



169. Majority Element

Notes

 Description ([\(/problems/majority-element/description/\)](/problems/majority-element/description/)) Hints ([\(/problems/majority-element/hints/\)](/problems/majority-element/hints/))Submissions ([\(/problems/majority-element/submissions/\)](/problems/majority-element/submissions/))

Quick Navigation ▼

View in Article  ([\(/articles/majority-element/\)](/articles/majority-element/))

Approach #1 Brute Force [Time Limit Exceeded]

Intuition

We can exhaust the search space in quadratic time by checking whether each element is the majority element.

Algorithm

The brute force algorithm iterates over the array, and then iterates again for each number to count its occurrences. As soon as a number is found to have appeared more than any other can possibly have appeared, return it.

Java

Python3

 Copy

```
1 class Solution {
2     public int majorityElement(int[] nums) {
3         int majorityCount = nums.length/2;
4
5         for (int num : nums) {
6             int count = 0;
7             for (int elem : nums) {
8                 if (elem == num) {
9                     count += 1;
10                }
11            }
12
13            if (count > majorityCount) {
14                return num;
15            }
16        }
17
18        return -1;
19    }
20 }
21 }
```

 Notes

Complexity Analysis

- Time complexity : $\mathcal{O}(n^2)$

The brute force algorithm contains two nested for loops that each run for n iterations, adding up to quadratic time complexity.

- Space complexity : $\mathcal{O}(1)$

The brute force solution does not allocate additional space proportional to the input size.

Approach #2 HashMap [Accepted]

Intuition

We know that the majority element occurs more than $\lfloor \frac{n}{2} \rfloor$ times, and a HashMap allows us to count element occurrences efficiently.

Algorithm

We can use a HashMap that maps elements to counts in order to count occurrences in linear time by looping over `nums`. Then, we simply return the key with maximum value.

Java

Python3

 Copy

```

1  class Solution {
2      private Map<Integer, Integer> countNums(int[] nums) {
3          Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
4          for (int num : nums) {
5              if (!counts.containsKey(num)) {
6                  counts.put(num, 1);
7              }
8              else {
9                  counts.put(num, counts.get(num)+1);
10             }
11         }
12         return counts;
13     }
14
15     public int majorityElement(int[] nums) {
16         Map<Integer, Integer> counts = countNums(nums);
17
18         Map.Entry<Integer, Integer> majorityEntry = null;
19         for (Map.Entry<Integer, Integer> entry : counts.entrySet()) {
20             if (majorityEntry == null || entry.getValue() > majorityEntry.getValue()) {
21                 majorityEntry = entry;
22             }
23         }
24
25         return majorityEntry.getKey();
26     }
27 }

```

 Notes

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

We iterate over `nums` once and make a constant time `HashMap` insertion on each iteration. Therefore, the algorithm runs in $\mathcal{O}(n)$ time.

- Space complexity : $\mathcal{O}(n)$

At most, the `HashMap` can contain $n - \lfloor \frac{n}{2} \rfloor$ associations, so it occupies $\mathcal{O}(n)$ space. This is because an arbitrary array of length n can contain n distinct values, but `nums` is guaranteed to contain a majority element, which will occupy (at minimum) $\lfloor \frac{n}{2} \rfloor + 1$ array indices.

Therefore, $n - (\lfloor \frac{n}{2} \rfloor + 1)$ indices can be occupied by distinct, non-majority elements (plus 1 for the majority element itself), leaving us with (at most) $n - \lfloor \frac{n}{2} \rfloor$ distinct elements.

Approach #3 Sorting [Accepted]

Intuition

If the elements are sorted in monotonically increasing (or decreasing) order, the majority element can be found at index $\lfloor \frac{n}{2} \rfloor$ (and $\lfloor \frac{n}{2} \rfloor + 1$, incidentally, if n is even).

