PRODUCTO DE INTERVALOS

|  |  |
| --- | --- |
| Fecha de Elaboración: | Octubre 3, 2013 |
| Autores: | Diego Serrano |
| Colaboradores: | Diana Gómez |
| Fuente: | 12532 – Interval Product  UVa Online Judge |

## Problema

Es normal sentirse preocupado y tenso el día antes de un concurso de programación. Para relajarse, usted se fue a tomar una copa con unos amigos en un bar cercano. Para mantener su mente aguda para el día siguiente, usted decidió jugar el siguiente juego. Empezando, sus amigos le darán una secuencia de *N* números enteros  
*X1, X2,…, XN*.

Entonces, habrá *K* rondas, y en cada ronda, sus amigos emitirán un comando, que puede ser:

* Un comando de *cambio*, cuando sus amigos quieren cambiar uno de los valores en la secuencia,
* Un comando de *producto*, cuando sus amigos le dan dos valores  
  *I, J* y le preguntan si el producto *XI × XI+1 ×…× XJ-1 × XJ* es positivo, negativo o cero.

Ya que estás en un pub, se decidió que la pena por una respuesta incorrecta es beber una pinta de cerveza. A usted le preocupa que esto pueda afectarle negativamente en la competencia del día siguiente, y no desea comprobar si la teoría del pico de Ballmer es correcta. Afortunadamente, sus amigos le dieron el derecho a utilizar su ordenador portátil. Puesto que usted confía más sus habilidades de codificación que su matemática, usted decidió escribir un programa que le ayude en el juego.

**Entrada**

Cada caso de prueba es descrito por varias líneas. La primera línea contiene dos números enteros *N* y *K*, lo que indica el número de elementos de la secuencia y el número de rondas del juego, respectivamente (*1<=N, K<=105*). La segunda línea contiene *N* enteros *XI* que representan los valores iniciales de la secuencia (*-100<=XI<=100, I=1,2,…, N*). Cada una de las líneas *K* próximas describe un comando y comienza con una letra mayúscula que es o ‘C’ o ‘P’. Si el comando es 'C', la línea describe una orden de cambio, y la letra va seguida de dos números enteros I y V, que indican que la X*I* debe recibir el valor V (*1<=I<=N* y -*100<=V<=100*). Si el comando es 'P', la línea describe un comando de producto, y la letra va seguida de dos números enteros *I* y *J* que indica que el producto de *XI* hasta *XJ*, inclusive, debe calcularse (*1<=I<=J<=N*). Dentro de cada caso de prueba hay al menos un comando producto.

**Salida**

Para cada caso de prueba, imprima una línea con una cadena que representa el resultado de todos los comandos de producto en el caso de prueba. El *i*-ésimo carácter de la cadena representa el resultado del *i*-ésimo comando producto. Si el resultado de la orden es positiva el carácter de salida debe ser "+" (más), si el resultado es negativo, el carácter de salida debe ser "-" (menos), y si el resultado es cero, el carácter de salida debe ser "0" (cero).

**Ejemplo de Entrada**

4 6

-2 6 0 -1

C 1 10

P 1 4

C 3 7

P 2 2

C 4 -5

P 1 4

5 9

1 5 -2 4 3

P 1 2

P 1 5

C 4 -5

P 1 5

P 4 5

C 3 0

P 1 5

C 4 -5

C 4 -5

**Ejemplo de Salida**

0+-

+-+-0

## Planteamiento de la Solución BÁSICA

Para resolver el problema, sencillamente podríamos almacenar la secuencia en un arreglo unidimensional, y cuando llegue una operación de producto, utilizar un ciclo para multiplicar los valores de *I* hasta *J*. Para evitar el desbordamiento de bits, podemos cambiar los contenidos de la secuencia por -1 cuando sea un número negativo, 1 cuando sea un número positivo, y cero queda como cero.

En esta solución, la complejidad de ejecutar el comando de cambio es *O(1)* y la complejidad del comando producto es *O(N)*. En Uva Online Judge, el tiempo máximo para este problema es de 2 segundos, y esta solución excede éste límite.

## Código Fuente

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

public class Naive {

public static int[] x;

public static void main(String[] args) {

try {

InputStream input = System.in;

BufferedReader reader = new BufferedReader(new

InputStreamReader(input));

StringBuilder out = new StringBuilder();

int n, k;

String params;

String[] sp;

while ((params = reader.readLine()) != null &&

params.length() > 0) {

sp = params.split(" ");

n = Integer.parseInt(sp[0]);

k = Integer.parseInt(sp[1]);

x = new int[n];

params = reader.readLine();

sp = params.split(" ");

for (int i = 0; i < n; i++) {

x[i] = Integer.parseInt(sp[i]);

if (x[i] > 1)

x[i] = 1;

if (x[i] < -1)

x[i] = -1;

