Pérfil de una curva cóncava

Dados una serie de valores cuyo pérfil se ajusta al de una curva cóncava, siendo estrictamente decrecientes hasta un determinado valor a partir del cual son estrictamente crecientes, se pide encontrar el valor del mínimo. Se admiten líneas estrictamente crecientes o estrictamente decrecientes.

Requisitos de implementación.
Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.
La solución obtenida debe emplear la técnica de divide y vencerás
La función que resuelve el problema debe recibir los datos en un vector y devolver el valor del mínimo.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas, en la primera se indica el número de elementos del vector y en la segunda los valores del vector separados por blancos.

El número de elementos del vector es mayor que cero y no hay dos valores consecutivos iguales.

Salida

Para cada caso de prueba se escribe en una línea el mínimo de la curva.

Entrada de ejemplo

```
5
7 5 3 8 9
8
9 8 7 6 5 4 3 2
1
2
2
3 5
10
34 25 12 10 9 8 7 6 5 9
```

Salida de ejemplo

```
3
2
2
3
5
```

Autor: Isabel Pita.

Fuga de la prisión

Año 3016, sistema interestelar XG23. En la carcel de alta seguridad C78 se ha detectado la fuga de un preso del pabellón Z4.

Cada preso del pabellón se identifica mediante una letra del alfabeto anglosajón. Sabiendo que el primer preso de este pabellón tiene la letra *x1* y el último es el *x2* y que todos ellos son consecutivos, encuentra la letra del preso fugado lo más rápido posible para evitar que pueda robar una nave y salir del sistema interestelar.

Requisitos de implementación.

Indicar el coste de la solución obtenida.

La función que resuelve el problema debe recibir las letras de los presos en un vector y calcular el que falta. La implementación debe ser recursiva.

Dada dos variables de tipo *char* el resultado de restar su valor es la diferencia entre los códigos ASCII de los caracteres. Por ejemplo si la variable *x1* tiene el valor *'c'* y la variable *x2* el valor *'f'*, el resultado de la operación *x2-x1* es 3.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica la letra del primer y último presidiario del pabellón. En la segunda se indican las letras de los presidiarios presentes en el recuento, ordenados en orden ascendente.

Cada pabellón tiene al menos un preso.

Salida

Para cada caso de prueba se escribe en una línea la letra que falta.

Entrada de ejemplo

```
12
a e
a b c e
c h
d e f g h
c h
c e f g h
c h
c d f g h
c h
c d e g h
c h
c d e f h
c h
c d e f g
d h
e f g h
d h
d f g h
d h
d e g h
d h
d e f h
d h
d e f g
```

Salida de ejemplo

```
d
c
d
e
f
g
h
d
e
f
g
h
```

Autor: Isabel Pita.

El juego del bongo

En la residencia de mis tíos, se ha puesto de moda un nuevo juego al que llaman el *bongo*. Cansados de tachar el número que cantaba el cajero sobre su tarjeta de bingo, los ancianos han optado por tachar el número del cartón que coincida con el número cantado por el cajero mas su posición en el cartón (la primera posición del cartón es la cero). De esta forma ejercitan su capacidad de cálculo.

Asi, por ejemplo, si en el cartón están los números 2 4 7 9 12 15 22 y el número cantado es el 6, al anciano tachará el número 9, ya que su posición, la 3, más el número cantado coinciden con el valor. Si no encuentra ningún número que le coincida no tachará ninguno y esperará al siguiente número. Utilizan únicamente cartones con los números ordenados de forma creciente y para evitar que una persona pueda tachar dos números en una tirada, los cartones no tienen ninguna pareja de números consecutivos.

Hoy he estado jugando con ellos, pero me falta práctica y perdía algunos números. He decidido hacer un programa que dados los valores del cartón y el número que canta el cajero me enseñe el número que debo tachar o si no debo hacer nada, para poder estar a su nivel la próxima vez que juegue.

Requisitos de implementación.
Indicar el coste de la solución obtenida.
La función que resuelve el problema debe ser recursiva.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica la cantidad de números que hay en el cartón y el número cantado por el cajero. En la segunda línea se indican los valores del cartón separados por blancos.

Los cartones tienen al menos un elemento.

Salida

Para cada caso de prueba se escriben en una línea el número que se debe tachar o la palabra *NO* si no hay ninguno en su sitio.

Entrada de ejemplo

```
4
7 8
3 6 8 11 14 17 20
6 2
0 3 5 7 9 11
4 5
0 4 7 12
5 4
1 3 7 9 12
```

Salida de ejemplo

```
11
3
7
NO
```

Autor: Isabel Pita.

Obtener el mínimo de un vector

Consideramos un vector V[N] de números enteros, cuyo valores se han obtenido aplicando una rotación sobre un vector ordenado en orden estrictamente decreciente. Implementa un algoritmo que calcule el mínimo del vector con una complejidad O(log(n)). El número de elementos sobre los que se aplica la rotación para obtener el vector de entrada es un valor entre 0 y N y no se conoce.

Por ejemplo, un posible vector de entrada sería el vector 70 55 13 4 100 80 obtenido desplazando los dos primeros elementos del vector 100 80 70 55 13 4 al final del mismo.

Obtenido del examen final de septiembre de 2014.

Requisitos de implementación.

Se debe implementar una función recursiva (resolver) que dado el vector, con los datos de entrada ya leidos, obtenga el mínimo en tiempo logarítmico respecto al número de elementos del vector. Se pueden utilizar más parámetros si se considera necesario.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se indica el número de elementos del vector y en la segunda los valores del vector.

Salida

Para cada caso de prueba se escribe en una línea diferente el mínimo del vector.

Entrada de ejemplo

```
4
8 6 4 2
4
2 8 6 4
4
4 2 8 6
4
6 4 2 8
5
8 5 3 1 10
5
5 3 1 10 8
6
70 55 13 4 100 80
```

Salida de ejemplo

```
2
2
2
2
1
1
4
```

Autor: Isabel Pita.

Ejercicio 2

(1,5 puntos) Implementa un programa recursivo que dado un vector de números enteros positivos y consecutivos, del que se han eliminado todos los elementos impares menos uno encuentre cúal es el número impar que queda en el vector. La implementación realizada debe ser eficiente.

(0.5 puntos) Indica el coste de la solución implementada. Justifícalo escribiendo la ecuación de recurrencia del problema y desplegándola.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de 2 líneas. En la primera se indica el número de elementos del vector una vez eliminados los elementos impares. En la siguiente se indican los elementos del vector. El final de los casos se marca con el valor cero.

El vector tiene al menos un elemento. Se garantiza que siempre existe un elemento impar en el vector.

Salida

Para cada caso de prueba se escribe en una línea el elemento impar del vector.

Entrada de ejemplo

```
4
6 8 9 10
4
1 2 4 6
4
4 5 6 8
4
4 6 8 9
5
1 2 4 6 8
5
2 3 4 6 8
1
3
2
5 6
2
6 7
0
```

Salida de ejemplo

```
9
1
5
9
1
3
3
5
7
```

Ejercicio 2

(3 puntos) Se tiene un vector con $k > 0$ valores enteros diferentes. Cada valor se encuentra repetido k_i veces, encontrándose consecutivos todos los valores iguales del vector. Se pide encontrar el número de valores distintos.

El problema se debe resolver empleando la técnica de divide y vencerás. Si existen valores repetidos, el vector no debe recorrerse completo, evitando siempre que se pueda el acceso a los elementos repetidos.

Entrada

La entrada que espera el corrector automático consta de una serie de casos de prueba y acabará cuando se introduzca un 0. Cada caso de prueba consta de dos líneas: en la primera se indica el número de elementos del vector y en la segunda los valores separados por un espacio.

Salida

Para cada caso de prueba se escribe el número de valores diferentes del vector en una línea.

Entrada de ejemplo

```
4
3 3 1 1
3
2 2 2
8
6 6 3 3 8 8 1 1
10
3 3 3 6 6 2 2 2 2 2
14
7 7 7 7 7 4 4 4 4 9 9 2 2 2
20
5 5 5 5 5 5 5 5 4 4 4 4 4 4 1 1 1 7 9 2
1
5
6
6 5 5 5 5 2
6
8 7 6 5 4 3
0
```

Salida de ejemplo

```
2
1
4
3
4
6
1
3
6
```

Lucky Lucke en busca del culpable

Lucky Lucke es el vaquero más rápido del oeste. Alguien ha robado el oro de la cámara blindada del banco y Lucky Lucke debe encontrar el mejor algoritmo posible para dar con el culpable. Lucky Lucke conoce la altura del bandido porque después de cometer el atraco dejó su figura marcada en la parte superior de la puerta del banco.

El sheriff le ha mandado por mail un fichero con la altura de todos los sospechosos del oeste ordenados de menor a mayor. ¿Puedes ayudarle a encontrar al culpable? .

Requisitos de implementación.

Se implementarán dos funciones recursivas diferentes: La primera *buscarIz* encuentra el primer sospechoso de la lista con la altura pedida, la segunda, *buscarDr* encuentra la posición en la lista del último sospechoso con la altura pedida.

La función *resuelveCaso*, llamará a la función *buscarIz* para calcular la primera posición en que aparece la altura pedida y llamará a la función *buscarDr* para obtener la última posición de la altura pedida en la lista.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso comienza con el número de sospechosos seguido de la altura del bandido. A continuación en una línea aparecen las alturas de todos los sospechosos ordenados de menor a mayor.

El número de sospechosos es un valor 0 <= N <= 1.000.000 y las alturas de los sospechosos son valores enteros positivos menores de 250.

Salida

Para cada caso de prueba se muestra en un línea la posición que ocupa en la lista de sospechosos el primero con la altura buscada seguido de la posición del último sospechoso con la altura dada. Si sólo existe un sospechoso se mostrará solo su posición en la lista y si no exsite ninguno se escribirá *NO EXISTE*.

Entrada de ejemplo

```
4 8
2 5 7 8
3 5
5 5 5
5 8
3 4 6 7 9
```

Salida de ejemplo

```
3
0 2
NO EXISTE
```

Autor: Isabel Pita

Suficientemente disperso

El dueño del casino está muy preocupado por que los jugadores de ruleta puedan llegar a detectar que el crupier controla el número que saca en cada tirada. Todas las noches estudia la lista de números que han salido ese día y comprueba que sean *suficientemente dispersos*.

El dueño se conforma con que la diferencia entre el primer valor que se saca y el último sea mayor que una cantidad K. Además comprueba que los datos que salieron en la primera mitad de la noche y los datos que salieron en la segunda mitad sean tambien suficientemente dispersos, esto es que la diferencia entre el primer valor y el último tanto de la primera mitad como de la segunda sea mayor que K y además cada una de sus mitades sea también suficientemente dispersa.

Estudia únicamente secuencias con un número de elementos que sea potencia de dos para poder dividirlas siempre en dos partes iguales. Considera que un sólo valor siempre es suficientemente disperso.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se muestra el número de tiradas que se considera y el valor de dispersión que se precisa. En la segunda se muestran los números que han salido en cada tirada

El número de valores de cada caso de prueba es una potencia de 2. El valor de dispersión es un entero positivo mayor que cero. Los valores de cada tirada son números entre 0 <= N <= 1.048.576

Salida

Para cada caso de prueba se escribe en una línea *SI* si el vector está suficientemente disperso y *NO* si no lo está.

Entrada de ejemplo

```
4 3
6 1 3 9
4 3
3 10 12 14
8 4
20 2 0 4 14 8 5 10
```

Salida de ejemplo

```
SI
NO
SI
```

Vector parcialmente ordenado

Los valores de un vector pueden estar más o menos ordenados. En muchos casos nos conviene saber si un vector está *mas o menos* ordenado. Por ejemplo, el algoritmo *quicksort* tiene complejidad cuadrática si el vector está ordenado, mientras que la complejidad del algoritmo de inserción para vectores *casi* ordenados es *casi* lineal. En este problema diremos que un vector está *parcialmente ordenado*, si el valor máximo de su mitad derecha es mayor o igual que todos los valores de la mitad izquierda y el valor mínimo de su mitad izquierda es menor o igual que todos los valores de su mitad derecha. Además tanto la mitad izquierda como la derecha cumplen que están parcialmente ordenados.

Dado un vector de números enteros positivos se pide decidir si está parcialmente ordenado.

Requisitos de implementación.
Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de los valores del vector terminados con el valor cero que no forma parte del vector. El último caso de prueba consta únicamente del valor cero, y no debe tratarse.

El número de valores de cada caso de prueba es una potencia de 2.

Salida

Para cada caso de prueba se escribe en una línea *SI* si el vector está parcialmente ordenado y *NO* si no lo está.

Entrada de ejemplo

```
2 6 3 8 0
6 12 8 18 10 15 16 40 0
5 5 5 5 0
2 6 1 8 0
1 3 2 5 3 1 3 4 0
0
```

Salida de ejemplo

```
SI
SI
SI
NO
NO
```

Autor: Isabel Pita.

Degradado de una imagen

En diseño gráfico un degradado es un rango de colores ordenados con la intención de dar visualmente una transición suave y progresiva entre dos o más colores. En el software informático los degradados se consiguen reduciendo progresivamente el porcentaje del primer color mientras se aumenta el porcentaje del segundo. Si reducimos el porcentaje de blanco en un color y aumentamos el porcentaje de negro obtendremos colores más oscuros.

Vamos a representar los colores claros con valores pequeños (poca proporción de color negro) y los colores oscuros con valores altos (mayor proporción de color negro). Dada una imagen, representada mediante una matriz de números enteros, nos piden comprobar si es aproximadamente un degradado. En el problema consideramos que la matriz es un degradado si cada fila cumple que la suma de los colores de la mitad izquierda de la imagen es menor que la suma de los valores de la mitad derecha. Además, para conseguir un degradado uniforme cada mitad debe ser a su vez un degradado.

Requisitos de implementación.

Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.

La solución obtenida debe emplear la técnica de divide y vencerás.

La función que resuelve el problema debe recibir los datos de una fila en un vector y comprobar si es un degradado. Se pueden utilizar parámetros auxiliares para tratar la recursión.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba representa una matriz. Comienza con una línea que indica las dimensiones de la matriz n * m seguida de n líneas con m valores, que representan las filas de la matriz.

El número de valores en cada fila de la matriz m es una potencia de 2.

Salida

Para cada caso de prueba se escribe en una línea *SI* si la matriz es un degradado y *NO* si no lo es.

Entrada de ejemplo

```
3 4
1 3 2 6
2 5 4 11
2 3 3 10
3 8
2 6 7 12 1 8 15 22
2 5 2 6 2 3 3 4
1 2 3 4 5 6 7 8
2 8
2 5 2 16 2 8 16 50
1 3 4 5 2 8 14 15
1 4
1 3 2 3
```

Salida de ejemplo

```
SI
NO
SI
SI
```

Autor: Isabel Pita.

Battlestar Galactica

En un lejano lugar del universo, los *Cylon* han lanzado un ataque imprevisto contra las doce colonias de Kobol. Todo lo que queda de la humanidad es una nave de combate la *Battlestar Galactica* junto con un grupo de naves espaciales civiles. Juntos emprenden un viaje en busca de la decimotercera colonia: la Tierra, cuya ubicación es desconocida. El comandante *Adama* dirige la expedición desde la nave de combate y el resto de las naves se sitúan en línea detrás de la nave principal. El orden en que lo hacen es importante para poder defenderse mejor de los continuos ataques de los *Cylon*.

Después del último ataque las naves han quedado desordenadas y es preciso que recuperen el orden correcto. Sin embargo, el comandante sabe que los *Cylon* preparan un nuevo ataque para dentro de T unidades de tiempo. Si intercambiar el lugar de dos naves consecutivas lleva una unidad de tiempo, calcula si será posible restructurar la fila antes del ataque, en caso contrario, los pilotos *Apolo* y *Starbuck* tendrán que realizar una salida contra los *Cylon* para intentar retrasar el ataque.

Requisitos de implementación.

El problema se debe resolver de forma recursiva. Indicar el coste obtenido y la recurrencia .

La función que resuelve el problema debe recibir los datos en un vector y devolver el número de unidades de tiempo necesarias para reestructurar la fila de naves.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso consta de dos líneas, en la primera se indica el número de naves que viajan bajo la protección de la *Battlestar Galactica*. En la segunda línea se da la posición en que ha quedado cada nave después del ataque de los *Cylon*. Por ejemplo, si la entrada es 5 2 1 3 4, esto significa que la nave 5 (la que debería viajar en quinto lugar) está ahora en primera posición, la nave que debería estar en la segunda posición esta bien colocada, la nave 1, que debería estar en la primera posición, ocupa la tercera, y las naves que deberían estar en tercera y cuarta posición, han quedado en las posiciones 4 y 5.

Salida

Para cada caso de prueba se escribe en una línea el número de unidades de tiempo necesarias para reestructurar la fila.

Entrada de ejemplo

```
3
3 2 1
6
4 3 2 1 6 5
```

Salida de ejemplo

```
3
7
```

Autor: Isabel Pita.

Buscar el elemento perdido

Se tienen dos vectores, *a* y *b*, de enteros ordenados y distintos entre sí con *n* y *n-1* elementos respectivamente. Los elementos de *b* son los mismos que tiene *a* excepto uno que falta. Se pide implementar un algoritmo recursivo eficiente que encuentre ese valor que falta. Se debe indicar la recurrencia y el coste asintótico en el caso peor del algoritmo. (Examen junio 2017)

Requisitos de implementación.

Se debe implementar una función recursiva que reciba los dos vectores y devuelva el valor que falta. Pueden utilizarse más parámetros de entrada si hacen falta.

Se valorará la eficiencia de la solución, el uso eficiente de variables y parámetros, los comentarios sobre el algoritmo y sobre su coste.

Entrada

La entrada comienza con el número de casos a tratar. Cada caso consta de 3 líneas, en la primera se da el número de elementos del primer vector, en la segunda los elementos del primer vector y en la tercera los elementos del segundo vector.

Los valores de los vectores son números enteros.

Salida

Para cada caso de prueba se escribe en una línea el número que falta.

Entrada de ejemplo

```
5
5
10 20 30 40 50
10 30 40 50
5
10 20 30 40 50
10 20 30 40
5
10 20 30 40 50
20 30 40 50
1
6

2
1 8
1
```

Salida de ejemplo

```
20
50
10
6
8
```

Números caucásicos

Un vector de enteros mayores que 0 de longitud 2^n (donde n es un número natural) es *caucásico* si el valor absoluto de la diferencia entre el número de elementos pares de sus mitades es, a lo sumo, 2 y cada mitad también es *caucásica*. Un vector con un elemento es *caucasico*.

Algunos ejemplos:

- $\{2, 4, 6, 8 \parallel 1, 3, 5, 7\}$ No es *caucásico*, porque su primera mitad tiene 4 elementos pares y la segunda 0.
- $\{2, 4, 6, 8 \parallel 2, 8, 5, 10\}$ Es *caucásico*.
- $\{2, 4, 8, 12, 3, 7, 9, 21 \parallel 10, 20, 30, 1, 3, 5, 7, 40\}$ No es *caucásico* ya que la primera mitad no lo es.

Diseña un algoritmo *recursivo* que determine si un vector de longitud 2^n es *caucásico*.
Requisitos de implementación.
Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.
Comentar y explicar la solución propuesta.
El problema se debe resolver utilizando la técnica de divide y vencerás.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba se escribe en dos líneas. En la primera se indica el número de elementos de vector, en la segunda se muestran los valores. La entrada termina con un caso con 0 valores.

El número de valores de cada caso de prueba es 2^n siendo $0 \leq n \leq 20$. Los valores del vector son números enteros positivos.

Salida

Para cada caso de prueba se escribe en una línea *SI* si el vector es caucásico y *NO* si no lo es.

Entrada de ejemplo

```
4
2 6 3 8
8
6 12 8 18 10 15 16 40
8
6 12 8 18 11 15 17 41
16
1 3 2 5 3 1 3 4 4 6 8 2 3 5 7 2
0
```

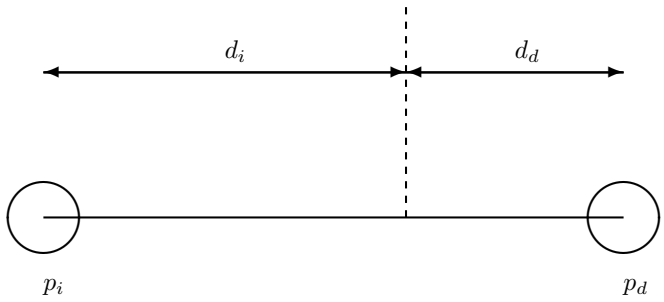
Salida de ejemplo

```
SI
SI
NO
NO
```

Autor: Isabel Pita.

