Solution Review: Longest Common Substring

In this lesson, we will look at different strategies to solve the longest common substring problem.

We'll cover the following

Solution 1: Simple recursion

```
def lcs_(str1, str2, i, j, count):
  # base case of when either of string has been exhausted
  if i >= len(str1) or j >= len(str2):
    return count
  # if i and j character matches, increment the count and compare the rest of the strings
  if str1[i] == str2[j]:
    count = lcs_(str1, str2, i+1, j+1, count+1)
  # compare str1[1:] with str2, str1 with str2[1:], and take max of current count and these two re
  return max(count, lcs_(str1, str2, i+1, j, 0), lcs_(str1, str2, i, j+1, 0))


def lcs(str1, str2):
  return lcs_(str1, str2, 0, 0, 0)

print(lcs("hello", "elf"))
```

Explanation

Let’s look at this problem on a smaller scale. We are comparing each character one by one with the other string. There can be three possibilities for `i`th character of `str1` and `j`th character of `str2`. If both characters match, these could be part of a common substring meaning we should count this length (*lines 6-7*). In the case that these characters do not match, we could have two further possibilities:

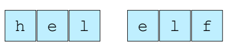
- `i`th character might match with `(j+1)`th character.
- `j`th character might match with `(i+1)`th character.

Thus, we take the max amongst all three of these possibilities (*line 9*). Look at the following visualization.

LCS("hel", "elf")

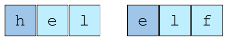
LCS("hel", "elf")

1 of 21



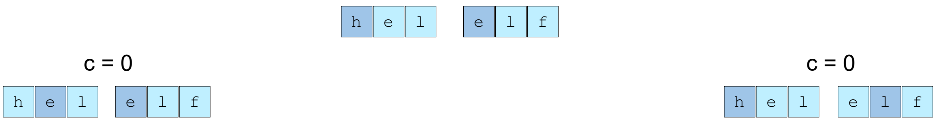
start with first characters of the both strings

2 of 21



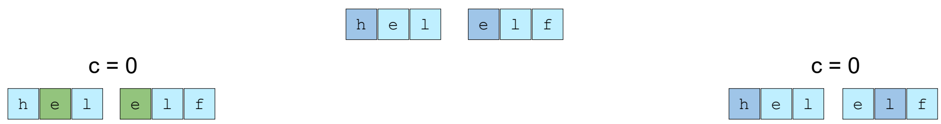
characters do not match; we will have two calls

3 of 21



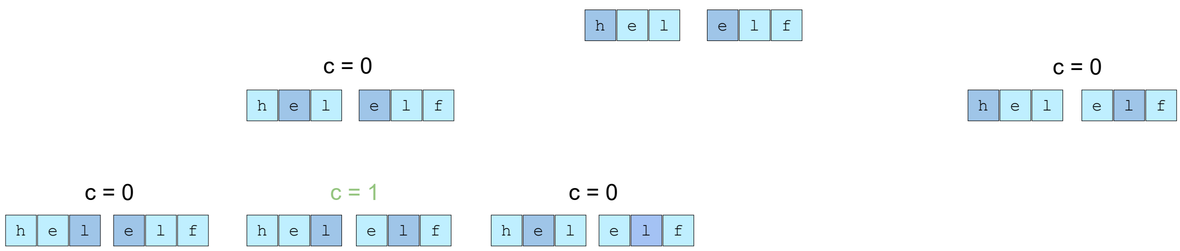
For the left recursive call from the root, we further make three calls, one additional when characters match. c is the count of characters matched so far

4 of 21



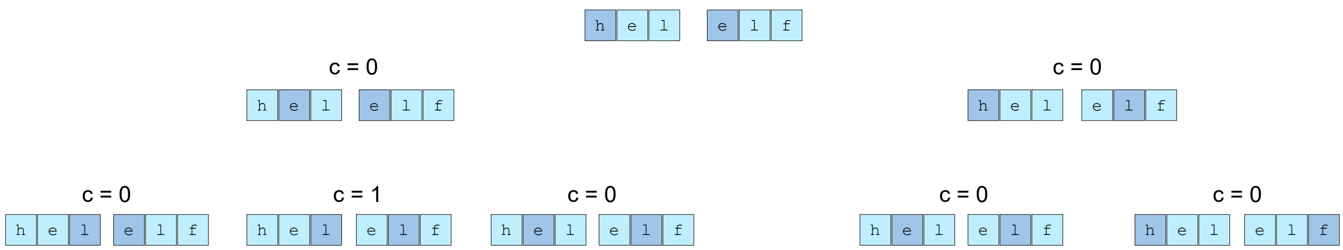
For the left recursive call from the root, we further make three calls, one additional when characters match. c is the count of characters matched so far

