386. Lexicographical Numbers

Medium Depth-First Search Trie

Problem Description

The problem at hand is to generate a list of integers from 1 to n and sort this list in lexicographical order. Lexicographical order is akin to how words are ordered in a dictionary, where the sequence is based on the alphabetical order of their component letters. When applied to numbers, it means ordering them as if they're strings, where '10' comes before '2' because '1' comes before '2' in the lexicographic sequence.

Constraints ensure that the algorithm is efficient, with the requirement being to run in linear time $O(n)$, which means that the time taken to solve the problem should be directly proportional to the size of the input, without any nested iterations that would increase the time complexity. Additionally, the solution must use constant extra space $O(1)$, excluding the output list, meaning the memory usage should not grow with the size of n.

Intuition

The problem prompts us to think about the properties of numbers in lexicographical order. Upon closer inspection, we can observe a pattern akin to performing a depth-first search (DFS) on a tree, where each number is treated as a node, and its subsequent numbers are its children in tree-like form.

To use this approach, we start from 1 and explore as deep (as big a number) as we can by multiplying by 10 until we're still within bounds of n. This is analogous to traversing down a branch of a tree. When we can't multiply by 10 anymore (either because we reach a number ending in 9 which can't have a digit appended or we exceed n), we backtrack by dividing by 10, which simulates going up one level in the tree, and then we move to the next node by adding 1, which is like visiting the next sibling in the tree.

We repeat this process until we've visited all valid nodes (numbers from 1 to n). Since we're visiting each number once, and every operation we're performing is constant time, the algorithm meets the time complexity requirement of O(n). Also, the only extra space used is for the output, with the variables inside the function using constant space, satisfying the space complexity constraint.

Applying this principle, the given solution efficiently constructs the lexicographical sequence without explicitly converting the numbers to strings, thereby avoiding the overhead of string manipulation and sorting, which would exceed the time complexity allowed.

Solution Approach

The implementation uses a simple integer to keep track of the current number and appends it to the result list (acting as a stack) until we've constructed all numbers from 1 to n in lexicographical order. Here's the step-by-step breakdown of the algorithm:

1. Initialize v to 1 as we need to start constructing our numbers from the smallest positive integer.
2. Create an empty list `ans` that will contain our final sorted numbers.
3. Iterate i from 0 to $n - 1$, which will allow us to fill in `ans` with the appropriate n numbers.
4. During each iteration, append the current value of v to `ans`.
5. If v multiplied by 10 is less than or equal to n, then replace v with $v * 10$. This step is like going deeper into the tree (taking a step to the next depth).
6. If we can't go deeper (either v has a last digit of 9 or $v * 10$ would exceed n), we do the following to backtrack and find the next number to explore:
   - We use a `while` loop to keep dividing v by 10 until we find a number that can have 1 added to it without exceeding n or ending with a 9 (this is the backtracking step, where we essentially go up in the tree to explore other branches).
   - Once we find such a number, we increment v by 1 to move to the next possible number (next sibling in the tree).
7. Repeat steps 4 to 6 until all n numbers have been generated and added to `ans`.
8. Return the `ans` list, which now contains all numbers from 1 to n sorted lexicographically.

The given solution can be linked to a depth-first search (DFS) algorithm in the following ways:

- Starting from 1 and exploring deeper numbers ($v * 10$) resembles visiting all child nodes in a path before backtracking.
- Once we can't go deeper, we backtrack by dividing by 10, analogous to going up to a previous node in DFS.
- Incrementing v is like visiting the next node on the same level in DFS.

By following this pattern, we ensure that every number is visited only once, following the desired sequence, and thus the time complexity is $O(n)$. There are no additional data structures used to store intermediate results; hence, the space complexity is $O(1)$ if we don't consider the space needed for the output list `ans`.

