1088. Confusing Number II

Hard Math Backtracking Leetcode Link

Problem Description

A confusing number is defined as a number that changes into a different VALID number when it is rotated by 180 degrees. Certain digits, when rotated, transform into other digits (0 → 0, 1 → 1, 6 → 9, 8 → 8, 9 → 6), while others (2, 3, 4, 5, 7) become invalid. A number is confusing if, after this transformation, it is a valid number and different from the original. The objective is to count all the confusing numbers within a given range [1, n].

Intuition

The solution approach is based on a depth-first search (DFS) algorithm. By building the numbers digit by digit, we can explore all numbers that could potentially be confusing numbers, skipping any numbers that contain invalid digits. As we generate a number, we simultaneously calculate its rotated version.

The main idea is to iterate through each digit position of the possible confusing number, selecting from the digits that have a valid rotation (0, 1, 6, 8, and 9) and appending them to the number being formed. For every digit added, the rotated counterpart is also computed. We keep track of whether the current path of numbers is following the leading number in the given range, `n`, as this determines the upper bound of the number that can be used in the current digit's place.

After forming the entire number by adding digits up to the length of `n`, a check is performed to see if it is a confusing number— that is, its rotated form must be a valid number and should not be equal to the original number. Only then is it counted as a confusing number.

To avoid making the check each time we add a digit, we add the check only when the full length of the number has been reached.

The code uses these helper functions to perform a systematic search for all possible confusing numbers in the given range, making sure the algorithm is both efficient and exhaustive.

Solution Approach

The solution uses recursive depth-first search (DFS) to build potential confusing numbers one digit at a time. Here's a breakdown of how the solution approach is implemented:

1. **Mapping Valid Rotations**: The variable `d` is an array that maps each digit to its corresponding rotation. If a digit results in an invalid number when rotated, it maps to `-1`. This mapping array is used to quickly check if a digit is valid and what it becomes upon rotation.
2. **Recursive DFS**: The function `dfs` is the core recursive function that performs the depth-first search. It takes three arguments:
   - `pos`: the current digit position we are trying to fill,
   - `limit`: a boolean flag that tells us whether we are restricted by the digit at position `pos` of the original number `n`, and
   - `x`: the number formed so far in the DFS exploration.
3. **Base Case for DFS**: When `pos` is equal to the length of the string representation of `n`, we've reached the end of a potential number. At this point, we check if `x` is a confusing number using the `check` function, which returns `1` if `x` is confusing, and `0` otherwise.
4. **Building Numbers Digit by Digit**: Within the `dfs` function, the loop variable `i` iterates over possible digits (from 0 to 9). However, we only process those that do not map to `-1` in `d`, which ensures we skip invalid digits.
5. **Limiting the Search Space**: If `limit` is `True`, meaning we are still following the digit pattern of `n`, we only consider digits up to the digit at position `pos` in `n`. Otherwise, we explore all valid digits (up to 9).
6. **Recursive Call**: For each valid digit `i`, a recursive call is made to `dfs` for the next position (`pos + 1`), with `limit` updated to indicate whether we are still bound by `n`'s digits (this is `True` only if `limit` was `True` and `i` is the same as the digit at `pos` in `n`). The number `x` is also updated by appending the digit `i`.
7. **Counting Confusing Numbers**: The `ans` variable accumulates the number of valid confusing numbers found by subsequent `dfs` calls.
8. **Function `check`**: This function is used to determine if the number `x` is a confusing number. It does so by rotating each digit of `x` and forming the transformed number `y`. If `x` is equal to `y`, then the number isn't confusing, and the function returns `False`. If `x` is not equal to `y`, the function returns `True`, and it's a confusing number.
9. **Entry Point**: The entry point of the solution is the call to `dfs(0, True, 0)`. The initial call is made with `pos` set to 0 (starting at the first digit), `limit` set to `True` (bound by the first digit of `n`), and `x` set to 0 (starting with an empty number).
10. **Result**: After the `dfs` calls have been made, the total number of confusing numbers found within the range [1, n] is returned.

