Another Solution of Problem Sequence

Sinya Lee

Jinshan High School, Swatow, China

Table of Contents

Analysis

Let's try to solve the problem using greedy algorithm.

We can imagine that each X_i and Y_i is a point on the number line. For example, the sample input can be displayed as **Figure 1**:

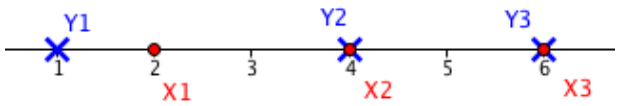

Figure 1

At first, let $Y_i = 1 + ((i - 1) * A)$, i.e. let each Y_i be in the leftmost places it can be. And then let's try to move them right to the best places Y_i should be. So we can design a transition T_i to represent moving Y_i to Y_N one unit to the right.

But actually T_i is not always available. We can't make more than $B - (1 + (i - 1) * A)$ transitions. And when $i > 1$, we can's make transition T_i more than $B - A$ times.

It can be easily proved that all the final states can be reached using this kind of transitions. We can make transition T_1 $(Y_1 - 1)$ time(s) and make T_i $(Y_i - Y_{i-1} - A)$ time(s) when $i > 1$.

So the problem is what transitions and in what order we need to make. In fact we can use greedy algorithm. We define P_i as the difference of the cost if we move Y_i 1 unit to the right. So the following formula is obvious.

$P_i = 1$ | *when* $Y_i \geq X_i$
$P_i = -1$ | *when* $Y_i < X_i$

Let's define $R_i = \sum_{j=i}^{N} P_i$, so R_i is the difference of the cost if we make transition T_i. So let's pick up the transition to minimize the difference of the cost, if it is negative.

Proof

We feel this algorithm is correct by intuition. But we need to prove it strictly. In the following we are trying to prove that every time we decided to make a transition T_x, any other transition T_i will not be better than T_x.

i. If $i < x$,
$\sum_{j=i}^{x-1} P_j$ is not negative, or R_x will be larger than R_i, and we will not choose T_x at all. On one hand, choosing T_i will not reduce the cost more than choosing T_x for now. On the other hand, if we choose T_i instead of T_x, some P_j $(i < j < x)$ may become *1* from *-1*, it won't be better for the future. So when $i < x$, choosing T_i will not be better than choosing T_x.

ii. if $i > x$,
$\sum_{j=k}^{i-1} P_k$ is not positive, or we will choose T_i instead of T_x. Let's define $d = \sum_{k=x}^{i-1} P_k$, so if we choose T_i instead of T_x, we will lose d unit(s) of cost for now. Since for all $x \leq j \leq i - 1$, $\sum_{k=j}^{i-1} P_k \leq k$, or we will choose T_j instead of T_x. It means that after we choose T_i for now, we won't reduce the cost more than k unit(s) in the future. So when $i > x$ choosing T_i won't be better than choosing T_x.

So we have proved that the greedy algorithm is correct.

Using Segment Trees

Let's try to analyze the time complexity. If we use the algorithm directly, every time we try to transform the state, we need $O(N)$ time to find the minimum R_i, $O(N)$ time to renew each P_i and R_i. And we may make transition for at most $O(Q)$ times. So the total time complexity is $O(N*Q)$, which is obviously too bad.

In fact, if we use segment trees, we can find the minimum R_i in $O(log\ N)$, we can also renew the P_i and R_i with lazy propagation. So the problem we have to face now is the factor Q. So we need some observations to move it away. We notice that if nothing happens after making a transition, the decision next time will be the same. That means we can merge these transitions together. We can use an array store how many times we can make transition T_i. To find how many same transitions T_i we can make without changing of any $R_j\ (j >= i)$, I used another segment tree to store the minimum $(X_j - Y_j)$ for those $P_i = -1$.

Because we can change a R_i from -1 to 1 at most $O(N)$ times. And we can make at most $O(N)$ transitions unavailable. So the total time complexity using segment trees is $O(N\ log\ N)$.

Relationship Between Two Solutions & Another Proof

Having discovered two available solutions, let's try to find out the relationship between these two algorithms. And we can also get another proof of the greedy algorithm in the following discussion.

Let's reviewing the first solution now. We define $f_i(x)$ is the minimum cost to ensure that Y_1 to Y_i match the condition and $Y_i = x$. And the domain of $f_i(x)$ is $[1 + A * (N - 2), Q]$.

If we use the greedy algorithm, $Y_N = 1 + A * (N - 1)$ at first. It is also the leftmost place Y_N can be. Because there is only one way to put Y_N there and ensure that Y_1 to Y_N match the condition, the cost at first is equal to $f_N(1 + A * (N - 1))$. There is an obvious observation: when we have made t transitions, $Y_N = 1 + A * (N - 1) + t$.

If we have a make $t - 1$ transition(s) and we have got the minimum cost for now, and if we can prove that we will get the minimum cost to make t transition(s) using the greedy algorithm, we can also prove no mater how many transitions we have made, our cost is minimum using mathematical induction.

So let's try to prove it.

