2341. Maximum Number of Pairs in Array

Easy Array Hash Table Counting

Leetcode Link

Problem Description

The problem presents you with an array of integers called `nums` and asks you to perform a series of operations on this array. In each operation, you must:

- Find two identical numbers in the array.
- Remove those two identical numbers from the array to form a 'pair'.

You continue performing this operation until it is no longer possible (i.e., there are no more identical numbers to pair up). The output is an array `answer` of size 2, where:

- `answer[0]` is the total number of pairs you have formed.
- `answer[1]` is the remaining number of integers left in `nums` after pairing up as much as you can.

The challenge is to calculate these two values efficiently.

Intuition

To find the solution to this problem, understanding the frequency of each number in the array is crucial. Two equal integers can form a pair, which means we need to count how often each integer appears in the array. The most straightforward approach is to use a frequency counter, which is a common method to store the number of occurrences of each element in a collection (like a list). This can be achieved elegantly in Python using the `Counter` class from the `collections` module. Once we have the counter, we can iterate through the values of this frequency counter:

- For each number in the counter, the number of pairs that can be formed is equal to `frequency of the number // 2`. We use integer division here because we can only form a whole pair of numbers; any extra would remain unpaired.
- We sum all these whole pairs to get the total number of pairs formed.
- To find out the number of leftover integers, we can sum the contributions of the leftover from each number by considering `frequency of the number % 2` (the remainder of the number of each integer when divided by 2). Since this can be a bit inefficient, we can also use the length of the original `nums` array and subtract twice the number of pairs formed (since each pair is made up of two numbers).

Using the `Counter` and these basic arithmetic operations allows us to arrive at the solution efficiently and accurately.

Solution Approach

The solution utilizes a `Counter` from Python's `collections` module to efficiently count the frequency of each number in the input `nums` array. By doing so, we create a data structure that maps each unique number to the number of times it appears in the array. This mapping is essential for understanding how many pairs we can form.

Once we have the `Counter`, the solution iterates over the values (which represent the count of occurrences of each number). For each count, it calculates the number of whole pairs that can be formed by that specific number using integer division `//`. Integer division by 2 gives us the number of pairs because it takes two identical integers to form one pair, and any excess number cannot form a pair on its own.

The pairs are then summed to get the total number of pairs formed which is represented by `s` in the solution code. This sum `s` is given by the expression `sum(v // 2 for v in cnt.values())`. Here `cnt` is the `Counter` for `nums`, `v` is each count value for the numbers in `cnt`, and the sum is over the number of pairs (integer division of `v` by 2).

After calculating the total sum of pairs, we need to determine the remaining unpaired integers. We can do this by subtracting twice the number of pairs from the total length of `nums`, because for every pair, two elements are removed from the array. This subtraction is done in the return statement `return [s, len(nums) - s * 2]`. The result is a list where the first element is the number of pairs `s`, and the second element is the number of leftover integers `len(nums) - s * 2`.

To summarize, the algorithm can be broken down into the following steps:

- Count the frequency of each unique integer in `nums` using a `Counter`.
- Calculate the number of pairs by summing up the integer division by 2 of each count in the `Counter`.
- Determine the leftover integers by subtracting twice the number of pairs from the original list's length.
- Return the number of pairs and the number of leftover integers as a list.

This approach is efficient as it only requires one pass over the data to create the `Counter`, and then another pass over the unique elements (the keys of the `Counter`) to calculate pairs. This results in a time complexity of O(n) where n is the number of elements in `nums`.

Example Walkthrough

Let's use a small example to illustrate how the solution approach works. Suppose the input array `nums` is `[4, 5, 6, 4, 5, 6, 6]`.

1. First, we create a frequency counter for `nums` using the `Counter` class. This results in the following counts of each number:
    - `4: 2`
    - `5: 2`
    - `6: 3`
2. Next, we iterate over these counts to determine the number of pairs we can form:
    - For number `4`, count is `2`. Number of pairs formed `2 // 2 = 1`.
    - For number `5`, count is `2`. Number of pairs formed `2 // 2 = 1`.
    - For number `6`, count is `3`. Number of pairs formed `3 // 2 = 1` (with `1` leftover).
3. We sum up the number of pairs. So, `1 (from 4) + 1 (from 5) + 1 (from 6) = 3`. Thus, the total number of pairs `s` is `3`.
4. We then need to find the number of leftover integers. There are `7` original numbers in `nums`, and after forming `3` pairs (which includes `6` numbers), we subtract `6` from `7`:
    - Leftover integers: `7 - 3 * 2 = 1`.
5. The result is returned as a list where the first element is the number of pairs, and the second element is the number of leftover integers. For this example, it would be `[3, 1]`.

