954. Array of Doubled Pairs

Medium Accepted Array Hash Table Sorting

Leetcode Link

Problem Description

Given an array `arr` of integers of even length, the task is to find out whether it is possible to rearrange the array such that, for every number `arr[2*i]`, there exists a corresponding number `arr[2*i + 1]` which is double the value of `arr[2*i]`. This condition must hold true for all values of `i` that range from `0` to `len(arr) / 2 - 1`. If such a reordering is possible, return `true`; otherwise, return `false`.

Intuition

The intuition behind the solution involves understanding that each element must have a counterpart that is exactly double its value. One effective way to approach this problem is to count the frequency of each element in the array using a hashmap or counter, which provides O(1) average time to check for the existence of the double value.

Sorting the keys of this hashmap by their absolute value helps deal with both positive and negative numbers equally because the sorted counterpart should be 2 times that number regardless of its sign. Starting with the smallest absolute value ensures that if we find a match (i.e., a number that is double the current number), we can be sure that we haven't erroneously attempted to match it with a number too large or too small.

The core idea of this solution is that for every key x, there should be at least as many numbers $2*x$ as there are x since we want to pair each x with a $2*x$. If at any point, the frequency of $2*x$ is lower than the frequency of x, then it is impossible to pair all x's with their corresponding double values, and therefore, the function should return false. If we successfully pair all elements, then we return true.

However, keep in mind that the case for 0 is special since it must be paired with itself. Hence if there is an odd number of zeros, it is not possible to pair them all up and we should return false right away.

Solution Approach

1. First, count the occurrence (frequency) of each element in the array using the `Counter` class from Python's `collections` module.
2. Check the special case for 0. If there is an odd number of 0s, return false because you cannot pair all 0s with double their value.
3. Sort the keys of the frequency counter by the absolute value to ensure proper pairing regardless of the signs of the numbers.
4. Iterate through each number x in the sorted keys:
   a. Check if there are fewer numbers at $2*x$ than x. If so, return false—while attempting to pair numbers, we have encountered more of a number than its pair.
   b. Decrease the frequency of $2*x$ by the frequency of x to "pair up" the x values with $2*x$ values.
5. If we successfully iterate over all numbers without returning false, it means we can pair all numbers with their doubles, so we return true.

Solution Approach

The solution leverages the `Counter` class from Python's `collections` module to create a frequency table to keep track of the occurrences of each element. This approach is efficient because it provides constant-time (`O(1)`) access to each element's count, a crucial feature for the solution's performance.

1. Initialize the frequency counter for the array elements:

```
freq = Counter(arr)
```

2. Special case handling for 0:

```
if freq[0] & 1:
    return False
```

Here, we're using the bitwise AND operator `&` to check if the number of zeros is odd. If it's odd, it's not possible to pair 0 with double its value (also 0), so we return `False`.

3. Sort the keys of the frequency counter by their absolute value:

```
for x in sorted(freq, key=abs):
```

The keys are sorted in ascending order of their absolute value, which allows us to iterate from the smallest to the largest by considering their magnitude. This is essential to handle negative numbers correctly.

4. Iterate through each sorted key and try to pair it with its double:

```
if freq[x << 1] < freq[x]:
    return False
freq[x << 1] -= freq[x]
```

First, we check if the frequency of the double of x (achieved by left-shifting x or $x << 1$, which is equivalent to $2 * x$) is less than the frequency of x. If it is, we can't pair all x with $2 * x$, hence we return `False`.

If the condition passes, we reduce the frequency of $2 * x$ by the frequency of x. This effectively pairs up the x's and the $2 * x$'s.

5. If all pairs are successfully matched, the function returns `True`:

```
return True
```

The key principle in this approach is the management of the frequency counter and ensuring that for every element x with a non-zero count, there is a sufficient count of elements $2 * x$ to match with. Sorting by absolute value ensures that we pair elements starting from the smallest magnitude, making it impossible to create invalid pairings with elements that already should have been paired with smaller numbers.

