747. Largest Number At Least Twice of Others

Easy Array Sorting

Problem Description

In this problem, you are given an array of integers, `nums`, with the assurance that the largest integer is unique, which means it does not appear more than once in the array. The task is to check whether this largest number is at least twice as large as every other number present in the array. If such a condition is met, your function should return the index of this largest number. However, if there's no number in the array that is twice as large as every other number, or the largest number itself is not twice as big as at least one other number in the array, then the function should return `-1`.

Intuition

The intuition for solving this problem stems from the requirement that we must find the largest number and then compare it to all others. We can do this efficiently by tracking two numbers as we iterate through the array: the largest number so far, and the second-largest number so far. We don't actually need to check the largest number against all other numbers if we maintain the second-largest; it's enough to check if it's at least twice the second-largest.

The solution works as follows:

- Initialize two variables, `mx` and `mid`, to track the largest and second-largest values, respectively, and another variable, `ans`, to store the index of the largest number. Start all of them at `0`, with `ans` initialized to `-1`.
- Iterate through the array with their index and value (`i, v`).
- For each number `v`, check if it is greater than the current largest number `mx`. If it is, update `mid` to `mx` (since the largest will now become the second-largest), update `mx` to `v`, and store the current index `i` in `ans`.
- If `v` is not greater than `mx` but is greater than the second-largest number `mid`, then update `mid` to `v`.
- After iterating through the entire array, we need to check if our found largest number `mx` is at least twice as big as the second-largest `mid`. If it is, we return the stored index `ans`; otherwise, we return `-1`.

This approach ensures that we only need a single pass through the array, achieving the goal using O(n) time complexity without any extra space complexity.

Solution Approach

The solution approach employs a linear scan algorithm, which is a simple yet powerful technique used in problems that require comparison or finding elements with certain properties in an array or list. Specifically, the algorithm keeps track of the maximum (`mx`) and the second maximum (`mid`) values while scanning through the array once.

Let's take a more detailed look at the data structures and patterns used in the implementation:

- **Variables for Tracking**: We use two variables `mx` and `mid` to keep track of the largest and the second-largest values as we iterate over the array. An additional variable `ans` is used to note the index of the largest number.
- **Iteration**: We use a `for` loop to iterate through the array elements enumerated with their indices using the built-in `enumerate` function in Python. The `enumerate` function provides a tuple for each element in the list, containing the index (`i`) and the value (`v`).
- **Conditional Logic**: Inside the loop, we use conditional statements (`if`/`elif`) to compare the current element `v` with our tracked maximum (`mx`) and second maximum (`mid`). This helps us in updating these variables without having to compare `mx` with every other number in the array.
- **Updating Values**: If we find a new maximum value, we update `mid` to the old `mx` before updating `mx` to the new maximum. If the current value is not larger than `mx` but is larger than `mid`, we simply update `mid`.
- **Final Check and Return**: After completing the iteration, the final check outside the loop determines whether `mx` is at least twice as large as `mid`. If the condition is satisfied, the index stored in `ans` is returned, and if not, `-1` is returned as specified by the problem statement.

Here is how the implementation translates the solution approach:

1. **Initialization**: We set `mx = mid = 0` and `ans = -1`. These are our starting conditions.
2. **Iteration over `nums`**: Using `enumerate(nums)`, we loop over each `i, v` pair in the array.
3. **New Maximum Condition**: If `v > mx`, we first set `mid = mx`, then `mx = v`, followed by `ans = i`.
   - `mid = mx` ensures we remember the previous largest value.
   - `mx = v` sets the new largest value.
   - `ans = i` captures the index of the new largest value.
4. **New Second Maximum Condition**: If `v` is not the new maximum but is greater than the current `mid`, we set `mid = v`. It's important because we only care about the second-largest value for the comparison with the largest value.
5. **Return Condition**: Perform the final comparison using `return ans if mx >= 2 * mid else -1`. If `mx` is twice as large or larger than `mid`, we return `ans` since `mx` meets the criteria; otherwise, we return `-1`.

The simplicity of this algorithm lies in avoiding unnecessary operations and comparisons, making efficient use of a single-pass scan to acquire our solution.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following array of integers:

```
nums = [3, 6, 1, 0]
```

We want to find out if the largest number in `nums` is twice as large as all the other numbers, and if so, return its index.

Following the linear scan algorithm described in the solution approach:

1. **Initialization**: We start with `mx = mid = 0` and `ans = -1`. This means currently, our largest and second-largest numbers are both set to the first element of the array, and we have no valid answer yet.

2. **Iteration over `nums`**:

   - First iteration (i=0, v=3): Since this is the first element, `mx` and `mid` are already set to 3, and `ans` is set to 0.
   - Second iteration (i=1, v=6): Now `v` is greater than `mx`, we update `mid` to the current `mx` (3), update `mx` to 6, and set `ans` to 1.
   - Third iteration (i=2, v=1): Here, `v` is neither greater than `mx` (6) nor greater than `mid` (3), so no changes are made.
   - Fourth iteration (i=3, v=0): Similarly, `v` is not greater than `mx` (6) or `mid` (3), so no changes are made here as well.

3. **New Maximum and Second Maximum Conditions**: Have been applied during the iteration. After the loop, our current maximum `mx` is 6, our second maximum `mid` is 3, and `ans` indicating the largest number's index is 1.

4. **Return Condition**: We finally check if `mx` (6) is at least twice as large as `mid` (3). Since it is not (6 is not ≥ 2*3), we return `-1`.

