May 1, 2024

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Algorithm Engineering Lab 05

Azmayen Fayek Sabil, 190042122

Contents

1 Task 1

1.1 Palindromic Substring

Given a string s, the task is to find the longest palindromic substring in s.

```
    Example 1:
        Input: s = "babad"
        Output: "bab"
        Explanation: "aba" is also a valid answer.
    Example 2:
        Input: s = "cbbd"
        Output: "bb"

    Constraints:
        1 <= s.length <= 1000
        s consist of only digits and English letters.
```

1.1.1 Approach to Finding the Longest Palindromic Substring

1. Iterate through each character in the string.
2. For each character, consider it as the center of a potential palindrome.
3. Expand around the center to check if the substring formed is a palindrome.
4. Keep track of the longest palindrome found so far.
5. Repeat steps 2-4 for both odd-length and even-length palindromes.
6. Return the longest palindromic substring found.

1.1.2 Python Solution

```python
def longest_palindromic_substring(s: str) -> str:
    if len(s) < 2:
        return s

    start = 0
    max_length = 1

    for i in range(len(s)):
        # Check odd-length palindromes
        left, right = i - 1, i + 1
        while left >= 0 and right < len(s) and s[left] == s[right]:
            if right - left + 1 > max_length:
                start = left
                max_length = right - left + 1
            left -= 1
            right += 1

        # Check even-length palindromes
        left, right = i, i + 1
        while left >= 0 and right < len(s) and s[left] == s[right]:
            if right - left + 1 > max_length:
                start = left
                max_length = right - left + 1
            left -= 1
            right += 1
```

```
26
27      return s[start:start + max_length]
28
29  # Example usage
30  s1 = "babad"
31  print(longest_palindromic_substring(s1))  # Output: "bab" or "aba"
32
33  s2 = "cbbd"
34  print(longest_palindromic_substring(s2))  # Output: "bb"
```

Listing 1: Python solution to find the longest palindromic substring

1.1.3 Output Explanation

For the given example input:

- In Example 1, the longest palindromic substring in ”babad” is ”bab” or ”aba”.
- In Example 2, the longest palindromic substring in ”cbbd” is ”bb”.

1.2 Complexity Analysis

1.2.1 Time Complexity

The time complexity of the solution is $O(n^2)$, where n is the length of the input string s. This is because we iterate through each character in the string and for each character, we expand around it to find the longest palindromic substring centered at that character.

1.2.2 Time Complexity

Space Complexity The space complexity of the solution is $O(1)$ because we use only a constant amount of extra space for storing variables like indices and lengths.

2 Task 2

2.1 Container With Most Water

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]). Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. Notice that you may not slant the container.

2.1.1 Python Implementation

```python
def max_area(height):
    left = 0
    right = len(height) - 1
    maxi = 0
    while left < right:
        w = right - left
        h = min(height[left], height[right])
        area = h * w
        maxi = max(maxi, area)
        if height[left] < height[right]:
            left += 1
        elif height[left] > height[right]:
            right -= 1
        else:
            left += 1
            right -= 1
    return maxi

# Example usage:
height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
print(max_area(height))  # Output: 49
```

Listing 2: Python solution for finding the maximum area of water trapped

2.1.2 Explanation

This Python code provides a procedural implementation to find the maximum area of water that can be trapped between vertical lines represented by the input array of heights. Here's a breakdown of the code:

1. Initialize two pointers, `left` and `right`, pointing to the start and end of the height array, respectively.
2. Initialize a variable `maxi` to store the maximum area of water.
3. Use a `while` loop to iterate until the `left` pointer is less than the `right` pointer.
4. Calculate the width of the container as the difference between `right` and `left`.
5. Calculate the height of the container as the minimum of the heights at the `left` and `right` pointers.
6. Calculate the area of the container as the product of width and height.
7. Update `maxi` to store the maximum of the current area and the previous maximum.
8. Move the pointers:

- If the height at `left` is less than the height at `right`, increment `left`.
- If the height at `left` is greater than the height at `right`, decrement `right`.
- If the heights are equal, increment `left` and decrement `right`.

9. Return `maxi` as the maximum area of water trapped.

2.1.3 Output Analysis

For the example usage provided, the output would be 49, which represents the maximum area of water that can be trapped between the vertical lines represented by the input array of heights.

2.2 Complexity

2.2.1 Time Complexity

The time complexity of this solution is $O(n)$, where n is the number of elements in the input array. This is because we are iterating through the array once with two pointers.

2.2.2 Space Complexity

The space complexity of this solution is $O(1)$ because we are using a constant amount of extra space for variables regardless of the size of the input array.