Código completo

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))

long* recebeCanteiros(int size){
    long *lst = (long*) malloc(size * sizeof(long));
    for (int i = 0; i < size; i++)
        scanf("%ld", &lst[i]);

    return lst;
}

void recalcMinBloco(long *lst, long *mins, int block_id, int block_size, int size){
    int start = block_id * block_size;
    int end = (block_id + 1) * block_size;

    // Garante que não ultrapassemos o fim do vetor lst
    if (end > size) end = size;

    long min_val = LONG_MAX; // começa com o maior valor possível
    for (int k = start; k < end; k++)
        min_val = MIN(min_val, lst[k]);

    // Armazena o mínimo encontrado para este bloco
    if (start < size) // evita acessar mins[-1] se size for 0
        mins[block_id] = min_val;
}

void empurraLazy(long *lst, long *lazy, int block_id, int block_size, int size){
    // Se não há valor preguiçoso, não faz nada
    if (lazy[block_id] == 0) return;

    int start = block_id * block_size;
    int end = (block_id + 1) * block_size;
    if (end > size) end = size;

    // Aplica o valor preguiçoso a todos os elementos reais do bloco
    for (int k = start; k < end; k++)
        lst[k] += lazy[block_id];

    // Limpa o valor preguiçoso
    lazy[block_id] = 0;
}

void atualizaReforco(long *lst, long *mins, long *lazy, int L, int R, long val, int block_size, int siz
    // Garante que o intervalo é válido
    if (L > R || L >= size || R < 0) return;
    // Ajusta R para os limites do vetor
```

```
    if (R >= size) R = size - 1;

    int bloco_L = L / block_size;
    int bloco_R = R / block_size;

    // Empurra o valor preguiçoso dos blocos de borda ANTES de modificá-los
    empurraLazy(lst, lazy, bloco_L, block_size, size);
    if (bloco_L != bloco_R)
        empurraLazy(lst, lazy, bloco_R, block_size, size);

    // Caso 1: A atualização inteira ocorre dentro de um único bloco
    if (bloco_L == bloco_R) {
        for (int k = L; k <= R; k++)
            lst[k] += val;

        recalcMinBloco(lst, mins, bloco_L, block_size, size);
    }
    // Caso 2: A atualização abrange múltiplos blocos
    else {
        // Borda Esquerda (do ponto L até o fim do bloco_L)
        for (int k = L; k < (bloco_L + 1) * block_size; k++)
            lst[k] += val;

        recalcMinBloco(lst, mins, bloco_L, block_size, size);

        // Blocos do Meio (apenas marca como 'lazy')
        for (int j = bloco_L + 1; j < bloco_R; j++)
            lazy[j] += val;

        // Borda Direita (do início do bloco_R até o ponto R)
        for (int k = bloco_R * block_size; k <= R; k++)
            lst[k] += val;

        recalcMinBloco(lst, mins, bloco_R, block_size, size);
    }
}

long minimoGlobal(long *mins, long *lazy, int num_blocks) {
    long min_global = LONG_MAX;
    for (int j = 0; j < num_blocks; j++)
        min_global = MIN(min_global, mins[j] + lazy[j]);

    return min_global;
}

int encontraMinimo(long *lst, int size, int delta, int block_size){
    int num_blocks = (size + block_size - 1) / block_size;
    long *mins = (long*) malloc(num_blocks * sizeof(long));
    long *lazy = (long*) calloc(num_blocks, sizeof(long));
    long min_global = LONG_MIN;

    // Preenchemos o vetor de minimos com os mínimos parciais
    for (int i = 0; i < num_blocks; i++)
        recalcMinBloco(lst, mins, i, block_size, size);
```

```
    atualizaReforco(lst, mins, lazy, 0, 0, delta, block_size, size);

    for (int i = 0; i < size; i++) {
        long current_min = minimoGlobal(mins, lazy, num_blocks);
        min_global = MAX(min_global, current_min);

        // Já consultamos o último estado (A_N-1), não precisa calcular o próximo
        if (i == size - 1) break;

        int L_sub = (i + 1) - delta; // (n-K)
        if (L_sub < 0) L_sub = 0;
        int R_sub = i; // (n-1)
        int P_add = i + 1; // (n)

        atualizaReforco(lst, mins, lazy, L_sub, R_sub, -1, block_size, size);
        atualizaReforco(lst, mins, lazy, P_add, P_add, delta, block_size, size);
    }

    free(mins);
    free(lazy);

    return min_global;
}

int main(){
    // PRIMEIRA ENTRADA: Recebe N e K
    int qt_canteiros;
    long reforco;
    scanf("%d %ld", &qt_canteiros, &reforco);

    // SEGUNDA ENTRADA: Recebe a lista de canteiros
    long *lista_canteiros = recebeCanteiros(qt_canteiros);
    int block_size = sqrt(qt_canteiros); // tamanho dos blocos que vamos dividir o vetor