This approach is efficient because it systematically generates all possible confusing numbers up to `n`, checking each one specifically upon reaching the final digit position and avoiding any unnecessary checks or generation of numbers with invalid digits.

Example Walkthrough

Let's consider a range [1, 25] to illustrate the solution approach.

1. **Creating the Valid Rotations Map**: We have an array `d` that maps digits to their rotations, where `d = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]`. Digits mapping to `-1` are invalid for our purposes.
2. **Recursive DFS Exploration**:
   - We start with `pos = 0`, `limit = True`, and `x = 0`.
   - The DFS recurses to build numbers and explore all possible confusing numbers within the range.
3. **Building Numbers**:
   - At `pos = 0`, we try digits `0, 1, 6, 8, 9` (since others are invalid).
   - Assume we first pick digit `1`. We update `x` to be `1`.
4. **Depth-First Search Moves**:
   - We progress to `pos = 1`. Since the second digit in `25` is `5`, which limits our choice to digits less than or equal to `5`, we only try `0, 1`.
   - With `1` already chosen, we now pick `0`, so `x` becomes `10`.
5. **Limit Checking**:
   - Now, `limit` is no longer `True` as the second digit `0` is not the same as `5` in `25`.
6. **Checking for Confusing Number**:
   - When the number has been fully constructed (we reach `pos` equal to the length of `n`), we check if it's a confusing number.
   - We do this by rotating each digit and comparing the rotated number to the original.
   - Since `10` becomes `01` after rotation, which is not a valid number because of the leading zero, it is not counted.
7. **Continuing the Search**:
   - We continue the search and try other combinations like `11`, `16` (becomes `91`, a confusing number), `18`, and `19` (becomes `61`, a confusing number) at the current level.
   - Each time we find a confusing number, we increment our count.
8. **Result**:
   - After exploring all possibilities, the DFS backtracks and tries new paths until all possible numbers are checked.
   - In our range [1, 25], we would find that `16` and `19` are the confusing numbers. Thus, the solution would return `2`.

The complete DFS search tree for this range is not fully shown here due to brevity, but it would consider all valid combinations, incrementing the count each time a confusing number is detected. The recursion ensures we explore all possible paths using valid digits and adhering to the constraints imposed by `n`.

Python Solution

```python
class Solution:
    def confusingNumberII(self, N: int) -> int:
        # Mapping of valid confusing number digits to their 180-degree rotation representation.
        # Note that 2, 3, 4, 5, and 7 are not valid since they don't form a digit when rotated.
        # -1 indicates invalid digits.
        digit_mapping = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]

        # Helper function to check if a number is a confusing number by comparing
        # the original number with its rotated version.
        def is_confusing(x: int) -> bool:
            rotated, original = 0, x
            while original:
                original, remainder = divmod(original, 10)
                rotated_digit = digit_mapping[remainder]
                # If the digit is not valid when rotated, return False
                if rotated_digit == -1:
                    return False
                rotated = rotated * 10 + rotated_digit
            # Confusing number if the original number is different from the rotated number
            return x != rotated

        # The main recursive function performs a depth-first search to generate all
        # possible numbers within the limit and counts the confusing ones.
        def dfs(position: int, is_limited: bool, current_number: int) -> int:
            # If we have constructed a number with the same number of digits as N
            if position == len(N_str):
                # Check if this number is a confusing number
                return int(is_confusing(current_number))

            count = 0
            # If we are limited by the most significant digit of N, up is that digit,
            # otherwise it's 9 because we can use any digit from 0 to 9.
            upper_bound = int(N_str[position]) if is_limited else 9

            # Try all digits from 0 to upper_bound
            for i in range(upper_bound + 1):
                # Only proceed if the digit i is valid when rotated
                if digit_mapping[i] != -1:
                    # Continue to the next position, update is_limited based on current upper bound
                    # and update current_number.
                    count += dfs(position + 1, is_limited and i == upper_bound, current_number * 10 + i)

            return count

        # Convert the input number N to a string to easily access each digit
        N_str = str(N)
        # Start the DFS from the first position, with the limit flag set, and starting number as 0
        return dfs(0, True, 0)

