# 2404. Most Frequent Even Element

Easy Array Hash Table Counting Leetcode Link

## Problem Description

The problem requires us to find the most frequently occurring even element in an array of integers. The steps are as follows:

1. Traverse through the list of integers.
2. Count the frequency of each even integer.
3. Determine the even integer that appears most frequently.

If there is a tie (more than one even number with the highest frequency), we should return the smaller of those numbers. If the array does not contain any even numbers, we should return -1. These specifics need to be taken into account while implementing the solution.

## Intuition

To solve this problem, we need to think about how to efficiently count the occurrences of each even number and then determine which one is the most frequent or the smallest among the most frequent. The steps include:

1. Filter out all the even numbers.
2. Count the occurrences of each even number.
3. Track the most frequent even number.
4. Handle the tie-breaking condition by comparing the current number with the most frequent one found so far, updating with the smaller number if their frequencies are the same.
5. Return -1 if there are no even numbers.

The `Counter` from Python's `collections` module is very handy for counting frequencies, and using a conditional generator expression `(x for x in nums if x % 2 == 0)` allows us to filter even numbers directly within the `Counter`'s constructor. Then, iterating over the items in the `Counter`, we can apply our logic to track the correct answer based on the frequency (`v`) and the number itself (`x`). We use two variables: `ans` to hold the current answer (initialized to -1) and `mx` to hold the maximum frequency seen so far (initialized to 0). The loop updates these variables as needed according to the problem rules.

## Solution Approach

The provided solution leverages the `Counter` class from Python's `collections` module to count the frequency of the even numbers. Here is how the solution is implemented:

1. **Filtering Even Numbers**: A generator expression is used to filter out even numbers from the `nums` list: `(x for x in nums if x % 2 == 0)`. This ensures that the `Counter` only processes even numbers.

2. **Counting Frequencies**: The `Counter` is populated with the filtered even numbers, resulting in a dictionary-like object that maps each even number to its count of occurrences: `Counter(x for x in nums if x % 2 == 0)`.

3. **Iterating Through Counts**: The solution iterates over the items of the `Counter` with `for x, v in cnt.items():`, where `x` is the number and `v` is its count (frequency).

4. **Finding the Most Frequent and Smallest**: The variables `ans` and `mx` are maintained to keep track of the most frequent even number and its corresponding count. During iteration, if the current count `v` exceeds the maximum count `mx`, or if the count equals `mx` but the current number `x` is smaller than `ans`, then `ans` is updated to `x` and `mx` is updated to `v`.

5. **Result**: The value of `ans` at the end of the iteration is the desired result. If no even numbers were found, `ans` remains `-1`, which is the required output for this case.

The algorithm is efficient because Counters are designed to provide constant time complexity, `O(1)`, for element counting, and the additional loop to find the most frequent even number runs in linear time, `O(n)`, where `n` is the size of the filtered list of even numbers. Thus, the overall time complexity of the algorithm is `O(n)`, where `n` is the length of the input list.

The pattern used here is quite common in problems related to frequency counts where tie-breaking rules apply. By carefully designing the conditionals within the iteration loop, the algorithm efficiently solves the problem without needing to sort the input or use additional data structures.

### Example Walkthrough

Let's consider an array of integers `nums` with the following elements: `[3, 5, 8, 3, 10, 8, 8, 10]`.

Following the solution approach:

1. **Filtering Even Numbers**: Firstly, we filter out the even numbers using the expression `(x for x in nums if x % 2 == 0)` which produces a generator for the sequence `[8, 10, 8, 8, 10]`.

2. **Counting Frequencies**: We use the `Counter` from the `collections` module on the filtered sequence to get a dictionary-like object representing the frequencies: `{8: 3, 10: 2}`. This indicates that the number 8 occurs three times and the number 10 occurs two times.

3. **Iterating Through Counts**: We then iterate through this dictionary. The loop checks both the number `x` and its frequency `v`.

4. **Finding the Most Frequent and Smallest**: To determine the most frequent and smallest even number, the variables `ans` and `mx` are set initially at `-1` and `0`, respectively. During the iteration:

    - For the first item `8:3`, `v` is 3, which is greater than `mx`, so `mx` is updated to 3, and `ans` is updated to 8.
    - Next, for `10:2`, `v` is 2, which is less than `mx`. So there is no update, and `ans` remains 8.

5. **Result**: At the end of the iteration, the value of `ans` is 8, which is the most frequent even number in our array `nums`. If there were no even numbers, `ans` would have remained `-1`.

This illustrates how the provided solution approach efficiently finds the most frequent even number or the smallest such number in the case of a tie.

## Python Solution

```python
from collections import Counter

class Solution:
    def mostFrequentEven(self, nums: List[int]) -> int:
        # Create a frequency counter for even numbers only
        even_count = Counter([x for x in nums if x % 2 == 0])

        # Initialize variables for the most frequent even number and its count
        # Set answer to -1 initially, which means there's no even number in the list
        most_frequent_even = -1
        highest_frequency = 0

        # Iterate over the frequency counter's items
        for number, frequency in even_count.items():
            # If a number has a higher frequency than the current or the same frequency but smaller in value
            if frequency > highest_frequency or (frequency == highest_frequency and number < most_frequent_even):
                # Update the most frequent number and the highest frequency
                most_frequent_even = number
                highest_frequency = frequency

        # Return the most frequent even number
        return most_frequent_even

# Note: The List type hint should be imported from the typing module for complete correctness,
# which depends on the Python version. If you are using Python 3.9 or newer, List type hints
# are built into the Python standard library. For Python 3.8 or earlier, use `from typing import List`.

