<https://leetcode.com/problems/longest-palindromic-subsequence/>

Given a string s, find *the longest palindromic* ***subsequence****'s length in* s.

A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

**Example 1:**

Input: s = "bbbab"

Output: 4

Explanation: One possible longest palindromic subsequence is "bbbb".

**Example 2:**

Input: s = "cbbd"

Output: 2

Explanation: One possible longest palindromic subsequence is "bb".

**Constraints:**

1 <= s.length <= 1000

s consists only of lowercase English letters.

**Attempt 1: 2023-05-11**

**Solution 1: Native DFS (10 min, TLE)**

**O(2^n) Brute force. If the two ends of a string are the same, then they must be included in the longest palindrome subsequence. Otherwise, both ends cannot be included in the longest palindrome subsequence.**

class Solution {

    public int longestPalindromeSubseq(String s) {

        return helper(s, 0, s.length() - 1);

    }

    private int helper(String s, int i, int j) {

        // Happens after "aa"

        if(i > j) {

            return 0;

        }

        if(i == j) {

            return 1;

        }

        // If the two ends of a string are the same, then they must be included

        // in the longest palindrome subsequence. Otherwise, both ends cannot be

        // included in the longest palindrome subsequence

        int result = 0;

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

            result += 2 + helper(s, i + 1, j - 1);

        } else {

            result += Math.max(helper(s, i + 1, j), helper(s, i, j - 1));

        }

        return result;

    }

}

Time Complexity : O(2^N)

Space Complexity : O(N)

**Refer to chatGPT**

**DFS Approach:**

In the DFS approach, we'll recursively try to find the longest palindromic subsequence by comparing characters from the beginning and the end of the string. If the characters match, we move inward; if they don't, we branch into two paths, either skipping the character from the left or from the right, and return the maximum result.

class Solution {

public int longestPalindromeSubseq(String s) {

return helper(s, 0, s.length() - 1);

}

private int helper(String s, int i, int j) {

// Base case: If the pointers cross, no valid subsequence is possible

if (i > j) {

return 0;

}

// Base case: If i == j, the longest subsequence is the single character itself

if (i == j) {

return 1;

}

// If the characters at i and j match, they can be part of the palindromic subsequence

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

return 2 + helper(s, i + 1, j - 1);

}

// If the characters don't match, we try both possibilities:

// 1. Skipping the character at i

// 2. Skipping the character at j

// And return the maximum result from both possibilities

int skipLeft = helper(s, i + 1, j);

int skipRight = helper(s, i, j - 1);

return Math.max(skipLeft, skipRight);

}

}

**Explanation:**

Base Cases:

If i > j: It means we’ve crossed the indices, and there’s no subsequence left, so we return 0.

If i == j: It means we are looking at a single character, which is a palindrome of length 1, so we return 1.

Recursive Case:

If s.charAt(i) == s.charAt(j): The characters match, and they can be part of the palindromic subsequence, so we add 2 to the result and recurse inward (i+1, j-1).

If s.charAt(i) != s.charAt(j): The characters don’t match, so we have two options:

Skip the character at i and recurse (i+1, j).

Skip the character at j and recurse (i, j-1).

We take the maximum of these two results.

**Time Complexity:**

Time Complexity: O(2^n) where n is the length of the string s. This is because we explore every possible subsequence, leading to exponential growth in recursive calls.

Space Complexity: O(n) due to the recursion stack.

This is the **native DFS** solution, and it's inefficient for large inputs due to its exponential time complexity.

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int n = s.length();

        Integer[][] memo = new Integer[n][n];

        return helper(s, 0, n - 1, memo);

    }

    private int helper(String s, int i, int j, Integer[][] memo) {

        if(memo[i][j] != null) {

            return memo[i][j];

        }

        // Happens after "aa"

        if(i > j) {

            return 0;

        }

        if(i == j) {

            return 1;

        }

        // If the two ends of a string are the same, then they must be included

        // in the longest palindrome subsequence. Otherwise, both ends cannot be

        // included in the longest palindrome subsequence

        int result = 0;

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

            result += 2 + helper(s, i + 1, j - 1, memo);

        } else {

            result += Math.max(helper(s, i + 1, j, memo), helper(s, i, j - 1, memo));

        }

        memo[i][j] = result;

        return result;

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N^2)

**Refer to**

<https://leetcode.com/problems/longest-palindromic-subsequence/editorial/>

**Approach 1: Recursive Dynamic Programming**

**Intuition**

If you are new to Dynamic Programming, please see our [Leetcode Explore Card](https://leetcode.com/explore/featured/card/dynamic-programming/) for more information on it!

