1807. Evaluate the Bracket Pairs of a String

Medium Array Hash Table String

Problem Description

This problem involves a string `s` which contains several bracket pairs. Each pair encapsulates a key. For instance, in this string: "(name)is(age)yearsold", there are two pairs of brackets with the keys "name" and "age".

There is also a 2D array called `knowledge` that holds pairs of keys and their corresponding values, such as `[["name","Alice"], ["age","12"]]`. This array tells us what value each key holds.

The task is to parse through the string `s` and replace each key within the brackets with its corresponding value from the `knowledge` array. If a key is not found in `knowledge`, it should be replaced with a question mark "?".

Every key will be unique and present just once in the knowledge base. The string `s` does not contain any nested brackets, simplifying the parsing process.

The goal is to return a new string with all bracket pairs evaluated and substituted by their corresponding values or a "?" if the key's value is unknown.

Intuition

The intuition behind the solution is to perform a linear scan of the string `s`, using a variable `i` to keep track of our position. We proceed through the string character by character until we encounter an opening bracket `(`.

When an opening bracket is detected, we know that a key is started. We then find the closing bracket `)` that pairs with the opening bracket. The substring within the brackets is the key we are interested in.

We check the dictionary `d` we've made from the `knowledge` array to see if the key exists:

- If the key has a corresponding value in `d`, we add that value to the result string.
- If the key does not exist in `d`, we append a question mark `?` to the result string.

Once we have replaced the key with its value or a question mark, we move `i` to the position just after the closing bracket since all characters within the brackets have been processed.

If we encounter any character other than an opening bracket, we simply add that character to the result string, as it does not need any substitution.

Our process continues until we've scanned all characters within the string `s`. In the end, we join all parts of our result list into a single string and return it as the solution.

Solution Approach

The solution leverages a dictionary and simple string traversal to effectively solve the problem. The approach follows these steps:

1. Convert the `knowledge` list of lists into a dictionary where each key-value pair is one key from the bracket pairs and its corresponding value. This allows for constant-time access to the values while we traverse the string `s`.

```
d = {a: b for a, b in knowledge}
```

2. Initialize an index `i` to start at the first character of the string `s` and a `ans` list to hold the parts of the new string we're building.
3. Start iterating over the string `s` using `i`. We need to check each character and decide what to do:
   - If the current character is an opening bracket `'('`, we must find the matching closing bracket `')'` to identify the key within.
   - We use the `find()` method to locate the index `j` of the closing bracket.
   - Obtain the key by slicing the string `s` from `i + 1` to `j` to get the content within brackets.

```
i = s.find(')', i + 1)
key = s[i + 1 : j]
```

4. Use the key to look up the value in the dictionary `d`. If the key is found, append the value to the `ans` list; otherwise, append a question mark `'?'`.

```
ans.append(d.get(key, '?'))
```

5. Update the index `i` to move past the closing bracket.
6. If the current character isn't an opening bracket, append it directly to the `ans` list because it's part of the final string.

```
ans.append(s[i])
```

7. Increment `i` to continue to the next character.
8. After the while loop completes, we will have iterated over the entire string, replacing all bracketed keys with their corresponding values or a question mark.
9. Finally, join the elements of the `ans` list into a string:

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

No advanced algorithms are needed for this problem; it primarily relies on string manipulation techniques and dictionary usage. The approach is efficient as it makes one pass through the string and accesses the dictionary values in constant time.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose we have the following string `s` and knowledge base:

```
s = "Hi, my name is (name) and I am (age) years old." knowledge = [["name","Bob"], ["age","25"]]
```

First, we convert the `knowledge` list into a dictionary, `d`:

```
d = {"name": "Bob", "age": "25"}
```

Now, we initialize our starting index `i = 0`, and an empty list `ans = []` to store parts of our new string.

We begin iterating over the string `s`. The first characters "Hi, my name is " are not within brackets, so we add them directly to `ans`.

