357. Count Numbers with Unique Digits

Medium Math Dynamic Programming Backtracking

Problem Description

In this problem, we are given a non-negative integer `n`, and we are asked to find out how many integers there are with unique digits such that the integer `x` satisfies `0 <= x < 10^n`. Unique digits mean that no digit in the number repeats. For example, the number `123` has unique digits, while the number `112` does not because the digit `1` is repeated.

Intuition

To solve this problem, we can approach it by counting the number of valid numbers rather than generating each possible number, which would be inefficient.

- For `n == 0`, the only number we can have is `0` itself, hence only one unique number.
- For `n == 1`, any digit from `0` to `9` is valid, which means there are `10` unique numbers (including `0`).

As soon as `n` is greater than 1, we start with `10` possibilities (from `0` to `9`) and choose the second digit. There are only `9` possible choices left for the second digit since it has to be different from the first (excluding the case where the first digit is `0`, as we've counted that in `n == 1`). For the third digit, there's one less choice than for the second (since now two digits are taken), and so on.

The solution follows these steps for `n > 1`:

- Start the answer with `10` cases (all single-digit numbers plus the number `0`).
- For each additional digit place, we multiply our current count of unique digits by the decreasing number of options available (starting from `9` for the second digit, `8` for the third, etc.).

The formula for the number of unique digit numbers that can be formed with `i+1` digits is `f(i+1) = f(i) * (10 - i)` where `f(i)` is the number of unique digit numbers with `i` digits and `i` begins at `1` and increments until `n-1`.

The solution code uses a loop to count the number of unique digit numbers for each number of digits from `1` up to `n` and adds them up to accumulate the total count.

Solution Approach

The implementation of the solution for counting unique digit numbers consists of the following steps:

- Start by checking for the base cases. If `n` is `0`, return `1` because only the number `0` fits the criteria. If `n` is `1`, return `10` because the numbers `0` through `9` are the only valid possibilities and they all have unique digits.
- For numbers with more than one digit (`n > 1`), we'll need to calculate the possibilities using a loop. Initialize the `ans` (answer) variable with `10`, to cover the one-digit numbers. Also, initialize a variable `cur` to `9`, representing the number of choices for the first digit, excluding `0`.
- Loop from `0` to `n - 1`. In each iteration, we will calculate the number of unique numbers that can be created with an additional digit. Multiply `cur` by `9 - i`, where `i` is the current iteration's index. This represents the decrease in available choices as we fix more digits in the number.
- Add the result of the multiplication to `ans`, updating it to include the count of unique numbers with the new number of digits.
- Continue this process until the loop ends.
- Finally, return `ans`, which now holds the total count of unique-digit numbers for all lengths up to `n` digits.

Python Solution Code

```python
class Solution:
    def countNumbersWithUniqueDigits(self, n: int) -> int:
        if n == 0:
            return 1
        if n == 1:
            return 10
        ans, cur = 10, 9
        for i in range(n - 1):
            cur *= 9 - i
            ans += cur
        return ans
```

This solution employs a mathematical pattern without using any complex data structures. The loop efficiently calculates the count for each number of digits, and the use of multiplication (`cur *= 9 - i`) within the loop follows the pattern of the decreasing number of choices for each subsequent digit.

Example Walkthrough

Let's illustrate the solution approach with `n = 3`. The task is to count numbers with unique digits where `0 <= x < 1000` (since `10^3 = 1000`).

For `n = 0`, there's only one number, `0`, so the answer is `1`.

For `n = 1`, any single digit number, `0` to `9`, is valid and unique. That's `10` possibilities.

Now, for `n > 2`, we need to calculate the possibilities for numbers having `2` and `3` digits.

- **Two-digit numbers (`10` to `99`):**
  - Start with `10` total unique numbers from the `n = 1` case.
  - For the first digit (tens place), we have `9` choices (`1` to `9`, as we're not including `0` here since that's accounted for in the `n = 1` case).
  - For the second digit (ones place), we have `9` choices again because it can be any digit except the one chosen for the tens place. This includes `0`.
  - So for two-digit numbers, we have `9 * 9 = 81` possibilities.
  - Now, our total is `10 + 81 = 91`.
- **Three-digit numbers (`100` to `999`):**
  - Continuing from `91` unique numbers.
  - For the first digit (hundreds place), we still have `9` choices (`1` to `9`).
  - For the second digit (tens place), we have `9` choices.
  - Now, for the third digit (ones place), we have `8` choices because two digits are already used.
  - Multiplying these together, for three-digit numbers, we have `9 * 9 * 8 = 648`.
  - Our total now is `91 + 648 = 739`.

Adding all these up, for `n = 3`, we would have `739` unique digit numbers where `0 <= x < 1000`. Using the pattern described in the Solution Approach, the loop calculates this same total. The pseudo-code for the loop would look like:

- Initialize `ans` with `10` (for `n = 1`).
- For each additional digit place (`i` from `0` to `n - 1 = 2`):
  - set `cur` to `9` for the first iteration.
  - multiply `cur` with `9 - i` to account for the already chosen digits.
  - add the result to `ans`
  - if `i = 0` (`2` digits), `cur = 9 * 9`, add `81` to `ans`; `ans` becomes `91`
  - if `i = 1` (`3` digits), `cur = 9 * 9 * 8`, add `648` to `ans`; `ans` becomes `739`

Therefore, for `n = 3`, the countNumbersWithUniqueDigits function returns `739`.

