исследуем другой путь. Вот как выглядит процесс:



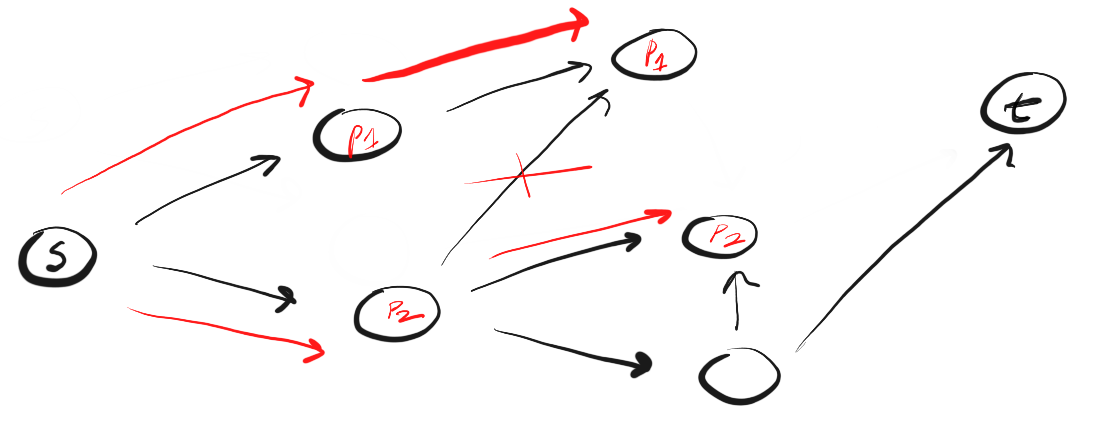
Здесь мы двигаемся по пути (p1) к ближайшей вершине и видим, что это не конец пути. Поэтому мы переходим к следующей вершине.



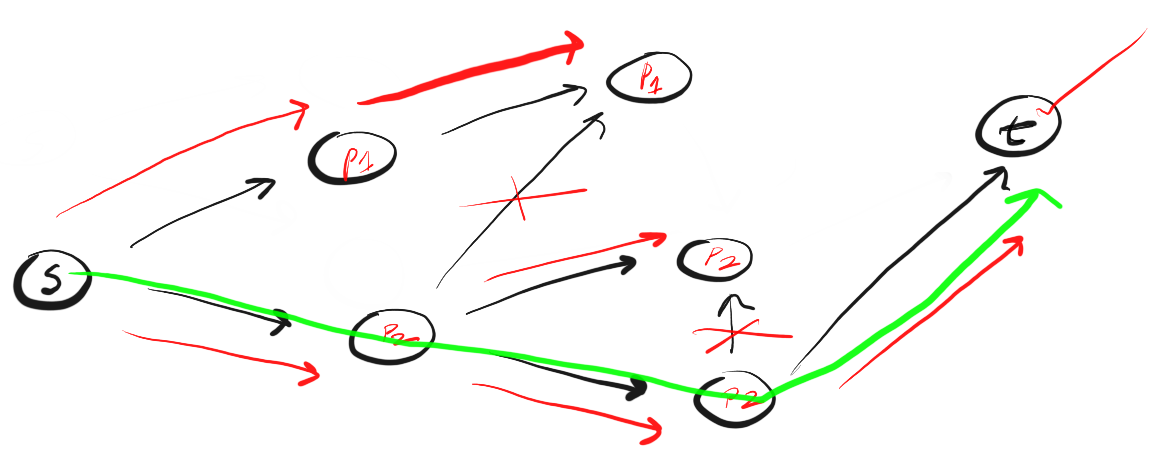
Мы достигли конца p1, но не нашли t, поэтому возвращаемся в s и двигаемся по второму пути.



Достигнув ближайшей к точке «s» вершины пути «p2» мы видим три возможных направления для дальнейшего движения. Поскольку вершину, венчающую первое направление, мы уже посещали, то двигаемся по второму.



Мы вновь достигли конца пути, но не нашли t, поэтому возвращаемся назад. Следуем по третьему пути и, наконец, достигаем искомой вершины «t».



Так работает DFS. Двигаемся по определенному пути до конца. Если конец пути — это искомая вершина, мы закончили. Если нет, возвращаемся назад и двигаемся по другому пути до тех пор, пока не исследуем все варианты.  
  
Мы следуем этому алгоритму применительно к каждой посещенной вершине.  
  
Необходимость многократного повторения процедуры указывает на необходимость использования рекурсии для реализации алгоритма.

Псевдокод поиска в глубину (рекурсивная реализация)

DFS(G, u)

u.visited = true

for each v ∈ G.Adj[u]

if v.visited == false

DFS(G, v)

init() {

for each u ∈ G

u.visited = false

for each u ∈ G

DFS(G, u)

}

Реализация поиска в глубину на Python

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start)

for next in graph[start] - visited:

dfs(graph, next, visited)

return visited

graph = {’0’: set([’1’, ‘2’]),

‘1’: set([’0’, ‘3’, ‘4’]),

‘2’: set([’0’]),

‘3’: set([’1’])

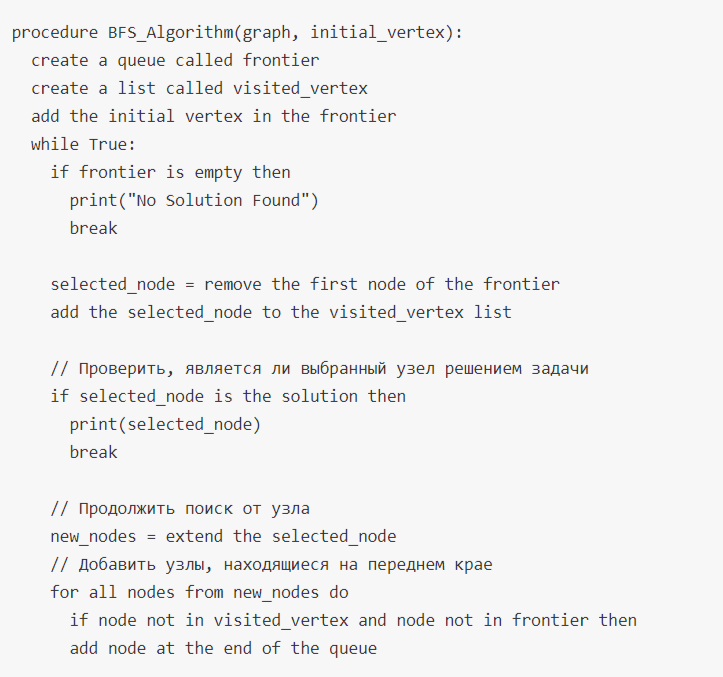
‘4’: set([’2’, ‘3’])}

dfs(graph, ‘0’)

