Pseudocode:

A: O(n)

```
create array Largest[size of input];
For i = 0 to array.size: O(n)
        If (i<K) Largest[i] = Input[i];
        else Largest[i] = max(Largest[i-1], Largest[i-1-K] + value[i], 0+value[i]);
return Largest[value.size()-1];
```

B: O(n)

```
create array new_Input[size of input] ;
if(value[i]>=0) new_value[i] = value[i];
for    i=0 to input size:
            if(i<K+1): sum += new_value[i];
            else: sum = sum - new_value[i-K-1] + new_value[i];
            if(sum >maximum): maximum=sum;
return maximum;
```

C: O(n)

```
create array Largest[size of input];
create array Smallest[size of input];
For i = 0 to array.size: O(n)
        If (i<K):
                Largest[i] = Input[i];
                Smallest[i] = Input[i];
        Else:
                Largest[i] = max(largest[i-1], Largest [i-1-K] * value[i], Smallest [i-1-
                K+1]*value[i], 1*value[i]);
                Smallest[i] = min(Smallest [i-1], Smallest [i-1-K] * value[i], Largest[i-1-
                K+1]*value[i], 1*value[i]);
return largest[value.size()-1];
```

Main: O(m*n)

```
create array Input[size of input];    array_size = n , input_line = m
Case "A": Do function A(); O(n)
Case "B": Do function B(); O(n*K) = O(n)    K is const.
Case "C": Do function C(); O(n)
Else : update Input array; O(n)
```

Time complexity analysis:

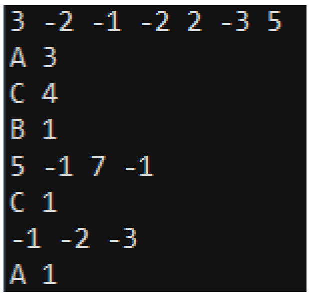
Assume:

Input file has m lines, and a input array has n numbers.

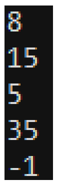
Time Complexity = **O(m*n)**

Experimental results:

Example:

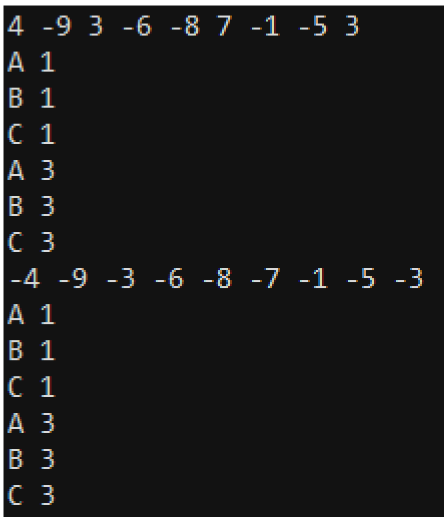
Input: 

```
3 -2 -1 -2 2 -3 5
A 3
C 4
B 1
5 -1 7 -1
C 1
-1 -2 -3
A 1
```

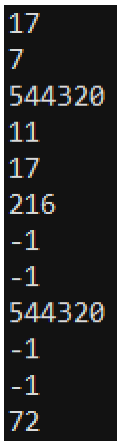
Output: 

```
8
15
5
35
-1
```

My own data:

Input: 

```
4 -9 3 -6 -8 7 -1 -5 3
A 1
B 1
C 1
A 3
B 3
C 3
-4 -9 -3 -6 -8 -7 -1 -5 -3
A 1
B 1
C 1
A 3
B 3
C 3
```

Output: 

```
17
7
544320
11
17
216
-1
-1
544320
-1
-1
72
```

Dynamic Programming v.s. Recursion:

Case A:

DP: O(n)

$\underline{f(n) = \max[f(n-1), f(n-K) + input(n)]}$

Recursion: $O(2^n)$

$f(n) = \underline{\max[f(n-1), f(n-K) + input(n)]}$

⇨ **$f(n-1) = \underline{\max[f(n-2), f(n-K-1) + input(n-1)]}$**

$f(n-K) = \underline{\max[f(n-K-1), f(n-K-K) + input(n-1)]}$

⇨ **$f(n-2), f(n-K), f(n-K-1), f(n-2K) \Rightarrow \ldots = O(2^n)$**

Case B:

DP: O(n)

$\underline{f(k)(n) = \max[f(k-1)(n-1), f(k-1)(n), f(k-1)(n-1)+input(n-k), f(k-1)(n-1)+ input(n)]} = f(n)-input(n-k)+ input(n)$

Recursion: $O(4^n)$

⇨ **$\underline{f(k-1)(n-1), f(k-1)(n), f(k-1)(n-1), f(k-1)(n-1)} \Rightarrow \ldots = O(4^n)$**

Case C:

DP: O(n)

f(n) = max[f(n-1), f(n-K) * input(n), min[f(n-1), f(n-K) * input(n)]*input(n)]

Recursion: $O(2^n)$

⇨ **f(n-1), f(n-K), f(n-1), f(n-K) => … => $O(4^n)$ = $O(2^n)$**

Dynamic programming method is much more faster than recursion method because it doesn’t need to compute the overlapping part. DP could read the record at the table so that it could reduce the calculation amount.