Sample Problems

Problem #1 : Reversing the first K elements of a Queue

Description - Given an integer k and a queue of integers, we need to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.
Only the following standard operations are allowed on the queue.

- enqueue(x) : Add an item x to rear of queue
- dequeue() : Remove an item from front of queue
- size((: Returns number of elements in queue.
- front() : Finds front item.

```
Input : Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
        k = 5
Output : Q = [50, 40, 30, 20, 10, 60, 70, 80, 90, 100]
```

Solution - The idea is to use an auxiliary stack and follow these steps to solve the problem -

1. Create an empty stack.
2. One by one dequeue items from a given queue and push the dequeued items to stack.
3. Enqueue the contents of stack at the back of the queue.
4. Reverse the whole queue.

Pseudo Code

```
/* Function to reverse the first K elements of the Queue */
void reverseQueueFirstKElements(k, Queue)
{
    if (Queue.empty() == true || k > Queue.size())
        return
    if (k <= 0)
        return
    stack  Stack
    /* Push the first K elements into a Stack*/
    for ( i = 1 to k) {
        Stack.push(Queue.front())
        Queue.pop()
    }
    /* Enqueue the contents of stack
       at the back of the queue*/
    while (!Stack.empty()) {
        Queue.push(Stack.top())
        Stack.pop()
    }
    /* Remove the remaining elements and
       enqueue them at the end of the Queue*/
    for (int i = 0 to i < Queue.size() - k) {
        Queue.push(Queue.front())
        Queue.pop()
    }
}
```

Time Complexity : O(n) , n : size of queue
Auxiliary Space : O(k)

Problem #2 : Sliding Window Maximum

Description - Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

```
Input :
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
k = 3
Output :
3 3 4 5 5 5 6
```

Solution : We create a Deque, Qi of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain Qi to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the Qi is the largest and element at rear of Qi is the smallest of current window.

```
void printKMax(arr[], n, k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    deque < int > Qi(k)

    /* Process first k (or first window) elements of array */
    for (i = 0; i < k; ++i) {
        // For every element, the previous smaller elements are useless so
        // remove them from Qi
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back() // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i)
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for (; i < n; ++i) {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        print (arr[Qi.front()])

        // Remove the elements which are out of this window
        while ((!Qi.empty()) && Qi.front() <= i - k)
            Qi.pop_front() // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back()

        // Add current element at the rear of Qi
        Qi.push_back(i)
    }

    // Print the maximum element of last window
    print (arr[Qi.front()])
}
```