1316. Distinct Echo Substrings

Hard | Trie | String | Hash Function | Rolling Hash

Leetcode Link

Problem Description

The challenge is to count the unique non-empty substrings of a given text string that can be expressed as the concatenation of the same string twice. In other words, for a given string `text`, we are looking for all the distinguishable substrings which satisfy the condition that substring `X` can be written as `X = a + a`, where `a` is some string. We regard two substrings as distinct if they appear at different positions in `text`, even if they are composed of the same characters.

Intuition

To solve this problem efficiently, we utilize a hash function to represent substrings, which allows us to compare the equality of two substrings in constant time instead of linear time. This hash function needs to be carefully designed to avoid collisions where different substrings may have the same hash value.

The solution involves the following steps:

1. Precompute hash values for all prefixes of the text string, as well as powers of the base used in the hash function. This is to ensure we can calculate the hash for any substring quickly.
2. Iterate over all possible lengths of `2*a`, where `a` is the substring we are trying to find. We do this by using two nested loops, the outer loop going from the start of the text to the second last character, and the inner loop stepping through the text with steps of size 2 (since we're considering pairs of substring `a`).
3. For every such potential concatenation, use the prefix hashes to compute the hashes of the two halves and compare them.
4. If they are the same, it indicates we found a substring `a` such that `a + a` is a valid concatenated form present in `text`.
5. To count the distinct substrings, we use a set to store the hash of any `a` we find. The set will automatically handle the uniqueness part.
6. In the end, the size of this set gives us the number of distinct echo substrings.

Solution Approach

This solution approach heavily relies on computational string hashing to efficiently compare substrings. The idea behind string hashing is to transform a string into a numerical value (the hash) in such a way that if two strings are equal, their hashes should also be equal. String hashing supports constant-time substring comparison, which is crucial for this solution.

Here's the detailed solution approach:

1. **Hashing Initialization**: Before we start comparing substrings, we precompute and store the hash values for all possible prefixes of the text, as well as the powers of our base number (chosen arbitrarily to be `131`). The modulus value `mod` is chosen to be a large prime number to reduce the chance of collisions. These precomputations enable us to later obtain the hash of any substring in constant time.

2. **Computing Prefix Hashes and Powers**: We iterate through the entire text, character by character. For each character, we calculate the running hash and corresponding power of the base.

   - The hash for the (i+1)th prefix `h[i + 1]` is computed as `(h[i] * base + t) % mod`, where `t` is the current character's numerical value.
   - The power `p[i + 1]` is simply the previous power times the base, modulo `mod`.
3. **Enumerating Echo Substrings**: An echo substring is a string that can be evenly split into two identical halves (`a + a`). We iterate over all potential starting positions for a substring by using an outer loop. For each of these starting points, the inner loop tries to find the echo substrings with that starting point by varying the length in steps of 2.

4. **Hash Comparison**: To check if a potential substring is an echo substring, we compute the hash values for both halves using the precomputed prefix hash array and compare them. For example, for a substring starting at index `i` and ending at index `j`, we want to find if the substring partitioned at index `k` (`i + j) >> 1`) is an echo substring, so we calculate `get(i + 1, k + 1)` and `get(k + 2, j + 1)` and check for equality.

5. **Storing Unique Hashes**: If the two halves are identical (their hashes match), we add the hash of the first half to a set named `vis`. Since a set automatically dismisses duplicates, this ensures that only unique echo substrings' hashes are stored.

6. **Return the Answer**: The total number of unique echo substrings is the size of the `vis` set, which is returned as the final result.

An important thing to note is that since we are dealing with indices, we're careful to use 1-based indexing for the hashes and powers, because the initial hash (empty string hash) and initial power should be `0` and `1` respectively.

Lastly, by utilizing a hash function, the complexity of checking whether two substrings are equal is reduced from `O(n)` to `O(1)`, which makes this solution computationally efficient for larger texts.

Example Walkthrough

Let's illustrate the solution approach with an example. Consider the text string `text = "ababaa"` and we want to find all distinct echo substrings.