Móviles

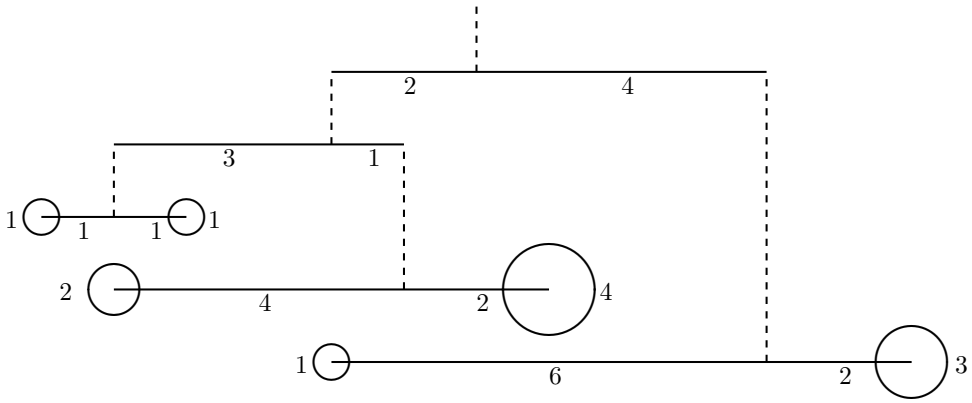
Antes de ser ese dispositivo de comunicación tan extendido, se entendía por móvil una estructura hecha con alambres y cuerdas de las que colgaban figuras coloridas, y que se colocaban sobre las cunas de los bebés para estimularles y entretenerles.



La figura representa un móvil simple. Tiene un único alambre colgado de una cuerda, con un objeto a cada lado. En realidad se puede ver como una "balanza" con el punto de apoyo en el sitio donde la cuerda está unida al alambre. Según el principio de la palanca, sabemos que está en equilibrio si el producto del peso de los objetos por sus distancias al punto de apoyo son iguales. Es decir si consideramos p_i como el peso colgado en el lado izquierdo, p_d el peso del lado derecho, y d_i la distancia desde el peso izquierdo a la cuerda y d_d de la cuerda al peso derecho, podremos decir que el móvil está en equilibrio si se cumple que $p_i \times d_i = p_d \times d_d$.

En móviles más complejos, cada peso puede ser sustituido por un "submóvil". En este caso se considera el peso del submóvil como la suma de los pesos de todos sus objetos, despreciando la cuerda y los alambres. Y consideraremos que está balanceado si $p_i \times d_i = p_d \times d_d$ y, además los submóviles de la izquierda y los de la derecha estan a su vez balanceados.

En ese caso no es tan trivial averiguar si está o no balanceado, por lo que te pedimos que nos escribas un programa que, dada una descripción de un móvil como entrada, determine si está o no en equilibrio.



Entrada

La entrada comienza con una línea con el número de casos de prueba que vienen a continuación.

Cada caso de prueba es un móvil, descrito con una o varias líneas, cada una de ellas conteniendo cuatro números enteros positivos, separados por un único espacio. Esos cuatro enteros representan las distancias de los extremos al punto de apoyo, así como sus pesos, en el orden p_i, d_i, p_d, d_d.

Si p_i o p_d (alguno de los pesos) es 0, en el extremo habrá colgado un *submóvil*, que estará descrito a continuación. Si un móvil tiene un submóvil en cada lado, primero se describirá el submóvil izquierdo.

Salida

Para cada caso de prueba, el programa indicará `SI` si el móvil que representa está en equilibrio, y `NO` en otro caso. Recuerda que se dice que un móvil está en equilibrio si *todos sus submóviles* y él mismo lo están.

Entrada de ejemplo

```
2
0 2 0 4
0 3 0 1
1 1 1 1
2 4 4 2
1 6 3 2
0 1 3 4
2 3 3 2
```

Salida de ejemplo

```
SI
NO
```

Notas

La implementación debe basarse en una función recursiva siguiente:

```
bool estaBalanceado(int &peso);
```

que lee de la entrada estandar la descripción de un móvil completo y devuelve si éste está balanceado y su peso (como parámetro de salida).

Nota

Este ejercicio debe verse en el contexto de la asignatura de Estructura de Datos y Algoritmos (EDA), FDI-UCM 2017/2018 (prof. Marco Antonio Gómez Martín). Por tanto *no* vale cualquier solución, sino sólo aquellas que utilicen los conceptos de EDA. Es muy posible que se den aclaraciones adicionales en clase a este respecto.