Algorithm

For this algorithm, we simply do exactly what is described: sort `nums`, and return the element in question. To see why this will always return the majority element (given that the array has one), consider the figure below (the top example is for an odd-length array and the bottom is for an even-length array):

Notes

{0, 1, 2, 3, 4, 5, 6}

{0, 1, 2, 3, 4, 5}

For each example, the line below the array denotes the range of indices that are covered by a majority element that happens to be the array minimum. As you might expect, the line above the array is similar, but for the case where the majority element is also the array maximum. In all other cases, this line will lie somewhere between these two, but notice that even in these two most extreme cases, they overlap at index $\lfloor \frac{n}{2} \rfloor$ for both even- and odd-length arrays. Therefore, no matter what value the majority element has in relation to the rest of the array, returning the value at $\lfloor \frac{n}{2} \rfloor$ will never be wrong.

Java

Python3

Copy

```
1 class Solution {
2     public int majorityElement(int[] nums) {
3         Arrays.sort(nums);
4         return nums[nums.length/2];
5     }
6 }
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n \lg n)$

Sorting the array costs $\mathcal{O}(n \lg n)$ time in Python and Java, so it dominates the overall runtime.

- Space complexity : $\mathcal{O}(1)$ or $\mathcal{O}(n)$

We sorted `nums` in place here - if that is not allowed, then we must spend linear additional space on a copy of `nums` and sort the copy instead.

 Notes

Approach #4 Randomization [Accepted?]

Intuition

Because more than $\lfloor \frac{n}{2} \rfloor$ array indices are occupied by the majority element, a random array index is likely to contain the majority element.

Algorithm

Because a given index is likely to have the majority element, we can just select a random index, check whether its value is the majority element, return if it is, and repeat if it is not. The algorithm is verifiably correct because we ensure that the randomly chosen value is the majority element before ever returning.

Java

Python3

 Copy

```

1 class Solution {
2     private int randRange(Random rand, int min, int max) {
3         return rand.nextInt(max - min) + min;
4     }
5
6     private int countOccurrences(int[] nums, int num) {
7         int count = 0;
8         for (int i = 0; i < nums.length; i++) {
9             if (nums[i] == num) {
10                 count++;
11             }
12         }
13         return count;
14     }
15
16     public int majorityElement(int[] nums) {
17         Random rand = new Random();
18
19         int majorityCount = nums.length/2;
20
21         while (true) {
22             int candidate = nums[randRange(rand, 0, nums.length)];
23             if (countOccurrences(nums, candidate) > majorityCount) {
24                 return candidate;
25             }
26         }
27     }

```

 Notes

Complexity Analysis

- Time complexity : $\mathcal{O}(\infty)$

It is technically possible for this algorithm to run indefinitely (if we never manage to randomly select the majority element), so the worst possible runtime is unbounded. However, the expected runtime is far better - linear, in fact. For ease of analysis, convince yourself that because the majority element is guaranteed to occupy *more* than half of the array, the expected number of iterations will be less than it would be if the element we sought occupied exactly *half* of the array. Therefore, we can calculate the expected number of iterations for this modified version of the problem and assert that our version is easier.

$$\begin{aligned}
 EV(itors_{prob}) &\leq EV(itors_{mod}) \\
 &= \lim_{n \rightarrow \infty} \sum_{i=1}^n i \cdot \frac{1}{2^i} \\
 &= 2
 \end{aligned}$$

Because the series converges, the expected number of iterations for the modified problem is constant. Based on an expected-constant number of iterations in which we perform linear work, the expected runtime is linear for the modified problem. Therefore, the expected runtime for our problem is also linear, as the runtime of the modified problem serves as an upper bound for it.

- Space complexity : $\mathcal{O}(1)$

Much like the brute force solution, the randomized approach runs with constant additional space.



Approach #5 Divide and Conquer [Accepted]

Intuition

If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.

Algorithm

Here, we apply a classical divide & conquer approach that recurses on the left and right halves of an array until an answer can be trivially achieved for a length-1 array. Note that because actually passing copies of subarrays costs time and space, we instead pass `lo` and `hi` indices that describe the relevant slice of the overall array. In this case, the majority element for a length-1 slice is trivially its only element, so the recursion stops there. If the current slice is longer than length-1, we must combine the answers for the slice's left and right halves. If they agree on the majority element, then the majority element for the overall slice is obviously the same¹. If they disagree, only one of them can be "right", so we need to count the occurrences of the left and right majority elements to determine which subslice's answer is globally correct. The overall answer for the array is thus the majority element between indices 0 and n .

Java

Python3

 Copy

