1.

919 . Insertador de árbol binario completo

Medio 971 92

Compañías

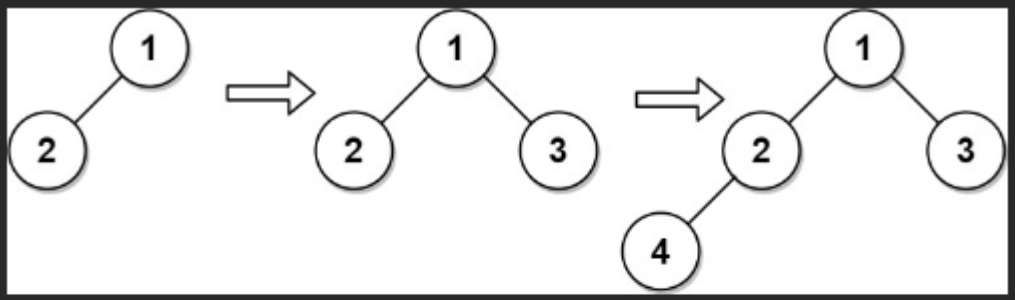
Un **árbol binario completo** es un árbol binario en el que todos los niveles, excepto posiblemente el último, están completamente llenos y todos los nodos están lo más a la izquierda posible.

Diseñe un algoritmo para insertar un nuevo nodo en un árbol binario completo manteniéndolo completo después de la inserción.

Implementar la `CBTInserter` clase:

- `CBTInserter(TreeNode root)` Inicializa la estructura de datos con la `root` del árbol binario completo.
- `int insert(int v)` Inserta a `TreeNode` en el árbol con valor `Node.val == val` para que el árbol permanezca completo y devuelve el valor del padre del insertado `TreeNode`.
- `TreeNode get_root()` Devuelve el nodo raíz del árbol.

Ejemplo 1:



Aporte

```
["CBTInserter", "insertar", "insertar", "get_root"]
[[[1, 2]], [3], [4], []]
```

Producción

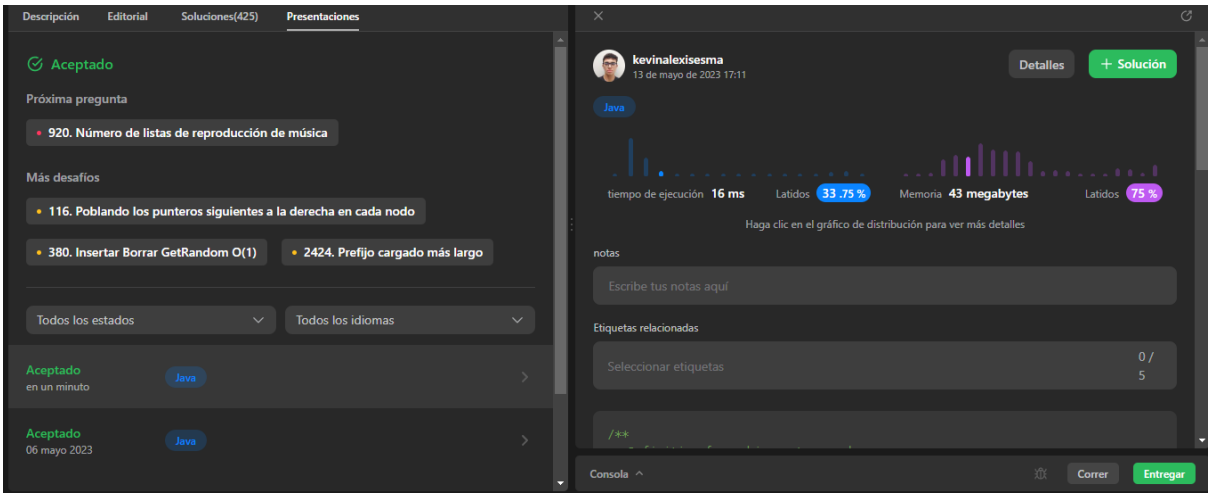
```
[nulo, 1, 2, [1, 2, 3, 4]]
```

Explicación

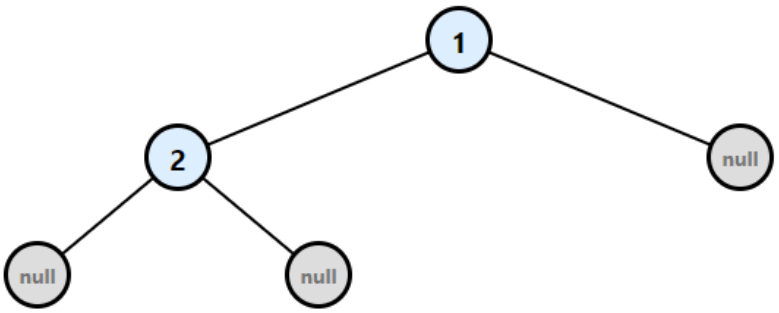
```
InsertadorCBTInsertorcBTInserter = new InsertadorCBTI ([1, 2]);
cBTInserter.insert(3); // devuelve 1
cBTInserter.insert(4); // devuelve 2
cBTInserter.get_root(); // devuelve [1, 2, 3, 4]
```

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class CBTInserter {

    List<TreeNode> tree;
    public CBTInserter(TreeNode root) {
        tree = new ArrayList<>();
```

```
        tree.add(root);
        for (int i = 0; i < tree.size(); ++i) {
            if (tree.get(i).left != null)
tree.add(tree.get(i).left);
            if (tree.get(i).right != null && tree.get(i).left
!= null) tree.add(tree.get(i).right);
        }
    }

    public int insert(int val) {
        TreeNode node = new TreeNode(val);
        int tamanio = tree.size();
        tree.add(node);
        if(tamanio%2 == 0){
            tree.get((tamanio-1)/2).right = node;
        }else{
            tree.get((tamanio-1)/2).left = node;
        }
        return tree.get((tamanio-1)/2).val;
    }

    public TreeNode get_root() {
        return tree.get(0);
    }
}

/**
 * Your CBTInserter object will be instantiated and called as
such:
 * CBTInserter obj = new CBTInserter(root);
 * int param_1 = obj.insert(val);
 * TreeNode param_2 = obj.get_root();
 */
```



EJEMPLO GRÁFICO: [[[1,2]] , [3], [4], []]



Se inserta el primer nodo, con un hijo a su izquierda, como dato inicial, al ser el primero, se ejecuta en el método `CBTInserter`, donde se crea el array List para ordenar las entradas y a su vez, por medio de condicionales, se verifica: si no tiene lado izquierdo, no agregar el derecho, es decir:

```
[1,2,3] <- permitir esto
[1, null, 3] <- esto no
```

[3], [4], [] el siguiente en el vector es el nodo 3.

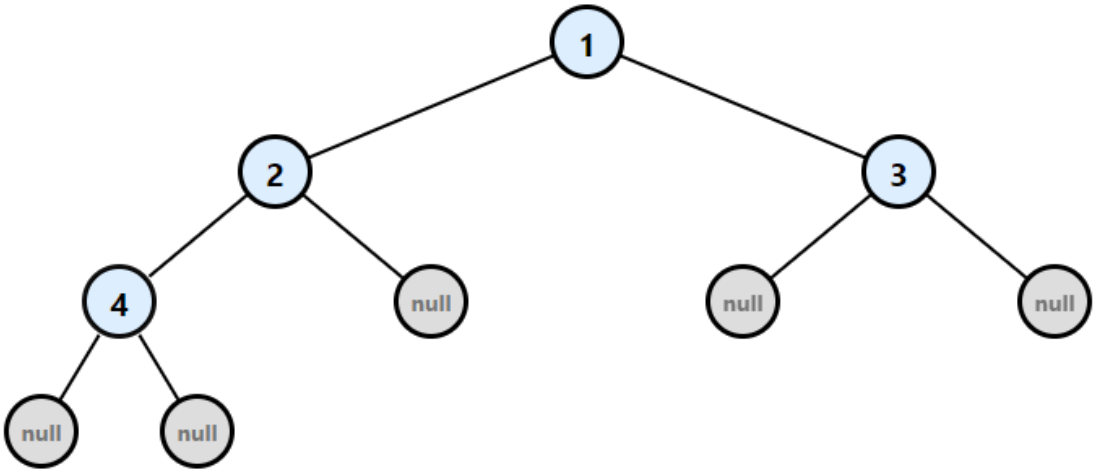
Ingresando por el método:

```
public int insert(int val)
```

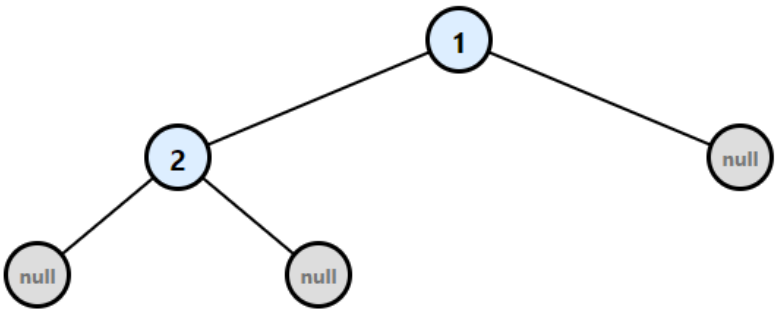
Se agrega al arraylist, pero antes de esto, se toma su tamaño para identificar su posición.

```
TreeNode node = new TreeNode(val);

Aquí se toma el tamaño antes de la agregación:
        int tamanio = tree.size();
```

```
Se agrega el nodo.
        tree.add(node);
Se pregunta si es par o impar el tamaño del árbol, si es par,
es porque el último en ingresar fue un nodo en el lado left,
entonces se inserta en su lado right:
        if(tamanio%2 == 0){
            tree.get((tamanio-1)/2).right = node;
        }else{

Si no es par, significa que el último en ingresar fue un nodo
en el lado right, por ende, este irá en el lado left.

