DUKE UNIVERSITY
Department of Computer Science

Test 2: CompSci 100e

Name (print): _____

Community Standard acknowledgment (signature): _____

	value	grade
Problem 1	14 pts.	
Problem 2	9 pts.	
Problem 3	10 pts.	
Problem 4	19 pts.	
Problem 5	10 pts.	
Problem 6	8 pts.	
TOTAL:	70 pts.	

This test has 13 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts only 75 minutes and there are 70 points on the exam. That means you should spend no more than *1 minute per point*.

You may consult your five (5) sheets, your textbooks, and no other resources. You may not use any computers, calculators, or cell phones. You may refer to any program text supplied in lectures or assignments.

Don't panic. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious.

If you think there is a syntax error or an ambiguity in a problem's specification, then please ask.

In writing code you do not need to worry about specifying the proper `import` statements. Assume that all libraries and packages we've discussed are imported in any code you write.

PROBLEM 1 : (*Short ones (14 points)*)

1. You are creating a boggle word search program and you want to find all valid words where the letters are adjacent. What data structure would be *best* suited to hold the dictionary (i.e. lexicon)?

   A. Queue

   B. Heap

   C. Trie

   D. AVL Tree

   E. Hash Table

2. 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; }`

3. **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. Collisions are resolved using chaining. 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 an *AVL tree* to represent a priority queue. The worst-case big-Oh of `add` (*insert*) and `poll` (*deleteMin*) do not change.

   III. 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.

   IV. Given the preorder and postorder traversals of a binary tree (i.e. printing out all of the elements but not the null nodes), it is possible to reconstruct the original tree.

   V. Given the preorder and inorder traversals of a binary tree, it is possible to reconstruct the original tree.

PROBLEM 2 : (*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 3 : (*Bits (10 points)*)

You would like to implement the set operations for the integers 0-31 using a *BitSet* class. The ith bit is a 1 if and only if i is in the set. For example,

00000000000000000000000000000000 indicates that there are no elements in the set.
10000000000000000000000000001010 indicates that 1, 3, and 31 are in the set.

The second set can be created with the following client code:

```
BitSet s = new BitSet();
s.set(1, true);
s.set(3, true);
s.set(31, true);
```

Below, we have given the constructor and the `set` method for *BitSet*.

```
public class BitSet {
  private int myBits;

  private final static int BITS_PER_INT = 32;

  public class BitSet(int bits)
  {
    // set all bits to 0
    myBits = 0;
  }

  /**
   * Sets the bit at the specified index to the specified value.
   * @param bitIndex a bit index that should be between 0 and 31
   * @param value a boolean value to set.
   */
  public void set(int bitIndex, boolean value)
  {
    if (bitIndex < 0 || bitIndex >= BITS_PER_INT)
      // out of bounds
      return;

    if (value) //
      myBits = myBits | (1 << bitIndex);  // set bit to on
    else
      myBits = myBits & ~(1 << bitIndex); // set bit to off
  }
```

A. Complete the `get` method. In the above example `s.get(3)` should return `true` while `s.get(7)` should return `false`.

```
/**
 * Returns the value of the bit with the specified index. The value
 * is true if the bit with the index bitIndex is currently set in
 * this BitSet; otherwise, the result is false.
 *
 * @param  bitIndex   the bit index.
 * @return the value of the bit with the specified index.
public boolean get(int bitIndex)
{
  // TODO: Complete get
```

B. You would like to implement the union operation for the `BitSet` class. The union operation ($\cup$) says that if $A = (1, 2, 3, 4, 5)$ and $B = (1, 3, 5, 7, 9)$, then $A \cup B = (1, 2, 3, 4, 5, 7, 9)$. Write the method union below.

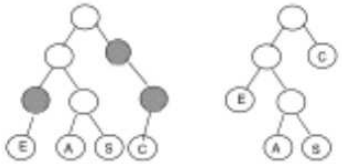
```
/**
 * Performs a union of this bit set with the bit set argument. This bit set is modified so that
 * a bit in it has the  value true if and only if it either already had the
 * value true or the corresponding bit in the bit set argument has the value true.
 *
 * @param set a bit set.
 */
 public void union(BitSet set) {
```

PROBLEM 4 : (*Huffman Trees (19 points)*)

The Huffman compression algorithm uses a tree to encode the codewords, where each node has either two or zero children. Someone has given you a tree that contains some nodes with only one child.

A. Why can such a tree not be created using the Huffman encoding algorithm discussed in class?

B. Write a function called *tighten* that given such an encoding tree will remove those nodes with one child. The diagram below shows a "loose" tree on the left and its tightened equivalent on the right. The three shaded nodes are the ones that were removed.



