Description | Solution | Discuss (999+) | Submissions

Quick Navigation

Average Rating: 4.41 (274 votes)

Approach 1: Brute Force

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

```
class Solution:
    def majorityElement(self, nums):
        majority_count = len(nums)//2
        for num in nums:
            count = sum(1 for elem in nums if elem == num)
            if count > majority_count:
                return num
```

Complexity Analysis

- Time complexity : $O(n^2)$.

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

- Space complexity : $O(1)$

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

Approach 2: HashMap

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

```
class Solution:
    def majorityElement(self, nums):
        counts = collections.Counter(nums)
        return max(counts.keys(), key=counts.get)
```

Complexity Analysis

- Time complexity : $O(n)$.

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

- Space complexity : $O(n)$

  At most, the `HashMap` size contains $n - \lfloor \frac{n}{2} \rfloor$ associations, so it occupies $O(n)$ 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.

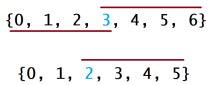
Approach 3: Sorting

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 $\lfloor \frac{n}{2} \rfloor + 1$, tentatively, 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).



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

```
class Solution:
    def majorityElement(self, nums):
        nums.sort()
        return nums[len(nums)//2]
```

Complexity Analysis

- Time complexity : $O(n \log n)$.

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

- Space complexity : $O(\log(n))$ or $O(n)$.

  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.

Approach 4: Randomization

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 it 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

```
import random

class Solution:
    def majorityElement(self, nums):
        majority_count = len(nums)//2
        while True:
            candidate = random.choice(nums)
            if sum(1 for elem in nums if elem == candidate) > majority_count:
                return candidate
```

Complexity Analysis

- Time complexity : $O(n)$.

  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 2 would be if the element we caught 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.

$$
\begin{aligned}
E[\text{iters}_{\text{mod}}] &\leq E[\text{iters}_{\text{mod}}] \\
&= \lim_{n \to \infty} \sum_{i=1}^{n} \frac{1}{2^i} \\
&= 2
\end{aligned}
$$

  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 : $O(1)$.

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

Approach 5: Divide and Conquer

Intuition

Python3

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:

        comp = len(nums)/2

        d1 = collections.Counter(nums)

        for k,v in d1.items() :
            if v > comp :
                return k
```

Notebook | Run Code Result | Debugger

Accepted Runtime: 63 ms

Your Input [3,2,3]

Output 3 | Diff

Expected 3

Console | Use Example Testcases

Run Code | Submit