**Поиск в ширину (BFS)**

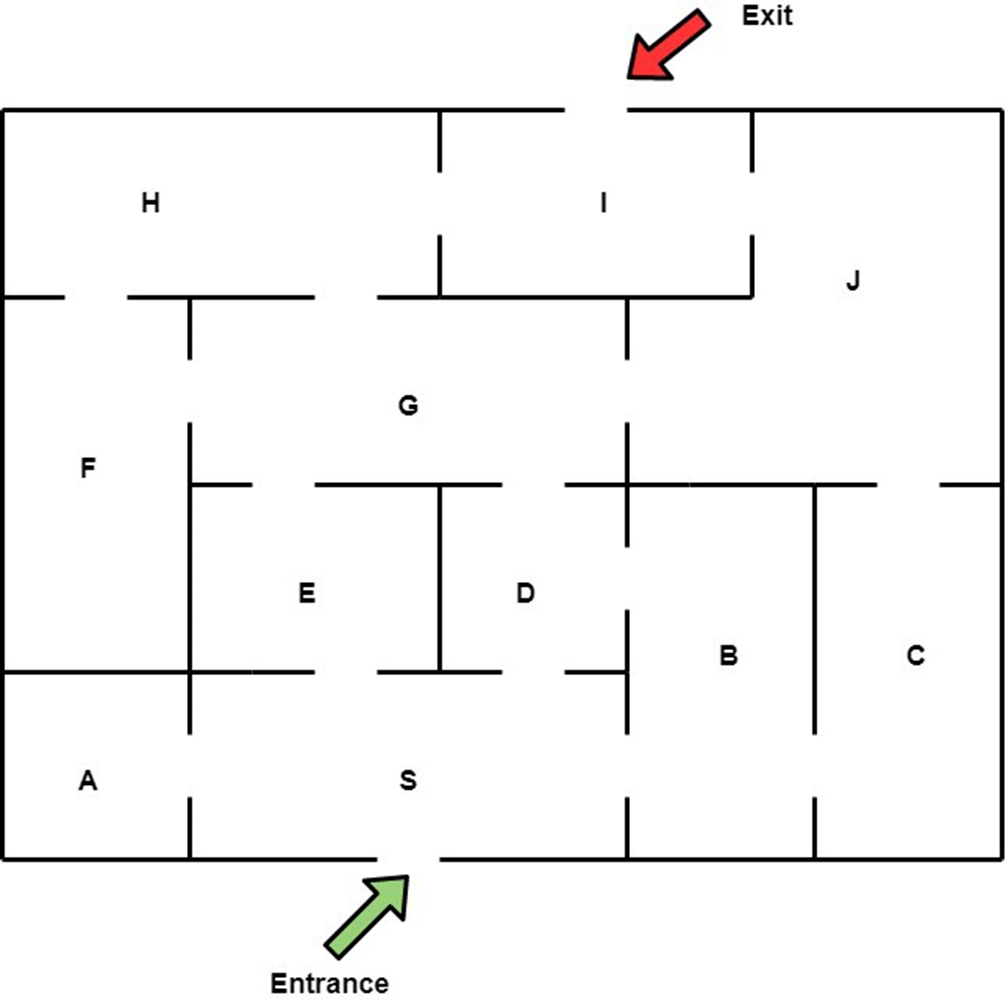
Алгоритмы поиска применяются для решения таких задач, которые можно смоделировать как графы. Каждый узел графа – это экземпляр задачи. Каждый поисковый алгоритм начинается с узла (исходный экземпляр – состояние) и наращивает вслед за этим узлом новые (то есть, новые экземпляры задачи), решая задачу допустимыми способами. Этот процесс останавливается, как только алгоритм находит решение (успех – конечное состояние) или не может создать ни одного нового узла (провал).

**Поиск в ширину** – это «слепой» алгоритм. Он называется «слепым», так как не учитывает стоимости перехода между вершинами графа. Алгоритм начинает работу с корневого узла (представляющего собой исходное состояние задачи) и исследует все узлы на рассматриваемом уровне, а только после этого переходит к узлам следующего уровня. Если алгоритм находит решение, то он возвращается и прекращает поиск, в противном случае наращивает от узла новое ребро и продолжает поиск. Алгоритм поиска в ширину является «полным» - это означает, что он всегда возвращает решение, если оно существует. Точнее, алгоритм возвращает то решение, которое ближе всего к корню. Поэтому в задачах, где переход от узла к любому его дочернему узлу стоит единицу, алгоритм BFS возвращает наилучшее решение. Кроме того, чтобы исследовать узлы уровень за уровнем, он использует структуру данных под названием [очередь](https://python.plainenglish.io/queue-data-strucure-theory-and-python-implementation-e58f3582c390), так что новые узлы добавляются в хвост очереди, а старые узлы удаляются из головы очереди. Псевдокод для алгоритма BFS выглядит так:

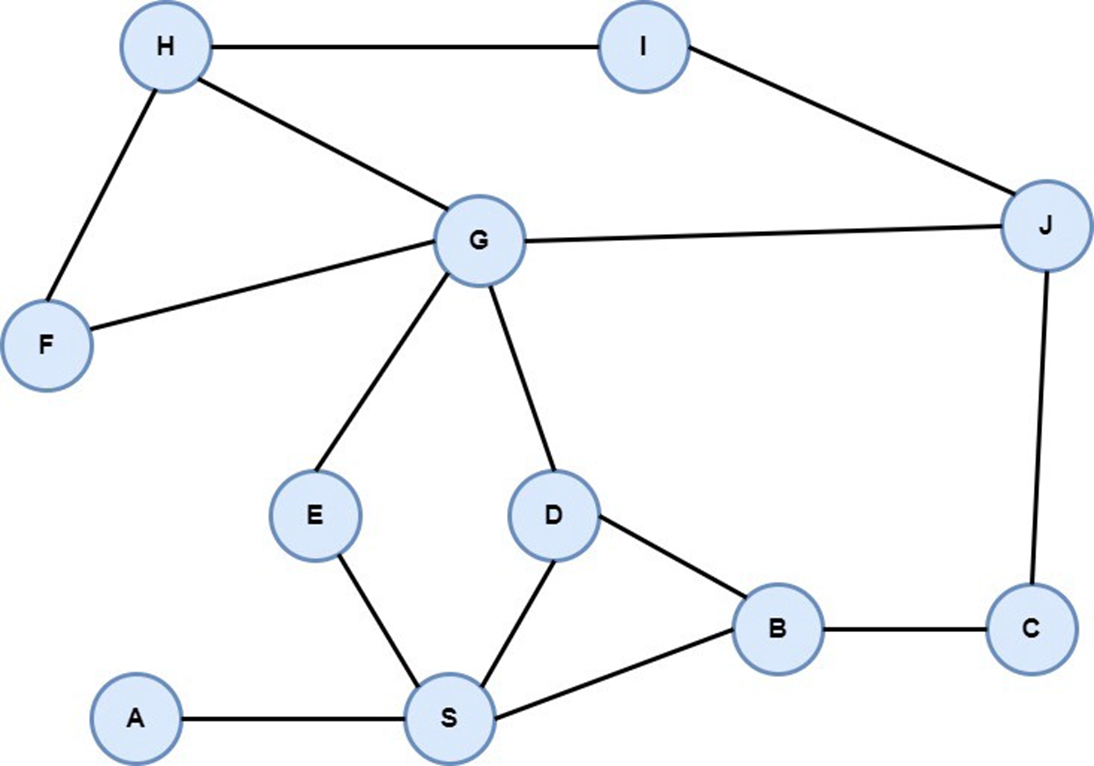


Из вышеприведенного кода очевидно, что для решения задачи при помощи алгоритма поиска, например, BFS, необходимо смоделировать задачу в форме графа и определить ее исходное состояние (начальный узел). После этого требуется найти такие правила, которым нужно будет следовать, чтобы наращивать узлы (экземпляры задачи). Эти правила определяются самой задачей. Последнее, что нам остается сделать – определить целевой узел или механизм, такой, при помощи которого алгоритм мог бы распознать целевой узел.

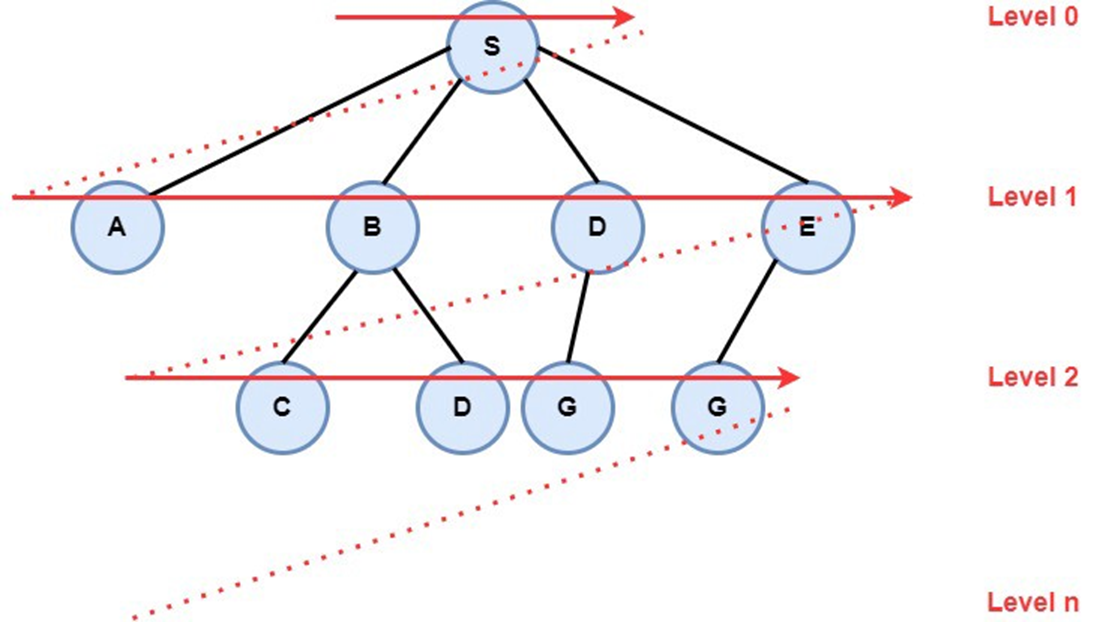
Итак, мы в общих чертах разобрались, как работает поиск в ширину, обсудим задачу, которую станем решать при помощи этого алгоритма. Допустим, имеется лабиринт, такой, как на следующем рисунке, и мы хотим перейти от входа к выходу за наименьшее возможное количество шагов. За один шаг будем считать любой переход из одной комнаты в другую. В нашем лабиринте 11 комнат, и у каждой из них – уникальное имя, например, “A”, “B”, т.д. Итак, наша цель – перейти из комнаты “S” в комнату “I”.



