Test 2 Review Questions

PROBLEM 1: (Short Ones)

A. What is the average (expected) time, in big-Oh notation, needed to determine whether a particular value appears in the following data structures (e.g., contains), assuming that the data structure contains $n$ values, and that an appropriate, fast algorithm is used. Assume the data is random.

   I. Linked list

   II. Unsorted array

   III. Binary search tree

   IV. HashMap

B. Is the worst-case time needed to search for a value different from the average (expected) time for any of the data structures in the previous question? If so, which ones, and what is (are) the worst-case time(s) for each one?

C. The values 1, 2, . . . , $n$ are added to an initially empty queue, and then $n/2$ values are removed from the queue. What is the big-O runtime of this sequence of queue operations where queues are implemented using doubly-linked lists with pointers to the first and list nodes; justify your answer briefly.

D. Suppose the same sequence of queue operations are performed, but queues are implemented with singly-linked lists and pointers to the first and last nodes. What is the big-O runtime; justify your answer.
For each of the following methods and data structures in the matrix below, give the Big-Oh. If worst-case and average-case are different, note that.

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>findMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted ArrayList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted ArrayList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PriorityQueue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trie</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PROBLEM 2:  (Analyze (11 points))

Suppose we want to create an interface `IFind` to support the following operations:

```java
public interface IFind {
    // returns true if and only if x is in data structure
    public boolean find(int x);
    // returns the maximum element
    public int findMax();
    // returns the median element
    public int findMedian();
}
```

Note: The median element is the middle element, if the elements were in sorted order. For example, the median of 3, 9, 11, 5, 7) is 7.

In each case, finding the element does not remove it from the data structure. Consider the following implementations for NDS. Assume operations are implemented efficiently. Give the worst case big-Oh analysis of each operation where there are $n$ elements in the structure.

<table>
<thead>
<tr>
<th></th>
<th>find(x)</th>
<th>findMax()</th>
<th>findMedian()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted doubly-linked list with head and tail pointers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min-Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PROBLEM 3:  (Drawing)

A. Draw what `head` and `list` are pointing to after the following code is executed. Draw all of the list nodes and their contents.

```java
String[] data = {"A", "B", "C", "D"};

ListNode head = new ListNode(data[0], null, null);
ListNode list = head;
for (int k=1; k < data.length; k++) {
    list.next = new ListNode(data[k], null, null);
    list = list.next;
}
head.prev = list;
```
B. Draw what `root` and `next` points to after executing the following code. Indicate if there is some kind of error.

```java
TreeNode root = new TreeNode(8, new TreeNode(3, new TreeNode(2, null, null),
                                  new TreeNode(7, null, null)),
                       new TreeNode(10, null, null));
TreeNode next = root.right.right;
```
PROBLEM 4: (Strands)

We would like to keep track of long strands of characters, (e.g. DNA), using a class, so we create the IStrand interface.

interface IStrand
{
    public void prepend(String s);
    public int length();
}

Below are two classes, StringStrand and LinkStrand that implement this interface.

class StringStrand implements IStrand
{
    private String myString;
    public StringStrand(){
        myString = "";
    }

    public void prepend(String s){
        myString = s + myString;
    }

    public int length(){
        return myString.length();
    }
}

class LinkStrand implements IStrand
{

    private Node myFront;
    private int myCount;

    public LinkStrand(){
        myFront = null;
        myCount = 0;
    }

    public void prepend(String s){
        myFront = new Node(s,myFront);
        myCount += s.length();
    }

    public int length(){
        return myCount;
    }
}
A. Which class do you think would be faster when performing \( n \) \texttt{prepend} operations? Briefly explain why.

B. Instead of just prepending to a Strand, we would like to splice in a String and insert it anywhere. That is, we would like to have \texttt{s.insert(k,str)} to add \texttt{str} at the \( k \)th position. Prepending could now also be performed as \texttt{s.insert(0,str)}. We would like this behavior in both \texttt{LinkStrand} and \texttt{StringStrand}. What do we need to modify in \texttt{IStrand}?

C. Write the \texttt{insert} method for \texttt{StringStrand} below.
D. Write the \texttt{insert} method for \texttt{LinkStrand} below.

E. Which \texttt{insert} method will be faster for large strands? Briefly explain why.
PROBLEM 5: (Counting trees (16 points))

The Counting BSTs APT asks you to create a class BSTcount that contains the method howMany, which takes an array of integers, and returns a long value that represents the number of distinct possible BSTs resulting from the given set of values.