5 of 21

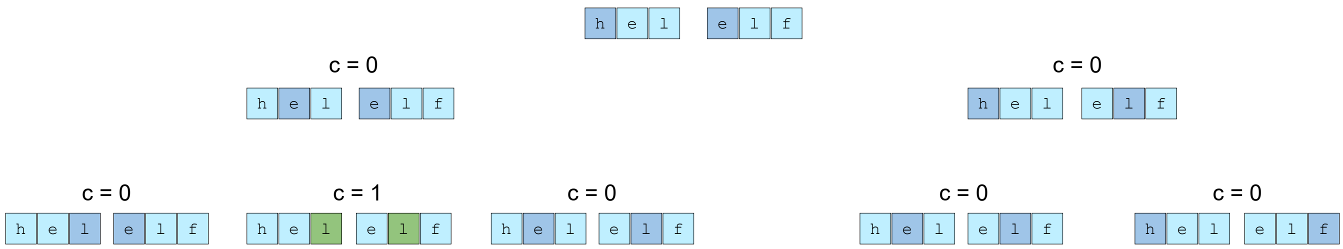


Since characters matched, increment c

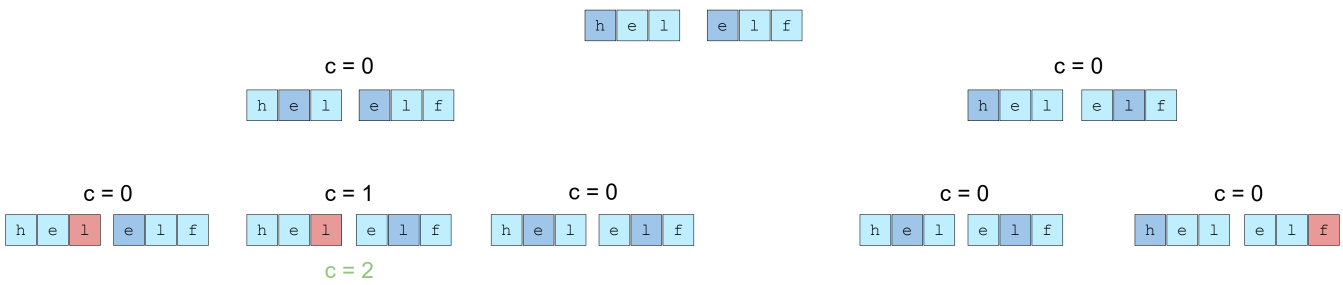
6 of 21



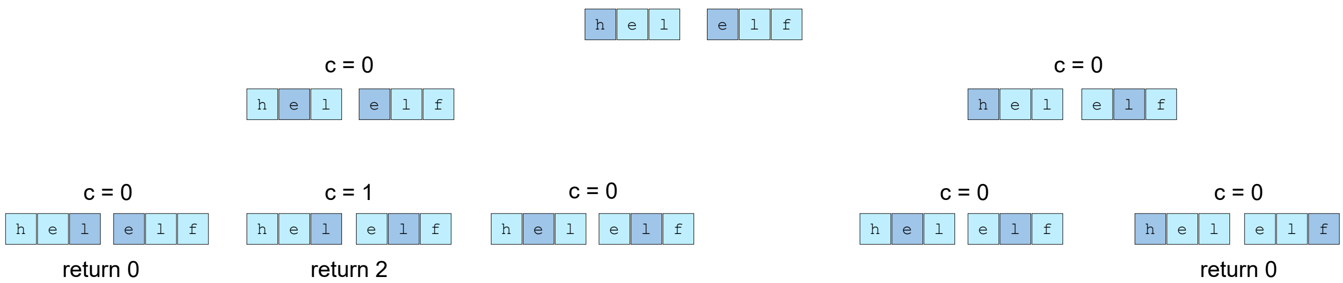
right side will have two cases



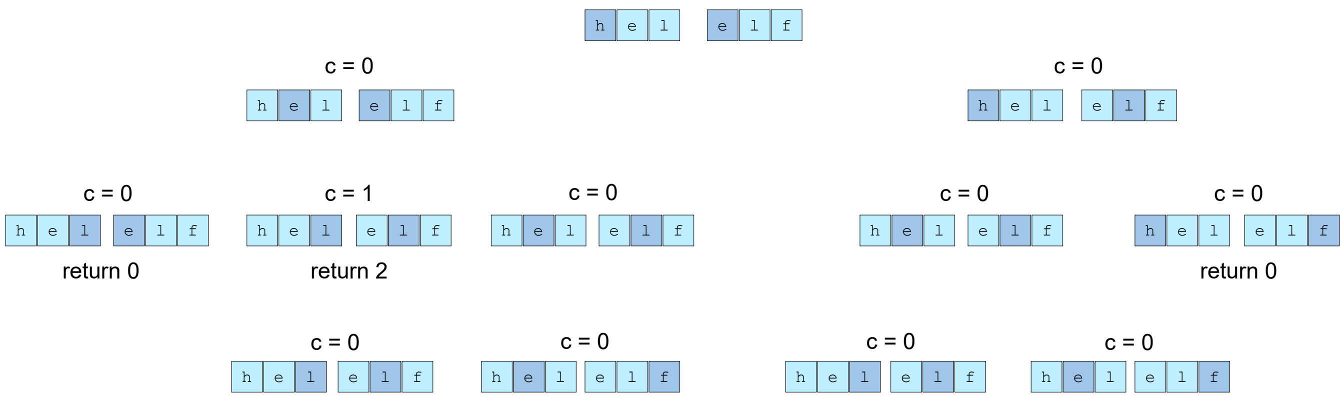
characters matching again, increment c for subsequent call



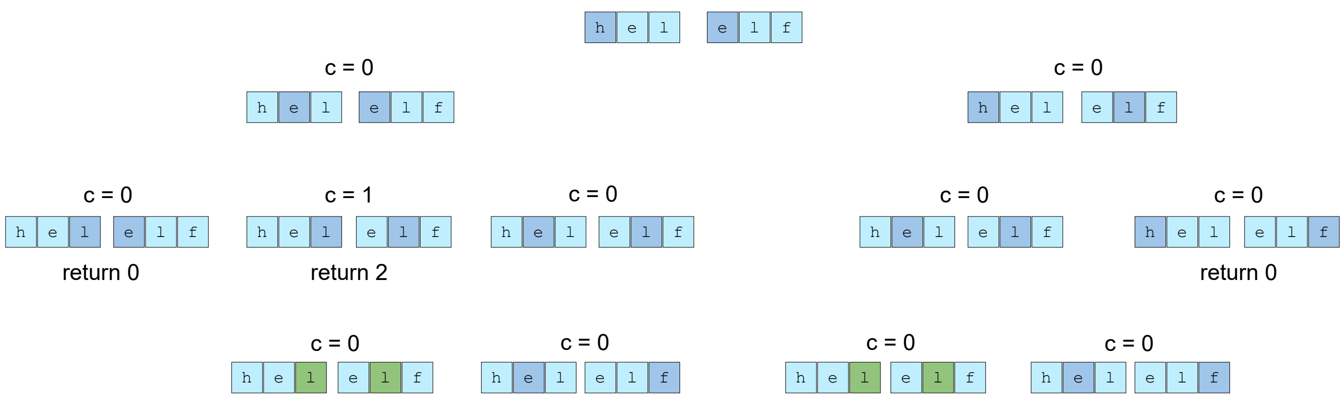
some calls have reached end of one string, leaving nothing to compare, these calls return value of c (updated if characters matched)