An intuitive approach to solve this problem is to generate all the subsequences of the given string and find the longest palindromic string among all the generated strings. There are a total of 2^n strings possible, where n denotes the length of the given string. We can use recursion to generate all possible strings.

If the first and last characters are the same, both characters are guaranteed to be considered in the final palindrome. As a result, we add 2 to our answer variable and recursively remove the first and last characters from the remaining substring.

If the first and last characters aren’t the same, they cannot both occur in the final palindrome. As a result, we recurse over the substring removing the first and also recurse over the substring removing the last character. As we want the longest palindromic subsequence, we pick the maximum out of both of these.

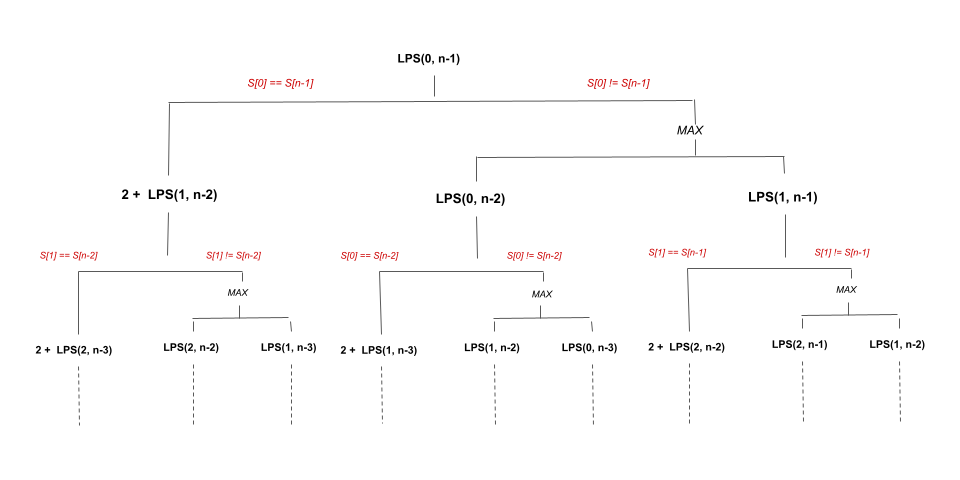
To perform this recursion, we use two pointers, i and j, where i is the index of the first character and j is the index of the last character, to form a substring of s that is being considered. As a result, the recursive relation can be written as follows:

1. If s[i] == s[j], perform answer = 2 + LPS(i + 1, j - 1).

2. Else, perform answer = max(LPS(i, j - 1), LPS(i + 1, j).

where LPS(int i, int j) is a recursive method that returns the longest palindromic subsequence of the substring formed from index i to index j in s. The solution is LPS(0, n - 1), where n is the length of s

The recursion tree of the above relation would look something like this:



Several subproblems, such as LPS(2, n - 2), LPS(1, n - 3), etc., are solved twice in the partial recursion tree shown above. If we draw the entire recursion tree, we can see that there are many subproblems that are solved repeatedly.

To avoid this issue, we store the solution of the subproblem in a 2D array when it is solved. When we encounter the same subproblem again, we simply refer to the array. This is called **memoization**.

**Algorithm**

Create an integer variable n and initialize it to the size of s.

Create a 2D-array called memo having n rows and n columns where memo[i][j] contains the length of the longest palindromic subsequence of the substring formed from index i to j in s.

Return lps(s, 0, n - 1, memo) where lps is a recursive method with four parameters: s, the starting index of the substring under consideration as i, the ending index of the substring as j and memo. We perform the following in this method:

If memo[i][j] != 0, it indicates that we have already solved this subproblem, so we return memo[i][j].

If i > j, the string is empty. We return 0.

If i == j, it is a substring having one character. As a result, we return 1.

If the first and the last characters are the same, i.e., s[i] == s[j], we include these two characters in the palindromic subsequence and add it to the longest palindromic subsequence formed using the substring from index i + 1 to j - 1 (inclusive). We perform memo[i][j] = lps(s, i + 1, j - 1, memo) + 2.

