**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMk1D1DY1PL#problem-statement)

Given a sequence, find the length of its Longest Palindromic Subsequence (LPS). In a palindromic subsequence, elements read the same backward and forward.

A [subsequence](https://en.wikipedia.org/wiki/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: "abdbca"  
Output: 5  
Explanation: LPS is "abdba".

**Example 2:**

Input: = "cddpd"  
Output: 3  
Explanation: LPS is "ddd".

### Basic Solution [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMk1D1DY1PL#basic-solution)

A basic brute-force solution could be to try all the subsequences of the given sequence. We can start processing from the beginning and the end of the sequence. So at any step, we have two options:

1. If the element at the beginning and the end are the same, we increment our count by two and make a recursive call for the remaining sequence.
2. We will skip the element either from the beginning or the end to make two recursive calls for the remaining subsequence.

If option one applies then it will give us the length of LPS; otherwise, the length of LPS will be the maximum number returned by the two recurse calls from the second option.

The time complexity of the recursive algorithm is exponential O(2^n)*O*(2​*n*​​), where ‘n’ is the length of the input sequence. The space complexity is O(n)*O*(*n*) which is used to store the recursion stack.

### Top-down Dynamic Programming with Memoization [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMk1D1DY1PL#top-down-dynamic-programming-with-memoization)

We can use an array to store the already solved subproblems.

The two changing values to our recursive function are the two indexes, startIndex and endIndex. Therefore, we can store the results of all the subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (startIndex + “|” + endIndex))

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMk1D1DY1PL#code-2)

Here is the code for this:

class LPS {

  public int findLPSLength(String st) {

    Integer[][] dp = new Integer[st.length()][st.length()];

    return findLPSLengthRecursive(dp, st, 0, st.length()-1);

  }

  private int findLPSLengthRecursive(Integer[][] dp, String st, int startIndex, int endIndex) {

    if(startIndex > endIndex)

      return 0;

    // every sequence with one element is a palindrome of length 1

    if(startIndex == endIndex)

      return 1;

    if(dp[startIndex][endIndex] == null) {

      // case 1: elements at the beginning and the end are the same

      if(st.charAt(startIndex) == st.charAt(endIndex)) {

        dp[startIndex][endIndex] = 2 + findLPSLengthRecursive(dp, st, startIndex+1, endIndex-1);

      } else {

        // case 2: skip one element either from the beginning or the end

        int c1 =  findLPSLengthRecursive(dp, st, startIndex+1, endIndex);

        int c2 =  findLPSLengthRecursive(dp, st, startIndex, endIndex-1);

        dp[startIndex][endIndex] = Math.max(c1, c2);

      }

    }

    return dp[startIndex][endIndex];

  }

  public static void main(String[] args) {

    LPS lps = new LPS();

    System.out.println(lps.findLPSLength("abdbca"));

    System.out.println(lps.findLPSLength("cddpd"));

    System.out.println(lps.findLPSLength("pqr"));

  }

}

**What is the time and space complexity of the above solution?** Since our memoization array dp[st.length()][st.length()] stores the results for all the subproblems, we can conclude that we will not have more than N\*N*N*∗*N* subproblems (where ‘N’ is the length of the input sequence). This means that our time complexity will be O(N^2)*O*(*N*​2​​).

The above algorithm will be using O(N^2)*O*(*N*​2​​) space for the memoization array. Other than that we will use O(N)*O*(*N*) space for the recursion call-stack. So the total space complexity will be O(N^2 + N)*O*(*N*​2​​+*N*), which is asymptotically equivalent to O(N^2)*O*(*N*​2​​).

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMk1D1DY1PL#bottom-up-dynamic-programming)

Since we want to try all the subsequences of the given sequence, we can use a two-dimensional array to store our results. We can start from the beginning of the sequence and keep adding one element at a time. At every step, we will try all of its subsequences. So for every startIndex and endIndex in the given string, we will choose one of the following two options:

1. If the element at the startIndex matches the element at the endIndex, the length of LPS would be two plus the length of LPS till startIndex+1 and endIndex-1.
2. If the element at the startIndex does not match the element at the endIndex, we will take the maximum LPS created by either skipping element at the startIndex or the endIndex.

