1060. Missing Element in Sorted Array

Medium Array Binary Search Leetcode Link

Problem Description

Given a sorted integer array `nums` with unique elements, our task is to find the *k*-th missing number from the array. A missing number in the array is defined as a number that is not included in the array but falls within the range of the array's first and last elements. For example, if our array is `[2, 3, 5, 9]`, and we are looking for the 1st missing number, the answer would be `4`, since it is the first number that does not exist in the array but falls within the range starting from the leftmost number `2`. If we were asked for the 2nd missing number, the answer would be `6`, and so on.

Intuition

The intuition behind the solution comes from understanding how to calculate the missing numbers in a particular segment of the sorted array. In a sorted array, where all elements are unique, the difference between the value of the element at a given position and its index (adjusting for the starting point of the series) tells how many numbers are missing up to that position.

To solve this problem, we can define a helper function `missing(i)` that returns the total count of missing numbers up to index `i`. Since the array is sorted, we can use binary search to find the smallest index `l` such that the count of missing numbers up to `l` is less than or equal to `k`.

The solution follows these steps:

1. Calculate the total number of missing numbers within the entire array. If `k` is greater than this number, we can simply add `k` to the last element of the array minus the total missing count.
2. Otherwise, perform a binary search between the start `l` = 0 and end `r` = `n` - 1 of the array to find the lowest index `l` where the count of missing numbers is less than `k`.
3. Using the binary search, when the count of missing numbers up to `mid` index is equal to or more than `k`, we move the right pointer `r` to `mid`. Otherwise, we move the left pointer `l` to `mid + 1`.
4. Finally, once the binary search is completed, the answer is the number at index `l-1` plus `k` minus the total count of missing numbers up to `l-1`.

In this approach, we achieve a logarithmic time complexity, making the algorithm efficient even for large arrays.

Solution Approach

The solution implements a binary search algorithm due to the array `nums` being sorted in ascending order. This approach is efficient for finding an element or a position within a sorted array in logarithmic time complexity, which is O(log n). The binary search pattern is used to quickly identify the segment of the array where the *k*-th missing number lies.

The solution has the following key components:

- A helper function `missing(i)`: this function calculates the total number of missing numbers up to index `i`. This is done by comparing the value at `nums[i]` with the starting value `nums[0]` and the index `i`. The mathematical expression is `nums[i] - nums[0] - i`.

- The main function `missingElement(nums, k)`: this function uses binary search to find the position where the *k*-th missing number is. The binary search is carried out with the help of two pointers, `l` (left) and `r` (right), which define the search boundaries.

- The while loop `while l < r`: it repeatedly splits the array in half to check the count of missing numbers up to the midpoint. Depending on whether the count of missing numbers is less than or greater than `k`, it adjusts the search space by moving the left or right pointers. The condition `missing(mid) >= k` is used to decide if we should move the right pointer.

- Calculating the result: after finding the right segment where the *k*-th missing number fits, the exact number is obtained using the expression `nums[l - 1] + k - missing(l - 1)`. This accounts for the missing numbers up to the position just before the one found by binary search.

The efficient use of binary search in this sorted array reduces the iteration count from potentially linear (if we were to iterate over each element) to logarithmic, making it suited for large-scale problems.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we have the following sorted array `nums` and we are asked to find the 3rd missing number (`k = 3`):

```
nums = [1, 2, 4, 7, 10]
```

To find the 3rd missing number, we perform the following steps:

1. Call the helper function `missing(i)` to calculate the number of missing numbers before several positions in the array. This works as follows:
   - For `i = 0` (element `1`), `missing(0)` is `0` because there are no missing numbers before the first element.
   - For `i = 2` (element `4`), `missing(2)` is `4 - 1 - 2 = 1` because there is one number (`3`) missing before it.
2. Compare `k` with the total number of missing numbers in the entire array to determine if `k` is within the array's range or beyond. In this case, since `nums` starts with `1` and ends with `10`, there are `10 - 1 - (5 - 1) = 5` missing numbers in total. Since `k = 3` is less than `5`, the 3rd missing number is within the range.

3. Perform a binary search to find the smallest index `l` such that the count of missing numbers up to `l` is less than or equal to `k`. Our binary search starts with `l = 0` and `r = 4` (the last index of `nums`):

   - First iteration: `mid = (0 + 4)/2 = 2`, `missing(mid) = 1 < k`, so update `l = mid + 1 = 3`.
   - Second iteration: `mid = (3 + 4)/2 = 3`, `missing(mid) = 3 (since 3, 5, and 6 are missing before 7) = k`, so update `r = mid = 3`.
   - The loop ends because `l < r` is no longer true.
4. The final index `l` we found will be `3`. The 3rd missing number is not in position `l` or `l-1` but after the number at `l-1`. Therefore, we calculate the 3rd missing number as `nums[l-1] + k - missing(l-1)`. In our case, this is `nums[2] + 3 - 1 = 4 + 3 - 1 = 6`.

Thus, our 3rd missing number is `6`. This process of binary search minimized the number of steps needed to find `k`, which results in significantly faster execution times for large arrays.

