Instructions

We're going to start class today with a "simulated" exam I've cut-and-pasted together from previous exams in the archive. Once class starts, I'm going to ask everyone to start work and try to do questions 1-3 in the first 40 minutes of class. In order to simulate the time pressure of the real exam, please don't start questions 1-3 until I say so.

If you get here early and are looking for something to do, feel free to try and solve questions 4-6. I won't go over them in detail, but I will have solutions posted on the website.
Consider the following class for a new type of stack called a PriorityStack. This class will have the usual stack operations of pop(), push() and peek(), but also have some additional operations. The new operations are peekMin() which returns the minimum value in the stack and popMin() which removes the minimum value from the stack. The implementation of the stack will be with a singly linked list.

Shown below is part of the class with some of the methods not implemented. Not all the methods are shown.

```java
public class PriorityStack {

    private Node start;  // points to the head of the linked list

    public static class Node {
        int value;
        Node next;

        Node(int val, Node ptr) {
            value = val;
            next = ptr;
        }
    }

    public PriorityStack() {
        start = null;
    }

    // pushes new value onto the stack
    public void push(int val) {
        start = new Node(val, start);
    }

    // returns top of stack, returns -1 if stack is empty
    public int pop() {
        if (start == null)
            return -1;
        int returnval = start.value;
        start = start.next;
        return returnval;
    }

    // return the top of the stack, if the stack is empty return -1
    public int peek() {
        if (start == null) {
            return -1;
        }
        return start.value;
    }

    // return the minimum value from the stack
```
PART A. (6 pts) Given the following code, show the result of the stack after this code executes. Note that print lists the elements in the stack from left to right on one line starting with the top element.

```java
PriorityStack pstack = new PriorityStack();
pstack.push(5);
pstack.push(8);
pstack.push(45);
pstack.push(7);
pstack.push(32);
pstack.push(11);
pstack.pop();
pstack.pop();
System.out.println("Top is " + pstack.peek());
pstack.push(20);
pstack.print();
System.out.println("min in stack is " + pstack.peekMin());
pstack.popMin();
pstack.print();
```

List the Output:

PART B. (4 pts) Write the method size that returns the number of elements in the stack.

```java
// returns the number of nodes in the stack
public int size() {
```
PART C. (6 pts) Write the method popMin() that removes and returns the minimum element in the stack. You may call peekMin() or any of the methods shown in the class. You do not need to write any methods in the class you call.

    // return and remove the minimum value from the stack
    // return -1 if stack is empty
    public int popMin()
    {
    
    

PART D. (3 pts) Consider the following print2 method and recursive rprint method that we would like to add to the PriorityStack class.

    public void print2()
    {
        rprint(start);
        System.out.println();
    }
    
    public void rprint(Node s)
    {
        if (s != null)
        {
            rprint(s.next);
            System.out.print(s.value + " ");
        }
    }

If the stack is from top to bottom: 8 4 9 7
show the result of calling print2();
PROBLEM 2:  \textit{(Stacks of Braces (16 points))}

The following strings of parentheses/braces/brackets are balanced in that each left-symbol is matched by the corresponding right-symbol.

\begin{verbatim}
" ( ) ( ) ( ) "
" ( [ { [ ] } ] ) "
" [ [ ( ) ] { } ] ]"
\end{verbatim}

The following strings are not balanced for the reason given on each line.

\begin{verbatim}
" ( ( ) ] "  the right-\} doesn't match the left-\{
" [ ( ( ) ) "  the left-[ isn't matched by a corresponding right-]
" [ { } ] ) "  the right-\}’s do not match any symbol
\end{verbatim}

The method \texttt{isBalanced} below returns true for all strings shown as balanced above as well as for every properly-balanced string. The method stores a number corresponding to each left-symbol on a stack and matches the top element when a right-symbol is found as the string of symbols is processed one character at-a-time. The method returns false for the first un-balanced string above, but returns true for the second one and throws an exception for the third one. Questions appear after the code.

