

# Yunyeong Kim

## 🔍 Question ▾

Given an integer array of size  $n$ , find all elements that appear more than  $\lfloor n/3 \rfloor$  times.

### ☰ Example ▾

**Input:** nums = [3,2,3]

**Output:** [3]

### ☰ Example ▾

**Input:** nums = [1]

**Output:** [1]

### ☰ Example ▾

**Input:** nums = [1,2]

**Output:** [1,2]

## 🔗 Definition ▾

- $\lfloor n/3 \rfloor$  times means  $\rightarrow 2/3 = 0$
- The floor of  $n$  divided by 3
- Floor of  $n$  over 3

## First Code : 10 min

```
class Solution(object):  
    def majorityElement(self, nums):  
        flag = int(len(nums)/3)  
        counter = {}
```

```

arr = []
for num in nums:
    if num in arr:
        continue
    if num in counter:
        counter[num] += 1
    else:
        counter[num] = 1
    if counter[num] > flag:
        arr.append(num)
return arr
# O(n) / O(n) - Dict

```

## Solution

```

class Solution:
    def majorityElement(self, nums: list[int]) -> list[int]:
        # Counters for the potential majority elements
        count1 = 0
        count2 = 0
        # Potential majority element candidates
        candidate1 = 0
        candidate2 = 0

        # First pass to find potential majority elements.
        for num in nums:
            # If count1 is 0 and the current number is not equal to
            candidate2, update candidate1.
            if count1 == 0 and num != candidate2:
                count1 = 1
                candidate1 = num

            # If count2 is 0 and the current number is not equal to
            candidate1, update candidate2.
            elif count2 == 0 and num != candidate1:
                count2 = 1
                candidate2 = num

            # Update counts for candidate1 and candidate2.
            elif candidate1 == num:
                count1 += 1
            elif candidate2 == num:
                count2 += 1

            # If the current number is different from both candidates,

```

```

decrement their counts.
    else:
        count1 -= 1
        count2 -= 1

result = []
threshold = len(nums) // 3 # Threshold for majority element

# Second pass to count occurrences of the potential majority
elements.
count1 = count2 = 0
for num in nums:
    if candidate1 == num:
        count1 += 1
    elif candidate2 == num:
        count2 += 1

# Check if the counts of potential majority elements are greater
than n/3 and add them to the result.
if count1 > threshold:
    result.append(candidate1)
if count2 > threshold:
    result.append(candidate2)

return result

```

## pigeonhole principle majority



A Pizza has  $n = 8$  slices,  
 - each person is allowed  $\lfloor n/3 \rfloor$  slices  
 -  $\lfloor n/3 \rfloor = 8/3 = 2.6 = 2$   
 if fair situation, 4 people can eat 2 slices.

Problem : more than  $\lfloor n/3 \rfloor$  times  $\rightarrow$  Maximum is set.

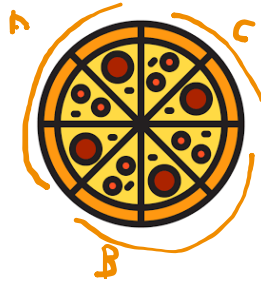
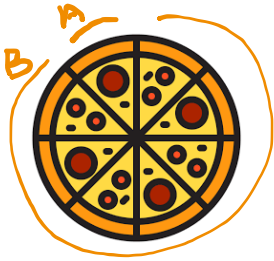
## Because the condition is ratio-based

- $\lfloor n/3 \rfloor$  is a ratio-based threshold derived from the array size  $n$ .
- Since the total number of elements, " $n$ ", is fixed,
- and " $n/3$ " serves as an absolute threshold,
- there is a mathematical limit to how many elements can exceed this threshold.



A Pizza has  $n = 8$  slices,  
 - each person is allowed  $\lfloor n/3 \rfloor$  slices  
 -  $\lfloor n/3 \rfloor = 8/3 = 2.6 = 2$   
 if greedy member eat more than pair slices,  
 - more than 2

How many member can be greedy?



Maximum = 2

$$k \times (\lfloor n/3 \rfloor + 1) \leq n \quad k \leq \frac{n}{\lfloor n/3 \rfloor + 1}$$

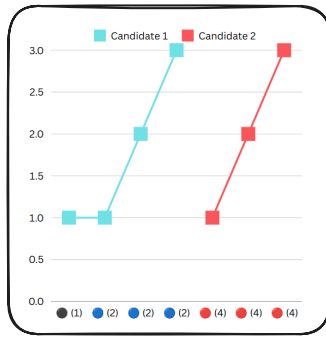
$$\lfloor 10/3 \rfloor = 3 \quad k \leq \frac{10}{4} = 2.5$$

## 📌 결론:

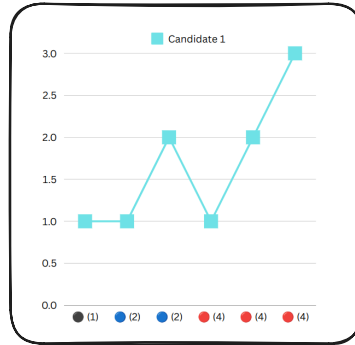
- **비둘기집 원리**를 확장해서, 배열 내 특정 기준 이상으로 등장할 수 있는 원소의 개수를 논리적으로 제한할 수 있습니다.
- $n/(k+1)$ 보다 많이 등장할 수 있는 원소는 **최대  $k$ 개**입니다.
- 이 논리를 기반으로 효율적인 알고리즘(예: Boyer-Moore Voting Algorithm 확장형)을 설계할 수 있습니다. 🚀

## Boyer-Moore Majority Voting

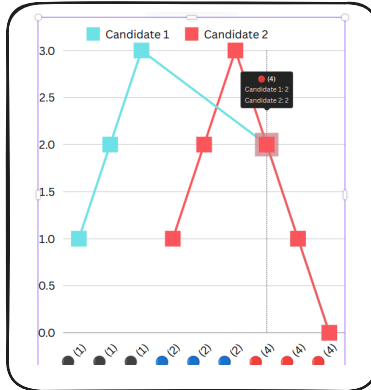
EX2 : Array = [1, 2, 2, 2, 4, 4, 4]  
Answer = [2, 4]



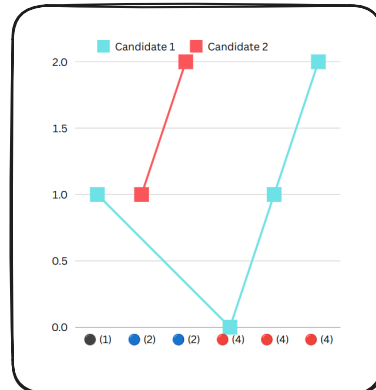
EX1 : Array = [1, 2, 2, 4, 4, 4]  
Answer = [4]



EX4 : Array = [1, 1, 1, 2, 2, 2, 4, 4, 4]  
Answer = []



EX1 : Array = [1, 2, 2, 4, 4, 4]  
Answer = [4]



## 📌 결론:

- Although both have a time complexity of  $O(n)$ ,
- **Boyer-Moore** is more efficient in terms of space complexity with  $O(1)$ .
- The second pass might make it seem more complex, but in the end, it uses less memory compared to the hash map approach.