Longest Common Subsequence

We'll cover the following

Problem Statement

Given two strings ‘s1’ and ‘s2’, find the length of the longest subsequence which is common in both the strings.

A subsequence (https://en.wikipedia.org/wiki/Subsequence) is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

```
Input: s1 = "abdca"
       s2 = "cbda"
Output: 3
Explanation: The longest common subsequence is "bda".
```

Example 2:

```
Input: s1 = "passport"
       s2 = "ppsspt"
Output: 5
Explanation: The longest common subsequence is "psspt".
```

Basic Solution

A basic brute-force solution could be to try all subsequences of ‘s1’ and ‘s2’ to find the longest one. We can match both the strings one character at a time. So for every index ‘i’ in ‘s1’ and ‘j’ in ‘s2’ we must choose between:

1. If the character `s1[i]` matches `s2[j]`, we can recursively match for the remaining lengths.

2. If the character `s1[i]` does not match `s2[j]`, we will start two new recursive calls by skipping one character separately from each string.

Here is the code:

Java | JS | Python3 | C++

```python
def find_LCS_length(s1, s2):
  return find_LCS_length_recursive(s1, s2, 0, 0)


def find_LCS_length_recursive(s1, s2, i1, i2):
  if i1 == len(s1) or i2 == len(s2):
    return 0

  if s1[i1] == s2[i2]:
    return 1 + find_LCS_length_recursive(s1, s2, i1 + 1, i2 + 1)

  c1 = find_LCS_length_recursive(s1, s2, i1, i2 + 1)
  c2 = find_LCS_length_recursive(s1, s2, i1 + 1, i2)

  return max(c1, c2)


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

The time complexity of the above algorithm is exponential $O(2^{m+n})$, where 'm' and 'n' are the lengths of the two input strings. The space complexity is $O(n+m)$ 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)).

Here is the code:

Java | JS | Python3 | C++

```python
def find_LCS_length(s1, s2):
  dp = [[-1 for _ in range(len(s2))] for _ in range(len(s1))]
  return find_LCS_length_recursive(dp, s1, s2, 0, 0)


```

```
def find_LCS_length_recursive(dp, s1, s2, i1, i2):
  if i1 == len(s1) or i2 == len(s2):
    return 0

  if dp[i1][i2] == -1:
    if s1[i1] == s2[i2]:
      dp[i1][i2] = 1 + find_LCS_length_recursive(dp, s1, s2, i1 + 1, i2 + 1)
    else:
      c1 = find_LCS_length_recursive(dp, s1, s2, i1, i2 + 1)
      c2 = find_LCS_length_recursive(dp, s1, s2, i1 + 1, i2)
      dp[i1][i2] = max(c1, c2)

  return dp[i1][i2]


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

Bottom-up Dynamic Programming

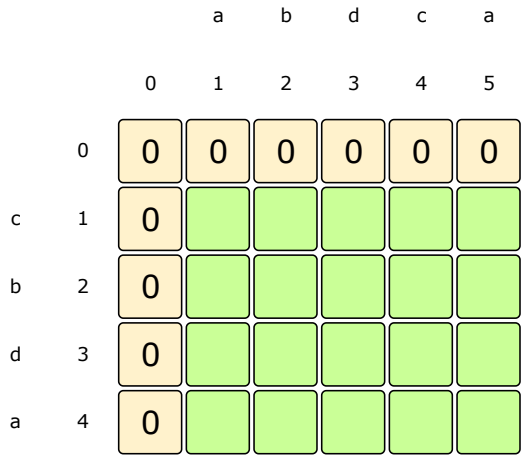
Since we want to match all the subsequences of the given two strings, we can use a two-dimensional array to store our results. The lengths of the two strings will define the size of the array's two dimensions. So for every index 'i' in string 's1' and 'j' in string 's2', we will choose one of the following two options:

1. If the character `s1[i]` matches `s2[j]`, the length of the common subsequence would be one plus the length of the common subsequence till the `i-1` and `j-1` indexes in the two respective strings.
2. If the character `s1[i]` does not match `s2[j]`, we will take the longest subsequence by either skipping ith or jth character from the respective strings.

