890. Find and Replace Pattern

Medium Array Hash Table String

Leetcode Link

Problem Description

The task is to find all words from a given list that match a specific pattern. A word matches the pattern if we can map each letter in the pattern to a different letter (forming a bijection) such that when we replace the letters in the pattern based on this mapping, we get the word from the list.

For example, if the pattern is "abb" and one of the words is "cdd", there is a match because we can map 'a' to 'c' and 'b' to 'd'. However, if another word in the list is "cde", it does not match the "abb" pattern because 'a' maps to 'c', but 'b' would have to map to both 'd' and 'e' for the word to match, which is not allowed since a bijection requires each letter to map to a unique letter.

The problem necessitates that no two different letters in the pattern can map to the same letter in the words, maintaining the uniqueness of the bijection.

Intuition

The solution approach involves checking for each word if there's a bijection with the pattern. This is done by creating two mappings: one for the word and one for the pattern. As we iterate through the characters of a word and the pattern simultaneously, we can establish a correspondence between the two.

The intuition is that if a word follows the pattern, each occurrence of a particular character in the pattern should be mapped to the same character in the word, and vice versa. For instance, pattern "abb" and word "xyz" do not match because 'a' is mapped to 'x', but 'b' has to be mapped to both 'y' and 'z', which violates the rules.

By utilizing two arrays indexed by the ASCII values of the characters, we can store the order in which each character appears. If at any point we find that the current mapping does not match (i.e., the current character of the word was previously mapped with a different character of the pattern or vice versa), we can conclude that the word does not match the pattern.

Therefore, we compare the mapping of the characters at each index. If they do not match, it means this particular word does not follow the pattern. If we reach the end without discrepancies, the word matches the pattern.

This comparison continues for each word in the list, and all matching words are collected and returned.

Solution Approach

In the provided solution, the objective was to compare a given word with a pattern. The essence of the solution lies in the algorithm that maps each character in the pattern to a character in the word and checks for consistency throughout the string.

Here is a step-by-step process of how the `Solution` class implements this algorithm:

1. A helper function `match` is defined, which takes two strings `s` (the word from the list) and `t` (the pattern) and determines whether `s` matches `t`.
2. Two arrays of integers, `m1` and `m2`, are initialized with a length of 128. This size is chosen to cover all standard ASCII values, ensuring that each character from the word and the pattern can be mapped to an index in these arrays. Initially, all elements of `m1` and `m2` are set to `0`.
3. The function then iterates through the characters of both the pattern and the word simultaneously using a `zip` function in Python.
4. For every pair of characters `(a, b)` from `s` and `t`, their ASCII values are used as indices in `m1` and `m2`.
5. On each iteration, indexed by `i` (starting at 1 to avoid the 0 initial values of `m1` and `m2`), it checks if `m1[ord(a)]` is not equal to `m2[ord(b)]`. If they are not equal, it means that the current character mapping is not consistent with previous mappings, and hence the function returns `False`.
6. If the condition is satisfied (i.e., the characters match the mapping), both mappings are updated with the current index `i`. This step marks the position of the latest mapping and serves to maintain consistency. If the same characters in `s` and `t` occur later, they must result in the same mapping index, or otherwise, they do not follow the pattern and the function would return `False`.
7. After completing the loop, if no inconsistencies are found, the function returns `True`, signifying that the word `s` matches the pattern `t`.
8. The main function in the `Solution` class uses a list comprehension that goes through all the words in the given list, applying the `match` function to each word along with the pattern.
9. Only those words for which the `match` function returns `True` are included in the final list that the main function returns. This final list represents all the words from the input that match the given pattern.

The solution makes use of array mapping and iteration, which are fundamental in allowing the comparison of strings with patterns. This method of creating a bijection through array indexing is efficient and straightforward, as it avoids complex data structures and makes use of simple arrays, leveraging their direct access property.

Example Walkthrough

Consider the pattern "abb" and the list of words `["xyz", "zzy", "yzz", "azb"]`. Let's walk through the solution approach with these examples and see which words match the pattern.

1. Take the first word "xyz" and create two arrays, `m1` and `m2`, initialized to `0`.
2. Start iterating through "xyz" and "abb" simultaneously. Map the ASCII values of 'x' to 'a', 'y' to 'b', and 'z' to 'b' using the arrays `m1` and `m2`.
3. When the first characters 'x' and 'a' are encountered, `m1[ord('x')]` and `m2[ord('a')]` are set to `1`.
4. Next, compare 'y' of "xyz" to 'b' of "abb". Since 'b' corresponds to 'y' and it's the first encounter, set `m1[ord('y')]` and `m2[ord('b')]` to `2`.
5. Lastly, 'z' is compared to 'b'. Since 'b' is already mapped to 'y', `m1[ord('z')]` is not equal to `m2[ord('b')]`, which is `2`. So, the mapping is inconsistent, and "xyz" does not match "abb".

Repeat the process for the remaining words:

1. "zzy":
   - 'z' maps to 'a'.
   - 'z' still maps to 'b' from the previous mapping.
   - 'y' maps to 'b', but 'b' is already mapped to 'z', so again, inconsistency found. "zzy" doesn't match "abb".
2. "yzz":
   - 'y' maps to 'a'.
   - First 'z' maps to 'b'.
   - Second 'z' also maps to 'b' (consistent with previous mapping).
   - Consistency is maintained, so "yzz" matches "abb".
3. "azb":
   - 'a' maps to 'a'.
   - 'z' maps to 'b'.
   - 'b' maps to 'b', but since there's a prior mapping of 'z' to 'b', this creates inconsistency.
   - "azb" does not match "abb".

In conclusion, among the words provided, only "yzz" matches the pattern "abb" using the bijection approach described in the solution.

