Technique: Sliding Window

**Level: Easy**

**Given an array of integers A, find the sum of each sliding window of size K.**

**Variation:** Instead of an array, what if you were presented with a stream of numbers. A new number can be added anytime. You want to find the sum of the last K elements.

Note: The above problem can be solved without a Queue as well - just maintain a sum with 2 pointers. However, a queue is needed for the Variation problem. We use this problem to illustrate the sliding window concept.

Questions to Clarify:
Q. Is K provided as input?
A. Yes.

Q. How do you want the output?
A. Print the sum of each sliding window.

Solution:
Whenever you see a sliding window problem, you should think of using a queue.

We keep adding elements to the queue until it is of size K. To add another element, we remove the element from the back and add the new element to the front, maintaining the size K of the queue.

Every time we add an element, we add its value to the sum. When we remove an element, we subtract its value from the sum. That way, the sum always contains the sum of the sliding window of size K.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few lines quickly to plan out your solution. Your actual solution should be in a real language and use good syntax.)

init Queue
init sum = 0
loop i through A:
     if (Queue size is K)
         remove element E from front
         sum = sum - E
     add A[i] to Queue
     sum = sum + A[i]
     if (Queue size is K)
         print sum to output
```

Test Cases:
Edge Cases: array empty, K is 0, K is 1
Base Cases: K is 2, array size is 1
Regular Cases: K is 3 or more, K is less than array size

Time Complexity: O(n)

Space Complexity: O(K), because we store at most K nodes in the queue

```
public static void slidingWindowSum(int[] a, int k) {
    if (a == null || k == 0 || a.length == 0)
        return; // confirm with interviewer what to do for this case

    // LinkedList implements Queue interface in Java
    Queue<Integer> q = new LinkedList<>();
    int sum = 0;
    for (int i = 0; i < a.length; i++) {
        if (q.size() == k) {
            int last = q.remove();
            sum -= last;
        }
        q.add(a[i]);
        sum += a[i];
        if (q.size() == k) {
            System.out.println(sum);
        }
    }
}
```

**Level: Medium**

**You are given stock prices and the corresponding day of each stock price.**

For example:
`(32, 1), (45, 1), (37,2), (42,3), ..`

**Here, 32 is the price and 1 is the day of the price.**

**Say you are given these prices as an input stream. You should provide a function for the user to input a stock price and day. Your system should be able to tell the maximum stock price in the last 3 days.**

Questions to Clarify:
Q. Can the user add a new stock price anytime?
A. Yes, the user can add a new price anytime using your function

Q. Can we assume that the user will provide the stock price for today?
i.e, will the stock prices be sorted by day?
A. Yes, you can assume that.

Q. Let's say we get a price on day 4. Should the window be of prices o day 2,3 and 4?
A. Yes

Q. If there is no price in the system, what should the max() function return?
A. Return 0.

Solution:
We maintain a sliding window, where each price in the window is within the last 3 days.

Every time a new stock price comes in, we add it to the sliding window and remove all elements that exceed 3 days from the back.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few lines quickly to plan out your solution. Your actual solution should be in a real language and use good syntax.)

init queue

addPrice(price, day)
     while back of queue is less than day-2
           remove the front of the queue
     add (price, day) to queue

getMax()
     find max of the queue by going through it.
```

Test Cases:
Edge Cases: empty price, empty day
Base Cases: empty queue, 1 item in queue
Regular Cases: no element in past 3 days, adding on new day

Time Complexity:

**`addPrice()`** - O(n) worst case, but O(1) amortized because we do N removals for N insertions
**`findMax()`** - O(n) because we go through entire sliding window. We can reduce this to O(1) by maintaining a Queue with Max, which we cover in the next section.

Space Complexity: O(n)

```
public class StockPriceWithTime {
    Queue<Price> q;
    int window;

    public StockPriceWithTime(int windowDays) {
        q = new LinkedList<>();
        window = windowDays;
    }

    public void addPrice(int price, int day) {
        while(!q.isEmpty() && q.peek().getDay() < (day - window + 1))
            q.remove();

        q.add(new Price(price, day));
    }

    // Returns max price in last 3 days
    public int getMax() {
        int maxPrice = 0;
        Iterator<Price> iter = q.iterator();
        while (iter.hasNext()) {
            int price = ((Price) iter.next()).getPrice();
            if (price > maxPrice)
                maxPrice = price;
        }
        return maxPrice;
    }
}


/*
 * Helper Code. Ask the interviewer if they want you to implement this.
 */
```

```
public class Price {
    int price;
    int day;

    public Price(int price, int day) {
        this.price = price;
        this.day = day;
    }

    public int getPrice() {
        return price;
    }

    public int getDay() {
        return day;
    }
}
```