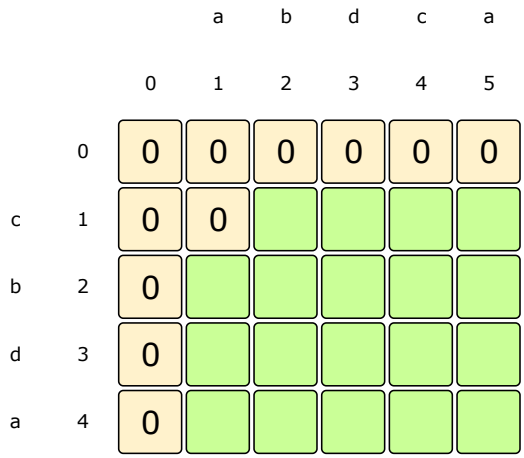
So our recursive formula would be:

```
if s1[i] == s2[j]
  dp[i][j] = 1 + dp[i-1][j-1]
else
  dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

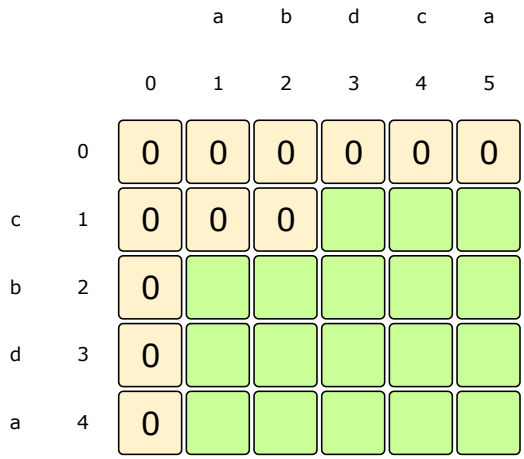
Let's draw this visually for "abcda" and "cbda". Starting with a subsequence of zero lengths, if any string has zero length then the common subsequence will be of zero length:



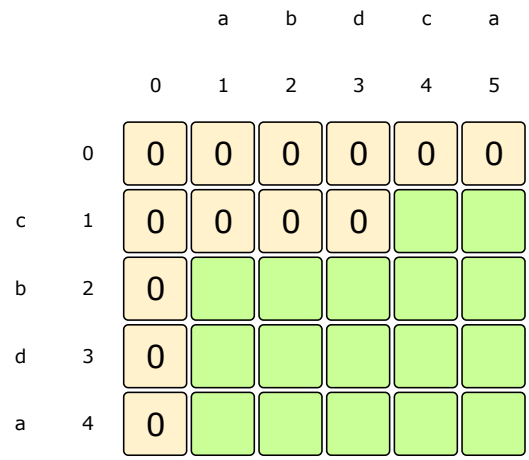
i:0, j:0-5 and i:0-4, j:0 => dp[i][j] = 0, as we don't have any common subsequence when one of the string is of zero length



i:1, j:1 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

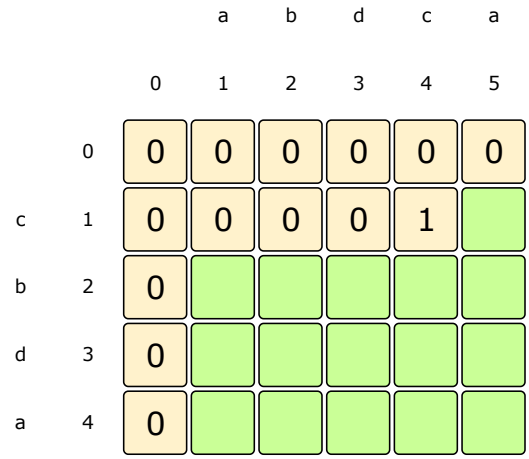


i:1, j:2 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]



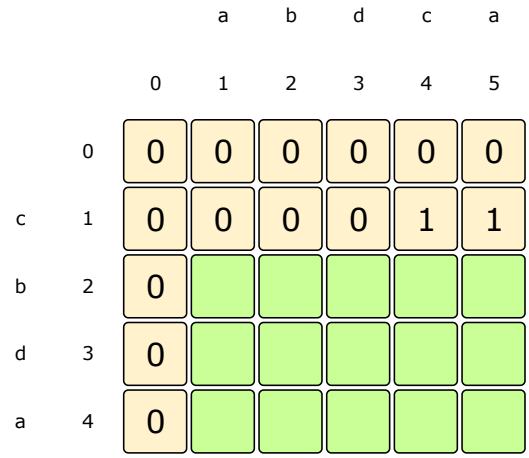
