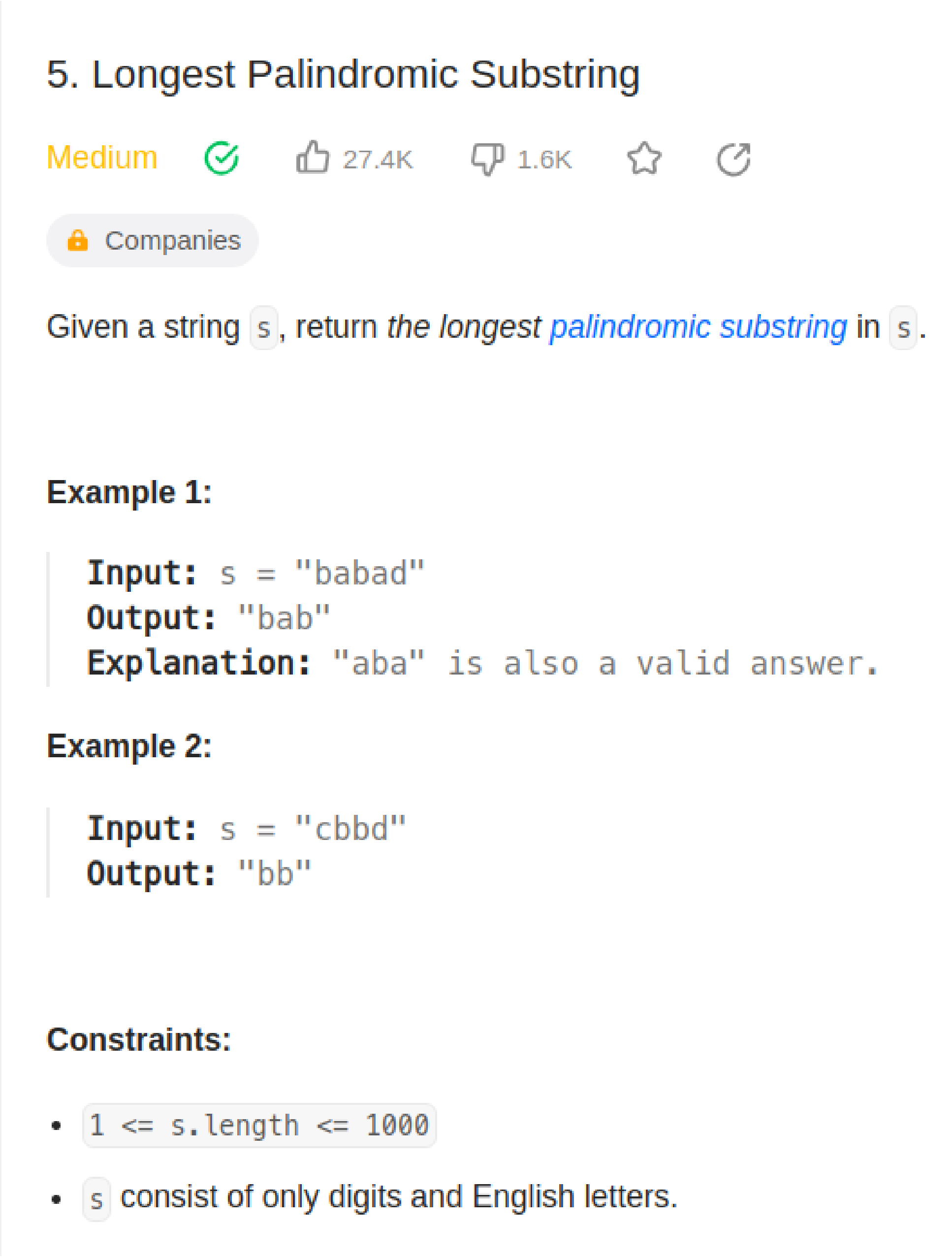
MCS – 253P ADVANCED PROGRAMMING AND PROBLEM SOLVING

HOMEWORK –2

Aswin Sampath
saswin@uci.edu

(53844684)

QUESTION 1: Longest Palindromic Substring



5. Longest Palindromic Substring

Medium ✓ 👍 27.4K 👎 1.6K ☆

Given a string `s`, return *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.

Understanding the Problem:

We are given a string s, and the task is to find the longest palindromic substring within it.

Identifying Edge Cases:

The constraints state that s has a length between 1 and 1000 and consists of only digits and English letters. Edge cases to consider:

- A single-character string: In this case, the answer should be the same character.
- A string without any palindromic substring: In this case, the answer should be any single character from the string.

Effective Test Cases:

To test our solution, consider the following cases:

1. s = "babad" (Example 1): The output should be "bab" or "aba."
2. s = "cbbd" (Example 2): The output should be "bb."
3. s = "abcde": This is a string with no palindromic substrings, so the output should be any single character from the string.
4. s = "a": The output should be "a" because it is a single character.
5. s = "abcdefedcba": The output should be the entire string because it is a palindrome.

Algorithmic Solution:

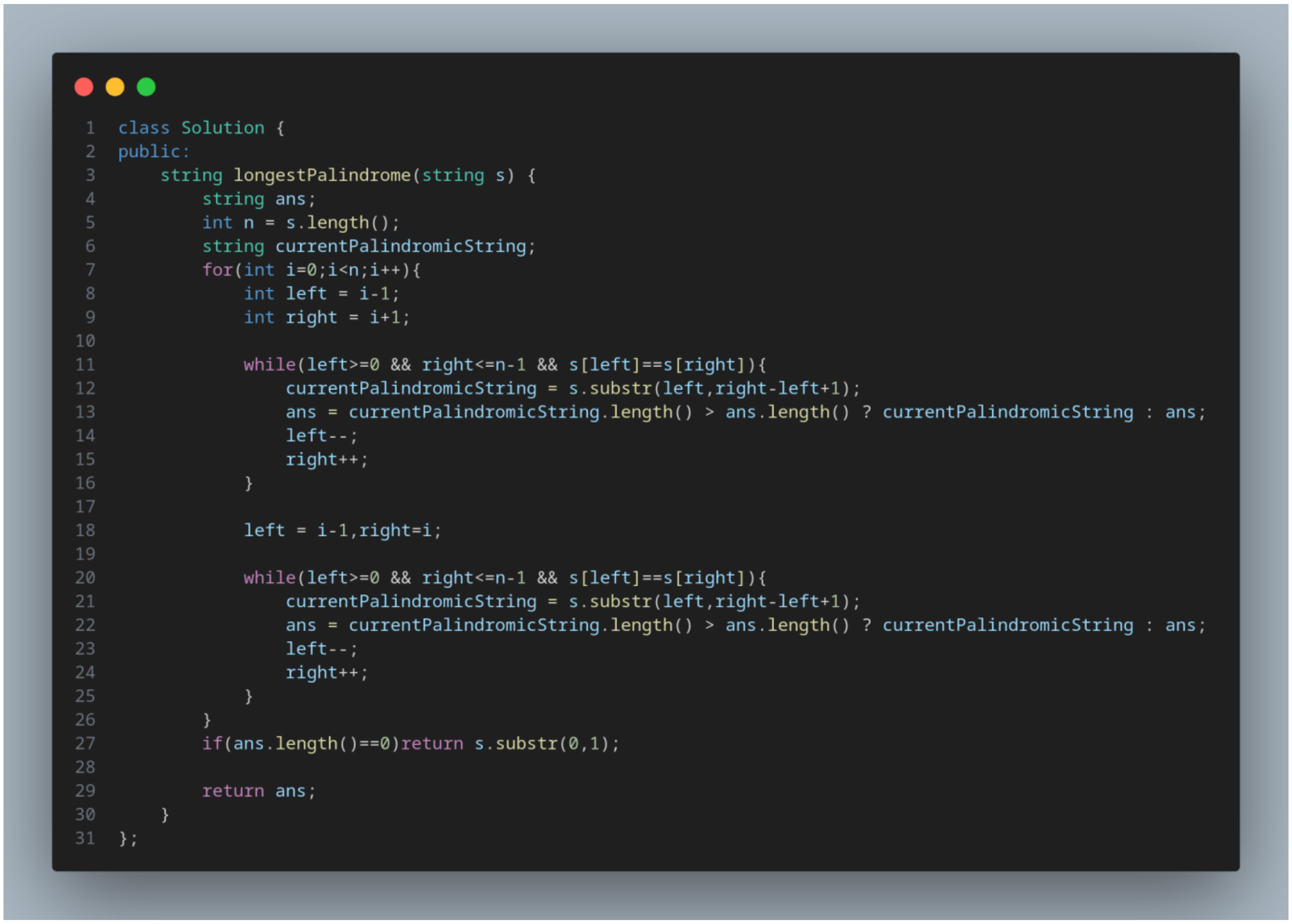
Our solution uses a straightforward approach of iterating through the string and expanding around each character to check for palindromic substrings. We consider both odd and even-length palindromes by checking for palindromes centered at each character.

