1215. Stepping Numbers

Medium Breadth-First Search Backtracking

Leetcode Link

Problem Description

We need to find all "stepping numbers" between two integers `low` and `high`, inclusive. A stepping number is defined as a number in which each digit is one more or one less than its neighboring digits. For example, `123` and `987` are stepping numbers, but `135` and `975` are not. The task is to produce a list of these particular numbers in sorted order without skipping any in the given range.

Intuition

The concept is similar to a breadth-first search (BFS) on the numbers, but with a unique condition that defines our graph's edges: a number can only connect to another if they differ by one at the last digit. Since single-digit numbers naturally fit the stepping number definition (except for `0` which has only one neighbor), we use them as the starting points for our BFS queue, except for `0` which we include directly into the results if it's within the given range.

From each single-digit number `1` to `9`, we construct new numbers by appending a digit either one less or one more than the last digit (if possible), ensuring this new number is still a stepping number. We continue this process in a BFS manner, checking at each step if our current number falls within the desired range `low` to `high`. If it does, we include it in the results. The stopping condition for our BFS is when a number exceeds `high`, ensuring we are not wasting resources by continuing the search beyond the given range.

By using BFS, we also take advantage of the fact that the queue keeps numbers approximately in the order of their magnitude, helping to fulfill the requirement that the list we return is sorted.

Solution Approach

The solution approach involves implementing a Breadth-First Search (BFS) algorithm. BFS is typically used to traverse or search tree or graph data structures. It starts at some arbitrary node of a graph (or root of a tree) and explores the neighbor nodes first, before moving to the next level neighbors. Here's how the BFS is used to find stepping numbers:

- First, we check if `0` is within the inclusive range given by `low` and `high`. If it is, we add `0` to the answer list because `0` is considered a stepping number.
- Then, we initialize a queue `q` and add the digits `1` to `9` to it. These are our starting points for generating stepping numbers since each single digit is trivially a stepping number.
- Now we follow the BFS pattern: We continuously process items from the queue until the queue is empty or we've exceeded our `high` limit. For each element `v` that we pop from the queue, we do the following:
  - Check if `v` is greater than `high`. If it is, we break from the loop as further numbers will only be larger and outside our range.
  - If `v` is within the range `[low, high]`, we add `v` to our list of answers, `ans`.
  - Next, we need to generate the potential stepping numbers that can be formed using `v` as the base. To do this, we consider the last digit `x` of `v`.
  - If `x > 0`, i.e., it's possible to subtract one from it without getting a negative digit, we generate a new number by appending `x - 1` to `v`, and we add `v * 10 + x - 1` to the queue.
  - If `x < 9`, i.e., we can add one to the digit without exceeding `9`, we create another number by appending `x + 1` to `v`, and we add `v * 10 + x + 1` to the queue.
- This process continues, growing numbers at the queue's front by one digit at a time, always checking that they are stepping numbers.
- Finally, when the queue is empty, or all remaining numbers in the queue are greater than `high`, the BFS search is complete. The `ans` list, which has been constructed in ascending order due to the nature of BFS, contains all the stepping numbers in the range `[low, high]`.

This algorithm employs BFS effectively to navigate the space of numbers, efficiently filtering and constructing stepping numbers. It uses the queue `q` to keep track of candidates for stepping numbers and a list `ans` to store the final results in sorted order.

Example Walkthrough

Let's take a small range to illustrate the solution approach. Assume our low is `10` and high is `21`. Here is how the algorithm would execute to find stepping numbers in this range:

