2375. Construct Smallest Number From DI String

Medium Stack Greedy String Backtracking Leetcode Link

Problem Description

In this problem, you're given a string pattern consisting of characters 'I' and 'D'. The 'I' stands for "increasing" and 'D' for "decreasing". Based on this pattern, you need to create another string `num` that follows these rules:

1. String `num` is formed by the digits '1' to '9', each digit can only be used once.
2. If `pattern[i] == 'I'`, then the ith element of `num` must be less than the (i+1)th element.
3. If `pattern[i] == 'D'`, then the ith element of `num` must be greater than the (i+1)th element.

The goal is to find the lexicographically smallest string `num` that satisfies the given pattern.

Intuition

To solve this problem, we can use a backtracking approach which is a type of depth-first search (DFS). The main idea is to build the string `num` one digit at a time, choosing the smallest possible digit at each step that satisfies the pattern requirement.

When we are at index `u` in the `num` string, we have some options to consider for `num[u]`. We can choose any digit from '1' to '9', but only if that digit has not been used before (because each digit should appear at most once), and it must also satisfy the increasing or decreasing conditions as per the pattern.

As soon as we place a digit at index `u`, we recursively call the function to handle index `u+1`. If we reach the end of the pattern and have built a valid `num` string, we store the result and stop the search.

The reason why this generates the lexicographically smallest `num` is that we always try to place the smallest possible digit at each step. If at any step, no digit satisfies the condition, we backtrack, which means we undo the last step and try a different digit.

This process continues until all the possibilities are explored, or we find the answer. As soon as the answer is found, we stop the search to ensure we have the lexicographically smallest solution.

Solution Approach

The solution approach uses a backtracking algorithm, which is implemented through a recursive function named `dfs`. It explores all possible combinations of the digits from '1' to '9' to build the string `num`. Here's a run-down of the key aspects of the backtracking approach:

1. The recursion is controlled by the function `dfs(u)`, where `u` represents the current index in `num` that we are trying to fill.
2. A list `vis` of length 10 is used to keep track of the digits that have been used so far. `vis[i]` is `True` if the digit `i` is already used in the string.
3. The `t` list is used to construct the `num` string in progress. When a digit is placed at index `u`, it is appended to `t`.
4. The base case of the recursion occurs when `u == len(pattern) + 1`. This means we have filled all the positions in `num` and we have a candidate solution. We then join all the characters in `t` to form the string `ans`, and the search is halted since we only want the lexicographically smallest solution.
5. Within the recursive function, a for-loop iterates through digits `i` from 1 to 9 to try each as the next character of `num`. For each digit, there are two main conditions to check:
   - If the current index `u` is not zero and the pattern at `u-1` is 'I', the current digit `i` should only be placed if it is greater than the last placed digit `t[-1]`.
   - Similarly, if the pattern at `u-1` is 'D', the digit `i` should be less than `t[-1]`.

   If either condition is not satisfied, it skips the current digit.
6. If the digit `i` satisfies the condition, it is marked as visited (`vis[i] = True`), added to the temporary list `t`, and the function recursively calls itself with the next index (`dfs(u + 1)`).
7. After the recursive call returns, whether it found a solution or not, the chosen digit is unmarked (`vis[i] = False`) and removed from `t` (backtracking) so that future recursive calls can consider it.
8. The recursion and backtracking continue until all valid digit permutations that meet the pattern conditions are explored or until the solution is found.

The `ans` variable is used to store the lexicographically smallest number that is formed. Since the algorithm always tries the smallest digit first and goes in increasing order, it ensures that the first complete number that is formed will be the lexicographically smallest.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using the pattern `"IDID"`. Our aim is to find the lexicographically smallest `num` that follows this pattern.

1. Start with an empty `num` string and `vis` list initializing all elements to 'False' indicating that no digits have been used yet.
2. Call `dfs(0)` to fill `num[0]`. At this stage, our `num` string and the pattern look like this: `num = ""`, pattern = `"IDID"`.
3. Since `u` is 0, we don't have any previous digits in `num`. We can choose any digit from `1` to `9`. We start by choosing the smallest available digit, '1'. Before we move on to the next index, we mark '1' as used in `vis`.
4. Now, we recursively call `dfs(1)`. The pattern at `u - 1` (pattern[0]) is 'I', which means we need `num[1]` to be greater than '1'. The smallest available digit that satisfies this is '2'. `num` now looks like this: `num = "12"`.
5. Next, we call `dfs(2)`. The pattern at `u - 1` (pattern[1]) is 'D'. Hence, `num[2]` needs to be less than '2'. We choose the smallest available digit that is less than '2', which is '1', but since '1' is already used, our next available smallest option is '3', this does not satisfy the condition so we move to the next digit which is not used that satisfies the 'D', which is '1'.
6. With `num = "121"`, we recursively call `dfs(3)`. The pattern at `u - 1` (pattern[2]) is 'I', so `num[3]` should be greater than '1'. The smallest available digit is '2', but it's already used. The next smallest available is '3', so we choose '3' and `num` becomes "1213".
7. Finally, we call `dfs(4)`. Since we're at the end of the pattern, we've generated a valid `num` that adheres to the rules. The recursion base case is reached and we store `num = "1213"` as `ans`.
8. Since we attempt to place the digits in increasing order and stop once the valid `num` is completed, we guarantee that our solution is the lexicographically smallest possible `num`.

