CSE 331

Stack Applications

Instructor Sebnem Onsay

Min stack Problem

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.
Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Min Stack

Design a Stack ADT that retrieves the min element in constant time

Explanation

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Min Stack

Constraints:

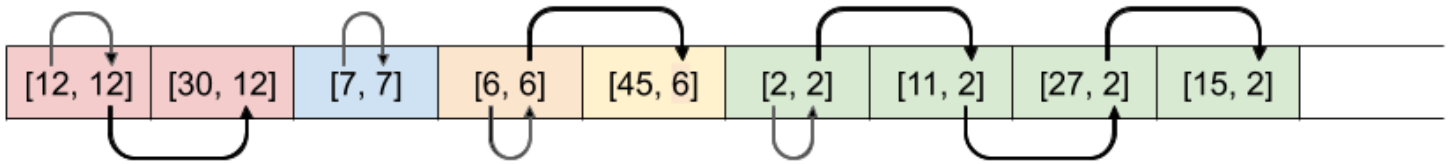
- $-2^{31} <= val <= 2^{31} - 1$
- Methods `pop`, `top` and `getMin` operations will always be called on **non-empty** stacks.
- At most $3 * 10^4$ calls will be made to `push`, `pop`, `top`, and `getMin`.

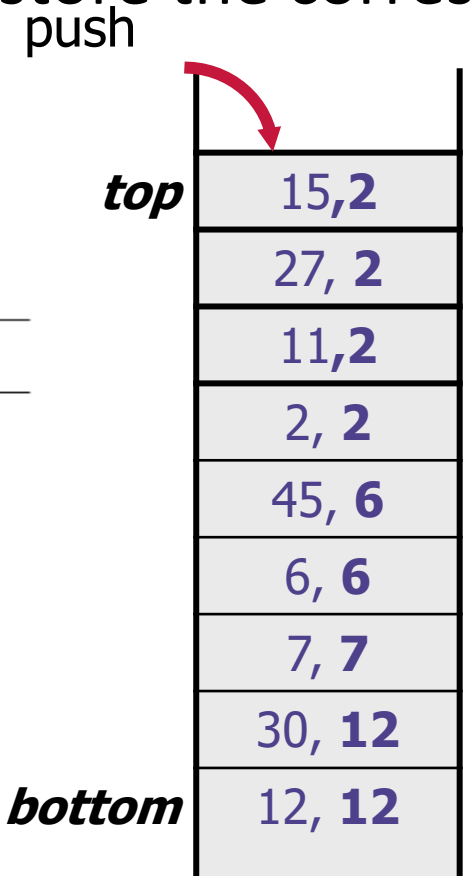
Min Stack :Approach 1

- With a Stack, we only ever add and remove values from the top.
- Therefore, an important invariant of a Stack is that when a new number, which we'll call **val1** is placed on a Stack, the numbers below it will not change for as long as number **val1** remains on the Stack.
- So, whenever number **val1** is the top of the Stack, the minimum will always be the same, as it's simply the minimum out of **val1** and all the numbers below it.

Min Stack :Strategy for Approach 1

- We are using a Python's list…
- When we are pushing an element, we can store the corresponding min with the value we push..

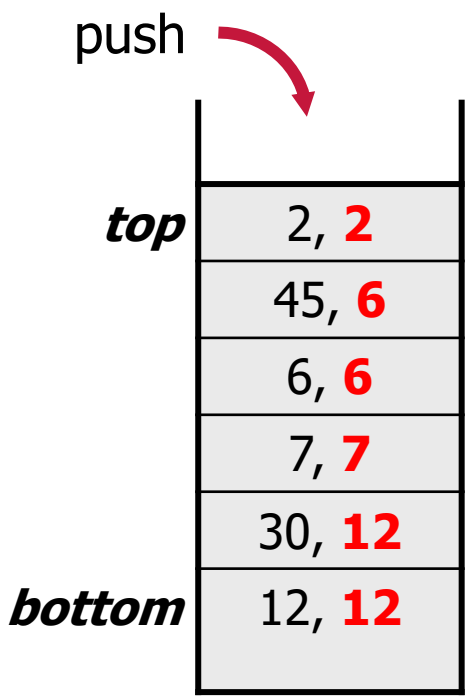




Min Stack :Strategy for Approach 1

- We are using a Python's list…
- When we are pushing an element, we can store the corresponding min with the value we push..

```python
# PROBLEM 1 --> STACKS
# APPROACH 1
class MinStack1:

    def __init__(self):
        self.stack = []

    def push(self, x: int) -> None:
        # If the stack is empty, then the min value
        # must just be the first value we add

        pass

    def pop(self) -> None:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1][0]

    def getMin(self) -> int:
        return self.stack[-1][1]
```