A solution is given below that is syntactically correct, but does not currently solve any of the test cases.

```java
public class BSTcount {

    /**
     * Return the number of distinct possible BSTs resulting from the given values
     * @param values elements are distinct
     */
    public long howMany(int[] values) {
        return howMany(values.length);
    }

    /**
     * Return the number of distinct possible BSTs with size elements
     */
    public long howMany(int size) {
        long trees = 0;
        for (int left=0; left <= size-1; left++) {
            int right = size - 1 - left;
            trees += howMany(left)*howMany(right);
        }
        return trees;
    }
}
```
A. The solution above returns 0 for both `values = {90, 12}` and `values = {90, 13, 2, 3}` instead of the correct answers (2 and 14 respectively). Explain why and fix the code above. Your fix should be one or two lines of code.

B. After your fix, your program works but it is not able to complete in time for the APT grader for large arrays. Using memoization, you can avoid repeating the calculation of previous calls to `howMany` and significantly reduce the running time of your algorithm. Add code as necessary. You may use the space below if necessary.

C. Explain why the memoized solution is so much faster by stating the recurrences for the unmemoized and memoized versions. Do not solve the recurrence. You can use \ldots to indicate patterns like $n + (n - 1) + \ldots + 1$ is the sum of the sequence of numbers from $n$ to 1.
PROBLEM 6: (**Towering Fir? None!** *(X points)*)  

The picture below shows a $R_3$-tower. The bottom rectangle is a $3 \times 4$ rectangle, topped by a $2 \times 3$ rectangle which is topped by a $1 \times 2$ rectangle. In an $R_4$ tower a $4 \times 3$ rectangle would be added below the $3 \times 4$ rectangle shown and in general in an $R_N$-tower there are $N$ rectangles and the bottom rectangle is a $N \times (N - 1)$ rectangle.

A. The $R_3$-tower is 6 units tall. How tall is a $R_6$-tower? Justify your answer briefly.

B. How tall is a $R_{10}$-tower? Justify briefly.

C. A $R_3$-tower is made of $12 + 6 + 2 = 20$ blocks as shown. How many blocks are there in a $R_6$-tower?

D. By definition we define $B(n)$ to be the number of blocks in an $R_n$-tower so that $B(1) = 2$, $B(2) = 8$, and $B(3) = 20$. Explain why the following relationship/equation is true for any $n > 1$.

$$B(n) = B(n - 1) + (n^2 + n)$$
E. Write a method `tblocks` so that `tblocks(n)` returns the number of blocks in an $R_n$-tower, e.g., so that `tblocks(3)` returns 20 and `tblocks(2)` returns 8.

To earn 5/5 points you must write the method recursively. You can earn 4/5 points for writing a correct iterative method.

```java
public int tblocks(int n) {
}
```
F. Using big-Oh (in terms of \(n\)), how tall is an \(R_n\)-tower? For example, a \(3\)-tower is 6 units tall. Justify your answer briefly. The correct answer is either \(n\), \(n^2\), \(n^3\), or \(2^n\).

G. Using big-Oh (in terms of \(n\)), how many \textbf{total blocks} are in an \(R_{n^2}\)-tower? For example, there are 40 blocks in an \(R_{2^2}\)-tower (which is an \(R_4\)-tower). Justify your answer briefly. The correct answer is either \(n\), \(n^2\), \(n^3\), or \(n^4\).
PROBLEM 7: (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 \texttt{count} above.

A. Complete \texttt{overlaps} above.

B. Complete \texttt{countIntervals} above.
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 the grid is a 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.

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) {
```
PROBLEM 9:  (Puzzle Hunt)

You are given a matrix of positive integers to represent a game board, where the (0, 0) entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move does not take you off the board. You are interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.

Consider the initial game board to the left, and notice that the upper left corner is occupied by a 2. That means you can take either two steps to the right, or two steps down (but not two steps to the left or above, because that would carry you off the board). Suppose you opt to go right so that you find yourself in the configuration to the right.

After that, you could continue along as follows:

This series of moves illustrates just one of potentially several paths you could take from upper left to lower right. Your task is to write a method called `numPaths`, which takes a 2-d array of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way of marking that you’ve been there. Just be sure to restore the original value as you exit the recursive call. You may want to write a helper function to handle the recursion and a utility function to decide it you are on the board or not.
A. Write `numPaths` below.