evaluate rest of the calls

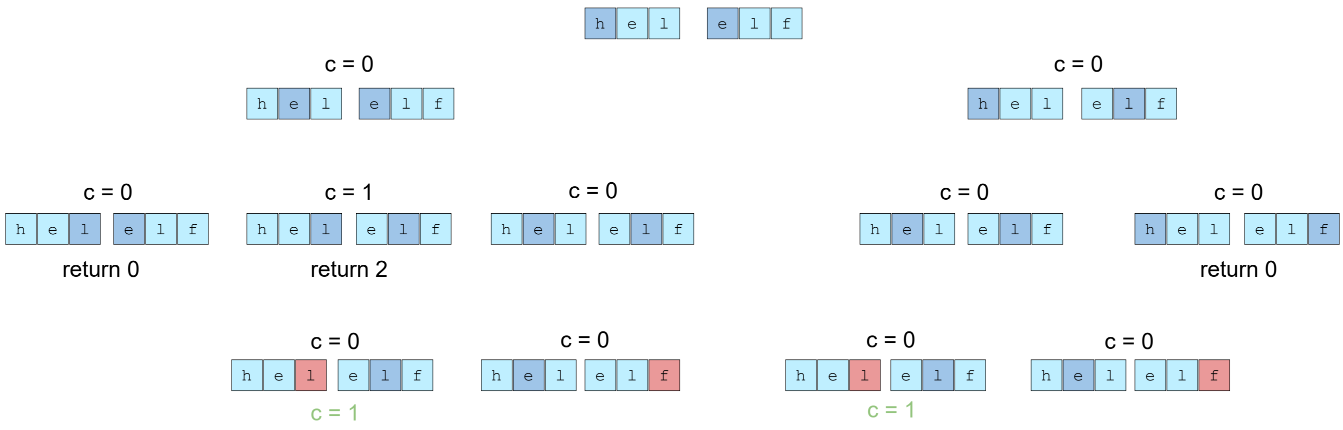


for the cases on left of the both calls, since we have character match we will have three cases; one for character match



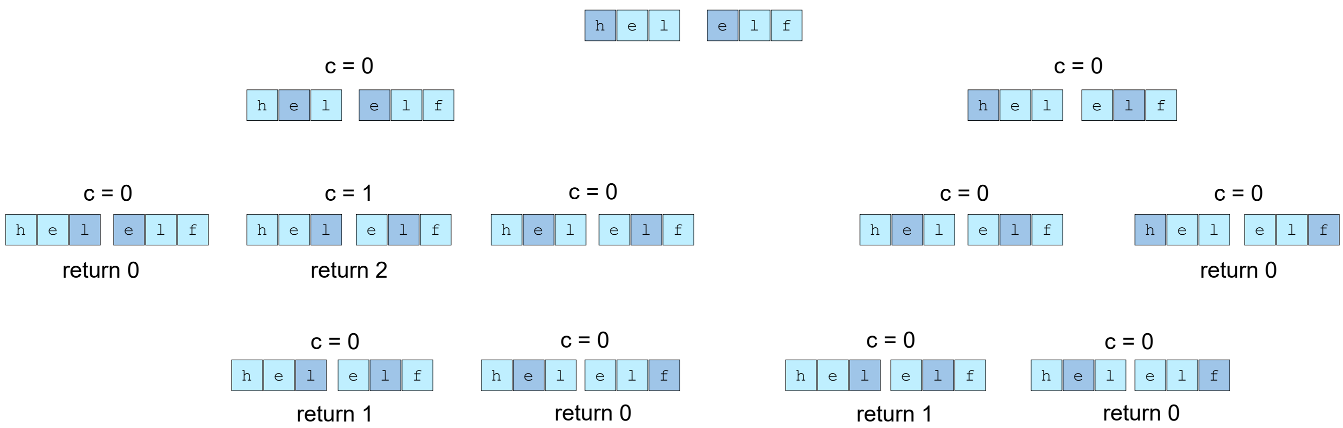
characters matching, increment c for subsequent call

12 of 21



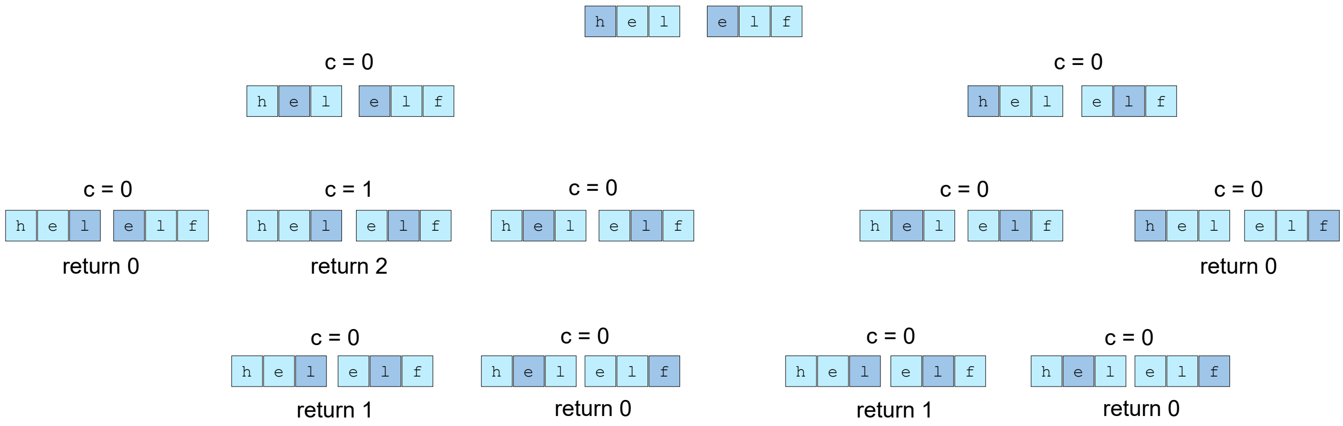
some calls have reached end of one string, leaving nothing to compare, these calls return value of c (updated if characters matched)

13 of 21



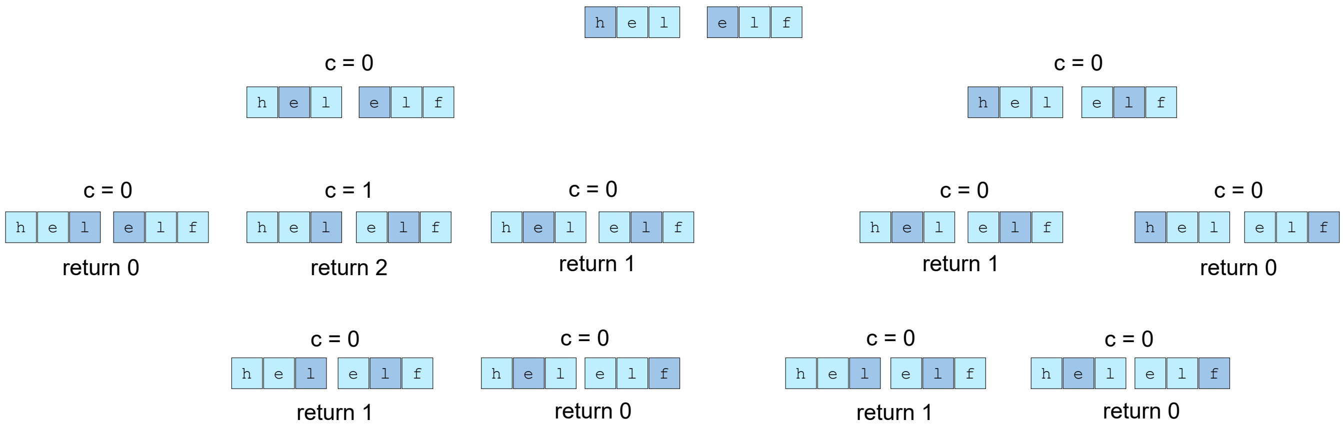
each call will return max value from all the recursive calls

