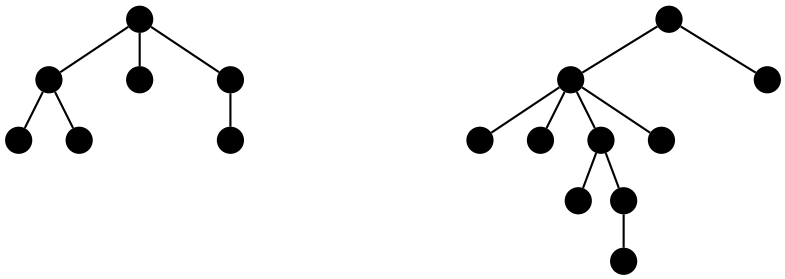
17
Altura de un árbol general

En un *árbol general* el número de hijos de cada nodo es variable, desde cero en el caso de una *hoja* hasta cierto número máximo que se llama el *grado* del árbol. La *altura* de una hoja es 1; si el árbol tiene hijos entonces su altura es 1 más el máximo de las alturas de sus hijos.

De los siguientes árboles generales, el de la izquierda tiene altura 3 y el de la derecha tiene altura 5.



Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en la descripción de un árbol general en una línea: primero aparece el número de hijos que tiene la raíz y a continuación aparecen las descripciones, como árboles generales, de cada uno de sus hijos, de izquierda a derecha.

Salida

Para cada árbol, se escribirá una línea con su altura.

Entrada de ejemplo

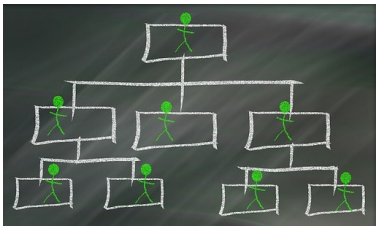
```
2
3 2 0 0 0 1 0
2 4 0 0 2 0 1 0 0 0
```

Salida de ejemplo

```
3
5
```

Autor: Alberto Verdejo.

18
Los becarios precarios



Aunque la definición de *becario* está relacionada con la obtención de subvenciones para realizar estudios o investigaciones[1], hoy en día cuando pensamos en ellos nos vienen a la cabeza estudiantes en los últimos cursos de sus estudios haciendo *prácticas en empresas* (con o sin remuneración) y estudiantes de doctorado.

En el árbol de jerarquía de la empresa, los becarios están siempre en la parte más baja recibiendo órdenes para realizar las tareas que nadie quiere hacer, y sin tener a nadie de menor rango a quien derivárselas.

Sin embargo, aunque desde fuera parece que todos los becarios están en las mismas condiciones, la realidad es que hay una especie de escalafón de becarios dependiendo de lo precaria que sea su situación. En concreto los becarios más precarios son aquellos que tienen a mucha gente por encima (o lo que es lo mismo, además de su jefe directo tienen muchos jefes indirectos).

Lo que queremos, dada la jerarquía de una empresa, es contar cuántos becarios precarios trabajan en ella.

Entrada

La entrada está formada por diferentes casos de prueba. Cada caso consiste en dos líneas. La primera contiene un entero $K \geq 1$ que indica el número de jefes que tiene que tener al menos un becario para ser considerado precario. En la segunda línea aparece la descripción de la jerarquía de la empresa: primero aparece el número de subordinados directos del jefe supremo (raíz de la jerarquía) y a continuación aparece la descripción de las subjerarquías que tienen como raíz a cada uno de estos subordinados, usando la misma técnica. Los becarios son los únicos que no tienen subordinados, y aparecen representados por un `0`.

Ninguna empresa tiene más de 10.000 empleados y nunca un becario tiene más de 1.000 jefes.

Salida

Para cada caso de prueba se escribirá el número de becarios precarios que tiene la empresa.

Entrada de ejemplo

```
1
3 2 0 0 0 2 0 0
2
3 2 0 0 0 2 0 0
3
3 2 0 0 0 2 0 0
```

Salida de ejemplo

```
5
4
0
```

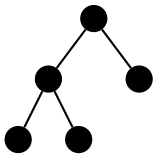
Autores: Marco Antonio Gómez Martín y Alberto Verdejo.

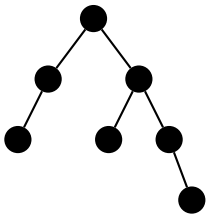
[1] Así lo dice la Real Academia Española.

19
Número de nodos, hojas y altura de un árbol binario

Dado un árbol binario, queremos calcular su número de nodos, cuántos de ellos son hojas, y cuál es la altura del árbol. Para calcular estas tres propiedades no es necesario conocer el elemento presente en cada nodo. Es suficiente conocer cómo están organizados en el árbol.

Por ejemplo, de los siguientes árboles, el de la izquierda tiene 5 nodos, 3 hojas y altura 3, y el de la derecha tiene 7 nodos, 3 hojas y altura 4.





