320. Generalized Abbreviation

Medium | Bit Manipulation | String | Backtracking | Leetcode Link

Problem Description

The problem asks us to generate all possible generalized abbreviations for a given word. A generalized abbreviation for a word is a transformation where any number of non-overlapping and non-adjacent substrings can be replaced with their respective lengths. The key points to remember are:

- Substrings that we replace with numbers must not overlap with each other.
- Substrings that we replace must not be adjacent.
- We can use the whole word as a substring, which means the entire word can be replaced with a single number indicating its length.
- The returned list does not need to be in any specific order.

To understand it better, let's look at an example with the word "abc". Possible generalized abbreviations for "abc" include "abc" (no change), "1bc" (replacing "a" with "1"), "a2" (replacing "bc" with "2"), "ab1" (replacing "c" with "1"), "1b1" (replacing the first and last characters with "1"), and "3" (replacing the entire word with "3").

Intuition

The process to solve this problem can be approached using recursion and backtracking. We can think of each character in the word as a decision point: at each character, we can decide to:

- Replace a substring starting at this character with a number that represents its length.
- Skip the character to keep it as is.

While performing these operations, we must ensure that we are not violating the rules of replacing non-overlapping and non-adjacent substrings. Therefore, each time we replace a substring with its length, we skip the next character to maintain the non-adjacency condition.

To arrive at the solution, we start by writing a recursive function that can generate the aforementioned abbreviations by considering each character. This function should have the ability to record the current state (as an abbreviation in progress) and be able to backtrack to explore different abbreviation possibilities until all possibilities have been exhausted.

The algorithm works as follows:

1. At each recursion level, the function is given the remaining string to process and the current abbreviation state.
2. If the remaining string is empty, the current state is one of the solutions, so add it to the answer list.
3. For each possible length of the substring to be abbreviated (from 1 to the length of the remaining string):
   - Append the length number to the current state.
   - If there are characters remaining after the substring, append the next character to enforce non-adjacency, and recurse with the remaining string after this character.
   - Backtrack: remove the added elements from the current state to explore other abbreviation possibilities.
4. Also, consider keeping the current character as is and recurse with the remaining string after this character.

The solution to the problem emerges as we systematically explore each abbreviation possibility while adhering to the established rules.

Solution Approach

The implemented solution uses depth-first search (DFS) and backtracking to generate all possible generalized abbreviations of the input word. The code defines a recursive function `dfs` that accepts the remaining string `s` to process and a temporary list `t` which keeps track of the current abbreviation being constructed.

Here is a step-by-step walkthrough of the implementation:

1. **Base Case**: If the remaining string `s` is empty, it means we have reached the end of the string and have constructed a valid abbreviation. Hence, we join the elements in the temporary list `t` into a string and append it to `ans`, the final list of abbreviations.

2. **Recursive Case**:

   - We iterate over the range from `1` to `len(s) + 1` which represents the length of the substring we are considering for replacement.
   - For each length `i`, we have two cases to consider:
     - **Abbreviation**: We append the length `i` to `t` as a substring replacement. If `i` is less than the length of `s`, we also append the next character to `t` (to satisfy the non-adjacency condition) and then recursively call `dfs` with the remaining string `s[i+1:]`. After the recursive call, we backtrack by popping the added elements from `t`.
     - **No Abbreviation (End of Recursion)**: If the length `i` is equal to the length of `s`, we simply perform a recursive call without adding any more characters, as we would be considering the entire remaining string as one substring.
   - Additionally, we must consider the case where we do not replace the current character with its abbreviation. Therefore, we append the current character `s[0]` to `t` and recursively call `dfs` with `s[1:]`. We backtrack by popping the character after recursion.

