18.Расстояние Левенштейна. Алгоритм поиска.

Идея: задача состоит в следующем: нужно узнать узнать минимальное количество действий, чтобы из одной строки получить другую

Действия: вставить один симвоол, удалить один символ и заменить символ.

Похоже на наибольшую общую подпослежовательность, динамическое прроограммирование.

Применимость: нужен заранее известный алфавит.

```
A=input() #- строка, которая "правильная"
B=input() #- строка, которую "исправляем"
n=len(A)
m=len(B)
F=[[0]*(n+1) for i in range (m+1)]
for i in range (n+1):
    F[0][i]=i
for j in range (m+1):
    F[j][0]=j
for i in range (1, m+1):
    for j in range (1, n+1):
        a=A[:j]
        b=B[:i]
        if a[-1]==b[-1]:
            F[i][j]=F[i-1][j-1]
        else:
            F[i][j]=min(F[i-1][j-1], F[i-1][j], F[i][j-1])+1
for i in range (m+1):
    print(F[i])

print(F[-1][-1])
```

```
каток
карты
[0, 1, 2, 3, 4, 5]
[1, 0, 1, 2, 3, 4]
[2, 1, 0, 1, 2, 3]
[3, 2, 1, 1, 2, 3]
[4, 3, 2, 1, 2, 3]
[5, 4, 3, 2, 2, 3]
3
```

19.Пи-функция строки. Алгоритм Кнута-Морриса-Пратта.

```
Z-функция строки TODO применимость
```

```
s='' - строка
n=len(s)
l=0
r=0
z=[0]*n
for i in range (1,n):
    if i>r:
        while i+z[i]<n and s[z[i]]==s[z[i]+i]:
            z[i]+=1
        if z[i]>0:
            l, r=i, i+z[i]-1
    else:
        z[i]=min(r-i+1,z[i-l])
        while i+z[i]<n and s[z[i]]==s[z[i]+i]:
            z[i]+=1
        if z[i]>0:
            l, r=i, i+z[i]-1
```

Префикс-функция - есть строчка, есть итый элемент, смотрим длину наибольшего префикса строки, который начинается в итом символе. строки O(n)

```
s=''
p=[0]*len(s)
for i in range (len(s)):
    k=p[i-1]
    while k>0 and s[i]!=s[k]:
        k=p[k-1]
    if s[i]==s[k]:
        k+=1
    p[i]=k
```

Алгоритм КМП заключается в нахождении вхождений строки s в строку t. Делаем это через соединение этих двух строк в строку 's'+'#'+'t', затем применяем префикс-функцию, смотрим, где принимает максимальное значение, равное длине строки s. Следовательно в строке t строка s входит, начиная с индекса i-2n

20.Очереди FIFO и LIFO. Корректность скобочного выражения с несколькими видами скобок

FIFO - структура данных "First In First Out" - очередь.

Операции:

```
push — добавить в начало очереди новый элемент;
pop — извлечь из очереди последний элемент;
top — узнать значение последнего элемента (не удаляя его);
size — узнать количество элементов в очереди.
```

LIFO - структура данных "Last In First Out" - стек.

Операции:

```
push — добавить (положить) в конец стека новый элемент;
pop — извлечь из стека последний элемент;
top — узнать значение последнего элемента (не удаляя его);
size — узнать количество элементов в стеке.
```

Модуль collections

Этот модуль реализует специализированные типы данных контейнеров, предоставляя альтернативы встроенным контейнерам общего назначения Python, dict, list, set, и tuple.

Алгоритм для скобок

Идея такая: делаем стэк, добавляем в него открывающие скобки. Когда попадается закрывающая, то проверяем, что у нее такой же тип, как на вершине стэка. Если такой же, то убиарем скобкус вершины стэка. Иначе скобочная послежлвательность некорректна.

```
def push (stack, x):
    stack.append(x)

def pop (stack):
    return stack.pop()

def size (stack):
    return len(stack)

def top (stack):
    return stack[len(stack)-1]

s=''
stack=[]
br1='{[('
br2='}])'
for i in range (len(s)):
    if s[i] in br1:
        push(stack, s[i])
    if s[i] in br2:
        x=br2.index(s[i])
        if size(stack)>0:
            if br1.index(top(stack))==x:
                pop(stack)
            else:
                print('NO')
                exit()
        else:
            print('NO')
            exit()
if size(stack)>0:
    print('NO')
else:
    print('YES')
```

13.Динамичное программирование снизу. Задачи про кузнечика на числовой прямой.

Динамическое программирование снизу - начинаем решение с маленьких задач, используем ответ для решения более больших.

TODO полное условие задачи про кузнечика

Пусть P[i] - стоимость попадания в i-ю точку числовой прямой, а F[i] - минимальная стоимость всей траектории от 1-й до i-й

```
P = list(map(int, input().split()))

F=[0]*(n+1)
F[0]=999999999999
F[1]=P[1]
for i in range (2,n+1):
    F[i]=min(F[i-1],F[i-2])+P[i]

print(F[-1])
```

14.Максимальная сумма подотрезка числовой последовательности. Однопроходный алгоритм

Пусть дан массив чисел a длины n, требуется найти такой его отрезок a[l:r+1], что сумма в нем максимальна.

Для решения данной задачи рассмотрим алгоритм, предложенный Джеем Каданом (Jay Kadane) в 1984 г.

Сам алгоритм выглядит следующим образом. Будем идти по массиву и накапливать в некоторой переменной s текущую частичную сумму. Если в какой-то момент s окажется отрицательной, то присвоим s = 0. Максимум из всех значений переменной s, случившихся за время работы, и будет ответом на задачу:

```
ressum=0
cursum=0
lind=0
rind=0
minind=-1
for i in range (len(a)):
    cursum+=a[i]
    if cursum<0:
        cursum=0
        indmin=i
    if ressum<cursum:
        ressum=cursum
        lind=minind+1
        rind=i
```

15.Наидлиннейшая возрастающая подпоследовательность

Идея следующая: если левее текущего элемента a[i] нет меньшего, то длина наибольшей возрастающей последовательности, заканчивающейся в a[i] равна 1, запишем это в L[i]. Если же нашлись такие a[j1], a[j2], ..., что они левее и меньше a[i], то L[i] = max(L[j1], L[j2], ...)

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

16.Длина набольшей по длине общей подпоследовательности двух последовательностей.

Воспользуемся динамическим программированием. F[i][j] - длина наибольшей по длине общей подпоследовательности последовательности A[:i+1] и B[:j+1] (то есть берём в A первые i элементов, а в B - первые j)

Если текущие элементы A и B равны, то F[i][j] = F[i-][j-1]+1, то есть мы продлили общую подпоследовательность новым символом

Иначе мы выбираем наилучший из уже подсчитанных ответов для F[i-1][j] и F[i][j-1]

```
A = list(map(int, input().split()))
B = list(map(int, input().split()))

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])
```

17.Алгоритм укладки рюкзака с дискретными массами предметов

Пусть W - список масс предметов, P - список цен каждого из предмета, K - максимальная грузоподъёмность рюкзака, n -

Будем для каждой возможной массы k хранить информацию о способе набора этой массы и наибольшую стоимость предметов, которые можно набрать в рюкзак данной массы.

Определим функцию $F(i, k)$ – 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.

Если же предмет с номером 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 предметами, 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.

Применимость: количество ограничено + целочисленные значения.

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