169. Majority Element

1107 100

Description (/problems/majority-element/description/) Hints (/problems/majority-element/hints/) Submissions (/problems/majority-element/s

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

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

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` can contain $n - \lfloor \frac{n}{2} \rfloor$ 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.

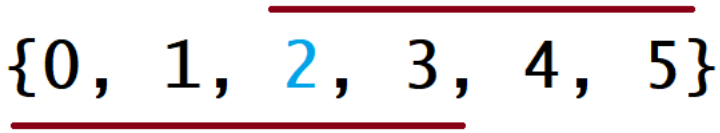
Approach 3: Sorting

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



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 {
    public int majorityElement(int[] nums) {
        Arrays.sort(nums);
        return nums[nums.length/2];
    }
}
```

Complexity Analysis

- Time complexity : $O(nlgn)$

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

- Space complexity : $O(1)$ 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 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

Complexity Analysis

- Time complexity : $O(\infty)$

  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.

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

  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 modifed problem. Therefore, the expected runtime for our problem is also linear, as the runtime of the modifed 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

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 (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 : $O(lgn)$

  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.

Quick Navigation
Java Python3 Copy
View in Article (/articles/majority-element/)
Notes

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.

Gregable.com

Majority Voting Algorithm

Find the majority element in a list of values

Oct 6, 2013

I haven't done an algorithms post in awhile, so the usual disclaimer first: If you don't find programming algorithms interesting, stop reading. This post is not for you.

Problem Statement

Imagine that you have a non-sorted list of values. You want to know if there is a value that is present in the list for more than half of the elements in that list. If so what is that value? If not, you need to know that there is no majority element. You want to accomplish this as efficiently as possible.

One common reason for this problem could be fault-tolerant computing. You perform multiple redundant computations and then verify that a majority of the results agree.

Simple Solution

Sort the list, if there is a majority value it must now be the middle value. To confirm it's the majority, run another pass through the list and count it's frequency.

The simple solution is `O(n lg n)` due to the sort though. We can do better!

Boyer-Moore Algorithm

The Boyer-Moore algorithm is presented in this paper: Boyer-Moore Majority Vote Algorithm. The algorithm uses `O(1)` extra space and `O(N)` time. It requires exactly 2 passes over the input list. It's also quite simple to implement, though a little trickier to understand how it works.

In the first pass, we generate a single candidate value which is the majority value if there is a majority. The second pass simply counts the frequency of that value to confirm. The first pass is the interesting part.

In the first pass, we need 2 values:

1. A `candidate` value, initially set to any value.
2. A `count`, initially set to `0`.

For each element in our input list, we first examine the `count` value. If the count is equal to 0, we set the `candidate` to the value at the current element. Next, first compare the element's value to the current `candidate` value. If they are the same, we increment `count` by 1. If they are different, we decrement `count` by 1.

In python:

```
candidate = 0
count = 0
for value in input:
  if count == 0:
    candidate = value
  if candidate == value:
```

```
      count += 1
    else:
      count -= 1
```

At the end of all of the inputs, the `candidate` will be the majority value if a majority value exists. A second O(N) pass can verify that the `candidate` is the majority element (an exercise left for the reader).

Explanation

To see how this works, we only need to consider cases that contain a majority value. If the list does not contain a majority value, the second pass will trivially reject the candidate.

First, consider a list where the first element is not the majority value, for example this list with majority value `0`:

```
[5, 5, 0, 0, 0, 5, 0, 0, 5]
```

When processing the first element, we assign the value of 5 to `candidate` and 1 to `count`. Since 5 is not the majority value, at some point in the list our algorithm must find another value to pair with every 5 we've seen so far, thus `count` will drop to `0` at some point before the last element in the list. In the above example, this occurs at the 4th' element:

List Value:

```
[5, 5, 0, 0, ...
```

Count value:

```
[1, 2, 1, 0, ...
```

At the point that `count` returns to `0`, we have consumed exactly the same number of 5's as other elements. If all of the other elements were the majority element as in this case, we've consumed 2 majority elements and 2 non-majority elements. This is the largest number of majority elements we could have consumed, but even still the majority element must still be a majority of the *remainder* of the input list (in our example, the remainder is ... **`0, 5, 0, 0,`** `5]`). If some of the other elements were not majority elements (for example, if the value was 4 instead), this would be even more true.

We can see similarly that if the first element was a majority element and `count` at some point drops to `0`, then we can also see that the majority element is still the majority of the remainder of the input list since again we have consumed an equal number of majority and non-majority elements.

This in turn demonstrates that the range of elements from the time `candidate` is first assigned to when `count` drops to `0` can be discarded from the input without affecting the final result of the first pass of the algorithm. We can repeat this over and over again discarding ranges that prefix our input until we find a range that is a suffix of our input where `count` never drops to `0`.

Given an input list suffix where count never drops to `0`, we must have more values that equal the first element than values that do not. Hence, the first element (`candidate`) must be the majority of that list and is the only possible candidate for the majority of the full input list, though it is still possible there is no majority at all.

Fewer comparisons

The above algorithm makes 2 passes through our list, and so requires `2N` comparisons in the worst case. It requires another `N` more if you consider the comparisons of `count` to 0. There is another, more complicated, algorithm that operates using only `3N/2 - 2` comparisons, but requires `N` additional storage. The paper (Finding a majority among N votes) also proves that `3N/2 - 2` is optimal.

Their approach is to rearrange all of the elements so that no two adjacent elements have the same value and keep track of the leftovers in a "bucket".

In the first pass, you start with an empty rearranged list and an empty "bucket". You take elements from your input and compare with the last element on the rearranged list. If they are equal you place the element in the "bucket". If they are not equal, you add the element to the end of the list and then move one element from the bucket to the end of the list as well. The last value on your list at the end of this phase is your majority candidate.

In the second pass, you repeatedly compare the candidate to the last value on the list. If they are the same, you discard two values from the end of the list. If they are different, you discard the last value from the end of the list and a value from the bucket. In this way you always pass over two values with one comparison. If the bucket ever empties, you are done and have no majority element. If you remove all elements from the rearranged list without emptying the bucket your candidate is the majority element.

Given the extra complexity and storage, I doubt this algorithm would have better real performance than Boyer-Moore in all but some contrived cases where equality comparison is especially expensive.

Distributed Boyer-Moore

Of course, Gregable readers probably know that I like to see if these things can be solved in parallel on multiple processors. It turns out that someone has done all of the fun mathematical proof to show how to solve this in parallel: Finding the Majority Element in Parallel.

Their solution boils down to an observation (with proof) that the first phase of Boyer-Moore can be solved by combining the results for sub-sequences of the original input as long as both the `candidate` **and** `count` values are preserved. So for instance, if you consider the following array:

`[1, 1, 1, 2, 1, 2, 1, 2, 2]` (Majority = 1)

If you were to run Boyer-Moore's first pass on this, you'd end up with:

```
candidate = 1
count = 1
```

If you were to split the array up into two parts and run Boyer-Moore on each of them, you'd get something like:

split 1:

```
[1, 1, 1, 2, 1]
candidate = 1
count = 3
```

split 2:

```
[2, 1, 2, 2]
candidate = 2
count = 2
```

You can then basically run Boyer-Moore over the resulting `candidate`, `count` pairs the same as you would if it were a list containing only the value `candidate` repeated `count` times. So for instance, Part 1's result could be considered the same as [1, 1, 1] and Part 2's as [2, 2]. However knowing that these are the same value repeated means you can generate the result for each part in constant time using something like the following python:

```
candidate = 0
count = 0
for candidate_i, count_i in parallel_output:
  if candidate_i = candidate:
    count += count_i
  else if count_i > count:
    count = count_i - count
    candidate = candidate_i
  else:
    count = count - count_i
```

This algorithm can be run multiple times as well to combine parallel outputs in a tree-like fashion if necessary for additional performance.

As a final step, a distributed count needs to be performed to verify the final candidate.