The choice of algorithm and data structure:

- **Depth-First Search (DFS)**: Used to explore all possible abbreviation combinations by diving deeper into the recursive tree structure based on the decisions made (to replace or not to replace substrings).
- **Backtracking**: Used to undo the decision at each step (once all possibilities after that decision are explored) to backtrack and explore other paths in the decision tree.
- **Temporary List (`t`)**: It keeps track of the current state of the abbreviation and is modified in place during recursion to save space.
- **Final List (`ans`)**: Used to store all the possible abbreviation combinations generated.

Here is the code excerpt that illustrates the recursive approach:

```python
def dfs(s, t):
    if not s:
        ans.append(''.join(t))
        return
    for i in range(1, len(s) + 1):
        t.append(str(i))
        if i < len(s):
            t.append(s[i])
            dfs(s[i + 1:], t)
            t.pop()  # Backtrack
        else:
            dfs(s[i:], t)  # End of Recursion
        t.pop()  # Backtrack
    t.append(s[0])
    dfs(s[1:], t)
    t.pop()  # Backtrack
```

The function `generateAbbreviations` initializes the final answer `ans` and calls `dfs` with the initial word and an empty temporary list to start the process. After all the possible abbreviations are generated, the `ans` list is returned.

Example Walkthrough

Let's assume our given word is "word". We use the recursive implementation of the solution approach to generate all possible generalized abbreviations. We'll use a step-by-step walk through to illustrate how the algorithm works.

First, we call the `dfs` function with the full string "word" and an empty temporary list `t`.

1. In the first level of recursion, `s` is "word".

   - We consider all lengths of substrings starting from 1 to the length of "word".

   a. Length 1: We replace "w" with "1". - Temp list `t` becomes ["1"]. - We call `dfs` with the remaining string "ord" and `t` as ["1", "o"] (enforcing non-adjacency). - This recursion continues, similarly replacing one character at a time and enforcing non-adjacency, until the base case is reached and the possible abbreviations like "1o2", "1or1", and "1ord" are added to `ans`.

   b. Length 2: We replace "wo" with "2". - Temp list `t` becomes ["2"]. - We call `dfs` with the remaining string "rd" and `t` as ["2", "r"] (enforcing non-adjacency). - Following the similar pattern, "2r1" and "2rd" are added to `ans`.

   c. This pattern continues for length 3 and 4, adding "3d" and "4" respectively to `ans`.

2. Additionally, we have the option to keep "w" as is and recurse with the remaining string "ord".

   - Temp list `t` is ["w"].
   - This recursion follows the same pattern as before, exploring all options with the prefix "w" and generating abbreviations like "w1r1", "w2", "wo1d", "wor1", and "word".

These steps will systematically continue for each possibility, exploring further branches in the recursion tree and using backtracking to generate all possible abbreviations. Once all recursion paths are explored, we have our complete list `ans` with all generated generalized abbreviations. The recursive calls ensure that we don't create overlapping abbreviations and that no two numbers are adjacent, maintaining the problem's constraints.

The final output for the given word "word" will hence include entries like "word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "3d", "w3", and "4", among others.

Python Solution

```python
class Solution:
    def generateAbbreviations(self, word: str) -> List[str]:
        # Helper function to perform depth-first search to find all possible abbreviations
        def dfs(remaining, current_abbreviation):
            if not remaining:
                # No more characters left to abbreviate, add the current abbreviation to answers
                answers.append(''.join(current_abbreviation))
                return

            # Consider abbreviation for every possible length starting from each index
            for i in range(1, len(remaining) + 1):
                # Append the number (length of abbreviation)
                current_abbreviation.append(str(i))

                # If there are characters remaining, append the next character after abbreviation
                if i < len(remaining):
                    current_abbreviation.append(remaining[i])
                    dfs(remaining[i + 1:], current_abbreviation)
                    # Backtrack to previous state
                    current_abbreviation.pop()
                else:
                    # All characters are abbreviated, call dfs with an empty remaining string
                    dfs(remaining[i:], current_abbreviation)

                # Backtrack to consider next abbreviation possibility
                current_abbreviation.pop()

            # Consider not abbreviating the first character and recursively find further abbreviations
            current_abbreviation.append(remaining[0])
            dfs(remaining[1:], current_abbreviation)
            # Backtrack after exploring abbreviations with the first character included
            current_abbreviation.pop()

        answers = []  # List to store all possible abbreviations
        dfs(word, [])  # Start the depth-first search with the full word and empty current abbreviation
        return answers