```

1 class Solution {
2     private int countInRange(int[] nums, int num, int lo, int hi) {
3         int count = 0;
4         for (int i = lo; i <= hi; i++) {
5             if (nums[i] == num) {
6                 count++;
7             }
8         }
9         return count;
10    }
11
12    private int majorityElementRec(int[] nums, int lo, int hi) {
13        // base case; the only element in an array of size 1 is the majority
14        // element.
15        if (lo == hi) {
16            return nums[lo];
17        }
18
19        // recurse on left and right halves of this slice.
20        int mid = (hi-lo)/2 + lo;
21        int left = majorityElementRec(nums, lo, mid);
22        int right = majorityElementRec(nums, mid+1, hi);
23
24        // if the two halves agree on the majority element, return it.
25        if (left == right) {
26            return left;
27        }

```

 Notes

Complexity Analysis

- Time complexity : $\mathcal{O}(n \lg n)$

Each recursive call to `majority_element_rec` performs two recursive calls on subslices of size $\frac{n}{2}$ and two linear scans of length n . Therefore, the time complexity of the divide & conquer approach can be represented by the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + 2n$$

By the master theorem

([https://en.wikipedia.org/wiki/Master_theorem_\(analysis_of_algorithms\)](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms))), the recurrence satisfies case 2, so the complexity can be analyzed as such:

$$\begin{aligned}
 T(n) &= \Theta(n^{\log_b a} \log n) \\
 &= \Theta(n^{\log_2 2} \log n) \\
 &= \Theta(n \log n)
 \end{aligned}$$

- Space complexity : $\mathcal{O}(\lg n)$

Although the divide & conquer does not explicitly allocate any additional memory, it uses a non-constant amount of additional memory in stack frames due to recursion. Because the algorithm "cuts" the array in half at each level of recursion, it follows that there can only be $\mathcal{O}(\lg n)$ "cuts" before the base case of 1 is reached. It follows from this fact that the resulting recursion tree is balanced, and therefore all paths from the root to a leaf are of length $\mathcal{O}(\lg n)$. Because the recursion tree is traversed in a depth-first manner, the space complexity is therefore equivalent to the length of the longest path, which is, of course, $\mathcal{O}(\lg n)$.

Approach #6 Boyer-Moore Voting Algorithm [Accepted]

Intuition

If we had some way of counting instances of the majority element as $+1$ and instances of any other element as -1 , summing them would make it obvious that the majority element is indeed the majority element.

Algorithm

Essentially, what Boyer-Moore does is look for a suffix *suf* of *nums* where *suf*[0] is the majority element in that suffix. To do this, we maintain a count, which is incremented whenever we see an instance of our current candidate for majority element and decremented whenever we see anything else. Whenever *count* equals 0, we effectively forget about everything in *nums* up to the current index and consider the current number as the candidate for majority element. It is not immediately obvious why we can get away with forgetting prefixes of *nums* - consider the following examples (pipes are inserted to separate runs of nonzero *count*).

[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]

Here, the 7 at index 0 is selected to be the first candidate for majority element. *count* will eventually reach 0 after index 5 is processed, so the 5 at index 6 will be the next candidate. In this case, 7 is the true majority element, so by disregarding this prefix, we are ignoring an equal number of majority and minority elements - therefore, 7 will still be the majority element in the suffix formed by throwing away the first prefix.

[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | **5, 5, 5, 5**]

Now, the majority element is 5 (we changed the last run of the array from 7 s to 5 s), but our first candidate is still 7. In this case, our candidate is not the true majority element, but we still cannot discard more majority elements than minority elements (this would imply that *count* could reach -1

before we reassign `candidate` , which is obviously false).

Therefore, given that it is impossible (in both cases) to discard more majority elements than minority elements, we are safe in discarding the prefix and attempting to recursively solve the majority element problem for the suffix. Eventually, a suffix will be found for which `count` does not hit `0` , and the majority element of that suffix will necessarily be the same as the majority element of the overall array.

Java Python3

Copy Notes

```
1 class Solution {
2     public int majorityElement(int[] nums) {
3         int count = 0;
4         Integer candidate = null;
5
6         for (int num : nums) {
7             if (count == 0) {
8                 candidate = num;
9             }
10            count += (num == candidate) ? 1 : -1;
11        }
12
13        return candidate;
14    }
15 }
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

Boyer-Moore performs constant work exactly n times, so the algorithm runs in linear time.

