




leetcode 169: majority element

Dan

June 13, 2019

 Description [click to see the question on Leetcode](#)

169. Majority Element

Easy  1674  147  Favorite  Share

Given an array of size n , find the majority element. The majority element is the element that appears **more than** $\lfloor n/2 \rfloor$ times.

You may assume that the array is non-empty and the majority element always exist in the array.

Example 1:

Input: [3,2,3]

Output: 3

Example 2:

Input: [2,2,1,1,1,2,2]

Output: 2

Figure 1: 问题描述截图



Solution 1

建立一个字典 dict, 记录元素的值和出现的个数, 以元素值为 key, 出现次数为字典值。代码如下:

 Solution 1 - code:

```
def majorElement(nums):  
    record = {}
```

```
    for i in nums:  
        if i in record.keys():  
            record[i] += 1  
        else:  
            record[i] = 1  
    candidate, count = 0, 0  
    for key, val in record.items():  
        if val > count:  
            candidate = key  
            count = val  
    return candidate
```

-  时间复杂度: 遍历两遍, $2*n$, 用大 O 表示法是 $O(n)$
-  空间复杂度: $O(n)$, 因为平均情况下需要创建长度为 $n/2$ 的字典。

Solution 2

使用 Python 内建的数据结构, 不重复造轮子: 使用 Counter。需要 import from collections.

Counter 对象的 `most_common([n])` 方法返回的是一个 list, list 里面的元素是 tuple: (val, count). 所以我们取 list 的第 0 个元素 (即第一个元组) 的第 0 个值 (即对应的出现最多的数字)。

这个是本文中 3 种方法中最快的……

 Solution 1 - code:

```
from collections import Counter  
def majorityElement(nums):  
    counter = Counter(nums)  
    return counter.most_common(1)[0][0]
```



🕒 时间复杂度: 遍历 1 遍, $2*n$, 用大 O 表示法是 $O(n)$ 。得到的结果居然是 8。Leetcode 所用的算法显然也是这个但是没有包括 `counter.most_common()` 的运行时间。算法, 因为得出的结果和这个一样。

🕒 空间复杂度: $O(n)$, 因为创建一个大小为 n 的 `counter`。

🕒 这个算法有什么优势呢?

🌟 Solution 3

在介绍第 3 种算法前, 介绍一个算法, 具体的原理请看

🕒 Description [A fast majority vote algorithm](#)

有动态的图解和论文。

这个算法的原理前提是, 在 majority 元素存在。注意, majority 元素是出现超过 $\text{floor}(n/2)$ 次的元素。

假设一个投票表决会场, 有 n 个投票者, 给 m 个候选者投票。主持人派一个秘书依次采访投票者, 并记录暂时当选的候选人和票数相对值。流程如下:

🌟 秘书查看手中**票数相对值**, 如果值为 0, 把这会儿采访的人支持的大佬记录为**当前候选人**, 并将**票数相对值**加一。然后访问下一个投票者。这个算初始化, 这个所谓的初始化可能会做多次, 只要**票数相对值**为 0 了, 就得做一次。

🌟 秘书查看手中**票数相对值**, 如果值不为 0, 则问一下这会儿采访的人支持的大佬是不是刚好是手里记的**当前候选人**, 如果不是, 把**票数相对值**减一, 也就是该候选人的票被抵消了一票, 因为投票人支持的不是他, 好惨啊。如果刚好这会儿采访的人支持的也是当前候选人, 就喜滋滋地将**票数相对值**加一。

🌟 对每个投票者都重复上述过程, 直到采访完毕。呼~

如何来理解这个原理呢? 我们可以理解为“配对式抵消”。先临时地将遇到的采访者的支持对象作为**当前候选人**, 然后一次与别的票型进行配对, 相符则加一, 不相符就减一。由于 majority 是出现超过 $\text{floor}(n/2)$ 次的, 那么 majority 一定会至少多一票, 也就是至少剩下一票抵消不掉。

但是注意这个算法的前提, 如果数组中有两个或多个元素出现同样多的次数, 那就无法得到正确的结果了。得到的结果可能甚至与这几个出现次数最多的元素无关, 而仅是与出现的位置有关。例如, 对于输入 `[3,2,3,4,5,6,6,7,8]`,

当输入数组的长度固定时, 这个算法并没有特别的优势, 毕竟别的算法也是 $O(n)$, 即线性时间的时间复杂度。但是当输入的数组的长度不是固定的时候, 这个算法就有优势了, 因为当数组有所变动时, 不必再回头看一遍数组之前的元素, 而是直接与新进来的元素相“配对”即可。别的方法会因这种变动, 而产生 $O(n^2)$ 的时间复杂度。注意, 在这种情况下, 仍需要保证 majority 元素是确定存在的!

此外, 这个算法的空间复杂度是 $O(1)$! 因为只要一个“秘书”就够了。

🕒 Solution 3 - code:

```
def majorityElement(nums):
    candidate, vote = 0, 0
    for i in range(len(nums)):
        if vote == 0:
            candidate = nums[i]
            vote += 1
        elif candidate == nums[i]:
            vote += 1
        else: vote -= 1
    return candidate
```