Итак, мы определили задачу, теперь нужно смоделировать ее в виде графа. Чаще всего для этого создается *вершина на каждую комнату* и *ребро на каждую дверь* в лабиринте. После такого моделирования получается показанный ниже граф – в нем 11 вершин и 15 рёбер.



Итак, от каждой вершины мы можем перейти к соседним, начиная от вершины “S” (это начальное состояние) до вершины “I” (это целевой узел, по достижении которого задача решена). Как я уже упоминал выше, алгоритм поиска в ширину обследует все узлы на актуальном уровне, а потом перейдет к узлам следующего уровня – как показано на следующей картинке.



**Восстановление пути. Алгоритм Дейкстры.**

**Алгоритм Дейкстры** находит кратчайший путь между двумя вершинами графа. Следовательно, если математические задачи моделируется при помощи графа, используя алгоритм Дейкстры, можно найти кратчайший путь между вершинами.

Перед непосредственным составлением кода, осветим ключевые моменты темы:

Главное условие: отрицательных длин ребер не бывает.

Алгоритм Дейкстры изначально создавался для поиска кратчайшего пути между двумя конкретными узлами. Однако сегодня он также широко используется для поиска кратчайших путей между узлом источника и всеми остальными узлами.

**Обзор кода алгоритма Дейкстры на Python**

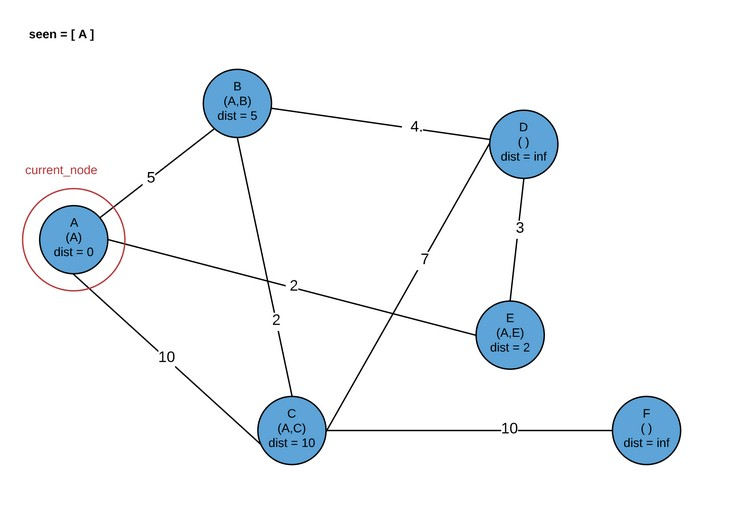
**Инициализация**

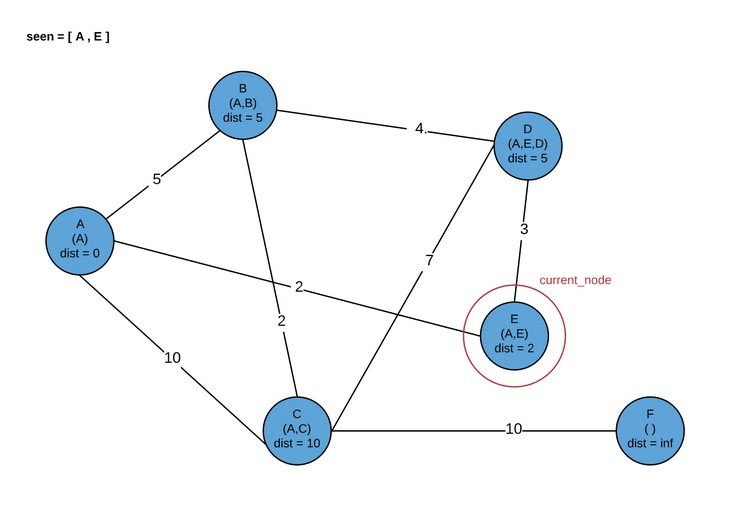
1. Установите provisional\_distance для всех узлов от исходного узла до бесконечности.
2. Определите пустой набор seen\_nodes. Данный набор гарантирует, что узел, у которого уже есть кратчайший путь, не будет повторно рассмотрен, а также то, что не будут рассматриваться пути через узел, у которого более короткий путь к источнику, чем текущий путь. Помните, что узлы входят в seen\_nodes только после доказательства того, что в наличии есть абсолютное кратчайшее расстояние (а не только предварительное расстояние). Набор используется для получения времени поиска O(1) вместо многократного выполнения поиска через массив O(n).
3. Установите provisional\_distance для исходного узла со значением 0, и массив, представляющий перескоки для простого включения самого исходного кода. Это будет полезно позже, когда мы проследим выбранный для графа путь для расчета минимального расстояния.

**Процедура итерации**

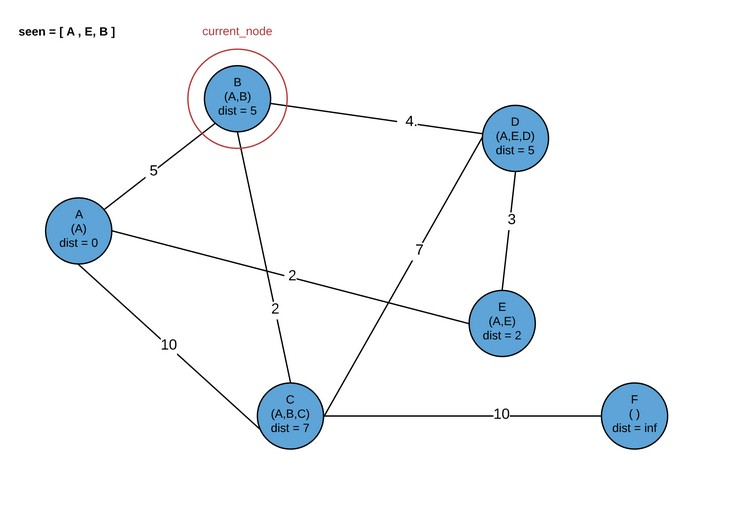
1. Пока (while) все узлы увидеть (seen) не удалось. Или, в случае поиска одного узла назначения, пока не удалось увидеть (seen) данный узел назначения.
2. Установите current\_node для узла c самым малым предварительным расстоянием provisional\_distance во всем графе. Обратите внимание, что для первой итерации это будет исходный узел source\_node, так как предварительное расстояние provisional\_distance установлено на 0.
3. Добавьте текущий узел current\_node к набору просмотренных узлов seen\_nodes.
4. Обновите provisional\_distance каждого соседнего элемента current\_node до (абсолютного) расстояния от current\_node до source\_node вдобавок к длине ребра от current\_node к данному соседнему элементу, ЕСЛИ данное значение меньше, чем текущее значение соседнего provisional\_distance. Если у соседнего элемента еще не было набора предварительного расстояния, помните, что он инициализирован до бесконечности, и по этой причине должен быть больше, чем данная сумма. При обновлении provisional\_distance также обновляются «перескоки», которые были сделаны для получения расстояния, задействуя конкатенацию перескоков current\_node к исходному узлу с самим current\_node.
5. Завершение цикла while.