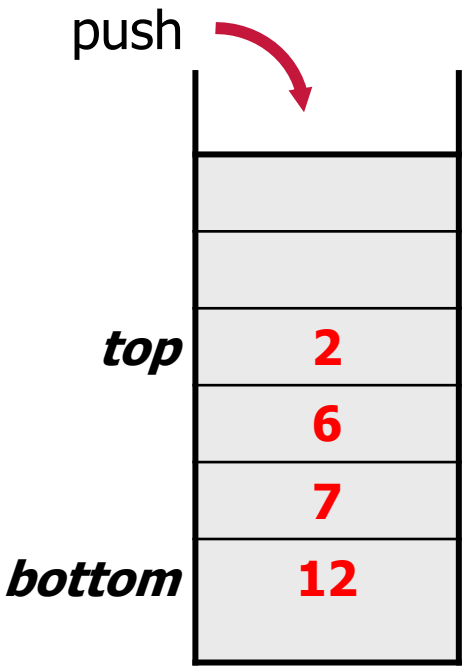


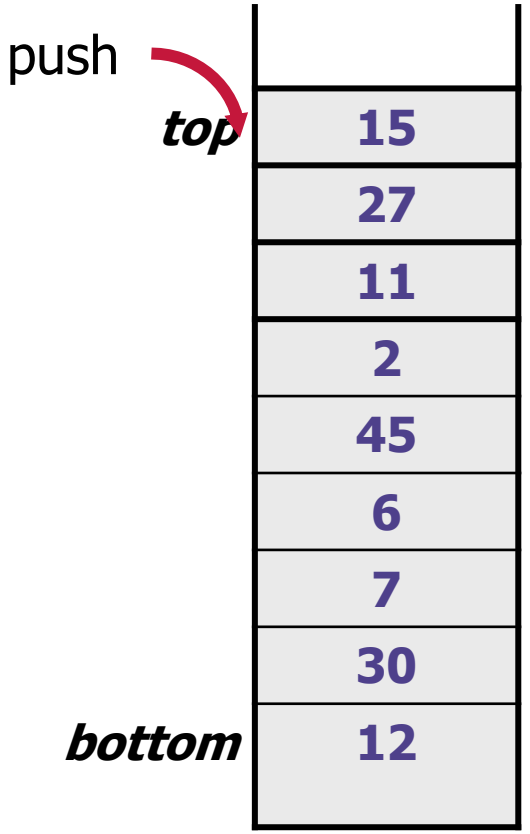
Implementation of Approach 1

```python
# PROBLEM 1 --> STACKS
# APPROACH 1
class MinStack1:

    def __init__(self):
        self.stack = []

    def push(self, x: int) -> None:
        # If the stack is empty, then the min value
        # must just be the first value we add
        if not self.stack:
            self.stack.append((x, x))
            return

        current_min = self.stack[-1][1]
        self.stack.append((x, min(x, current_min)))

    def pop(self) -> None:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1][0]

    def getMin(self) -> int:
        return self.stack[-1][1]
```

Min Stack :Strategy for Approach 2

- How about using another stack …





Potential pitfall

- The push(...) method for this implementation of MinStack is straightforward.
  - Items should always be pushed onto the main Stack*, but they should only be pushed onto the min-tracker Stack if they are smaller than the current top of it*. Well, that's mostly correct.
  - There's one potential pitfall here that we'll look at soon.

Potential pitfall

- MinStack's pop(...) method. The value we need to pop is always on the top of the main underlying Stack.
  - However, if we simply popped it from there, the min-tracker Stack would become incorrect once its top value had been removed from the main Stack!
- A logical solution would be to do the following additional check and modification to the min-tracker Stack when MinStack's pop(...) method is called.

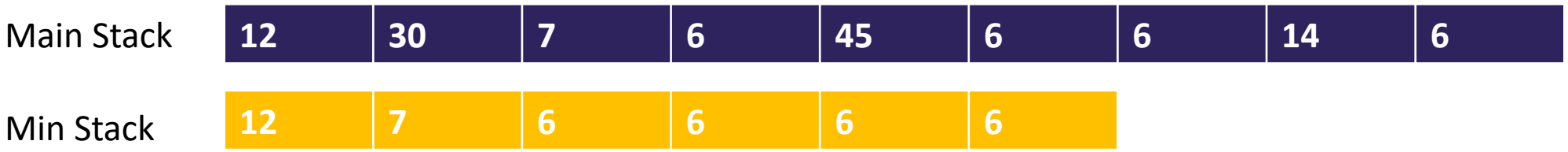
```
If top of main_stack == top of min_tracker_stack:
    min_tracker_stack.pop()
```

- This way, the new minimum would now be the top of the min-tracker Stack

Min Stack : Strategy for Approach 2

- Push only onto the min-tracker Stack if, and only if, it was less than or equal to the current minimum.

- One downside of this solution is that
  - if the same number is pushed repeatedly onto MinStack, and that number also happens to be the current minimum, there'll be a lot of needless repetition on the min-tracker Stack.



Min Stack: Approach 2 -> Using another Stack ?

```python
# APPROACH II
class MinStack2:

    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, x: int) -> None:
        pass
    def pop(self) -> None:
        pass

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.min_stack[-1]
```

Approach 3: Improvement on Approach 2

- An improvement is to put pairs onto the min-tracker Stack.
- The first value of the pair would be the same as before, and the second value would be how many times that minimum was repeated.

Approach 3 -→ You Go to work!

Fix it so it would be like below….

- Improve Approach 2…

Complete the code

```python
#APPROACH III
class MinStack3:

    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, x: int) -> None:

        # We always put the number onto the main stack.


        # If the min stack is empty, or this number is smaller than
        # the top of the min stack, put it on with a count of 1.


        # Else if this number is equal to what's currently at the top
        # of the min stack, then increment the count at the top by 1.


        pass

    def pop(self) -> None:

        # If the top of min stack is the same as the top of stack
        # then we need to decrement the count at the top by 1.


        # If the count at the top of min stack is now 0, then remove
        # that value as we're done with it.


        # And like before, pop the top of the main stack.
        pass

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.min_stack[-1][0]
```

CSE 331

SLIDING WINDOW PRACTICE

Instructor: Stephen Ramsay

Purpose of this exercise is to learn different ways to improve Run time complexity from O(n^2) to O(n) or better ☺

Problem : Fixed window length k

Find the maximum sum of a subarray

Given an array consisting of n integers, find the contiguous subarray of given length k that has the maximum sum value.

