1096. Brace Expansion II

Hard Stack Breadth-First Search String Backtracking Leetcode Link

Problem Description

This problem involves interpreting a custom grammar to generate a set of words that an input expression represents. This grammar operates on lowercase English letters and curly braces `{}`.

The rules are as follows:

1. A single letter like `a` represents the set `{a}`.
2. A comma-separated list within curly braces like `{a,b,c}` creates the union of the single letters, resulting in `{a, b, c}`.
3. Concatenation of sets means forming all possible combinations of words where the first part comes from the first set and the second part from the second, like `ab + cd` will give `{abcd}`.

The task is, given a string expression that uses this grammar, to return a sorted list of all the unique words that it represents.

Intuition

The intuition of the solution relies on the principles of Depth-First Search (DFS). We want to explore all possible combinations of the strings as described by the expression. The idea behind this code is to solve the expression recursively by expanding the braces from the innermost to the outermost.

The process can be broken down into the following steps:

1. Locate the innermost right brace '}' and find the corresponding left brace '{' that starts the current scope.
2. Split the content within the braces by commas to obtain individual segments that need to be combined.
3. Build new expressions by replacing the content within the current braces with each of the segments and recursively process these new expressions.
4. As we're using a set `s` to store the results, it will automatically handle duplicates and give us a unique set of words.
5. Once all recursive calls are completed, return the words in sorted order as required.

This approach follows a divide-and-conquer strategy, breaking down the expression into smaller parts until it can't be divided anymore, i.e., when no more braces are found. At this point, the simplest expressions are single words that are added to the set `s`. As the recursion unwinds, the expressions are built up and added to `s` until the original expression has been fully expanded.

Solution Approach

The solution algorithm is designed using a recursive function `dfs`. The idea of this function is to expand the input expression by replacing the expressions within the braces step by step until no more braces exist.

Here's how the algorithm and its components work:

- **Data Structure Used**: A set `s` is used to store unique combinations of the expressions as they're being expanded. `set` in Python takes care of both storing the elements and ensuring all elements are unique.
- **Recursive Function `dfs`**: The function `dfs` is defined which will do most of the heavy lifting. It takes a string `exp` (partial expression) as its argument.
- **Base Case**: First, it checks whether there is a '}' in the current expression. If there isn't, it means the expression is fully expanded (i.e., has no more braces to process), and it can be added directly to the set `s`.
- **Finding Braces Pairs**: If a '}' is found, it locates the left brace '{' that matches with this right brace by using `exp.rfind('{', 0, j - 1)`, which searches for the last '{' before the found '}'.
- **Processing Innermost Expression**:
  - It splits the content located between these braces on commas, resulting in a list of smaller expressions that are not yet combined.
  - It generates new partial expressions by taking the part of the expression before the found '{' (`a`), each of the smaller expressions (`b`), and the part after the '}' (`c`) and calls `dfs(a + b + c)` on them.
- **Recursion**: This recursive call continues to expand more braces within the new partial expressions until all possibilities are added to the set `s`.
- **Sorting**: Once the recursion is completed, the function `braceExpansionII` returns `sorted(s)`, which is a list of all the unique words represented by the original expression, sorted in lexicographical order.

To summarize, the algorithm uses DFS with recursion to expand the expressions by iteratively replacing the curly braces with their possible combinations, collecting results in a set to maintain uniqueness, and then returning the sorted list of those results.

Example Walkthrough

Let's say we have the following input expression:

```
"{a,b}{c,d}"
```

According to the problem description and the rules of the given grammar, this expression could generate the following set of words: `{ac, ad, bc, bd}`.

Here's a step-by-step illustration of the solution approach using this example:

1. **Start DFS**: The expression has braces, so our `dfs` function starts processing it.
2. **Locate Braces**: The `dfs` function finds the rightmost '}' at index 7 and then finds its corresponding left brace '{' at index 4.
3. **Split Content**: The content within these braces is `"c,d"`. The `dfs` function splits this into `["c", "d"]`.
4. **Build Expressions**: The expression before the '{' is `{a,b}`, and there's nothing after '}'. So we create new expressions by replacing `{c,d}` with `c` and with `d` from the split results. The new partial expressions are `{a,b}c` and `{a,b}d`.
5. **Recursion**: Now, we recursively call `dfs` on `{a,b}c` and `{a,b}d`:
   - For `{a,b}c`:
     - Find '}' at index 3 and corresponding '{' at index 0.
     - Split "a,b" into `["a", "b"]`.
     - Create new expressions `ac` and `bc`.
     - As these new expressions contain no braces, add them directly to the set `s`.
   - For `{a,b}d`:
     - Repeat the same process as for `{a,b}c`.
     - Split "a,b" into `["a", "b"]`.
     - Create new expressions `ad` and `bd`.
     - Add these to the set `s`, too.
6. **Completion of DFS**: The set `s` now contains `{"ac", "bc", "ad", "bd"}`.
7. **Return Sorted Result**: Finally, once all recursion is done, `sorted(s)` returns:

```
["ac", "ad", "bc", "bd"]
```