So our recursive formula would be:

1

2

3

4

if st[endIndex] == st[startIndex]

  dp[startIndex][endIndex] = 2 + dp[startIndex + 1][endIndex - 1]

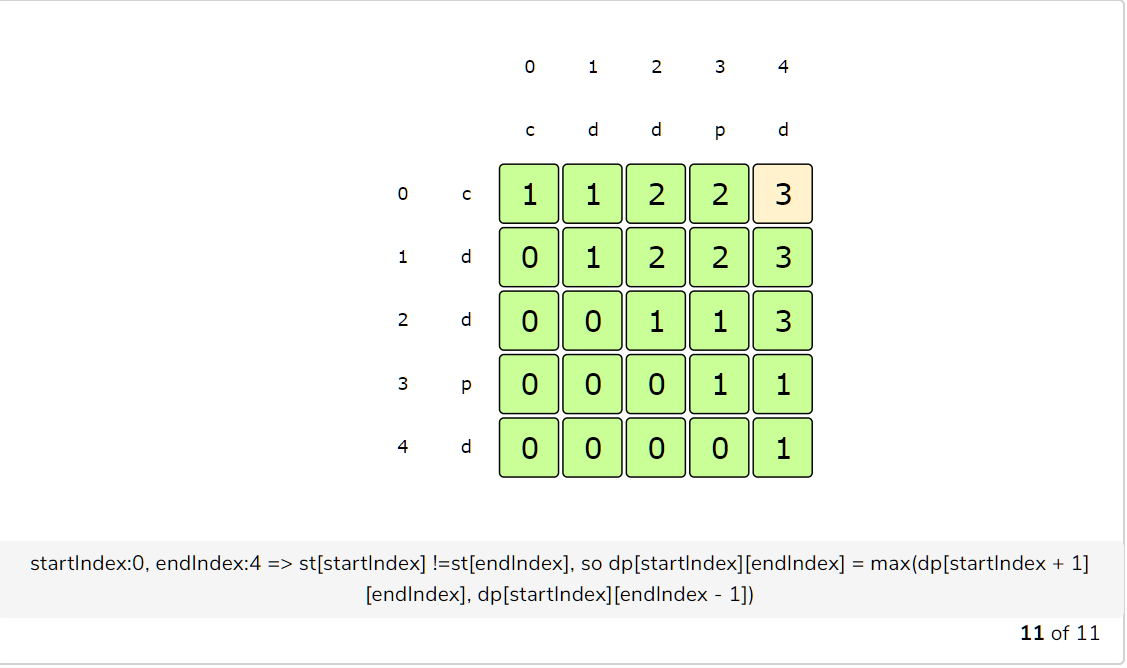
else

  dp[startIndex][endIndex] = Math.max(dp[startIndex + 1][endIndex], dp[startIndex][endIndex - 1])





Let’s draw this visually for “cddpd”, starting with a subsequence of length ‘1’. As we know, every sequence with one element is a palindrome of length 1:



From the above visualization, we can clearly see that the length of LPS is ‘3’ as shown by dp[0][4].

Here is the code for our bottom-up dynamic programming approach:

class LPS {

  public int findLPSLength(String st) {

    // dp[i][j] stores the length of LPS from index 'i' to index 'j'

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

    // every sequence with one element is a palindrome of length 1

    for (int i = 0; i < st.length(); i++)

      dp[i][i] = 1;

    for (int startIndex = st.length() - 1; startIndex >= 0; startIndex--) {

      for (int endIndex = startIndex + 1; endIndex < st.length(); endIndex++) {

        // case 1: elements at the beginning and the end are the same

        if (st.charAt(startIndex) == st.charAt(endIndex)) {

          dp[startIndex][endIndex] = 2 + dp[startIndex + 1][endIndex - 1];

        } else { // case 2: skip one element either from the beginning or the end

          dp[startIndex][endIndex] = Math.max(dp[startIndex + 1][endIndex], dp[startIndex][endIndex - 1]);

        }

      }

    }

    return dp[0][st.length() - 1];

  }

  public static void main(String[] args) {

    LPS lps = new LPS();

    System.out.println(lps.findLPSLength("abdbca"));

    System.out.println(lps.findLPSLength("cddpd"));

    System.out.println(lps.findLPSLength("pqr"));

  }

}

The time and space complexity of the above algorithm is O(n^2)*O*(*n*​2​​), where ‘n’ is the length of the input sequence.