Time and Space Complexity Analysis:

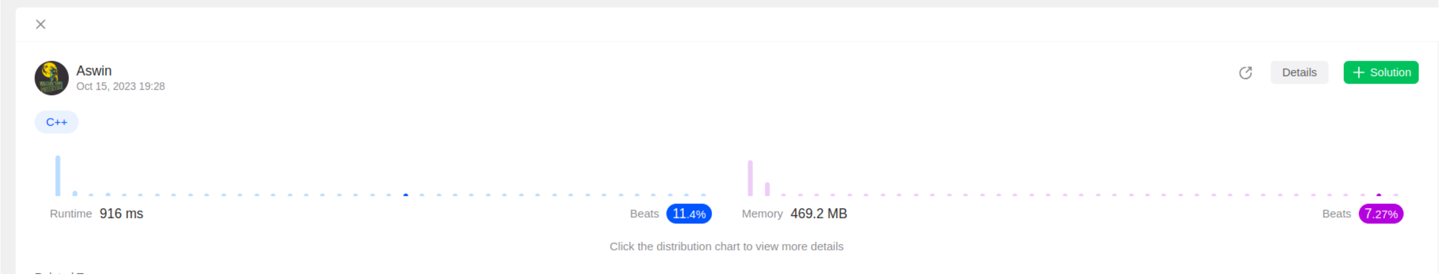
Time Complexity: Our solution has a time complexity of O(N^2), where N is the length of the input string. This is because we iterate through each character, and for each character, we can potentially expand to both the left and right to check for a palindrome.

Space Complexity: Our solution has a space complexity of O (1), as we use a constant amount of extra space to store variables and the answer string.

Code:

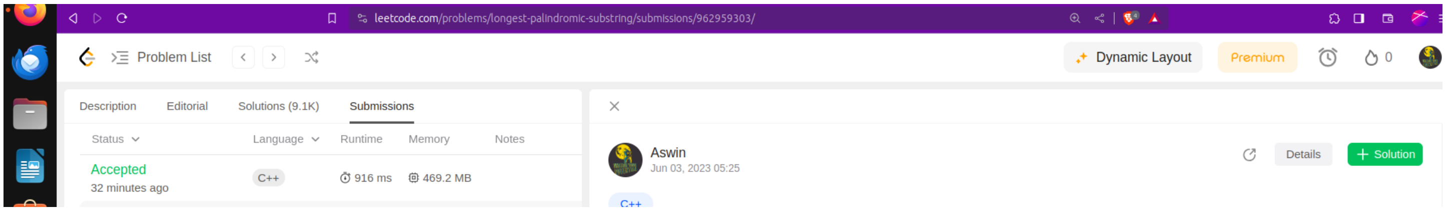


```cpp
class Solution {
public:
    string longestPalindrome(string s) {
        string ans;
        int n = s.length();
        string currentPalindromicString;
        for(int i=0;i<n;i++){
            int left = i-1;
            int right = i+1;

            while(left>=0 && right<=n-1 && s[left]==s[right]){
                currentPalindromicString = s.substr(left,right-left+1);
                ans = currentPalindromicString.length() > ans.length() ? currentPalindromicString : ans;
                left--;
                right++;
            }

            left = i-1,right=i;

            while(left>=0 && right<=n-1 && s[left]==s[right]){
                currentPalindromicString = s.substr(left,right-left+1);
                ans = currentPalindromicString.length() > ans.length() ? currentPalindromicString : ans;
                left--;
                right++;
            }
        }
        if(ans.length()==0)return s.substr(0,1);

        return ans;
    }
};
```



