2122. Recover the Original Array

Hard Array Hash Table Enumeration Sorting

Leetcode Link

Problem Description

Alice had an initial array `arr` containing `n` positive integers. To create variability, she decided to generate two new arrays, `lower` and `higher`, each also containing `n` elements. She chose a positive integer `k` and subtracted `k` from each element in `arr` to form `lower`, and added `k` to each element in `arr` to form `higher`. Thus, `lower[i] = arr[i] - k` and `higher[i] = arr[i] + k`.

Unfortunately, Alice lost all three arrays `arr`, `lower`, and `higher`. She now only remembers a single array `nums` that contains all the elements of `lower` and `higher` mingled together without any indication as to which element belongs to which array. The task is to help Alice by reconstructing the original array `arr`.

The constraints are that `nums` consists of `2n` integers with exactly `n` integers from `lower` and `n` from `higher`. We need to return any valid construction of `arr`. It is important to remember that multiple valid arrays might exist, so any of them will be considered correct. Furthermore, the problem guarantees that there is at least one valid array `arr`.

Intuition

Since the original array `arr` is lost, and the elements in `nums` cannot be directly distinguished as belonging to either `lower` or `higher`, we need to analyze the properties of numbers in `lower` and `higher`. Recognizing that each element in `higher` is generated by adding the same fixed value `k` to the respective elements in `lower`, we can sort `nums` to sequentially establish pairs of elements that could potentially correspond to the original and its modification by `k`.

The first step in uncovering `arr` is sorting `nums`. After the sort, knowing that the smallest value must be from the `lower` array and the next smallest that forms a valid pair must be from `higher`, one can attempt to determine the value of `k`. This value should be twice the chosen `k` because it's the difference between counterpart elements from `higher` and `lower`.

For each potential `k` (which is the difference between a pair of elements in `nums`), we check if it's even and not zero. Since `k` has to be a positive integer, this test eliminates invalid candidates.

Once a potential `k` is found, we proceed to form the pairs. A boolean array, `vis`, is used to track elements already used in pairs. If the right elements are not found for pairing, we know the current `k` candidate is invalid, and we move to the next candidate.

The pairing logic relies on finding elements `nums[l]` and `nums[r]` where `nums[r] - nums[l]` equals the difference `d` we're exploring as `2k`. We keep appending the midpoint of these pairs `(nums[l] + nums[r]) / 2` to the `ans` list until either we run out of elements or can no longer find valid pairs, which indicates a complete `arr`.

If a complete `arr` is found where the number of found pairs equals `n`, we return `arr`. If not, we continue testing other differences until a valid array is constructed or all possibilities have been exhausted, in which case an empty list is returned.

Solution Approach

The implementation of the solution involves a few key steps that utilize algorithms and data structures effectively to reconstruct Alice's initial array `arr`.

1. **Sorting `nums`:** The very first operation in the code is sorting the `nums` array. This is a common preprocessing step in many problems, as it often simplifies the problem by ordering the elements. In our case, sorting is essential because it allows us to pair elements in `nums` that could represent an original and its counterpart modified by `k`.

2. **Finding the potential `k`:** The for-loop starts with `i` at index 1, iterating through the sorted `nums`. Each iteration examines whether `nums[i] - nums[0]` is a potential `2k` (the actual `k` multiplied by 2). Only even and non-zero differences are considered valid because we need a positive integer `k`.

3. **Reconstructing `arr`:** A boolean array `vis` is created to keep track of which elements have already been paired. Initially, it marks the element corresponding to the current candidate `k` (`nums[i]` at the top of the loop) as used. The code maintains two pointers, `l` and `r`, which start searching for pairs from the beginning of `nums`. These pointers skip used elements tracked by `vis`.

4. **Pairing elements using two pointers:** The paired elements are chosen by verifying that `nums[r] - nums[l]` equals `d` (the current candidate `2k`). For each such pair, the midpoint `(nums[l] + nums[r]) / 2`, representing an element in `arr`, is calculated and appended to `ans`.

5. **Handling edge cases and completing the array:** The pairing process continues until it's no longer possible to pair elements according to the current `d` value. If a complete `arr` is reconstructed such that the number of pairs equals `n` (which is half the length of `nums`), then `ans` is returned as a valid original array.

6. **Returning the result:** If, during the pairing process, an inconsistency is found, the algorithm breaks out of the inner while loop, meaning the current `d` is not valid, and the for-loop continues with the next candidate. If the algorithm finds a correct sequence of pairs for `ans`, it is returned as a valid candidate. Otherwise, the pairing attempt will fail for all differences `d`, and the function will return an empty list, though this is guaranteed not to happen as per the problem statement.

The above approach takes advantage of sorted arrays, the two-pointer technique, and the boolean visitation array to implement an efficient and effective solution.

Example Walkthrough

Let's walk through the solution approach with a small example. Suppose Alice remembers the following mingled array `nums` that has elements from both `lower` and `higher` arrays: `nums = [3, 9, 5, 7]`.

According to the problem description and solution approach, we want to reconstruct the original array `arr`. Follow these steps:

1. **Sort `nums`:** After sorting, the array `nums` becomes `[3, 5, 7, 9]`. Sorting ensures that the smallest element, which must be from the `lower` array, is at the beginning.

