





Solution Review: Longest Palindromic Subsequence

In this lesson, we will see the solution to the longest palindromic subsequence problem and see another similar problem to the longest palindromic substring.

We'll cover the following

- Solution: Using longest common subsequence function
 - Explanation
 - Time and space complexity
- Longest palindromic substring
 - The correct solutions
 - Brute force
 - Explanation
 - Bottom-up dynamic programming
 - Explanation
 - Time and space complexity

Solution: Using longest common subsequence function

<div>main.py</div> <div>lcs.py</div>	All code files are copied to end of the page...
<div></div> <div>  </div>	

Explanation

That's it. All the magic happens in the `LCS` function and we have already seen how that function works in the [previous solution review](#). We have simply called the `LCS` function with the actual string `str` and its reversed form `str[::-1]`. Let's see why this works.

We know that we wrote our **LCS** algorithm to compare characters from the start of the strings towards the end. In this case, we want to compare the same string starting from the opposite ends. One way to exploit our implementation of **LCS** is to simply pass it the original string and its reversed version.





Time and space complexity

Time taken in reversing a string is $O(n)$, while we know that **LCS** has a time complexity of $O(nm)$ given strings of sizes n and m . Since here we have two versions of the same string, they will have the same length, n . Thus, the time complexity would be $O(n^2)$. Similarly, the space complexity would also be $O(n^2)$.

Longest palindromic substring

While we're at it, let's look at another almost identical problem. Given a string, find the longest palindromic substring in it. Remember the difference between subsequence and substring: **substring** is a contiguous part of a string while **subsequence** doesn't necessarily have to be contiguous. You can take a look at the problem of the [longest common substring](#) that we solved in the previous chapter.

Even though the problems look similar, they don't have similar solutions. Here is an implementation for this problem, which clearly gives the wrong answer.

<div>main.py</div> <div>lcs.py</div>	All code files are copied to end of the page...
<div></div> <div>  </div>	

Can you find any palindromic substring in the above string of length three? No. We need to understand that the **LPStr** function finds the longest common substring between two strings. If we reverse the **"racercar"** string, we get **"racrecar"**. These strings have common substrings **"rac"** and **"car"**, which are not palindromic. Thus, when working with these kinds of problems, we need to be careful of the problems that look similar but are not.

The correct solutions

This is another very famous coding problem, and the most famous solutions for this problem are not dynamic programming solutions. With dynamic

programming, as we will see, we can solve this problem at best in $O(n^2)$. However,

there is [Manacher's algorithm](#), which solves this problem in $O(n)$ as well. Here, we will take a look at two solutions to this problem.

Brute force

```
def isPalindrome(string):
    for i in range(len(string)):
        if string[i] != string[len(string) - i - 1]:
            return False
    return True

def LPStr(string):
    maximum = 0
    # Find all possible substrings
    for i in range(len(string)):
        for j in range(i, len(string)):
            # Check if this substring is palindrome or not
            if isPalindrome(string[i:j+1]) and maximum < len(string[i:j+1]):
                maximum = len(string[i:j+1])
    return maximum

print(LPStr("racercar"))
```



Explanation

The idea is to find all the possible substrings and check them individually whether they are palindromes or not. Finding all the substrings of a string of size n entails $O(n^2)$. Next, we check each of these substrings whether it is a palindrome or not, which takes $O(n)$. Thus, we end up with the time complexity of $O(n^3)$.

Let's see how we can improve this algorithm. We ultimately need to find all n^2 substrings, there is no way around that. But can we reduce the time complexity of determining whether a substring is a palindrome? If we knew that a smaller substring was a palindrome, how would that help us in determining whether a larger substring is formed by appending two characters at either side of the smaller substring? You can see how this is moving towards a bottom-up algorithm.

Bottom-up dynamic programming

Let's look at the bottom-up dynamic programming implementation for this problem.

```
def LPStr(string):
    dp = [[False for _ in range(len(string)) for _ in range(len(string))]

    # base cases of substrings of size 1
    for i in range(len(string)):
        dp[i][i] = True

    # to keep track of maximum sized palindromes
    maximum = 1

    # base cases of substrings of size 2
    for i in range(len(string)-1):
        dp[i][i+1] = string[i] == string[i+1]
        if dp[i][i+1]:
            maximum = 2

    # starting from substrings of size 3 up to size of n
    for j in range(2, len(string)):
        # finding all substrings of size j
        for i in range(len(string)-j):
            # this substring is palindrome if first and last characters match and rest of the string is a palindrome
            dp[i][i+j] = dp[i+1][i+j-1] and string[i] == string[i+j]
            # to update maximum
            if dp[i][i+j] and maximum < (j+1):
                maximum = j+1

    return maximum
print(LPStr("racercar"))
```



Explanation

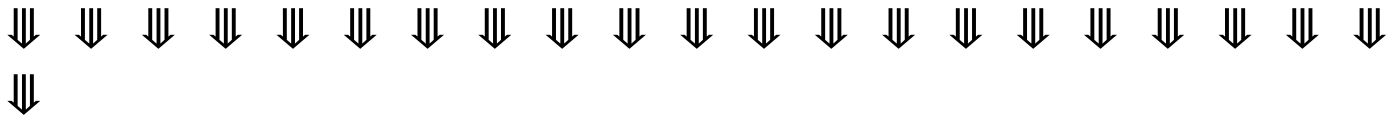
We start off filling our `dp` table with the smallest subproblems first. All the substrings of length one are palindromes by default, so we mark them `True` (lines 5-6). The substrings of size two are palindromes only if both characters in the string are the same (line 12-15). We use the `maximum` variable to keep track of the maximum length of a palindrome as we run through all possible substrings. Now that we are done with the base case, we can move on to the general case. We now check all substrings starting from the length three and going up to n (lines 18-20). To check whether a substring is a palindrome, we simply see if the first and last characters of this substring match and if the remainder of the substring, obtained by deleting the first and last characters, is a palindrome or not (line 22). We update the `maximum` variable throughout the iteration (lines 24-25) and return it at the end.

Time and space complexity

Since we iterate through all possible substrings, which are n^2 , and check whether a substring is $O(1)$, our total time complexity is $O(n^2)$. Since we have to save the results of all these substrings, our space complexity becomes $O(n^2)$ as well.

In the next lesson, we will go through the final coding challenge of this course.

Code Files Content !!!



```
-----  
|  main.py [1]  
-----
```

```
from lcs import LCS  
# call longest common subsequence function as follows:  
#   LCS(str1, str2)  
#   returns integer for length of the longest common subsequence in str1 and str2  
  
def LPS(str):  
    return LCS(str, str[::-1])  
  
print(LPS("racercar"))
```

```
-----  
|  lcs.py [1]  
-----
```

```
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  
  
    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  
            else:  
                thisrow[i] = max(dp[i], thisrow[i-1]) # if character don't match, use i-th result from previous row  
        dp = thisrow    # after evaluating thisrow, set dp equal to this row to be used in the next iteration  
    return dp[n]
```

```
*****
```

```
-----  
|  main.py [2]  
-----
```

```
from lcs import LCStr  
# call longest common subsequence function as follows:  
#   LCS(str1, str2)  
#   returns integer for length of the longest common subsequence in str1 and str2  
  
def LPStr(str):  
    return LCStr(str, str[::-1])  
  
print(LPStr("racercar"))
```

```
-----  
|  lcs.py [2]  
-----
```

```
def LCStr(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 maxLength  
            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 iteration  
    return maxLength
```

```
*****
```