

2301. Match Substring After Replacement

Hard Array Hash Table String String Matching

[Leetcode Link](#)

Problem Description

The given problem presents the task of determining whether we can make one string (**sub**) a substring of another string (**s**) by performing a series of character replacements according to a set of given mappings (**mappings**). The challenge lies in figuring out if, after performing zero or more of these allowed replacements, **sub** can be found as a contiguous sequence of characters within **s**.

Here's what we know about the problem:

- We're given two strings: **s** (the main string) and **sub** (the substring we want to match within **s**).
- mappings** is a 2D array of character pairs, where each pair indicates a permissible replacement (**old_i**, **new_i**), allowing us to replace **old_i** in **sub** with **new_i**.
- Each character in **sub** can be replaced only once, ensuring that we cannot keep changing the same character over and over again.
- A "substring" by definition is a contiguous sequence of characters - meaning the characters are adjacent and in order - within a string.

The goal is to return **true** if **sub** can be made into a substring of **s** or **false** otherwise.

Intuition

The solution approach can be envisioned in a few logical steps. Given that we must find if **sub** is a substring of **s** after some amount of character replacements (which are dictated by **mappings**), the straightforward strategy is to iterate through **s** and check every possible substring that is the same length as **sub**.

For each potential match, we iterate over the characters of **sub** and the corresponding characters of **s**. We then check two conditions for each character pair:

- The characters are the same, in which case no replacement is needed.
- The character from **s** is in the set of allowed replacements for the corresponding character in **sub** (as per the **mappings**).

We maintain a mapping dictionary **d** where each key is a character from **sub**, and each value is a set of characters that the key can be replaced with. This allows for quick lookup to check if a character from **s** is a valid substitute for a character in **sub**.

If all the characters in a potential match satisfy one of those conditions, we can conclude that it's possible to form **sub** from that segment of **s** and return **true**. If no matches are found after checking all possible segments of **s**, we return **false**.

Solution Approach

The implementation of the solution follows a straightforward algorithm which leverages a dictionary to store the mappings for quick lookup, and iterates through the main string **s** to find a possible match for **sub**. Here are the steps in detail:

- First, the algorithm uses a **defaultdict** from Python's **collections** module to create a dictionary (**d**) where each key will reference a **set** of characters. This **defaultdict** is a specialized dictionary that initializes a new entry with a default value (in this case, an empty set) if the key is not already present.
- It then populates this dictionary with the mappings. For each pair (old character, new character) given in the **mappings** list, the algorithm adds the new character to the **set** corresponding to the old character.
- The next step is to iterate through the main string **s**. The algorithm checks every substring of **s** that is the same length as **sub** which is done by using the range **for i in range(len(s) - len(sub) + 1)**. This loop will go over each starting point for a potential substring within **s**.
- For each starting index **i**, the algorithm performs a check to determine if the substring of **s** starting at **i** and ending at **i + len(sub)** can be made to match **sub** by replacements. This is done by using the **all()** function combined with a generator expression: **all(a == b or a in d[b] for a, b in zip(s[i : i + len(sub)], sub))**. This generator expression creates tuples of corresponding characters from the potential substring of **s** and from **sub**.
- Each tuple consists of a character **a** from **s** and a character **b** from **sub**. The expression checks if **a** equals **b** (no replacement needed), or if **a** is an acceptable replacement for **b** as per the dictionary **d**. If all character comparisons satisfy one of these conditions, then the substring of **s** starting at **i** is a match for **sub**, and the function returns **true**.
- If no such starting index **i** is found where **sub** can be matched in **s** after all necessary replacements, the algorithm reaches the end of the function and returns **false**, indicating that it is not possible to make **sub** a substring of **s** with the given character replacements.

The solution effectively combines data structure utilization (dictionary of sets for mapping) with the concept of sliding windows (iterating over substrings of **s** that are of the same length as **sub**).

Example Walkthrough

Let's illustrate the solution approach with a simple example:

Say we have the main string **s** as "dogcat", the target substring **sub** as "dag", and the **mappings** as [(**'a'**, **'o'**), (**'g'**, **'c'**)]. We are to determine if we can make **sub** a substring of **s** by using the given character replacements.

