Find maximum/minimum subarray sum of size k;

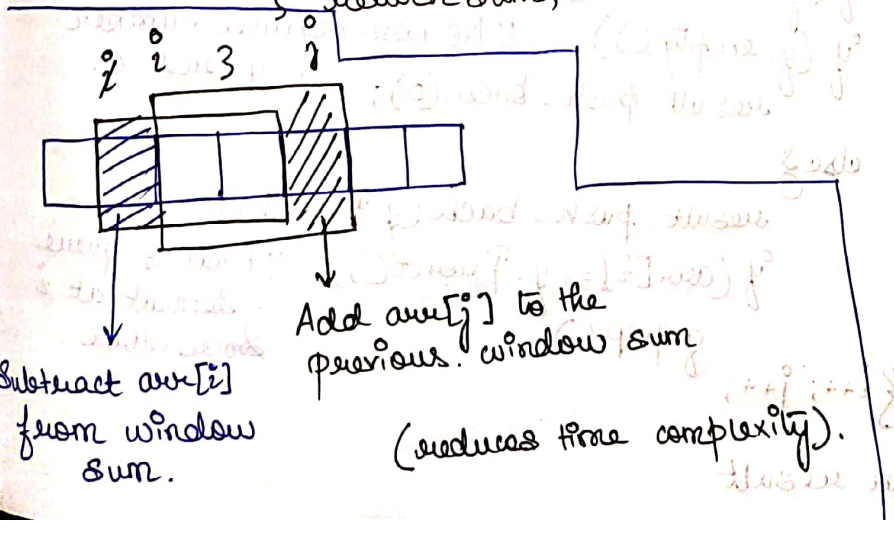
Problem Statement – Given an array of integers of size N and a window of size k;
Return max/min sum of a subarray of size k.

— x —

Code

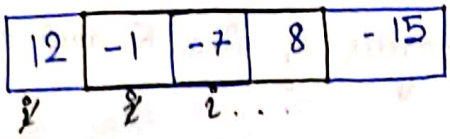
```
int maxSubarray(int* arr, int k, int N){
    int maximum = INT_MIN;
    int sum = 0;
    int i=0, j=0;   // start and end index of window.

    while(j < N){   // until end index pointer reaches the end.
        sum += arr[j];
        if(j-i+1 < k)   // end index does not reach window length
            j++;        // increment end pointer
        else if(j-i+1 == k){
            maximum = max(maximum, sum);
            sum = sum - arr[i];
            i++; j++;
        }
    }
    return sum;
```



(reduces time complexity).

First Negative number in every window of Size K.



30 16 28 ...

O/P ← -1 -1 -7 -15 -15

Queue → -1 -7 -15

Removing i ← As soon as i matches with the element at the front of the queue pop it. Otherwise keep it. [If element at i is +ve; don't do anything]

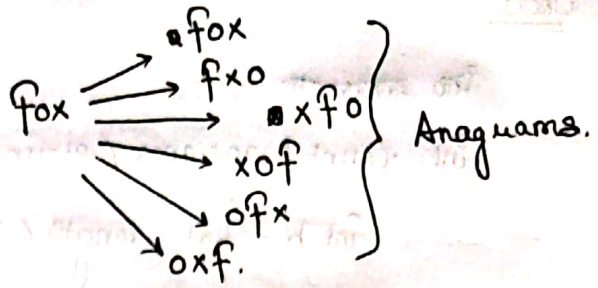
Adding j ← Every non-negative integer is pushed inside the queue.

Code

```
vector<int> firstNegative(int* arr, int n, int k){
    int i=0, j=0;
    vector<int> result;
    queue<int> q;
    while(j<n){
        if(arr[j]<0)
            q.push(arr[j])        // enqueuing every -ve number to the queue
        else if(j-i+1<k)
            j++;
        else if(j-i+1==k){
            if(q.empty())         // No non-negative number append 0.
                result.push_back(0);
            else{
                result.push_back(q.front())
                if(arr[i]==q.front())   // front of queue = element at i So we move.
                    q.pop()
            }i++; j++;
    }
    return result.
}
```

Count occurrences of Anagrams in a text.

Anagrams. → Permutations of a String



Algorithm:-
1) The length of the pattern is the window size. (Slide over the text)
2) Create a hashmap with each character of the pattern.
3) Keep a count variable // number of chars in the pattern.
count = hashmap.size();

4) Use String Matching.

a: 2 1 0
b: 1

For each character match reduce its frequency by 1.
As soon as the freq. of a character becomes 0;
reduce count by 1. (∵ that character Quantity already matched)

Adding j →

Removing i → If removing element is found in map increase its frequency by 1.
(If freq. == 1; increase count by 1).

If count == 0; anagram is found.
Increase countofanagrams.

This algorithm can be used as a String matching algorithm and used in any problem.

(Separate code written in C# • Divine: Top Int Algorithms)

Code

```
int CountAnagrams (string pat, string txt){
    int k = pat.length();          // Window length.
    int i=0, j=0;
    map<int, int> hash;            // Freq. Map of characters
    for(int i=0; i<pat.length(); i++){
        if (hash.find(pat[i]) == hash.end())
            hash[pat[i]] = 1;
        else
            hash[pat[i]]++;
    }
    int count = hash.size();
    int result = 0;                // Stores count of anagrams.

    while (j < txt.length()){      // String matching
        if (hash.find(txt[j]) != hash.end()){ // Found
            hash[txt[j]]--;
            if (hash[txt[j]] == 0)
                count--;
        }
        if (j-i+1 < k)
            j++;
        else if (j-i+1 == k){
            if (count==0)
                result++;
            if (hash.find(txt[i])!=hash.end()){
                hash[txt[i]]++;
                if (hash[txt[j]] == 1)
                    count++;
            }
            i++; j++;
        }
    }
    return result;
}
```

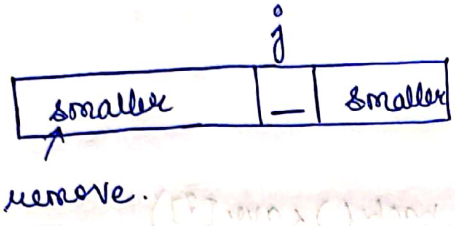
// Removing i and moving forward

// Checking = 1 bcoz only if the character count is 1 it has come and count increased from before. If char is present no need to increase char count

Maximum of all Subarrays of size k.

Given an array arr[] of size N and an integer k. Find the maximum for each and every contiguous subarray of size k.

— x —



While adding element at index j to the deque.
Check for elements to the left of j.
Smaller elements to the left of element at j index can never be a candidate to become maximum for that window so is removed.

However smaller elements to the right have probability to be max. in other windows so added.

Adding element at j → Every element of the array is added to the deque. However; before adding we check for elements less than it in the deque and remove them.

This ensures that the maximum for the window always resides at the front of the deque.

We need to access elements from other sides of the queue (back and front). So we need a double ended queue.

Removing i → If arr[i] == front. of the deque.
we pop the element from the deque
and increment i.

Code

```
vector<int> max_of_subarrays(int* arr, int n, int k){
    int i=0, j=0;
    vector<int> result;
    deque<int> dq;

    while(j<n){
        while(dq.size()>0 && dq.back()<arr[i])
            dq.pop_back();
        dq.push_back(arr[j]);
        if (j-i+1<k)
            j++;

        else if (j-i+1==k)
            result.push_back(dq.front());
            if (arr[i]==dq.front())
                dq.pop_front();

            i++; j++;
        }
    }
    return result;
}
```