# 2206. Divide Array Into Equal Pairs

Easy | Bit Manipulation | Array | Hash Table | Counting

## Problem Description

You have an array named `nums` with an even number of integers, precisely `2 * n` integers. Your objective is to determine if this array can be separated into `n` pairs where each pair consists of two identical elements. Every element should be used exactly once and only in one pair. The challenge is to figure out if it's possible to create such pairs evenly across the entire array. If it can be done, you should return `true`. If it's not possible and there's at least one element that can't form a pair with an identical counterpart, then the result should be `false`.

## Intuition

One way to approach this problem is to think about the conditions under which the array can be divided into `n` pairs. Since we need to pair equal elements, a straightforward requirement is that each unique number must occur an even number of times—specifically 2 times since we're creating pairs. Should there be any number with an odd count, it would be impossible to form the required pairs.

Hence, arriving at the solution involves counting how many times each unique value appears in the array. This is known as creating a frequency count. We use the `Counter` class from Python's `collections` module to make this very easy. Once we have the counts, we check each of them to ensure they are all even (a number having a count that is a multiple of 2). If even one count is odd, it means there's at least one element that cannot form a pair and thus, we cannot divide the array as needed.

## Solution Approach

The solution makes use of a hash map to count the occurrences of each number in the input array `nums`. In Python, this is conveniently accomplished using the `Counter` class from the `collections` module, which automatically creates a hash map (dictionary) where each key is a unique number from the array, and its value is the count of how many times it appears.

Here's a breakdown of the algorithm implemented in the provided solution:

1. Import the `Counter` from `collections` to create the frequency count. `cnt = Counter(nums)` will iterate over `nums`, and for each element in the array, it will increment its corresponding count in the `cnt` hash map.
2. Next, we need to verify whether each number appears an even number of times. We use a generator expression `v % 2 == 0 for v in cnt.values()` for this. This expression goes through each value (which represents the count of occurrences for a number) in the `cnt` hash map and checks if it is even.
3. The `all()` function wraps the generator expression to check if all elements in the expression evaluate to true. For a count `v`, if `v % 2 == 0` is true, then `v` is even.
4. If all counts are even, then `all()` returns `true`, meaning every element can be paired. If any count is odd, then `all()` returns `false`, meaning it's not possible to evenly divide the array into pairs.

The crux of the solution is to ensure two paired elements (which must be equal) are available for all the elements. If there's an odd number of any element, a pair cannot be formed, violating the problem's constraint.

The reference solution approach utilizes the Constant Time Operation property of the hash map which allows very efficient counting and retrieval operations, and the logical interpretation of the pair-formation rule (even number of elements) to arrive at a succinct and effective solution.

### Example Walkthrough

Let's take the array `nums = [1, 2, 1, 2, 1, 3, 3, 1]` for our example:

1. **Initialization**: We import the `Counter` class and initialize it with our array, `nums`. `Counter` tallies the counts of each distinct element, resulting in a hash map.

```
from collections import Counter
cnt = Counter(nums)  # Counter({1: 4, 2: 2, 3: 2})
```

   The resulting `cnt` dictionary contains keys that are the unique values from `nums`, with the counts as their respective values.

2. **Check Evenness**: We create a generator expression to evaluate whether each count is even. The expression is `v % 2 == 0 for v in cnt.values()`, which in our case will generate the following sequence of booleans:

```
[True, True, True]  # every count is even: 4 % 2 == 0, 2 % 2 == 0, 3 % 2 == 0
```

   Here, `v` refers to the count of each unique number in the array. Since `Counter({1: 4, 2: 2, 3: 2})` is our frequency map, the expression checks `(4 % 2 == 0, 2 % 2 == 0, 2 % 2 == 0)` corresponding to counts for `1`, `2`, and `3` respectively.

3. **Apply `all()` Function**: Utilizing the `all()` function, we confirm if every value in the generator expression is `True`.

```
can_pair = all(v % 2 == 0 for v in cnt.values())  # True
```

4. **Result**: The `all()` functioned returned `True` indicating that every count is even, which means that every unique number in the array appears an even number of times. Hence, it is possible to divide the array into pairs of identical elements.

Thus, in this case, the given array `nums` can indeed be paired up in the manner described in the problem statement, and the function would return `true`.