Requisitos de implementación.

Se admiten dos soluciones (y conviene que se practiquen ambas). La primera consiste en extender la clase `bintree` con métodos públicos que devuelvan cada una de las tres propiedades. Estos métodos llamarán a otros privados y recursivos que recibirán como parámetro la raíz del árbol. La segunda solución consiste en implementar una función externa recursiva que recibirá como parámetro un árbol binario y devolverá las tres propiedades, recorriendo el árbol una sola vez.

Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en una cadena de caracteres con la descripción de un árbol binario (correspondiente al recorrido en preorden): el árbol vacío se representa con un punto (`.`); un árbol no vacío se representa con un `*` (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

Salida

Para cada árbol, se escribirá una línea con el número de nodos del árbol, el número de hojas y su altura, separados por un espacio en blanco.

Entrada de ejemplo

```
4
***..*..*..
***...**..*.*..
.
*..
```

Salida de ejemplo

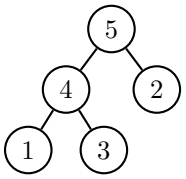
```
5 3 3
7 3 4
0 0 0
1 1 1
```

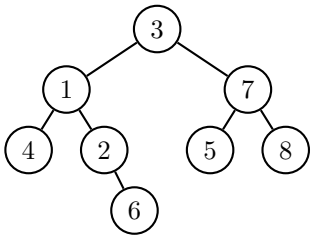
Autor: Alberto Verdejo.

20
La frontera

La *frontera* de un árbol binario es la secuencia formada por los elementos almacenados en las hojas del árbol, tomados de izquierda a derecha.

Por ejemplo, los siguientes árboles tienen como frontera 1, 3, 2 y 4, 6, 5, 8, respectivamente.





Requisitos de implementación.

Se puede extender la clase `bintree` con un método público que calcule la frontera (devolviéndola en un `vector`, por ejemplo), o implementar una función externa a la clase. En cualquier caso, el coste de la operación debe ser lineal en el número de nodos del árbol.

Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario: primero aparece su raíz (un entero no negativo), y a continuación la descripción del hijo izquierdo y después la del hijo derecho. El número –1 indica el árbol vacío.

Salida

Para cada árbol se escribirá su frontera en una línea, separando los elementos por espacios.

Entrada de ejemplo

```
4
5 4 1 -1 -1 3 -1 -1 2 -1 -1
3 1 4 -1 -1 2 -1 6 -1 -1 7 5 -1 -1 8 -1 -1
-1
2 -1 -1
```

Salida de ejemplo

```
1 3 2
4 6 5 8

2
```

Autor: Alberto Verdejo.

21
Elemento mínimo de un árbol

Dado un árbol binario no vacío, cuyos nodos almacenan valores que se pueden ordenar (se supone que tienen definido el operador $<$), queremos saber el elemento menor.

Requisitos de implementación.
El problema puede resolverse de diferentes maneras: extendiendo la clase `bintree` con un método público que devuelva el menor elemento; implementando una función externa a la clase recursiva, que explore el árbol buscando el menor; utilizando iteradores. En cualquier caso, la función tiene que ser genérica, es decir, no puede conocer el tipo `T` de los elementos almacenados en el árbol. Además, el coste de la operación debe ser lineal en el número de nodos del árbol.

Entrada

Cada caso de prueba ocupa dos líneas. En la primera aparecerá una `N` si el árbol es de números enteros positivos, o una `P` si el árbol es de palabras. En la segunda línea aparecerá la descripción del árbol: primero la raíz, después la descripción del hijo izquierdo y después la descripción del hijo derecho. Si el árbol es de números, se utilizará `-1` para indicar el árbol vacío; si es de palabras, se utilizará `#`.

Salida

Para cada árbol se escribirá una línea con el valor menor almacenado en sus nodos. En el caso de un árbol de palabras se entiende que el orden es el que proporciona el operador $<$ del tipo `string`.

Entrada de ejemplo

```
N
5 4 1 -1 -1 3 -1 -1 2 -1 -1
P
ramon luis maria # # carlos # # sara # #
```

Salida de ejemplo

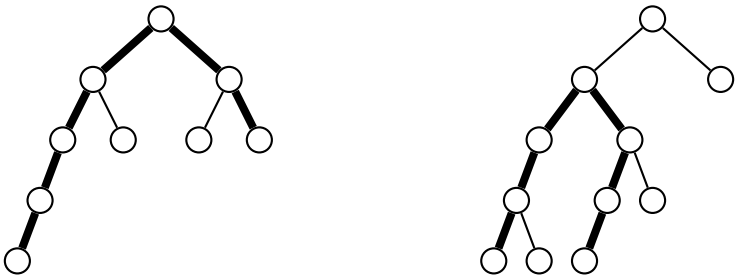
```
1
carlos
```

Autor: Alberto Verdejo.