14 of 21



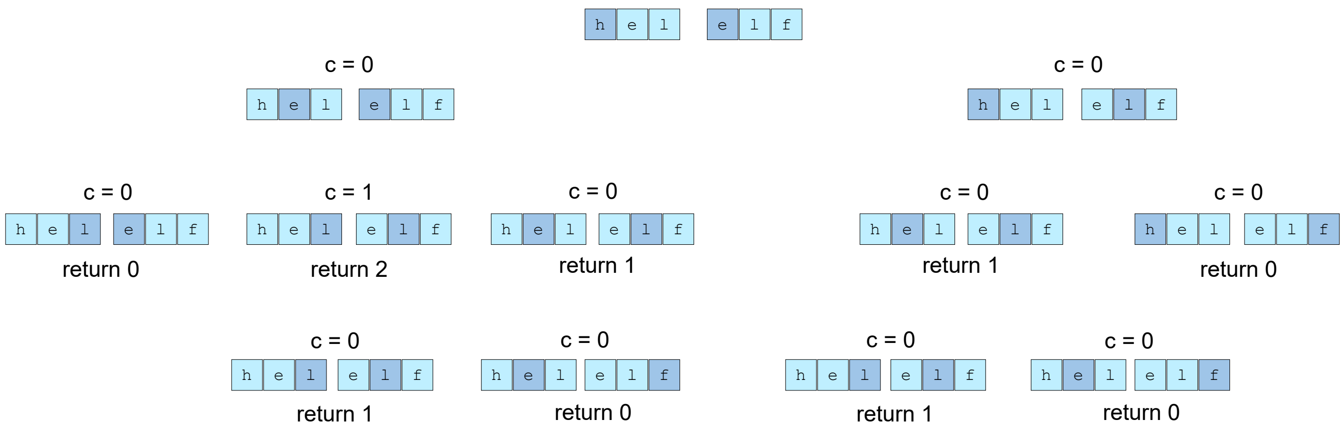
each call will return max value from all the recursive calls

15 of 21



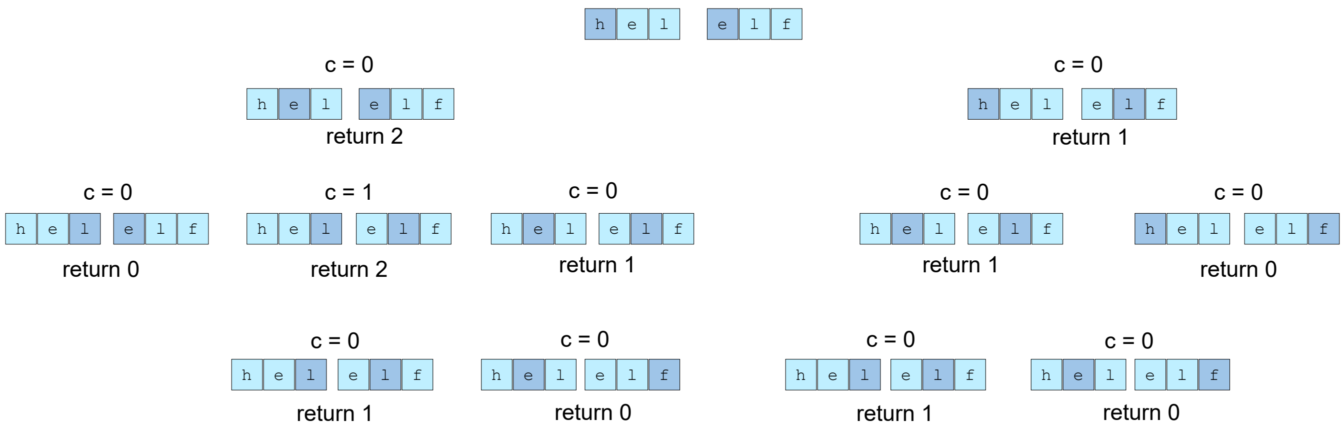
each call will return max value from all the recursive calls

16 of 21



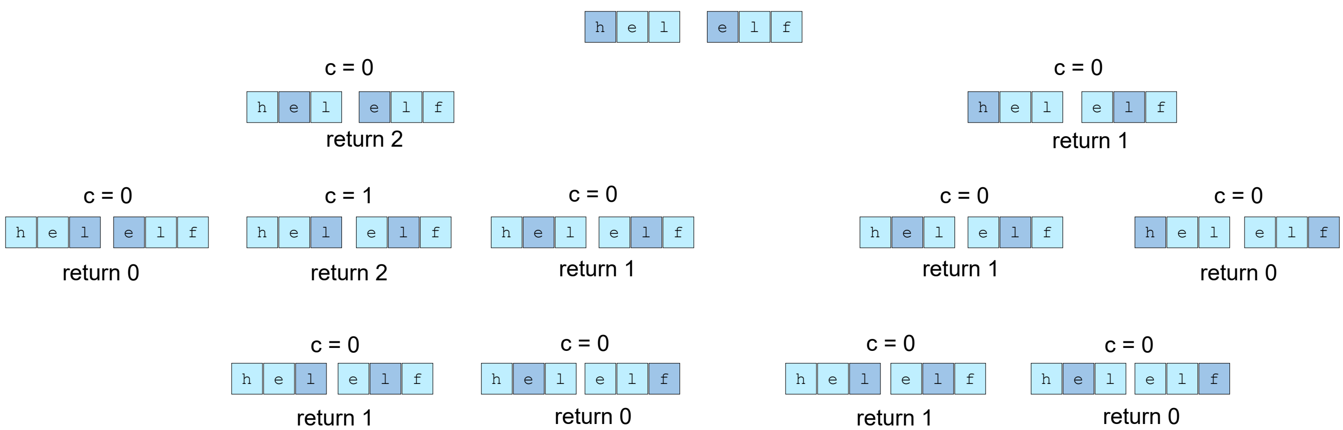
each call will return max value from all the recursive calls

