Let’s go to work

Find the majority element in an array

A ***majority element*** in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Optimum Run time : O(N)

Optimum Aux. Space Complexity: O(1)

Examples :

Input : [3, 3, 4, 2, 4, 4, 2, 4, 4]

Output : 4

Explanation: The frequency of 4 is 5 which is greater than the half of the size of the array size.

Input : [1, 2, 3, 5, 1, 1, 1, 1, 1]

Output :1

Explanation: The frequency of 1 is 6 which is greater than the half of the size of the array size.

Approach 1 : Brute Force

Approach 1 : Brute Force

Let’s solve it with Brute Force First

- We can exhaust the search space in quadratic time by checking whether each element is the majority element.
- Algorithm
- Brute force:
  - This approach 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.

SOLUTION 1:

Run Time $O(N^2)$

Space O(1)

Can this be improved?

```python
# Brute Force
def find_majority_element_bf(nums: List[int]) -> int:
    """
    param nums: int container to look up
    :return: frequent value in array
    :pre-cond: there will be at least one elem in list
    """

    pass
```

Let’s solve it with Sorting in mind

Use Sorted to sort the list and a nested loop!

SOLUTION 1:

Run Time O(NlogN)

Space O(1)

Can this be improved?

```python
# Brute Force
def find_majority_element_bf(nums: List[int]) -> int:
    """
    param nums: int container to look up
    :return: frequent value in array
    :pre-cond: there will be at least one elem in list
    """

    pass
```

Lets try another approach

Approach 2 : Sorting

Sorting

- If the elements are sorted in monotonically increasing (or decreasing) order, the majority element can be found at index $\lfloor n/2 \rfloor$ (and also at $\lfloor n/2 \rfloor - 1$, if n is even).

How to implement ?

Sort nums, and return the element in question.

Array with even number of elements:

[4, 4, 2, 4, 3, 5, 4]

After sorting, see the mid point

[2, 3, 4, 4, 4, 4 ,5]

Array with odd number of elements

[1, 2, 3, 5, 1, 1, 1, 1, 1]

[1, 1, 1, 1, 1, 1 ,2, 3, 5,]

Let’s complete this:

```python
def find_majority_element_v1(self, nums: List[int]) -> int:
    pass
```

