890. Find and Replace Pattern

String

to 'x', but 'b' has to be mapped to both 'y' and 'z', which violates the rules.

follow the pattern. If we reach the end without discrepancies, the word matches the pattern.

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

Hash Table

Problem Description

<u>Array</u>

Medium

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

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

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.

ntuition

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

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

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.

whether s matches t.

Python.

return False.

Example Walkthrough

m1 and m2.

the match function to each word along with the pattern.

final list represents all the words from the input that match the given pattern.

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.

A helper function match is defined, which takes two strings s (the word from the list) and t (the pattern) and determines

Two arrays of integers, m1 and m2, are initialized with a length of 128. This size is chosen to cover all standard ASCII values,

The function then iterates through the characters of both the pattern and the word simultaneously using a zip function in

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,

For every pair of characters (a, b) from s and t, their ASCII values are used as indices in m1 and m2.

and hence the function returns False. 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

After completing the loop, if no inconsistencies are found, the function returns True, signifying that the word s matches the

Only those words for which the match function returns True are included in the final list that the main function returns. This

- pattern t. The main function in the Solution class uses a list comprehension that goes through all the words in the given list, applying
- 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.

The solution makes use of array mapping and iteration, which are fundamental in allowing the comparison of strings with

these examples and see which words match the pattern. Take the first word "xyz" and create two arrays, m1 and m2, initialized to 0.

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

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

When the first characters 'x' and 'a' are encountered, m1[ord('x')] and m2[ord('a')] are set to 1. 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.

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".

"zzy":

"yzz":

∘ 'y' maps to 'a'.

∘ 'a' maps to 'a'.

∘ 'z' maps to 'b'.

from typing import List

def is match(s, t):

return True

Python

Java

class Solution {

/**

class Solution:

First 'z' maps to 'b'.

5.

∘ '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".

 Second 'z' also maps to 'b' (consistent with previous mapping). Consistency is maintained, so "yzz" matches "abb". "azb":

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

∘ 'b' maps to 'b', but since there's a prior mapping of 'z' to 'b', this creates inconsistency. "azb" does not match "abb".

Repeat the process for the remaining words:

solution. Solution Implementation

def find and replace pattern(self, words: List[str], pattern: str) -> List[str]:

Initialize two maps for characters in s and t

for index, (char s, char t) in enumerate(zip(s, t), 1):

map s[ord(char s)] = map t[ord(char t)] = index

List comprehension to find and collect words that match the pattern

if map s[ord(char_s)] != map_t[ord(char_t)]:

Enumerate through the pair(s) from s and t

map s. map t = [0] * 128, [0] * 128

return False

print(matching_words) # Output would be ["mee","agg"]

* @return a list of words that match the pattern

// Iterate through each word in the array

matchedWords.add(word);

for (String word : words) {

return matchedWords;

List<String> matchedWords = new ArrayList<>();

if (doesWordMatchPattern(word, pattern)) {

* @param pattern the pattern to which words are to be matched

public List<String> findAndReplacePattern(String[] words, String pattern) {

// If the word matches the pattern, add it to the result list

Inner function to determine if the given string s matches the pattern t

Each map stores the index of the character from s/t encountered in the string

If the mappings are not equal, then s does not match the pattern t

If the loops completes without returning False, s matches the pattern t

matching words = sol.find and replace pattern(["abc","deq","mee","aqq","dkd","ccc"], "abb")

Update the mapping for the current character in both s and t to the current index

It invokes 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()

/** * 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

```
* Checks if the given word matches the given pattern.
                   the word to match against the pattern
   * @param word
   * @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) {</pre>
            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++
#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 = [](const string& word, const 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) {</pre>
                // 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.

matchingWords.emplace_back(word); // Add word to list if it matches the pattern.

// Initialize two maps to store the character to index mappings for the word and pattern

// Update the mappings to reflect the most recent character mapping.

wordToPatternMapping[word[i]] = i + 1:

// The full word matches the pattern.

return true;

for (const auto& word : words) {

if (isMatch(word, pattern))

};

patternToWordMapping[pattern[i]] = i + 1;

// Check each word in the list to see if it matches the pattern.

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

const wordToPatternMap = new Map<string, number>();

const patternToWordMap = new Map<string, number>();

// Iterate over each character in the word

```
// 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) => {
```

};

TypeScript

```
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;
   });
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 mappings 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
```

The time complexity of the function findAndReplacePattern is O(N * K), where N is the length of the words list and K is the

print(matching_words) # Output would be ["mee","agg"]

List comprehension to find and collect words that match the pattern

The size of these arrays does not scale with the input size, thus it is constant.

return [word for word in words if is_match(word, pattern)]

It invokes the is match function on each word with the given pattern

matching words = sol.find and replace pattern(["abc","deq","mee","aqq","dkd","ccc"], "abb")

return True

Time and Space Complexity

Example of usage:

sol = Solution()

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 0(1) since we only use two fixed-size arrays m1 and m2 of size 128 (the number of ASCII characters).