In summary, the smallest `num` that follows the pattern `"IDID"` is "1213".

Python Solution

```python
class Solution:
    def smallest_number(self, pattern: str) -> str:
        # Helper function for depth-first search to build valid numbers
        def dfs(position):
            nonlocal smallest
            # If a valid number is found, stop further search
            if smallest:
                return
            # Check if all positions are filled satisfying the pattern
            if position == len(pattern) + 1:
                smallest = ''.join(current_number)
                return
            # Try all possible digits from 1 to 9 for the next character
            for digit in range(1, 10):
                if not visited[digit]:
                    # If 'I' is encountered, the digit must be greater than the previous digit
                    if position and pattern[position - 1] == 'I' and int(current_number[-1]) >= digit:
                        continue
                    # If 'D' is encountered, the digit must be smaller than the previous digit
                    if position and pattern[position - 1] == 'D' and int(current_number[-1]) <= digit:
                        continue
                    # Mark the digit as used and add it to the current number
                    visited[digit] = True
                    current_number.append(str(digit))
                    # Recursively continue to the next position
                    dfs(position + 1)
                    # Backtrack: unmark the digit and remove it from the current number
                    visited[digit] = False
                    current_number.pop()

        # Initialize the list to keep track of visited digits
        visited = [False] * 10
        # Initialize the list to construct the current number
        current_number = []
        # Variable to keep track of the smallest number found
        smallest = None
        # Start DFS from the first digit
        dfs(0)
        # Return the smallest number that fits the given pattern
        return smallest

```

Java Solution

```java
class Solution {
    // Array to keep track of visited digits
    private boolean[] visited = new boolean[10];
    // StringBuilder to construct the sequence incrementally
    private StringBuilder sequence = new StringBuilder();
    // String to store the given pattern
    private String pattern;
    // String to store the final answer sequence
    private String answer;

    public String smallestNumber(String pattern) {
        this.pattern = pattern;
        // Starting the depth-first search (DFS)
        dfs(0);
        // Return the final answer sequence
        return answer;
    }

    // Helper method for the DFS
    private void dfs(int position) {
        // If an answer is already found, stop the recursion
        if (answer != null) {
            return;
        }
        // If the length of sequence equals the length of pattern + 1, we have a complete sequence
        if (position == pattern.length() + 1) {
            // Set the current sequence as the answer
            answer = sequence.toString();
            return; // Stop further recursion
        }
        // Iterate through all possible digits (1 to 9)
        for (int i = 1; i < 10; ++i) {
            // If the current digit i has not been used yet
            if (!visited[i]) {
                // If the last added digit should be less according to the pattern 'I'
                if (position > 0 && pattern.charAt(position - 1) == 'I' && sequence.charAt(position - 1) - '0' >= i) {
                    continue; // Skip this digit since it would break the pattern
                }
                // If the last added digit should be more according to the pattern 'D'
                if (position > 0 && pattern.charAt(position - 1) == 'D' && sequence.charAt(position - 1) - '0' <= i) {
                    continue; // Skip this digit since it would break the pattern
                }
                // Mark the digit as used
                visited[i] = true;
                // Add the digit to the sequence
                sequence.append(i);
                // Recurse to the next position with updated sequence and visited digits
                dfs(position + 1);
                // Backtrack: remove the last digit from the sequence
                sequence.deleteCharAt(sequence.length() - 1);
                // Mark the digit as not used (undo the previous marking)
                visited[i] = false;
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string smallestNumber = "";     // To store the answer
    string pattern;                 // To store the input pattern
    vector<bool> visited;           // To keep track of visited digits
    string tempNumber = "";         // Temporary number for DFS traversal

    // Entry function to find the smallest number satisfying the pattern
    string smallestNumber(string pattern) {
        this->pattern = pattern;    // Initialize the class member with the input pattern
        visited.assign(10, false);  // All digits are initially not visited
        dfs(0);                     // Start the Depth-First Search (DFS) from index 0
        return smallestNumber;      // Return the smallest number that satisfies the pattern
    }

    // Recursive function to perform DFS and find the solution
    void dfs(int index) {
        if (smallestNumber != "") return; // If already found the solution, exit the function
        if (index == pattern.size() + 1) { // If the size of tempNumber is correct
            smallestNumber = tempNumber;   // Assign tempNumber to smallestNumber
            return;
        }
        for (int i = 1; i < 10; ++i) { // Loop through digits 1 to 9
            if (!visited[i]) { // If digit i has not been used
                // Skip if current pattern requires increase and the last digit in tempNumber is not less than i
                if (index > 0 && pattern[index - 1] == 'I' && tempNumber.back() - '0' >= i) continue;
                // Skip if current pattern requires decrease and the last digit in tempNumber is not greater than i
                if (index > 0 && pattern[index - 1] == 'D' && tempNumber.back() - '0' <= i) continue;
                visited[i] = true;               // Mark digit i as visited
                tempNumber += to_string(i);      // Append digit i to the tempNumber
                dfs(index + 1);                  // Recur for the next index
                tempNumber.pop_back();           // Backtrack: remove the last digit from tempNumber
                visited[i] = false;              // Backtrack: mark digit i as not visited
            }
        }
    }
};