17 of 21



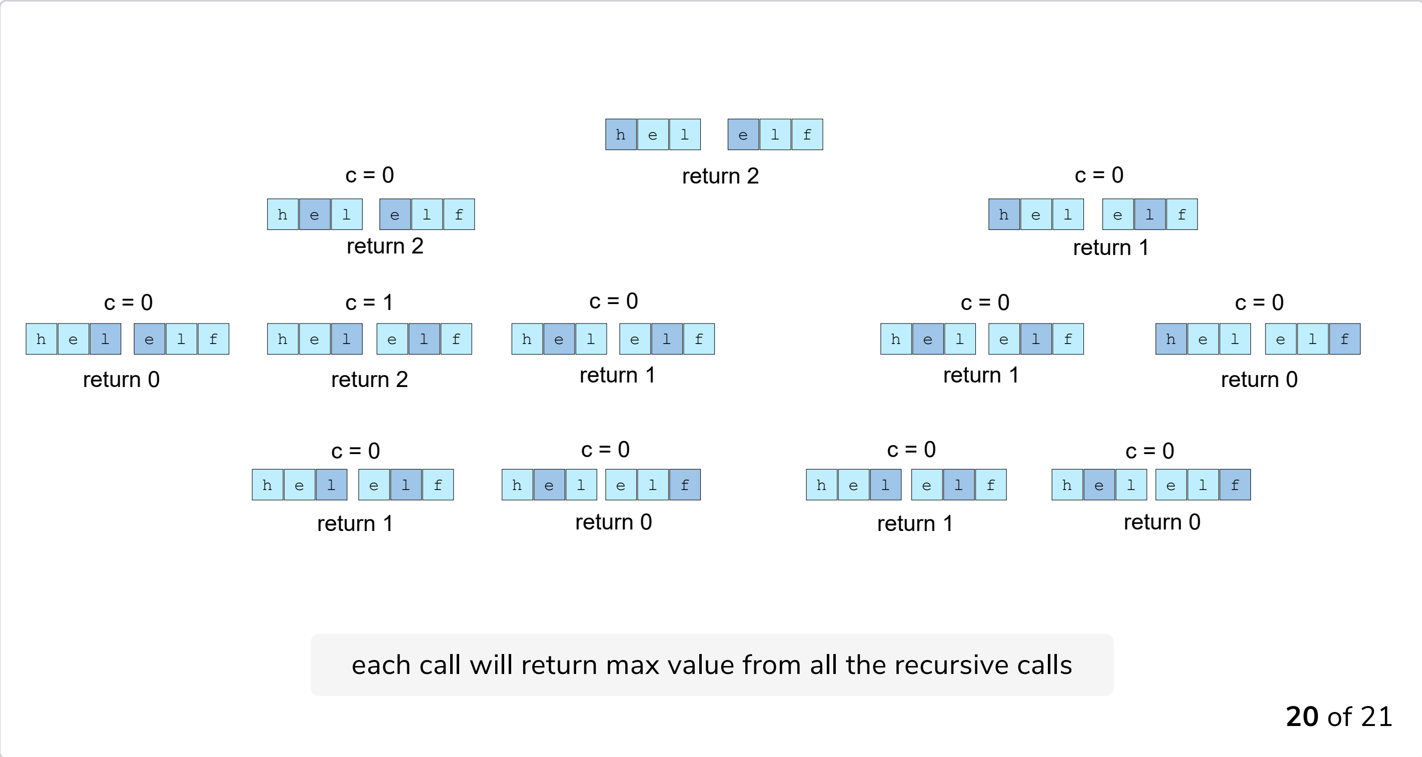
each call will return max value from all the recursive calls

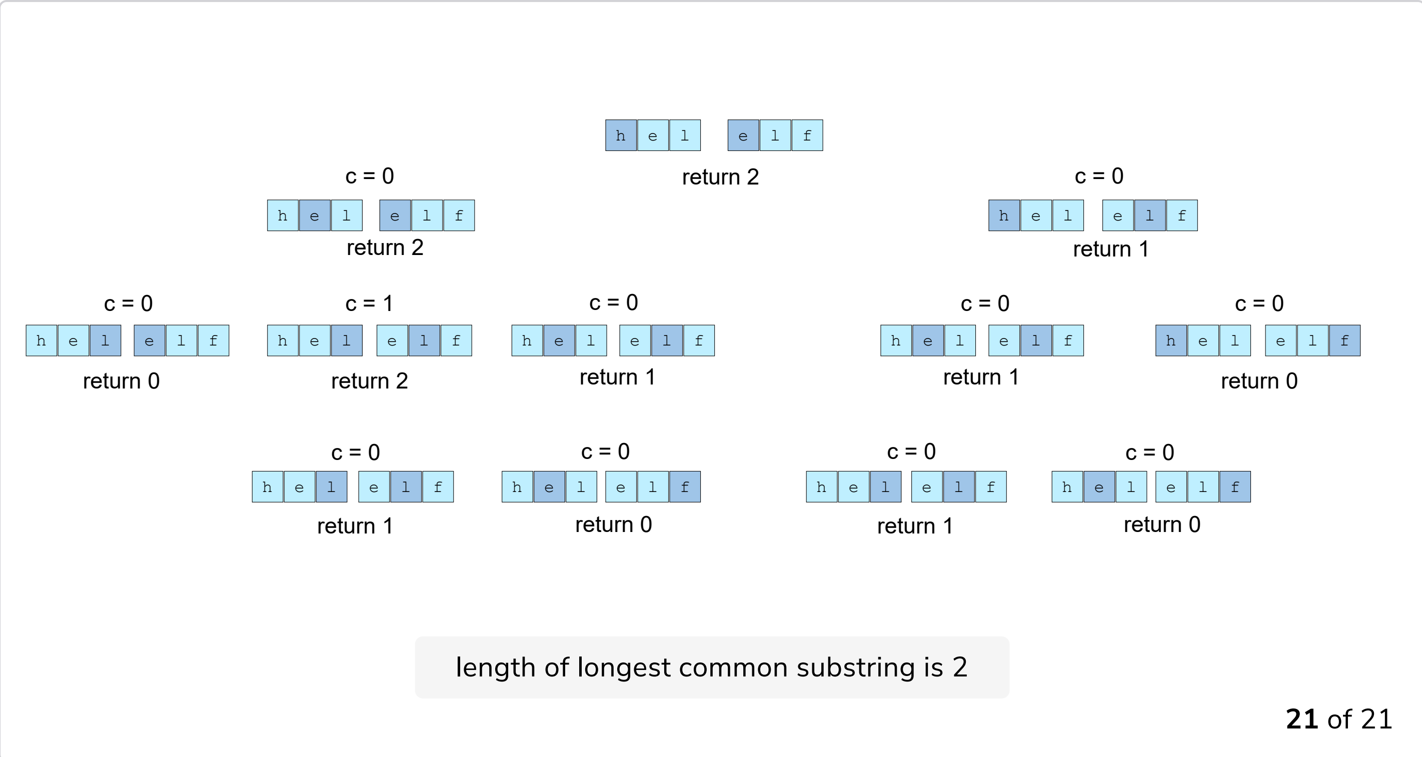
18 of 21



each call will return max value from all the recursive calls

19 of 21





Time complexity

The time complexity of this algorithm is exponential. At every step we can have at most three possibilities, thus, if the length of strings is `m` and `n`, we can have three calls, `mxn` times. Therefore, the overall time complexity is $\mathbf{O(3^{m+n})}$.