En los dos casos, para hallar la posición en que entrarán en
su lado respectivo, se hace la siguiente ecuación:

(Tamaño-1)/2 = si hay 4 nodos, sería 3/2, que daría 1.5,
función piso, que está determinada en java, sería uno.
Posición uno del array:
[nodo1, nodo2, nodo3, nodo4]
            tree.get((tamanio-1)/2).left = node;
```



En este caso sería el nodo 2, entonces en el nodo dos, lado derecho, entraría el nodo 5.
En nuestro caso original:



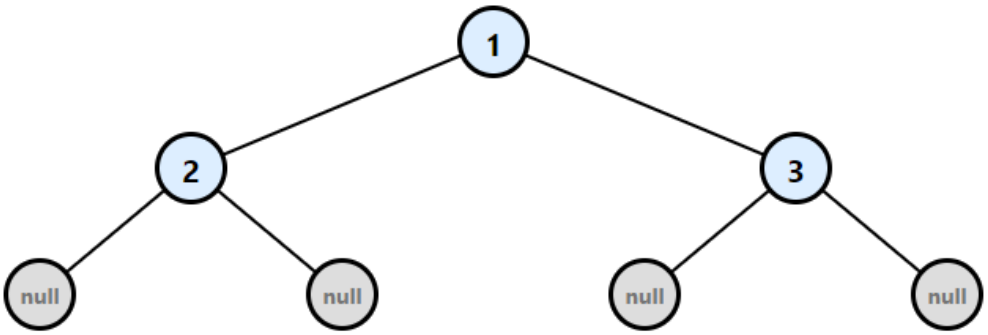
El nodo 3, en el array hay 2 nodos, con la función: (tamanio-1)/2,

$$\frac{(2-1)}{2} = \frac{1}{2} = 0.5,\ función\ piso\ de\ java:\ 0$$

Entonces en el array posición 0, sería el nodo 1:

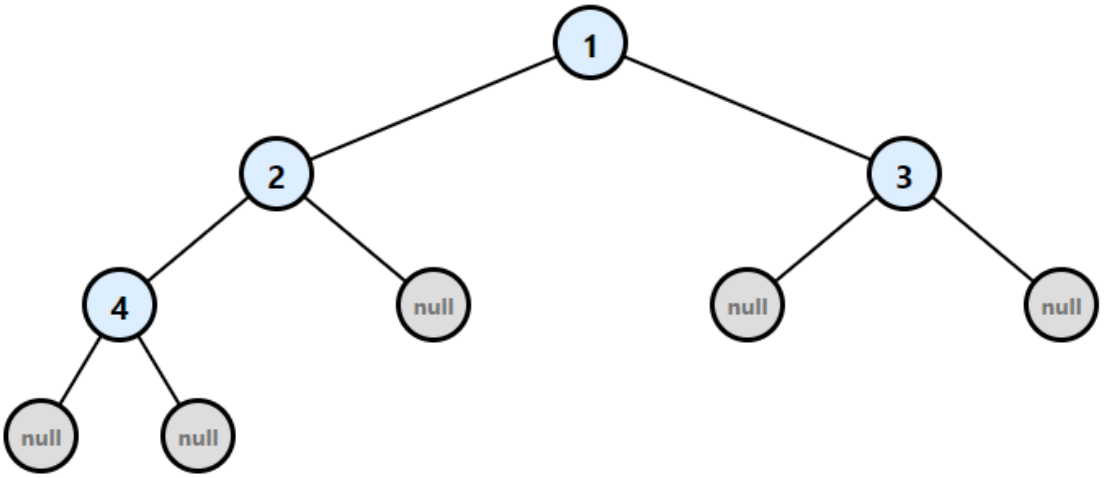
```
tree.get((tamanio-1)/2).right = node;
```

Para agregarlo quedaría:



Así mismo con la [4]:

```
tree.get((3-1)/2).left = node4;
tree.get(2/2).left = node4;
tree.get(1).left = node4;
```



2.

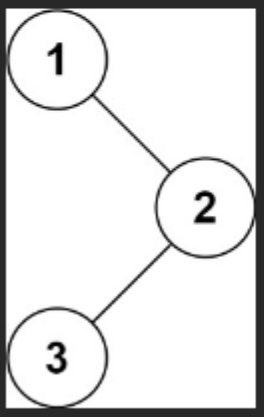
94 . Recorrido en orden de árbol binario

Fácil 11.4K 576

Compañías

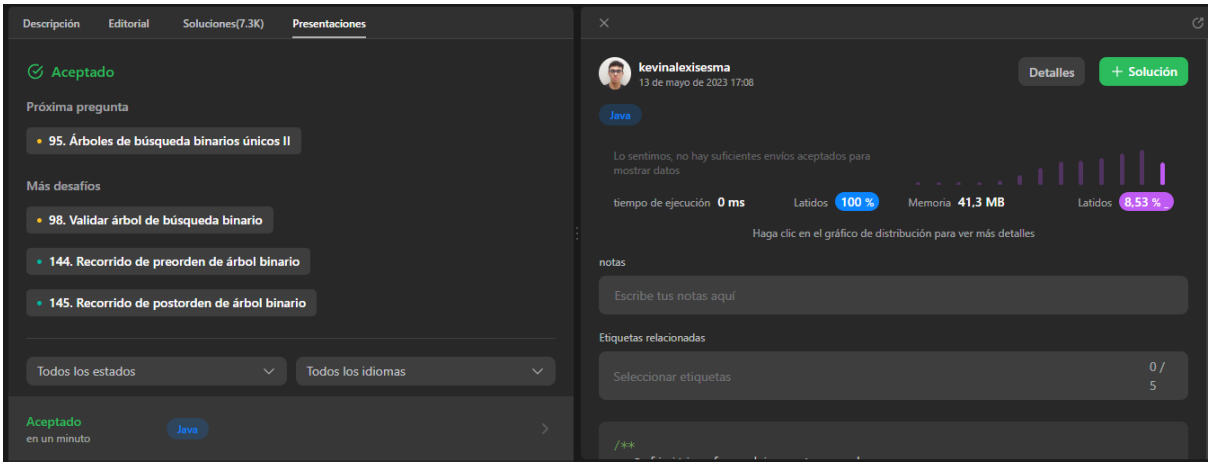
Dada la `root` de un árbol binario, devuelve *el recorrido en orden de los valores de sus nodos* .

Ejemplo 1:

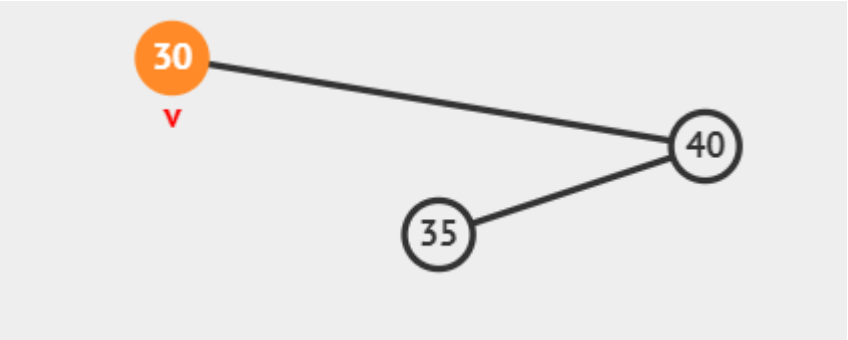


Entrada: raíz = [1, nulo, 2,3]
Salida: [1,3,2]

```java
class Solution {
    List<Integer> inorder = new ArrayList<>();
    public List<Integer> inorderTraversal(TreeNode root) {
        inorderT(root);
        return inorder;
    }

    public void inorderT(TreeNode root) {
        if(root != null){
            if(root.left !=  null){
                inorderT(root.left);
            }
            inorder.add(root.val);
            if(root.right != null){
                inorderT(root.right);
            }
        }
        return;
    }
}
```

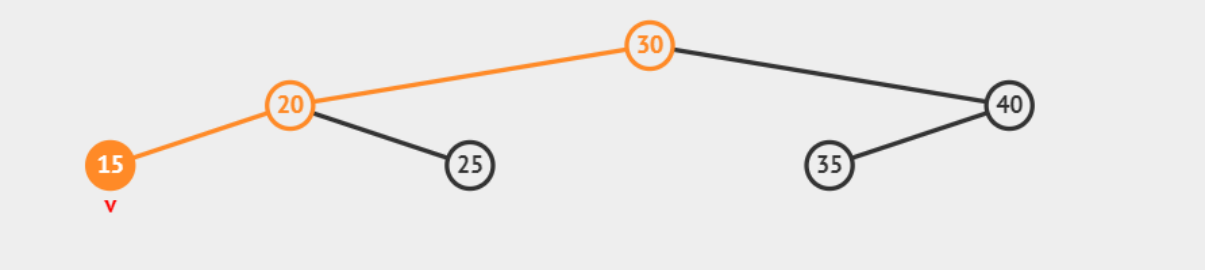


Se tiene este árbol:

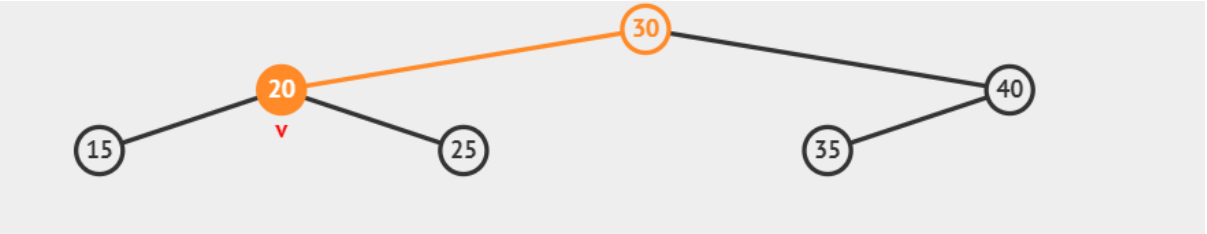


Al mostrar sus valores en Inorden, sería lo siguiente:

```
Nos dan la raíz como primer parámetro
public void inorderT(TreeNode RAIZ) {
        Pregunta si la raiz es diferente de nulo, para no
realizar nada cuando nos den un árbol vació y como caso base
para la recursividad.
        if(RAIZ != null){
            Se valida si la raíz tiene izquierda, si es así,
se autollama pero con el nodo left: inorderT(RAIZ.left)
            if(RAIZ.left !=  null){
                inorderT(RAIZ.left);
            }
ya cuando llega a su izquierda máxima, se agrega su valor a
la Lista de enteros.
            inorder.add(RAIZ.val);
y se pregunta si tiene derecha par hacer lo mismo que la
izquierda:
            if(RAIZ.right != null){
                inorderT(RAIZ.right);
            }
        }
        return;
```

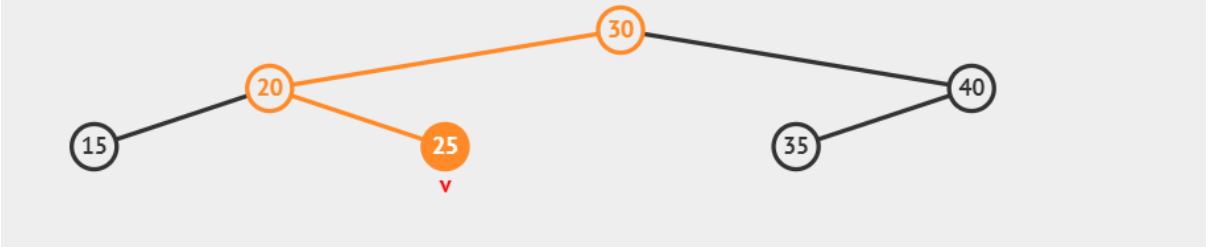


Una vez en el ultimo de la izquierda, guarda su valor en la lista:
{15}
se retorna a su padre:



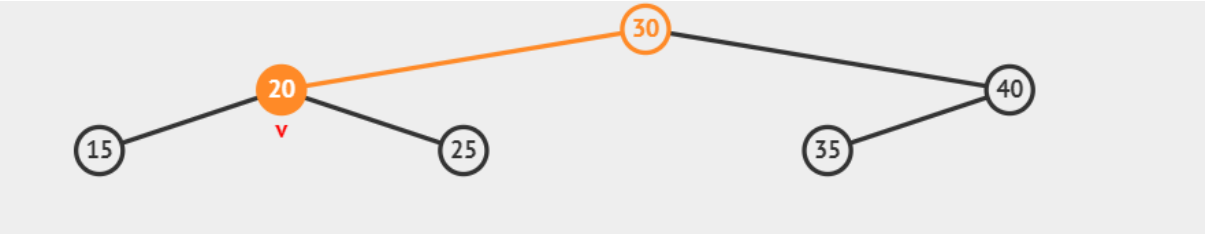

```
                inorderT(root.left);  Estamos aquí
            }
          Luego de terminar con toda su izquierda, vuelve a
su raíz, guarda su valor y va por la derecha:
            inorder.add(root.val);
            if(root.right != null){
                inorderT(root.right);
            }
```

{15,20}
Pregunta por sus derechas:

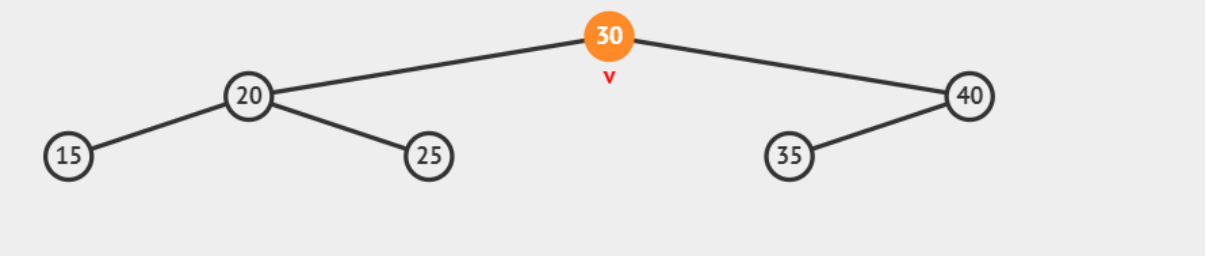


Al encontrar su ultima derecha, lo guarda:
{15,20,25}

```
            if(root.right != null){
                inorderT(root.right);
            }
Cuando termine con sus derechas, entonces se devuelve a su
raíz.
```



Como ya estamos en el final del código, se devuelve a su padre raíz:

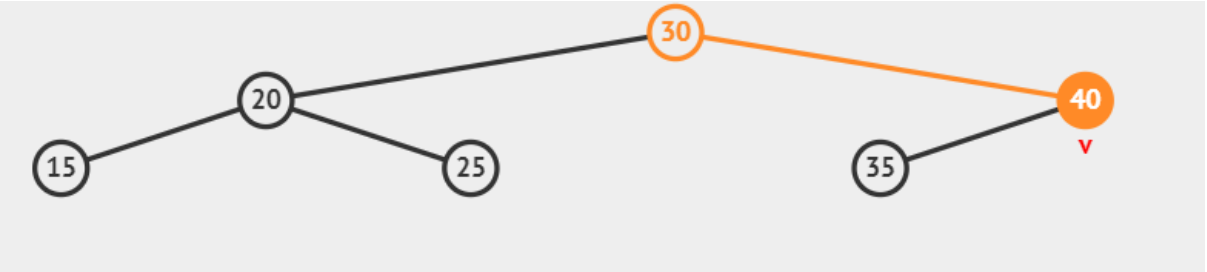


guarda la raíz:

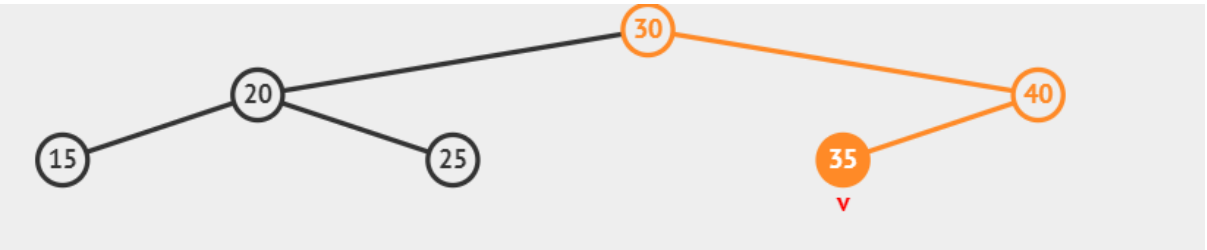
```
inorder.add(root.val);
```

{15,20,25,30}
Y ahora empieza su lado derecho:

```
if(root.right != null){
    inorderT(root.right);
}
```



Aquí se va hacia todas su izquierdas:



Después de que no tiene más, se guarda:

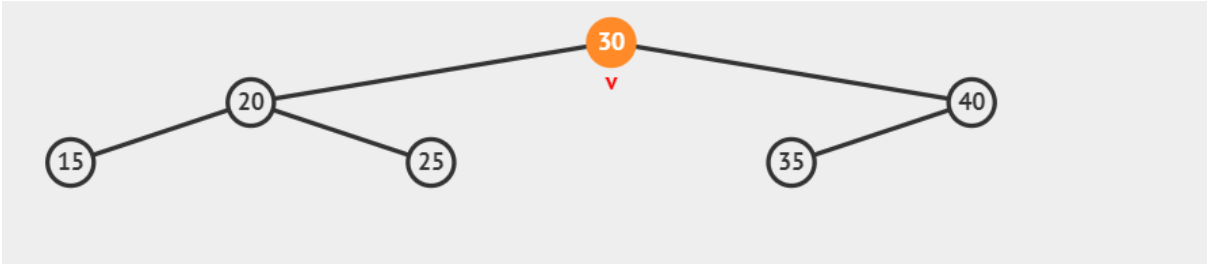
```
inorder.add(root.val);
```

{15,20,25,30,35}
Y ahora empieza su lado derecho:

```
if(root.right != null){
    inorderT(root.right);
}
```

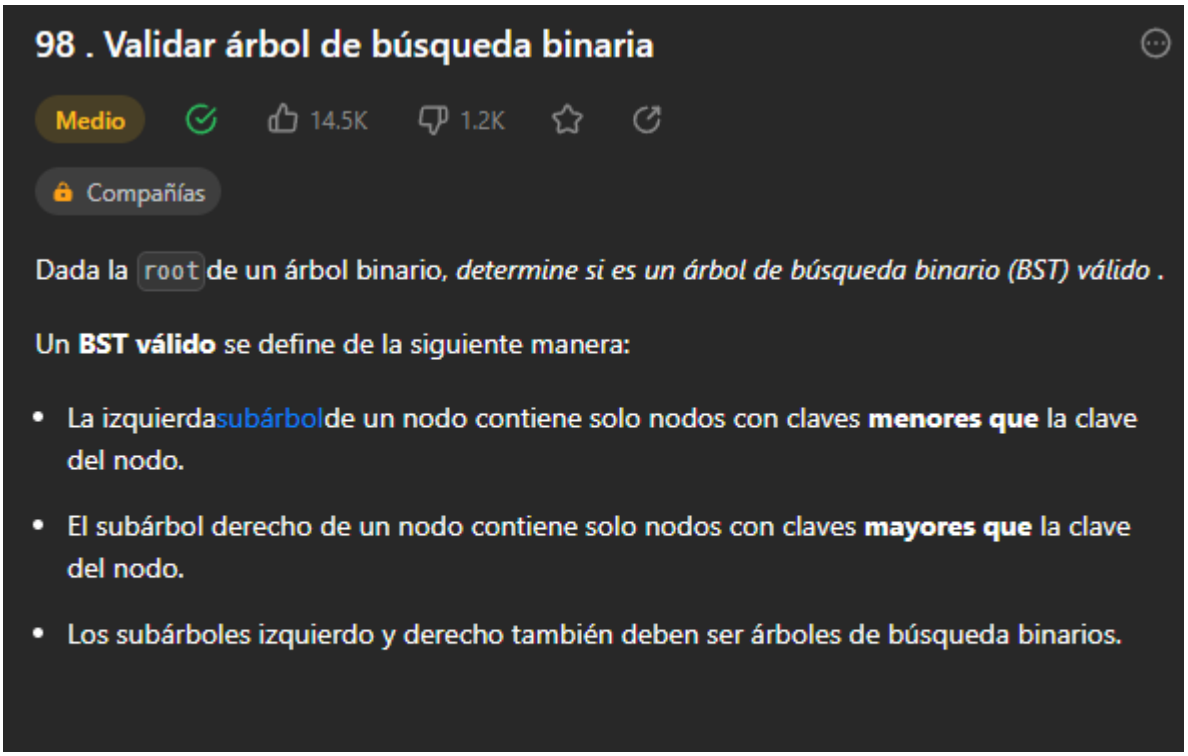
Como no tiene porque es una hoja, se devuelve a su raíz:

Aquí guarda su valor:
{15,20,25,30,35, 40}
y se devuelve para finalizar el método recursivo.

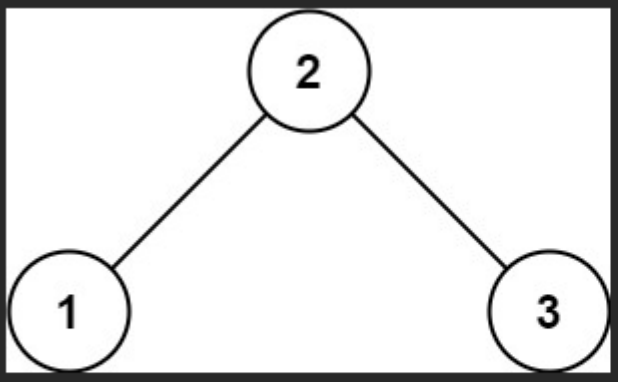


y aquí termina su método recursivo
y devuelve la lista: {15,20,25,30,35, 40}

3.

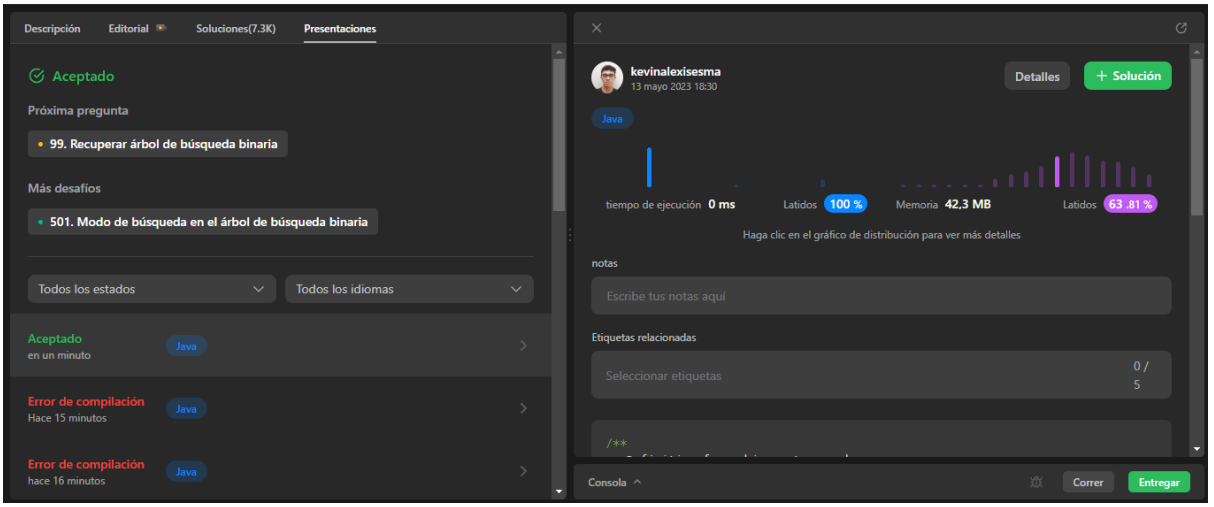


Ejemplo 1:



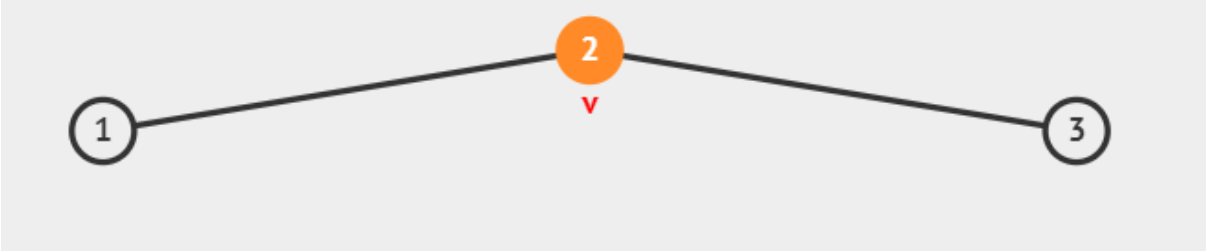
Entrada: raíz = [2,1,3]
Salida: verdadero

```
class Solution {
    boolean bst = true;
    TreeNode previous = null;
    public void isValid(TreeNode root){
        if(root==null){
            return;
        }
        isValid(root.left);
        if(previous!=null && previous.val>=root.val){
            bst = false;
        }
        previous = root;
        isValid(root.right);
    }
    public boolean isValidBST(TreeNode root) {
        isValid(root);
        return bst;
    }
}
```




```java
boolean bst = true;
TreeNode previous = null;
public void isValid(TreeNode root){
    if(root==null){
        return;
    }
    isValid(root.left);
    if(previous!=null && previous.val>=root.val){
        bst = false;
    }
    previous = root;
    isValid(root.right);
}
```

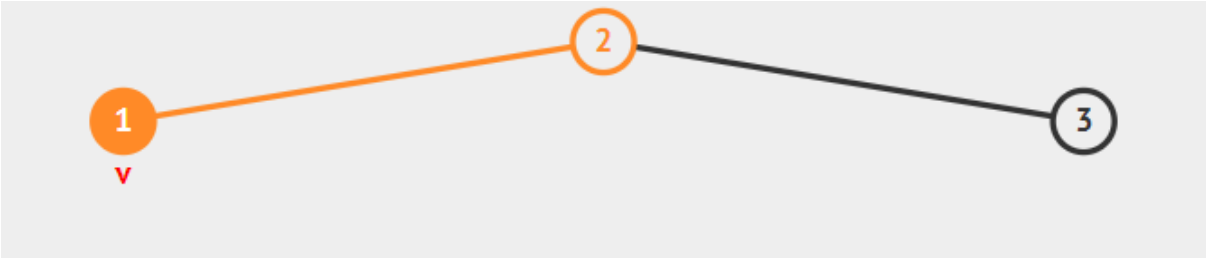
Para determinar si es un BST, se tiene una variable bst, que en el dado caso que un condicional no cumpla, se convertirá en falso.




```java
        if(root==null){
            return;
        }
```

Con este condicional se verifica que la raíz o el actual root no sea null, funciona como caso base para el método y para recursividad.

```java
isValid(root.left); Validamos el segundo.
```

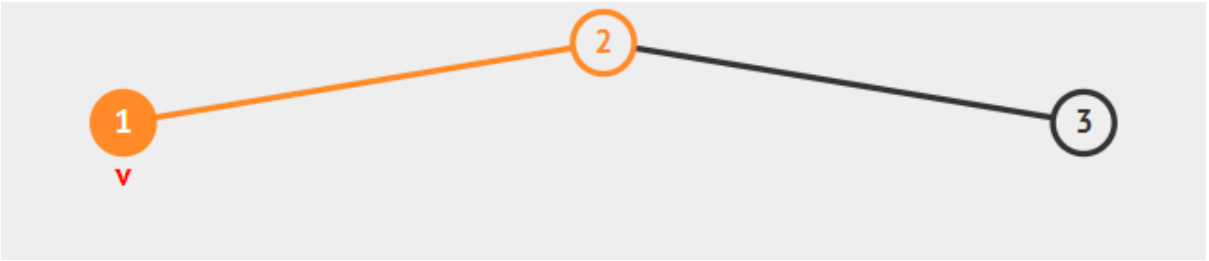


Al estar aquí también se valida si es diferente de null y se va hacia su izquierda:

```
    if(root==null){
            return;
        }
        isValid(root.left);
```

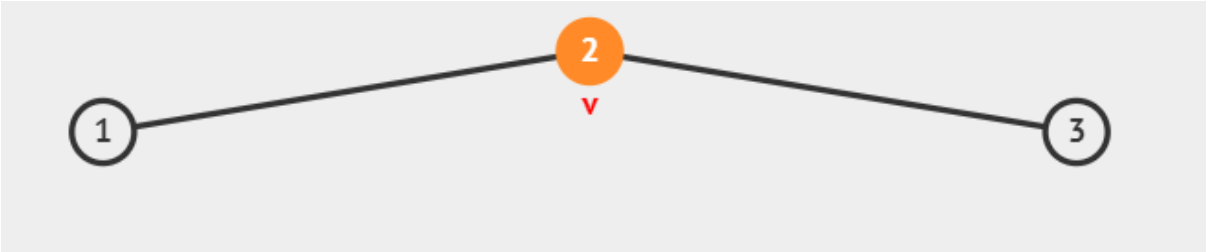
Al entrar a isValid(root.left), irá null porque el uno no tiene hijos:

```
    if(root==null){ Es decir, que será más recursivo después
de este punto, entonces solo se retornará.
            return;
        }
Una vez retornado, estará en el 1:
```



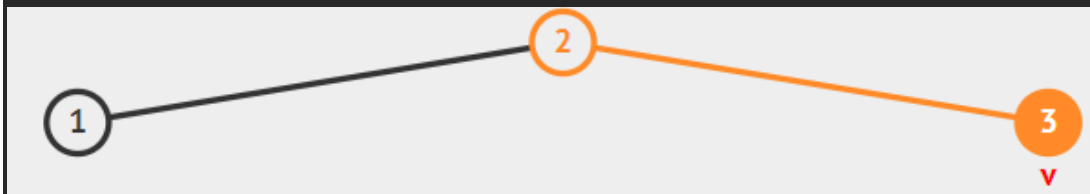

