## 516. Longest Palindromic Subsequence



**Leetcode Link** 

## **Problem Description**

a sequence that can be obtained from another sequence by deleting zero or more characters without changing the order of the remaining characters. Unlike substrings, subsequences are not required to occupy consecutive positions within the original string. A palindromic subsequence is one that reads the same backward as forward.

The problem asks us to find the length of the longest palindromic subsequence within a given string s. A subsequence is defined as

Intuition

subsequence of the whole string. The key idea is to create a 2D array dp where dp[i][j] represents the length of the longest palindromic subsequence of the substring s[i:j+1].

The intuition behind the solution is to use dynamic programming to build up the solution by finding the lengths of the longest

palindromic subsequences within all substrings of s, and then use these results to find the length of the longest palindromic

To fill this table, we start with the simplest case: a substring of length 1, which is always a palindrome of length 1. Then, we gradually consider longer substrings by increasing the length and using the previously computed values.

When we look at a substring [i, j] (where i is the starting index and j is the ending index):

1. If the characters at positions i and j are the same, then the length of the longest palindromic subsequence of [i, j] is two plus

the length of the longest palindromic subsequence of [i + 1, j - 1]. 2. If the characters are different, the longest palindromic subsequence of [i, j] is the longer of the longest palindromic

subsequences of [i + 1, j] and [i, j - 1].

We continue this process, eventually filling in the dp table for all possible substrings, and the top-right cell of the table (dp[0][n-1],

where n is the length of the original string) will contain the length of the longest palindromic subsequence of s.

Solution Approach The implementation uses dynamic programming to solve the problem as follows:

1. Initialize a 2D array, dp, of size n x n, where n is the length of the input string s. Each element dp[i][j] will represent the length

#### of the longest palindromic subsequence in the substring s[i...j].

characters.

[j-1], dp[i][j-1], and dp[i+1][j].

2. Populate the diagonal of dp with 1s because a single character is always a palindrome with a length of 1. We're certain that the

represents the substring from the first character to the last.

subsequence. We will use the implementation steps mentioned to find this length.

- longest palindromic subsequence in a string of length 1 is the string itself. 3. The array dp is then filled in diagonal order. This is because to calculate dp[i][j], we need to have already calculated dp[i+1]
- 4. For each element dp[i][j] (where i < j), two cases are possible:

o If s[i] == s[j], the characters at both ends of the current substring match, and they could potentially be part of the

longest palindromic subsequence. Therefore, dp[i][j] is set to dp[i+1][j-1] + 2, adding 2 to account for the two matching

o If s[i] != s[j], the characters at both ends of the substring do not match. We must find whether the longer subsequence

- occurs by excluding s[i] or s[j]. So, dp[i][j] is set to the maximum of dp[i][j-1] and dp[i+1][j]. 5. After filling up the array, dp [0] [n-1] will contain the length of the longest palindromic subsequence in the entire string, since it
- The solution approach efficiently computes the longest palindromic subsequence by systematically solving smaller subproblems and combining them to form the solution to the original problem.

Example Walkthrough Let's use the string s = "bbbab" to illustrate the solution approach. The goal is to find the length of the longest palindromic

### Initially, dp looks like this (zero-initialized for non-diagonal elements):

subsequences for different substrings.

[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0]

After this step, dp looks like this:

diagonal (substrings of length 3), and so on.

Filling dp for the substring of length 2:

1 dp = [ 2 [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],

1. Initialize the 2D array dp of size 5×5 (since the length of s is 5). This dp will store lengths of the longest palindromic

2. Populate the diagonal with 1s, because any single character is itself a palindrome of length 1.

```
dp = [
      [1, 0, 0, 0, 0],
     [0, 1, 0, 0, 0],
     [0, 0, 1, 0, 0],
     [0, 0, 0, 1, 0],
      [0, 0, 0, 0, 1]
```

and dp[i+1][j].

5 [0, 1, 0, 0, 0]

6 [0, 0, 1, 0, 0]

7 [0, 0, 0, 1, 0] 8 [0, 0, 0, 0, 1]

2 [1, 2, 3, 3, 4]

3 [0, 1, 2, 2, 3]

**Python Solution** 

1 class Solution:

11

12

13

14

15

16

17

18

23

24

25

26

27

28

29

30

30

31

32

33

34

36

35 }

C++ Solution

1 class Solution {

1 dp[0][1] (s[0]: 'b', s[1]: 'b') => dp[1][0] (which is 0, a non-existing substring) + 2 = 2 3 dp array becomes: 4 [1, 2, 0, 0, 0]

3. Now, we start filling in the dp array in diagonal order, just above the main diagonal (i.e., substrings of length 2), then the next

4. When s[i] == s[j], we set dp[i][j] to dp[i+1][j-1] + 2. When s[i] != s[j], we set dp[i][j] to the maximum of dp[i][j-1]

Similarly, filling dp for all substrings of length 2 to length n-1 (n = 5):

4 [0, 0, 1, 1, 3] 5 [0, 0, 0, 1, 2] 6 [0, 0, 0, 0, 1]

def longest\_palindrome\_subseq(self, s: str) -> int:

dp = [[0] \* length for \_ in range(length)]

# so we populate the diagonals with 1

for i in range(j - 1, -1, -1):

for i in range(length):

for j in range(1, length):

dp[i][i] = 1

else:

return dp[0][length - 1]

return dp[0][n - 1];

# palindromic subsequence between indices i and j in string s

# Loop over pairs of characters from end to start of the string

dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

# A single character is always a palindrome of length 1,

dp[i][j] = dp[i + 1][j - 1] + 2

# The length of the input string

length = len(s)

