1287. Element Appearing More Than 25% In Sorted Array

Easy Array

Leetcode Link

Problem Description

The problem provides us with an integer array that is sorted in non-decreasing order. It states that there is exactly one integer in the array whose occurrence exceeds 25% of the size of the array. Our task is to identify and return this integer. To put it simply, we need to find the integer that appears more than a quarter of the time in the list of numbers.

Intuition

The first thing to notice is that since the array is sorted in non-decreasing order, all instances of any given number will be consecutive elements in the array. To check if a number occurs more than 25% of the time, you don't need to count every occurrence of each number. Instead, you can look at specific points in the array that would indicate where the 25% threshold would fall.

For example, if an element occurs more than 25% in an array of size `n`, then this element must be the same as the element that is exactly a quarter of the length of the array (`n / 4`) away from it. In other words, the element at index `i` will be the same as the element at index `i + n/4` for the integer that occurs more than 25% of the time.

This is because the sorted nature of the array ensures that a recurring element would occupy a contiguous block. If the block is big enough to cover more than 25% of the array, then the start and the point at a quarter length away must be part of this block and hence, contain the same value.

The solution takes advantage of this fact. It iterates through the array and checks if the current element is the same as the element `n/4` indices away. The bitwise operation `(n >> 2)` is a more efficient way of calculating `n/4`, by shifting the binary representation of `n` to the right by 2 bits.

If it finds such an element, it returns it immediately, since the problem statement guarantees that there is exactly one such integer. If, for some reason, the loop ends without finding an integer (which should not happen given the problem's constraints), it returns 0.

Solution Approach

The implementation of the solution is straightforward and takes advantage of the fact that the list is already sorted. Here is a step-by-step walk-through of the solution approach:

1. Determine the length of the array and store it in variable `n`.
2. Loop through each element of the array with a `for` loop. With `enumerate()`, you have both the index `i` and the value `val` at that index.
3. Within the loop, check if the current element `val` is equal to the element a quarter length away from it in the array (`arr[i + (n >> 2)]`). The expression `(n >> 2)` efficiently calculates `n/4` by using bitwise right shift operator, which essentially divides `n` by 2 twice.
4. If such an element that meets the condition is found, return it immediately. This is the element that occurs more than 25% of the time in the array because the elements in that range of the array are the same due to the array's sorted nature.
5. If no such element is found by the end of the loop (which theoretically should not happen given the problem guarantees an element exists), the function defaults to returning 0.

No additional data structure or complex pattern is used here. The solution capitalizes on the sequence's sort order to minimize the number of checks needed to find the required element. The correct identification of the element is predicated on the guarantee provided by the problem statement, which is that there must be an integer that satisfies the condition, thus making additional checks unnecessary.

Example Walkthrough

Let's consider an example with the following sorted integer array to illustrate the solution approach:

```
arr = [1, 2, 2, 2, 3]
```

The size of this array `n` is 5. According to the problem, we are looking for an integer that occurs more than 25% of the time. Since 25% of the size of the array is `5 * 0.25 = 1.25`, our threshold is more than 1.25 occurrences. Therefore, we are seeking an integer that appears at least 2 times since we cannot have a fraction of an occurrence.

Following the solution approach:

1. We determine that `n = 5`.
2. We begin to loop through each element using a `for` loop. We'll be checking if the element at the current index `i` is the same as the element at index `i + (n >> 2)`. Remember `(n >> 2)` effectively calculates `int(5 / 4) = 1`, so we'll be comparing elements one index apart.
3. As we iterate:
   - For `i = 0`: `arr[0]` is 1, and `arr[0 + 1]` is 2. No match, we move on.
   - For `i = 1`: `arr[1]` is 2, and `arr[1 + 1]` is also 2. We have a match.
4. Since `arr[1]` is equal to `arr[1 + 1]`, we found the element that occurs more than 25% of the time in the array (2 appears more than once).
5. We return the value `2` immediately without needing to check the rest of the array.

There's no need to continue iterating because by the problem's constraint, there's exactly one integer that must meet the condition, and we've already found it. If the array had been larger, our loop would continue checking this condition until the end, but this step is unnecessary in our example.