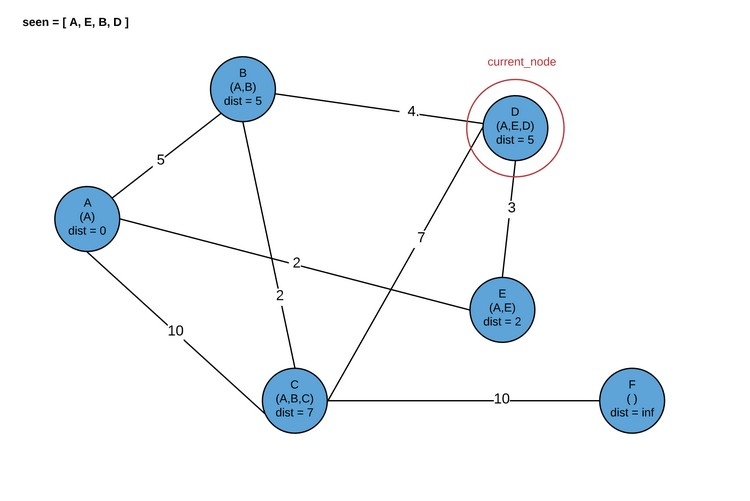
**Алгоритм Дейкстры через схемы и изображения**

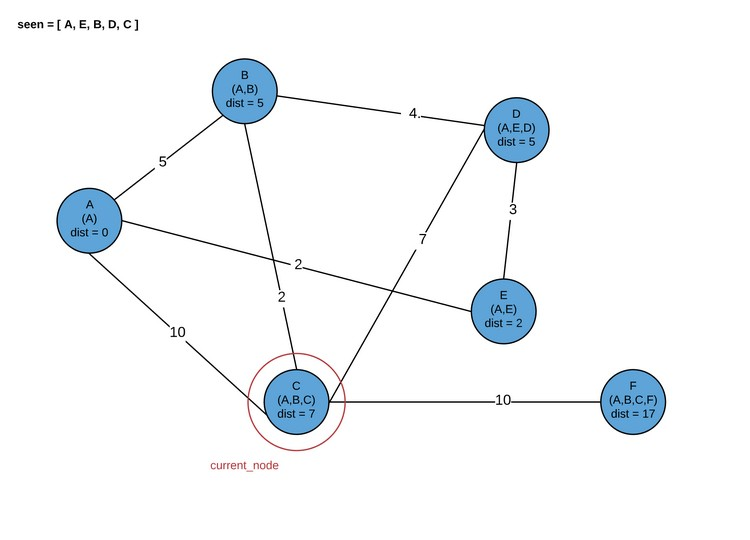
****

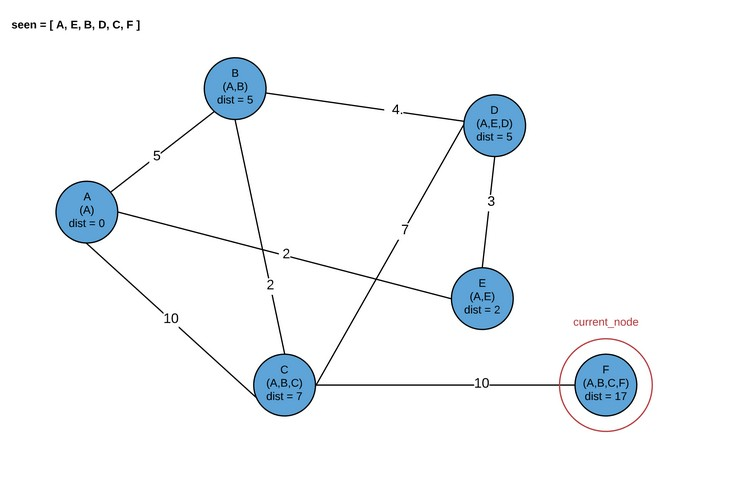
****

Обратите внимание, что в дальнейшем можно посетить либо D, либо B. Сейчас мы посетим B.

****

****

****

****

**Здесь программа завершается. В результате мы получаем кратчайшие пути для каждого узла графа.**

**Python код для алгоритма Дейкстры**

**def**dijkstra(**self**,node):

*# Получает индекс узла (или поддерживает передачу int)*

nodenum=**self**.get\_index\_from\_node(node)

*# Заставляет массив отслеживать расстояние от одного до любого узла*

*# в self.nodes. Инициализирует до бесконечности для всех узлов, кроме*

*# начального узла, сохраняет "путь", связанный с расстоянием.*

*# Индекс 0 = расстояние, индекс 1 = перескоки узла*

dist=[**None**]\*len(**self**.nodes)

**for**i**in**range(len(dist)):

dist[i]=[float("inf")]

dist[i].append([**self**.nodes[nodenum]])

dist[nodenum][0]=0

*# Добавляет в очередь все узлы графа*

*# Отмечает целые числа в очереди, соответствующие индексам узла*

*# локаций в массиве self.nodes*

queue=[i**for**i**in**range(len(**self**.nodes))]

*# Набор увиденных на данный момент номеров*

seen=set()

**while**len(queue)>0:

*# Получает узел в очереди, который еще не был рассмотрен*

*# и который находится на кратчайшем расстоянии от источника*

min\_dist=float("inf")

min\_node=**None**

**for**n**in**queue:

**if**dist[n][0]<min\_dist **and**n**notin**seen:

min\_dist=dist[n][0]

min\_node=n

*# Добавляет мин. расстояние узла до увиденного, убирает очередь*

queue.remove(min\_node)

seen.add(min\_node)

*# Получает все следующие перескоки*

connections=**self**.connections\_from(min\_node)

*# Для каждой связи обновляет путь и полное расстояние от*

*# исходного узла, если полное расстояние меньше*

*# чем текущее расстояние в массиве dist*

**for**(node,weight)**in**connections:

tot\_dist=weight+min\_dist

**if**tot\_dist<dist[node.index][0]:

dist[node.index][0]=tot\_dist

dist[node.index][1]=list(dist[min\_node][1])

dist[node.index][1].append(node)

**return**dist

**Тема 9. Динамика на одномерных и двумерных массивах. Восстановление ответа.**

**Динамическое программирование**

**Общие принципы**

Рассмотрим такую задачу: найти N-е число Фибоначчи.

Числа Фибоначчи определяются так: F0 = 0, F1 = 1, Fn = Fn-2 + Fn-1.

Эту задачу можно решить рекурсивно:

def fibonacchi(n):

if n == 0:

return 0

if n == 1:

return 1

return fibonacchi(n - 2) + fibonacchi(n - 1)

print fibonacchi(20)

6765

Однако это будет работать очень долго. 20-е число посчитать еще можно будет, а 40-е число - нет. И не потому, что числа большие. А потому, что мы будем делать слишком много лишней работы. Число операций будет экспоненциально относительно N.

Потому что, чтобы посчитать N-е число, нам нужно будет независимо посчитать (N-1)-е число и (N-2)-е число, и это в минимум в два раза больше действий, чем нужно для (N-2). А значит, для подсчета N-го числа Фибоначчи необходимо 2 раза посчитать (N-2)-е число, и это занимает в два раза больше времени, а значит это хотя бы 2N/2 действий. Это слишком долго, и главное, что это легко исправляется. Давайте просто не считать лишних действий - если мы один раз посчитали Fk то давайте запомним, чему оно равно, и в следующий раз, когда оно нам понадобится, мы используем его сразу. Удобнее всего сохранить числа Фибоначчи прямо в массиве:

N = 20

fib = [None] \* (N + 1) # создали массив из None от 0 до N включительно

fib[0] = 0

fib[1] = 1 # заполнили изначальные позиции

for i in range(2, N + 1): # обходим массив по порядку слева направо

fib[i] = fib[i - 2] + fib[i - 1] # формула считает новое число через предыдущие!

print fib[N] # ответ лежит на N-ом месте

6765

И этот способ легко работает для больших N, так как работает за O(N) - всего один проход по массив.

N = 200000

fib = [0] \* (N + 1)

fib[1] = 1

for i in range(2, N + 1):

fib[i] = (fib[i - 2] + fib[i - 1]) % (10 \*\* 9)

print fib[N]

175853125

Это и называется динамическим программированием (или динамикой, ДП). Основная идея состоит в том, чтобы \* свести задачу для N к задаче для чисел, меньших, чем N (с помощью формулы) \* хранить все ответы в массиве \* заполнить начало массива вручную (для которых формула не работает) \* обойти массив и заполнить ответы по формуле \* вывести ответ откуда-то из этого массива

Чтобы решить задачу по динамике вы должны ответить на 5 вопросов: \* Что лежит в массиве? (самый важный вопрос чаще всего) \* Как инициализировать начало массива? \* Как обходить массив? (чаще всего слева направо, но не всегда) \* Какой формулой считать элементы массива? \* Где в массиве лежит ответ?

**Одномерная динамика: кузнечик**

Рассмотрим такую задачу:

Есть полоска 1 \* N, кузнечик стоит на первой клетке, он может прыгать вперед на 1, 2, 3 клетки. Сколько есть способов добраться от начальной клетки до последней?