Otherwise, if the first and the last characters do not match, we recursively search for the longest palindromic subsequence in both the substrings formed after ignoring the first and last characters. We pick the maximum of these two. We perform memo[i][j] = max(lps(s, i + 1, j, memo), lps(s, i, j - 1, memo)).

Return memo[i][j].

**Implementation**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int n = s.length();

        int[][] memo = new int[n][n];

        return lps(s, 0, n - 1, memo);

    }

    private int lps(String s, int i, int j, int[][] memo) {

        if (memo[i][j] != 0) {

            return memo[i][j];

        }

        if (i > j) {

            return 0;

        }

        if (i == j) {

            return 1;

        }

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

            memo[i][j] = lps(s, i + 1, j - 1, memo) + 2;

        } else {

            memo[i][j] = Math.max(lps(s, i + 1, j, memo), lps(s, i, j - 1, memo));

        }

        return memo[i][j];

    }

}

**Complexity Analysis**

Here, n is the length of s.

Time complexity: O(n^2)

Initializing the memo array takes O(n^2) time.

Since there are O(n^2)states that we need to iterate over, the recursive function is called O(n^2) times.

Space complexity: O(n^2)

The memo array consumes O(n^2) space.

The recursion stack used in the solution can grow to a maximum size of O(n). When we try to form the recursion tree, we see that there are maximum of two branches that can be formed at each level (when s[i]!= s[j]). The recursion stack would only have one call out of the two branches. The height of such a tree will be O(n) because at each level we are decrementing the length of the string under consideration by '1'. As a result, the recursion tree that will be formed will have O(n) height. Hence, the recursion stack will have a maximum of O(n) elements.

**Solution 3: 2D DP (10 min)**

**Style 1: Similar to Longest Palindrome Substring way, loop i down and loop j up**

**This way is deduced by previous Native DFS solution based on 顶底之术，since in Native DFS solution we have i begin with 0 end with n - 1, j begin with n - 1 end with 0, so in 2D DP, we just reverse it as for loop i begin with n - 1 end with 0, j begin with 0 end with n - 1, but another condition is j always after i, so j begin with i + 1 end with n - 1**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int n = s.length();

        // dp[i][j] contains the answer of the longest palindromic

        // subsequence of the substring formed from index i to j in s

        int[][] dp = new int[n][n];

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

            dp[i][i] = 1;

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

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

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

                } else {

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

                }

            }

        }

        // The return dp[0][n - 1] means start index i = 0 to the end index

        // j = n - 1 dp result which cover the full length of the string

        return dp[0][n - 1];

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N^2)

**Style 2: Loop i up and loop j down**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int n = s.length();

        // dp[i][j] contains the answer of the longest palindromic

        // subsequence of the substring formed from index i to j in s

        int[][] dp = new int[n][n];

        for(int i = 0; i < n; i++) {

            dp[i][i] = 1;

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

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

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

                } else {

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

                }

            }

        }

        // The return dp[n - 1][0] means start index i = 0 to the end index

        // j = n - 1 dp result which cover the full length of the string

        return dp[n - 1][0];

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N^2)

**Refer to**

<https://leetcode.com/problems/longest-palindromic-subsequence/editorial/>

**Approach 2: Iterative Dynamic Programming**

**Intuition**

We used memoization in the preceding approach to store the answers to subproblems in order to solve a larger problem. We can also use a bottom-up approach to solve such problems without using recursion. We build answers to subproblems iteratively first, then use them to build answers to larger problems.

Using the same method as before, we create a 2D-array dp, where dp[i][j] contains the answer of the longest palindromic subsequence of the substring formed from index i to j in s. Our answer would be dp[0][n - 1], where n is the size of s. The state transition would be as follows:

1. If s[i] == s[j], perform dp[i][j] = 2 + dp[i + 1][j - 1].

2. Otherwise, perform dp[i][j] = max(dp[i][j - 1], dp[i + 1][j].

The dp array can be filled in a variety of ways. A few of them are briefly discussed below:

Building from smaller to larger strings: We can begin by selecting all possible substrings of length '1', then find the largest palindromic subsequence in all substrings of length '2', then in length '3', and so on to obtain the answer for the entire string.

