2007. Find Original Array From Doubled Array

Medium | Greedy | Array | Hash Table | Sorting | Leetcode Link

Problem Description

This problem provides an array called `changed`, which has been created by taking an original array of integers, doubling each element, appending these doubled numbers to the original array, and then shuffling the entire collection of numbers.

Our task is to reconstruct the original array from the `changed` array. However, there are some constraints:

1. If the `changed` array has an odd number of elements, it's impossible to obtain the original array since doubling each element of the original array and then merging would result in an even number of total elements.
2. For each element in the `original` array, there must be a corresponding element in `changed` that is double its value.
3. If `changed` does not satisfy the property of being a "doubled array," as described, we need to return an empty array.

We are allowed to return the elements of the `original` array in any order, adding flexibility to our solution.

Intuition

To find the solution, the following intuition can guide us:

1. **Sorting**: We start by sorting `changed` since this will arrange all double elements after their corresponding originals (since 2 * x > x for all x > 0). Sorting also helps us to handle duplicates effectively.

2. **Counting**: Using a counter (frequency map) over the sorted `changed` array helps in keeping track of the elements we've used for forming the `original` array and the elements that need to be paired with their doubles.

3. **Pairing and Elimination**: We iterate over the sorted `changed` array and look for the double of each element. If for any element `x`, the element `2x` does not exist in sufficient quantity (or does not exist at all), we cannot form the `original` array, hence, we return an empty array.

4. **Building the Original Array**: If an element `x` and its double `2x` are found, we pair them and reduce their count in the frequency map. The element `x` is added to the `original` array.

5. **Validation**: In the end, if the generated `original` array has exactly half the length of the `changed` array, we have successfully reformed the `original` array. Otherwise, the `changed` array wasn't a doubled array, to begin with, or we couldn't pair all elements correctly, and we return an empty array.

Using this approach caters to all conditions provided in the problem, thus guaranteeing a correct solution when one is possible.

Solution Approach

The implementation follows the intuition closely using Python's built-in data structures and sorting algorithm.

Here's the step-by-step breakdown of the solution approach:

1. **Check for the Odd Number of Elements**: A quick check to confirm if the length of the `changed` array is odd. Since the doubled array should be even in length, return an empty array immediately if this is the case.

```python
n = len(changed)
if n & 1:
    return []
```

2. **Use a Counter for Frequency Tracking**: A `Counter` object from the `collections` module is utilized to keep a frequency map of the elements in `changed`.

```python
cnt = Counter(changed)
```

3. **Sort the `changed` Array**: The `changed` array is sorted to ensure that the elements and their doubles are aligned in increasing order.

```python
changed.sort()
```

4. **Iterate to Build the `original` Array**: A for-loop iterates over each element in the sorted `changed` array, applying the following logic:

   - **Skip Processed Elements**: If an element `x` has already been used (i.e., its count is 0), skip it.

```python
if cnt[x] == 0:
    continue
```

   - **Check for Double's Existence**: If there is not enough of the double of `x` remaining (i.e., `cnt[2 * x]` is 0 or negative), the `changed` cannot be a doubled array, so return an empty list.

```python
if cnt[x * 2] <= 0:
    return []
```

   - **Add Element to `original` and Update Counts**: If a valid double is found, append `x` to the `original` array and decrement the counts for `x` and `2 * x`.

```python
ans.append(x)
cnt[x] -= 1
cnt[x * 2] -= 1
```

5. **Final Validation and Return**: After the loop is done, check if the length of the `original` array is exactly half the `changed` array (which implies each element was paired correctly). If not, the `changed` array couldn't have been a doubled array, and thus, return an empty array.

```python
return ans if len(ans) == n // 2 else []
```

By utilizing a sorted array and a frequency map, the algorithm ensures that all elements can be paired with their corresponding doubles, maintaining linearithmic time complexity due to sorting, with the remainder of operations being linear within the sorted array. The space complexity is linear due to the extra space used for the `Counter` and the resulting `original` array.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following `changed` array:

```python
changed = [1, 3, 4, 2, 6, 8]
```

Step-by-step, we perform the following actions:

1. **Check for Odd Number of Elements**: The length of the `changed` array is 6, which is even. We can proceed.

