1410. HTML Entity Parser

Medium Hash Table String

Leetcode Link

Problem Description

In this problem, we are asked to implement an HTML entity parser. HTML uses special character sequences known as entities to represent certain characters. Each entity begins with an ampersand (`&`) and ends with a semicolon (`;`). In this problem, we are given several predefined entities and their corresponding characters that they represent. For instance, the entity `&quot;` stands for the double quote character `"`. Our task is to write a parser that takes an input string containing these entities and replaces them with their associated characters. The entities and their corresponding characters we need to handle are:

- Quotation Mark: `&quot;` → `"`
- Single Quote Mark: `&apos;` → `'`
- Ampersand: `&` → `&`
- Greater Than Sign: `>` → `>`
- Less Than Sign: `<` → `<`
- Slash: `&frasl;` → `/`

The challenge is to correctly parse the input string and make the substitutions wherever an entity is found so that the output string is the equivalent string with special characters.

Intuition

To solve this problem, we can take a string matching approach. We know that entities are fixed patterns, and we can map each of these patterns to their corresponding characters. The solution involves scanning the input string and checking for the presence of an entity. When an entity is detected, it is replaced with the actual character it represents.

Our solution maintains a dictionary (hash map) `d` where the keys are the entities we want to check for (`&quot;`, `&apos;`, etc.) and the values are the characters those entities represent (`"`, `'`, etc.). As we scan the input string `text`, we use a loop to extract a substring starting from the current position `i` and check if it matches any entity. We examine possible substrings up to a length of 7 (as the longest entity `&frasl;` has a length of 7 characters including `&` and `;`).

Here's the step-by-step approach we use in the solution code:

1. Initialize a pointer `i` to 0 and an empty list `ans` to store the result characters.
2. Iterate over the input string `text` while `i` is less than the length of `text`.
3. For each position `i`, check substrings of length 1 to 7, to see if any of them match an entity in dictionary `d`.
4. If a match is found, append the corresponding character to `ans`, and update `i` to skip the matched entity.
5. If no matching entity is found, append the current character to `ans` and increment `i` by 1 to move to the next character.
6. Continue this process until the end of the input string is reached.
7. Finally, join all the characters in `ans` into a single string and return that as the result.

The key part of this solution is the use of a dictionary for efficient entity look-up and the careful iteration over potential entity substrings to accommodate the necessary replacements.

Solution Approach

The implementation of the solution follows a straightforward pattern matching and substitution approach leveraging a dictionary for fast lookups. Below is an in-depth walk-through of how the algorithm, data structures, and patterns are used in the provided solution code:

1. **Dictionary for Entity Mapping**: A dictionary `d` is initialized to store the mapping of HTML entities to their corresponding characters. The use of a dictionary (hash map) allows for constant-time look-up which is efficient for our substitution needs.

```
d = {
    '&quot;': '"',
    '&apos;': "'",
    '&amp;': '&',
    '&gt;': '>',
    '&lt;': '<',
    '&frasl;': '/',
}
```

2. **Scanning the Input String**: Two pointers are initiated, `i` to iterate through the string, and `j` to check for entities within a window that slides forward from `i`.

   - `i` starts at `i` and moves through the string character by character.
   - `j` is the length of the input string `text`.

3. **Checking for Entities**: Within the while loop, which continues until `i` is less than `n`, the code uses a nested loop to check for entities of varying lengths (from 1 to 7).

```
while i < n:
    for j in range(1, 8):
        s = i + j
        if s in d:
```

4. **Appending Replaced Characters or Original Characters**: When a matching entity is found within the current window of the string (signified by the range `(i, j)`), the corresponding character is appended to the `ans` list and `i` is updated to be `j` to continue scanning after the entity. If no entity is found, the current character at position `i` is appended instead and `i` is incremented by one.

```
ans.append(text[i:j])
i = j
```

or if no match is found:

```
ans.append(text[i])
i += 1
```

5. **Joining and Returning the Result**: After the entire string has been scanned and entities have been replaced, the `ans` list is joined to form a string which is returned as the final output.

```
return ''.join(ans)
```

The algorithm effectively does the following:

- It processes the input string character by character, but also examines potential entity matches by looking ahead up to 7 characters.
- It uses a dictionary for efficient mapping from entities to characters.
- It maintains the order of characters within the input string, only replacing entities when found.
- It compiles the output dynamically, making it responsive to any length of the input text without the need for pre-sizing.