Using two pointers: We can use two pointers, i and j, where i points to the first character of the substring under consideration and j points to the last character. Using dp entries corresponding to all the substrings formed by selecting indices within the range from i to j (inclusive), we form answers for all the substrings that start index i - 1. The pointer j moves from j = i - 1 to j = n - 1 to cover all possible substrings that start at index i - 1. (we can also choose to move from i to j + 1, i.e., from left to right). From the end of the string, we move from right to left, decrementing i by 1 until we reach the index 0. This is the approach we take here.

**Algorithm**

Create an integer variable n and initialize it to the size of s.

Create a 2D-array called dp having n rows and n columns where dp[i][j] contains the length of the longest palindromic subsequence of the substring formed from index i to j in s.

We iterate using two loops. The outer loop iterates from i = n - 1 to i = 0 decrementing i by 1 after each iteration. At the end of each iteration, we will have the length of longest palindromic subsequence in all the substrings that start from index i in s. For each i, we first mark dp[i][i] = 1 because it denotes just one character and then we iterate over j = i + 1 to j = n - 1 and perform the following:

If the first and the last characters are the same, i.e., s[i] == s[j], we include these two characters in the palindromic subsequence and add it to the longest palindromic subsequence formed using the substring from index i + 1 to j - 1 (inclusive). We perform dp[i][j] = dp[i + 1][j - 1] + 2. We already have the answer for dp[i + 1][j - 1] because we computed it for substrings starting at index i + 1 in the previous iteration of outer loop.

Otherwise, if the first and the last characters do not match, we look for the longest palindromic subsequence in both the substrings formed after ignoring the first and last characters. We pick the maximum of these two. We perform dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]).

Return dp[0][n - 1].

**Implementation**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int[][] dp = new int[s.length()][s.length()];

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

            dp[i][i] = 1;

            for (int j = i + 1; j < s.length(); j++) {

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

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

                } else {

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

                }

            }

        }

        return dp[0][s.length() - 1];

    }

}

**Complexity Analysis**

Here, n is the length of s.

Time complexity: O(n^2)

Initializing the dp array takes O(n^2)time.

We fill the dp array which takes O(n^2)time.

Space complexity: O(n^2)

The dp array consumes O(n^2)space.

**Style 3: Loop with length of current string as outer loop, and use i with length to deduce j**

**Refer to chatGPT**

To convert the above **DFS solution** for LeetCode 516 (Longest Palindromic Subsequence) into a **2D Dynamic Programming (DP) solution**, we can follow these steps:

**Steps to Convert to DP:**

Instead of making recursive calls, we will use a 2D DP table dp[i][j] to store the result for the substring s[i...j]. This will avoid recalculating the same subproblem multiple times.

**The DP table dp[i][j] will store the length of the longest palindromic subsequence in the substring s[i...j].**

Base case:

When i == j, the longest palindromic subsequence is the single character itself, so dp[i][j] = 1.

Recursive case:

If s.charAt(i) == s.charAt(j), then dp[i][j] = dp[i + 1][j - 1] + 2 (we include both characters s[i] and s[j]).

If s.charAt(i) != s.charAt(j), then dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]) (we either skip s[i] or s[j]).

We fill the DP table bottom-up, i.e., for each pair (i, j), where i <= j, and gradually solve the problem for larger substrings.

**2D DP Code:**

class Solution {

public int longestPalindromeSubseq(String s) {

int n = s.length();

        // dp[i][j] contains the answer of the longest palindromic

        // subsequence of the substring formed from index i to j in s

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

// Base case: A single character is a palindrome of length 1

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

dp[i][i] = 1;

}

// Fill the DP table, starting from shorter substrings to longer substrings

for (int len = 2; len <= n; len++) { // length of substring

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

int j = i + len - 1; // endpoint of the current substring

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

dp[i][j] = dp[i + 1][j - 1] + 2; // Characters match

} else {

dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]); // Characters don't match

}

}

}

// The answer for the whole string will be stored in dp[0][n-1]

return dp[0][n - 1];

}

}

**Explanation:**

Base Case:

For substrings of length 1 (i.e., i == j), we initialize dp[i][i] = 1 because a single character is always a palindrome of length 1.

Filling the DP Table:

We fill the DP table for substrings of increasing lengths (starting from length 2 to n).

