214. Shortest Palindrome

Hard String String Matching Hash Function Rolling Hash Leetcode Link

Problem Description

In this problem, we are given a string `s`. Our task is to make the smallest palindrome by adding characters at the front of `s`. A palindrome is a string that reads the same forward and backward. The key here is that we can only add characters to the beginning of `s` to form the palindrome; we cannot modify or add characters at the end. We need to find and return the shortest possible palindrome that can be obtained from `s` through such transformations.

Intuition

To solve this problem effectively, we need to identify the longest palindrome that starts at the beginning of `s`. Once we find such a palindrome, we can mirror the remaining part of the string (that isn't included in the palindrome) and add it to the front to create the shortest palindrome string.

The crucial part of the solution is to find the longest palindromic prefix efficiently. We can achieve this by comparing prefixes and suffixes of the string while respecting the palindrome property.

The algorithm uses a hash-based approach to compare the palindromic prefix and suffix, which can be efficiently computed using a rolling hash technique. Here's how it goes:

1. Initialize two hash variables `prefix` and `suffix` to 0. These will be used to calculate the hash of the prefix (from the start of the string) and the suffix (from the end of the string) respectively.
2. Use a `base` for the hash function and a `mod` to prevent overflow issues with large hash values. Go over each character in the string and update the hash values for both the prefix and suffix.
3. If at any point the hash of the prefix and the hash of the suffix are equal, it indicates that we have a palindrome from the beginning of the string to the current index `i`.
4. Remember the furthest index `idx` where the prefix and suffix hashes match (indicating the longest palindromic prefix).

After we identify the longest palindromic prefix, the remaining substring (from `idx` to the end), when reversed and added to the front of the original string `s`, will result in the shortest palindrome.

So, the result will be the reverse of the substring from `idx` to the end concatenated to the original string. If the entire string `s` is a palindrome, we just return `s` as it is already the shortest palindrome we can achieve.

Solution Approach

For the implementation of the solution, we leverage a rolling hash polynomial hashing algorithm. The rolling hash technique is an efficient way to compute and compare hash values for substrings quickly. This is particularly useful in our case, where we need to compare prefixes and suffixes of the given string.

Here's a step-by-step explanation of the implementation:

1. We choose a base number, `base`, for the rolling hash functions and a large prime number, `mod`, to take the modulus and prevent integer overflow.
2. We initialize `prefix` and `suffix` hash values to `0`, `mul` to `1`, and `idx` to `0`. The `idx` will hold the position of the last character of the longest palindrome prefix.
3. We iterate through the string `s` using a for loop, and on each iteration, we do the following:
   - Update the `prefix` hash by multiplying the current `prefix` hash by `base` and then adding the value of the current character shifted by `1` to avoid `0` in the calculation (`ord(c) - ord('a') + 1`). We wrap around with modulus `mod` to manage large numbers.
   - Update the `suffix` hash by adding to it the value of the current character multiplied by `mul` (which represents `base` raised to the power of the character's position). Again, we take the result modulo `mod`.
   - Update `mul` by multiplying it by `base` and taking modulus `mod`. This step effectively computes `base` to the power of `i` modulo `mod` for the `i`-th character.
   - If at any point the `prefix` and `suffix` hashes are equal, it means we have a palindrome starting from the beginning of the string to the current character. We update `idx` to `i + 1`.
4. After the loop completes, if `idx` is equal to the length of the string, the entire string is a palindrome. We return `s` in that case.
5. If the whole string isn't a palindrome, we need to create a palindrome by adding characters to the beginning of the string. We do this by taking the substring from `idx` to the end of `s`, reversing it, and concatenating it with the original string `s`.

The final result is generated by the expression `s[idx:][::-1] + s`, where `s[idx:][::-1]` creates the needed prefix by reversing the part of the string that is not included in the palindrome and appending it to the front of `s` to form the shortest palindrome.

This approach is efficient as it has a linear time complexity with respect to the length of the string, and it leverages hashing to quickly identify the longest palindrome prefix.

Example Walkthrough

Let's walk through the solution approach with a small example. Suppose our given string is `s = "abacd"`. We want to find the shortest palindrome by adding characters at the beginning of `s`.

