248. Strobogrammatic Number III

Hard · Recursion · Array · String · Leetcode Link

Problem Description

The challenge presented involves identifying a special class of numbers known as "strobogrammatic numbers." A strobogrammatic number is one that retains its original value when rotated by `180` degrees. Imagine flipping the number upside down; if it still reads the same, it's strobogrammatic. For example, `69` and `88` are strobogrammatic, while `70` is not.

Given two string inputs, `low` and `high`, which represent the lower and upper bounds of a numeric range, the task is to calculate the total count of strobogrammatic numbers that fall within this range, including the bounds themselves. We need to ensure that we convert `low` and `high` from strings to integers because we are working with numeric comparisons.

This is a problem that combines counting, string manipulation, and number theory. Solvers must understand the nature of strobogrammatic numbers and devise a strategy to generate and count all valid strobogrammatic numbers within the specified interval.

Intuition

To approach this solution, we need to generate strobogrammatic numbers in an efficient way, which requires careful consideration given the potentially large range. The direct approach of checking each number within the range will be inefficient, especially for large bounds.

Here are the critical steps to the algorithm:

1. We observe that strobogrammatic numbers are symmetrical and recursively build them from the middle outwards.
2. For a given length `u`, we can construct strobogrammatic numbers by placing pairs of strobogrammatic digits at the start and end of an already-formed strobogrammatic number of length `u - 2`. This uses a depth-first search (DFS) approach.
3. The valid pairs to form strobogrammatic numbers are `('0', '0'), ('1', '1'), ('8', '8'), ('6', '9'), and ('9', '6')`.
4. We include '0' at the ends only if we are not at the outermost layer, since a number cannot start with '0'.
5. We execute this DFS approach in a loop starting from the length of the `low` string to the length of the `high` string, building all possible strobogrammatic numbers of each length.
6. We check if each generated strobogrammatic number falls within the `low` to `high` range after converting it to an integer.
7. Increment a counter each time we find a valid strobogrammatic number within the range.

This approach focuses on generating only potentially valid strobogrammatic numbers rather than searching through the entire range, thus reducing the number of necessary checks and improving efficiency.

Solution Approach

The provided solution involves the use of recursion to generate strobogrammatic numbers with a depth-first search (DFS) approach. To do so, a helper function called `dfs` is used to construct these numbers.

Here's a detailed explanation of how the solution operates:

- The `dfs` function is defined to construct strobogrammatic numbers of a given length `u`. It has two base cases:
  - If `u == 0`, the function returns an empty list containing just an empty string `['']`, since there are no digits in a zero-length number.
  - If `u == 1`, the function returns a list of single-digit strobogrammatic numbers `['0', '1', '8']`.
- For other cases, `dfs` is called recursively on `u - 2` to return the list of strobogrammatic numbers that are two digits shorter. We can sandwich pairs of strobogrammatic digits around each returned number to form new strobogrammatic numbers of length `u`.
- These digit pairs are added only if the resulting number is not longer than the maximum length (`n`) being checked. The pairs used are `('1', '1')`, `('8', '8')`, `('6', '9')`, and `('9', '6')`. If the length is not the full target length `n`, we can also use the pair `('0', '0')`, but leading zeros are not added to full-length numbers.
- After defining the `dfs` function, the lengths `a` and `b` represent the lengths of the `low` and `high` strings, respectively, which are used to get the length range of strobogrammatic numbers.
- The main part of the solution iterates over each length from `a` to `b`, inclusive, generating strobogrammatic numbers of that length using the `dfs` function.
- The generated strings are checked to determine if they fall within the specified numeric range `[low, high]`. This is done by converting the strobogrammatic string to an integer and comparing it against the numeric `low` and `high`. If it falls within the range, the counter `ans` is incremented.
- Finally, the `ans` value containing the count of strobogrammatic numbers in the specified range is returned.

Throughout the implementation, key algorithmic patterns such as recursion, DFS, and generating combinatorial output based on constraints are used to build an efficient solution for counting strobogrammatic numbers within a given range.

Example Walkthrough

Let's consider a small example where the `low` string is "10" and the `high` string is "100". We need to find out how many strobogrammatic numbers exist between 10 and 100.

Using the solution approach, we would start by finding strobogrammatic numbers of different lengths within the inclusive range of the lengths of "10" (length 2) and "100" (length 3).

We iterate through lengths 2 and 3 since no strobogrammatic number of length 1 falls between 10 and 100.

For length 2 (same as the length of "10"), the possible strobogrammatic numbers are:

- `11`, which is not strobogrammatic because it doesn't retain its value when flipped.
- `69`, which is strobogrammatic.
- `88`, which is strobogrammatic.
- `96`, which is strobogrammatic.
- `00` is excluded as it's not a valid two-digit number because numbers cannot start with '0'.

Out of these, only `69`, `88`, and `96` are valid and fall within the given range (10 to 100).

Next, for length 3 (same as the length of "100"), the possible strobogrammatic numbers would need to have a form such as "x0x", "x1x", or "x8x" (the middle digit can be '0', '1', or '8'). But we quickly realize that none of these forms can create a valid strobogrammatic number due to the nature of the digit '0' in the middle. As a result, there are no valid 3-digit strobogrammatic numbers between 10 and 100.

As such, the total count of strobogrammatic numbers between 10 and 100 is 3.

In this case, the `dfs` function would have worked by first generating numbers of length 2 by sandwiching the central parts `['']` with all valid pairs except `('0', '0')` and then generating numbers of length 3 by sandwiching the central parts `['0', '1', '8']` with valid pairs. However, since all 3-digit combinations fall outside the range, they would not be counted.

The final answer would therefore be `3`, representing the strobogrammatic numbers `69`, `88`, and `96`.

Python Solution

```python
class Solution:
    def strobogrammaticInRange(self, low: str, high: str) -> int:
        # Helper function to generate strobogrammatic numbers of length 'length'
        def generate_strobogrammatic(length):
            # Base case for a strobogrammatic number of length 0 is an empty string
            if length == 0:
                return ['']
            # Base case for length 1 (single digit strobogrammatic numbers)
            if length == 1:
                return ['0', '1', '8']
            sub_ans = []
            # Recursive call to get the inner strobogrammatic number
            for sub_number in generate_strobogrammatic(length - 2):
                # Adding the strobogrammatic pairs to the sub_number
                for pair in ('11', '88', '69', '96'):
                    sub_ans.append(pair[0] + sub_number + pair[1])
                # Numbers like '060', '080' etc. cannot be at the beginning or end
                # So we add them only when we're not at the outermost level
                if length != num_length:
                    sub_ans.append('0' + sub_number + '0')
            return sub_ans

        min_length, max_length = len(low), len(high)
        low, high = int(low), int(high)
        count = 0  # Counter for strobogrammatic numbers within the range

        # Loop through all lengths from min_length to max_length
        for num_length in range(min_length, max_length + 1):
            # generate strobogrammatic numbers of length 'num_length'
            for num_str in generate_strobogrammatic(num_length):
                # Convert the string to an integer and check if it's within range
                if low <= int(num_str) <= high:
                    count += 1
        return count  # Return the count of strobogrammatic numbers within the range

