540. Single Element in a Sorted Array

Medium Array Binary Search Leetcode Link

Problem Description

In this LeetCode problem, we have a sorted array where each number normally appears twice, except for one particular number that appears only once. The challenge is to identify the single number that doesn't have a pair. Moreover, the solution must be efficient, running in logarithmic time complexity, which suggests that we should use an algorithm like binary search, and the space complexity must be constant, not dependent on the size of the input array.

Intuition

To solve this problem, we leverage the sorted nature of the array and the requirements for time and space complexity to determine that binary search is the right approach. Binary search helps us cut down the problem space in half with each iteration, making our time complexity O(log n).

The key insight here is to understand how pairs of the same numbers are arranged in the array. Since the array is sorted, identical numbers are adjacent. If we take the first occurrence of a pair, it should be at an even index, and its pair should be immediately next, at an odd index, and this pattern continues until we hit the single element. After the single element, this pattern will flip because of the missing pair.

By examining the middle index and its adjacent numbers, we can decide whether the single number lies on the left or right of our current middle point. Specifically, we use XOR operator to quickly check if a number at an even index does not have its pair at the next odd index, or vice versa for an odd index. The XOR operation is `mid ^ 1`, which essentially gives us the index that should hold the identical number to the one at `mid`. If the condition is true, it means the single number is at `mid` or left of it, so we shift our search window to the left. Otherwise, the single non-duplicate must be to the right, so we adjust our window accordingly.

We repeat this until we narrow down to one element, `left` index ends up pointing to the single element, as `right` converges towards it. Hence, the element at the `left` index is the non-duplicated number and our answer.

Solution Approach

The provided solution approach uses binary search, which is an efficient algorithm for finding an item in a sorted array, using a divide and conquer strategy. The basic idea of binary search is to repeatedly divide the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Here's a step-by-step breakdown of the implementation:

1. Initialize two pointers, `left` and `right`, to the start and end of the array, respectively. These pointers will define the bounds of our search space.
2. Use a while loop to iterate as long as `left` is less than `right`. This condition ensures that the search continues until the search space is reduced to one element.
3. Calculate the `mid` index, which is central to binary search's divide-and-conquer approach, using `(left + right) >> 1`. The rightward shift operator `>>` effectively divides the sum by two but is faster than standard division.
4. Determine if the element at `mid` is part of a pair that starts with an even index (`mid` being even and `nums[mid] == nums[mid ^ 1]`) or an odd index (`mid` being odd and `nums[mid] == nums[mid ^ 1]`). The bitwise XOR operator `^` is used here to compare the element with its adjacent pair member based on the parity of `mid`.
   - If `nums[mid] == nums[mid ^ 1]`, the single element must be to the right of `mid`, thus `left` is set to `mid + 1`.
   - Conversely, if `nums[mid] != nums[mid ^ 1]`, the single element must be to the left of `mid` (including `mid` itself), so `right` is set to `mid`.
5. This process halves the search interval with each iteration, quickly homing in on the single non-duplicate number.
6. The loop terminates when `left` equals `right`, signifying that the element at the `left` index is the non-duplicated number.
7. Finally, return the value at the `left` index.

By following this approach, we harness binary search's efficiency in concert with the array's particular properties (sorted with pairs of numbers) to achieve the desired logarithmic time complexity and constant space complexity.

Example Walkthrough

Let's go through an example to illustrate the solution approach described above using the array `nums = [1, 1, 2, 3, 3]`. We need to find the single number that does not have a pair.

1. Initialize `left = 0` and `right = 4` (the start and end indices of the array).
2. Start the while loop. Since `left < right` (0 < 4), we proceed with binary search.
3. Calculate `mid`. `(left + right) >> 1` is `(0 + 4) >> 1`, which equals 2. So, `mid = 2`, and `nums[mid] = 2`.
4. We need to check the parity of `mid` and use the XOR operator to determine which side to search next.
   - Since `mid` is even, we compare `nums[mid]` and `nums[mid ^ 1]`. We compute `mid ^ 1` as `2 ^ 1`, which equals 3.
   - `nums[mid]` is 2 and `nums[mid ^ 1]` is 3. They're not equal (`2 != 3`), indicating that the single number must be to the left of `mid`, including `mid` itself.
5. Set `right` to `mid` (now `right = 2`).
6. The while loop continues. Now, `left` is 0 and `right` is 2. Check if `left < right`—yes, 0 < 2.
7. Calculate the new `mid`. `(left + right) >> 1` is `(0 + 2) >> 1`, which equals 1. So now, `mid = 1` and `nums[mid] = 1`.
8. Check the adjacency condition again.
   - `mid` is odd, so we check if `nums[mid] == nums[mid ^ 1]`. We compute `mid ^ 1` as `1 ^ 1`, which equals 0.
   - `nums[mid]` is 1 and `nums[mid ^ 1]` is also 1 (`nums[0]`), they are equal (`1 == 1`). So this means the single number should be to the right of `mid`.
9. Set `left` to `mid + 1` (now `left = 2`).
10. Repeat the loop. `left` is now 2, and `right` is also 2, which means `left` is not less than `right`, so the loop ends.
11. The single number is at the `left` index, which is 2. `nums[left]` is 2, which is the single number we were looking for.

By following these steps, we used binary search efficiently to find the single non-duplicate number in a sorted array.