1. We choose `base` as a small prime number, for simplicity, let's use `3`, and let `mod` be a large prime number, `mod = 10007` to avoid integer overflow.
2. Initialize `prefix` and `suffix` hashes to `0`, `mul` to `1`, and `idx` to `0`.
3. We'll iterate through the string and compute hashes for the prefix starting from the beginning and the suffix from the end simultaneously:

i	Character (c)	Prefix Hash (new)	Suffix Hash (new)	mul (new)	idx
0	'a'	(0*3 + 1) mod 10007	(0 + 1*1) mod 10007	3	1
1	'b'	(1*3 + 2) mod 10007	(1 + 2*3) mod 10007	3*3	1
2	'a'	(5*3 + 1) mod 10007	(7 + 1*9) mod 10007	3*3*3	3
3	'c'	(16*3 + 3) mod 10007	(16 + 3*27) mod 10007	3*3*3*3	3
4	'd'	(51*3 + 4) mod 10007	(97 + 4*81) mod 10007	3*3*3*3*3	3

   At each step, we are updating the `prefix` hash by multiplying it by `base` and adding the current character's value, and updating the `suffix` hash by adding the current character's value multiplied by `mul`. `mul` is updated by multiplying it by `base` at each step.

   At index `i = 2`, the prefix and suffix hashes match (`16 mod 10007`), meaning that we have a palindrome from the start of the string to the character at index `2` (`"aba"`). So we update `idx` to `3`.

4. After completing the iteration, since `idx` is not equal to the length of the string, we conclude that the entire string is not a palindrome.
5. To form the shortest palindrome, we take the substring from `idx` to the end (`"acd"`) and reverse it to get `"dca"`. We then concatenate this reversed substring to the front of the original string `s`.

The resulting palindrome is `"dca" + "abacd"` = `"dcaabacd"`, which is the shortest palindrome that we can form by adding characters only at the beginning of the string `s`. This example illustrates the efficiency of the solution, which finds the longest palindromic prefix and adds the minimum required characters to the front to form the palindrome.

Python Solution

```python
class Solution:
    def shortest_palindrome(self, s: str) -> str:
        base = 131  # Base for polynomial rolling hash.
        mod = 10**9 + 7  # Modulus for hash to avoid overflow.
        n = len(s)  # Length of the input string.
        prefix_hash = 0  # Hash value of the prefix.
        suffix_hash = 0  # Hash value of the suffix.
        multiplicator = 1  # Multiplicator value used for hash computation.
        longest_palindrome_idx = 0  # End index of the longest palindromic prefix.

        # Compute rolling hash from both ends.
        for i, ch in enumerate(s):
            # Update the prefix hash by appending character.
            prefix_hash = (prefix_hash * base + (ord(ch) - ord('a') + 1)) % mod

            # Update the suffix hash by adding character (considered at the beginning).
            suffix_hash = (suffix_hash + (ord(ch) - ord('a') + 1) * multiplicator) % mod

            # Update the multiplicator for the next character.
            multiplicator = (multiplicator * base) % mod

            # If the prefix and suffix hashes match, update the longest prefix palindrome index.
            if prefix_hash == suffix_hash:
                longest_palindrome_idx = i + 1

        # If the entire string is a palindrome, return it.
        if longest_palindrome_idx == n:
            return s

        # Otherwise, append the reverse of the remaining suffix to the front to make the shortest palindrome.
        return s[longest_palindrome_idx:][::-1] + s
```

Java Solution

```java
public class Solution {

    // This method finds the shortest palindrome starting from the first character by appending characters to the front
    public String shortestPalindrome(String s) {
        // we use a prime number as a base for computing rolling hash
        final int base = 131;
        // modular multiplication factor, initially 1
        int multiplier = 1;
        // we will use a large prime number to mod the result to avoid overflow
        final int mod = (int) 1e9 + 7;
        // rolling hash from the front
        int prefixHash = 0;
        // rolling hash from the back
        int suffixHash = 0;
        // the index till the string is a palindrome
        int palindromeIdx = 0;
        // length of the string
        int length = s.length();

        // iterate through the string to update the prefix and suffix hashes
        for (int i = 0; i < length; ++i) {
            // convert character to number (assuming lowercase 'a' to 'z')
            int charValue = s.charAt(i) - 'a' + 1;
            // update the prefix hash and ensure it is within the bounds by taking modulo
            prefixHash = (int) (((long) prefixHash * base + charValue) % mod);
            // update the suffix hash and ensure it is within the bounds by taking modulo
            suffixHash = (int) ((suffixHash + (long) charValue * multiplier) % mod);
            // update the multiplier for the next character
            multiplier = (int) (((long) multiplier * base) % mod);

            // if the prefix and suffix are equal, then we know the string up to index i is a palindrome
            if (prefixHash == suffixHash) {
                palindromeIdx = i + 1;
            }
        }

        // If the whole string is a palindrome, return it as is
        if (palindromeIdx == length) {
            return s;
        }

        // We need to add the reverse of the substring from palindromeIdx to the end to the front
        // to make the string a palindrome
        String suffixToBeAdded = new StringBuilder(s.substring(palindromeIdx)).reverse().toString();

        // Return the string with the suffix added in front to form the shortest palindrome
        return suffixToBeAdded + s;
    }
}
```

