914. X of a Kind in a Deck of Cards

Easy | Array | Hash Table | Math | Counting | Number Theory

Problem Description

The problem presents us with an array named `deck`, where each element in the array represents a number on a card. We need to determine if it's possible to divide these cards into one or more groups such that two conditions are met:

1. Each group must contain the *exact same* number of cards, `x`, where x is greater than 1.
2. All cards within any given group must have the *same number* written on them.

If such a division is possible, we should return `true`. Otherwise, the function should return `false`.

Intuition

The intuition behind the solution is to use number frequency and the greatest common divisor (GCD). The key idea is that if all the card numbers' frequency counts have a GCD greater than 1, we can form groups that satisfy the problem's conditions.

Here's why this works:

1. Count the frequency of each number in the deck using a frequency counter (`Counter(deck)`). This tells us how many times each unique number appears in the deck.
2. The crucial insight is that if we can find a common group size `x` that divides evenly into all counts, we can create the groups required. In mathematical terms, `x` must be a common divisor for all frequencies. We then use `reduce` and `gcd` to calculate the GCD across all frequency counts.
3. If the GCD of these counts is at least 2, this means there is a common divisor for all frequencies, and hence we can partition the deck into groups of size GCD or any multiple of the GCD that is also a divisor of all frequencies. If the GCD is 1, this means there's no common divisor greater than 1, and it would be impossible to partition the deck as desired.

By following this reasoning, we arrive at the provided solution approach.

Solution Approach

The implementation makes use of Python's standard library, particularly the `collections.Counter` class to calculate the frequency of each card and the `functools.reduce` function in combination with `[math](/problems/math-basics).gcd` to find the greatest common divisor (GCD) of the card frequencies.

Here's a step-by-step explanation:

1. We create a frequency counter for the `deck` array with `Counter(deck)` that returns a dictionary with card values as keys and their respective frequencies as values.
2. We extract the values (frequencies) from the counter and store them in the variable `vals` using `vals = Counter(deck).values()`.
3. The `reduce` function is used to apply the `gcd` (greatest common divisor) function repeatedly to the items of `vals` - effectively finding the GCD of all frequencies. The syntax `reduce(gcd, vals)` computes the GCD of the first two elements in `vals`, then the GCD of that result with the next element, and so on, until all elements are processed.
4. Finally, the GCD result is compared against the integer 2. If the GCD is greater than or equal to 2 (`reduce(gcd, vals) >= 2`), this means all card frequencies have a common divisor greater than 1. This allows them to be divided into groups satisfying the problem's constraints and thereby returning `true`. If the GCD is less than 2, it means there is no common divisor that can form the groups as required, and we return `false`.

The solution leverages the mathematical property that any common divisor of two numbers also divides their GCD. By finding the GCD of all frequency counts, we ensure that this number can be a valid group size for partitioning the deck accordingly.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the array `deck = [1,2,3,4,4,3,2,1,4,4]`.

1. First, we will count the frequency of each number using `Counter(deck)`. Applying this to our `deck` gives us the count `{1: 2, 2: 2, 3: 2, 4: 4}`. The key-value pairs here represent the card number and its frequency, respectively.
2. Next step is to extract these frequencies into a list: `vals = [2, 2, 2, 4]`. These are the counts of cards `1, 2, 3`, and `4` respectively.
3. Now we use the `reduce` function combined with the `gcd` function to find the GCD of all these frequency counts. For our example, the GCD of `[2, 2, 2, 4]` is `2`. This is calculated by starting with the GCD of the first two elements: `gcd(2, 2)`, which is `2`. Then it continues by taking this result and applying the GCD with the next element: `gcd(2, 2)` → `2`, and finally `gcd(2, 4)` → `2`.
4. After figuring out that our GCD is `2`, which is greater than or equal to 2, we can conclude that we can divide the deck into groups that meet both conditions mentioned. We have `1` pair, `2` pair, `3` pair, and two pairs of `4` which fits the criteria as each group (pair in this case) has the same number of cards, and all cards in each group have the same number.

Therefore, for this deck array `[1,2,3,4,4,3,2,1,4,4]`, the answer to whether the division is possible is `true`.

