5. Longest Palindromic Substring String ] Medium **Dynamic Programming Leetcode Link** 

## **Problem Description** The goal of this problem is to find the longest palindromic substring within a given string s. A palindromic string is a string that reads

To understand the problem, let's consider what makes up a palindrome:

A single character is always a palindrome.

A substring of three or more characters is a palindrome if its first and last characters are the same, and the substring obtained

- by removing them is also a palindrome. Given these observations, we need an algorithm that can check for palindromic substrings efficiently and keep track of the longest

the same backward as forward, such as 'radar' or 'level'.

Two characters are a palindrome if they are identical.

one found so far. Intuition

The solution involves Dynamic Programming (DP), an optimization technique that solves complex problems by breaking them into simpler subproblems, storing the solution to each subproblem, and reusing those solutions.

## table in a bottom-up manner. For every substring length (from 2 to n), we set dp[i][j] to true if the corresponding substring is a

palindrome.

palindrome. Here's the process: For substrings of length 1, each is trivially a palindrome.

Solution 1: Dynamic Programming The idea is to use a 2D table dp to store whether a substring s[i..j] is a palindrome. We fill this

 For substrings of length 2, they are palindromes if both characters are the same. • For longer substrings, we check if the first and last characters are the same and if the substring obtained by removing them (dp[i+1][j-1]) is a palindrome.

If dp[i][j] is true, we check if it is the longest palindrome so far. If it is, we update the starting index and maximum length of the

Solution 2: Enumerating the Palindrome Center An alternative approach is to consider each possible center of the palindrome

- (which could be a character or between two characters), and expand outwards from the center to see how far the palindrome can be
- extended. We keep track of the length and starting point of the longest palindrome found during the process.

checking if removing the matching characters still leaves a palindrome.

Implementing the DP Solution The provided Python code uses the dynamic programming approach. Here's a breakdown of its parts: • It initializes the dp table (f in the code) to all True for the length 1 substrates, and then iterates backwards over the string.

• For each pair (i, j) it checks if s[i] matches s[j], then it sets f[i][j] to whatever the value at f[i+1][j-1] is, effectively

 It keeps track of the start position k and the max length mx of the longest palindrome. The time complexity of this approach is O(n²) because it examines all possible substrings, making it practical for reasonably short

strings.

The solution to finding the longest palindromic substring employs two main algorithms - Dynamic Programming and Center Expansion. Below is a walkthrough for both.

Initialize a n by n matrix dp with True values along the diagonal, indicating that all single characters are palindromes.

For each length L, iterate over all possible starting indices i from which a substring of length L can be obtained.

Iterate over all possible substring lengths L starting from 2 to the length of the input string n.

Track the starting index and maximum length of the longest palindromic substring found so far.

Use the ending index j = i+L-1 to cover the substring of length L starting at index i.

Dynamic Programming Solution The dynamic programming approach creates a table dp where each entry dp[i][j] records whether

 Set dp[i][j] to True if and only if the end characters match (s[i] == s[j]) and the internal substring s[i+1..j-1] is a palindrome (dp[i+1][j-1] == True).

8 for L in range(2, n + 1):

10

11

17

18

8

9

9

10

11

13

14

15

19

20

21

10

time.

2 k, mx = 0, 1

diagonal element below it.

else:

1 f = [[True] \* n for \_ in range(n)]

for j in range(i + 1, n):

f[i][j] = False

f[i][j] = True

if mx < j - i + 1:

**if** s[i] != s[j]:

# Odd Length Palindromes

mx\_len = odd\_len

# Even Length Palindromes

mx\_len = even\_len

if even\_len > mx\_len:

17 def expandFromCenter(s, n, l, r):

l -= 1

r += 1

programming approach.

the application at hand.

Example Walkthrough

0

0 0 0 1

0

0

0

starting at index 1, both with a length of 3.

 $dp = [[False] * n for _ in range(n)]$ 

for i in range(n - L + 1):

 $mx_len = L$ 

start = i

dp[i][j] = (s[i] == s[j])

def longestPalindrome(self, s: str) -> str:

# Table to store the palindrome status

n = len(s) # Length of the string

dp = [[True] \* n for \_ in range(n)]

if dp[i][j] and L > mx\_len:

longest\_palindrome = s[start:start+mx\_len]

j = i + L - 1

if L == 2:

else:

Python Solution

1 class Solution:

4. We then proceed to substrings of length 4 and 5 but find no longer palindromes.

Here is a snippet of Python code that reflects the process for this example:

dp[i][j] = (s[i] == s[j] and dp[i + 1][j - 1])

print(f'The longest palindrome in the string is: "{longest\_palindrome}"')

# dp[i][j] will be 'True' if the string from index i to j is a palindrome.

start\_index = 0 # Start index of the longest palindrome found

dp[i][j] = False # Initially set to False

if s[i] == s[j] and dp[i + 1][j - 1]:

if  $max_length < j - i + 1$ :

start\_index = i

return s[start\_index : start\_index + max\_length]

# Return the longest palindrome substring

# Check if the current substring is a palindrome

// Extract the longest palindromic substring from the string.