Let’s try another approach

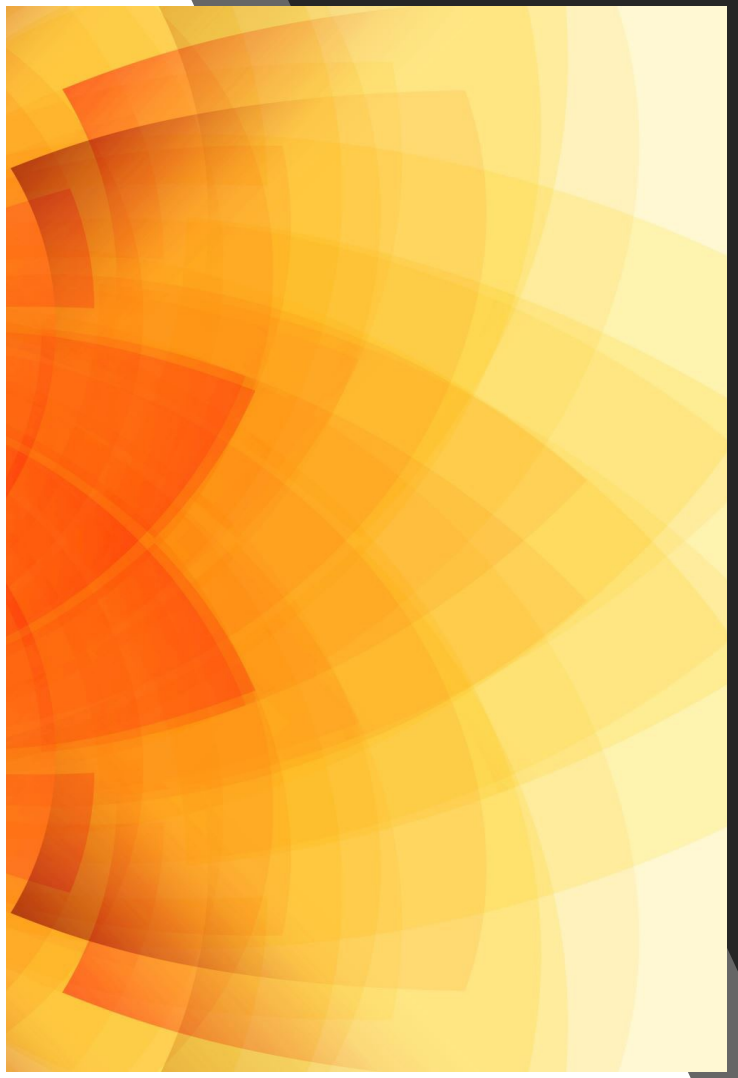
Approach 3 : Using a Hash Map

Using a Hash Map

- We know that the majority element occurs more than ⌊n/2⌋ times, and a HashMap allows us to count element occurrences efficiently.

How do we implement this?

- 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.



Let’s complete this

```
def find_majority_element_v2(self, nums: List[int]) -> int:
    pass
```

Analysis

- 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.

SOLUTION 3:

Using a dictionary

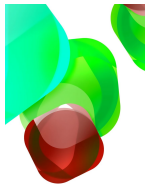
Run Time O(N)

Space O(N)

Let’s try another approach

Approach 4 : Using a Hash Map with a twist

Using Collections



Let's complete this

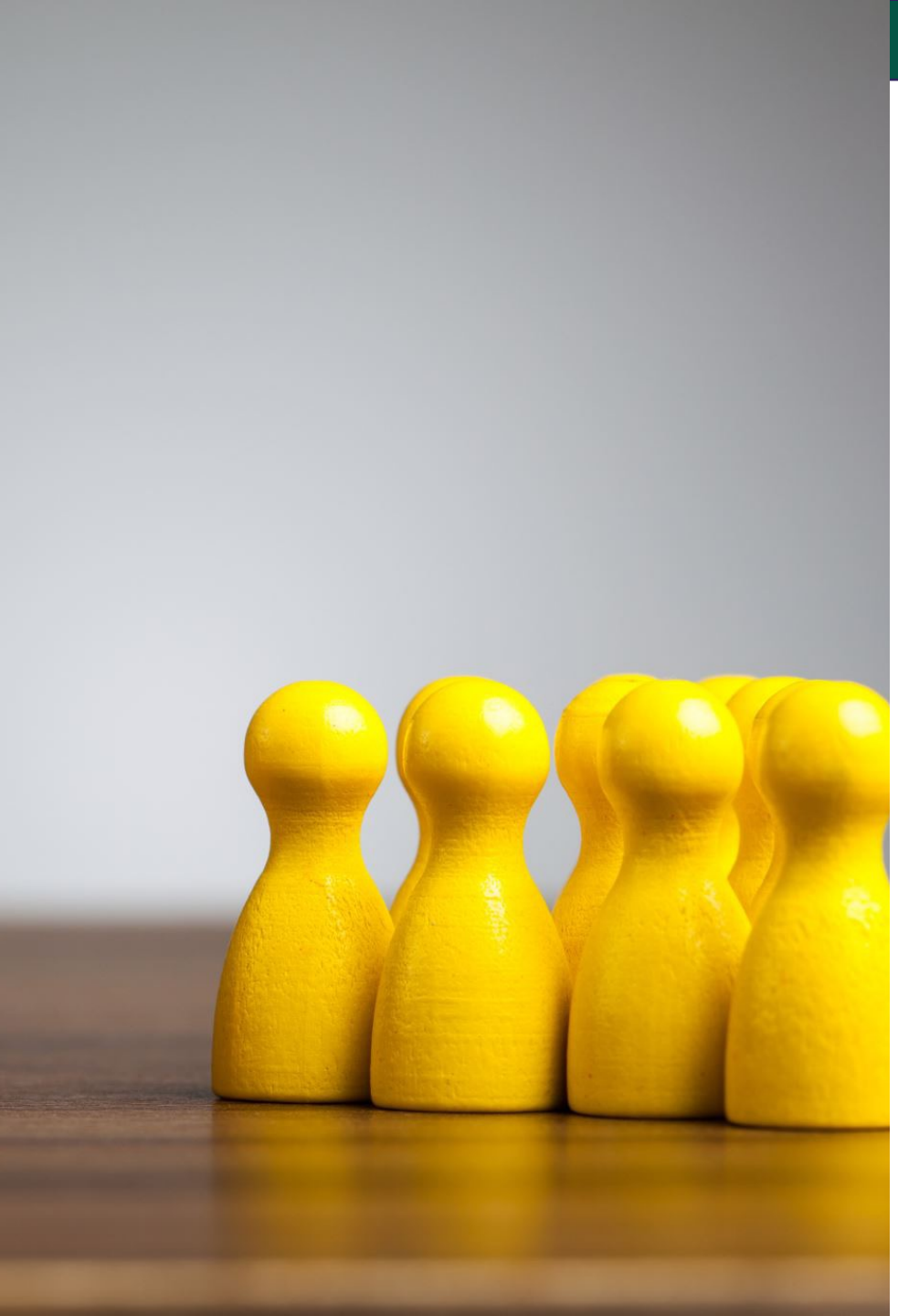
```python
# using collections module
def find_majority_element_v3(self, nums: List[int]) -> int:
    pass
```