For each substring s[i...j]:

If s[i] == s[j], we include both characters, so the value is dp[i + 1][j - 1] + 2.

If s[i] != s[j], we take the maximum result between skipping the current i or the current j, so the value is Math.max(dp[i + 1][j], dp[i][j - 1]).

Final Answer:

The value in dp[0][n - 1] will contain the length of the longest palindromic subsequence for the entire string.

**Example:**

Let's take s = "bbbab":

Initial DP Table:

0 1 2 3 4

b b b a b

0 1 0 0 0 0

1 0 1 0 0 0

2 0 0 1 0 0

3 0 0 0 1 0

4 0 0 0 0 1

After filling DP table:

0 1 2 3 4

b b b a b

0 1 2 3 3 4

1 0 1 2 2 3

2 0 0 1 1 3

3 0 0 0 1 1

4 0 0 0 0 1

The longest palindromic subsequence is "bbbb", so the result is 4.

**Time Complexity:**

Time Complexity: O(n^2) where n is the length of the string. This is because we fill a 2D DP table of size n x n and each cell is filled in constant time.

Space Complexity: O(n^2) for the 2D DP table.

**What's the difference between Style 3 and Style 1 ?**

**Refer to chatGPT**

The style of the **2D DP solution** you provided is functionally equivalent to the one I shared, but the difference lies in **the order in which the DP table is filled**:

**Key Differences:**

Order of filling the DP table:

Style 3 solution: Fills the DP table by starting from **shorter substrings to longer substrings**. The outer loop iterates over the length of the substring (len), and the inner loops over the starting index i and computes the corresponding ending index j.

Style 1 solution: Fills the DP table **in reverse order**, starting from **the last character** (i = n - 1) and working backward. The inner loop starts from j = i + 1, meaning it processes the characters from the right of the string toward the left.

Initialization:

Both solutions initialize dp[i][i] = 1 for single characters (since a single character is a palindrome of length 1).

In Style 3 solution, this is done outside the main loops, while in the Style 1 solution, it's done inside the loop for i when i == j.

Traversal Direction:

Style 3 solution: The traversal is **bottom-up** from shorter substrings to longer substrings.

Style 1 solution: The traversal is **right-to-left** for i and **left-to-right** for j.

**Why the Style 1 Works:**

In the Style 1 solution, the **bottom-left diagonal cells** of the DP table are filled first (i.e., smaller substrings), and the values in those cells are used to fill the **top-right diagonal** cells (i.e., larger substrings).

Since dp[i][j] depends on dp[i + 1][j - 1] (for matching characters) and dp[i + 1][j] or dp[i][j - 1] (for non-matching characters), the reverse order of filling (starting from i = n - 1) ensures that all the required subproblems are already solved when calculating larger substrings.

**Which Approach to Use:**

Both approaches are correct and will yield the same result because they follow the same recurrence relation.

My solution might be more intuitive for beginners because it progresses by calculating the result for smaller substrings first and gradually builds toward larger ones.

The **alternative solution** uses a more efficient loop structure by leveraging the reverse index order to avoid nested loop dependencies, making it a bit more optimized in terms of memory access patterns.

**Example:**

For s = "bbbab":

**Style 1 solution** starts filling from the bottom of the DP table:

Initial DP Table:

0 1 2 3 4

b b b a b

0 1 0 0 0 0

1 0 1 0 0 0

2 0 0 1 0 0

3 0 0 0 1 0

4 0 0 0 0 1

After filling in reverse:

0 1 2 3 4

b b b a b

0 1 2 3 3 4

1 0 1 2 2 3

2 0 0 1 1 3

3 0 0 0 1 1

4 0 0 0 0 1

Both solutions give the longest palindromic subsequence length as 4.

**Solution 4: DP with Space Optimization (10 min)**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int n = s.length();

        // dp[j] stores the length of longest palindromic subsequence

        // of the substring from index i to j in s. It is similar to

        // what dp[i][j] stored in previous approach

        int[] dp = new int[n];

        // dpPrev[j] stores the length of the longest palindromic

        // subsequence of the substring from index i + 1 to j in s.