i:1, j:3 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

4 of 16



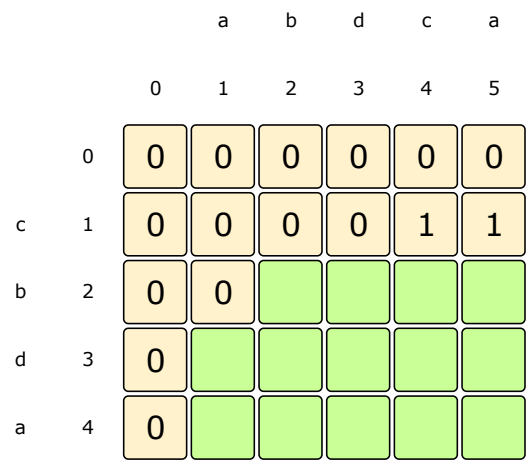
i:1, j:4 => dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

5 of 16



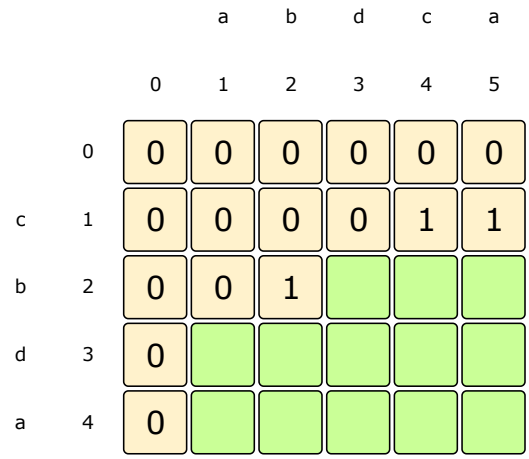
i:1, j:5 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

6 of 16



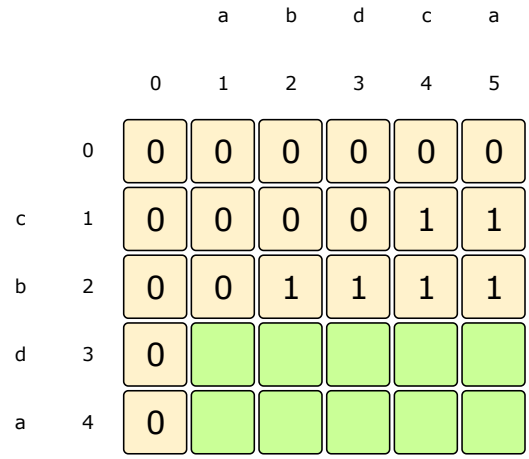
i:2, j:1 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

7 of 16



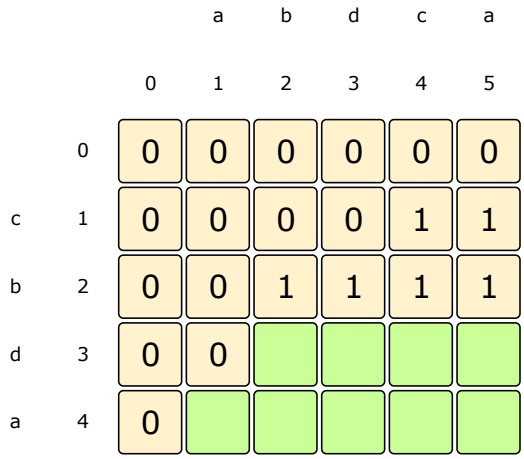
i:2, j:2=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

8 of 16



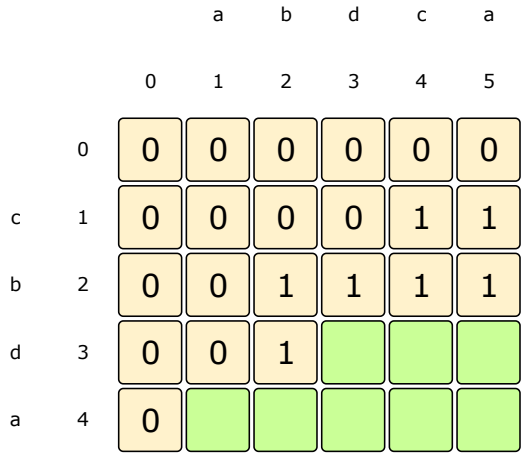
i:2, j:3-5 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

9 of 16



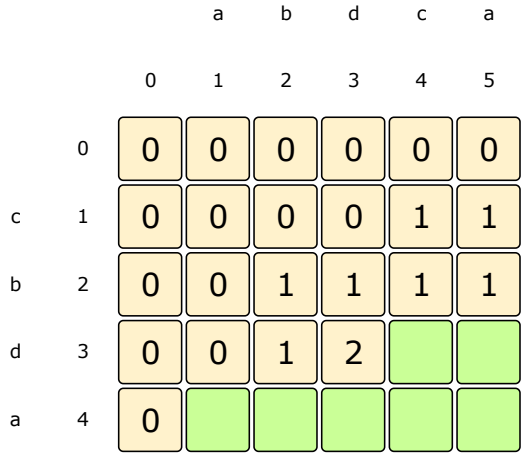
i:31, j:1 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

10 of 16



