April 1, 2022

BOYER MOORE MAJORITY VOTE ALGORITHM

Nikhil Kumar Singh
Vrishchik

DURATION
9min

CATEGORIES

Competitive Programming

Search

TAGS

C++

The Boyer-Moore voting method is one of the most often used optimum algorithms for determining the majority element among elements with more than N/2 occurrences. This works wonderfully for finding the majority element, which requires two traversals over the provided items and is O(N) time and O(1) space complexity.

This method is often asked about in interviews with companies like Google, Facebook, Microsoft, Salesforce, and Amazon.

FINDING THE MAJORITY ELEMENT PROBLEM

In this task, we must discover the number in the supplied array with a frequency more than half the array's length, i.e. N/2 times. The length of the array is denoted by N.

Example 1:
Input:{4,7,4,4,3,4,6,4}
Output: 4
Explanation: The length of the array is 8 and the frequency of 2 is 5 > 8/2

Example 2:
Input:{4,7,0,5,3,4,6,4}
Output: -1
Explanation: No majority element

APPROACH #1: BRUTE FORCE

Use nested for loops to solve the majority element problem, counting each element and if count>n/2 for each element. However, this method is inefficient since it requires O(n2) time.

Code:

```
int findMajorityElement(vector < int > & nums) {

int n = nums.size(), count;
for (int i = 0; n && i <= n / 2; i++) {

  count = 1;
  for (int j = i + 1; j < n; j++)
    if (nums[j] == nums[i])
      count++;

  if (count > n / 2)
    return nums[i];
}

return -1;
}
```

Time complexity: O(n2), nested loops
Space complexity: O(1) only two variables

APPROACH #2: MAP APPROACH

Instead of checking the frequency for every number in the array we just store the frequency of each number in the map and then check if any number has the required frequency.

Code:

```
int findMajorityElement(vector < int > & nums) {

unordered_map < int, int > mp;

for (auto x: nums)
  mp[x]++;
```

```
    for (auto x: mp)
      if (x.second > n / 2)
        return x.first;

    return -1;
  }

```

Time complexity: O(n), we make only two iterations through the array.
Space complexity: O(n), sorting frequency of each number.

APPROACH #3: BOYER-MOORE VOTING ALGORITHM

In its most basic form, the algorithm seeks out a majority element if one exists. A majority element is one that appears more than half of the time in the input elements. However, if there is no majority, the algorithm will not recognize this and will continue to output one of the items.

The algorithm is divided into two parts. A first pass identifies an element as a majority, and a second pass confirms that the element identified in the first pass is indeed a majority.

The method will not identify the majority element if it does not exist, and will thus return an arbitrary element.

ALGORITHM:

1. Begin by initializing two variables, num and a counter count, both of which are set to 0.
2. Now we'll begin iterating over the number and for each element x.
3. We first check to see if the count is 0, and then we assign num to x.
4. Then we check to see if the current num is equal to x, if not, we decrease the count by one, else we increment it by one.
5. Reset the counter to zero.
6. Find the frequency of the num variable by looping through the nums array.
7. Now, if the count is larger than N/2, we return num; otherwise we return -1.

ILLUSTRATION:

Input: { 1,7,2,7,8,7,7}

Index 0: num = 1, count =1
Index 1: num = 1, count =0 (7 not equal to 1)

Index 2: num = 2 (as the count is 0 we initalise num to current), count =1

Index 3: num = 2, count =0 (7 not equal to 2)

Index 4: num = 8 (as the count is 0 we initalise num to current), count =1

Index 5: num = 8, count =0 (7 not equal to 8)

Index 6: num = 7 (as the count is 0 we initalise num to current), count =1

Now we can check for the frequency of 7, i.e, 4 which is > 7/2.

Code:

```
int findMajority(vector < int > nums) {

 int n = nums.size();
 int num, count = 0;
 for (auto x: nums) {
  if (count == 0)
   num = x;

  count += (x == num ? 1 : -1);
 }

 count = 0;
 for (auto x: nums)
  if (x == num)
   count++;

 if (count > n / 2)
  return num;

 return -1;
}
```

Time complexity: O(n), we make only two iterations through the array.

Space complexity: O(1), only two variables.