        // It is analogous to dp[i + 1][j] in the previous approach.

        int[] dpPrev = new int[n];

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

            dp[i] = 1;

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

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

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

                } else {

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

                }

            }

            dpPrev = dp.clone();

        }

        // Return dp[n - 1] (or dpPrev[n - 1] as both are similar)

        return dp[n - 1];

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/longest-palindromic-subsequence/editorial/>

**Approach 3: Dynamic Programming with Space Optimization**

**Intuition**

The state transition, as we discussed in previous approaches, is:

1. If s[i] == s[j], perform dp[i][j] = 2 + dp[i + 1][j - 1].

2. Otherwise, perform dp[i][j] = max(dp[i][j - 1], dp[i + 1][j].

If we look closely at this transition, to fill dp[i][j] for a particular i and all possible values of j, we only need the values from the current and previous rows. To fill row i + 1 in the dp grid, we need the values from row i + 1(dp[i + 1][j - 1]) and previously computed value in the ith row itself (dp[i][j - 1]). Values in rows i + 2, i + 3, and so on are no longer needed.

Our task is complete if we can store the values of the previous iteration, i.e., for row i + 1 after each iteration of the outer loop.

We can solve this by using two 1D arrays of size n, dp and dpPrev, where n is the size of s. We repeat the previous approach by running two loops. The outer loop runs from i = n - 1to i = 0, and the inner loop runs from j = i + 1 to j = n - 1.

Now, dp[j] stores the length of longest palindromic subsequence of the substring from index i to j in s. It is similar to what dp[i][j] stored in previous approach.

The other array dpPrev is important to understand. It helps us by remembering the previous state that we completed previously. dpPrev[j] stores the length of the longest palindromic subsequence of the substring from index i + 1 to j in s. It is analogous to dp[i + 1][j] in the previous approach.

Because dpPrev stores the answers of substrings beginning with index i + 1 and dp stores the answers of substrings beginning with index i we must copy the elements of dp to dpPrev after iterating over all the substrings beginning with index i to prepare for the next iteration. After we copy dp to dpPrev, for the next iteration which considers substrings from i - 1, dpPrev will hold values of substrings beginning at index i which is exactly what we want.

**Algorithm**

Create an integer variable n and initialize it to the size of s.

Create two arrays called dp and dpPrev of size n.

We iterate using two loops with outer loop running from i = n - 1 to i = 0 decrementing i by 1after each iteration. For each i, we first mark dp[i] = 1since it denotes just one character at index i and then we iterate over j = i + 1to j = n - 1and perform the following:

If the first and the last characters are the same, i.e., s[i] == s[j], we include these two characters in the palindromic subsequence and add it to the longest palindromic subsequence formed using the substring from index i + 1to j - 1(inclusive). We perform dp[j] = dpPrev[j - 1] + 2. Note that we already have computed answer for substrings starting from index i + 1in the previous iteration of outer loop. We have it in dpPrev.

Otherwise, if the first and the last characters do not match, we check for the longest palindromic subsequence in both the substrings formed after ignoring the first and last characters. We pick the maximum of these two. We perform dp[j] = max(dpPrev[j], dp[j - 1]).

After the completion of inner loop, we copy dp to dpPrev.

Return dp[n - 1](or dpPrev[n - 1]as both are similar).

**Implementation**

class Solution {

    public int longestPalindromeSubseq(String s) {

        int n = s.length();

        int[] dp = new int[n];

        int[] dpPrev = new int[n];

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

            dp[i] = 1;

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

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

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

                } else {

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

                }

            }

            dpPrev = dp.clone();

        }

        return dp[n - 1];

    }

}

**Complexity Analysis**

Here, n is the length of s.

Time complexity: O(n^2)

Initializing the dp and dpPrev arrays take O(n) time.

To get the answer, we use two loops that take O(n^2) time.

Space complexity: O(n)

The dp and dpPrev arrays take O(n) space each.

**Refer to**

[L72.Edit Distance (Ref.L115,L712,L1143)](note://998ECECFA22D4E1E9533D9AD624A6345)

[L712.Minimum ASCII Delete Sum for Two Strings (Ref.L72,L583,L1143)](note://WEBa910883e40bb3571527f8753e2b859be)

[L583.Delete Operation for Two Strings (Ref.L712,L72,L1143)](note://WEB12ebad84cb577589b9576d7f802df5ee)

[L1143.Longest Common Subsequence (Ref.L516,L583,L712)](note://F9909063BABB4A67B28268E5F3054AED)