Example Walkthrough

Let's illustrate the solution approach using $n = 21$ as a small example:

1. Start by initializing v to 1 because we must construct numbers starting from the smallest positive integer.
2. Create an empty list `ans` to contain our lexicographically ordered numbers.
3. Now, we iterate from $i = 0$ to $n - 1$ (until we have n numbers in `ans`):
   - For $i = 0$:
     - $v = 1$. Since $v * 10 = 10$ is less than or equal to 21, we can go deeper in the lexicographical tree. We append v to `ans` making it $[1]$.
   - For $i = 1$:
     - Now, $v = 10$. Since $v * 10 = 100$ is greater than 21, we can't go deeper. So, we add v to `ans` to have $[1, 10]$.
     - We enter the backtracking while loop:
       - $v / 10 = 1$, which can't have 1 added to it because it ends with a 9 after incrementing (this step is not applicable for this value but will be later).
       - We increment v to 11, since it's within bounds and not ending with a 9.
   - Continuing this process, our `ans` progresses through: $[1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]$.
   - After reaching 19, we can't multiply by 10 because it would exceed 21. So, we backtrack:
     - Divide 19 by 10 to get 1, increment by 1 to get 2 which is valid.
   - We append 2 to `ans` to get $[1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2]$.
   - We continue this process, v becomes 20, which we append to `ans`, then it becomes 21 which is also appended.
   - Since $v * 10$ is greater than n, we try to increment. However, 21 ends in a 9 after backtracking and incrementing, so we would stop the process here.
4. After all iterations, our list `ans` is $[1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2, 20, 21]$.
5. We then reorder `ans` to get it into the correct lexicographical order manually: $[1, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2, 20, 21]$ becomes $[1, 2, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]$.

Note that steps 4 and 5 occur automatically as part of the algorithm's design; they are effectively illustrating what the algorithm is doing under the hood.

By following each iteration's decision to either "go deeper" or "backtrack and move next" as outlined in the solution approach, the lexicographical order is achieved with $O(n)$ time complexity and $O(1)$ space complexity, excluding the space of the output list `ans`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def lexicalOrder(self, n: int) -> List[int]:
        current_value = 1
        lex_order_list = []

        # Generate n lexicographic numbers
        for _ in range(n):
            # Append current value to the result list
            lex_order_list.append(current_value)

            # If the current value times 10 is less than or equal to n,
            # then 10 will still give us a lexicographically smaller number
            if current_value * 10 <= n:
                current_value *= 10
            else:
                # If current_value + 10 is larger than n, we check:
                # If the current value is the end of the range (ends in 9)
                # or adding one to current value would exceed n,
                # we continuously divide current_value by 10 until it's not.
                while current_value % 10 == 9 or current_value + 1 > n:
                    current_value //= 10

                # Finally, we increment current_value by one to get the next valid number
                # in lexicographical order that is also less than or equal to n.
                current_value += 1

        # Return the list containing all n numbers in lexicographical order
        return lex_order_list
```

Java

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

class Solution {

    public List<Integer> lexicalOrder(int n) {
        // Initialize an ArrayList to store the lexicographically ordered numbers
        List<Integer> result = new ArrayList<>();
        // Start with the smallest lexicographically number 1
        int current = 1;

        for (int i = 0; i < n; ++i) {
            // Add the current number to the result list
            result.add(current);

            // Check if the next lexicographical step is to multiply by 10
            if (current * 10 <= n) {
                current *= 10; // If so, go down one level of the lexical tree
            } else {
                // If reached the end of this lexical level (e.g., n is 13 and current is 9),
                // or the increment leads past n, go up one level and increment
                while (current % 10 == 9 || current + 1 > n) {
                    current /= 10;
                }
                current++; // Increment to the next number in lexical order
            }
        }

        // Return the list containing the lexicographically ordered numbers
        return result;
    }

}
```

