233. Number of Digit One Recursion Math Dynamic Programming Leetcode Link Hard

Problem Description The problem statement is asking us to calculate the total count of the digit 1 in the range of all non-negative integers less than or

equal to a given integer n. In other words, if n is 13, for instance, we should count how many times the digit 1 appears in the following sequence of numbers: 0, 1, 2, 3, ..., 13. This would include occurrences of 1 in numbers like 10, 11, 12, and 13, not just the single occurrence in the number 1.

Intuition

The solution provided leverages a method known as "Digital Dynamic Programming" (Digital DP). This approach optimizes the counting process by breaking it down into a digit-by-digit examination, considering the recurrence and patterns of ones appearing in each positional digit.

The intuitive approach for this problem might involve iterating through all the numbers from 0 to n and count the digit 1 in each

number. However, such a method would not be efficient for larger values of n.

state (pos, cnt, limit) is computed only once, further improving efficiency.

The process is as follows: 1. Reverse engineer the number n to obtain a digit array a, which stores the digits of n in reverse order. For instance, if n is 213, a

will be [3,1,2].

- 2. We define a recursive function dfs(pos, cnt, limit) where pos indicates the current position we're checking in a, cnt counts the number of ones we've found so far, and limit is a flag indicating if we've reached the upper limit (the original number n).
- 1s (cnt). 4. For each recursive call, if limited by n (limit is true), we use the digit in a [pos] as our upper bound. This represents counting up

3. The base condition of the recursive function is when pos <= 0, meaning we've checked all positions and we return the count of

- to the current digit in the original number n. If not limited, we can go up to 9 (all possibilities for a decimal digit). 5. The recursive calls iterate from 0 up to the upper bound, incrementing cnt if the current digit is 1 and proceeding to the next
- position to the left (pos 1). 6. A dynamic programming optimization is applied by caching the results of the recursive calls. This ensures that each distinct
- subproblems to avoid redundant calculations, resulting in more optimal time complexity compared to simple iterative counting.

In essence, this Digital DP solution breaks down the large problem into smaller subproblems and caches the solutions to these

The solution approach for the problem utilizes a recursive algorithm with memoization (a dynamic programming technique) to efficiently compute the count of digit 1 in the given range. Let's walk step-by-step through the implementation details:

The input integer n is broken down into its constituent digits and stored in reverse order in an array a. This reversal puts the

When pos is 0, the recursion returns cnt, which means no more digits are left to process, so we've counted all occurrences

least significant digit at a [1] and progresses towards the most significant digit. This is convenient for our recursive function,

which will build the count from the least significant digit upward. 2. Recursive Function (Dynamic Programming):

3. Base Condition:

of 1.

4. Building Count and Recursing:

Data Structures and Patterns

Example Walkthrough

2. Recursive Function Initialization:

4. Recursive Calls and Count Building:

Calls dfs(1, 0, True) for 0

Calls dfs(0, cnt, True) for 0

Calls dfs(0, cnt, True) for 2

the stored result instead of recomputing it.

@lru_cache(maxsize=None)

if position == 0:

ans = 0

n //= 10

length += 1

return dfs(length, 0, True)

private int[] digits = new int[12];

n /= 10;

for (int[] row : memo) {

Arrays.fill(row, -1);

return dfs(length, 0, true);

// Perform a depth-first search.

if (position <= 0) {</pre>

private int[][] memo = new int[12][12];

return count_ones

def dfs(position, count_ones, is_limit):

Determine the upper bound for the current digit.

upper_bound = digits[position] if is_limit else 9

Initialize the answer for this position

for digit in range(upper_bound + 1):

Otherwise, it is 9 (the maximum value for a digit).

Iterate over all possible digits for this position

digits[length + 1] = n % 10 # Store the digit

// The 'a' array holds the digits of the number in reverse order.

// The 'dp' array memoizes results for dynamic programming approach.

// This method counts the number of digit '1's in numbers from 1 to 'n'.

// Start the depth-first search from the most significant digit.

// If we have processed all the digits, return the count of '1's.

digits[++length] = n % 10; // Store the last digit of n.

memset(memo, -1, sizeof memo); // Initialize the memoization array to -1.

int ans = 0; // Initialize the answer for the current recursion level.

ans += dfs(pos - 1, count + (i == 1), limit && i == upperBound);

memo[pos][count] = ans; // If not at the limit, memoize the result.

return dfs(length, 0, true); // Start the depth-first search from the most significant digit.

return count; // Base case: If all positions are traversed, return the count of '1's.

let upperBound = limit ? digits[pos] : 9; // Determine the upper bound for the current digit.

return memo[pos][count]; // If we are not at the limit and we have a memoized result, return it.

// Calculate the count of '1's for the next position, updating count if the current digit is '1'.

17 // This function uses depth-first search to count the number of occurrences of the digit 'l'.

ans += dfs(pos - 1, count + (i === 1 ? 1 : 0), limit && i === upperBound);

memo[pos][count] = ans; // If not at the limit, memoize the result.

return ans; // Return the computed answer for the current digit position.

return dfs(length, 0, true); // Start the DFS from the most significant digit.