2. **Use a Counter for Frequency Tracking**: Generate a frequency map:

```python
cnt = Counter([1, 3, 4, 2, 6, 8]) # Counter({1: 1, 3: 1, 2: 1, 4: 1, 6: 1, 8: 1})
```

3. **Sort the `changed` Array**: The sorted `changed` array looks like this:

```python
changed = [1, 2, 3, 4, 6, 8]
```

4. **Iterate to Build the `original` Array**:

   - For the first element `1`, we find that its double `2` exists, so we add `1` to `original` and update the counts:

```python
ans = [1]
cnt = {1: 0, 3: 1, 2: 0, 4: 1, 6: 1, 8: 1}
```

   - We skip `2` since its count is now `0`.

   - Next, for `3`, we find its double `6`, add `3` to `original`, and update counts:

```python
ans = [1, 3]
cnt = {1: 0, 3: 0, 2: 0, 4: 1, 6: 0, 8: 1}
```

   - We skip `4` as we don't have `8` (double of `4`) in enough quantity (count is `0`), and since we can't find a valid double, we would normally return an empty array. However, for demonstration purposes, we will assume the double `8` exists and continue:

```python
ans = [1, 3, 4]
cnt = {1: 0, 3: 0, 2: 0, 4: 0, 6: 0, 8: 0}
```

5. **Final Validation and Return**: Now `ans` has 3 elements, which is half of the `changed` array's length. Therefore, assuming that a double for each element existed, our `original` array is `[1, 3, 4]`.

This simplified example demonstrates how each step logically brings us closer to the reconstructed `original` array or leads us to conclude that reconstruction is not possible if the conditions are not met.

Python Solution

```python
from collections import Counter

class Solution:
    def findOriginalArray(self, changed: List[int]) -> List[int]:
        # Get the length of the array
        n = len(changed)
        # If the length of the array is odd, we cannot form a doubled array
        if n % 2 == 1:
            return []
        # Count the frequency of each element in the array
        element_counter = Counter(changed)
        # Sort the array to handle pairs in ascending order
        changed.sort()
        # Initialize the original array
        original_array = []

        # Iterate over the sorted array
        for number in changed:
            # Skip the number if it has already been paired
            if element_counter[number] == 0:
                continue
            # If there isn't a double of this number, we can't form a valid array
            if element_counter[number * 2] <= 0:
                return []
            # Add the number to the original array and adjust the counts for the number and its double
            original_array.append(number)
            element_counter[number] -= 1
            element_counter[number * 2] -= 1

        # Return the original array only if it is half the size of the changed array; otherwise, return an empty array
        return original_array if len(original_array) == n // 2 else []

# Here is the correct usage of list and typing import for the List type annotation
from typing import List

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public int[] findOriginalArray(int[] changed) {
        // Find the length of the changed array
        int length = changed.length;

        // if the length is odd, there cannot be an original array
        // because the original and double elements aren't in pairs
        if (length % 2 == 1) {
            return new int[0];
        }

        // Sort the array to ensure the paired items can be found easily
        Arrays.sort(changed);

        // Initialize the count array to keep track of the frequency of each number
        int[] frequency = new int[changed[length - 1] + 1];
        for (int num : changed) {
            frequency[num]++;
        }

        // Initialize the resulting array with half the length of the changed array
        int[] result = new int[length / 2];
        int resultIndex = 0;

        // Go through the elements in changed array to find the original numbers
        for (int num : changed) {
            // Skip already paired numbers
            if (frequency[num] == 0) {
                continue;
            }
            // If the double value is out of the frequency array's range or already used up
            if (num * 2 >= frequency.length || frequency[num * 2] <= 0) {
                return new int[0];
            }
            // If a valid pair is found, put it in the result
            result[resultIndex++] = num;
            // Decrement the counts for the number and its double
            frequency[num]--;
            frequency[num * 2]--;
        }

        // Check if we've successfully found the original array
        return resultIndex == length / 2 ? result : new int[0];
    }
}

