CBook

Денис Кириенко

ЛКШ-2013, параллель C.python

Содержание

1 Квадратичные сортировки

1.1 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 $A[i] \leq A[i+1]$ 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 $A[i] \geq A[i+1]$, 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 с 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, то есть количество выполняемых им действий. 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 нужна $n-1$ операция. Потом нужно $n-2$ операции и т. д. Общее число операций равно $n + (n-1) + (n-2) + ... + 1 = \frac{n(n+1)}{2} = O(n^2)$. Таким образом, сортировка выбором — квадратичный алгоритм, время его работы пропорционально квадрату от размера списка.

1.2 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 срез $A[:i+1] = [1, 4, 5, 6, 8, 8]$. 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. $O(n)$. 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 $1+2+\ldots+(n-1)+n = \frac{n(n+1)}{2} = O(n^2)$.

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 $\frac{n(n+1)}{4} = O(n^2)$. То есть в среднем этот алгоритм также имеет квадратичную сложность.

1.3 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 $O(n^2)$. Что произойдет, если запустить сортировки пузырьком на уже остортированном списке? Ни одной перестановки не будет произведено, но алгоритм все равно выполнит $O(n^2)$ 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.

1.4 Сортировка подсчетом

Еще одним алгоритмом, который, казалось бы, решает аналогичную задачу, является алгоритм сортировки подсчетом. Правда, у него ограниченная область применения, его нельзя использовать, например, для строковых данный или чисел, типа float. Он применим только для сортировки списков, элементы которого — небольшие целые числа.

Пусть список состоит из целых неотрицательных чисел, меньших K. Давайте подсчитаем, сколько раз каждое из чисел $0, 1, 2, \ldots, K-1$ встречается в списке A. Для этого заведем список Count из K элементов (с индексами от 0 до K - 1), где значение Count[i] равно количеству появления числа i в списке A. Для этого пройдемся переменной elem по всему списку и увеличим на 1 значение A[elem]. Затем создадим отсортированный список. В этот список должно войти число 0 столько раз, чему равно Count[0], число 1 должно войти Count[1] раз и т. д. Для создания такого списка используем

вложенный цикл. Значение K можно выбрать, как наибольшее значение в списке, увеличенное на 1.

```
def CountSort(A):
    K = max(A) + 1
    Count = [0] * K
    for elem in A:
        Count[elem] += 1
    Result = []
    for val in range(K):
        Result += [val] * Count[val]
    return Result
```

Заметим, что этот алгоритм не упорядочивает элементы списка, он создает новый список, который состоит из тех же значений, что и значения исходного списка. Именно поэтому алгоритм сортировки подсчетом нельзя считать алгоримом сортировки — он не упорядочивает список, а создает новый список. Легко видеть, что сложность этого алгоритма: $O(n+K)$. То есть при маленьких значениях K этот алгоритм имеет линейную сложность. Наоборот, если K существенно больше чем n, то сортировка подсчетом бессмысленна, например, не следует сортировать подсчетом элементы списка длины 100, если они могут принимать значения до 10^6 — любой квадратичный алгоритм быстрее справится с этой задачей.

2 Быстрые сортировки

2.1 Сортировка слиянием

Алгоритм сортировки слиянием позволяет отсортировать список Он основан на идее, что два отсортированных списка можно слить в один отсортированный список за время, равное суммарной длине этих списков.

Для этого сравним первые элементы данных списков. Тот элемент, который меньше, скопируем в конец результирующего списка (который первоначально пуст) и в этом списке перейдем к следующему элементу. Будем повторять этот процесс (выбираем из начала двух списков наименьший элемент, копируем его в результирующий список), пока один из исходных списков не кончится. После этого оставшиеся элементы (один из двух исходных списков будет непуст) скопируем в результирующий список.

Для того, чтобы не удалять начальные элементы из списков, заведем два индекса i и j, указывающие на текущие элементы в каждом списке. Вместо удаления элементов будем передвигать эти индексы. В конце добавим к результирующему списку оставшиеся элементы из двух исходных списков A[i:] + B[j:] (один из этих срезов будет пустым, это не должно нас смущать).

```
def merge(A, B):
    Res = []
    i = 0
    j = 0
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            Res.append(A[i])
```

```
            i += 1
        else:
            Res.append(B[j])
            j += 1
    Res += A[i:] + B[j:]
    return Res
```

Заметим, что сложность работы функции merge — линейная от суммарных длин списков A и B, так как каждый элемент обрабатывается ровно один раз за O(1).

Теперь можно реализовать сортировку слиянием. Это — рекурсивная функция, которая получает на вход исходный список и список, составленный из тех же элементов, но отсортированный. Если длина исходного списка равна 1 или 0, то он уже отсортирован и сортировать его не надо. Если же длина списка больше 1, то разобьем его на две части равной (или почти равной, если длина исходного списка нечетная). Отсортируем обе эти части, затем сольем их вместе при помощи функции merge.

```
def MergeSort(A):
    if len(A) <= 1:
        return A
    else:
        L = A[:len(A) // 2]
        R = A[len(A) // 2:]
        return merge(MergeSort(L), MergeSort(R))
```

Оценим сложность этого алгоритма. Пусть массив содержит n элементов. Тогда за $O(n)$ его можно разделить на две части и после сортировки слить их вместе. Каждая из этих двух частей имеет размер $n/2$, и за $O(n)$ шагов каждую из них можно поделить на две части размером $n/4$ и затем после сортировки слить их вместе. Аналогично, четыре части размером $n/4$ за суммарное $O(n)$ шагов делятся на части размером $n/8$ и сливаются вместе. Этот процесс "в глубину" продолжается столько раз, сколько раз можно число n делить на 2, до тех пор, пока размер части не станет равен 1, то есть $\log_2 n$. Итого, общая сложность этого алгоритма равна $O(n \log_2 n)$.

Одним из недостатков сортировки слиянием является тот факт, что он требует много вспомогательной памяти (столько же, каков размер исходного массива) для реализации.

3 Структуры данных

3.1 Стек

Стеком (англ. *stack*) называется структура данных, в которой элементы хранятся в виде последовательности, при этом работать можно только с одним элементом – который был последним добавлен в стек. Стек можно представить себе в виде стакана, в который можно класть данные, причем достать можно только последний элемент, который был положен в стакан.

Стек также называют структурой типа LIFO (last in, first out — последним пришел, первым ушел). Стек должен поддерживать следующие операции:

push — добавление элемента в конец стека.

pop — удаление элемента из стека. Удаляется последний элемент, который был добавлен в стек. Как правило, операция **pop** возвращает значение удаленного элемента.

top — узнать значение последнего элемента в стеке (без удаления его из стека).

size — узнать количество элементов в стеке.

При этом операция **top** может отсутствовать (ее можно заменить на пару операций **pop** и **push**), а вместо операции **size** может быть более простая операция **isempty** проверки стека на пустоту. Возможна и операция **clear**, которая очищает стек.

Для реализации стека в языке Python можно использовать обычный список, поскольку он поддерживает операции добавления элемента в конец списка при помощи метода **append** и удаления элемента из конца списка при помощи метода **append**. Пустой стек соответствует пустому списку. Реализуем операции над стеком в виде функций, получающих в качестве параметра список **Stack**, в котором хранятся элементы стека.

```
def push(Stack, data):
    Stack.append(data)

def pop(Stack):
    return Stack.pop()

def top(Stack):
    return Stack[-1]

def size(Stack):
    return len(Stack)

def isempty(Stack):
    return len(Stack) == 0

def clear(Stack):
    Stack[:] = []
```

Использовать эти операции можно, например, следующим образом:

```
Stack = []
push(Stack, 1)
push(Stack, 2)
push(Stack, 3)
while size(Stack) > 0:
    print(pop(Stack))
```

В языках программирования, в которых нет списков (массивов) динамически изменяющегося размера, можно реализовать стек на базе массива, при этом отдельно нужно хранить размер стека, например, в переменной **stacksize**. Тогда при добавлении элемента в стек значение **stacksize** увеличивается на 1, а при удалении элемента из стека — уменьшается на 1. Соответствующий код может выглядеть, например, следующим образом:

```
def push(data):
    stacksize = stacksize + 1
    Stack[stacksize] = data

def pop():
    data = Stack[stacksize]
    stacksize = stacksize - 1
    return data
```

Но в языке Python списки могут динамически изменять свой размер, при этом добавление и удаление элементов в конец списка занимает небольшое время, в среднем не зависящее от размера списка. В этом случае говорят, что операции добавления и удаления элементов в стек имеют сложность $O(1)$.

Запишем и другую форму реализации стека с использованием объектно-ориентированного подхода. В этом случае мы определим класс **Stack**, у которого будут методы **push**, **pop**, **top** и т.д. Для хранения элементов стека будет использоваться список **__elems**, являющийся закрытым членом класса, то есть обратиться к таким членам класса можно только через методы стека **push**, **pop** и т.д. Такая технология называется *инкапсуляцией*.

```
class Stack:
    def __init__(self):
        self.__elems = []

    def push(self, data):
        self.__elems.append(data)

    def pop(self):
        return self.__elems.pop()

    def top(self):
        return self.__elems[-1]

    def size(self):
        return len(self.__elems)

    def isempty(self):
        return len(self.__elems) == 0

    def clear(self):
        self.__elems[:] = []
```

В этом случае использование стека будет выглядеть по-другому:

```
S = Stack()
S.push(1)
S.push(2)
S.push(3)
while S.size() > 0:
    print(S.pop())
```

Стеки широко используются в компьютерных программах. Во-первых, при помощи стека организован процесс вызова функций и рекурсии. Во-вторых, многие алгоритмы используют стеки, например, алгоритм проверки правильности скобочной последовательности, алгоритм обхода графа в глубину, алгоритм Грэхема построения выпуклой оболочки.

3.2 Очередь

Очередью (англ. *queue*) называется структура данных, из которой удаляется первым тот элемент, который был первым в очередь добавлен. То есть очередь в программировании соответствует "бытовому" понятию очереди, когда первым обслуживается тот человек, который первым встал в очередь. Очередь также называют структурой типа FIFO (first in, first out — первым пришел, первым ушел). Очередь поддерживает те же операции, что и стек, за исключением того, что операции **pop** и **top** работают с другим концом очереди.

Казалось бы, очередь можно просто реализовать, если удалять из списка не последний элемент, а первый, если передать методу **pop** списка индекс удаляемого элемента 0. Приведем начало такой реализации:

```
class Queue:
    def __init__(self):
        self.__elems = []

    def push(self, data):
        self.__elems.append(data)

    def pop(self):
        return self.__elems.pop(0)

    def top(self):
        return self.__elems[0]
```

Остальные методы очереди аналогичны соответствующим методам списка, поэтому приводить их здесь не будем.

Но проблема такой реализации будет заключаться в том, что удаление элемента из начала списка требует сдвига всех элементов списка к его началу, поэтому производится за время, пропорциональное длине списка (то есть имеет сложность $O(n)$, где n — количество элементов списка). Можно поступить следующим образом — при выполнении операции **pop** не удалять элемент из списка, а просто считать, что начало очереди сдвигается влево, то есть первым элементом очереди становится следующий элемент списка. Для этого будем в отдельной переменной (поле класса) **__queuestart** хранить номер элемента, который будет первым элементом в очереди. Тогда удаление элемента из очереди сводится к увеличению этой переменной на 1, а в начале списка будут идти элементы, которые когда-то находились в очереди, а потом были удалены из нее.

```
class Queue:
    def __init__(self):
        self.__elems = []
```

```
        self.__queuestart = 0

    def push(self, data):
        self.__elems.append(data)

    def top(self):
        return self.__elems[self.__queuestart]

    def pop(self):
        self.__queuestart += 1
        return self.__elems[self.__queuestart - 1]
```

В этом случае реализация метода **size** будет другой — длина очереди это размер списка минус все удаленные элементы:

```
class Queue:
    def size(self):
        return len(self.__elems) - self.__queuestart
```

У такой реализации есть другая проблема — начало очереди засоряется "ненужными" элементами, которые не используются. Если из очереди было удалено много элементов, то большая часть очереди будет состоять из этих ненужных элементов, что приводит к расходу памяти.

Есть несколько способов решения этой проблемы. Можно при наличии большого числа неиспользуемых элементов в начале списка новые элементы, добавляемые в конец очереди, можно размещать в начале списка на месте удаленных из очереди элементов. Получается список, "замкнутый" в кольцо, в котором за последним элементом списка идет первый элемент. При заполнении последних элементов списка очередь начинает "расти" от его начала. В такой реализации необходимо хранить номер начального элемента в очереди и номер последнего элемента в очереди (или количество элементов в очереди).

Другой способ заключается в том, чтобы время от времени реально удалять неиспользуемые элементы из начала очереди. То есть если начальных неиспользуемых элементов в очереди немного, они остаются на месте. А если накопилось много неиспользованных элементов, то можно потратить большое время и "очистить" очередь, удалив неиспользуемые элементы из начала списка.

Такая технология называется "ленивое удаление". Суть технологии в том, что ненужный элемент не удаляется реально из структуры данных, а просто помечается, как неиспользуемый, что осуществляется за $O(1)$. Если накопилось много таких неиспользуемых элементов, то проводится операция реального удаления элементов из структуры данных — такая операция требует много времени, но и проводиться будет редко.

Критерим для реального удаления неиспользуемых элементов будет условие, что неиспользуемые элементы занимают более половины от списка. Получаем следующую реализацию операции **pop**:

```
class Queue:
    def pop(self):
        result = self.__elems[self.__queuestart]
        self.__queuestart += 1
```

```
        if self.__queuestart > len(self.__elems) / 2:
            self.__elems[:self.__queuestart] = []
            self.__queuestart = 0
        return result
```

Сложность такой реализации будет следующей. В лучшем случае операция удаления будет выполняться за $O(1)$, в худшем — за $O(n)$, но если с очередью выполнить n операций, то суммарное время выполнения всех операций удаления будет $O(n)$. В этом случае говорят, что среднее (или амортизационное) время работы операции удаления будет $O(1)$.

3.3 Дек

Деком (англ. *deque* от "double-ended queue" — очередь с двумя концами) называется структура данных, в которой все операции добавления и удаления элементов можно проводить с каждым из двух концов дека. То есть в деке есть две операции добавления в начало и в конец дека и также две операции удаления из начала и конца дека.

Реализовать дек можно на базе списка с добавлением элементов в начало или конец списка, но в этом случае операции с началом дека будут иметь сложность $O(n)$. Можно реализовать дек на базе закольцованного списка, если хранить две переменные — индекс первого и последнего элемента в деке. Но в модуле **collections** языка Python есть класс **deque**, содержащий реализацию стандартного дека, которую можно использовать и в качестве очереди. У этого класса есть следующие методы:

append(x) — добавить элемент x в конец дека (или справа).

appendleft(x) — добавить элемент x в начало дека (или слева).

pop() — удалить последний (самый правый) элемент дека. Метод возвращает значение удалямого элемента.

popleft() — удалить первый (самый левый) элемент дека. Метод возвращает значение удалямого элемента.

clear() — очистить дек.

rotate(n) — циклически сдвинуть элементы дека на n элементов вправо, эквивалентно n-кратному удалению элемента справа и добавлением его слева. Если $n < 0$, то осуществляется сдвиг влево на $|n|$ элементов.

Также можно узнать количество элементов в классе **deque** при помощи функции **len**, а также можно обращаться к элементам дека по индексу, то есть если **D** является объектом класса **deque**, то **D[0]** является самым первым (левым) элементом дека, **D[1]** — следующим за ним, и т.д. **D[-1]** является последним (правым) элементом дека, **D[-2]** — предпоследним и т.д. Пример использования дека:

```
import collections
D = collections.deque()
D.appendleft(1)
D.appendleft(2)
D.appendright(3)
# Сейчас дек имеет вид  [2, 1, 3]
# Вывод дека слева направо
while len(D) > 0:
    print(D[0])
```

```
D.popleft()
```

При этом ввиду особенностей реализации операции добавления элементов в начало и конец дека, обращения к первым и последним элементам дека выполняются за $O(1)$, а операции доступа к элементам, остоящим от концов дека на расстояние n выполняются за $O(n)$.

4 Поиск элемента в списке

4.1 Линейный поиск

Рассмотрим задачу линейного поиска элемента в массиве. Необходимо реализовать функцию, которая проверяет, содержится ли в данном списке **A** данный элемент **key**. Функция должна возвращать значение **True** или **False**. Другие модицификации алгоритма линейного поиска могут возвращать значение индекса элемента, равного **key**.

Это можно сделать, например, перебрав все элементы списка при помощи цикла for с проверкой условия равенства текущего элемента значению **key** внутри цикла:

```
def search(A, key):
    for i in range(len(A)):
        if A[i] == key:
            return True
    return False
```

В языке Python удобней использовать цикл по значению элемента (for elem in A), а не по индексу элемента (for i in range(len(A)). В этом случае цикл будет выглядеть проще:

```
def search(A, key):
    for elem in A:
        if elem == key:
            return True
    return False
```

Другая возможная реализация линейного поиска использует цикл while без дополнительного условия внутри цикла. Будем по очереди просматривать все элементы цикла до тех пор, пока не найдем элемент, равный **key** или не выйдем за границы списка. В этом случае переменная i указывает на просматриваемый элемент списка, в начале алгоритма i=0. В цикле значение i увеличивается, пока значение i не выйдет за границу списка, или если будет найден элемент, равный **key**. В результате после окончания цикла либо A[i] == key, либо i вышло за границу списка, в этом случае i==len(A). Для проверки того, был ли найден элемент, равный **key**, необходимо проверить условие $i < len(A)$.

```
def search(A, key):
    i = 0
    while i < len(A) and A[i] != key:
        i += 1
    return i < len(A)
```

Алгоритм линейного поиска можно модицировать так, чтобы он возвращал индекс первого элемента, равного **key**, а если такой элемент не найден, то возвращается специальное значение -1. Вариант алгоритма линейного поиска индекса элемента с использованием цикла for:

```
def search(A, key):
    for i in range(len(A)):
        if A[i] == key:
            return i
    return -1
```

Реализация алгоритма линейного поиска элемента с использованием цикла while:

```
def search(A, key):
    i = 0
    while i < len(A) and A[i] != key:
        i += 1
    if i < len(A):
        return i
    else:
        return -1
```

Поскольку все эти алгоритмы должны в худшем случае (ели искомый элемент не содержится в списке) просмотреть весь список в поисках данного элемента, то сложность данных алгоритмов будет $O(n)$ (n — длина списка).

Если гарантируется, что элемент **key** есть в списке и необходимо найти индекс его первого вхождения, то можно убрать проверку на выход за границу списка:

```
def search(A, key):
    i = 0
    while A[i] != key:
        i += 1
    return i
```

Но если в списке элемент, равный **key** может отстутствовать, то можно добавить значение **key** в конец списка, тогда линейный поиск заведомо остановится на этом новом элементе (если элемента равного **key** не было в исходном списке) или раньше. Тогда также можно не выполнять проверку на выход индекса за границу списка. Такой элемент называется *барьерным*, поскольку он является ограничителем для индекса элемента в поиске. После окончания поиска барьерный элемент необходимо удалить из списка. Приведем код алгоритма линейного поиска, проверяющего наличие элемента в списке, с использованием барьерного элемента.

```
def search(A, key):
    A.append(ley)
    i = 0
    while A[i] != key:
        i += 1
    A.pop()
    return i < len(A)
```

4.2 Двоичный поиск

Если исходный список уже отсортирован, то элемент в нем можно найти гораздо быстрее, если воспользоваться двоичным поиском. Идея двоичного поиска заключается в делении списка пополам, после чего в зависимости от значения среднего элемента в списке мы переходим либо к левой, либо к правой половине списка. Тем самым длина части, в которой мы ищем элемент, сокращается в два раза на каждом шаге цикла, а, значит, общая сложность алгоритма двоичного поиска будет $O(log_2 n)$ (количество шагов, необходимых для сокращения списка из n элементов до одного элемента делением пополам).

