Solution Review: Longest Common Subsequence

In this lesson, we will go over some solutions to the classic dynamic programming problem of the longest common subsequence.

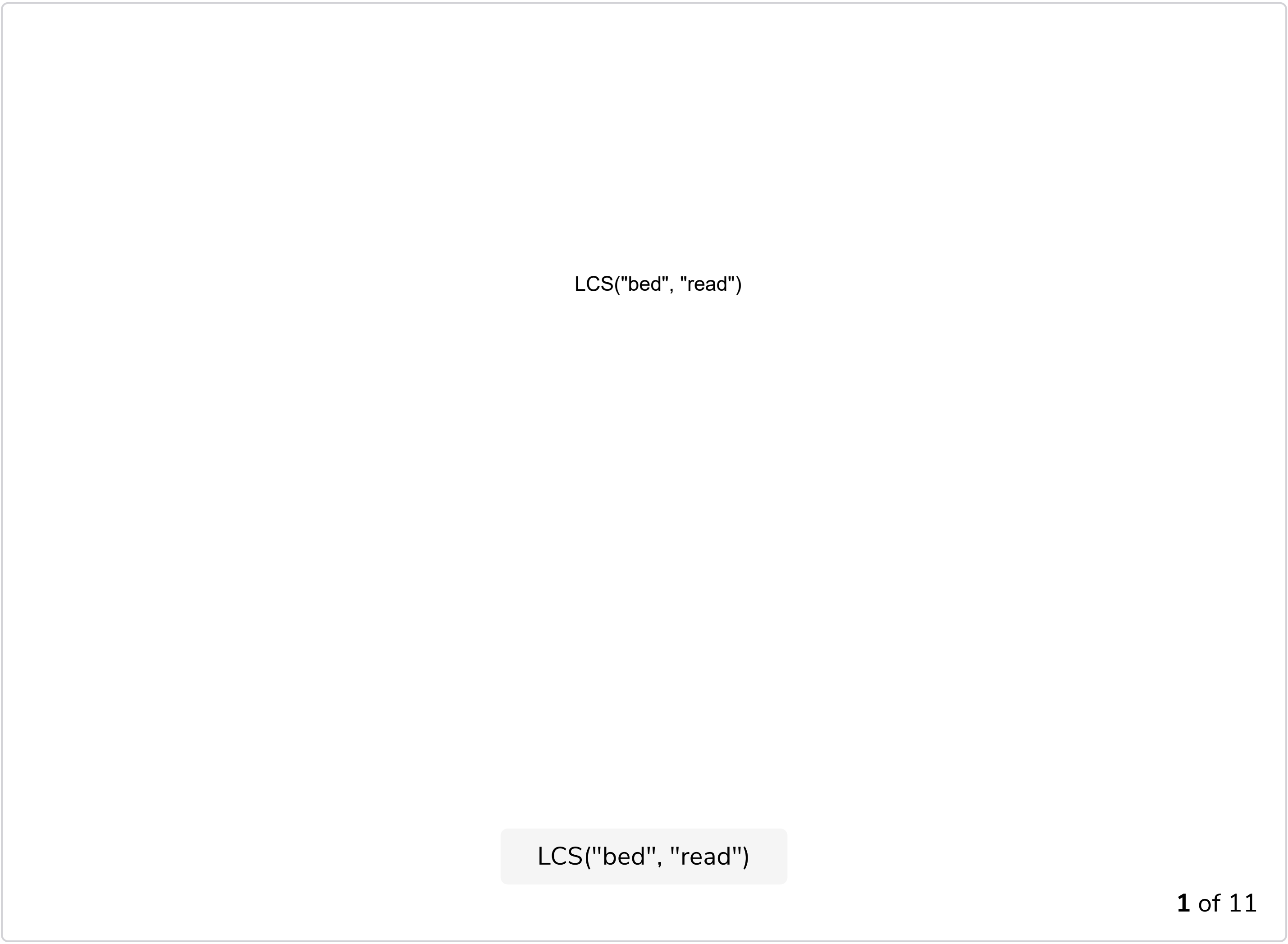
We'll cover the following

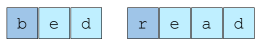
Solution 1: Simple recursion

```python
# helper function with updated signature: i is current index in str1, j is current index in str2
def LCS_(str1, str2, i, j):
    if i == len(str1) or j == len(str2): # base case
        return 0
    elif str1[i] == str2[j]:  # if current characters match, increment 1
        return 1 + LCS_(str1, str2, i+1, j+1)
    # else take max of either of two possibilities
    return max(LCS_(str1, str2, i+1, j), LCS_(str1, str2, i, j+1))

def LCS(str1, str2):
    return LCS_(str1, str2, 0, 0)

print(LCS("bed", "read"))
```

Explanation

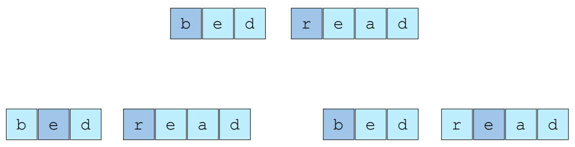
The algorithm is pretty intuitive, we start comparing `str1` and `str2` at the first characters. If the characters match we can simply move one position ahead in both `str1` and `str2` (*lines 5-6*). If the characters do not match, we need to find the possibility that yields maximum count from either moving one position ahead in `str1` or moving one position ahead in `str2` (*line 8*). As our base case, we return 0 when either of our strings end is reached (*lines 3-4*).





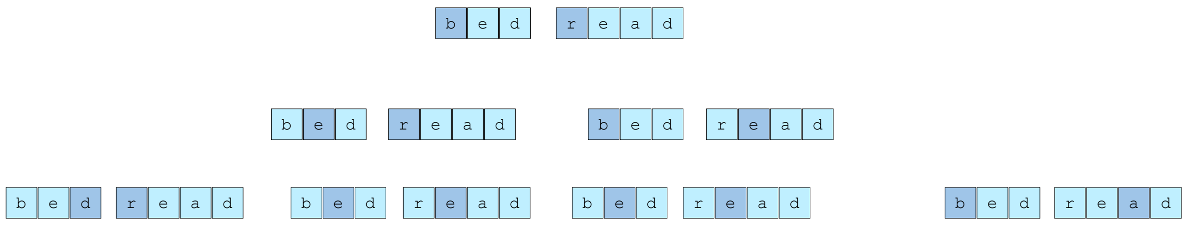
We will start from the first character of the both strings

2 of 11



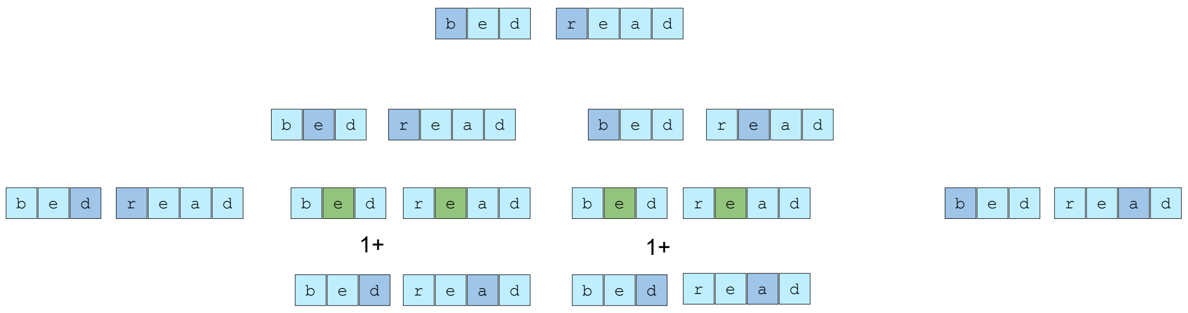
since characters do not match, there will be two recursive calls