This is the sorted list of all unique words represented by the input expression.

By following the given solution approach, we can see how the problem of expanding and combining expressions is broken down into manageable steps using DFS and recursion. Each level of recursion deals with a simpler expression, and the set `s` ensures that all computed words are unique and stored efficiently. Once all possibilities are explored, the sorted content of this set is the answer.

Python Solution

```python
from typing import List

class Solution:
    def braceExpansionII(self, expression: str) -> List[str]:
        # Helper function to perform depth-first search on the expression
        def dfs(exp: str):
            # Find the position of the first closing brace
            closing_brace_index = exp.find('}')

            # Base case: If there is no closing brace, add the entire expression to the set
            if closing_brace_index == -1:
                expanded_set.add(exp)
                return

            # Find the position of the last opening brace before the found closing brace
            opening_brace_index = exp.rfind('{', 0, closing_brace_index)

            # Divide the expression into three parts: before, inside, and after the braces
            before_brace = exp[:opening_brace_index]
            after_brace = exp[closing_brace_index + 1 :]

            # Split the contents of the braces by commas and recurse for each part
            for inside_brace in exp[opening_brace_index + 1 : closing_brace_index].split(','):
                # Recursively call the dfs function with the new expression
                dfs(before_brace + inside_brace + after_brace)

        # Initialize an empty set to store the expanded expressions
        expanded_set = set()
        # Call the dfs helper function with the initial expression
        dfs(expression)
        # Return the expanded expressions as a sorted list
        return sorted(expanded_set)

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;
import java.util.TreeSet;

public class Solution {

    // Use TreeSet for unique elements and automatic sorting of the results
    private TreeSet<String> resultSet = new TreeSet<>();

    public List<String> braceExpansionII(String expression) {
        // Start depth-first search from the initial expression
        depthFirstSearch(expression);
        // Convert the sorted set to a list for returning as the result
        return new ArrayList<>(resultSet);
    }

    private void depthFirstSearch(String exp) {
        // Search for the closing brace '}' character
        int closingIndex = exp.indexOf('}');

        // If there is no closing brace, we are at the base condition of recursion
        if (closingIndex == -1) {
            // Add the expression to the result set and return
            resultSet.add(exp);
            return;
        }

        // Find the last opening brace '{' before the found '}' to isolate the expression segment
        int openingIndex = exp.lastIndexOf('{', closingIndex);

        // Isolate the parts of the expression before and after the braces segment
        String beforeBraces = exp.substring(0, openingIndex);
        String afterBraces = exp.substring(closingIndex + 1);

        // Split the internal string within braces by ',' for each option and recursively solve
        for (String insideBraces : exp.substring(openingIndex + 1, closingIndex).split(",")) {
            // Recombine the new strings and continue the depth-first search
            depthFirstSearch(beforeBraces + insideBraces + afterBraces);
        }
    }

    // Standard entry point for testing the class functionality
    public static void main(String[] args) {
        // Create solution instance
        Solution solution = new Solution();

        // Call braceExpansionII with a sample expression and print the results
        List<String> results = solution.braceExpansionII("{a,b}c{d,e}f");
        for (String result : results) {
            System.out.println(result);
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <set>
#include <string>
#include <sstream>
using namespace std;

class Solution {
public:
    // Public method that initializes the process and returns the result as a vector of strings.
    vector<string> braceExpansionII(string expression) {
        // Start the depth-first search with the input expression.
        depthFirstSearch(expression);

        // Convert the set of strings to a vector and return it as a result.
        return vector<string>(expandedExpressions.begin(), expandedExpressions.end());
    }

private:
    set<string> expandedExpressions; // Set to store unique expanded expressions.

    // Helper method to perform depth-first search and generate all possible expansions.
    void depthFirstSearch(string exp) {
        // Find the first closing brace in the expression.
        int closingBraceIndex = exp.find_first_of('}');
        // If there are no closing braces left, insert the current expression to set and return.
        if (closingBraceIndex == string::npos) {
            expandedExpressions.insert(exp);
            return;
        }
        // Find the last opening brace before the found closing brace.
        int openingBraceIndex = exp.rfind('{', closingBraceIndex);
        // Separate the expression into three parts: before, inside, and after the braces.
        string beforeBraces = exp.substr(0, openingBraceIndex);
        string afterBraces = exp.substr(closingBraceIndex + 1);
        stringstream insideBracesStream(exp.substr(openingBraceIndex + 1, closingBraceIndex - openingBraceIndex - 1));
        string insideBraces;

        // Iterate over comma separated values inside the braces.
        while (getline(insideBracesStream, insideBraces, ',')) {
            // Recursively call dfs on the new expression formed by the combination of parts.
            depthFirstSearch(beforeBraces + insideBraces + afterBraces);
        }
    }
};