This approach ensures that all entities are replaced appropriately, and the string is processed in a single pass, which offers a good balance of time complexity (linear in relation to the length of the text) and space complexity (constant for the dictionary plus linear for the output string).

Example Walkthrough

Let's go through the solution approach with a small example.

Consider the input string `text = "Hello & World > Universe &frasl; Spaces"`. In this example, we have HTML entities that correspond to an ampersand, a greater-than sign, and a forward slash, which need to be replaced with their respective characters `&`, `>`, and `/`.

Using our solution approach:

1. We initialize our dictionary `d` that maps entities to their corresponding characters.
2. We set `i = 0`, which will be used to iterate over `text`, and `ans`, an empty list to store the result characters.
3. Start with the while loop: `while i < len(text)`.
4. Now, looking at the first characters starting from index 0, we see "Hello ". Since none of these characters match an entity, they are appended to `ans`.
5. When we reach `&`, our loop checks substring lengths from 1 to 7 characters starting with `i`. It finds that the substring "&" matches an entity in `d`.
   - We append `&` to `ans`.
   - Set `i` to the index after the matched entity.
6. We repeat this process for the rest of the string.
   - " World " is added directly to `ans`.
   - `>` translates to `>`, so `>` is added to `ans`.
   - " Universe " is added directly to `ans`.
   - `&frasl;` translates to `/`, so `/` is added to `ans`.
   - The rest of the string " Spaces" is added directly to `ans`.
7. After the entire string is processed, we join the list `ans` into a single string:

```
'Hello & World > Universe / Spaces'
```

And that is the expected output, which we return.

This walkthrough demonstrates how the algorithm scans the input string, checks for entities, appends the corresponding characters or the original ones, and ultimately joins them together for the result.

Python Solution

```
class Solution:
    def entityParser(self, text: str) -> str:
        # Dictionary to map HTML entities to their corresponding characters
        entity_to_char = {
            "&quot;": '"',
            "&apos;": "'",
            "&amp;": "&",
            "&gt;": ">",
            "&lt;": "<",
            "&frasl;": "/",
        }

        i, text_length = 0, len(text)  # Initialize pointer i and get the length of the text
        parsed_text = []  # List to hold the parsed characters

        # Loop through each character in the text
        while i < text_length:
            # Check for entities with a maximum length of 7 characters
            for entity_length in range(1, 8):
                end_index = i + entity_length
                # If a substring matches a key in the dictionary, append the corresponding character
                if text[i:end_index] in entity_to_char:
                    parsed_text.append(entity_to_char[text[i:end_index]])
                    i = end_index  # Move the pointer past the entity
                    break
            else:
                # If no entity is found, append the current character
                parsed_text.append(text[i])
                i += 1  # Move the pointer to the next character

        # Join all the parsed characters to form the final string
        return ''.join(parsed_text)
```

Java Solution

```
import java.util.Map;
import java.util.HashMap;

class Solution {
    public String entityParser(String text) {
        // Create a map for the HTML entity to its corresponding character.
        Map<String, Character> htmlEntityMap = new HashMap<>();
        htmlEntityMap.put("&quot;", '"');
        htmlEntityMap.put("&apos;", '\'');
        htmlEntityMap.put("&amp;", '&');
        htmlEntityMap.put("&gt;", '>');
        htmlEntityMap.put("&lt;", '<');
        htmlEntityMap.put("&frasl;", '/');

        // StringBuilder to hold the final parsed string.
        StringBuilder parsedString = new StringBuilder();

        // Variable to track the current index in the input string.
        int currentIndex = 0;

        // Length of the input text.
        int textLength = text.length();

        // Iterate over the input text to find and replace entities.
        while (currentIndex < textLength) {
            // Flag to mark if we found an entity match.
            boolean isEntityFound = false;
            // Try all possible lengths for an entity (entities are between 1 and 7 characters long).
            for (int length = 1; length <= 7 && currentIndex + length <= textLength; ++length) {
                // Extract a substring of the current length from the current index.
                int endIndex = currentIndex + length;
                String currentSubstring = text.substring(currentIndex, endIndex);
                // Check if the current substring is a known entity.
                if (htmlEntityMap.containsKey(currentSubstring)) {
                    // If it's an entity, append the corresponding character to parsedString.
                    parsedString.append(htmlEntityMap.get(currentSubstring));
                    // Move the current index forward past the entity.
                    currentIndex += length;
                    // Indicate that we found and handled an entity.
                    isEntityFound = true;
                    // Break out of the loop as we only want to match the longest possible entity.
                    break;
                }
            }
            // If no entity was found, append the current character to parsedString and move to the next character.
            if (!isEntityFound) {
                parsedString.append(text.charAt(currentIndex++));
            }
        }
        // Return the fully parsed string with all entities replaced.
        return parsedString.toString();
    }
}
```