1. **Hashing Initialization**:

   - We choose a base, let's say 131, and a large prime as the modulus for hashing.
   - Prefix hashes and powers of the base will be precomputed. Assume an arbitrary prime number, for example, `mod = 10^9 + 7`.
2. **Computing Prefix Hashes and Powers**:

   - Let's initialize our hash array `h` and power array `p` with `h[0] = 0` and `p[0] = 1`.
   - Now, we calculate `h[1] = (h[0] * 131 + 'a') % mod`, `h[2] = (h[1] * 131 + 'b') % mod`, and so on.
   - Similarly, we calculate `p[1] = (p[0] * 131) % mod`, `p[2] = (p[1] * 131) % mod`, and so forth.
3. **Enumerating Echo Substrings**:

   - We explore all possible echo substrings. Start from the first character and consider all even-length substrings.
   - For instance, we'll look at substring lengths of 2, 4, up to the length of `text`.
4. **Hash Comparison**:

   - To check for echo substrings, we compare the computed hashes. For example, with the substring `text[0:1]` ("ab"), we compare the hash of `text[0:0]` ("a") and `text[1:1]` ("b").
   - Since these hashes will differ, we move on.
5. **Storing Unique Hashes**:

   - When we come across the substring `text[0:3]` ("abab"), we can split this into `text[0:1]` ("ab") and `text[2:3]` ("ab"). If hashes match, it's an echo substring.
   - We would then store the hash of `text[0:1]` in our `vis` set.
6. **Return the Answer**:

   - As we continue this process along `text`, we find that the only other echo substring is `text[2:5]` ("abaa"), which splits into `text[2:3]` ("ab") and `text[4:5]` ("aa").
   - The hash of `text[2:3]` ("ab") is already in `vis`, but `text[4:5]` ("aa") is new and will be added to `vis`.
   - Hence, the number of unique echo substrings is the size of `vis`, which contains two elements, so the answer is 2.

By following these steps, we efficiently process the text to find all unique echo substrings, ensuring that the operation scales well for larger strings by using constant-time hash comparisons.

Python Solution

```python
class Solution:
    def distinctEchoSubstrings(self, text: str) -> int:
        # Calculate the hash value of the substring from index l to index r
        def get_hash(l: int, r: int) -> int:
            return (prefix_hashes[r] - prefix_hashes[l - 1] * pow_base[r - l + 1]) % mod

        length_of_text = len(text)
        base = 131
        mod = 10**9 + 7

        # Initialize prefix hash and base power arrays
        prefix_hashes = [0] * (length_of_text + 1)
        pow_base = [1] * (length_of_text + 1)

        # Pre-compute hashes and powers of base for all prefixes
        for i, character in enumerate(text):
            char_code = ord(character) - ord('a') + 1
            prefix_hashes[i + 1] = (prefix_hashes[i] * base + char_code) % mod
            pow_base[i + 1] = (pow_base[i] * base) % mod

        # Use a set to record unique echo substrings by their hash values
        seen_echo_hashes = set()

        # Check for echo substrings in the text
        for i in range(length_of_text - 1):
            for j in range(i + 1, length_of_text, 2):
                middle = (i + j) // 2
                hash_first_half = get_hash(i + 1, middle + 1)
                hash_second_half = get_hash(middle + 2, j + 1)
                # If both halves are identical, record the hash of the first half
                if hash_first_half == hash_second_half:
                    seen_echo_hashes.add(hash_first_half)

        # The number of distinct echo substrings is the size of the hash set
        return len(seen_echo_hashes)

```

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

class Solution {
    private long[] prefixHashes;
    private long[] powersOfBase;

    public int distinctEchoSubstrings(String text) {
        int length = text.length();
        int base = 131; // A prime number chosen as the base for hashing
        prefixHashes = new long[length + 10];
        powersOfBase = new long[length + 10];
        powersOfBase[0] = 1;

        // Precompute prefix hashes and powers of base for the text
        for (int i = 0; i < length; ++i) {
            int charValue = text.charAt(i) - 'a' + 1;
            prefixHashes[i + 1] = prefixHashes[i] * base + charValue;
            powersOfBase[i + 1] = powersOfBase[i] * base;
        }

        // Use a hash set to store echo substrings' hashes without duplication
        Set<Long> vis = new HashSet<>();
        for (int i = 0; i < length - 1; ++i) {
            // The j index is incremented by 2 to ensure the substring can be split into two equal parts
            for (int j = i + 1; j < length; j += 2) {
                int mid = (i + j) / 2;
                long firstHalfHash = getHash(i + 1, mid + 1);
                long secondHalfHash = getHash(mid + 2, j + 1);

                // If the hashes match, the substrings are equal, so add the hash to the set
                if (firstHalfHash == secondHalfHash) {
                    vis.add(firstHalfHash);
                }
            }
        }

        // The number of distinct echo substrings is the size of the set
        return vis.size();
    }

    // Helper function to compute the hash of a substring using prefix hashes and the precomputed powers of base
    private long getHash(int start, int end) {
        return prefixHashes[end] - prefixHashes[start - 1] * powersOfBase[end - start + 1];
    }
}

```

C++ Solution

```cpp
#include <string>
#include <vector>
#include <unordered_set>

typedef unsigned long long ull; // Define `ull` for simplicity.