22
Diámetro de un árbol binario

Definimos un *camino* en un árbol binario como una secuencia de nodos $n_1 n_2 \dots n_k$ sin repeticiones (por cada nodo del árbol se pasa como mucho una vez) tal que para todo par de nodos consecutivos $n_i n_{i+1}$ $(1 \leq i < k)$ uno de ellos siempre es padre del otro (n_i es padre de n_{i+1} o n_{i+1} es padre de n_i). Definimos la *longitud* de un camino $n_1 n_2 \dots n_k$ como el número de nodos que lo forman, k. Y definimos el *diámetro* de un árbol como la longitud del camino más largo del árbol.

Por ejemplo, los dos árboles siguientes tienen diámetro 7 y un camino de esa longitud aparece resaltado con trazo más grueso en los árboles.



Dado un árbol binario queremos averiguar su diámetro.

Requisitos de implementación.

Se implementará una función externa a la clase `bintree` que explore el árbol averiguando su diámetro. Esta función debe tener un coste lineal con respecto al número de nodos del árbol.

Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en una cadena de caracteres con la descripción de un árbol binario: el árbol vacío se representa con un '`.`'; un árbol no vacío se representa con un '`*`' (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

Salida

Para cada caso, se escribirá una línea con el diámetro del árbol correspondiente.

Entrada de ejemplo

```
5
.
*..
*.*..
*****....*..**..*..
*****..*...***...*..*..
```

Salida de ejemplo

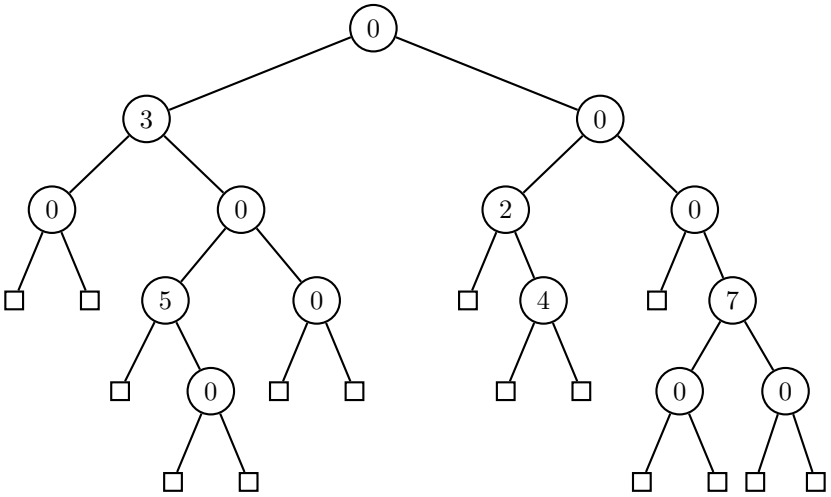
```
0
1
2
7
7
```

Autor: Alberto Verdejo.

23
Excursionistas atrapados

Durante el fin de semana varios grupos de excursionistas han intentado subir a una montaña. Debido a las condiciones meteorológicas han tenido que desistir en su intento y se encuentran atrapados en diversos puntos de la falda de la montaña. Las rutas que suben a la montaña se estructuran en forma de árbol binario. De la base de la montaña parten muchas rutas que se van juntando en diversas intersecciones hasta llegar solo uno o dos caminos a la cima. Se conoce la localización de cada grupo (se encuentran todos ellos en intersecciones de caminos) y su número de componentes.

Se están organizando equipos de rescate para ir a buscarlos. Para facilitar el rescate partirá un equipo del punto de la base de la montaña más cercano a cada grupo atrapado en la parte baja. Los equipos luego irán subiendo por los caminos de la montaña rescatando a los grupos que se encuentran más cerca de la cima. Los equipos nunca bajan para buscar a grupos de excursionistas. Por lo tanto se necesitan tantos equipos como grupos haya que no tengan otro grupo en ninguna de las rutas que suben hasta ese punto.



En el ejemplo, tenemos 5 grupos de excursionistas perdidos por la montaña (los nodos que no son 0). Se necesita un equipo de rescate para el grupo de 5, este equipo rescatará también al grupo de 3 que se encuentra en la ruta entre el 5 y la cima. Otro equipo rescatará al grupo de 4 y después al grupo de 2 que se encuentra encima. Un último equipo rescatará al grupo de 7. Si a un grupo lo pudieran rescatar dos equipos, uno por cada camino, es indiferente cuál de ellos lo rescata.

Requisitos de implementación.

Para resolver el problema implementa una función externa a la clase `bintree` que explore el árbol. El coste de esta función debe ser lineal en el número de nodos del árbol.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso consiste en la descripción de un árbol binario: la raíz seguida de la descripción del hijo izquierdo y del hijo derecho. Los valores de los nodos serán números naturales indicando el número de excursionistas de cada intersección. Los árboles vacíos se representan con el valor `-1`.