Python Solution

```python
from typing import List

class Solution:
    def find_and_replace_pattern(self, words: List[str], pattern: str) -> List[str]:
        # Inner function to determine if the given string s matches the pattern t
        def is_match(s, t):
            # Initialize two maps for characters in s and t
            # Each map stores the index of the character from s/t encountered in the string
            map_s, map_t = [0] * 128, [0] * 128
            # Enumerate through the pair(s) from s and t
            for index, (char_s, char_t) in enumerate(zip(s, t), 1):
                # If the mapped values are not equal, then s does not match the pattern t
                if map_s[ord(char_s)] != map_t[ord(char_t)]:
                    return False
                # Update the mapping for the current character in both s and t to the current index
                map_s[ord(char_s)] = map_t[ord(char_t)] = index
            # If the loops completes without returning False, s matches the pattern t
            return True

        # List comprehension to find and collect words that match the pattern
        # It includes the is_match function on each word with the given pattern
        return [word for word in words if is_match(word, pattern)]

# Example of usage:
# sol = Solution()
# matching_words = sol.find_and_replace_pattern(["abc","deq","mee","aqq","dkd","ccc"], "abb")
# print(matching_words) # Output would be ["mee","aqq"]
```

Java Solution

```java
class Solution {

    /**
     * Filters the array of words, selecting only those that match the given pattern.
     *
     * @param words  an array of words to be filtered based on the pattern
     * @param pattern the pattern to which words are to be matched
     * @return a list of words that match the pattern
     */
    public List<String> findAndReplacePattern(String[] words, String pattern) {
        List<String> matchedWords = new ArrayList<>();
        // Iterate through each word in the array
        for (String word : words) {
            // If the word matches the pattern, add it to the result list
            if (doesWordMatchPattern(word, pattern)) {
                matchedWords.add(word);
            }
        }
        return matchedWords;
    }

    /**
     * Checks if the given word matches the given pattern.
     *
     * @param word    the word to match against the pattern
     * @param pattern the pattern to be matched with the word
     * @return true if the word matches the pattern, false otherwise
     */
    private boolean doesWordMatchPattern(String word, String pattern) {
        // Arrays to keep track of character mappings for both the word and the pattern
        int[] wordToPatternMapping = new int[128];
        int[] patternToWordMapping = new int[128];

        // Iterate through characters of the word and the pattern simultaneously
        for (int i = 0; i < word.length(); ++i) {
            char wordChar = word.charAt(i);
            char patternChar = pattern.charAt(i);

            // If current mapping does not match, return false (patterns do not match)
            if (wordToPatternMapping[wordChar] != patternToWordMapping[patternChar]) {
                return false;
            }

            // Update the mappings (using i+1 to avoid default 0 value of int array)
            wordToPatternMapping[wordChar] = i + 1;
            patternToWordMapping[patternChar] = i + 1;
        }
        // If all characters matched the pattern, return true
        return true;
    }
}
```

C++ Solution

```cpp
#include <string>
#include <vector>

class Solution {
public:
    // findAndReplacePattern finds all the words that match the given pattern.
    // @param words: a list of strings representing the words to be matched.
    // @param pattern: a string representing the pattern to match.
    // @return: a list of strings that match the pattern.
    vector<string> findAndReplacePattern(vector<string>& words, string pattern) {
        vector<string> matchingWords;  // Will hold the words that match the pattern.

        // Lambda function to check if a word matches the pattern.
        auto isMatch = [](string word, string pattern) {
            int wordToPatternMapping[128] = {0}; // Maps characters in word to pattern.
            int patternToWordMapping[128] = {0}; // Maps characters in pattern to word.

            // Loop through the characters in the words and pattern.
            for (int i = 0; i < word.size(); ++i) {
                // Check if current mapping does not match.
                if (wordToPatternMapping[word[i]] != patternToWordMapping[pattern[i]])
                    return false; // The words do not follow the same pattern so return false.

                // Update the mappings to reflect the most recent character mapping.
                wordToPatternMapping[word[i]] = i + 1;
                patternToWordMapping[pattern[i]] = i + 1;
            }
            // The full word matches the pattern.
            return true;
        };

        // Check each word in the list to see if it matches the pattern.
        for (const auto& word : words) {
            if (isMatch(word, pattern))
                matchingWords.emplace_back(word); // Add word to list if it matches the pattern.
        }

        return matchingWords; // Return the list of matching words.
    }
};
```

Typescript Solution

```typescript
// Function to find all words that match the given pattern
function findAndReplacePattern(words: string[], pattern: string): string[] {
    // Filter the words array to include only the words that match the pattern
    return words.filter(word => {
        // Initialize two maps to store the character to index mappings for the word and pattern
        const wordToPatternMap = new Map<string, number>();
        const patternToWordMap = new Map<string, number>();

        // Iterate over each character in the word
        for (let i = 0; i < word.length; i++) {
            // Check if there is a mismatch between the current mappings of the word and pattern
            if (wordToPatternMap.get(word[i]) !== patternToWordMap.get(pattern[i])) {
                // If a mismatch is found, exclude this word
                return false;
            }
            // Update the mappings with the current index
            wordToPatternMap.set(word[i], i);
            patternToWordMap.set(pattern[i], i);
        }

        // If all characters have been mapped successfully, include this word
        return true;
    });
}
```

Time and Space Complexity

The time complexity of the function `findAndReplacePattern` is $O(N * K)$, where N is the length of the `words` list and K is the length of each word (and the pattern). This is because for each of the N words, the function `match` is called, which iterates over the entire length K of the word and the pattern once.

The space complexity is $O(1)$ since we only use two fixed-size arrays `m1` and `m2` of size 128 (the number of ASCII characters). The size of these arrays does not scale with the input size, thus it is constant.