Python Solution

```python
from typing import List

class Solution:
    def findSpecialInteger(self, arr: List[int]) -> int:
        # Length of the array
        length = len(arr)

        # Iterate over the array's elements
        for index, value in enumerate(arr):
            # Compare the current element with the element at a quarter length away
            # Using bitwise right shift operator to divide length by 4
            if value == arr[index + (length >> 2)]:
                # Since both are equal and are at least 25% apart, return the value
                return value

        # In case no special integer is found, return 0 (though the problem statement implies there should be one)
        return 0
```

Java Solution

```java
class Solution {
    public int findSpecialInteger(int[] arr) {
        // The length of the input array
        int length = arr.length;

        // Iterate through the array with the intent to find the special integer.
        // The special integer is the one that appears more than 25% of the time.
        for (int index = 0; index < length; ++index) {
            // Check if the current element is the same as the element which is away
            // a quarter of the array's length. If it is, we've found the special integer.
            // (n >> 2) is equivalent to dividing n by 4 (or a quarter of its length).
            if (arr[index] == arr[index + (length >> 2)]) {
                // Return the special integer if found.
                return arr[index];
            }
        }

        // If no such element is found, return 0. This line should never be reached if the
        // input always contains such a special integer, as per the problem's statement.
        return 0;
    }
}
```

C++ Solution

```cpp
#include <vector> // Include the required header for using vectors

// The Solution class encapsulates the problem's solution.
class Solution {
public:
    // Method to find the element that appears more than 25% of the time in a sorted array.
    int findSpecialInteger(std::vector<int>& arr) {
        int n = arr.size(); // Obtain the size of the array
        // Iterate over each element in the array. We use the bitwise operator to divide 'n' by 4 (n >> 2).
        // This is the same as n/4 but is potentially faster for powers of 2.
        for (int i = 0; i <= n - (n >> 2); ++i) { // The loop now stops at `n - (n >> 2)` to avoid accessing out-of-range elements.
            // Check if the current element is the same as the element 25% ahead in the array.
            // If it is, we return the value since it appears more than 25% of the time.
            if (arr[i] == arr[i + (n >> 2)]) {
                return arr[i];
            }
        }
        // If no element appears more than 25% of the time, which theoretically should not happen in a valid input, return 0.
        // Though given the problem constraints, this line should be unreachable.
        return 0;
    }
};
```

Typescript Solution

```typescript
/**
 * This function takes an array of numbers and returns the element that appears more than 25% of the time in the array.
 * It uses a sliding window, leveraging the fact that if an element appears more than 25% of the time,
 * then it must appear at the current index and again at a quarter of the array's length away.
 * @param arr - The array of numbers to be checked.
 * @return The special element occurring more than 25% of the time.
 */
function findSpecialInteger(arr: number[]): number {
    // Get the length of the array
    const lengthOfArray: number = arr.length;

    // Iterate through the array
    for (let i = 0; i < lengthOfArray; ++i) {
        // Check if the current element is the same as the element a quarter array length away.
        if (arr[i] === arr[i + (lengthOfArray >> 2)]) {
            // Return the element if it is found to repeat after a quarter array length.
            return arr[i];
        }
    }

    // If no such element is found, return 0 (which can be seen as an error or a "not found" indication)
    return 0;
}
```

Time and Space Complexity

The given code snippet is designed to find an element in an array `arr` that appears more than `25%` of the time. To determine the time and space complexity, let's analyze the given code:

Time Complexity: The code involves a single loop where `i` traverses from `0` to `n-1`, where `n` is the length of the array. The key operation to check is `arr[i + (n >> 2)]`. Shifting `n` by `2` to the right is equivalent to dividing `n` by `4` (`n >> 2` is `n/4`). This operation finds the element at a quarter distance ahead of the current index `i`. Since it's done within a single loop, with no nested loops or recursive calls, the time complexity is linear with respect to the size of the input array. Hence, the time complexity is `O(n)`.

Space Complexity: The algorithm only uses a constant amount of extra space for variables such as `i`, `val`, and `n`, with no additional data structures dependent on the input size. Therefore, the space complexity is `O(1)`, which represents constant space complexity.