Salida

Para cada caso de prueba se escribirá el número de equipos de rescate necesarios para socorrer a todos los excursionistas seguido del número de excursionistas que se encuentran atrapados en la ruta que tiene un mayor número de excursionistas.

Entrada de ejemplo

```
3
0 3 0 -1 -1 0 5 -1 0 -1 -1 0 -1 -1 0 2 -1 4 -1 -1 0 -1 7 0 -1 -1 0 -1 -1
1 0 3 -1 -1 -1 0 4 -1 0 -1 -1 0 -1 -1
0 0 -1 -1 -1
```

Salida de ejemplo

```
3 8
2 5
0 0
```

Autores: Isabel Pita y Alberto Verdejo.

Problema número 203

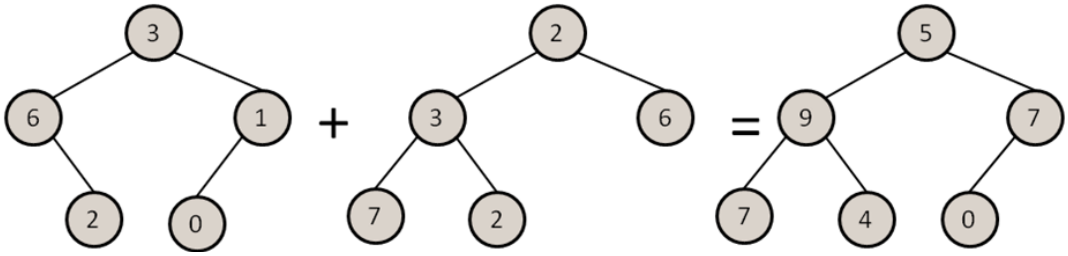
Suma de árboles

Tiempo máximo: 4,000 s Memoria máxima: 4096-4906 KiB

http://www.aceptaelreto.com/problem/statement.php?id=203

Si tenemos dos árboles binarios, podemos *sumarlos*. La idea de la suma consiste en poner un árbol encima del otro solapándolos, de forma que las raíces de ambos queden una encima de otra, sus hijos izquierdos también, etc. El árbol final tiene por tanto la estructura combinada de ambos árboles y, en los casos donde dos nodos coincidían, el valor del nodo final es la suma del contenido de los dos nodos originales.

Como ejemplo, aquí aparecen dos árboles y su árbol *suma*:



Entrada

Cada caso de prueba consiste en dos árboles de enteros. Cada uno de ellos comienza con el contenido de su raíz (un entero no negativo), al que le sigue la descripción del hijo izquierdo y después la del hijo derecho. El número `-1` indica la *ausencia de hijo* o lo que es lo mismo, el *árbol vacío*.

Los casos de prueba terminarán cuando *ambos* árboles sean vacíos. Ese caso de prueba no generará ninguna salida.

Salida

Para cada caso de prueba se escribirá una línea en la que aparecerá la descripción del árbol *suma* siguiendo el mismo formato que en la entrada (ten en cuenta que al final del último número *no* debe aparecer un espacio).

Se garantiza que el árbol resultado no tendrá nunca más de 10.000 nodos, y que cada uno de ellos no tendrá un número mayor que 10^9.

Entrada de ejemplo

```
1 -1 -1
2 -1 -1
1 2 -1 -1 3 -1 -1
5 -1 6 -1 -1
-1
-1
```

Salida de ejemplo

```
3 -1 -1
6 2 -1 -1 9 -1 -1
```

Autor: Marco Antonio Gómez Martín.

Revisor: Pedro Pablo Gómez Martín.

Problema número 204

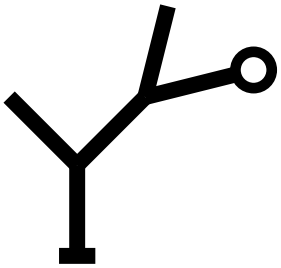
Árbol de navidad

Tiempo máximo: 1,000-3,000 s Memoria máxima: 4096-4906 KiB

http://www.aceptaelreto.com/problem/statement.php?id=204

Es aceptado universalmente que para que un árbol de navidad se considere que está bien decorado, los ornamentos deben estar distribuidos correctamente por todas sus ramas. Esto tiene dos razones de ser: la primera es estética y la segunda de equilibrio. Si todos los adornos se concentran en la misma región del árbol, éste se caerá.

Algunos informáticos, además, solo admiten en casa árboles de navidad *binarios*. Esos árboles empiezan con un tronco que se divide en dos subárboles distintos. Cada uno de los dos subárboles puede a su vez, dividirse en otros dos subárboles más pequeños, etc., hasta que se llega a lo que podemos llamar *hoja*. Es en esas hojas donde se puede poner la decoración, colocando las famosas bolas de navidad o alguna otra figurita.