3 of 11



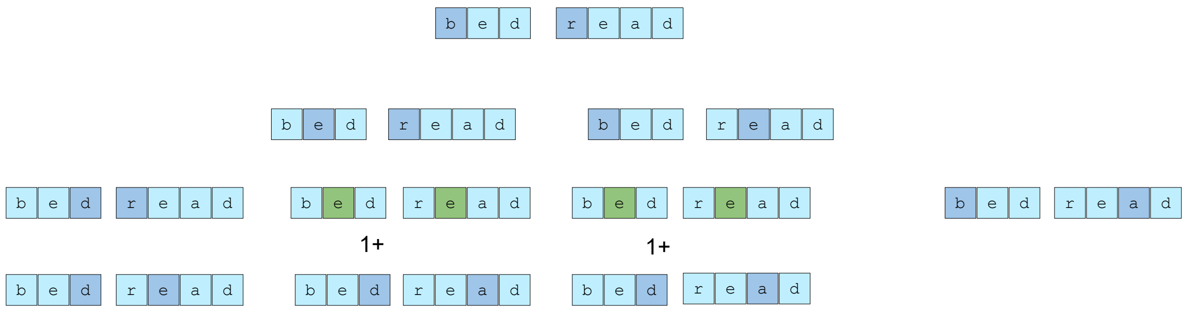
since characters do not match again, there will be two recursive calls for each of them

4 of 11



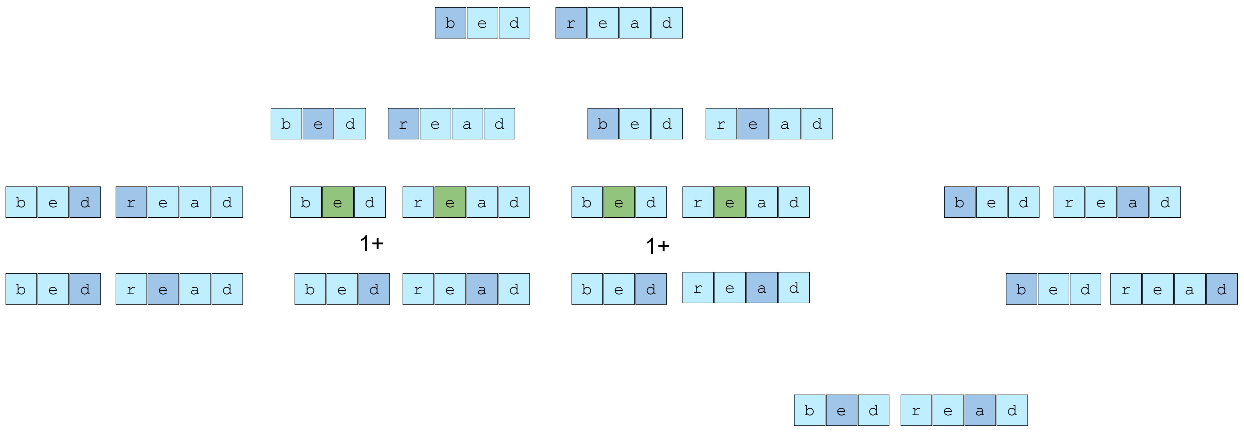
Now, we have the case where characters are matching, this time we will only have one recursive call with 1 incremented

5 of 11



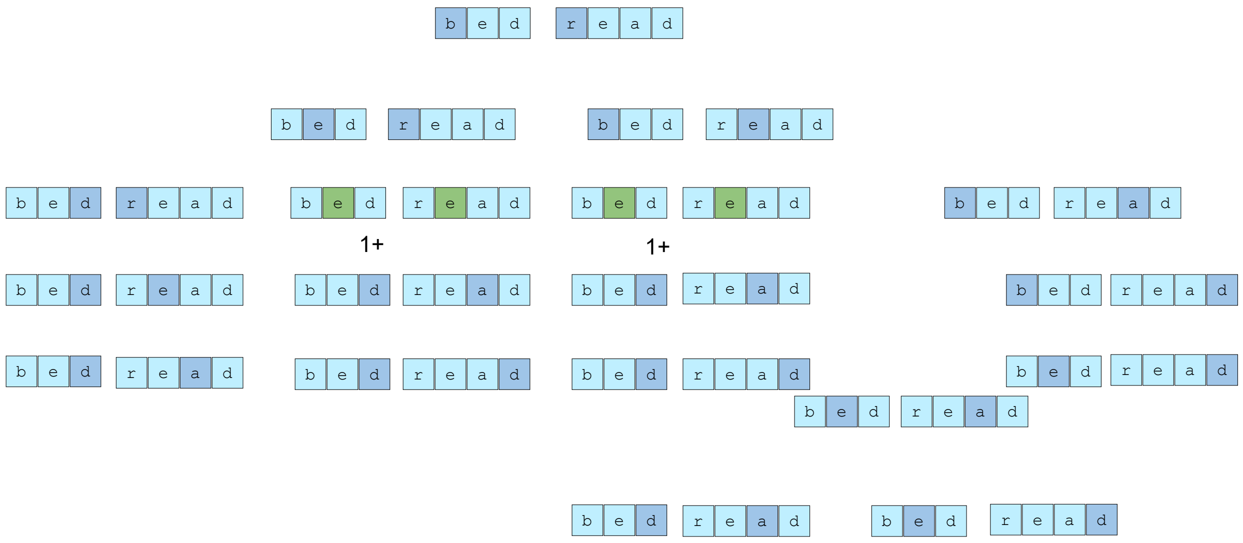
On the left call, we are on the last character of first string, although our algorithm will make two calls but we know the call where a string has finished returns in 0, so for simplicity we will have only one call

