2376. Count Special Integers

Hard Math Dynamic Programming

Leetcode Link

Problem Description

The essence of this problem is to find how many positive integers up to a given number `n` have all distinct digits – that is, no digit is repeated within the number. For example, the number `12345` is a special number because all of its digits are unique, while `11345` is not because the digit `1` is repeated.

To compute this, we need to consider all integers from `1` to `n` and count only those that meet the criteria of having distinct digits. It's important to note that `0` cannot be a starting digit of any special number as the numbers are positive integers.

Intuition

To solve this problem efficiently without checking every number individually, we need to use combinatorics to count the number of potential combinations of distinct digits at different magnitudes (thousands, hundreds, tens, units, etc.).

Here's the general intuition behind the solution approach:

- We define a helper function `A(m, n)` which calculates the number of permutations of `n` distinct elements taken `m` at a time. This is important for understanding how many distinct digit numbers we can create with a given set of digits.
- We start by counting the number of special numbers for lengths less than the length of `n`. If `n` has `x` digits, we first count all special numbers with `x-1` digits or fewer. For each of these lengths, the first digit can be chosen in 9 ways (1 through 9), and the remaining digits can be chosen using the `A(m, n)` function since the order matters and we cannot repeat digits.
- For the numbers that have the same number of digits as `n`, we have to be careful not to exceed `n`. We do this by going digit by digit from the most significant to the least significant digit in `n`.
- We maintain an array `vis` which represents which digits have been visited so far to ensure that we are only considering numbers with unique digits.
- Working digit by digit from the highest to the lowest, for each digit in `n`, we count the number of special numbers we can form without exceeding that digit.
- at any point, we reach a digit in `n` that has already been used in the number we're forming, we stop the process because any larger numbers would not be special.
- Finally, when we finish processing each digit, we have counted all the special numbers up to `n`.

Overall, the approach is to smartly count all possible combinations of distinct digits without having to enumerate each special number individually.

Solution Approach

The provided solution implements a combinatorics-based approach backed by permutation mathematics and logical partitioning of the problem space. This approach utilizes an array to keep track of visited digits and a permutation function to count valid numbers. The implementation details are key to understanding how the solution works effectively to count special numbers.

Here's the step-by-step solution approach:

- We define a permutation function `A(m, n)` which calculates permutations of `n` elements taken `m` at a time recursively. This is used to figure out how many arrangements are possible for a set of digits.
- The variable `vis` is an array of boolean flags indicating which digits (0-9) have already been used in a number under construction. This way, we ensure the uniqueness of digits.
- We create a variable `ans` to store the total count of special numbers and a list `digits` which contains the individual digits of `n` in reverse order for ease of processing from the lowest to the highest digit.
- We count special numbers for lengths less than the length of `n`. For a number of length `i`, there are `9` options for the first digit (1-9) and then we calculate the arrangements of the remaining `i-1` digits using the permutation function `A(9, i - 1)`.
- Next, we process digits with the same length as `n`. Starting from the most significant digit and moving towards the least significant, we count the number of special numbers smaller than `n` that can be formed with each digit.
  - For each digit `n` (considered in reverse order), we iterate from the lowest allowed digit (1 if it's the most significant digit, and 0 otherwise) up to but not including the current digit `n` of `x`, using our `x` function to count permutations constrained by the number of distinct digits available and the length of the substring formed so far.
  - If we encounter a digit that has been already visited (`vis[v]` is true), it means we've already accounted for all special numbers that can be formed with the current prefix, and we break the loop.
  - We mark the digit we're currently at as visited (`vis[v] = True`).
  - If we reach the least significant digit, we perform one final check. If all digits were unique, we also count the number itself by incrementing `ans` by 1.

By combining these steps, the algorithm manages to count all distinct digits numbers up to `n` without enumerating each potentially vast set of numbers. The use of the permutation function significantly reduces the computational complexity by taking advantage of the properties of numbers and the definition of "special" in the problem.

Example Walkthrough

Let's use the number `n = 320` as an example to illustrate the solution approach described.