Como ejemplo, el árbol de la figura de la derecha tiene un tronco que se divide en dos subárboles. El de la izquierda termina directamente en una hoja, mientras que el de la derecha se divide a su vez en dos subárboles no divididos, uno de ellos con decoración y el otro sin ella.

Pues bien, se entiende que un árbol de navidad binario está decorado correctamente cuando en todos los sitios donde una rama (o el tronco) se divide en dos suárboles, el número de bolas de ambos no difiere en más de una unidad y, además, los dos subárboles están, vistos de forma individual, bien decorados. Con esta definición el árbol de la figura estaría bien decorado, mientras que si en la rama vacía *hermana* de la que tiene la bola colocáramos otra bola más, dejaría de estarlo.

Entrada

La entrada consta de una serie de árboles binarios de navidad, descritos cada uno en una línea que no superará los 100.000 caracteres. La línea contiene el recorrido en *preorden* del árbol binario. Más concretamente, cada carácter representa un nodo del árbol, ya sea interno o un nodo hoja. El carácter '`*`' indica la aparición de un adorno navideño (recuerda que sólo aparecen en las hojas), mientras que el carácter '`.`' representa una hoja sin adorno. Por su parte, el carácter '`Y`' indica una división en el árbol, al que seguirá la descripción de los dos subárboles.

Salida

Para cada caso de prueba se mostrará una línea indicando si el árbol de navidad está correctamente decorado o si corre el riesgo de caerse. Si la decoración está bien colocada se escribirá `OK`, mientras que si no lo está, se escribirá `KO`.

Entrada de ejemplo

```
Y.Y*.
Y.Y**
*
```

Salida de ejemplo

```
OK
KO
OK
```

Autor: Marco Antonio Gómez Martín.

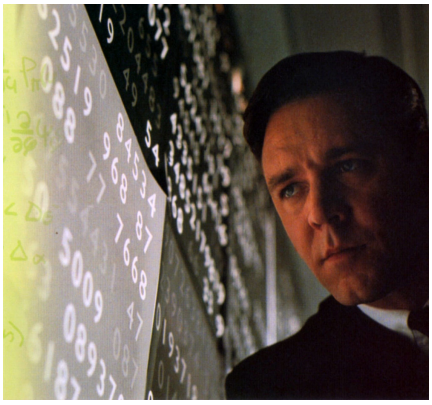
Revisor: Pedro Pablo Gómez Martín.

Problema número 228

Codificación espejo

Tiempo máximo: 4,000 s Memoria máxima: 4096 KiB

http://www.aceptaelreto.com/problem/statement.php?id=228



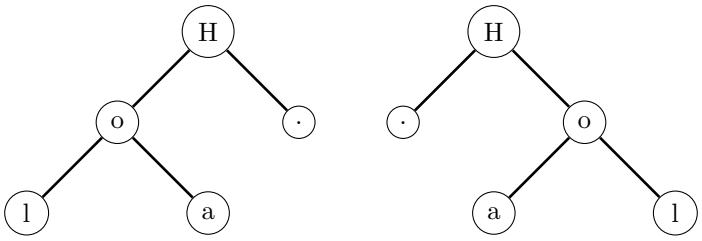
El envío de mensajes cifrados para evitar miradas indiscretas se lleva estudiando desde la antigüedad. El método más simple consiste en manejar tablas de traducción que contienen, para cada letra, por qué otra letra se sustituirá en el mensaje cifrado. El descifrado maneja las tablas inversas que permiten recomponer el mensaje original. Además, para dificultar el trabajo a los posibles espías, muchos de estos mecanismos primitivos *no* consideran el espacio separador de palabras como parte del mensaje por lo que el receptor al decodificar ve todas las letras seguidas y tiene que ser él, a la vista de las mismas, el que separe las palabras.

Existe otro mecanismo simple que consiste en, sencillamente, *cambiar de orden* las letras, siguiendo unas determinadas reglas. Si esas reglas están bien elegidas, el proceso de codificación y el de decodificación es el mismo, por lo que no se necesita implementar algoritmos distintos en cada lado de la comunicación.

El método que hoy proponemos cumple, parcialmente, esta propiedad. Y, como decíamos antes, no codifica los espacios, por lo que el receptor tendrá que colocarlos él mismo a partir del significado de lo que vaya leyendo.

El mecanismo de codificación/decodificación se basa en representar el mensaje como un árbol binario tal que su recorrido en *preorden* es el mensaje a codificar (sin espacios); lo normal es que existan muchos posibles árboles que cumplan lo anterior, cualquiera de ellos vale. La codificación espejo entonces construye un nuevo árbol invirtiendo el original para conseguir su *imagen especular*. El recorrido en preorden de ese nuevo árbol es el mensaje que se envía.

A modo de ejemplo, imaginemos que queremos enviar el mensaje "`Hola.`". En la figura de la izquierda aparece un posible árbol binario cuyo recorrido en preorden coincide con el mensaje que queremos transmitir. A la derecha aparece su imagen especular, cuyo recorrido en preorden es el que se envía.