At first we should figure out how can we get all the values of $\{Y_i\}$ and let $Y_N = 1 + A * (N - 1) + t$ to minimize the cost. Because Y_N is fixed, Δ_N is also fixed. The only thing we should do now is to minimize $f_{N-1}(x)$, which is $\sum_{j=1}^{N-1} \Delta_j$. But Y_N is fixed, as a result, the domain of $f_{N-1}(x)$ differs with its initial domain. x need to be in $[max\{Y_N - B, 1 + A * (N - 2)\}, Y_N - A]$. If we can find the value of x in the new domain to minimize $f_{N-1}(x)$, let it be the value of Y_{N-1}. So the value of Y_{N-1} is also fixed, so that we can get the value of Y_{N-2}, and then Y_{N-3}, Y_{N-4}, Y_{N-5} ... Y_1 using the same method.

Now let's try to find out how each Y_i changes when Y_N changes from $Y_N = 1 + A * (N - 1) + t$ to $Y_N' = Y_N + 1 = 1 + A * (N - 1) + t + 1$.

When Y_N is not changed, we define D as the domain of $f_{N-1}(x)$, which is $[max\{Y_N - B, 1 + A * (N - 2)\}, Y_N - A]$.

When $Y_N' = Y_N + 1$, we define E as the domain of $f_{N-1}(x)$, which is $[max\{Y_N + 1 - B, 1 + A * (N - 2)\}, Y_N + 1 - A]$.

Let's observe the function $f_{N-1}(x)$ in domain $D \cup E$. And when $Y_{N-1} = d$ ($d \in D \cup E$), $f_{N-1}(x)$ is minimum.

i. When $d \in D$ and $d \notin E$ **(Figure 2)**
Since $f_{N-1}(x)$ is a concave function, $f_{N-1}(x)$ is monotonically increasing in $D \cup E$. So in $f_{N-1}(d + 1)$ is minimum in E. So when $Y_N' = Y_N + 1$, $Y_{N-1}' = Y_{N-1} + 1 = d + 1$.

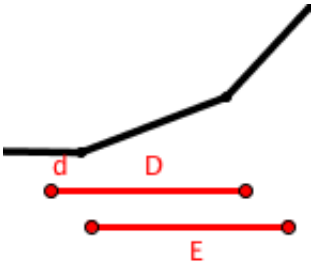

Figure 2

ii. When $d \notin D$ and $d \in E$ **(Figure 3 or 4)**
Since $f_{N-1}(x)$ is a concave function, $f_{N-1}(x)$ is monotonically decreasing in $D \cup E$. So in $f_{N-1}(d - 1)$ is minimum in D. So when $Y_N' = Y_N + 1$, $Y_{N-1}' = Y_{N-1} + 1 = d - 1 + 1 = d$.

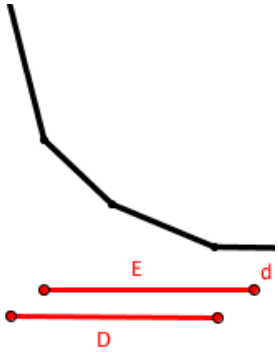

Figure 3

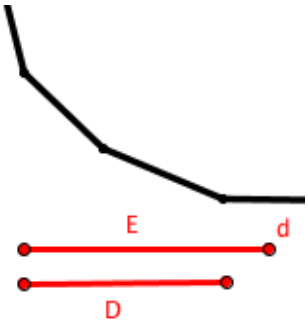

Figure 4

iii. When $d \in D$ and $d \in E$ **(Figure 5 or 6)**
It is obvious that when $Y_N' = Y_N + 1$, $Y_{N-1}' = Y_{N-1} = d$.

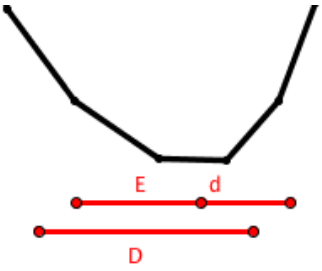

Figure 5

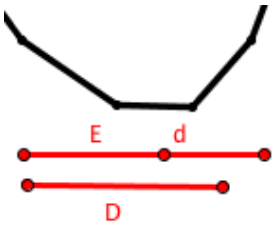

Figure 6

We know that Y_{N-1}' is equal to either $Y_{N-1} + 1$ or Y_{N-1}.

- When $Y_{N-1}' = Y_{N-1} + 1$, we can use the same method to figure out Y_{N-2}', if $Y_{N-2}' = Y_{N-2} + 1$, we can figure out Y_{N-3}, and then Y_{N-4} ...
- When $Y_{N-1}' = Y_{N-1}$ the domain of Y_{N-2} won't change at all, so the value of Y_{N-2} will also not change. And then Y_{N-3}, Y_{N-4} ... Y_1 won't change too.

So we have figure out how each Y_i changes when Y_N changes from $t - 1$ to t. We have a value of i: if $j >= i$, $Y_j' = Y_j + 1$; if $j < i$, $Y_j' = Y_j$. That is same as transition T_i. So if we find the transition that will reduce the cost most, we will get the minimum cost and ensure that $Y_N = t$.