C++ Solution

```cpp
typedef unsigned long long ull;

class Solution {
public:
    string shortestPalindrome(string s) {
        // Define constants and initial values
        const int kBase = 131; // Base for polynomial hashing
        ull prefixHash = 0;    // Hash value for the prefix
        ull suffixHash = 0;    // Hash value for the suffix
        ull currentMultiplier = 1; // Used to compute hash values
        int palindromeEndIndex = 0; // Index marking the end of the longest palindrome starting at position 0
        int n = s.size(); // Size of the input string

        // Loop through the string character by character
        for (int i = 0; i < n; ++i) {
            int charValue = s[i] - 'a' + 1; // Convert char to int (1-based for 'a' to 'z')
            prefixHash = prefixHash * kBase + charValue; // Update prefix hash polynomially
            suffixHash = suffixHash + currentMultiplier * charValue; // Update suffix hash
            currentMultiplier *= kBase; // Update the base multiplier for the next character

            // If the current prefix is a palindrome (checked by comparing its hash with the suffix hash)
            if (prefixHash == suffixHash) {
                palindromeEndIndex = i + 1; // Update the end index of the longest palindrome found
            }
        }

        // If the whole string is a palindrome, return it as is
        if (palindromeEndIndex == n) return s;

        // Otherwise, construct the shortest palindrome by appending the reverse of the remaining substring
        string remainingSubstring = s.substr(palindromeEndIndex, n - palindromeEndIndex);
        reverse(remainingSubstring.begin(), remainingSubstring.end());
        return remainingSubstring + s; // Concatenate the reversed substring with the original string
    }
};
```

Typescript Solution

```typescript
// Define a type for unsigned long long equivalent in TypeScript
type ULL = bigint;

// Converts a lowercase character to an integer (1-based)
const charToInt = (char: string): number => char.charCodeAt(0) - 'a'.charCodeAt(0) + 1;

// Reverses a string in place
const reverseString = (s: string): string => s.split('').reverse().join('');

// Computes the shortest palindrome that can be formed by adding characters in front of the given string
const shortestPalindrome = (s: string): string => {
    // Constants and initial values
    const base: ULL = BigInt(131); // Base for polynomial hashing
    let prefixHash: ULL = BigInt(0); // Hash value for the prefix
    let suffixHash: ULL = BigInt(0); // Hash value for the suffix
    let currentMultiplier: ULL = BigInt(1); // Used to compute hash values
    let palindromeEndIndex = 0; // Index marking the end of the longest palindrome at start
    const n = s.length; // Size of the input string

    // Loop through the string character by character
    for (let i = 0; i < n; ++i) {
        const charValue = charToInt(s[i]); // Convert char to int
        // Update prefix hash polynomially and suffix hash
        prefixHash = prefixHash * base + BigInt(charValue);
        suffixHash = suffixHash + currentMultiplier * BigInt(charValue);
        // Update the base multiplier for hash computation
        currentMultiplier *= base;

        // If the current prefix is a palindrome (checked by comparing hashes)
        if (prefixHash === suffixHash) {
            palindromeEndIndex = i + 1; // Update the end index of the longest palindrome found
        }
    }

    // If the whole string is a palindrome, return it
    if (palindromeEndIndex === n) return s;

    // Construct the shortest palindrome by appending the reversed suffix to the original string
    const remainingSubstring = s.substring(palindromeEndIndex);
    return reverseString(remainingSubstring) + s;
};

// Example usage
// const result = shortestPalindrome("example");
// console.log(result);
```

Time and Space Complexity

The given Python code implements an algorithm for finding the shortest palindrome by appending characters to the beginning of the given string `s`. The algorithm is based on calculating hash values from both ends (prefix and suffix) and checking for palindromes.

Time Complexity

The time complexity of this code primarily comes from a single for loop that iterates through each character in the string once. Inside the loop, it computes the prefix and suffix hash values, and compares them to check if they are equal.

Here's the breakdown:

- The hash operations and comparison inside the for loop are O(1) operations as they are done using arithmetic calculations.
- The for loop runs n times, where n is the length of the string `s`.

Therefore, the time complexity of this code is `O(n)`, where `n` is the length of the input string.

Space Complexity

The space complexity of the code is determined by the storage used which is independent of the length of the input string `s`.

Here's the breakdown:

- Variables `prefix`, `suffix`, `mul`, and `idx` are integers which occupy constant space.
- The slice and reverse operation `s[idx:][::-1]` creates a new string of at most `n-1` characters when the input string is not already a palindrome.

Even though a new string is created in the worst-case scenario, the space complexity is proportional to the input string size which gives us `O(n)`. However, if we consider only the additional space excluding the input and output, the space complexity is actually `O(1)` since we're only using a fixed amount of additional storage regardless of the input size.