Python Solution

```python
from typing import List

class Solution:
    def missingElement(self, nums: List[int], k: int) -> int:
        # Helper function to calculate the number of missing elements
        # before the current index 'i'
        def count_missing_before_index(i: int) -> int:
            # The count is the difference between the current value and the
            # first value minus the number of steps from the beginning
            return nums[i] - nums[0] - i

        # Calculate the length of the input array 'nums'
        num_length = len(nums)

        # If 'k' is greater than the number of missing numbers before the last element
        # then the missing element is beyond the last element of the array
        if k > count_missing_before_index(num_length - 1):
            return nums[num_length - 1] + k - count_missing_before_index(num_length - 1)

        # Initialize left and right pointers for binary search
        left, right = 0, num_length - 1

        # Use binary search to find the smallest index 'left' such that
        # the number of missing numbers is at least 'k'
        while left < right:
            mid = (left + right) // 2
            if count_missing_before_index(mid) >= k:
                right = mid
            else:
                left = mid + 1

        # The actual missing element is the element at index 'left - 1' plus
        # 'k' minus the number of missing numbers before 'left - 1'
        return nums[left - 1] + k - count_missing_before_index(left - 1)

# Example usage:
# solution = Solution()
# result = solution.missingElement([4,7,9,10], 1)
# print(result)  # Outputs 5, which is the first missing number.

```

Java Solution

```java
class Solution {

    // Function to find the k-th missing element in the sorted array nums
    public int missingElement(int[] nums, int k) {
        int len = nums.length;

        // Check if k-th missing number is beyond the last element of the array
        if (k > missingCount(nums, len - 1)) {
            return nums[len - 1] + k - missingCount(nums, len - 1);
        }

        // Binary search initialization
        int left = 0, right = len - 1;

        // Binary search to find the k-th missing element
        while (left < right) {
            int mid = (left + right) / 2; // Find the middle index
            // Check if missing count from start to mid is >= k,
            // if true, k-th missing number is to the left side of mid
            if (missingCount(nums, mid) >= k) {
                right = mid;
            } else {
                // Otherwise, k-th missing number is to the right side of mid
                left = mid + 1;
            }
        }

        // Once binary search is complete, compute and return the k-th missing element
        return nums[left - 1] + k - missingCount(nums, left - 1);
    }

    // Helper function to calculate the number of missing numbers from start till index i
    private int missingCount(int[] nums, int idx) {
        return nums[idx] - nums[0] - idx;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the k-th missing element in a sorted array.
    int missingElement(vector<int>& nums, int k) {
        // Lambda function to calculate the number of missing elements
        // up to the index `i` in the sorted array.
        auto countMissingUpToIndex = [&](int index) {
            return nums[index] - nums[0] - index;
        };

        // Get the size of the input array.
        int size = nums.size();

        // If k is beyond the range of missing numbers in the array,
        // calculate the result directly.
        if (k > countMissingUpToIndex(size - 1)) {
            return nums[size - 1] + k - countMissingUpToIndex(size - 1);
        }

        // Initialize binary search bounds.
        int left = 0, right = size - 1;

        // Perform binary search to find the smallest element
        // such that the number of missing elements up to that
        // element is equal or greater than k.
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (countMissingUpToIndex(mid) >= k) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        // Calculate the k-th missing element using the
        // element at index `left - 1`.
        return nums[left - 1] + k - countMissingUpToIndex(left - 1);
    }
};

```

Typescript Solution

```typescript
// Function to find the k-th missing element in a sorted array.
function missingElement(nums: number[], k: number): number {
  // Lambda function to calculate the number of missing elements
  // up to the index `i` in the sorted array.
  const countMissingUpToIndex = (index: number) => {
    return nums[index] - nums[0] - index;
  };

  // Get the size of the input array.
  const size: number = nums.length;

  // If k is beyond the range of missing numbers in the array,
  // return the k-th element beyond the last element.
  if (k > countMissingUpToIndex(size - 1)) {
    return nums[size - 1] + k - countMissingUpToIndex(size - 1);
  }

  // Initialize binary search bounds.
  let left: number = 0;
  let right: number = size - 1;

  // Perform a binary search to find the smallest element
  // such that the number of missing elements up to that
  // element is equal to or greater than k.
  while (left < right) {
    let mid: number = left + Math.floor((right - left) / 2);
    if (countMissingUpToIndex(mid) >= k) {
      right = mid;
    } else {
      left = mid + 1;
    }
  }

  // Calculate the k-th missing element using the
  // element at index `left - 1`.
  return nums[left - 1] + k - countMissingUpToIndex(left - 1);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed as follows:

- The `missing` function is a simple calculation that performs in constant time, O(1).
- The `missingElement` function runs a binary search algorithm over the array of size `n`. Binary search cuts the search space in half with each iteration, resulting in a logarithmic time complexity, O(log n).
- Since the missing function is called within the binary search loop, and it is called in constant time, it does not affect the overall time complexity of O(log n).

Therefore, the overall time complexity of the code is `O(log n)`.

Space Complexity

The space complexity can be analyzed as follows:

- The given solution uses only a fixed amount of extra space for variables such as `n, l, r, mid`, which does not depend on the input size.
- No additional data structures or recursive calls that use additional stack space are made.

Hence, the overall space complexity is `O(1)`, which implies constant space is used regardless of the input size.