// This method uses depth-first search to count the number of occurrences of the digit '1'.

return count; // Base case: If all positions are traversed, return the count of '1's.

int upperBound = limit ? digits[pos] : 9; // Determine the upper bound for the current digit.

return memo[pos][count]; // If we are not at the limit and we have a memoized result, return it.

// Calculate the count of '1's for the next position, updating count if the current digit is '1'.

n /= 10; // Remove the last digit from n.

// Enumerate possibilities for the current digit.

digits[++length] = n % 10; // Store the last digit of n.

n = Math.floor(n / 10); // Remove the last digit from n.

// Reset the memoization array to -1 for each new computation.

memo = Array.from(Array(12), () => Array(12).fill(-1));

function dfs(pos: number, count: number, limit: boolean): number {

let ans = 0; // Initialize the answer for the current recursion level.

if (!limit && memo[pos][count] !== -1) {

for (let i = 0; i <= upperBound; ++i) {

// Enumerate possibilities for the current digit.

int dfs(int pos, int count, bool limit) {

if (!limit && memo[pos][count] != -1) {

for (int i = 0; i <= upperBound; ++i) {</pre>

private int dfs(int position, int countOfOnes, boolean isLimited) {

// Initialize memoization array with -1 to represent uncalculated states.

limited by the original number's digits.

are still within the bounds of the original number.

arguments of the recursive function to avoid redundant calculations.

resulting in the final count of the digit 1 in all numbers less than or equal to n.

through all numbers and leverages subproblem solutions.

1. Digit Array Preparation:

Algorithm

Solution Approach

• The core of the solution is a recursive function dfs(pos, cnt, limit) which is defined within the class. It carries the current position pos in the array a, the current count cnt of ones, and a boolean limit that determines whether the current path is

• The recursive function spans across all possible digits from 0 up to a [pos] if limit is True, or up to 9 if limit is False. For each digit, the function calls itself with pos - 1, and incrementing count if the current digit is 1, passing on the limit if we

Memoization

Memoization is achieved via the @cache decorator from the functools module (in Python 3.9+). This caches the results based on the

 Digit Array: An array a is used to hold the individual digits of the number in reverse order. • Caching: The @cache decorator on top of the recursive function implicitly creates a dictionary (or similar data structure) where each unique state (combinations of pos, cnt, and limit) is stored alongside the computed result. Digital Dynamic Programming (Digital DP): This pattern involves dissecting a numerical problem into digital-level subproblems,

The Solution class applies this algorithm by first initializing the digit array a and calling the dfs function with the starting parameters,

This approach is significantly more efficient for larger numbers compared to a brute force method, as it avoids counting digit by digit

and solving them using the principles of dynamic programming, allowing overlap and reuse of subproblem solutions.

equal to n. 1. Digit Array Preparation:

First, we break down 23 into its constituent digits 2 and 3 and reverse their order to get the digit array a = [3, 2].

Let's assume we have n = 23. We want to calculate the total count of the digit 1 in the range of all non-negative integers less than or

pos is the length of the array a), with a cnt of 0 because we haven't counted any 1's yet, and with the limit boolean set to True because we can't exceed the number n. 3. Initial Recursive Call:

We call dfs(2, 0, True) corresponding to the most significant digit (2 in number 23), with a count of 0 and limit as True.

It iterates through digits from 0 to 2 (since limit is True and a [2] is 2).

This will iterate from 0 to 3 (since limit is True and a [1] is 3).

Calls dfs(0, cnt, True) for 1 (found one '1', so cnt will increase by 1)

Calls dfs(0, cnt, True) for 3 (limit is now False for 3 because it's not equal to a[1])

• We initialize our recursive function dfs(pos, cnt, limit) and start at the most significant position (in a, the starting position

Calls dfs(1, 0, True) for 1 (found one '1', so cnt will increase by 1) Calls dfs(1, cnt, True) for 2 (limit remains True for 2 because it's equal to a[2])

o dfs(2, 0, True)

o dfs(1, cnt, True)

recursive calls.

6. Memoization:

7. Counting Ones:

9

11

12

13

14

15

16

17

18

19

20

21

30

31

32

33

34

35

8

9

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

39

9

10

11

12

13

14

16

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

15 }

38 };

while (n) {

if (pos <= 0) {

if (!limit) {

Typescript Solution

while (n > 0) {

if (pos <= 0) {

if (!limit) {

1 let digits: number[] = Array(12).fill(0);

5. Base Condition and Count Accumulation: odfs(0, cnt, limit) will hit the base condition pos == 0, and return the cnt which is the count of ones accumulated from the