The algorithm's time complexity primarily depends on the sorting procedure, which operates in `O(n log n)` time, where n is the length of the array. The subsequent iteration has a linear run time concerning the number of unique elements. The space complexity depends on the number of unique elements as well, which determines the size of the frequency counter.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Given the input array `arr = [4, 8, 1, 2, 3, 6]`, we need to check if it can be rearranged so that for every number at `arr[2*i]`, there is a number at `arr[2*i + 1]` that is double its value.

1. First, we count the frequency of each element:

```
freq = {4: 1, 8: 1, 1: 1, 2: 1, 3: 1, 6: 1}
```

2. There are no zeros in this example, so we don't need to worry about the special case for 0.
3. We sort the keys of the frequency counter by absolute value to get the numbers in the correct pairing order:

```
sorted keys by absolute value = [1, 2, 3, 4, 6, 8]
```

4. Now, we iterate through these sorted keys, checking and pairing each number with its double:
   - For `1`, there is a corresponding `2` (double of `1`), so we pair them up and reduce the frequency of `2` by the frequency of `1`:

```
freq = {4: 1, 8: 1, 1: 0, 2: 0, 3: 1, 6: 1}
```

   - Next, we have `3`, and its double `6` is also available. We pair them and adjust the frequencies:

```
freq = {4: 1, 8: 1, 1: 0, 2: 0, 3: 0, 6: 0}
```

   - Finally, we pair `4` with `8`:

```
freq = {4: 0, 8: 0, 1: 0, 2: 0, 3: 0, 6: 0}
```

5. All elements have been successfully paired with their doubles, so we return `True`

If, at any point during the iteration, the frequency of the current element's double was insufficient to pair with every instance of the current element, we would return `False`. However, in this example, the pairing is successful, so the given array meets the conditions defined in the problem.

Python Solution

```
from collections import Counter

class Solution:
    def canReorderDoubled(self, arr: List[int]) -> bool:
        # Create a frequency counter for elements in the array
        element_frequency = Counter(arr)

        # If the frequency of 0 is odd, pairs of twice 0 can't be formed
        if element_frequency[0] % 2 != 0:
            return False

        # Sort the unique elements by their absolute values
        for element in sorted(element_frequency, key=abs):
            # The number of times the double of this element should be
            # at least the number of times the element itself occurs
            if element_frequency[element * 2] < element_frequency[element]:
                return False
            # Decrement the frequency of the double element
            element_frequency[element * 2] -= element_frequency[element]

        # If all elements can be paired with their doubles, return true
        return True
```

Java Solution

```
class Solution {

    public boolean canReorderDoubled(int[] arr) {
        // Create a frequency map to count occurrences of each number in the array
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int number : arr) {
            frequencyMap.put(number, frequencyMap.getOrDefault(number, 0) + 1);
        }

        // If there's an odd number of zeroes, it's not possible to reorder as per the rules
        if (frequencyMap.getOrDefault(0, 0) % 2 != 0) {
            return false;
        }

        // Create a list of unique keys (numbers) from the map and sort them by their absolute values
        List<Integer> keys = new ArrayList<>(frequencyMap.keySet());
        keys.sort(Comparator.comparingInt(Math::abs));

        // Iterate through the keys
        for (int key : keys) {
            // If the frequency of the double is less than the frequency of the current key, return false
            if (frequencyMap.getOrDefault(key * 2, 0) < frequencyMap.get(key)) {
                return false;
            }
            // Decrease the frequency of the doubled number by the frequency of the current key
            frequencyMap.put(key * 2, frequencyMap.getOrDefault(key * 2, 0) - frequencyMap.get(key));
        }

        // If the loop completes without returning false, reordering is possible
        return true;
    }
}
```

