Test 2 Review

PROBLEM 1:  (Short Ones)

A. Why should you use a HashMap rather than a TreeMap? Choose the best answer.
   I. To conserve memory.
   II. Because the performance of get and put is better
   III. Because the Java library implementation is optimized.
   IV. To guarantee that lookups (get) will be faster than inserts (put).
   V. Because it supports a fast sort implementation.

B. Suppose that I take a sorted linked list of N integers, and break it into a list of M equal-sized sorted
   lists of integers (that is, put the first N/M integers into the first list, the next N/M into the second,
   etc.). What is the worst-case time for finding whether an integer x is in anywhere in this list of lists?
   If for some fixed N, you can choose M, what M should you choose for maximum lookup speed?

C. 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).
   A. public int hashCode () { return super.hashCode(); }
   B. public int hashCode () { return 42; }
   C. public int hashCode () { return x; }
   D. public int hashCode () { return x + y; }
   E. public int hashCode () { return x * 3 + y; }
   F. public int hashCode () { return x * 1000 + y; }

G. Given the following recursive method mystery:

   int mystery(int n)
   {
       if (n < 0)
           return -mystery(-n);
       else if (n < 10)
           return n;
       else
           return mystery(n/10 + n % 10);
   }

   What are would following calls evaluate to?
   I. mystery(7)
II. mystery(~512)

H. True or False State whether the following statement is true or false. If false, you should give a specific counterexample.

I. A certain hash table contains \( N \) integer keys, all distinct, and each of its buckets contains at most \( K \) elements. Assuming that the hashing function and the equality test require constant time, the time required to find all keys in the hash table that are between \( L \) and \( U \) is \( O(K \times (U - L)) \) in the worst case.

II. Instead of using a heap, we use a sorted ArrayList to represent a priority queue. The worst-case big-Oh of add and poll do not change.

I. Draw the list (i.e. that is a box-and-pointer diagram) resulting from the following code.

```java
Node m = l.next.next;
Node n = new Node("5", m);
```

Your answer should clearly note what \( l, m, \) and \( n \) point to.

PROBLEM 2 : (Hash (6 pts))

In the Markov assignment, the WordNgram class encapsulated \( N \) words/strings so that the group of \( N \) words can be treated as a key in a map.

```java
public class WordNgram implements Comparable<WordNgram>{
    private String[] myWords;
    ...
    /**
     * Return true if this N-gram is the same as the parameter: all words the same.
     * @param o is the WordNgram to which this one is compared
     * @return true if o is equal to this N-gram
     */
    public boolean equals(Object o){
        WordNgram wg = (WordNgram) o;
        if (myWords.length != wg.myWords.length)
            return false;
        for(int k = 0; k < myWords.length; k++)
            if (!(myWords[k].equals(wg.myWords[k]))
                return false;
        return true;
    }
}
```

In this problem, you will answer questions about possible implementations of hashCode.

A. Will get and put still work in a HashMap if the WordNgram’s hashCode implementation is as follows? Explain why or why not. What will be the average run-time for get and put if there are \( n \) WordNGrams stored in the HashMap.
public int hashCode(){
    // TODO return a better hash value
    return 15;
}

B. Why is the following hashCode implementation flawed?

public int hashCode(){
    int result = 0;
    for(int k = 0; k < myWords.length; k++)
        result += myWords[k].hashCode();
}

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: (Boggle)
The game of Boggle is usually played using sixteen letter cubes. A letter cube can be represented as a string of length 6, one character for each face on the cube. Given a list of letter cubes and a target word, you want to determine whether it is possible to spell that word using those letter cubes, where each cube can be used at most once in spelling the word.
Examples: given the list of cubes {"etaoin", "shrdlu", "qwerty"}, it is possible to spell the words "as" and law! but not "weld" or "toe".
Write a recursive, backtracking method `canSpell` that takes a target word along with an `ArrayList` of letter cubes. The function should return `true` if it is possible to spell the word using the cubes in the list and `false` if otherwise. Assume that the word and letter cubes will only contain lowercase letters.

```java
public static boolean canSpell(String word, ArrayList<String> cubes)
{
}
```

**PROBLEM 5: (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:

![Game Board Illustration]

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)
    {

    }

PROBLEM 6 : (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 “sri” 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
class Pair {
    public Pair(String first, String second) {
        this.first = first;
        this.second = second;
    }
    // getters and setters...
}

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 B (4 points)
What is the running time of this code for two strings of N-characters? Use big-O and justify your answer.

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”.

The definition for the map follows:

```
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:

```
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).

PROBLEM 7: (Stacks and Queues (6 points))

A. Describe the contents of stack s after the method convert executes. That is, describe the contents in a general manner based on what's in s before the code executes.

```
public void convert(Stack<Object> s){
    ArrayList<Object> list = new ArrayList<Object>();
    while (s.size() > 0) {
        list.add(s.pop());
    }
```
for(Object o : list) {
    s.push(o);
}

B. What happens if a queue is used instead of a stack in the code above, e.g.,

public void convert(Queue<Object> q){
    ArrayList<Object> list = new ArrayList<Object>();
    while (q.size() > 0) {
        list.add(q.remove());
    }
    for(Object o : list) {
        q.add(o);
    }
}

PROBLEM 8: (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.

" ( ) ( ) ( ) "
" ( [ { [ ] } ] ) "
" [ [ ( ) ] ] ] "

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

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

The method 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 {
                if (st.isEmpty() || st.peek() != -leftIndex - 1) return false;
                st.pop();
            }
        }
        return st.isEmpty();
    }

}
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 in the code so that this string is correctly 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.

"( [ ] ) "
" < < > > > ( ) {}"
II. A pre-order traversal:
III. A post-order traversal:

PROBLEM 10: *(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. [3pts] 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. [7pts] 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
/\  
14 8
/\ /\  
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
 * @param pq is a minheap containing heap.length-1 elements
 * @param target
 */
int lessCount(int[] pq, int target)
```
/*
 * Returns # elements in subheap rooted/starting at index
 * that are less than target
 * @param pq is a minheap containing heap.length-1 elements
 * in indices [1,heap.length-1]
 * @param index is in the range [1,heap.length-1]
 */
public int lessAux(int[] pq, int index, int target)
{

PROBLEM 11: (Tradeoffs (8 points))
You are given an array of n ints (where n is very large) and are asked to find the largest m of them (where m is much less than n).

A. Design an efficient algorithm to find the largest m elements.
   You can assume the existence of all data structures we discussed in class. You do not have to explain how any of the standard methods (e.g. constructing a heap) work. Be specific, however, about which data structures you are using and how these data structures are interconnected.
   Your algorithm should work well for all values of m and n, from very small to very large.

B. What is the running time of your algorithm? What is it for small m? What is it as m → n (i.e. as m approaches n)?