1. **Initial setup**:
   - The number `320` has three digits.
   - We initialize `ans` to `0` for counting special numbers.
   - We define a permutation function `A(m, n)` that calculates permutations of `m` elements taken `n` at a time.
   - We prepare a `vis` array to mark digits that have been used.
2. **Counting numbers with fewer digits**:
   - First, we count special numbers with one digit. There are 9 options (1-9), so `9 = A(9, 0) = 9`.
   - Next, we count special numbers with two digits. The first digit has 9 options, and the second has 9 options (0-9, excluding the first digit), so `9 = A(9, 1) = 81`.
   - Add these to `ans` for a subtotal of `9 + 81 = 90`.
3. **Processing the same length as `n`**:
   - Starting with the most significant digit, we have:
     1. **Hundreds place**: The first digit can be 1 or 2 because 3 is the hundreds digit of `n`. Each option allows for `A(9, 2)` permutations of the remaining digits.
        - When the first digit is 1, we can use any digit (0-9 excluding 1) for the tens and the units place: `1 * A(9, 2) = 72`.
        - When the first digit is 2, it's the same situation: `1 * A(9, 2) = 72`.
        - Add these to `ans` for a new subtotal: `90 + 72 + 72 = 234`.
     2. **Tens place**: Next, we consider the digit 2 as fixed (from 3) and look at the tens digit. No tens digits have been used yet (except 2), so we have options (0, 1). Each option allows for `A(8, 1)` permutations of the units digit.
        - For 0 as the tens digit: `1 * A(8, 1) = 8`.
        - For 1 as the tens digit: `1 * A(8, 1) = 8`.
        - Update `ans: 234 + 8 + 8 = 250`.
     3. **Units place**: Lastly, we fix the tens digit to 2 (from 3) and consider the units digit. We can only use 0 or 1 without repeating any digits.
        - Both 0 and 1 are valid and lead to unique numbers, so we add 2 more to our answer: `250 + 2 = 252`.
   - Since we have considered all the digits of `n = 320`, we conclude with `ans = 252`.

In our example, there are 252 positive integers less than or equal to 320 that have all distinct digits. This walkthrough demonstrates the effectiveness of using combinatorics and permutation mathematics to solve the problem without checking each number individually.

Python Solution

```python
class Solution:
    def count_special_numbers(self, n: int) -> int:
        # Recursive function to calculate permutations A(m, n) = m! / (m-n)!
        def permute(m:int, n):
            if n == 0:
                return 1
            else:
                return permute(m-1, n-1) * (m - n + 1)

        # List to keep track of visited digits
        visited = [False] * 10

        # Variable to store the count of special numbers
        answer = 0

        # Convert the number into a list of its digits in reverse order
        # so we can process the least significant digit first
        digits = [int(d) for d in str(n)][::-1]

        # Length of the number (number of digits)
        num_length = len(digits)

        # Count all special numbers with length less than the length of n
        for i in range(1, num_length):
            answer += 9 * permute(9, i - 1)

        # Count special numbers with the same length as n
        for i in range(num_length - 1, -1, -1):
            current_digit = digits[i]

            # First digit cannot be 0, others can
            start = 1 if i == num_length - 1 else 0

            while start < current_digit:
                if not visited[start]:
                    answer += permute(10 - (num_length - i), i)
                start += 1

            # If the current digit has already been visited, stop the loop
            if visited[current_digit]:
                break

            # Mark the digit as visited
            visited[current_digit] = True

            # If it's at the last digit and haven't broken the loop,
            # then we need to account for this number itself being a special number
            if i == 0:
                answer += 1

        return answer

# Example usage:
sol = Solution()
# print(sol.count_special_numbers(20))  # The output will be the count of special numbers less than or equal to 20
```