Problem : Fixed window length k

Find the maximum sum of a subarray

Input: nums = [1, 4, 2, 10, 2] k=4

Output: Maximum sum of subarray is 18.

Input: nums= [1, 4, 2, 10, 2, 3, 1, 0, 20] k=4

Output: Maximum sum of subarray is 24.

What is Sliding Window?

It is a technique which is useful for solving problems in array or string.

Its main goal is

reducing the time complexity from $O(n^2)$ to $O(n)$.

There are two types of sliding window:

Type 1:

Fixed window length k: the length of the window is fixed, in general it is used in cases where the algorithm is to find something in the window such as

the **maximum sum of all windows**,

the **maximum or median number of each window**.

In general, we will need variables to maintain the state of the window, some are as simple as an integer or it could be as complicated as some advanced data structure such as list, queue or deque.

There are two types of sliding window:

- **Type 2:**
- **Two pointers + criteria**: the window size is not fixed in this case
- In general, it is used in cases where the algorithm is to find a subarray that meets the criteria:
- Example: given an array of integers, find the number of subarrays whose sum is equal to a target value.

Let’s go to work

Problem : Fixed window length k

Find the maximum sum of a subarray

Given an array consisting of n integers, find the contiguous subarray of given length k that has the maximum sum value.

Problem : Fixed window length k

Find the maximum sum of a subarray

Input: nums = [1, 4, 2, 10, 2] k=4

Output: Maximum sum of subarray is 18.

Input: nums= [1, 4, 2, 10, 2, 3, 1, 0, 20] k=4

Output: Maximum sum of subarray is 24.

Let’s solve the problem first…
Approach 1:
Brute Force

Let’s go to work
Using Brute Force

```
def find_max_sum_brute(self, mylist, n, k):
    # Initialize result

    # Consider all blocks
    # starting with i.
    pass
```


Approach 2:
Sliding Window

Find the maximum sum of a subarray

This is a typical problem that falls into the first type of **sliding window** problems since the length of the window is fixed.

Steps to follow:

1. We find the first window with size *k* and maintain a variable **curSum** which equals the sum of all the elements within the current window
2. As we are moving the window one step at a time from left to right, we subtract the leftmost element in the current window and add the next element of the array until we hit the end of the array.
3. Finally, we return the **maxSum**.

Visualize

Below is a list with 4 elements, let k=4 (fixed window size)

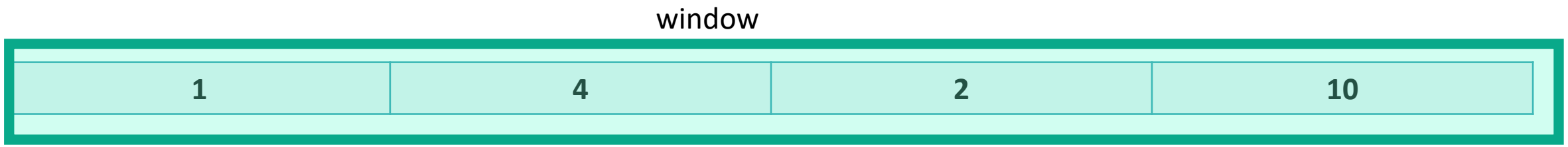
The window will cover the whole list.

1	4	2	10

Visualize

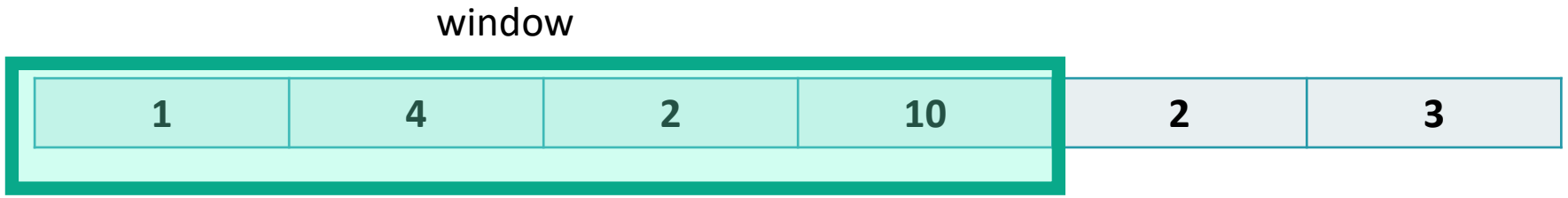
Below is a list with 4 elements, let k=4(fixed window size)

The window will cover the entire list.



Visualize

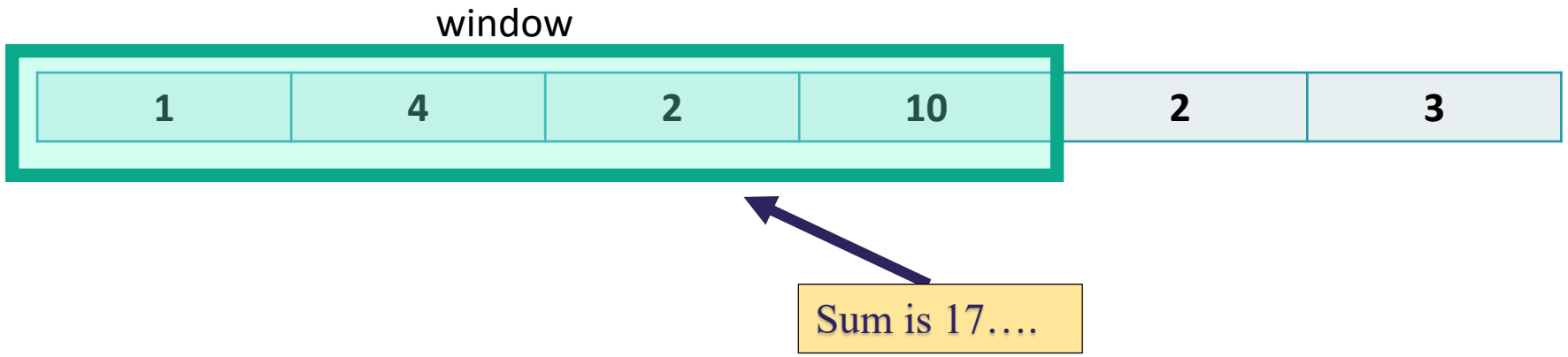
Now imagine you have a bigger list with 6 elements, k=4 (fixed window size)