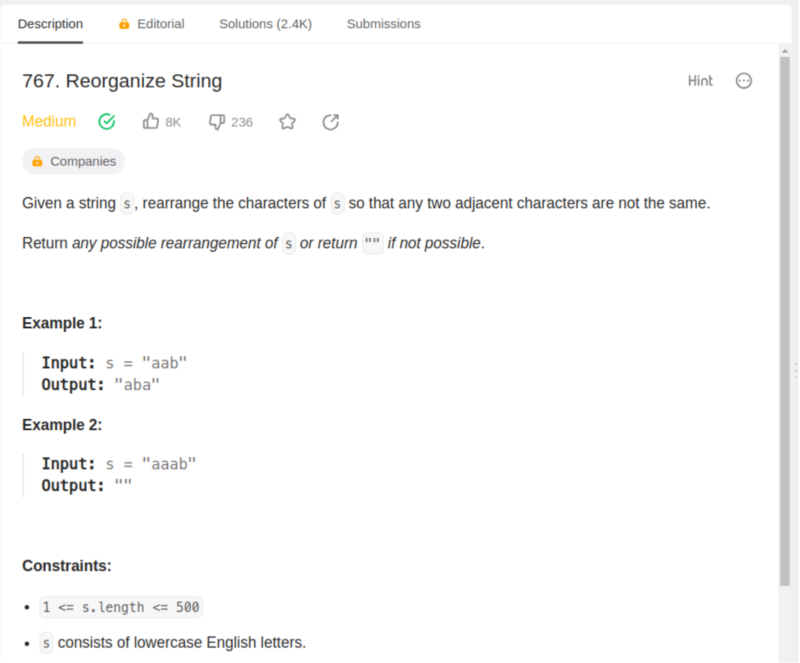
Runtime 916 ms Beats 11.4%

Memory 469.2 MB Beats 7.27%



Status	Language	Runtime	Memory	Notes
Accepted 32 minutes ago	C++	916 ms	469.2 MB	

QUESTION 2: Reorganize String



767. Reorganize String

Given a string `s`, rearrange the characters of `s` so that any two adjacent characters are not the same.

Return *any possible rearrangement of* `s` *or return* `""` *if not possible.*

Example 1:

```
Input: s = "aab"
Output: "aba"
```

Example 2:

```
Input: s = "aaab"
Output: ""
```

Constraints:

- `1 <= s.length <= 500`
- `s` consists of lowercase English letters.

Understanding the Problem:

We are given a string s, and the task is to rearrange the characters of s in such a way that no two adjacent characters are the same. If such a rearrangement is possible, we return any rearrangement; otherwise, return an empty string.

Identifying Edge Cases:

The constraints state that s has a length between 1 and 500 and consists of lowercase English letters. Here are some edge cases to consider:

- A single character string: In this case, it is already a valid rearrangement.
- A string with all the same characters: This is not possible to rearrange.

Effective Test Cases:

1. s = "aab" (Example 1): The output should be "aba."
2. s = "aaab" (Example 2): The output should be an empty string.
3. s = "abc": This is a random arrangement, so the output can be any valid rearrangement.
4. s = "a": The output should be "a" since it is a single character.
5. s = "aa": The output should be an empty string because you cannot rearrange it.

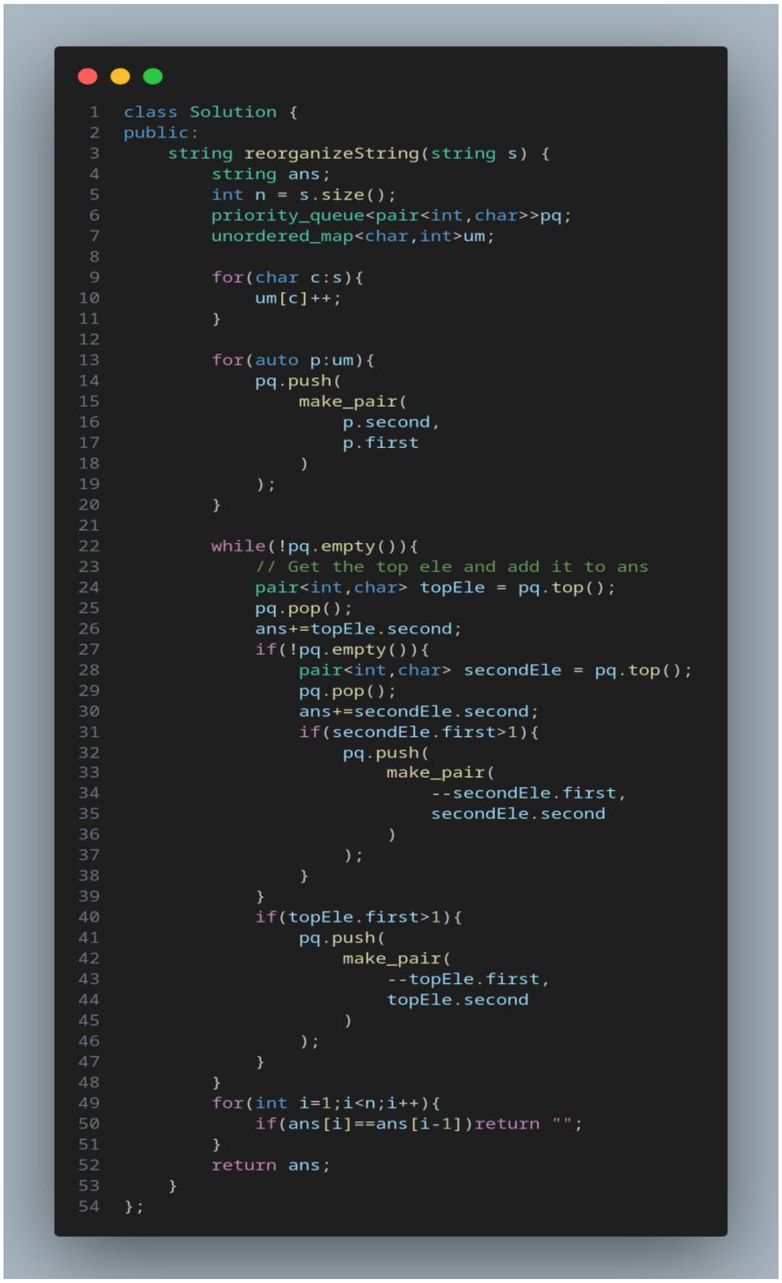
Algorithmic Solution:

Our solution uses a priority queue (max-heap) to store characters along with their frequencies. We iterate through the characters of the input string, count their frequencies, and insert them into the priority queue. Then, we repeatedly extract characters with the highest frequency from the priority queue and append them to the result string, making sure that the same character is not added consecutively. We decrease the frequency of characters as we add them back to the queue. Finally, we check if the rearranged string is valid (no adjacent characters are the same) and return it or an empty string accordingly.

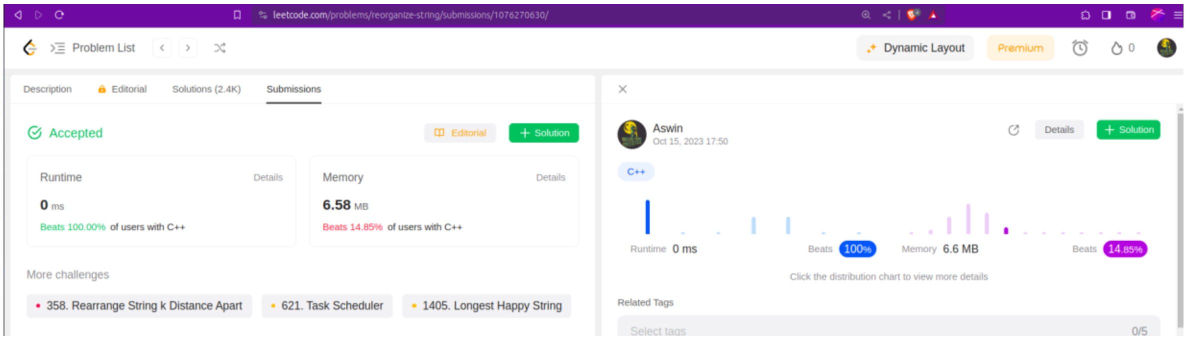
Time and Space Complexity Analysis:

Time Complexity: Our solution has a time complexity of O (N * log(N)), where N is the length of the input string. The main loop iterates through the characters of the string, and in each iteration, we perform operations like pushing and popping from the priority queue (log(N) time).
Space Complexity: Our solution has a space complexity of O(N) because we use a priority queue and an unordered map to store character frequencies.

Code:



```cpp
class Solution {
public:
    string reorganizeString(string s) {
        string ans;
        int n = s.size();
        priority_queue<pair<int,char>>pq;
        unordered_map<char,int>um;

        for(char c:s){
            um[c]++;
        }

        for(auto p:um){
            pq.push(
                make_pair(
                    p.second,
                    p.first
                )
            );
        }

        while(!pq.empty()){
            // Get the top ele and add it to ans
            pair<int,char> topEle = pq.top();
            pq.pop();
            ans+=topEle.second;
            if(!pq.empty()){
                pair<int,char> secondEle = pq.top();
                pq.pop();
                ans+=secondEle.second;
                if(secondEle.first>1){
                    pq.push(
                        make_pair(
                            --secondEle.first,
                            secondEle.second
                        )
                    );
                }
            }
            if(topEle.first>1){
                pq.push(
                    make_pair(
                        --topEle.first,
                        topEle.second
                    )
                );
            }
        }
        for(int i=1;i<n;i++){
            if(ans[i]==ans[i-1])return "";
        }
        return ans;
    }
};
```



Accepted

Runtime Details

0 ms

Beats 100.00% of users with C++

Memory Details

6.58 MB

Beats 14.85% of users with C++

More challenges

- 358. Rearrange String k Distance Apart
- 621. Task Scheduler
- 1405. Longest Happy String

Aswin
Oct 15, 2023 17:50

Runtime 0 ms Beats 100%

Memory 6.6 MB Beats 14.85%