// Finds the longest palindromic substring in a given string 's'

// Create a 2D vector 'dp' to store palindrome information.

// Initialize one-character and two-character palindromes

// dp[i][j] will be true if the substring s[i..j] is a palindrome.

std::vector<std::vector<bool>> dp(n, std::vector<bool>(n, false));

dp[i][i] = true; // Substrings of length 1 are palindromes.

max\_length = 2; // Update the max\_length to 2

dp[i][j] = true; // Mark as palindrome

int start\_index = 0; // Starting index of the longest palindrome found

int max\_length = 1; // Length of the longest palindrome found, at least 1 because single characters are palindromes.

dp[i][i + 1] = true; // Substrings of length 2 are palindromes if both characters are equal.

// Check if the current substring is a palindrome using previously calculated sub-problems

start\_index = i; // Update the starting index of the longest palindrome

int j = i + len - 1; // Calculate the end index of the current substring

// Update max\_length and start\_index if a bigger palindrome is found

// Extract and return the longest palindrome substring from the original string

std::string longestPalindrome(std::string s) {

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

int n = s.size(); // Length of the given string

if (i < n - 1 && s[i] == s[i + 1]) {

// Check for palindromes of length 3 and more

for (int i = 0; i < n - len + 1; ++i) {

if(len > max\_length) {

return s.substr(start\_index, max\_length);

start\_index = i;

max\_length = len;

 $if(s[i] == s[j] && dp[i + 1][j - 1]) {$ 

for (int len = 3; len <= n; ++len) {</pre>

return s.substring(startIdx, startIdx + maxLength);

dp[i][j] = True # Update the palindrome status

 $max_length = j - i + 1$  # Update  $max_length$ 

# Check if the current palindrome is the longest found so far

Following the dynamic programming strategy:

return r - l - 1

 $start = i - mx_len // 2$ 

if odd len > mx len:

odd\_len = expandFromCenter(s, n, i, i)

 $start = i - (mx_len - 1) // 2$ 

while l >= 0 and r < n and s[l] == s[r]:

even\_len = expandFromCenter(s, n, i, i + 1)

3 for i in range(n - 2, -1, -1):

for i in range(n - L + 1):

 $mx_len = L$ 

start = i

j = i + L - 1

if L == 2:

**Solution Approach** 

the substring s[i..j] is a palindrome.

dp = [[False] \* n for \_ in range(n)] for i in range(n): dp[i][i] = True # Base case for one-letter palindrome mx len = 1 start = 0

12 dp[i][j] = (s[i] == s[j])13 else: dp[i][j] = (s[i] == s[j] and dp[i + 1][j - 1])14 15 if dp[i][j] and L > mx\_len: 16

• The time complexity of this approach is  $0(n^2)$  as we check all n(n-1)/2 substrings and updating the dp table takes constant

In the code, a 2D list f represents the dp table, where we fill the table starting from the second to last row to the first (i = n - 2) to 0

and from left to right (j = i + 1 to n) within each row. This is because dp[i][j] depends on dp[i + 1][j - 1], which is the next

k, mx = i, j - i + 111 12 else: 13 f[i][j] = FalseHere, the variable k tracks the starting index of the longest palindrome, and mx tracks the length of the longest palindrome. Center Expansion Solution The center expansion algorithm doesn't require any additional space except for keeping track of the longest palindrome found. 1 mx len = 12 start = 0 for i in range(n):

if j - i == 1 or f[i + 1][j - 1]:

The time complexity for this approach is also  $0(n^2)$  because each expansion may take 0(n) time and there are 0(n) potential

odd and even centers are compared to update the maximum palindrome length mx\_len and the starting index start.

centers to consider. However, this approach does not use extra space, making it more space-efficient than the dynamic

Let's consider a small example to illustrate the solution approach, specifically the dynamic programming solution.

Suppose the input string is s = "babad". We are interested in finding the longest palindromic substring.

Here, the expandFromCenter function checks for the longest palindrome centered at 1 (for odd lengths) and between 1 and r (for

even lengths), expanding its boundaries outwards as long as the characters match. The lengths of the longest palindromes for both

Both approaches are efficient for different scenarios and can be used depending on the space-time trade-off that is more critical for

True, representing that every single character is a palindrome by itself. Our dp table initially looks something like this (where 0 indicates False and diagonals are implicitly True):

2. We check two-character substrings (length L = 2) for palindromicity. We notice that "ba" is not a palindrome, "ab" is not a

palindrome, "ba" again is not a palindrome, but "ad" is not a palindrome either. So, our dp table does not change.