class Solution {
public:
    // Function to calculate the number of distinct echo substrings in the given text.
    int distinctEchoSubstrings(std::string text) {
        int length = text.size();
        int base = 131; // Base for polynomial rolling hash function.
        std::vector<ull> powers(length + 10); // Precompute the powers of base.
        std::vector<ull> hashValues(length + 10); // Hash values for prefixes of text.
        powers[0] = 1;

        // Initialize the powers and hashValues vectors.
        for (int i = 0; i < length; ++i) {
            int letterValue = text[i] - 'a' + 1;
            powers[i + 1] = powers[i] * base;
            hashValues[i + 1] = hashValues[i] * base + letterValue;
        }

        std::unordered_set<ull> uniqueEchoHashes; // Set to keep track of unique echo substrings.

        // Iterate over the text to find echo substrings.
        for (int i = 0; i < length - 1; ++i) {
            // Only need to check even length substrings for "echoes".
            for (int j = i + 1; j < length; j += 2) {
                int mid = (i + j) / 2;
                ull firstHalfHash = calculateHash(i + 1, mid + 1, powers, hashValues);
                ull secondHalfHash = calculateHash(mid + 2, j + 1, powers, hashValues);

                // If both halves match, add the hash to the set.
                if (firstHalfHash == secondHalfHash) {
                    uniqueEchoHashes.insert(firstHalfHash);
                }
            }
        }

        // The number of distinct echo substrings is the size of the set.
        return uniqueEchoHashes.size();
    }

private:
    // Helper function to calculate the hash of a substring using precomputed powers and hash values.
    ull calculateHash(int left, int right, const std::vector<ull>& powers, const std::vector<ull>& hashValues) {
        return hashValues[right] - hashValues[left - 1] * powers[right - left + 1];
    }
};

```

Typescript Solution

```typescript
// Define 'ull' for simplicity.
type ull = bigint;

// Base for polynomial rolling hash function.
const base: ull = 131n;

// Function to calculate the hash of a substring using precomputed powers and hash values.
function calculateHash(left: number, right: number, powers: ull[], hashValues: ull[]): ull {
  return hashValues[right] - hashValues[left - 1] * powers[right - left + 1];
}

// Function to calculate the number of distinct echo substrings in the given text.
function distinctEchoSubstrings(text: string): number {
  const length: number = text.length;

  // Precompute the powers of base.
  const powers: ull[] = Array(length + 10).fill(0n);
  // Hash values for prefixes of text.
  const hashValues: ull[] = Array(length + 10).fill(0n);
  powers[0] = 1n;

  // Initialize the powers and hashValues arrays.
  for (let i = 0; i < length; ++i) {
    const letterValue: ull = BigInt(text.charCodeAt(i) - 'a'.charCodeAt(0) + 1);
    powers[i + 1] = powers[i] * base;
    hashValues[i + 1] = hashValues[i] * base + letterValue;
  }

  // Set to keep track of unique echo substring hashes.
  const uniqueEchoHashes: Set<ull> = new Set();

  // Iterate over the text to find echo substrings.
  for (let i = 0; i < length - 1; ++i) {
    // Only need to check even length substrings for "echoes".
    for (let j = i + 1; j < length; j += 2) {
      const mid: number = Math.floor((i + j) / 2);
      const firstHalfHash: ull = calculateHash(i + 1, mid + 1, powers, hashValues);
      const secondHalfHash: ull = calculateHash(mid + 2, j + 1, powers, hashValues);

      // If both halves match, add the hash to the set.
      if (firstHalfHash === secondHalfHash) {
        uniqueEchoHashes.add(firstHalfHash);
      }
    }
  }

  // The number of distinct echo substrings is the size of the set.
  return uniqueEchoHashes.size;
}

// Example usage:
// const result: number = distinctEchoSubstrings("yourtext");
// console.log(result);

```

Time and Space Complexity

Time Complexity

The time complexity of the code is mainly determined by the two nested loops that run over the string to find all possible echo substrings. For a string of length `n`, the outer loop runs `n - 1` times and the inner loop runs in the order of `n/2` times on average (since it increments by `2` each time). Within the inner loop, the time to calculate hashes is constant, thanks to the precomputed hash and power values.

So, the total time complexity is O(n^2) considering the nested loops and the fact that each hash computation within the loops is done in constant time.

Space Complexity

Space complexity comes from the storage of precomputed hash values `h` and power values `p`, as well as the hash set `vis` used to store unique hash values corresponding to echo substrings. This results in O(n) space complexity.

- `h` and `p` each require O(n) space.
- `vis` may potentially store up to O(n/2) distinct hashes (in the worst case, every substring might be an echo substring).

Therefore, the space complexity is O(n) + O(n) + O(n/2), which simplifies to O(n).