Problem 1

Part A
Top is 7
20 7 45 8 5
min in stack is 5
20 7 45 8

Part B
    public int size()
    {
        int sum = 0;
        Node temp = start;
        while(temp != null)
        {
            sum++;
            temp = temp.next;
        }
        return sum;
    }

Part C
    public int popMin()
    {
        // empty list
        if (start == null) return -1;
        // calculate the min
        int min = peekMin();
        // min at the front of the list
        if (start.value == min)
        {
            start = start.next;
            return min;
        }
        // min not the first node in the list
        Node prev = start;
        Node current = start.next;
        while (current.value != min)
        {
            current = current.next;
            prev = prev.next;
        }
// delete the node
prev.next = current.next;
return min;
}

Part D
7 9 4 8

Problem 2

public class Balancer {
private static String LEFTS = "{";
private static String RIGHTS = "";"

public boolean isBalanced(String s){
    Stack<Integer> st = new Stack<Integer>();
    for(int k=0; k < s.length(); k++){
        char ch = s.charAt(k);

        if (Character.isWhitespace(ch)) continue;
        int leftIndex = LEFTS.indexOf(ch);

        if (leftIndex >= 0){
            st.push(leftIndex);
        } else {
            // PART C
            if (st.isEmpty()) return false;
            int rightIndex = RIGHTS.indexOf(ch);
            if (st.peek() != rightIndex) return false;
            st.pop();
        }
    }

    // PART B
    return st.isEmpty();
}

A. [4pts] LEFT and RIGHT serve as maps of brace or parens to an integer, providing the same integer value for corresponding braces, e.g., ( maps to 2 and ) maps to 2. Using a stack ensures checking balance since it’s LIFO so () matches since ( is on top of stack when ) is seen. Similarly if we assume […] works with . . . checked, then [ matches ] by the LIFO structure.
B. [4pts] See code
C. [4pts] See code
D. [4pts]
add < and > to the same location in LEFT/RIGHT respectively. For example:
LEFT = LEFT + "<";
RIGHT = RIGHT + ">";

**Problem 3**

```java
public int groupSize(boolean[][] grid, int row, int col) {
    if (row < 0 || row >= grid.length || col < 0 || col >= grid[0].length)
        return 0;
    if (!grid[row][col])
        return 0;
    grid[row][col] = false;
    return 1 + groupSize(grid, row+1, col) + groupSize(grid, row, col-1) +
    groupSize(grid, row, col+1);
}

public int numGroups(boolean[][] grid) {
    int count = 0;
    for (int i=0; i < grid.length; i++)
        for (int j=0; j < grid[0].length; j++)
            if (numGroups(grid, i, j) > 0)
                count += 1;
    return count;
}
```

**Problem 4**

```java
public boolean overlap(Interval b) {
    if (right < b.left)
        return false;
    if(left > b.right)
        return false
    return true;
}
```

// code for countIntervals
if(a[i].left <= max)
    count++;
delete(int x): O(n)
deleteMin(): O(n)
B. insert(int x): O(n)
delete(int x): O(n)
D. insert(int x): O(n)
delete(int x): O(n)
deleteMin(): O(1)

Problem 6

A.
1. If one of the words has no characters, there is no lcs
2. If the words share a first-letter, it will be part of lcs, find the rest recursively and combine them
3. Otherwise (no first letter in common), check word a with what comes after the first letter in word b, and vice-versa/symmetric case. This checks all combinations.

C1. Can use TreeMap since Pair implements comparable.
C2. if (myMemo.containsKey(p)) return myMemo.get(p);
C3. myMemo.put(p, after);
Also need to store t1 or t2 with p before returning in third case (when first chars do not match).