```

Java Solution

```java
class Solution {
    private final int[] digitMapping = {0, 1, -1, -1, -1, -1, 9, -1, 8, 6}; // Mapping of digits to their confusing number counterp
    private String targetNumberString; // The target number converted to string to facilitate index-based access

    // This method calculates the number of confusing numbers less than or equal to n
    public int confusingNumberII(int n) {
        targetNumberString = String.valueOf(n); // Convert to string to get each digit by index
        return depthFirstSearch(0, 1, 0); // Start the recursive depth-first search from position 0 with limit 1
    }

    // Helper method for the recursive depth-first search
    private int depthFirstSearch(int position, int limitFlag, int currentNumber) {
        // Base case: if the current position is at the end of the string
        if (position >= targetNumberString.length()) {
            // Check if the current number is a confusing number and return 1 if true, 0 otherwise
            return isConfusing(currentNumber) ? 1 : 0;
        }

        // Determine the upper bound for this digit - if we are at the limit, we take the digit from the target
        int upperBound = limitFlag == 1 ? targetNumberString.charAt(position) - '0' : 9;
        int count = 0; // Initialize the count of confusing numbers

        // Iterate through all digits up to upperBound
        for (int digit = 0; digit <= upperBound; ++digit) {
            // Check if the digit maps to a valid confusing number digit
            if (digitMapping[digit] != -1) {
                // Recursive call to explore further digits, updating the limitFlag and currentNumber accordingly
                count += depthFirstSearch(position + 1, limitFlag == 1 && digit == upperBound ? 1 : 0, currentNumber * 10 + digit);
            }
        }

        // Return the total count of confusing numbers found
        return count;
    }

    // Helper method to check if a number is a confusing number
    private boolean isConfusing(int number) {
        int transform = 0; // This will hold the transformed confusing number
        for (int temp = number; temp > 0; temp /= 10) {
            int digit = temp % 10; // Get the last digit of temp
            transform = transform * 10 + digitMapping[digit]; // Map the digit and add it to the transformed number
        }
        return number != transform; // Return true if the original and transformed numbers are different
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function is used to convert a given integer into its confusing number equivalent.
    // A confusing number is one that when rotated 180 degrees becomes a different number.
    // It returns true if the number is confusing, false otherwise.
    bool isConfusingNumber(int x) {
        string rotatedDigits = "01689"; // Only these digits are valid after rotation.
        int rotated = 0; // The rotated number.
        int original = x; // The original number for comparison.
        int mapping[10] = {0, 1, -1, -1, -1, -1, 9, -1, 8, 6}; // Mapping from original to rotated digits.

        while (x > 0) {
            int digit = x % 10; // Get the last digit.
            if (mapping[digit] == -1) // If it's not a valid digit for rotation, return false.
                return false;
            rotated = rotated * 10 + mapping[digit]; // Append the rotated digit.
            x /= 10; // Remove the last digit from x.
        }
        return original != rotated; // The number is confusing if the original and the rotated are different.
    }

    // This recursive function explores all the combinations of valid digits to form confusing numbers.
    // It accepts the current position in the digits string, a limit flag to indicate if this digit should
    // not exceed the corresponding digit in N, and the current formed number.
    int dfs(int pos, bool limit, int x, const string& s) {
        if (pos == s.size()) { // If we've considered all digits
            return isConfusingNumber(x) ? 1 : 0;
        }
        int maxDigit = limit ? s[pos] - '0' : 9; // Determine the maximum digit we can use.
        int count = 0; // Count of confusing numbers.
        for (int digit = 0; digit <= maxDigit; ++digit) { // Explore all possible digits.
            if (rotatedDigits.find(digit) != string::npos) { // Skip invalid digits.
                // If the current digit equals the maxDigit, we set limit to true.
                // Otherwise, we can freely choose any valid digit, so limit is false.
                count += dfs(pos + 1, limit && digit == maxDigit, x * 10 + digit, s);
            }
        }
        return count;
    }

    // This function returns the total count of confusing numbers less than or equal to N.
    int confusingNumberII(int N) {
        // Convert N to string for easier manipulation.
        string numberString = to_string(N);

        // Start the DFS from position 0, with the limit flag set (since at the
        // first digit, we can't exceed the first digit of N), and initial number as 0.
        return dfs(0, true, 0, numberString);
    }
};