At index 14, we encounter an opening bracket `(`. Now we need to find the corresponding closing bracket `)`:

- We use `s.find(')', i + 1)` and find the closing bracket at index 19.
- The key within the brackets is `s[i + 1 : j]`, which yields `"name"`.
- We search for `"name"` in `d` and find that it corresponds to the value `"Bob"`.
- `ans.append(d.get(key, '?'))` will result in appending `"Bob"` to `ans`.

Next, we update `i` to be just past the closing bracket; `i = 20`.

Continuing to iterate, we add the characters " and I am " directly to `ans`, as they are outside of brackets.

Upon reaching the next opening bracket at index 29, we repeat the process:

- We find the closing bracket at index 33.
- We identify the key as `"age"`.
- We search for `"age"` and obtain the value `"25"` from `d`.
- We append `"25"` to `ans`.

Finally, `i` is updated to index 34, and we add the remaining characters " years old." to `ans`.

After processing the entire string, we join all parts of `ans` using `''.join(ans)` to get the final string:

```
"Hi, my name is Bob and I am 25 years old."
```

This is the string with the keys in the original string `s` replaced by their corresponding values from the `knowledge` base, which is the desired outcome. If at any point a key had not been found in `d`, a `'?'` would have been appended instead.

Solution Implementation

Python

```
class Solution:
    def evaluate(self, expression: str, knowledge: List[List[str]]) -> str:
        # Convert the `knowledge` list of lists into a dictionary for fast lookups
        knowledge_dict = {key: value for key, value in knowledge}

        index, length of expression = 0, len(expression)
        result = [] # This list will collect the pieces of the evaluated expression

        # Iterate through the expression string
        while index < length of expression:
            # If the current character is '(', find the corresponding ')'
            if expression[index] == '(':
                closing_paren_index = expression.find(')', index + 1)
                # Extract the key between the parentheses
                key = expression[index + 1 : closing_paren_index]
                # Append the value from the knowledge dictionary if it exists, otherwise '?'
                result.append(knowledge_dict.get(key, '?'))
                # Move the index past the closing parenthesis
                index = closing_paren_index
            else:
                # Append the current character to the result if it's not part of a key
                result.append(expression[index])
            # Move to the next character
            index += 1

        # Join all parts of the result list into a single string
        return ''.join(result)
```

Java

```
class Solution {

    public String evaluate(String s, List<List<String>> knowledge) {

        // Create a dictionary from the provided knowledge list
        Map<String, String> dictionary = new HashMap<>(knowledge.size());
        // Populate the dictionary with key-value pairs from the knowledge list
        for (List<String> entry : knowledge) {
            dictionary.put(entry.get(0), entry.get(1));
        }

        // StringBuilder to construct the final evaluated string
        StringBuilder evaluatedString = new StringBuilder();

        // Iterate over the entire input string character by character
        for (int i = 0; i < s.length(); ++i) {

            // If current character is '(', it's the start of a key
            if (s.charAt(i) == '(') {

                // Find the corresponding closing ')' to get the key
                int j = s.indexOf(')', i + 1);

                // Extract the key from the input string
                String key = s.substring(i + 1, j);

                // Append the value for the key from the dictionary to the result
                // If the key is not found, append "?"
                evaluatedString.append(dictionary.getOrDefault(key, "?"));

                // Move the index to the character after the closing ')'
                i = j;
            } else {
                // If current character is not '(', append it directly to the result
                evaluatedString.append(s.charAt(i));
            }
        }

        // Return the fully evaluated string
        return evaluatedString.toString();
    }
}
```

C++

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

class Solution {
public:
    // Function that takes a string s and a knowledge base as input, and replaces
    // all substrings enclosed in parentheses with their corresponding values from
    // the knowledge base. If a substring does not exist in the knowledge base,
    // it is replaced with "?".
    string evaluate(string s, vector<vector<string>>& knowledge) {
        // Creating a dictionary (hash map) to store the key-value pairs
        // from the knowledge base for quick lookup.
        unordered_map<string, string> knowledgeMap;
        for (auto& entry : knowledge) {
            knowledgeMap[entry[0]] = entry[1];
        }

