Approach 1: Brute Force

Idea:

Count the frequency of each element in the array and return the element that has a count greater than n/2.

Algorithm:

1. Iterate over the array and for each element, count its occurrences in the array.
2. If an element's count is greater than n/2, return that element.

Complexity:

- **Time Complexity:** O(n^2) - For each element, you are counting its occurrences in the entire array.
- **Space Complexity:** O(1) - No extra space used.

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int n = nums.size();
        for(int i = 0; i < n; i++) {
            int count = 0;
            for(int j = 0; j < n; j++) {
                if(nums[j] == nums[i])
                    count++;
            }
            if(count > n / 2)
                return nums[i];
        }
        return -1; // This line won't be reached as the majority element always exists.
    }
};
```

Approach 2: Better Approach (HashMap)

Idea:

Use a HashMap (or unordered_map in C++) to count the frequency of each element. Return the element that has a frequency greater than n/2.

Algorithm:

1. Traverse the array and store the frequency of each element in a hash map.
2. Iterate through the hash map and return the element with a frequency greater than n/2.

Complexity:

- **Time Complexity:** O(n) - You are traversing the array twice.
- **Space Complexity:** O(n) - Hash map to store the frequency of elements.

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        unordered_map<int, int> countMap;
        int n = nums.size();

        for(int num : nums) {
            countMap[num]++;
        }

        for(auto &entry : countMap) {
            if(entry.second > n / 2) {
                return entry.first;
            }
        }

        return -1; // This line won't be reached as the majority element always exists.
    }
};
```

Approach 3: Best Approach (Boyer-Moore Voting Algorithm)

Idea:

Use the Boyer-Moore Voting Algorithm to find the majority element. The algorithm works by maintaining a count and a candidate. The count is incremented when we encounter the same element and decremented when we encounter a different one. If the count drops to zero, we choose a new candidate.

Algorithm:

1. Initialize count to 0 and candidate to None.
2. Traverse the array:
    - If count is 0, set the current element as the candidate.
    - If the current element is the same as candidate, increment count.
    - Otherwise, decrement count.
3. Return candidate as the majority element.

Complexity:

- **Time Complexity:** O(n) - Single pass through the array.

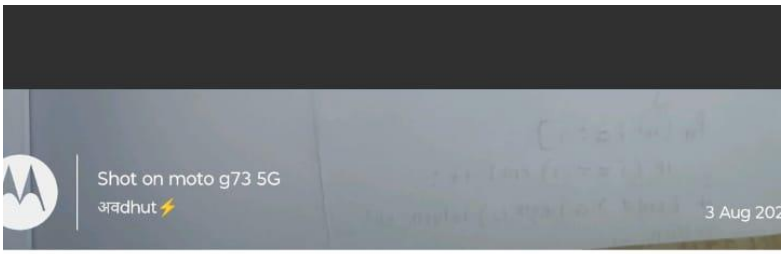
- **Space Complexity:** O(1) - No extra space used.