```
Y continuará con la siguiente línea de código:
if(previous!=null && previous.val>=root.val){ En este caso,
como es la primera vez que se llega a este punto, el previous
es = null, entonces no entra.
            bst = false;
        }
Se establece el 1 como previous.
        previous = root;
y se valida su derecha, pero como no tiene, simplemente
retorna.
        isValid(root.right);
```

Cuando ya termina el método, se retorna a su nodo padre, es decir:



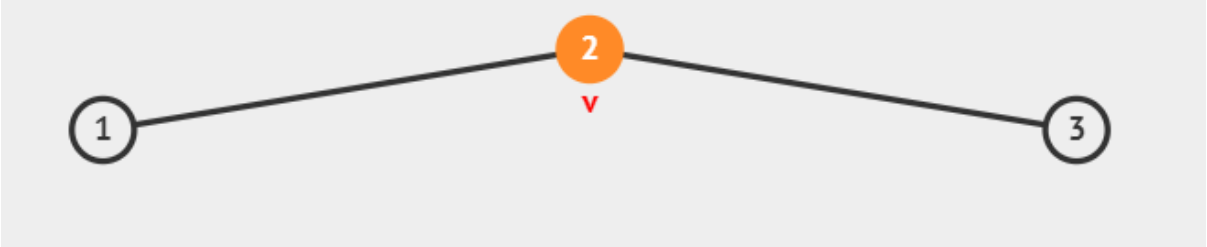
una vez aquí,

```
Llega a este condicional, entonces pregunta:
¿Previos es vacío?, no, porque tiene al 1.
¿Previos.val, que es igual a 1, es mayor igual que root.val,
que es 2?, es decir, ¿1 >= 2?

Lo cual es falso, entonces no entra al if.
if(previous!=null && previous.val>=root.val){
            bst = false;
        }
        previous = root; el previos pasa a ser 2.
        isValid(root.right); y se dirige hacia su derecha.}
```




```
Pregunta:
     if(root==null){ como es diferente de null, no entra.
            return;
        }
        isValid(root.left); Aquí valida por su lado
izquierdo, pero como es null, solo retorna.

     En el condicional, pregunta:
     Previos != null, es verdadero, tiene al 2.
     Previos.val >= root.val, es falso, porque:
     ¿2>=3?, no es mayor igual a 3, entonces no es verdadero,
esto quiere decir que bst sigue siendo verdadero.
        if(previous!=null && previous.val>=root.val){
            bst = false;
```

```
        }
        previous = root; previos pasa a ser el 3.
        isValid(root.right); aquí como no tiene derecha,
entonces retorna.
```



Estamos parados en la raíz, ya después del `isValid(root.right);` Entonces, solo queda ir al método donde comenzó todo:

```
public boolean isValidBST(TreeNode root) {
        isValid(root); Después de esto, bst sigue en true.
        return bst; devuelve true, indicando que el árbol si
es bst.
    }
```

Ejercicios del 4 al 6

1) Resolverlo en la plataforma

- ***Ejercicio 4 - Árboles de altura mínima***

Un árbol es un grafo no dirigido en el que dos vértices cualesquiera están conectados *exactamente* por un camino. En otras palabras, cualquier gráfico conexo sin ciclos simples es un árbol.

Dado un árbol de n nodos etiquetados de 0 a n - 1, y una matriz de n - 1 edges donde indica que hay un borde no dirigido entre los dos nodos y en el árbol, puede elegir cualquier nodo del árbol como raíz. Cuando selecciona un nodo como raíz, el árbol de resultados tiene altura . Entre todos los árboles enraizados posibles, aquellos con altura mínima (ie) se denominan **árboles de altura mínima** (MHT).edges[i] = [ai, bi]aibixhmin(h)

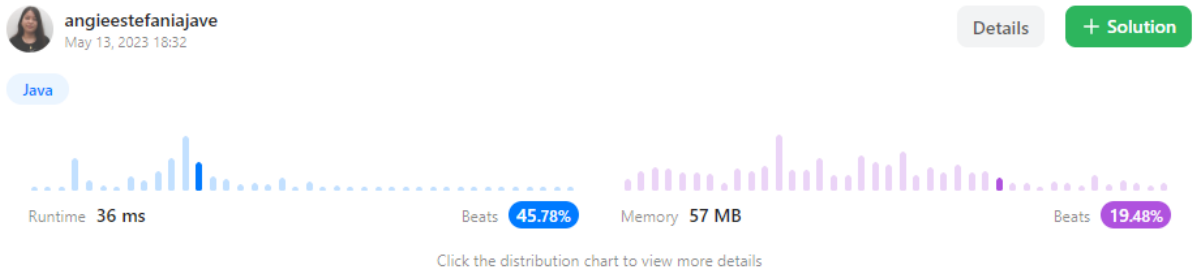
Devuelve *una lista de las etiquetas raíz* ***de todos los MHT*** . Puede devolver la respuesta en **cualquier orden** .

La **altura** de un árbol con raíces es el número de aristas en el camino descendente más largo entre la raíz y una hoja.

Solución

```java
class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges)
{
         if (n == 1) {
            List<Integer> list = new ArrayList<>();
            list.add(0);
            return list;
        }

        List<Set<Integer>> a = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            a.add(new HashSet<>());
        }
        for (int[] edge : edges) {
            a.get(edge[0]).add(edge[1]);
            a.get(edge[1]).add(edge[0]);
        }

        List<Integer> hojas = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (a.get(i).size() == 1) {
                hojas.add(i);
            }
        }
        while (n > 2) {
            n -= hojas.size();
            List<Integer> newHojas = new ArrayList<>();
            for (int hoja : hojas) {
                int vecino = a.get(hoja).iterator().next();
                a.get(vecino).remove(hoja);
                if (a.get(vecino).size() == 1) {
                    newHojas.add(vecino);
                }
            }
            hojas = newHojas;
        }
        return hojas;
    }
}
```

Evidencia



- ***Ejercicio 5 - Encuentre un nodo correspondiente de un árbol binario en un clon de ese árbol***

Dados dos árboles binarios original y cloned dada una referencia a un nodo target en el árbol original.

El cloned árbol es una **copia del** árbol original.

Devuelve *una referencia al mismo nodo* en el cloned árbol.

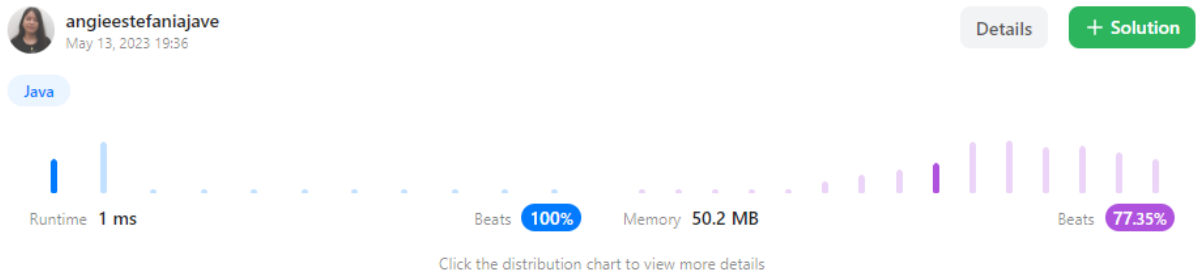
Tenga en cuenta que **no puede** cambiar ninguno de los dos árboles o el target nodo y la respuesta **debe ser** una referencia a un nodo en el cloned árbol.

Solución

```
/**

 * Definition for a binary tree node.

 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */


class Solution {
    public final TreeNode getTargetCopy(final TreeNode original,
final TreeNode cloned, final TreeNode target) {

        if (original == null || cloned == null || target == null)
{
            return null;
        }
        if (original == target) {
            return cloned;
        }
        TreeNode left = getTargetCopy(original.left, cloned.left,
target);
        if (left != null) {
            return left;
        }
```

```
        return getTargetCopy(original.right, cloned.right,
target);
    }
}
```

Evidencia



- ***Ejercicio 6 - Encuentra si la ruta existe en el gráfico***

Hay un gráfico **bidireccional** n con vértices, donde cada vértice está etiquetado de 0 a n - 1(**inclusive**). Los bordes del gráfico se representan como una matriz de enteros 2D edges, donde cada uno denota un borde bidireccional entre vértice y vértice . Cada par de vértices está conectado **como máximo por una** arista, y ningún vértice tiene una arista en sí mismo.edges[i] = [u_i, v_i] u_iv_i

Desea determinar si existe una **ruta válida** de vértice source a vértice destination.

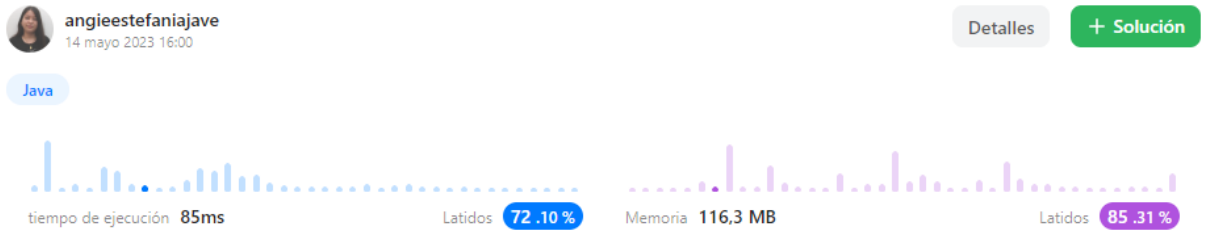
Dado edges y los enteros n, source y destination, devuelve n true *si hay una* ***ruta válida*** *de* source *a* destination*, o* false *de lo contrario .*

Solución

