3b. Implement the above algorithm in Java

```java
public class Main {
    public static void main(String[] args) {
        // Main initialises Jugs, which then runs everything else.
        Jugs j = new Jugs();
    }
}


import java.util.*;

public class Jugs {
    //These store the maximum capacity of water that can fit in each jug.
    public int a;
    public int b;
    public int c;

    //I used an Arraylist to store my states, as it can be dynamically changed, as more states are created.
    //Each state is stored in a 'node' object - which stores the amount of water in each jug, as well as the index of that node's parent. This allows each node to be traced back through the tree.
    ArrayList<node> nodes = new ArrayList<node>();
    //root is the start node, and the only one hardcoded in.
    node root = new node(0,0,0,-1);
    //testNode is used as a temporary node that is changed regularly, but also used globally.
    node testNode;

    //The constructor for the Jugs class. It coordinates the algorithm and makes everything is done in the correct order.
    public Jugs(){
        getJugCapacities();
        //This makes sure the arrayList nodes has at least one node to begin with, and that the first node is root.
        nodes.add(root);
        createFirstChildren();
        createTreeArray();
        printArray();
    }

    //This receives the jug capacities from the user for jugs A, B and C, sorting it so A has the largest capacity and C has the smallest.
    public void getJugCapacities(){
        //Using Java's Scanner utility it takes the next line of the console as input.
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the capacity of the first jug:");
        int x = scan.nextInt();
        System.out.println("Enter the capacity of the second jug:");
        int y = scan.nextInt();
        System.out.println("Enter the capacity of the third jug:");
        int z = scan.nextInt();
        //validation to make sure that jug a holds the largest capacity, followed by jug b, then jug c with the smallest.
        if(x <= y && x <= z){
            c = x;
            if(y<=z){
                b = y;
                a = z;
            }else{
                a = y;
                b = z;
            }
        }else if(y <= x && y <= z){
            c = y;
            if(x<=z){
                b = x;
                a = z;
            }else{
                a = x;
                b = z;
            }
        }else{
            c = z;
            if(x<=y){
                b = x;
                a = y;
            }else{
                a = x;
                b = y;
            }
        }
        System.out.println("A: " + a + ", B: " + b + ", C: " + c);
    }

    //This function initialises the start state and makes its children, making sure
    that there are nodes in the main arrayList, so that the for loop doesn't finish
    prematurely (as it depends on the size of the arrayList)
    public void createFirstChildren(){
        testNode = root;
        createChildren(0);
    }

    //This loops through the arrayList of all nodes and calls createChildren, which
    checks all possible actions. If the created nodes from that are distinct from
    others in the ArrayList, they are appended to the end of it.
    //This makes the nodes arrayList longer, and means the for loop continues going.
    This means that the for loop only finishes when it has checked every single node
    for the possibility of creating more. This means that it finishes after every
    possibility has been checked.
    public void createTreeArray(){
        for(int x=1;x<nodes.size();x++){
            testNode = nodes.get(x);
            createChildren(x);
        }
    }

    //createChildren runs through all the possible actions that can be done on a
    particular node. First it checks whether you can fill any of the Jugs, then
    empty any of them, then pour any of them into any of the others.
    //it passes along the parent index in the form of the int x parameter. This
    allows for traceability later on, as you can backtrack from any node to it's
    parent, and then continue upwards until the parent integer is -1, which
    indicates you've reached the start state.
    public void createChildren(int x){
        fillA(x);
        fillB(x);
        fillC(x);
        emptyA(x);
        emptyB(x);
        emptyC(x);
        pourAtoB(x);
        pourAtoC(x);
        pourBtoA(x);
        pourBtoC(x);
        pourCtoA(x);
        pourCtoB(x);
    }

    //All the functions called in createChildren call searchArray, with the
    parameters for a new node. searchArray searches the array and if the numbers
    will make a unique node, the node is created and appended to the nodes arrayList.

    //fillA, fillB and fillC are all very similar, checking whether a jug is full or
    not, and if it is not full, it submits the new numbers for a new node to the
    searchArray function, where the selected jug now has maximum capacity. It also
    sends the parent index as that will be needed to create a new node.
    public void fillA(int parent){
        if(testNode.a != a){
            searchArray(a,testNode.b,testNode.c,parent);
        }
    }
    public void fillB(int parent){
        if(testNode.b != b){
            searchArray(testNode.a,b,testNode.c,parent);
        }
    }
    public void fillC(int parent){
        //if jug c isn't full, this operation can proceed.
        if(testNode.c != c){
            searchArray(testNode.a,testNode.b,c,parent);
        }
    }
    //emptyA, emptyB and emptyC are all similar as well. They check whether a jug is already empty and if it is not, then it submits the numbers, where the selected jug now is 0.
    public void emptyA(int parent){
        if(testNode.a != 0){
            searchArray(0,testNode.b,testNode.c,parent);
        }
    }
    public void emptyB(int parent){
        if(testNode.b != 0){
            searchArray(testNode.a,0,testNode.c,parent);
        }
    }
    public void emptyC(int parent){
        if(testNode.c != 0){
            searchArray(testNode.a,testNode.b,0,parent);
        }
    }
    //all the pour functions work in a very similar way. They check that there is water to pour from the initial jug, and that there is at least some space in the second jug.