Solution 2: Top-down dynamic programming

Let’s see if this problem satisfies both conditions of dynamic programming.

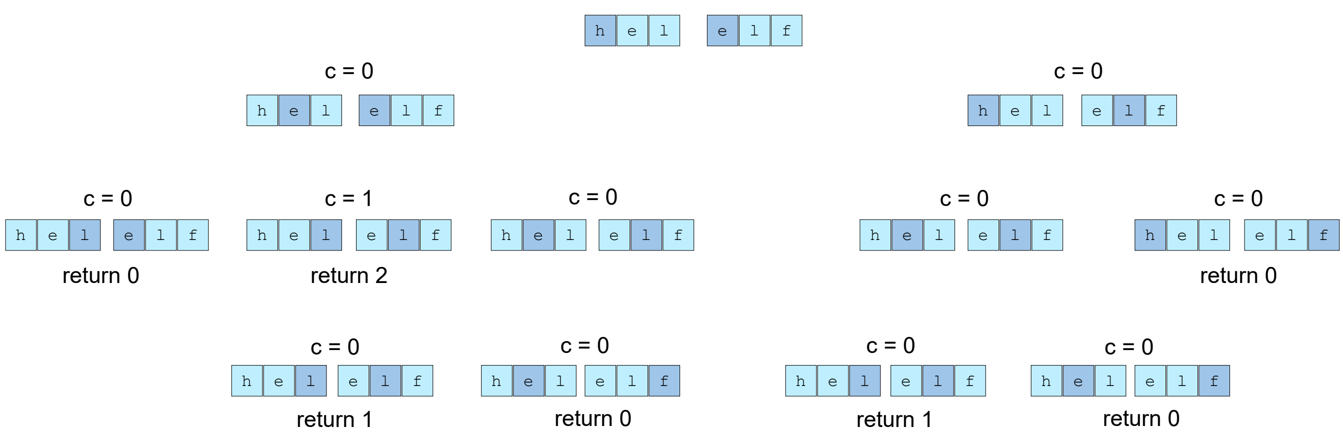
Optimal substructure

If we have a pair of strings, `str1` and `str2`, with lengths of `n` and `m`. We could construct their optimal solution if we had answers to the following three subproblems:

- The solution of substrings of `str1` and `str2` formed by removing the first characters. (i+1, j+1)
- The solution of the substring of `str1` formed by removing its first character and `str2` as it is. (i+1, j)
- The solution of the substring of `str2` formed by removing its first character and `str1` as it is. (i, j+1)

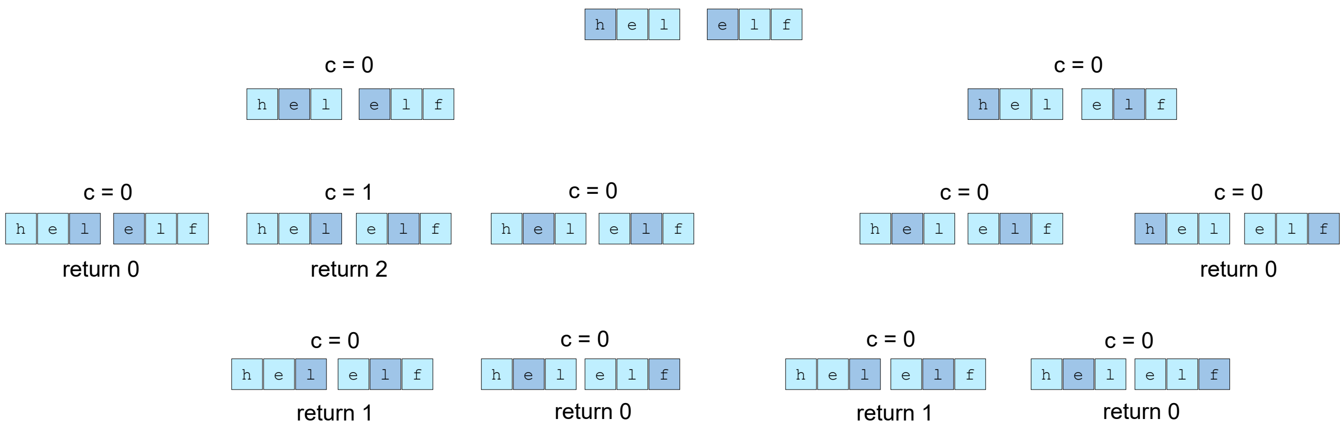
Overlapping subproblem

If we look at the visualization given above, we can see some overlapping problems. The following visualization highlights these overlapping subproblems.