```

Java Solution

```java
class Solution {
    // Pairs of strobogrammatic numbers that are each other's reflection.
    private static final int[][] STROBO_PAIRS = {{1, 1}, {8, 8}, {6, 9}, {9, 6}};
    private int targetLength;

    // Public method to count the strobogrammatic numbers in a given range.
    public int strobogrammaticInRange(String low, String high) {
        int minLength = low.length(), maxLength = high.length();
        long lowerBound = Long.parseLong(low), upperBound = Long.parseLong(high);
        int count = 0;

        // Loop through each length from low to high
        for (targetLength = minLength; targetLength <= maxLength; ++targetLength) {
            // Generate all strobogrammatic numbers of the current length.
            for (String num : generateStrobogrammaticNumbers(targetLength)) {
                long value = Long.parseLong(num);
                // Check if the generated number is within the range, if so, increment the count.
                if (lowerBound <= value && value <= upperBound) {
                    ++count;
                }
            }
        }
        return count;
    }

    // Helper method to generate strobogrammatic numbers of a given length.
    private List<String> generateStrobogrammaticNumbers(int length) {
        // Base case for recursion: if length is 0, return a list with an empty string.
        if (length == 0) {
            return Collections.singletonList("");
        }
        // If the length is 1, we can use '0', '1', and '8' as they look same even after rotation.
        if (length == 1) {
            return Arrays.asList("0", "1", "8");
        }
        List<String> result = new ArrayList<>();
        // Get all the strobogrammatic numbers of length minus two.
        for (String middle : generateStrobogrammaticNumbers(length - 2)) {
            // Surround the middle part with each pair of STROBO_PAIRS.
            for (int[] pair : STROBO_PAIRS) {
                result.add(pair[0] + middle + pair[1]);
            }
            // If this is not the outermost layer, we can add '0' at both ends as well.
            if (length != targetLength) {
                result.add("0" + middle + "0");
            }
        }
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <functional> // For std::function
#include <utility> // For std::pair

using std::vector;
using std::string;
using std::function;
using std::pair;
using std::stoll; // For converting string to long long

using ll = long long; // Define 'll' as an alias for 'long long'

class Solution {
public:
    // Define pairs that are strobogrammatic (they look the same when rotated 180 degrees)
    const vector<pair<char, char>> strobogrammaticPairs = {{'1', '1'}, {'8', '8'}, {'6', '9'}, {'9', '6'}};

    int strobogrammaticInRange(string low, string high) {

        // Declare the current size of strobogrammatic numbers to generate
        int currentSize;

        // Depth-First Search function to generate strobogrammatic numbers of a certain size
        function<vector<string>(int)> generateStrobogrammatic = [&](int size) {
            // Base cases
            if (size == 0) return vector<string>{""};
            if (size == 1) return vector<string>{"0", "1", "8"};

            vector<string> results;
            // Generate smaller strobogrammatic numbers and append new pairs to them
            for (auto& smallerStr : generateStrobogrammatic(size - 2)) {
                for (auto& [left, right] : strobogrammaticPairs) {
                    results.push_back(left + smallerStr + right);
                }
                // If not at the outermost layer, we can add '0' at both ends
                if (size != currentSize) {
                    results.push_back('0' + smallerStr + '0');
                }
            }
            return results;
        };

        // Get sizes of the provided range
        int sizeLow = low.size(), sizeHigh = high.size();

        // Initialize counter for valid strobogrammatic numbers within the range
        int count = 0;

        // Convert string bounds to long long for numerical comparison
        ll lowerBound = stoll(low), upperBound = stoll(high);

        // Generate strobogrammatic numbers for sizes within the inclusive range [sizeLow, sizeHigh]
        for (currentSize = sizeLow; currentSize <= sizeHigh; ++currentSize) {

            // Generate strobogrammatic numbers of current size
            for (auto& strobogrammaticNum : generateStrobogrammatic(currentSize)) {

                // Convert the strobogrammatic string to a number
                ll value = stoll(strobogrammaticNum);

                // Check if the number is within the given range
                if (lowerBound <= value && value <= upperBound) {
                    ++count;
                }
            }
        }
        // Return the total count of strobogrammatic numbers in the range
        return count;
    }
};

