1023. Camelcase Matching

Medium | Trie | Two Pointers | String | String Matching | Leetcode Link

Problem Description

The problem gives us an array of strings called `queries` and a single string called `pattern`. Our goal is to determine which strings in the `queries` array match the `pattern`. We say that a query string matches the pattern if it's possible to insert lowercase English letters into the pattern such that after insertion, the pattern is equal to the query. Importantly, these insertions can happen at any index within the pattern, but we cannot add any new characters into the query string. The output of the problem is a boolean array `answer` where each element `answer[i]` is `true` if `queries[i]` matches the pattern, otherwise `false`.

For example, if the query is `"FoBar"` and the pattern is `"FB"`, we can see that by adding `"o"` between `"F"` and `"B"` and appending `"ar"` after `"B"`, the pattern matches the query (after the insertions, the pattern would be `"FoBar"`). Therefore, the answer for this query would be `true`.

Intuition

The solution involves defining a match-checking function that uses two pointers to traverse and compare characters of the `queries[i]` and `pattern` one by one. By using an iterative process, we can validate if we can morph the pattern into each query.

The intuition for the solution is based on a couple of observations:

1. The characters in both `queries[i]` and `pattern` must match in order, and the matching characters in `queries[i]` must be in the same positions as in `pattern`. Uppercase letters in the query are "anchors" that must be present and in the correct order in the pattern for a match to be possible.
2. Lowercase letters in `queries[i]` can be skipped because we're allowed to insert lowercase letters into the pattern. However, uppercase letters can't be skipped or inserted because they define the structure of the pattern that must be followed.

This allows us to implement a pointer-based approach:

- Initialize two pointers, `i` and `j`, to traverse `queries[i]` and `pattern`, respectively.
- Iterate over `queries[i]` with pointer `i`, and for each character, check if it matches the `j`th character of `pattern`.
- If the characters match, move both pointers forward.
- If the characters do not match, and the `i`th character of `queries[i]` is lowercase, we can attempt to skip it by moving pointer `i` forward.
- If we encounter an uppercase letter that doesn't match or we reach the end of `queries[i]` without fully traversing the pattern, that query does not match the pattern.
- After pointer `j` has traversed the pattern, we need to ensure that all remaining characters in `queries[i]` are lowercase; otherwise, it means there are extra uppercase letters that make the query not match the pattern.

The match-checking function embodied the above logic, consistently applying it to every query in the array, thus creating the desired boolean output array.

The time complexity is linear with respect to the total number of characters in all queries and the length of the pattern, i.e., effectively O(n*m), where n is the number of queries and m is the length of the longest query string.

Solution Approach

The solution to the given problem involves creating a helper function, `check(s, t)`, which is designed to determine if the string `s` (a query) matches the string `t` (the pattern). This function is crucial as it encapsulates the logic for matching based on the rules outlined in the problem description.

Helper Function: `check(s, t)`

This function uses the two-pointer technique, which is a common pattern used in string manipulation algorithms. Here's a breakdown of the `check` function implementation using the `s` and `t` variables:

1. Initialize two pointers: `i` starts at the beginning of `s` while `j` starts at the beginning of `t`.
2. Iterate through `s` using the pointer `i`:
   - If `s[i]` is a lowercase letter that does not match `t[j]`, increment `i` to skip it. This step embodies the rule that allows the insertion of lowercase letters.
   - If `s[i]` matches `t[j]`, increment both pointers `i` and `j` to check the next characters since a match dictates we move in both `queries` and `pattern`.
   - If the characters do not match and `s[i]` is not a lowercase letter, return `false` as we can't insert uppercase letters or skip non-matching uppercase letters.
3. Once `j` has reached the end of `t`, we check the remainder of `s` to ensure all leftover characters are lowercase. If we encounter an uppercase letter or `i` hasn't yet reached the end of `s`, we return `false` as the additional characters can't be part of the pattern.
4. If `i` has reached the end of `s`, we return `true`, indicating a successful match.

Main Logic

The main part of the solution consists of iterating over each string in the `queries` array and applying the `check` function:

1. Use a list comprehension to iterate over all query strings in `queries`.
2. For each query, call the `check` function with the query and the pattern.
3. Convert the result of the `check` function into boolean values and store them in the final output list.

Time Complexity Analysis

The overall time complexity of the solution is `O(n * m)`, where `n` is the number of query strings in the `queries` array, and `m` is the length of the longest query string. This is because we must potentially traverse each character in each query string in the worst-case scenario, consuming linear time relative to the length of each individual query.

In summary, this approach is simple yet effective. It leverages the two-pointer pattern to navigate two strings simultaneously and verifies compliance with the pattern-matching rules. The algorithm is sharp in that it halts and decides as soon as a non-compliance is detected, ensuring optimal performance.