Árbol del mensaje original Árbol del mensaje codificado

Para que el receptor del mensaje pueda recomponer el árbol enviado a partir únicamente del recorrido en preorden, se aprovecha que los espacios del propio mensaje no se envían. Gracias a eso, se pueden utilizar de forma segura para indicar ausencia de hijo. Es por esto que la forma de transmitir el árbol de la derecha es:

- Primero aparece la raíz del árbol, la letra `H`.
- A continuación aparece el hijo izquierdo completo. Es decir el carácter `.` seguido de dos espacios que indican que ese nodo no tiene hijos.
- Posteriormente aparece el hijo derecho, cuya codificación comienza con su raíz, `o`, seguida del hijo izquierdo (`a` y dos espacios) y el hijo derecho (`l` y dos espacios adicionales).

Entrada

La entrada consiste en diversas líneas, cada una con un mensaje codificado utilizando la codificación espejo. Se garantiza que cada línea representará un árbol binario válido de un máximo de 5.000 nodos y cuya altura no será mayor que 3.000.

Importante: en el ejemplo de entrada las tres líneas terminan con *dos* espacios tras el último carácter visible; en caso contrario la línea no representaría correctamente el árbol binario.

Salida

Para cada caso de prueba, el programa escribirá una línea con el mensaje descifrado. Ten en cuenta que lo que escribimos es el mensaje descifrado y no el recorrido en preorden del árbol espejo, por lo que en la salida *no* aparecerán los espacios.

Entrada de ejemplo

```
H.  oa  l
Hol  a  .
as  do  i
```

Salida de ejemplo

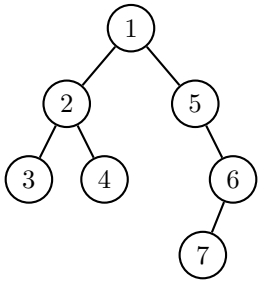
```
Hola.
H.oal
adios
```

Autor: Marco Antonio Gómez Martín.

Revisor: Pedro Pablo Gómez Martín.

24
Reconstrucción de un árbol binario

El semestre pasado estuvimos diseñando árboles binarios, con números enteros en los nodos. Los más bonitos eran aquellos que tenían todos los valores distintos, como este:



Cuando acabamos con ellos, guardamos los más bonitos en un archivo, serializados según su recorrido en preorden (ya sabes: raíz, hijo izquierdo, hijo derecho). Por aquello de la tolerancia a fallos, los guardamos en un segundo archivo, pero ahí serializados según su recorrido en inorden (hijo izquierdo, raíz, hijo derecho). *Y ¡menos mal!* Creemos que con uno solo de los recorridos no habríamos podido ahora reconstruir los árboles y tampoco estamos seguros de poderlo hacer teniendo los dos. ¿Nos ayudas?

Requisitos de implementación.

Se implementará una función que reciba los recorridos en preorden e inorden de un árbol (por ejemplo en dos vectores) y devuelva un objeto de la clase `bintree` que represente un árbol que tenga esos dos recorridos.

Entrada

La entrada está formada por una serie de casos. Cada caso consta de dos líneas: la primera contiene el recorrido en preorden de un árbol binario (cuyos nodos tenían valores entre 1 y 10^6) y la segunda el recorrido en inorden de ese mismo árbol.

Salida

Para cada caso, se escribirá el recorrido en postorden del árbol reconstruido.

Entrada de ejemplo

```
1 2 3 4 5 6 7
3 2 4 1 5 7 6
2 4 6
2 4 6
1 2 4 8 5 3 6 7
8 4 2 5 1 6 3 7
```

Salida de ejemplo

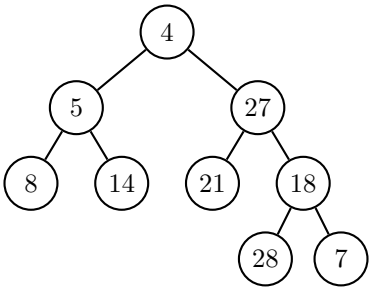
```
3 4 2 7 6 5 1
6 4 2
8 4 5 2 6 7 3 1
```

Autor: Alberto Verdejo.

25
La barrera de los primos

En un árbol binario cuyos nodos almacenan números naturales, decimos que un nodo es *accesible* si el camino que va desde la raíz hasta él no contiene ningún número primo. Estamos interesados en encontrar el múltiplo de 7 en un nodo accesible lo más cercano a la raíz (en caso de empates, preferimos el que se encuentre más a la izquierda).

Por ejemplo, en el siguiente árbol el número buscado es el 21, que se encuentra a profundidad 3.



Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario: primero aparece su raíz (un número natural mayor que 1 y menor que 5.000), y a continuación la descripción del hijo izquierdo y después la del hijo derecho, dadas de la misma manera. El número –1 indica el árbol vacío.