```

## Java Solution

```java
class Solution {
    public int mostFrequentEven(int[] nums) {
        // Create a HashMap to keep track of the count of even numbers
        Map<Integer, Integer> countMap = new HashMap<>();

        // Iterate over all the elements in the array
        for (int num : nums) {
            // Check if the current element is even
            if (num % 2 == 0) {
                // If it is even, increment its count in the HashMap
                // Using merge function to handle both existing and new numbers
                countMap.merge(num, 1, Integer::sum);
            }
        }

        // Variable to keep track of the most frequent even number
        int mostFrequentNum = -1;
        // Variable to keep track of the maximum frequency
        int maxFrequency = 0;

        // Iterate over the entries in the HashMap
        for (var entry : countMap.entrySet()) {
            int number = entry.getKey();
            int frequency = entry.getValue();
            // Check if the current frequency is greater than the maxFrequency found so far
            // or if the frequency is same as maxFrequency and number is smaller than current mostFrequentNum
            if (maxFrequency < frequency || (maxFrequency == frequency && mostFrequentNum > number)) {
                // Update the most frequent number and the maximum frequency
                mostFrequentNum = number;
                maxFrequency = frequency;
            }
        }

        // Return either the most frequent even number or -1 if no such number was found
        return mostFrequentNum;
    }
}

```

## C++ Solution

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

class Solution {
public:
    int mostFrequentEven(vector<int>& nums) {
        unordered_map<int, int> frequencyMap; // Map to store the frequency of each even number
        for (int num : nums) { // Iterate through the array
            if (num % 2 == 0) { // If the element is even
                ++frequencyMap[num]; // Increment the frequency count for this number
            }
        }

        int mostFrequent = -1; // Variable to store the most frequent even number. Initialized to -1.
        int maxFrequency = 0; // Variable to store the maximum frequency of an even number

        // Iterate through the frequency map to find the most frequent even number
        for (auto& keyValue : frequencyMap) {
            int number = keyValue.first; // Current number
            int frequency = keyValue.second; // Frequency of the current number

            // Update mostFrequent if the current frequency is greater than maxFrequency
            // or if the frequency is the same but the number is smaller
            if (maxFrequency < frequency || (maxFrequency == frequency && mostFrequent > number)) {
                mostFrequent = number;
                maxFrequency = frequency; // Update the max frequency
            }
        }
        return mostFrequent; // Return the most frequent even number, or -1 if none exists
    }
};

```

## Typescript Solution

```typescript
// Finds the most frequent even element in an array of numbers.
// If multiple elements are equally frequent, returns the smallest one.
// If there are no even numbers, returns -1.
function mostFrequentEven(nums: number[]): number {
    // Map to store the count of even numbers.
    const countMap: Map<number, number> = new Map();

    // Iterate over the array and count the even numbers.
    for (const num of nums) {
        if (num % 2 === 0) {
            countMap.set(num, (countMap.get(num) ?? 0) + 1);
        }
    }

    // Variable to keep track of the most frequent even number.
    let answer: number = -1;
    // Variable to keep track of the highest occurrence count found so far.
    let maxCount: number = 0;

    // Iterate over the countMap to find the most frequent even number.
    for (const [number, count] of countMap) {
        // If the count for the current number is greater than maxCount,
        // or if the count is equal but the number is smaller than the current answer,
        // update answer and maxCount.
        if (maxCount < count || (maxCount === count && answer > number)) {
            answer = number;
            maxCount = count;
        }
    }

    // Return the most frequent even number or -1 if there are no even numbers.
    return answer;
}

```

## Time and Space Complexity

The given code snippet is a Python function that finds the most frequent even element in a list. To compute the time complexity and space complexity, let's analyse the given operations:

1. A `Counter` object is constructed with a generator expression: `Counter(x for x in nums if x % 2 == 0)`. This step goes through all `n` elements of `nums`, checking for `x % 2 == 0` to consider only even numbers. Therefore, this step has a time complexity of `O(n)`.

2. The `ans` and `mx` variables are initialized. This is a constant time operation, so its time complexity is `O(1)`.

3. A `for` loop iterates over each item in the `Counter` object: `for x, v in cnt.items()`. This could be up to `n` iterations in the worst case (if all numbers in `nums` are even and unique). However, since the total number of unique elements in the `Counter` object is unlikely to be `n`, let's assume there are `k` unique even numbers after filtering. The time complexity for this loop is `O(k)`.

4. Inside the loop, there are a few constant time comparisons, and possibly an assignment operation. These constant time operations inside the loop do not affect the overall time complexity of `O(k)` for the loop.

Therefore, the total time complexity is the sum of all these steps, which would be `O(n)` + `O(k)`. Since `k` is bounded by `n`, the worst-case time complexity simplifies to `O(n)`.

As for space complexity:

1. The `Counter` object will hold at most `k` unique even numbers, which requires `O(k)` space.

2. The `ans` and `mx` variables are constant space, `O(1)`.

The total space complexity is `O(k)`. Since `k` is bounded by `n`, the worst-case space complexity is `O(n)`.

In conclusion:

- **Time Complexity**: `O(n)`
- **Space Complexity**: `O(n)`