2549. Count Distinct Numbers on Board

Easy | Array | Hash Table | Math | Simulation

Problem Description

In this problem, you have a board with a single positive integer `n` on it. You are required to follow a daily procedure for `10^9` days, which involves two steps. In the first step, for every number `x` currently on the board, you need to find all the numbers `i` in the range `1 <= i <= n`, which satisfy the condition `x % i == 1`. In the second step, all the numbers you've found in the first step are added to the board. The task is to determine the number of *distinct* integers on the board after `10^9` days.

A key thing to note is that once a number is placed on the board, it stays there for the entire duration.

Intuition

The first intuition is that the process seems complex and potentially time-consuming, especially given the `10^9` days. However, we notice that the nature of the modulo operation being performed (`x % i == 1`) simplifies things. This is because for a given `x`, as `i` increases, the set of `i`'s that satisfy the condition doesn't change past a certain point, since we are solely interested in `i` values that result in a remainder of `1` when `x` is divided by `i`.

By analyzing smaller cases, we can observe that the numbers added to the board on the first day are `[2, 3, 4, ..., n]` — because `n % i == 1` for `i` ranging from `1` to `n-1`. From the next day onwards, additional multiples do not get added to the board since these would only generate sequences similar to the first day but with bigger numbers which do not go up to `n`. The operation `x % i == 1` only yields numbers smaller than `x`, so the numbers produced are already on the board.

Given this, we can deduce that after the first day, there would be no new numbers added to the board. Thus, the distinct integers present on the board would be all the numbers from `2` up to `n`, inclusive. This gives us a total count of `n-1` distinct numbers (as we start counting from `2`). If `n` is `1`, there are no additional numbers that we can add, so the result would be `1`. This knowledge lets us solve the problem in constant time with the simple solution `max(1, n - 1)`.

Solution Approach

The reference solution provided is a direct translation of the intuition behind the problem into code. Since we have deduced that no new numbers will be added to the board after the first day, we don't need to simulate the entire `10^9` days. Instead, we just need to calculate the total number of distinct numbers after the first day.

No complex algorithms or data structures are needed here, and there's no need for a pattern as the observation leads to a one-step solution.

The implementation is straightforward:

```
class Solution:
    def distinctIntegers(self, n: int) -> int:
        return max(1, n - 1)
```

The function `distinctIntegers` simply returns the result of `max(1, n - 1)`. The `max()` function is used to handle the edge case where `n` is `1`. In this case, there can only be one number (which is `1` itself) on the board. For all other values of `n` greater than `1`, the function returns `n - 1`, which represents all the distinct numbers from `2` to `n` inclusive.

It's important to note that the efficiency of this solution is O(1), meaning that the time complexity does not depend on the size of the input `n`. Hence, it is an extremely efficient solution both in terms of time and space complexity.

Example Walkthrough

Let's illustrate the solution approach using a small example with `n = 5`. According to the problem statement, we start with the integer `5` on the board:

- **Day 1:**
  - We perform the operation `x % i == 1` for each `i` such that `1 <= i <= n`. In this case, `x` is the initial number `5`.
  - We find the integers: `5 % 1 == 0`, `5 % 2 == 1`, `5 % 3 == 2`, `5 % 4 == 1`, and `5 % 5 == 0`.
  - The only `i` values that satisfy the condition `x % i == 1` are `2` and `4`.
  - Therefore, we add the numbers `2` and `4` to the board.
  - So, at the end of Day 1, the board contains `2`, `4`, and the initial `5`.
- **After Day 1:**
  - We will notice that any `x` on the board going forward will have been generated by using the `%` operation with a starting number of `5`.
  - Now, repeating the operation `x % i == 1` for each new `x` (which are `2` and `4`) and each `i` in the range `1` to `5` will not provide any new integers satisfying the condition that are not already on the board.
  - Since `2` and `4` are less than our starting `n (5)`, all the resultant numbers on applying `x % i == 1` are less than `2` and `4`, respectively, and are thus not within the range `1` to `5`. This means no new numbers will be added to the board after Day 1.

Therefore, the board will contain the original number `5` plus every number from `2` to `5 - 1` (inclusive). This is exactly `5 - 1 = 4` numbers, which are `2`, `3`, `4`, and `5`.

No matter how many days pass, no new numbers will be added to the board since any number `x` will not yield a new `i` that satisfies `x % i == 1` within the range `1` to `n`.

The solution then is the function:

```
class Solution:
    def distinctIntegers(self, n: int) -> int:
        return max(1, n - 1)
```

For our example with `n = 5`, the function call `distinctIntegers(5)` will return `max(1, 5 - 1)`, which is `max(1, 4)`, resulting in `4`. This matches our walkthrough, where the distinct numbers on the board after `10^9` days are `2`, `3`, `4`, and `5`.

Solution Implementation

Python

```
class Solution:
    def distinctIntegers(self, n: int) -> int:
        # This function calculates the maximum number of distinct integers.
        # It ensures there is at least one distinct integer by returning a minimum of 1
        # and otherwise returns one less than the input number `n`.

        # Use max() to ensure the result is at least 1
        return max(1, n - 1)
```

Java

```
class Solution {
    // Method to calculate the number of distinct integers
    public int distinctIntegers(int n) {
        // Returns the maximum between 1 and (n - 1)
        // This ensures that for n less than 2, the result is always 1
        return Math.max(1, n - 1);
    }
}
```

C++

```
class Solution {
public:
    // Function to calculate the number of distinct integers
    int distinctIntegers(int num) {
        // The logic assumes that the minimum number of distinct integers
        // you can have is 1 (when num is either 0 or 1)
        // and the maximum is when you subtract 1 from num (for num > 1),
        // since the problem may refer to counting distinct integers given a certain condition,
        // which is not mentioned here, hence the simplified approach.

        // Using std::max to ensure the number returned is at least 1
        return std::max(1, num - 1);
    }
};
```

TypeScript

```
// Returns the count of distinct integers that can be formed with 'n' elements
// where the difference between each element must be at least one
// except for the base case where n is 0 or 1, return 1 (only one distinct integer can be formed - zero itself)
function distinctIntegers(n: number): number {
    // Ensure there's at least one distinct integer when n is 0 or 1
    // For n greater than 1, return n - 1 because we can have a maximum of n-1 distinct integers
    // with a difference of at least 1 between each integer
    return Math.max(1, n - 1);
}


class Solution:
    def distinctIntegers(self, n: int) -> int:
        # This function calculates the maximum number of distinct integers.
        # It ensures there is at least one distinct integer by returning a minimum of 1
        # and otherwise returns one less than the input number `n`.

        # Use max() to ensure the result is at least 1
        return max(1, n - 1)
```

Time and Space Complexity

Time Complexity

The given Python function `distinctIntegers` executes a direct comparison and returns the result of `max(1, n - 1)`. It does not have any loops or recursive calls. As such, the number of operations is constant, regardless of the size of `n`. Therefore, the time complexity of the function is `O(1)`.

Space Complexity

The function only uses a fixed amount of space for the input variable `n` and does not allocate any additional space that depends on the input size. It also does not use any auxiliary data structures like arrays, lists, or dictionaries. Consequently, the space complexity of the function is also `O(1)`.