Solution Implementation

Python

```python
from collections import Counter
from functools import reduce
from math import gcd
from typing import List

class Solution:
    def hasGroupsSizeX(self, deck: List[int]) -> bool:
        # Count the occurence of each card value in the deck
        card_count = Counter(deck).values()

        # Use the reduce function to find the greatest common divisor (GCD) among all counts
        gcd_result = reduce(gcd, card_count)

        # Check if the GCD is at least 2 (which means there is a possible group size X that is divisible by all card counts)
        return gcd_result >= 2
```

Java

```java
class Solution {
    // Determines if we can partition the deck into groups with the same size and each group having the same integer
    public boolean hasGroupsSizeX(int[] deck) {
        // Create an array to hold the frequency of each value
        int[] count = new int[10000];
        // Count the occurrences of each number in the deck
        for (int num : deck) {
            count[num]++;
        }

        // Variable to store the greatest common divisor of all counts
        int gcdValue = -1;
        // Calculate the GCD of all the frequencies
        for (int frequency : count) {
            if (frequency > 0) {
                // If gcdValue is still -1, this is the first non-zero frequency found, so assign it directly
                // Otherwise, get the GCD of the current gcdValue and the new frequency
                gcdValue = gcdValue == -1 ? frequency : gcd(gcdValue, frequency);
            }
        }

        // Return true if the GCD of all frequencies is at least 2 (we can form groups of at least 2)
        return gcdValue >= 2;
    }

    // Recursive method to calculate the greatest common divisor (GCD) of two numbers
    private int gcd(int a, int b) {
        // The GCD of b and the remainder of a divided by b. When b is 0, a is the GCD.
        return b == 0 ? a : gcd(b, a % b);
    }
}
```

C++

```cpp
#include<vector>
#include<numeric> // For std::gcd (C++17 and above)

class Solution {
public:
    bool hasGroupsSizeX(vector<int>& deck) {

        // Array to count the occurrences of each number in the deck
        int counts[10000] = {0};
        for (int& value : deck) {
            // Increment the count for this number
            counts[value]++;
        }

        // Variable to store the greatest common divisor of all counts,
        // initial value -1 indicates that we haven't processed any count yet
        int gcdOfCounts = -1;
        for (int& count : counts) {
            if (count) { // Checking if the count is not zero
                if (gcdOfCounts == -1) {
                    // This is the first non-zero count, we assign it to gcdOfCounts
                    gcdOfCounts = count;
                } else {
                    // Calculate the GCD of the current gcdOfCounts and this count
                    gcdOfCounts = std::gcd(gcdOfCounts, count);
                }
            }
        }

        // A valid group size exists if the GCD of all counts is at least 2
        return gcdOfCounts >= 2;
    }
};
```

TypeScript

```typescript
function gcd(a: number, b: number): number {
  // Base case for recursion: If b is 0, gcd is a
  if (b === 0) return a;
  // Recursive case: gcd of b and the remainder of a divided by b
  return gcd(b, a % b);
}

function hasGroupsSizeX(deck: number[]): boolean {
  // Object to count the occurrences of each number in the deck
  const counts: { [key: number]: number } = {};

  // Count occurrences of each card
  for (const value of deck) {
    if (counts[value]) {
      counts[value]++;
    } else {
      counts[value] = 1;
    }
  }

  // Variable to store the greatest common divisor of all counts.
  // Initial value -1 indicates that we haven't processed any count yet.
  let gcdOfCounts = -1;

  // Iterate over card counts to find gcd
  for (const count of Object.values(counts)) {
    if (count) { // Checking if the count is not zero
      if (gcdOfCounts === -1) {
        // This is the first non-zero count, we assign it to gcdOfCounts
        gcdOfCounts = count;
      } else {
        // Calculate the GCD of the current gcdOfCounts and this count
        gcdOfCounts = gcd(gcdOfCounts, count);
      }
    }
  }

  // A valid group size exists if the GCD of all counts is at least 2
  return gcdOfCounts >= 2;
}
```


```python
from collections import Counter
from functools import reduce
from math import gcd
from typing import List

class Solution:
    def hasGroupsSizeX(self, deck: List[int]) -> bool:
        # Count the occurence of each card value in the deck
        card_count = Counter(deck).values()

        # Use the reduce function to find the greatest common divisor (GCD) among all counts
        gcd_result = reduce(gcd, card_count)

        # Check if the GCD is at least 2 (which means there is a possible group size X that is divisible by all card counts)
        return gcd_result >= 2
```


Time and Space Complexity

Time Complexity

The time complexity of the code involves a few operations. First, the counting of elements using `Counter`, which takes `O(n)` time, where `n` is the number of cards in the `deck`. Secondly, `reduce(gcd, vals)` computes the Greatest Common Divisor (GCD) of the counts. Calculating the GCD using Euclid's algorithm has a worst-case complexity of `O(log(min(a, b)))` for two numbers `a` and `b`. Since `reduce` applies gcd pair-wise to the values, the complexity in the worst case will be `O(m * log(k))`, where `m` is the number of unique cards in the deck and `k` is the smallest count of a unique card.

Overall, the time complexity is `O(n + m * log(k))`.

Space Complexity

The space complexity is `O(m)` due to the space used by the `Counter` to store the count of unique cards. `m` is the number of unique cards. The space used by the reduce operation is `O(1)` as it only needs space for the accumulator to store the ongoing GCD.