Day 14

Prepared By Azan Imtiaz

Palindromic Substring

Given a string `s`, return *the number of **palindromic substrings** in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

Example 1:

```
Input: s = "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".
```

Example 2:

```
Input: s = "aaa"
Output: 6
Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".
```

Intuition

The goal is to count all palindromic substrings in a given string. A palindrome reads the same backward as forward. The intuition behind this approach is to check for palindromes by expanding around potential centers.

Approach

1. Treat each character and each pair of adjacent characters in the string as potential centers of palindromes.
2. For each center, expand outwards and count palindromes while the characters on both sides match.
3. Consider two scenarios:
   - Odd-length palindromes with a single character center.
   - Even-length palindromes with two consecutive characters as the center.

4. The `countPalindromesAroundCenter` method expands outwards from the center while the characters match, counting all valid palindromes.

Solution

```
class Solution {
    public int countSubstrings(String s) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            count += countPalindromesAroundCenter(s, i, i); // Odd length palindromes
            count += countPalindromesAroundCenter(s, i, i + 1); // Even length
palindromes
        }
        return count;
    }

    private int countPalindromesAroundCenter(String s, int left, int right) {
        int count = 0;
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            count++;
            left--;
            right++;
        }
        return count;
    }
}
```

Time Complexity

The time complexity is O(n^2), where n is the length of the string. This is because, for each character, we potentially scan outward in both directions.

Detail Dry Run

Let's consider the string `"abc"`:

1. Start with `i=0` (character 'a'):
   - Check odd length: `countPalindromesAroundCenter("abc", 0, 0)` finds "a".
   - Check even length: `countPalindromesAroundCenter("abc", 0, 1)` finds no palindrome.
2. Move to `i=1` (character 'b'):
   - Check odd length: `countPalindromesAroundCenter("abc", 1, 1)` finds "b".

- Check even length: `countPalindromesAroundCenter("abc", 1, 2)` finds no palindrome.

3. Finally, `i=2` (character 'c'):
   - Check odd length: `countPalindromesAroundCenter("abc", 2, 2)` finds "c".
   - Check even length: `countPalindromesAroundCenter("abc", 2, 3)` goes beyond the string length and finds no palindrome.

Total count of palindromic substrings is 3.

Minimum Window Substring

Problem:

Given two strings `s` and `t` with lengths `m` and `n` respectively, find the smallest substring in `s` that contains all characters in `t` (including duplicates). If no such substring exists, return an empty string `""`. The solution must be unique for the given test cases.

Explanation:

The challenge is to locate the shortest segment of `s` that fully encompasses `t`. This means every letter from `t` must appear at least as many times in that segment of `s`.

Intuition:

The idea is to use a sliding window to scan through `s` while tracking the frequency of characters in both `s` and `t`. When the window includes all characters of `t`, we try to shrink it without losing any necessary characters.

Approach:

1. Create a frequency map for `t`.
2. Use two pointers to create a sliding window of `s`.
3. Expand the window by moving the right pointer and update the frequency map for `s`.
4. When all characters of `t` are within the window, try to shrink it by moving the left pointer.
5. Keep track of the minimum length window that contains all characters of `t`.
6. Return the smallest window or an empty string if no such window exists.

Solution:

```
class Solution {
    public String minWindow(String s, String t) {
        if (s == null || t == null || s.length() < t.length()) {
            return "";
        }

        HashMap<Character, Integer> tMap = new HashMap<>();
        for (char c : t.toCharArray()) {
            tMap.put(c, tMap.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0;
        int minLength = Integer.MAX_VALUE, minStart = 0, count = 0;
        HashMap<Character, Integer> sMap = new HashMap<>();

        while (right < s.length()) {
            char rightChar = s.charAt(right);
            sMap.put(rightChar, sMap.getOrDefault(rightChar, 0) + 1);

            if (tMap.containsKey(rightChar) && sMap.get(rightChar).intValue() <=
tMap.get(rightChar).intValue()) {
                count++;
            }

            while (count == t.length()) {
                if (right - left + 1 < minLength) {
                    minLength = right - left + 1;
                    minStart = left;
                }

                char leftChar = s.charAt(left);
                sMap.put(leftChar, sMap.get(leftChar) - 1);

                if (tMap.containsKey(leftChar) && sMap.get(leftChar).intValue() <
tMap.get(leftChar).intValue()) {
                    count--;
                }

                left++;
            }

            right++;
        }

        return minLength == Integer.MAX_VALUE ? "" : s.substring(minStart, minStart +
minLength);
    }
}
```

Time Complexity:

The time complexity is O(m + n), where m is the length of string `s` and n is the length of string `t`. This is because each character in `s` is visited at most twice, once by the right pointer and once by the left pointer.

Dry Run:

Let's dry run the provided Example 1:

- Input: `s = "ADOBECODEBANC"`, `t = "ABC"`
- Initialize maps and variables.
- Expand window until `right = 5` (`"ADOBEC"`), now `count = 3` (all `t` characters are found).
- Start shrinking from `left` until `left = 2` (`"BEC"`), `count` drops below `t.length()`.
- Continue expanding and shrinking to find `minLength` which is `"BANC"`.
- Return `"BANC"`.