2048. Next Greater Numerically Balanced Number

Medium Math Backtracking **Enumeration**

number one by one until it finds the solution.

number is numerically balanced or not.

Problem Description

that digit within the number itself. For example, the number 122 is numerically balanced because there is exactly one 1 and two 2 S. The problem at hand involves finding the smallest numerically balanced number that is strictly greater than a given integer n.

An integer is considered *numerically balanced* if for each digit d that appears in the number, there are exactly d occurrences of

That means if n is 1, then the next numerically balanced number is 22 because it's the smallest number greater than 1 that satisfies the numerically balanced condition (two instances of the digit 2). Intuition

The intuition behind the solution is to start from the number immediately greater than n and check each successive number to see if it meets the criteria of being numerically balanced. This is essentially a brute-force approach since it examines each

The function check(num) assesses whether a number num is numerically balanced. It creates a list counter that keeps track of the occurrences of each digit. The number is converted into a string to iterate over its digits, updating counter accordingly. Following that, the function verifies whether the occurrence of each digit in counter matches its value, thus confirming the

This method is not the most efficient solution, particularly for large n, because it potentially evaluates many numbers until it finds the next numerically balanced number. However, it's a straightforward and easy-to-understand approach. **Solution Approach**

The solution uses a straightforward brute-force approach. Here's a step-by-step guide to how the algorithm is implemented: Validation Function (check): A key part of the solution is the check function, which takes an integer num as input and

Convert num to a string to iterate over its digits.

 Increment the counter at the index corresponding to each digit d by 1 for each occurrence of d in num. o Iterate over the digits of num again, and for each digit d, check if counter[d] is equal to the digit d itself. This step confirms that the

- number of occurrences of each digit d is equal to its value. If any digit d does not meet the balance condition, return False.
 - n.

Start iterating from n+1, since we are looking for a number strictly greater than n.

determines if num is numerically balanced. The function does the following:

Initialize a counter list of 10 zeros, with each index representing digits 0 to 9.

- If all digits satisfy the condition, return True indicating the number is numerically balanced. Main Function (nextBeautifulNumber): This function looks for the smallest numerically balanced number strictly greater than
- Call the check function for each number i in the iteration. Stop the iteration and return the current number i as soon as a numerically balanced number is found. • The upper limit of 10**7 is an arbitrary large number to ensure that the next numerically balanced number is found within practical

Efficiency: While the provided approach is correct, it's important to note that the efficiency isn't optimal for large numbers

because it may require checking a large number of candidates. Advanced techniques or optimizations such as precomputing

possible balanced numbers or using combinatorics to generate candidates more intelligently could potentially improve

computational time.

basic patterns used in brute-force algorithm implementations.

which meets the requirement of being numerically balanced.

significant, which highlights the inefficiency for larger inputs.

Initialize a counter for each digit from 0 to 9.

If the number is beautiful, return it.

Increment the counter for each digit found in the number.

Check if the frequency of each digit is equal to the digit itself.

If no beautiful number is found (which is highly improbable), return -1.

// Finds the next beautiful number that is greater than a given number `n`

// Start from the next integer value and check up to an upper limit

// Utilize the check method to determine if the number is beautiful

efficiency. The algorithm uses iteration and simple counting, making it easily understandable. The check function uses a list as a counting

Starting from 301 (the next integer after n), we apply check(num) to each subsequent integer:

Example Walkthrough Let's take an integer n = 300. According to the problem description, we need to find the smallest numerically balanced number

mechanism, which is an instance of the array data structure, and the iterations over the sequence of numbers and their digits are

and one 1, so 301 is not numerically balanced. Check 302, 303, ..., up until 312: None of these numbers will be numerically balanced because there will never be three 3s, or two 2s until at least 322 (the instance where the digit matches its count).

Check 301: Is there one 3, zero 0's, and one 1? No, because there has to be zero 0's (which is correct) but also three 3's

Check 312: It is not numerically balanced either because there's only one 1 and one 3, but there should be three 3 s and one

Skip forward to 322: This is the first number where the digit 2 appears twice, satisfying its requirement. But we need three

greater than n.

occurrences of 3 and none of the numbers 313 to 322 meet this criteria. Proceeding with this method, the first number to satisfy the checking criteria after 300 is 333. In 333, there are exactly three 3s,

be returned by the nextBeautifulNumber function. Thus, applying the brute-force approach outlined in the solution, we conclude that the next numerically balanced number greater than 300 is 333. While this method is straightforward, it is also clear that for a large number the number of iterations can become

This example shows that the check function will iterate over the numbers 301, 302, 303, ..., 332, 333, and when it reaches 333,

it will return True, indicating that this is the smallest numerically balanced number greater than 300. The number 333 will then

Python class Solution: def nextBeautifulNumber(self, n: int) -> int: # Define a function to check if a number is 'beautiful'.

A number is beautiful if the frequency of each digit is equal to the digit itself.

digit = int(digit char) if frequency counter[digit] != digit: return False return True # Iterate through numbers greater than n to find the next beautiful number.

This is a safeguard, and in practice, this return statement should not be reached. return -1

Java

class Solution {

Solution Implementation

def is beautiful(num):

frequency_counter = [0] * 10

for digit char in str(num):

for digit char in str(num):

for i in range(n + 1, 10**7):

public int nextBeautifulNumber(int n) {

if (isBeautiful(i)) {

for (int i = n + 1; i < 10000000; ++i) {

for (int i = n + 1; i < 100000000; ++i) {

if (isBeautiful(i)) {

return i;

bool isBeautiful(int num) {

return -1;

private:

};

TypeScript

// If i is a beautiful number, return it.