```
class Solution {

    public boolean validPath(int n, int[][] edges, int source, int
destination) {

        List<Integer>[] auxLista = new List[n];

         for (int i = 0; i < n; i++) {
             auxLista[i] = new ArrayList<>();
         }

         for (int[] edge : edges) {

             int u = edge[0], v = edge[1];
             auxLista[u].add(v);
             auxLista[v].add(u);
         }

         boolean[] visited = new boolean[n];

         visited[source] = true;
```

```
            Stack<Integer> pila = new Stack<>();

            pila.push(source);

            while (!pila.isEmpty()) {

                int u = pila.pop();
                if (u == destination) {
                    return true;
                }

                for (int v : auxLista[u]) {

                    if (!visited[v]) {

                        visited[v] = true;
                        pila.push(v);
                    }
                }
            }

            return false;
        }
    }
```

Evidencias



2) Escribir un programa que genere dos ficheros TXT, un fichero con datos de entrada y otro con datos de salida. Estos datos deben servir para probar la misma solución anterior en otra plataforma. La prueba debe realizarla (según lo explicado en clase) usando redirección de I/O por comandos < in.txt >out.txt

- ***Ejercicio 4***

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
```

```
public class TestSolution4 {

    public static void main(String[] args) throws IOException {
        // Configuración de las pruebas
        Random random = new Random();
        int numTests = 5;
        int minNodes = 3;
        int maxNodes = 9;
        int maxEdges = 15;

        // Crear las pruebas
        for (int i = 1; i <= numTests; i++) {
            // Generar datos aleatorios para la prueba
            int n = random.nextInt(maxNodes - minNodes + 1) + minNodes;
            int[][] edges = generateRandomTreeEdges(random, n);

            // Ejecutar la solución
            Solution4 solution = new Solution4();
            List<Integer> result = solution.findMinHeightTrees(n,
edges);

            // Verificar y corregir el caso de ciclo infinito
            if (result.size() == n) {
                // El grafo tiene un ciclo infinito, se debe corregir
                result = new ArrayList<>();
                result.add(0); // Agregar cualquier nodo como resultado
            }

            // Escribir los datos de entrada en un archivo
            String inputFilename = "test" + i + "_in.txt";
            FileWriter inputWriter = new FileWriter(new
File(inputFilename));
            inputWriter.write(n + "\n");
            for (int[] edge : edges) {
                inputWriter.write(edge[0] + " " + edge[1] + "\n");
            }
            inputWriter.close();

            // Escribir los datos de salida en un archivo
            String outputFilename = "test" + i + "_out.txt";
            FileWriter outputWriter = new FileWriter(new
File(outputFilename));
            for (int r : result) {
```

```
                outputWriter.write(r + " ");
            }
            outputWriter.close();
        }
    }

    private static int[][] generateRandomTreeEdges(Random random, int
n) {
        int[][] edges = new int[n - 1][2];
        Set<Integer> nodes = new HashSet<>();

        // Agregar todos los nodos al conjunto
        for (int i = 0; i < n; i++) {
            nodes.add(i);
        }

        // Generar los bordes del árbol
        int parent = random.nextInt(n);
        nodes.remove(parent);

        for (int i = 0; i < n - 1; i++) {
            int child = nodes.iterator().next();
            nodes.remove(child);
            edges[i][0] = parent;
            edges[i][1] = child;
            parent = child;
        }

        return edges;
    }
}
```

Ejercicio 5

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class TestSolution5 {

    public static void main(String[] args) throws IOException {
        // Configuración de las pruebas
```

```
        Random random = new Random();
        int numTests = 20;
        int minVal = 1;
        int maxVal = 50;

        // Crear las pruebas
        for (int i = 1; i <= numTests; i++) {
            // Generar datos aleatorios para la prueba
            Solution5 solution = new Solution5();
            Solution5.TreeNode originalRoot =
generateRandomTree(random, minVal, maxVal);
            Solution5.TreeNode clonedRoot = cloneTree(originalRoot);
            Solution5.TreeNode targetNode =
getRandomNode(originalRoot);

            // Ejecutar la solución
            Solution5.TreeNode result =
solution.getTargetCopy(originalRoot, clonedRoot, targetNode);

            // Escribir los datos de entrada en un archivo
            String inputFilename = "test" + i + "_in.txt";
            FileWriter inputWriter = new FileWriter(new
File(inputFilename));
            writeTree(inputWriter, originalRoot);
            inputWriter.write(targetNode.val + "\n");
            inputWriter.close();

            // Escribir los datos de salida en un archivo
            String outputFilename = "test" + i + "_out.txt";
            FileWriter outputWriter = new FileWriter(new
File(outputFilename));
            if (result != null) {
                outputWriter.write(result.val + "\n");
            } else {
                outputWriter.write("null\n");
            }
            outputWriter.close();
        }
    }

    private static Solution5.TreeNode generateRandomTree(Random random,
int minVal, int maxVal) {
        if (random.nextBoolean()) {
```

```
            return null;
        }
        int val = random.nextInt(maxVal - minVal + 1) + minVal;
        Solution5.TreeNode node = new Solution5().new TreeNode(val);
        node.left = generateRandomTree(random, minVal, maxVal);
        node.right = generateRandomTree(random, minVal, maxVal);
        return node;
    }

    private static Solution5.TreeNode cloneTree(Solution5.TreeNode
node) {
        if (node == null) {
            return null;
        }
        Solution5.TreeNode clone = new Solution5().new
TreeNode(node.val);
        clone.left = cloneTree(node.left);
        clone.right = cloneTree(node.right);
        return clone;
    }

    private static Solution5.TreeNode getRandomNode(Solution5.TreeNode
root) {
        if (root == null) {
            return null;
        }
        Random random = new Random();
        int targetVal = root.val;
        while (targetVal == root.val) {
            int minVal = 1;
            int maxVal = 50;
            targetVal = random.nextInt(maxVal - minVal + 1) + minVal;
        }
        return findNode(root, targetVal);
    }

    private static Solution5.TreeNode findNode(Solution5.TreeNode node,
int targetVal) {
        if (node == null || node.val == targetVal) {
            return node;
        }
        Solution5.TreeNode leftResult = findNode(node.left, targetVal);
        if (leftResult != null) {
```

```
            return leftResult;
        }
        return findNode(node.right, targetVal);
    }

    private static void writeTree(FileWriter writer, Solution5.TreeNode
node) throws IOException {
        if (node == null) {
            writer.write("null\n");
        } else {

            writer.write(node.val + "\n");
            writeTree(writer, node.left);
            writeTree(writer, node.right);
        }
    }
}
```

- ***Ejercicio 6***

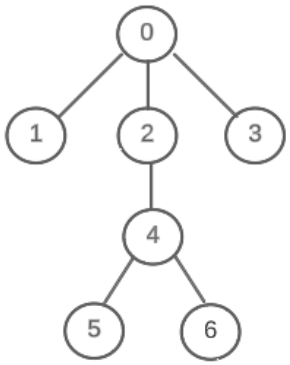
```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class TestSolution6 {

    public static void main(String[] args) throws IOException {
        // Configuración de las pruebas
        Random random = new Random();
        int numTests = 20;
        int minNodes = 3;
        int maxNodes = 100;
        int maxEdges = 500;

        // Crear las pruebas
        for (int i = 1; i <= numTests; i++) {
            // Generar datos aleatorios para la prueba
            int n = random.nextInt(maxNodes - minNodes + 1) + minNodes;
            int[][] edges = generateRandomEdges(random, n, maxEdges);
            int source = random.nextInt(n);
            int destination = random.nextInt(n);

            // Ejecutar la solución
```

```
            Solution6 solution = new Solution6();
            boolean result = solution.validPath(n, edges, source,
destination);

            // Escribir los datos de entrada en un archivo
            String inputFilename = "test" + i + "_in.txt";
            FileWriter inputWriter = new FileWriter(new
File(inputFilename));
            inputWriter.write(n + " " + edges.length + " " + source + "
" + destination + "\n");
            for (int[] edge : edges) {
                inputWriter.write(edge[0] + " " + edge[1] + "\n");
            }
            inputWriter.close();

            // Escribir los datos de salida en un archivo
            String outputFilename = "test" + i + "_out.txt";
            FileWriter outputWriter = new FileWriter(new
File(outputFilename));
            outputWriter.write(result ? "true\n" : "false\n");
            outputWriter.close();
        }
    }

