

Longest Repeating Subsequence

We'll cover the following



- Problem Statement
- Basic Solution
 - Code
- Top-down Dynamic Programming with Memoization
 - Code
- Bottom-up Dynamic Programming
 - Code

Problem Statement

Given a sequence, find the length of its longest repeating subsequence (LRS). A repeating subsequence will be the one that appears at least twice in the original sequence and is not overlapping (i.e. none of the corresponding characters in the repeating subsequences have the same index).

Example 1:

Input: "t o m o r r o w"

Output: 2

Explanation: The longest repeating subsequence is "or" {tomorrow}.

Example 2:

Input: "a a b d b c e c"

Output: 3

Explanation: The longest repeating subsequence is "a b c" {a a b d b c e c}.

Example 3:

Input: "f m f f"

Output: 2

Explanation: The longest repeating subsequence is "f f" {f m f f, f m f f}. Please note the second last character is shared in LRS.

Basic Solution

The problem is quite similar to the Longest Common Subsequence

(<https://www.educative.io/collection/page/5668639101419520/5633779737559040/5657535201673216>) (LCS), with two differences:

1. In LCS, we were trying to find the longest common subsequence between the two strings, whereas in LRS we are trying to find the two longest common subsequences within one string.
2. In LRS, every corresponding character in the subsequences should not have the same index.

A basic brute-force solution could be to try all subsequences of the given sequence to find the longest repeating one, but the problem is how to ensure that the LRS's characters do not have the same index. For this, we can start with two indexes in the given sequence, so at any step we have two choices:

1. If the two indexes are not the same and the characters at both the indexes are same, we can recursively match for the remaining length (i.e. by incrementing both the indexes).
2. If the characters at both the indexes don't match, we start two new recursive calls by incrementing each index separately. The LRS would be the one with the highest length from the two recursive calls.

Code #

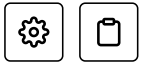
Here is the code:

Java	JS	Python3	C++
------	----	---------	-----

```
1 def find_LRS_length(str):
2     return find_LRS_length_recursive(str, 0, 0)
3
4
5 def find_LRS_length_recursive(str, i1, i2):
6     if i1 == len(str) or i2 == len(str):
7         return 0
8
9     if i1 != i2 and str[i1] == str[i2]:
10        return 1 + find_LRS_length_recursive(str, i1 + 1, i2 + 1)
11
12    c1 = find_LRS_length_recursive(str, i1, i2 + 1)
13    c2 = find_LRS_length_recursive(str, i1 + 1, i2)
14
15    return max(c1, c2)
16
17
18 def main():
19     print(find_LRS_length("tomorrow"))
20     print(find_LRS_length("aabdbcec"))
21     print(find_LRS_length("fmff"))
22
23
24 main()
25
```

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' is the length of the input sequence. The space complexity is $O(n)$ which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization



We can use an array to store the already solved subproblems.

The two changing values to our recursive function are the two indexes, $i1$ and $i2$. Therefore, we can store the results of all the subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string ($i1 + "|" + i2$)).

Code #

Here is the code:

Java	JS	Python3	C++
------	----	---------	-----

```
1 def find_LRS_length(str):
2     n = len(str)
3     dp = [[-1 for _ in range(n)] for _ in range(n)]
4     return find_LRS_length_recursive(dp, str, 0, 0)
5
6
7 def find_LRS_length_recursive(dp, str, i1, i2):
8     n = len(str)
9     if i1 == n or i2 == n:
10        return 0
11
12    if dp[i1][i2] == -1:
13        if i1 != i2 and str[i1] == str[i2]:
14            dp[i1][i2] = 1 + find_LRS_length_recursive(dp, str, i1 + 1, i2 + 1)
15        else:
16            c1 = find_LRS_length_recursive(dp, str, i1, i2 + 1)
17            c2 = find_LRS_length_recursive(dp, str, i1 + 1, i2)
18            dp[i1][i2] = max(c1, c2)
19
20    return dp[i1][i2]
21
22
23 def main():
24     print(find_LRS_length("tomorrow"))
25     print(find_LRS_length("aabdbcec"))
26     print(find_LRS_length("fmff"))
27
28
29 main()
```

Bottom-up Dynamic Programming

Since we want to match all the subsequences of the given string, we can use a two-dimensional array to store our results. As mentioned above, we will be tracking two indexes to overcome the overlapping problem. So for each of the two indexes, ' $i1$ ' and ' $i2$ ', we will choose one of the following options:









1. If 'i1' and 'i2' are different and the character `str[i1]` matches the character `str[i2]` then the length of the LRS would be one plus the length of LRS up to `i1-1` and `i2-1` indexes.
2. If the character at `str[i1]` does not match `str[i2]`, we will take the LRS by either skipping 'i1'th or 'i2'th character.

So our recursive formula would be:

```
1 if i1 != i2 && str[i1] == str[i2]
2     dp[i1][i2] = 1 + dp[i1-1][i2-1]
3 else
4     dp[i1][i2] = max(dp[i1-1][i2], dp[i1][i2-1])
```

Code

Here is the code for our bottom-up dynamic programming approach:

 Java	 JS	 Python3	 C++
<pre>1 def find_LRS_length(str): 2 n = len(str) 3 dp = [[0 for _ in range(n+1)] for _ in range(n+1)] 4 maxLength = 0 5 # dp[i1][i2] will be storing the LRS up to str[0..i1-1][0..i2-1] 6 # this also means that subsequences of length zero(first row and column of 7 # dp[][]), will always have LRS of size zero. 8 for i1 in range(1, n+1): 9 for i2 in range(1, n+1): 10 if i1 != i2 and str[i1 - 1] == str[i2 - 1]: 11 dp[i1][i2] = 1 + dp[i1 - 1][i2 - 1] 12 else: 13 dp[i1][i2] = max(dp[i1 - 1][i2], dp[i1][i2 - 1]) 14 15 maxLength = max(maxLength, dp[i1][i2]) 16 17 return maxLength 18 19 20 def main(): 21 print(find_LRS_length("tomorrow")) 22 print(find_LRS_length("aabdbcec")) 23 print(find_LRS_length("fmff")) 24 25 26 main() 27</pre>			
<div></div>			

The time and space complexity of the above algorithm is $O(n^2)$, where 'n' is the length of the input sequence.

[< Back](#)[Next >](#)

 Mark as Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/longest-repeating-subsequence__pattern-5-longest-common-substring__grokking-dynamic-programming-patterns-for-coding-interviews)

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.