```

Java Solution

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

class Solution {
    // Class variable to store the final list of abbreviations.
    private List<String> abbreviations;

    // Public method that initiates the generation of abbreviations for a word.
    public List<String> generateAbbreviations(String word) {
        abbreviations = new ArrayList<>();

        // Temporary list to keep track of current abbreviation progression.
        List<String> currentAbbreviation = new ArrayList<>();

        // Start the depth first search (DFS) from the beginning of the word.
        dfs(word, currentAbbreviation);

        // Return the list of all possible abbreviations after DFS is complete.
        return abbreviations;
    }

    // Helper method to perform depth first search to generate all abbreviations.
    private void dfs(String remainingSubstring, List<String> currentAbbreviation) {
        // If the remaining substring is empty, join all collected parts and add to result.
        if ("".equals(remainingSubstring)) {
            abbreviations.add(String.join("", currentAbbreviation));
            return;
        }

        // Try all possible lengths of numbers to abbreviate (up to the length of the remaining substring).
        for (int i = 1; i <= remainingSubstring.length(); ++i) {
            // Add number abbreviation for the current length.
            currentAbbreviation.add(Integer.toString(i));

            // If there are characters left, add the next character and continue DFS.
            if (i < remainingSubstring.length()) {
                currentAbbreviation.add(String.valueOf(remainingSubstring.charAt(i)));
                dfs(remainingSubstring.substring(i + 1), currentAbbreviation);

                // Backtracking - remove the last character to explore new paths.
                currentAbbreviation.remove(currentAbbreviation.size() - 1);
            } else {
                // If no characters are left, continue DFS with the current abbreviation.
                dfs(remainingSubstring.substring(i), currentAbbreviation);
            }

            // Backtracking - remove the number abbreviation to try a different abbreviation length.
            currentAbbreviation.remove(currentAbbreviation.size() - 1);
        }

        // Try keeping the first character as is and continue DFS for the rest of the string.
        currentAbbreviation.add(String.valueOf(remainingSubstring.charAt(0)));
        dfs(remainingSubstring.substring(1), currentAbbreviation);

        // Backtracking - remove the character to keep the path tree clean for new paths.
        currentAbbreviation.remove(currentAbbreviation.size() - 1);
    }
}

```

C++ Solution

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

using namespace std;

class Solution {
private:
    vector<string> abbreviations; // Vector to store the final list of abbreviations.

    // Helper method to perform depth-first search to generate all abbreviations.
    void dfs(const string& remainingSubstring, vector<string>& currentAbbreviation) {
        // If the remaining substring is empty, concatenate all collected parts and add to the result.
        if (remainingSubstring.empty()) {
            abbreviations.push_back(join(currentAbbreviation, ""));
            return;
        }

        // Try all possible lengths of numbers to abbreviate (up to the length of the remaining substring).
        for (int i = 1; i <= remainingSubstring.size(); ++i) {
            // Add number abbreviation for the current length.
            currentAbbreviation.push_back(to_string(i));

            // If there are characters left, add the next character and continue DFS.
            if (i < remainingSubstring.size()) {
                currentAbbreviation.push_back(string(1, remainingSubstring[i]));
                dfs(remainingSubstring.substr(i + 1), currentAbbreviation);

                // Backtracking - remove the last character to explore new paths.
                currentAbbreviation.pop_back();
            } else {
                // If no characters are left, continue DFS with the current abbreviation.
                dfs(remainingSubstring.substr(i), currentAbbreviation);
            }

            // Backtracking - remove the number abbreviation to try a different abbreviation length.
            currentAbbreviation.pop_back();
        }

        // Try keeping the first character as is and continue DFS for the rest of the string.
        currentAbbreviation.push_back(string(1, remainingSubstring[0]));
        dfs(remainingSubstring.substr(1), currentAbbreviation);

        // Backtracking - remove the character to keep the path tree clean for new paths.
        currentAbbreviation.pop_back();
    }

    // Helper method to join strings in a vector into a single string with a given delimiter.
    string join(const vector<string>& parts, const string& delimiter) {
        string result;

        // Iterating over each part and appending to the result string.
        for (const auto& part : parts) {
            if (&part != &parts[0]) { // Avoid adding delimiter before the first element.
                result += delimiter;
            }
            result += part;
        }
        return result;
    }

public:
    // Public method that initiates the generation of abbreviations for a word.
    vector<string> generateAbbreviations(const string& word) {
        abbreviations.clear();

        // Temporary vector to keep track of the current abbreviation progression.
        vector<string> currentAbbreviation;

        // Start the depth-first search (DFS) from the beginning of the word.
        dfs(word, currentAbbreviation);

        // Return the vector of all possible abbreviations after DFS is complete.
        return abbreviations;
    }
};