Как решать такие задачи? Нужно придумать рекуррентную формулу, как ответ для N зависит от ответа для меньших чисел.

Очень помогает посмотреть на маленькие числа (!! одна из самых важных идей для придумывания решений):

Пусть dp[x] - это количество способов добраться от 1 клетки до клетки номер x.

* dp[1] = 1 способ (стоять на месте)
* dp[2] = 1 способ (1 -> 2)
* dp[3] = 2 способа (1 -> 2 -> 3 и 1 -> 3)
* dp[4] = 4 способа (1 -> 2 -> 3 -> 4 и 1 -> 3 -> 4 и 1 -> 2 -> 4 и 1 -> 4)
* dp[5] = 7 способов (1 -> 2 -> 3 -> 4 -> 5 и 1 -> 3 -> 4 -> 5 и 1 -> 2 -> 4 -> 5 и 1 -> 4 -> 5 и 1 -> 2 -> 3 -> 5 и 1 -> 3 -> 5 и 1 -> 2 -> 5)

Дальше становится сложнее. Но можно заметить закономерность. А можно и не заметить, но зато если мы сейчас придумаем формулу, мы легко проверим, работает ли она. Заодно мы получили наши значения на маленьких числах, которые нам все равно понадобится вбить в программу.

Какой последний прыжок кузнечика в его пути до N-й клетки? Один из трёх вариантов: \* (N - 1) -> N \* (N - 2) -> N \* (N - 3) -> N

То есть все пути до N разбиваются на 3 группы. Причем мы знаем сколько путей в каждой группе. В первой из них ровно dp[N - 1] путей - столько путей идут до (N-1)-й клетки, и дальше идет еще один прыжок. Во второй и третьей группах поэтому тоже dp[N - 2] и dp[N-3] путей.

Так что формула получается такой: dp[N] = dp[N - 3] + dp[N - 2] + dp[N - 1].

Очень похоже на числа Фибоначчи, да? Можете посмотреть на числа, которые мы уже выписали, там все отлично подходит:

dp[4] = 4 = 1 + 1 + 2 = dp[1] + dp[2] + dp[3]

dp[5] = 7 = 1 + 2 + 4 = dp[2] + dp[3] + dp[4].

Так что программа пишется легко:

N = 20

dp = [0] \* (N + 1)

dp[1] = 1

dp[2] = 1

dp[3] = 2

for i in range(4, N + 1):

dp[i] = dp[i - 3] + dp[i - 2] + dp[i - 1]

print(dp)

[0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, 5768, 10609, 19513, 35890, 66012]

Давайте изменим немного задачу: Теперь некоторые из клеток закрыты. То есть нам известно про конкретные клетки, что на них кузнечик прыгать не может. Тогда задача все еще решается так же, только нужно убедиться, что dp[x] = 0 для всех запрещенных x!

Также немного перепишем код, чтобы не писать отдельно случаи для 2 и 3, а также чтобы не писать в формуле сумму трех чисел (а представьте, что в задаче не 3, а 100). Будем инициализировать только dp[1]. А ко всем следующим значениям dp[i] будет прибавлять dp[i - k], где k = 1, 2, 3. Причем, если i - k < 1, то мы будем игнорировать такие клетки, и этим самым мы избавились от необходимости прописывать ответ для dp[2] и dp[3].

N = 20

BAD\_CELLS = [2, 3, 6, 13]

dp = [0] \* (N + 1)

is\_bad = [0] \* (N + 1)

for bad in BAD\_CELLS:

is\_bad[bad] = 1

print(is\_bad)

[0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

if not is\_bad[1]:

dp[1] = 1

for i in range(2, N + 1):

if not is\_bad[i]:

for k in range(1, 4):

if i - k >= 1:

dp[i] += dp[i - k]

print(dp)

[0, 1, 0, 0, 1, 1, 0, 2, 3, 5, 10, 18, 33, 0, 51, 84, 135, 270, 489, 894, 1653]

**Двумерная динамика: черепашка**

Теперь рассмотрим такую задачу:

На каждой клетке двумерной таблички написано, сколько там лежит монет. Черепашка стоит в клетке 1x1 (верхней левой), и может двигаться только на одну клетку вниз, или на одну клетку вправо. Нужно найти максимальное число монет, которое может набрать черепашка по пути к нижней правой клетке NxM.

Первое, что приходит в голову - это просто идти черепашкой в ту клетку из соседних, где лежит больше монет. К сожалению, эта жадная стратегия не всегда работает. Например, на такой доске жадная черепашка пошла бы по следу из единичек, хотя гораздо выгоднее пойти сначала по нулям, а потом найти там большие горстки монет (40, 70, 100):

COINS = [

[0, 1, 1, 1, 1, 1],

[0, 0, 0, 0, 0, 1],

[0, 40, 70, 0, 0, 1],

[100, 0, 0, 0, 0, 1]

]

N = 4

M = 6

Тут нас снова спасает динамика. Давайте сводить задачу к предыдущей! Задачей назовем “сколько максимально монет можно набрать на пути от 0 \* 0 до i \* j’’ (заменим 1-нумерацию на 0-нумерацию). Будем хранить это в двумерном массиве dp в клетке dp[i][j].

Сразу понятны некоторые свойства этого массива: \* Он размера NxM \* dp[0][0] = COINS[0][0] \* ответ на всю задачу лежит в dp[N - 1][M - 1]

Но гораздо важнее придумать формулу для подсчета dp[i][j] через предыдущие. Легко посчитать первую строку и первый столбец: \* dp[0][k] = dp[0][k - 1] + COINS[0][k] \* dp[k][0] = dp[k - 1][0] + COINS[k][0]

Так как до этих клеток есть ровно один путь.

Но что делать, если есть много путей до клетки dp[i][j]? Снова разобьем их на на несколько групп в зависимости от последнего хода (! важный трюк, запомните). Последний ход был: \* либо из [i][j - 1] \* либо из [i - 1][j]

Поэтому формула для максимального числа монет такая: dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + COINS[i][j].

Ну все, достаточно пройтись правильно по двумерному массиву (построчно сверху вних, а в каждой строке слева направо) и заполнить этот массив.

dp = [[None] \* M for i in range(N)]

for i in range(N):

for j in range(M):

if i == 0 and j == 0:

dp[0][0] = COINS[0][0]

elif i == 0:

dp[0][j] = dp[0][j - 1] + COINS[0][j]

elif j == 0:

dp[i][0] = dp[i - 1][0] + COINS[i][0]

else:

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + COINS[i][j]

print(dp[i])

[0, 1, 2, 3, 4, 5]

[0, 1, 2, 3, 4, 6]

[0, 41, 111, 111, 111, 112]

[100, 100, 111, 111, 111, 113]

В отличие от жадного алгоритма, динамическое программирование находит оптимальное решение - это, в данном случае, 113 монет.

**Восстановление ответа**

В последней задаче было здорово найти, что в оптимальном пути черепашки набирается 113 монет, но интересно, что именно это за путь. Такую задачу называют восстановлением ответа в динамике.

Есть два способа, которыми можно это сделать.

1. Хранить в массив prev откуда ты пришел в эту клетку.

Когда мы выбираем максимум из левой и верхней клетки, мы на самом деле решаем, какой последний ход будет в оптимальном пути до этой клетки - сверху или слева, и берем ответ для той клетки, сложнный с монетами в этой клетке. Давайте координаты клетки, откуда мы пришли, хранить в массиве prev. Или, в данном случае, можно хранить не координаты а просто 1, если пришли слева, и 0, если пришли сверху.

dp = [[None] \* M for i in range(N)]

prev = [[None] \* M for i in range(N)]

for i in range(N):

for j in range(M):

if i == 0 and j == 0:

dp[0][0] = COINS[0][0]

prev[0][0] = -1 # это самое начало, предыдущей клетки нет

elif i == 0:

dp[0][j] = dp[0][j - 1] + COINS[0][j]

prev[0][j] = 0 # слева пришли

elif j == 0:

dp[i][0] = dp[i - 1][0] + COINS[i][0]

prev[i][0] = 1 # сверху пришли

else:

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + COINS[i][j]

if dp[i - 1][j] > dp[i][j - 1]:

prev[i][j] = 1

else:

prev[i][j] = 0

print(prev[i])

[-1, 0, 0, 0, 0, 0]

[1, 1, 1, 1, 1, 1]

[1, 1, 0, 0, 0, 0]

[1, 0, 1, 0, 0, 1]

И, чтобы восстановить ответ, надо просто пройтись с конца по этим клеткам до самого начала, и развернуть получившуюся последовательность.

i, j = N - 1, M - 1

answer = []

answer\_directions = []

while i > 0 or j > 0:

if prev[i][j] == 1:

i -= 1

answer\_directions.append('DOWN')

else:

j -= 1

answer\_directions.append('RIGHT')

answer.append((i, j))

print answer[::-1] # reverse

print answer\_directions[::-1] # reverse

[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5)]

['RIGHT', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']

1. Вместо хранения массива prev догадаться по массиву dp, откуда именно черепашка пришла в эту клетку.

