W9 - Data Structure & Algorithms

Problem 1: Maximum Number of Non-Overlapping Tasks

Overview: Given N tasks, each with a start and end time, determine the maximum number of tasks a person can complete without any time overlap. This problem is a classic example of the activity selection problem, where the goal is to select the maximum number of activities that don't overlap in time.

Algorithm Steps:

1. **Sort Tasks**:
    - First, sort the tasks based on their end times. This sorting is crucial as it allows selecting the maximum number of non-overlapping tasks.
2. **Select Tasks**:
    - Start with the first task in the sorted list as it finishes the earliest.
    - For each subsequent task, if its start time is greater than or equal to the end time of the previously selected task, select it.
3. **Implementation Details**:
    - Define a class Task with start and end as attributes.
    - Implement a comparator to sort the tasks based on end times.
4. **Code**

```
import java.util.Arrays;

public class TaskScheduler {
   static class Task {
       int start;
       int end;

       Task(int start, int end) {
           this.start = start;
           this.end = end;
       }
   }

   // Function to find the maximum number of non-overlapping tasks
   public static int maxTasks(Task[] tasks) {
       Arrays.sort(tasks, (a, b) -> a.end - b.end);

       int count = 0;
       int lastEndTime = Integer.MIN_VALUE;
       for (Task task : tasks) {
           if (task.start >= lastEndTime) {
               count++;
               lastEndTime = task.end;
           }
       }
       return count;
   }

   // Main method to test the maxTasks function
   public static void main(String[] args) {
       Task[] tasks = {
           new Task(1, 3),
           new Task(2, 5),
           new Task(4, 6),
           new Task(6, 7),
           new Task(5, 8),
           new Task(8, 9)
```

```
        };

        System.out.println("Maximum number of tasks that can be completed: " + maxTasks(tasks));
    }
}
```

Problem 2, 3

Problem 2: Sort an Array of Integers Using Heap Sort

Overview: Implement the Heap Sort algorithm to sort an array of integers. Heap Sort is a comparison-based sorting technique based on a Binary Heap data structure. It's an in-place algorithm, but it is not a stable sort.

Algorithm Steps:

1. **Build a Max Heap**:
    - Rearrange the array into a max heap, a complete binary tree where each parent node is greater than or equal to its children.
2. **Heapify the Array**:
    - Starting from the last non-leaf node, heapify each node in a bottom-up manner to ensure each subtree is a max heap.
3. **Sort the Array**:
    - Repeatedly remove the maximum element from the heap and move it to the end of the array, reducing the heap size each time.
    - After removing the maximum element, heapify the root node.
4. **Code**

```
public class HeapSort {

 // Function to sort an array using heap sort
 public static void heapSort(int[] arr) {
     int n = arr.length;

     // Build a max heap
     for (int i = n / 2 - 1; i >= 0; i--) {
         heapify(arr, n, i);
     }

     // Extract elements from the heap one by one
     for (int i = n - 1; i > 0; i--) {
         // Move current root to end
         int temp = arr[0];
         arr[0] = arr[i];
         arr[i] = temp;

         // Call max heapify on the reduced heap
         heapify(arr, i, 0);
     }
 }

 // Function to heapify a subtree rooted with node i
 public static void heapify(int[] arr, int n, int i) {
     int largest = i; // Initialize largest as root
     int left = 2 * i + 1; // left child
     int right = 2 * i + 2; // right child

     // If left child is larger than root
     if (left < n && arr[left] > arr[largest]) {
         largest = left;
```

```
        }

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }

        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    // Main method to test the heapSort function
    public static void main(String[] args) {
        int[] arr = {12, 11, 13, 5, 6, 7};
        heapSort(arr);
        System.out.println("Sorted array is: ");
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

Problem 3: Solve the Two Water Jugs Problem

Overview: Implement a program to solve the two water jugs problem. Given two jugs with capacities x and y liters, and a goal of z liters, determine the steps to measure exactly z liters if it is possible. The jugs do not have any markings to measure smaller quantities.

This is a classic problem in artificial intelligence and algorithms, often solved using breadth-first search (BFS) to find the shortest sequence of actions that leads to the goal state.

Algorithm Steps:

1. **State Representation**:
   - Represent the state of the jugs as a pair of integers (a, b), where a and b are the current amounts of water in each jug, respectively.
2. **BFS for State Exploration**:
   - Use BFS to explore all possible states starting from the initial state (0, 0).
   - Each state transition represents one of the following actions: fill a jug, empty a jug, or pour water from one jug to the other until either the first jug is empty or the second jug is full.
3. **Goal Check**:
   - At each step, check if any of the jugs has the target quantity z. If so, return the sequence of actions leading to this state.
4. **Handling Edge Cases**:
   - Ensure that z can be measured using the given jugs. If z is greater than x and y combined, or if z is not an integer multiple of the greatest common divisor of x and y, then the problem has no solution.
5. **Code**

```
import java.util.*;

public class WaterJugSolver {

   static class State {
       int a, b;
```

```
    State(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof State) {
            State state = (State) o;
            return this.a == state.a && this.b == state.b;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(a, b);
    }
}

// Function to check if a state is the goal state
public static boolean isGoalState(State state, int z) {
    return state.a == z || state.b == z;
}

// Function to solve the Water Jug problem using BFS
public static void solveWaterJugProblem(int x, int y, int z) {
    if (z > x + y) {
        System.out.println("No solution possible");
        return;
    }

    Queue<State> queue = new LinkedList<>();
    Set<State> visited = new HashSet<>();
    State initial = new State(0, 0);

    queue.add(initial);
    visited.add(initial);

    while (!queue.isEmpty()) {
        State current = queue.poll();

        if (isGoalState(current, z)) {
            System.out.println("Reached goal with " + current.a + " liters in Jug A and " + current.b + " lit
            return;
        }

        // Generate all possible next states and add them to the queue
        List<State> nextStates = generateNextStates(current, x, y);
        for (State nextState : nextStates) {
            if (!visited.contains(nextState)) {
                queue.add(nextState);
                visited.add(nextState);
            }
        }
    }

    System.out.println("No solution found");
```

```java
    }

    // Function to generate all possible next states
    public static List<State> generateNextStates(State current, int x, int y) {
        List<State> states = new ArrayList<>();
        // Fill Jug A
        states.add(new State(x, current.b));
        // Fill Jug B
        states.add(new State(current.a, y));
        // Empty Jug A
        states.add(new State(0, current.b));
        // Empty Jug B
        states.add(new State(current.a, 0));
        // Pour from A to B
        int pourAtoB = Math.min(current.a, y - current.b);
        states.add(new State(current.a - pourAtoB, current.b + pourAtoB));
        // Pour from B to A
        int pourBtoA = Math.min(current.b, x - current.a);
        states.add(new State(current.a + pourBtoA, current.b - pourBtoA));

        return states;
    }

    // Main method to test the solveWaterJugProblem function
    public static void main(String[] args) {
        int x = 4; // Capacity of Jug A
        int y = 3; // Capacity of Jug B
        int z = 2; // Goal quantity

        solveWaterJugProblem(x, y, z);
    }
}
```