Visualize

Now imagine you have a bigger list with 6 elements, k=4 (fixed window size)

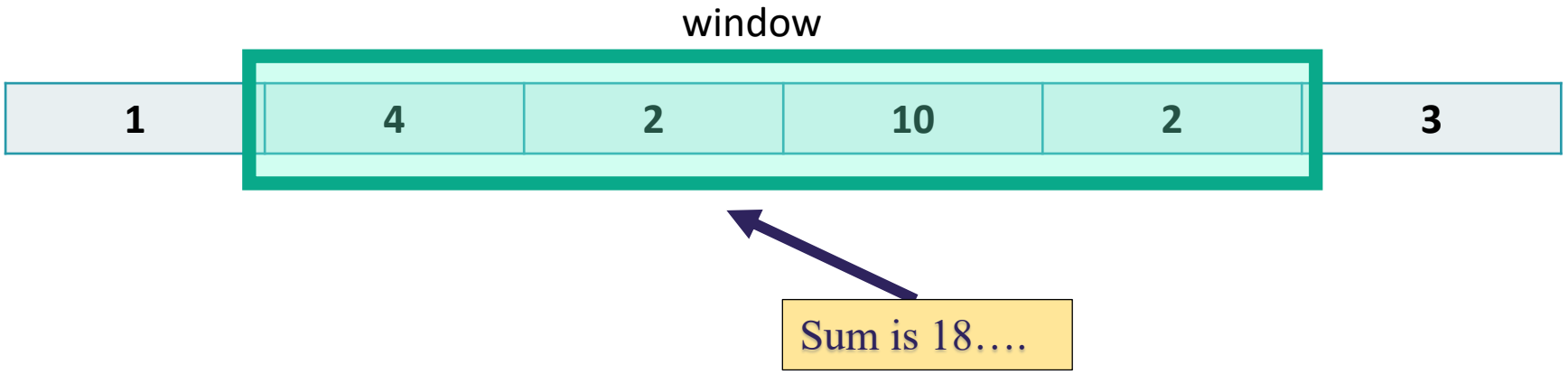
Start from the beginning with your window, that is the first 4 elements:



Visualize

Now imagine you have a bigger list with 6 elements, k=4 (fixed window size)

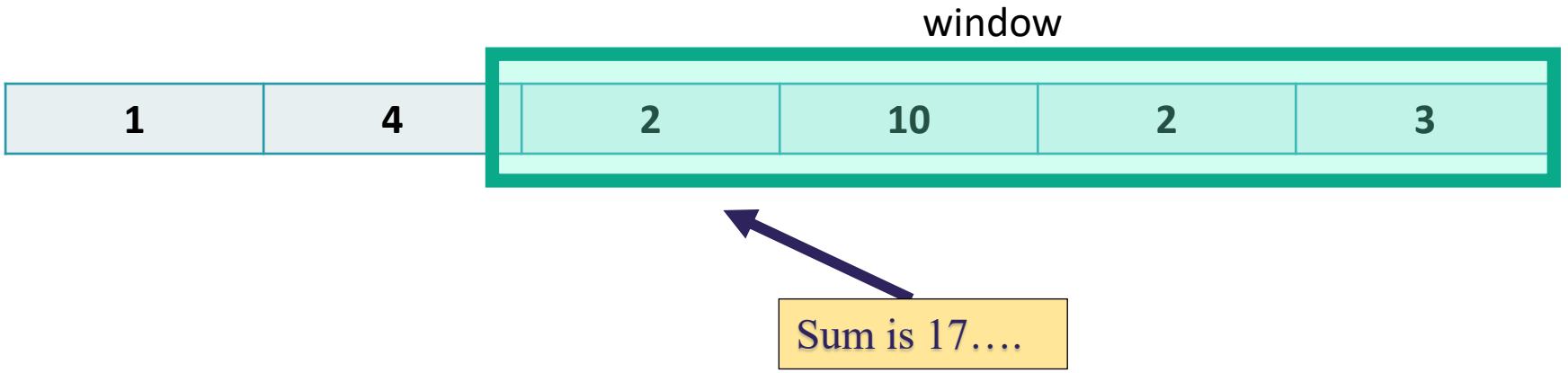
Now slide your window by 1:



Visualize

Now imagine you have a bigger list with 6 elements, k=4 (fixed window size)

Now slide your window by 1:



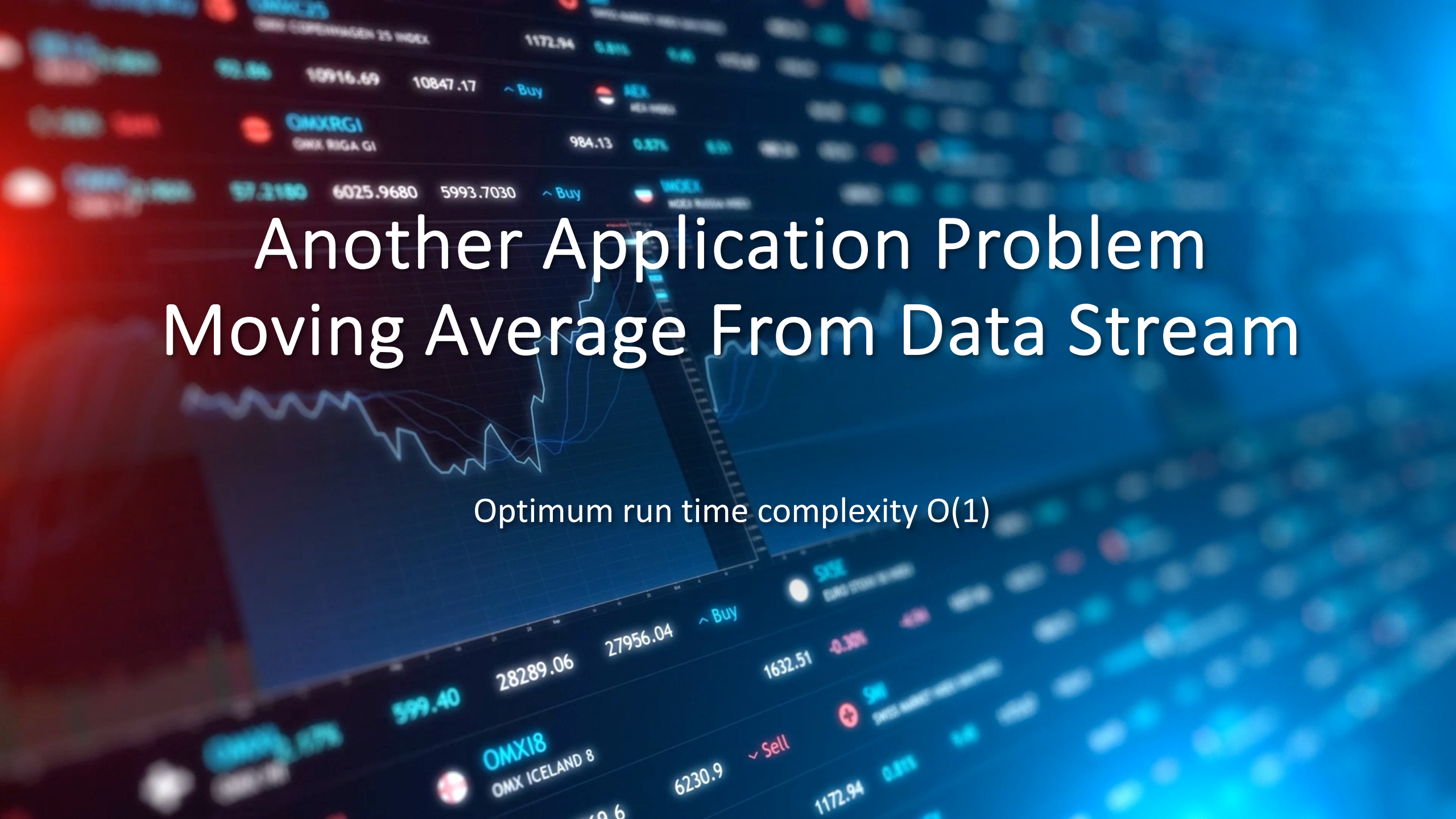
Program returns 18 as the max sum.

Let’s Go to work!

- You can log into Codio and find CC7 or
- Download the starter code from D2L and Open it up in PyCharm.
- Find **solution.py**
- Find the function named as **find_max_sum** function using this approach.
- Optimum run time complexity requirement: O(n).

O(n) Solution

```
def find_max_sum(self, data, n, k):
    # k must be smaller than n

    # Compute sum of first
    # window of size k



    # Compute sums of remaining windows by
    # removing first element of previous
    # window and adding last element of
    # current window.
    pass
```



Another Application Problem Moving Average From Data Stream

Optimum run time complexity O(1)

Problem: Moving Average From Data Stream

Given a stream of integers and a fixed window size, calculate the moving average of all integers in the sliding window.

Optimum run time: O(1)

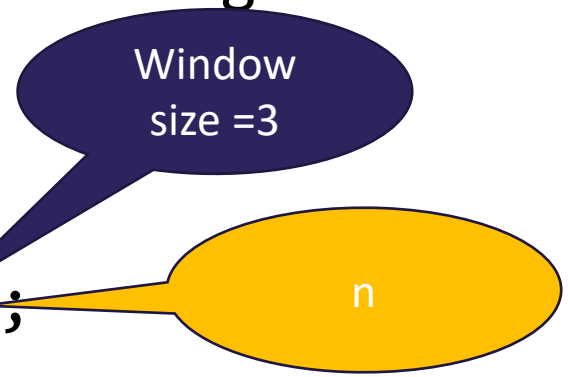
Implement the **MovingAverage1** class:

MovingAverage(int size) Initializes the object with the size of the window size.

double **next**(int val) Returns the moving average of the last size values of the stream.