C++

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to generate the lexicographical order of numbers from 1 to n
    vector<int> lexicalOrder(int n) {
        vector<int> result;
        int current = 1;
        for (int i = 0; i < n; ++i) {
            // Add the current number to the result
            result.push_back(current);
            // If multiplying current by 10 is less than or equal to n, keep going to the next depth
            if (current * 10 <= n) {
                current *= 10;
            } else {
                // When current reaches the end of a depth (a multiple of 10 - 1) or n
                // We go back to the closest ancestor which can have a right sibling
                // and increment it by 1 in the lexicographical sequence
                while (current % 10 == 9 || current + 1 > n) {
                    current /= 10;
                }
                ++current;
            }
        }
        return result; // Return the list in lexicographically increasing order
    }
};
```

TypeScript

```typescript
// Defines the lexicalOrder function that returns an array of numbers in lexicographical order up to n
const lexicalOrder = (n: number): number[] => {
    // Initialize the answer array which will hold the numbers in lexicographical order
    let answer: number[] = [];

    // A depth-first search function to explore each possible number within range
    const dfs = (current: number) => {
        // If the current number exceeds n, return since it's out of bounds
        if (current > n) {
            return;
        }
        // Add the current number to the answer array
        answer.push(current);

        // Iterate through the next possible digits (0 through 9)
        for (let i = 0; i < 10; ++i) {
            // Recursively call dfs with the next potential number
            dfs(current * 10 + i);
        }
    }

    // Start the depth-first search with initial digits (1 through 9)
    for (let i = 1; i < 10; ++i) {
        dfs(i);
    }

    // Return the final array of numbers in lexicographical order
    return answer;
};
```

```python
from typing import List

class Solution:
    def lexicalOrder(self, n: int) -> List[int]:
        current_value = 1
        lex_order_list = []

        # Generate n lexicographic numbers
        for _ in range(n):
            # Append current value to the result list
            lex_order_list.append(current_value)

            # If the current value times 10 is less than or equal to n,
            # then 10 will still give us a lexicographically smaller number
            if current_value * 10 <= n:
                current_value *= 10
            else:
                # If current_value + 10 is larger than n, we check:
                # If the current value is the end of the range (ends in 9)
                # or adding one to current value would exceed n,
                # we continuously divide current_value by 10 until it's not.
                while current_value % 10 == 9 or current_value + 1 > n:
                    current_value //= 10

                # Finally, we increment current_value by one to get the next valid number
                # in lexicographical order that is also less than or equal to n.
                current_value += 1

        # Return the list containing all n numbers in lexicographical order
        return lex_order_list
```

Time and Space Complexity

Time Complexity

The provided algorithm generates numbers in lexicographical order from 1 to n. For each number, it performs a series of steps to determine the next number in the order. The complexity analysis is as follows:

- The algorithm uses a for loop that runs n times (once for each number from 1 to n).
- Inside the loop, if the current number v multiplied by 10 is less than or equal to n, it immediately finds the next number by multiplying by 10, which is a constant time operation $O(1)$.
- If the current number v is not directly preceding a number by multiplication of 10, the algorithm may execute one or more while loop iterations. Each iteration of the loop divides v by 10 or increments v by 1, depending on the condition.
- In the worst case scenario, the division by 10 occurs at most $O(\log n)$ times per number because v can at most have $O(\log n)$ digits for a base 10 representation.

However, despite the potential multiple while loop iterations, every number from 1 to n is processed exactly once, and the while loop adjusts the next start of the sequence. Therefore, each digit is checked a constant number of times.

The overall time complexity is $O(n)$. While there are additional $O(\log n)$ operations per element, those operations do not multiply the total complexity, because they relate to the transition between elements rather than processing of individual elements themselves.

Space Complexity

The space complexity of the algorithm includes the space needed for the output as well as any additional data structures used in the computation:

- The output list `ans` contains n elements, resulting in a space complexity of $O(n)$.
- The variable v and the loop index i are of constant size, adding $O(1)$ space.

Hence, the total space complexity, considering the space used to store the output, is $O(n)$. If the space for the output is not considered as part of the complexity analysis, the space complexity of the algorithm itself is $O(1)$.