287. Find the Duplicate Number

Medium | Bit Manipulation | Array | Two Pointers | Binary Search | Leetcode Link

Problem Description

In this problem, we're given an array called `nums` which contains `n + 1` integers, where every integer is within the range of `1` to `n`, both inclusive. There's an important constraint in the problem: there is exactly one number in the array `nums` that appears more than once, and our task is to find that one repeated number. The challenge is to solve this problem under the following two conditions: we are not permitted to alter the original array, and we have to solve it using only a constant amount of space, which eliminates the possibility of using additional data structures that can grow with the size of the input.

Intuition

We can approach this problem with a binary search technique despite the fact that the array is not sorted. This might seem counterintuitive at first because binary search usually requires a sorted array. However, the key insight here is to use binary search not on the elements of the array itself, but rather on the range of numbers between `1` to `n`.

The intuition is based on the Pigeonhole Principle which states that if you have more pigeons than pigeonholes, at least one pigeonhole must contain more than one pigeon. In this context, if there are more numbers in the array than the range it covers (`n` numbers in the range `1` to `n`), one of the numbers must be a duplicate.

We start by considering the entire range of numbers from `1` to `n`. Then, we use binary search to split this range into two halves: the first half (from `1` to `mid`) and the second half (from `mid + 1` to `n`). The helper function, `f`, calculates how many numbers in the array are less than or equal to a given middle value, `x`. If the count is greater than `x`, we know the duplicate number must be in the first half; otherwise, it's in the second half.

By repeatedly halving the search space, we can eventually narrow down the range to a single number, which is the duplicate we're looking for. The `bisect_left` function in Python assists us in performing this binary search, and the key `f`, is passed to determine whether we should go left or right in our search.

Solution Approach

The solution approach for finding the duplicate number in the array leverages binary search, which is an efficient algorithm for finding an item from a sorted list by repeatedly dividing the search interval in half. Although the array itself is not sorted, we use binary search on the range of possible numbers (`1` to `n`) to find the duplicate.

Let's walk through the implemented solution step by step:

1. **Define the Helper Function `f(x)`:** This function takes an integer `x` and returns `True` if the number of elements in `nums` less than or equal to `x` is greater than `x` itself. Otherwise, it returns `False`. This function is essential because it determines whether the duplicate number lies in the lower half (`1` to `x`) or the upper half (`x + 1` to `n`) of the current search interval.

2. **Binary Search with `bisect_left`:** We use Python's `bisect_left` function from the `bisect` module to apply binary search. The `bisect_left` function takes three arguments:

   - The range to perform the search on, which is `range(len(nums))`. Note that this range goes from `0` to `n` inclusive since the array `nums` has `n + 1` elements.
   - A boolean value that we are trying to find in the hypothetical sorted array of booleans, `True` in this case since we are looking for the point where `f(x)` transitions from `False` to `True`.
   - The `key` function, which in our solution is the helper function `f`. This function is applied to the middle value in the current search interval to guide the binary search process.
3. **Finding the Duplicate Number:** The binary search proceeds by checking the middle of the current interval. If `f(mid)` is `True`, it means there are more numbers in `nums` that are less than or equal to `mid` than there should be, indicating that the duplicate number must be less than or equal to `mid`. If `f(mid)` is `False`, it means that the duplicate is greater than `mid` and we shift our search to the upper half. This process continues until the algorithm converges on the duplicate number, which will be the point at which `f(x)` changes from `False` to `True`.

By repeatedly narrowing the search interval, we eventually find the duplicate number with `O(log n)` search iterations, with each iteration involving an `O(n)` operation to calculate the sum within the helper function. Overall, the solution thus takes `O(n log n)` time with constant space complexity, as we do not use any additional data structures that are dependent on the size of the input.

Example Walkthrough

Let's consider an example to understand the solution approach. Imagine we have an array `nums` with the size of `n + 1`, and it looks like this: `[1, 3, 4, 2, 2]`. The `n` in this case is `4` since the range of numbers is from `1` to `4`.

Here's how we would walk through the problem step by step:

1. **Define the Helper Function `f(x)`:**

   - We need to implement a function `f(x)` that returns `True` if the count of numbers in `nums` that are less than or equal to `x` is greater than `x`. For example, `f(3)` would count how many numbers in `nums` are `<= 3`. In our example, `f(3)` would return `True` because there are 4 numbers that fit the condition (1, 2, 2, and 3), which is greater than 3.
2. **Binary Search with `bisect_left`:**

   - Now we initiate a binary search on the range of numbers from `1` to `n` (`1` to `4` in our example).
   - The `bisect_left` function will effectively split this range and use our function `f` to decide whether to look in the lower half or the upper half.
   - In the first iteration, the middle value `mid` between `1` and `4` is `2`. We calculate `f(2)`, which is `False` since there are only 2 numbers in `nums` that are less than or equal to `2`, which is not greater than `2`. This tells us that the duplicate must be larger than `2`.
   - In the next iteration, the middle value between `3` and `4` is `3`. We calculate `f(3)` and, as stated earlier, it returns `True`. This tells us to search in the lower half, but since only `3` and `4` is left, we have narrowed down `3` as the potential duplicate.
3. **Finding the Duplicate Number:**

   - Once we have the bounds narrowed down to a single number where `f(x)` transitions from `False` to `True`, or vice versa, we know we have found the duplicate number. In our example, when `f(3)` returns `True` and `f(2)` returned `False`, we know that `3` is the value where the transition happens, thus `3` is the duplicate number.