Итак, перед нами стоит задача — выяснить, содержится ли элемент **key** в некотором списке или в его части. Мы будем сокращать часть списка, в которой мы ищем элемент **key**. А именно, введем две границы — **left** и **right**. При этом мы будем знать, что элемент **A[right]** строго больше, чем **key**, то же самое можно сказать и про элементы, которые правее элемента с индексом **right**. Про элемент **A[left]** и те элементы, которые левее него мы будем знать, что все они меньше или равны key. А про элементы, которые лежат строго между **A[left]** и **A[right]** (то есть про элементы, чьи индексы больше **left**, но меньше **right**), мы ничего не знаем. То есть между элементами с индексами **left** и **right** находятся те элементы списка, про которые в настоящий момент ничего неизвестно. Алгоритм двоичного поиска будет сокращать эту область до тех пор, пока все элементы не будут отнесены либо к левой, либо к правой части, а элементов, про которые ничего не известно, не останется.

В самом начале мы ничего не знаем про все элементы массива, поэтому указатели **left** и **right** должны указывать на фиктивные элементы: **left** указывает на элемент, следующий перед элементом с индексом 0, а **right** на элемент, следущий за элементом с индексом **right**. Поэтому присвоим left = -1 и right = len(A). Можно представить это так — к концам массива добавляются два фиктивных элемента, в левый конец добавляется элемент, в который записывается минус бесконечность (т. е. значение, заведомо меньшее, чем key), и этот элемент имеет индекс -1, а в правый элемент дописывается элемент, равный плюс бесконечности, и его индекс равен len(A). Соответственно, переменные left и right первоначально указывают на эти фиктивные элементы.

Затем разделим отрезок от left до right на две части и возьмем средний элемент между ними. Его индекс равен middle = (left + right) // 2. Сравним значение этого элемента со значением key. Если A[middle] строго больше чем key это означает, что сам элемент A[middle] и все, что правее него, должно попасть в правую часть. Это означает, что нужно сделать присваивание right = middle. Иначе (если A[middle] <= key), то элемент A[middle] и все, что левее него, должно попасть в левую часть, то есть необходимо присвоить left = middle. Будем повторять этот процесс, пока между двумя границами left и right еще есть элементы, то есть что right > left + 1. Получаем следующий алгоритм — прообраз алгоритма двоичного поиска.

```
left = -1
right = len(A)
while right > left + 1:
```

```
    middle = (left + right) // 2
    if A[middle] > key:
        right = middle
    else:
        left = middle
```

Что будет после завершения этого алгоритма? left и right указывают на два соседних элемента, при этом A[right] > key, A[left] <= key. Таким образом, если элемент key содержится в списке, то A[left] == key, а если не содержится, то A[left] < key. Но возможна ситуация, когда left == -1 (если в списке A все элементы строго больше key, то в двоичном поиске будет двигаться только границам right и двоичный поиск сойдется к значениям left == -1 и right == 0). Поэтому для ответа на вопрос «содержится ли в списке A элемент key» необходимо проверить условие left >= 0 and A[left] == key. Итак, можно записать алгоритм двоичного поиска следующим образом:

```
def BinarySearch(A, key):
    left = -1
    right = len(A)
    while right > left + 1:
        middle = (left + right) // 2
        if A[middle] > key:
            right = middle
        else:
            left = middle
    return left >= 0 and A[left] == key
```

Но этот алгоритм позволяет решить куда более интересную задачу, чем просто проверка наличия элемента в списке. Для этого обратим внимание на значение right после окончания алгоритма? Это элемент в списке, который строго больше, чем key. Иными словами, на место элемента A[right] можно вставить элемент со значением key (сдвинув при этом всю правую часть списка, которая будет строго больше key, на один элемент), сохраняя упорядоченность списка, при этом right есть самая правая позиция, куда можно вставить в список элемент key, сохраняя упорядоченность. В этом случае говорят, что значение right является “верхней границей” для элемента key: правее этой позиции нельзя вставить элемент key, сохраняя список упорядоченным. Вот функция, которая в заданном списке A находит «верхнюю границу» для заданного элемента key:

```
def UpperBound(A, key):
    left = -1
    right = len(A)
    while right > left + 1:
        middle = (left + right) // 2
        if A[middle] > key:
            right = middle
        else:
            left = middle
    return right
```

Теперь модифицируем алгоритм двоичного поиска так, чтобы элементы,

в точности равные key, оказывались в правой части списка, а не в левой, как раньше. Это потребует изменения неравенства в условии внутри цикла:

```
def LowerBound(A, key):
    left = -1
    right = len(A)
    while right > left + 1:
        middle = (left + right) // 2
        if A[middle] >= key:
            right = middle
        else:
            left = middle
    return right
```

На каких значениях left и right закончится данный цикл? A[left] < key, A[right] >= key. Тем самым right указывает на первый элемент, равный (или больше) key, а элементы с индексами меньшими, чем right будут строго меньше, чем key. То есть элемент, равный key, можно вставить на место элемента с индексом right, сохраняя упорядоченность списка, и это минимальная позиция, куда это можно сделать. Иными словами, данная функция возвращает "нижнюю границу" для элемента key. Следует отметить, что поскольку UpperBound возвращает индекс первого элемента, который больше чем key, а LowerBound возвращает индекс первого элемента, который равен key (а если такого нет — то того, который больше, чем key), то значение разности UpperBound(A, key) - LowerBound(A, key) равно числу вхождений элемента key в список A.

При помощи LowerBound также можно проверить вхождение элемента key в список A при помощи условия right < len(A) and A[right] == key после выполнения алгоритма (аналогично, нужно проверить, что в списке под индексом right стоит элемент, равный key, но при этом не происходит выхода за границу списка, возможного в случае, когда все элементы списка будут строго меньше key, в этом случае алгоритм LowerBound остановится на значении left == len(A) - 1, right == len(A)).

4.3 Поиск корня функции делением пополам

Идею двоичного поиска можно использовать для нахождения корня непрерывной функции. Пусть дана функция $f(x)$, то есть можно вычислить значение функции в произвольной точке на отрезке $[a, b]$, при этом известно, что $f(a) < 0$, $f(b) > 0$, следовательно если функция непрерывна, то на отрезке $[a, b]$ у нее есть корень. Разделим отрезок пополам и в зависимости от знака значения функции на середине отрезка перейдем либо к левому, либо к правому отрезку, сохраняя инвариант $f(a) < 0$, $f(b) > 0$, то есть выбирая отрезок так, чтобы корень функции оставался на отрезке $[a, b]$, а длина отрезка при этом сократилась в два раза. Цикл продолжается до тех пор, пока длина отрезка не станет меньше той точности, с которой необходимо определить корень функции. Эта точность задается значением переменной EPS.

```
def root(a, b):
    while b - a > EPS :
```

```
        x = (right + left) / 2
        if f(x) > 0:
            b = x
        else :
            a = x
    return (a + b) / 2
```

4.4 Двоичный поиск по ответу

Двоичный поиск позволяет иногда решать задачи, которые, казалось бы, никакого отношения ни к поиску элемента в массиве, ни к поиску корня функции, не имеют никакого отношения. Рассмотрим идею двоичного поиска по ответу на примере следующей задачи.

Есть два принтера, первый принтер печатает одну страницу за a секунд, второй принтер — за b секунд. Необходимо напечатать n страниц, за какое наименьшее время это можно сделать?

Эту задачу можно решить двоичным поиском по ответу. Задачи, решаемые двоичным поиском по ответу, легко можно узнать по следующим особенностям условий:

1. В задаче фигурирует некоторый параметр, от которого зависит наличие ответа (в данной задаче этот параметр — время).
2. Условие монотонности — если есть решение задачи для некоторого значения параметра, то и для большего (меньшего) значения параметра решение тоже есть (в данной задаче если за некоторое время t можно напечатать нужное число страниц, то и за любое большее время это можно сделать).
3. Необходимо найти наименьшее (наибольшее) значение параметра, при котором существует решение задачи.
4. Для фиксированного значения параметра (например, при фиксированном времени) легко проверяется, является ли данное значение параметра решением.

В этой задаче все эти условия соблюдены. Чтобы проверить, можно ли за время t напечатать n страниц, нужно посчитать, сколько страниц каждый из двух принтеров напечатает за t секунд и сложить результат. Проверить, что за время t можно решить поставленную задачу можно при помощи условия t // a + t // b >= n. В задаче необходимо найти такое минимальное значение t, для которого выполнено это условие, значит, для значения $t - 1$ и всех меньших значений это условие невыполнено. Это можно сделать простым перебором всех значений t начиная с нуля, пока не будет найдено нужное значение:

```
t = 0
while t // a + t // b < n:
    t += 1
```

Но это решение будет работать долго, для ускорения решения такое минимальное значение t, для которого выполнено условие t // a + t // b >= n можно найти двоичным поиском.

Для этого заведем две переменные left и right. left будет таким временем, при котором задача не имеет решения. А right будет таким временем, при котором задача, наоборот, имеет решение. Для значений времени между left и right мы ничего (пока) не знаем.

Далее будем сдвигать границы left и right. Берем середину отрезка между ними t = (left + right) // 2. Считаем, сколько страниц можно напечатать за время t. Если это число больше или равно n, то необходимо присвоить right = t (для времени t задача разрешима, поэтому двигаем правый конец), иначе присвоим left = t (за время t задача неразрешима, двигаем левый конец). Цикл продолжается, пока между right и left есть еще нерассмотренные значения. Цикл закончится, когда right == left + 1, в этом случае right будет равно минимальному времени, за которое можно напечатать n страниц, так как за время right - 1 это сделать будет уже нельзя. В качестве начального значения для left возьмем число -1 (за время -1 точно ничего напечатать нельзя), а в качестве начального значения для right необходимо взять время, за которое точно можно напечатать n страниц, например, можно взять значение an.

```
left = -1
right = a * n
while right > left + 1:
    t = (left + right) // 2
    if t // a + t // b >= n:
        right = t
    else:
        left = t
print(right)
```

Обратите внимание, что в качестве значения left нельзя выбрать значение 0, так как при $n = 0$ правильным ответом является 0, так как за время 0 можно напечатать 0 листов бумаги, в то время как начальным значением для left должно быть такое значение времени t, при котором задача не разрешима, поэтому в качестве значения left выбирается невозможное значение -1.

4.5 Линейный поиск в Python

В языке Python есть встроенные функции линейного поиска. Для проверки принадлежности элемента списку можно использовать бинарные операторы **in** и **not in**, возвращающее логическое значение. Например, 2 in [1, 2, 3] вернет True, а 2 not in [1, 2, 3] вернет False.

Для нахождения индекса появления некоторого элемента в списке можно использовать метод index.

5 Динамическое программирование

5.1 Одномерное динамическое программирование

Рассмотрим следующую задачу. На числовой прямой сидит кузнечик, который может прыгать вправо на одну или на две единицы. Первоначально

кузнечик находится в точке с координатой 0. Определите количество различных маршрутов кузнечика, приводящих его в точку с координатой n.

Обозначим количество маршрутов кузнечика, ведущих в точку с координатой n, как $F(n)$. Теперь научимся вычислять функцию $F(n)$. Прежде всего заметим, что $F(0) = 1$ (это вырожденный случай, существует ровно один маршрут из точки 0 в точку 0 — он не содержит ни одного прыжка), $F(1) = 1$, $F(2) = 2$. Как вычислить $F(n)$? В точку n кузнечик может попасть двумя способами — из точки $n - 2$ при помощи прыжка длиной 2 и из точки $n - 1$ прыжком длины 1. То есть число способов попасть в точку n равно сумме числа способов попасть в точку $n - 1$ и $n - 2$, что позволяет выписать рекуррентное соотношение: $F(n) = F(n - 2) + F(n - 1)$, верное для всех $n \geq 2$.

Теперь мы можем оформить решение этой задачи в виде рекурсивной функции:

```
def F(n):
    if n < 2:
        return 1
    else:
        return F(n - 1) + F(n - 2)
```

Но при попытке вычислить решение этой функции для уже не очень больших n, например, для $n = 40$, окажется, что эта функция работает крайне медленно. И при этом время работы функции с увеличением n растет экспоненциально, то есть такое решение неприемлемо по сложности. Причина этого заключается в том, что при вычислении рекурсивной функции подзадачи, для которых вычисляется решение, “перекрываются”. То есть для того, чтобы вычислить $F(n)$ нам нужно вызвать $F(n-1)$ и $F(n-2)$. В свою очередь $F(n - 1)$ вызовет $F(n - 2)$ и $F(n - 3)$. То есть функция $F(n - 2)$ будет вызвана два раза — один раз это будет сделано при вычислении $F(n - 1)$ и один раз — при вычислении $F(n - 2)$. Значение $F(n - 3)$ будет вычислено уже три раза, а значение $F(n - 4)$ будет вычисляться уже пять раз. При увеличении глубины рекурсии количество “перекрывающихся” вызовов функций будет расти экспоненциально. То есть одна из причин неэффективности рекурсивного решения — одно и то же значение функции вычисляется несколько раз, так как оно используется для вычисления нескольких других значений функции.

На самом деле несложно видеть, что значения рекурсивной функции в данном случае будут совпадать с числами Фибоначчи, так как вычисляются по тем же рекуррентным соотношениям. А для вычисления чисел Фибоначчи можно использовать цикл, а не рекурсию — следующее число Фибоначчи определяется, как сумма двух предыдущих.

```
F = [0] * (n + 1)
F[0] = 1
F[1] = 1
for i in range(2, n + 1):
    F[i] = F[i - 2] + F[i - 1]
```

Сложность такого решения будет $O(n)$. Сложность вычисления уменьшается за счет того, что для каждого промежуточного i значение $F(i)$ вы-

числяется один раз и сохраняется в списке, чтобы впоследствии использовать это значение несколько раз для вычисления $F(i+1)$ и $F(i+2)$.

Такой прием называется динамическим программированием. Динамическое программирование использует те же рекуррентные соотношения, что и рекурсивное решение, но в отличии от рекурсии в динамическом программировании значения вычисляются в цикле и сохраняются в списке. При этом заполнение списка идет от меньших значений к большим, в то время как в рекурсии — наоборот, рекурсивная функция вызывается для больших значений, а затем вызывает сама себя для меньших значений.

Модифицируем задачу. Пусть кузнечик прыгает на одну, две или три единицы, необходимо также вычислить количество способов попасть в точку n. В рекуррентном соотношении добавится еще одно слагаемое: $F(n) = F(n-1) + F(n-2) + F(n-3)$. И начальные значения для вычисления теперь должны состоять из трех чисел: $F(0)$, $F(1)$, $F(2)$. Решение изменится не сильно:

```
F = [0] * (n + 1)
F[0] = 1
F[1] = F[0]
F[2] = F[1] + F[0]
for i in range(3, n + 1):
    F[i] = F[i - 3] + F[i - 2] + F[i - 1]
```

Еще раз модифицируем задачу. Пусть некоторые точки являются “запретными” для кузнечика, он не может прыгать в эти точки. “Карта” запрещенных точек задается при помощи списка Map: если Map[i] == 0, то в точку номер i кузнечик не может прыгать, а если Map[i] == 1, то данная точка является разрешенной для кузнечика. Как и в предыдущей задаче, необходимо найти количество маршрутов в точку n.

В данном случае также придется модифицировать вид рекуррентного соотношения: если Map[i] == 0, то F[i] = 0, то есть если точка “запрещенная”, то количество способов попасть в эту точку равно 0, так как нет ни одного допустимого маршрута, заканчивающегося в этой точке. Если же Map[i] == 1, то значение F[i] вычисляется по тем же рекуррентным соотношениям, что и ранее. Получаем следующее решение:

```
F = [0] * (n + 1)
F[0] = 1
for i in range(1, n + 1):
    if Map[i] == 0:
        F[i] = 0
    else:
        F[i] = sum(F[max(0, i - 3): i])
```

Здесь используется немного другой код для вычисления суммы F[i - 3] + F[i - 2] + F[i - 1] для того, чтобы крайние значения F[1] и F[2] также можно было вычислить при помощи этого кода в основном цикле, а не перед ним.

Рассмотрим еще одну задачу. Пусть кузнечик прыгает на одну или две единицы, а за прыжок в каждую точку необходимо заплатить определенную стоимость, различную для различных точек. Стоимость прыжка в точку i задается значением Price[i] списка Price. Необходимо найти минимальную стоимость маршрута кузнечика из точки 0 в точку n.

На этот раз нам необходимо модифицировать определение целевой функции. Пусть $C(n)$ - минимальная стоимость пути из 0 в n. Выведем рекуррентное соотношение для $C(n)$. Чтобы попасть в точку n мы должны попасть в точку $n-1$ или $n-2$. Минимальные стоимости этих маршрутов будут равны $C(n-1)$ и $C(n-2)$ соответственно, к ним придется добавить значение Price[n] за прыжок в клетку n. Но из двух клеток $n-1$ и $n-2$ нужно выбрать тот маршрут, который имеет наименьшую стоимость. Получили рекуррентное соотношение: $C(n) = \min(C(n-1), C(n-2)) + Price(n)$.

Вычислить значение целевой функции также лучше при помощи динамического программирования, а не рекурсии:

```
C = [0] * (n + 1)
C[1] = Price[1]
for i in range(2, n + 1):
    C[i] = min(C[i - 1], C[i - 2]) + Price[i]
```

После выполнения этого цикла в списке C будет записана минимальная стоимость маршрута для всех точек от 0 до n.

Но помимо нахождения наименьшей стоимости маршрута, разумеется, хотелось бы найти и сам маршрут минимальной стоимости. Такая задача называется задачей “восстановления ответа”.

Для восстановления ответа будем для каждой точки i запоминать номер точки Prev[i], из которой кузнечик попал в точку i, если он будет передвигаться по пути минимальной стоимости. То есть Prev[i] — это точка, предшествующая точке с номером i на пути минимальной стоимости (также говорят, что Prev — это массив предшественников). Как определить Prev[i]? Если $C[i-1] < C[i-2]$, то кузнечик попал в точку i из точки i -1, поэтому Prev[i] = i - 1, иначе Prev[i] = i - 2. Модифицируем алгоритмы вычисления значений целевой функции, одновременно вычисляя значения Prev[i].

```
C = [0] * (n + 1)
C[1] = Price[1]
Prev[1] = 0
for i in range(2, n + 1):
    if C[i - 1] < C[i - 2]:
        C[i] = C[i - 1] + Price[i]
        Prev[i] = i - 1
    else:
        C[i] = C[i - 2] + Price[i]
        Prev[i] = i - 2
```

Теперь для восстановления пути необходимо начать с точки n и переходить от каждой точки к ее предшественнику, пока путь не дойдет до начальной точки с номером 0. Номера всех точек будем добавлять в список Path.

```
Path = []
i = n
while i > 0:
    Path.append(i)
    i = Prev[i]
Path.append(0)
Path = Path[::-1]
```

В конце в список Path добавляется начальная точка номер 0, которая не была обработана в основном цикле, а затем весь список Path разворачивается в обратном порядке (т. к. точки добавляются в обратном порядке, от конечной к начальной).

