516. Longest Palindromic Subsequence

Dynamic Programming

Problem Description

String]

Medium

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

Intuition 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

subsequence of the whole string.

the longest palindromic subsequence of [i + 1, j - 1].

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]. 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

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 <u>dynamic programming</u> to solve the problem as follows:

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 longest palindromic subsequence in a string of length 1 is the string itself.

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

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][j-1], dp[i][j-1], and <math>dp[i+1][j].

it represents the substring from the first character to the last.

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

and combining them to form the solution to the original problem.

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

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 characters.

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]. After filling up the array, dp[0][n-1] will contain the length of the longest palindromic subsequence in the entire string, since

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

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

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

The solution approach efficiently computes the longest palindromic subsequence by systematically solving smaller subproblems

subsequences for different substrings. Initially, dp looks like this (zero-initialized for non-diagonal elements):

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

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

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

After this step, dp looks like this:

dp = [

dp = [

dp array becomes:

[1, 2, 0, 0, 0]

[0, 1, 0, 0, 0]

[0, 0, 1, 0, 0]

[0, 0, 0, 1, 0]

[0, 0, 0, 0, 1]

[1, 2, 3, 3, 4]

[0, 1, 2, 2, 3]

[0, 0, 1, 1, 3]

[0, 0, 0, 1, 2]

Solution Implementation

length = len(s)

for i in range(length):

for i in range(1, length):

dp[i][i] = 1

else:

Python

class Solution:

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

[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]

```
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
 diagonal (substrings of length 3), and so on.
 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] and dp[i+1][j].
 Filling dp for the substring of length 2:
dp[0][1] (s[0]: 'b', s[1]: 'b') => dp[1][0] (which is 0, a non-existing substring) + 2 = 2
```

- Similarly, filling dp for all substrings of length 2 to length n-1 (n = 5): dp array after filling in:
- [0, 0, 0, 0, 1] 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 example provided demonstrates the use of dynamic programming to calculate the length of the longest palindromic

The string s = "bbbab" has a longest palindromic subsequence of length 4, which can be bbbb or bbab.

subsequence step by step by building solutions to smaller subproblems.

def longest palindrome subseq(self, s: str) -> int:

A single character is always a palindrome of length 1,

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

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

// Initialize a 2D array 'dp' to store the length of the longest

// Base case: single letters are palindromes of length 1

The length of the input string

so we populate the diagonals with 1

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

if s[i] == s[i]:

between i + 1 and j - 1

// palindromic subsequence for substring (i, j)

// Build the table 'dp' bottom-up such that:

for (int i = i - 1; i >= 0; ---i) {

if (s.charAt(i) == s.charAt(j)) {

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

// Function to find the length of the longest palindromic subsequence

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

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

// If the characters at the current start and end positions are equal,

dp[start][end] = Math.max(dp[start + 1][end], dp[start][end - 1]);

// Otherwise, we take the maximum of the two possible subsequence lengths:

// we can extend the length of the palindrome subsequence by 2.

// excluding the start character or the end character

dp[start][end] = dp[start + 1][end - 1] + 2;

function longestPalindromeSubseq(s: string): number {

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

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

if (s[start] === s[end]) {

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

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

// Store the length of the string

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

const length: number = s.length;

// initialized with zeros

dp[i][i] = 1;

} else {

int[][] dp = new int[n][n];

for (int i = 0; i < n; ++i) {

for (int j = 1; j < n; ++j) {

} else {

dp[i][i] = 1;

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

Initialize a 2D array with zeros, where dp[i][i] will hold the length of the longest # palindromic subsequence between indices i and j in string s dp = [[0] * length for _ in range(length)]

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

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

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

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

- Add 2 to the length of the longest palindromic subsequence we found

Otherwise, take the maximum of the lengths found by:

// We start by considering all substrings of length 2, and work our way up to n.

// they can form a palindrome with the palindrome from substring (i+1, j-1)

// within (i, i) is the maximum of (i+1, i) or (i, i-1) since we can

// If the characters at i and i do not match, then the longest palindrome

// exclude one of the characters and seek the longest within the remaining substring

// If the characters at positions i and i are the same

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

```
public int longestPalindromeSubseq(String s) {
   // Length of the input string
    int n = s.length();
```

class Solution {

Java

```
// The top-right corner of 'dp' will hold the result for the entire string (0, n-1)
        return dp[0][n - 1];
C++
class Solution {
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));
        // Each single character is a palindrome of length 1.
        for (int i = 0; i < length; ++i) {</pre>
            dp[i][i] = 1;
        // Build the DP table in a bottom-up manner.
        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,
                // we can extend the length of the palindrome subsequence by 2.
                if (s[start] == s[end]) {
                    dp[start][end] = dp[start + 1][end - 1] + 2;
                } else {
                    // Otherwise, we take the maximum of the two possible subsequence lengths:
                    // excluding the start character or the end character
                    dp[start][end] = max(dp[start + 1][end], dp[start][end - 1]);
        // The answer is in dp[0][length - 1], which represents the longest palindromic
        // subsequence of the entire string.
        return dp[0][length - 1];
};
```

TypeScript

```
// The answer is in dp[0][length - 1], which represents the longest palindromic
   // subsequence of the entire string
   return dp[0][length - 1];
class Solution:
   def longest palindrome subseq(self, s: str) -> int:
       # The length of the input string
        length = len(s)
       # Initialize a 2D array with zeros, where dp[i][i] will hold the length of the longest
       # palindromic subsequence between indices i and j in string s
       dp = [[0] * length for _ in range(length)]
       # A single character is always a palindrome of length 1,
       # so we populate the diagonals with 1
       for i in range(length):
           dp[i][i] = 1
       # Loop over pairs of characters from end to start of the string
       for i in range(1, length):
            for i in range(j - 1, -1, -1):
               # If characters at index i and i are the same, they can form a palindrome:
               # - Add 2 to the length of the longest palindromic subsequence we found
                   between i + 1 and j - 1
               if s[i] == s[i]:
                   dp[i][j] = dp[i + 1][j - 1] + 2
               else:
                   # Otherwise, take the maximum of the lengths found by:
                   \# - Excluding the j-th character (considering subsequence from i to j - 1)
                   # - Excluding the i-th character (considering subsequence from i + 1 to j)
                   dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])
       # The entire string's longest palindromic subsequence length is at dp[0][length-1]
       return dp[0][length - 1]
```

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

Time and Space Complexity

uses 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 0(n^2). **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.