Program

sequence_2.cpp

```
// For Junde Huang.
// sequence, CTSC 2009.
// Written by Sinya in September, 2009.

#include <iostream>
#include <cstring>

#define maxn 500100
#define maxn4 maxn*4
#define maxq 1000000000

using namespace std;

int n;
int q, a, b;
int x[maxn];
int sum[maxn4], minsum[maxn4], pos[maxn4];
```

```
int timer[maxn4], rec[maxn4], p1[maxn4];
int remain[maxn];
int sol;

void change1(int n, int l, int r, int x, int y) {
    if ( (x < l) || (x > r) ) return;
    if (r == l) {
        if (y != 0)
            sum[n] = y;
        pos[n] = x;
        if (remain[x] == 0)
            minsum[n] = maxn;
        else
            minsum[n] = y;
        return;
    }
    int lc = n << 1, rc = lc + 1, mid = (r + l) >> 1;
    change1(lc, l, mid, x, y);
    change1(rc, mid + 1, r, x, y);
    sum[n] = sum[lc] + sum[rc];
    if (minsum[lc] + sum[rc] < minsum[rc]) {
        minsum[n] = minsum[lc] + sum[rc];
        pos[n] = pos[lc];
    }
    else {
        minsum[n] = minsum[rc];
        pos[n] = pos[rc];
    }
}

void change2(int n, int l, int r, int x, int y, int delta) {
    if ( (x < l) || (x > r) ) return;
    delta += rec[n];
    if (r == l) {
        timer[n] = y - delta;
        p1[n] = l;
        return;
    }
    int lc = n << 1, rc = lc + 1, mid = (r + l) >> 1;
    change2(lc, l, mid, x, y, delta);
    change2(rc, mid + 1, r, x, y, delta);
    if (timer[lc] + rec[lc] < timer[rc] + rec[rc]) {
        timer[n] = timer[lc] + rec[lc];
        p1[n] = p1[lc];
    }
    else {
        timer[n] = timer[rc] + rec[rc];
        p1[n] = p1[rc];
    }
}

void change3(int n, int l, int r, int x, int y) {
    if (r < x) return;
    if (l >= x) {
        rec[n] += y;
```

```
        return;
    }
    int lc = n << 1, rc = lc + 1, mid = (r + l) >> 1;
    change3(lc, l, mid, x, y);
    change3(rc, mid + 1, r, x, y);
    if (timer[lc] + rec[lc] < timer[rc] + rec[rc]) {
        timer[n] = timer[lc] + rec[lc];
        p1[n] = p1[lc];
    }
    else {
        timer[n] = timer[rc] + rec[rc];
        p1[n] = p1[rc];
    }
}

void get(int n, int l, int r, int p, int delta, int* np, int* time) {
    if (r < p) return;
    delta += rec[n];
    if (l >= p) {
        if (delta + timer[n] < *time) {
            *time = delta + timer[n];
            *np = p1[n];
        }
        return;
    }
    int lc = n << 1, rc = lc + 1, mid = (r + l) >> 1;
    get(lc, l, mid, p, delta, np, time);
    get(rc, mid + 1, r, p, delta, np, time);
}

void input() {
    scanf("%d%d%d%d", &n, &q, &a, &b);

    memset(sum, 0, sizeof(sum));
    memset(minsum, 0, sizeof(minsum));
    memset(pos, 0, sizeof(pos));
    memset(timer, 0, sizeof(timer));
    memset(rec, 0, sizeof(rec));
    sol = 0;

    int temp = 1;

    for (int i = 1; i <= n; ++i) {
        scanf("%d", &x[i]);
        sol = sol + abs(x[i] - temp);

        if (i == 1)
            remain[i] = maxn;
        else
            remain[i] = b - a;

        if (temp >= x[i]) {
            change1(1, 1, n, i, 1);
            change2(1, 1, n, i, maxq, 0);
        }
```

```
        else {
            change1(1, 1, n, i, -1);
            change2(1, 1, n, i, x[i] - temp, 0);
        }

        temp += a;
    }
    remain[n+1] = 0;
}

void work() {
    int now = 0, maxt = q - ( 1 + a * (n - 1) );
    while (minsum[1] < 0) {
        int p = pos[1];
        int np, time = maxq;
        get(1, 1, n, p, 0, &np, &time);
        int mt = min(time, remain[p]);

        if ( mt + now >= maxt) {
            sol += minsum[1] * (maxt - now);
            return;
        }
        sol += minsum[1] * mt;

        change3(1, 1, n, p, -mt);

        if (time < remain[p]) {
            remain[p] -= mt;
            change1(1, 1, n, np, 1);
            change2(1, 1, n, np, maxq, 0);
        }
        else {
            remain[p] -= mt;
            change1(1, 1, n, p, 0);
        }

        now += mt;
    }
}

int main() {
    freopen("sequence.in", "r", stdin);
    freopen("sequence.out", "w", stdout);
    input();
    work();
    printf("%d", sol);
    return 0;
}
```