30. Substring with Concatenation of All Words

Hard | Hash Table | String | Sliding Window

Leetcode Link

Problem Description

You are given a string `s` and an array of strings `words`. All strings within `words` have the same length. Your objective is to find all the starting indices of substrings in `s` where a substring is a concatenation of every word in any possible permutation of `words`.

Consider `words` as a set of building blocks, where each word is a block of the same size. You have to find all position in `s` where you can construct a substring using every block exactly once, arranging them in any sequence.

A "concatenated substring" is such that:

- It consists of all words from the `words` array.
- The words can be in any order.
- Each word from `words` appears exactly as it is (unmodified, and in full).

For example, if `words` is `["ab","cd","ef"]`, then `"abcdef"`, `"abefcd"`, and `"cdabef"` are concatenated strings if they occur in `s`. On the other hand, `"acdbef"` is not a concatenated substring because it doesn't represent any permutation of `words`.

You need to return a list of starting indices of such concatenated substrings found in `s`, the order of indices does not matter.

Intuition

To arrive at the solution for this problem, we have to consider that searching for each possible permutation of `words` in `s` would be computationally expensive. We need a more efficient way to check for concatenated substrings.

The intuition behind the solution lies in the pattern recognition, sliding window, and hash table techniques:

- **Pattern Recognition**: Since all words in `words` have the same length, our window of the substring to search in `s` will also be a constant size, which is the sum of lengths of all words in `words`.
- **Sliding Window**: As we slide this window across `s`, we can incrementally check whether the substring window contains a valid concatenation of the words.
- **Hash Table**: By using a hash table to count the occurrences of the words we've seen in the current window, we can keep track of the words and their counts to determine if the current window forms a valid concatenated substring.

The sliding window starts from the beginning of `s`, and we move it to the right one word-length at a time. We continue this process for all possible positions the first word of the window could start from (i.e., 0 to word-length - 1).

At each step, when a new word is included in the sliding window:

- We add it to a current word frequency count hash table.
- If this new word isn't in the original words count hash table, we reset the current one, as it's not a valid continuation.
- If the count of the newly added word exceeds its expected count, we slide the window's left bound to the right to exclude enough occurrences, so the counts match.
- Whenever the number of words within the sliding window is equal to the size of `words` and all word counts correspond, we record the starting index.

By following this strategy, we avoid computing all permutations of `words`, with the sliding window efficiently narrowing down possible starting indices.

Solution Approach

The solution approach involves using a hash table and a sliding window, as already suggested by the intuition.

1. **Hash Table for Counting Words**: A hash table is used to count the number of times each word appears in the `words` array. This is the `cnt` hash table in the code.

2. **Setting Up for Sliding Window**: The key variables are set up for the sliding window algorithm such as the length of the string (`m`), the number of words (`n`), and the length of each word which is assumed to be uniform (`k`). The variable `ans` is an array that will collect our answer - the starting indices of valid concatenated substrings.

3. **Iterating Starting Points**: We start iterating over the string `s` with variable `i` which represents the start point of the sliding window. This loop essentially allows us to accommodate words in `s` starting from different positions up to the word length, making sure we cover all possible alignments of words within `s`.

4. **Sliding Window Mechanism**: Inside the loop, we initialize a counter for the current window (`cnt1`), the left (`l`) and right (`r`) boundaries of the window, and the variable `t` to keep track of the total number of valid words encountered in the current window.

5. **Processing Words**: We then continue to shift our window to the right word by word, capturing each word using `s[r:r+k]`. We also keep checking if the captured word is in the `cnt` hash table. If it's not in `cnt`, we reset our counters and move the window forward because the word does not belong to any concatenation of `words`.

6. **Updating Word Count**: If the word is valid, we increment its count in `cnt1` and also the total count `t`.

7. **Validating Counts**: If there are too many occurrences of the word in the current window, we increment the left boundary of the window `l` to reduce the count of the word in the window. This is necessary to match the word count exactly to that of the input words count.

8. **Storing Valid Indices**: If the total number of words `t` in the sliding window equals the number of words in `words` (`n`), it means we found a valid concatenation starting at index `l`. This index is stored in the answer array `ans`.

By the end of the outer loop, we have considered all possible alignments and have added all valid starting points to the `ans` array, which is then returned.

The sliding window algorithm is made efficient by the fact that it maintains a balance of counts dynamically as it shifts over the string `s`, ensuring the constraints are satisfied before adding a starting index to the answer array.

Example Walkthrough

Let's go through a simple example to illustrate the solution approach:

Suppose the input string `s` is `"wordgoodgoodgoodbestword"` and `words` is `["word","good","best"]`, where each word in `words` is of length 4.