We conclude the iteration, and since the largest number (6) is not at least twice as large as the second-largest number in the array, the function would return `-1`. Our implementation correctly follows the solution approach and therefore, validates its effectiveness.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def dominantIndex(self, nums: List[int]) -> int:
        # Initialize maximum and second maximum values and the index of the maximum value
        max_value = second_max = -1
        max_index = -1

        # Iterate through the numbers with their indices
        for index, value in enumerate(nums):
            # If the current value is greater than the maximum value found so far
            if value > max_value:
                # Assign the old max value to second max before updating max_value
                second_max, max_value = max_value, value
                # Record the index of the new maximum value
                max_index = index
            # Else if the value is not greater than max_value but is greater than second_max
            elif value > second_max:
                # Update the second maximum value
                second_max = value

        # Check if the maximum value is at least twice as much as the second maximum
        # If so, return the index of the maximum value. Otherwise, return -1.
        return max_index if max_value >= 2 * second_max else -1
```

Java

```java
class Solution {
    public int dominantIndex(int[] nums) {
        // Initialize two variables to store the largest and second largest numbers
        int max = Integer.MIN_VALUE;
        int secondMax = Integer.MIN_VALUE;

        // The index of the largest number will be stored in this variable
        int indexOfMax = -1;

        // Iterate through the array to find the largest and second largest numbers
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] > max) {
                // If the current number is greater than the largest found so far,
                // update secondMax to max, and max to the current number
                secondMax = max;
                max = nums[i];

                // Update the index of the largest number
                indexOfMax = i;
            } else if (nums[i] > secondMax) {
                // If the current number is only greater than secondMax,
                // update the secondMax to the current number
                secondMax = nums[i];
            }
        }

        // Check if the largest number is at least twice as much as the second largest number
        // If so, return the index of the largest number, otherwise return -1
        return max >= secondMax * 2 ? indexOfMax : -1;
    }
}
```

C++

```cpp
#include <vector> // Required for using the vector container.

class Solution {
public:
    // Function to find whether there exists a dominant index.
    // The dominant index is an index where the element is at least twice as
    // large as every other element in the array. If such an element exists, return its index.
    // Otherwise, return -1.
    int dominantIndex(vector<int>& nums) {
        int maxElement = 0; // Holds the value of the largest element.
        int secondMaxElement = 0; // Holds the value of the second largest element.
        int dominantIndex = 0;  // Initialize the dominant index to 0.

        // Iterate over the array to find the largest and second largest elements.
        for (int i = 0; i < nums.size(); ++i) {
            // If current element is greater than the largest element found so far
            if (nums[i] > maxElement) {
                secondMaxElement = maxElement; // Update the second max to be the previous max
                maxElement = nums[i]; // Update the max to be the current element
                dominantIndex = i; // Update the index of the largest element
            }
            // If current element is not greater than max but is greater than the second max
            else if (nums[i] > secondMaxElement) {
                secondMaxElement = nums[i]; // Update the second max to be the current element
            }
        }

        // If the largest element is at least twice as big as the second largest element,
        // return the index of the largest element, otherwise return -1.
        return maxElement >= secondMaxElement * 2 ? dominantIndex : -1;
    }
};
```

TypeScript

```typescript
/**
 * This TypeScript function finds the index of the dominant element in the array.
 * An element is dominant if it is greater than twice all other elements.
 * @param {number[]} nums - An array of numbers.
 * @returns {number} - The index of the dominant element or -1 if no such element exists.
 */
const dominantIndex = (nums: number[]): number => {
    let largest: number = 0;
    let secondLargest: number = 0;
    let dominantIndex: number = 0;

    // Loop through all elements in the nums array
    for (let i = 0; i < nums.length; ++i) {
        if (nums[i] > largest) { // If current element is larger than the largest element found so far
            secondLargest = largest; // Update the second largest element
            largest = nums[i]; // Update the largest element
            dominantIndex = i; // Update the index of the dominant element
        } else if (nums[i] > secondLargest) { // If current element is larger than the second largest element
            secondLargest = nums[i]; // Update the second largest element
        }
    }

    // If the largest element is at least twice as large as the second largest element, return its index.
    // Otherwise, return -1 indicating there is no dominant element.
    return largest >= secondLargest * 2 ? dominantIndex : -1;
};

// Example usage of the function
const testArray: number[] = [3, 6, 1, 0];
const index: number = dominantIndex(testArray);
console.log('The dominant index is:', index); // Outputs the index or -1 if no dominant element is found
```


```python
from typing import List

class Solution:
    def dominantIndex(self, nums: List[int]) -> int:
        # Initialize maximum and second maximum values and the index of the maximum value
        max_value = second_max = -1
        max_index = -1

        # Iterate through the numbers with their indices
        for index, value in enumerate(nums):
            # If the current value is greater than the maximum value found so far
            if value > max_value:
                # Assign the old max value to second max before updating max_value
                second_max, max_value = max_value, value
                # Record the index of the new maximum value
                max_index = index
            # Else if the value is not greater than max_value but is greater than second_max
            elif value > second_max:
                # Update the second maximum value
                second_max = value

        # Check if the maximum value is at least twice as much as the second maximum
        # If so, return the index of the maximum value. Otherwise, return -1.
        return max_index if max_value >= 2 * second_max else -1
```


Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the input list `nums`. This is because the code consists of a single for loop that iterates through all the elements of the list exactly once to find the largest and the second largest elements.

During each iteration, the code performs a constant-time operation to compare the current value `v` to the current maximum `mx` and the second maximum `mid`. Assignments and comparisons are basic operations that take constant time. Thus, the for loop constitutes the major time-consuming part of the algorithm.

Space Complexity

The space complexity of the code is `O(1)`, which means it uses constant additional space. The only extra variables used in the function are `mx`, `mid`, and `ans`, and these do not depend on the size of the input list. All other operations are done in-place and do not require allocation of additional storage that scales with the input size.