}

while (k-- > 0) {

params = reader.readLine();

sp = params.split(" ");

if (sp[0].equals("C")) {

int ind = Integer.parseInt(sp[1]);

int val = Integer.parseInt(sp[2]);

ind--;

if (val > 1)

val = 1;

if (val < -1)

val = -1;

x[ind] = val;

} else {

int start = Integer.parseInt(sp[1]);

int endInd = Integer.parseInt(sp[2]);

start--;

endInd--;

int res = 1;

for (int i=start; i<=endInd; i++) {

res \*= x[i];

}

if (res == 1)

out.append("+");

else if (res == -1)

out.append("-");

else

out.append("0");

}

}

out.append("\n");

}

System.out.print(out);

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

## Planteamiento de la Solución INTERMEDIA

Teniendo en cuenta las propiedades de la multiplicación, podemos extraer datos que pueden servir para resolver de forma más eficiente.

Cuando se multiplica cualquier número por 0, su resultado es cero. Así que si en la secuencia hay un cero, entonces el resultado va a ser 0, invariablemente. Para que una multiplicación de como resultado un numero negativo, se necesitan un número impar de factores negativos. Así que contando el número de factores impares en la secuencia podríamos saber si es negativo.

Para implementar esta solución, usaremos programación dinámica, y usaremos dos arreglos unidimensionales auxiliares, que nos guardan *Z[t]* el número de ceros en la secuencia, desde *0* hasta *t*, y *G[t]* el número de negativos en la secuencia desde *0* hasta *t*. *Z* y *G* pueden pre-calcularse, usando programación dinámica donde:

De esta manera, aplicando la operación *Z[J]-Z[I-1]* tendremos la cantidad de ceros en el caso de prueba, y con la operación *G[J]-G[I-1]* tendremos la cantidad de negativos. Si hay al menos un cero, entonces la respuesta es *0*. Si no es así, y hay un número impar de negativos, entonces la respuesta es "-". De otra forma, la respuesta es "+".

La complejidad para crear los arreglos auxiliares es lineal *O(N)*, el tiempo para responder el comando producto es constante *O(1)*, pero la complejidad de realizar un comando cambio es lineal *O(N)*. En Uva Online Judge, el tiempo máximo para este problema es de 2 segundos, y esta solución excede éste límite.

## Código Fuente

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

public class DP {

public static int[] x;

public static int[] z;

public static int[] g;

public static void main(String[] args) {

try {

InputStream input = System.in;

BufferedReader reader = new BufferedReader(new

InputStreamReader(input));

StringBuilder out = new StringBuilder();

int n, k;

String params;

String[] sp;

while ((params = reader.readLine()) != null &&

params.length() > 0) {

sp = params.split(" ");

n = Integer.parseInt(sp[0]);

k = Integer.parseInt(sp[1]);

x = new int[n + 1];

z = new int[n + 1];

g = new int[n + 1];

z[0] = 0;

g[0] = 0;

params = reader.readLine();

sp = params.split(" ");

for (int i = 1; i <= n; i++) {

x[i] = Integer.parseInt(sp[i - 1]);

z[i] = (x[i] == 0) ? z[i - 1] + 1 : z[i - 1];

g[i] = (x[i] < 0) ? g[i - 1] + 1 : g[i - 1];

if (x[i] > 1)

x[i] = 1;

if (x[i] < -1)

x[i] = -1;

}

while (k-- > 0) {

params = reader.readLine();

sp = params.split(" ");

if (sp[0].equals("C")) {

int ind = Integer.parseInt(sp[1]);

int val = Integer.parseInt(sp[2]);

if (val > 1)

val = 1;

if (val < -1)

val = -1;

if ((val == 0 && x[ind] != 0)

|| (val < 0 && x[ind] >= 0)

|| (val > 0 && x[ind] <= 0)) {

x[ind] = val;

for (int i = ind; i <= n; i++) {

z[i] = (x[i] == 0) ?

z[i - 1] + 1 : z[i - 1];

g[i] = (x[i] < 0) ?

g[i - 1] + 1 : g[i - 1];

}

}

} else {

int start = Integer.parseInt(sp[1]);

int endInd = Integer.parseInt(sp[2]);

int numZeroes =

z[endInd] - z[start - 1];

int numNegatives =

g[endInd] - g[start - 1];

if (numZeroes > 0) {

out.append("0");

} else if (numNegatives % 2 == 1) {

out.append("-");

} else {

out.append("+");

}

}

}

out.append("\n");

}

System.out.print(out);

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

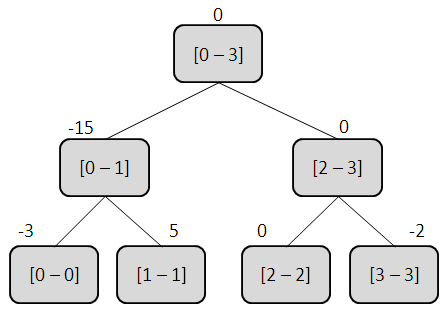
}

}

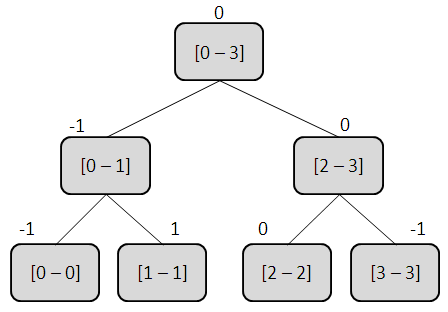
## Planteamiento de la Solución ÓPTIMA

Usando arboles de segmento, podemos reducir el tiempo del comando *producto* a *O(logN)*, donde *N* es la longitud de la secuencia.

La idea principal es construir un árbol de segmento de los productos de los sub-intervalos de nuestra secuencia. Por ejemplo, si tenemos la secuencia *[-2, 6, 0, -1]*, tendríamos el siguiente árbol:



Sin embargo, si manejamos el valor real de los productos para nuestros cálculos, fácilmente veremos que nuestras variables sufrirán de desbordamiento de bits. Dado que el problema solo nos llama a considerar si es positivo, negativo o cero, podemos definir a los productos positivos como 1, los negativos como -1, y los de cero, quedarían de la misma manera. Luego de aplicar las operaciones, nuestro nuevo árbol luce así:



La complejidad para crear el árbol es lineal *O(N)*, y el tiempo para responder cada comando es *O(logN)*, por lo cual, la complejidad de responder a todos los comandos es *O(K logN)*.

La solución fue enviada a Uva Online Judge el 2 de octubre de 2013, con un tiempo de ejecución de 1.009 segundos.

## Código Fuente

import java.io.BufferedReader;

import java.io.InputStream;

import java.io.InputStreamReader;

public class Main {

public static int *M* = 100000;

public static int[] *x* = new int[*M* + 10];

public static int[] *segTree* = new int[4 \* *M* + 100];

public static void buildTree(int at, int lo, int hi) {

if (lo == hi) {

*segTree*[at] = *x*[lo];

return;

}

int mid = (lo + hi) / 2;

*buildTree*(2 \* at, lo, mid);

*buildTree*(2 \* at + 1, mid + 1, hi);

*segTree*[at] = *segTree*[2 \* at] \* *segTree*[2 \* at + 1];

}

public static void update(int at, int index, int value, int lo, int hi) {

if (index < lo || index > hi)

return;

if (lo == hi) {

*segTree*[at] = value;

return;

}

int mid = (lo + hi) / 2;

*update*(2 \* at, index, value, lo, mid);

*update*(2 \* at + 1, index, value, mid + 1, hi);

*segTree*[at] = *segTree*[2 \* at] \* *segTree*[2 \* at + 1];

}

public static int query(int at, int lo, int hi, int start, int endInd) {

if (start > hi || endInd < lo)

return 1;

if (lo >= start && hi <= endInd)

return *segTree*[at];

int mid = (lo + hi) / 2;

return *query*(2 \* at, lo, mid, start, endInd)

\* *query*(2 \* at + 1, mid + 1, hi, start, endInd);

}

public static void main(String[] args) {

try {

InputStream input = System.*in*;

BufferedReader reader = new BufferedReader(new

InputStreamReader(input));

StringBuilder out = new StringBuilder();

int n, k;

String params;

String[] sp;

while ((params = reader.readLine()) != null &&

params.length() > 0) {

sp = params.split(" ");

n = Integer.*parseInt*(sp[0]);

k = Integer.*parseInt*(sp[1]);

params = reader.readLine();

sp = params.split(" ");

for (int i = 0; i < n; i++) {

*x*[i] = Integer.*parseInt*(sp[i]);

if (*x*[i] > 1)

*x*[i] = 1;

if (*x*[i] < -1)

*x*[i] = -1;

}

*buildTree*(1, 0, n - 1);

while (k-- > 0) {

params = reader.readLine();

sp = params.split(" ");

if (sp[0].equals("C")) {

int ind = Integer.*parseInt*(sp[1]);

int val = Integer.*parseInt*(sp[2]);

ind--;

if (val > 1)

val = 1;

if (val < -1)

val = -1;

*update*(1, ind, val, 0, n - 1);

*x*[ind] = val;

} else {

int start = Integer.*parseInt*(sp[1]);

int endInd = Integer.*parseInt*(sp[2]);

start--;

endInd--;

int res = *query*(1, 0, n - 1, start,

endInd);

if (res == 1)

out.append("+");

else if (res == -1)

out.append("-");

else

out.append("0");

}

}

out.append("\n");

}

System.*out*.print(out);

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}