Let’s try another approach

Approach 5 : We will try to improve our space complexity

BOYER-MOORE MAJORITY VOTING ALGORITHM

Run Time O(N)
Space O(1)



BOYER-MOORE MAJORITY VOTING ALGO

- The algorithm maintains in its local variables
  - a sequence element(candidate)
  - and a counter, with the counter initially zero.

- It then processes the elements of the sequence, one at a time.

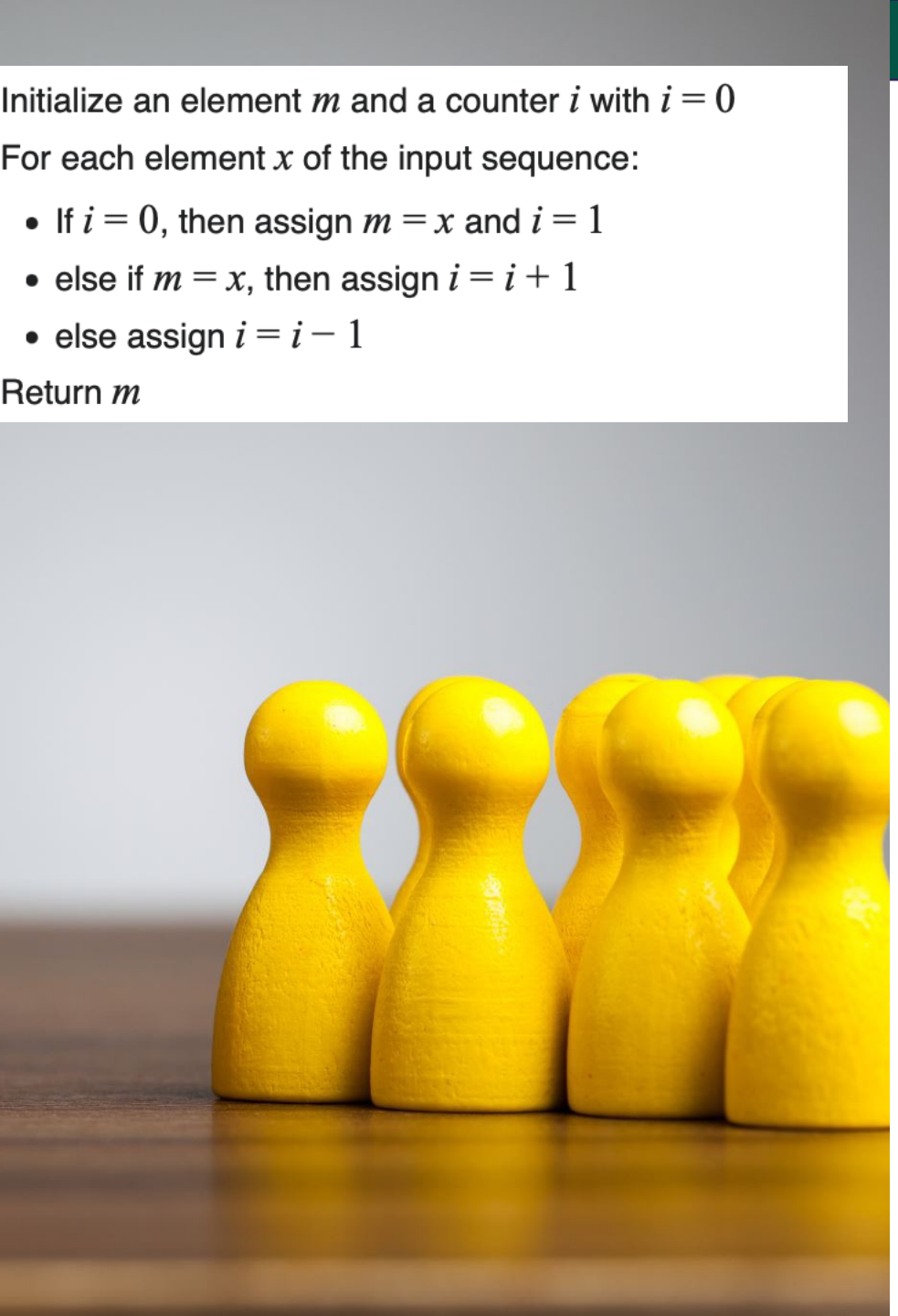
- Initialize an element m and a counter i with $i = 0$
- For each element x of the input sequence:
  - If $i = 0$, then assign $m = x$ and $i = 1$
  - else if $m = x$, then assign $i = i + 1$
  - else assign $i = i - 1$
- Return m