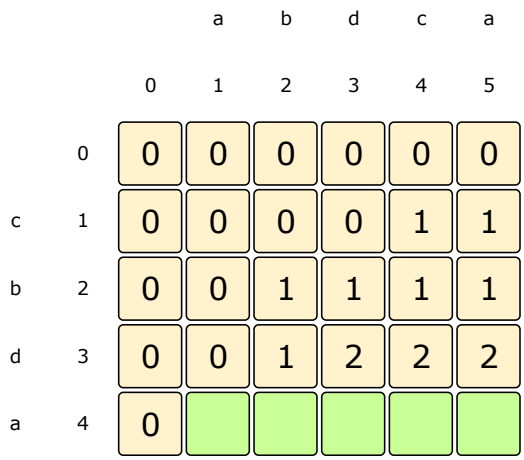
i:3, j:2 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]

11 of 16

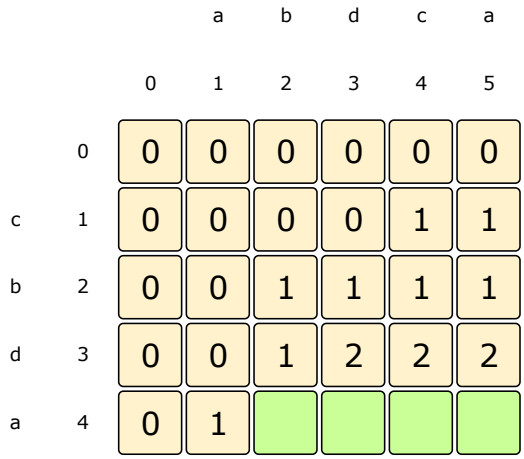


i:3, j:3=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

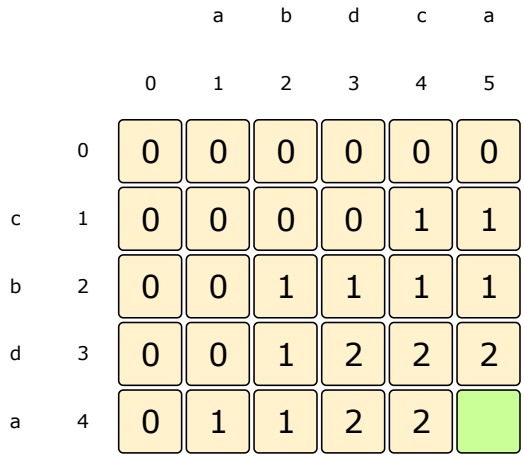
12 of 16



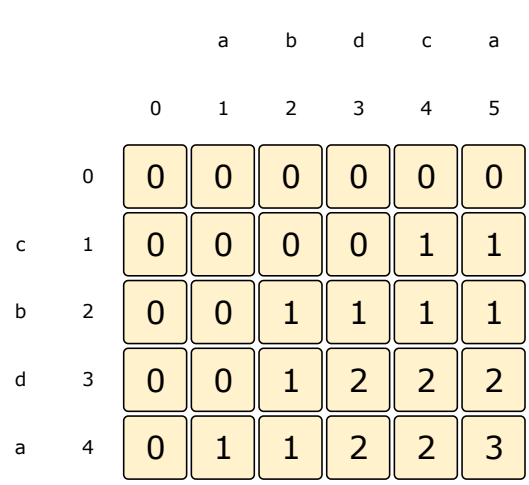
i:3, j:4-5 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]



i:4, j:1=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]



i:4, j:2-4 => dp[i][j] = max(dp[i-1][j], dp[i][j-1]), as s1[i] != s2[j]



i:4, j:5=> dp[i][j] = 1 + dp[i-1][j-1], as s1[i] == s2[j]

16 of 16

From the above visualization, we can clearly see that the longest common subsequence is of length '3' – as shown by `dp[4][5]`.

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

Java JS Python3 C++