```

Typescript Solution

```typescript
// Recursive function to perform the depth-first search on the expression.
// It computes the different possible strings after expanding the braces.
const depthFirstSearch = (exp: string, result: Set<string>) => {
    // Find the position of the first closing brace '}'.
    const closingBraceIndex = exp.indexOf('}');
    if (closingBraceIndex === -1) {
        // If there are no more closing braces, add the current expression to the result.
        result.add(exp);
        return;
    }

    // Find the last opening brace '{' before the found closing brace '}'.
    const openingBraceIndex = exp.lastIndexOf('{', closingBraceIndex);

    // Divide the expression into three parts: before, inside, and after the braces.
    const beforeBrace = exp.substring(0, openingBraceIndex);
    const afterBrace = exp.substring(closingBraceIndex + 1);
    // Split the substring within the braces by ',' to get individual elements to expand.
    const insideBrace = exp.substring(openingBraceIndex + 1, closingBraceIndex).split(',');

    // Recursively process each element of the expansion.
    for (const element of insideBrace) {
        depthFirstSearch(beforeBrace + element + afterBrace, result);
    }
};

// Function for expanding all possible brace expressions and sorting them.
const braceExpansionII = (expression: string): string[] => {
    // Initialize a set to hold unique expanded strings.
    const resultSet: Set<string> = new Set();

    // Start the recursive depth-first search from the expression.
    depthFirstSearch(expression, resultSet);

    // Convert the set of strings to an array, sort it, and return.
    return Array.from(resultSet).sort();
};

```

Time and Space Complexity

Time Complexity

The provided code performs a depth-first search (DFS) to generate all possible combinations of strings that can result from the given `expression`, following the rules of brace expansion.

The time complexity of the algorithm is hard to analyze precisely without more context regarding the nature of `expression`. However, given that the code involves recursive calls for every substring result, we can make some general observations:

- Each recursive call is made after finding a pair of braces `{}` and for each element in the comma-separated list inside.
- For every split by comma inside the braces, a new recursive DFS call is made, therefore, in the worst case, the number of recursive calls for each level of recursion is `O(n)`, where `n` is the size of the split list.
- The maximum depth of recursion is related to the number of nested brace pairs. If we assume the maximum number of nested brace pairs is `k`, then to some degree we can say the depth of recursion is `O(k)`.
- The cost of string concatenation in Python is `O(m)`, where `m` is the length of the resulting string.

Considering these factors, in the worst-case scenario where the recursion is deep and the number of elements to combine at each level is large, the time complexity could grow exponentially with respect to the number of brace pairs and the number of individual elements within them. Thus, it can be considered `O(n^k)` or worse. This is a rough estimate as the actual time complexity can vary dramatically with the structure of the `expression`.

Space Complexity

The space complexity of the algorithm is also influenced by several factors:

- The recursion depth `k` contributes to the space complexity due to call stack usage. Hence, the implicit stack space is `O(k)`.
- The set `s` stores all unique combinations which can be at most `O(2^n)` where `n` is the number of elements in `expression`, since each element inside a brace can either be included or excluded.
- Each recursive call involves creating new strings, and if memory allocated for these strings is not efficiently handled by the Python's memory management, it could add to the space complexity.

Taking these into account, the worst-case space complexity can be considered `O(2^n + k)` for storing combinations and stack space, but similar to the time complexity, this heavily depends on the input `expression` structure.