1. Initialize the result list `ans` and check if `0` is within the range `10` to `21`. It is not, so we do not add `0` to `ans`.
2. Initialize the queue `q` and add the digits `1` to `9` to it, because these are already stepping numbers.
3. Start the BFS by dequeuing the front of `q` and processing it. The process starts with 1.
4. Since `1` is less than `low`, it is not added to the `ans`, but we will use it to generate potential stepping numbers. We check for the last digit `x` of `1`, which is `1`. We can subtract `1` to get `0` and add `1` to get `2`. We generate `10` and `12` and add them to the queue.
5. Continue the BFS. The next number in the queue would be `2`, and we will follow the same steps to generate `21` and `23` and add to the queue. Since `21` is within the range [10, 21], it is added to `ans`.
6. The same process continues for queue elements `3` through `9` but all the numbers each would generate (`34` and `32` for `3`, `43` and `45` for `4`, and so on) would be greater than the `high` of `21`, so they do not contribute to the `ans` list.
7. At this stage our BFS is mostly generating numbers that are higher than `21`. Once the number at the front of the queue is `22` or higher, the BFS stops processing new numbers.
8. The final `ans` list contains the stepping numbers in the given range: `[10, 12, 21]`.

The algorithm uses the BFS to systematically explore larger and larger stepping numbers starting from the smallest possible ones (the single-digit numbers) while checking against the `low` and `high` constraints to build a sorted result list.

Python Solution

```python
from collections import deque
from typing import List

class Solution:
    def count_stepping_numbers(self, low: int, high: int) -> List[int]:
        # Initialize an empty list to store stepping numbers
        stepping_numbers = []

        # Add 0 to the list if the range starts from 0 because 0 is a stepping number
        if low == 0:
            stepping_numbers.append(0)

        # Create a queue and initialize it with numbers 1 through 9
        # These serve as starting points for generating stepping numbers
        queue = deque(range(1, 10))

        # Process the queue until it's empty
        while queue:
            # Pop the first element in the queue
            current = queue.popleft()

            # If the current number exceeds the high limit, exit the loop
            if current > high:
                break

            # If the current number is within the specified range, add it to the result list
            if current >= low:
                stepping_numbers.append(current)

            # Get the last digit of the current number to calculate next possible stepping numbers
            last_digit = current % 10

            # Generate the next number by appending a digit smaller by 1 (if possible) and larger by 1 (if possible)
            if last_digit:
                queue.append(current * 10 + last_digit - 1)
            if last_digit < 9:
                queue.append(current * 10 + last_digit + 1)

        # Return the list of stepping numbers
        return stepping_numbers

# Example of how the code can be used:
# solution = Solution()
# print(solution.count_stepping_numbers(0, 21))
```

Java Solution

```java
class Solution {

    // Function to return all stepping numbers between the range [low, high]
    public List<Integer> countSteppingNumbers(int low, int high) {
        List<Integer> steppingNumbers = new ArrayList<>();

        // Add 0 as a stepping number if it's within the range
        if (low == 0) {
            steppingNumbers.add(0);
        }

        // Create a queue to perform Breadth First Search (BFS)
        Deque<Long> queue = new ArrayDeque<>();

        // Seed the queue with numbers 1 to 9 as they are the single-digit stepping numbers
        for (long i = 1; i < 10; i++) {
            queue.offer(i);
        }

        // Perform BFS to find all stepping numbers
        while (!queue.isEmpty()) {
            long currentNumber = queue.pollFirst();

            // Terminate BFS when the current number exceeds the upper bound
            if (currentNumber > high) {
                break;
            }

            // Add the current number to the result list if it's within the range
            if (currentNumber >= low) {
                steppingNumbers.add((int) currentNumber);
            }

            // Get the last digit of the current number
            int lastDigit = (int) currentNumber % 10;

            // Generate next stepping number by appending a valid digit

            // If the last digit is not 0, append (lastDigit - 1)
            if (lastDigit > 0) {
                queue.offer(currentNumber * 10 + lastDigit - 1);
            }

            // If the last digit is not 9, append (lastDigit + 1)
            if (lastDigit < 9) {
                queue.offer(currentNumber * 10 + lastDigit + 1);
            }
        }

        // Return the complete list of stepping numbers
        return steppingNumbers;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <queue>

class Solution {
public:
    // Function to find all stepping numbers between low and high
    vector<int> countSteppingNumbers(int low, int high) {
        // Container to hold the final list of stepping numbers
        vector<int> steppingNumbers;

        // If zero is within the range, add it to the list
        if (low == 0) {
            steppingNumbers.push_back(0);
        }

        // Queue to facilitate the breadth-first search
        queue<long long> bfsQueue;

        // Initialize the queue with numbers 1 through 9
        for (int digit = 1; digit < 10; ++digit) {
            bfsQueue.push(digit);
        }

        // Perform BFS to generate stepping numbers
        while (!bfsQueue.empty()) {
            // Extract the next element from the queue
            long long currentNumber = bfsQueue.front();
            bfsQueue.pop();

            // If the current number is greater than the high limit, stop the search
            if (currentNumber > high) {
                break;
            }

            // If the current number lies within the range, add it to the result list
            if (currentNumber >= low) {
                steppingNumbers.push_back(currentNumber);
            }

            // Calculate the last digit of the current number
            int lastDigit = currentNumber % 10;

            // Generate the next stepping number and add it to the queue if the last digit is not 0
            if (lastDigit > 0) {
                long long nextSteppingNumber = currentNumber * 10 + lastDigit - 1;
                bfsQueue.push(nextSteppingNumber);
            }

            // Generate the next stepping number and add it to the queue if the last digit is not 9
            if (lastDigit < 9) {
                long long nextSteppingNumber = currentNumber * 10 + lastDigit + 1;
                bfsQueue.push(nextSteppingNumber);
            }
        }

        // Return the final list of stepping numbers
        return steppingNumbers;
    }
};
```