Example Walkthrough

Let's walk through an example to illustrate the solution approach described above.

Suppose we have the following `queries` array and `pattern`:

```
queries = ["After", "AFter", "AFilter", "BFilter"]
pattern = "AF"
```

To determine which query strings match the pattern, we apply the `check` function to each one:

1. "After":
   - Initialize pointers `i=0` and `j=0`. Since `queries[i] = 'A'` and `pattern[j] = 'A'`, they match, and we increment both pointers.
   - Now `i=1` and `j=1`, and we have `queries[i] = 'f'` and `pattern[j] = 'F'`. They do not match because one is lowercase and the other is uppercase, but since `queries[i]` is lowercase, we can skip it by incrementing `i`.
   - With `i=2` and `j=1`, we now have `queries[i] = 't'` and `pattern[j] = 'F'`. The letters do not match, so we increment `i` as 't' is lowercase.
   - At `i=3`, `j=1`, we have `queries[i] = 'e'`, still lowercase. We skip it.
   - Now `i=4`, `j=1`, and `queries[i] = 'r'`. Another lowercase to skip.
   - Since `j` has not moved and `i` reached the end of `queries[i]` without matching 'F' from `pattern`, we return `false`.
2. "AFter":
   - We set `i=0` and `j=0`. They match ('A' with 'A'), so both pointers increment.
   - At `i=1`, `j=1`, `queries[i] = 'F'` and `pattern[j] = 'F'`, they match, so increment both pointers again.
   - Now `i=2`, and since `j` has reached the end of `pattern`, we only need to check that the remaining characters in `queries[i]` are lowercase.
   - As 't', 'e', and 'r' are all lowercase, we move `i` to the end of the string.
   - Given that all conditions are satisfied, `i` has reached the end of `s`, and `j` has reached the end of `t`, we return `true`.
3. "AFilter":
   - Both pointers start at the beginning. `i=0`, `j=0`, and 'A' matches 'A'. Both increment.
   - At `i=1`, `j=1`, 'F' matches 'F'. Increment both again.
   - `i=2` now points to 'i' which is lowercase, so we can skip it.
   - Skipping all lowercase 'l', 't', 'e', 'r' by incrementing `i` each time.
   - Having processed all characters of `pattern` and ensured no uppercase letters in the remaining `queries[i]`, we return `true`.
4. "BFilter":
   - Start with `i=0` and `j=0`. We find that 'B' doesn't match 'A' in `pattern` and it's uppercase, which means we cannot skip it or change the pattern to match it.
   - Immediately we return `false`, as it's not possible for this query to match the pattern.

The resulting list based on the `check` function would be `[false, true, true, false]`. This methodically applies the rules from the problem description to each query against the pattern. By utilizing this process, the `check` function can efficiently validate pattern matching for each string in the `queries` array.

Python Solution

```python
class Solution:
    def camelMatch(self, queries: List[str], pattern: str) -> List[bool]:
        # Helper function to check if the query matches the pattern.
        def matches_pattern(query: str, pattern: str) -> bool:
            query_length = len(query)
            pattern_length = len(pattern)
            pattern_index = query_index = 0

            # Traverse the pattern characters.
            while pattern_index < pattern_length:
                # Skip lower case characters from query that are not a part of the pattern.
                while query_index < query_length and query[query_index] != pattern[pattern_index] and query[query_index].islower():
                    query_index += 1

                # If query runs out of characters or does not match pattern character, return False.
                if query_index == query_length or query[query_index] != pattern[pattern_index]:
                    return False

                # Move to the next character in both query and pattern.
                query_index += 1
                pattern_index += 1

            # All remaining characters in the query should be lowercase.
            while query_index < query_length and query[query_index].islower():
                query_index += 1

            # True if the end of the query has been reached, False otherwise.
            return query_index == query_length

        # Use list comprehension to apply the helper function to all queries in the input list.
        return [matches_pattern(query, pattern) for query in queries]

