17. Letter Combinations of a Phone Number

Medium Hash Table String Backtracking

Leetcode Link

Problem Description

The problem presents a scenario where we're given a string that contains digits from `2-9`. The task is to generate all possible letter combinations that the number could represent, similar to the way multi-press text entry worked on old phone keypads. Each number corresponds to a set of letters. For instance, '2' maps to "abc", '3' maps to "def", and so on, just as they do on telephone buttons. We should note that the digit '1' does not map to any letters. The goal here is to return all the possible letter combinations in any order.

Intuition

The intuition behind the solution is that we can solve the problem using a form of backtracking—generating all the possible combinations by traversing over the input digits and mapping them to the corresponding letters. We start with an initial list containing just an empty string, which represents the starting point of our combination. For each digit in the input string, we look up the corresponding string of characters it can represent and then generate new combinations by appending each of these letters to each of the combinations we have so far.

The solution uses a bread-first search-like approach (without using a queue). Initially, since we only have one combination (an empty string), for the first digit, we will generate as many new combinations as there are letters corresponding to that digit. For each subsequent digit, we take all the combinations generated so far and expand each by the number of letters corresponding to the current digit. This process is repeated until we have processed all digits in the input. The beauty of this approach is that it incrementally builds up the combination list with valid letter combinations corresponding to the digits processed so far, and it ensures that all combinations of letters for the digits are covered by the end.

Solution Approach

The implemented solution makes use of a list comprehension, which is a compact way to generate lists in Python. Here's a step-by-step walk-through of the approach:

1. Initialize a list `d` with strings representing the characters for each digit from '2' to '9'. For example, `d[0]` is `'abc'` for the digit '2', `d[1]` is `'def'` for the digit '3', and so on.
2. Start with a list `ans` containing an empty string. The empty string serves as a starting point for building the combinations.
3. Loop through each digit in the given `digits` string.
   - Convert the digit to an integer and subtract 2 to find the corresponding index in the list `d` (since the list `d` starts with the letters for '2').
   - Retrieve the string `s` representing the possible characters for that digit.
   - Generate new combinations by taking every element `a` in `ans` and concatenating it with each letter `b` in the string `s`. This is done using a list comprehension: `[a + b for a in ans for b in s]`.
   - Replace `ans` with the newly generated list of combinations.
4. After the loop finishes, `ans` contains all of the letter combinations that the input number can represent, and the function returns `ans`.

It's interesting to note that this solution has a linear time complexity with respect to the number of digits in the input, but the actual complexity depends on the number of letters that each digit can map to, since the size of `ans` can grow exponentially with the number of digits.

Example Walkthrough

Let's take "23" as an example to illustrate the solution approach.

1. We initialize a list `d` representing the characters for each digit from '2' to '9'. Thus, `d[0] = 'abc'` for '2' and `d[1] = 'def'` for '3'.
2. We start with a list `ans` containing one element, an empty string: `ans = [""]`.
3. Loop through each digit in "23":
   - For the first digit '2':
     - Convert '2' to an integer and subtract 2 to find the corresponding index in list `d` which is `0`.
     - Retrieve the string `s` which is `'abc'`.
     - Generate new combinations by appending each letter in `'abc'` to the empty string in `ans`. So `ans` becomes `["a", "b", "c"]`.
   - For the second digit '3':
     - Convert '3' to an integer and subtract 2 to find the corresponding index in list `d` which is `1`.
     - Retrieve the string `s` which is `'def'`.
     - Generate new combinations by concatenating each element in `ans` ("a", "b", "c") with each letter in `'def'`. Using list comprehension, the new `ans` becomes:

       ```
       [
         "a" + "d", "a" + "e", "a" + "f",
         "b" + "d", "b" + "e", "b" + "f",
         "c" + "d", "c" + "e", "c" + "f"
       ]
       ```

     - So now `ans` is `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`.
4. After processing both digits, `ans` contains all possible letter combinations for "23". Hence, the final output is `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`.

This example demonstrates the backtracking nature of the solution where each step builds upon the previous steps' results, expanding them with all possible letter combinations for the next digit.

Python Solution

```python
from typing import List

class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        # If the input string is empty, return an empty list
        if not digits:
            return []

        # Mappings of digits to letters as found on a phone dialpad
        digit_to_chars = ['abc', 'def', 'ghi', 'jkl', 'mno', 'pqrs', 'tuv', 'wxyz']

        # Initialize the list of combinations with an empty string
        combinations = ['']

        # Iterate through each digit in the input string
        for digit in digits:
            # Determine the corresponding letters from the mapping
            # Subtract 2 because 'abc' corresponds to digit '2'
            letters = digit_to_chars[int(digit) - 2]

            # Build new combinations by appending each possible letter to each existing combination
            combinations = [prefix + letter for prefix in combinations for letter in letters]

        # Return the list of all letter combinations
        return combinations