В данном примере это довольно легко. Если мы уже посчитали весь массив dp, то теперь можно начиная с конца легко понять, пришла черепашка туда сверху или слева в оптимальном маршруте - она пришла из клетки с максимальным числом монет.

i, j = N - 1, M - 1

answer = []

answer\_directions = []

while i > 0 or j > 0:

if i != 0 and (j == 0 or dp[i - 1][j] > dp[i][j - 1]):

i -= 1

answer\_directions.append('DOWN')

else:

j -= 1

answer\_directions.append('RIGHT')

answer.append((i, j))

print answer[::-1] # reverse

print answer\_directions[::-1] # reverse

[(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5)]

['RIGHT', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'RIGHT', 'RIGHT', 'DOWN']

**Тема 10. Интересные задачи на динамическое программирование (классический рюкзак, НВП, НОП)**

**Рюкзак**

В самой простой форме задача о рюкзаке формулируется так: > Даны n предметов с весами a1, ... , an. Определите, на какой максимальный вес можно набрать предметов в рюкзак вместимости W.

Для решения этой задачи воспользуемся динамическим программированием. Обозначим за dp[i][j] состояние, когда мы рассмотрели первые i предметов и набрали ими

j веса. dp[i][j] = True, если такая ситуация возможна, иначе dp[i][j] = False.

Для каждого состояния dp[i][j], которое возможно получить, мы можем либо взять предмет номер i и попробовать обновить ответ из состояния dp[i - 1][j - a[i]], либо не брать его и обновить ответ из dp[i - 1][j]. Очевидно, что мы можем получить 0 веса, рассмотрев 0 предметов.

dp[0][0] = True

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

for j in range(0, W + 1):

dp[i][j] = dp[i - 1][j]

if a[i] <= j and dp[i - 1][j - a[i]]:

dp[i][j] = True

Ответом будет максимальное i, такое что dp[n][j] = True. Таким образом, мы получили решение за O(nW).

**Наибольшая возрастающая подпоследовательность**

Пусть, дана последовательность из n чисел a1, ..., an. Требуется найти длину ее наибольшей возрастающей подпоследовательности (НВП), то есть длину такой наибольшей последовательности индексов i1 < i2 < ... < ik, что a[i1] < a[i2] < ... < a[ik].

Пример: в последовательности 100, 20, 75, 0, -40, 80, -10, 120, 110 наибольшей возрастающей подпоследовательность является 20, 75, 80, 120: она имеет длину 4. Возрастающих подпоследовательностей длины 5 здесь нет.

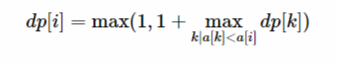
**НВП за O(N2)**

Давайте решать наивно через динамческое программирование - то есть хранить в dp[i] ровно то, что нам надо найти - длину НВП для первых i чисел.

dp[0] = 0. Но как найти формулу, выражающую dp[i] через предыдущин значения?

Есть два варианта: \* i-ое число не входит в НВП. Тогда dp[i] = 1 \* i-ое число входит в НВП. Тогда dp[i] = 1 + dp[k], где k - индекс предыдущего числа в этой НВП. Так давайте просто его переберем. При этом надо учесть, что a[k] должно быть меньше, чем a[i]!

Итогвоая формула получается такая:



Этот алгоритм работает за O(N2): у нас O(N) состояний динамики, и каждое из них мы считаем за O(N) действий, пока ищем этот максимум.

Ответ восстанавливается тем же способом: для каждого состояния нужно сохранить, где был этот максимум - там и есть предыдущее число в НВП.

**НВП за O(N log N)**

Решим эту задачу чуть более нестандартным динамическим программированием, где

min\_end[i] будет обозначать минимальное число, на которое может заканчиваться НВП длины i. При этом мы будем постепенно обрабатывать числа слева направо, и в этом массиве будет храниться только информация про все НВП в уже обработанном начале последовательности.

Изначально min\_end[0] = -∞ , min\_end[i] = ∞ для i > 0. В качестве ∞ надо выбрать число, которое заведомо больше любого из a[i], аналогично с - ∞.

Рассматривая очередной элемент, попробуем продлить им каждую подпоследовательность:

n = 6

a = [6, 1, 5, 3, 4, 2] # НВП: 1, 3, 4

inf = 100

min\_end = [-inf] + [inf] \* n

for i in range(n):

for j in range(n):

if min\_end[j - 1] < a[i] < =min\_end[j]:

min\_end[j] = a[i]

print(dp)

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

NameError Traceback (most recent call last)

<ipython-input-1-930ff2a8d8b0> in <module>()

7 if min\_end[j - 1] < a[i] < min\_end[j]:

8 min\_end[j] = a[i]

----> 9 print(dp)

NameError: name 'dp' is not defined

Ответом будет максимальный такой индекс j, что min\_end[j] != 0. Это решение работает за O(N2).

Его можно значительно ускорить, заметив два факта: - На любом шаге min\_end[i - 1] <= min\_end[i]. Это легко доказать от противного. - Из предыдущего факта следует, что любое a[i] обновит максимум одно значение динамики, так как попадет максимум в один интервал.

Значит, для поиска j, которое обновится можно воспользоваться бинарным поиском. Это решение уже работает за O(n log n).

**Наибольшая общая подпоследовательность**

Даны две последовательности a1, ... , an и b1, ... , bm. Требуется найти длину их наибольшей общей подпоследовательности (НОП), то есть длину наибольшей таких последовательностей

i1 < ... < ik и j1 < ... < jk, что a[i1] = b[j1], ... , a[ik] = b[jk].

Решим эту задачу с помощью динамического программирования, где dp[i][j] будет обозначать длину НОП, если мы рассмотрели префиксы последовательностей длины

i и j.

Тогда заметим, что есть две ситуации, когда мы считаем dp[i][j]: \* ai != bj, тогда хотя бы один из этиз символов не содержится в НОП, иначе она заканчивается на два разных символа. В этом случае dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) \* a1 = bj, тогда несложно доказать, что точно есть максимальная НОП, в которую входят ОБА этих символа, а значит dp[i][j] = 1 + dp[i - 1][j - 1].

А на пустых префиксах ответ 0.

a = [1, 100, 2, 100, 3]

b = [10, 10, 1, 2, 3, 10] # НОП: 1,2,3

n = len(a)

m = len(b)

dp = [[0 for j in range(m + 1)] for i in range(n + 1)]

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

for j in range(1, m + 1):

dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

if a[i - 1] == b[j - 1]:

dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + 1)

print(dp[i])

[0, 0, 0, 1, 1, 1, 1]

[0, 0, 0, 1, 1, 1, 1]

[0, 0, 0, 1, 2, 2, 2]

[0, 0, 0, 1, 2, 2, 2]

[0, 0, 0, 1, 2, 3, 3]

Ответом является максимальное число в массиве dp. Решение работает за O(nm).

Ответ при это восстанавливается классическим способом - с конца. Нам все еще нужно просто в каждой ячейке смотреть - если символы в ней равны, то нужно уменьшить i и j, иначе только один из них - так, чтобы НОП был максимален.

