1410. HTML Entity Parser String Medium Hash Table **Leetcode Link** 

## **Problem Description**

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 " 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: " → "

In this problem, we are asked to implement an HTML entity parser. HTML uses special character sequences known as entities to

- Single Quote Mark: ' → ¹ Ampersand: & → &
- Greater Than Sign: > → > Less Than Sign: < → <</li>
- Slash: ⁄ → / 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 (", ', 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

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.

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.

longest entity ⁄ has a length of 7 characters including & and;).

- 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 perform the necessary replacements.
- **Solution Approach**

The implementation of the solution follows a straightforward pattern matching and substitution approach leveraging a dictionary for

5. If no matching entity is found, append the current character to ans and increment i by 1 to move to the next character.

fast lookups. Below is an in-depth walk-through of how the algorithm, data structures, and patterns are used in the provided solution code:

characters. The use of a dictionary (hash map) allows for constant-time look-up which is efficient for our substitution needs. '"': '"' ''': "'"

1. Dictionary for Entity Mapping: A dictionary d is initialized to store the mapping of HTML entities to their corresponding

### 2. Scanning the Input String: Two pointers are initiated, i to iterate through the string, and j to check for entities within a window

incremented by one.

1 ans.append(text[i])

1 return ''.join(ans)

The algorithm effectively does the following:

2 i += 1

characters.

"<": '<'

that slides forward from i.

n is the length of the input string text.

entities of varying lengths (from 1 to 7).

"⁄": '/',

1 while i < n: for l in range(1, 8): if text[i:j] 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

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

1 ans.append(d[text[i:j]]) 2 i = jor if no match is found:

to form a string which is returned as the final output.

i starts at 0 and moves through the string character by character.

 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

correspond to an ampersand, a greater-than sign, and a forward slash, which need to be replaced with their respective characters &,

It processes the input string character by character, but also examines potential entity matches by looking ahead up to 7

5. Joining and Returning the Result: After the entire string has been scanned and entities have been replaced, the ans list is joined

Let's go through the solution approach with a small example. Consider the input string text = "Hello & World > Universe ⁄ Spaces". In this example, we have HTML entities that

Using our solution approach:

matches an entity in d.

○ We append & to ans.

>, and /.

**Example Walkthrough** 

for the output string).

3. Start with the while loop: while i < len(text).

1. We initialize our dictionary d that maps entities to their corresponding characters.

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 "&"

4. Now, looking at the first characters starting from index 0, we see "Hello". Since none of these characters match an entity, they

2. We set i = 0, which will be used to iterate over text, and ans, an empty list to store the result characters.

 ⁄ 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:

def entityParser(self, text: str) -> str:

entity\_to\_char = {

'"': '"'

''': "'"

'&': '&',

'>': '>',

'<': '<',

'⁄': '/',

while i < text\_length:</pre>

or the original ones, and ultimately joins them together for the result.

# Dictionary to map HTML entities to their corresponding characters

# Check for entities with a maximum length of 7 characters

# If no entity is found, append the current character

i += 1 # Move the pointer to the next character

# Join all the parsed characters to form the final string

parsed\_text = [] # List to hold the parsed characters

# Loop through each character in the text

for length in range(1, 8):

end\_index = i + length

parsed\_text.append(text[i])

isEntityFound = true;

parsedString.append(text.charAt(currentIndex++));

// This function converts HTML entities in a string to their corresponding characters.

bool entityFound = false; // Flag to indicate if an HTML entity is found.

// Check if the substring is within the text boundaries.

if (htmlEntityToChar.count(currentSubstring)) {

entityFound = true; // Set the flag.

string currentSubstring = text.substr(index, length);

parsedText += htmlEntityToChar[currentSubstring];

index = endIndex; // Move the index past the HTML entity.

// Try to match the longest possible HTML entity by checking substrings of lengths 1 to 7.

int endIndex = index + length; // Calculate the end index of the current substring.

// If the current substring is an HTML entity, append the respective character.

break; // Break out of for-loop once an HTML entity is found and handled.

// Dictionary to map HTML entities to their respective characters.

unordered\_map<string, string> htmlEntityToChar = {

// The resulting string after parsing HTML entities.

for (int length = 1; length <= 7; ++length) {</pre>

if (endIndex <= textLength) {</pre>

// Loop through each character of the input text.

// Return the fully parsed string with all entities replaced.

break;

if (!isEntityFound) {

return parsedString.toString();

string entityParser(const string& text) {

{""", "\""},

{"'", "'"},

{"&", "&"},

{">", ">"}, {"<", "<"},

{"⁄", "/"}

string parsedText = "";

int textLength = text.size();

while (index < textLength) {</pre>

int index = 0;

i, text\_length = 0, len(text) # Initialize pointer i and get the length of the text

# If a substring matches a key in the dictionary, append the corresponding character

1 "Hello & World > Universe / Spaces"

Python Solution

1 class Solution:

4

6

9

10

11

12

13

14

15

16

17

18

19

20

21

27

28

29

30

31

32

33

39

40

41

43

44

45

46

48

49

50

51

53

52 }

C++ Solution

#include <string>

class Solution {

**}**;

public:

9

10

11

12

13

14

15

16

17

18

19

20

21

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

#include <unordered\_map>

using namespace std;

Java Solution

1 import java.util.Map;

"Universe" is added directly to ans.

