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:

1. The characters are the same, in which case no replacement is needed.
2. 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:

1. 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.

2. 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.

3. 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`.

4. 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`.

5. 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`.

6. 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:

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

   ```
   d = {
       'a': {'o'},
       'g': {'c'}
     }
   ```

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

2. 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"`.

3. 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

```python
from collections import defaultdict

class Solution:
    def matchReplacement(self, string: str, substring: str, mappings: list[list[str]]) -> bool:
        # Create a dictionary to store the mappings of characters that can be replaced.
        replacement_dict = defaultdict(set)

        # Populate the replacement dictionary with the mappings provided.
        for original, replacement in mappings:
            replacement_dict[original].add(replacement)

        # Iterate over the string checking for each possible starting index of the substring.
        for start_index in range(len(string) - len(substring) + 1):
            # For each character pair (from the main string and the substring starting at the current index),
            # check if they are the same or if the character from the main string can replace the one
            # from the substring according to the provided mappings.
            if all(char_from_string == char_from_substring or char_from_string in replacement_dict[char_from_substring]
                   for char_from_string, char_from_substring in zip(string[start_index : start_index + len(substring)], substring)):
                # If all characters match (directly or through replacements),
                # the substring can be matched at this index, and True is returned.
                return True

        # If no match was found, return False.
        return False

```

Java Solution

```java
class Solution {
    public boolean matchReplacement(String s, String sub, char[][] mappings) {
        // Create a map to hold characters from `sub` and their allowable replacements.
        Map<Character, Set<Character>> allowedReplacements = new HashMap<>();

        // Populate the map with the mappings provided.
        for (char[] mapping : mappings) {
            // If the character from `sub` is not in the map, add it.
            // Then add the replacement character to the corresponding set.
            allowedReplacements.computeIfAbsent(mapping[0], k -> new HashSet<>()).add(mapping[1]);
        }

        int stringLength = s.length(), subStringLength = sub.length();

        // Try to match the sub string with a segment of string `s`, considering replacements
        for (int i = 0; i <= stringLength - subStringLength; ++i) {
            boolean isMatch = true; // Flag to track if the segment matches with replacements

            // Check each character in the segment
            for (int j = 0; j < subStringLength && isMatch; ++j) {
                char currentChar = s.charAt(i + j);     // Current character from `s`
                char subChar = sub.charAt(j);           // Current character from `sub`

                // Check if the current character matches the sub character or is an allowed replacement
                if (currentChar != subChar && !allowedReplacements.getOrDefault(subChar, Collections.emptySet()).contains(currentChar
                    isMatch = false; // If not, the segment does not match
                }
            }

            // If a match is found, return true
            if (isMatch) {
                return true;
            }
        }

        // If no match is found after checking all segments, return false
        return false;
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
public:
    // Function to determine if 'sub' after replacements can match any substring in 's'
    bool matchReplacement(string s, string sub, vector<vector<char>>& mappings) {
        // Create a map to hold all replacement options for each character
        unordered_map<char, unordered_set<char>> replacementDict;

        // Iterate through the mappings and fill the replacement dictionary
        for (auto& mapping : mappings) {
            // Add the replacement (mapping[1]) for the key character (mapping[0])
            replacementDict[mapping[0]].insert(mapping[1]);
        }

        int mainStrLength = s.size();         // length of the main string 's'
        int subStrLength = sub.size();        // length of the substring 'sub'

        // Iterate over 's' to check each possible starting position
        for (int i = 0; i <= mainStrLength - subStrLength; ++i) {
            bool matches = true; // Flag to determine if a match has been found

            // Iterate over 'sub' to check character by character
            for (int j = 0; j < subStrLength && matches; ++j) {
                char charMain = s[i + j];  // Character from 's'
                char charSub = sub[j];     // Corresponding character from 'sub'

                // Check if characters match or if there's a valid mapping in replacementDict
                if (charMain != charSub && !replacementDict[charSub].count(charMain)) {
                    matches = false; // Characters do not match and no mapping exists
                }
            }

            // If all characters match (or have valid mappings), return true
            if (matches) {
                return true;
            }
        }

        // No matching substring found, return false
        return false;
    }
};

```

Typescript Solution

```typescript
type Mapping = Array<[string, string]>;

// Function to determine if 'sub' after replacements can match any substring in 's'
// s: the main string in which to search for the substring
// sub: the substring to find in the main string 's' after applying the replacements
// mappings: an array of tuples where each tuple is a legal mapping (replacement) from one character to another
function matchReplacement(s: string, sub: string, mappings: Mapping): boolean {
    // Map to hold all replacement options for each character
    const replacementDict: Map<string, Set<string>> = new Map();

    // Populate the replacement dictionary with the mappings
    mappings.forEach(([key, value]) => {
        if (!replacementDict.has(key)) {
            replacementDict.set(key, new Set());
        }
        replacementDict.get(key)!.add(value);
    });

    const mainStrLength: number = s.length;   // Length of the main string 's'
    const subStrLength: number = sub.length;  // Length of the substring 'sub'

    // Iterate over 's' to check each possible starting position for matching with 'sub'
    for (let i = 0; i <= mainStrLength - subStrLength; i++) {
        let matches: boolean = true; // Flag to track if a match is found during iteration

        // Iterate over 'sub', checking character by character
        for (let j = 0; j < subStrLength && matches; j++) {
            const charMain: string = s[i + j];  // Character from main string 's'
            const charSub: string = sub[j];     // Corresponding character from substring 'sub'

            // Check if characters match or there's a valid replacement mapping
            if (charMain !== charSub && (!replacementDict.get(charSub)?.has(charMain))) {
                matches = false; // Characters do not match and no valid replacement exists
            }
        }

        // If all characters match or have valid replacements, return true
        if (matches) {
            return true;
        }
    }

    // No matching substring found, return false
    return false;
}

```

Time and Space Complexity

Time Complexity

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

1. 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.

2. The second part of the code involves iterating over `s` and checking whether `sub` matches 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

1. 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.

2. No additional significant space is used in the rest of the code.

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