In conclusion, even though the number array is not sorted, we used the properties of binary search on the range `1` to `n` to efficiently find the duplicate number. Since `f(x)` only involves counting elements, and we only needed a range to apply `bisect_left`, we maintained constant space usage as per the problem's constraints.

Python Solution

```python
from typing import List
from bisect import bisect_left

class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        # Define a helper function that will check if the count of numbers less than
        # or equal to x is greater than x itself.
        def is_duplicate_above_x(x: int) -> bool:
            # Count the numbers less than or equal to x
            count = sum(num <= x for num in nums)
            # If the count is greater than x, we might have a duplicate above x
            return count > x

        # Use binary search (implemented as bisect_left) to find the duplicate.
        # The search range is from 1 to len(nums) - 1 as len(nums) could be the maximum number possible
        # since there is exactly one duplicate.
        duplicate_number = bisect_left(range(1, len(nums)), True, key=is_duplicate_above_x)

        return duplicate_number

```

Java Solution

```java
class Solution {
    public int findDuplicate(int[] nums) {
        // Initializing the low and high pointers for binary search.
        int low = 0;
        int high = nums.length - 1;

        // Binary search to find the duplicate number.
        while (low < high) {
            // Calculating the middle index.
            int middle = (low + high) / 2; // same as (low + high) >> 1 but clearer to understand
            int count = 0; // Counter for the number of elements less than or equal to middle.

            // Iterate over the array and count elements less than or equal to middle.
            for (int value : nums) {
                if (value <= middle) {
                    count++;
                }
            }

            // Determine if the duplicate is in the lower half or upper half.
            // If the count is greater than middle, the duplicate is in the lower half.
            if (count > middle) {
                high = middle; // Narrow the search to the lower half.
            } else {
                low = middle + 1; // Narrow the search to the upper half.
            }
        }

        // When low == high, we have found the duplicate number.
        return low;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        // Initialize the search range
        int left = 0;
        int right = nums.size() - 1;

        // Use binary search to find the duplicate
        while (left < right) {
            // Find the midpoint of the current search range
            int mid = left + (right - left) / 2;

            // Count how many numbers are less than or equal to 'mid'
            int count = 0;
            for (int num : nums) {
                if (num <= mid) {
                    count++;
                }
            }

            // If the count is more than 'mid', then the duplicate is in the left half
            if (count > mid) {
                right = mid; // Search in the left half
            } else {
                left = mid + 1; // Search in the right half
            }
        }

        // 'left' is the duplicate number
        return left;
    }
};

```

Typescript Solution

```typescript
function findDuplicate(nums: number[]): number {
    // Define the search range start and end, initially set to 1 and the number of elements - 1
    let left = 1;
    let right = nums.length - 1;

    while (left < right) {
        // Calculate the midpoint of the current search range
        const mid = Math.floor((left + right) / 2);
        let count = 0;

        // Count how many numbers in the array are less than or equal to the midpoint
        for (const value of nums) {
            if (value <= mid) {
                count++;
            }
        }

        // If the count is greater than the midpoint, this indicates that the duplicate
        // is within the range [left, mid], so we focus the search there.
        // Otherwise, the duplicate is in the range [mid + 1, right].
        if (count > mid) {
            right = mid; // Narrow the search to the left half
        } else {
            left = mid + 1; // Narrow the search to the right half
        }
    }

    // Once left meets right, we've found the duplicate number
    return left;
}

```

Time and Space Complexity

The time complexity of the given code snippet is `O(n * log n)`. This is because the `bisect_left` function performs binary search, which has a time complexity of `O(log n)`, and it calls the `f` function on each step of the binary search. The `f` function has a time complexity of `O(n)` since it iterates over all `n` elements in the `nums` list to calculate the sum of all elements less than or equal to `x`. Since the binary search is performed in the range of `len(nums)`, which is `n`, the `f` function is called `O(log n)` times, resulting in an overall time complexity of `O(n * log n)`.

The space complexity of the code is `O(1)`. The code uses a constant amount of additional space: the `f` function and the binary search do not use any extra space that grows with the input size. Therefore, regardless of the size of the input list `nums`, the additional space required by the algorithm remains constant.