Test 2 Review Questions

PROBLEM 1:  (Short Answer)

A. In order to use the class `Point` containing fields `x` and `y` in a HashSet, you are considering multiple hash functions. Of these hash functions, which one would give the best performance in a HashSet? Assume that your points are likely to be between (0, 0) and (1280, 1024) (the size of the average computer monitor).

   I. `public int hashCode () { return super.hashCode(); }`
   II. `public int hashCode () { return 42; }`
   III. `public int hashCode () { return x; }`
   IV. `public int hashCode () { return x + y; }`
   V. `public int hashCode () { return x * 3 + y; }`
   VI. `public int hashCode () { return x * 1000 + y; }

B. Compare and contrast
   I. Abstract Class & Interface
   II. Class & Object
   III. LinkedList & ArrayList
   IV. Union-find with weighted union operations where we keep track of the size of each tree & quick-union
   V. Union-find with path compression on find operations & quick union

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.

E. 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;
   ```
F. 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 2: (TriPyramid (20 points))

The picture below shows a four-rowed, two-dimensional pyramid constructed of triangles – this will be called a 4-pyramid in this problem because it has four rows. The top triangle is number one, then the triangles are numbered left-to-right in a row and top-to-bottom as shown. In the fourth row there are seven triangles; in general in the \(N^{th}\) row there are \(2N - 1\) triangles. In answering questions below assume we’re discussing an \(N\)-pyramid with \(N\) rows for a large value of \(N\).

Part A (2 points)
What is the exact value or number of the right-most triangle in the seventh row?

Part B (2 points)
What is the exact value of the right-most triangle in the 40\(^{th}\) row?

Part C (2 points)
What is the exact value of the left-most triangle in the 61\(^{st}\) row?

Part D (2 points)
Using big-Oh what is the number of triangles in the bottom row of a pyramid with \(N\) rows. Justify your answer.

Part E (2 points)
Using big-Oh what is the number of triangles in the bottom row of a pyramid with \(N^2\) total triangles. Justify your answer.

Part F (2 points)
Using big-Oh what is the number of triangles in the bottom row of a pyramid with \(N^4\) total triangles. Justify your answer.
Part G (8 points)
The diagram below shows four 3-pyramids combined to make an 6-pyramid. Assume there is a function or method `combine` that takes an $N$-pyramid as a parameter and returns a new pyramid created by combining four $N$-pyramids as shown. For example, the pyramid shown below would be returned by the call `combine(3)`.

![Diagram of 6-pyramid](image)

G.1 (2 points)
What is the big-Oh number of triangles in the bottom row of the pyramid returned by the call `combine(N)`. Justify your answer.

G.2 (2 points)
What is the big-Oh value of the rightmost pyramid in the bottom row (the triangle with the largest number) in the pyramid returned by the call `combine(combine(N))`. Justify your answer.

G.3 (4 points)
Consider the pseudo-code below for a sequence of calls to create a pyramid. For example, when $N = 2$ the initial pyramid $p$ is constructed before the loop with two rows and three triangles in the bottom row; the loop then executes twice. The first time through the loop results in $p$ having four rows. The second time through the loop results in $p$ having eight rows (with 15 triangles in the bottom row and the value printed is 64).

```java
Pyramid p = new Pyramid(N); // create pyramid with N rows
int size = p.rows(); // set size to N
for(int k=0; k < size; k++){
  p = combine(p);
}
System.out.println("biggest number is "+p.lastPyramid());
```

If the value of $N = 10$ so that the initial pyramid has 10 rows in which the last pyramid is numbered 100 what is the value printed by the code above? Justify your answer. Your answer should be exact, but can be expressed using exponentiation and multiplication, e.g., $3^5 \times 100^2$ is acceptable as an answer.

PROBLEM 3: (Reverse (9 points))
Each of the Java functions on the left take a string $s$ as input, and returns its reverse. For each of the following, state the recurrence (if applicable) and give the big-Oh complexity bound. Recall that concatenating two strings in Java takes time proportional to the sum of their lengths, and extracting a substring takes constant time.
A. public static String reverse1(String s) {
    int N = s.length();
    String reverse = "",
    for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}

B. public static String reverse2(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse2(right) + reverse2(left);
}

C. public static String reverse3(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
        a[i] = s.charAt(N-i-1);
    return new String(a);
}

PROBLEM 4: (Interval)
From Sedgewick: Consider the following data type, for intervals on the line:

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 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 on the previous page 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 on the previous page. 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 ... to indicate patterns like $n + (n - 1) + \ldots + 1$ is the sum of the sequence of numbers from $n$ to 1.

PROBLEM 6: (BST redux (16 points))

For this problem binary trees, note that store integer values instead of strings and no duplicate values appear in the tree.

By definition a binary search tree is a binary tree if each of the following properties hold for every node in the tree.

1. All values in the left subtree of a node are less than the node’s value.
2. All values in the right subtree of a node are greater than the node’s value.
3. Both the left and right subtree of a node are binary search trees.

A. The methods getMin and getMax below return the smallest and largest values in a tree, respectively. They are used in writing isSearchTree that returns a boolean value indicating whether a tree is a search tree using the definition of a search tree given above.