- Space complexity : $\mathcal{O}(1)$

Boyer-Moore allocates only constant additional memory.

Footnotes

Analysis written by: @emptyset (<https://leetcode.com/emptyset>)

Approaches and time complexities itemized by @ts () and @1337c0d3r ()

1. This is a constant optimization that could be excluded without hurting our overall runtime. ↩

Join the conversation

Signed in as **tan7**.

Post a Reply

nalimov commented last week

iscuss.leetcode.com/user/nalimov
Thank you for explaining Boyer-Moore algorithm, that's very interesting. I was curious is it possible to optimize $O(n)$ time solution to use only $O(1)$ space. Turns out it is possible :)

Notes

pande commented 2 weeks ago

iscuss.leetcode.com/user/pande
Boyce-Moore Algorithms is the one which is worth understanding in this.

__ commented last month

iscuss.leetcode.com/user/__
Python's sort takes $O(n)$ space. It's only sort of in-place.

rohan13 commented 2 months ago

iscuss.leetcode.com/user/rohan13
class Solution {
public int majorityElement(int[] nums) {
if (nums.length == 0)
return -1;

```
int majorElem = nums[0], count = 1;  
  
for (int i=1; i<nums.length; i++) {  
    if (nums[i] == majorElem) {  
        count++;  
    } else {  
        count --;  
    }  
    if (count == 0) {  
        majorElem = nums[i];  
        count = 1;  
    }  
}  
  
if (count > nums.length/2)  
    return majorElem;  
  
count = 0;  
  
for (int i=0; i<nums.length; i++) {  
    if (nums[i] == majorElem) {  
        count++;  
    }  
  
    if (count > nums.length/2)  
        break;  
}  
  
return count > nums.length/2 ? majorElem : -1;  
}  
  
}
```

Notes

uday.joshi88 commented 4 months ago

```
public int majorityElement(int[] nums) {
```

[iscuss.leetcode.com/user/uday-](https://discuss.leetcode.com/user/uday-joshi88)

```
    HashMap<Integer, Integer> ht = new HashMap<>();
    int value = 0;
    for(int i : nums){
        if(ht.containsKey(i)){
            ht.put(i, 1+ht.get(i));
            if(ht.get(i)>nums.length/2){
                value=i;
            }
        }
        else{
            ht.put(i, 1);
        }
    }
    return nums.length==1 ? nums[0] : value;
}
```

Notes

emptyset commented 4 months ago

[@zpng \(https://discuss.leetcode.com/uid/7399\)](https://discuss.leetcode.com/user/emptyset) The problem states that only arrays with a majority element will be passed as input. Your array does not have a majority element, so it is not valid input. For the version of the problem that works with arrays that may or may not have a majority element, you need to make a final linear pass over the array, verifying that the candidate is indeed a majority element.

zpng commented 4 months ago

[iscuss.leetcode.com/user/zpng](https://discuss.leetcode.com/user/zpng)
[7, 7, 5, 7, 5, 1, 5, 7, 5, 5, 7, 7, 6, 6, 6, 6]
approach #6 result is 6 not expected 7, so the approach is wrong!

Ark-kun commented 4 months ago

[iscuss.leetcode.com/user/ark-](https://discuss.leetcode.com/user/ark-kun)
Another easy solution: K being a majority item means that every bit value of K is majority. We can then reconstruct K from the majority bit values.

emptyset commented 5 months ago

@swatisaoji1 (<https://discuss.leetcode.com/uid/37850>) That array does not have a majority element, so it is not a valid input for this problem.

[View original thread \(https://discuss.leetcode.com/topic/110772\)](https://discuss.leetcode.com/topic/110772)

Load more comments...

Notes

Copyright © 2018 LeetCode

[Contact Us \(/support/\)](/support/) | [Frequently Asked Questions \(/faq/\)](/faq/) | [Terms of Service \(/terms/\)](/terms/) | [Privacy Policy \(/privacy/\)](/privacy/)

 [United States \(/region/\)](/region/)