Но можно обойтись и без списка Prev. Мы в любой момент можем определить, из какой точки кузнечик пришел в точку i, если сравним C[i-1] и C[i-2]. Поэтому решение о том, к какой точке переходить при восстановлении ответа можно принимать непосредственно при восстановлении ответа, сравнив C[i-1] и C[i-2]. Путь восстанавливается через ту точку, для которой значение C будет меньше.

```
Path = []
i = n
while i > 0:
    if C[i - 1] < C[i - 2]:
        prev = i - 1
    else:
        prev = i - 2
    Path.append(prev)
    i = prev
Path.append(0)
Path = Path[::-1]
```

5.2 Двумерное динамическое программирование

Рассмотрим теперь вместо одномерных задач на движение по прямой перемещения в двумерном пространстве — например, перемещения на шахматной доске или на клетчатом листе бумаги.

5.2.1 Подсчет числа маршрутов

Рассмотрим шахматную доску в левом верхнем углу которой находится король. Король может перемещаться только вправо, вниз или по диагонали вправо-вниз на одну клетку. Необходимо определить количество различных маршрутов короля, приводящих его в правый нижний угол.

Сопоставим каждой клетке ее координаты (i, j), где i будет обозначать номер строки на доске, j — номер столбца. Нумеровать строки будем сверху вниз, столбцы — слева направо, нумерация начинается с 0. Тогда начальное положение короля будет клетка $(0, 0)$.

Обозначим через $F(i, j)$ количество способов прийти из клетки $(0, 0)$ в клетку (i, j). В клетку (i, j) можно прийти из трех клеток — слева из $(i, j-1)$, сверху из $(i-1, j)$ и по диагонали из $(i-1, j-1)$. Поэтому число маршрутов ведущих в клетку равно числу маршрутов из всех ее предшественников, а именно:

$F(i, j) = F(i, j-1) + F(i-1, j) + F(i-1, j-1)$

Отдельно нужно задать значения для граничных клеток, то есть когда $i = 0$ или $j = 0$. В результате получится таблица заполненная следующим образом:

1	1	1	1	1
1	3	5	7	9
1	5	13	25	41
1	7	25	63	129
1	9	41	129	321

Для заполнения этой таблицы и подсчета числа маршрутов можно использовать следующую программу, в которой сначала создается двумерный список, затем заполняются крайние клетки (первый столбец и первая строка), затем заполняются остальные элементы таблицы при помощи приведенного выше рекуррентного соотношения. В данном примере n — число строк, m — число столбцов на доске.

```
F = [[0] * m for i in range(n)]
for i in range(n):
    F[i][0] = 1
for j in range(m):
    F[0][j] = 1
for i in range(1, n):
    for j in range(1, m):
        F[i][j] = F[i][j - 1] + F[i - 1][j] + F[i - 1][j - 1]
```

На этом примере можно составить общий план решения задачи методом динамического программирования. Этот план можно использовать для решения любых задач при помощи динамического программирования:

1. Записать то, что требуется найти в задаче, как целевую функцию от некоторого набора аргументов (числовых, строковых или еще каких-либо).

2. Свести решение задачи для произвольного набора параметров к решению аналогичных подзадач для других наборов параметров (как правило, с меньшими значениями параметров). Если задача несложная, то полезно бывает выписать явное рекуррентное соотношение, задающее значение функции для данного набора параметров.

3. Задать начальные значения функции, то есть те наборы аргументов, при которых задача тривиальна и можно явно указать значение функции.

4. Создать массив (или другую структуру данных) для хранения значений функции. Как правило, если функция зависит от одного целочисленного параметра, то используется одномерный массив, для функции от двух целочисленных параметров - двумерный массив и т. д.

5. Организовать заполнение массива с начальных значений, определяя очередной элемент массива при помощи выписанного на шаге 2 рекуррентного соотношения или алгоритма.

Для заполнения первой строки и первого столбца таблицы мы использовали “специальную” формулу, отличающуюся от общего случая. Но в некоторых задачах удобней бывает все значения вычислять по одной и той же формуле, а для граничных значений функции ввести специальные “фиктивные” элементы. В данной задаче тоже можно так поступить — введем

специальную “каемочку” из одного фиктивного столбца слева и одной фиктивной строки сверху таблицы.

Для того, чтобы значения в остальной таблице вычислялись по общим формулам, во все клетки каемочки нужно записать число 0, кроме клетки $(0,0)$, в которую будет записано значение 1:

1	0	0	0	0	0
0	1	1	1	1	1
0	1	3	5	7	9
0	1	5	13	25	41
0	1	7	25	63	129
0	1	9	41	129	321

Теперь во всех остальных клетках таблицы значения могут быть вычислены по общей формуле: $F(i,j) = F(i,j-1) + F(i-1,j) + F(i-1,j-1)$, а программа может выглядеть так:

```
F = [[0] * (m + 1) for i in range(n + 1)]
F[0][0] = 1
for i in range(1, n + 1):
    for j in range(1, m + 1):
        F[i][j] = F[i][j - 1] + F[i - 1][j] + F[i - 1][j - 1]
```

5.2.2 Маршрут наименьшей стоимости

Теперь решим задачу о нахождении маршрута минимальной стоимости из левого верхнего угла в правый нижний, считая что для каждой клетке указана стоимость прохода через эту клетку. Сразу же будем считать, что таблица снабжена “каемочкой”, поэтому начальная клетка будет иметь индексы $(1,1)$, конечная клетка - (n,m), а строка и столбец с индексом 0 будут относится к фиктивной каемочке.

Если считать, что стоимость прохода через клетку (i,j) записана в отдельном списке $Price[i,j]$, то обозначив через (i,j) стоимость кратчайшего пути из начальной клетки $(1,1)$ в клетку (i,j) получим рекуррентное соотношение:

$C(i,j) = \min(C(i,j-1), C(i-1,j), C(i-1,j-1)) + Price[i,j]$

При этом при вычислении граничных значений (в первом столбце и первой строке) необходимо учитывать только клетки из этого столбца и этой строки (но не из предыдущего столбца и предыдущей строки). Это удобно реализовать, если заполнить предыдущую строк и предыдущий столбец “каемочкой”, записав для клеток каемочки значения функции C равными некоторому очень большому числу. Это число мы будем обозначать как INF, в качестве значения INF следует взять число, заведомо больше, чем максимальное число, которое может быть записано в таблице C. А в угол “каемочки” нужно записать число 0: $C[0][0] = 0$.

```
INF = 10 ** 20
C = [[0] * (m + 1) for i in range(n + 1)]
C[0][0] = 0
for i in range(1, n + 1):
    C[i][0] = INF
for j in range(1, m + 1):
```

```
    C[0][j] = INF
for i in range(1, n + 1):
    for j in range(1, m + 1):
        C[i][j] = min(C[i][j - 1], C[i - 1][j],
                      C[i - 1][j - 1]) + Price[i][j])
```

Теперь напишем восстановление ответа. Точно так же, как и в одномерном случае будем идти на конечной клетке в начальную, на каждом шаге выбирая ту из возможных предшествующих клеток, для которой значение функции C меньше.

```
Answer = []
i = n
j = m
while (i, j) != (0, 0):
    Answer.append((i, j))
    prev_C = min(C[i][j - 1], C[i - 1][j], C[i - 1][j - 1])
    if C[i][j - 1] == prev_C:
        i, j = i, j - 1
    elif C[i - 1][j] == prev_C:
        i, j = i - 1, j
    else:
        i, j = i - 1, j - 1
Answer = Answer[::-1]
```

5.2.3 Простые позиционные игры

Рассмотрим игру “Ферзя в угол” для двоих игроков. В левом верхнем углу доски размером $n \times m$ находится ферзь, который может двигаться только вправо-вниз. Игроки по очереди двигают ферзя, то есть за один ход игрок может переместить ферзя либо по вертикали вниз, либо по горизонтали вправо, либо во диагонали вправо-вниз. Игрок, который не сможет сделать хода — проигрывает, иными словами, выигрывает игрок, который поставит ферзя в правый нижний угол. Необходимо определить, какой из игроков может выиграть в этой игре независимо от ходов другого игрока.

Эту задачу также можно решить при помощи динамического программирования. Будем заполнять доску знаками “+” и “-”. Знак “+” будет означать, что данная клетка является выигрышной для ходящего игрока (то есть если ферзь стоит в этой клетке, то игрок, который делает ход, может всегда выиграть), а знак “-” означает, что ходящий игрок — проигрывает. Клетки последней строки, последнего столбца и диагонали, ведущей из правого нижнего угла необходимо отметить, как “+”, так как если ферзь стоит в этой клетке, то ходящий игрок может выиграть одним ходом:

+					+
	+				+
		+			+
			+		+
				+	+
+	+	+	+	+	-

Но в правом нижнем углу необходимо поставить знак “-” — если ферзь стоит в углу, то тот игрок, которых должен делать ход, уже проиграл.

Теперь рассмотрим две клетки, из которых можно пойти только в те клетки, в которых записан знак “+”. В этих клетках нужно записать знак “-” — если ферзь стоит в этих клетках, то какой бы ход не сделал ходящий игрок, ферзь окажется в клетке, в которой стоит знак “+”, то есть выигрывает ходящий игрок. Значит, тот, кто сейчас ходит — всегда проигрывает.

+					+
	+				+
		+			+
			+	-	+
			-	+	+
+	+	+	+	+	-

Но теперь в те клетки, из которых можно попасть в клетку, в которой стоит знак “-” за один ход, необходимо записать знак “+” — если ферзь стоит в этой клетке, то игрок, который делает ход, может выиграть, если передвинет ферзя в клетку, в которой стоит знак “-”.

Дальше таблица заполняется аналогично. В клетке ставиться знак “+”, если есть ход, который ведет в клетку, в которой стоит знак “-”. В клетке ставится знак “-”, если все ходы из этой клетки ведут в клетки, в которых записан знак “+”. Продолжая таким образом можно определить выигрывающего игрока для любой начальной клетки.

+	+	-	+	+	+
-	+	+	+	+	+
+	+	+	+	+	+
+	+	+	+	-	+
+	+	+	-	+	+
+	+	+	+	+	-

5.3 Наибольшая общая подпоследовательность

Рассмотрим две строки (или числовые последовательности) A и B. Пусть первая строка состоит из n символов $a_0...a_{n-1}$, вторая строка состоит из m символов $b_0...b_{m-1}$. Подпоследовательностью данной строки (последовательности) называется некоторое подмножество символов исходной строки, следующих в том же порядке, в котором они идут в исходной строке, но не обязательно подряд. Если в строке n символов, то у нее 2^n различных подпоследовательностей: каждый из n символов строки может либо входить, либо не входить в любую выбранную подпоследовательность. Пустая подпоследовательность не содержит ни одного элемента и также является подпоследовательностью любой строки.

Рассмотрим задачу — для двух данных строк найти такую строку наибольшей длины, которая была бы подпоследовательностью каждой из них. Например, если A=’abcabaac’, B=’baccbca’ то у строк A и B них есть общая подпоследовательность длины 4, например, ’acba’ или ’acbc’.

Данную задачу можно решить перебором — например, перебрав все 2^n подпоследовательностей первой строки и для каждой их них проверив, является ли она подпоследовательностью второй строки. Но при помощи динамического программирования эту же задачу можно решить за сложность $O(nm)$.

Рассмотрим последние символы данных строк a_{n-1} и b_{m-1}. Если эти

символы совпадают, то они обязательно войдут последними символами и в наибольшую общую подпоследовательность данных строк. Тогда можно свести задачу нахождения наибольшей общей подпоследовательности для строк $A = a_0...a_{n-1}$ и $B = b_0...b_{m-1}$ к задаче нахождения наибольшей общей подпоследовательности для строк, полученных отбрасыванием от данных строк последнего символа, то есть для $a_0...a_{n-2}$ и $b_0...b_{m-2}$. Затем к ответу для "укороченных" строк добавим последние (равные) символы исходных строк (a_{n-1} или b_{m-1}) и получим ответ для исходных строк.

Если же последние символы исходных строк не совпадают, то эти символы (a_{n-1} и b_{m-1}) не могут одновременно входить в наибольшую общую подпоследовательность, поэтому можно один из них отбросить. Тогда задача сводится к нахождению наибольшей общей подпоследовательности для одного из двух случаев - для строк $a_0...a_{n-2}$ и $b_0...b_{m-1}$ или для строк $a_0...a_{n-1}$ и $b_0...b_{m-2}$.

Мы научились сводить задачу нахождения наибольшей общей подпоследовательности двух строк к меньшей задаче - нахождения наибольшей общей подпоследовательности для строк, полученных отбрасыванием последних символов от исходных строк, то есть для префиксов исходных строк. Для дальнейшего решения задачи будем следовать принципу построения решения при помощи динамического программирования.

Рассмотрим префикс A' первой строки из i символов: $A' = a_0...a_{i-1}$ и префикс B' второй строки из j символов: $B' = b_0...b_{j-1}$. В терминах срезов языка Питон можно считать, что $A' = A[:i]$ и $B' = B[:j]$. Обозначим через $F(i, j)$ длину наибольшей общей подпоследовательности для A' и B'.

Теперь выпишем рекуррентные соотношения. Они зависят от того, совпадают ли последние символы рассматриваемых строк A' и B'. Если $a_{i-1} = b_{j-1}$, то тогда $F(i, j) = F(i-1, j-1)+1$ - нужно решить задачу для строк, полученных отбрасыванием последних символов рассматриваемых строк и добавить 1 символ к ответу. В противном случае нужно рассмотреть два случая: $F(i-1, j)$ и $F(i, j-1)$, которые соответствуют отбрасыванию по одному символу от конца каждой из рассматриваемых строк. В этом случае $F(i, j) = \max(F(i-1, j), (i, j-1))$.

Начальные значения функции F задаются просто: если одна из строк — пустая, то общая подпоследовательность также пустая, то есть имеет длину 0: $F(0, j) = F(i, 0) = 0$.

Далее необходимо завести двумерный массив размером $(n+1) \times (m+1)$ и заполнить его значениями по указанным рекуррентным соотношениям. Сначала весь массив заполним нулями (что задаст граничные значения), затем двумя вложенными циклами по i и по j заполним оставшуюся часть массива:

```
n = len(A)
m = len(B)
F = [[0] * (m + 1) for i in range(n + 1)]
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if A[i - 1] == B[j - 1]:
            F[i][j] = F[i - 1][j - 1] + 1
        else:
            F[i][j] = max(F[i - 1][j], F[i][j - 1])
```

```
print(F[n][m])
```

В таблице ниже приведен пример заполнения массива для строки 'abcabaac' и 'baccbca'. Длина наибольшей общей подпоследовательности для данных строк равна 4.

		b	**a**	**c**	**c**	**b**	**c**	**a**
	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1
b	0	1	1	1	1	2	2	2
c	0	1	1	2	2	2	3	3
a	0	1	1	2	2	2	3	4
b	0	1	1	2	2	2	3	4
a	0	1	1	2	2	2	3	4
a	0	1	1	2	2	2	3	4
c	0	1	2	3	3	3	4	4

Этот код находит длину наибольшей общей подпоследовательности. Для нахождения самой общей подпоследовательности необходимо восстановить ответ. Для этого выполним "обратный проход" по массиву F начиная с последнего элемента. В каждой рассматриваемой ячейке $F[i][j]$ выясним, как было получено значение в этой ячейке. Это зависит от последних символов рассматриваемых префиксов. Если $a_{i-1} = b_{j-1}$, то тогда ответ для элемента $F[i][j]$ получен из $F[i-1][j-1]$ добавлением 1, поэтому перейдем к элементу $F[i-1][j-1]$, а к ответу добавим символ $a_{i-1} = b_{j-1}$. Иначе нужно перейти к тому элементу $F[i-1][j]$ или $F[i][j-1]$, значение в котором совпадает со значением $F[i][j]$. Алгоритм восстановления ответа записан ниже:

```
Ans = []
i = n
j = m
while i > 0 and j > 0:
    if A[i - 1] == B[j - 1]:
        Ans.append(A[i - 1])
        i -= 1
        j -= 1
    elif F[i - 1][j] == F[i][j]:
        i -= 1
    else:
        j -= 1
Ans = Ans[::-1]
```

5.4 Наибольшая возрастающая подпоследовательность

Рассмотрим числовую последовательнсоть из n элеметов $a_0 ... a_{n-1}$. Рассмотрим задачу нахождения среди всех возможных подпоследовательностей данной последовательности монотонно возрастающей подпоследовательности наибольшей длины.

У данной задачи есть несколько способов решения.

Первый способ — отсортируем последовательность в порядке неубывания, удалим из нее повторяющиеся элементы (то есть получим строго возрастающую последовательность B из элементов $b_0, \ldots, b_{m-1}$. Теперь для

последовательностей A и B найдем наибольшую общую подпоследовательность. Понятно, что эта подпоследовательность будет подпоследовательностью A, будет монотонно возрастать и будет иметь наибольшую длину из всех таких последовательностей. Сложность такого алгоритма будет $O(n^2)$.

Возможен и другой способ решения задачи при помощи динамического программирования. Обозначим через $F(i)$ длину наибольшей возрастающей подпоследовательности, последним элементом которой будет элемент a_i. Тогда для вычисления значения $F(i)$ рассмотрим предпоследний элемент этой последовательности. Пусть это элемент a_j, тогда $j < i$ и $a_j < a_i$. Длина наибольшой возрастающей подпоследовательности, заканчивающейся a_j есть $F(i)$, значит, необходимо найти такое подходящее j, что $F(j)$ будет наибольшим. Итак, $F(i) = 1 + \min\limits_{j<i, a_j<a_i} F(j)$. Если же ни одного такого подходящего j нет (то есть все $a_j \geq a_i$ при $j < i$), то $F(i) = 1$.

Соответствующая программа вычисления значений функции F будет выглядеть так:

```
F = [0] * len(A)
for i in range(len(A)):
    for j in range(i):
        if A[j] < A[i] and F[j] > F[i]:
            F[i] = F[j]
    F[i] += 1
```

Ответом (длиной наибольшей возрастающей подпоследовательности) будет наибольшее значение $F(i)$.

Для восстановления ответа найдем такой элемент последовательности a_i, что $F(i)$ будет максимальным. Это будет последний элемент наибольшей возрастающей подпоследовательности. Теперь найдем предыдущий элемент. Это такой элемент a_j, что $j < i$ и $F(j) = F(i) - 1$. Будем повторять поиск предыдущего элемента до тех пор, пока не дойдем до такого j, что $F(j) =$, это будет первым элементом наибольшей возрастающей подпоследовательности.

Легко видеть, что сложность такого алгоритма $O(n^2)$. Тем не менее, можно придумать решение, которое будет иметь сложность $O(n \log n)$.

Рассмотрим следующую функцию: $F(i)$ - наименьший элемент последовательности, которым может заканчиваться наибольшая возрастающая подпоследовательность длины i. то есть будем для каждой возможной длины i возрастающей подпроследовательности пытаться выбрать подпоследовательность из i элементов, при этом минимизировать последний элемент последовательности.

Также добавим фиктивные элементы: $F(0) = -\inf$, а также $F(n+1) = +\inf$, что имеет следующий смысл - последовательность длины $n+1$ не существует, то есть считаем, что последний элемент бесконечной большой (к ней ничего больше нельзя добавить). А последовательность длины 0 наоборот имеет последним элементом “минус бесконечность”, то есть к ней можно добавить что угодно.

Теперь будем по одному рассматривать элементы исходной последовательности, начиная с самого первого. При этом начальная инициализация такая: $F(0) = -\inf$, $F(i) = +\inf$ при $i > 0$, то есть в самом начале не известно ни одной возрастающей подпоследовательности, даже длины 1.