In conclusion, the solution approach correctly determines that from the input array `[4, 5, 6, 4, 5, 6, 6]`, we can form `3` pairs and we are left with `1` unpaired integer. The final answer is `[3, 1]`.

Python Solution

```python
from collections import Counter

class Solution:
    def numberOfPairs(self, nums: List[int]) -> List[int]:
        # Create a counter for all elements in the nums list
        num_counter = Counter(nums)

        # Calculate pairs by summing the integer division of counts by 2
        # This gives us the number of pairs for each unique number
        pairs = sum(count // 2 for count in num_counter.values())

        # Return the number of pairs and the number of leftover elements
        # Leftovers are calculated by subtracting twice the number of pairs from the total number of elements
        return [pairs, len(nums) - pairs * 2]

# Comment:
# The `numberOfPairs` method takes a list of integers (`nums`) and returns a list
# where the first element is the number of pairs that can be formed and the second
# element is the number of elements that cannot be paired.

```

Java Solution

```java
class Solution {

    public int[] numberOfPairs(int[] nums) {
        // Create an array to count occurrences of each number, assuming the given range is 0-100
        int[] count = new int[101];

        // Count the number of times each number appears in the input array
        for (int num : nums) {
            ++count[num]; // Increment the count for the current number
        }

        // Initialize a variable to keep track of the total number of pairs
        int totalPairs = 0;

        // Calculate the number of pairs for each number and add to the totalPairs
        for (int occurrences : count) {
            totalPairs += occurrences / 2; // A pair is two of the same number, hence we divide by 2
        }

        // The total number of leftovers is the original array size minus twice the number of pairs
        int leftovers = nums.length - totalPairs * 2;

        // Return an array containing the total number of pairs and the leftovers
        return new int[] {totalPairs, leftovers};
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find the number of pairs that can be made in the vector nums
    vector<int> numberOfPairs(vector<int>& nums) {
        // Initialize a vector to hold the count of each number up to 100
        vector<int> count(101, 0);

        // Increment the count for each number in the input vector
        for (int num : nums) {
            ++count[num];
        }

        // Initialize the number of pairs to zero
        int pairs = 0;

        // Count the number of pairs for each count value
        for (int frequency : count) {
            pairs += frequency / 2; // Each pair requires two of the same number
        }

        // Calculate the number of leftover elements by subtracting the
        // number of elements forming pairs (pairs * 2) from the total number of elements
        int leftovers = static_cast<int>(nums.size()) - pairs * 2;

        // Return the number of pairs and leftovers as a vector of two elements
        return {pairs, leftovers};
    }
};

```

Typescript Solution

```typescript
// Function takes an array of numbers and returns the number of pairs
// along with the number of leftover elements after pairing them.
function numberOfPairs(nums: number[]): number[] {
    // Get the length of the array.
    const length0fNums = nums.length;
    // Initialize an array to keep a count of each number (up to 100 as per the constraint).
    // If the constraints change, this upper limit should be updated accordingly.
    const counts = new Array(101).fill(0);
    // Iterate through the `nums` array and count occurrences of each number.
    for (const num of nums) {
        counts[num]++;
    }
    // Calculate the sum of pairs by adding half the count of each number
    // (using bitwise right shift to quickly divide by 2).
    const totalPairs = counts.reduce((runningTotal, currentCount) => runningTotal + (currentCount >> 1), 0);
    // Calculate the remaining number of elements after forming pairs.
    const remainingElements = length0fNums - totalPairs * 2;
    // Return the total number of pairs and the remaining elements.
    return [totalPairs, remainingElements];
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function primarily depends on two operations: the construction of the counter `cnt` and the subsequent iteration over its values to sum the number of pairs.

1. Constructing the counter `cnt` from `nums` has a time complexity of `O(n)`, where `n` is the length of the input list `nums`. This is because `Counter` goes through each element in the list once.
2. Iterating over the values of the counter `cnt` to calculate the sum takes `O(k)`, where `k` is the number of unique elements in `nums`. In the worst case, if all elements are distinct, `k` is equal to `n`.

Therefore, the overall time complexity of the function is `O(n + k)`. However, since `k <= n`, we can simplify this to `O(n)`.

Space Complexity

The space complexity is determined by the space required to store the counter `cnt`.

1. The counter `cnt` can store at most `k` key-value pairs, where `k` is the number of unique elements in `nums`. In the worst case, the space complexity would thus be `O(k)`.
2. The list `nums` itself is not modified, and no additional space that is dependent on `n` is used beyond the counter `cnt`.

Assuming the unique elements in `nums` are bounded by `n`, the space complexity can be simplified to `O(n)`, which accounts for the space required to store the counter for each unique element in the list.