792. Number of Matching Subsequences

Medium | Trie | Array | Hash Table | String | Binary Search | Dynamic Programming | Sorting | Leetcode Link

Problem Description

The problem requires you to determine how many strings in the array `words` are subsequences of the string `s`. A subsequence of a string is defined as a sequence that can be derived from the original string by removing some characters without altering the order of the remaining characters. For instance, "ace" is a subsequence of "abcde" because you can remove 'b' and 'd' to get "ace". Your goal is to return the count of strings in `words` that can be formed in this way from the string `s`.

Intuition

To solve this problem, the key idea is to track the progress of each word in the `words` array as we iterate through the string `s`. Instead of checking each word against the entire string `s` at once, which would be inefficient, we use a smarter approach. We create a dictionary that maps the first character of each word to a queue holding tuples that represent the words. Each tuple consists of an index `i` and a position `j`, where `i` is the index of the word in the `words` array, and `j` is the position up to which we have successfully matched the word in `words[i]` with characters in `s`.

As we iterate through the string `s`, we get the queue of words (tuples) that are expecting the current character. For each word in the queue, if we find a matching character in `s`, we move to the next character in the word by incrementing `j`. If we reach the end of a word, it means we have successfully found it as a subsequence in `s`, and we increment our answer count (`ans`).

We use a queue because it efficiently supports removing elements from the front, allowing us to process the words in the order they are expecting to match characters from `s`. By organizing the words based on the next expected character, we only consider the relevant part of `words` for each character in `s`, making the process efficient.

Solution Approach

The solution approach makes use of the following algorithms, data structures, and patterns:

- **Dictionary with Queue**: A `defaultdict(deque)` from Python's `collections` module is used to map the first character of each word in `words` to a queue containing tuples of the form `(i, j)`. Here `i` is the index of the word in the array `words`, and `j` is the current character position we have matched in the word so far.
- **Character Iteration and Matching**: The solution iterates through each character `c` in the string `s` and uses it to reference the queue of tuples waiting for that character (`d[c]`).
- **Queue Processing**: Within this iteration, another loop runs for the length of the queue `d[c]`, which processes each tuple. For each tuple `(i, j)` dequeued from `d[c]`, the `j` is incremented to point to the next character since the current one `c` is a match.
- **Subsequence Completion Check**: If the incremented `j` equals the length of the word `words[i]`, it means all the characters in `words[i]` are matched in sequence with some characters of `s`, therefore `words[i]` is a subsequence of `s`. When this condition is met, the answer counter `ans` is incremented.
- **Updating the Dictionary with Queues**: If `j` is not equal to the length of the word, the tuple `(i, j)` is not completed yet and must wait for the next character `words[i][j]`. Hence, it is appended back into the queue corresponding to this next expected character `d[words[i][j]]`.

In essence, the algorithm processes `s` character by character, moving 'pointers' through each word in `words` when a matching character is found, and recounts every time a word is completely matched as a subsequence.

The algorithm complexity is mainly O(N + M), where N is the length of the string `s`, and M is the total length of all words in `words`. It's because each character of each word is processed at most once when the corresponding character in `s` is found.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach in action. Consider the string `s = "abcde"` and the list of words `words = ["a", "bb", "acd", "ace"]`. We want to find how many words are subsequences of `s`.

To begin, we initialize our dictionary with queues, `d`, that will hold the progress for each word in `words` as we iterate over `s`. For our example, the initial state of the dictionary will be as follows since each queue will contain tuples with the index of the word in `words` and the initial character position `0`.

```
{
  'a': deque([(0, 0), (2, 0), (3, 0)]),
  'b': deque([(1, 0)]),
}
```

Now we start iterating over `s`, character by character:

1. For the first character `'a'` in `s`, we look in `d` to find the queue for `'a'`, which is `deque([(0, 0), (2, 0), (3, 0)])`. We have three words waiting to match the character `'a'` in `s`.

   a. dequeue (0, 0): Since 'a' in `s` matches the first character of `words[0]` which is "a", it's a complete match. We increment `ans` to 1.

   b. dequeue (2, 0): Now we're looking at `words[2]` which is "acd". The character 'a' matched, so we increment `j` to 1 and requeue the tuple since "acd" is not yet fully matched. Now the tuple is (2, 1).

   c. dequeue (3, 0): Similarly, for `words[3]` which is "ace", we have a match on 'a', increment `j` to 1, and requeue the tuple to (3, 1).

   Now our dictionary looks like this:

   ```
   {
     'b': deque([(1, 0)]),
     'c': deque([(2, 1), (3, 1)]),
   }
   ```

2. The second character of `s` is `'b'`. We check our dictionary for the queue under `'b'`, which is `deque([(1, 0)])`.

   a. dequeue (1, 0): We look at `words[1]` which is "bb". Since 'b' matches the first character, we increment `j` to 1 and enqueue the tuple (1, 1) back because it's still awaiting another 'b'.