    private static int[][] generateRandomEdges(Random random, int n,
int maxEdges) {
        int numEdges = random.nextInt(maxEdges + 1);
        int[][] edges = new int[numEdges][2];
        for (int i = 0; i < numEdges; i++) {
            edges[i][0] = random.nextInt(n);
            edges[i][1] = random.nextInt(n);
        }
        return edges;
    }
}
```

3) Explicar la solución con un ejemplo gráfico, con colores, paso a paso. Por ejemplo, si se requiere armar un árbol y luego recorrerlo, debe mostrarse un ejemplo completo de manera gráfica. Si se deben considerar varios escenarios o casos borde, deben incluirse en esta explicación.

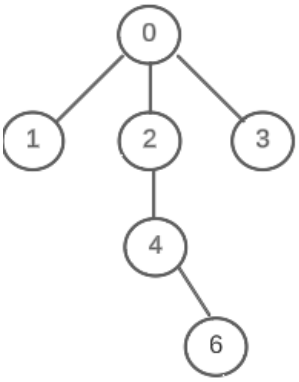
Ejercicio 4

1. Grafo inicial



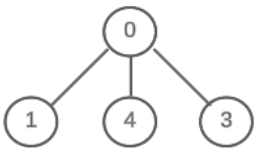
2. Identificación de las hojas:
   Hojas: [1, 2, 5]

3. Iteración 1:
   Eliminando hojas: [1, 2, 5]



4. Iteración 2:
   Hojas actuales: [1, 2, 6]

5. Iteración 3:



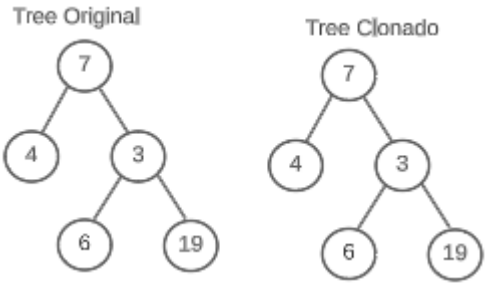
   Hojas actuales: [1, 3, 4]

6. Iteración 4:

Hojas actuales: [3]

7. Finalización:
   Nodos restantes: [0, 3]

Ejercicio 5



En este caso, queremos encontrar el nodo con el valor 6 en el árbol original y obtener su correspondiente nodo en el árbol clonado.

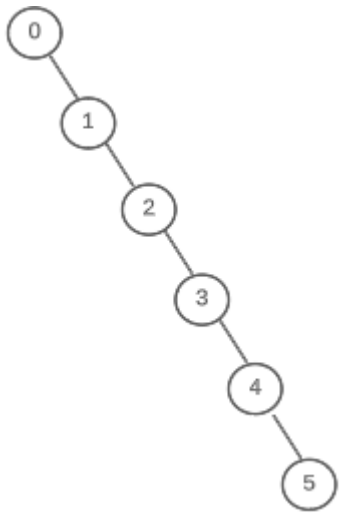
El proceso de búsqueda se realiza de la siguiente manera:

1. Comenzamos en la raíz del árbol original y del árbol clonado.
2. Comparamos el nodo actual en el árbol original con el nodo objetivo (valor 6). Si son iguales, hemos encontrado el nodo objetivo y devolvemos el nodo correspondiente en el árbol clonado.
3. Si no son iguales, nos movemos al subárbol izquierdo y repetimos el proceso desde el paso 2.
4. Si no se encuentra el nodo objetivo en el subárbol izquierdo, nos movemos al subárbol derecho y repetimos el proceso desde el paso 2.

En este caso, el nodo objetivo con el valor 6 se encuentra en el subárbol derecho del árbol original. Al seguir el proceso de búsqueda, llegamos al nodo correspondiente en el árbol clonado.

Ejercicio 6

n = 6
edges = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]]
source = 0
destination = 5



El objetivo es determinar si hay un camino válido desde el nodo fuente (0) hasta el nodo destino (5).

El proceso de búsqueda se realiza de la siguiente manera:

1. Creamos una lista de adyacencia llamada auxLista para representar las conexiones entre los nodos.
2. Recorremos la matriz edges y agregamos las conexiones a auxLista. En este caso, obtenemos:

   auxLista[0] = [1]
   auxLista[1] = [0, 2]
   auxLista[2] = [1, 3]
   auxLista[3] = [2, 4]
   auxLista[4] = [3, 5]
   auxLista[5] = [4]

3. Inicializamos un arreglo de booleanos llamado ‘visited’ para rastrear los nodos visitados. Marcamos el nodo fuente (0) como visitado.

4. Creamos una pila ‘pila’ y agregamos el nodo fuente (0) a la pila.

5. Mientras la pila no esté vacía, realizamos lo siguiente:

   - Tomamos el elemento superior de la pila (último nodo visitado) y lo sacamos de la pila. En este caso, obtenemos el nodo 0.
   - Comparamos si el nodo actual es igual al nodo destino. Si es así, encontramos un camino válido y devolvemos true.
   - Recorremos los nodos adyacentes al nodo actual (0) y verificamos si no han sido visitados. En este caso, tenemos el nodo 1 como adyacente.
   - Marcamos el nodo adyacente (1) como visitado y lo agregamos a la pila.

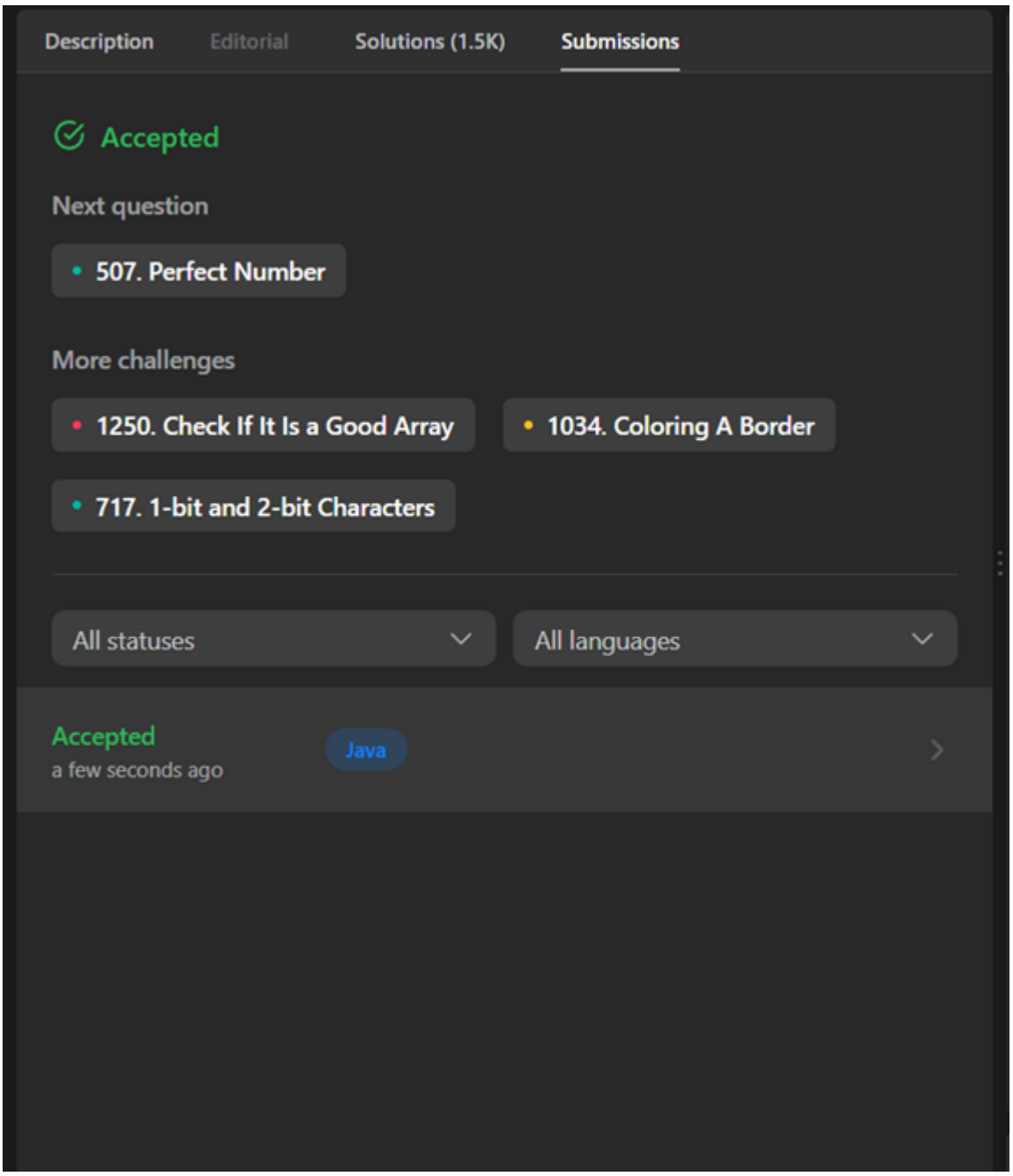
6. Si hemos recorrido todos los nodos adyacentes y no encontramos el nodo destino, retornamos false.

En este ejemplo, el algoritmo seguirá los siguientes pasos:

1. La pila contiene el nodo 0.
2. El nodo 0 se saca de la pila y se verifica si es igual al nodo destino. Como no son iguales, continuamos.
3. El nodo adyacente 1 se marca como visitado y se agrega a la pila.
4. La pila contiene los nodos 1, 0.
5. El nodo 1 se saca de la pila y se verifica si es igual al nodo destino. Como no son iguales, continuamos.
6. Los nodos adyacentes 0 y 2 ya han sido visitados, por lo que no se agregan a la pila.
7. La pila contiene el nodo 0.
8. El nodo 0 se saca de la pila y se verifica si es igual al nodo destino. Como no son iguales, continuamos.
9. Los nodos adyacentes 1 y 2 ya han sido visitados, por lo que no se agregan a la pila.

4) Publicar todo lo anterior en github en un repositorio, usando el README para la documentación. Subir a UVIRTUAL un ZIP completo del repositorio.
Link de gitHub:
https://github.com/kevinalexisesma/EstructurasDeDatosAvanzadas.git
https://github.com/AngieJaimes25/EstructuraDeDatosAvanzados/tree/main

7. Relative Ranks


Description
Editorial
Solutions (1.5K)
Submissions
Accepted
Next question
507. Perfect Number
More challenges
1250. Check If It Is a Good Array
1034. Coloring A Border
717. 1-bit and 2-bit Characters
All statuses
All languages
Accepted
a few seconds ago
Java

506. Relative Ranks

Easy 1K 53

Companies

You are given an integer array `score` of size `n`, where `score[i]` is the score of the `i^th` athlete in a competition. All the scores are guaranteed to be **unique**.

The athletes are **placed** based on their scores, where the `1^st` place athlete has the highest score, the `2^nd` place athlete has the `2^nd` highest score, and so on. The placement of each athlete determines their rank:

- The `1^st` place athlete's rank is `"Gold Medal"`.
- The `2^nd` place athlete's rank is `"Silver Medal"`.
- The `3^rd` place athlete's rank is `"Bronze Medal"`.
- For the `4^th` place to the `n^th` place athlete, their rank is their placement number (i.e., the `x^th` place athlete's rank is `"x"`).

Return an array `answer` of size `n` where `answer[i]` is the **rank** of the `i^th` athlete.

Example 1:

```
Input: score = [5,4,3,2,1]
Output: ["Gold Medal","Silver Medal","Bronze Medal","4","5"]
Explanation: The placements are [1st, 2nd, 3rd, 4th, 5th].
```

Example 2:

```
Input: score = [10,3,8,9,4]
Output: ["Gold Medal","5","Bronze Medal","Silver Medal","4"]
Explanation: The placements are [1st, 5th, 3rd, 2nd, 4th].
```

