Problem Description

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"

The problem gives us an array of strings called queries and a single string called pattern. Our goal is to determine which strings in

would be true. Intuition

after "B", the pattern matches the query (after the insertions, the pattern would be "FoBar"). Therefore, the answer for this query

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:

forward.

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

The solution involves defining a match-checking function that uses two pointers to traverse and compare characters of 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 jth character of pattern. If the characters match, move both pointers forward.

• If the characters do not match, and the ith character of queries[i] is lowercase, we can attempt to skip it by moving pointer i

- 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.
- otherwise, it means there are extra uppercase letters that make the query not match the pattern.

of the check function implementation using the s and t variables:

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

After pointer j has traversed the pattern, we need to ensure that all remaining characters in queries [i] are lowercase;

- 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

This function uses the two-pointer technique, which is a common pattern used in string manipulation algorithms. Here's a breakdown

o 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

outlined in the problem description. Helper Function: check(s, t)

query) matches the string t (the pattern). This function is crucial as it encapsulates the logic for matching based on the rules

insertion of lowercase letters. o 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.

2. Iterate through s using the pointer 1:

o 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.

Main Logic

4. If i has reached the end of s, we return true, indicating a successful match.

1. Use a list comprehension to iterate over all query strings in queries.

1. Initialize two pointers: i starts at the beginning of s while j starts at the beginning of t.

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.

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

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 worstcase 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

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

Let's walk through an example to illustrate the solution approach described above. Suppose we have the following queries array and pattern:

is lowercase.

are lowercase.

1. "After":

2. "AFter":

Example Walkthrough

Time Complexity Analysis

is detected, ensuring optimal performance.

1 queries = ["After", "AFter", "AFilter", "BFilter"]

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'

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.

At i=1, j=1, 'F' matches 'F'. Increment both again.

change the pattern to match it.

matching for each string in the queries array.

query_length = len(query)

pattern_length = len(pattern)

pattern_index = query_index = 0

Traverse the pattern characters.

query_index += 1

return False

query_index += 1

query_index += 1

return query_index == query_length

matches.add(isMatch(query, pattern));

private boolean isMatch(String query, String pattern) {

++queryIndex;

return false;

return queryIndex == queryLength;

++queryIndex;

++queryIndex;

for (auto& query : queries) {

++patternIndex;

// Check if query ended or characters don't match

// Move to the next character in both the query and the pattern

while (queryIndex < queryLength && islower(query[queryIndex])) {</pre>

// If we reached the end of the query, all checks are passed

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

// Iterate through all the queries to check against the pattern

answers.push_back(matchesPattern(query, pattern));

// Function to match camel case patterns within a set of query strings

function camelMatch(queries: string[], pattern: string): boolean[] {

if (queryIndex == queryLength || query[queryIndex] != pattern[patternIndex]) {

// All characters in the pattern are matched, check if the remaining characters in the query are all lowercase

int queryLength = query.length(); // Length of query string

int patternLength = pattern.length(); // Length of the pattern

pattern_index += 1

while pattern_index < pattern_length:</pre>

• 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.

def matches_pattern(query: str, pattern: str) -> bool:

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

• 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]

Both pointers start at the beginning. i=0, j=0, and 'A' matches 'A'. Both increment.

 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":

Since j has not moved and i reached the end of queries[i] without matching 'F' from pattern, we return false.

 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

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

Python Solution class Solution: def camelMatch(self, queries: List[str], pattern: str) -> List[bool]: # Helper function to check if the query matches the pattern.

Move to the next character in both query and pattern.

All remaining characters in the query should be lowercase.

while query_index < query_length and query[query_index].islower():</pre>

True if the end of the query has been reached, False otherwise.

// Add the result of matching each query with the pattern to the matches list

int queryIndex = 0, patternIndex = 0; // Indices to track position within query and pattern

// Helper method to check if a single string 'query' matches the camel case 'pattern'

Skip lower case characters from query that are not a part of the pattern.

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]:

Immediately we return false, as it's not possible for this query to match the pattern.

29 # Use list comprehension to apply the helper function to all queries in the input list. 30 31 return [matches_pattern(query, pattern) for query in queries] 32

while query_index < query_length and query[query_index] != pattern[pattern_index] and query[query_index].islower():</pre>

```
1 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) {
```

return matches;

Java Solution

9

10

12

13

14

15

16

18

19

20

21

22

24

25

26

27

28

10

11

12

13

14

15

16

17

18

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

30

41

43

45

48

50

49 };

```
19
           // Iterate through the characters in 'pattern'
20
           while (patternIndex < patternLength) {</pre>
21
               // Advance in 'query' while current character does not match the pattern character
               // and it's a lowercase letter
22
23
               while (queryIndex < queryLength && query.charAt(queryIndex) != pattern.charAt(patternIndex)</pre>
24
                        && Character.isLowerCase(query.charAt(queryIndex))) {
25
                    queryIndex++;
26
27
               // 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)) {
28
29
                    return false;
30
31
               // Move to the next character in both 'query' and 'pattern'
32
                queryIndex++;
33
                patternIndex++;
34
35
36
           // After the end of 'pattern', all remaining characters in 'query' must be lowercase for it to be a match
37
           while (queryIndex < queryLength && Character.isLowerCase(query.charAt(queryIndex))) {</pre>
38
                queryIndex++;
39
40
           // Return true if the end of 'query' is reached, otherwise false
41
           return queryIndex == queryLength;
42
43
44
45
C++ Solution
 1 #include <vector>
 2 #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
10
           auto matchesPattern = [](const string& query, const string& pattern) {
11
                int queryLength = query.size();
12
13
                int patternLength = pattern.size();
                int queryIndex = 0, patternIndex = 0;
14
15
               // Loop through each character in the pattern
16
               while (patternIndex < patternLength) {</pre>
17
                    // Advance through the query string to find the match for the current pattern character
18
                    while (queryIndex < queryLength && query[queryIndex] != pattern[patternIndex] && islower(query[queryIndex])) {</pre>
19
```

Typescript Solution

```
// 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) {</pre>
10
               // Skip lowercase characters in query that do not match the current pattern character
               while (queryIndex < queryLength &&</pre>
13
                      query[queryIndex] !== pattern[patternIndex] &&
                      query.charCodeAt(queryIndex) >= 97) { // ASCII 97 = 'a'
14
                   ++queryIndex;
15
16
               // Pattern character does not match, or query ended before pattern
17
               if (queryIndex === queryLength || query[queryIndex] !== pattern[patternIndex]) {
19
                   return false;
20
21
           // 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
33
       for (const query of queries) {
           results.push(checkPattern(query, pattern));
34
35
       return results;
36
37 }
38
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
```

28 29

• 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.

complexity in terms of the length of the guery string. The check function is called once for each query in the queries list.

The time complexity of the function is determined as follows:

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 0(k * (m + n)), since for every query we have a separate traversal through the pattern string and the query string.

In the worst case, each character of the query string will be checked exactly once against the pattern. This results in a linear

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.