## Solution Implementation

### Python

```
from collections import Counter
from typing import List

class Solution:
    def divideArray(self, nums: List[int]) -> bool:
        # Create a counter dictionary to store the frequency of each number in the list
        num_counter = Counter(nums)

        # Use all() to check for each count in the counter values
        # Return True if all counts are even, which means pairs can be formed for each unique number
        # If there is any count that is not even, False is returned indicating pairs cannot be formed
        return all(count % 2 == 0 for count in num_counter.values())
```

### Java

```
class Solution {
    // Function to check if it's possible to divide the array into pairs such that each pair contains equal numbers.
    public boolean divideArray(int[] nums) {
        // Create an array to count the frequency of elements.
        // The maximum value for any element is considered to be 500, hence an array of size 510 is created for safe margin.
        int[] count = new int[510];

        // Iterate over the input array to count the frequency of each element.
        for (int num : nums) {
            // Increment the count of the current element.
            count[num]++;
        }

        // Iterate over the frequency array to check if all elements have an even count.
        for (int frequency : count) {
            // If an element has an odd count, we cannot divide the array into pairs with equal numbers.
            if (frequency % 2 != 0) {
                return false; // Return false as soon as an odd frequency is found.
            }
        }

        // If all elements have even counts, return true indicating that division into pairs is possible.
        return true;
    }
}
```

### C++

```
#include <vector>
using namespace std;

class Solution {
public:
    bool divideArray(vector<int>& nums) {
        // Create a frequency array with a fixed size,
        // assuming that 500 is the maximum number expected to be in the `nums` array.
        vector<int> frequency(501, 0); // Initialize all elements to 0

        // Increment the frequency count for each number in the `nums` array.
        for (int num : nums) {
            frequency[num]++;
        }

        // Check if all numbers occur an even number of times.
        for (int freq : frequency) {
            // If a number occurs an odd number of times, it's not possible
            // to divide the array into pairs such that each pair consists of equal numbers.
            if (freq % 2 != 0) {
                return false;
            }
        }

        // If the code reaches this point, all numbers occur an even number of times,
        // and the array can be divided into pairs of equal numbers.
        return true;
    }
};
```

### TypeScript

```
// Define the function to check if an array can be divided into pairs with equal numbers.
function divideArray(nums: number[]): boolean {
  // Create a frequency array with a fixed size,
  // assuming 500 is the maximum number expected to be in the `nums` array.
  // Initialize all elements to 0.
  const frequency: number[] = new Array(501).fill(0);

  // Increment the frequency count for each number in the `nums` array.
  for (const num of nums) {
    frequency[num]++;
  }

  // Check if all numbers occur an even number of times.
  for (const freq of frequency) {
    // If a number occurs an odd number of times, it's not possible
    // to divide the array into pairs such that each pair consists of equal numbers.
    if (freq % 2 !== 0) {
      return false;
    }
  }

  // If the code reaches this point, all numbers occur an even number of times,
  // and the array can be divided into pairs of equal numbers.
  return true;
}
```

```
from collections import Counter
from typing import List

class Solution:
    def divideArray(self, nums: List[int]) -> bool:
        # Create a counter dictionary to store the frequency of each number in the list
        num_counter = Counter(nums)

        # Use all() to check for each count in the counter values
        # Return True if all counts are even, which means pairs can be formed for each unique number
        # If there is any count that is not even, False is returned indicating pairs cannot be formed
        return all(count % 2 == 0 for count in num_counter.values())
```

## Time and Space Complexity

### Time Complexity

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

- Creating the `Counter` object from `nums` involves iterating over all `n` elements of the list to count the occurrences of each unique number. This step has a time complexity of `O(n)`.
- The `all` function then iterates over the values in the counter, which could be at most `n/2` if all numbers are unique and each appears exactly twice (which represents the case with the maximum possible number of keys). In the worst case, this will take `O(n/2)` which simplifies to `O(n)`.

Combining these two steps, the overall time complexity remains `O(n)` since constants are dropped in Big O notation.

### Space Complexity

Analyzing the space complexity involves considering the extra space taken by the algorithm aside from the input:

- The `Counter` object will store each unique number as a key and its frequency as a value. In the worst case, if all numbers are unique, the space required will be `O(n)`. However, this is the worst-case scenario; in the best case, if all numbers are the same, the space complexity would be `O(1)`.
- The `Counter` values are iterated in place using the `all` function, which doesn't take additional significant space.

Hence, the worst-case space complexity is `O(n)`.