Now let's follow the steps of the approach to find the starting indices of the concatenated substrings:

1. **Create the Hash Table**: Create a hash table (`cnt`) to store the frequency of each word in `words`.

   `cnt = {"word": 1, "good": 1, "best": 1}`

2. **Set Up Sliding Window Variables**: The length of each word (`k`) is 4, `s` length (`m`) is 22, and there are 3 words (`n`) in `words`.

   Starting indices of `ans` will be collected in an array.

3. **Iterate Starting Points**: Iterate with `i` from 0 to less than `k` (since `k` is 4, `i` will take values 0, 1, 2, and 3).

4. **For `i = 0`**: (First window position)

   - Initialize `cnt1` (current window frequency), `l` (left boundary of window), `r` (right, initially `i`), and `t` (total valid words in current window, initially 0).
   - Start the right boundary `r` at index 0 and `l` at index 0.
   - Slide `r` by `k` to capture a word. (first word `word` from indices 0 to 4).
   - Since `word` is in `cnt`, add to `cnt1` and increment `t`.
   - Now, `r = 4`, `l = 0`, and `t = 1`.
   - Continue shifting `r` and repeat the process.
   - At `r = 16`, the window captures `best`, add it to `cnt1` and increment `t`.
   - At this point, `r = 20`, `l = 0`, and `t = 3`.
   - Since `t` equals the number of words in `words`, we have a valid starting index at `l`, which is 0.

   Now `ans` array has `[0]`.

5. **For `i = 1`** (second possible window position):

   - Similar process, we initialize the variables again, but we start checking from the first index.
   - No match will be found starting at `i = 1`.

6. **For `i = 2`** (third possible window position):

   - Similarly, initialize and check from index 2.
   - No match will be found starting at `i = 2`.

7. **For `i = 3`** (fourth possible window position):

   - Initialize and start checking from index 3.
   - A match is found for the second occurrence of the valid concatenated substring starting at index `l = 9`.

   We add 9 to our `ans` array, now `ans` becomes `[0, 9]`.

By the end of the algorithm, we've checked all possible alignments within `s` for concatenations of all words in `words`. Thus, the final `ans` array contains `[0, 9]`, which are the starting indices where the concatenated substrings appear in `s`.

The solution is efficient because it does not calculate all permutations of `words`; instead, it builds the concatenated substring as it slides over `s`, and validates the counts of encountered words against `cnt` to determine valid starting points.

Python Solution

```python
from collections import Counter

class Solution:
    def findSubstring(self, s: str, words: List[str]) -> List[int]:
        # Count the frequency of each word in the 'words' list
        word_count = Counter(words)
        total_length = len(s)
        num_words = len(words)
        word_length = len(words[0])

        # This will hold the start indices of the substrings
        start_indices = []

        # Check every word_length characters to find valid substrings
        for offset in range(word_length):
            left = right = offset  # Initialize two pointers
            window_word_count = Counter()
            total_matched_words = 0

            # Move the window rightwards in the string 's' by word_length
            while right + word_length <= total_length:
                word = s[right: right + word_length]
                right += word_length

                # If the word is not in our word_count, reset window
                if word not in word_count:
                    left = right
                    window_word_count.clear()
                    total_matched_words = 0
                    continue

                # Increase the count for the new word in our window
                window_word_count[word] += 1
                total_matched_words += 1

                # If there are more instances of the word than needed, shrink window
                while window_word_count[word] > word_count[word]:
                    word_to_remove = s[left: left + word_length]
                    left += word_length
                    window_word_count[word_to_remove] -= 1
                    total_matched_words -= 1

                # If the window contains exactly 'num_words' words, we found a substring starting at 'left'
                if total_matched_words == num_words:
                    start_indices.append(left)

        return start_indices

