2nd LGE Code Jam 2019 - Problem C(English)

Created by 박선현 sunhyun.park, last modified by 신동철 dc.shin on 2019/07/11

Stock Market

This is a problem to find the best suitable minimum length of range which has sum greater or equal than some number.

The search space N is maximum 10^5 size, and time limit is 1 second.

Brute Force - O(N^2)

This approach is searching every range which has sum greater or equal than given number.

We can use prefix sum data structure to compute sum of continuous range.

By using prefix sum data structure, we can get the sum of continuous range in array at O(1) time complexity.

```
sum of range [i,j] = prefix_sum [j] - prefix_sum [i-1]
```

Actual benefit can get for each day can be obtained by subtracting purchasing price from selling price.

We're using this value to compute actual benefit and make prefix sum.

The search space limitation for subtask1 is N<=5000. And this is small enough for naive O(N^2) approach to solve it within given time bound.

C-sub1-solution.cpp

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using ll = long long;
using pii = std::pair<int, int>;

using namespace std;

void solve() {
    int n, current_cash, cost;
    ll profit_goal;
    cin >> n >> current_cash >> cost >> profit_goal;

    vector<ll> profit(n+1, 0);
    for (int i = 1; i <= n; ++i) {
        cin >> profit[i];
        profit[i] += profit[i - 1] - cost;
    }

    int can_have = 0;
    if (cost) {
        can_have = current_cash / cost;
    } else {
        can_have = n;
    }

    pii best = {n + 1, 0};
    for (int i=0; i<n; ++i) {
        for (int j=i+1; j<=n; ++j) {
            if (profit[j] - profit[i] >= profit_goal) {
                best = min(best, pii(j - i, -i));
            }
        }
    }

    if (best.first > can_have) {
        cout << -1 << endl;
    } else {
        cout << -best.second + 1 << " " << -best.second + best.first << endl;
    }
}

int main(int argc, char* argv[]) {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int tcs;
    cin >> tcs;

    while (tcs-- > 0) solve();
}
```

C-sub1-solution.java

```java
import java.util.*;

public class Main {
    private static int T;
    private static int N;
    private static long X; // money
    private static long Y; // price
    private static long Z; // min

    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        T = sc.nextInt();
        for (int i = 0; i < T; i++) {
            N = sc.nextInt();
            X = sc.nextLong();
            Y = sc.nextLong();
            Z = sc.nextLong();
            long[] Vs = new long[N+1];
            for (int j = 1; j <= N; j++) {
                Vs[j] = Vs[j - 1] + sc.nextLong() - Y;
            }
            solve(Vs);
        }
        sc.close();
    }