1 dp array after filling in:

```
5. Finally, dp [0] [n-1] which is dp [0] [4] contains the length of the longest palindromic subsequence of the entire string s. In this
    case, it is 4.
    The string s = "bbbab" has a longest palindromic subsequence of length 4, which can be bbbb or bbab.
The example provided demonstrates the use of dynamic programming to calculate the length of the longest palindromic
subsequence step by step by building solutions to smaller subproblems.
```

# Initialize a 2D array with zeros, where dp[i][j] will hold the length of the longest

# If characters at index i and j are the same, they can form a palindrome:

# - Excluding the j-th character (considering subsequence from i to j - 1)

# - Excluding the i-th character (considering subsequence from i + 1 to j)

# Otherwise, take the maximum of the lengths found by:

# The entire string's longest palindromic subsequence length is at dp[0][length-1]

// The top-right corner of 'dp' will hold the result for the entire string (0, n-1)

#### 19 # - Add 2 to the length of the longest palindromic subsequence we found 20 between i + 1 and j - 1**if** s[i] == s[j]: 21

```
31
Java Solution
   class Solution {
       public int longestPalindromeSubseq(String s) {
           // Length of the input string
           int n = s.length();
           // Initialize a 2D array 'dp' to store the length of the longest
           // palindromic subsequence for substring (i, j)
           int[][] dp = new int[n][n];
9
10
           // Base case: single letters are palindromes of length 1
           for (int i = 0; i < n; ++i) {
               dp[i][i] = 1;
14
           // Build the table 'dp' bottom-up such that:
15
16
           // We start by considering all substrings of length 2, and work our way up to n.
           for (int j = 1; j < n; ++j) {
17
               for (int i = j - 1; i >= 0; ---i) {
18
                   // If the characters at positions i and j are the same
19
                   // they can form a palindrome with the palindrome from substring (i+1, j-1)
20
                   if (s.charAt(i) == s.charAt(j)) {
21
22
                       dp[i][j] = dp[i + 1][j - 1] + 2;
23
                   } else {
24
                       // If the characters at i and j do not match, then the longest palindrome
25
                       // within (i, j) is the maximum of (i+1, j) or (i, j-1) since we can
26
                       // exclude one of the characters and seek the longest within the remaining substring
27
                       dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
28
29
```

```
2 public:
       // Function to find the length of the longest palindromic subsequence.
       int longestPalindromeSubseq(string s) {
           // Store the length of the string.
           int length = s.size();
           // Create a 2D DP array with 'length' rows and 'length' columns.
           vector<vector<int>> dp(length, vector<int>(length, 0));
11
           // Each single character is a palindrome of length 1.
12
           for (int i = 0; i < length; ++i) {</pre>
13
               dp[i][i] = 1;
14
15
           // Build the DP table in a bottom-up manner.
16
           for (int end = 1; end < length; ++end) {</pre>
                for (int start = end - 1; start >= 0; --start) {
                   // If the characters at the current start and end positions are equal,
20
                   // we can extend the length of the palindrome subsequence by 2.
                   if (s[start] == s[end])
21
22
                       dp[start][end] = dp[start + 1][end - 1] + 2;
23
                   } else {
                       // Otherwise, we take the maximum of the two possible subsequence lengths:
24
25
                       // excluding the start character or the end character
26
                        dp[start][end] = max(dp[start + 1][end], dp[start][end - 1]);
27
28
29
30
31
           // The answer is in dp[0][length - 1], which represents the longest palindromic
32
           // subsequence of the entire string.
33
           return dp[0][length - 1];
34
35 };
36
Typescript Solution
   // Function to find the length of the longest palindromic subsequence
   function longestPalindromeSubseq(s: string): number {
       // Store the length of the string
       const length: number = s.length;
```

# Time and Space Complexity

// If the characters at the current start and end positions are equal, 18 // we can extend the length of the palindrome subsequence by 2. 19 **if** (s[start] === s[end]) { 20 dp[start][end] = dp[start + 1][end - 1] + 2;21 } else { // Otherwise, we take the maximum of the two possible subsequence lengths: 23 // excluding the start character or the end character 24 25 dp[start][end] = Math.max(dp[start + 1][end], dp[start][end - 1]); 26 27 28

// Create a 2D DP array with 'length' rows and 'length' columns,

// Each single character is a palindrome of length 1.

for (let start = end - 1; start >= 0; start--) {

const dp: number[][] = new Array(length).fill(0).map(() => new Array(length).fill(0));

// The answer is in dp[0][length - 1], which represents the longest palindromic

// initialized with zeros

dp[i][i] = 1;

for (let i = 0; i < length; i++) {</pre>

// Build the DP table in a bottom-up manner.

for (let end = 1; end < length; end++) {</pre>

// subsequence of the entire string

return dp[0][length - 1];

9

10

11

12

13

14

15

16

17

29

30

31

32

34

33 }

#### Time Complexity The time complexity of the code is $0(n^2)$ , where n is the length of the string s. This complexity arises because the algorithm uses

**Space Complexity** 

The space complexity of the code is also  $0(n^2)$ . This is due to the dp table which is a 2-dimensional list of size n \* n. Each cell of the table is filled out exactly once, resulting in a total space usage directly proportional to the square of the length of the input string.

two nested loops: the outer loop runs from 1 to n - 1 and the inner loop runs in reverse from j - 1 to 0. Each time the inner loop

executes, the algorithm performs a constant amount of work, resulting in a total time complexity of O(n^2).