Following the solution approach:

- First, we create a **defaultdict** to store the mappings. It will look like this after populating it with the given **mappings**:

```
1 d = {}
2   'a': {'o'},
3   'g': {'c'}
4 }
```

This tells us that 'a' can be replaced with 'o', and 'g' can be replaced with 'c'.

- Next, we'll iterate through the string **s** to find potential substrings that match the length of **sub** (3 characters). The substrings of **s** we'll check are "dog", "ogc", and "gca".

- The first potential match is "dog". We compare it with "dag", the desired **sub**. We see that:
 - The first characters 'd' match.
 - The second characters 'o' and 'a' do not match, but since 'o' is in the set of allowed characters for 'a' in the dictionary **d**, this is an acceptable replacement.
 - The third characters 'g' and 'g' match.

Since all characters are either matching or can be replaced accordingly, this substring of **s** ("dog") can be made to match **sub**.

Thus, the function would return **true**, indicating that "dag" can indeed be made into a substring of "dogcat" by using the allowed character replacements.

Python Solution

```
1 from collections import defaultdict
2
3 class Solution:
4     def matchReplacement(self, string: str, substring: str, mappings: list[list[str]]) -> bool:
5         # Create a dictionary to store the mappings of characters that can be replaced.
6         replacement_dict = defaultdict(set)
7
8         # Populate the replacement dictionary with the mappings provided.
9         for original, replacement in mappings:
10             replacement_dict[original].add(replacement)
11
12         # Iterate over the string checking for each possible starting index of the substring.
13         for start_index in range(len(string) - len(substring) + 1):
14             # For each character pair (from the main string and the substring starting at the current index),
15             # check if they are the same or if the character from the main string can replace the one
16             # from the substring according to the provided mappings.
17             if all(char_from_string == char_from_substring or char_from_string in replacement_dict[char_from_substring]
18                   for char_from_string, char_from_substring in zip(string[start_index : start_index + len(substring)], substring)):
19                 # If all characters match (directly or through replacements),
20                 # the substring can be matched at this index, and True is returned.
21                 return True
22
23         # If no match was found, return False.
24         return False
25
```

Java Solution

```
1 class Solution {
2     public boolean matchReplacement(String s, String sub, char[][] mappings) {
3         // Create a map to hold characters from 'sub' and their allowable replacements.
4         Map<Character, Set<Character>> allowedReplacements = new HashMap<>();
5
6         // Populate the map with the mappings provided.
7         for (char[] mapping : mappings) {
8             // If the character from 'sub' is not in the map, add it.
9             // Then add the replacement character to the corresponding set.
10            allowedReplacements.computeIfAbsent(mapping[0], k -> new HashSet<>()).add(mapping[1]);
11        }
12
13        int stringLength = s.length(), subStringLength = sub.length();
14
15        // Try to match the sub string with a segment of string 's', considering replacements
16        for (int i = 0; i <= stringLength - subStringLength; ++i) {
17            boolean isMatch = true; // Flag to track if the segment matches with replacements
18
19            // Check each character in the segment
20            for (int j = 0; j < subStringLength && isMatch; ++j) {
21                char currentChar = s.charAt(i + j); // Current character from 's'
22                char subChar = sub.charAt(j); // Current character from 'sub'
23
24                // Check if the current character matches the sub character or is an allowed replacement
25                if (currentChar != subChar && !allowedReplacements.getOrDefault(subChar, Collections.emptySet()).contains(currentChar))
26                    isMatch = false; // If not, the segment does not match
27            }
28
29            // If a match is found, return true
30            if (isMatch) {
31                return true;
32            }
33        }
34
35        // If no match is found after checking all segments, return false
36        return false;
37    }
38 }
39
40
```