6 of 11



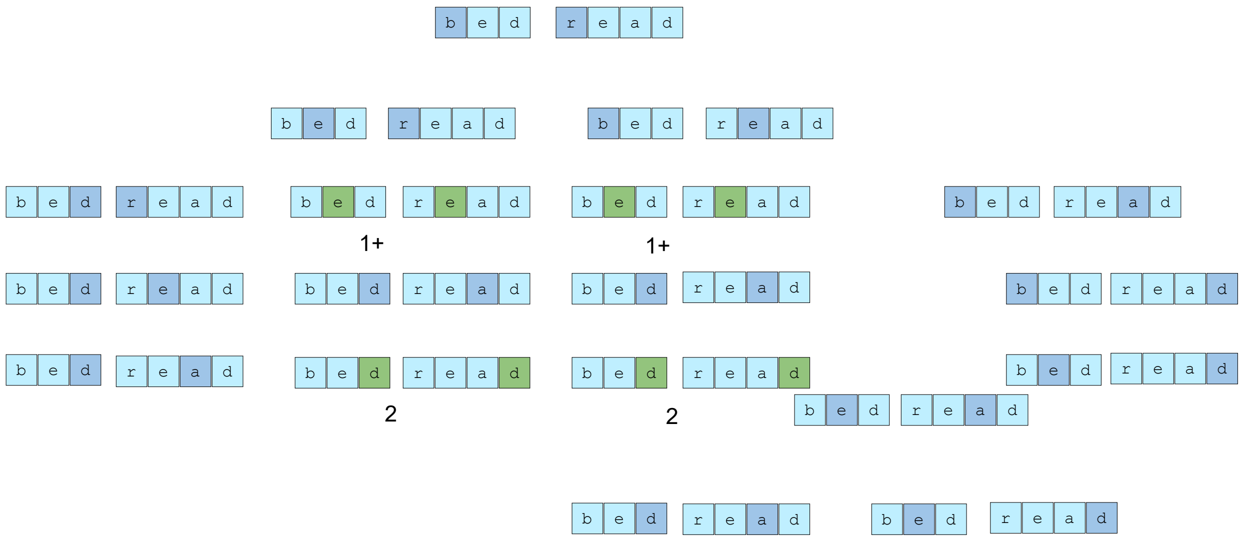
right call will be as before

7 of 11

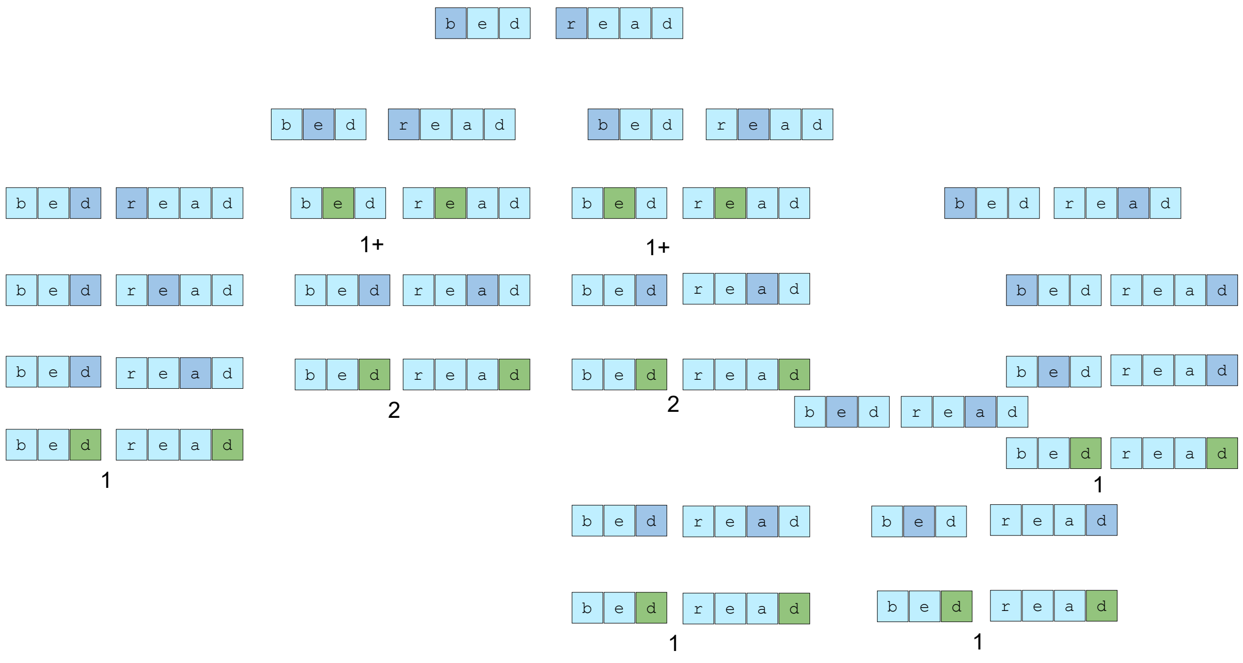


Again no character matches, so we will have two calls, (one call for cases where we are at end of one string)

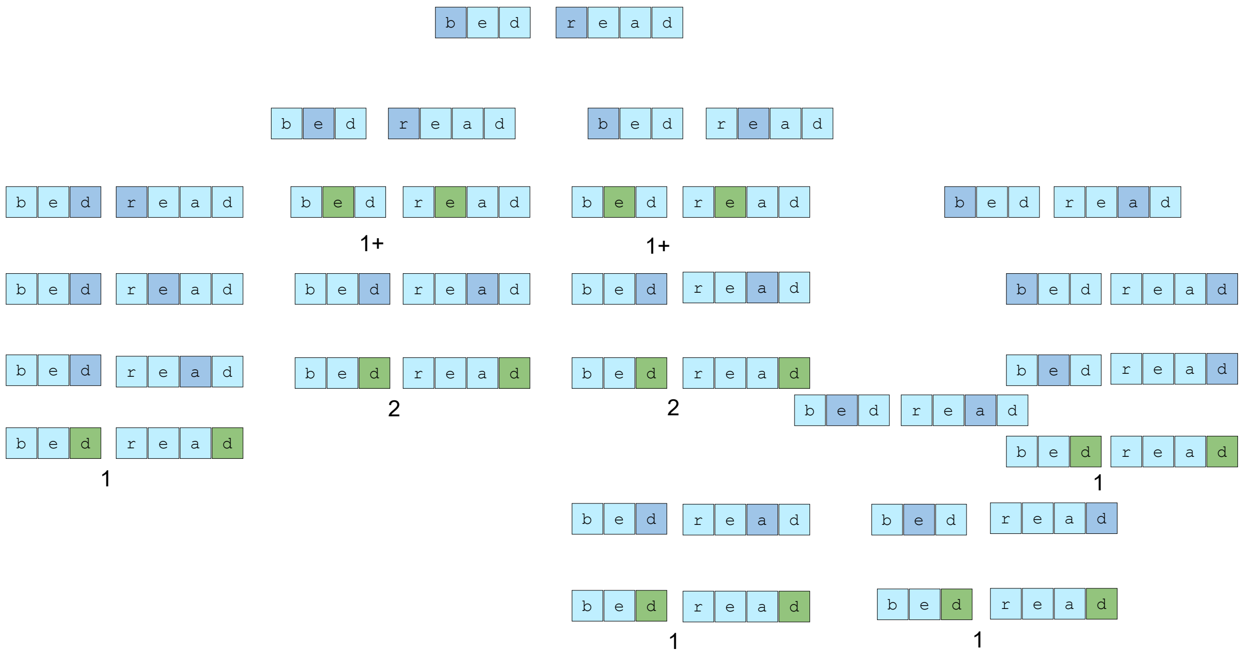
8 of 11



Here, we have the case of matching characters, but we have run out of characters to match as well so we will end up with 2 on these calls



If we continue this way, we will end up with this recursion tree



Thus optimal answer we get is 2

Time complexity

At each point we can have a max of two possibilities, we can move one character ahead in either string. If lengths of strings are m and n respectively, we will have an overall time complexity bounded by **O(2^{m+n})**.

Solution 2: Top-down dynamic programming

Let’s take a look at how this problem satisfies both the prerequisites 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 following three subproblems:

The Case of Match:

- The solution of substrings of `str1` and `str2` formed by removing the first characters. (i+1, j+1)

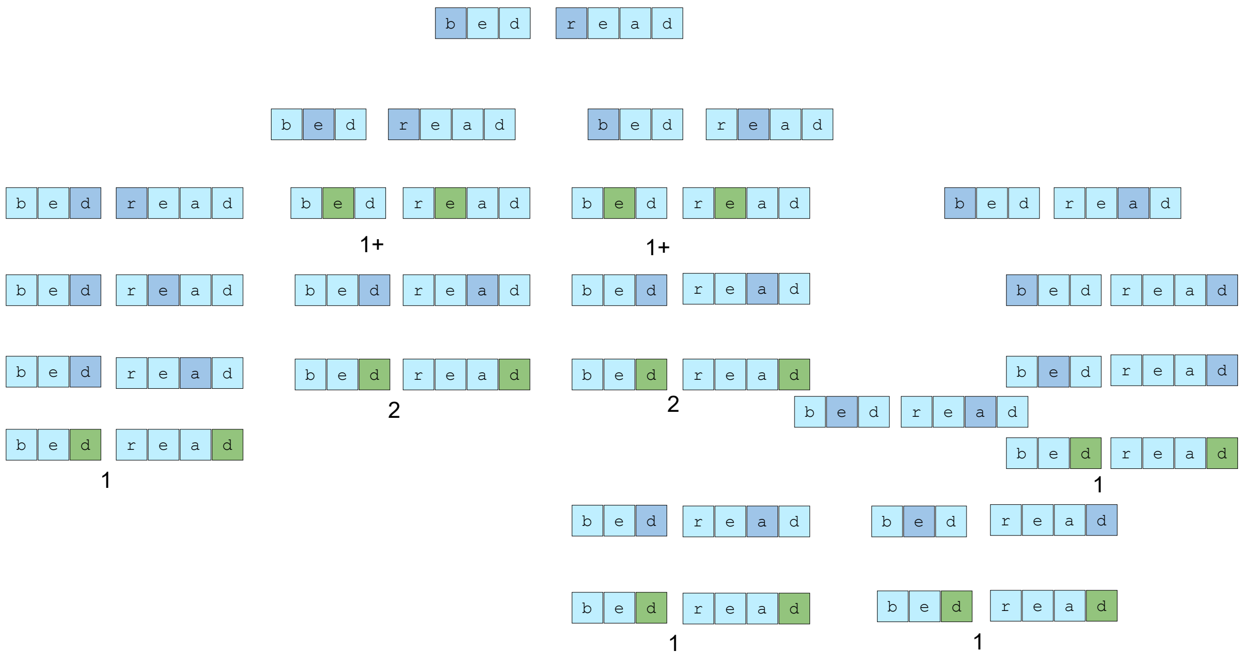
The Case of Mismatch:

- 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)

We only need the optimal answer to these three subproblems to get the optimal answer to the main problem; thus, the property of optimal substructure is satisfied.

