2416. Sum of Prefix Scores of Strings

Hard Trie Array String Counting Leetcode Link

Problem Description

The problem presents us with an array of non-empty strings called `words`. The task is to calculate a score for each string, which represents the count of other strings in the array that have this string as a prefix. Specifically, for a given string `word` from the array, its score is determined by how many times it serves as a prefix for the other strings in `words`. It's worth highlighting that a string is always considered a prefix of itself.

As an example, if `words` contains `["a", "ab", "abc", "cab"]`, the score for the string `"ab"` is `2` because it is a prefix for both `"ab"` and `"abc"`. We are asked to calculate and return an array `answer` where `answer[i]` is equal to the sum of scores for all non-empty prefixes of `words[i]`.

Intuition

The core idea to solve this problem is to use a Trie data structure. A Trie is an efficient information retrieval data structure that is used to represent the "retrieval" of data. It specializes in managing strings that are in a particularly large dataset, allowing for fast and efficient prefix-based searches.

Here's why using Trie is an effective solution:

1. When we insert each word into the Trie, we simultaneously increase the count at each node. This count can be interpreted as the number of words in which the string represented by the path to that node serves as a prefix.
2. Inserting all the words into the Trie ensures that every prefix is accounted for, and their respective scores are updated properly throughout the Trie nodes.
3. To find the total score of all non-empty prefixes of a word, we search the Trie for that word. As we go through each character of the word, we add to the answer the counts encountered at each node. These counts represent the score of the prefix ending at that node.
4. By doing this for each word in `words`, we end up with an array of total prefix scores corresponding to each word, as required.

Using this approach, we can efficiently calculate the score for each word without re-comparing each string with all other strings in every iteration, avoiding a brute-force solution that would be much less efficient.

Solution Approach

The provided solution uses a Trie data structure to efficiently compute the sum of scores for the prefixes of each word in the input array `words`. Here's a step-by-step explanation of how the solution works:

Class Design

A class `Trie` is defined with two primary attributes: `children` and `cnt`. `children` is a list of length 26, which corresponds to the 26 letters of the English alphabet, and `cnt` is an integer representing the count of words that pass through this particular node.

1. **Initialization**: The `Trie` instance is initialized with `children` set to an array of `None` values (since it's empty initially) and the `cnt` count set to `0`.
2. **Insert Method**: The `insert` method takes a word (`w`) and adds it to the Trie. With each character insertion, the respective character index is computed (`ord(c) - ord('a')`) to identify the relative position of the node in `children`. If the node is `None`, a new `Trie` node is created. Every time a character is inserted, `node.cnt` is incremented, indicating that one more word that has this prefix has been added to the Trie.
3. **Search Method**: The `search` method computes the total score of prefixes for a given word (`w`). It iterates through each character of the word, moving through the Trie nodes and summing up the `cnt` values encountered at each step. If it encounters `None`, meaning that no further prefix exists, it returns the sum collected so far. Otherwise, it continues until all characters of the word have been processed, accumulating the total score.

Solution Class

The `Solution` class contains the `sumPrefixScores` method, which accepts the `words` list and returns a list of total prefix scores for each word in `words`.

1. A new `Trie` instance is created.
2. Each word in the `words` array is inserted into the Trie using the `insert` method of the `Trie` class. This builds the Trie structure with the correct `cnt` values for each node, representing the scores of the corresponding prefixes.
3. Finally, the `search` method of the `Trie` class is used to calculate the total score of non-empty prefixes for each word. This is done inside a list comprehension that returns the resulting list of scores.

By following this approach, the algorithm efficiently calculates the score of each word by leveraging the Trie structure, avoiding redundancy and excessive comparisons that would be inherent in a brute-force approach.

Example Walkthrough

Let's walk through the solution approach using the example array `words = ["top", "tops", "toss", "taco", "tap"]`. We'd like to calculate the score for each string in the array by determining the number of times it serves as a prefix for other strings in the array.

Initial Trie Construction

1. We start by creating an empty Trie.
2. We insert the first word "top". Now "t", "to", and "top" all have a count of 1 in their respective nodes.
3. We insert the second word "tops". Since "top" is already there, when we insert "s", "tops" also has a count of 1, and the "top" node count increases to 2.
4. We proceed similarly with "toss", increasing the count of "tos" and "toss" to 1, and the node "t" will have a count of 3 because 3 words have been inserted starting with "t".
5. We insert "taco", similarly building up counts for "t", "ta", "tac", and "taco".
6. We insert "tap", also building up counts for those nodes.