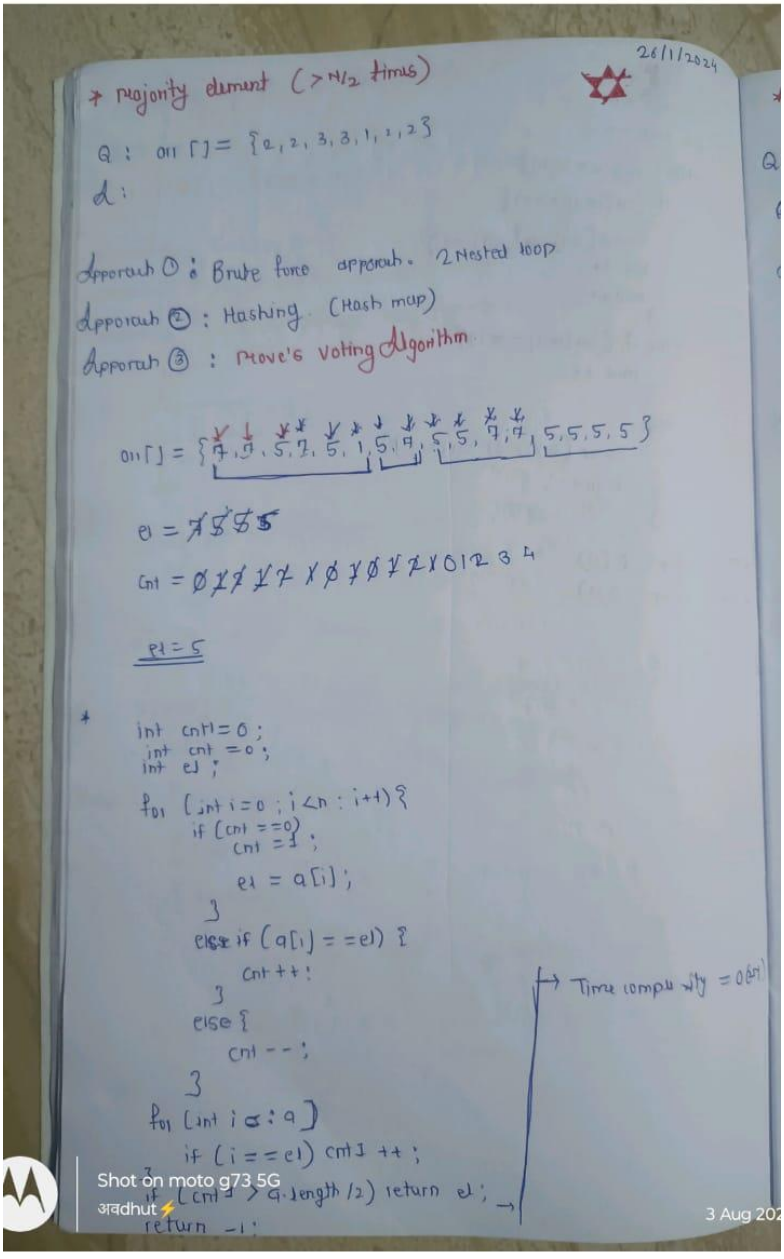
```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int count = 0;
        int candidate = 0;

        for(int num : nums) {
            if(count == 0) {
                candidate = num;  // Choose a new candidate.
            }

            // Adjust the count based on whether the current number supports the
candidate.
            if(num == candidate) {
                count = count + 1;  // Increment count.
            } else {
                count = count - 1;  // Decrement count.
            }
        }

        return candidate;  // The candidate is the majority element.
    }
};
```

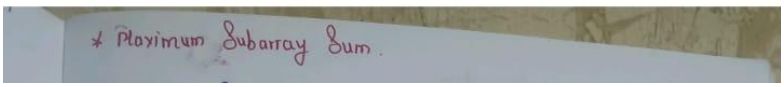
Explanation of Boyer-Moore Voting Algorithm:

- The key idea is that if a candidate is the majority element, it will remain as the candidate even after balancing out the votes.
- When count becomes 0, it means the candidate has an equal number of elements against it, so we choose a new candidate.
- By the end of the array, the candidate is guaranteed to be the majority element since the majority element always exists.

This approach is optimal in both time and space complexity.


Shot on moto g73 5G
अवधुत⚡
3 Aug 202


* majority element (>N/2 times)
26/1/2024
Q : arr[] = {2, 2, 3, 3, 1, 2, 2}
A :
Approach ① : Brute force approach. 2 Nested loop
Approach ② : Hashing (Hash map)
Approach ③ : Moore's voting Algorithm
arr[] = {7, 7, 5, 7, 5, 1, 5, 7, 5, 5, 7, 7, 5, 5, 5, 5}
el = 7 7 7 5
cnt = 0 1 0 1 0 1 0 1 2 1 0 1 2 3 4
el = 5
int cnt1 = 0;
int cnt = 0;
int el;
for (int i = 0; i < n; i++) {
if (cnt == 0)
cnt = 1;
el = a[i];
}
else if (a[i] == el) {
cnt++;
}
else {
cnt--;
}
for (int i : a)
if (i == el) cnt1 ++;
if (cnt1 > a.length/2) return el;
return -1;
→ Time complexity = O(n)
Shot on moto g73 5G
अवधुत⚡
3 Aug 202


* Maximum Subarray Sum.