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 {
                // add code for C

                int rightIndex = RIGHTS.indexOf(ch);
                if (st.peek() != rightIndex) return false;
                st.pop();
            }
        }

        return true;  // replace for B
    }
}
A. [4pts] Explain how the strings LEFT and RIGHT combined with the use of the method `indexOf` and a stack result in all balanced strings being recognized correctly by `isBalanced`. In other words, describe how the method works at a high-level. Extra credit for appropriate use of the word `map` in your explanation.

B. [4pts] The method `isBalanced` returns true for the string "[(())]" although the string isn't balanced because the first bracket isn't matched. Replace the last return statement so that correctly-balanced strings are still identified, but the unbalanced string "[(())]" is identified as unbalanced as are other strings with at least one unmatched symbol.

C. [4pts] The string "[{[]}])" is not correctly identified as unbalanced because the code throws an empty-stack exception. Insert a statement in the place identified in the code so that this string is properly identified as unbalanced and no exception is thrown.

D. [4pts] Show in code or describe in words how to add the symbols < and > so that the strings below are correctly identified as balanced, e.g., so that in addition to parentheses, braces, and brackets the less-than and greater-than symbols can be used in left-right symbol expressions.

"[(><)]"
"<><> >> ()[]"
PROBLEM 3: (Graded 18 points)

Consider a $N$-by-$N$ grid in which some squares are occupied by black circles. Two squares belong to the same group if they share a common edge.
In the picture to the right, there are

- 1 group of 4 occupied squares
- 1 group of 3 occupied squares
- 2 groups of 2 occupied squares
- 2 groups of individually occupied squares

Given that grid is an two-dimensional array where $grid[i][j] == true$ if and only if grid cell $(i,j)$ is occupied, the following questions ask you to find the groups.

A. [10pts] Given a grid and a grid cell location $(row, col)$, $groupSize$ should compute the size of the group including that square. For example, in the example above $groupSize(grid, 1, 3)$ should return 2.

```java
public int groupSize(boolean[] [] grid, int row, int col)
{
```

Your code may modify the state of the grid
B. [8pts] `numGroups` should return the number of different groups in a grid. For the example of the previous page, `numGroups` should return `1 + 1 + 2 + 2` or 6. You may add helper methods as you feel necessary.

```java
public int numGroups(boolean[][] grid)
```
PROBLEM 1: (Interval)
From Sedgewick: Consider the following data type, for intervals on the line:

```java
public class Interval implements Comparable<Interval>
{
    public int left;
    public int right;

    public Interval(int l, int r) {
        left = l; right = r;
    }

    public int compareTo(Interval b) {
        return left - b.left;
    }

    /**
     * Returns true if and only if this Interval overlaps with b
     */
    public boolean overlap(Interval b) {
        // TODO: complete in part A
    }
}

public static int countIntervals(Interval[] a) {
    Arrays.sort(a);
    int count = 1;
    int max = a[0].right;
    for (int i = 1; i < a.length; i++) {
        // TODO: complete missing line of code for part B
        if (a[i].right > max)
            max = a[i].right;
    }
    return count;
}
```

For a particular application, clusters of intervals are of importance. To find clusters, replace any pair of intervals that intersects (by even an endpoint) by the union of the two intervals, continuing until all intervals do not intersect. For example, the following set of intervals has 3 clusters:

```
____   _______   _____
```

Note that you are guaranteed to have Intervals with non-negative numbers. Given an array of intervals, how
many clusters are there? The brute-force algorithm that calculates the overlap of all entries is quadratic. It can also be done in $O(n \log n)$ time with method count above.

A. Complete overlaps above.

B. Complete countIntervals above.
PROBLEM 7: (Analyze (15 points))

Suppose we want to create a data structure called DataStore that has the following operations:

- insert(int x) - inserts x into DataStore
- delete(int x) - if x is in DataStore, deletes x from DataStore
- deleteMin() - removes and returns the minimum value in DataStore

Consider the following implementations for DataStore. Assume operations are implemented efficiently. Give the worst case big-Oh analysis of each operation.