Теперь рассмотрим очередной элемент a_i. Его можно добавить в конец любой возрастающей подпоследовательности, которая заканчивается числом, меньшим, чем a_i. Если есть возрастающая подпоследовательность длины k, которая заканчивается каким-то значением x, то есть $F(k) = x$, при этом $x < a_i$, то возможно построить последовательность длины $k+1$ с последним элементом, равным a_i, добавив его в конец подпоследовательности длины k. При этом это должно приводить к улучшению значения $F(k+1)$, то есть $F(k+1)$ должно быть не меньше, чем a_i. Это означает, что необходимо среди элементов списка F найти такое значение k, что $F(k) < a_i$, $F(k+1) \geq a_i$, после чего необходимо установить $F(k+1) = a_i$. Это можно сделать при помощи функции двоичного поиска LowerBound, которая позволяет в списке F найти первое значение, не меньшее, чем a_i, почле чего ему необходимо присвоить a_i. Заметим, что при этом монотонность значений в списке F сохраняется, что делает оправданным двоичный поиск.

Соответствующий алгоритм может выглядеть следующим образом:

```
INF = 10 ** 10
F = [INF] * (len(A) + 1)
F[0] = -INF
for i in range(len(A)):
    left = 0
    right = len(A)
    while right - left > 1:
        middle = (left + right) // 2
        if F[middle] >= A[i]:
            right = middle
        else:
            left = middle
    F[right] = A[i]
```

Ответом (длиной наибольшей возрастающей подпоследовательности) является такое наибольшее i, что $F(i) \neq +\inf$.

Восстановление ответа для данной задаче оставим в качестве самостоятельного упражнения. Подскажем, что для восстановления ответа необходимо хранить для каждого элемента предыдущий элемент, а также для каждой возможной длины i необходимо хранить номер последнего элемента возрастающей подпоследовательности длины i в исходной последовательности.

5.5 Задача об укладке рюкзака

5.5.1 Задача “Банкомат”

Рассмотрим следующую задачу. В банкомате имеется банкноты n различных номиналов $a_1, a_2, \ldots, a_n$. Клиент хочет получить сумму в K денежных единиц. Необходимо определить, при помощи какого минимального числа банкнот можно выдать эту сумму (а при необходимости восстановления ответа - определить способ выдачи, использующий минимальное число банкнот).

Очевидно приходящий в голову “жадный алгоритм” — выдавать банкноты наиболее крупного номинала, пока это возможно, затем переходить к

более мелкому номиналу, в общем случае неверен. Например, пусть номиналы банкнот $a_1 = 1$, $a_2 = 20$, $a_3 = 90$, а сумма для выдачи $K = 100$. Тогда "жадный" алгоритм выдаст банкноту в 90 и 10 банкнот по 1, в то время как существует решение, использующее всего лишь две банкноты по 50.

На помощь придет динамическое программирование. Пусть $F(k)$ — минимальное число банкнот, при помощи которых можно выдать сумму в k рублей. Выберем одну из банкнот, входящую в оптимальный способ выдачи. Пусть это банкнота a_i. Тогда необходимо выдать оставшуюся сумму $k - a_i$, что можно сделать при помощи $F(k - a_i)$ банкнот. То есть $F(k) = 1 + F(k - a_i)$. Далее необходимо взять минимум по всем возможным значениями i: $F(k) = 1 + min_i F(k - a_i)$.

Начальные значения удобно сделать такими: $F(0) = 0$, $F(k) = +\infty$ при $k < 0$. Значение $+\infty$ будет означать невозможность выдачи суммы вообще, то есть "очень плохой вариант", когда банкнот потребуется бесконечно много. Можно добавить к списку значений функции $F(k)$ "каемку" для отрицательных значений k, но лучше просто рассматривать только такие значения k и i, когда $k - a_i \geq 0$.

В данном примере программы будем считать, что номиналы банкнот хранятся в списке A и нумерация банкнот начинается с числа 0.

```
INF = 10 ** 10
F = [INF] * (K + 1)
F[0] = 0
for k in range(1, K + 1):
    for i in range(len(A)):
        if k - A[i] >= 0 and F[k - A[i]] < F[k]:
            F[k] = F[k - A[i]]
    F[k] += 1
```

Для восстановления ответа будем опять идти к началу списка, уменьшая сумму k, выбирая такую банкноту a_i, что $F(k) = F(k - a_i) + 1$. Номиналы банкнот, которые будут при этом использоваться для восстановления ответа, будут записаны в список Ans.

```
Ans = []
k = K
while k != 0:
    for i in range(len(A)):
        if k - A[i] >= 0 and F[k] == F[k - A[i]] + 1:
            Ans.append(A[i])
            k -= A[i]
```

5.5.2 Задача "Золотые слитки"

Предыдущий алгоритм также решал задачу проверки возможности выдачи любой суммы при помощи заданного номинала банкнот — сумму k возможно выдать, если $F(k) < +\infty$.

Также в предыдущей задаче банкнот каждого номинала было неограниченно много. Теперь рассмотрим задачу, в которой существует ровно одна банкнота каждого номинала (но номиналы банкнот могут повторяться),

необходимо проверить, можно ли заданную сумму K выдать при помощи данных банкнот.

Этой задаче можно придать и такой смысл: есть n золотых слитков массами a_1, a_2, ..., a_n. Какую максимальную массу золота можно унести, если она не может превышать K (то есть "грузоподъемность" не превосходит K).

Задачу также можно решать при помощи динамического программирования. Пусть $F(k)$ — признак того, можно ли набрать слитков на массу в точности k, то есть одно из двух значений True (или 1) или False (или 0).

Будем по очереди рассматривать все слитки, обновляя значения $F(k)$. При рассмотрении слитка a_i необходимо пометить $F(k) = 1$, если $F(k-a_i) = 1$, то есть если массу в точности k можно набрать, если ранее было возможно набрать массу в точности $k - a_i$.

```
F = [0] * (K + 1)
F[0] = 1
for i in range(len(A)):
    F_new = F[:] # копия списка F для обновления
    for k in range(A[i], K + 1):
        if F[k - A[i]] == 1:
            F_new[k] = 1
    F = F_new
```

Данный алгоритм нуждается в пояснении. В самом начале список F заполняется значением 0, кроме $F(0) = 1$, что означает, что нулевую массу можно набрать не используя ни одного слитка, а любую другую массу не используя ни одного слитка, набрать нельзя.

Внутренний цикл начинается со значения a_i, так как массу k можно набрать, если ранее было возможно набрать массу $k-a_i$, то есть минимальное значение для k равно a_i.

Кроме того, нельзя вносить исправления сразу же в список F, потому что в этом случае один предмет будет учтен более одного раза, то есть будут помечены единицами $F(a_i)$, затем $F(2a_i)$, так как $F(a_i) = 1$, затем $F(3a_i)$ и т.д. Чтобы избежать этой ошибки, в этом алгоритме создается копия списка F и в него вносятся изменения, тем самым не будут учитываться изменения в списке, сделанные при добавлении слитка a_i.

Другой способ избежать этой проблемы — обходить список F с конца — от большего значения к меньшему.

```
F = [0] * (K + 1)
F[0] = 1
for i in range(len(A)):
    for k in range(K, A[i] - 1, -1):
        if F[k - A[i]] == 1:
            F[k] = 1
```

Для восстановления ответа для каждого значения k будем хранить массу слитка, при помощи которого он был получен, будем хранить его в списке $Prev$. Значение списка $Prev$ будет обновляться при записи $F(k) = 1$:

```
F = [0] * (K + 1)
F[0] = 1
Prev = [0] * (K + 1)
```

```
for i in range(len(A)):
    for k in range(K, A[i] - 1, -1):
        if F[k - A[i]] == 1:
            F[k] = 1
            Prev[k] = A[i]
```

Теперь для восстановления ответа будем уменьшать значение k на $Prev(k)$ пока не получим $k = 0$.

```
Ans = []
k = K
while k > 0:
    Ans.append(Prev[k])
    k -= Prev[k]
```

5.5.3 Дискретная задача об укладке рюкзака

Следующим обобщением задачи про золотые слитки является "задача об укладке рюкзака". В этой задаче также имеется несколько предметов, для каждого предмета заданы две характеристики: вес $w_i > 0$ и стоимость («полезность») предмета $p_i > 0$. Необходимо выбрать множество предметов суммарной максимальной стоимости, при этом суммарная масса выбранных предметов должна быть ограничена значением K.

Одним из способов решения этой задачи является полный перебор всех подмножеств из n предметов, которых будет 2^n и выбор среди них наилучшего подмножества, удовлетворяющего условиям задачи. Такой алгоритм будет иметь сложность $O(2^n)$.

Если же массы всех предметов являются целыми числами (так называемая "дискретная" задача), то в данном случае возможно придумать решение сложности $O(nK)$ при помощи динамического программирования.

Как и в задаче про золотые слитки, будем для каждой возможной массы k хранить информацию о способе набора этой массы, но в отличии от задачи про слитки будем хранить не возможность набора данной массы (0 или 1), а наилучшее решение для данной массы, то есть наибольшую стоимость предметов, которые можно набрать в рюкзак данной массы.

Формально определим так: $F(i, k)$ - максимальная стоимость предметов, которые можно уложить в рюкзак массы k, если можно использовать только первые i предметов.

Выведем рекуррентное соотношение для $F(i, k)$ уменьшив значение i. Есть две возможности собрать рюкзак, используя первые i предметов - взять предмет с номером i или не брать.

Если не брать предмет с номером i, то в этом случае $F(i, k) = F(i-1, k)$, так как рюкзак массы k будет собран только с использованием первых $i-1$ предмета. В этом случае $F(i, k) = F(i-1, k)$.

Если же предмет номер i войдет в рюкзак (это можно сделать только при $k \geq w_i$), то останется свободная вместимость рюкзака $k - w_i$, которую можно будет заполнить первыми $i-1$ предметом, максимальная стоимость рюкзака в этом случае будет $F(i-1, k-w_i)$. Но поскольку предмет номер i был включен в рюкзак, то стоимость рюкзака увеличится на p_i. То есть в этом случае $F(i, k) = F(i-1, k-w_i) + p_i$.

Из двух возможных вариантов нужно выбрать вариант наибольшей стоимости, то есть $F(i,k) = \max(F(i-1,k), F(i-1,k-w_i) + p_i)$.

Для хранения значения функции F будем использовать двумерный список. При этом массы предметов хранятся в списке W, их стоимости — в списке P. Будем считать (для простоты записи программы), что предметы пронумерованы от 1 до n.

```
F = [ [0] * (K + 1) for i in range(n + 1)]
for i in range(1, n + 1):
    for k in range(1, K + 1):
        if k >= W[i]:
            F[i][k] = max(F[i - 1][k], F[i - 1][k - W[i]] + P[i])
        else:
            F[i][k] = F[i - 1][k]
```

Для восстановления ответа будем перебирать все предметы "с конца" от n до 1. В переменной k будет храниться текущая вместимость рюкзака. Рассматривая предмет номер i определим, как было получено значение $F(i,k)$. Если $F(i,k) = F(i-1,k)$, то можно не включать предмет i в рюкзак и перейти к предмету $i-1$ не меняя значения k. Иначе предмет i нужно включить в рюкзак, при этом значение k уменьшается на w_i.

```
k = K
for i in range(n, 0, -1):
    if F[i][k] != F[i - 1][k]:
        Ans.append(i)
        k -= W[i]
```

В этой реализации случай, когда $F(i,k) = F(i-1,k)$ просто пропускается, и рассматривается только случай $F(i,k) \neq F(i-1,k)$. После окончания алгоритма в списке Ans будут храниться номера предметов, входящих в рюкзак.

6 Рекурсивный перебор

Важную роль среди алгоритмов играют алгоритмы перебора различных комбинаторных структур. Например, можно перебирать все подмножества данного множества, все подмножества, содержащие заданное число элементов, все перестановки данного набора элементов. Основным способом перебора комбинаторных структур является рекурсия.

6.1 Перебор всех подмножеств

Пусть дано некоторое множество, содержащее n элементов $a_0, a_1, \ldots, a_{n-1}$. Необходимо перебрать все его подмножества. Общее число подмножеств n-элементного множества равно 2^n. Например, у множества из 3 элементов $\{1,2,3\}$ восемь подмножеств: {} (пустое подмножество), $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{1,2,3\}$.

Каждое подмножество будем кодировать строкой из n символов, где i-й символ будет равен 0, если a_i не входит в выбранное подмножество, или равен 1, если a_i входит в выбранное подмножество. То есть строка из одних

нулей соответствует пустому множеству, а строка из одних единиц соответствует подмножеству, совпадающему со всем множеством.

Таким образом, задача сводится к перебору всех двоичных строк (т. е. состоящих из символов 0 или 1) длины n. Строки будем перебирать в лексикографическом порядке, то есть для $n = 3$ порядок будет таким:

```
000
001
010
011
100
101
110
111
```

В лексикографическом порядке все строки упорядочены по первому символу, т.е. сначала нужно вывести те строки, у которых первый символ равен 0, затем те строки, у которых первый символ равен 1. Те строки, у которых первые символы равны, упорядочены по второму символу, затем — по третьему и т. д.

Если посмотреть на пример для $n = 3$ то можно видеть, что результат перебора построен по следующему принципу. Сначала нужно взять первый символ строки равный 0, затем к нему всеми возможными способами дописать следующие $n - 1$ символ также в лексикографическом порядке. Потом поставим на первое место символ, равный 1 и допишем к нему всеми возможными способами следующие $n - 1$ символ.

В свою очередь, если у нас уже есть какая-то сформированная начальная часть строки (например, первый символ равен 0), то нужно к нему добавить сначала символ 0 (получим начальную часть вида 00) и так же рекурсивно дописать оставшиеся $n - 2$ символа, затем добавить символ 1 (получим начальную часть вида 01), и рекурсивно добавить к этой начальной части еще $n - 2$ символа.

Решение оформим в виде рекурсивной функции generate с двумя параметрами. Первый параметр - число n равное количеству символов строки, которые нужно сгенерировать. Второй параметр — строка prefix, в которой хранится уже сгенерированная начальная часть строки. Функция будет добавлять по одному символу к уже построенной части prefix, сначала добавляя символ “0”, затем символ “1”, после чего функция будет рекурсивно вызывать себя для построения $n - 1$ оставшегося символа.

Окончание рекурсии — случай $n = 0$, в этом случае функция больше не должна ничего строить и должна вывести построенную строку, которая целиком будет храниться в переменной prefix. Если требуется не вывести полученные строки на экран, а как-то обработать данное подмножество, то вместо функции print нужно вызвать функцию обработки подмножества.

Во всех остальных случаях функция дважды вызывает себя рекурсивно для построения строки длины $n - 1$ — сначала добавив к строке prefix “0”, затем добавив “1”. Функция может быть реализована следующим образом:

```
def generate(n, prefix):
    if n == 0:
        print(prefix)
```

```
    else:
        generate(n - 1, prefix + "0")
        generate(n - 1, prefix + "1")
```

Для того, чтобы вывести все двоичные строки длины 5, нужно вызвать эту функцию так: generate(5,)

Попробуем модифицировать эту функцию. Допустим, нужно вывести все двоичные строки, в которых нет двух символов «1» подряд. Это означает, что добавить символ “1” можно только в том случае, если prefix оканчивается на символ “0”, а также в случае пустой строки. Нужно добавить только одно условие:

```
def generate(n, prefix):
    if n == 0:
        print(prefix)
    else:
        generate(n - 1, prefix + "0")
        if prefix == "" or prefix[-1] == "0":
            generate(n - 1, prefix + "1")
```

6.2 Перебор всех k-элементных подмножеств

Рассмотрим задачу построения всех подмножеств данного множества, содержащих ровно k единиц. Такой объект (k-элементное подмножество n-элементного множества) можно задать несколькими способами.

Первый способ — двоичная строка длины n в которой ровно k единиц. Для построения такой строки модифицируем функцию generate, добавив в нее еще один дополнительный параметр — число единиц k, которое необходимо добавить к исходной строке. Добавляя к строке prefix символ “0”, рекурсию нужно вызывать с параметрами $(n-1, k)$, то есть нужно сгенерировать еще $n-1$ символ, среди которых должно быть k единиц, т.к. новых единиц добавлено не было. А если к строке prefix был добавлен символ “1”, то рекурсию нужно будет вызывать с параметрами $(n-1, k-1)$.

Но нужно поставить еще дополнительные условия, ограничивающие случаи, когда функцию можно вызывать рекурсивно. А именно, символ “1” можно добавить только в том случае, когда $k > 0$. А символ “0” можно добавить при условии, что $k < n$, так как при $k = n$ все оставшиеся символы должны быть единицами.

```
def generate(n, k, prefix):
    if n == 0:
        print(prefix)
    else:
        if k < n:
            generate(n - 1, k, prefix + "0")
        if k > 0:
            generate(n - 1, k - 1, prefix + "1")
```

Другой способ представления множества — список выбранных элементов. Пусть в множестве n элементов — числа от 1 до n. Тогда каждое множество - это некоторый набор из k неповторяющихся чисел от 1 до n. Чтобы

представление каждого подмножества было единственным, будем считать, что числа в наборе упорядочены по возрастанию. Например, при $n = 4$ и $k = 2$ есть 6 различных множеств: $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$.

Таким образом, задача сводится к задаче построения всех возрастающих последовательностей длины k составленных из чисел от 1 до n. Для построения всех таких последовательностей можно использовать следующую функцию:

```
def generate(n, k, prefix):
    if k == 0:
        print(prefix)
    else:
        if len(prefix) == 0:
            next = 1
        else:
            next = prefix[-1] + 1
        while next + k - 1 <= n:
            generate(n, k - 1, prefix + [next])
            next += 1
```

В этом годе параметр prefix — это список уже построенного начала последовательности, поэтому вызывать функцию generate нужно передавая последним параметром пустой список. Параметр n — это максимальное число элементов последовательности, параметр k — это количество элементов последовательности, которое еще необходимо добавить к списку prefix.

Если $k = 0$, то больше добавлять к prefix нечего и рекурсия заканчивается. Во всех остальных случаях перебирается следующий элемент, который необходимо добавить к prefix. Его значение перебирается в цикле по переменной next. Минимальное значение next на 1 больше последнего элемента списка prefix, а если prefix пуст, то минимальное значение next равно 1. Дальше возможное значение next увеличивается на 1, при это элемент next добавляется к списку prefix. Максимальное значение next равно $n - k + 1$, т. к. после числа next необходимо записать еще $k - 1$ число, большее next, но не превосходящее n.

6.3 Перебор всех перестановок

Напишем алгоритм перебора всех перестановок чисел от 1 до n, то есть последовательности, полученной из списка чисел от 1 до n изменением порядка их следования. Известно, что таких перестановок существует $n!$, напишем функцию, которая выводит все перестановки в лексикографическом порядке.

Как и ранее, функция получает в качестве параметра значение n (длина перестановки) и prefix — уже построенное начало перестановки. Далее перебираются все числа от 1 до n в порядке возрастания в качестве возможного продолжения перестановки. Если число не содержится в списке prefix, то к списку prefix добавляется следующий элемент последовательности и функция вызывается рекурсивно.

Окончание рекурсии происходит в случае, если список prefix имеет длину n, то есть все числа от 1 до n уже содержатся в списке prefix.

```
def generate(n, prefix):
    if len(prefix) == n:
        print(prefix)
    else:
        for next in range(1, n + 1):
            if next not in prefix:
                generate(n, prefix + [next])
```

7 Комбинаторные объекты

В задачах этой главы более подробно будут рассматриваться различные комбинаторные объекты — все подмножества n-элементного множества, все k-элементные подмножества множества, перестановки, правильные скобочные последовательности, разбиения на слагаемые. Наиболее типичные задачи, решаемые для этих объектов: подсчет количества объектов, перебор всех объектов, построение предыдущего и следующего объекта в лексикографическом порядке, определение порядкового номера объекта и построение объекта по его порядковому номеру.