Initialize an element m and a counter i with $i = 0$

For each element x of the input sequence:

- If $i = 0$, then assign $m = x$ and $i = 1$
- else if $m = x$, then assign $i = i + 1$
- else assign $i = i - 1$

Return m



BOYER-MOORE MAJORITY VOTING ALGO

- When processing an element *x*, if the counter is zero, the algorithm stores *x* as its remembered sequence element(candidate) and sets the counter to one.
- Otherwise, it compares *x* to the stored element(candidate) and either increments the counter (if they are equal) or decrements the counter (otherwise).
- At the end of this process, if the sequence has a majority, it will be the element stored by the algorithm.

- Initialize an element m and a counter i with $i = 0$
- For each element x of the input sequence:
  - If $i = 0$, then assign $m = x$ and $i = 1$
  - else if $m = x$, then assign $i = i + 1$
  - else assign $i = i - 1$
- Return m

BOYER-MOORE MAJORITY VOTING ALGORITHM

Run Time O(N)
Space O(1)

Boyer-Moore

N=9
There are total of 5 2's
9//2 =4
So: 2 is our majority element.
We will initialize 2 variables.
Candidate is the majority element we are looking for
Count is the number of times we have seen this element

candidate = 0

count = 0

element = 2

- Initialize an element m and a counter i with $i = 0$
- For each element x of the input sequence:
  - If $i = 0$, then assign $m = x$ and $i = 1$
  - else if $m = x$, then assign $i = i + 1$
  - else assign $i = i - 1$
- Return m

Boyer-Moore

Starting from index 0 , count =0
candidate becomes 2
Increment count by 1

candidate = 2

count = 1

element = 2

Boyer-Moore

At index 1:
current candidate is 2
Compare it with the value in index 1 ,
if they are different , decrement count

