784. Letter Case Permutation

Medium Bit Manipulation String Backtracking Leetcode Link

Problem Description

In this problem, we're given a string `s` consisting of alphabetical characters. We have the freedom to change each of these characters individually to either lowercase or uppercase, which can lead to various permutations of the string with different cases for each letter. The task is to generate all possible strings with these different case transformations and return them as a list in any order.

An example to illustrate this: for the string "a1b2", the output will be `["a1b2", "a1B2", "A1b2", "A1B2"]`.

This is a typical backtracking problem, as we'll consider each character and explore the possibilities of both keeping it as it is or changing its case, accumulating all different strings that can be formed by these choices.

Intuition

To achieve our solution, we can apply a method known as depth-first search (DFS), which is a type of backtracking algorithm. The DFS approach will help us traverse through all possible combinations of uppercase and lowercase transformations of the characters within the string. Here's the intuition breakdown for solving this problem:

1. **Start from the first character**: We begin the process with the first character of the string.
2. **Recursive exploration**: For each character, we have two choices (provided it's an alphabetical character and not a digit):
    - Keep the current character as is and proceed to the next one.
    - Change the current character's case (from lowercase to uppercase or vice versa) and move to the next one.
3. **Base case**: If we reach the end of the string (i.e., index `i` is equal to the string length), we've completed a permutation and it's added to the list of results.
4. **Backtracking**: After exploring one option (like keeping the character as is) for the current position, we backtrack to explore the other option (changing the case).
5. **Result accumulation**: As we hit the base case at different recursive call levels, we collect all permutations of the string.

This recursive approach works because it explores all possible combinations by trying each option at each step. The use of the bitwise XOR operation `^ 32` is a clever trick to toggle the case of an alphabetical character since converting between lowercase to uppercase (or vice versa) in ASCII entails flipping the 6th bit.

Solution Approach

The provided solution uses a Depth-First Search (DFS) algorithm along with a recursive helper function named `dfs`. This function is defined within the scope of the `letterCasePermutation` method.

Here's a step-by-step walk-through of the solution code:

1. A new list `t` is created from the original string `s` which is going to hold the characters that we are currently exploring through DFS.
2. An empty list `ans` is initialized which will eventually contain all the possible permutations of the string.
3. The `dfs` function defined inside the `letterCasePermutation` takes a single argument `i`, which is the index of the character in `t` we are currently exploring.
4. In the body of the `dfs` function, the base case checks if the index `i` is greater than or equal to the length of `s`. If it is, it means we've constructed a valid permutation of string `s` in the list `t` and we append this permutation to `ans`.
5. If the base case condition is not met, we first call `dfs(i + 1)` without altering the current character. This recurses on the decision to keep the current character as is.
6. Next, if the current character is alphabetic (`t[i].isalpha()`), we toggle its case using `t[i] = chr(ord(t[i]) ^ 32)`. The `ord` function gets the ASCII value of the character, `^ 32` toggles the 6th bit which changes the case, and `chr` converts the ASCII value back to the character. After changing the case, we call `dfs(i + 1)` again, exploring the decision tree path where the current character's case is toggled.
7. By recursively calling `dfs` with 'unchanged' and 'changed' states of each character, all combinations are eventually visited, and the resulting strings appended to `ans`.

The DFS here is a recursive pattern that explores each possibility and undoes (backtracks) that choice before exploring the next possibility. The data structure used is a basic Python list which conveniently allows us to both change individual elements in place and join them into strings when we need to add a permutation to the results.

Example Walkthrough

Let's illustrate the solution approach with a small example using the string "0a1".