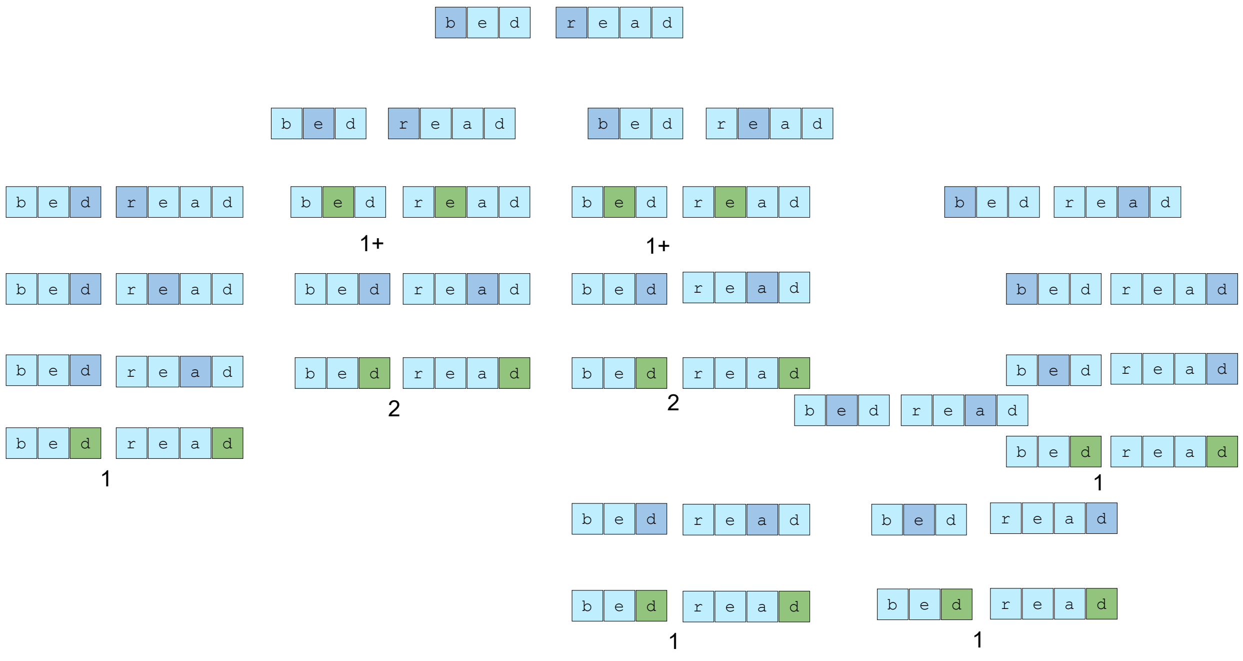
Overlapping subproblems

Let’s revisit the above visualization to see some overlapping subproblems.

