Unique Length-3 Palindromic Subsequences

A. Madhur

Introduction

In what follows, several methods are presented. The intention is to demonstrate the versatility of `Rust`'s syntax and, in doing so, provide a concise refresher.

Statement: Given a string `s`, return the number of unique palindromes of length three that are subsequences of `s`.

Note that even if there are multiple ways to obtain the same subsequence, it is counted only once.

A palindrome is a string that reads the same forwards and backwards.

A subsequence of a string is a new string generated from the original string by deleting zero or more characters without changing the relative order of the remaining characters.

For example, `"ace"` is a subsequence of `"abcde"`.

`https://leetcode.com/problems/unique-length-3-palindromic-subsequences/description`

Naive Solution [AC]

The idea is simple, as the problem significantly constrains the solution space by limiting candidates to palindromes of length three. Consequently, the relevant patterns reduce to subsequences of the form "`a_a`".

Interestingly, `String` in `Rust` is UTF-8 encoded and stored contiguously in memory (similar to `Java`). It also exposes a comparable mechanism for traversal via `.chars()`, analogous to `Java`'s `toCharArray()`. The underlying design philosophy shares certain characteristics with that of `Erlang`.

N.B. Every code block has been formatted using `rustfmt`.

```rust
impl Solution {
    pub fn count_palindromic_subsequence(s: String) -> i32 {
        let n = s.len();
        let sc = s.chars().collect::<Vec<char>>();
        let mut res = 0;

        for left in 0..n {
            if sc[0..left].contains(&sc[left]) {
                continue;
            }

            let mut right = n - 1;
            while right > left && sc[right] != sc[left] {
                right -= 1;
            }

            let mut seen = vec![0; 26];
            for i in (left + 1)..right {
                let j = (sc[i] as u8 - b'a') as usize;
                if seen[j] == 0 {
                    res += 1;
                }
                seen[j] += 1;
            }
        }

        return res;
    }
}
```

In the worst case, this approach runs in $O(n^2)$ time. While the logic is correct, the implementation is not optimal. A more structured version of the same idea, shown below, uses precomputed indices along with a `HashSet` to improve clarity. Its worst-case time complexity is still $O(n^2)$, though membership checks benefit from the expected $O(1)$ performance of hashing.

```
impl Solution {
    pub fn count_palindromic_subsequence(s: String) -> i32 {
        let sb = s.as_bytes();

        let mut left = vec![-1; 26];
        let mut right = vec![-1; 26];

        for (i, &ch) in sb.iter().enumerate() {
            let j = (ch - b'a') as usize;
            if left[j] == -1 {
                left[j] = i as i32;
            }
            right[j] = i as i32;
        }

        let mut res = std::collections::HashSet::<(u8, u8)>::new();

        for i in 0..26 {
            if left[i] != -1 && left[i] != right[i] {
                for mid in (left[i] + 1) as usize..right[i] as usize {
                    res.insert((i as u8 + b'a', sb[mid]));
                }
            }
        }

        return res.len() as i32;
    }
}
```

A more refined version precomputes frequencies from the right.

```
impl Solution {
    pub fn count_palindromic_subsequence(s: String) -> i32 {
        let n = s.len(); // safe for lowercase ASCII; otherwise use s.as_bytes().
    len()
        let sb = s.as_bytes();

        let mut right_freq = vec![0; 26];
        for &b in sb {
            right_freq[(b - b'a') as usize] += 1;
        }

        let mut seen = vec![false; 26];
        let mut res = std::collections::HashSet::<(u8, u8)>::new();

        for j in 0..n {
            let left = (sb[j] - b'a') as usize;
            right_freq[left] -= 1;

            for c in 0..26 {
                if seen[c] && right_freq[c] > 0 {
                    res.insert((c as u8, left as u8));
                }
            }

            seen[left] = true;
        }

        return res.len() as i32;
    }
}
```

Using Bitmask [AC]

Performance can be improved by using a `bitmask` instead of a `HashSet`, thereby avoiding hashing overhead and heap allocations.

```
impl Solution {
    pub fn count_palindromic_subsequence(s: String) -> i32 {
        let n = s.len();
        let sb = s.as_bytes();

        let mut left = vec![n; 26];
        let mut right = vec![0; 26];

        for i in 0..n {
            let j = (sb[i] - b'a') as usize;
            left[j] = left[j].min(i);
            right[j] = i;
        }

        let mut res = 0;

        for c in 0..26 {
            if left[c] >= right[c] {
                continue;
            }
            let mut mask: u32 = 0;

            for i in (left[c] + 1)..right[c] {
                let mid = (sb[i] - b'a') as u32;
                mask |= 1 << mid;
            }

            res += mask.count_ones() as i32;
        }

        return res;
    }
}
```

Using bitset-style `struct` [AC]

This version uses a small bitset-style `struct` instead of a raw `u32` mask. This improves semantic clarity and more closely resembles `std::bitset` in `C++`.

```rust
#[derive(Clone, Copy)]
struct BitSet26 {
    bits: u32,
}

impl BitSet26 {
    fn new() -> Self {
        Self { bits: 0 }
    }

    fn set(&mut self, idx: usize) {
        self.bits |= 1 << idx;
    }

    fn count(&self) -> u32 {
        self.bits.count_ones()
    }
}

impl Solution {
    pub fn count_palindromic_subsequence(s: String) -> i32 {
        let n = s.as_bytes().len();
        let bytes = s.as_bytes();

        let mut first = vec![n; 26];
        let mut last = vec![0; 26];

        for i in 0..n {
            let j = (bytes[i] - b'a') as usize;
            first[j] = first[j].min(i);
            last[j] = i;
        }

        let mut res = 0;

        for c in 0..26 {
            if first[c] >= last[c] {
                continue;
            }

            let mut seen = BitSet26::new();

            for i in (first[c] + 1)..last[c] {
                let mid = (bytes[i] - b'a') as usize;
                seen.set(mid);
            }

            res += seen.count() as i32;
        }

        return res;
    }
}
```

Functional Approach [AC]

This version is intentionally written in a functional style and visually resembles solutions commonly written in functional languages such as `Erlang`.

```rust
impl Solution {
    pub fn count_palindromic_subsequence(s: String) -> i32 {
        let bs = s.as_bytes();

        let (first, last) = s.iter().enumerate().fold(
            (vec![usize::MAX; 26], vec![0usize; 26]),
            |(mut left, mut right), (i, &b)| {
                let j = (b - b'a') as usize;
                left[j] = left[j].min(i);
                right[j] = i;
                (left, right)
            },
        );

        (0..26)
            .filter_map(|ch| {
                let left = first[ch];
                let right = last[ch];

                if left < right {
                    Some(
                        (left + 1..right)
                            .map(|i| (bs[i] - b'a') as usize)
                            .collect::<std::collections::HashSet<usize>>()
                            .len(),
                    )
                } else {
                    None
                }
            })
            .sum::<usize>() as i32
    }
}
```