Set i to the index after the matched entity.

6. We repeat this process for the rest of the string.

> translates to >, so > is added to ans.

"World" is added directly to ans.

- 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
- 22 if text[i:end\_index] in entity\_to\_char: 23 parsed\_text.append(entity\_to\_char[text[i:end\_index]]) 24 i = end\_index # Move the pointer past the entity 25 break 26 else:

return ''.join(parsed\_text)

```
2 import java.util.HashMap;
   class Solution {
       public String entityParser(String text) {
           // Create a map for the HTML entity to its corresponding character.
 6
           Map<String, String> htmlEntityMap = new HashMap<>();
           htmlEntityMap.put(""", "\"");
 8
           htmlEntityMap.put("'", "'");
 9
           htmlEntityMap.put("&", "&");
10
11
           htmlEntityMap.put(">", ">");
12
           htmlEntityMap.put("<", "<");</pre>
13
           htmlEntityMap.put("⁄", "/");
14
15
           // StringBuilder to hold the final parsed string.
16
           StringBuilder parsedString = new StringBuilder();
17
18
           // Variable to track the current index in the input string.
19
           int currentIndex = 0;
20
21
           // Length of the input text.
22
           int textLength = text.length();
23
24
           // Iterate over the input text to find and replace entities.
25
           while (currentIndex < textLength) {</pre>
26
               // Flag to mark if we found an entity match.
27
               boolean isEntityFound = false;
               // Try all possible lengths for an entity (entities are between 1 and 7 characters long).
28
29
                for (int length = 1; length <= 7 && currentIndex + length <= textLength; ++length) {</pre>
30
                   // Extract a substring of the current length from the current index.
31
                   String currentSubstring = text.substring(currentIndex, currentIndex + length);
32
                   // Check if the current substring is a known entity.
33
                   if (htmlEntityMap.containsKey(currentSubstring)) {
                       // If it's an entity, append the corresponding character to parsedString.
34
35
                        parsedString.append(htmlEntityMap.get(currentSubstring));
36
                       // Move the current index forward past the entity.
37
                        currentIndex += length;
                       // Indicate that we found and handled an entity.
38
```

// Break out of the loop as we only want to match the longest possible entity.

// If no entity was found, append the current character to parsedString and move to the next character.

### 49 50 51 52

```
43
 44
 45
 46
                 // If no HTML entity is found, append the current character to the result.
                 if (!entityFound) {
 47
 48
                     parsedText += text[index++];
             return parsedText;
 53
 54
    };
 55
Typescript Solution
  1 // TypeScript type definition for a mapping from HTML entities to characters.
  2 type EntityToCharMap = { [entity: string]: string };
    // A dictionary to map HTML entities to their respective characters.
    const htmlEntityToChar: EntityToCharMap = {
       """: '"'
       "'": "'"
      "&": "&",
      ">": ">",
      "<": "<"
 10
       "⁄": "/"
 11
 12 };
 13
 14 // This function converts HTML entities in a string to their corresponding characters.
 15 function entityParser(text: string): string {
      // The resulting string after parsing HTML entities.
 17
       let parsedText = "";
 18
 19
      // Loop through each character of the input text.
       let index = 0;
 20
       const textLength = text.length;
 21
 22
 23
       while (index < textLength) {</pre>
 24
         let entityFound = false; // Flag to indicate if an HTML entity is found.
 25
 26
         // Try to match the longest possible HTML entity by checking substrings of lengths 1 to 7.
 27
         for (let length = 1; length <= 7; ++length) {</pre>
 28
           const endIndex = index + length; // Calculate the end index of the current substring.
 29
 30
           // Check if the substring is within the text boundaries.
           if (endIndex <= textLength) {</pre>
 31
 32
             const currentSubstring = text.substring(index, endIndex);
 33
             // If the current substring is an HTML entity, append the respective character.
 34
 35
            if (currentSubstring in htmlEntityToChar) {
 36
               parsedText += htmlEntityToChar[currentSubstring];
 37
               index = endIndex; // Move the index past the HTML entity.
               entityFound = true; // Set the flag.
 38
 39
               break; // Break out of for-loop once an HTML entity is found and handled.
 40
 41
 43
        // If no HTML entity is found, append the current character to the result.
 44
        if (!entityFound) {
 45
           parsedText += text[index++];
 46
 47
 48
 49
```

# Time and Space Complexity

// const originalText = "Hello & welcome to <coding&gt;!";

// console.log(convertedText); // Output should be "Hello & welcome to <coding>!"

// const convertedText = entityParser(originalText);

return parsedText;

// Example usage:

50

51

52

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 0(1) because dictionary lookups in Python are constant time, and the loop for range 1 doesn't significantly affect the time complexity as it's bounded (maximum length of the strings in the dictionary d 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.