```

Java Solution

```java
class Solution {
    // Function to check the list of strings 'queries' match the camel case pattern 'pattern'
    public List<Boolean> camelMatch(String[] queries, String pattern) {
        List<Boolean> matches = new ArrayList<>(); // List to hold the results
        // Iterate over each query
        for (String query : queries) {
            // Add the result of matching each query with the pattern to the matches list
            matches.add(isMatch(query, pattern));
        }
        return matches;
    }

    // Helper method to check if a single string 'query' matches the camel case 'pattern'
    private boolean isMatch(String query, String pattern) {
        int queryLength = query.length(); // Length of query string
        int patternLength = pattern.length(); // Length of the pattern
        int queryIndex = 0, patternIndex = 0; // Indices to track position within query and pattern

        // Iterate through the characters in 'pattern'
        while (patternIndex < patternLength) {
            // Advance in 'query' while current character does not match the pattern character
            // and it's a lowercase letter
            while (queryIndex < queryLength && query.charAt(queryIndex) != pattern.charAt(patternIndex)
                    && Character.isLowerCase(query.charAt(queryIndex))) {
                queryIndex++;
            }
            // If the end of 'query' is reached or characters do not match, the query doesn't follow the pattern
            if (queryIndex == queryLength || query.charAt(queryIndex) != pattern.charAt(patternIndex)) {
                return false;
            }
            // Move to the next character in both 'query' and 'pattern'
            queryIndex++;
            patternIndex++;
        }

        // After the end of 'pattern', all remaining characters in 'query' must be lowercase for it to be a match
        while (queryIndex < queryLength && Character.isLowerCase(query.charAt(queryIndex))) {
            queryIndex++;
        }

        // Return true if the end of 'query' is reached, otherwise false
        return queryIndex == queryLength;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to determine if each query matches the given pattern
    vector<bool> camelMatch(vector<string>& queries, string pattern) {
        vector<bool> answers;
        // Lambda function to check if a single query matches the pattern
        auto matchesPattern = [](const string& query, const string& pattern) {
            int queryLength = query.size();
            int patternLength = pattern.size();
            int queryIndex = 0, patternIndex = 0;

            // Loop through each character in the pattern
            while (patternIndex < patternLength) {
                // Advance through the query string to find the match for the current pattern character
                while (queryIndex < queryLength && query[queryIndex] != pattern[patternIndex] && islower(query[queryIndex])) {
                    ++queryIndex;
                }

                // Check if query ended or characters don't match
                if (queryIndex == queryLength || query[queryIndex] != pattern[patternIndex]) {
                    return false;
                }

                // Move to the next character in both the query and the pattern
                ++queryIndex;
                ++patternIndex;
            }

            // All characters in the pattern are matched, check if the remaining characters in the query are all lowercase
            while (queryIndex < queryLength && islower(query[queryIndex])) {
                ++queryIndex;
            }

            // If we reached the end of the query, all checks are passed
            return queryIndex == queryLength;
        };

        // Iterate through all the queries to check against the pattern
        for (auto& query : queries) {
            answers.push_back(matchesPattern(query, pattern));
        }

        return answers;  // Return a vector with the results for each query
    }
};

```

Typescript Solution

```typescript
// Function to match camel case patterns within a set of query strings
function camelMatch(queries: string[], pattern: string): boolean[] {
    // Helper function to check if a single query matches the camel case pattern
    const checkPattern = (query: string, pattern: string): boolean => {
        const queryLength = query.length;
        const patternLength = pattern.length;
        let queryIndex = 0;
        let patternIndex = 0;
        // Iterate through both query and pattern
        for (; patternIndex < patternLength; ++queryIndex, ++patternIndex) {
            // Skip lowercase characters in query that do not match the current pattern character
            while (queryIndex < queryLength &&
                   query[queryIndex] !== pattern[patternIndex] &&
                   query.charCodeAt(queryIndex) >= 97) {  // ASCII 97 = 'a'
                ++queryIndex;
            }
            // Pattern character does not match, or query ended before pattern
            if (queryIndex === queryLength || query[queryIndex] !== pattern[patternIndex]) {
                return false;
            }
        }
        // Skip all trailing lowercase characters in the query
        while (queryIndex < queryLength && query.charCodeAt(queryIndex) >= 97) {
            ++queryIndex;
        }
        // If all characters of query are checked, it's a match
        return queryIndex == queryLength;
    };

    // Array to store results for each query
    const results: boolean[] = [];
    // Check each query and add the result to the results array
    for (const query of queries) {
        results.push(checkPattern(query, pattern));
    }
    return results;
}

```

Time and Space Complexity

The given code implements a function to check if each query in a list of queries matches a given camel case pattern.

Time Complexity

The time complexity of the function is determined as follows:

- The `check` function goes through each character in the query string `s` and the pattern string `t` at most once. The inner while loop runs as long as there are lowercase characters in `s` that are not in `t`, and the outer while loop runs as long as there are characters in `t`.
- In the worst case, each character of the query string will be checked exactly once against the pattern. This results in a linear complexity in terms of the length of the query string.
- The `check` function is called once for each query in the `queries` list.

If the average length of the queries is `m` and the length of the pattern is `n`, and there are `k` queries, then the time complexity is `O(k * (m + n))`, since for every query we have a separate traversal through the pattern string and the query string.

Space Complexity

The space complexity of the function is considered as follows:

- The `check` function uses a constant amount of extra space since it only uses a few integer variables to keep track of the indices.
- The output list has the same length as the number of queries. This is where the space is used. However, this is not counted towards the extra space complexity as this is required for the output of the function.

So, the space complexity of the function is `O(1)` (constant space complexity) for the auxiliary space.