Python Solution

```python
from typing import List

class Solution:
    def single_non_duplicate(self, nums: List[int]) -> int:
        # Initialize the search space
        left, right = 0, len(nums) - 1

        # Perform binary search to find the non-duplicate integer
        while left < right:
            # Calculate the middle index of the current search space
            mid = (left + right) // 2

            # Check if the middle element's value is equal to the next
            # or the previous based on whether mid is odd or even
            # Using XOR operation. If mid is even, mid ^ 1 will be mid + 1,
            # and if mid is odd, mid ^ 1 will be mid - 1.
            if nums[mid] != nums[mid ^ 1]:
                # If they are not equal, move the right pointer to mid.
                # We do this because the single element must be in the
                # first half if the pair is not complete in the mid.
                right = mid
            else:
                # If they are equal, this means the single element is in the
                # second half of the array. Move the left pointer to mid + 1.
                left = mid + 1

        # When left == right, the search space has been narrowed down to one element.
        # This remaining element is the non-duplicate integer we're looking for.
        return nums[left]

# Example usage:
# sol = Solution()
# result = sol.single_non_duplicate([1, 1, 2, 3, 3, 4, 4, 8, 8])
# print(result)  # Output should be 2

```

Java Solution

```java
class Solution {
    public int singleNonDuplicate(int[] nums) {
        // Initialize the left and right pointers
        int left = 0;
        int right = nums.length - 1;

        // Continue searching while the left pointer is less than the right pointer
        while (left < right) {
            // Calculate the middle index
            int mid = left + (right - left) / 2;

            // The XOR operation here is a clever trick. Since we're looking for
            // the single non-duplicate number, pairs will be adjacent.
            // For even mid index, mid ^ 1 will be the next index, which should be identical in the case of pairs.
            // For odd mid index, mid ^ 1 will be the previous index, which, again, should be identical in case of pairs.
            // If they're not identical, then the single element must be to the left, so adjust the right pointer.
            if (nums[mid] != nums[mid ^ 1]) {
                right = mid;
            } else {
                // If they are identical, the single element must be to the right, so adjust the left pointer.
                left = mid + 1;
            }
        }

        // When left == right, we have found the single non-duplicate element.
        return nums[left];
    }
}

```

C++ Solution

```cpp
#include <vector> // Include necessary header for the vector container

// Solution class to encapsulate the method that finds the single non-duplicate number
class Solution {
public:
    // The singleNonDuplicate method takes a vector of integers and returns the single non-duplicate number.
    int singleNonDuplicate(std::vector<int>& nums) {
        // Initialize the left and right pointers for binary search
        int left = 0;
        int right = nums.size() - 1;

        // Execute binary search while the left pointer is less than the right pointer
        while (left < right) {
            // Calculate the middle index
            int mid = left + (right - left) / 2; // Avoid potential overflow by using left + (right - left) / 2 instead of (left + ri

            // Check for the single element
            // XORing the index 'mid' with 1 will give us the pair index for even 'mid'(mid ^ 1 = mid + 1) and
            // the previous index for odd 'mid'(mid ^ 1 = mid - 1)
            if (nums[mid] != nums[mid ^ 1]) {
                // If nums[mid] is not the same as its adjacent (pair), we found our single element or it is to the left.
                // Hence, we move the 'right' pointer to 'mid'
                right = mid;
            } else {
                // If nums[mid] is the same as its adjacent, the single element must be to the right of 'mid'
                // So, move the 'left' pointer to 'mid + 1'
                left = mid + 1;
            }
        }

        // At the end of the loop, 'left' will have converged to the single non-duplicate element
        return nums[left];
    }
};

```

Typescript Solution

```typescript
// Function to find the single non-duplicate number in a sorted array.
// All numbers except one appears exactly twice, the non-duplicate number appears only once.
function singleNonDuplicate(nums: number[]): number {
    // Define pointers for the binary search
    let leftPointer = 0;
    let rightPointer = nums.length - 1;

    // Start binary search
    while (leftPointer < rightPointer) {
        // Calculate the middle index using bit manipulation
        // (right shift by 1 is equivalent to dividing by 2)
        const middleIndex = (leftPointer + rightPointer) >> 1;

        // Check if the middle element is not equal to its neighbor.
        // XOR with 1 will check the neighbor, for even mid it will check next, for odd mid it will check previous.
        if (nums[middleIndex] != nums[middleIndex ^ 1]) {
            // If it's not equal, the single element must be on the left side.
            // Move the right pointer to the middle index.
            rightPointer = middleIndex;
        } else {
            // Otherwise, the single element is on the right side.
            // Move the left pointer to one past the middle index.
            leftPointer = middleIndex + 1;
        }
    }
    // At the end of the loop, leftPointer will point to the single element.
    // Return the element at the leftPointer index.
    return nums[leftPointer];
}

// Example usage:
// const result = singleNonDuplicate([1,1,2,3,3,4,4,8,8]);
// console.log(result); // Outputs: 2

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(log n)`. This is because the algorithm applies a binary search over the input array `nums`. During each iteration of the while loop, the search space is halved by updating either the `left` or `right` pointers, which results in a logarithmic number of steps relative to the size of the input array.

Space Complexity

The space complexity of the code is `O(1)`. No additional data structures that scale with input size are used within the method. The variables `left`, `right`, and `mid` occupy constant space, so the space usage does not depend on the input size.