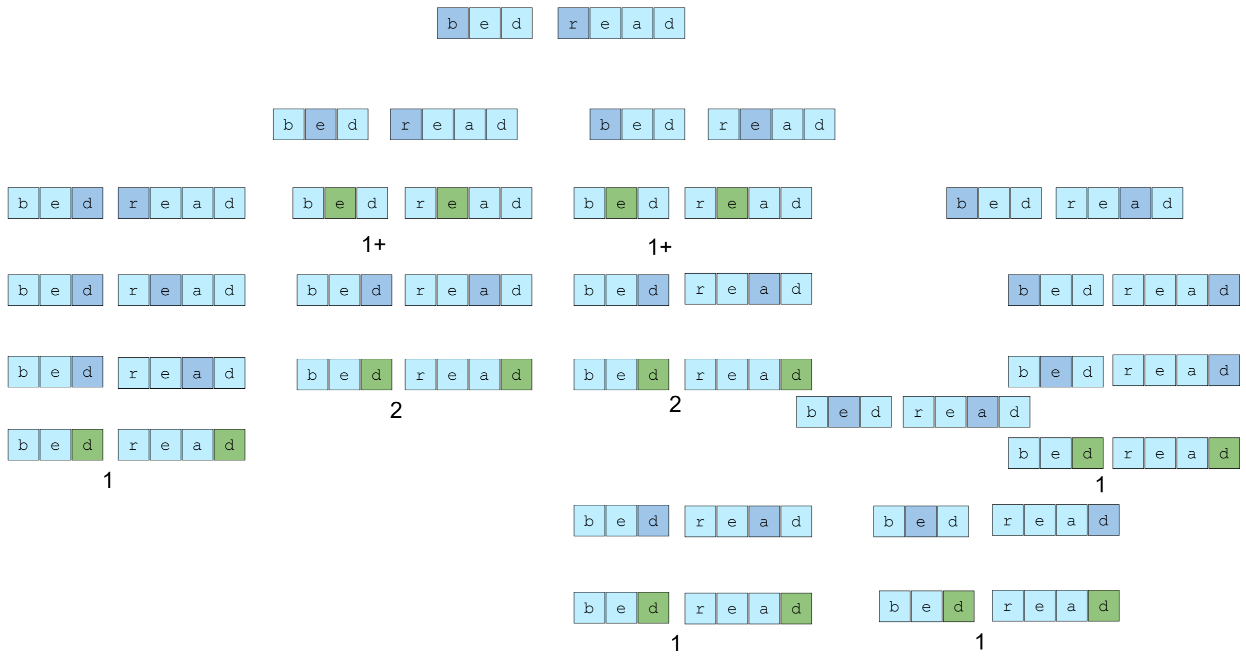


Let's try to find some repeating subproblems

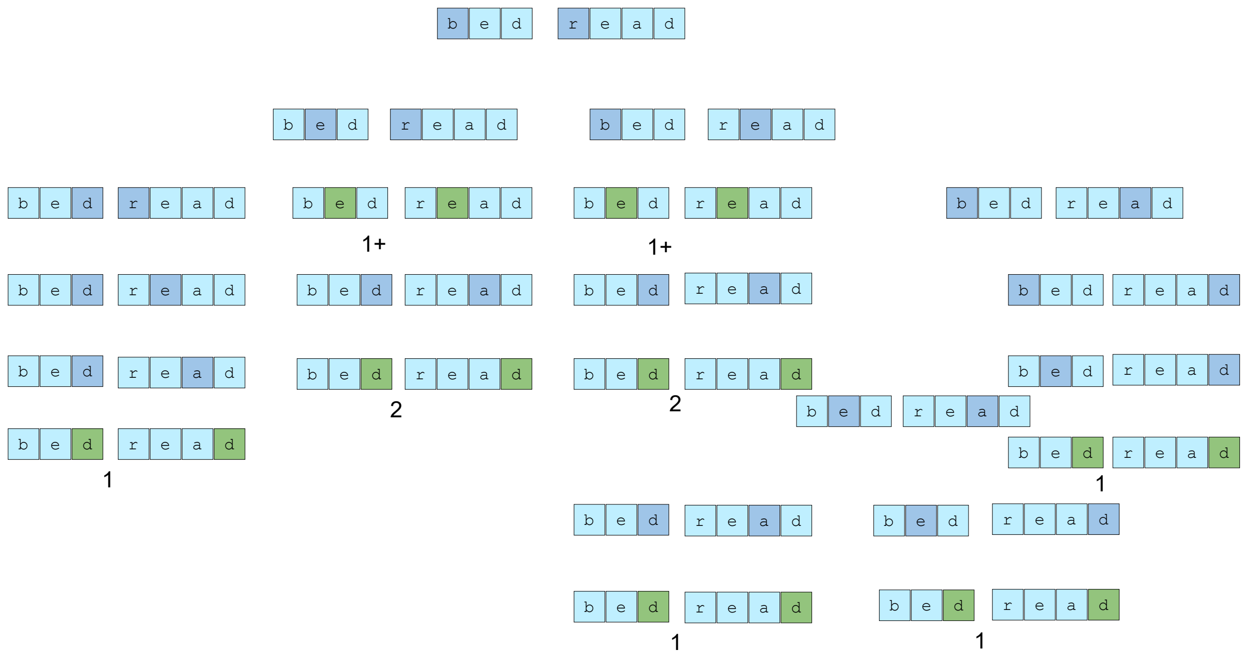
1 of 5



2 of 5

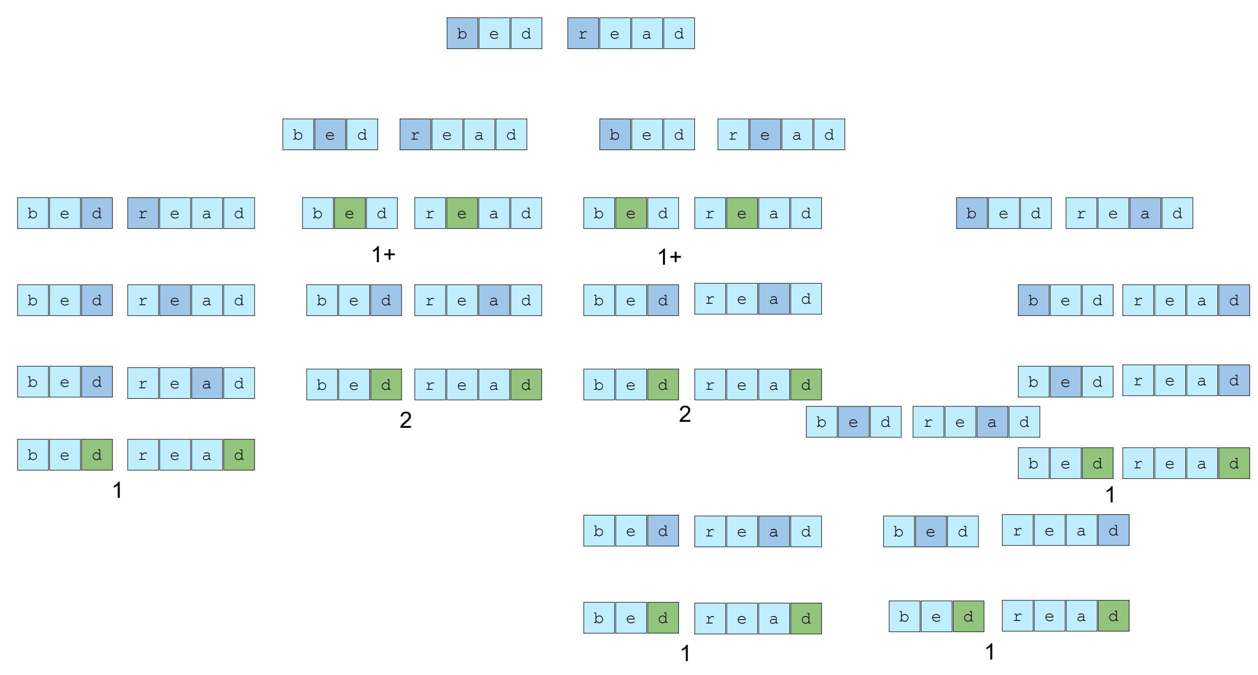


3 of 5



Let's look at repeating computations due to these subproblems

4 of 5



Let's look at repeating computations due to these subproblems

5 of 5

This clearly shows our algorithm can benefit from tabulation or memoization. Let's first look at a solution with memoization.

```
# helper function with updated signature: i is current index in str1, j is current index in str2
def LCS_(str1, str2, i, j, memo):
    if i == len(str1) or j == len(str2): # base case
        return 0
    elif (i,j) in memo:
        return memo[(i,j)]
    elif str1[i] == str2[j]:  # if current characters match, increment 1
        memo[(i,j)] = 1 + LCS_(str1, str2, i+1, j+1, memo)
        return memo[(i,j)]
    # else take max of either of two possibilities
    memo[(i,j)] = max(LCS_(str1, str2, i+1, j, memo), LCS_(str1, str2, i, j+1, memo))
    return memo[(i,j)]

def LCS(str1, str2):
    memo = {}
    return LCS_(str1, str2, 0, 0, memo)

print(LCS("bed", "read"))
```

Explanation

We know the drill now! All we have to do is save the evaluated results in `memo`, so we can look back at `memo` when these results are needed again instead of reevaluating. Since in our case `i` and `j` are responsible for uniquely identifying a pair of substrings, we use them for indexing `memo`.

Time and space complexity

Let's think of this in terms of the keyspace mapped by the tuple of `i` and `j`. For strings of length m and n, `i` can go from 0 to m while j can go from 0 to n. Thus, the total number of unique subproblems to evaluate and store are mn. Thus, the time and space complexity of this algorithm is **O(mn)**.

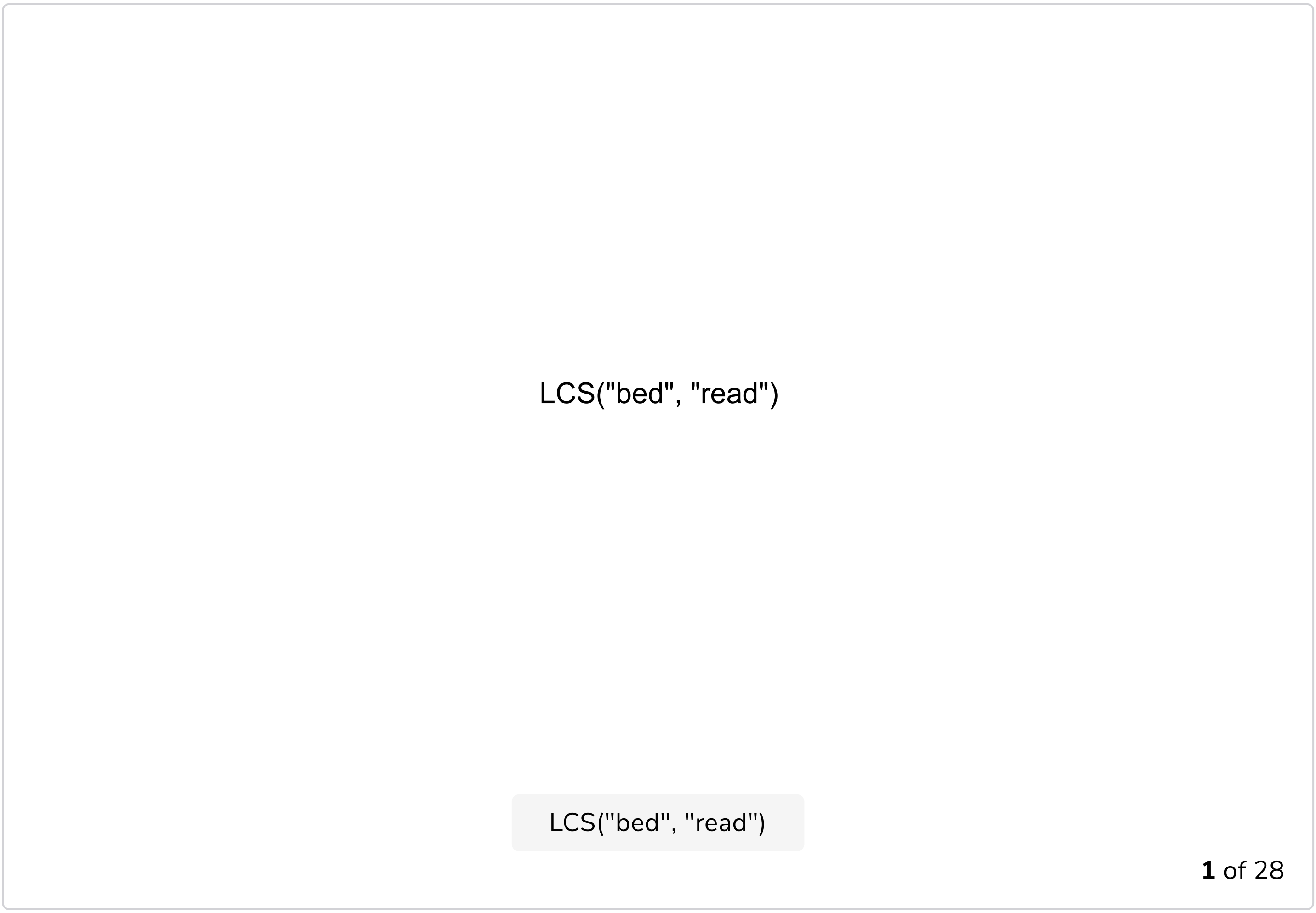
Solution 3: Bottom-up dynamic programming

```
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

    # iterating to fill table
    for i in range(1, n+1):
        for j in range(1, m+1):
            # if characters at this position match,
            if str1[i-1] == str2[j-1]:
                # add 1 to the previous diagonal and store it in this diagonal
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                # if character don't match, take max of last two positions vertically and horizont
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[n][m]

print(LCS("bed", "read"))
```

Explanation

We first make a 2-d list of size $m+1 \times n+1$ where n is the length of `str1` and m is the length of `str2`. This list is initialized to 0. We need the first row and column to be 0 for the base case. Any entry in this list given by `dp[i][j]` is the length of the longest common subsequence between `str1` up till `i` th position and `str2` up to the `j` th position. As we saw in the recursive algorithm, when we had a match of the characters, we could simply move one position ahead in both strings and add one to the result. This is exactly what we do here as well. We add 1 to the previous diagonal result (*line 13*). In case we have a mismatch, we need to take the max of two subproblems: either we could move one position ahead in `str1` (the subproblem `dp[i-1][j]`) or we could move one step ahead in `str2` (the subproblem `dp[i][j-1]`) (*line 16*). In the end, we have the optimal answer for `str1` and `str2` in last position, i.e., `dp[n][m]` (*line 17*).