C++ Solution

```
1 #include <string>
2 #include <vector>
3 #include <unordered_map>
4 #include <unordered_set>
5
6 using namespace std;
7
8 class Solution {
9 public:
10     // Function to determine if 'sub' after replacements can match any substring in 's'
11     bool matchReplacement(string s, string sub, vector<vector<char>>& mappings) {
12         // Create a map to hold all replacement options for each character
13         unordered_map<char, unordered_set<char>> replacementDict;
14
15         // Iterate through the mappings and fill the replacement dictionary
16         for (auto& mapping : mappings) {
17             // Add the replacement (mapping[1]) for the key character (mapping[0])
18             replacementDict[mapping[0]].insert(mapping[1]);
19         }
20
21         int mainStrLength = s.size(); // length of the main string 's'
22         int subStrLength = sub.size(); // length of the substring 'sub'
23
24         // Iterate over 's' to check each possible starting position
25         for (let i = 0; i <= mainStrLength - subStrLength; ++i) {
26             bool matches = true; // Flag to determine if a match has been found
27
28             // Iterate over 'sub' to check character by character
29             for (int j = 0; j < subStrLength && matches; ++j) {
30                 char charMain = s[i + j]; // Character from main string 's'
31                 char charSub = sub[j]; // Corresponding character from 'sub'
32
33                 // Check if characters match or if there's a valid mapping in replacementDict
34                 if (charMain != charSub && !replacementDict[charSub].count(charMain)) {
35                     matches = false; // Characters do not match and no mapping exists
36                 }
37             }
38
39             // If all characters match (or have valid mappings), return true
40             if (matches) {
41                 return true;
42             }
43
44             // No matching substring found, return false
45             return false;
46         }
47     }
48 };
49
```

Typescript Solution

```
1 type Mapping = Array<[string, string]>;
2
3 // Function to determine if 'sub' after replacements can match any substring in 's'
4 // s: the main string in which to search for the substring
5 // sub: the substring to find in the main string 's' after applying the replacements
6 // mappings: an array of tuples where each tuple is a legal Mapping (replacement) from one character to another
7 function matchReplacement(s: string, sub: string, mappings: Mapping): boolean {
8     // Map to hold all replacement options for each character
9     const replacementDict: Map<string, Set<string>> = new Map();
10
11     // Populate the replacement dictionary with the mappings
12     mappings.forEach((key, value) => {
13         if (!replacementDict.has(key)) {
14             replacementDict.set(key, new Set());
15         }
16         replacementDict.get(key)!.add(value);
17     });
18
19     const mainStrLength: number = s.length; // Length of the main string 's'
20     const subStrLength: number = sub.length; // Length of the substring 'sub'
21
22     // Iterate over 's' to check each possible starting position for matching with 'sub'
23     for (let i = 0; i <= mainStrLength - subStrLength; i++) {
24         let matches: boolean = true; // Flag to track if a match is found during iteration
25
26         // Iterate over 'sub', checking character by character
27         for (let j = 0; j < subStrLength && matches; j++) {
28             const charMain: string = s[i + j]; // Character from main string 's'
29             const charSub: string = sub[j]; // Corresponding character from substring 'sub'
30
31             // Check if characters match or there's a valid replacement mapping
32             if (charMain !== charSub && (!replacementDict.get(charSub)?.has(charMain))) {
33                 matches = false; // Characters do not match and no valid replacement exists
34             }
35         }
36
37         // If all characters match or have valid replacements, return true
38         if (matches) {
39             return true;
40         }
41     }
42
43     // No matching substring found, return false
44     return false;
45 }
46
```

Time and Space Complexity

Time Complexity

The time complexity of the code is mainly determined by the two loops present.

- Building the dictionary **d** from the **mappings** list has a time complexity of **O(M)**, where **M** is the length of the **mappings** list. Since each mapping is added once to the dictionary.
- The second part of the code involves iterating over **s** and checking whether any substring of **s** with respect to the replacement rules given by **mappings**. The outer loop runs **O(N - L + 1)** times, where **N** is the length of **s** and **L** is the length of **sub**. For each iteration, it runs an inner loop over the length of **sub**, which contributes **O(L)**.

For each character in the substring of **s**, the check **a == b or a in d[b]** is made, which takes **O(1)** on average (with a good hash function for the underlying dictionary in Python). However, in the worst case, when the hash function has many collisions, this could degrade to **O(K)**, where **K** is the maximum number of mappings for a single character.

Therefore, the overall worst-case time complexity can be expressed as **O(M + (N - L + 1) * L * K)**. The average case would be **O(M + (N - L + 1) * L)** assuming constant time dictionary lookups.

Space Complexity

- The space complexity for the dictionary **d** is **O(M * K)**, where **M** is the number of unique original characters in **mappings**, and **K** is the average number of replacements for each character.
- No additional significant space is used in the rest of the code.

Combining these factors, the overall space complexity is **O(M * K)**.