1497. Check If Array Pairs Are Divisible by k

Medium Array Hash Table Counting

Problem Description

The problem presents an array of integers `arr` with an even number of elements `n` and an integer `k`. The task is to determine whether `arr` can be partitioned into `n/2` pairs, where the sum of the integers in each pair is divisible by `k`. If such a partitioning is possible, the function should return `true`; otherwise, it should return `false`.

Intuition

The solution is built on the property of divisibility by an integer `k`. If two numbers `a` and `b` have a sum that is divisible by `k`, it means that `(a + b) % k == 0`. In other words, the remainder when `a + b` is divided by `k` is 0.

For any integer `x`, `x % k` will give a remainder in the range `[0, k-1]`. If `x % k` is `0`, then `x` is divisible by `k`. To make a pair with a sum divisible by `k`, each number `x` in the array that has a non-zero remainder `x % k` requires a corresponding number `y` such that `(x + y) % k == 0`.

We can think of the remainders as forming pairs themselves. For a non-zero remainder `i`, there must be an equal number of elements with remainder `k - i`.

To arrive at the solution, we use a counter to track the frequency of each remainder in the array. We then check two conditions:

1. The count of elements with a remainder of `0` must be even because they can only be paired with other elements with a remainder of `0`.
2. For each non-zero remainder `i`, there should be an equal count of elements with remainder `i` and `k - i`.

Solution Approach

The implementation of the solution involves using the `Counter` class from Python's `collections` module to efficiently count the occurrences of each remainder when elements of the array are divided by `k`. The `Counter` object will map each unique remainder to the number of times it appears in the array.

Here's a step-by-step walkthrough of the solution:

1. Calculate the remainder of each element in the array when divided by `k`. This is done using a list comprehension: `[x % k for x in arr]`.
2. Feed this list of remainders into `Counter` to get the frequency count of each remainder: `Counter(x % k for x in arr)`.
3. Verify the first condition for elements with zero remainders:
   - `cnt[0] % 2 == 0`. This checks if the count of elements that are exactly divisible by `k` is even. These elements can only be paired with themselves, so an odd count would make it impossible to form pairs where the sum is divisible by `k`.
4. Verify the second condition for elements with non-zero remainders:
   - Loop through the range from 1 to `k - 1` and for each `i`, check if the count of elements with a remainder of `i` is equal to the count of elements with a remainder of `k - i`.
   - This check is performed by the list comprehension: `all(cnt[i] == cnt[k - i] for i in range(1, k))`.
   - The `all` function ensures that this condition must hold true for all remainders in the specified range for the function to return `true`.

The use of the `Counter` and the `all` function makes the algorithm efficient and concise. The time complexity of this algorithm is primarily dependent on the time it takes to traverse the array and calculate the remainders, which is (O(n)), and the time to check the pair frequencies, which is (O(k)). This makes the overall time complexity (O(n + k)).

Here's the relevant snippet from the Python solution for reference:

```python
class Solution:
    def canArrange(self, arr: List[int], k: int) -> bool:
        cnt = Counter(x % k for x in arr)
        return cnt[0] % 2 == 0 and all(cnt[i] == cnt[k - i] for i in range(1, k))
```

By adhering to these steps, the algorithm efficiently solves the problem while respecting the constraints of even length arrays and the divisibility requirement.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have an array `arr = [2, 4, 1, 5]` and an integer `k = 3`. We need to check if we can partition this array into pairs such that the sum of each pair is divisible by `k`. Since there are `4` elements in `arr`, we are looking for `n/2 = 2` pairs.

Following the solution steps:

1. Calculate the remainders of each element when divided by `k = 3`:
   - `2 % 3 = 2`
   - `4 % 3 = 1`
   - `1 % 3 = 1`
   - `5 % 3 = 2` The remainders are `[2, 1, 1, 2]`.
2. Count the frequency of each remainder using the `Counter`:
   - Remainder `1` appears `2` times.
   - Remainder `2` appears `2` times.
3. Verify the first condition for elements with zero remainders:
   - Our example does not have any element with a remainder of `0`, so we can skip this.
4. Verify the second condition for elements with non-zero remainders:
   - Check each `i` from `1` to `k - 1`, which in this case is just `1` and `2` since `k = 3`.
   - For `i = 1`: Count is `2`.
   - For `k - i = 3 - 1 = 2`: Count is `2`.
   - Since the counts are equal for `i` and `k - i`, the condition is met.

Since the array only has non-zero remainders and each remainder `i` has an equal count to `k - i`, we have successfully verified that it is possible to partition the array into `n/2` pairs where the sum of each pair is divisible by `k`. Thus, the function would return `true`.