```

Typescript Solution

```typescript
// Global variable to store the final list of abbreviations.
let abbreviations: string[] = [];

// Function that initiates the generation of abbreviations for a word.
function generateAbbreviations(word: string): string[] {
    abbreviations = [];

    // Temporary array to keep track of current abbreviation progression.
    let currentAbbreviation: string[] = [];

    // Start the depth-first search (DFS) from the beginning of the word.
    dfs(word, currentAbbreviation);

    // Return the list of all possible abbreviations after DFS is complete.
    return abbreviations;
}

// Helper function to perform depth-first search to generate all abbreviations.
function dfs(remainingSubstring: string, currentAbbreviation: string[]): void {
    // If the remaining substring is empty, join all collected parts and add to the result.
    if (remainingSubstring === "") {
        abbreviations.push(currentAbbreviation.join(""));
        return;
    }

    // Try all possible lengths of numbers to abbreviate (up to the length of the remaining substring).
    for (let i = 1; i <= remainingSubstring.length; i++) {
        // Add number abbreviation for the current length.
        currentAbbreviation.push(i.toString());

        // If there are characters left, add the next character and continue DFS.
        if (i < remainingSubstring.length) {
            currentAbbreviation.push(remainingSubstring.charAt(i));
            dfs(remainingSubstring.substring(i + 1), currentAbbreviation);

            // Backtracking - remove the last character to explore new paths.
            currentAbbreviation.pop();
        } else {
            // If no characters are left, continue DFS with the current abbreviation.
            dfs("", currentAbbreviation);
        }

        // Backtracking - remove the number abbreviation to try a different abbreviation length.
        currentAbbreviation.pop();
    }

    // Try keeping the first character as is and continue DFS for the rest of the string.
    currentAbbreviation.push(remainingSubstring.charAt(0));
    dfs(remainingSubstring.substring(1), currentAbbreviation);

    // Backtracking - remove the character to keep the path tree clean for new paths.
    currentAbbreviation.pop();
}

```

Time and Space Complexity

The time complexity of this code can be analyzed by considering the number of different abbreviation combinations that can be generated for a word of length `n`. At every character position in the word, we have two choices: abbreviate by using a number or use the character itself. This results in a recursion tree with a maximum branch factor of `n`, as we can abbreviate up to `n` characters in one go, forming a sequence of overlapping subproblems. Therefore, the time complexity is `O(2^n)` since the recursion depth is `n` and at every level we are making two choices. However, because of the way the abbreviations are generated (with numbers that can represent multiple characters), the actual time complexity is less strictly bounded and is dependent on the structure of the recursion.

The space complexity is determined by the depth of the recursion stack and the space required to store the intermediate strings. In the worst case, the recursion depth is `n` and the intermediate strings can also be of length `n`, leading to a space complexity of `O(n)`.