Let's look at a visualization of this algorithm.



			r	e	a	d
		0	1	2	3	4
	0					
b	1					
e	2					
d	3					

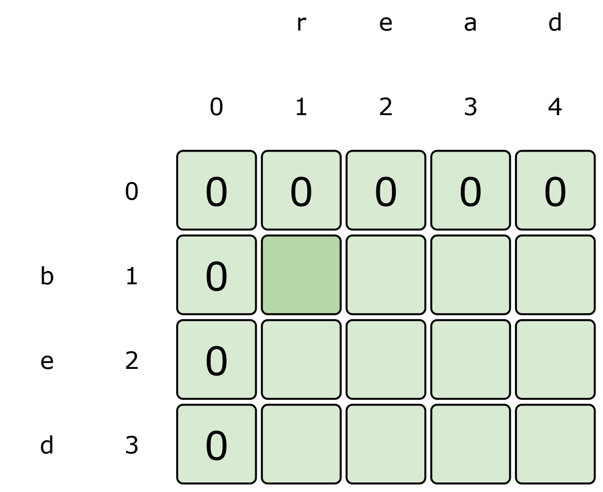
Let's make dp table of size 3x4

2 of 28

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0				
e	2	0				
d	3	0				

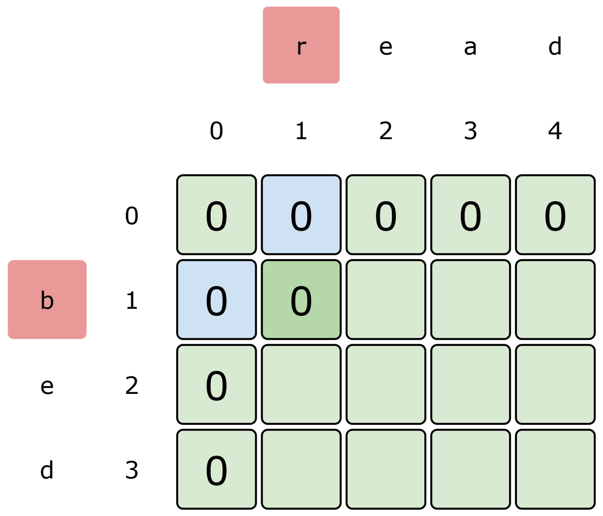
For base case make first row and column zeros

3 of 28



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0				
e	2	0				
d	3	0				

starting from 1,1 index to start filling up the table



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0			
e	2	0				
d	3	0				

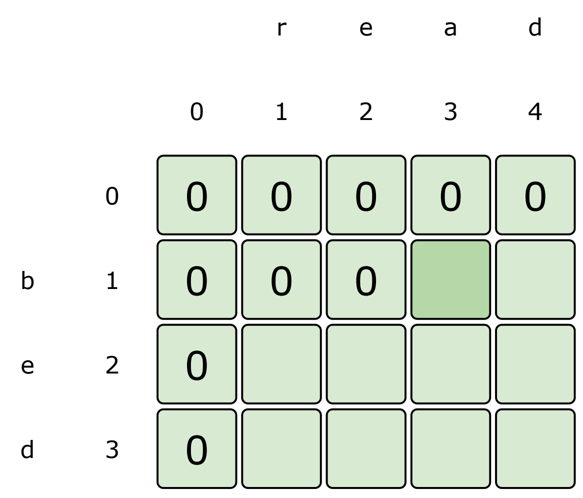
since characters do not match, we take max of position to the left, and position above

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0			
e	2	0				
d	3	0				

moving on

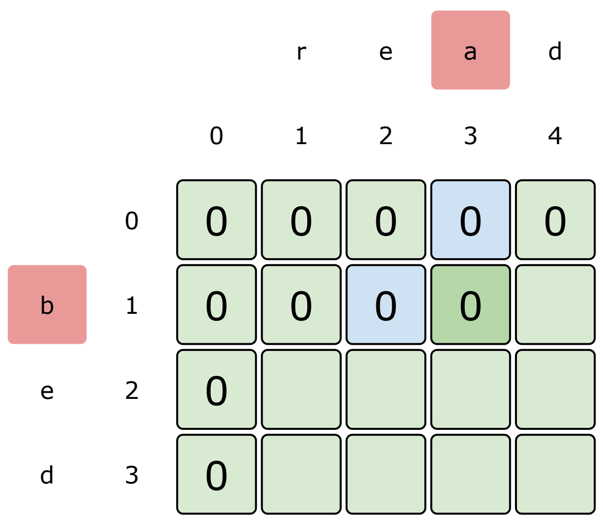
			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0		
e	2	0				
d	3	0				

since characters do not match, we take max of position to the left, and position above



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0		
e	2	0				
d	3	0				

moving on



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	
e	2	0				
d	3	0				

since characters do not match, we take max of position to the left, and position above

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	
e	2	0				
d	3	0				

moving on

10 of 28

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0				
d	3	0				

since characters do not match, we take max of position to the left, and position above

11 of 28

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0				
d	3	0				

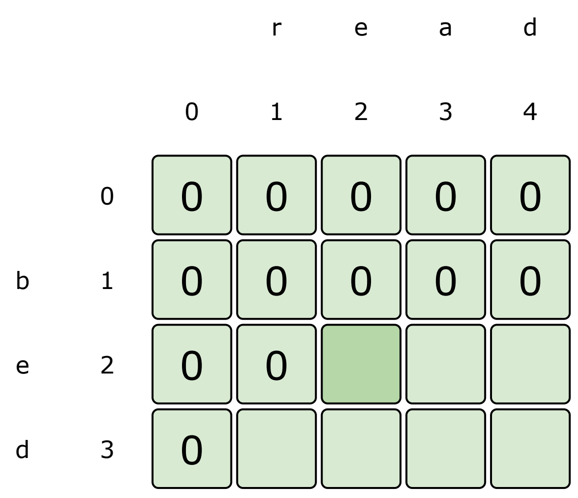
moving on

12 of 28

			🟥 r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
🟥 e	2	0	0			
d	3	0				

since characters do not match, we take max of position to the left, and position above