The corresponding Python function would process the information as follows:

```python
from collections import Counter

def canArrange(arr: List[int], k: int) -> bool:
    cnt = Counter(x % k for x in arr)
    # Since there is no remainder of 0, we skip checking cnt[0] % 2
    # Validate the second condition using a list comprehension and all()
    return all(cnt[i] == cnt[k - i] for i in range(1, k))

# Example array and k
arr = [2, 4, 1, 5]
k = 3

# Call the function with the example inputs
print(canArrange(arr, k))  # Output: True
```

By adhering to the solution steps, we are given a working example of the algorithm applied to this simple array, confirming the algorithm's correctness in this scenario.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def canArrange(self, arr: List[int], k: int) -> bool:
        # Count the frequency of each remainder when each element in arr is divided by k
        remainder_count = Counter(x % k for x in arr)

        # Check if the number of elements that are divisible by k is even
        if remainder_count[0] % 2 != 0:
            return False

        # Check if each non-zero remainder has a complementary count of elements
        # Such that remainder + complementary = k
        # The counts of remainder and its complementary need to be the same
        for remainder in range(1, k):
            if remainder_count[remainder] != remainder_count[k - remainder]:
                return False

        # All checks have passed, hence return True
        return True
```

Java

```java
class Solution {

    public boolean canArrange(int[] arr, int k) {
        // Create an array to store counts of modulo results
        int[] count = new int[k];

        // Loop through each number in the input array
        for (int number : arr) {
            // Increment the count at the index equal to the number's modulo k,
            // taking into account negative numbers by adding k before modding
            count[(number % k + k) % k]++;
        }

        // Check pairs for each possible modulo result except for 0
        for (int i = 1; i < k; ++i) {
            // For a valid pair, count of modulo i should be equal to count of modulo (k - i)
            if (count[i] != count[k - i]) {
                // If they are not equal, the condition is not met, so return false
                return false;
            }
        }

        // Check that count of numbers that are exactly divisible by k (modulo result is 0)
        // is an even number since they need to be paired among themselves
        return count[0] % 2 == 0;
    }
}
```

C++

```cpp
class Solution {
public:
    // This function checks if it's possible to rearrange 'arr' such that the sum of every pair of elements is divisible by 'k'.
    bool canArrange(vector<int>& arr, int k) {
        // Create a count array to store frequencies of mod values
        vector<int> count(k, 0);

        // Increment the count for each element's mod value with 'k', properly handling negative numbers
        for (int number : arr) {
            int modValue = ((number % k) + k) % k;
            ++count[modValue];
        }

        // Check pairs from 1 to k-1. For every 'i', there must be equal number of elements with 'k-i' as mod.
        for (int i = 1; i < k; ++i) {
            if (count[i] != count[k - i]) {
                // If counts are not equal, pairs cannot be formed to satisfy the condition.
                return false;
            }
        }

        // The count of elements evenly divisible by 'k' must be even for them to be paired.
        return count[0] % 2 == 0;
    }
};
```

TypeScript

```typescript
// Function checks if it's possible to rearrange 'arr' so that the sum of every pair of elements is divisible by 'k'.
function canArrange(arr: number[], k: number): boolean {
    // Create an array to store frequencies of mod values
    let modCount: number[] = new Array(k).fill(0);

    // Increment the count for each element's mod value with 'k',
    // properly handling negative numbers
    arr.forEach(number => {
        let modValue: number = ((number % k) + k) % k;
        modCount[modValue]++;
    });

    // Check pairs from 1 to k - 1. For every 'i', there must be an equal number of elements with 'k - i' as their mod value.
    for (let i = 1; i < k; i++) {
        if (modCount[i] !== modCount[k - i]) {
            // If counts are not equal, pairs cannot be formed to satisfy the condition.
            return false;
        }
    }

    // The count of elements evenly divisible by 'k' must be even for them to be paired successfully.
    return modCount[0] % 2 === 0;
}

// Example usage:
// let result: boolean = canArrange([1, 2, 3, 4, 5, 10, -10], 5);
// console.log(result); // This should output true or false based on the array and k value.
```

```python
from collections import Counter

class Solution:
    def canArrange(self, arr: List[int], k: int) -> bool:
        # Count the frequency of each remainder when each element in arr is divided by k
        remainder_count = Counter(x % k for x in arr)

        # Check if the number of elements that are divisible by k is even
        if remainder_count[0] % 2 != 0:
            return False

        # Check if each non-zero remainder has a complementary count of elements
        # Such that remainder + complementary = k
        # The counts of remainder and its complementary need to be the same
        for remainder in range(1, k):
            if remainder_count[remainder] != remainder_count[k - remainder]:
                return False

        # All checks have passed, hence return True
        return True
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed by looking at the operations performed:

- The list comprehension `x % k for x in arr` iterates over each element in `arr`, resulting in `O(n)` time complexity, where `n` is the length of `arr`.
- The `Counter` from the standard `collections` module constructs the frequency dictionary in once again `O(n)` time complexity.
- The `return` statement consists of two parts:
  - Checking if `cnt[0] % 2 == 0` is a constant time `O(1)` operation because we access a dictionary key and then check for evenness.
  - The list comprehension `all(cnt[i] == cnt[k - i] for i in range(1, k))` could iterate up to `k` times. In the worst case, this is `O(k)`. However, since there are only 'n' unique modulo results possible (all elements `x` in `arr` are used to compute `x % k`), we are effectively iterating up to `min(n, k)`, hence `O(min(n, k))`.

Based on these observations, the total time complexity is `O(n) + O(1) + O(min(n, k))` which simplifies to `O(n + min(n, k))`. Since `n` and `k` are independent variables, we cannot simplify this further without knowing the relationship between `n` and `k`.

Space Complexity

The space complexity is determined by the auxiliary space used by the program. In this case:

- The frequency dictionary `cnt` may contain up to `min(n, k)` different keys (since these are the unique modulo results) and therefore has a space complexity of `O(min(n, k))`.
- The space complexity for the list comprehension is `O(1)` since it's not stored but used directly in the `all` function.

Consequently, the space complexity of the whole code is dominated by the space requirement of the frequency dictionary, which is `O(min(n, k))`.