    //Then, if the total liquid is less than the full capacity of the destination jug, then the numbers are submitted where the initial jug is 0 and the destination jug is the sum of both numbers.
    //Otherwise, if the total liquid is more than the full capacity of the destination jug, then the numbers are submitted where the destination jug is at its full capacity and the initial jug holds the remainder.
    public void pourAtoB(int parent){
        //Checks that the initial jug isn't empty and the destination jug isn't full
        if(testNode.a != 0 && testNode.b != b) {
            if(testNode.a+testNode.b>=b){
                searchArray(testNode.a-(b-testNode.b),b,testNode.c,parent);
            }
            if(testNode.a+testNode.b<b){
                searchArray(0,testNode.a+testNode.b,testNode.c,parent);
            }
        }
    }
    public void pourAtoC(int parent){
        //Checks that the initial jug isn't empty and the destination jug isn't full
        if(testNode.a != 0 && testNode.c != c) {
            if(testNode.a+testNode.c>=c){
                searchArray(testNode.a-(c-testNode.c),testNode.b,c,parent);
            }
            if(testNode.a+testNode.c<c){
                searchArray(0,testNode.b,testNode.a+testNode.c,parent);
            }
        }
    }
    public void pourBtoA(int parent){
        //Checks that the initial jug isn't empty and the destination jug isn't full
        if(testNode.b != 0 && testNode.a != a) {
            if(testNode.b+testNode.a>=a){
                searchArray(a,testNode.b-(a-testNode.a),testNode.c,parent);
            }
            if(testNode.b+testNode.a<a){
                searchArray(testNode.b+testNode.a,0,testNode.c,parent);
            }
        }
    }
    public void pourBtoC(int parent){
        //Checks that the initial jug isn't empty and the destination jug isn't full
        if(testNode.b != 0 && testNode.c != c) {
            if(testNode.b+testNode.c>=c){
                searchArray(testNode.a,testNode.b-(c-testNode.c),c,parent);
            }
            if(testNode.b+testNode.c<c){
                searchArray(testNode.a,0,testNode.b+testNode.c,parent);
            }
        }
    }
    public void pourCtoA(int parent){
        //Checks that the initial jug isn't empty and the destination jug isn't full
        if(testNode.c!= 0 && testNode.a != a) {
            if(testNode.c+testNode.a>=a){
                searchArray(a,testNode.b,testNode.c-(a-testNode.a),parent);
            }
            if(testNode.c+testNode.a<a){
                searchArray(testNode.c+testNode.a,testNode.b,0,parent);
            }
        }
    }
    public void pourCtoB(int parent){
        //Checks that the initial jug isn't empty and the destination jug isn't full
        if(testNode.c!= 0 && testNode.b != b) {
            if(testNode.c+testNode.b>=b){
                searchArray(testNode.a,b,testNode.c-(b-testNode.b),parent);
            }
            if(testNode.c+testNode.b<b){
                searchArray(testNode.a,testNode.c+testNode.b,0,parent);
            }
        }
    }

    //searchArray starts by looping through the array and checking whether the combination of the numbers that were 'submitted' to it are already in the array. If they are not, then it creates a new node and adds it to the nodes arrayList.
    public void searchArray(int testA, int testB, int testC, int parent) {
        //I use included as the boolean that stores whether or not the submitted values already exist in the array. False means that the node is not included in the array, where true means that it already exists in the array.
        boolean included = false;
        for(int x=0;x<nodes.size();x++){
            if(testA == nodes.get(x).a && testB == nodes.get(x).b && testC == nodes.get(x).c){
                included = true;
            }
        }
        //If the numbers aren't already in the array, this if statement passes and a new node is added to the ArrayList.
        if(!included){
            nodes.add(new node(testA,testB,testC,parent));
        }
    }

    //printArray prints the List out. It iterates through the nodes ArrayList and prints each state out, using printNode to print each node in a uniform way. It uses the parent variable to organise the array.
    //When the parent of the selected node is different than the parent of the previous node, it adds a line break and prints out the parent node of the following nodes, so there is some traceability.
    public void printArray(){
        printNode(0);
        for(int x = 1;x<nodes.size();x++){
            if(nodes.get(x).parent != nodes.get(x-1).parent){
                System.out.print("\nNodes from state: " + "("+nodes.get(nodes.get(x).parent).a + ", "+nodes.get(nodes.get(x).parent).b + ", "+nodes.get(nodes.get(x).parent).c + ")\n");
            }
            printNode(x);
        }
        System.out.println("There are " + nodes.size() + " possible states.");
    }

    //printNode takes an integer representing the index of the nodes arrayList and prints out the state in a uniform way.
    public void printNode(int x){
        System.out.println("("+nodes.get(x).a + ", "+nodes.get(x).b + ", "+nodes.get(x).c + ")");
    }
}

//Node is an object that stores our data, almost exactly like an integer array.
//Node stores the amount of water in jug a, b and c in the variables a, b and c. The index of a node's parent is stored in parent.
public class node {
    public int a;
    public int b;
    public int c;
    public int parent;

    public node(int a, int b, int c, int parent){
        this.a = a;
        this.b = b;
        this.c = c;
        this.parent = parent;
    }
}
```