Lab: Dynamic Programming

Please submit your solutions (source code) to all below-described problems in Judge.

1. Fibonacci

Write a **dynamic programming** solution for finding n^th Fibonacci members.

- $F_0 = 0$
- $F_1 = 1$

Examples

Input	Output
20	6765
50	12586269025

2. Move Down/Right

Given a **matrix of N by M** cells filled with positive integers, find the path from **top left** to **bottom right** with the **highest sum** of cells by moving only down or right.

Examples

Input	Output

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

3
3
1 0 6
8 3 7
2 4 9
	[0, 0] [1, 0] [1, 1] [1, 2] [2, 2]

8
7
2 6 1 8 9 4 2
1 8 0 3 5 6 7
3 4 8 7 2 1 8
0 9 2 8 1 7 9
	[0, 0] [0, 1] [1, 1] [2, 1] [2, 2]
	[2, 3] [3, 3] [4, 3] [4, 4] [4, 5]
	[5, 5] [5, 6] [6, 6] [7, 6]

2 7 1 9 7 8 2
4 5 6 1 2 5 6
9 3 5 2 8 1 9
2 3 4 1 7 2 8	

3. Longest Common Subsequence

Considering **two sequences S_1 and S_2**, the **longest common subsequence** is a sequence that is a subsequence of both S_1 and S_2. For instance, if we have two strings (sequences of characters), "abc" and "adb", the LCS is "ab" – it is a subsequence of both sequences and it is the longest (there are two other subsequences – "a" and "b").

Input

- On the first line, you will receive a string – **`str1`** – first string.
- On the second line, you will receive a string – **`str2`** – second string.

Output

- Print the **length of the longest common subsequence**.

Examples

Input	Output
abc adb	2
ink some beer drink se ber	10
tree team	2

Solution

Dynamic Programming Approach

Just like the LIS problem, we can solve the LCS problem by **solving sub-problems** and keeping track of the solutions to the sub-problems (**memoization**). In the LIS problem, we used an array, but here we'll be comparing two sequences, therefore, we'll need a matrix like the one below:

		t	e	a	m
	LCS("", "")	LCS("", t)	LCS("", te)	LCS("", tea)	LCS("", team)
t	LCS(t, "")	LCS(t, t)	LCS(t, te)	LCS(t, tea)	LCS(t, team)
r	LCS(tr, "")	LCS(tr, t)	LCS(tr, te)	LCS(tr, tea)	LCS(tr, team)
e	LCS(tre, "")	LCS(tre, t)	LCS(tre, te)	LCS(tre, tea)	LCS(tre, team)

e	LCS(tree, "")	LCS(tree, t)	LCS(tree, te)	LCS(tree, tea)	LCS(tree, team)

The rows will represent subsequences (substrings) of the first string ("tree"); the first row will represent a substring with length 0 (an empty string), the second row will represent a substring of length 1 ("t"), the third row will represent a substring of length 2 ("tr"), etc. The last row will represent a substring of length 4 which is the entire string "tree".

The columns will represent the substrings of the second string ("team"), again starting with an empty string and ending with the entire string.

In each cell, we'll enter the length of the LCS of the two substrings – the substring of the first string (the rows) and the second string (the columns). E.g., in the table above, cells (2, 2) will represent the LCS of "tr" and "te". Note that we assume that an empty string does not have anything in common with any other string, therefore row 0 and column 0 will be filled with zeros.

Find the LCS for Every Combination of Substrings

We know what to do – create a matrix of integers and calculate the LCS length for each cell. Let's begin.

The matrix should have 1 more row than the number of characters in the first string and 1 more column than the number of characters in the second string (the first row and column are the empty substrings). Therefore:

```python
first = input()
second = input()

rows = len(first) + 1
cols = len(second) + 1
lcs = []
[lcs.append([0] * cols) for _ in range(rows)]
```

Now, we must iterate each cell of **`lcs`** from top to bottom and from left to right and decide what number to put in that cell. Remember, at each step, we already have the results from previous steps, so we can build on that. We have two distinct cases:

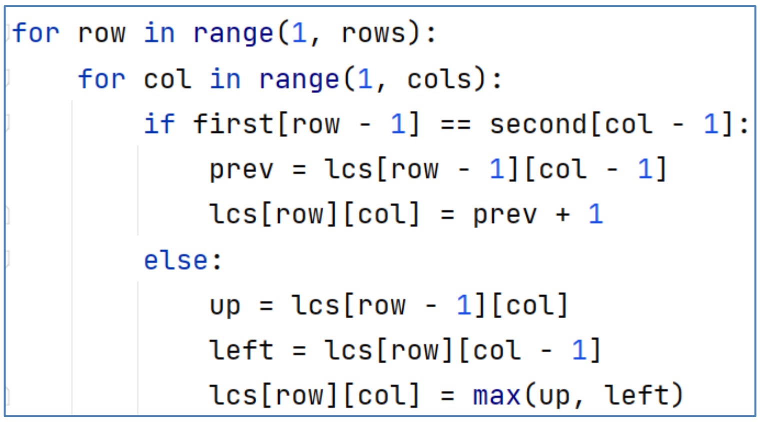
1) The last character of the first substring is equal to the last character of the second substring.

This means that, compared to the cell which is to the left and up of the current one, the length of the current cell's LCS is greater by 1. Why? The cell to the left and up of the current one will hold the LCS of two substrings which are shorter by 1 than the current substrings; basically, the last character (which is the same) won't be present. Adding that same character to both substrings, we'll obtain the current cell and an LCS greater by 1.

2) The last character of the first substring is different from the last character of the second substring.

We know the LCS of all substrings is shorter than the current ones. The longest LCS so far should be one of two – the one directly above or the one directly to the left of the current cell. Adding a character to one of the substrings used to calculate these two LCSs doesn't have any effect, therefore, the current cell's LCS is the larger of the two.

Complete the if-statement following the logic above:



```
for row in range(1, rows):
    for col in range(1, cols):
        if first[row - 1] == second[col - 1]:
            prev = lcs[row - 1][col - 1]
            lcs[row][col] = prev + 1
        else:
            up = lcs[row - 1][col]
            left = lcs[row][col - 1]
            lcs[row][col] = max(up, left)
```

Once done, the matrix should be filled with the length of each LCS, like so:

		t	e	a	m
	0 LCS("", "") = ""	0 LCS("", t) = ""	0 LCS("", te) = ""	0 LCS("", tea) = ""	0 LCS("", team) = ""
t	0 LCS(t, "") = ""	1 LCS(t, t) = t	1 LCS(t, te) = t	1 LCS(t, tea) = t	1 LCS(t, team) = t
r	0 LCS(tr, "") = ""	1 LCS(tr, t) = t	1 LCS(tr, te) = t	1 LCS(tr, tea) = t	1 LCS(tr, team) = t
e	0 LCS(tre, "") = ""	1 LCS(tre, t) = t	2 LCS(tre, te) = te	2 LCS(tre, tea) = te	2 LCS(tre, team) = te
e	0 LCS(tree, "") = ""	1 LCS(tree, t) = t	2 LCS(tree, te) = te	2 LCS(tree, tea) = te	2 LCS(tree, team) = te

Recover the LCS

Once the table is filled, all we need to do is recover what we need from it. Let's do this in a separate method.

We iterate the matrix starting from the bottom-right corner until we reach row 0 or column 0. We'll fill the characters in a **deque** (this way we can easily keep the letters in the correct order).

Again, we have two distinct cases:

1) The last characters of the two substrings are the same – add the character to the list and move to the cell which is to the left and above the current one. The logic is the same as the one we used to fill the matrix.
2) The characters are different – we need to decide where to go next – up or left. We go to the cell which has the same LCS length as the current one (if both have the same length, it doesn't matter).

```
lcs_letters = deque()
row = rows - 1
col = cols - 1
while row >= 0 and col >= 0:
    if first[row - 1] == second[col - 1]:
        lcs_letters.appendleft(first[row - 1])
        row -= 1
        col -= 1
    elif lcs[row - 1][col] > lcs[row][col - 1]:
        row -= 1
    else:
        col -= 1
```