    private static void solve(long[] Vs) {
        long start = 0;
        long range = N + 1;
        for (int i = 0; i <= N; i++) {
            for (int j = i + 1; j <= N; j++) {
                if (Vs[j] >= Vs[i] + Z) {
                    if (range > j - i || (range == j - i && i >= start)) {
                        start = i + 1;
                        range = j - i;
                    }
                }
            }
        }
        if (range == N + 1) {
            System.out.println("-1");
        } else {
            System.out.println(start + " " + (start + range - 1));
        }
    }
}
```

Using sliding window and 2-pointers - O(N)

Search space limitation for subtask2 is N<=10^5. O(N^2) algorithm may have time limit exceeding failure.

At least, O(N log N) or faster algorithm can be a solution.

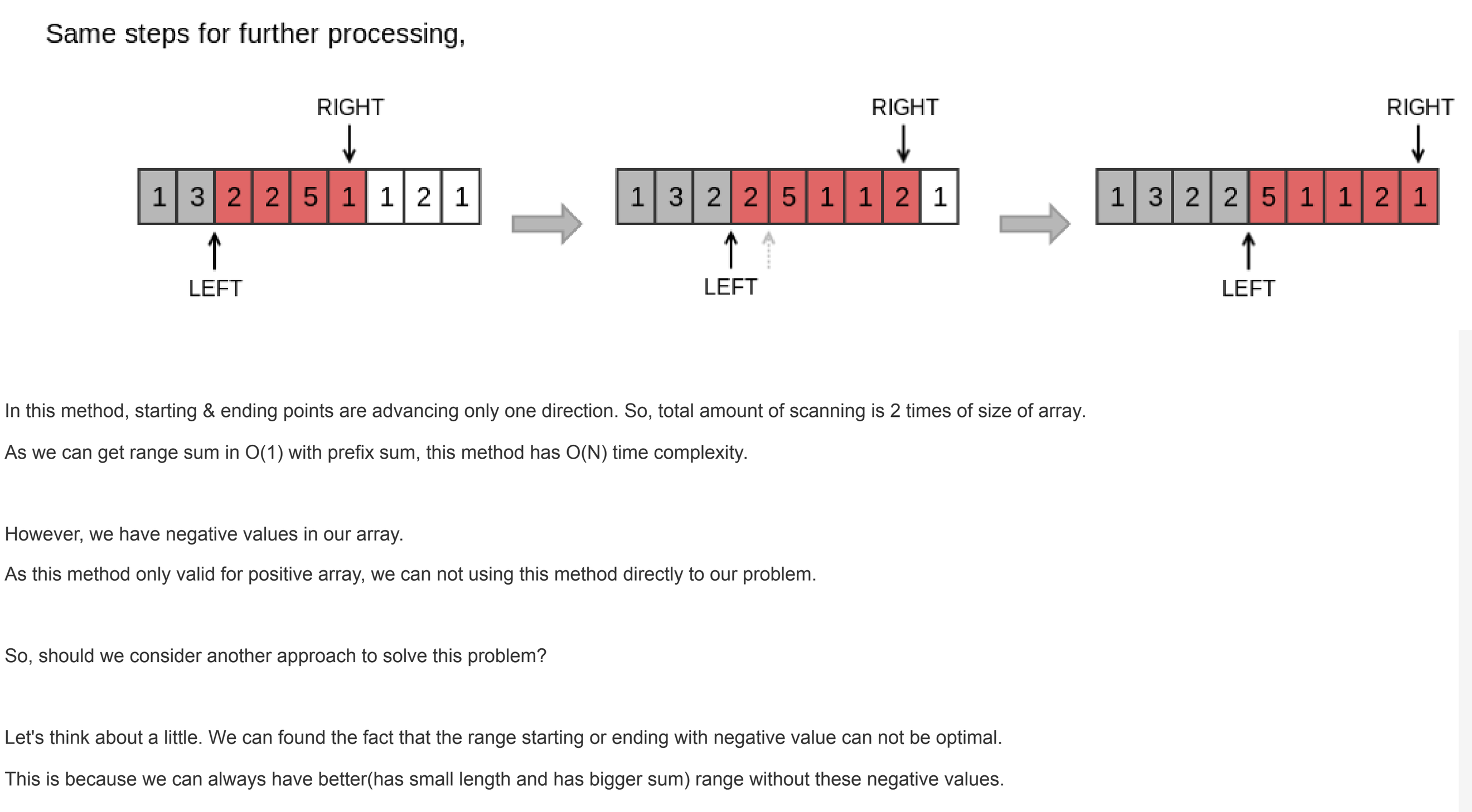
Is there a way to find the possible range with this time complexity? There is a method which called "Sliding Window".

"Sliding Window" is a method to compensate time complexity only checks the differences while sliding interesting range.

In general, sliding window use fixed size window to compute range operations,

However in this problem, we're going to use variable length sliding window with two pointers(these pointers refer starting & ending accordingly) technique.

Below is an example of finding range which has sum at least 10 using slinding window with two pointers.



In this method, starting & ending points are advancing only one direction. So, total amount of scanning is 2 times of size of array.

As we can get range sum in O(1) with prefix sum, this method has O(N) time complexity.

However, we have negative values in our array.

As this method only valid for positive array, we can not using this method directly to our problem.

So, should we consider another approach to solve this problem?