2. **Finding potential `k`:** Starting from the second smallest number in `nums`, which is `5`, we take the difference with the smallest number `3`, resulting in `2`. Since the difference must be `2k` (and `k` is a positive integer), valid potential `k` values are `1`, the only number that when doubled gives `2`.

3. **Reconstructing `arr`:** Initialize `vis = [false, false, false, false]` since no elements have been paired yet. We will look for pairs that have a difference of `2` (our potential `2k`).

4. **Pairing elements using two pointers:**
   - We start with the first element `3` and look for an element that has a difference of `2`. The next element is `5`, and `5 - 3 = 2` which is our `2k`. We mark elements `3` and `5` as visited: `vis = [true, true, false, false]`.
   - Now, we calculate the midpoint, `(3 + 5) / 2 = 4`, which is a candidate for `arr`. Hence, `arr = [4]`.
   - We continue to the next unvisited element which is `7`, and look for its pair. We find `9` (`9 - 7 = 2`). Now `vis = [true, true, true, true]`.
   - The midpoint of `7` and `9` is `(7 + 9) / 2 = 8`, which is added to `arr`. Now `arr = [4, 8]`.

5. **Completing the array:** At this point, all elements in `nums` have been visited and paired correctly. There are two pairs, and this matches `n` (since `nums` is of length `2n` and `arr` of length `n`). The process is complete, and we have successfully reconstructed `arr`.

6. **Returning the result:** The array `arr = [4, 8]` is returned as a valid construction of Alice's original array.

By following these steps, we have used the sorted array, the two-pointer technique, and a boolean array that tracks paired elements to effectively reconstruct the array `arr` that Alice had before the arrays were lost.

Python Solution

```python
from typing import List

class Solution:
    def recoverArray(self, nums: List[int]) -> List[int]:
        # Sort the input array in ascending order.
        nums.sort()
        n = len(nums)

        # Try to find the difference between a pair of original and added numbers.
        for i in range(1, n):
            difference = nums[i] - nums[0]

            # Ignore differences of zero or that are odd, since
            # the original problem constraint requires that the difference is even.
            if difference == 0 or difference % 2 == 1:
                continue

            # Create an array to keep track of visited numbers.
            visited = [False] * n
            visited[i] = True

            # Initialize the array to be returned.
            original_numbers = [(nums[0] + nums[i]) // 2]

            # Initialize pointers for the smaller value (l)
            # and the larger value (r) in a candidate pair.
            l, r = 1, i + 1

            # While there are more elements to process:
            while r < n:
                # Move the l pointer to the next unvisited element.
                while l < n and visited[l]:
                    l += 1

                # Move the r pointer to the next element
                # that makes the difference exactly 'difference'.
                while r < n and nums[r] - nums[l] < difference:
                    r += 1

                # If r reaches the end or the difference is incorrect, break the loop.
                if r == n or nums[r] - nums[l] > difference:
                    break

                # If a valid pair is found, add to visited and original_numbers array.
                visited[r] = True
                original_numbers.append((nums[l] + nums[r]) // 2)

                # Move both pointers to the next potential pair.
                l, r = l + 1, r + 1

            # If the size of the original_numbers is half the size of the input array,
            # then we have found all pairs and can return the result.
            if len(original_numbers) == n // 2:
                return original_numbers

        # If no valid configuration is found, return an empty array.
        return []

# Example use:
# solution = Solution()
# result = solution.recoverArray([1, 3, 4, 2])
# print(result)  # Output: The recovered array of original integers.