```java
/**
 * Calculates total number of distinct ways one could travel from the 
 * upper left corner of grid to the lower right corner, given the 
 * constraint that no single path should ever visit the same location twice. 
 * 
 * @param board square matrix board[i][j] is the number of squares 
 * one can advance vertically or horizontally from (i,j) 
 * @return the number of possible paths from (0,0) to the lower 
 * right corner of board (board.length-1, board[0].length - 1) 
 */
public static int numPaths(int[][] board)
{
}

// HELPER FUNCTIONS
/**
 * @return true if (row,col) is within the bounds of the board 
 * (i.e. 0 <= row < board.length and 0 <= col < board[0].length) 
 * false otherwise 
 */
public static boolean onBoard(int[][] board, int row, int col) 
{
}

/**
 * @return the number of possible paths from (row,col) to the lower 
 * right corner of board (board.length-1, board[0].length - 1) 
 */
public static int numPaths(int[][] board, int row, int col)
{
}
```

B. Give a recurrence for your solution. You do not need to solve the recurrence.

**PROBLEM 10 : (Grids)**
In recitation, we discussed a maze generation algorithm for a $N \times N$ maze listed below.

- Start with each cell in the grid in its own set numbered $[0, N^2 - 1]$
- Create all interior and exterior walls
- Repeat until all cells are in the same set
  - Choose an interior wall at random
  - If the two cells (call them $x$ and $y$) adjacent to this wall are in different sets
* remove the wall
* perform a \textit{union} on $x$ and $y$

The cells are stored with a union-find data structure similar to the ones that implemented the \texttt{IUnionFind} interface in the Percolation assignment. Consider the complexity for the following implementations of union-find data structures - repeated from the lecture notes.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>init</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>$N$</td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td>quick-union</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>weighted quick-union</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>weighted quick-union w/ path compression</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

**A.** What is the big-Oh of the maze-generating algorithm above for the following if we use the following structures to keep track of the sets of cells.

Assume that when you choose an interior wall at random, the two adjacent cells are in different sets half (50\%) of the time.

1. quick-find

2. quick-union

3. weighted quick-union w/ path compression
B. You are given a Maze class that represents the internal state of a maze. The implementation of the methods is omitted, but each of the methods properly implements its specification described in the comments.

```java
public class Maze {
    // constructs new randomly generated (height x width) maze
    // all cells are set to be not visited
    public Maze(int height, int width) { // ... }

    // returns the width of maze in # columns
    public int getWidth() { // ... }

    // returns the height of the maze in # rows
    public int getHeight() { // ... }

    // returns true iff (row, col) is the goal cell
    public boolean isGoal(int row, int col) { ... }

    // mark cell as visited
    public void markVisited(int row, int col) { ... }

    // mark cell as NOT visited
    public void clearVisited(int row, int col) { ... }

    // returns true iff cell has been marked visited
    public boolean isVisited(int row, int col) { ... }

    // returns true iff (row1, col1) and (row2, col2) are adjacent
    // and there is no wall between the cells
    public boolean isBlocked(int row1, int col1, int row2, col2) {}
}
```

C. Write a method pathExists that returns true if and only if a path exists from (row, col) to the goal for a particular Maze object. Your method should use the public methods described above to access the internal state of the Maze. You can write any helper methods that you find helpful.

```java
public boolean pathExists(int row, int col)
```

PROBLEM 11: (Tries)
Given the following definition of a Trie Node:

```java
public static class Node {
    boolean isWord;
    Map<Character, Node> children;

    Node() {
        isWord = false;
        children = new TreeMap<Character, Node>();
    }
}
```
A. Write a method to add a word to a Trie

```java
/**
 * Adds a string to the trie rooted at t
 */
public void add(Node t, String s) {
```

B. In this problem, you will write a method `trieIntersect` that returns a trie that is the intersection of two tries.

You may assume a method `copyTrie` is already written. Its prototype is below:

```java
public Node copyTrie(Node n) {
    // implementation omitted
}
```

You might want to model your method after the following definition of `trieUnion`:

```java
/**
 * returns a trie that is the union of lhs and rhs i.e., every word in
 * lhs
 * or in rhs is in returned trie
 */
public Node trieUnion(Node lhs, Node rhs) {
    if (lhs == null)
        return copyTrie(rhs);
    else if (rhs == null)
        return copyTrie(lhs);
    else {
        Node temp = new Node();
        boolean somethingBelow = false;
        // get set of all children
        HashSet<Character> allChildren = new HashSet<Character>(lhs.children.keySet());
        allChildren.addAll(rhs.children.keySet());
        for (char ch : allChildren) {
```
Node uchild = trieUnion(lhs.children.get(ch),
                         rhs.children.get(ch));
if (uchild != null) {
    // something found
    somethingBelow = true;
    temp.children.put(ch, uchild);
}
}
temp.isWord = lhs.isWord || rhs.isWord; // in union?
if (somethingBelow || temp.isWord)
    // something to return
    return temp;
else
    return null;
}

Steps in writing trieIntersect

1. Make the recursive calls to see what is in the intersection below
2. Examine isWord fields to determine if the current node/path is in the intersection
3. Return an empty trie unless there is something below or the current node/path is in the intersection
// returns a trie that is the intersection of a and b
// i.e., every word in a AND in b is in returned trie
Node* trieIntersect(Node a, Node b) {