   And the dictionary updates to:

   ```
   {
     'c': deque([(2, 1), (3, 1)]),
     'b': deque([(1, 1)]),
   }
   ```

3. Next, `s` gives us 'c'. We process the queue under `'c'`, which is `deque([(2, 1), (3, 1)])`.

   a. dequeue (2, 1): "acd" in words[2] has matched 'a' and now 'c', so `j` becomes 2. Since `j` is not equal to the length of "acd", we enqueue tuple (2, 2) since we are now waiting for 'd'.

   b. dequeue (3, 1): Similarly, "ace" in words[3] has matched 'a' and 'c', and `j` is incremented to 2, indicating we are waiting for 'e' next.

   Updated dictionary:

   ```
   {
     'b': deque([(1, 1)]),
     'd': deque([(2, 2)]),
     'e': deque([(3, 2)]),
   }
   ```

4. The next character in `s` is `'d'`, which affects the queue under `'d'`.

   a. dequeue (2, 2): The tuple for "acd" is fully matched now because 'd' was the last needed character, so we increment `ans` by 1 (total `ans` is now 2).

   Dictionary after the operation:

   ```
   {
     'b': deque([(1, 1)]),
     'e': deque([(3, 2)]),
   }
   ```

5. The last character in `s` is `'e'`, and we process the queue under `'e'`.

   a. dequeue (3, 2): Finally, "ace" is fully matched, and our total count `ans` is incremented by one more (total `ans` is now 3).

The end state of the dictionary is not relevant anymore since we have processed all of `s`. The algorithm finishes with `ans = 3` because we've found that three out of four words in `words` are subsequences of `s`. The word "bb" was not a subsequence since the second 'b' couldn't be matched.

Python Solution

```python
from collections import defaultdict, deque

class Solution:
    def num_matching_subseq(self, string: str, words: List[str]) -> int:
        """
        Count the number of words from the list 'words' that are subsequences of the string 'string'.

        Args:
        string (str): the string to be searched for subsequences.
        words (List[str]): the list of words to check as potential subsequences.

        Returns:
        int: the number of words that are subsequences of 'string'.
        """

        # Initialize a dictionary to hold queues of tuples for each character in words.
        # Each tuple contains an index of a word in words and the next character index to search for.
        char_to_word_indices = defaultdict(deque)

        # Populate the dictionary with initial character indices for each word.
        for index, word in enumerate(words):
            char_to_word_indices[word[0]].append((index, 0))

        # Counter for the number of matching subsequences found.
        matching_count = 0

        # Loop through each character in the input string.
        for char in string:
            # Process all tuples (word index, character index) for the current character.
            for _ in range(len(char_to_word_indices[char])):
                word_index, char_index = char_to_word_indices[char].popleft()
                char_index += 1  # Move to the next character in the current word

                # If we reached the end of the word, increment the count of matching subsequences.
                if char_index == len(words[word_index]):
                    matching_count += 1
                else:
                    # Else, append the tuple (word index, next character index) to the appropriate list.
                    next_char = words[word_index][char_index]
                    char_to_word_indices[next_char].append((word_index, char_index))

        # Return the total count of matching subsequences.
        return matching_count
```

Remember to include the necessary import statement at the top for `List` from `typing`:

```python
from typing import List

```

Java Solution

```java
class Solution {
    public int numMatchingSubseq(String s, String[] words) {
        // Create an array of dequeues to store positions of characters in words
        Deque<int[]>[] waitingChars = new Deque[26];

        // Initialize each dequeue in the array
        Arrays.setAll(waitingChars, k -> new ArrayDeque<>());

        // Fill the deques with the first characters of each word in words array
        for (int i = 0; i < words.length; ++i) {
            waitingChars[words[i].charAt(0) - 'a'].offer(new int[]{i, 0});
        }

        int matches = 0; // Store the number of matching subsequences

        // Iterate through each character of the string s
        for (char c : s.toCharArray()) {
            // Get the deque corresponding to the current character
            Deque<int[]> queue = waitingChars[c - 'a'];

            // Process all elements in queue
            for (int i = queue.size(); i > 0; --i) {
                // Dequeue the first element
                int[] pair = queue.pollFirst();
                int wordIndex = pair[0], charIndex = pair[1] + 1;

                // If the whole word is found, increment the match count
                if (charIndex == words[wordIndex].length()) {
                    ++matches;
                } else {
                    // Otherwise, enqueue the pair back with the updated character index
                    waitingChars[words[wordIndex].charAt(charIndex) - 'a'].offer(new int[]{wordIndex, charIndex});
                }
            }
        }
        // Return the total number of matching subsequences found
        return matches;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of words from the `words` vector that are subsequences of the string `s`
    int numMatchingSubseq(string s, vector<string>& words) {
        // Create a vector of queues to store pairs of indices.
        // Each element represents a queue for words starting with the corresponding character ('a' to 'z').
        vector<queue<pair<int, int>>> charQueues(26);

        // Populate the queues with pairs: the index of the word in the `words` vector and the character index being checked.
        for (int i = 0; i < words.size(); ++i) {
            charQueues[words[i][0] - 'a'].emplace(i, 0);
        }

        // Initialize the answer to 0.
        int ans = 0;

        // Iterate over each character in the string `s`
        for (char& c : s) {
            // Reference to the queue corresponding to the current character
            auto& queue = charQueues[c - 'a'];

            // Process each pair in the queue
            for (int t = queue.size(); t > 0; --t) {
                // Get the front pair (word index, char index) and remove it from the queue
                auto [wordIndex, charIndex] = queue.front();
                queue.pop();

                // Increment the char index to check the next character of the word
                ++charIndex;

                // If all characters in the word have been found in `s`, increment the answer
                if (charIndex == words[wordIndex].size()) {
                    ++ans;
                } else {
                    // Otherwise, push the pair (word index, next char index) to the corresponding queue
                    // for the next character to be checked in `words[wordIndex]`
                    charQueues[words[wordIndex][charIndex] - 'a'].emplace(wordIndex, charIndex);
                }
            }
        }
        // Return the total count of matching subsequences
        return ans;
    }
};