candidate = 2

count =0

element = 1

Boyer-Moore

candidate = 2

count =1

element = 2

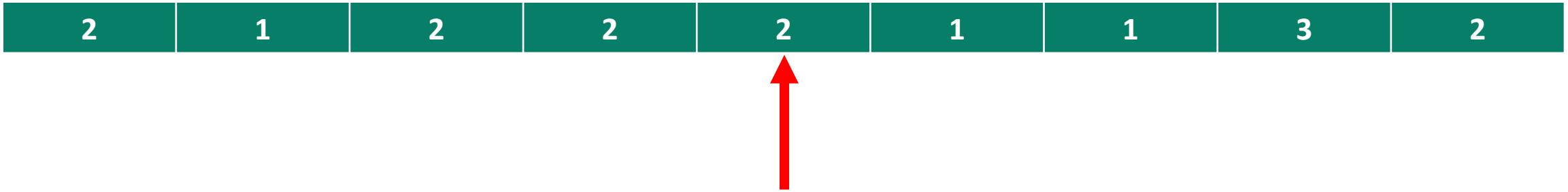
Boyer-Moore

candidate = 2

count =2

element = 2

Boyer-Moore

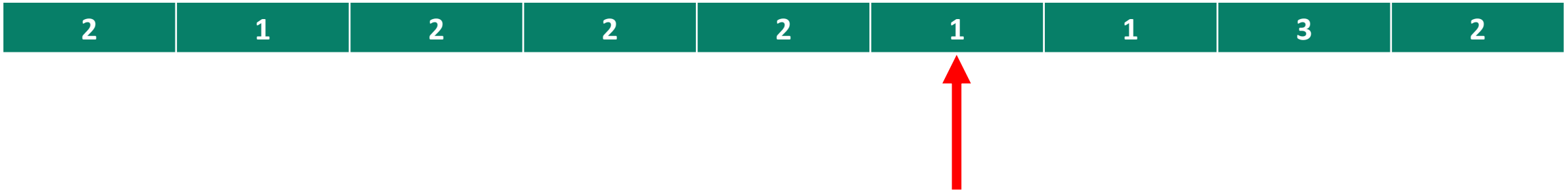


candidate = 2

count =3

element = 2

Boyer-Moore

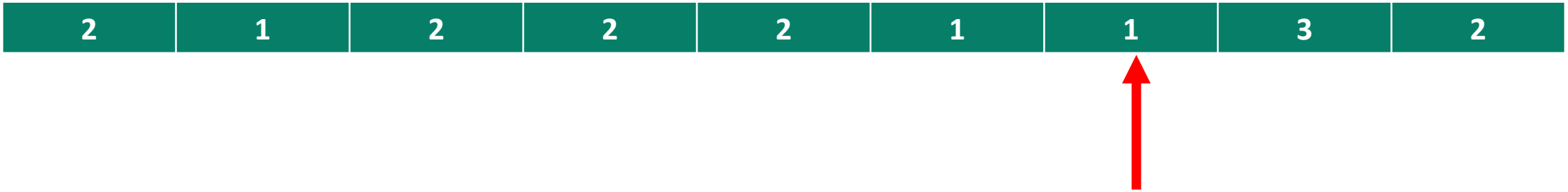


candidate = 2

count =2

element = 1

Boyer-Moore

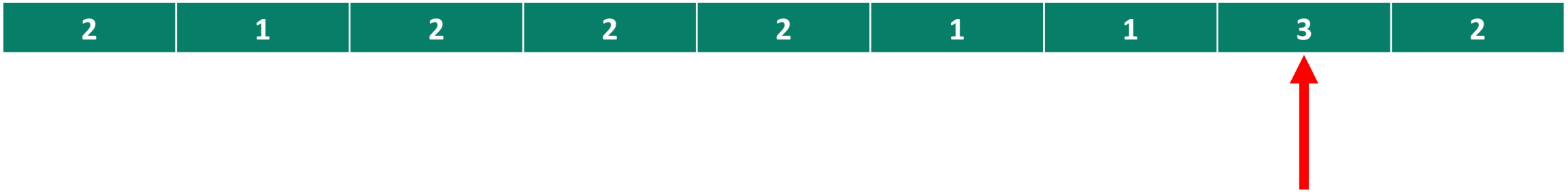


candidate =2

count =1

element = 1

Boyer-Moore

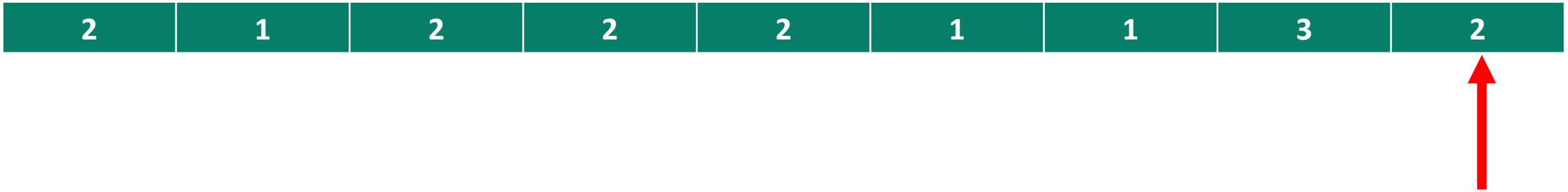


candidate = 2

count =0

element = 3

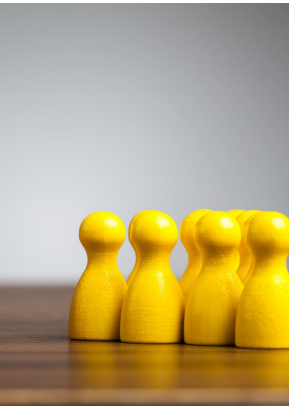
Boyer-Moore



candidate = 2

count =1

element = 2



Let's complete this

```
def find_majority_element_Boyer_Moore(self, nums: List[int]) -> int:
    pass
```

Boyer-Moore Analysis

Majority element means it is an element that appears more than n/2 time, n being the length of the array) in an array in

O(N) run time and O(1) Space complexity.