```

Java Solution

```java
class Solution {

    public List<Integer> findSubstring(String s, String[] words) {
        Map<String, Integer> wordCount = new HashMap<>();

        // Create and populate a map with the count of each unique word
        for (String word : words) {
            wordCount.merge(word, 1, Integer::sum);
        }

        int strLength = s.length(), numOfWords = words.length;
        int wordLength = words[0].length(); // Assume all words are the same length
        List<Integer> indices = new ArrayList<>();

        // Iterate over all possible word start indices to check for valid substrings
        for (int i = 0; i < wordLength; ++i) {
            Map<String, Integer> currentCount = new HashMap<>();
            int left = i, right = i;
            int totalWords = 0;

            // Expand the window to the right, adding words into current window count
            while (right + wordLength <= strLength) {
                String sub = s.substring(right, right + wordLength);
                right += wordLength;

                // If the word is not in the original word list, reset the window
                if (!wordCount.containsKey(sub)) {
                    currentCount.clear();
                    left = right;
                    totalWords = 0;
                    continue;
                }

                // Increase the count for the current word in the window
                currentCount.merge(sub, 1, Integer::sum);
                ++totalWords;

                // If a word count exceeds its count in wordCount, reduce from left side
                while (currentCount.get(sub) > wordCount.get(sub)) {
                    String removed = s.substring(left, left + wordLength);
                    left += wordLength;
                    currentCount.merge(removed, -1, Integer::sum);
                    --totalWords;
                }

                // If the total words reached the number of words, a valid substring is found
                if (totalWords == numOfWords) {
                    indices.add(left);
                }
            }
        }
        return indices;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function searches for all starting indices of substring(s) in 's' that is a concatenation of each word in 'words' exactly
    vector<int> findSubstring(string s, vector<string>& words) {
        // Count the frequency of each word in 'words'.
        unordered_map<string, int> wordCount;
        for (auto& word : words) {
            ++wordCount[word];
        }

        int stringSize = s.size(), wordCountSize = words.size(), wordSize = words[0].size();
        vector<int> substrIndices;

        // Iterate over the string 's'.
        for (int i = 0; i < wordSize; ++i) {
            unordered_map<string, int> windowCount;
            int left = i, right = i;
            int totalCount = 0;

            // Slide a window over the string 's'.
            while (right + wordSize <= stringSize) {
                string currentWord = s.substr(right, wordSize);
                right += wordSize;

                // Skip the current segment if the word is not in 'words'.
                if (!wordCount.count(currentWord)) {
                    windowCount.clear();
                    left = right;
                    totalCount = 0;
                    continue;
                }

                // Update the count for the current word in the window.
                ++windowCount[currentWord];
                ++totalCount;

                // If there are more occurrences of 'currentWord' in the window than in 'words', remove from the left.
                while (windowCount[currentWord] > wordCount[currentWord]) {
                    string wordToRemove = s.substr(left, wordSize);
                    left += wordSize;
                    --windowCount[wordToRemove];
                    --totalCount;
                }

                // If the total count of words match and all words frequencies are as expected, add to result.
                if (totalCount == wordCountSize) {
                    substrIndices.push_back(left);
                }
            }
        }

        return substrIndices;
    }
};

```

Typescript Solution

```typescript
function findSubstring(s: string, words: string[]): number[] {
    // Create a map to store the frequency of words.
    const wordCountMap: Map<string, number> = new Map();
    // Populate the word frequency map.
    for (const word of words) {
        wordCountMap.set(word, (wordCountMap.get(word) || 0) + 1);
    }
    const stringLength: number = s.length;
    const wordArrayLength: number = words.length;
    const wordLength: number = words[0].length;
    const indices: number[] = [];

    // Iterate through the string in increments of word length
    for (let i = 0; i < wordLength; ++i) {
        const tempCountMap: Map<string, number> = new Map();
        let left = i;
        let right = i;
        let matchedWordCount = 0;

        // Scan the string in chunks the size of the words' length
        while (right + wordLength <= stringLength) {
            const currentWord = s.slice(right, right + wordLength);
            right += wordLength;

            // Skip the word if it's not in the frequency map
            if (!wordCountMap.has(currentWord)) {
                tempCountMap.clear();
                left = right;
                matchedWordCount = 0;
                continue;
            }

            // Update the temporary count map
            tempCountMap.set(currentWord, (tempCountMap.get(currentWord) || 0) + 1);
            ++matchedWordCount;

            // If the current word has been seen more times than it is present in words array, slide the window to the right
            while ((tempCountMap.get(currentWord)! - wordCountMap.get(currentWord)!) > 0) {
                const wordToLeft = s.slice(left, left + wordLength);
                tempCountMap.set(wordToLeft, tempCountMap.get(wordToLeft)! - 1);
                left += wordLength;
                --matchedWordCount;
            }

            // Check if all words match; if so, add to results
            if (matchedWordCount === wordArrayLength) {
                indices.push(left);
            }
        }
    }

    return indices;
}

```

Time and Space Complexity

The time complexity of the given code is `O(m * k)` where `m` is the length of the string `s` and `k` is the length of each word within the words list. This stems from the fact that we iterate over the string `s` in increments of `k` for loops starting at each of the first `k` characters.

The space complexity is `O(n * k)` where `n` is the number of words in the given list `words` and `k` is the length of each word. This is due to two counters `cnt` and `cnt1` storing, at most, `n` different words of `k` length each, along with the list `ans` that in the worst case could store up to `m / k` starting indices if every substring is a valid concatenation.