```

C++ Solution

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

class Solution {
public:
    vector<int> findOriginalArray(vector<int>& changed) {
        int n = changed.size();

        // If the size is odd, it's impossible to form an original array
        if (n % 2 != 0) {
            return {};
        }

        // Sort the array to make sure that for every element x, x*2 comes after x if it exists
        sort(changed.begin(), changed.end());

        // Create a frequency array that accounts for all elements in 'changed'
        vector<int> frequency(changed.back() + 1, 0);
        for (int x : changed) {
            frequency[x]++;
        }

        // Initialize the vector to store the original array elements
        vector<int> originalArray;

        // Iterate over the sorted 'changed' array
        for (int x : changed) {
            if (frequency[x] == 0) {
                // If the current element's count is already 0, skip it
                continue;
            }

            if (x * 2 >= frequency.size() || frequency[x * 2] <= 0) {
                // If there are no elements double the current or outside of the count range, return empty
                return {};
            }

            // Decrement the count of the element and its double
            originalArray.push_back(x);
            frequency[x]--;
            frequency[x * 2]--;
        }

        // If the size of formed originalArray is exactly half of the 'changed' array, return it
        // Otherwise, return an empty vector indicating no valid original array could be formed
        return originalArray.size() == n / 2 ? originalArray : vector<int>();
    }
};

```

Typescript Solution

```typescript
function findOriginalArray(changed: number[]): number[] {
    // Check if the array length is odd; if so, there can't be an original array
    const length = changed.length;
    if (length % 2 !== 0) {
        return [];
    }

    // Create a map to count occurrences of each number
    const frequencyCounter = new Map<number, number>();

    // Populate the frequency counter with the frequency of each number
    for (const number of changed) {
        frequencyCounter.set(number, (frequencyCounter.get(number) || 0) + 1);
    }

    // Sort the array to process pairs in order
    changed.sort((a, b) => a - b);

    // Initialize an array to store the original array
    const originalArray: number[] = [];

    // Iterate through the sorted array to find and validate pairs
    for (const number of changed) {
        // If the current number is already processed, skip it
        if (frequencyCounter.get(number) === 0) {
            continue;
        }

        // If there is no valid pair for the current number, return an empty array
        if ((frequencyCounter.get(number * 2) || 0) <= 0) {
            return [];
        }

        // Add the current number to the original array
        originalArray.push(number);

        // Decrement the frequency of the current number and its pair
        frequencyCounter.set(number, frequencyCounter.get(number)! - 1);
        frequencyCounter.set(number * 2, frequencyCounter.get(number * 2)! - 1);
    }

    // If the original array's length is half of the changed array, return it; otherwise, return an empty array
    return originalArray.length * 2 === length ? originalArray : [];
}

```

Time and Space Complexity

Time Complexity

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

1. Getting the length of `changed` (`n = len(changed)`) is an `O(1)` operation.
2. Checking for even length (`if n & 1`) is also an `O(1)` operation.
3. Creating a counter (`cnt = Counter(changed)`) counts the frequency of each number in `changed`, which is an `O(n)` operation.
4. Sorting the array (`changed.sort()`) takes `O(n log n)` time.
5. The loop runs through the sorted list of numbers (`for x in changed`). In the worst case, it runs `n` times. Each operation inside is `O(1)` because accessing and modifying the counter is constant time, given a good hash function.
6. Condition checks inside the loop and counter updates are all `O(1)`.
7. The last return statement (`return ans if len(ans) == n // 2 else []`) is an `O(1)` operation.

The most time-consuming steps are the counter creation and the sorting. Adding the complexities, we get:

`O(n)` (for counting) + `O(n log n)` (for sorting) + `O(n)` (for iterating through the list) = `O(n log n)`.

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

Space Complexity

For space complexity:

1. Additional space is used to store the counter (`cnt`), which can be up to `O(n)` if all numbers are unique.
2. Space for the output array (`ans`), which is half of the input array's size in the best case, so `O(n/2)` simplifies to `O(n)`.

Hence, the overall space complexity of the algorithm is `O(n)`.