1960. Maximum Product of the Length of Two Palindromic Substrings

Problem Explanation

You are given a string `s` and your task is to find two non-intersecting palindromic substrings of odd length such that the product of their lengths is maximized. Both the substrings should be palindromes and have odd lengths.

A palindrome is a string that is the same forward and backward. A substring is a contiguous sequence of characters in a string.

Example

Let's walk through an example to understand the problem better.

Input: s = "ababbb" Output: 9

In this example, we have two substrings, "aba" and "bbb", which are palindromes with odd lengths. We can calculate the product as follows: `3 * 3 = 9`, and thus, the output is 9.

Approach

The approach we will use for this problem is the Manacher's Algorithm. Here are the steps we will follow:

1. For each position in the string `s`, find the length of the longest palindrome that has the center at that position.
2. Find two non-overlapping longest palindromes.
3. Return the product of the lengths of the two longest non-overlapping palindromes.

Manacher's Algorithm

Manacher's Algorithm is an efficient method to find the longest palindromic substring in linear time complexity. It works by calculating the lengths of all palindromic substrings for each position in the given string, hence finding the longest palindrome substring.

Algorithm Steps

1. Implement the Manacher's Algorithm and store the length of the longest palindrome centered at each position in an array.
2. Find the left to right and right to left longest palindromes using the array derived in step 1.
3. Iterate through the array, finding two non-overlapping longest palindromes.
4. Return the product of the lengths of the two non-overlapping longest palindromes.

Now, let's implement the solution in different languages.

C++ Solution

```
cpp
class Solution {
    public:
        long long maxProduct(string s) {
            const int n = s.length();
            long long ans = 1;
            // l[i] := max length of palindromes in s[0..i)
            vector<int> l = manacher(s, n);
            // r[i] := max length of palindromes in s[i..n)
            vector<int> r = manacher(string(rbegin(s), rend(s)), n);
            reverse(begin(r), end(r));

            for (int i = 0; i + 1 < n; ++i)
                ans = max(ans, (long long)l[i] * r[i + 1]);

            return ans;
        }

    private:
        vector<int> manacher(const string& s, int n) {
            vector<int> maxExtends(n);
            vector<int> l2r(n, 1);
            int center = 0;

            for (int i = 0; i < n; ++i) {
                const int r = center + maxExtends[center] - 1;
                const int mirrorIndex = center - (i - center);
                int extend = i > r ? 1 : min(maxExtends[mirrorIndex], r - i + 1);
                while (i - extend >= 0 && i + extend < n &&
                       s[i - extend] == s[i + extend]) {
                    l2r[i + extend] = 2 * extend + 1;
                    ++extend;
                }
                maxExtends[i] = extend;
                if (i + maxExtends[i] >= r)
                    center = i;
            }

            for (int i = 1; i < n; ++i)
                l2r[i] = max(l2r[i], l2r[i - 1]);

            return l2r;
        }
};
```

We will now implement the solution in other languages.## Python Solution

```
python
class Solution:
    def maxProduct(self, s: str) -> int:
        n = len(s)
        ans = 1
        l = self.manacher(s, n)
        r = self.manacher(s[::-1], n)
        r.reverse()

        for i in range(n - 1):
            ans = max(ans, l[i] * r[i + 1])

        return ans

    def manacher(self, s: str, n: int) -> list:
        maxExtends = [0] * n
        l2r = [1] * n
        center = 0

        for i in range(n):
            r = center + maxExtends[center] - 1
            mirrorIndex = center - (i - center)
            extend = 1 if i > r else min(maxExtends[mirrorIndex], r - i + 1)
            while i - extend >= 0 and i + extend < n and s[i - extend] == s[i + extend]:
                l2r[i + extend] = 2 * extend + 1
                extend += 1
            maxExtends[i] = extend
            if i + maxExtends[i] >= r:
                center = i

        for i in range(1, n):
            l2r[i] = max(l2r[i], l2r[i - 1])

        return l2r
```

JavaScript Solution

```
javascript
class Solution {
    maxProduct(s) {
        const n = s.length;
        let ans = 1;
        const l = this.manacher(s, n);
        const r = this.manacher(s.split('').reverse().join(''), n);
        r.reverse();

        for (let i = 0; i + 1 < n; ++i)
            ans = Math.max(ans, l[i] * r[i + 1]);

        return ans;
    }

    manacher(s, n) {
        const maxExtends = new Array(n).fill(0);
        const l2r = new Array(n).fill(1);
        let center = 0;

        for (let i = 0; i < n; ++i) {
            const r = center + maxExtends[center] - 1;
            const mirrorIndex = center - (i - center);
            let extend = i > r ? 1 : Math.min(maxExtends[mirrorIndex], r - i + 1);
            while (i - extend >= 0 && i + extend < n &&
                   s[i - extend] === s[i + extend]) {
                l2r[i + extend] = 2 * extend + 1;
                ++extend;
            }
            maxExtends[i] = extend;
            if (i + maxExtends[i] >= r)
                center = i;
        }

        for (let i = 1; i < n; ++i)
            l2r[i] = Math.max(l2r[i], l2r[i - 1]);

        return l2r;
    }
}
```

Java Solution

```
java
class Solution {
    public int maxProduct(String s) {
        int n = s.length();
        int ans = 1;
        int[] l = manacher(s, n);
        int[] r = manacher(new StringBuilder(s).reverse().toString(), n);
        int[] reverseR = new int[n];
        for (int i = 0; i < n; ++i) {
            reverseR[i] = r[n - i - 1];
        }

        for (int i = 0; i + 1 < n; ++i) {
            ans = Math.max(ans, l[i] * reverseR[i + 1]);
        }

        return ans;
    }

    private int[] manacher(String s, int n) {
        int[] maxExtends = new int[n];
        int[] l2r = new int[n];
        int center = 0;

        for (int i = 0; i < n; ++i) {
            int r = center + maxExtends[center] - 1;
            int mirrorIndex = center - (i - center);
            int extend = i > r ? 1 : Math.min(maxExtends[mirrorIndex], r - i + 1);
            while (i - extend >= 0 && i + extend < n &&
                   s.charAt(i - extend) == s.charAt(i + extend)) {
                l2r[i + extend] = 2 * extend + 1;
                ++extend;
            }
            maxExtends[i] = extend;
            if (i + maxExtends[i] >= r) {
                center = i;
            }
        }

        for (int i = 1; i < n; ++i) {
            l2r[i] = Math.max(l2r[i], l2r[i - 1]);
        }

        return l2r;
    }
}
```