Typescript Solution

```typescript
// Function to find all stepping numbers in a given range.
function countSteppingNumbers(low: number, high: number): number[] {
    // Initializing an array to store the stepping numbers.
    const steppingNumbers: number[] = [];

    // If low is 0, we include it as it's technically a stepping number.
    if (low === 0) {
        steppingNumbers.push(0);
    }

    // Initialize a queue to perform breadth-first search.
    const queue: number[] = [];

    // Seed the queue with numbers 1 through 9, the single-digit stepping numbers.
    for (let digit = 1; digit < 10; ++digit) {
        queue.push(digit);
    }

    // Execute breadth-first search to find all stepping numbers up to 'high'.
    while (queue.length) {
        // Fetch the first number in queue.
        const currentNum = queue.shift()!;

        // Stop processing if the current number exceeds the high bound.
        if (currentNum > high) {
            break;
        }

        // If the current number is within the range, add it to the result.
        if (currentNum >= low) {
            steppingNumbers.push(currentNum);
        }

        // Check the last digit of the current number.
        const lastDigit = currentNum % 10;

        // If the last digit is not 0, append a valid stepping number by subtracting one.
        if (lastDigit > 0) {
            queue.push(currentNum * 10 + lastDigit - 1);
        }

        // If the last digit is not 9, append a valid stepping number by adding one.
        if (lastDigit < 9) {
            queue.push(currentNum * 10 + lastDigit + 1);
        }
    }

    // Return the array of stepping numbers found in the range.
    return steppingNumbers;
}
```

Time and Space Complexity

The time complexity of the algorithm is `O(10 * 2^log(M))`, where `M` is the highest number you have in the range. Since the algorithm only ever has numbers with up to `log(M)` digits in the queue and for each such number the algorithm generates at most two more numbers, it ends up with a factor of `2^log(M)` operations. Multiplying by the initial range of numbers `1-9` gives us the `10` factor.

The space complexity is `O(2^log(M))` because the queue can grow up to include all stepping numbers less than `high`. The maximum number of elements that the queue can hold is determined by the number of stepping numbers with `log(M)` digits, which is the number of digits in `high`. The stepping number sequence grows exponentially as more digits are added to the numbers, and thus the space required by the queue grows exponentially with the number of digits in `high`, resulting in a space complexity of `O(2^log(M))`.