**Тема 11. Бинарный поиск. Инвариант.**

**Бинарный поиск – это алгоритм поиска определенного элемента в списке. Предположим, у нас есть список из тысяч элементов, и нужно получить индексную позицию определенного элемента. Мы можем очень быстро найти позицию индекса элемента, используя алгоритм двоичного поиска.**

**Существует множество поисковых алгоритмов, но наиболее популярным среди них является бинарный поиск.**

**Элементы в списке должны быть отсортированы для применения алгоритма двоичного поиска. Если элементы не отсортированы, сначала отсортируйте их. Давайте разберемся с концепцией бинарного поиска.**

**Концепция двоичного поиска**

**В алгоритме двоичного поиска мы можем найти позицию элемента, используя следующие методы:**

* **рекурсивный метод;**
* **итерационный метод.**

**Принцип «разделяй и властвуй» лежит в основе рекурсивного метода. В нем функция вызывается снова и снова, пока не найдет элемент в списке.**

**Набор операторов повторяется несколько раз, чтобы найти позицию индекса элемента в итеративном методе. Цикл while используется для выполнения этой задачи.**

**Двоичный поиск более эффективен, чем линейный поиск, потому что нам не нужно искать каждый индекс списка. Список должен быть отсортирован для выполнения алгоритма двоичного поиска.**

**Рассмотрим пошаговую реализацию бинарного поиска.**

**У нас есть отсортированный список элементов, и мы ищем позицию индекса 45.**

**[12, 24, 32, 39, 45, 50, 54]**

**Итак, мы устанавливаем два указателя в нашем списке. Один указатель используется для обозначения меньшего значения, называемого низким, а второй указатель используется для обозначения самого высокого значения, называемого высоким.**

Далее мы вычисляем значение среднего элемента в массиве.

mid = (low+high)/2

Here, the low is 0 and the high is 7.

mid = (0+7)/2

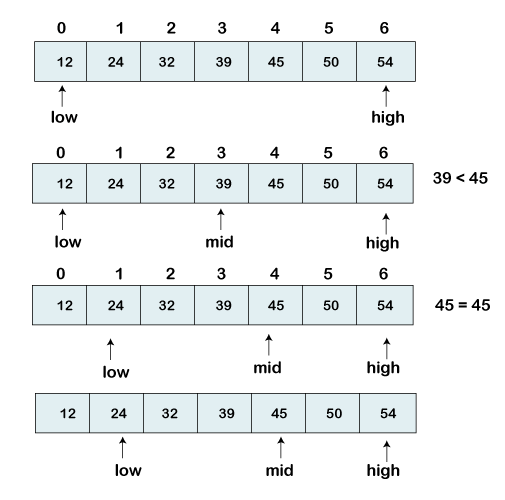
mid = 3 (Integer)

Теперь мы сравним искомый элемент со средним значением индекса. В этом случае 32 не равно 45. Поэтому нам нужно провести дальнейшее сравнение, чтобы найти элемент.

Если число, которое мы ищем, равно mid. Затем верните mid, иначе переходите к дальнейшему сравнению.

Число для поиска больше среднего числа, мы сравниваем n со средним элементом элементов справа от mid и устанавливаем low на low = mid + 1.

В противном случае сравните n со средним значением элементов слева от mid и установите high в high = mid – 1.



Повторяйте, пока не будет найден номер, который мы ищем.

**Итерационный бинарный поиск в Python**

Сначала мы реализуем двоичный поиск с итерационным методом. Мы будем повторять набор операторов и перебирать каждый элемент списка. Мы найдем среднее значение, пока поиск не завершится.

Разберемся в следующей программе итерационного метода.

Пример:

def binary\_search(list1, n):

low = 0

high = len(list1) - 1

mid = 0

while low n:

high = mid - 1

else:

return mid

return -1

list1 = [12, 24, 32, 39, 45, 50, 54]

n = 45

result = binary\_search(list1, n)

if result != -1:

print("Element is present at index", str(result))

else:

print("Element is not present in list1")

Выход:

Element is present at index 4

Объяснение:

В приведенной выше программе:

1. Мы создали функцию под названием binary\_search(), которая принимает два аргумента – список для сортировки и число для поиска.
2. Мы объявили две переменные для хранения наименьшего и наибольшего значений в списке. Начальное значение low присваивается 0, high – len (list1) – 1, а mid – 0.
3. Затем мы объявили цикл while с условием, что наименьшее значение равно и меньше наибольшего. Цикл while будет повторяться, если число еще не найдено.
4. В цикле while мы находим среднее значение и сравниваем значение индекса с искомым числом.
5. Если значение среднего индекса меньше n, мы увеличиваем среднее значение на 1 и присваиваем ему значение. Поиск перемещается влево.
6. В противном случае уменьшите среднее значение и назначьте его максимальному. Поиск переместится в правую сторону.
7. Если n равно среднему значению, верните mid.
8. Это будет происходить до тех пор, пока минимум не станет равным и меньше максимума.
9. Если мы дойдем до конца функции, значит, элемента нет в списке. Возвращаем -1 вызывающей функции.

**Рассмотрим рекурсивный метод двоичного поиска.**

**В бинарном поиске можно использовать метод рекурсии. В этом случае мы определим рекурсивную функцию, которая будет вызывать сама себя до тех пор, пока не выполнит условие.**

**Пример:**

**def binary\_search(list1, low, high, n):**

**if low n:**

**return binary\_search(list1, low, mid - 1, n)**

**else:**

**return binary\_search(list1, mid + 1, high, n)**

**else:**

**return -1**

**list1 = [12, 24, 32, 39, 45, 50, 54]**

**n = 32**

**res = binary\_search(list1, 0, len(list1)-1, n)**

**if res != -1:**

**print("Element is present at index", str(res))**

**else:**

**print("Element is not present in list1")**

**Выход:**

**Element is present at index 2**

**Объяснение:**

**Вышеупомянутая программа аналогична предыдущей. Мы объявили рекурсивную функцию и ее базовое условие. Условие: наименьшее значение меньше или равно наибольшему значению.**

1. **Среднее число вычисляем как в прошлой программе.**
2. **Мы используем оператор if, чтобы продолжить двоичный поиск.**
3. **Если среднее значение равно числу, которое мы ищем, возвращается среднее значение.**
4. **Если среднее значение меньше значения, мы снова ищем нашу рекурсивную функцию binary\_search(), увеличиваем среднее значение на единицу и присваиваем низкое значение.**
5. **Если среднее значение больше, чем значение, которое мы ищем, тогда наша рекурсивная функция binary\_search() снова уменьшит среднее значение на единицу и присвоит ему низкое.**

**В последней части мы написали нашу основную программу. Это то же самое, что и предыдущая программа, но с той лишь разницей, что мы передали два параметра в функцию binary\_search().**

**Это потому, что мы не можем присвоить начальные значения low, high и mid в рекурсивной функции. Каждый раз, когда вызывается рекурсивный метод, значение этих переменных сбрасывается. Это даст неправильный результат.**

**Сложность**

**Сложность алгоритма двоичного поиска в лучшем случае составляет O (1). Это произойдет, если элемент, который мы ищем, найден при первом сравнении. O (logn) – это наихудшая и средняя сложность двоичного поиска, зависит от количества выполняемых поисков, чтобы найти элемент, который мы ищем.**

**Инварианты**

Инвариант в программировании — логическое выражение, которое определяет непротиворечивость состояния — набора данных.

Разберёмся на примере. Когда мы описали конструктор и селекторы для рациональных чисел, то неявно подразумевали выполнение следующих инвариантов:

num = make\_rational(numer, denom)

numer == get\_numer(num)

# True

denom == get\_denom(num)

# True

Когда мы передаем в конструктор рационального числа числитель и знаменатель, то ожидаем, что получим те же числа, если применим селекторы к этому рациональному числу. Так определяется корректность работы данной абстракции. Этот код практически является тестами.

Инварианты существуют относительно любой операции. И иногда они довольно хитрые. Например, рациональные числа можно сравнивать между собой, но не прямым способом. Одни и те же дроби можно представлять разными способами: 1/2 и 2/4. Код, который не учитывает этого факта, работает некорректно:

num1 = make\_rational(2, 4)

num2 = make\_rational(8, 16)

num1 == num2

# False

**Тема 12. Длинная арифметика. Сложение, вычитание, умножение на короткое/длинное число.**

**Длинная арифметика**

Длинная арифметика — это набор программных средств (структуры данных и алгоритмы), которые позволяют работать с числами гораздо больших величин, чем это позволяют стандартные типы данных.

**Виды целочисленной длинной арифметики**

Вообще говоря, даже только в олимпиадных задачах набор средств достаточно велик, поэтому произведём классификацию различных видов длинной арифметики.

**Классическая длинная арифметика**

Основная идея заключается в том, что число хранится в виде массива его цифр.

Цифры могут использоваться из той или иной системы счисления, обычно применяются десятичная система счисления и её степени (десять тысяч, миллиард), либо двоичная система счисления.

Операции над числами в этом виде длинной арифметики производятся с помощью "школьных" алгоритмов сложения, вычитания, умножения, деления столбиком. Впрочем, к ним также применимы алгоритмы быстрого умножения: Быстрое преобразование Фурье и Алгоритм Карацубы.