Let's think about a little. We can found the fact that the range starting or ending with negative value can not be optimal.

This is because we can always have better(has small length and has bigger sum) range without these negative values.

Let's checks some more details in this point of view.

(It is very important to check conditions for an algorithm before use it)

> **About starting point *x*, Ending point *y* and prefix sum array *P***
>
> **- optimal(y) : x for x<=y and the smallest length [x,y] with P[y]-P[x-1]>=Z (P[x-1]<=P[y]-Z)**
>
> **- If x1<x2 and P[x2-1] <= P[x1-1], optimal(y) can not be x1.**
>
> **Because P[y] - P[x1-1] >= Z, shortly P[x1-1] <= P[y] - Z then,**
>
> **We can see P[x2-1] <= P[x1-1] <= P[y] - Z**
>
> **And in this case, x1 is a better choice comparing with x2 because it is smaller in size.**
>
> **So, if ending point is determined with y, we can only consider x for increasing P[x].**
>
> **- At the same point of view, the ending points y1 < y2,**
>
> **If optimal(y1)==x, then we don't need to reconsider x at checking y2(It is always [x,y1] is smaller in size)**

Generally speaking, when using sliding window with two pointers we should maintain monotonic queue while sliding.

In this problem, adding index to queue should always strictly increasing total amount of queue.

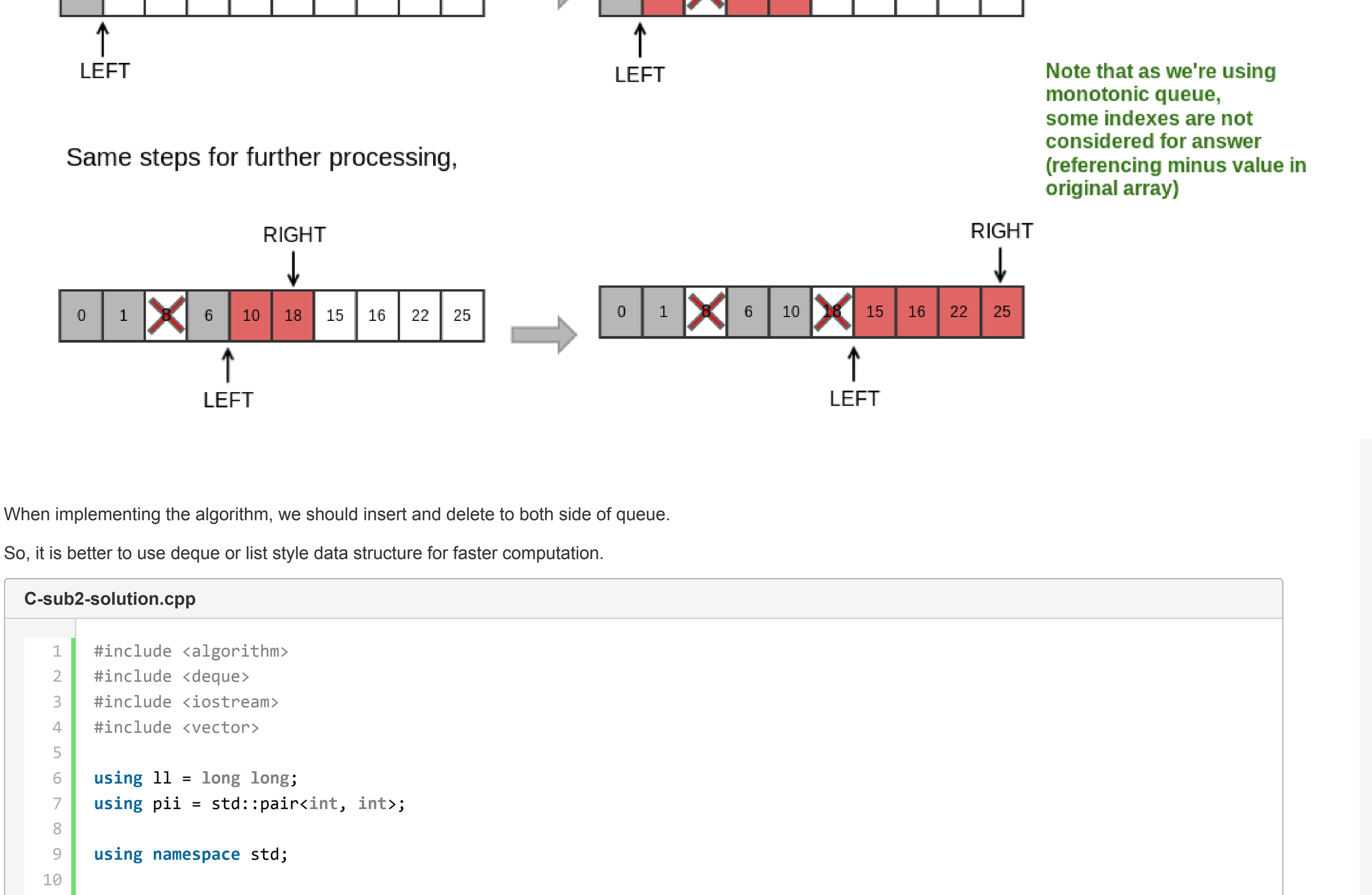
And removing index from window should always strictly decreasing total amount of queue as well.

If there're some indexes which are not appropriate for this, then we don't need to consider them.

We just remove them greedily, and maintaining monotonic queue while processing.

(As we are computing range sum with prefix sum array, removing some indexes between starting & ending doesn't have effect for this)

The workflow is shown below.



When implementing the algorithm, we should insert and delete to both side of queue.

So, it is better to use deque or list style data structure for faster computation.

C-sub2-solution.cpp

