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

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 same[1]. 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

```
class Solution:
    def majorityElement(self, nums, lo=0, hi=None):
        def majority_element_rec(lo, hi):
            # base case; the only element in an array of size 1 is the majority
            # element.
            if lo == hi:
                return nums[lo]

            # recurse on left and right halves of this slice.
            mid = (hi-lo)//2 + lo
            left = majority_element_rec(lo, mid)
            right = majority_element_rec(mid+1, hi)

            # if the two halves agree on the majority element, return it.
            if left == right:
                return left

            # otherwise, count each element and return the "winner".
            left_count = sum(1 for i in range(lo, hi+1) if nums[i] == left)
            right_count = sum(1 for i in range(lo, hi+1) if nums[i] == right)

            return left if left_count > right_count else right

        return majority_element_rec(0, len(nums)-1)
```

Complexity Analysis

- Time complexity : $O(nlgn)$

  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:

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

  By the master theorem, 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 : $O(lgn)$

  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 $O(lgn)$ "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 $O(lgn)$. 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, $O(lgn)$.

Approach 6: Boyer-Moore Voting Algorithm

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

```
class Solution:
    def majorityElement(self, nums):
        count = 0
        candidate = None

        for num in nums:
            if count == 0:
                candidate = num
            count += (1 if num == candidate else -1)

        return candidate
```

Complexity Analysis

- Time complexity : $O(n)$

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

- Space complexity : $O(1)$

  Boyer-Moore allocates only constant additional memory.

Footnotes

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

Report Article Issue

Retrieve last submitted code

Python3 Autocomplete

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

        comp = len(nums)/2

        di = collections.Counter(nums)

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

Testcase Run Code Result Debugger

Accepted Runtime: 53 ms

Your input [3,2,3]

Output 3 Diff

Expected 3

Console Use Example Testcases

Run Code Submit