```
def find_LCS_length(s1, s2):
  n1, n2 = len(s1), len(s2)
  dp = [[0 for _ in range(n2+1)] for _ in range(n1+1)]
  maxLength = 0
  for i in range(1, n1+1):
    for j in range(1, n2+1):
      if s1[i - 1] == s2[j - 1]:
        dp[i][j] = 1 + dp[i - 1][j - 1]
      else:
        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

      maxLength = max(maxLength, dp[i][j])
  return maxLength


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))


main()
```

The time and space complexity of the above algorithm is $O(m * n)$, where 'm' and 'n' are the lengths of the two input strings.

Challenge

Can we further improve our bottom-up DP solution? Can you find an algorithm that has $O(n)$ space complexity?

Hide Hint

We only need one previous row to find the optimal solution!

Java | JS | Python3 | C++

```
def find_LCS_length(s1, s2):
  # TODO: Write your code here
  return -1

```

Solution

```
def find_LCS_length(s1, s2):
  n1, n2 = len(s1), len(s2)
  dp = [[0 for _ in range(n2+1)] for _ in range(2)]
  maxLength = 0
  for i in range(1, n1+1):
    for j in range(1, n2+1):
      if s1[i - 1] == s2[j - 1]:
        dp[i % 2][j] = 1 + dp[(i - 1) % 2][j - 1]
      else:
        dp[i % 2][j] = max(dp[(i - 1) % 2][j], dp[i % 2][j - 1])

      maxLength = max(maxLength, dp[i % 2][j])

  return maxLength


def main():
  print(find_LCS_length("abdca", "cbda"))
  print(find_LCS_length("passport", "ppsspt"))

main()
```

Back

Longest Common Substring

Next

Minimum Deletions & Insertions to Tra...

Mark as Completed