```

Typescript Solution

```typescript
function confusingNumberII(n: number): number {
    // Converts the number 'n' to its string representation.
    const numberAsString = n.toString();

    // Mapping of digits to their respective confusing number transformations.
    // A confusing number transformation maps 0→0, 1→1, 6→9, 8→8, 9→6.
    // Digit that cannot be in a confusing number are marked with -1.
    const digitMap: number[] = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6];

    // Helper function to check whether a number is confusing,
    // meaning its digits rotate to be a different number.
    const isConfusing = (x: number) => {
        let rotatedNumber = 0;
        // Rotate each digit and form the rotated number.
        for (let temp = x; temp > 0; temp = Math.floor(temp / 10)) {
            const digit = temp % 10;
            rotatedNumber = rotatedNumber * 10 + digitMap[digit];
        }
        // A number is confusing if it is different from its rotated form.
        return x !== rotatedNumber;
    };

    // Recursive function to perform depth-first search on the digits.
    const dfs = (position: number, isLimited: boolean, currentNumber: number): number => {
        // Base case: all digits have been processed.
        if (position >= numberAsString.length) {
            // Check if the formed number is confusing.
            return isConfusing(currentNumber) ? 1 : 0;
        }
        // Set the upper limit for the current digit based on limit.
        const upperLimit = isLimited ? parseInt(numberAsString[position]) : 9;
        let count = 0; // Store the count of confusing numbers found.
        // Try all possible digits from 0 to the upper limit for current position.
        for (let i = 0; i <= upperLimit; ++i) {
            // Only continue if the digit is a valid confusing digit (not -1).
            if (digitMap[i] !== -1) {
                // Perform DFS on the next position with updated limits and number.
                count += dfs(position + 1, isLimited && i === upperLimit, currentNumber * 10 + i);
            }
        }
        // Return the total count of confusing numbers for this path.
        return count;
    };

    // Call the depth-first search starting from the first digit.
    return dfs(0, true, 0);
}

```

Time and Space Complexity

The given code defines a function `confusingNumberII` that computes the number of confusing numbers less than or equal to `n`. A confusing number is defined as a number that does not equal its rotated 180 degrees representation.

The main algorithm involves a recursive depth-first search (DFS) function `dfs`. This function has three main parameters:

- `pos`: The current digit position being considered
- `limit`: A boolean that indicates if the current path is bounded by the maximum number of `n`
- `x`: The current number being constructed

The time complexity of the DFS depends on the length of the number `n`, as recursion is going digit by digit. At each level of recursion, the algorithm iterates through possible digits (up to 10, or `up + 1` which is the digit limit at the current `pos` if `limit` is True).

The recursive tree's branching factor on average is less than 10 due to the filtering out of invalid digits (where `d[i] == -1`). The depth of the tree is `len(s)` where `s` is the string representation of `n`. Therefore, in terms of `n`, the time complexity is roughly `O(10^len(s))`, which can be seen as `O(n)` because the number of confusing numbers is less than or equal to `n`.

The space complexity is primarily determined by the depth of the recursion call stack, which goes as deep as `len(s)`. Hence, the space complexity is `O(len(s))`.

Additionally, the auxiliary space used is constant for the array `d` and the c integers used in the algorithm (`x`, `y`, `t`, `v`, and temporary variables used for iteration and recursion).

To give a formal computational complexity:

- Time Complexity: `O(n)` where `n` is the input number
- Space Complexity: `O(len(str(n)))` where `len(str(n))` represents the number of digits in `n`