7.1 Подмножества

Начнем с самый простых объектов — всех подмножеств данного n-элементного множества. Нетрудно подсчитать общее число подмножеств — каждый из n объектов может независимо от других входить или не входить в подмножество, поэтому общее число подмножеств есть $2 \cdot 2 \cdot \ldots \cdot 2 = 2^n$. Перенумеруем все элементы множества числами от 0 до $n-1$. Можно закодировать каждое из подмножеств строкой из n чисел 0 или 1, считая, что i-й элемент строки равен 1, если i-й элемент подмножества включается в множество или 0 если не включается. Тогда каждому подмножеству сопоставляется строка из n символов 0 или 1. Алгоритм перебора всех таких строк был рассмотрен в предыдущей главе:

```
def generate(n, prefix):
    if n == 0:
        print(prefix)
    else:
        generate(n - 1, prefix + "0")
        generate(n - 1, prefix + "1")
```

Если такую строку рассмотреть, как запись некоторого числа в двоичной системе счисления, то каждому подмножеству сопоставляется некоторое число, содержащее не более n цифр в двоичной системе счисления, то есть число от 0 до $2^n - 1$. Поэтому вместо рекурсивного перебора можно просто перебрать все числа от 0 до $2^n - 1$. Для перевода числа в двоичную форму воспользуемся функцией bin, удалив из значения, которое возвращает функция bin первые два символа — префикс `0b`, который функция bin всегда дописывает в начало возвращаемой строки, затем дополним строку слева нулями до длины n:

```
for i in range(2 ** n):
```

```
S = bin(i)[2:]
S = "0" * (n - len(S)) + S
print(S)
```

Представление каждого подмножества в виде двоичного кода позволяет удобно работать с этими объектами, поскольку для всех двоичных строк длины n их лексикографический порядок соответствует арифметическому порядку соответствующих им чисел, например:

$$\begin{aligned} 0_{10} &= 000_2 \\ 1_{10} &= 001_2 \\ 2_{10} &= 010_2 \\ 3_{10} &= 011_2 \\ 4_{10} &= 100_2 \\ 5_{10} &= 101_2 \\ 6_{10} &= 110_2 \\ 7_{10} &= 111_2 \end{aligned}$$

Если занумеровать все последовательности числами, то для определения последовательности по номеру можно использовать функцию bin (как в примере выше), а для определения номера по последовательности нужно преобразовать строку в целое число, считая, что строка содержит запись целого числа в двоичной систему счисления. Для этого можно использовать функцию int, передав ей в качестве второго параметра число 2 — основание используемой системы счисления.

Другая распространенная задача — получение следующего и предыдущего в установленном порядке комбинаторного объекта. Например, для двоичной строки '101' следующим объектом будет объект '110', а предыдущим — '100'. Для построения следующей двоичной строки можно соответствующее ей число увеличить на 1, а для построения предыдущего — уменьшить на 1.

Пример функции, генерирующую следующую в лексикографическом порядке двоичную последовательность:

```
def NextBinarySequence(S):
    n = len(S)
    if S == '1' * n:
        return None
    else:
        S = bin(int(S, 2) + 1)[2:]
        return '0' * (n - len(S)) + S
```

Функция получает на вход строку из нулей и единиц и возвращает следующую в лексикографическом порядке строку из нулей и единиц такой же длины. Если же следующей двоичной строки не существует, то есть исходная строка уже состояла из одних единиц, то функция возвращает значение None.

Функция построения предыдущей последовательности выглядит аналогично:

```
def PrevBinarySequence(S):
    n = len(S)
    if S == '0' * n:
        return None
    else:
        S = bin(int(S, 2) - 1)[2:]
        return '0' * (n - len(S)) + S
```

Задачу построения следующего и предыдущего объекта можно решить и без использования целочисленной арифметики. Общий алгоритм построения следующего в лексикографическом порядке комбинаторного объекта такой:

1. Найти самый правый (то есть имеющий наибольший индекс) элемент комбинаторного объекта, который допускает увеличение.
2. Увеличить этот элемент на минимально возможную величину.
3. Элементы, которые стоят правее, изменить так, чтобы они образовывали минимально возможный объект в лексикографическом порядке.

Для двоичных последовательностей увеличение какого-либо элемента последовательности — это замена символа 0 на 1. Значит, нужно найти самый правый элемент последовательности, который равен 0. Если такого элемента нет, то последовательность состоит из одни единиц и является максимальной в лексикографическом порядке. Найденный элемент необходимо увеличить, то есть заменить на 1, а все элементы правее него сделать минимально возможными, то есть заменить на 0.

```
def NextBinarySequence(S):
    n = len(S)
    i = n - 1
    while i > 0 and S[i] == "1":
        i -= 1
    if i < 0:
        return None
    else:
        return S[:i] + "1" + "0" * (n - i - 1)
```

Аналогично устроен алгоритм нахождения предыдущей последовательности: необходимо найти самый правый символ, равный 1, заменить его на символ 0, все символы правее него сделать максимально возможными в лексикографическом порядке, то есть заменить на символ 1.

```
def PrevBinarySequence(S):
    n = len(S)
    i = n - 1
    while i > 0 and S[i] == "0":
        i -= 1
    if i < 0:
        return None
    else:
        return S[:i] + "0" + "1" * (n - i - 1)
```

Используя функцию построения следующего (или предыдущего) комбинаторного объекта можно реализовать другой подход к построению всех комбинаторных объектов: начнем с минимально возможного в лексикографическом порядке объекта и будем строить следующий объект до тех пор, пока это возможно:

```
S = "0" * n
while S:
    print(S)
    S = NextBinarySequence(S)
```

7.2 Перестановки

Теперь изучим комбинаторные алгоритмы, связанные с перестановками всех чисел от 1 до n. Общеизвестно, что всех перестановок n различных чисел существует $n!$: первый элемент перестановки можно выбрать n способами, второй — $n-1$ и т.д. Алгоритм рекурсивного перебора всех перестановок был изложен выше:

```
def generate(n, prefix):
    if len(prefix) == n:
        print(prefix)
    else:
        for next in range(1, n + 1):
            if next not in prefix:
                generate(n, prefix + [next])
```

Научимся строить следующую и предыдущую в лексикографическом порядке перестановку. Рассмотрим, например, перестановку $(2, 9, 6, 5, 8, 7, 4, 3, 1)$ $(n = 9)$. В соответствии с общим принципом построения следующих в лексикографическом порядке комбинаторных объектов необходимо найти такой самый правый элемент перестановки, который можно увеличить, не модифицируя элементы, стоящие левее. Таким элементом является число 5, так как следующие числа $(8, 7, 4, 3, 1)$ образуют убывающую последовательность, поэтому переставляя только последние пять элементов данной перестановки нельзя получить большую перестановку. Итак, для получения следующей перестановки нужно заменть элемент 5 на больший, не модифицируя предыдущие элементы перестановки.

Элемент 5 нужно заменить на значение 7, поскольку мы переставляем только элементы, следующие за элементом 5, и наименьшим элементом, на который можно заменить 5, увеличив его, является значение 7. Поменяем элементы 5 и 7 местами, получим перестановку $(2, 9, 6, 7, 8, 5, 4, 3, 1)$. Теперь необходимо из элементов, следующих за элементом 7, сделать минимальную в лексикографическом порядке перестановку, для чего их необходимо упорядочить по возрастанию. Поскольку они уже были упорядочены по убыванию, а перестановка элементов 5 и 7 сохраняет это свойство, то достаточно просто все элементы, следующие за увеличенным элементом, развернуть в обратном порядке. Реализуем этот алгоритм в виде функции NextPermutation. Эта функция генерирует следующую в лексикографическом порядке перестановку непосредственно в том списке, в котором хранится переданная ей перестановка

```
def NextPermutation(P):
    i = len(P) - 2
    while i >= 0 and P[i] > P[i + 1]:
        i -= 1
    if i < 0:
        P[:] = []
        return
    j = len(P) - 1
    while P[j] < P[i]:
        j -= 1
    P[i], P[j] = P[j], P[i]
    P[i + 1:] = P[:i:-1]
```

Сначала находится самый правый элемент перестановки P[i], который меньше, чем следующих за ним элемент P[i + 1]. Это делается внутри первого цикла while, который является линейным поиском такого i, что P[i] < P[i + 1] начиная с конца списка. Если такого элемента не найдено, то это означает, что вся перестановка убывает и поэтому является минимальной в лексикографическом порядке. В этом случае алгоритм заменяет переданную перестановку на пустой список и заканчивает работу.

Затем вторым линейным поиском находится такой самый правый элемент P[j], который больше P[i]. Поскольку все элементы, которые стоят правее элемента P[i] убывают, и поскольку есть хотя бы один элемент, который больше, чем P[i] (например, P[i + 1] > P[i]), то линейный поиск обязательно найдет такой элемент, причем это будет наименьший из всех элементов, которые больше P[i]. Затем алгоритм переставляет местами P[i] и P[j] и разворачивает все элементы, идующие после элемента P[i] в обратном порядке при помощи среза.

Аналогичным образом ищется предыдущая в лексикографическом порядке перестановка:

```
def PrevPermutation(P):
    i = len(P) - 2
    while i >= 0 and P[i] < P[i + 1]:
        i -= 1
    if i < 0:
        P[:] = []
        return
    j = len(P) - 1
    while P[j] > P[i]:
        j -= 1
    P[i], P[j] = P[j], P[i]
    P[i + 1:] = P[:i:-1]
```

Научимся опеределять номер перестановки. Все перестановки пронумеруем числами от 0 до $(n-1)!$ в лексикографическом порядке. Вот пример нумерации для $n = 4$:

$$
\begin{aligned}
P_0 &= (1,2,3,4)\\
P_1 &= (1,2,4,3)\\
P_2 &= (1,3,2,4)\\
P_3 &= (1,3,4,2)\\
P_4 &= (1,4,2,3)\\
P_5 &= (1,4,3,2)\\
P_6 &= (2,1,3,4)\\
P_7 &= (2,1,4,3)\\
P_8 &= (2,3,1,4)\\
P_9 &= (2,3,4,1)\\
P_{10} &= (2,4,1,3)\\
P_{11} &= (2,4,3,1)\\
P_{12} &= (3,1,2,4)\\
P_{13} &= (3,1,4,2)\\
P_{14} &= (3,2,1,4)\\
P_{15} &= (3,2,4,1)\\
P_{16} &= (3,4,1,2)\\
P_{17} &= (3,4,2,1)\\
P_{18} &= (4,1,2,3)\\
P_{19} &= (4,1,3,2)\\
P_{20} &= (4,2,1,3)\\
P_{21} &= (4,2,3,1)\\
P_{22} &= (4,3,1,2)\\
P_{23} &= (4,3,2,1)
\end{aligned}
$$

Видим, что все $n!$ перестановок разбиваются на блоки: сначала идут перестановки, которые начинаются с 1, затем начинающиеся с 2 и т.д. В каждом таком блоке $(n-1)!$ различных перестановок, поэтому по первому элементу данной перестановки можно определить, какое число полных блоков идет раньше этой перестановки и сколько перестановок содержится в этих полных блоках. Например, если перестановка начинается с 1, то сначала идет 0 полных блоков, если начинается с 2, то один полный блок и т.д. То есть число пропущенных полных блоков равно количеству элементов в перестановке, которые могут идти раньше первого элемента перестановки.

Определив количество полных блоков, идущих ранее этой перестановки, необходимо также определить номер перестановки внутри блока, в который попала эта перестановка. В этом блоке содержатся все перестановки, начинающиеся с одного и того же элемента, их всего $(n-1)!$. Отбросим первый элемент перестановки и теперь определим номер укороченный на один элемент перестановки среди всех возможных перестановок тех же элементов. Это можно сделать при помощи рекурсии.

Заметим, что в результате отбрасывания первого элемента из перестановки, мы будем рассматривать не перестановки всех чисел от 1 до n, а перестановки некоторого подмножества чисел от 1 до n. То есть решается более общая задача — по данной перестановки из нескольких различных чисел необходимо найти номер данной перестановки среди всех возможных перестановок этих чисел. Например, пусть нужно найти номер перестановки $(6, 4, 1, 9)$. Таких перестановок $4! = 24$, они все разбиваются на 4 блока по $3! = 6$ перестановок, наша перестановка попадает в третий по счету блок, пропуская два полных блока (так как $6 > 1$ и $6 > 4$). То есть к ответу нужно добавить $2 \cdot 6 = 12$ перестановок за счет пропусков двух полных блоков, после чего задача сводится к нахождению номера перестановки $(4, 1, 9)$. Здесь пропускается один блок из $2! = 2$ перестановок (так как $4 > 1$), поэтому к ответу нужно добавить еще 2 и задача сводится к перестановке $(1, 9)$. Далее к ответу ничего не добавляется, так как первый элемент перестановки является минимальным задача сводится к перестановке (9), которая является единственной среди всех перестановок из одного элемента. Если считать, что нумерация перестановок идет с нуля, то ответом будет число $12 + 2 = 14$.

Реализовать алгоритм можно в виде рекурсивной функции:

```
def NumberByPermutation(S):
    n = len(S)
    if n <= 1:
        return 0
    k = 0
    for elem in S:
        if elem < S[0]:
            k += 1
    return math.factorial(n - 1) * k + NumberBySequence(S[1:])
```

Или в виде нерекурсивного алгоритма:

```
def NumberByPermutation(S):
    ans = 0
    n = len(S)
    for i in range(n):
        k = 0
        for elem in S[i + 1:]:
            if elem < S[i]:
                k += 1
        ans += math.factorial(n - 1 - i) * k
    return ans
```

Обратный алгоритм определения перестановки по ее номеру устроен аналогично. Первый элемент перестановки можно определить по частному от деления номера перестановки на $(n - 1)!$, для определения оставшейся части нужно взять остаток от деления номера перестановки на $(n - 1)!$ и свести задачу к нахождения перестановки по номеру, если разрешается использовать только те элементы, которые не были использованы ранее.

Пример рекурсивной функции, возвращающей перестановку по ее номеру.

```
def SequenceByNumber(num, prefix, unused):
```

```
    if len(unused) == 0:
        return prefix
    n = len(unused)
    k = num // math.factorial(n - 1)
    prefix.append(unused[k])
    unused.pop(k)
    num %= math.factorial(n - 1)
    SequenceByNumber(num, prefix, unused)
    return prefix
```

Параметры функции: num — номер искомой перестановки, prefix — уже построенная часть перестановки, которую необходимо добавить к ответу, unused — список еще не использованных элементов перестановки, из которых будет составлена оставшаяся часть перестановки. Например, для определения 100-й по счету перестановки из $n = 5$ элементов функцию необходимо вызывать так:

```
SequenceByNumber(100, [], [1, 2, 3, 4, 5]):
```

Функция определяет по номеру перестановки значение k — индекс элемента из списка unused, который будет идти в начале перестановки, добавляет его в конец списка prefix, удаляет его из списка unused, после чего вызывается рекурсивно для построения оставшейся части перестановки.

Приведем и нерекурсивную реализацию данного алгоритма:

```
def SequenceByNumber(n, num):
    ans = []
    unused = [i for i in range(1, n + 1)]
    while n > 0:
        n -= 1
        k = num // math.factorial(n)
        ans.append(unused[k])
        unused.pop(k)
        num %= math.factorial(n)
    return ans
```

8 Куча

Рассмотрим структуру данных, которая поддерживает следующие операции:

1. Добавить элемент в структуру данных.
2. Извлечь из структуры данных наибольший (вариант - наименьший) элемент. Извлеченный элемент удаляется из структуры.

При этом в структуре могут храниться одинаковые элементы.

Если реализовать такую структуру на базе списка, то добавлять элементы можно в конец списка за $O(1)$. Но поиск наибольшего элемента будет занимать $O(n)$, если n — число элементов в списке, так как придется просматривать все элементы списка и выбирать из них наибольший.

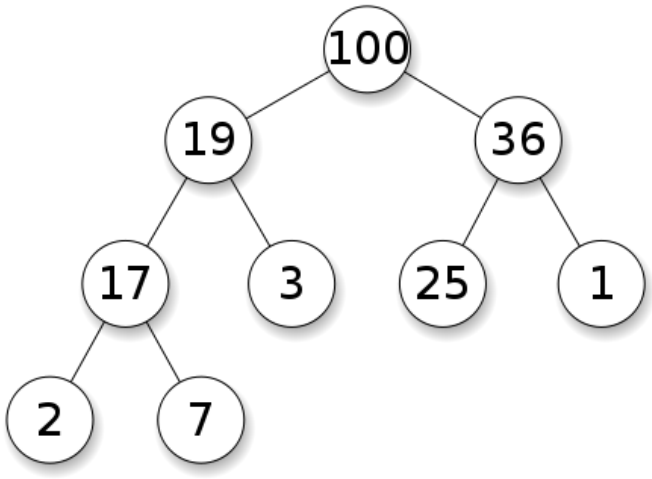
Если же хранить элементы в списке упорядочив их по неубыванию, то извлечение наибольшего будет занимать $O(1)$, но добавление элемента в список — $O(n)$, так как придется сдвигать элементы, уже находящиеся в списке.

Специальная структура данных “Куча” (англ. heap)позволяет эти операции выполнять за $O(logn)$.

В куче элементы хранятся в виде двоичного дерева, то есть у элементов есть два потомка — левый и правый. В вершине кучи находится один элемент, у него два потомка на следующем уровне, у них, в свою очередь, по два потомка на третьем уровне (итого 4 элемента на третьем уровне) и т. д. Уровни заполняются в порядке увеличения номера уровня, а сам уровень заполняется слева направо. У элементов последнего уровня нет ни одного потомка, возможно, что и у некоторых элементов предпоследнего уровня нет потомков. Также в куче может быть один элемент, у которого только один потомок (левый).

При этом для элементов кучи верно следующее свойство: каждый из элементов кучи большее или равен всех своих потомков. В частности это означает, что в вершине кучи хранится наибольший элемент.

На картинке приведен пример правильной кучи из 9 элементов.



Удобно элементы кучи хранить в списке, начиная с корневого элемента. Для простоты нумерации пропустим нулевой элемент списка, то есть вершина кучи будет храниться в элементе списка с индексом 1. Остальные элементы кучи хранятся подряд в элементах списка с индексами 2, 3, 4 и т. д. То есть для примера выше:

Heap[1] == 100
Heap[2] == 19
Heap[3] == 36
Heap[4] == 17
Heap[5] == 3
Heap[6] == 25

```
Heap[7] == 1
Heap[8] == 2
Heap[9] == 7
```

Легко видеть, что у элемента H[i] левым потомком является элемент H[2*i], а правым потомком - элемент H[2*i+1]. А родителем элемента H[i] является элемент H[i//2].

Напишем реализацию структуры данных "куча" в виде класса Heap, в котором сами данные будут храниться в виде закрытого поля __data. Метод size класса Heap будет возвращать размер кучи, то есть длину списка __data минус 1, так как список __data содержит один фиктивный элемент. Метод top будет возвращать первый (наибольший) элемент кучи, то есть __data[1].

```
class Heap():
    __data = [None]

    def size(self):
        return len(self.__data) - 1

    def top(self):
        return self.__data[1]
```

Добавление элемента в кучу

Элемент добавляется в кучу следующим образом. Сначала он добавляется в самый конец кучи, то есть становится последним элементов (это можно сделать при помощи метода append списка). При этом возможно нарушение главного свойства кучи (каждый элемент больше своих потомков). Свойство могло нарушиться для элемента, который является родителем добавляемого. В этом случае нужно поменять элемент с его родителем. Процесс нужно повторять до тех пор, пока условие нарушается, то есть у добавленного элемента есть родитель (то есть элемент не корневой) и этот родитель меньше добавляемого. То есть добавляемый элемент «поднимается» вверх к вершине кучи, пока не займет надлежащее место.