If time permits….

One more problem to work on…

Logger Rate Limiter

Design a logger system that receives a stream of messages along with their timestamps.

Each **unique** message should only be printed **at most every 10 seconds** (i.e. a message printed at timestamp **t** will prevent other identical messages from being printed until timestamp **t + 1**0).

All messages will come in chronological order.

Several messages may arrive at the same timestamp.

Logger Rate Limiter

Implement the Logger class:

Logger() Initializes the logger object.

bool shouldPrintMessage(int timestamp, string message) Returns true if the message should be printed in the given timestamp, otherwise returns false.

Optimum Run Time O(1)

Space Complexity O(M) → M being the number of incoming messages

Practicing Hash Table

Logger logger = new Logger();

logger.shouldPrintMessage(1, "foo"); // return true, next allowed timestamp for "foo" is 1 + 10 = 11

logger.shouldPrintMessage(2, "bar"); // return true, next allowed timestamp for "bar" is 2 + 10 = 12

logger.shouldPrintMessage(3, "foo"); // 3 < 11, return false

logger.shouldPrintMessage(8, "bar"); // 8 < 12, return false

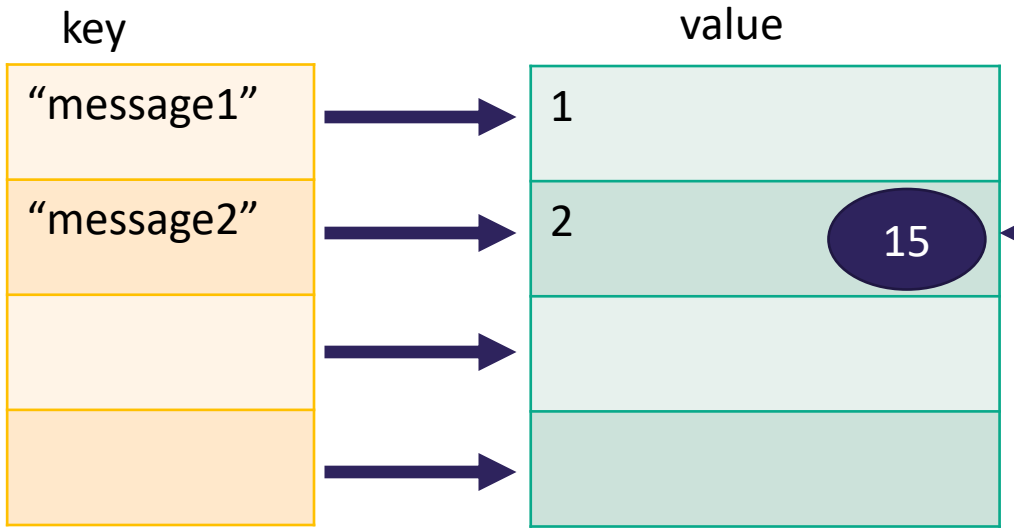
logger.shouldPrintMessage(10, "foo"); // 10 < 11, return false

logger.shouldPrintMessage(11, "foo"); // 11 >= 11, return true, next allowed timestamp for "foo" is 11 + 10 = 21

Practicing with Hash Maps

We can have a hash table/dictionary with the **message as key**, and **its timestamp as the value**.

The hash table keeps all the unique messages along with the latest timestamp that the message was printed.



shouldprint(“message2”, 15)

There is an entry in the hash table with the message2 and the timestamp 2.
Then there comes another message2 with the timestamp 15.
Since the message was printed 13 seconds before (*i.e.* beyond the buffer window),
it is therefore eligible to print again the message.
As a result, the timestamp of the message2 would be updated to 15.

