Understanding the Problem Statement

Consider an array [10,4,2,11,3,15,12] and k= 3. Now for every continuous subarrays of size 3, i.e [10,4,2], [4,2,11], [2,11,3], [11,3,15], [3,15,12] we have to find maximum and print them i.e 10,11,11,15,15.

Approach 1

Find subarray by increasing index and find maximum in the subarray using linear search. If array size is n, then we will have (n-k+1) subarrays of size k. In each subarray of size k, to finding max using linear search takes O(n) time. Therefore O(n-k+1) x O(k) = O(nk) Space Complexity takes O(1) because we are not using any extra space

Time Complexity: O(nk) Space Complexity: O(1)

We cannot reduce O(n) i.e no of subarrays but we can reduce O(k) i.e finding max using different datastructure other than array. May be BST or maxheap.

Approach 2

Now instead of using array in above approach, if we use Binary Search Tree? When we slide from one window to another window of size k, we have to find maximum, delete an element and insert new element. Now we will have BST of size k. finding maximum takes O(logk),deleting an element takes O(logk) and inserting new element takes O(logk). This is only incase if BST is balanced binary search tree or else delete takes O(n). so O(n) x O(logk+logk+logk) so total complexity is O(nlogk) and space complexity is O(k) because we are using binary search tree of size k.

BST : Time Complexity : O(nlogk) Space Complexity : O(k)

Now instead of using array and BST if we use Max heap, finding maximum takes O(1), deleting an element takes O(k) because we have to search element first in entire tree of size k and then delete actually it takes O(klogk), and inserting an element takes O(logk). So Total time complexity is O(nk). Space complexity is O(k) for tree.

Max Heap: Time Complexity : O(nk) Space Complexity : O(k)

Approach 3

1.First create Double Linked List with first three elements. 2.While inserting element, we have to see whether previously existing elements are still useful or else we have to delete them For example we have 11,2,3 in DLL. Now we are planning to insert 9. If we insert 9, 2 and 3 will no longer be useful so we can delete them . Now DLL will contain only 11 and 9. 3.Maximum element will always be the first element in DLL i.e 11. 4.Whenever we move to next window, starting element of previous window has to be deleted. i.e 11,2,3,9 if we go to 2,3,9 subarray then 11 should be deleted. Now to findout 11 is present in DLL or not,we store indices of the elements in DLL not the elements themselves 5.We repeat this process of removing useless elements from last and removing first element if we move to next window, and printing maximum i.e first element in DLL. 6.So each element will be inserted and deleted from DLL once. Insertion and deletion takes O(1). So Total Time Complexity is O(n) and space complexity is O(k) for DLL. Note: we may think that why to maintain all k elements in DLL. just store maximum in window like 10. if new element is greater than 10, print that or else print 10 and the continue windowwise. But if we have 10,9,8,2. we just store 10 and compare with 2? 9 can be maximum in 9,8,2. So we have to store all potential maximum elements in window k. Maximum k and minimum 1. In this case we store 10,9,8 suppose if we have 8,9,10,2 the we store only 10 as per algorithm because once we have 10,9 and 8 becomes useless. Time Complexity : O(n) Space Complexity: O(k)

Implementation

In [1]:
```
class Node:
    def __init__(self,data):
        self.prev = None
        self.data = data
        self.next = None

    @staticmethod
    def createSampleNode(k):
        newnode = Node(k)
        return newnode
```

In [2]:
```
def addFirst(data):
    global head,tail
    temp = Node.createSampleNode(data)
    if head==None:
        head=tail=temp
    else:
        temp.next = head
        head.prev = temp
        head = temp

def addLast(data):
    global head,tail
    temp=Node.createSampleNode(data)
    if head==None:
        head=tail=temp
    else:
        tail.next = temp
        temp.prev = tail
        tail = temp

def removeFirst():
    global head,tail
    if head==tail:
        temp = head
        head = tail = None
        del(temp)
    else:
        temp = head
        head = head.next
        head.prev = None
        del(temp)

def removeLast():
    global head,tail
    if head==tail:
        temp = head
        head = tail = None
        del(temp)
    else:
        temp = tail
        tail = tail.prev
        tail.next = None
        del(temp)

def peekFirst():
    global head,tail
    return head.data

def peekLast():
    global head,tail
    return tail.data

def isListEmpty():
    if head==None:
```

```
        return 1
    return 0

def printFirst():
    global head,tail
    if head!=None:
        return head.data

def traverseDoubleLinkedList(a):
    temp = a
    while(temp):
        print(temp.data)
        temp = temp.next
```

In [3]:
```
head = tail = None

def slidingWindowMax(arr,n,k=3):
    addFirst(0)
    for i in range(1,k):
        while( not isListEmpty() and arr[i]>=arr[peekLast()]):
            removeLast()
        addLast(i)
    for i in range(k,n):
        print(arr[printFirst()],end=",")
        if peekFirst() == i-k:
            removeFirst()
        while( not isListEmpty() and arr[i]>=arr[peekLast()]):
            removeLast()
        addLast(i)
    print(arr[printFirst()])
```

In [4]:
```
arr = [10,4,2,11,3,15,12]
slidingWindowMax(arr,7,3)
# ans 10,11,11,15,15
```

```
10,11,11,15,15
```

In [5]:
```
arr=[15,9,4,17,18,12,6,26,27,16,1,12,14,21,35,29]
slidingWindowMax(arr,16,4)
# ans 17,18,18,18,26,27,27,27,27,16,21,35,35
```

```
17,18,18,18,26,27,27,27,27,16,21,35,35
```

In [ ]: