PROTECCIÓN DEL AMBIENTE

|  |  |
| --- | --- |
| Fecha de Elaboración: | Octubre 10, 2013 |
| Autores: | Diego Serrano |
| Fuente: | 12528 – Environment Protection  UVa Online Judge |

## Problema

Arsenic & Cyanide Mining (ACM) es una empresa que ha decidido recientemente iniciar el desarrollo de sus minas en las tierras cercanas a su ciudad natal. Como miembro del comité de reglamentación de la ciudadanía para las operaciones de ACM, su tarea consiste en controlar que tanto puede extraer la corporación en esas tierras, de tal manera que se puedan mantener los puestos de trabajo y otros beneficios sin sacrificar el medio ambiente y la salud de la población local.

ACM tiene planes de realizar minería en varios parches rectangulares de tierra. Un pedazo de tierra tiene anchura *W*, puede ser excavado hasta una profundidad máxima *D*, y tiene una superficie la cual se considera que esta en profundidad 0. Los minerales en un parche se organizan en tres capas, que pueden variar en su profundidad a lo largo de la anchura del parche, pero siempre tienen el mismo perfil a lo largo de toda su longitud. Esta es la razón por la cual ACM sólo está interesado en el perfil a lo largo de la anchura de cada parche, y ha realizado trabajos de exploración con el fin de determinar con precisión su forma. Como resultado, descubrieron que las dos interfaces entre las tres capas de los minerales pueden ser representadas por dos funciones *y1(x)* y *y2(x)*, donde la primera describe el límite entre la capa superior y la capa media, y la segunda describe el límite entre la capa intermedia y la capa inferior. Estas funciones son siempre de la forma:

*-D<y2(x)<y1(x)<0 para 0≤x≤W,*

De tal manera que los límites de las capas no se tocan entre sí. Además, cada función tiene la forma *yi (x) = pi(x)/qi(x)*, donde

para *i = 1, 2* y un cierto número entero *K*. La figura siguiente muestra los perfiles de dos parches de tierra en la forma en que la ACM los representa. El parche de la izquierda tiene un ancho *W = 6* y profundidad *D = 9*, mientras que el parche de la derecha tiene *W = 8* y *D = 10*. Los límites de las capas de cada parche se describen por las funciones definidas debajo de ellos.



ACM cavará todo en un pedazo de tierra hasta una profundidad de excavación *d*, y luego venderá todos los minerales obtenidos para su ganancia. Sin embargo, los minerales en la parte superior y las capas inferiores no tienen valor, por lo que la ganancia de toda la operación proviene exclusivamente de los minerales en la capa media. De hecho, la ganancia es proporcional al área *A* de la capa media cuando la profundidad es *d*. Dado el perfil de un pedazo de tierra y un entero *A*, a usted le gustaría saber la profundidad de excavación *d* que debe permitir cavar a ACM para que puedan obtener un área de minerales de la capa intermedia de exactamente *A*. En la figura anterior se puede ver la respuesta para los dos casos de prueba en la entrada de la muestra. Para el parche en el lado izquierdo, con el fin de conseguir un área *A = 4* la profundidad de la excavación debe ser *d = 4.00000*, mientras que para el parche a la derecha un área *A = 14* requiere una profundidad de excavación *d = 5.51389*.

**Entrada**

Cada caso de prueba se describe mediante cinco líneas. La primera línea contiene cuatro enteros *W*, *D*, *A* y *K* , donde *W* es la anchura de la superficie de la tierra que ACM quiere excavar (*1 ≤ W ≤ 8*), *D* es la profundidad (*1 ≤ D ≤ 10*), *A* es el área de la capa media en el perfil que ACM debe obtener (*1 ≤ A ≤ W × D*), y *K* permite la definición de los perfiles *y1(x )* y *y2(x)* como se explicó anteriormente (*0 ≤ K ≤ 8*).

Cada una de las otras líneas contiene K + 1 números enteros entre -108 y 108, inclusive. La segunda línea contiene el coeficientes de *p1(x)* de *P1,0* a *P1,K*. La tercera línea contiene el coeficientes de *q1(x)* de *Q1,0* a *Q1,K*. La cuarta línea contiene el coeficientes de *p2(x)* de *P2,0* a *P2,K*. La quinta línea contiene el coeficientes de *q2(x)* de *Q2,0* a *Q2,K*. Dentro de cada caso de prueba, *A* es estrictamente menor que el área total de la capa media y existe un único valor *d* tal que una profundidad de excavación *d* produce una superficie de minerales de la capa media en el perfil de exactamente *A*. Además, *q1(x) = 0*, *q2 (x) = 0* y *-D <y2(x) < y1(x) < 0*, para *0 ≤ x ≤ W*.

**Salida**

Para cada caso de prueba se imprime una línea con un número racional que representa la profundidad *d* que se le debe permitir cavar a ACM para que puedan obtener un área de minerales de la capa intermedia de exactamente *A*. El resultado debe ser un número racional con exactamente cinco dígitos después del punto decimal, redondeado si es necesario.

**Ejemplo de Entrada**

6 9 4 1

-10 1

2 0

-16 1

2 0

8 10 14 4

-1392 864 -216 24 -1

1312 -864 216 -24 1

-73 36 -54 36 -9

17 -4 6 -4 1

**Ejemplo de Salida**

4.00000

5.51389

## Planteamiento de la Solución

Dado la dificultad para integrar *p(x)* y *q(x)*, debemos recurrir a una aproximación. Recordemos la clase de métodos numéricos, y en especial al matemático Thomas Simpson. En el siglo XVI, Simpson propuso una aproximación numérica para integrales definidas usando interpolares cuadráticos. Específicamente, se define la siguiente aproximación:

La regla de Simpson puede derivarse aproximando la integral *f(x)*, representada en la figura por la línea azul, por la interpolación cuadrática *P(x)*, representada por la línea roja.

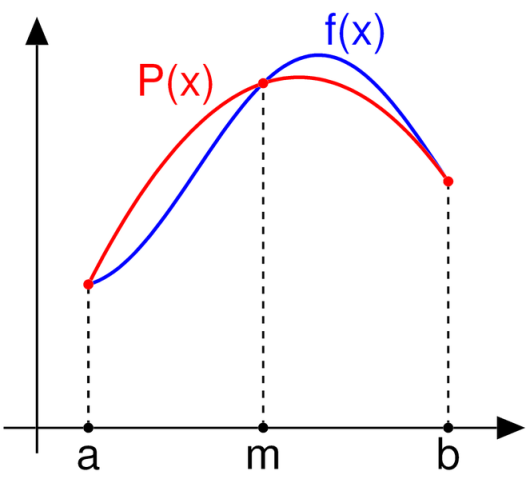


Figura 1. Ilustración de la regla de Simpson. Fuente: Wikipedia

Sabiendo como calcular el área entre las dos funciones, aproximadamente, aún nos queda encontrar el punto *d* hasta el cual se le permitirá cavar. Para esto, simplemente hacemos una búsqueda binaria entre el punto más alto, que es la superficie *high=0*, y la máxima profundidad posible *low=-D*. Para este tipo de búsqueda, encontramos que toma *22* iteraciones para encontrar la profundidad, con la precisión deseada.

Cuando corremos esta solución, nos damos cuenta que aún no es una solución lo suficientemente rápida. 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.InputStream;

import java.io.InputStreamReader;

import java.text.DecimalFormat;

import java.util.StringTokenizer;

public class Naive {

public static double W, D, A, K;

public static double[] p1 = new double[10];

public static double[] q1 = new double[10];

public static double[] p2 = new double[10];

public static double[] q2 = new double[10];

public static double eval(double x, double line) {

double num1, den1, num2, den2;

num1 = num2 = den1 = den2 = 0.0;

for (int i = (int) K; i >= 0; i--) {

num1 += Math.pow(x, i) \* p1[i];

den1 += Math.pow(x, i) \* q1[i];

num2 += Math.pow(x, i) \* p2[i];

den2 += Math.pow(x, i) \* q2[i];

}

double integral1, integral2;

integral1 = num1 / den1;

integral2 = num2 / den2;

if (line > integral1) {

return 0;

} else if (line < integral2) {

return integral1 - integral2;

} else

return integral1 - line;

}

public static double simps(double a, double b, double line) {

return ((b - a) / 6.0)

\* (eval(a, line) +

(4 \* eval((a + b) / 2.0, line)) +

eval(b, line));

}

public static void main(String[] args) {

try {

InputStream input = System.in;

BufferedReader reader = new BufferedReader(new

InputStreamReader(input));

StringTokenizer tokenizer = new StringTokenizer(" ");

String params;

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

tokenizer = new StringTokenizer(params);

W = Integer.parseInt(tokenizer.nextToken());

D = Integer.parseInt(tokenizer.nextToken());

A = Integer.parseInt(tokenizer.nextToken());

K = Integer.parseInt(tokenizer.nextToken());

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

p1[i] =

Double.parseDouble(tokenizer.nextToken());

}

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

q1[i] =

Double.parseDouble(tokenizer.nextToken());

}

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

p2[i] =

Double.parseDouble(tokenizer.nextToken());

}

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

q2[i] =

Double.parseDouble(tokenizer.nextToken());

}

double eps = 1e-4;

double x;

double low = -D;

double high = 0;

for (int i = 0; i < 22; i++) {

double totalArea = 0.0;

x = (low + high) / 2.0;

for (double a = 0;

a + eps - 1e-5 < W;

a += eps) {

double b = a + eps;

totalArea += simps(a, b, x);

}

if (totalArea < A) {

high = x;

} else {

low = x;

}

}

DecimalFormat numberFormat =

new DecimalFormat("#.00000");

System.out.println(numberFormat.format(-low));

}

reader.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

## Planteamiento de la Solución ÓPTIMA

Como vimos anteriormente, el tiempo que emplea la anterior solución no fue suficientemente bueno. El problema está al calcular el resultado de los polinomios.

En matemáticas no hay diferencia entre calcular el valor de un polinomio usando la potencia de números enteros positivos y multiplicación repetida. Por ejemplo, en el siguiente polinomio:

Con la forma estándar de calcular el valor, tendríamos que hacer 3 sumas y 6 multiplicaciones:

Sin embargo, gracias al matemático británico William George Horner, podemos hacerla de una forma más óptima. El método de Horner describe un algoritmo para calcular polinomios, que consiste en transformar monomios en una forma eficiente de calcular para una máquina.

Dado el polinomio:

Esta operación podemos representarla de la forma:

Lo cual en nuestro ejemplo, tendría finalmente 3 sumas y 3 multiplicaciones:

En el caso general, en la forma estándar de calcular el valor de un polinomio de grado *n*, necesitamos máximo *n* sumas y *(n2 + n)/2* multiplicaciones. Usando el método de Horner, podemos reducir el número de multiplicaciones a *2n - 1*.

Aplicando el método de Horner en nuestro código, vemos como los casos de prueba son resueltos casi inmediatamente.

La solución fue enviada a Uva Online Judge el 9 de octubre de 2013, con un tiempo de ejecución de 4.382 segundos.

## Código Fuente

import java.io.BufferedReader;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.text.DecimalFormat;

import java.util.StringTokenizer;

public class Main {

public static double W, D, A, K;

public static double[] p1 = new double[10];

public static double[] q1 = new double[10];

public static double[] p2 = new double[10];

public static double[] q2 = new double[10];

public static double eval(double x, double line) {

double num1, den1, num2, den2;

num1 = num2 = den1 = den2 = 0.0;

for (int i = (int) K; i >= 0; i--) {

num1 = num1 \* x + p1[i];

den1 = den1 \* x + q1[i];

num2 = num2 \* x + p2[i];

den2 = den2 \* x + q2[i];

}

double integral1, integral2;

integral1 = num1 / den1;

integral2 = num2 / den2;

if (line > integral1) {

return 0;

} else if (line < integral2) {

return integral1 - integral2;

} else

return integral1 - line;

}

public static double simps(double a, double b, double line) {

return ((b - a) / 6.0)

\* (eval(a, line) +

(4 \* eval((a + b) / 2.0, line)) +

eval(b, line));

}

public static void main(String[] args) {

try {

InputStream input = System.in;

BufferedReader reader = new BufferedReader(new

InputStreamReader(input));

StringTokenizer tokenizer = new StringTokenizer(" ");

String params;

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

tokenizer = new StringTokenizer(params);

W = Integer.parseInt(tokenizer.nextToken());

D = Integer.parseInt(tokenizer.nextToken());

A = Integer.parseInt(tokenizer.nextToken());

K = Integer.parseInt(tokenizer.nextToken());

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

p1[i] =

Double.parseDouble(tokenizer.nextToken());

}

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

q1[i] =

Double.parseDouble(tokenizer.nextToken());

}

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

p2[i] =

Double.parseDouble(tokenizer.nextToken());

}

params = reader.readLine();

tokenizer = new StringTokenizer(params);

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

q2[i] =

Double.parseDouble(tokenizer.nextToken());

}

double eps = 1e-4;

double x;

double low = -D;

double high = 0;

for (int i = 0; i < 22; i++) {

double totalArea = 0.0;

x = (low + high) / 2.0;

for (double a = 0;

a + eps - 1e-5 < W;

a += eps) {

double b = a + eps;

totalArea += simps(a, b, x);

}

if (totalArea < A) {

high = x;

} else {

low = x;

}

}

DecimalFormat numberFormat =

new DecimalFormat("#.00000");

System.out.println(numberFormat.format(-low));

}

reader.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}