```

Typescript Solution

```typescript
// Function to find the smallest number that matches the given pattern
function smallestNumber(pattern: string): string {
    const patternLength = pattern.length;
    // Result array to hold the sequence of digits forming the smallest number
    const result = new Array(patternLength + 1).fill('');
    // Visited array to keep track of used digits
    const visited = new Array(patternLength + 1).fill(false);

    // Depth-first search function to build the result sequence
    // i: current position in the pattern, currentNum: current digit to consider
    const depthFirstSearch = (i: number, currentNum: number) => {
        // Base case: if we've reached the end of the pattern, return
        if (i === patternLength) {
            return;
        }

        // If currentNum has been used, backtrack and try a different number
        if (visited[currentNum]) {
            visited[currentNum] = false;
            if (pattern[i] === 'I') { // 'I' means increasing; we go backwards with a smaller number
                depthFirstSearch(i - 1, currentNum - 1);
            } else { // 'D' means decreasing; we go backwards with a larger number
                depthFirstSearch(i - 1, currentNum + 1);
            }
            return;
        }

        // Mark the current number as used
        visited[currentNum] = true;
        // Assign current number to the result array at position i
        result[i] = currentNum;

        // If the current pattern character is 'I', explore the next numbers in increasing order
        if (pattern[i] === 'I') {
            for (let j = result[i] + 1; j <= patternLength + 1; j++) {
                if (!visited[j]) { // If the number has not been used
                    depthFirstSearch(i + 1, j); // Recur with the next position and the current number
                    return;
                }
            }
            // If no valid number is found, backtrack
            visited[currentNum] = false;
            depthFirstSearch(i, currentNum - 1);
        } else { // If the current pattern character is 'D', explore numbers in decreasing order
            for (let j = result[i] - 1; j > 0; j--) {
                if (!visited[j]) { // If the number has not been used
                    depthFirstSearch(i + 1, j); // Recur with the next position and the current number
                    return;
                }
            }
            // If no valid number is found, backtrack
            visited[currentNum] = false;
            depthFirstSearch(i, currentNum + 1);
        }
    };

    // Start the DFS with the first number being 1
    depthFirstSearch(0, 1);
    // Once the DFS is complete, fill in the last available number in the result
    for (let i = 1; i <= patternLength + 1; i++) {
        if (!visited[i]) {
            result[patternLength] = i; // The last number in the result should be the unvisited number
            break;
        }
    }

    // Convert the result array to a string and return it
    return result.join('');
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the number of recursive calls to the `dfs` function, which is dependent on the length of the `pattern` string (`n`) and the branching factor at each step of the recursion.

With each recursive call to `dfs`, the function tries to append each number from `1` to `9` that hasn't already been used to the temporary array `t`. This means that in the worst case, the first recursive call will have `9` options, the second will have `8` options, and so on, resulting in a factorial time complexity.

Let's denote the length of the pattern as `n`. The number of recursive calls can be bounded by `9!` (factorial) for small patterns, since we have at most `9` digits to use, and it decreases for each level of the recursion. However, for longer patterns, the maximum branching factor will diminish as the pattern increases beyond `9`, so it will be less than `9!` for patterns longer than `9`.

Therefore, the time complexity can be approximated as `O(9!)` for patterns up to length `9`. For patterns longer than `9`, the time complexity is still bounded by `O(9!)` due to the early termination of the recursion once all digits are used.

Space Complexity

The space complexity is determined by the depth of the recursion (which impacts the call stack size) and the additional data structures used (such as the `vis` array and the `t` list).

Since the maximum depth of the recursion is equal to the length of the pattern plus one (`n + 1`), the contribution to the space complexity from the call stack is `O(n)`.

The `vis` array is always of size `10`, representing the digits `1` through `9`. The size of `t` corresponds to the depth of the recursion, which is `O(n)`. Therefore, the space requirements for `vis` are `O(1)` whereas for `t` are `O(n)`.

Combining the contributions, the total space complexity is `O(n)` due also to the recursive call stack size being at most `n` for patterns longer than `9`.

To summarize:

- The time complexity is `O(9!)`.
- The space complexity is `O(n)`.