```

Java Solution

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

class Solution {

    // Let's declare a method to generate all possible letter combinations for a given phone number.
    public List<String> letterCombinations(String digits) {
        // A result list to store the final combinations.
        List<String> result = new ArrayList<>();

        // Return an empty list if the input digits string is empty.
        if (digits.length() == 0) {
            return result;
        }

        // Add an empty string as an initial value to start the combinations.
        result.add("");

        // Mapping of digit to letters as seen on a phone keypad.
        String[] digitToLetters = new String[] {"abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

        // Iterate over each digit in the input string.
        for (char digit : digits.toCharArray()) {
            // Get the corresponding letters for the current digit.
            String letters = digitToLetters[digit - '2'];

            // Temp list to hold the combinations for the current digit.
            List<String> temp = new ArrayList<>();

            // Combine each result in the list with each letter for the current digit.
            for (String combination : result) {
                for (char letter : letters.toCharArray()) {
                    // Add the new combination to the temp list.
                    temp.add(combination + letter);
                }
            }

            // Update the result list with the new set of combinations.
            result = temp;
        }

        // Return the complete list of combinations.
        return result;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Generates all combinations of letters by mapping the digits to corresponding letters on a phone keypad
    std::vector<std::string> letterCombinations(std::string digits) {
        // If the input string is empty, return an empty vector
        if (digits.empty()) return {};

        // Create a mapping for the digits to their corresponding letters
        std::vector<std::string> digitToChars = {
            "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"
        };

        // Initialize the answer vector with an empty string to start the combinations
        std::vector<std::string> combinations = {""};

        // Loop through each digit in the input string
        for (char digit : digits) {
            // Get the string that corresponds to the current digit
            std::string chars = digitToChars[digit - '2'];

            // Temporary vector to store the new combinations
            std::vector<std::string> tempCombinations;

            // Loop through each combination we have so far
            for (std::string &combination : combinations) {
                // Append each character that corresponds to the current digit
                for (char ch : chars) {
                    tempCombinations.push_back(combination + ch);
                }
            }

            // Replace the combinations with the updated ones
            combinations = std::move(tempCombinations);
        }
        // Return the vector containing all combinations
        return combinations;
    }
};

```

Typescript Solution

```typescript
// Mapping of digits to corresponding letters as found on a phone dial pad.
const digitToLetterMap: { [digit: string]: string[] } = {
    '2': ['a', 'b', 'c'],
    '3': ['d', 'e', 'f'],
    '4': ['g', 'h', 'i'],
    '5': ['j', 'k', 'l'],
    '6': ['m', 'n', 'o'],
    '7': ['p', 'q', 'r', 's'],
    '8': ['t', 'u', 'v'],
    '9': ['w', 'x', 'y', 'z'],
};

/**
 * Generates all possible letter combinations that the number could represent.
 * @param digits - A string containing digits from 2-9.
 * @returns An array of strings representing all possible letter combinations.
 */
function letterCombinations(digits: string): string[] {
    const lengthOfDigits = digits.length;
    // Base case: if the input string is empty, return an empty list.
    if (lengthOfDigits === 0) {
        return [];
    }
    // List to store the combinations.
    const combinations: string[] = [];

    /**
     * Helper function to generate combinations using depth-first search.
     *
     * @param index - The current index in the input digit string.
     * @param currentStr - The current combination of letters being built.
     *
     */
    const generateCombinations = (index: number, currentStr: string) => {
        // If the current combination is the same length as the digits string, add to results.
        if (index === lengthOfDigits) {
            combinations.push(currentStr);
            return;
        }
        // Loop through the letters mapped to the current digit and recurse for the next digit.
        for (const char of digitToLetterMap[digits[index]]) {
            generateCombinations(index + 1, currentStr + char);
        }
    };

    // Start the recursion with the first digit.
    generateCombinations(0, '');
    // Return the list of generated combinations.
    return combinations;
}

```

Time and Space Complexity

The given Python code generates all possible letter combinations that each number on a phone map to, based on a string of digits. Here is an analysis of its time and space complexity:

- **Time Complexity:** Let n be the length of the input string `digits`. The algorithm iterates through each digit, and for each digit, iterates through all the accumulated combinations so far to add the new letters.

  For a digit `i`, we can have at most 4 letters (e.g., for digits mapping to 'pqrs', 'wxyz') which increases the number of combinations by a factor of up to 4 each time. Thus, in the worst case scenario, the time complexity can be expressed as $O(4^n)$, as each step could potentially multiply the number of combinations by 4.

- **Space Complexity:** The space complexity is also $O(4^n)$ because we store all the possible combinations of letters. Although the list `ans` is being used to keep track of intermediate results, its size grows exponentially with the number of digits in the input. At each step, the new list of combinations is up to 4 times larger than the previous one until all digits are processed.