Здесь описана работа только с неотрицательными длинными числами. Для поддержки отрицательных чисел необходимо ввести и поддерживать дополнительный флаг "отрицательности" числа, либо же работать в дополняющих кодах.

**Структура данных**

Хранить длинные числа будем в виде вектора чисел int, где каждый элемент — это одна цифра числа.

typedef vector<int> lnum;

Для повышения эффективности будем работать в системе по основанию миллиард, т.е. каждый элемент вектора lnum содержит не одну, а сразу 9 цифр:

const int base = 1000\*1000\*1000;

Цифры будут храниться в векторе в таком порядке, что сначала идут наименее значимые цифры (т.е. единицы, десятки, сотни, и т.д.).

Кроме того, все операции будут реализованы таким образом, что после выполнения любой из них лидирующие нули (т.е. лишние нули в начале числа) отсутствуют (разумеется, в предположении, что перед каждой операцией лидирующие нули также отсутствуют). Следует отметить, что в представленной реализации для числа ноль корректно поддерживаются сразу два представления: пустой вектор цифр, и вектор цифр, содержащий единственный элемент — ноль.

**Вывод**

Самое простое — это вывод длинного числа.

Сначала мы просто выводим самый последний элемент вектора (или , если вектор пустой), а затем выводим все оставшиеся элементы вектора, дополняя их нулями до 9 символов:

printf ("%d", a.empty() ? 0 : a.back());

for (int i=(int)a.size()-2; i>=0; --i)

printf ("%09d", a[i]);

(здесь небольшой тонкий момент: нужно не забыть записать приведение типа (int), поскольку в противном случае число a.size() будет беззнаковым, и если a.size() <= 1, то при вычитании произойдёт переполнение)

**Чтение**

Считываем строку в string, и затем преобразовываем её в вектор:

for (int i=(int)s.length(); i>0; i-=9)

if (i < 9)

a.push\_back (atoi (s.substr (0, i).c\_str()));

else

a.push\_back (atoi (s.substr (i-9, 9).c\_str()));

Если использовать вместо string массив char’ов, то код получится ещё компактнее:

for (int i=(int)strlen(s); i>0; i-=9) {

s[i] = 0;

a.push\_back (atoi (i>=9 ? s+i-9 : s));

}

Если во входном числе уже могут быть лидирующие нули, то их после чтения можно удалить таким образом:

while (a.size() > 1 && a.back() == 0)

a.pop\_back();

**Сложение**

Прибавляет к числу a число b и сохраняет результат в a:

int carry = 0;

for (size\_t i=0; i<max(a.size(),b.size()) || carry; ++i) {

if (i == a.size())

a.push\_back (0);

a[i] += carry + (i < b.size() ? b[i] : 0);

carry = a[i] >= base;

if (carry) a[i] -= base;

}

**Вычитание**

Отнимает от числа a число b (a >= b) и сохраняет результат в a:

int carry = 0;

for (size\_t i=0; i<b.size() || carry; ++i) {

a[i] -= carry + (i < b.size() ? b[i] : 0);

carry = a[i] < 0;

if (carry) a[i] += base;

}

while (a.size() > 1 && a.back() == 0)

a.pop\_back();

Здесь мы после выполнения вычитания удаляем лидирующие нули, чтобы поддерживать предикат о том, что таковые отсутствуют.

**Умножение длинного на короткое**

Умножает длинное a на короткое b (b < base) и сохраняет результат в a:

int carry = 0;

for (size\_t i=0; i<a.size() || carry; ++i) {

if (i == a.size())

a.push\_back (0);

long long cur = carry + a[i] \* 1ll \* b;

a[i] = int (cur % base);

carry = int (cur / base);

}

while (a.size() > 1 && a.back() == 0)

a.pop\_back();

Здесь мы после выполнения деления удаляем лидирующие нули, чтобы поддерживать предикат о том, что таковые отсутствуют.

(Примечание: способ **дополнительной оптимизации**. Если скорость работы чрезвычайно важна, то можно попробовать заменить два деления одним: посчитать только целую часть от деления (в коде это переменная carry), а затем уже посчитать по ней остаток от деления (с помощью одной операции умножения). Как правило, этот приём позволяет ускорить код, хотя и не очень значительно.)

**Умножение двух длинных чисел**

Умножает a на b и результат сохраняет в c:

lnum c (a.size()+b.size());

for (size\_t i=0; i<a.size(); ++i)

for (int j=0, carry=0; j<(int)b.size() || carry; ++j) {

long long cur = c[i+j] + a[i] \* 1ll \* (j < (int)b.size() ? b[j] : 0) + carry;

c[i+j] = int (cur % base);

carry = int (cur / base);

}

while (c.size() > 1 && c.back() == 0)

c.pop\_back();

**Деление длинного на короткое**

Делит длинное a на короткое b (b < base), частное сохраняет в a, остаток в carry:

int carry = 0;

for (int i=(int)a.size()-1; i>=0; --i) {

long long cur = a[i] + carry \* 1ll \* base;

a[i] = int (cur / b);

carry = int (cur % b);

}

while (a.size() > 1 && a.back() == 0)

a.pop\_back();

**Длинная арифметика в факторизованном виде**

Здесь идея заключается в том, чтобы хранить не само число, а его факторизацию, т.е. степени каждого входящего в него простого.

Этот метод также весьма прост для реализации, и в нём очень легко производить операции умножения и деления, однако невозможно произвести сложение или вычитание. С другой стороны, этот метод значительно экономит память в сравнении с "классическим" подходом, и позволяет производить умножение и деление значительно (асимптотически) быстрее.

Этот метод часто применяется, когда необходимо производить деление по непростому модулю: тогда достаточно хранить число в виде степеней по простым делителям этого модуля, и ещё одного числа — остатка по этому же модулю.

**Длинная арифметика по системе простых модулей (Китайская теорема или схема Гарнера)**

Суть в том, что выбирается некоторая система модулей (обычно небольших, помещающихся в стандартные типы данных), и число хранится в виде вектора из остатков от его деления на каждый из этих модулей.

Как утверждает Китайская теорема об остатках, этого достаточно, чтобы однозначно хранить любое число в диапазоне от 0 до произведения этих модулей минус один. При этом имеется Алгоритм Гарнера, который позволяет произвести это восстановление из модульного вида в обычную, "классическую", форму числа.

Таким образом, этот метод позволяет экономить память по сравнению с "классической" длинной арифметикой (хотя в некоторых случаях не столь радикально, как метод факторизации). Крому того, в модульном виде можно очень быстро производить сложения, вычитания и умножения, — все за асимптотически однаковое время, пропорциональное количеству модулей системы.

Однако всё это даётся ценой весьма трудоёмкого перевода числа из этого модульного вида в обычный вид, для чего, помимо немалых временных затрат, потребуется также реализация "классической" длинной арифметики с умножением.

Помимо этого, производить **деление** чисел в таком представлении по системе простых модулей не представляется возможным.

**Виды дробной длинной арифметики**

Операции над дробными числами встречаются в олимпиадных задачах гораздо реже, и работать с огромными дробными числами значительно сложнее, поэтому в олимпиадах встречается только специфическое подмножество дробной длинной арифметики.

**Длинная арифметика в несократимых дробях**

Число представляется в виде несократимой дроби , где и — целые числа. Тогда все операции над дробными числами нетрудно свести к операциям над числителями и знаменателями этих дробей.

Обычно при этом для хранения числителя и знаменателя приходится также использовать длинную арифметику, но, впрочем, самый простой её вид — "классическая" длинная арифметика, хотя иногда оказывается достаточно встроенного 64-битного числового типа.

**Выделение позиции плавающей точки в отдельный тип**

Иногда в задаче требуется производить расчёты с очень большими либо очень маленькими числами, но при этом не допускать их переполнения. Встроенный -байтовый тип , как известно, допускает значения экспоненты в диапазоне , чего иногда может оказаться недостаточно.

Приём, собственно, очень простой — вводится ещё одна целочисленная переменная, отвечающая за экспоненту, а после выполнения каждой операции дробное число "нормализуется", т.е. возвращается в отрезок , путём увеличения или уменьшения экспоненты.

При перемножении или делении двух таких чисел надо соответственно сложить либо вычесть их экспоненты. При сложении или вычитании перед выполнением этой операции числа следует привести к одной экспоненте, для чего одно из них домножается на в степени разности экспонент.

Наконец, понятно, что не обязательно выбирать в качестве основания экспоненты. Исходя из устройства встроенных типов с плавающей точкой, самым выгодным представляется класть основание равным .