    // SAIDA
    int minimo = encontraMinimo(lista_canteiros, qt_canteiros, reforco, block_size);
    printf("%d", minimo);

    free(lista_canteiros);
    return 0;
}
```

Análise das funções individuais

Tomando n = quantidade de canteiros e ignorando alguns termos de custo constante em prol de facilitar os cálculos,

recebeCanteiros

```
long *lst = (long*) malloc(size * sizeof(long));
for (int i = 0; i < size; i++)
    scanf("%ld", &lst[i]);
```

```
return lst;
```

Conforme verificamos no **problema 1**, que possui uma função com a mesma lógica desta, podemos afirmar diretamente que $C(n) = O(n)$.

recalcMinBloco

```
int start = block_id * block_size;
int end = (block_id + 1) * block_size;

if (end > size) end = size;

long min_val = LONG_MAX;
for (int k = start; k < end; k++)                <- # = k = end - start - 1
    min_val = MIN(min_val, lst[k]);              <- # = 1 * laço

if (start < size)
    mins[block_id] = min_val;
```

Primeiramente, devemos encontrar o valor de `end` e `start` em relação à quantidade de canteiros. Sabemos que `block_id` vai de 0 a $\sqrt{N}$, e `block_size`, no pior caso, é $\frac{N}{\sqrt{N}} \approx \sqrt{N}$. Na primeira iteração do loop externo que chama essa função, temos $start = 0$ e $end = \sqrt{N}$, logo

$$C(n) = \sum_{k=start}^{end-1} 1 = \sum_{k=0}^{\sqrt{n}-1} 1$$

Como o intervalo de laço é o mesmo para toda iteração, no pior caso, logo $C(n) = O(\sqrt{N})$.

empurraLazy

```
if (lazy[block_id] == 0) return;

int start = block_id * block_size;
int end = (block_id + 1) * block_size;
if (end > size) end = size;

for (int k = start; k < end; k++)
    lst[k] += lazy[block_id];

lazy[block_id] = 0;
```

Verificamos o mesmo intervalo de loop que a função anterior, logo $C(n) = O(\sqrt{N})$.

atualizaReforco

```
if (L > R || L >= size || R < 0) return;
if (R >= size) R = size - 1;

int bloco_L = L / block_size;
int bloco_R = R / block_size;

empurraLazy(lst, lazy, bloco_L, block_size, size);              <- # = 1
if (bloco_L != bloco_R)
    empurraLazy(lst, lazy, bloco_R, block_size, size);          <- # = 1
```

```
// Caso 1: A atualização inteira ocorre dentro de um único bloco
if (bloco_L == bloco_R) {
    for (int k = L; k <= R; k++)
        lst[k] += val;

    recalcMinBloco(lst, mins, bloco_L, block_size, size);       <- # = 1
}
// Caso 2: A atualização abrange múltiplos blocos
else {
    // Borda Esquerda (do ponto L até o fim do bloco_L)
    for (int k = L; k < (bloco_L + 1) * block_size; k++)
        lst[k] += val;

    recalcMinBloco(lst, mins, bloco_L, block_size, size);       <- # = 1

    // Blocos do Meio (apenas marca como 'lazy')
    for (int j = bloco_L + 1; j < bloco_R; j++)
        lazy[j] += val;

    // Borda Direita (do início do bloco_R até o ponto R)
    for (int k = bloco_R * block_size; k <= R; k++)
        lst[k] += val;

    recalcMinBloco(lst, mins, bloco_R, block_size, size);       <- # = 1
}
```

No caso dessa função, o pior caso ocorre quando $L \neq R$ (2 chamadas da função *empurraLazy*) e a atualização do reforço atinge mais de um bloco (3 laços e 2 chamadas da função *recalcMinBloco*).

Primeiramente, precisamos encontrar os valores de L e R. Analisando a função *encontraMinimo*, verificamos que, no pior caso, $L = n - K$ e $R = n - 1$, ou seja,

$$bloco\_L = \frac{n-K}{\sqrt{n}} = \sqrt{n}\frac{n-K}{n} bloco\_R = \frac{n-1}{\sqrt{n}} = \sqrt{n}\frac{n-1}{n}$$

No caso 2 (caso que estamos analisando), a estrutura dos três laços é a mesma, ou seja, a complexidade de cada um deve ser aproximadamente a mesma. Além disso, como ambos vão iterar sobre o termo geral $c_1\sqrt{n}\frac{n-c_2}{n}, c_1$ *e* c_2 constantes, podemos tomar o caso mais simples (*bloco_R*) e calcular para o caso geral. Assim,

$$C(n) = 4 \cdot O(\sqrt{n}) + \sum_{k=inicio\ loop}^{R} 1$$

Analisando a lógica do laço, vemos que, no pior caso, as iterações vão do começo de um bloco até o seu fim. Em outras palavras, o laço vai de $(n-2)\sqrt{n}$ a n. Conforme a definição do tamanho de um bloco, podemos concluir que o laço itera $\sqrt{n}$ vezes. Assim,

$$\begin{aligned} C(n) &= 4 \cdot O(\sqrt{n}) + \sqrt{n} \\ &= c_0(4\sqrt{n}) + \sqrt{n} \text{ (definição formal)} \\ &= \sqrt{n}(4c_0 + 1) \\ &= c_1\sqrt{n},\ c_1 = 4c_0 + 1 \end{aligned}$$

Portanto, $C(n) = O(\sqrt{n})$.

minimoGlobal

