**Longest Palindromic Substring**

**WE'LL COVER THE FOLLOWING**

* + - [Problem Statement](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#problem-statement)
    - [Basic Solution](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#basic-solution)
    - [Top-down Dynamic Programming with Memoization](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#top-down-dynamic-programming-with-memoization)
    - [Bottom-up Dynamic Programming](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#bottom-up-dynamic-programming)
    - [Manacher’s Algorithm](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#manachers-algorithm)

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#problem-statement)

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

**Example 1:**

Input: "abdbca"  
Output: 3  
Explanation: LPS is "bdb".

**Example 2:**

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

**Example 3:**

Input: = "pqr"  
Output: 1  
Explanation: LPS could be "p", "q" or "r".

**Basic Solution**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#basic-solution)

This problem follows the [Longest Palindromic Subsequence](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5748119283171328/) pattern. The only difference is that in a palindromic subsequence characters can be non-adjacent, whereas in a substring all characters should form a palindrome. We will follow a similar approach though.

The brute-force solution will be to try all the substrings of the given string. We can start processing from the beginning and the end of the string. So at any step, we will have two options::

1. If the element at the beginning and the end are the same, we make a recursive call to check if the remaining substring is also a palindrome. If so, the substring is a palindrome from beginning to the end.
2. We will skip either the element from the beginning or the end to make two recursive calls for the remaining substring. The length of LPS would be the maximum of these two recursive calls.

Here is the code:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)

class LPS {

public int findLPSLength(String st) {

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

}

private int findLPSLengthRecursive(String st, int startIndex, int endIndex) {

if (startIndex > endIndex)

return 0;

// every string with one character is a palindrome

if (startIndex == endIndex)

return 1;

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

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

int remainingLength = endIndex - startIndex - 1;

// check if the remaining string is also a palindrome

if (remainingLength == findLPSLengthRecursive(st, startIndex + 1, endIndex - 1))

return remainingLength + 2;

}

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

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

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

return Math.max(c1, c2);

}

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"));

}

}

RUN

SAVERESET

Due to the three recursive calls, the time complexity of the above algorithm is exponential O(3^n)*O*(3​*n*​​), where ‘n’ is the length of the input string. 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/m2yRjwxBY7A#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))

Here is the code for this:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)

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 string with one character is a palindrome

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)) {

int remainingLength = endIndex - startIndex - 1;

// check if the remaining string is also a palindrome

if (remainingLength == findLPSLengthRecursive(dp, st, startIndex + 1, endIndex - 1)) {

dp[startIndex][endIndex] = remainingLength + 2;

return dp[startIndex][endIndex];

}

}

// case 2: skip one character 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"));

}

}

RUN

SAVERESET

The above algorithm has a time and space complexity of O(n^2)*O*(*n*​2​​) because we will not have more than n\*n*n*∗*n* subproblems.

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#bottom-up-dynamic-programming)

Since we want to try all the substrings of the given string, we can use a two-dimensional array to store the subproblems’ results. So dp[i][j] will be ‘true’ if the substring from index ‘i’ to index ‘j’ is a palindrome.

We can start from the beginning of the string and keep adding one element at a time. At every step, we will try all of its substrings. So for every endIndex and startIndex in the given string, we need to check the following thing:

If the element at the startIndex matches the element at the endIndex, we will further check if the remaining substring (from startIndex+1 to endIndex-1) is a substring too.

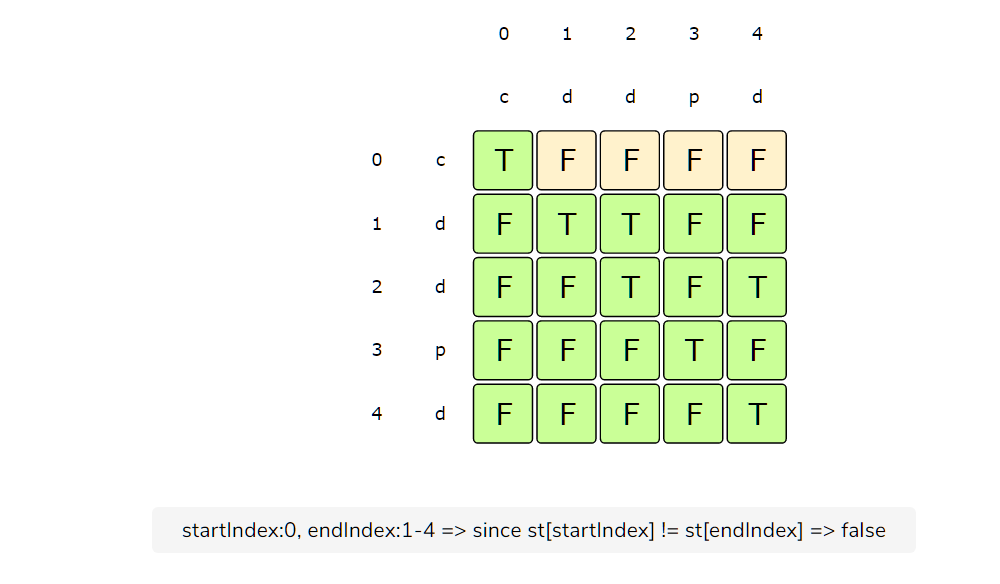
So our recursive formula will look like:

if st[startIndex] == st[endIndex], and

if the remaing string is of zero length or dp[startIndex+1][endIndex-1] is a palindrome then

dp[startIndex][endIndex] = true

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



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

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A)

class LPS {

public int findLPSLength(String st) {

// dp[i][j] will be 'true' if the string from index 'i' to index 'j' is a

// palindrome

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

// every string with one character is a palindrome

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

dp[i][i] = true;

int maxLength = 1;

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

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

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

// if it's a two character string or if the remaining string is a palindrome too

if (endIndex - startIndex == 1 || dp[startIndex + 1][endIndex - 1]) {

dp[startIndex][endIndex] = true;

maxLength = Math.max(maxLength, endIndex - startIndex + 1);

}

}

}

}

return maxLength;

}

public static void main(String[] args) {

LPS lps = new LPS();

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

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

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

}

}

RUN

SAVERESET

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

**Manacher’s Algorithm**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2yRjwxBY7A#manachers-algorithm)

The best-known algorithm to find the longest palindromic substring which runs in linear time O(n)*O*(*n*) is [Manacher’s Algorithm](https://en.wikipedia.org/wiki/Longest_palindromic_substring" \t "_blank). However, it is a non-trivial algorithm that doesn’t use DP. Please take a look to familiarize yourself with this algorithm, however, no one expects you to come up with such an algorithm in a 45 minutes coding interview.

**COMPLETED**

[**←    Back**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMk1D1DY1PL)

Longest Palindromic Subsequence

[**Next    →**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/xV73LEk5rx9)

Count of Palindromic Substrings

Stuck? Get help on

[**DISCUSS**](https://discuss.educative.io/c/grokking-dynamic-programming-patterns-for-coding-interviews-design-gurus/pattern-3-palindromic-subsequence-longest-palindromic-substring)

Send feedback

16 Recommendations