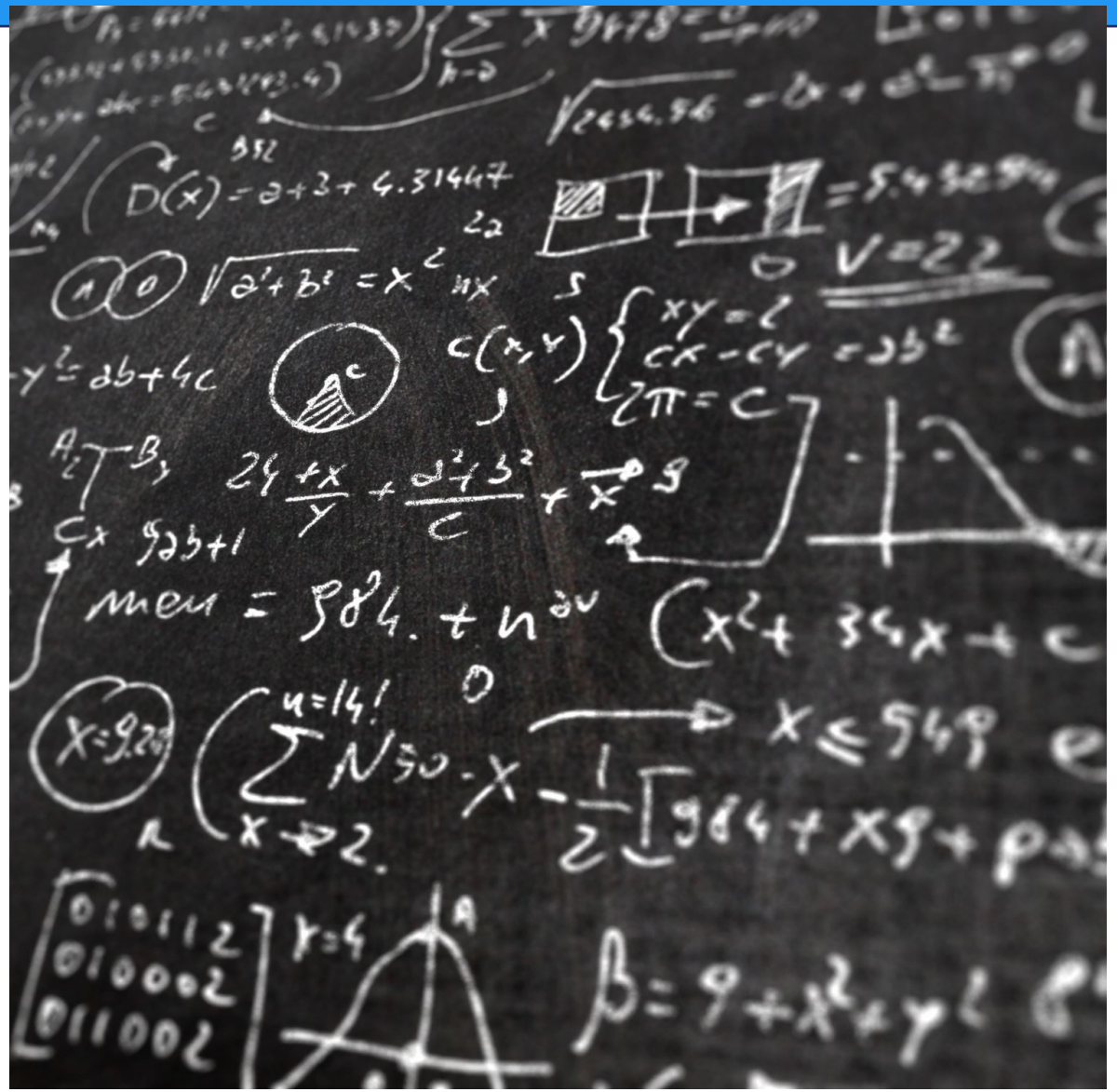
Problem: Moving Average From Data Stream

Given a stream of integers and a window size, calculate the moving average of all integers **in the sliding window**.



Explanation

```
MovingAverage movingAverage = new MovingAverage(3);
movingAverage.next(1); // return 1.0 = 1 / 1
movingAverage.next(10); // return 5.5 = (1 + 10) / 2
movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3
movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3
```

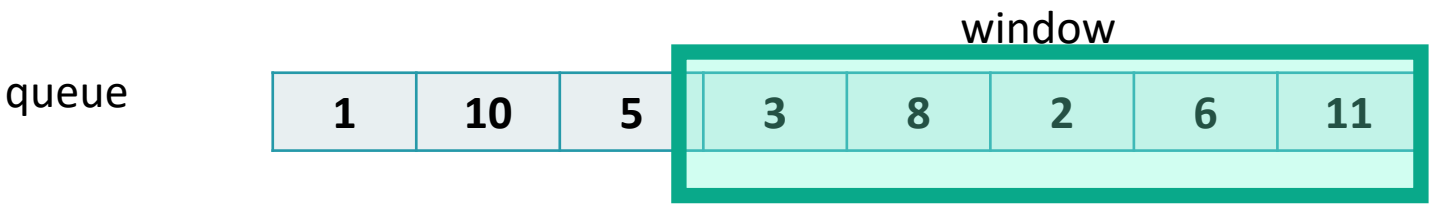


Let’s solve the problem first…

Approach 1:

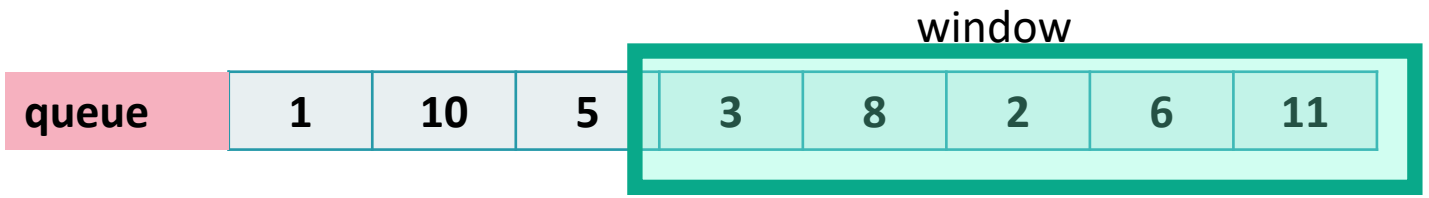
How to approach this?

- We could keep track of all the incoming values with the data structure of an *Array* (or *List*.)
- Then from the data structure, later we retrieve the necessary elements to calculate the average.



Algorithm

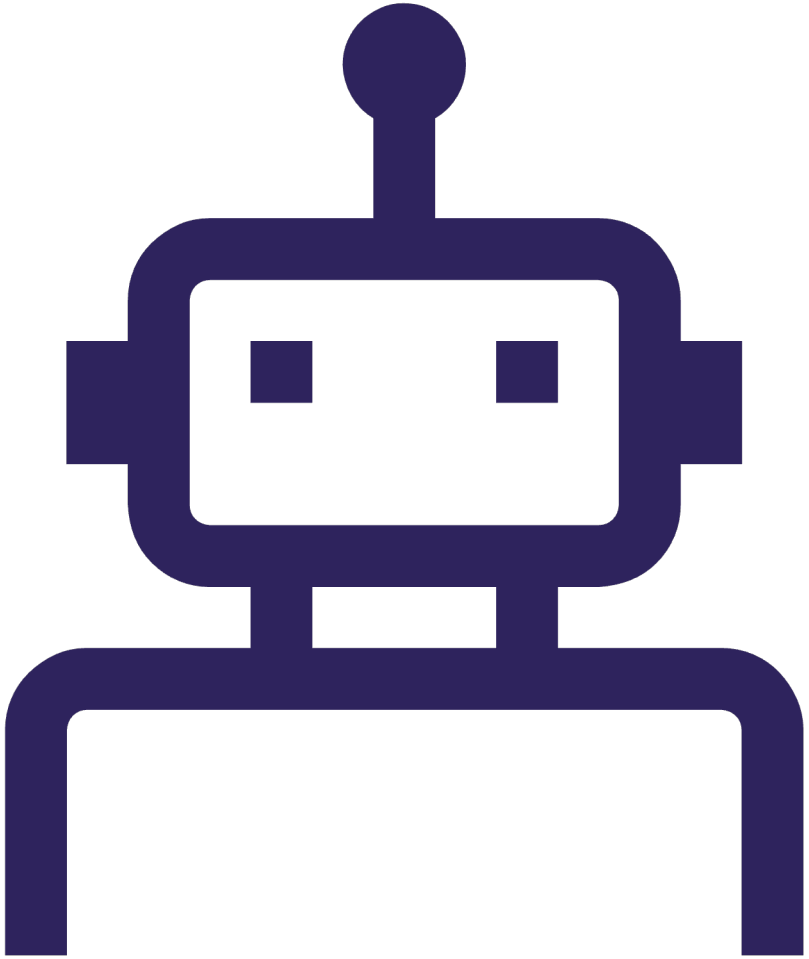
- First, we initialize a variable queue to store the values from the data stream, and the variable n for the size of the moving window. (We create our queue by using a Python's List.)
- At each invocation of **next**(val),
  - We first append the value to the queue.
  - We then retrieve the last n values from the queue, in order to calculate the average.



Run time complexity of this approach is O(N)
N is the size of the moving window, since we need to retrieve N elements from the queue at each invocation of next(val) function.