Реализация соответствующего алгоритма (чуть более оптимизированная) приведена ниже:

```
class Heap():
    def add(self, x):
        H = self.__data
        H.append(x)
        i = len(H) - 1
        while i > 1 and x > H[i // 2]:
            H[i] = H[i // 2]
            i //= 2
        H[i] = x
```

В метод add передается добавляемый элемент x. Для упрощения читаемости кода обозначим через H ссылку на список, в котором хранится куча, т.е. self.__data. Сначала в конец кучи H добавляется новый элемент,

переменной i присваивается индекс добавленного элемента. Затем все предки добавленного элемента должны сдвинуться вниз, если они были меньше добавленного элемента. Это реализовано в цикле while, в конце которого значение i меняется на родителя текущего элемента. Цикл останавливается на том значении i, у которого нет родителя, или если родитель элемента с индексом i больше, чем x. Поэтому после окончания цикла на место элемента i записывается значение elem.

Удаление элемента из кучи

Из кучи можно удалить наибольший элемент, то есть тот, который хранится в вершине куче. На его место нужно поставить какой-нибудь элемент кучи. Поставим последний элемент кучи, удалив его из конца. Теперь в вершине кучи может нарушиться свойство кучи, значит, верхний элемент нужно сместить вниз, обменяв его с одним из своих потомков. При этом из двух потомков нужно выбрать наибольший и если этот наибольший потомок больше стоящего в вершине кучи, обменяем их местами.

Тем самым элемент, который был взят снизу кучи, спустится на один уровень вниз. Будем дальше опускать этот элемент до тех пор, пока оба его потомка не станут меньше его (или у него не будет потомков, также необходимо аккуратно обработать случай одного потомка.

```
class Heap:
    def pop(self):
        H = self.__data
        if self.size() == 1:
            return H.pop()
        result = H[1]
        H[1] = H.pop()
        i = 1
        while (2 * i + 1 < len(H) and
               H[i] < max(H[2 * i:2 * i + 2]))):
            if H[2 * i] > H[2 * i + 1]:
                j = 2 * i
            else:
                j = 2 * i + 1
            H[i], H[j] = H[j], H[i]
            i = j
        if 2 * i == len(H) - 1 and H[i] < H[2 * i]:
            H[i], H[2 * i] = H[2 * i], H[i]
        return result
```

В этом примере сохраняется значение на вершине кучи в переменной result, затем последний элемент удаляется из кучи и ставится на вершину кучи. Отдельно обрабатывается случай, когда куча состояла ровно из одного элемента, т. е. после удаления она становится пустой. Далее в основном цикле элемент опускается вниз. Цикл продолжается пока у элемента два потомка и хотя бы один из потомков больше текущего элемента. В этом случае элемент меняется местами с наибольшим из потомков и цикл повторяется заново.

После окончания цикла отдельно обрабатывается случай, когда у элемента ровно один потомок (нет правого потомка) и единственный левый потомок больше данного элемента, в этом случае необходимо провести еще один обмен.

Сложность операций с кучей

Все операции с кучей (добавление и удаление элемента) требуют $O(h)$ операций, где h - высота кучи. Пусть в куче n элементов, а ее высота равна h. Тогда наибольшее число элементов, которое может быть в куче высоты h есть $1 + 2 + 4 + ... + 2^{h-1} = 2^h - 1$.

Таким образом, $2^{h-1} \leq n < 2^h$, откуда видно, что h примерно равно двоичному логарифму числа n, то есть сложность всех операций с кучей - $O(\log n)$.

Применения кучи

Одно из наиболее известных применений кучи - сортировка при помощи кучи или пирамидальная сортировка (англ. heapsort). В данной сортировки из элементов списка снача строится куча, потом элементы по одному удаляются из кучи - сначала наибольший элемент, потом - наибольший из оставшихся и т. д. При этом кучу можно хранить там же, где хранятся элементы самого списка, тем самым пирамидальная сортировка имеет сложность $O(n \log n)$, но при этом не требует дополнительной памяти (как сортировка слиянием) и не является вероятностной (как быстрая сортировка Хоара).

Также при помощи кучи можно организовать структуру данных «очередь с приоритетами». В очереди каждому элементу сопоставляется приоритет - некоторое целое число. При удалении элемента из очереди удаляется не тот элемент, который был добавлен раньше (как в обычной очереди), а элемент с наибольшим приоритетом. То есть элементы в очереди с приоритетами можно хранить в куче, сравнивая их при этом по приоритету.

В очереди с приоритетами также есть операция изменения приоритета элемента. Для этого реализованы две функции - повышения и понижения приоритета. При повышении приоритета элемент поднимается вверх, поэтому эта функция реализована аналогично операции добавления элемента. При понижении приоритета элемент спускается вниз, как в операции удаления элемента.

9 Хеширование

9.1 Полиномиальное хеширование строк

Использование технологии хеширование позволяет решать некоторые задачи, связанные со строками. Для начала рассмотрим алгоритм вычисления хеш-функции для строк.

Для начала определим хеш-функцию от строки из одного символа. Каждому возможному символу строки сопопоставим уникальное целое положительное число — значение хеш-функции от этого символа. Например, можно взять ASCII-код или Unicode символа, или если строка, например,

состоит только из строчных латинских букв, можно считать, что h('a') = 1, h('b') = 2, ..., h('z') = 26. Пустой строке будет соответствовать нулевое значение хеш-функции, поэтому не следует использовать число 0 для значения хеш-функции от одного символа.

Для строк длины 2 желательно выбрать такой алгоритм вычисления хеш-функции, который обладал бы следующим свойством:

1. Значение хеш-функции для всех строк длины 1 и 2 были бы различными.

2. Использовались бы значения хеш-функции для одного символа, определенные ранее.

3. Алгоритм легко обобщается на строки большей длины.

Простое решение — для строки a_0a_1 значение хеш-функции вычислить по формуле $h(a_0a_1) = h(a_0)b + h(a_1)$. Если при этом выбрать значение b большим, чем максимальное возможное значение хеш-функции от одного символа (например, если используются только латинские буквы, то можно взять $b = 27$), то тогда у всех строк длины 2 значения хеш-функции будут различными.

Для строк длины три можно определить значение хеш-функции следующим образом: $h(a_0a_1a_2) = h(a_0)b^2 + h(a_1)b + h(a_2)$ и т.д. Например, если взять $b = 27$, то хеш-функция от строки 'one' будет равна $hash$('one') = $hash$('o')$b^2 + hash$('n')$b + hash$('e') = $15 \cdot 27^2 + 14 \cdot 27 + 5 = 11318$.

Соответственно, для строки длины n значение хеш-функции можно определить так: $h(a_0a_1a_2 \dots a_{n-1}) = h(a_0)b^{n-1} + h(a_1)b^{n-2} + ... + h(a_{n-2})b + h(a_{n-1})$. Это значение также равно значению числа $\overline{a_0a_1a_2 \dots a_{n-1}}$ в системе счисления с основанием b, то есть числом, записанного символами $h(a_0a_1a_2 \dots a_{n-1}$, если считать, что один символ соответствует одной цифре в системе счисления с основанием b, а значением цифры является значение хеш-функции от этого символа. Также можно рассмотреть это, как значение многочлена степени $n - 1$ с коэффициентами a_0, a_1, ..., a_{n-1}, вычисленное при $x = b$.

Но при длинной строке результатом вычисления по этой формуле будет очень большое число (длина которого будет порядка длины строки), поэтому использовать такой хеш для решения задач нельзя (так как основным смыслом хеш-функции является сопоставление большому объекту некоторого относительно небольшого числа, поэтому в качестве значения хеш-функции возьмем остаток от деления полученного числа на некоторое большое число M. В качестве значения M рекомендуется выбирать большое простое число, также часто в качестве значения M выбирают степень числа 2, равную размеру целочисленного типа данных, например, 2^{32} или 2^{64}. Обязательное требование к числу M — оно должно быть взаимно простым с числом b. Итак, определим значение хеш-функции от строки:

$$h(a_0a_1a_2 \dots a_{n-1}) = (h(a_0)b^{n-1} + h(a_1)b^{n-2} + ... + h(a_{n-2})b + h(a_{n-1})) \bmod M$$

Значение определенной таким образом хеш-функции удобно вычислять, как и значение многочлена по схеме Горнера пользуясь свойством:

$$h(a_0a_1a_2\dots a_{n-1}) =$$
$$= \big((h(a_0)b^{n-2} + h(a_1)b^{n-3} + ... + h(a_{n-2}))b + h(a_{n-1})\big) \bmod M =$$
$$= (h(a_0a_1a_2\dots a_{n-2})b + h(a_{n-1})) \bmod M$$

То есть чтобы вычислить хеш-функцию от строки $a_0a_1a_2\dots a_{n-1}$ необходимо взять вычислить хеш-функцию от строки без одного последнего символа $a_0a_1a_2\dots a_{n-2}$, умножить результат на b, добавить значение последнего символа a_{n-1} и взять остаток от деления результата на M. Например, это можно сделать при помощи следующей функции за время, пропорциональное длине строки:

```
b = 27
M = 10 ** 17 + 3

def Hash(S):
    result = 0
    for char in S:
        result *= b
        result += ord(char) - ord('a') + 1
        result %= M
    return result
```

Если же в конец некоторой данной строки S добавить строку T длиной в m символов, то число, соответствующее хеш-функции строки S необходимо сдвинуть на m символов влево, что соответствует умножению на b^m, после чего добавить хеш-функцию строки T.

$$h(ST) = (h(S)b^m + h(T)) \bmod M$$

То есть легко вычислить хеш-функцию от конкатенации двух строк S и T, если известны хеш-функции от данных строк.

С другой стороны, пусть известна хеш-функция $h(ST)$ и хеш-функция от ее префикса $h(S)$. Тогда можно вычислить хеш-функцию от строки T:

$$h(T) = (h(ST) - h(S)b^m) \bmod M$$

Таким образом, если дана строка S и необходимо вычислить хеш-функцию от среза этой строки $S[i:j]$ (напомним, что срез $S[i:j]$ состоит из символов строки с индексами от i до $j-1$ и имеет длину $j-i$), то это можно сделать, зная значения хеш-функции от префиксов строки длины i и j:

$$h(S[i:j]) = \big(h(S[:j]) - h(S[:i])b^{j-i}\big) \bmod M$$

То есть если известны значения хеш-функции на всех префиксах данной строки S, то значение хеш-функции от любой ее подстроки можно вычислить за $O(1)$, если также известно значение $b^m \bmod M$. Это можно сделать при помощи следующей функции:

```
def Hash(i, j, Prefix, Power):
    return (Prefix[j] - Prefix[i] * Power[j - i]) mod M
```

Эта функция получает на вход следующие параметры: границы среза i и j некоторой строки S. Сама строка S не передается в функцию, вместо нее передаются два списка. В списке Prefix хранятся значения хеш-функции для всех префиксов строки S, то есть значение Prefix[i] равно значению хеш-функции для префикса S[i] строки S. В списке Power хранятся значения степеней числа b по модулю M, то есть Power[i] $= b^i \bmod M$.

Значения списка Prefix и Power необходимо вычислисть один раз в начале, выполнив так называемый "предподсчет", то есть предварительную подготовку, необходимую для реализации алгоритма. Для выполнения предподсчета используются две функции: функция CalcPrefix получает на вход строку S и возвращает список Prefix длины $n + 1$ (если n — длина строки S), содержащий значение хеш-функции для всех префиксов строки S, и функция CalcPower, получающая на вход число n и возвращающая список остатков от деления на M всех степеней числа b от 0 до n.

```
b = 27
M = 10 ** 17 + 3

def CalcPrefix(S):
    n = len(S)
    Prefix = [0] * (n + 1)
    for i in range(n):
        Prefix[i + 1] = (Prefix[i] * b + h(S[i])) % M
    return Prefix

def CalcPower(n):
    Power = [1] * (n + 1)
    for i in range(n):
        Power[i + 1] = Power[i] * b % M
    return Power

def h(char):
    return ord(char) - ord('a') + 1
```

Для выполенения предподсчета для некоторой строки S даные функции необходимо вызывать так:

```
Prefix = CalcPrefix(S)
Power = CalcPower(len(S))
```

Предподсчет выполняется за время $O(n)$, но после однократно выполненного предподсчета функция Hash позволяет за $O(1)$ вычислять значение хеш-функции для любой подстроки данной строки, что позволяет, например, быстро проверять на равенство две подстроки.

9.1.1 Поиск наибольшей общей подстроки

Пусть даны две строки $S = s_0 s_1 \dots s_{n-1}$ длины n и $T = t_0 t_1 \dots t_{m-1}$ длины m. Необходимо найти наибольшую общую подстроку этих строк, то есть такую подстроку, которая одновременно встречается и в строке S, и в строке T. В отличии от общей подпоследовательности, символы подстроки должны идти подряд.

Прежде всего научимся проверять, если у двух данных строк общая подстрока длины k. Сравнение всех подстрок длины k двух данных строк занимает много времени, но если вместо строк сравнивать значения хеш-функций от этих подстрок, которые после предобработки можно вычислять быстро, то это можно сделать существенно быстрее. Вычислим значения хеш-функции для всех подстрок длины k первой подстроки, добавляя результат в структуру данных S типа "множество" (set), позволяющую быстро проверять наличие элемента в ней. После этого переберем все подстроки длины k второй строки, вычислим значение хеш-функции для каждой такой подстроки и проверим наличие такого значения в множестве set. Если для какой-то подстроки длины k второй строки вычисленное значение хеш-функции содержится в множестве S, то у данных строк есть общая подстрока длины k и функция возвращает True.

Реализуем эту часть алгоритма в виде функции IsCommonSubstring. Функция получает четыре параметра: длину искомой подстроки k, два списка Prefix1 и Prefix2 со значениями хеш-функций на префиксах двух данных срок (сами строки не нужны, для вычисления хеш-функции для любой подстроки данной строки нужно знать только значение хеш-функции на всех префиксах строки), и список Power со значениями степеней числа b для быстрого вычисления хеш-функции.

```
def IsCommonSubstring(k, Prefix1, Prefix2, Power):
    S = set()
    for i in range(k, len(Prefix1)):
        S.add(Hash(i - k, i, Prefix1, Power))
    for i in range(k, len(Prefix2)):
        if Hash(i - k, i, Prefix2, Power) in S:
            return True
    return False
```

Для нахождения длины наибольшей общей подстроки двух строк S и T воспользуемся двоичным поиском по ответу: если у двух строк есть общая подстрока длины k, то есть и общая подстрока длины $k-1$, поэтому наибольшее значение k, при котором у двух строк есть общая подстрока длины k можно найти двоичным поиском. Сначала предподсчитаем значения хеш-функции на префиксах двух строк и посчитаем значения степений числа b:

```
Prefix1 = CalcPrefix(S)
Prefix2 = CalcPrefix(T)
Power = CalcPower(max(len(S), len(T)))
```

В качестве начального значения для левой границы двоичного поиска выберем значение left = 0, так как можно считать, что у двух строк всегда есть общая подстрока длины 0. В качестве правой границы можно выбрать right = min(len(S), len(T)) + 1, так как такое значение больше длины хотя бы одной из двух данных строк, поэтому общей подстроки такой длины быть не может. Затем будем сдвигать границы left и right сохраняя инвариант: у данных строк есть общая подстрока длины left, но нет общей подстроки длины right. После окончания двоичного поиска значение right будет равно left + 1, поэтому длина наибольшей общей подстроки будет равна значение left.

```
left = 0
right = min(len(S), len(T)) + 1
while right > left + 1:
    middle = (left + right) // 2
    if IsCommonSubstring(middle, Prefix1, Prefix2, Power):
        left = middle
    else:
        right = middle
print(left)
```

10 Алгоритмы на графах

10.1 Основные понятия теории графов

Многие объекты, возникающие в жизни человека, могут быть смоделированы (представлены в памяти компьютера) при помощи графов. Например, транспортные схемы (схема метрополитена и т. д.) изображают в виде станций, соединенных линиями. В терминах графов станции называются вершинами графа а линии – ребрами.

Графом называется конечное множество *вершин* и множество *ребер*. Каждому ребру сопоставлены две вершины — концы ребра.

Бывают различные варианты определения графа. В данном определении концы у каждого ребра равноправны. В этом случае нет разницы где начало, а где — конец ребра. Но, например, в транспортных сетях бывают случаи одностороннего движения по ребру, тогда говорят об ориентированном графе — графе, у ребер которого одна вершина считается начальной, а другая — конечной.

Если некоторое ребро u соединяет две вершины A и B графа, то говорят, что ребро u инцидентно вершинам A и B, а вершины в свою очередь инцидентны ребру u. Вершины, соединенные ребром, называются смежными.

Ребра называются кратными, если они соединяют одну и ту же пару вершин (а в случае ориентированного графа — если у них совпадают начала и концы). Ребро называется петлей, если у него совпадают начало и конец. Во многих задачах кратные ребра и петли не представляют интереса, поэтому могут рассматриваться только графы без петель и кратных ребер. Такие графы называю простыми.

Степенью вершины в неориентированном графе называется число инцидентных данной вершине ребер (при этом петля считается два раза, то есть степень — это количество “концов” ребер, входящих в вершину). Довольно очевидно, что сумма степеней всех вершин равна удвоенному числу ребер в графе. Отсюда можно посчитать максимальное число ребер в простом графе — если у графа n вершин, то степень каждой из них равна $n-1$, а, значит, число ребер есть $n(n-1)/2$. Граф, в котором любые две вершины соединены одним ребром, называется полным графом.

Также легко заметить следующий факт: в любом графе число вершин нечетной степени четно. Этот факт называется “леммой о рукопожатиях” — в любой компании число людей, сделавших нечетное число рукопожатий всегда четно.

Пути, циклы, компоненты связности

Путем на графе называется последовательность ребер $u_1, u_2, \ldots, u_k$, в которой конец одного ребра является началом следующего ребра. Начало первого ребра называется началом пути, конец последнего ребра — концом пути. Если начало и конец пути совпадают, то такой путь называется циклом.

Путь, который проходит через каждую вершину не более одного раза называется простым путем. Аналогично определяется простой цикл.

Граф называется связным, если между любыми двумя его вершинами есть путь. Если граф несвязный, то его можно разбить на несколько частей (подграфов), каждая из которых будет связной. Такие части называются компонентами связности. Возможно, что некоторые компоненты связности будут состоять всего лишь из одной вершины.

Понятно, что в графе из n вершин может быть от 1 до n компонент связности.

Деревья

Рассмотрим связный граф из n вершин. Какое минимальное число ребер может быть в нем?

Несложно построить пример графа, содержащего $n-1$ ребро - например, можно взять одну вершину графа и соединить ее с остальными вершинами при помощи $n-1$ ребра. Нетрудно также понять, что в таком графе не должно быть простых циклов (иначе в простом цикле можно выбросить одно ребро и граф останется связным). Такие графы называются деревьями.

Определение: *деревом* называется связный граф не содержащий простых циклов.

Нетрудно видеть, что в дереве нельзя удалить ни одного ребра, чтобы граф остался связным. Поэтому дерево является минимальным связным графом.

Основным свойством дерева является следующая теорема:

Дерево из n вершин содержит $n-1$ ребро.

Эту теорему можно доказать математической индукцией по n, используя лемму о висячей вершине - в каждом дереве есть хотя бы одна вершина степени 1. Эту вершину можно удалить и далее применить предположение индукции для меньшего числа вершин.