Algorithm

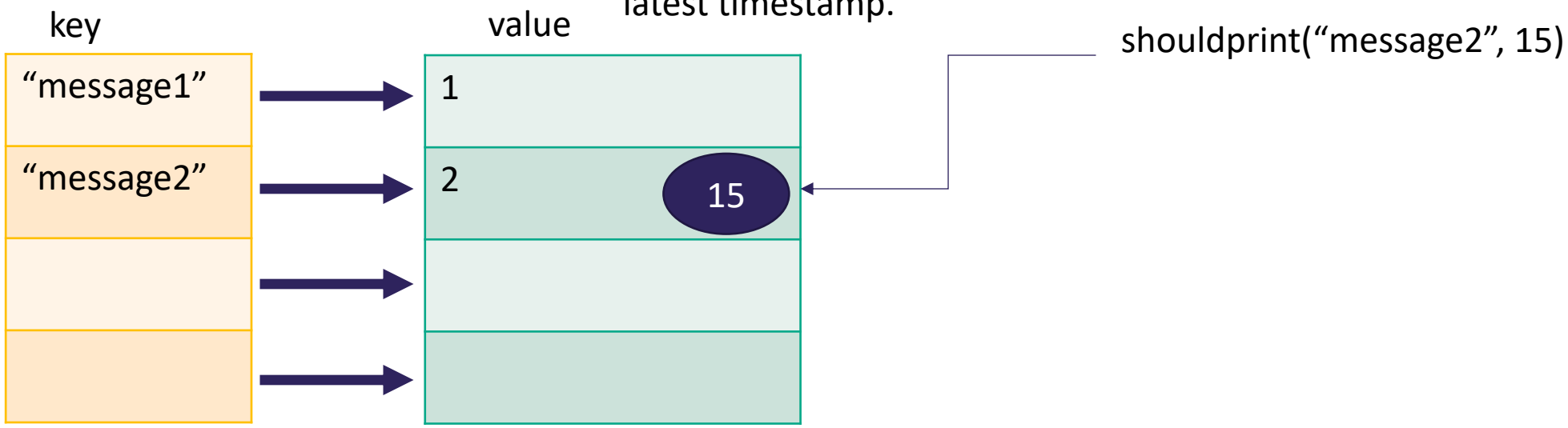
Algorithm

•We initialize a hash table/dictionary to keep the messages along with the timestamp.

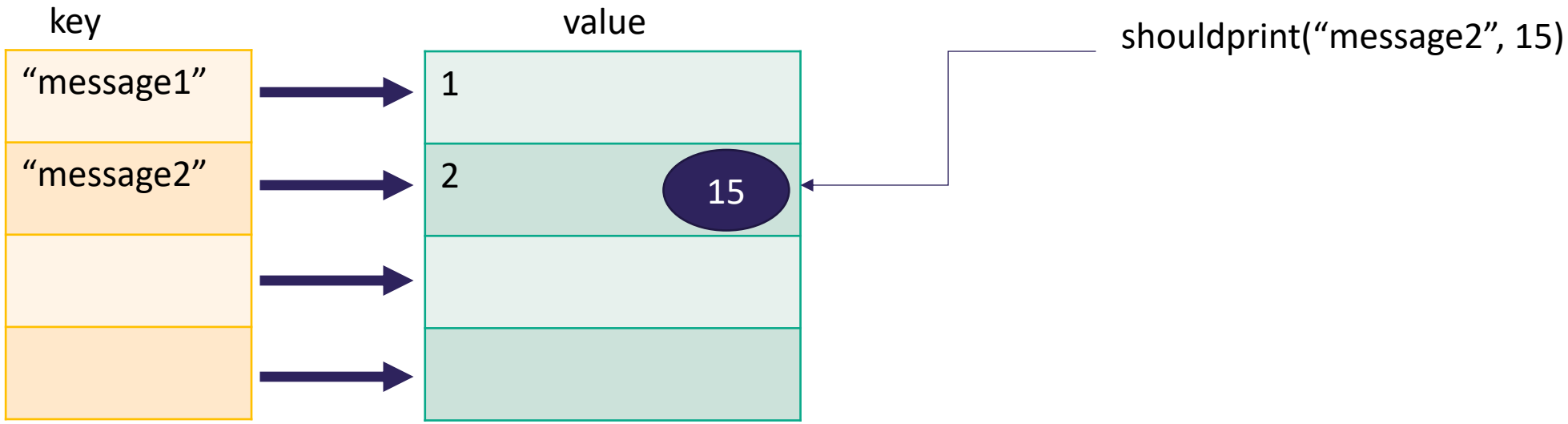
•At the arrival of a new message, the message is eligible to be printed with either of the two conditions as follows:

- case 1). we have never seen the message before.
- case 2). we have seen the message before, and it was printed more than 10 seconds ago.

•In both of the above cases, we would then update the entry that is associated with the message in the hash table, with the latest timestamp.



Algorithm



Let’s complete this

```
def shouldPrintMessage(self, timestamp, message) -> bool:
    """
    :return  true if the message should be printed in the given timestamp, otherwise returns false.
    """
    pass
```