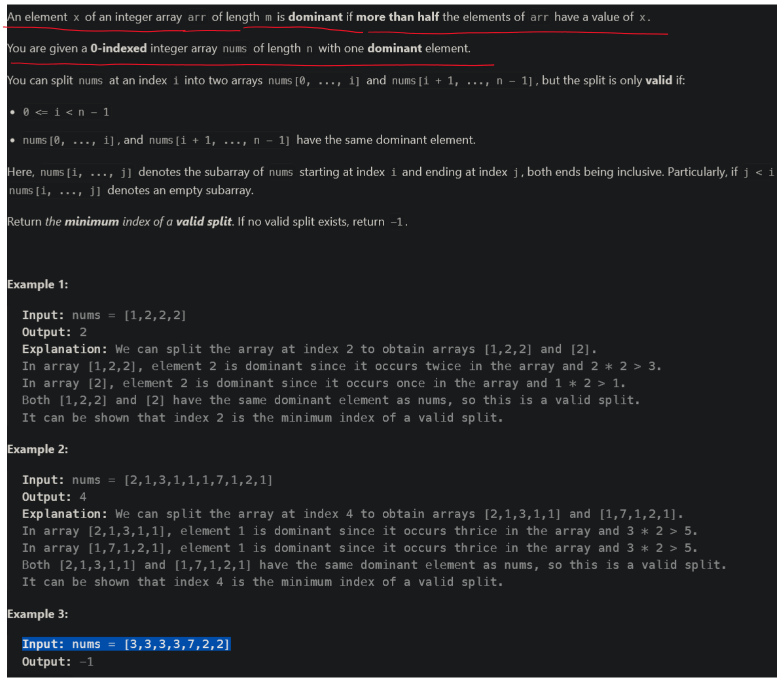
Minimum Index of a Valid split :-



An element `x` of an integer array `arr` of length `m` is **dominant** if **more than half** the elements of `arr` have a value of `x`.

You are given a **0-indexed** integer array `nums` of length `n` with one **dominant** element.

You can split `nums` at an index `i` into two arrays `nums[0, ..., i]` and `nums[i + 1, ..., n - 1]`, but the split is only **valid** if:

- `0 <= i < n - 1`
- `nums[0, ..., i]`, and `nums[i + 1, ..., n - 1]` have the same dominant element.

Here, `nums[i, ..., j]` denotes the subarray of `nums` starting at index `i` and ending at index `j`, both ends being inclusive. Particularly, if `j < i` `nums[i, ..., j]` denotes an empty subarray.

Return *the **minimum** index of a **valid split***. If no valid split exists, return `-1`.

Example 1:

```
Input: nums = [1,2,2,2]
Output: 2
Explanation: We can split the array at index 2 to obtain arrays [1,2,2] and [2].
In array [1,2,2], element 2 is dominant since it occurs twice in the array and 2 * 2 > 3.
In array [2], element 2 is dominant since it occurs once in the array and 1 * 2 > 1.
Both [1,2,2] and [2] have the same dominant element as nums, so this is a valid split.
It can be shown that index 2 is the minimum index of a valid split.
```

Example 2:

```
Input: nums = [2,1,3,1,1,1,7,1,2,1]
Output: 4
Explanation: We can split the array at index 4 to obtain arrays [2,1,3,1,1] and [1,7,1,2,1].
In array [2,1,3,1,1], element 1 is dominant since it occurs thrice in the array and 3 * 2 > 5.
In array [1,7,1,2,1], element 1 is dominant since it occurs thrice in the array and 3 * 2 > 5.
Both [2,1,3,1,1] and [1,7,1,2,1] have the same dominant element as nums, so this is a valid split.
It can be shown that index 4 is the minimum index of a valid split.
```

Example 3:

```
Input: nums = [3,3,3,3,7,2,2]
Output: -1
```

eg: i/p: nums = [1, 2, 2 | 2]

o/p: 2

1 2 3

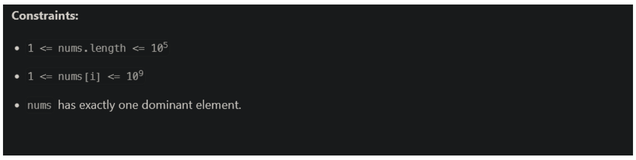
de(nums) = x

de(n_{0-j}, $n_{j+1}-n$) = x

find min j

$x \geq \frac{j}{2}$ also rem $(x) \geq \left(\frac{n-j}{2}\right)$ rem/2

also $x \geq n/2$



Constraints:

- `1 <= nums.length <= 10^5`
- `1 <= nums[i] <= 10^9`
- `nums` has exactly one dominant element.

Approach 1 :- By Sorting :-

```
class Solution {
 public:
     int minimumIndex(vector<int> &nums) {
        int n = nums.size();
         vector<int> copy = nums;
         sort ( copy.begin() , copy.end());
         int key = -1, val = INT_MIN;
        for(int i= 0; i<n; i++){
            int j= i+1;
             int cnt = 1;
             while (j<n && copy[j]== copy[i]){
                        cnt ++;
                          j++;
                        }
                if(cnt  > val) {
                      key = copy[i];
                         = cnt; }
```

```
      if(cnt
            key = copy[i];
            val = cnt; }
              i=j-1;

        }
     if(val <= n/2) return -1;
     int v = 0;

     for (int  i= 0; i<n; i++){
              if(nums[i]==key){
                        v++;
                  }

        int  rem1 = n-i-1 , rem2 = val-v;
        if(v > (i+1)/2  && rem2 > (rem1/2))return i;

      }
    return -1;
  }};
```

T.C. O(NlogN)

S.C. O(1).

Approach 2:- Hash Map

Algorithm:-

- Initialize:
  - n to the size of nums.
  - first Map & secondMap as hashMaps to track the numbers in the first & second half of the split, respectively.
- Iterate through nums, adding each element to secondMap.
- Iterate through nums again. For each number, num at index:
  - Decrement second Map[num] by 1.
  - Increment first Map[num] by 1.
  - If firstMap[num] *2 > index + 1 and second Map [num] * 2 > n-index-1, return index, since num is the dominant element in both halves of th

index, since num is the dominant element in both halves of the current split.

- Return -1, indicating that no valid split was found.

```
class Solution{
public:
  int minimumIndex(vector<int> & nums){
   unordered_map <int ,int > firstMap, secondMap;
   int n = nums.size();

 //add all elements of nums to second half
   for (auto & num : nums){
        second Map[num]++; }

 for(int index = 0; index<n ; index++){

 //create split at current index
    int  num = nums[index];
    secondMap[num]--;
    firstMap[num]++;

  //check if valid split
 if(firstMap[num]*2  >index+1  && secondMap[num]*2 >
     n-index -1) return index;}}
 // no valid split exists
 return -1;  }};
```

T.C. O(n)

S.C. O(n)

Approach 3:- Boyer - Moore Majority Voting Algorithm :-

Intuition :-

In the previous approach, we used hashmaps to keep track of element frequencies in each split, but this required extra space proportional to the size of nums. Since maintaining these frequency maps can be costly in terms of memory, we need a way to determine the dominant element without storing counts for every possible number.

Boyer-Moore Majority Voting Algorithm efficiently finds a ... linear time without using extra coun [illegible] 0

Boyer-Moore Majority Voting Algorithm efficiently finds a majority element (if it exists) in linear time without using extra space. The key observation behind it is that if an element appears more than n/2 times, then it must remain after canceling out other elements. By iterating through nums while maintaining a candidate element & a counter, we can determine the element x that appears the most.

```
class Solution{
public:
    int minimumIndex(vector<int>& nums){
     // Find the majority element
int x= nums[0] , count = 0, xcount = 0 , n= nums.size();
for(auto &num: nums){
        if(num == x) count ++;
          else count --;
        if(count == 0) {
                x = num;
                count = 1;
            }
        }

// Count frequency of majority element
for(auto &num: nums){
        if(num == x){
             xcount ++;
          }
        }

// Check if valid split is possible.
count = 0;
for(int index=0; index<n; index++){
     if(nums[index]== x){
            count ++; }
  int remainingCount = xCount - count;
  if(count * 2> index+1  && remainingCount * 2 > n-index-1){
           return index;
      }
    }
  return -1;
  }
};
```

T. C - $O(N)$

S - C. $O(1)$