A. (3 pts) The implementation is a singly linked list not in any order with a pointer to the start of the linked list. Suppose n items are in DataStore. What are the worst case times of the following operations?

- insert(int x):
- delete(int x):
- deleteMin():

B. (3 pts) The implementation is a singly linked list stored in increasing sorted order with a pointer to the start of the linked list. Suppose n items are in DataStore. What are the worst case times of the following operations?

- insert(int x):
- delete(int x):

D. (3 pts) The implementation is an array sorted in decreasing order (with the largest element in slot 0). Assume there is plenty of space for the array to grow. Suppose n items are in DataStore. What are the worst case times of the following operations?

- insert(int x):
- delete(int x):
- deleteMin():
6

PROBLEM: (Nothing in Common (15 points))

The Longest Common Subsequence or lcs of two strings is the longest sequence of characters in order, not necessarily adjacent, that is in common to both strings. This has applications in text processing, genomics, and web searching.

For example, the lcs of the strings "sorting" and "describe" is "si" and the lcs of the strings "human" and "chimpanzee" is "hman".

The code below on the left correctly returns the longest common subsequence of two strings.

```java
public String lcs(String a, String b){
    if (a.length() == 0 || b.length() == 0){ // CASE 1
        return "";
    }

    // placeholder A
    if (a.charAt(0) == b.charAt(0)){ // CASE 2
        String after = lcs(a.substring(1),b.substring(1));
        after = a.charAt(0) + after;

        // placeholder B
        return after;
    }
    // CASE 3
    String t1 = lcs(a.substring(1),b);
    String t2 = lcs(a,b.substring(1));
    if (t1.length() > t2.length()) return t1;
    return t2;
}
```

Part A (3 points)

Describe in words what the three cases in the code are and why these cases make the code correct.
Part C (8 points)

The list of words on the left of the previous page shows some of the calls for the method lcs for the words "sorting" and "describe" - the list shows the two parameters passed to lcs and the number of calls with these parameters. For example, there are 491 calls with parameters "g" and "be". For the strings "sorting" and "describe" there are 6,238 calls made to find the longest common subsequence of "sri".

To make the method faster you will memoize so that results are stored and retrieved rather than being recomputed. For this problem you'll describe how to implement memoization and how to make it work. The idea is to make at most one recursive call for each pair of parameters. To do this you'll use the class Pair below.

The idea is to modify lcs so that the result returned for parameters (a,b) as a Pair is stored in a map and retrieved if it is stored rather than recomputed recursively.

You'll do three things for this part of the problem:

- Comment on the definition for the map which would be an instance variable.
- Show how to check the map and return the stored value if it's present.
- Show how to store a value in the map so it will be available for subsequent calls.

Using this memoization technique reduces the calls from 6,238 to 100 for the strings "sorting" and "describe".

```java
public class Pair implements Comparable<Pair> {
    String a;
    String b;
    public Pair(String s, String q) {
        a = s;
        b = q;
    }
    public int hashCode() {
        return a.hashCode() * 100 + b.hashCode();
    }
    public boolean equals(Object o) {
        Pair p = (Pair) o;
        return a.equals(p.a) && b.equals(p.b);
    }
    public int compareTo(Pair o) {
        int f = a.compareTo(o.a);
        if (f == 0) return b.compareTo(o.b);
        return f;
    }
    public String toString() {
        return "(" + a + ", " + b + ")";
    }
}
```

The definition for the map follows:

```java
Map<Pair, String> myMemo = new HashMap<Pair, String>();
```

C.1

Can you replace HashMap with TreeMap in the definition above and have the rest of the code work (after modifying lcs to use the memo/cache)? Justify your answer.

C.2

Put code in lcs at the location marked placeholder A to check the cache and return the result for a Pair p defined as:

```java
Pair p = new Pair(a,b);
```

C.3

Put code at the location marked placeholder B so that a value is stored properly in the memo/cache for subsequent retrieval. Where else in the code would you also have to store values in the map (label on the lcs code page).