```cpp
#include <algorithm>
#include <deque>
#include <iostream>
#include <vector>

using ll = long long;
using pii = std::pair<int, int>;

using namespace std;

void solve() {
    ll n, current_cash, cost, profit_goal;
    cin >> n >> current_cash >> cost >> profit_goal;

    ll can_have = cost ? current_cash / cost : n;

    vector<ll> profit(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        cin >> profit[i];
        profit[i] -= cost;

        // making cumulative
        profit[i] += profit[i - 1];
    }

    pii best = {n + 1, 0};
    deque<ll> dq;
    for (int i = 0; i <= n; ++i) {
        while (!dq.empty() && profit[i] <= profit[dq.back()]) dq.pop_back();

        while (!dq.empty() && profit[i] >= profit[dq.front()] + profit_goal) {
            int l = dq.front();
            dq.pop_front();
            best = min(best, pii(i-l, -l));
        }

        dq.push_back(i);
    }

    if (best.first > can_have) {
        cout << -1 << endl;
    } else {
        cout << -best.second + 1 << " " << -best.second + best.first << endl;
    }
}

int main(int argc, char* argv[]) {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int tcs;
    cin >> tcs;

    while (tcs-- > 0) solve();
}
```

C-sub2-solution.java

```java
import java.util.*;

public class Main {
    private static int T;
    private static int N;
    private static long X; // money
    private static long Y; // price
    private static long Z; // min

    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        T = sc.nextInt();
        for (int i = 0; i < T; i++) {
            N = sc.nextInt();
            X = sc.nextLong();
            Y = sc.nextLong();
            Z = sc.nextLong();
            long[] Vs = new long[N+1];
            for (int j = 1; j <= N; j++) {
                Vs[j] = Vs[j - 1] + sc.nextLong() - Y;
            }
            solve(Vs);
        }
        sc.close();
    }

    private static void solve(long[] Vs) {
        long start = 0;
        long range = N + 1;
        Deque<Integer> deq = new LinkedList<>();
        for (int i = 0; i <= N; i++) {
            while (!deq.isEmpty() && Vs[i] <= Vs[deq.getLast()]) {
                deq.removeLast();
            }
            while (!deq.isEmpty() && Vs[i] >= Vs[deq.getFirst()] + Z) {
                int s = deq.removeFirst();
                if (range > i - s || (range == i - s && s >= start)) {
                    start = s + 1;
                    range = i - s;
                }
            }
            deq.addLast(i);
        }
        if (range == N + 1) {
            System.out.println("-1");
        } else {
            System.out.println(start + " " + (start + range - 1));
        }
    }
}
```