C++ Solution

```
#include <string>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // This function converts HTML entities in a string to their corresponding characters.
    string entityParser(string& text) {
        // Dictionary to map HTML entities to their respective characters.
        unordered_map<string, string> htmlEntityToChar = {
            {"&quot;", "\""},
            {"&apos;", "'"},
            {"&amp;", "&"},
            {"&gt;", ">"},
            {"&lt;", "<"},
            {"&frasl;", "/"}
        };

        // The resulting string after parsing HTML entities.
        string parsedText = "";

        // Loop through each character of the input text.
        int index = 0;
        int textLength = text.size();
        while (index < textLength) {
            bool entityFound = false; // Flag to indicate if an HTML entity is found.

            // Try to match the longest possible HTML entity by checking substrings of lengths 1 to 7.
            for (int length = 1; length <= 7; ++length) {
                int endIndex = index + length; // Calculate the end index of the current substring.

                // Check if the substring is within the text boundaries.
                if (endIndex <= textLength) {
                    string currentSubstring = text.substr(index, length);

                    // If the current substring is an HTML entity, append the respective character.
                    if (htmlEntityToChar.count(currentSubstring)) {
                        parsedText += htmlEntityToChar[currentSubstring];
                        index = endIndex; // Move the index past the HTML entity.
                        entityFound = true; // Mark that an entity was found.
                        break; // Break out of for-loop once an HTML entity is found and handled.
                    }
                }
            }

            // If no HTML entity is found, append the current character to the result.
            if (!entityFound) {
                parsedText += text[index++];
            }
        }

        return parsedText;
    }
};
```

Typescript Solution

```
// Typescript type definition for a mapping from HTML entities to characters.
type EntityToCharMap = { [entity: string]: string };

// A dictionary to map HTML entities to their respective characters.
const htmlEntityToChar: EntityToCharMap = {
    "&quot;": '"',
    "&apos;": "'",
    "&amp;": "&",
    "&gt;": ">",
    "&lt;": "<",
    "&frasl;": "/"
};

// This function converts HTML entities in a string to their corresponding characters.
function entityParser(text: string): string {
    // The resulting string after parsing HTML entities.
    let parsedText = "";

    // Loop through each character of the input text.
    let index = 0;
    const textLength = text.length;
    while (index < textLength) {
        let entityFound = false; // Flag to indicate if an HTML entity is found.

        // Try to match the longest possible HTML entity by checking substrings of lengths 1 to 7.
        for (let length = 1; length <= 7; ++length) {
            const endIndex = index + length; // Calculate the end index of the current substring.

            // Check if the substring is within the text boundaries.
            if (endIndex <= textLength) {
                const currentSubstring = text.substring(index, endIndex);

                // If the current substring is an HTML entity, append the respective character.
                if (currentSubstring in htmlEntityToChar) {
                    parsedText += htmlEntityToChar[currentSubstring];
                    index = endIndex; // Move the index past the HTML entity.
                    entityFound = true; // Mark that an entity was found and handled.
                    break; // Break out of for-loop once an HTML entity is found and handled.
                }
            }
        }

        // If no HTML entity is found, append the current character to the result.
        if (!entityFound) {
            parsedText += text[index++];
        }
    }

    return parsedText;
}

// Example usage:
// let convertedText = "Hello &amp; welcome to &lt;coding&gt;!";
// const convertedText = entityParser(originalText);
// console.log(convertedText); // Output should be "Hello & welcome to <coding>!"
```

Time and Space Complexity

The time complexity of the algorithm is $O(n)$, where n is the length of the input string `text`. This is because, in the worst case, each character in the string is visited once. The check `text[i:j] in d` is $O(1)$ because dictionary lookups in Python are constant time, and the loop for range `i` doesn't significantly affect the time complexity as it's bounded (maximum length of the strings in the dictionary is 7, which is a constant).

The space complexity of the code is $O(n)$ as well, where n is the length of the text. This is because a list `ans` is being used to construct the output, and it can grow up to the length of the input string in the case where no entity is replaced.