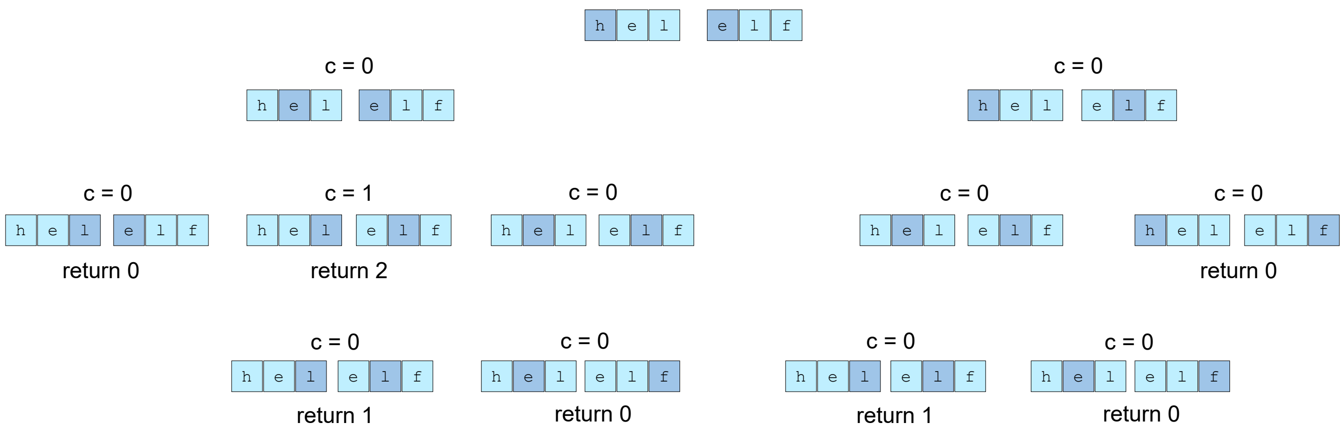
Look at the repeating subproblems

1 of 3



we could save recomputing by saving the answer to this subproblem

2 of 3



3 of 3

Let’s look at the top-down dynamic programming solution where we use memoization.

```
def lcs_(str1, str2, i, j, count, memo):
  # base case of when either of string has been exhausted
  if i >= len(str1) or j >= len(str2):
    return count
  # check if result available in memo
  if (i,j,count) in memo:
    return memo[(i,j,count)]
  c = count
   # if i and j character matches, increment the count and compare the rest of the strings
```

```
    if str1[i] == str2[j]:
      c = lcs_(str1, str2, i+1, j+1, count+1, memo)
    # compare str1[1:] with str2, str1 with str2[1:], and take max of current count and these two re
    # memoize the result
    memo[(i,j,count)] = max(c, lcs_(str1, str2, i+1, j, 0, memo), lcs_(str1, str2, i, j+1, 0, memo))
    return memo[(i,j,count)]

def lcs(str1, str2):
    memo = {}
    return lcs_(str1, str2, 0, 0, 0, memo)

print(lcs("hel", "elf"))

# testing with longer strings
import random
import string

st1 = ''.join(random.choice(string.ascii_lowercase) for _ in range(40))
st2 = ''.join(random.choice(string.ascii_lowercase) for _ in range(60))
print(lcs(st1, st2+st1))
```

Explanation

The only change in this solution is the memoization of results. We check in the `memo` before evaluating something (*lines 6-7*) and store the result in `memo` after the evaluation (*line 14*). An important detail here is that we have three different parameters, `i`, `j`, and `count`, that uniquely define every subproblem. Thus, we use all three in the process of memoization.

Time and space complexity

Let's look at this in the context of our keyspace, i.e., tuples of `i`, `j`, and `count`. If the lengths of our strings are m and n, where m is the length of the larger string, i.e., $n \leq m$, then the total keyspace mapped by this tuple would be mn^2. `i` could go from 0 to m, `j` could go from 0 to n, and `count` would also be able to go from 0 to n because the length of the common substring cannot be greater than the length of the smaller of the two strings. Thus, we have mn^2 unique problems to evaluate and store in the worst case making the time and space complexity **O(mn^2)**.

This solution is equivalent to finding all the substrings of the smaller string, which are n^2, and then finding them in the larger string of size m. The time complexity of such a solution would also be O(mn^2).

Solution 3: Bottom-up dynamic programming

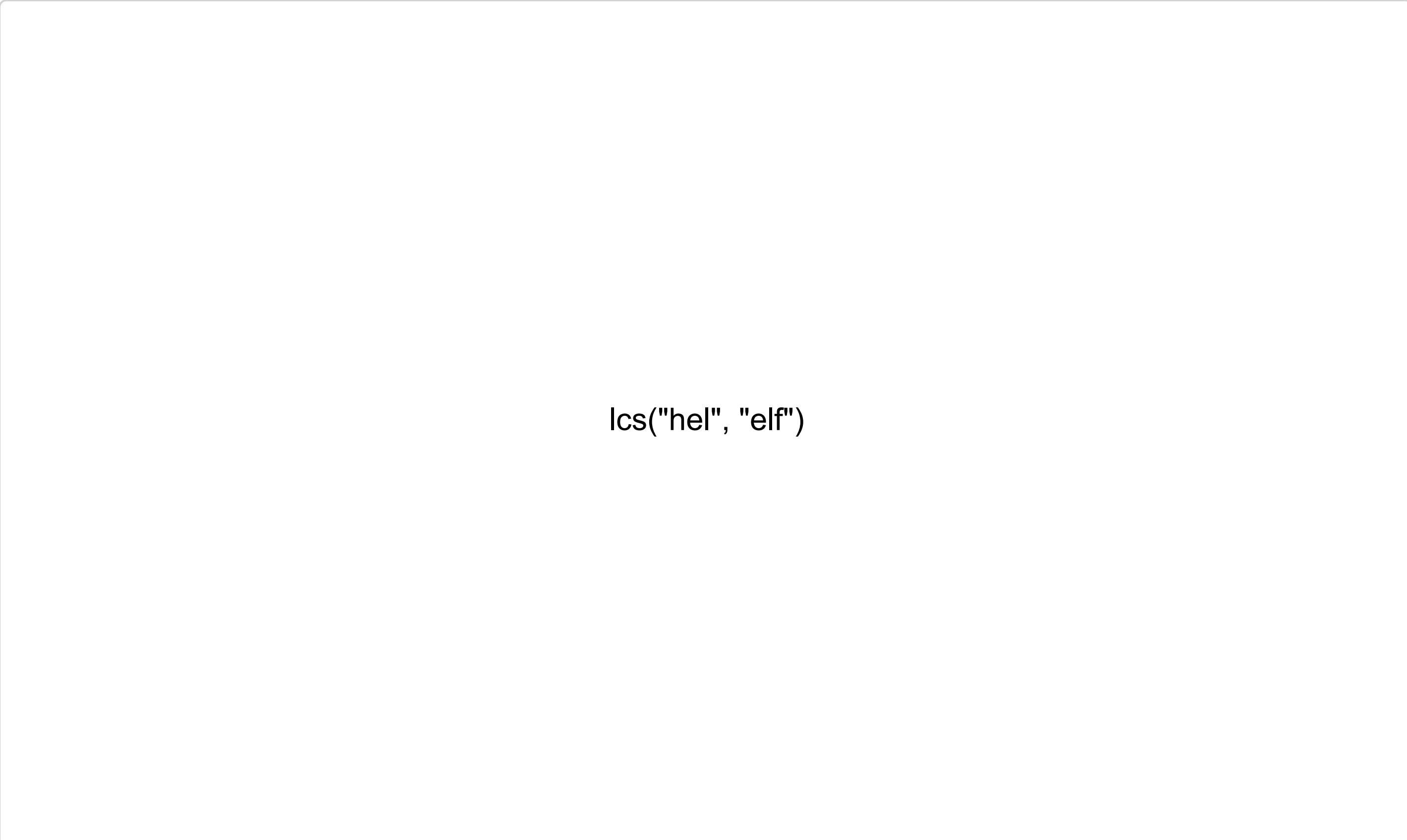
Let's look at the non-recursive implementation of this algorithm. The major bit here is tabulation. If we are able to tabulate the problem properly we will be able

