1392. Longest Happy Prefix

Hard | String | String Matching | Hash Function | Rolling Hash

Leetcode Link

Problem Description

The problem presents a string processing task where we need to find the longest possible substring that serves two roles simultaneously: it is a prefix and a suffix of the given string `s`. A prefix is a start part of the string, and a suffix is an ending part of the string. It's important to note however that the 'happy prefix' we are looking for cannot be the entire string itself, it should be strictly non-empty and shorter than the full string.

Intuition

The intuition behind the solution is fairly straightforward. We want to find the longest sequence at the beginning of the string (`s`) that is repeated at the end. The naive approach would be to consider each possible prefix of the string and check if it's also a suffix.

To implement this, we can start checking the longest possible prefix/suffix first and narrow down to the shorter ones. This is done by slicing the string: for a prefix starting from the beginning of the string `s` and ending at `s[:-i]`, check if there's an equivalent suffix starting at `s[i:]` and going to the end of the string. If such a match is found, then `s[i:]` is our longest 'happy prefix'. If no match is found all the way up to the shortest prefix (a single character), return an empty string.

The solution starts the loop from 1 (as opposed to starting from 0) because we want a non-empty prefix/suffix. Additionally, we loop until `len(s)` because `s[:-len(s)]` would give us an empty string which is not a valid 'happy prefix', and we're looking for a prefix/suffix pair that does not include the entire string itself.

Solution Approach

The provided solution uses a straightforward brute-force approach. There's no need for complex algorithms or additional data structures. The approach is based on string comparison and slicing only.

Here's a step-by-step explanation of the implemented solution:

1. The solution defines a method `longestPrefix` within the `Solution` class that takes a string `s` as its argument.
2. The method initiates a loop that starts at 1 and ends just before the length of the string. The loop iterates with `i` indicating the current size of the suffix and prefix that are being compared. By using `s[:-i]` and `s[i:]`, we effectively slice the string to create both the prefix and the suffix to compare.
3. At each iteration, the algorithm checks whether the current prefix (`s[:-i]`) is equal to the current suffix (`s[i:]`). If these two substrings are the same, it means we have found a 'happy prefix'.
4. The loop starts checking from the longest possible happy prefix and goes down to the shortest. The reasoning behind this order is that we are interested in the longest happy prefix. If a match is found, it will be the longest one due to the order of the loop, and the method can return immediately without checking the rest.
5. If no 'happy prefix' is found throughout the loop, the method returns an empty string `''`, indicating that no such prefix exists for the given string.

To visualize, let's consider `s = "abab"` as an example:

- In the first iteration, `i = 1`, the prefix `s[:-1]` is "aba" and the suffix `s[1:]` is "bab". They do not match.
- In the second iteration, `i = 2`, the prefix `s[:-2]` is "ab" and the suffix `s[2:]` is "ab". They match, so "ab" is returned as the longest happy prefix.

This solution approach has a time complexity of O(n^2), where n is the length of the string, due to the fact that string comparison is done in each iteration over slices of the string that decrease in size by one character each time.

Example Walkthrough

Let's walk through a small example using the string `s = "level"` to illustrate the solution approach. We are looking for the longest substring that is both a prefix and a suffix of `s` but is not `s` itself.

1. **Iteration 1:** `i = 1`
   - Prefix: `s[:-1]` is "leve"
   - Suffix: `s[1:]` is "evel"
   - Comparison: "leve" is not equal to "evel", so we continue to the next iteration.
2. **Iteration 2:** `i = 2`
   - Prefix: `s[:-2]` is "lev"
   - Suffix: `s[2:]` is "vel"
   - Comparison: "lev" is not equal to "vel", so we continue to the next iteration.
3. **Iteration 3:** `i = 3`
   - Prefix: `s[:-3]` is "le"
   - Suffix: `s[3:]` is "el"
   - Comparison: "le" is not equal to "el", so we continue to the next iteration.
4. **Iteration 4:** `i = 4`
   - Prefix: `s[:-4]` is "l"
   - Suffix: `s[4:]` is "l"
   - Comparison: "l" is equal to "l"
   - Since we have found that the prefix is equal to the suffix, "l" is the longest "happy prefix".

In this case, since "l" is the longest substring that meets the criteria, the method returns "l". If no matching substrings had been found, the method would return an empty string `''`.

The solution is efficient because it uses a loop to progressively shorten the prefixes and suffixes from the longest to the shortest, stopping early when a match is found, which is more optimal compared to checking in the opposite direction. Additionally, by not using additional data structures, the approach keeps memory usage minimal.