```

Typescript Solution

```typescript
// Use the 'bigint' type to handle large integer values in TypeScript
type ll = bigint;

// Define pairs that are strobogrammatic (they look the same when rotated 180 degrees)
const strobogrammaticPairs: Array<[string, string]> = [['1', '1'], ['8', '8'], ['6', '9'], ['9', '6']];

// Depth-First Search function to generate strobogrammatic numbers of a certain size
const generateStrobogrammatic = (size: number, maxSize: number): string[] => {
    // Base cases: return arrays of empty string or single strobogrammatic digits
    if (size === 0) return [''];
    if (size === 1) return ['0', '1', '8'];

    let results: string[] = [];

    // Generate smaller strobogrammatic numbers and append new pairs to them
    const smallerNumbers = generateStrobogrammatic(size - 2, maxSize);
    for (const smallerStr of smallerNumbers) {
        for (const [left, right] of strobogrammaticPairs) {
            results.push(`${left}${smallerStr}${right}`);
        }
        // If not at the outermost layer, we can add '0' at both ends
        if (size !== maxSize) {
            results.push(`0${smallerStr}0`);
        }
    }
    return results;
};

// Function that calculates the count of strobogrammatic numbers within a given range
const strobogrammaticInRange = (low: string, high: string): number => {
    // Initialize counter for valid strobogrammatic numbers within the range
    let count: number = 0;

    // Get sizes of the provided range
    const sizeLow: number = low.length;
    const sizeHigh: number = high.length;

    // Convert string bounds to 'bigint' for numerical comparison
    const lowerBound: ll = BigInt(low);
    const upperBound: ll = BigInt(high);

    // Generate strobogrammatic numbers for sizes within the inclusive range [sizeLow, sizeHigh]
    for (let currentSize: number = sizeLow; currentSize <= sizeHigh; ++currentSize) {

        // Generate strobogrammatic numbers of the current size
        const strobogrammaticNumbers = generateStrobogrammatic(currentSize, currentSize);
        for (const numStr of strobogrammaticNumbers) {
            // Convert the strobogrammatic string to a number
            const value: ll = BigInt(numStr);

            // Check if the number is within the given range
            if (lowerBound <= value && value <= upperBound) {
                count++;
            }
        }
    }

    // Return the total count of strobogrammatic numbers in the range
    return count;
};

```

Time and Space Complexity

The given code defines a `Solution` class with a method `strobogrammaticInRange` to find strobogrammatic numbers within a given range. A strobogrammatic number is a number that looks the same when rotated 180 degrees (e.g., 69, 88, 818).

Time Complexity

The time complexity of the solution can be analyzed as follows:

- The recursive function `dfs(u)` generates all strobogrammatic numbers of length `u`.
- For `u=0` and `u=1`, it returns a fixed set of values, so this is constant time, `O(1)`.
- For `u > 1`, it recursively calls `dfs(u - 2)` and then iterates over the result, which we'll call `k`, prepending and appending pairs of strobogrammatic digits to each string. Since there are four pairs it can append (except for the first and last digits, for which there's an additional pair), the number of operations for each recursive call relates to `4 * k + k` (when `u` is not equal to `n`, considering zero-padded numbers), where `k` is the number of results from `dfs(u - 2)`.
- The number of strobogrammatic numbers of length `u` grows exponentially since every pair of digits can lead to 5 possibilities (including the '00' pair except at the top level). Therefore, approximately, the recursion's time complexity can be described by `T(u) = 5 * T(u - 2)` for `u > 1`, which indicates exponential growth.
- The full search for generating strobogrammatic numbers ranges from length `a` (`len(low)`) to length `b` (`len(high)`), and the generation complexity would roughly be `O(1) + O(5^((a-1)/2)) + ... + O(5^((b-1)/2))`, which is dominated by the largest term `O(5^((b-1)/2))` when `b` is even or `O(5^(b/2))` when `b` is odd.

Considering the final for-loop that iterates over `n` from `a` to `b` inclusive and checks against the range, the overall time complexity would be approximated by `O(b * 5^(b/2))`, where `b` is the length of `high`.

Space Complexity

The space complexity can be analyzed as follows:

- The recursion `dfs(u)` will have a maximum stack depth equivalent to `b/2` (since each recursive step reduces `u` by 2).
- Additionally, the space to store `ans` increases exponentially with the recursion, similar to time complexity, since every level of recursion generates a list of numbers that grows exponentially.
- The space is freed once each recursive call completes, but the maximum held at any time relates to the maximum depth of the recursion tree, meaning the space complexity is also dominated by the output size of the deepest recursion call.

Given the above, the space complexity is also `O(5^(b/2))`, where `b` is the length of `high`.