4. Longest Increasing Subsequence

Let's have a sequence of numbers **S = {a_1, a_2, … a_n}**. An **increasing** subsequence is a sequence of numbers within **S** where each number is **larger** than the previous. We **do not change the relative positions** of the numbers, e.g. we do not move smaller elements to the left to obtain longer sequences. If several sequences with equal length exist, find the left-most of them

Examples

Input	Output
`1 2 5 3 4`	`1 2 3 4`
`4 3 2 1`	`4`
`4 2 -1 3 5 5`	`2 3 5`

Solution

Dynamic Programming Approach

The LIS problem can be solved by dividing it into sub-problems – for each element at **index `I`** of the **sequence S**, find the LIS in the range [**S_0** … **S_i**].

Example for **S** = { 3, 14, 5, 12, 15, 7, 8, 9, 11, 10, 1 }:

- LIS { 3 } => { 3 }
- LIS { 3, 14 } => { 3, 14 }
- LIS { 3, 14, 5 } => { 3, 5 }
- LIS { 3, 14, 5, 12 } => { 3, 5, 12 }
- etc.

For each index, we'll **keep track of the length of the LIS up to that index** and the **previous index** of the LIS. E.g., the length of the LIS at index **5** is **3**, the longest sequence ending in **`seq[5]`** is {3, 5, 7} and the index of the previous element of the subsequence (the number 5) is 2. The table below illustrates these computations:

`index`	0	1	2	3	4	5	6	7	8	9	10

S[]	3	14	5	12	15	7	8	9	11	10	1
len[]	1	2	2	3	4	3	4	5	6	6	1
prev[]	-1	0	0	2	3	2	5	6	7	7	-1
LIS	{3}	{3,14}	{3,5}	{3,5,12}	{3,5,12,15}	{3,5,7}	{3,5,7,8}	{3,5,7,8,9}	{3,5,7,8,9,11}	{3,5,7,8,9,10}	{1}

We need to calculate the info in the above table for every element of the original sequence **S**, so we'll need two additional arrays with lengths equal to the length of **S**. Translating this into code within our method, we get:

```
length = [0] * len(nums)
parent = [0] * len(nums)
```

We also need to keep track of the **length of the longest subsequence** found so far and the index at which it ends (we'll use **-1** to mark that there is no such index found currently):

```
best_len = 0
best_idx = -1
```

Calculate LIS at Each Index

To obtain the longest increasing sequence up to a given index, we just have to find the LIS up to that point to which the current element can be appended as the largest. That is why, when considering the sequence {3, 14, 5} we obtained {3, 5}; we want to know the longest sequence in which the current number (5) participates.

We'll do the following:

- loop through each number in the sequence.
- find the longest sequence up to that point which ends with a number that is smaller than the current.

Remember that we keep track of the length of each LIS in the **`length`** array.

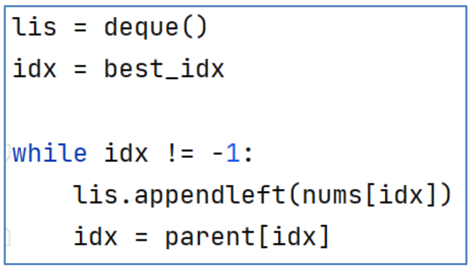
```
for curr_idx in range(len(nums)):
    curr_num = nums[curr_idx]
    curr_len = 1
    curr_parent = -1

    for prev_idx in range(curr_idx - 1, -1, -1):
        prev_number = nums[prev_idx]
        prev_len = length[prev_idx]
        if curr_num > prev_number and prev_len + 1 >= curr_len:
            curr_len = prev_len + 1
            curr_parent = prev_idx

    length[curr_idx] = curr_len
    parent[curr_idx] = curr_parent
```

Don't forget to keep track of the length of the longest increasing subsequence and the index of its last element:

```
if curr_len > best_len:
    best_len = curr_len
    best_idx = curr_idx
```

Recover the LIS

Knowing the index of the last element of the LIS, we can get the whole sequence by continuously taking each previous element using the info we keep in the **`parent`** array. Store the elements in a deque (this way we can easily keep the correct order of the elements):



```
lis = deque()
idx = best_idx

while idx != -1:
    lis.appendleft(nums[idx])
    idx = parent[idx]
```