Java Solution

```java
class Solution {
    public int countSpecialNumbers(int n) {
        // Create a list to hold the individual digits of the number in reverse order
        List<Integer> digits = new ArrayList<>();
        while (n > 0) {
            digits.add(n % 10);
            n /= 10;
        }

        // Determine the number of digits in the number
        int numDigits = digits.size();

        // This will hold our final count of special numbers
        int count = 0;

        // Count special numbers with fewer digits than the input number
        for (int i = 1; i < numDigits; ++i) {
            count += 9 * permute(9, i - 1);
        }

        // Visit tracking array for digits
        boolean[] visited = new boolean[10];

        // Iterate over each digit, starting with the most significant digit
        for (int i = numDigits - 1; i >= 0; --i) {
            // Count special numbers with this digit
            // Count the special numbers smaller than the current number at digit `i`
            for (int j = i == numDigits - 1 ? 1 : 0; j < currentValue; ++j) {
                if (!visited[j]) {
                    count += permute(10 - (numDigits - i), i);
                }
            }
            // If digit was seen before, no need to continue
            if (visited[currentValue]) {
                break;
            }
            // Mark the digit as visited
            visited[currentValue] = true;
            // If we've reached the least significant digit, include it in the count
            if (i == 0) {
                ++count;
            }
        }
        return count;
    }

    // Recursion method to compute permutations, denoted as A(m, n) = m! / (m - n)!
    private int permute(int m, int n) {
        return n == 0 ? 1 : permute(m, n - 1) * (m - n + 1);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int countSpecialNumbers(int n) {
        int answer = 0; // Holds the count of special numbers.
        vector<int> digits; // Stores individual digits of the number n.

        // Extract digits of number and store in reverse order.
        while (n) {
            digits.push_back(n % 10);
            n /= 10;
        }

        int m = digits.size(); // Number of digits in n.
        vector<bool> visited(10, false); // Flags to keep track of digits used.

        // Count special numbers less than the given number with fewer digits.
        for (int i = 1; i < m; ++i) {
            answer += 9 * permute(9, i - 1);
        }

        // Count the special numbers less than the given number with same number of digits.
        for (int i = m - 1; i >= 0; --i) {
            int currentDigit = digits[i];

            // Counting arrangements for each digit less than currentDigit.
            for (int j = i == m - 1 ? 1 : 0; j < currentDigit; ++j) {
                if (!visited[j]) {
                    answer += permute(10 - (m - i), i);
                }
            }

            // If currentDigit has already been seen, stop and exit.
            if (visited[currentDigit]) {
                break;
            }

            // Mark this digit as used.
            visited[currentDigit] = true;

            // If this is the last digit and hasn't been visited, include this number.
            if (i == 0) {
                ++answer;
            }
        }

        // Return the total count of special numbers.
        return answer;
    }

    // Helper function to calculate permutations, P(m, n) = m! / (m-n)!.
    int permute(int m, int n) {
        if (n == 0) return 1; // Base case for permutation.
        return permute(m - 1, n - 1) * (m - n + 1); // Recursive count of permutations.
    }
};
```

Typescript Solution

```typescript
// Global array to keep track of visited digits.
const visited: boolean[] = new Array(10).fill(false);

// Helper function to calculate permutations, P(m, n) = m! / (m-n)!.
function permute(m:number, n: number): number {
    if (n == 0) return 1; // Base case for permutation.
    return permute(m - 1, n - 1) * (m - n + 1); // Recursive count of permutations.
}

// Main function to count special numbers up to a given number n.
function countSpecialNumbers(n: number): number {
    let answer = 0; // Holds the count of special numbers.
    const digits: number[] = []; // Stores individual digits of the number in reverse order.

    // Extract digits of n and store in reverse order.
    while (n) {
        digits.push(n % 10);
        n = Math.floor(n / 10);
    }

    const m: number = digits.length; // The number of digits in n.

    // Count special numbers less than n with fewer digits.
    for (let i = 1; i < m; ++i) {
        answer += 9 * permute(9, i - 1);
    }

    // Count the special numbers less than n with the same number of digits.
    for (let i = m - 1; i >= 0; --i) {
        const currentDigit: number = digits[i];

        // Counting arrangements for each digit less than currentDigit.
        for (let j = i == m - 1 ? 1 : 0; j < currentDigit; ++j) {
            if (!visited[j]) {
                answer += permute(10 - (m - i), i);
            }
        }

        // If currentDigit has been seen before, break from the loop.
        if (visited[currentDigit]) {
            break;
        }

        // Mark the current digit as visited.
        visited[currentDigit] = true;

        // If last digit hasn't been visited, include this number in the count.
        if (i == 0) {
            ++answer;
        }
    }

    // Reset the visited array for future calls to countSpecialNumbers.
    visited.fill(false);

    // Return the total count of special numbers.
    return answer;
}
```

Time and Space Complexity

The time complexity of the given code is primarily determined by two nested loops: an outer loop and an inner loop.

The outer loop runs for `i` from 1 to `n`, where `n` is the length of `n` in terms of number of digits. This loop runs in O(m).

Inside the outer loop, there is an inner loop that continues until `j` reaches the digit `x`. In the worst-case scenario, this loop could iterate 9 times (if `x` is 9 and `j` starts from 0). However, this doesn't run for every digit because once a repeated digit is detected, the loop breaks. Thus, this contributes less than O(9*m) complexity.

The function `A(m, n)` is a recursive implementation of arranging `n` elements in `n` places, which takes O(n) time since it calls itself recursively `n` times.

The combination of these loops and the recursive function `A(m, n)` means that the overall time complexity is O(m * 9 * n), simplifying to O(m^2) considering that `n` is bounded by `m`.

For space complexity, the code uses a `vis` list of constant size 10 to keep track of visited digits. Aside from this and some integer variables, there's no additional scaling with input size, making the space complexity O(1), or constant space.