93. Restore IP Addresses

Medium String Backtracking

Leetcode Link

Problem Description

The task is to generate all possible valid IP addresses from a given string containing only digits. A valid IP address has four integer sections separated by dots, and each integer must be within the range of 0 to 255. Importantly, the integers must not have leading zeros, except for the digit "0" itself. To solve this problem, it is necessary to insert dots in the correct positions within the given string without changing the order of digits to create valid IP addresses. Each section of the IP address is essentially a substring of the original string. The challenge is to determine the partitions that yield a valid sequence of four integers, respecting the constraints stated above.

Intuition

To arrive at the solution, we adopt a depth-first search (DFS) strategy, exploring each potential segment step by step and backtracking if a segment turns out to be invalid. The DFS algorithm tries to place the dots in all allowable positions, and upon placing each dot, it checks if the resulting segment is a valid integer between 0 and 255. It is important to note that a leading zero is permissible if and only if the segment is a single zero.

The steps of the algorithm are as follows:

1. Proceed with the DFS traversal from the first digit of the string.
2. In each function call, iterate over the possible lengths for the next segment, which could be 1, 2, or 3 digits long.
3. At each iteration, check if the segment is a valid integer within the IP address criteria (0 to 255 and no leading zeros, unless the integer is 0).
4. If the segment is valid, add it to a temporary list and recursively call the function with the updated index.
5. If we reach a point where all four segments are valid and we have used all digits of the string, we add the combination to the result list.
6. If at any point the segment is invalid or we've got 4 segments but the string is not fully used, we backtrack and try another segment length or configuration.

By exploring each possible dot placement and segment length, and ensuring that each constitutes a valid part of the IP address, we can guarantee that all valid IP addresses formed from the input string are found.

Solution Approach

The provided solution uses a depth-first search (DFS) recursive algorithm to explore all possible combinations of the digits in the string that could result in a valid IP address. The following is a detailed explanation of the implementation:

- **Recursive Function (DFS)**: The `dfs(i: int)` function is a recursive function that takes one argument, `i`, which represents the current index in the string `s`. This function is called recursively to explore all potential valid IP segments starting at index `i`.
- **Base Case and Result Collection**: In the recursive function, there are two base cases. One checks if we are at the end of the string (`i == n`) and have exactly four segments in the temporary list (`len(t) == 4`), which is a condition for a valid IP address. If these conditions are met, we join the segments with dots and append the result to the answer list `ans`. The other base case handles the situation where reaching the end of the string without having exactly four segments or having four segments but not using all characters, which leads to termination of the current path.
- **Checking for Valid Segments**: A helper function `check(s: int, j: int)` is used to determine if the substring from index `s` to `j` (inclusive) constitutes a valid IP segment. The check function accounts for leading zeros and the valid integer range. A leading zero is only valid if it is a singular zero (e.g., "0"). Any segment leading with a "0" and having additional digits (e.g., "01") is invalid.
- **Exploration and Backtracking**: The core of the DFS is a for-loop that iterates from the current index `i` to the minimum between `i + 3` and `n` (the length of the string). This accounts for the fact that IP address segments have at most three digits. If a valid segment (substring) is found using the `check` function, it's added to the current temporary list `t`, and the DFS continues from the next index `j + 1`. After the recursive call returns, the added segment is removed (backtracking), and the loop continues to explore the next possible segment.
- **Parallel Variables**: The solution maintains several variables in parallel: `n`, which is the length of the input string `s`; `ans`, the list holding all the valid IP addresses found; and `t`, which is a temporary list used to store the current segments of the IP address being explored.
- **Initiation of DFS**: The process begins by calling `dfs(0)`, which triggers the recursive exploration from the first character in the string.

The DFS pattern used here is crucial not only to generate all possible segmentations but also to ensure efficiency by halting early in paths that cannot lead to valid IP addresses. The use of recursion with backtracking enables the algorithm to explore different segment lengths and combinations effectively.

Example Walkthrough

Let's consider a simple example to illustrate how the solution works with the given string "255255111135".

1. Begin by calling the recursive DFS function from the first character: `dfs(0)`.
2. In the first level of recursion, start with an empty list `t = []`. Iterate over string lengths 1-3 starting from the first character `s[0]` as long as we remain within the string length (dot placement possibilities), here let's try all three:
   - Try taking "2" as the first segment: `t = ["2"]`, call `dfs(1)`.
   - Try taking "25" as the first segment: `t = ["25"]`, call `dfs(2)`.
   - Try taking "255" as the first segment: `t = ["255"]`, call `dfs(3)`.
3. Following the depth-first search strategy, let's dig into the case where the first segment is "255": `dfs(3)`.
4. With `t = ["255"]`, proceed with the next character `s[3]`, trying segment lengths of 1-3.
   - Try taking "2" as the next segment: `t = ["255", "2"]`, call `dfs(4)`.
   - Try taking "25" as the next segment: `t = ["255", "25"]`, is invalid as "25525" is not a possible IP address. Backtracking occurs, no recursive call is made.
   - Try taking "255" as the next segment: `t = ["255", "255"]`, which is valid, so proceed with `dfs(6)`.
5. Now consider `t = ["255", "255"]`. At `dfs(6)`, try out different length again.
   - Taking "1" as the next segment, `t = ["255", "255", "1"]`, call `dfs(7)`.
   - Taking "11" as the next segment, `t = ["255", "255", "11"]`, call `dfs(8)`.
   - Taking "111" as the next segment is not possible here since we would run out of characters for the last segment. No recursive call happens.
6. Consider the state `t = ["255", "255", "1"]`. Now call `dfs(7)` and follow the same process.
   - Taking "3" as the next segment: `t = ["255", "255", "1", "3"]` and calling `dfs(8)`.
   - Taking "35" as the next segment: `t = ["255", "255", "1", "35"]` is valid but doesn't use all characters, no recursive call is made.
   - Taking "135" as the final segment is not possible since the start index would be greater than string length. No recursive call is made.
7. In `dfs(8)` with `t = ["255", "255", "1", "3"]`, we are at the end of the string, so join `t` with dots to get "255.255.1.3" and add this to the answer list `ans`.
8. Now backtrack to the state `t = ["255", "255", "1"]` and attempt the next possible recursion which would have been with the segment "35" or "135". Both these were ruled out in step 6.
9. Continue backtracking to previous recursion levels and trying different segment lengths and combinations in this manner until all possibilities have been exhausted.

By the end of the described example, we will have all valid IP addresses that can be formed from the given string stored in the `ans` list.

Python Solution

```python
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        # Helper function to check if the part of the string can be a valid IP address segment
        def is_valid_segment(start: int, end: int) -> bool:
            if start > end or (s[start] == "0" and end - start > 0):  # Leading zero is not allowed unless the segment is '0'
                return False
            return 0 <= int(s[start: end + 1]) <= 255  # Check if the segment is in the range 0-255

        # Recursive function to generate all possible valid IP addresses
        def backtrack(current_index: int):
            # If at the end of the string and exactly 4 segments are found, add the IP to the solutions
            if current_index == segment_length and len(current_ip) == 4:
                solutions.append(".".join(current_ip))
                return
            if current_index == string_length or len(current_ip) == 4:  # If it's not a valid IP, just return
                return
            # Try to form a valid segment by choosing 1 to 3 digits
            for length in range(1, min(4, segment_length - current_index + 1, string_length)):
                if is_valid_segment(current_index, j):
                    current_ip.append(s[current_index: j + 1])  # Add the new segment
                    backtrack(j + 1)  # Recurse for the next segment
                    current_ip.pop()  # Backtrack and remove the last segment before the next iteration

        string_length = len(s)  # Store the length of the string
        solutions = []  # This will store all the valid IP addresses
        current_ip = []  # Temporary list to store the parts of the current IP address
        backtrack(0)  # Start the recursive backtrack function
        return solutions

# Example usage
# sol = Solution()
# valid_ips = sol.restoreIpAddresses("255255111135")
# print(valid_ips) # Output would be ["255.255.11.135", "255.255.111.35"]
```

Java Solution

```java
class Solution {
    private int stringLength; // Length of the input string
    private String inputString; // The input string representing the digits of the IP address
    private List<String> validIPAddresses = new ArrayList<>(); // List to hold the valid IP addresses
    private List<String> currentSegment = new ArrayList<>(); // List to hold the current segments of the IP address being construct

    // Public method to restore IP addresses from the given string.
    public List<String> restoreIpAddresses(String s) {
        stringLength = s.length();
        inputString = s;
        backtrack(0, 0); // Begin the depth-first search (DFS) from the first character of the string
        return validIPAddresses;
    }

    // Helper method to perform a DFS to build all valid IP addresses.
    private void backtrack(int index, int segmentCount) {
        // Check if we have processed the entire string and have exactly 4 segments
        if (index == stringLength && currentSegment.size() == 4) {
            // Join the segments and add the resulting IP address to the list
            validIPAddresses.add(String.join(".", currentSegment));
            return;
        }
        // If we've processed the entire string or have more than 4 segments, backtrack
        if (index == stringLength || currentSegment.size() == 4) {
            return;
        }

        // Initialize an integer to store the numeric value of current segment
        int segmentValue = 0;
        // Consider 1 to 3 digit long segments (as an IP segment ranges from 0 to 255)
        for (int j = index; j < Math.min(index + 3, stringLength); ++j) {
            segmentValue = segmentValue * 10 + inputString.charAt(j) - '0'; // Convert current segment to integer

            // Check for leading zeroes and if segmentValue is greater than 255
            if (segmentValue > 255 || (inputString.charAt(index) == '0' && index != j)) {
                break; // If any of these checks fail, stop exploring further and backtrack
            }

            // Add the current segment to our list and continue the search
            currentSegment.add(inputString.substring(index, j + 1));
            backtrack(j + 1, segmentCount + 1);
            currentSegment.remove(currentSegment.size() - 1); // Remove the last added segment to backtrack
        }
    }
}
```

C++ Solution

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

class Solution {
public:
    // Restore IP Addresses from a string.
    std::vector<std::string> restoreIpAddresses(std::string s) {
        int stringLength = s.size(); // Store the length of the input string.
        std::vector<std::string> validIPs; // Store all valid IP addresses.
        std::vector<std::string> currentIP; // Store the current IP sections.

        // Define the recursive function to perform depth-first search.
        std::function<void(int)> dfs = [&](int index) {
            // If index has reached the end of the string and the current IP has exactly 4 parts, a valid IP is found.
            if (index == stringLength && currentIP.size() == 4) {
                validIPs.push_back(currentIP[0] + "." + currentIP[1] + "." + currentIP[2] + "." + currentIP[3]);
                return;
            }
            // If index is beyond the string length or more than 4 parts are formed, there's no valid IP.
            if (index >= stringLength || currentIP.size() >= 4) {
                return;
            }
            int segmentValue = 0; // Value of the current IP part.

            // Iterate through the string, ensuring to not go beyond length and within the 3 characters segment limit.
            for (int j = index; j < std::min(stringLength, index + 3); ++j) {
                segmentValue = segmentValue * 10 + s[j] - '0'; // Calculate the segment value.

                // If the segment value is more than 255 or if it starts with a '0' but is not single digit, it's invalid.
                if (segmentValue > 255 || (j > index && s[index] == '0')) {
                    break;
                }
                // Add the current segment to the currentIP and continue with the next part.
                currentIP.push_back(s.substr(index, j - index + 1));
                dfs(j + 1); // Recurse to the next part of the string.
                currentIP.pop_back(); // Backtrack to explore other possibilities.
            }
        };
        dfs(0); // Start DFS from the 0th index of the input string.
        return validIPs; // Return all the valid IP addresses.
    }
};
```

Typescript Solution

```typescript
function restoreIpAddresses(s: string): string[] {
    const lengthOfString = s.length;
    const validIPAddresses: string[] = []; // This will hold all the valid IP addresses
    const currentIPParts: string[] = []; // Temporarily stores the current parts of the IP address

    // Helper function to perform a depth-first search for IP addresses segments starting from index 'idx'
    const findIPAddresses = (idx: number) => {
        // Check if we have four numbers and we have exactly 4 parts for an IP address
        if (idx == lengthOfString && currentIPParts.length == 4) {
            result.push(currentIPParts.join(".")); // Join the parts and add them to the result list
            return;
        }

        // If we have already 4 parts or have traversed the string, no need to proceed further
        if (idx >= lengthOfString || currentIPParts.length >= 4) {
            return;
        }

        let currentSegmentValue = 0;
        for (let j = idx; j < idx + 3 && j < lengthOfString; ++j) {
            // Compute the current segment value
            currentSegmentValue = currentSegmentValue * 10 + s[j].charCodeAt(0) - '0'.charCodeAt(0);

            // Check if the current segment value is greater than 255 or it has a leading zero
            if (currentSegmentValue > 255 || (j > idx && s[idx] === '0')) {
                break;
            }

            // Add the current segment to the current IP parts and proceed recursively
            currentIPParts.push(currentSegmentValue.toString());
            findIPAddresses(j + 1); // Recur for the next part of the string
            currentIPParts.pop(); // Backtrack to try different segment combinations
        }
    };

    // Start the depth-first search from the first index
    findIPAddresses(0);
    // Return the list of valid IP addresses found
    return result;
}
```

Time and Space Complexity

The given Python code defines a method to restore possible IP addresses from a string by implementing a depth-first search (DFS) algorithm.

Time Complexity

The time complexity of the algorithm can be considered as $O(1)$ in terms of the input string's length, since an IP address consists of 4 parts, and each part can have a maximum of 3 digits. The check function is called at each step of the DFS and runs in $O(1)$ time since it operates on a constant size substring (at most 3 characters).

The DFS function will attempt to place a dot after every 1 to 3 digits, but since IP addresses are fixed length (4 parts of at most 3 digits each), the maximum depth of the recursive call stack will be 4, and there will be at most 3^4 possible combinations to check (3 choices at each of the 4 levels of the decision tree). This results in a total of 81 iterations in the worst case, each taking constant time.

Thus, the overall time complexity is $O(1)$ since the size of the input is not a factor beyond a certain length (the length must be between 4 and 12 for a valid IP address).

Space Complexity

The space complexity of the solution mainly depends on the size of the recursive call stack and the space used to store the intermediate and final solutions. As previously mentioned, the recursive call stack will have at most 4 levels due to the nature of IP addresses. Plus, a single path `t` in the recursion tree is a list that can have at most 4 strings, each up to 3 characters long.

The list `ans` will contain all the valid IP addresses we find. In the worst-case scenario, every partitioning will lead to a valid IP address, but this is highly unlikely. However, if we consider every single character as a digit and each digit forms a valid part of an IP address, the maximum number of valid IP addresses would be 3^4 (though actually it would be less due to the leading zero and value >255 restrictions).

Hence, the space complexity for the output list is $O(1)$, and the overall space complexity of the algorithm including the recursive call stack and the temporary list `t` is also $O(1)$, since the problem's constraints limit the input size and, consequently, the recursion depth and output size.