        // String to store the final result after all replacements are done.
        string result;
        // Iterating over the input string to find and replace values.
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(') { // Check if the current character is an opening parenthesis.
                int j = s.find(')', i + 1); // Find the corresponding closing parenthesis.
                // Extract the key between the parentheses.
                string key = s.substr(i + 1, j - i - 1);
                // Lookup the key in the knowledge map and append to result.
                // if key not found, append "?".
                result += knowledgeMap.count(key) ? knowledgeMap[key] : "?";
                i = j; // Move the index to the position of the closing parenthesis.
            } else {
                // Append the current character to result if it's not an opening parenthesis.
                result += s[i];
            }
        }
        return result; // Return the final result string.
    }
};
```

TypeScript

```
function evaluate(expression: string, knowledgePairs: string[][]): string {
    // Get the length of the expression.
    const expressionLength = expression.length;
    // Create a map to hold the knowledge key-value pairs.
    const knowledgeMap = new Map<string, string>();
    // Populate the map using the knowledgePairs array.
    for (const [key, value] of knowledgePairs) {
        knowledgeMap.set(key, value);
    }
    // Initialize an array to hold the parts of the answer as we process the expression.
    const answerParts = [];
    // Initialize the index to iterate through the expression.
    let index = 0;

    // Iterate over the characters in the expression.
    while (index < expressionLength) {
        // Check if the current character is the beginning of a key placeholder.
        if (expression[index] === '(') {
            // Find the closing parenthesis for the current key placeholder.
            const closingIndex = expression.indexOf(')', index + 1);
            // Extract the key from inside the parenthesis.
            const key = expression.slice(index + 1, closingIndex);
            // Retrieve the value associated with the key from the map, defaulting to '?'.
            const value = knowledgeMap.get(key) ?? '?';
            // Add the value to the answerParts array.
            answerParts.push(value);
            // Move the index to the position after the closing parenthesis.
            index = closingIndex;
        } else {
            // If the current character is not a parenthesis, add it to the answerParts array as is.
            answerParts.push(expression[index]);
        }
        // Move to the next character.
        index++;
    }

    // Join all parts of the answer to form the final string and return it.
    return answerParts.join('');
}
```

```
class Solution:
    def evaluate(self, expression: str, knowledge: List[List[str]]) -> str:
        # Convert the `knowledge` list of lists into a dictionary for fast lookups
        knowledge_dict = {key: value for key, value in knowledge}

        index, length of expression = 0, len(expression)
        result = [] # This list will collect the pieces of the evaluated expression

        # Iterate through the expression string
        while index < length of expression:
            # If the current character is '(', find the corresponding ')'
            if expression[index] == '(':
                closing_paren_index = expression.find(')', index + 1)
                # Extract the key between the parentheses
                key = expression[index + 1 : closing_paren_index]
                # Append the value from the knowledge dictionary if it exists, otherwise '?'
                result.append(knowledge_dict.get(key, '?'))
                # Move the index past the closing parenthesis
                index = closing_paren_index
            else:
                # Append the current character to the result if it's not part of a key
                result.append(expression[index])
            # Move to the next character
            index += 1

        # Join all parts of the result list into a single string
        return ''.join(result)
```

Time and Space Complexity

The time complexity of the code is $O(n + k)$, where n is the length of the string `s` and k is the total number of characters in all keys of the dictionary `d`. Here's the breakdown:

- We loop through the entire string `s` once, which gives us $O(n)$.
- Constructing the dictionary `d` has a time complexity of $O(k)$, assuming a constant time complexity for each insert operation into the hash table, and `k` is the total length of all keys present in `knowledge`.

The space complexity of the code is $O(n + m)$, where m is the space required to store the dictionary `d` and n is the space for the answer string (assuming each character can be counted as taking $O(1)$ space):

- The dictionary `d` will have a space complexity of $O(k)$, where k is the sum of the lengths of all keys and values in the `knowledge` list.
- The list `ans` that accumulates the answer will have at most the same length as the input string `s`, since each character in `s` is processed once, leading to $O(n)$ space.

Therefore, the overall space complexity is the sum of the space complexities of `d` and `ans`, i.e., $O(n + m)$.