Python Solution

```python
class Solution:
    def longest_prefix(self, s: str) -> str:
        # Initialize the maximum length of the longest prefix that is also a suffix
        max_prefix_length = 0

        # Start checking for prefixes and suffixes from the shortest length towards the longest
        for i in range(1, len(s)):
            # Check if the prefix is equal to the suffix
            # We slice the string s from the beginning to the end minus i (prefix)
            # And compare it to the substring from i to the end (suffix)
            if s[:-i] == s[i:]:
                # If they are equal, we update the maximum length of the prefix
                max_prefix_length = len(s) - i

        # Return the longest prefix that is also a suffix
        # We slice the string s to the length of the maximum prefix found
        return s[:max_prefix_length]

# Example of usage:
# solution = Solution()
# print(solution.longest_prefix("level"))  # Output: "l"

```

Java Solution

```java
class Solution {
    // Arrays to store the precomputed hash values and powers of the base
    private long[] power;
    private long[] hash;

    /**
     * Function to find the longest prefix which is also a suffix.
     *
     * @param s the input string
     * @return the longest prefix which is also a suffix without overlapping
     */
    public String longestPrefix(String s) {
        int base = 131; // A prime number used as the base for hashing
        int n = s.length();
        power = new long[n + 10]; // Expanded size to prevent array index out of bounds
        hash = new long[n + 10];
        power[0] = 1; // Initializing the first element of power to 1

        // Precompute the powers and hash values for the input string
        for (int i = 0; i < n; ++i) {
            power[i + 1] = power[i] * base;
            hash[i + 1] = hash[i] * base + s.charAt(i);
        }

        // Iterate from longest possible prefix to the shortest
        for (int length = n - 1; length > 0; --length) {
            // If the prefix hash and suffix hash matches, return the prefix
            if (getHash(1, length) == getHash(n - length + 1, n)) {
                return s.substring(0, length);
            }
        }

        // No prefix matches the suffix, return an empty string
        return "";
    }

    /**
     * Helper function to get the hash of a substring.
     *
     * @param left  left index of the substring (inclusive, 1-indexed)
     * @param right right index of the substring (inclusive, 1-indexed)
     * @return the hash value of the substring
     */
    private long getHash(int left, int right) {
        // Compute the substring's hash by subtracting the hash before the substring
        // and adjusting with the power to avoid hash collision
        return hash[right] - hash[left - 1] * power[right - left + 1];
    }
}

```

C++ Solution

```cpp
#include <string>
using std::string;

typedef unsigned long long ull;

class Solution {
public:
    // Calculate the longest prefix which is also a suffix
    string longestPrefix(string s) {
        int base = 131; // The base for the polynomial hash
        int n = s.size(); // Length of the string
        ull power[n + 10]; // Stores the powers of base, power[i] = base^i
        ull hash[n + 10]; // Stores the hash values for the prefix ending at each index

        // Initializing the power and hash arrays
        power[0] = 1;
        hash[0] = 0;

        // Populate power and hash arrays
        for (int i = 0; i < n; ++i) {
            power[i + 1] = power[i] * base;
            hash[i + 1] = hash[i] * base + s[i];
        }

        // Iterate from the end and check if a prefix is also a suffix
        for (int length = n - 1; length > 0; --length) {
            ull prefixHash = hash[length]; // Hash value of the current prefix
            ull suffixHash = hash[n] - hash[n - length] * power[length]; // Hash value of the current suffix

            // If the computed prefix and suffix hash values are equal, we've found the longest prefix-suffix
            if (prefixHash == suffixHash) {
                return s.substr(0, length);
            }
        }

        // If no proper prefix-suffix pair is found, return an empty string
        return "";
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the longest prefix which is also a suffix of the string.
 * The prefix must be non-empty and less than the string length.
 *
 * @param {string} str - The string to analyze.
 * @returns {string} The longest prefix which is also a suffix.
 */
function longestPrefix(str: string): string {
    // Get the length of the string
    const lengthOfString: number = str.length;

    // Iterate backwards from the end of the string
    for (let i = lengthOfString - 1; i >= 0; i--) {
        // Compare substring from start to the current index (prefix)
        // with substring from the end of the string with the same length (suffix)
        if (str.slice(0, i) === str.slice(lengthOfString - i)) {
            // If they are equal, return the prefix as the longest prefix which is also a suffix
            return str.slice(0, i);
        }
    }

    // No prefix which is also suffix was found, return empty string
    return '';
}

```

Time and Space Complexity

Time Complexity

The provided code snippet calculates the longest prefix of a given string `s` that is also a suffix by checking each substring. The outer for loop runs `(len(s) - 1)` times. For each iteration of the loop, the slicing operation `[i:]` and `[:-i]` has a time complexity of `O(n)`, where `n` is the length of the string. Since the slicing happens twice for each iteration and the comparison also takes `O(n)` for each iteration, the total time complexity is:

`O((n - 1) * 2n)` = `O(2n^2 - 2n)` which simplifies to `O(n^2)`.

Therefore, the time complexity of the code is `O(n^2)`.

Space Complexity

Since the algorithm only uses a constant amount of extra space for variables like `i` and the space needed for storing the prefixes/suffixes which are being compared is also not additional as they are references to existing substrings in `s`, the space complexity is `O(1)`. There is no additional space being used that grows with the input size.