Salida

Para cada árbol se escribirá el múltiplo de 7 que aparezca en un nodo accesible lo más cercano a la raíz (y en caso de empate el colocado más a la izquierda) y la profundidad a la que se encuentra dicho nodo. En caso de que no haya ninguno se escribirá `NO HAY`.

Entrada de ejemplo

```
4
4 5 8 -1 -1 14 -1 -1 27 21 -1 -1 18 28 -1 -1 7 -1 -1
4 6 8 -1 -1 14 -1 -1 27 21 -1 -1 18 28 -1 -1 7 -1 -1
7 14 -1 -1 21 -1 -1
14 21 -1 -1 28 -1 -1
```

Salida de ejemplo

```
21 3
14 3
NO HAY
14 1
```

Autor: Alberto Verdejo.

Problema número 218

Ductilidad de los árboles binarios

Tiempo máximo: 1,000-3,000 s Memoria máxima: 4096 KiB

http://www.aceptaelreto.com/problem/statement.php?id=218

La maleabilidad es la propiedad de un material sólido de adquirir una deformación mediante una compresión sin fracturarse; cuando un material es maleable se pueden hacer láminas muy finas con él. De forma similar, la ductilidad es otra propiedad que, en vez de láminas, permite hacer alambres o hilos muy finos.

Los árboles binarios están hechos de un material muy dúctil, que nos permite estirar el árbol en un hilo muy fino, de tan solo un elemento de grosor. Dependiendo del tipo de manipulación utilizada para conseguir el hilo, los átomos (nodos) originales del árbol (que contienen sus elementos) quedan colocados de formas distintas. Hay cuatro formas de manipulación básicas: la obtención por el mecanismo de preorden, de inorden, de postorden y de niveles.

Un buen joyero de árboles es capaz de recuperar el árbol original a partir de dos finos cables sacados de dos árboles iguales. ¿Eres un buen joyero?

Entrada

La entrada consistirá en distintos casos de prueba, cada uno de ellos ocupando tres líneas. La primera línea contiene el número N de nodos del árbol original. A continuación vienen dos líneas cada una con N enteros positivos, que se corresponden con el recorrido del árbol en inorden y postorden respectivamente. Se garantiza que el árbol original (y por tanto los recorridos dados) no tiene valores repetidos.

La entrada termina con una línea con un `0`.

Salida

Para cada caso de prueba se escribirá el recorrido en preorden del árbol. Se garantiza que el árbol no tendrá más de 1.000 nodos.

Entrada de ejemplo

```
1
1
1
3
0 1 2
0 1 2
3
0 1 2
0 2 1
0
```

Salida de ejemplo

```
1
2 1 0
1 0 2
```

Autor: Marco Antonio Gómez Martín.

Revisor: Pedro Pablo Gómez Martín.

Problema número 231

Conversor de expresiones

Tiempo máximo: 2,000-3,000 s Memoria máxima: 4096 KiB

http://www.aceptaelreto.com/problem/statement.php?id=231

Una forma habitual en informática de representar las expresiones aritméticas es utilizar la que se conoce como *notacion postfija*. En ella los operadores en vez de aparecer entre los dos operandos (como en "`3 + 5`"), aparecen *después* de ellos ("`3 5 +`").

La ventaja de esta notación es que no presenta ambigüedad en el orden de evaluación de los operadores y por lo tanto nunca son necesarios los paréntesis. Además, hacer la evaluación es fácil por medio de una pila: cada vez que nos encontramos un operando se añade en la pila, y cuando se lee un operador se extraen los dos primeros elementos de la pila, se combinan ambos en función del operador, y se apila el resultado obtenido. Cuando el procesado termina, en la pila queda un único elemento que se corresponde con la evaluación de la expresión completa.

Como ejemplo sencillo, podemos evaluar la expresión `7 / (5 - 2)` que en notación *postfija* se escribe como `7 5 2 - /`. La evaluación comienza apilando los tres operandos que nos encontramos, el 7, 5 y 2; al encontrarnos con el `-` se extraen los dos últimos y se restan (`5 - 2`), el resultado (`3`) se añade en la pila. Por último, el procesado del `/` implica extraer el 3, luego el 7, hacer la división y apilar el `2` que es el resultado de la expresión.

Existe una notación distinta que permite evaluar la expresión utilizando una idea similar pero usando una *cola* en vez de una pila. La expresión anterior se expresa con `2 5 - 7 /`. Con una cola, se añade primero el 2 y el 5 en la cola, se extraen para restarlos, y el resultado se inserta de nuevo en la cola; acto seguido se añade el 7 y por último se dividen los dos elementos.

A partir de una expresión que utiliza notación *postfija* evaluable con una pila, ¿eres capaz de convertirla al formato que permite la evaluación con una cola?

Entrada

La entrada consiste en una sucesión de expresiones en evaluación *postfija*, cada una en una línea. Los operadores de las expresiones posibles serán los operadores binarios de suma, resta, multiplicación, división y módulo (`+`, `-`, `*`, `/` y `%`). Los operandos serán siempre números con un único dígito. No habrá espacios en las expresiones que, además, tendrán longitud menor de 20.000 caracteres.

Salida

Para cada caso de prueba se mostrará la expresión equivalente que permita la evaluación utilizando una cola en lugar de una pila, también sin espacios.

Entrada de ejemplo

```
752-/
3
```

Salida de ejemplo

```
25-7/
3
```

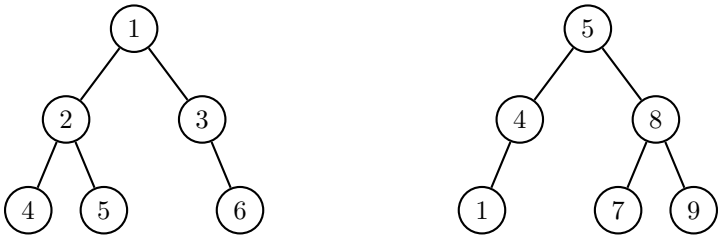
Autor: Marco Antonio Gómez Martín.

Revisor: Pedro Pablo Gómez Martín.

26
¿Es un árbol binario de búsqueda?

Un *árbol binario de búsqueda* es un árbol binario cuyos nodos almacenan valores que se mantienen ordenados de la siguiente manera: la raíz del árbol contiene un valor que es estrictamente mayor que todos los valores en el hijo izquierdo y estrictamente menor que todos los valores en el hijo derecho; además, ambos hijos son árboles binarios de búsqueda.

De los siguientes árboles (con números enteros como valores) solamente el de la derecha es un árbol binario de búsqueda.



Dado un árbol binario, el problema consiste en decidir si es o no un árbol binario de búsqueda.

Entrada

La entrada comienza con el número de casos que vienen a continuación. Cada caso de prueba consiste en una línea con la descripción de un árbol binario: primero aparece su raíz (un entero no negativo), y a continuación la descripción del hijo izquierdo y después la del hijo derecho. El número –1 indica el árbol vacío. Los árboles nunca contendrán más de 4.000 nodos.

Salida

Para cada árbol se escribirá `SI` si el árbol es un árbol binario de búsqueda y `NO` si no lo es.

Entrada de ejemplo

```
4
1 2 4 -1 -1 5 -1 -1 3 -1 6 -1 -1
5 4 1 -1 -1 -1 8 6 -1 -1 9 -1 -1
-1
2 2 -1 -1 -1
```

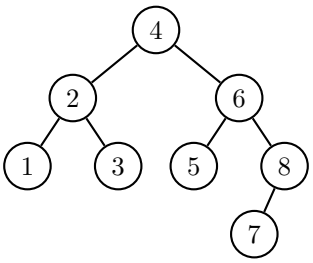
Salida de ejemplo

```
NO
SI
SI
NO
```

Autor: Alberto Verdejo.

27
Reconstrucción de un árbol binario de búsqueda

Este semestre nos ha tocado diseñar bonitos árboles binarios de búsqueda, con números enteros en los nodos. En estos árboles la raíz contiene un valor que es estrictamente mayor que todos los del hijo izquierdo y estrictamente menor que todos los del hijo derecho y, además, ambos hijos son también árboles binarios de búsqueda. Aquí tenemos un ejemplo:



Cuando hemos acabado con ellos, para evitar problemas también los hemos guardado en dos archivos, en uno serializados según su recorrido en preorden (raíz, hijo izquierdo, hijo derecho) y en el otro serializados según su recorrido en inorden (hijo izquierdo, raíz, hijo derecho). Pero ha ocurrido un desastre y hemos perdido el segundo fichero. ¿Tendremos que volver a empezar a diseñar árboles? Aunque suene muy divertido, en realidad no lo es tanto. ¿Podremos reconstruir los árboles originales a partir solamente de su recorrido en preorden?

Requisitos de implementación.

Se implementará una función que reciba el recorrido en preorden de un árbol binario de búsqueda (por ejemplo en un `vector`) y devuelva un objeto de la clase `bintree` que represente un árbol que tenga ese recorrido.

Entrada

La entrada está formada por una serie de casos. Cada caso ocupa una línea y contiene el recorrido en preorden de un árbol binario de búsqueda (cuyos nodos tenían valores entre 1 y 10^6).

Salida

Para cada caso, se escribirá el recorrido en postorden del árbol reconstruido.

Entrada de ejemplo

```
4 2 1 3 6 5 8 7
2 4 6
12 8 10 20
```

Salida de ejemplo

```
1 3 2 5 7 8 6 4
6 4 2
10 8 20 12
```

Autor: Alberto Verdejo.