```java
class Solution {
    public String[] findRelativeRanks(int[] score) {
        int[] nums=score;
        String[] result = new String[nums.length];
        int max = 0;
        for (int i : nums) {
            if (i > max) max = i;
        }
        int[] hash = new int[max + 1];
        for (int i = 0; i < nums.length; i++) {
            hash[nums[i]] = i + 1;
        }
        int place = 1;
        for (int i = hash.length - 1; i >= 0; i--) {
            if (hash[i] != 0) {
                if (place == 1) {
                    result[hash[i] - 1] = "Gold Medal";
                } else if (place == 2) {
                    result[hash[i] - 1] = "Silver Medal";
                } else if (place == 3) {
                    result[hash[i] - 1] = "Bronze Medal";
                } else {
                    result[hash[i] - 1] = String.valueOf(place);
                }
                place++;
            }
        }
        return result;
    }
}
```

8) Heap Operations

Kass07	681C - Heap Operations	GNU C++17 (64)	Accepted	140 ms	32100 KB

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    ios_base::sync_with_stdio(false);
    string str;
    int n,val;
    cin>>n;
    vector< pair<string,int> >v;
    multiset<int>ans;
    while(n--)
    {
        cin>>str;
        if(str=="insert") {cin>>val;ans.insert(val);v.emplace_back(make_pair(str,val));}
        else if(str=="removeMin")
        {
            auto it=ans.begin();
            if(it!=ans.end()){
            ans.erase(it);
            }
            else {
            v.emplace_back(make_pair("insert",0));
            }
            v.emplace_back(make_pair(str,0));
        }
        else if(str=="getMin")
        {
            cin>>val;
            auto it=ans.begin();
            if(val==(*it)&&it!=ans.end())
            {
                v.emplace_back(make_pair(str,val));
            }
            else
            {
                while(!ans.empty())
                {
                it=ans.begin();
                if(val==(*it)||(val<(*it))) break;
                v.emplace_back(make_pair("removeMin",0));
                ans.erase(it);
                }
```

```
            if(ans.empty())
            {
                v.emplace_back(make_pair("insert",val));
                ans.insert(val);
            }
            else if((val<(*it)))
            {
                v.emplace_back(make_pair("insert",val));
                ans.insert(val);
            }
            v.emplace_back(make_pair(str,val));
        }
    }
}
int sz=(int)v.size();
cout<<sz<<endl;
for(int i=0;i<sz;i++)
{
    if(v[i].first=="removeMin") cout<<"removeMin"<<'\n';
    else cout<<v[i].first<<" "<<v[i].second<<'\n';
}
return 0;
}
```

9) Count Distinct Integers

B - Count Distinct Integers Editorial

Time Limit: 2 sec / Memory Limit: 1024 MB

Score : 200 points

Problem Statement

In a sequence of N positive integers $a = (a_1, a_2, \ldots, a_N)$, how many different integers are there?

Constraints

- $1 \leq N \leq 1000$
- $1 \leq a_i \leq 10^9$ $(1 \leq i \leq N)$
- All values in input are integers.

Input

Input is given from Standard Input in the following format:

N
a_1 $\ldots$ a_N

Output

Print the answer.

Sample Input 1 Copy

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

Sample Output 1 Copy

```
3
```

There are three different integers: $1, 2, 4$.

Sample Input 2 Copy

```
1
1
```

Task: - Language: - Status: - User: Reset Search

Submission Time	Task	User	Language	Score	Code Size	Status	Exec Time	Memory	
2023-05-16 17:54:48	B - Count Distinct Integers	Kennedy07	Java (OpenJDK 1.8.0)	200	463 Byte	AC	131 ms	30456 KB	Detail

```
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        Set<Integer> color = new HashSet<>();
        int[] numArray = new int[N];
        for(int i = 0; i < N; i++) {
                numArray[i] = sc.nextInt();
                color.add(numArray[i]);
        }
        System.out.println(color.size());
    }
}
```

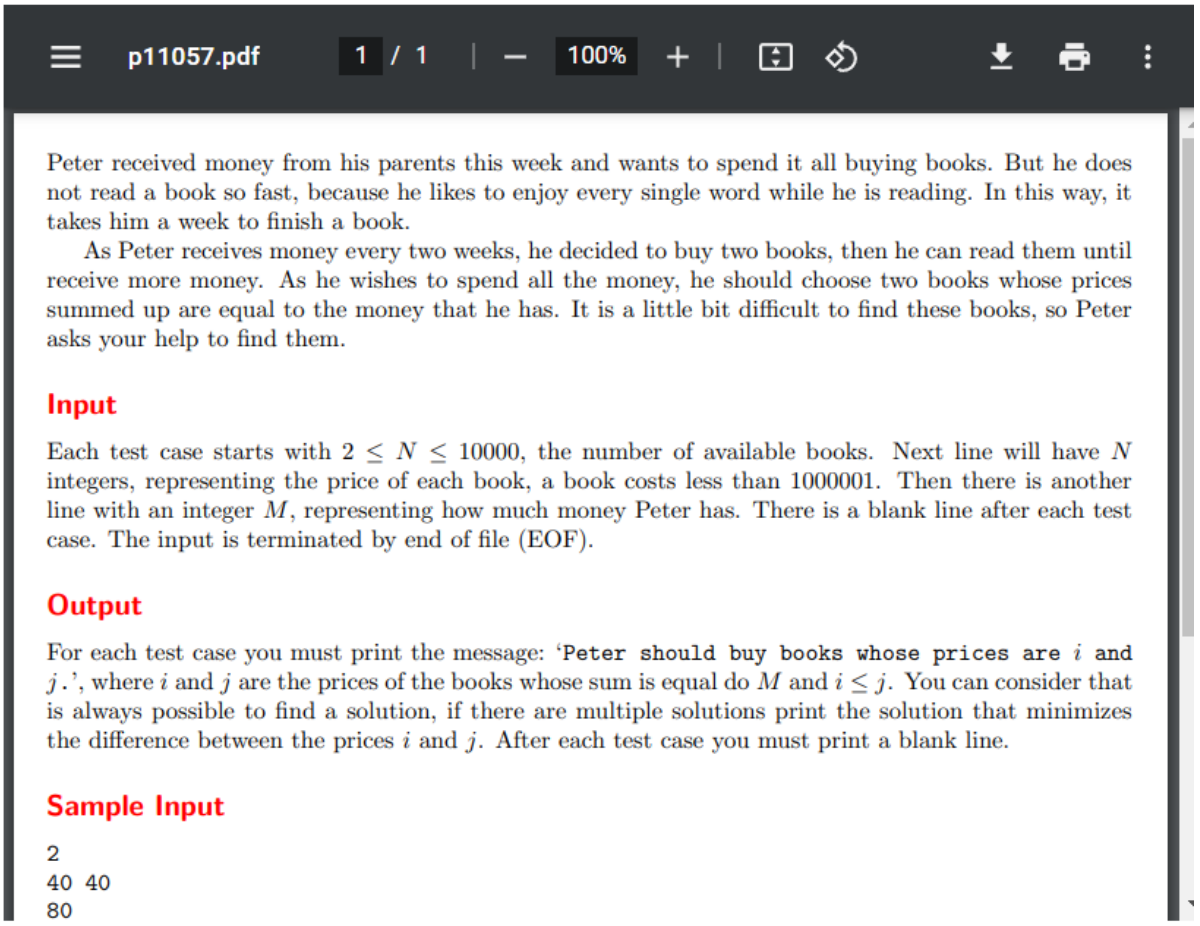
Submission Info

Submission Time	2023-05-16 17:54:48
Task	B - Count Distinct Integers
User	Kennedy07
Language	Java (OpenJDK 1.8.0)
Score	200
Code Size	463 Byte
Status	AC
Exec Time	131 ms

10) Exact Sum

11057 - Exact Sum

Time limit: 3.000 seconds



Peter received money from his parents this week and wants to spend it all buying books. But he does not read a book so fast, because he likes to enjoy every single word while he is reading. In this way, it takes him a week to finish a book.

As Peter receives money every two weeks, he decided to buy two books, then he can read them until receive more money. As he wishes to spend all the money, he should choose two books whose prices summed up are equal to the money that he has. It is a little bit difficult to find these books, so Peter asks your help to find them.

Input

Each test case starts with $2 \leq N \leq 10000$, the number of available books. Next line will have N integers, representing the price of each book, a book costs less than 1000001. Then there is another line with an integer M, representing how much money Peter has. There is a blank line after each test case. The input is terminated by end of file (EOF).

Output

For each test case you must print the message: '`Peter should buy books whose prices are` i `and` j`.`', where i and j are the prices of the books whose sum is equal do M and $i \leq j$. You can consider that is always possible to find a solution, if there are multiple solutions print the solution that minimizes the difference between the prices i and j. After each test case you must print a blank line.

Sample Input

```
2
40 40
80
```

My Submissions

#	Problem	Verdict	Language	Run Time	Submission Date
28471269	11057 Exact Sum	Accepted	JAVA	0.250	2023-05-17 00:21:46

```java
public class ExactSum {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(System.out);
        while (br.ready()) {
            int n = Integer.parseInt(br.readLine());
            HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
            StringTokenizer st = new StringTokenizer(br.readLine());
            for (int i = 0; i < n; i++) {
                int x = Integer.parseInt(st.nextToken());
                if (map.containsKey(x)) {
                    int val = map.get(x);
                    map.put(x, val + 1);
                } else {
                    map.put(x, 1);
                }
            }
            int m = Integer.parseInt(br.readLine());
            int i = 0, j = 0;
            int dif = Integer.MAX_VALUE;
            for (Entry<Integer, Integer> e : map.entrySet()) {
                int key = e.getKey();
                int req = m - key;
                if (req == key) {
                    if (e.getValue() > 1) {
                        dif = 0;
                        i = key;
                        j = key;
                    }
                } else {
                    if (map.containsKey(req)) {
                        int curDif = Math.abs(req - key);
                        if (curDif < dif) {
                            dif = curDif;
                            i = Math.min(key, req);
                            j = Math.max(key, req);
                        }
                    }
                }
            }
            out.append("Peter should buy books whose prices are " + i + " and " + j + ".\n\n");
            br.readLine();
        }
        out.flush();
    }
}
```