2376. Count Special Integers 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

Hard

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 @ cannot be a starting digit of any special number as the numbers are positive integers. Intuition

potential combinations of distinct digits at different magnitudes (thousands, hundreds, tens, units, etc.). Here's the general intuition behind the solution approach:

To solve this problem efficiently without checking every number individually, we need to use combinatorics to count the number of

larger numbers would not be special.

• We define a helper function A(m, n) which calculates the number of permutations of m distinct elements taken n 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 m digits, we first count all

- special numbers with lengths from 1 to m-1 digits. 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. If 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
- 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.

reverse order for ease of processing from the lowest to the highest digit.

We define a permutation function A(m, n) which calculates permutations of melements taken n 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

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

 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.

numbers that can be formed with the current prefix, and we break the loop.

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

the properties of numbers and the definition of "special" in the problem.

We initialize ans to 0 for counting special numbers.

 For each digit in 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 v of n, using our A 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

 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

Example Walkthrough

 We prepare a vis array to mark digits that have been used. 2. Counting numbers with fewer digits: \circ First, we count special numbers with one digit. There are 9 options (1-9), so $9 \times A(9, 0) = 9$.

Next, we count special numbers with two digits. The first digit has 9 options, and the second has 9 options (0-9, excluding

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)

2. Tens place: Next, we consider the digit 2 as fixed (from n) 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.

We define a permutation function A(m, n) that calculates permutations of m elements taken n at a time.

Starting with the most significant digit, we have:

individually.

2

4

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

28

29

30

31

32

33

34

35

36

37

44

45

46

47

48

49

50

51

52

56

43

44

45

46

47

48

49

50

52

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

57

58

59

51 }

C++ Solution

public:

1 class Solution {

return count;

private int permute(int m, int n) {

int countSpecialNumbers(int n) {

digits.push_back(n % 10);

for (int i = 1; i < m; ++i) {

for (int i = m - 1; i >= 0; --i) {

if (!visited[j]) {

int currentDigit = digits[i];

while (n) {

n /= 10;

Python Solution

1. Initial setup:

The number 320 has three digits.

the first digit), so $9 \times A(9, 1) = 81$.

3. Processing the same length as n:

Add these to ans for a subtotal of 9 + 81 = 90.

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: $1 \times A(9, 2) = 72$.

• For 0 as the tens digit: $1 \times A(8, 1) = 8$.

When the first digit is 2, it's the same situation: 1 x A(9, 2) = 72.

• Add these to ans for a new subtotal: 90 + 72 + 72 = 234.

- For 1 as the tens digit: 1 × A(8, 1) = 8. Update ans: 234 + 8 + 8 = 250.
 - 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. \circ 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

3. Units place: Lastly, we fix the tens digit to 2 (from n) and consider the units digit. We can only use 0 or 1 without

class Solution: def count_special_numbers(self, n: int) -> int: # Recursive function to calculate permutations A(m, n) = m! / (m-n)!def permutations(m, n): if n == 0:

Variable to store the count of special numbers

so we can process the least significant digit first

return permutations(m, n - 1) * (m - n + 1)

Convert the number into a list of its digits in reverse order

answer += permutations(10 - (num_length - i), i)

If we're at the last digit and haven't broken the loop,

then we need to account for this number itself being a special number

print(sol.count_special_numbers(20)) # The output will be the count of special numbers less than or equal to 20

return 1

visited = [False] * 10

num_length = len(digits)

List to keep track of visited digits

digits = [int(c) for c in str(n)[::-1]]

for i in range(num_length - 1, -1, -1):

First digit cannot be 0, others can

start = 1 if i == num_length - 1 else 0

current_digit = digits[i]

while start < current_digit:</pre>

start += 1

if i == 0:

return answer

answer += 1

if not visited[start]:

visited[current_digit] = True

Length of the number (number of digits)

else:

answer = 0

22 23 # Count all special numbers with length less than the length of n 24 for i in range(1, num_length): 25 answer += 9 * permutations(9, i - 1)26 27 # Count special numbers with the same length as n

```
38
                # If the current digit has already been visited, stop the loop
39
                if visited[current_digit]:
40
41
                    break
42
43
                # Mark the digit as visited
```

53 # Example usage:

54 # sol = Solution()

```
Java Solution
    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<>();
  4
             while (n != 0) {
                 digits.add(n % 10);
                 n /= 10;
  8
  9
 10
             // Determine the number of digits in the number
 11
             int numDigits = digits.size();
 12
 13
             // This will hold our final count of special numbers
 14
             int count = 0;
 15
 16
             // Count special numbers with fewer digits than the input number
 17
             for (int i = 1; i < numDigits; ++i) {</pre>
 18
                 count += 9 * permute(9, i - 1);
 19
 20
 21
             // Visit tracking array for digits
 22
             boolean[] visited = new boolean[10];
 23
 24
             // Iterate over each digit, starting with the most significant digit
 25
             for (int i = numDigits - 1; i \ge 0; --i) {
                 int currentValue = digits.get(i);
 26
 27
                 // Count the special numbers smaller than the current number at digit `i`
 28
                 for (int j = i == numDigits - 1 ? 1 : 0; j < currentValue; ++j) {</pre>
                     if (!visited[j]) {
 29
                         count += permute(10 - (numDigits - i), i);
 30
 31
 32
 33
                 // If digit was seen before, no need to continue
 34
                 if (visited[currentValue]) {
 35
                     break;
 36
 37
                 // Mark the digit as visited
                 visited[currentValue] = true;
 38
 39
                 // If we've reached the least significant digit, include it in the count
 40
                 if (i == 0) {
 41
                     ++count;
 42
```

// Recursive method to compute permutations, denoted as A(m, n) = m! / (m - n)!

return n == 0 ? 1 : permute(m, n - 1) * (m - n + 1);

int answer = 0; // Holds the count of special numbers.

// Extract digits of number and store in reverse order.

int m = digits.size(); // Number of digits in n.

answer += 9 * permutation(9, i - 1);

vector<int> digits; // Stores individual digits of the number n.

vector<bool> visited(10, false); // Flags to keep track of digits used.

// Count special numbers less than the given number with fewer digits.

// Counting arrangements for each digit less than currentDigit.

for (int $j = i == m - 1 ? 1 : 0; j < currentDigit; ++j) {$

answer += permutation(10 - (m - i), i);

// Count the special numbers less than n with the same number of digits.

// Counting arrangements for each digit less than currentDigit.

for (let $j = i === m - 1 ? 1 : 0; j < currentDigit; ++j) {$

// If currentDigit has been seen before, break from the loop.

// Reset the visited array for future calls to countSpecialNumbers.

// If last digit hasn't been visited, include this number in the count.

answer += permutation(10 - (m - i), i);

// Count the special numbers less than the given number with same number of digits.

39 40 41

```
29
 30
 31
 32
                 // If currentDigit has already been seen, stop and exit.
 33
                 if (visited[currentDigit]) {
 34
                     break;
 35
 36
 37
                 // 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) {
 42
                     ++answer;
 43
 44
 45
 46
             // Return the total count of special numbers.
 47
             return answer;
 48
 49
         // Helper function to calculate permutations, P(m, n) = m! / (m-n)!.
 50
 51
         int permutation(int m, int n) {
 52
             if (n == 0) return 1; // Base case for permutation.
 53
             return permutation(m, n - 1) * (m - n + 1); // Recursive count of permutations.
 54
 55 };
 56
Typescript Solution
  1 // 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 permutation(m: number, n: number): number {
         if (n === 0) return 1; // Base case for permutation.
         return permutation(m, n - 1) * (m - n + 1); // Recursive count of permutations.
  8
  9
    // Main function to count special numbers up to a given number n.
    function countSpecialNumbers(n: number): number {
         let answer: number = 0; // Holds the count of special numbers.
 12
 13
         const digits: number[] = []; // Stores individual digits of the number n in reverse order.
 14
 15
         // Extract digits of n and store in reverse order.
         while (n) {
 16
             digits.push(n % 10);
 17
 18
             n = Math.floor(n / 10);
 19
 20
 21
         const m: number = digits.length; // The number of digits in n.
 22
         // Count special numbers less than n with fewer digits.
 23
 24
         for (let i = 1; i < m; ++i) {
 25
             answer += 9 * permutation(9, i - 1);
```

Time and Space Complexity

for (let i = m - 1; i >= 0; --i) {

if (!visited[j]) {

if (visited[currentDigit]) {

visited[currentDigit] = true;

// Mark the current digit as visited.

// Return the total count of special numbers.

break;

if (i === 0) {

visited.fill(false);

return answer;

recursively n times.

++answer;

const currentDigit: number = digits[i];

The outer loop runs for i from 1 to m, where m 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 v. In the worst-case scenario, this loop could iterate 9 times (if v is 9 and j starts from 0). However, this doesn't run for every digit because once a repeated digit is detected, the

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

loop breaks. Thus, this contributes less than O(9*m) complexity. The function A(m, n) is a recursive implementation of arranging melements in n places, which takes O(n) time since it calls itself

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(m²) 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.