Through these recursive calls, if we encounter the same pos, cnt, and limit state, the @cache decorated dfs function will use

```
    From the iteration on dfs(1, 0, True), we have found '1' at the second position only once in the numbers from 0-9 (i.e., 1,

        since every other number from 10-19 is outside our limit, it is counted only once).
      • From the iteration on dfs(1, 1, True), when the first digit is '1', we add the instances, we get the numbers 10, 11, 12, 13
        (which counts for another 1), 14, 15, 16, 17, 18, 19 - where '1' occurs ten times due to the second digit ranging from 0-9.
By summing these up, we get a total count of the digit '1' in the range of all non-negative integers less than or equal to n = 23, which
is 13 times. This involves counting the singular '1's in the range 1-9, the tens digit '1' in the range 10-19, and the one's digit '1' in 21.
Python Solution
   from functools import lru_cache
   class Solution:
       def countDigitOne(self, n: int) -> int:
           # Use lru_cache decorator to memoize the results of the recursive calls
```

Base case: if no more digits to process, return the count of ones found

If is_limit is True, the upper bound is the current digit in the number.

22 ans += dfs(position - 1, count_ones + (digit == 1), is_limit and digit == upper_bound) 23 return ans 24 25 # Convert the number to a list of its digits, reversed. 26 digits = [0] * 1227 length = 028 while n:

Calculate the answer recursively. Increase the count if the current digit is 1.

Limit flag is carried over and set to True if we reached the upper_bound.

Move to the next digit

Invoke the recursive DFS function starting with the most significant digit

```
10
        public int countDigitOne(int n) {
11
            int length = 0;
12
13
            // Split the integer 'n' into its digits.
           while (n > 0) {
14
15
                digits[++length] = n % 10;
```

Java Solution

1 public class Solution {

```
32
                 return countOfOnes;
 33
 34
 35
             // If we are not limited by the most significant digit and we have computed this state before, return the memoized result.
 36
             if (!isLimited && memo[position][countOfOnes] != -1) {
 37
                 return memo[position][countOfOnes];
 38
 39
 40
             // Determine the upper bound of the current digit we can place.
 41
             int upperBound = isLimited ? digits[position] : 9;
 42
             int sum = 0;
 43
 44
             // Try all possible digits at the current position.
 45
             for (int digit = 0; digit <= upperBound; ++digit) {</pre>
                 // Accumulate results from deeper levels, adjusting the count of '1's and the limit flag.
 46
 47
                 sum += dfs(position - 1, countOfOnes + (digit == 1 ? 1 : 0), isLimited && digit == upperBound);
 48
 49
 50
             // If we are not limited, memoize the result for the current state.
 51
             if (!isLimited) {
                 memo[position][countOfOnes] = sum;
 52
 53
 54
 55
             // Return the calculated sum for the current state.
 56
             return sum;
 57
 58
 59
         // Helper method to fill the 'dp' array with a value.
 60
         private static void fillDp(int[][] dp, int value) {
             for (int[] row : dp) {
 61
 62
                 Arrays.fill(row, value);
 63
 64
 65
 66
C++ Solution
  1 class Solution {
    public:
         int digits[12];
         int memo[12][12];
  5
  6
         // This method calculates the number of digit '1's that appear when counting from 1 to the given number n.
         int countDigitOne(int n) {
             int length = 0; // Initialize the length to store the number of digits in n.
  8
             // Store the digits of n in reverse order.
  9
```

let memo: number[][] = Array.from(Array(12), () => Array(12).fill(-1)); 4 // This function calculates the number of digit '1's that appear when counting from 1 to the given number n. function countDigitOne(n: number): number { let length = 0; // Initialize the length to store the number of digits in n. // Store the digits of n in reverse order.

```
34
 35
         return ans; // Return the computed answer for the current digit position.
 36 }
 37
Time and Space Complexity
Time Complexity
The time complexity of the given Python code involves analyzing the depth-first search (dfs) function. The dfs function is called
recursively with several key conditions that impact its running time:
  • The recursion depth is determined by the parameter pos, which can be as large as the number of digits in n. Let's denote the
   number of digits in n as d.

    For each call to dfs, there is a loop that iterates at most 10 times (digits 0 through 9), represented by the variable up.

  • The function uses memoization via the @cache decorator, significantly reducing the number of calculations by caching and
   reusing results of previous computations.
```

Considering these points, the time complexity can be approximated by 0(d * 10 * d), since dfs will be called for d levels, and at

each level, it will iterate through up to 10 possibilities, and the work done at each level is 0(d) to handle the limiting cases where

limit is True. Memoization ensures that results for each unique combination of (pos, cnt, limit) are not recomputed, which might

The space complexity comprises the space used by the recursive call stack and the space required to store the memoized results: The recursion can go as deep as d, representing the space used by the call stack.

Space Complexity

```
• The @cache decorator uses space to store results of unique combinations of arguments to the dfs function. The number of
 unique argument combinations can be up to d * 2 * d, since pos can take up to d values, cnt can take up to d values (as it
```

otherwise lead to an exponential time complexity of 0(10^d).

Hence, with memoization, the final time complexity is $0(d^2 * 10)$.

counts the number of 1s and there are at most d ones), and limit can be either True or False. Thus, the space complexity is $O(d^2)$ for the memoization and the call stack together.

Overall, the space complexity can be represented as $O(d^2)$.