```java
public int getMax(TreeNode root){
    if (root == null) return Integer.MIN_VALUE;
    return Math.max(root.info,
                    Math.max(getMax(root.left), getMax(root.right)));
}

public int getMin(TreeNode root){
    if (root == null) return Integer.MAX_VALUE;
    return Math.min(root.info,
                    Math.min(getMin(root.left), getMin(root.right)));
}

public boolean isSearchTree(TreeNode root){
    if (root == null) return true;
    return isSearchTree(root.left) &&
```
A student suggests different implementations of `getMax` and `getMin` because in their use in the code above they are only called when it is known that the tree is a search tree. The alternate implementations are below.

```java
public String getMax(TreeNode root){
    while (root.right != null){
        root = root.right;
    }
    return root.info;
}

public String getMin(TreeNode root){
    while (root.left != null){
        root = root.left;
    }
    return root.info;
}
```

Explain what the recurrences would be in the average and worst case for `isSearchTree` using these alternative implementations of `getMin` and `getMax`. You do not need to solve the recurrences, you just need to give them.

Another student says that the alternative implementations can be used even when the body of `isSearchTree` is below, where the recursive calls are swapped with the calls to `getMin` and `getMax`. In a few words indicate whether you think this code will work correctly with the alternative implementations and why you think this.

```java
public boolean isSearchTree(TreeNode root){
    if (root == null) return true;
    return
    root.info > getMax(root.left) &&
    root.info < getMin(root.right) &&
    isSearchTree(root.left) &&
    isSearchTree(root.right) &&
}
```

PROBLEM 7: (Grids (18 points))

```
public boolean isSearchTree(TreeNode root){
    if (root == null) return true;
    return
    root.info > getMax(root.left) &&
    root.info < getMin(root.right) &&
    isSearchTree(root.left) &&
    isSearchTree(root.right) &&
}
```
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.

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) {
    // Existing code changes to this function:
    // 
    // Instead of marking that you've been there. Just be sure to restore the original
    // you should change the value at any given location to a zero as a way
    // share a common sub
    // path twice, but two paths are considered to be distinct even if they
    // corner of the board. Note that you never want to count the same
    // constraint that no single path should ever visit the same location twice.
    // write a function called
    // ways you could take from upper left to lower right. Your job here is to
    // move
    // So, this series of moves illustrates just one of potentially several paths
    // squares
    // 
    // You're
    // Problem 4: Pascal's Travels
    // Given that
    // grid
    // 
    // of integers
    // 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.

/*
 * 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 9: (Heaps of Trouble (10 points))
A minheap is a heap such that each node obeys the heap value property – the value in each node is less than
the value in the node’s subtrees. A min heap is implemented via an array with the root at index 1 as we
discussed in class. The children of the node stored in element i of the array being at positions 2i and 2i + 1.

A. Suppose that there are n distinct elements in a minheap. Which positions (indices in the array) could
possibly be occupied by the fourth smallest element? Circle all that apply.
   I. 1
   II. 2 or 3
   III. 4, 5, 6, or 7
   IV. 8 through 15
   V. 16 and higher

B. Complete the method lessCount to determine the number of elements in a minheap that are less than
a target number. A
In the example below, lessCount(heap,13) should return 3 (6, 8, and 10 are less than 13), and
lessCount(heap,8) should return 1 (only 6 is less than 8).

```
6
/\     /\  
|  |   |  |
22 36 10 63
/      /   \
38     24
```

The worst case running time of your function should be \(O(k)\) where \(k\) is the number of elements less
than the key. We provide an auxiliary function header that can be called from lessCount, but you
don’t have to use it.

/**
 * Returns the number of elements in the heap that are
 * less than target
 */
int lessCount(int[] pq, int target) {
}

public int lessAux(int[] pq, int index, int target) {
}