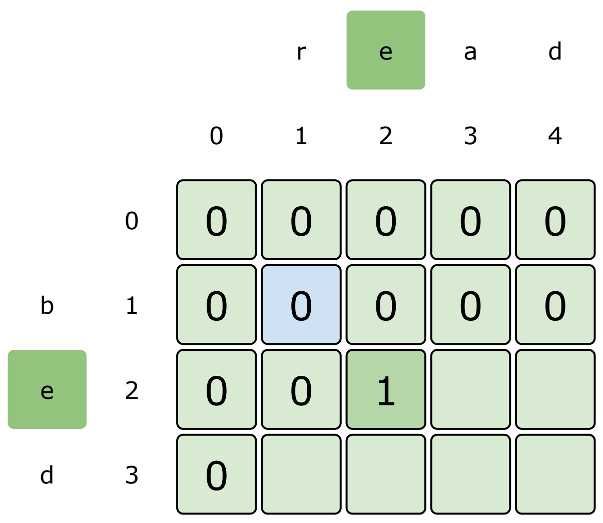
13 of 28



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0			
d	3	0				

moving on

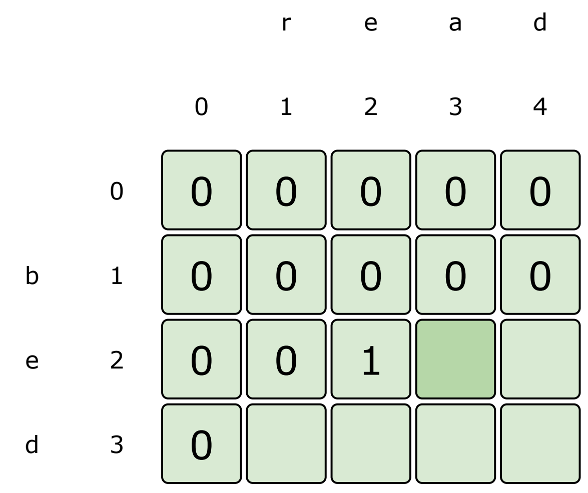
14 of 28



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1		
d	3	0				

Now we have a match of the characters, so we will add 1 to the previous diagonal

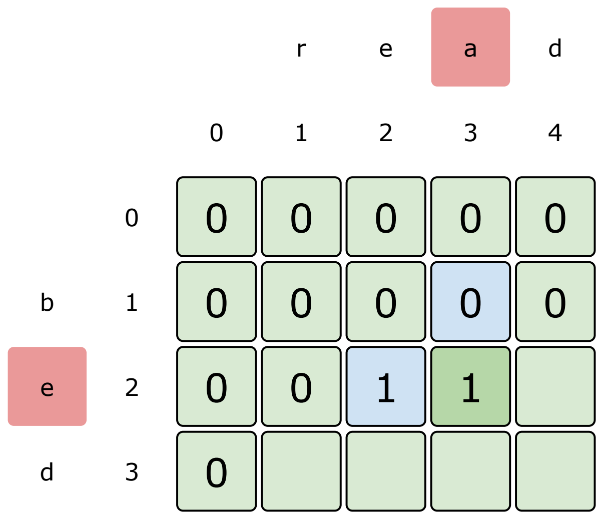
15 of 28



$$\begin{array}{cc|ccccc} & & & r & e & a & d \\ & & 0 & 1 & 2 & 3 & 4 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 0 & 0 & 0 \\ e & 2 & 0 & 0 & 1 & & \\ d & 3 & 0 & & & & \end{array}$$

moving on

16 of 28



$$\begin{array}{cc|ccccc} & & & r & e & a & d \\ & & 0 & 1 & 2 & 3 & 4 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 0 & 0 & 0 \\ e & 2 & 0 & 0 & 1 & 1 & \\ d & 3 & 0 & & & & \end{array}$$

since characters do not match, we take max of position to the left, and position above

17 of 28

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	
d	3	0				

moving on

18 of 28

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	1
d	3	0				

since characters do not match, we take max of position to the left, and position above

19 of 28

		r	e	a	d
	0	1	2	3	4
0	0	0	0	0	0
b 1	0	0	0	0	0
e 2	0	0	1	1	1
d 3	0				

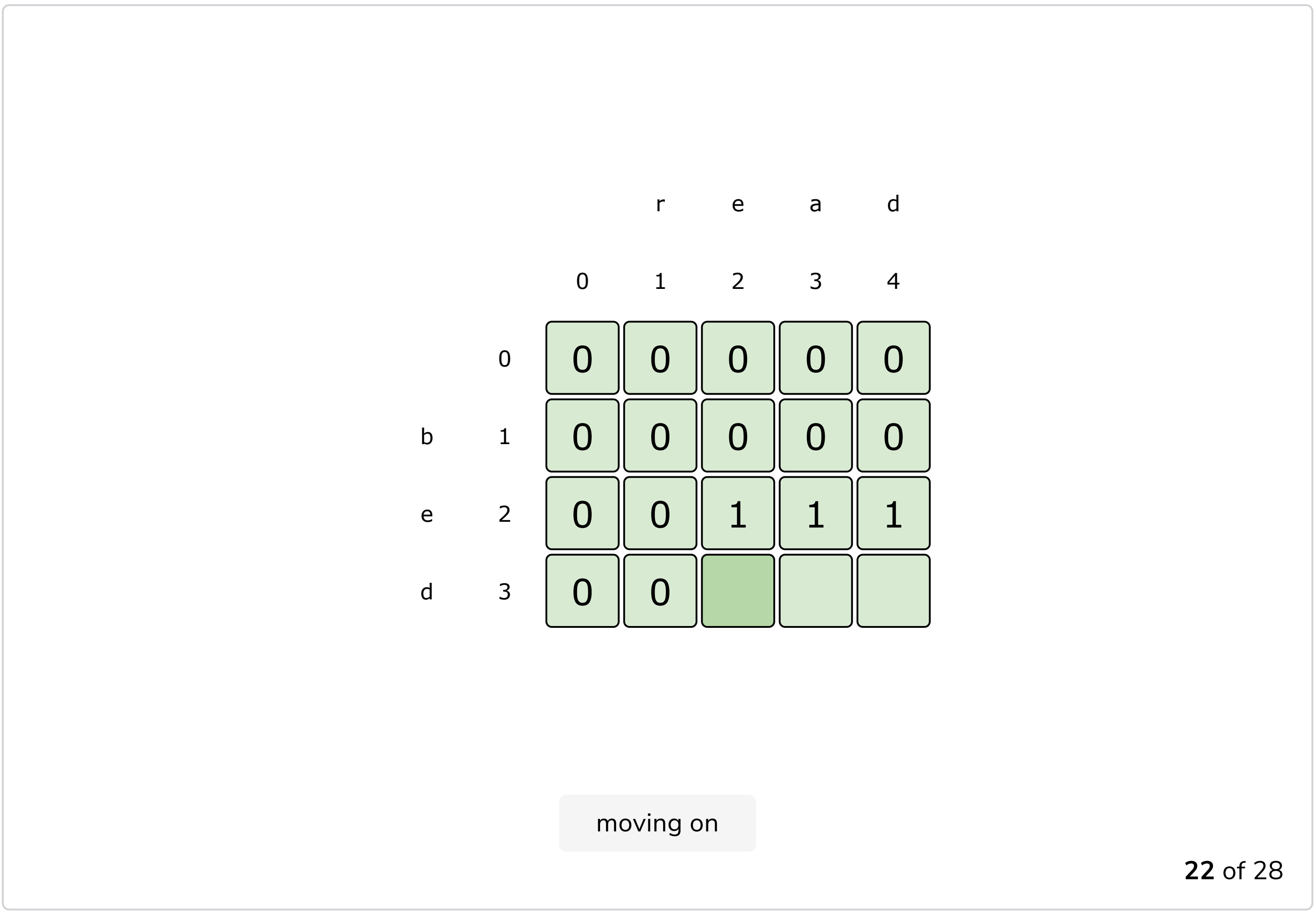
moving on

20 of 28

		r	e	a	d
	0	1	2	3	4
0	0	0	0	0	0
b 1	0	0	0	0	0
e 2	0	0	1	1	1
d 3	0	0			

since characters do not match, we take max of position to the left, and position above