to solve the problem with bottom-up dynamic programming. If you look at the visualization of the recursive algorithm, you would notice how each recursive call takes count of previously matched consecutive characters of both strings. This gives us some idea of what we need to tabulate. We can have a 2-d array of size `mxn`, where any position is given by `i`th row and `j`th column gives us the maximum count of character matches between the first string up to `i`th position and the second string up to `j`th position. Look at the implementation of the algorithm below:

```
def lcs(str1, str2):
  n = len(str1)   # length of str1
  m = len(str2)   # length of str1

  dp = [[0 for j in range(m+1)] for i in range(n+1)]  # table for tabulation of size m x n
  maxLength = 0   # to keep track of longest substring seen

  for i in range(1, n+1):           # iterating to fill table
    for j in range(1, m+1):
      if str1[i-1] == str2[j-1]:    # if characters at this position match,
        dp[i][j] = dp[i-1][j-1] + 1 # add 1 to the previous diagonal and store it in this diagonal
        maxLength = max(maxLength, dp[i][j])  # if this substring is longer, replace it in maxleng
      else:
        dp[i][j] = 0 # if character don't match, common substring size is 0
  return maxLength

stressTesting = True  # to only check if your recursive solution is correct, set it to false
testForBottomUp = True   # to test a top down implementation set it to false

print(lcs("hel", "elf"))

# testing with longer strings
import random
import string

st1 = ''.join(random.choice(string.ascii_lowercase) for _ in range(400))
st2 = ''.join(random.choice(string.ascii_lowercase) for _ in range(600))
print(lcs(st1, st2+st1))
```

Explanation

We start off by constructing a 2-d array of size `mxn` for tabulation where `n` is the size of `str1` and `m` is the size of `str2`. We initialize this 2-d array to zeros (*line 5*). Now we start filling the array starting from position 1,1. Each entry in this array tells us the count of the last characters matched between both strings up to `i`th and `j`th positions in `str1` and `str2` respectively. So for example, if `dp[3][4]` returns `2`, this means the last two characters of `str1` and `str2` up to positions `3` and `4` match, i.e., `str1[2] = str2[3]` and `str1[1] = str2[2]`. By the end of the execution of this algorithm, we will have the length of the longest common substring in the variable `maxLength` since we take its max with every entry of the `dp` table. The following visualization shows a dry run of this algorithm.



lcs("hel" , "elf")

1 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0			
e	2	0			
l	3	0			

start from index 1,1 to fill the dp table

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0			
e	2	0			
l	3	0			

if the characters match, add 1 to the previous diagonal else put a 0

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0			
e	2	0			
l	3	0			

if the characters match, add 1 to the previous diagonal else put a 0

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0		
e	2	0			
l	3	0			

since characters do not match put a 0

5 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0		
e	2	0			
l	3	0			

moving to next position

6 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	
e	2	0			
l	3	0			

since characters do not match put a 0

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	
e	2	0			
l	3	0			

moving to next position

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0			
l	3	0			

since characters do not match put a 0

9 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0			
l	3	0			

moving to next position

10 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0			
l	3	0			

since characters are matching, add 1 to diagonal

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1		
l	3	0			

since characters are matching, add 1 to diagonal

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1		
l	3	0			

moving to next position

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	
l	3	0			

since characters do not match put a 0

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	
l	3	0			

moving to next position

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0			

since characters do not match put a 0

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0			

moving to next position

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0		

since characters do not match put a 0

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0		

moving to next position

19 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0		

since characters are matching, add 1 to diagonal

20 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0	2	

since characters are matching, add 1 to diagonal

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0	2	

moving to next position

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0	2	0

since characters do not match put a 0

23 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0	2	0

dp table filled

24 of 25

			e	l	f
		0	1	2	3
	0	0	0	0	0
h	1	0	0	0	0
e	2	0	1	0	0
l	3	0	0	2	0

return the highest value from table i.e. 2

25 of 25

Time and space complexity

If you look at the figure, you will see we are only filling up a table of size `mxn` which entails a time complexity of **O(nm)**. Similarly, as we can see that the size of the table is `mxn`, the space complexity would also be **O(nm)**.

Solution 4: Space optimized bottom-up dynamic programming

If you notice in the above illustration, when filling up a row, we only require the row above it and not any previous row. This means we only need to maintain the state of the last row instead of all `m` rows. Thus, we can reduce space complexity from *O(mn)* to **O(n)**.

```
def lcs(str1, str2):
  n = len(str1)   # length of str1
  m = len(str2)   # length of str1

  dp = [0 for i in range(n+1)]  # table for tabulation, only maintaining state of last row
```

```
  maxLength = 0    # to keep track of longest substring seen

  for j in range(1, m+1):           # iterating to fill table

    thisrow = [0 for i in range(n+1)] # calculate new row (based on previous row i.e. dp)
    for i in range(1, n+1):
      if str1[i-1] == str2[j-1]:    # if characters at this position match,
        thisrow[i] = dp[i-1] + 1 # add 1 to the previous diagonal and store it in this diagonal
        maxLength = max(maxLength, thisrow[i])  # if this substring is longer, replace it in maxle
      else:
        thisrow[i] = 0 # if character don't match, common substring size is 0
    dp = thisrow   # after evaluating thisrow, set dp equal to this row to be used in the next ite
  return maxLength

stressTesting = True  # to only check if your recursive solution is correct, set it to false
testForBottomUp = True   # to test a top down implementation set it to false

print(lcs("hel", "elf"))

# testing with longer strings
import random
import string

st1 = ''.join(random.choice(string.ascii_lowercase) for _ in range(400))
st2 = ''.join(random.choice(string.ascii_lowercase) for _ in range(600))
print(lcs(st1, st2+st1))
```

Explanation

Since we only need one row, we save it in `dp` (*line 5*) and use it to evaluate the next row (*lines 10-15*). Once the next row has been evaluated in the form of `thisrow`, update `dp` and store this newly computed row in it (*line 16*) so it can be used in the next iteration.

Time and space complexity

The time complexity of this solution remains the same as before since we still need to compute the values of all `m` rows, each of size `n`. Thus, the time complexity remains **O(nm)**. Space complexity, however, reduces to **O(n)** since we only keep the state of one row now.

In the next lesson, we will see a comparison between bottom-up dynamic programming and top-down dynamic programming.