```
long min_global = LONG_MAX;
for (int j = 0; j < num_blocks; j++)
    min_global = MIN(min_global, mins[j] + lazy[j]);

return min_global;
```

Como a função *MIN* tem custo constante, podemos considerar apenas o custo do laço j. Uma vez que `num_blocks` $\approx \sqrt{N}$, logo $C(n) = O(\sqrt{n})$.

encontraMinimo

```
int num_blocks = (size + block_size - 1) / block_size;
long *mins = (long*) malloc(num_blocks * sizeof(long));
long *lazy = (long*) calloc(num_blocks, sizeof(long));
long min_global = LONG_MIN;

// Preenchemos o vetor de minimos com os mínimos parciais
for (int i = 0; i < num_blocks; i++)
    recalcMinBloco(lst, mins, i, block_size, size); <- # = i * custo da função

atualizaReforco(lst, mins, lazy, 0, 0, delta, block_size, size); <- # =  1 * custo da função

for (int i = 0; i < size; i++) {
    long current_min = minimoGlobal(mins, lazy, num_blocks); <- # = i * custo da função
    min_global = MAX(min_global, current_min);

    // Já consultamos o último estado (A_N-1), não precisa calcular o próximo
    if (i == size - 1) break;

    int L_sub = (i + 1) - delta; // (n-K)
    if (L_sub < 0) L_sub = 0;
    int R_sub = i; // (n-1)
    int P_add = i + 1; // (n)

    atualizaReforco(lst, mins, lazy, L_sub, R_sub, -1, block_size, size); <- # = i * custo da função
    atualizaReforco(lst, mins, lazy, P_add, P_add, delta, block_size, size); <- # = i * custo da função
}

free(mins);
free(lazy);

return min_global;
```

O custo da função *encontraMinimo* é dado por

$$C(n) = O(\sqrt{n}) + \sum_{i=0}^{\sqrt{n}-1} O(\sqrt{n}) + \sum_{i=0}^{n-1} 3 \cdot O(\sqrt{n}).$$

Contudo, pela condição de parada interna do segundo laço `if (i == size - 1)`, é mais correto afirmar que o custo da função é

$$C(n) = O(\sqrt{n}) + \sum_{i=0}^{\sqrt{n}-1} O(\sqrt{n}) + \sum_{i=0}^{n-1} O(\sqrt{n}) + \sum_{i=0}^{n-2} 2 \cdot O(\sqrt{n}).$$

Desenvolvendo a função de custo,

$$\begin{aligned} C(n) &= c_0\sqrt{n} + \sqrt{n} \cdot c_1\sqrt{n} + n \cdot c_2\sqrt{n} + (n-1) \cdot 2c_3\sqrt{n} \\ &= c_0\sqrt{n} + c_1 n + c_2 n\sqrt{n} + 2c_3 n\sqrt{n} - 2c_3\sqrt{n} \\ &= \sqrt{n}(c_0 - 2c_3) + c_1 n + n\sqrt{n}(c_2 + 2c_3) \\ &= c_4\sqrt{n} + c_1 n + c_5 n\sqrt{n}, \ c_4 = c_0 + 2c_3 \text{ e } c_5 = c_2 + 2c_3 \end{aligned}$$

Para n suficientemente grande, podemos afirmar que o custo será determinado pelo valor que cresce mais rapidamente. Como, nesse caso, $\sqrt{n} < n < n\sqrt{n}$, portanto podemos afirmar que $C(n) \approx c_5 n\sqrt{n} \implies C(n) \approx O(n\sqrt{n})$.

Conclusão

main

```
// PRIMEIRA ENTRADA: Recebe N e K
int qt_canteiros;
long reforco;
scanf("%d %ld", &qt_canteiros, &reforco);

// SEGUNDA ENTRADA: Recebe a lista de canteiros
long *lista_canteiros = recebeCanteiros(qt_canteiros);<- # = custo da função
int block_size = sqrt(qt_canteiros);

// SAIDA
int minimo = encontraMinimo(lista_canteiros, qt_canteiros, reforco, block_size); <- # = custo da função
printf("%d", minimo);

free(lista_canteiros);
return 0;
```

Conforme a análise das funções individuais, o custo final será dado por

$$C(n) = O(\sqrt{n}) + O(n\sqrt{n})$$

Para n suficientemente grande,

$$C(n) = \sqrt{n}(c_0 + nc_1) \approx n\sqrt{n}$$

Portanto, o custo final do algoritmo completo é $\approx O(n\sqrt{n})$.