```
public class App {
    public static void main(String[] args) throws Exception {
        SearchAlgorithm search = new SearchAlgorithm();
        System.out.println("\nTHE TABLE OF BFS PROCESS:");
        for(int i=0; i<170; i++) System.out.print("-");
            System.out.println();
        search.BFS(4, 3, 2);
    }
}
import java.util.Set;
import java.util.HashSet;


public class States {
    int jug1,jug2;
    States parent; // Create a parent to store the parent state
    Set<States> adjacent = new HashSet<States>(); // Create a set to store the adjacent states
of the (jug1,jug2) state
```

```
    States(){}
    States (int jug1, int jug2, States parent) {
        this.jug1 = jug1;
        this.jug2 = jug2;
        this.parent = parent;
    }
    States (States state) {
        this.jug1 = state.jug1;
        this.jug2 = state.jug2;
        this.parent = state.parent;
    }

    @Override // Display the state
    public String toString() {
        return "(" + jug1 + "," + jug2 + ")";
    }
    @Override // Compare the state
    public boolean equals(Object obj) {
        if (obj==null) return false;
        if (obj == this) return true;
        States s = (States) obj;
        return jug1 == s.jug1 && jug2 == s.jug2;
    }
    @Override // Hash the state to store in the set to support checking if the state is in the
set
    public int hashCode() {
        return jug1 * 10 + jug2;
    }
```

```
    // Verify if the state is goal state
    public boolean isGoal(int target) {
        return jug1 == target || jug2 == target;
    }
    // Verify if the state is valid
    public boolean isValid(int maxJug1, int maxJug2) {
        return jug1 >= 0 && jug2 >= 0 && jug1 <= maxJug1 && jug2 <= maxJug2;
    }
```

```
    // Add the adjacent states to the adjacent set
    public void Action(int maxJug1, int maxJug2, Set<States> adjacent, States parent) {
```

```
        adjacent.add(new States(maxJug1, jug2, parent));// Fill jug1
        adjacent.add(new States(jug1, maxJug2, parent));// Fill jug2
        adjacent.add(new States(0, jug2, parent));// Empty jug1
        adjacent.add(new States(jug1, 0, parent));// Empty jug2
        int temp = Math.min(jug1, maxJug2 - jug2); // verify the amount of water that can be
poured
        adjacent.add(new States(jug1 - temp, jug2 + temp,parent));// Pour jug1 to jug2
        temp = Math.min(jug2, maxJug1 - jug1); // verify the amount of water that can be
poured
        adjacent.add(new States(jug1 + temp, jug2 - temp,parent));// Pour jug2 to jug1
```

```
    }
}
```

```
import java.util.*;

public class SearchAlgorithm {
    public void BFS(int maxJug1, int maxJug2, int target)
    {
        Queue<States> open = new LinkedList<>(); // Create a queue to store the states is
going to be visited
        Queue<States> closed = new LinkedList<>(); // Create a queue to store the visited
states
        Set<States> adjacent = new LinkedHashSet<>(); // Create a set to store the adjacent
states of the current state
        int step = 1; // Create a variable to store the step number
        Object[] title = {"STEP", "CURRENT ", "ADJACENT STATES", "OPENED", "CLOSED"}; //
Create a string array to store the title of the table
```

```
        // Create the initial state
        States initial = new States(0, 0, null);
        open.add(initial);
        while (!open.isEmpty()) {
            // Get the current state from the open list
            States current= open.poll();

            // verify the state isValid and not in the closed list
            if (current.isValid(maxJug1, maxJug2) && !closed.contains(current)) {
                // Add the adjacent states to adjacent set
                current.Action(maxJug1, maxJug2, adjacent, current);
                // Add the adjacent states to the open list
                for (States temp : adjacent)
                    if (temp.isValid(maxJug1, maxJug2) && !closed.contains(temp)
&& !temp.equals(current))
                        open.add(temp);
                closed.add(current); // Add the current state to the closed list
            }
            // if the state is not valid or in the closed list, continue to the next state
            else {
                for(int i=0; i<170; i++) System.out.print("-");
                System.out.printf("\n|\tVIOLATE STATES: " + current);
                for(int i=0; i<140; i++) System.out.print(" "); System.out.printf("|\n");
                continue;
            }
```

```
            // verify if the state is goal state
            if (current.isGoal(target)) {
                for(int i=0; i<170; i++) System.out.print("-");
                System.out.println("\nGOAL STATE FOUND " + current);
                Stack<States> path = new Stack<>();
                // Add the path to the stack
```

```
                while (!current.equals(initial)) {
                    path.push(new States(current));
                    current = current.parent;
                    // Get the parent of the current state to add to the path from the goal
state to the initial state
                }
                path.push(initial);
                System.out.print("PATH: ");
                while (!path.isEmpty()) {
                    System.out.print(path.pop());
                    if (!path.isEmpty()) System.out.print(" -> ");
                }
                return;
            }
```

```
            // Display the BFS process
            if(step==1) System.out.printf("| %-5s| %-10s | %-40s | %-30s | %-70s |\n", title);
            for(int i=0; i<170; i++) System.out.print("-");
            System.out.printf("\n| %-5s| %-10s | %-40s | %-30s | %-70s
|\n",step++,current,adjacent,open,closed);
            adjacent.clear();

        }
        // If no solution found
        for(int i=0; i<170; i++) System.out.print("-");
        System.out.println("\nNO STATES FOUND");
    }

}
```