Solution Implementation

Python

```python
class Solution:
    def countNumbersWithUniqueDigits(self, n: int) -> int:
        # Base case: If n is 0, there's only one number (0 itself) that can be formed
        if n == 0:
            return 1
        # Base case: If n is 1, the numbers 0-9 are all unique, so there are 10
        if n == 1:
            return 10

        # Initialize the count for unique digit numbers with the total for n = 1
        unique_digit_numbers_count = 10
        # Variable to keep track of the count of unique digits for the current number of digits
        current_count = 9  # Starting with 9 because we have 1 to 9 as options for the first digit

        # Loop through the number of digits from 2 to n, as we have already covered n = 1
        for i in range(n - 1):
            # The count of unique numbers for the current digit length is reduced by one less option each time
            # since we're using one more digit and can't repeat any of the lower digits.
            current_count *= 9 - i
            # Add the count for the current number of digits to the overall count
            unique_digit_numbers_count += current_count

        # Return the total count of unique digit numbers for all lengths up to n
        return unique_digit_numbers_count
```

Java

```java
class Solution {

    // This method counts the numbers with unique digits up to a certain length.
    public int countNumbersWithUniqueDigits(int n) {
        // If n is 0, there's only one number which is 0 itself
        if (n == 0) {
            return 1;
        }

        // If n is 1, we have digits from 0 to 9, resulting in 10 unique numbers
        if (n == 1) {
            return 10;
        }

        // Initialize answer with the count for n = 1
        int answer = 10;

        // Current number of unique digits as we increase the length
        int currentUniqueNumbers = 9;

        // Loop to calculate the number of unique digit numbers for lengths 2 to n
        for (int i = 0; i < n - 1; ++i) {
            // Compute the count for the current length by multiplying with the digits
            // available considering we can't reuse any we have already used
            currentUniqueNumbers *= (9 - i);

            // Add the current length's count to the total answer so far
            answer += currentUniqueNumbers;
        }

        // Return the total count of unique numbers with digits up to length n
        return answer;
    }
}
```

C++

```cpp
class Solution {
public:
    int countNumbersWithUniqueDigits(int n) {
        // Base cases:
        // If n is 0, there's only 1 number (0 itself)
        if (n == 0) return 1;
        // If n is 1, there are 10 unique digit numbers (0 to 9)
        if (n == 1) return 10;

        // Start with the count for a 1-digit number
        int count = 10;

        // Current number of unique digits we can use starting from 9
        int uniqueDigits = 9;

        // Loop through the number of digits from 2 up to n
        for (int i = 2; i <= n; i++) {
            // Calculate the number of unique numbers that can be formed with i digits
            // by multiplying the current number of unique digits we can use
            uniqueDigits *= (11 - i);
            // Add the count of unqiue numbers for the current number of digits to the total count
            count += uniqueDigits;
        }

        // Return the total count of numbers with unique digits
        return count;
    }
};
```

TypeScript

```typescript
/**
 * Counts the numbers with unique digits up to the given number of digits n.
 * @param {number} n - The number of digits to consider.
 * @returns {number} - The count of numbers with unique digits.
 */
function countNumbersWithUniqueDigits(n: number): number {
    // Base case for 0 digits
    if (n === 0) return 1;

    // Base case for 1 digit
    if (n === 1) return 10;

    // Initialize count with the total for a single digit
    let count: number = 10;

    // Initialize uniqueDigits with the possible unique digits (9, not including 0)
    let uniqueDigits: number = 9;

    // Iterate through the number of digits from 2 up to n
    for (let i: number = 2; i <= n; i++) {
        // Calculate the count for the current digit position by multiplying with the
        // remaining unique digits (10 - i: since one digit is already used)
        uniqueDigits *= (11 - i);
        // Accumulate the count for the current number of digits
        count += uniqueDigits;
    }

    // Return the total count of numbers with unique digits
    return count;
}
```

```python
class Solution:
    def countNumbersWithUniqueDigits(self, n: int) -> int:
        # Base case: If n is 0, there's only one number (0 itself) that can be formed
        if n == 0:
            return 1
        # Base case: If n is 1, the numbers 0-9 are all unique, so there are 10
        if n == 1:
            return 10

        # Initialize the count for unique digit numbers with the total for n = 1
        unique_digit_numbers_count = 10
        # Variable to keep track of the count of unique digits for the current number of digits
        current_count = 9  # Starting with 9 because we have 1 to 9 as options for the first digit

        # Loop through the number of digits from 2 to n, as we have already covered n = 1
        for i in range(n - 1):
            # The count of unique numbers for the current digit length is reduced by one less option each time
            # since we're using one more digit and can't repeat any of the lower digits.
            current_count *= 9 - i
            # Add the count for the current number of digits to the overall count
            unique_digit_numbers_count += current_count

        # Return the total count of unique digit numbers for all lengths up to n
        return unique_digit_numbers_count
```

Time and Space Complexity

The provided Python code defines a function `countNumbersWithUniqueDigits` which calculates the number of n-digit integers that have unique digits.

- **Time Complexity:** The time complexity of the function is primarily determined by the for loop that iterates `n - 1` times. Within the for loop, there are only constant-time operations. Therefore, the overall time complexity is `O(n)`.
- **Space Complexity:** The space complexity of the function is `O(1)` because the space used does not grow with the input size `n`. The function only uses a constant amount of additional space for variables `ans` and `cur`.