// Helper function to check if a number is beautiful.

// Initialize a counter for each digit (0-9).

// Increment the counter for each digit found in the number.

// Check if the count of each digit matches its value.

if (digitCounter[digit - '0'] != digit - '0') {

// All digit counts match their values, so the number is beautiful.

// Convert the number to a string.

string numStr = to_string(num);

vector<int> digitCounter(10, 0);

++digitCounter[digit - '0'];

for (char digit : numStr) {

for (char digit : numStr) {

return false;

return true;

if is beautiful(i):

return i

digit = int(digit char)

frequency_counter[digit] += 1

```
// Return the first beautiful number found
                return i;
        // Return -1 if no beautiful number is found (should not occur with the given constraints)
        return -1;
    // Helper method to check if a number is beautiful
    private boolean isBeautiful(int num) {
        // Initialize a counter to store the frequency of each digit
        int[] digitCounter = new int[10];
        // Convert the number to a character array
        char[] chars = String.valueOf(num).toCharArray();
       // Increment the count for each digit found
        for (char c : chars) {
            digitCounter[c - '0']++;
       // Check if each digit appears exactly as many times as the digit itself
        for (char c : chars) {
            if (digitCounter[c - '0'] != c - '0') {
                // If not, the number is not beautiful
                return false;
       // If all digits match the criteria, the number is beautiful
        return true;
C++
#include <vector>
#include <string>
using namespace std;
class Solution {
public:
    // Function to find the next beautiful number greater than a given number n.
    int nextBeautifulNumber(int n) {
        // Loop starts from the next number after n and stops when it finds a beautiful number.
```

// Return -1 if no beautiful number is found (this won't happen given the problem constraints).

// A beautiful number has each digit appearing exactly d times where d is the actual digit.

// If any digit count doesn't match its value, the number isn't beautiful.

```
function nextBeautifulNumber(n: number): number {
   // Start checking numbers greater than the given number
   for (let answer = n + 1; ; answer++) {
        // Check if the current number is "beautiful"
        if (isValid(answer)) {
           // If it's a beautiful number, then return it as the answer
            return answer;
function isValid(n: number): boolean {
   // Initialize digit occurrence record with zeros
   let digitFrequency = new Array(10).fill(0);
   // While there are still digits to process
   while (n > 0) {
       // Get the rightmost digit of the number
       const currentDigit = n % 10;
       // Increment the frequency of the currentDigit
       digitFrequency[currentDigit]++;
       // Remove the rightmost digit from the number
       n = Math.floor(n / 10);
   // Check all digits from 0 to 9
   for (let i = 0; i < 10; i++) {
       // If a digit occurs and its frequency is not equal to the digit itself, it's not valid
        if (digitFrequency[i] && digitFrequency[i] != i) return false;
   // If all conditions are met, return true
   return true;
class Solution:
   def nextBeautifulNumber(self, n: int) -> int:
       # Define a function to check if a number is 'beautiful'.
       # A number is beautiful if the frequency of each digit is equal to the digit itself.
       def is beautiful(num):
           # Initialize a counter for each digit from 0 to 9.
            frequency counter = [0] * 10
           # Increment the counter for each digit found in the number.
            for digit char in str(num):
               digit = int(digit char)
               frequency_counter[digit] += 1
           # Check if the frequency of each digit is equal to the digit itself.
            for digit char in str(num):
               digit = int(digit char)
               if frequency counter[digit] != digit:
                   return False
           return True
       # Iterate through numbers greater than n to find the next beautiful number.
       for i in range(n + 1, 10**7):
           if is beautiful(i):
```

Time Complexity

Time and Space Complexity

return -1

digit itself.

return i

If the number is beautiful, return it.

If no beautiful number is found (which is highly improbable), return -1.

This is a safeguard, and in practice, this return statement should not be reached.

The time complexity of the check function is as follows: 1. Converting a number to its string representation is <code>0(log(num))</code>, because the length of the string is proportional to the number of digits in the number. 2. Counting the digits by using an array of size 10 (for each possible digit) is done in O(log(num)), where log(num) is the number of digits in num. 3. Again, it iterates over each digit to verify if the number is beautiful, which is also <code>0(log(num))</code>.

The function nextBeautifulNumber increments a number, starting from n + 1, and checks each number to see if it is a "beautiful"

number as defined by the function check. The check function verifies if the number of times each digit appears is equal to the

The loop can be considered to go on until the next beautiful number is found, which is the worst-case scenario since we do not know the distribution of beautiful numbers. If we say finding the next beautiful number from n has a maximum range of r, the total time complexity is O(r * log(r)).

Combining these steps, the function check alone is O(log(num)). However, we need to consider that it is called for each number

The exact value of r is not straightforward to determine as it depends on the pattern distribution of beautiful numbers, but given the constraint that the for loop can go up to 10^{7} , in the worst case scenario r would be close to 10^{7} , making it $0(7 * 10^{7})$,

Space Complexity The space complexity is primarily impacted by:

which simplifies to $0(10^7)$.

starting from n + 1.

- 1. The counter array which holds 10 integers, representing each digit, so it remains constant 0(1). 2. The string representation of the number being checked inside the loop, which has a space complexity of O(log(num)) for each number num.
- Since the counter array is static and doesn't grow with the input, and the string representation of the current number is only stored temporarily for each check, the space complexity can be simplified to 0(1) because the logarithmic space required for the

string representation does not affect the overall space complexity which is dominated by the fixed size of the counter array.