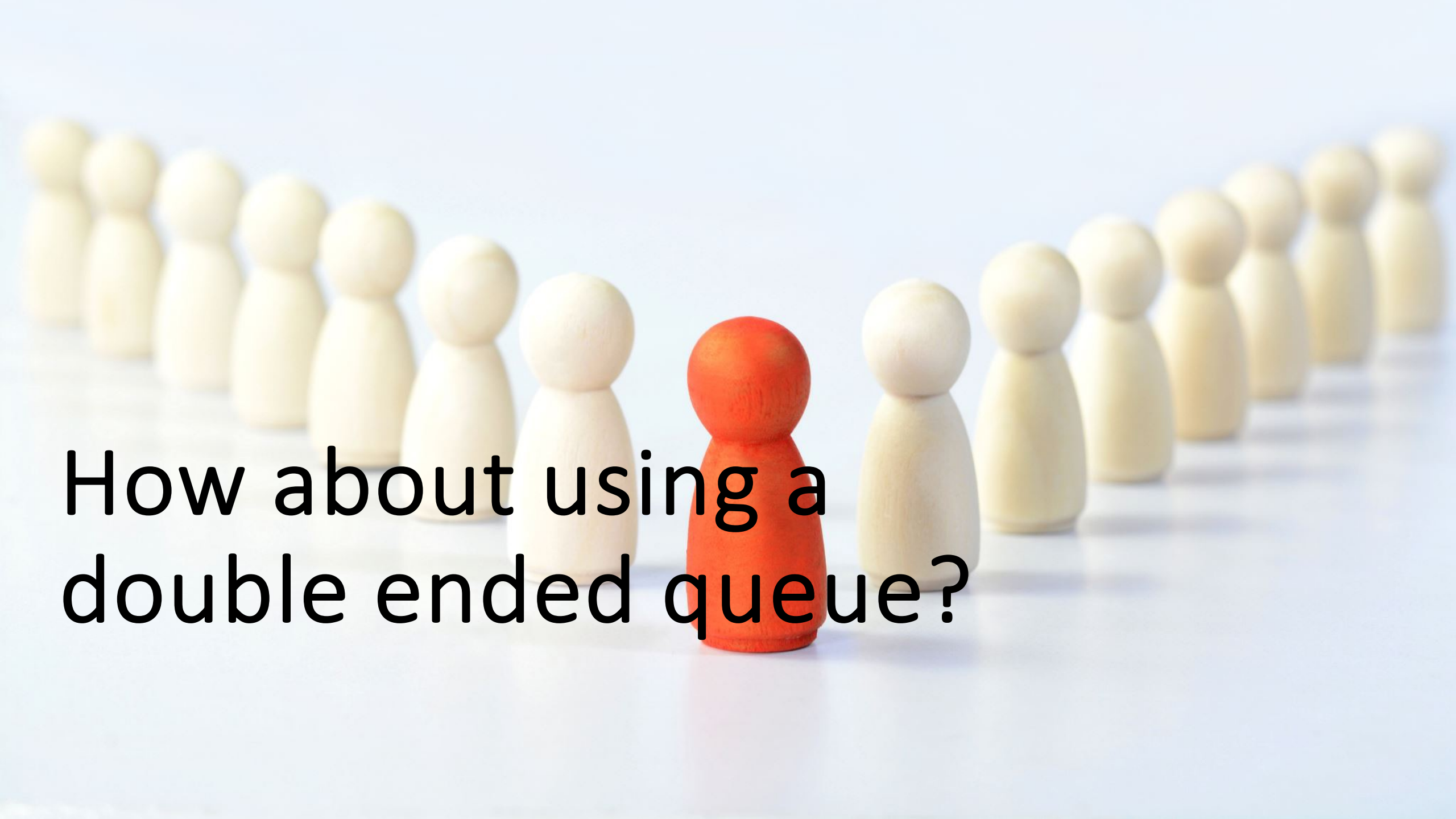
Let’s go to work Using a Queue

```python
class MovingAverage1:

    def __init__(self, window_size: int):
        self.window_size = window_size
        self._data = []

    def next(self, val: int) -> float:
        # window_size, self._data = self.window_size, self.queue

        self._data.append(val)
        # calculate the sum of the moving window
        pass
```



Can we improve the run time complexity?

Aim for O(1)?


How about using a
double ended queue?

Approach 2 : Using *Double-ended Queue*

Step1: We do not need to keep all values from the data stream, we only need the **last n values** which falls into the moving window.

Approach 2 : Using a *Double-ended Queue (Deque)*

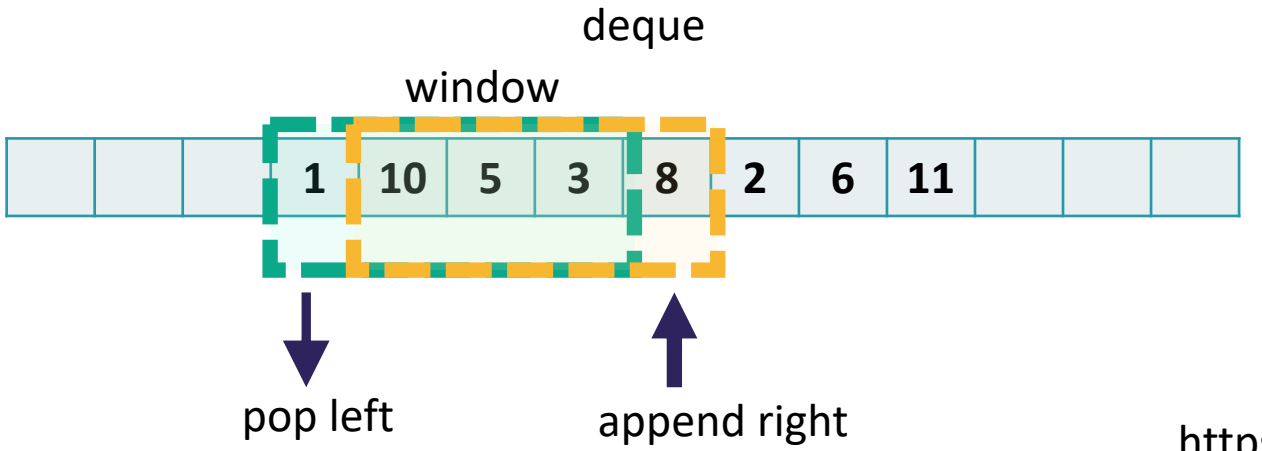
Step2: We know that at each step, we add a new element to the window, and at the same time we remove the oldest element from the window.

Here, we could apply a data structure called double-ended queue (a.k.a **deque**) to implement the moving window, which would have the constant time complexity O(1) to add or remove an element from both its ends.

Approach 2 : Use *Double-ended Queue : Deque*

Step3: In order to calculate the sum, we do not need to reiterate the elements in the moving window.

- We could keep the sum of the previous moving window, then in order to obtain the sum of the new moving window,
- we simply add the new element and deduce the oldest element. With this measure, we then can reduce the time complexity to constant.



https://docs.python.org/2.5/lib/deque-objects.html

Let’s go to work
Using Deque

```python
class MovingAverage2:
    # About dequeue:
    # https://docs.python.org/2.5/lib/deque-objects.html
    def __init__(self, size: int):
        self.size = size
        self.queue = deque()
        # number of elements seen so far
        self.window_sum = 0
        self.count = 0

    def next(self, val: int) -> float:
        self.count += 1
        # calculate the new sum by shifting the window
        self.queue.append(val)
        #calculate the new sum by shifting the window
        #if the count exceeds the size, popleft ( means remove from front)

        pass
```