1. We start by setting up an empty list `ans` that will store our final permutations and a list `t` from the string `s`, which is `"0a1"`. List `t` will hold the characters that we are currently exploring through the DFS algorithm.
2. Since the first character '0' is not an alphabetical character, we only have one option for it; we keep it as is. So, we move on to the next character.
3. The next character is 'a'. Here, we have two choices:
    - Keep 'a' as it is and proceed to the next character, which will be '1'. This sequence is "0a1". Since '1' is not alphabetical, we keep it as is and add "0a1" to `ans`.
    - Toggle the case of 'a' to 'A' and then proceed to the next character, '1'. This sequence is "0A1". Again, we keep '1' as it is and add "0A1" to `ans`.
4. At this point, the `dfs` function has recursively explored and returned from both branches for character 'a', and the two permutations of the string are saved in `ans`, which now contains `["0a1", "0A1"]`.
5. Since there are no more characters to explore, the recursion terminates, and the `ans` list now holds all the possible permutations regarding the case of alphabetic characters.

And that's the end of the example. The original list "0a1" only had one alphabetical character that could be toggled, which resulted in two different string permutations: "0a1" and "0A1".

Python Solution

```python
class Solution:

    def letterCasePermutation(self, s: str) -> List[str]:
        # Helper function for depth-first search
        def dfs(index):
            # Base case: if the current index is equal to the length of the string
            if index == len(s):
                # Append the current permutation to the answer list
                permutations.append(''.join(current_string))
                return
            # Recursive case: move to the next character without changing the case
            dfs(index + 1)

            # If the current character is alphabetic, we can change its case
            if current_string[index].isalpha():
                # Flip the case: if the current character is uppercase, make it lowercase and vice versa
                current_string[index] = chr(ord(current_string[index]) ^ 32)
                # Continue the search with the flipped case character
                dfs(index + 1)
                # Revert the change for backtracking purposes
                current_string[index] = chr(ord(current_string[index]) ^ 32)

        # Convert the input string to a list, making it mutable
        current_string = list(s)
        # Initialize a list to store all possible permutations
        permutations = []
        # Start the depth-first search from the first character
        dfs(0)
        # Return the computed permutations
        return permutations

# Example of usage:
# sol = Solution()
# result = sol.letterCasePermutation("a1b2")
# print(result)  # Output: ["a1b2", "a1B2", "A1b2", "A1B2"]
```

Java Solution

```java
class Solution {
    private List<String> permutations = new ArrayList<>(); // List to hold the string permutations
    private char[] charArray; // Character array to facilitate permutation generation

    // Public method to generate letter case permutations for a given string
    public List<String> letterCasePermutation(String s) {
        charArray = s.toCharArray(); // Convert string to character array for modification
        generatePermutations(0); // Start recursion from the first character
        return permutations; // Return all generated permutations
    }

    // Helper method to recursively generate permutations
    private void generatePermutations(int index) {
        // Base case: If the end of the array is reached, add the current permutation to the list
        if (index >= charArray.length) {
            permutations.add(new String(charArray));
            return;
        }
        // Recursively call the method to handle the next character without modification
        generatePermutations(index + 1);

        // If the current character is alphabetic, toggle its case and proceed with recursion
        if (Character.isLetter(charArray[index])) {
            // Toggle the case: lowercase to uppercase or vice versa using XOR operation with the space character
            charArray[index] ^= ' ';
            generatePermutations(index + 1);

            // IMPORTANT: revert the character back to its original case after the recursive call
            charArray[index] ^= ' '; // This step ensures the original structure is intact for future permutations
        }
        // If the character is not alphabet, it is left as-is and the recursion takes care of other characters
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to generate all permutations of a string where each letter can be lowercase or uppercase
    std::vector<std::string> letterCasePermutation(std::string str) {
        // Resultant vector to store all permutations
        std::vector<std::string> permutations;

        // Utility lambda function to perform a Depth-First Search (DFS) for all possible letter case combinations
        std::function<void(int)> dfs = [&](int index) {
            // Base case: if we reach the end of the string, add the current permutation to the results
            if (index >= str.size()) {
                permutations.emplace_back(str);
                return;
            }

            // Recursive case: continue with the next character without changing the current one
            dfs(index + 1);

            // Only if the current character is alphabetical (either uppercase or lowercase)
            // we toggle its case and recursively continue
            if (isalpha(str[index])) {
                // Toggle the case of the current character: lowercase to uppercase or vice versa
                // XOR with 32 takes advantage of the fact that the ASCII values of upper and lower case
                // letters differ by exactly 32
                str[index] ^= 32;

                // Recursive call with the case of the current character toggled
                dfs(index + 1);

                // Optional: Toggle the case back if you want to use the original string elsewhere
                // after this function call. If str is not used after this, you can omit this step.
                // str[index] ^= 32;
            }
        };

        // Start DFS from index 0
        dfs(0);

        // Return all generated permutations
        return permutations;
    }
};
```