Now, for example, if we consider the node representing "ta", it has a count of 2 because it serves as a prefix for "taco" and "tap".

Computing Scores

1. For each word in `words`, we will traverse the Trie node by node, summing the counts.
2. Taking "top" as an example, the traversal would be 't' → 'to' → 'top', and for each of those we would sum the counts encountered: 3 (at 't') + 3 (at 'to') + 2 (at 'top') = 8.
3. Similarly, we traverse for "tops", resulting in counts of 3 (at 't') + 3 (at 'to') + 2 (at 'top') + 1 (at 'tops') = 9.
4. We repeat this for each word: "toss" (8), "taco" (5), and "tap" (6).

Result

As a result, for the given 'words' array, the `sumPrefixScores` function will return the array `[8, 9, 8, 5, 6]`. Each element of the returned array corresponds to the total score of all the non-empty prefixes for each word in the `words` array.

Thus, the Trie-based approach efficiently computes the prefix scores for all words in the given array without redundant comparisons.

Python Solution

```python
class Trie:
    def __init__(self):
        # Initialize each Trie node with 26 pointers for each lowercase letter in the alphabet
        self.children = [None] * 26
        # Initialize a variable to keep count of the number of words passing through this node
        self.count = 0

    def insert(self, word):
        # Start from the root node
        node = self
        for char in word:
            # Convert character to the corresponding index (0-25 for 'a'-'z')
            index = ord(char) - ord('a')
            # If the child node for this character doesn't exist, create it
            if node.children[index] is None:
                node.children[index] = Trie()
            # Move to the child node
            node = node.children[index]
            # Increment the count for each node that is part of a word
            node.count += 1

    def search(self, word):
        # Start from the root node
        node = self
        # To gather the cumulative count of all prefixes of the searched word
        total_count = 0
        for char in word:
            # Convert character to index
            index = ord(char) - ord('a')
            # If the node doesn't have a child at this index, return current total count
            if node.children[index] is None:
                return total_count
            # Move to the child node
            node = node.children[index]
            # Add the node's count to the total count
            total_count += node.count
        # Return the total count which represents the sum of the prefix scores for the word
        return total_count


class Solution:
    def sum_prefix_scores(self, words):
        # Create an instance of the Trie
        trie = Trie()
        # Insert each word into the trie, building up prefix counts
        for word in words:
            trie.insert(word)
        # Retrieve and return the sum of the prefix scores for each word
        return [trie.search(word) for word in words]