1. Initialize the dp table for a string of length 5 (since "babad" is 5 characters long). Each cell dp[i][i] along the diagonal is set to

## 3. Now we move on to substrings of length 3. We find "bab" is a palindrome and so is "aba". The table updates to:

0

0

= "babad"

13

14

15

16

17

16

17

18

19

20

21

22

23

24

25

26

27

28

26

27

28

29

30

31

33

32 }

C++ Solution

public:

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

42

43

44

45

46

47

48

49

**}**;

Typescript Solution

1 #include <vector>

2 #include <string>

class Solution {

Java Solution

class Solution {

0

0

for i in range(n): dp[i][i] = True  $6 \text{ mx\_len} = 1$ 7 start = 0 for L in range(2, n + 1):

When this code is executed, the longest palindrome "bab" or "aba" (whichever comes first in the updating process) will be printed.

The 2D dp matrix serves as a memory that helps avoid recalculating whether a substring is a palindrome, thus saving computational

time. Each step relies on the information gathered in the previous steps, making this a bottom-up dynamic programming approach.

5. The dynamic programming algorithm records the longest palindrome we found, which is "bab" starting at index 0 or "aba"

10 max\_length = 1 # Length of the longest palindrome found, initially 1 character 11 12 # Bottom-up dynamic programming approach. # Start from the end of the string and move towards the beginning. 13 for i in range(n - 2, -1, -1): # i is the start index of the substring 14 for j in range(i + 1, n): # j is the end index of the substring 15

```
public String longestPalindrome(String s) {
            int n = s.length(); // Get the length of the string.
           boolean[][] dp = new boolean[n][n]; // Create a dynamic programming (DP) table.
           // Initialize all substrings of length 1 (single character) as a palindrome.
6
           for (boolean[] row : dp) {
               Arrays.fill(row, true);
9
10
           int startIdx = 0; // Starting index of the longest palindromic substring found.
11
           int maxLength = 1; // Length of the longest palindromic substring found, initialized with length 1.
12
13
           // Build the DP table in a bottom-up manner.
14
           for (int i = n - 2; i \ge 0; --i) { // Start from the second last character and move backwards.
15
                for (int j = i + 1; j < n; ++j) { // Compare it with characters ahead of it.
16
                    dp[i][j] = false; // Initialize the current substring (i, j) as not palindrome.
18
                    if (s.charAt(i) == s.charAt(j)) { // If the characters match,
19
                        dp[i][j] = dp[i + 1][j - 1]; // Check if removing them gives a palindrome.
20
                       // Update the start position and max length if a larger palindrome is found.
                        if (dp[i][j] \&\& maxLength < j - i + 1) {
21
22
                            maxLength = j - i + 1;
23
                            startIdx = i;
24
25
```

## 36 37 38 39 40 41

function longestPalindrome(s: string): string { // Get the length of the string. const length = s.length; // Initialize a 2D array f to track palindromes, initializing all entries to true. const isPalindrome: boolean[][] = Array(length) .fill(0) .map(() => Array(length).fill(true)); // Initialize a variable to store the starting index of the longest palindrome found. 9 let startIndex = 0; 10 // Initialize a variable to store the length of the longest palindrome found. 11 12 let maxLength = 1; 13 // Loop through the string from end to beginning, adjusting the isPalindrome matrix. 14 for (let  $i = length - 2; i >= 0; --i) {$ 15 for (let j = i + 1; j < length; ++j) { // Invalidate the current state since it's not a single character string. 17 18 isPalindrome[i][j] = false; // If the characters at i and j are the same, check the inner substring. 19 **if** (s[i] === s[j]) { 20 21 // Set the state based on the substring inside the current bounds. isPalindrome[i][j] = isPalindrome[i + 1][j - 1]; 23 // If the substring is a palindrome and larger than the current maxLength, update maxLength and startIndex. 24 if (isPalindrome[i][j] && maxLength < j - i + 1) { 25 maxLength = j - i + 1;startIndex = i; 26 27 28 29 30 31 // Get the longest palindrome substring from the given string based on startIndex and maxLength. 32 33 return s.slice(startIndex, startIndex + maxLength); 34 } 35

The time complexity of the code provided is  $0(n^2)$ , as it includes two nested loops that both iterate over the string length n. steps in terms of the length of the input string s.

Time and Space Complexity

Specifically, the outer loop counts down from n-2 to 0, and the inner loop counts up from i+1 to n, resulting in a quadratic number of The space complexity, however, differs from the reference answer. The code creates a 2D list f sized n by n, which is used to store boolean values representing whether a substring is a palindrome or not. This means the space complexity is not 0(1), but in fact 0(n^2) as well, because storing these values requires a space proportional to the square of the length of the input string s.