```

Typescript Solution

```typescript
// Define a method to count the number of words from the `words` array that are subsequences of the string `s`.
function numMatchingSubseq(s: string, words: string[]): number {
    // Create an array of queues to store pairs of indices.
    // Each element represents a queue for words starting with the corresponding character ('a' to 'z').
    const charQueues: Array<Queue<[number, number]>> = Array.from(
        { length: 26 },
        () => new Queue<[number, number]>());

    // Populate the queues with pairs: the index of the word in the `words` array and the character index being checked.
    for (let i = 0; i < words.length; ++i) {
        const charIndex = words[i].charCodeAt(0) - 'a'.charCodeAt(0);
        charQueues[charIndex].enqueue([i, 0]);
    }

    // Initialize the answer to 0.
    let ans = 0;

    // Iterate over each character in the string `s`.
    for (const c of s) {
        // Reference to the queue corresponding to the current character.
        const queue = charQueues[c.charCodeAt(0) - 'a'.charCodeAt(0)];

        // Process each pair in the queue.
        const size = queue.size();
        for (let t = 0; t < size; ++t) {
            // Get the front pair (word index, char index) and remove it from the queue.
            const [wordIndex, charIndex] = queue.dequeue();

            // Increment the char index to check the next character of the word.
            const nextCharIndex = charIndex + 1;

            // If all characters in the word have been found in `s`, increment the answer.
            if (nextCharIndex === words[wordIndex].length) {
                ans++;
            } else {
                // Otherwise, push the pair (word index, next char index) to the corresponding queue
                // for the next character to be checked in `words[wordIndex]`.
                const nextCharCode = words[wordIndex].charCodeAt(nextCharIndex) - 'a'.charCodeAt(0);
                charQueues[nextCharCode].enqueue([wordIndex, nextCharIndex]);
            }
        }
    }

    // Return the total count of matching subsequences.
    return ans;
}

// Helper class for Queue
class Queue<T> {
    private data: T[] = [];

    enqueue(item: T): void {
        this.data.push(item);
    }

    dequeue(): T {
        return this.data.shift();
    }

    size(): number {
        return this.data.length;
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed as follows:

- Initializing the dictionary `d` with a `deque` for each unique character has a space complexity that depends on the number of unique characters in `words` list, but initializing the deques themselves takes O(1) time.
- The first loop goes through all the words in `words` list to populate the dictionary `d`. Each word is processed in O(1) time to append its index and starting character position to the corresponding deque, resulting in O(n) time complexity for this step, where n is the number of words in `words`.
- The second loop goes through each character in the input string `s`. For each character `c`, it processes all indices `(i, j)` in `d[c]`. Since each word is exactly processed once, and each character in `s` results in at most one deque operation (either `popleft` or `append`), the number of operations is proportional to the length of `s` plus the total length of all words. Hence, the time complexity, in this case, is O(m + k), where m is the length of `s` and k is the sum of lengths of all words in `words`.

Thus, the overall time complexity combines the complexities of both loops, resulting in O(n + m + k).

Space Complexity

The space complexity of the code is determined by:

- The dictionary `d` which contains as many keys as there are unique starting characters in the `words`, and each key has a `deque` that contains at most n tuples `(i, j)`, where `n` is the number of words. However, since each word can only be in one deque at a time, it's more precise to say that it contains a total of n tuples across all deques.
- There are no other data structures that require significant space.

Therefore, the space complexity is O(n), where n is the number of words in `words`.