Typescript Solution

```typescript
// Function that generates all possible permutations of a string
// with letters toggled between uppercase and lowercase.
// It uses Depth-First Search (DFS) to explore all variations.
function letterCasePermutation(s: string): string[] {
    const length = s.length;                 // The length of the input string
    const charArray = [...s];                // Converts the string to an array of characters
    const results: string[] = [];            // Stores the resulting permutations

    // Helper function to perform DFS
    const dfs = (index: number) => {
        if (index === length) {              // If the end of the string is reached,
            results.push(charArray.join('')); // join the characters to form a permutation and add to results.
            return;
        }
        dfs(index + 1);                      // Recursive call with the next character (no change)

        // Check if the current character is a letter and can change case
        if (isLetter(charArray[index])) {
            toggleCase(charArray, index);    // Toggle the case of the character at the current index
            dfs(index + 1);                  // Recursive call with the toggled character
        }
    };

    // Function to check if a character is a letter (could be replaced with regex or other methods)
    function isLetter(c: string): boolean {
        return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
    }

    // Function that toggles the case of a letter by using bitwise XOR with the space character (32)
    function toggleCase(arr: string[], index: number): void {
        arr[index] = String.fromCharCode(arr[index].charCodeAt(0) ^ 32);
    }

    dfs(0); // Start the DFS from the first character
    return results; // Return the array of permutations
}
```

Time and Space Complexity

The given Python code generates all possible permutations of a string `s` where the permutation can include lowercase and uppercase characters. A depth-first search (DFS) algorithm is employed to explore all possibilities.

Time Complexity:

To analyze the time complexity, let us consider the number of permutations possible for a string with `n` characters. Each character can either remain unchanged or change its case if it's an alphabetic character. For each alphabetic character, there are 2 possibilities (uppercase or lowercase), while non-alphabetic characters have only 1 possibility (as they don't have case). If we assume there are `m` alphabetic characters in the string, there would be `2^m` different permutations.

The DFS function is recursively called twice for each alphabetic character (once without changing the case, once with the case changed), and it's called once for each non-alphabetic character. Thus, the time complexity of this process can be represented as:

`O(2^m * n)`, where `n` is the length of the string and `m` is the number of alphabetic characters in the string as each permutation includes building a string of length `n`.

Space Complexity:

The space complexity includes the space needed for the recursion call stack as well as the space for storing all the permutations. The maximum depth of the recursive call stack would be `n`, as that's the deepest the DFS would go. The space needed for `ans` will be `O(2^m * n)` as we store `2^m` permutations, each of length `n`.

However, one might consider the space used for each layer of the recursion separately from the space used to store the results since the latter is part of the problem's output. If we only consider the space complexity for the recursion (excluding the space for the output), then the space complexity would be `O(n)`.

Combining both the space for the recursion and the output, the space complexity can be represented as:

`O(2^m * n + n)`, which simplifies to `O(2^m * n)` when considering `2^m` to be significantly larger than `n`.

In conclusion, the time complexity is `O(2^m * n)` and the space complexity is `O(2^m * n)` when accounting for both the recursion stack and the output list.