```

Java Solution

```java
class Trie {
    private Trie[] children = new Trie[26]; // Each trie node can have up to 26 children for each letter of the alphabet
    private int count; // This variable counts the number of times a node is visited during insertions

    // Function to insert a word into the Trie
    public void insert(String word) {
        Trie node = this;
        for (char c : word.toCharArray()) {
            int index = c - 'a'; // Obtain the index by subtracting the ASCII value of 'a' from the character
            if (node.children[index] == null) {
                node.children[index] = new Trie(); // Create a new Trie node if it does not exist
            }
            node = node.children[index];
            node.count++; // Increment the count for each node visited
        }
    }

    // Function to compute the sum of the prefix scores for the given word
    public int search(String word) {
        Trie node = this;
        int sum = 0;
        for (char c : word.toCharArray()) {
            int index = c - 'a'; // Obtain the index as done in insert function
            if (node.children[index] == null) {
                return sum; // Return the current sum if the node is null (prefix does not exist)
            }
            node = node.children[index];
            sum += node.count; // Add the count of the visited node to the sum
        }
        return sum;
    }
}

class Solution {
    // Function to compute the sum of prefix scores for each word in the array
    public int[] sumPrefixScores(String[] words) {
        Trie trie = new Trie(); // Initialize a new Trie
        for (String word : words) {
            trie.insert(word); // Insert each word from the array into the Trie
        }
        int[] answer = new int[words.length]; // Create an array to hold the result
        for (int i = 0; i < words.length; i++) {
            answer[i] = trie.search(words[i]); // Compute the sum of prefix scores for each word in the array
        }
        return answer; // Return the computed scores
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
using namespace std;

// Trie node structure to store the children and the word count.
class Trie {
private:
    vector<Trie*> children; // Children nodes of the current node.
    int count; // Count of words passing through this node.

public:
    // Constructor initializes the children to hold 26 Trie pointers, one for each letter.
    Trie()
        : children(26), count(0) {}

    // Insert a word into the Trie.
    void insert(const string& word) {
        Trie* node = this;
        for (char letter : word) {
            int index = letter - 'a'; // Convert character to index (0-25).
            // If no child exists at the index, create a new Trie node.
            if (!node->children[index]) {
                node->children[index] = new Trie();
            }
            node = node->children[index];
            ++node->count; // Increment count on traversed nodes.
        }
    }

    // Search for a word in the Trie and return the sum of the counts of all its prefixes.
    int search(const string& word) {
        Trie* node = this;
        int sum = 0;
        for (char letter : word) {
            int index = letter - 'a'; // Convert character to index (0-25).
            // If the child at the index doesn't exist, return the current sum.
            if (!node->children[index]) return sum;
            node = node->children[index];
            sum += node->count; // Add count at the node to the sum.
        }
        return sum; // Return the total sum for the word.
    }
};

// Main Solution class to solve the problem using the Trie.
class Solution {
public:
    // Method to calculate the sum of prefix scores for each word in the input vector.
    vector<int> sumPrefixScores(vector<string>& words) {
        Trie* trie = new Trie(); // Create a new Trie.
        // Insert each word from the input vector into the Trie.
        for (const string& word : words) {
            trie->insert(word);
        }
        vector<int> answer; // Vector to store the result.
        // Search each word and calculate the sum of prefix scores.
        for (const string& word : words) {
            answer.push_back(trie->search(word));
        }
        return answer; // Return the resulting vector of sums.
    }
};

```

Typescript Solution

```typescript
// Declare the trie node structure.
interface TrieNode {
  children: TrieNode[];
  count: number;
}

// Create a function to initialize a trie node.
function createTrieNode(): TrieNode {
  return {
    children: new Array(26),
    count: 0
  };
}

// Initialize the trie root node.
let root: TrieNode = createTrieNode();

// Function to insert a word into the trie.
function insert(word: string): void {
  let node = root;
  for (const char of word) {
    const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
    if (!node.children[index]) {
      node.children[index] = createTrieNode();
    }
    node = node.children[index];
    node.count++;
  }
}

// Function to search for a word in the trie and calculate the sum of counts of its prefixes.
function search(word: string): number {
  let node = root;
  let sum = 0;
  for (const char of word) {
    const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
    if (!node.children[index]) {
      return sum;
    }
    node = node.children[index];
    sum += node.count;
  }
  return sum;
}

// Function to calculate the prefix scores for a list of words.
function sumPrefixScores(words: string[]): number[] {
  // Reset the trie for new entries.
  root = createTrieNode();

  // Insert words to the trie.
  for (const word of words) {
    insert(word);
  }

  // Compute and collect the prefix scores for each word.
  let scores: number[] = [];
  for (const word of words) {
    scores.push(search(word));
  }

  // Return the collected scores.
  return scores;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `insert` method in the `Trie` class is `O(L)`, where `L` is the length of the word being inserted. This is because we iterate through each character in the word once to insert it into the trie.

Given `N` words, inserting them all into the trie would take `O(N * L)` time, where `L` is the average length of the words.

The `search` method also runs in `O(L)` for a similar reason – we go through each character of the word to compute the accumulated scores.

Therefore, when searching for the prefix scores for all words, assuming `L` is again the average length of the words, this operation would also take `O(N * L)` time.

Overall, the combined time complexity for constructing the trie and then performing the search for all words is `O(N * L)`.

Space Complexity

The space complexity of the trie is `O(M * 26)` where `M` is the total number of unique nodes in the trie. `M` depends on the total number of unique letters in all inserted words. Since each node has an array of 26 elements to hold references to its children, the `26` factor is included. In the worst-case scenario where there is no overlap of prefixes, `M` can be as large as `N * L`, leading to a space complexity of `O(N * L * 26)`. However, in practice, since there is likely significant overlap (common prefixes) when inserting words in English, the actual space used is usually much less than this upper bound.

Thus, we can more accurately denote the space complexity as `O(M)`, acknowledging that in the average case, this does not reach the worst-case scenario of `O(N * L * 26)` because of the common prefixes in the trie.