C++ Solution

```
#include <vector>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    // Function to determine if it is possible to reorder the array such that
    // for every element x in the array, there exists another element 2 * x as well.
    bool canReorderDoubled(std::vector<int>& arr) {
        // Create a frequency map for all elements in the array.
        std::unordered_map<int, int> frequency_map;
        for (int value : arr) {
            ++frequency_map[value];
        }

        // If there's an odd number of zeros, it's not possible to pair all of them,
        // hence we can return false immediately.
        if (frequency_map[0] % 2 != 0) return false;

        // Extract keys from the frequency map to sort them based on their absolute values.
        std::vector<int> keys;
        for (const auto& element : frequency_map) {
            keys.push_back(element.first);
        }

        // Sorting the keys based on the absolute values of the elements.
        std::sort(keys.begin(), keys.end(), [](int a, int b) {
            return std::abs(a) < std::abs(b);
        });

        // Iterate over the sorted keys.
        for (int key : keys) {
            // If there are not enough double elements to pair with the current element,
            // it's not possible to reorder the array as required.
            if (frequency_map[key * 2] < frequency_map[key]) return false;

            // Decrement the count of the pair element by the frequency of the current element.
            frequency_map[key * 2] -= frequency_map[key];
        }

        // If all elements can be paired successfully, return true.
        return true;
    }
};
```

Typescript Solution

```
// Import necessary functions from the "lodash" library.
import { sortBy, countBy } from "lodash";

// Function to determine if it is possible to reorder the array such that
// for every element x in the array, there exists another element 2 * x as well.
function canReorderDoubled(arr: number[]): boolean {
    // Create a frequency map of all elements in the array using the countBy function.
    const frequencyMap: Record<number, number> = countBy(arr);

    // If there's an odd number of zeros, it's not possible to pair all of them,
    // hence we can return false immediately.
    if (frequencyMap[0] % 2 !== 0) return false;

    // Extract keys from the frequency map and sort them based on their absolute values.
    const keys: number[] = Object.keys(frequencyMap).map(Number);
    const sortedKeys: number[] = sortBy(keys, Math.abs);

    // Iterate over the sorted keys.
    for (const key of sortedKeys) {
        // If there are not enough double elements to pair with the current element,
        // it's not possible to reorder the array as required.
        if (frequencyMap[key * 2] || 0) < frequencyMap[key]) return false;

        // Decrement the count of the pair element by the frequency of the current element.
        frequencyMap[key * 2] -= frequencyMap[key] || 0) - frequencyMap[key];
    }

    // If all elements can be paired successfully, return true.
    return true;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be broken down into several parts:

1. **Counter Creation**: The `Counter` creation from the `arr` list runs in `O(n)` time, where `n` is the length of `arr`, as it needs to iterate through the list once to count the frequency of each element.
2. **Sorting**: The `sorted` function sorts the keys of the frequency counter. Sorting is typically `O(n log n)`. In this case, it's sorting based on the absolute values, but this doesn't change the time complexity.
3. **Iterating through Sorted Keys**: The for-loop iterates through each unique value in `arr` (after it's been sorted), which in the worst case can be `O(n)` when all elements are unique. However, for each element, it's only doing a constant amount of work by adjusting the frequency of the doubled element. Therefore, this step is `O(n)`.

Combining these, the sorting step dominates the time complexity, leading to an overall time complexity of `O(n log n)`.

Space Complexity

The space complexity of the given code is as follows:

1. **Counter**: A Counter is created to store the frequency of each element in `arr`. In the worst case, if all elements are unique, it requires `O(n)` space.
2. **Sorted List**: The `sorted` function returns a new list containing the sorted keys, which also requires `O(n)` space in the worst case if all elements are unique.

However, the space used by the new sorted list does not add to the space complexity because it's not allocating additional space that grows with the input size; it's dependent on the number of unique elements which is already accounted for by the Counter. So the overall space complexity remains `O(n)`.

Therefore, the final space complexity is also `O(n)`.