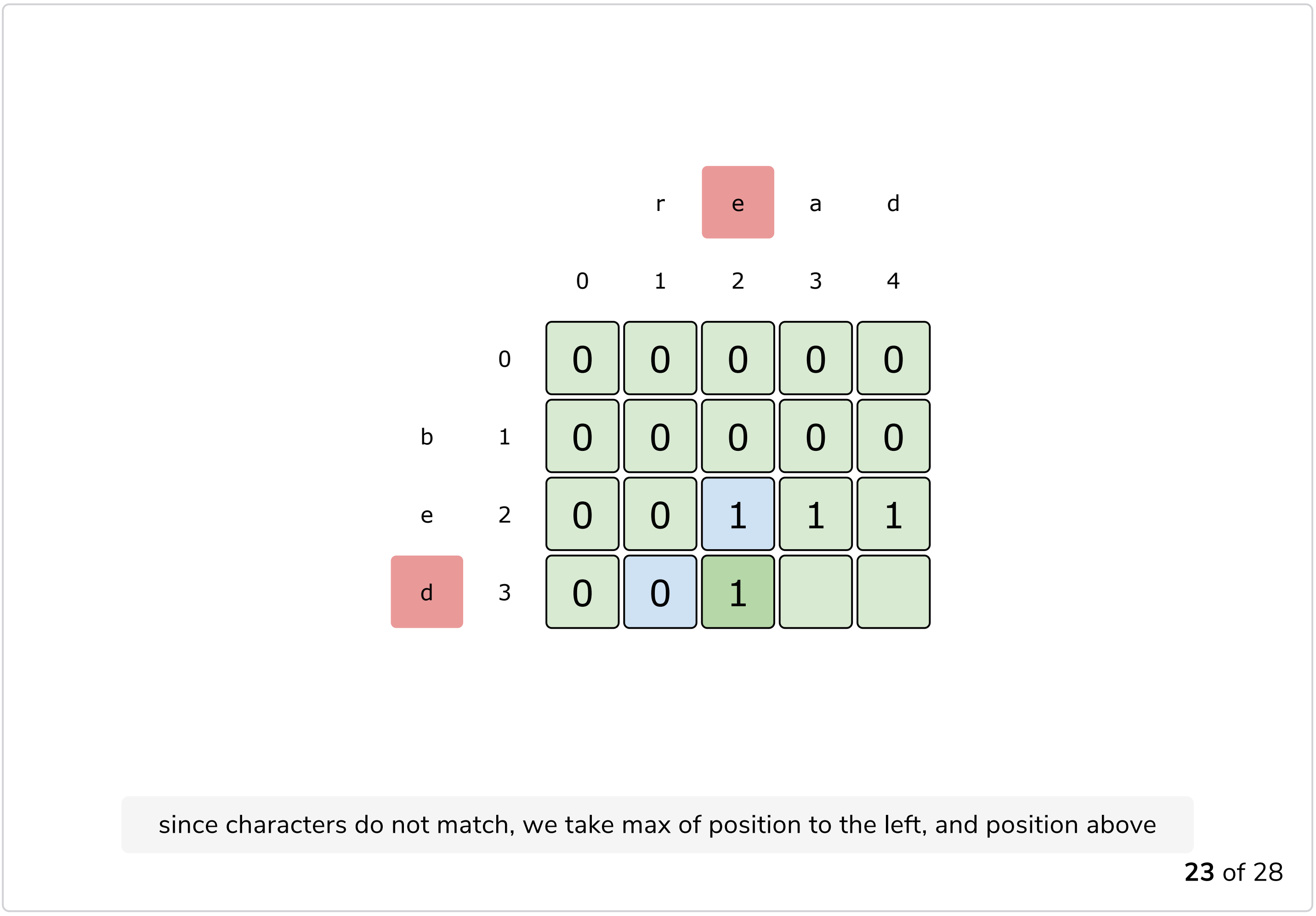
21 of 28



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	1
d	3	0	0			

moving on

22 of 28



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	1
d	3	0	0	1		

since characters do not match, we take max of position to the left, and position above

23 of 28

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	1
d	3	0	0	1		

moving on

			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	1
d	3	0	0	1	1	

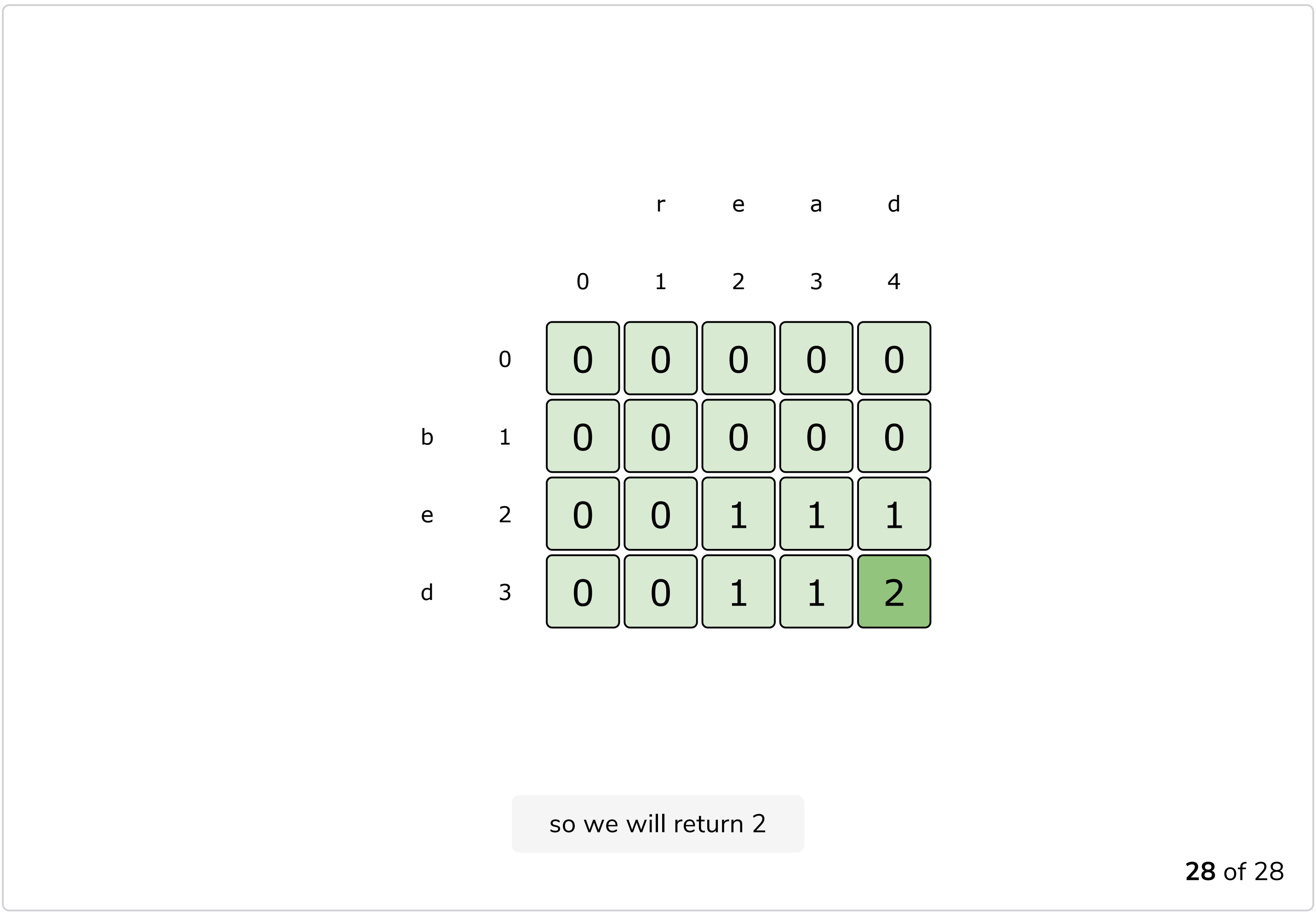
since characters do not match, we take max of position to the left, and position above

		r	e	a	d
	0	1	2	3	4
0	0	0	0	0	0
b 1	0	0	0	0	0
e 2	0	0	1	1	1
d 3	0	0	1	1	

moving on

		r	e	a	d
	0	1	2	3	4
0	0	0	0	0	0
b 1	0	0	0	0	0
e 2	0	0	1	1	1
d 3	0	0	1	1	2

Now we have a match of the characters again, so we will add 1 to the previous diagonal



			r	e	a	d
		0	1	2	3	4
	0	0	0	0	0	0
b	1	0	0	0	0	0
e	2	0	0	1	1	1
d	3	0	0	1	1	2

Time and space complexity

As we can see from the visualization as well, we only have $m \times n$ `dp` table to fill. Thus, the time complexity, as well as the space complexity, would be **O(mn)**.

Solution 4: Space optimized bottom-up dynamic programming

As we can see in the visualization above, we only use the results of the previous rows to evaluate the next row. Thus, there is no point in keeping the results of the complete $m \times n$ size table. Following is an implementation where instead of making a 2d list, we can work with only a 1-d list of size n.

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

    # table for tabulation, only maintaining state of last row
    dp = [0 for i in range(n+1)]
    for j in range(1, m+1):           # iterating to fill table
        # calculate new row (based on previous row i.e. dp)
        thisrow = [0 for i in range(n+1)]

        for i in range(1, n+1):
            # if characters at this position match,
            if str1[i-1] == str2[j-1]:
                # add 1 to the previous diagonal and store it in this diagonal
                thisrow[i] = dp[i-1] + 1
            else:
                # if character don't match, use i-th result from dp, and previous result from this
                thisrow[i] = max(dp[i], thisrow[i-1])
        # after evaluating thisrow, set dp equal to this row to be used in the next iteration
        dp = thisrow
    return dp[n]

print(LCS("who", "wow"))
```

Time and space complexity

The time complexity remains **O(mn)** but the space complexity has been reduced to **O(n)**.

In the next lesson, we will take a look at another dynamic programming coding challenge on strings.