```

Java Solution

```java
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

class Solution {
    public int[] recoverArray(int[] nums) {
        // Sort the input array to ensure the order
        Arrays.sort(nums);
        // Use i to scan through the array, starting from index 1
        for (int i = 1, length = nums.length; i < length; ++i) {
            // Calculate the difference between the current element and the first one
            int difference = nums[i] - nums[0];
            // Skip if the difference is zero or odd
            if (difference == 0 || difference % 2 == 1) {
                continue;
            }
            // Create a boolean array to keep track of visited elements
            boolean[] visited = new boolean[length];
            visited[i] = true; // Mark the current element as visited
            // Initialize a list to store the elements of the original array
            List<Integer> tempArray = new ArrayList<>();
            // Add the reconstructed element to the temp list
            tempArray.add((nums[0] + nums[i]) >> 1);
            // Use two pointers to traverse the array and try to reconstruct the original array
            for (int l = 1, r = i + 1; r < length; ++l, ++r) {
                // Advance the left pointer past any already visited elements
                while (l < length && visited[l]) {
                    ++l;
                }
                // Advance the right pointer until the condition is met
                while (r < length && nums[r] - nums[l] < difference) {
                    ++r;
                }
                // Break if the pointer has reached the end or condition is not met
                if (r == length || nums[r] - nums[l] > difference) {
                    break;
                }
                // Mark the right element as visited and add the reconstructed element
                visited[r] = true;
                tempArray.add((nums[l] + nums[r]) >> 1);
            }
            // If we've added the correct number of elements, the array is recovered
            if (tempArray.size() == (length >> 1)) {
                // Convert the list to an array and return it
                int[] ans = new int[tempArray.size()];
                int idx = 0;
                for (int elem : tempArray) {
                    ans[idx++] = elem;
                }
                return ans;
            }
        }
        // If no array was recovered, return null.
        return null;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    vector<int> recoverArray(vector<int>& nums) {
        // Sort the input array.
        sort(nums.begin(), nums.end());

        // Iterate through the array, trying to find the correct difference 'd'.
        for (int i = 1, n = nums.size(); i < n; ++i) {
            int diff = nums[i] - nums[0];

            // Skip if the difference is zero or odd, since the problem assumes an even difference.
            if (diff == 0 || diff % 2 == 1) continue;

            // Keep a visited array to mark the elements that have been used.
            vector<bool> visited(n, false);
            visited[i] = true;

            // This will store the original array.
            vector<int> originalArray;
            originalArray.push_back((nums[0] + nums[i]) / 2); // Add the first element.

            // Use two pointers to recover the original array using the current difference 'diff'.
            for (int left = 1, right = i + 1; right < n; ++left, ++right) {
                // Find the next unvisited element for the left pointer.
                while (left < n && visited[left]) ++left;

                // Find the corresponding pair for the left element such that the difference is 'diff'.
                while (right < n && nums[right] - nums[left] < diff) ++right;

                // Break if there is no pair found or the difference is larger than 'diff'.
                if (right == n || nums[right] - nums[left] > diff) break;

                visited[right] = true; // Mark the right element as visited.

                // Recover and add the original element to 'originalArray'.
                originalArray.push_back((nums[left] + nums[right]) / 2);
            }

            // If we have successfully recovered the whole array, return it.
            if (originalArray.size() == (n / 2)) return originalArray;
        }

        // Return an empty array if no valid solution is found.
        return {};
    }
};

```

Typescript Solution

```typescript
function recoverArray(nums: number[]): number[] {
    // Sort the input array.
    nums.sort((a, b) => a - b);

    // Iterate through the array, attempting to find the correct difference 'd'.
    for (let i = 1; i < nums.length; i++) {
        let diff = nums[i] - nums[0];

        // Skip if the difference is zero or odd, since the difference should be even.
        if (diff === 0 || diff % 2 === 1) {
            continue;
        }

        // Initialize an array to keep track of visited elements.
        let visited: boolean[] = new Array(nums.length).fill(false);
        visited[i] = true;

        // Initialize the array that will store the recovered original array.
        let originalArray: number[] = [];
        // Add the first element of the original array.
        originalArray.push((nums[0] + nums[i]) / 2);

        // Use two pointers to attempt to recover the original array using the current difference 'diff'.
        for (let left = 1, right = i + 1; right < nums.length;) {
            // Find the next unvisited element for the left pointer.
            while (left < nums.length && visited[left]) {
                left++;
            }

            // Find the corresponding pair for the left element that meets the difference 'diff'.
            while (right < nums.length && nums[right] - nums[left] < diff) {
                right++;
            }

            // If the pointers exceed bounds or the difference exceeds 'diff', break out.
            if (right >= nums.length || nums[right] - nums[left] > diff) {
                break;
            }

            // Mark the right element as visited.
            visited[right] = true;

            // Recover and add the next element to the 'originalArray'.
            originalArray.push((nums[left] + nums[right]) / 2);
            // Move to the next possible unvisited elements.
            left++;
            right++;
        }

        // If the original array was successfully recovered, return it.
        if (originalArray.length === nums.length / 2) {
            return originalArray;
        }
    }

    // Return an empty array if no valid solution is found.
    return [];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm can be analyzed as follows:

1. Sorting the `nums` array: Sorting an array of size `n` typically takes `O(n log n)` time.

2. The outer loop runs at most `n` times because it iterates through the sorted `nums` array starting from the second element.

3. For each element in the outer loop, there are two inner loops (while loops) that could potentially run `n` times each in the worst case.
   - The first inner loop increments `l` until it finds an unvisited element.
   - The second inner loop increments `r` until it finds the pair element that satisfies `nums[r] - nums[l] == d`.

However, each element from the `nums` array gets paired at most once due to the `vis` array tracking the visited status. This means that in total, each inner loop contributes at maximum `n` iterations across all iterations of the outer loop, not `n` per outer loop iteration.

Thus, the time complexity is primarily dictated by the outer loop and the pairing process, leading to a complexity of `O(n log n)` for sorting plus `O(n)` for pairing, which simplifies to `O(n log n)` for the entire algorithm.

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

Space Complexity

The space complexity can be considered by analyzing the storage used by the algorithm:

1. The `vis` array: An array of size `n` used to keep track of the visited elements, which occupies `O(n)` space.

2. The `ans` array: Potentially, this array could store `n/2` elements when all the correct pairings are made (since pairs are formed), so it uses `O(n)` space as well.

As such, the space complexity of the algorithm is determined by the additional arrays used for keeping track of visited elements and storing the answer, leading to `O(n)` space complexity.

In conclusion, the space complexity is `O(n)`.