Можно показать, что эквивалентны следующие определения дерева:

1. Деревом называется связный граф, не содержащий простых циклов.
2. Деревом называется связный граф, содержащий n вершин и $n-1$ ребро.
3. Деревом называется связный граф, который при удалении любого ребра перестает быть связным.
4. Деревом называется граф, в котором любые две вершины соединены ровно одним простым путем.

Очень часто в дереве выделяется одна вершина, называемая корнем дерева, дерево с выделенным корнем называют корневым или подвешенным деревом. Примером такого дерева является генеалогическое дерево.

10.2 Способы представления графов в памяти

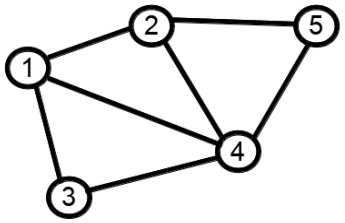
Представление графов в памяти — это способ хранения модели графа (то есть сведений о том, какие вершины соединены ребрами), позволяющий отвечать на следующие вопросы:

1. Для двух данных вершин u и b проверить, соединены ли вершины u и v ребром.

2. Перебрать все ребра, исходящие из данной вершины u.

При этом способ хранения графов в памяти должен учитывать возможности работы с ориентированными и неориентированными графами. По умолчанию будем предполагать, что хранимый граф является простым, но можно рассмотреть вопрос и о представлении графов с петлями и кратными ребрами.

Рассмотрим следующий граф:



При представлении графа матрицей смежности информация о ребрах графа хранится в квадратной матрице (двумерном списке), где элемент A[i][j] равен 1, если ребра i и j соединены ребром и равен 0 в противном случае. Для данного примера матрица смежности будет выглядеть так:

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	1
3	1	0	0	1	0
4	1	1	1	0	1
5	0	1	0	1	0

Если граф неориентированный, то матрица смежности всегда симметрична относительно главной диагонали.

При использовании матрицы смежности удобно проверять соединены ли две вершины ребром — это просмотр одного элемента матрицы A[i][j], но сложнее перебирать все ребра, исходящие из данной вершины (для этого необходимо перебрать все оставшиеся вершины и проверить, соединены ли они ребром). Также матрица смежности требует $O(n^2)$ памяти и может оказаться неэффективным способом хранения дерева или разреженных графов.

При представлении графа списками смежности для каждой вершины i хранится список W[i] смежных с ней вершин. Для рассмотренного примера:

```
W[1] = [2, 3, 4]
W[2] = [1, 4, 5]
W[3] = [1, 4]
```

```
W[4] = [1, 2, 3, 5]
W[5] = [2, 4]
```

Таким образом, весь граф можно представить одним двумерным списком:

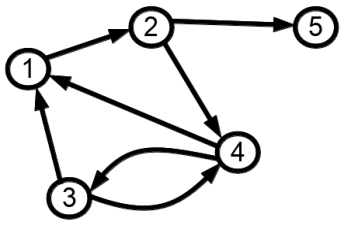
```
W = [[], [2, 3, 4], [1, 4, 5], [1, 4], [1, 2, 3, 5], [2, 4]]
```

Поскольку нумерация в нашем примере начинается с 0, то к списку добавлен еще один фиктивный элемент W[0].

В таком способе удобно перебирать ребра, выходящие из вершины i (это просто список W[i]), но сложно проверять наличие ребра между вершинами i и j — для этого необходимо проверить, содержится ли число j в списке W[i]. Но в языке Питон можно эту часть сделать более эффективной, если заменить списки на множества - тогда проверка существования ребра между двумя вершинами также будет выполняться за $O(1)$.

При помощи матриц смежности и списков смежности можно представлять и неориентированные графы. В случае матрицы смежности A[i][j] будет равно 1, если есть ребро, начинающееся в вершине i и заканчивающееся в вершине j. В случае списков смежности наличие ребра из вершины i в вершину j означает, что в списке W[i] есть число j.

Например, для такого графа:



Матрица смежностей будет следующей:

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	1
3	1	0	0	1	0
4	1	0	1	0	0
5	0	0	0	0	0

А списки смежности будут следующими:

```
W[1] = [2]
W[2] = [4, 5]
W[3] = [1, 4]
W[4] = [1, 3]
W[5] = []
```

Приведем код считывания графа. Будем считать, что граф задается следующим образом: в первой строке записано число вершин n и число ребер m графа. Далее записаны m строк, содержащих по два числа - номера начальной и конечной вершины ребра. Например, первый граф из первого примера можно задать так:

```
5 7
1 2
2 5
5 4
4 2
1 4
1 3
3 4
```

Пример заполнения матрицы смежности. Матрица создается размером $(n+1) \times (n+1)$, так как используется нумерация с единицы:

```
n, m = map(int, input().split())
A = [[0] * (n + 1) for i in range(n + 1)]
for i in range(m):
    u, v = map(int, input().split())
    A[u][v] = 1
    # A[v][u] = 1
```

Пример заполнения списков смежности, используются множества вместо списков:

```
n, m = map(int, input().split())
W = [set() for i in range(n + 1)]
for i in range(m):
    u, v = map(int, input().split())
    W[u].add(v)
    # W[v].add(u)
```

Здесь также используется нумерация с единицы. В обоих примерах закомментированная строчка нужна в случае неориентированного графа, тогда для каждого считанного ребра из u в v необходимо добавить обратное ребро из v в u.

Взвешенные графы

Очень часто рассматриваются графы, в которых каждому ребру приписана некоторая числовая характеристика — вес. Вес может означать длину дороги или стоимость проезда по данному маршруту. Соответствующие графы называются взвешенными.

При представлении графа матрицей смежности вес ребра можно хранить в матрице, то есть A[i][j] в данном случае будет равно весу ребра из i в j. При этом при отсутствии ребра можно хранить специальное значение, например, None. Во многих задачах удобно при отсутствии ребра хранить очень большое число, в этом случае отсутствие ребра аналогично наличию ребра очень большой стоимости.

При представлении графа списками смежности удобней всего множества смежных вершин использовать словарь, где ключом будет номер вершины, являющейся концом ребра, а значением — вес данного ребра. Тогда считывание графа можно реализовать так:

```
n, m = map(int, input().split())
W = [dict() for i in range(n + 1)]
```

```
for i in range(m):
    u, v, weight = map(int, input().split())
    W[u][w] = weight
```

Тогда для проверки наличия ребра между вершинами u и v можно использовать условие if v in W[u], а вес ребра будет храниться в W[u][v].

10.3 Поиск в ширину

Алгоритм поиска в ширину (англ. breadth-first search, BFS) позволяет найти кратчайшие пути из одной вершины невзвешенного (ориентированного или неориентированного) графа до всех остальных вершин. Под кратчайшим путем подразумевается путь, содержащий наименьшее число ребер.

Алгоритм построен на простой идее — пусть до какой-то вершины u найдено кратчайшее расстояние и оно равно d, а до вершины v кратчайшее расстояние не меньше, чем d. Тогда если вершины u и v - смежны, то кратчайшее расстояние до вершины v равно $d + 1$.

Через $d[i]$ будем обозначать кратчайшее расстояние до вершины i. Пусть начальная вершина имеет номер s, тогда $d[s] = 0$. Для всех вершин смежных с s расстояние равно 1, для вершин, смежных с теми, до которых расстояние равно 1, расстояние равно 2 (если только оно не равно 0 или 1) и т.д.

Таким образом, организовать процесс вычисления кратчайших расстояний до вершин можно следующим образом. Для каждой вершины в массиве d будем хранить кратчайшее расстояние до этой вершины, если же расстояние неизвестно — будем хранить значение None. В самом начале расстояние до всех вершин равно None, кроме начальной вершины, до которой расстояние равно 0. Затем перебираем все вершины, до которых расстояние равно 0, перебираем смежные с ними вершины и для них записываем расстояние равное 1. Затем перебираем все вершины, до которых расстояние равно 1, перебираем их соседей, записываем для них расстояние, равное 2 (если оно до этого было равно None). Затем перебираем вершины, до которых расстояние было равно 2 и тем самым определяем вершины, до которых расстояние равно 3 и т.д. Этот цикл можно повторять либо пока обнаруживаются новые вершины на очередном шаге, либо $n - 1$ раз (где n — число вершин в графе), так как длина кратчайшего пути в графе не может превосходить $n - 1$.

Такая реализация алгоритма будет неэффективной, если на каждом шаге перебирать все вершины, отбирая те, которые были обнаружены на последнем шаге. Для эффективной реализации следует использовать очередь.

В очередь будут закладываться вершины после того, как до них будет определено кратчайшее расстояние. То есть очередь будет содержать вершины, которые были “обнаружены” алгоритмом, но не были рассмотрены исходящие ребра из этих вершин. Можно также сказать, что это очередь на “обработку” вершин.

Из очереди последовательно извлекаются вершины, рассматриваются все исходящие из них ребра. Если ребро ведет в необнаруженную до этого вершину, то есть расстояние до новой вершины не определено, то оно устанавливается равным на единицу больше, чем расстояние до обрабатываемой вершины, а новая вершина добавляется в конец очереди.

Таким образом, если из очереди извлечена вершина с расстоянием d,

то в конец очереди будут добавляться вершины с расстоянием $d+1$, то есть в любой момент исполнения алгоритма очередь состоит из вершин, удаленных на расстояние d, за которыми следуют вершины, удаленные на расстояние $d+1$.

Запишем алгоритм поиска в ширину.

```
D = [None] * (n + 1)
D[start] = 0
Q = [start]
Qstart = 0
while Qstart < len(Q):
    u = Q[Qstart]
    Qstart += 1
    for v in V[u]:
        if D[v] is None:
            D[v] = D[u] + 1
            Q.append(v)
```

В этом алгоритме n — число вершин в графе, вершины пронумерованны от 1 до n. Номер начальной вершины (от которой ищутся пути) хранится в переменной start. Q — список, используемый для хранения очереди элементов, Qstart — первый элемент очереди. Добавление новой вершины в конец очереди — это вызов метода append для списка, удаление вершины из начала очереди — это увеличение Qstart на 1 (при этом первый элемент в очереди хранится в Q[Qstart]).

В самом начале в очередь добавляется только один элемент start, для которого в самом начале определено расстояние D[start] = 0 (для всех остальных элементов расстояние не определено). Цикл продолжается пока очередь не пуста, что проверяется условием Qstart < len(Q). В цикле из очереди удаляется первый элемент u. Затем перебираются все смежные с ним вершины v. Если вершина v не была обнаружена ранее, что проверяется при помощи условия D[v] is None, то расстояние до вершины v устанавливается равным расстоянию до вершины u, увеличенному на 1, затем вершина v добавляется в конец очереди.

Если в графе содержится n вершин и m ребер, то сложность такого алгоритма равна $O(n+m)$, так как алгоритму необходимо пройти по всем ребрам. Если граф хранится при помощи матрицы смежности, то сложность алгоритма равна $O(n^2)$, так как внутренний цикл перебора всех смежных вершин будет содержать n шагов для каждой обработанной вершины графа.

Записанный алгоритм находит только кратчайшие расстояния до каждой из вершин графа. Чтобы найти кратчайший путь необходимо для каждой вершины хранить все ребра, по которым совершалось открытие новых вершин, то есть для каждой вершины необходимо хранить номер её предшественника — вершины, из которой была открыта данная вершина. Все сохраненные ребра вместе образуют дерево кратчайших путей. Чтобы построить кратчайший путь из начальной вершины до какой-то другой достижимой из нее вершины, необходимо взять путь в этом дереве, соединяющий эти две вершины.

Предшественников будем хранить в списке:

```
Prev = [None] * (n + 1)
```

Значение Prev[i] есть номер предшествующей вершине i кратчайшего пути из вершины start. То есть чтобы построить кратчайший путь до вершины i необходимо построить кратчайший путь до вершины Prev[i], а затем добавить к нему ребро из Prev[i] в i.

При обнаружении новой вершины v в записанном алгоритме необходимо пометить, что данная вершина была достигнута проходом по ребру из вершины u, то есть предшественником вершины v является вершина u:

```
Prev[v] = u
```

Для восстановления ответа (кратчайшего пути от вершины start до некоторой вершины finish) заведем список Ans для сохранения ответа, затем будет последовательно переходить от каждой вершины к ее предшественнику, пока не дойдем до значения None, то есть отсутствия предшественника:

```
Ans = []
curr = finish
while curr is not None:
    Ans.append(curr)
    curr = Prev[curr]
```

В итоге список Ans будет хранить вершины на кратчайшем пути от start до finish, записанные в обратном порядке.

10.4 Обход в глубину

Алгоритм поиска (или обхода) в глубину (англ. depth-first search, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины.

Отличие поиска в глубину от поиска в ширину заключается в том, что (в случае неориентированного графа) результатом алгоритма поиска в глубину является некоторый маршрут, следуя которому можно обойти последовательно все вершины графа, доступные из начальной вершины. Этим он принципиально отличается от поиска в ширину, где одновременно обрабатывается множество вершин, в поиске в глубину в каждый момент исполнения алгоритма обрабатывается только одна вершина. С другой стороны, поиск в глубину не находит кратчайших путей, зато он применим в ситуациях, когда граф неизвестен целиком, а изучается, например, роботом, который ищет выход из лабиринта.

Если же граф ориентированный, то обход в глубину строит дерево путей из начальной вершины во все доступные из неё.

Обход в глубину можно представить себе следующим образом. Пусть исследователь находится в некотором лабиринте (графе) и он хочет обойти весь лабиринт (посетить все доступные вершины в графе). Исследователь находится в некоторой вершине и видит ребра, исходящие из этой вершины. Очевидная последовательность действий исследователя такая:

1. Пойти в какую-нибудь смежную вершину.
2. Обойти все, что доступно из этой вершины.
3. Вернуться в начальную вершину.

4. Повторить алгоритм для всех остальных вершин, смежных из начальной.

Видим, что алгоритм является рекурсивным — для обхода всего графа нужно переместиться в соседнюю вершину, после чего повторить для этой вершины алгоритм обхода. Но возникает проблема зацикливания: если из вершины A можно перейти в вершину B, то из вершины B можно перейти в вершину A и рекурсия будет бесконечной. Для борьбы с рекурсией нужно применить очень простую идею — исследователь не должен идти в ту вершину, в которой он уже был раньше, то есть которая не представляет для него интерес (считаем, что интерес для исследователя представляют только вершины, в которых он не был ранее). Итак, уточненный алгоритм может выглядеть следующим образом:

1. Пойти в какую-нибудь смежную вершину, не посещенную ранее.
2. Запустить из этой вершины алгоритм обхода в глубину
3. Вернуться в начальную вершину.
4. Повторить пункты 1-3 для всех не посещенных ранее смежных вершин.

Для реализации алгоритма понадобится отмечать, в каких вершинах был исследователь, а в каких — нет. Пометку будем делать в списке Visited, где Visited[i] == True для посещенных вершин, и Visited[i] == False для непосещенных. Пометка “о посещении вершиных” ставится при заходе в эту вершину.

Поскольку целью обхода в глубину зачастую является построение дерева обхода в глубину, то сразу же будем хранить предшественника для каждой вершины.

Алгоритм обхода в глубину оформим в виде рекурсивной функции DFS(start), где start — номер вершины, из которой запускается обход.

```
Visited = [False] * (n + 1)
Prev = [None] * (n + 1)
def DFS(start):
    Visited[start] = True
    for u in V[start]:
        if not Visited[u]:
            Prev[u] = start
            DFS(u)
```

В этом алгоритме n — число вершин в графе, вершины нумеруются числами от 1 до n, а V[u] хранит множество вершин смежных с u. Для запуска алгоритма, например, для вершины с номером start необходимо вызвать DFS(start). После этого вызова все вершины, доступные из start, будут отмечены в списке Visited, а при помощи списка Prev можно построить пути из вершины start до всех доступных вершин. Если не требуется строить дерево обхода в глубину, то можно убрать заполнение списка Start, в этом случае алгоритм DFS становится чрезвычайно простым.

Алгоритм обхода в глубину позволяет решать множество различных задач. Например, реализуем при помощи алгоритма обхода в глубину подсчет числа компонент связности в неориентированном графе.

Для этого будем обходить все вершины графа и проверять, была ли очередная вершина посещена ранее. Если не была - то это означает, что найдена новая компонента связности, для выделения всей компоненты связности необходимо запустить DFS от этой вершины.

```
Visited = [False] * (n + 1)
def DFS(start):
    Visited[start] = True
    for v in V[start]:
        if not Visited[v]:
            DFS(v)
NComp = 0
for i in range(1, n + 1):
    if not Visited(i):
        NComp += 1
        DFS(i)
```

Другое примение DFS — проверка графа на двудольность. Граф называется двудольным, если его вершины можно разбить на два множества так, что концы каждого ребра принадлежат разным множествам. Иными словами, можно покрасить вершины графа в два цвета так, что концы каждого ребра покрашены в разный цвет.

Модифицируем алгоритм DFS так, что он будет проверять граф на двудольность и строить покраску графа в два цвета (если он двудольный). Для этого заменим список Visited на список Color, в котором будем хранить значение 0 для непосещенных вершин, а для посещенных вершин будем гранить значение 1 или 2 - ее цвет.

Алгоритм DFS для каждого ребра будет проверять цвет конечной вершины этого ребра. Если вершина не была посещена, то она красится в цвет, неравный цвету текущей вершины. Если же вершина была посещена, то ребро либо пропускается, если его концы - разноцветные, а если его концы одного цвета, то делается пометка, что граф не является двудольным (переменной IsBipartite присваивается значение False, по ее значению можно судить о том, является ли граф двудольный).

```
Color = [0] * (n + 1)
IsBipartite = True

def DFS(start):
    global IsBipartite
    for u in V[start]:
        if Color[u] == 0:
            Color[u] = 3 - Color[start]
            DFS(u)
        else if Color[u] == Color[start]:
            IsBipartite = False

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

```
if Color[i] == 0:
    Color[i] = 1
    DFS(i)
```

Основная программа проходит по всем ребрам графа и при обнаружении ранее не обнаруженной вершины красит ее в цвет 1 и запускает DFS из этой вершины.

Еще одно применение DFS - поиск цикла в ориентированном графе. Цикл в ориентированном графе можно обнаружить по наличию ребра, ведущего из текущей вершины в вершину, которая в настоящий момент находится в стадии обработки, то есть алгоритм DFS зашел в такую вершину, но еще не вышел из нее. В таком алгоритме DFS будем красить вершины в три цвета. Цветом 0 (“белый“) будем обозначать еще непосещенные вершины. Цветом 1 (“серый”) будем обозначать вершины в процессе обработки, а цветом 2 (“черный”) будем обозначать уже обработанные вершины. Вершина красится в цвет 1 при заходе в эту вершину и в цвет 2 - при выходе. Цикл в графе существует, если алгоритм DFS обнаруживает ребро, конец которого покрашен в цвет 1.

```
Color = [0] * (n + 1)
CycleFound = False

def DFS(start):
    Color[start] = 1
    for u in V[start]:
        if Color[u] == 0:
            DFS(u)
        else if Color[start] == 1:
            CycleFound = True
    Color[start] = 2

for i in range(1, n + 1):
    if Color[i] == 0:
        DFS(i)
```

Наконец, еще одно важное применение поиска в глубину - топологическая сортировка. Пусть дан ориентированный граф не содержащий циклов. Тогда вершины этого графа можно упорядочить так, что все ребра будут идти от вершин с меньшим номером к вершинам с большим номером.

Для топологической сортировки графа достаточно запустить алгоритм DFS, при выходе из вершины добавляя вершину в конец списка с ответом. После окончания алгоритма список с ответом развернуть в противоположном порядке.

```
Visited = [False] * (n + 1)
Ans = []

def DFS(start):
    Visited[start] = True
    for u in V[start]:
        if not Visited[u]:
            DFS(u)
```

```
    Ans.append(start)

for i in range(1, n + 1):
    if not Visited(i):
        DFS(i)
Ans = Ans[::-1]
```

10.5 Алгоритм Дейкстры

Алгоритм Дейкстры назван в честь голландского ученого Эдсгера Дейкстры (Edsger Dijkstra). Алгоритм был предложен в 1959 году для нахождения кратчайших путей от одной вершины до всех остальных в ориентированном взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса.

Рассмотрим две модели хранения взвешенного графа в памяти. В первой модели (матрица весов, аналог матрицы смежности) будем считать, что вес ребра из вершины i в вершину j равен W[i][j], то есть в матрице W хранятся веса ребра для любых двух вершин. Если из вершины i в вершину j нет ребра, то W[i][j] == INF для некоторого специального значения константы INF. Значение INF следует выбирать исходя из задачи, например, если речь идет о расстояниях между какими-либо населенными пунктами Земли, то можно выбрать значение INF равным 10^9 километров.

Алгоритм Дейкстры относится к так называемым “жадным” алгоритмам. Пусть расстояние от начальной вершины start до вершины i хранится в массиве D[i]. Начальные значения D[start]=0, D[i]=INF для всех остальных вершин i. То есть в самом начале алгоритму известен путь из вершины start до вершины start длины 0, а до остальных вершин кратчайшие пути неизвестны. Между тем алгоритм будет постепенно улучшать значения в массиве D, в результате получит кратчашие расстояния до всех вершин.

Основная идея для улучшения называется «релаксацией ребра». Пусть из вершины i в вершину j есть ребро веса W[i][j], при этом выполнено неравенство D[i] + W[i][j] < D[j]. То есть можно построить маршрут из начальной вершины до вершины i и добавить к нему ребро из i в j, и суммарная стоимость такого маршрута будет меньше, чем известная ранее стоимость маршрута из начальной вершины в вершину j. Тогда можно улучшить значение D[j], присвоив D[j] = D[i] + W[i][j].

В алгоритме Дейкстры вершины красятся в два цвета, будем говорить, что вершина “неокрашенная” или “окрашенная”. Изначально все вершины неокрашенные. Если алгоритм Дейкстры покрасил вершину i, то это означает, что найденное значение D[i] является наилучшим возможным и в последствии не будет улучшаться, то есть значение D[i] является кратчайшим расстоянием от начальной вершины до вершины i. Если же вершина не покрашена, то величина D[i] для такой вершины i равна кратчайшему пути из вершины start до вершины i, который проходит только по покрашенным вершинам (за исключением самой вершины i).

На каждом шаге алгоритма Дейкстры красится одна новая вершина. В качестве такой вершины выбирается неокрашенная вершина i с наименьшим значением D[i]. Затем рассматриваются все ребра, исходящие из вершины i, и производится релаксация этих ребер, то есть улучшаются расстo-

яния до вершин, смежных с i.

Алгоритм заканчивается, когда на очередном шаге не останется неокрашенных вершин или если расстояние до всех неокрашенных вершин будет равно INF (то есть эти вершины являются недостижимыми).

Запишем алгоритм Дейкстры. Пусть N — число вершин в графе, вершины пронумерованы от 0 до N-1. Номер начальной вершины — start и веса ребер хранятся в матрице W.

```
INF = 10 ** 10
D = [INF] * N
D[start] = 0
 Colored = [False] * N
while True:
    min_dist = INF
    for i in range(N):
        if not Colored[i] and D[i] < min_dist:
            min_dist = D[i]
            min_vertex = i
    if min_dist == INF:
        break
    i = min_vertex
    Colored[i] = True
    for j in range(N):
        if D[i] + W[i][j] < D[j]:
            D[j] = D[i] + W[i][j]
print(D)
```

Список Colored будет хранить информацию о том, была ли покрашена вершина. Сначала инициализируются списки D и Colored. Затем запускается внешний цикл алгоритма, который выбирает неокрашенную вершину с минимальным расстоянием, номер этой вершины хранится в переменной min_vertex, а расстояние до этой вершины — в переменной min_dist. Если же min_dist оказывается равно INF, то значит все неокрашенные вершины являются недостижимыми и алгоритм заканчивает свою работу. Иначе найденная вершина окрашивается и после этого релаксируются все ребра, исходящие из этой вершины.

Данный алгоритм имеет сложность $O(n^2)$, так как внешний цикл может быть выполнен до n раз, внутри него содержится два цикла, каждый из которых также выполняется n раз.

Для восстановления ответа, то есть для нахождения пути из начальной вершины до всех остальных, необходимо построить дерево кратчайших путей. Это дерево будет состоять из тех ребер, которые были успешно срелаксированы в результате исполнения алгоритма. То есть если происходит релаксация ребра из i в j, то теперь кратчайший маршрут из вершины start до вершины j должен проходить через вершину i и затем содержать ребро i-j. Тем самым вершина i становится предшественником вершины j на кратчайшем пути из начальной вершины до вершины j.

Рассмотрим реализацию алгоритм Дейкстры с восстановлением ответа на графе, хранимым в виде списка смежности. Набор вершин, смежных с вершиной i будет храниться в множествве W[i]. Также необходимо хранить веса ребер, будем считать, что для хранения весов ребер используется

словарь Weight, где ключом является кортеж из двух вершин. То есть вес ребра из i в j хранится в элементе Weight[i, j] словаря весов.

```
D = [INF] * N
D[start] = 0
Prev = [None] * N
Colored = [False] * N
while True:
    min_dist = INF
    for i in range(N):
        if not Colored[i] and D[i] < min_dist:
            min_dist = D[i]
            min_vertex = i
    if min_dist == INF:
        break
    i = min_vertex
    Colored[i] = True
    for j in W[i]:
        if D[i] + Weight[i, j] < D[j]:
            D[j] = D[i] + Weight[i, j]
            Prev[j] = i
```

Для нахождения кратчайшего пути из вершины start до вершины j будем переходить от каждой вершины к ее предшественнику:

```
Path =[]
while j is not None:
    Path.append(j)
    j = Prev[j]
Path = Path[::-1]
```

Алгоритм Дейкстры применим только в том случае, когда веса всех ребер неотрицательные. Это гарантирует то, что после окраски расстояние до вершины не может быть улучшено. Если в графе могут быть ребра отрицательного веса, то следует использовать другие алгоритмы.

Есть модификация алгоритма Дейкстры, которая более эффективна на разреженных графах. Поскольку необходимо обновлять расстояния до вершин и выбирать из неокрашенных вершин ту, до которой расстояние наименьшее, то можно хранить все неокрашенные вершины в куче. Тогда выбор наименьшей вершины будет выполняться за $O(\log n)$, что позволит более оптимально выбирать очередную окрашиваемую вершину. Но обновление расстояния до вершины в этом случае будет выполняться также за $O(\log n)$, так как это требует перестройки кучи. Если в графе m ребер, то максимальное число релаксаций ребер также будет не больше m и суммарная сложность всех релаксаций будет $O(m \log n)$. Таким образом, алгоритм Дейкстры с использованием кучи будет иметь сложность $O(n \log n + m \log n) = O((n+m) \log n)$. Если граф — разреженный, то такой алгоритм работает существенно быстрее, чем обычный алгоритм Дейкстры.

10.6 Алгоритм Форда-Беллмана

Алгоритм Форда-Беллмана позволяет найти кратчайшие пути из одной вершины графа до всех остальных, даже для графов, в которых веса ребер могут быть отрицательными. Тем не менее, в графе не должно быть циклов отрицательного веса, достижимых из начальной вершины, иначе вопрос о кратчайших путях является бессмысленным. При этом алгоритм Форда-Беллмана позволяет определить наличие циклов отрицательного веса, достижимых из начальной вершины.

Алгоритм Форда-Беллмана использует динамическое программирование. Введем функцию динамического программирования:

F[k][i] — длина кратчайшего пути из начальной вершины до вершины i, содержащего не более k ребер.

Начальные значения зададим для случая k=0. В этом случае F[0][start] = 0, а для всех остальных вершин i F[0][i] = INF, то есть путь, состоящий из нуля ребер существует только от вершины start до вершины start, а до остальных вершин пути из нуля ребер не существует, что будем отмечать значением INF.

Далее будем вычислять значения функции F увеличивая число ребер в пути k, то есть вычислим кратчайшие пути, содержащие не более 1 ребра, кратчайшие пути, содержащие не более 2 ребер и т.д. Если в графе нет циклов отрицательного веса, то кратчайший путь между любыми двумя вершинами содержит не более $n-1$ ребра (n - число вершин в графе), поэтому нужно вычислить значения F[n-1][i], которые и будут длинами кратчайших путей от вершины start до вершины i).

Рассмотрим, как вычисляется значение F[k][i]. Пусть есть кратчайший маршрут из вершины start до вершины i, содержащий не более k ребер. Пусть последнее ребро этого маршрута есть ребро j-i. Тогда путь до вершины j содержит не более k-1 ребра и является кратчайшим путем из всех таких путей, значит, его длина равна F[k-1][j], а длина пути до вершины i равна F[k-1][j] + W[j][i], где W[j][i] есть вес ребра j-i. Дальше необходимо перебрать все вершины j, которые могут выступать в качестве предыдущих, и выбрать минимальное значение F[k-1][j] + W[j][i].

Получаем следующий алгоритм:

```
INF = 10 ** 10
F = [[INF] * N for i in range(N)]
F[0][start] = 0
for k in range(1, N):
    for i in range(N):
        F[k][i] = F[k - 1][i]
        for j in range(N):
            if F[k - 1][j] + W[j][i] < F[k][i]:
                F[k][i] = F[k - 1][j] + W[j][i]
```

Очевидно, что сложность такого алгоритма $O(n^3)$.

Теперь модифицируем этот алгоритм. Прежде всего, сделаем список F одномерным — "склеим" значения F[k][i] для разных значений k, будем хранить в списке F[i] кратчайшее известное расстояние до вершины i, улучшая его по ходу. Получим следующий код:

```
INF = 10 ** 10
```

```
F = [INF] * N
F[start] = 0
for k in range(1, N):
    for i in range(N):
        for j in range(N):
            if F[j] + W[j][i] < F[i]:
                F[i] = F[j] + W[j][i]
```

Последние две строчки есть ни что иное, как релаксация ребра j-i, как это делается в алгоритме Дейкстры. А два последних цикла по вершинам j и i с релаксацией ребра j-i просто являются релаксацией всех ребер в графе. Но если граф "разреженный", то его удобно хранить не в виде матрицы смежности, а в виде списков смежности, тогда перебор всех ребер в графе можно осуществить быстрее, чем перебирая все пары вершин.

Пусть граф задан списками смежности, а вес ребра j-i хранится в словаре W[j, i], где ключ — это кортеж из j, i, а значение — вес ребра. Такой способ хранения графа рассматривался в виде одной из возможных реализаций алгоритмы Дейкстры.

Тогда перебрать все ребра графа, можно организовав цикл по всем ключам словаря W и алгоритм Форда-Беллмана можно записать в виде:

```
INF = 10 ** 10
F = [INF] * N
F[start] = 0
for k in range(1, N):
    for j, i in W.keys():
        if F[j] + W[j, i] < F[i]:
            F[i] = F[j] + W[j, i]
```

То есть по сути алгоритм Форда-Беллмана можно сформулировать так:

1. Проинициализировать список F значениями F[start] = 0, F[i] = INF для остальных i.
2. Пройтись по всем ребрам j-i графа, пытаясь срелаксировать ребро j-i.
3. Пункт 2 повторить n-1 раз.

Сложность такого алгоритма равна $O(nm)$, где n - число вершин, m - число ребер графа. Видно, что для плотных графов (где $m \approx n^2$) сложность близка к сложности предыдущего варианта алгоритма, а вот для разреженных графов (где $m \approx n$) такой алгоритм будет существенно быстрее.

Восстановление пути делается точно так же, как в алгоритме Дейкстры. Для этого необходимо запоминать предка для каждой вершины, обновляя его при успешной релаксации ребра:

```
INF = 10 ** 10
F = [INF] * N
Prev = [None] * N
F[start] = 0
for k in range(1, N):
    for j, i in W.keys():
        if F[j] + W[j, i] < F[i]:
```

```
            F[i] = F[j] + W[j, i]
            Prev[i] = Prev[j]
```

Сама процедура восстановления пути совпадает с аналогичной процедурой алгоритма Дейкстры.

Алгоритм Форда-Беллмана можно остановить, если на очередном шаге ни одно ребро не было срелаксировано:

```
INF = 10 ** 10
F = [INF] * N
F[start] = 0
Stop = False
k = 1
while k < N and not Stop:
    k += 1
    Stop = True
    for j, i in W.keys():
        if F[j] + W[j, i] < F[i]:
            F[i] = F[j] + W[j, i]
            Stop = False
```

Также алгоритм Форда-Беллмана можно использовать для проверки того, есть ли в графе цикл отрицательного веса, достижимый из начальной вершины. Для этого нужно еще раз попробовать срелаксировать все ребра. Если хотя бы одна релаксация возможна, то граф содержит цикл отрицательного веса, достижимый из начальной вершины.

```
CycleFound = False
for j, i in W.keys():
    if F[j] + W[j, i] < F[i]:
        CycleFound = True
        break
```

10.7 Алгоритм Флойда

Алгоритм Флойда (или Флойда-Уоршелла, Floyd–Warshall) позволяет найти кратчайшее расстояние между любыми двумя вершинами в графе, при этом веса ребер могут быть как положительными, так и отрицательными. Данный алгоритм также использует идею динамического программирования.

Будем считать, что в графе n вершин, пронумерованных числами от 0 до $n-1$. Граф задан матрицей смежности, вес ребра $i-j$ хранится в w_{ij}. При отсутствии ребра $i-j$ значение $w_{ij}=+\infty$, также будем считать, что $w_{ii}=0$.

Пусть значение a_{ij}^k равно длине кратчайшего пути из вершины i в вершину j, при этом путь может заходить в промежуточные вершины только с номерами меньшими k (не считая начала и конца пути). То есть a_{ij}^0 - это длина кратчайшего пути из i в j, который вообще не содержит промежуточных вершин, то есть состоит только из одного ребра $i-j$, поэтому $a_{ij}^0=w_{ij}$. Значение $a_{ij}^1=w_{ij}$ равно длине кратчайшего пути, который может проходить через промежуточную вершину с номером 0, путь с весом a_{ij}^2 может

проходить через промежуточные вершины с номерами 0 и 1 и т.д. Путь с весом a_{ij}^n может проходить через любые промежуточные вершины, поэтому значение a_{ij}^n равно длине кратчайшего пути из i в j.

Алгоритм Флойда последовательно вычисляет a_{ij}^0, a_{ij}^1, a_{ij}^2, ..., a_{ij}^n, увеличивая значение параметра k. Начальное значение - $a_{ij}^0 = w_{ij}$.

Теперь предполагая, что известны значения a_{ij}^{k-1} вычислим a_{ij}^k. Кратчайший путь из вершины i в вершину j, проходящий через вершины с номерами, меньшими, чем k может либо содержать, либо не содержать вершину с номером $k-1$. Если он не содержит вершину с номером $k-1$, то вес этого пути совпадает с a_{ij}^{k-1}. Если же он содержит вершину $k-1$, то этот путь разбивается на две части: $i-(k-1)$ и $(k-1)-j$. Каждая из этих частей содержит промежуточные вершины только с номерами, меньшими $k-1$, поэтому вес такого пути равен $a_{i,k-1}^{k-1} + a_{k-1,j}^{k-1}$. Из двух рассматриваемых вариантов необходимо выбрать вариант наименьшей стоимости, поэтому:

$$a_{ij}^k = \min(a_{ij}^{k-1}, a_{i,k-1}^{k-1} + a_{k-1,j}^{k-1})$$

Запишем алгоритм в виде программы на языке Питон. Обратите внимание, что нам понадобится трехмерный массив, поскольку используются три индекса - k, i, j:

```
A = [[[INF for j in range(n)] for i in range(n)] for k in range(n + 1)]
for i in range(n):
    for j in range(n):
        A[0][i][j] = W[i][j]
for k in range(1, n + 1):
    for i in range(n):
        for j in range(n):
            A[k][i][j] = min(A[k - 1][i][j],
                             A[k - 1][i][k - 1] + A[k - 1][k - 1][j])
```

Внешний цикл в этом алгоритме последовательно перебирает все вершины, затем пытается улучшить пути из i в j, разрешив им проходить через выбранную вершину. Упростим этот алгоритм, избавившись от «трехмерности» массива A: будем только хранить значение кратчайшего пути из i в j в A[i][j], а при улучшении пути будем записать новую длину пути также в A[i][j]. Также изменим определение цикла по переменной k, заменив значение k-1 на k.

```
A = [[W[i][j] for j in range(n)] for i in range(n)]
for k in range(n):
    for i in range(n):
        for j in range(n):
            A[i][j] = min(A[i][j], A[i][k] + A[k][j])
```

Очевидно, что сложность такого алгоритма $O(n^3)$.

Обратите внимание, что при наличии ребер отрицательного веса значения A[i][j] могут уменьшатся. Поэтому может оказаться, что значение A[i][j] было равно INF, а затем оно уменьшилось благодаря наличию ребер отрицательного веса. В результате значение A[i][j] оказалось меньше INF (например, за счет объединения пути длиной INF и пути отрицательного веса), но при этом все равно пути между вершинами i и j нет. Поэтому

нужно либо ставить дополнительные проверки на то, что A[i][k] и A[k][j] не равны INF, либо значения, которые незначительно меньше INF, также считать отсутствием пути.

Алгоритм Флойда некорректно работает при наличии цикла отрицательного веса, но при этом если путь от i до j не содержит цикла отрицательного веса, то вес этого пути будет найден алгоритмом правильно. Также при помощи данного алгоритма можно определить наличие цикла отрицательного веса: если вершина i лежит на цикле отрицательного веса, то значение A[i][i] будет отрицательным после окончания алгоритма.

Для восстановления ответа необходим двумерный массив предшественников. Будем считать, что в Prev[i][j] хранится номер вершины, являющейся предшественником вершины j на кратчайшем пути из вершины i. Тогда при обновлении значения A[i][j] нужно также обновить предшественника. А именно, если путь $i - j$ был обновлен на путь, проходящий через вершину k, то теперь предшественником вершины j на пути из i становится вершина, которая была ее предшественником на пути из k, то есть необходимо присвоить Prev[i][j]=Prev[k][j].

При этом с самого начала необходимо выполнить инициализацию — предшественником вершины j является вершина на пути из i в j, если вершины соединены ребром:

```
for i in range(n):
    for j in range(n):
        if W[i][j] < INF:
            Prev[i][j] = i
```

Запишем алгоритм, который сохраняет предшественников, а также добавим проверки на существование пути:

```
A = [[W[i][j] for j in range(n)] for i in range(n)]
Prev = [[(i if i != j else None) for j in range(n)] for i in range(n)]
for k in range(n):
    for i in range(n):
        for j in range(n):
            if (A[i][k] < INF and A[k][j] < INF and
                    A[i][k] + A[k][j] < A[i][j]):
                A[i][j] = A[i][k] + A[k][j]
                Prev[i][j] = Prev[k][j]
```

Восстановление пути из i в j аналогично ранее рассмотренным алгоритмам, только необходимо учесть двумерность массива Path:

```
Path =[]
while j is not None:
    Path.append(j)
    j = Prev[i][j]
Path = Path[::-1]
```