The definition of a Huffman `TreeNode` is attached to the end of the test.

```
/**
 * remove nodes with one child from Huffman tree
 * @param root is the root of a Huffman tree (may be null)
 * @return tree where nodes with one child are removed
 */
public static TreeNode tighten(TreeNode root)
{
```

C. State the recurrence and the big-Oh for your solution.

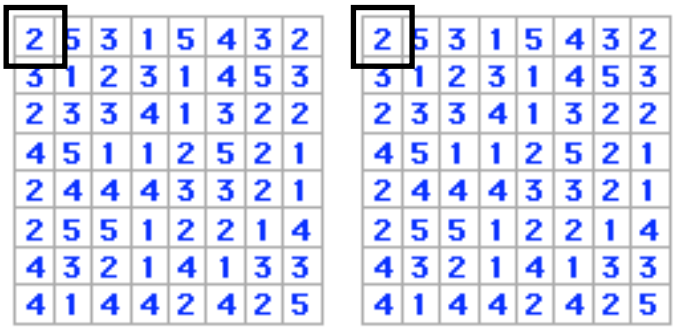
D. You would like to check to make sure the character counts in the tree sum up correctly. That is the the `weight` field of a node should equal the sum of the weights of its children. Write `validWeights` that checks to see if each node is the sum of its children.

```
/**
 *
 * @return true iff each internal node's weights is the sum of its
 * children's weights
 */
public static boolean validWeights(TreeNode root)
{
```

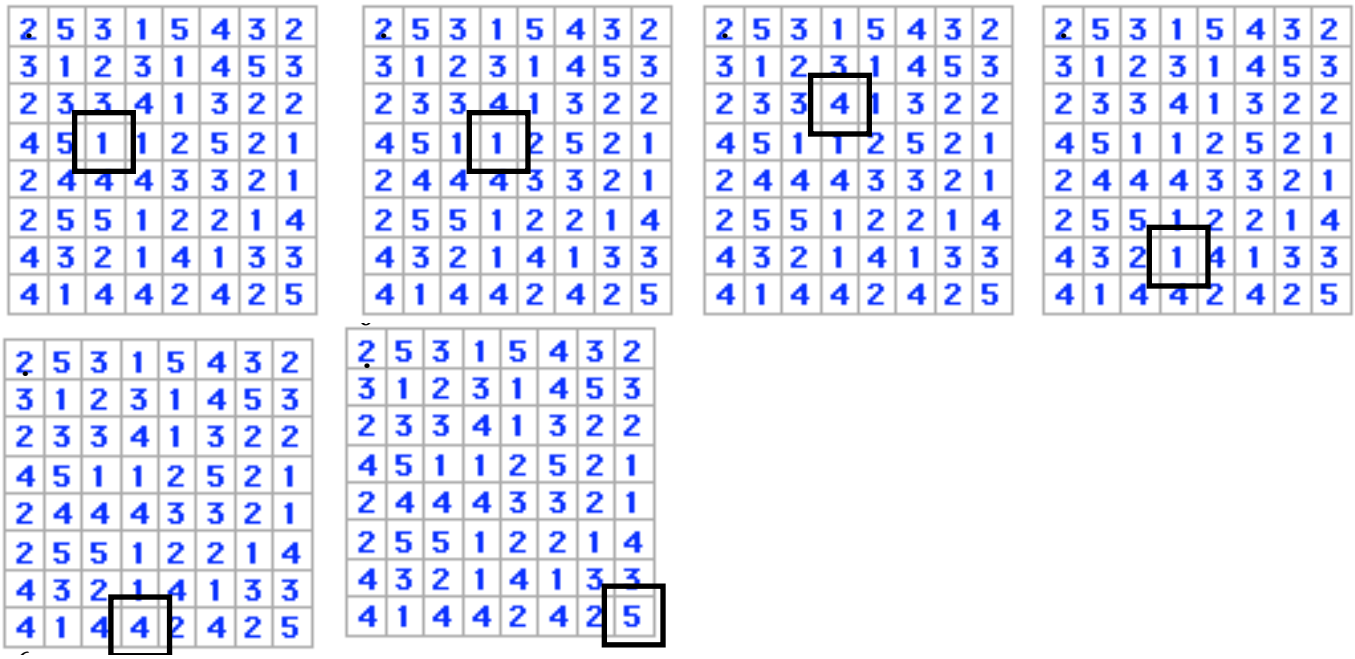
PROBLEM 5 : (*Puzzle Hunt*)

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:



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 horizonatally 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 6 : (*Tradeoffs (8 points)*)
You are given an array of n `int`s (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 do not need to write your solution in Java. Precise English or pseudocode will suffice.

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 $m \to n$ (i.e. as m approaches n)?

Throughout this test, assume that the following classes and methods are available. These classes are taken directly from the material used in class. There should be no methods you have never seen before here.

Definitions

Some common recurrences and their solutions.

`T(n) = T(n/2) + O(1)`	$O(\log n)$
`T(n) = T(n/2) + O(n)`	$O(n)$
`T(n) = 2T(n/2) + O(1)`	$O(n)$
`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
`T(n) = T(n-1) + O(1)`	$O(n)$
`T(n) = T(n-1) + O(n)`	$O(n^2)$

List Node

```
public class Node
{
    String info;
    Node next;
    Node(String s, Node link) {
        info = s;
        next = link;
    }
}
```

Node for Binary Trees

```
    public class TreeNode
    {
       String info;
       TreeNode left;
       TreeNode right;
       TreeNode parent;

       TreeNode (String s, TreeNode lt, TreeNode rt)
                 TreeNode  p)
       {
           info = s;
           left = lt;
           right = rt;
           parent = p;
       }
    }
```

String

```
public class String {
    /* Compares this string to the specified object.
     * The result is true if and only if the argument
     * is not null and is a String object that
     * represents the same sequence of characters */
    public boolean equals(Object anObject)

    /* Returns the index within this string of the
     *  first occurrence of the specified substring.
     * -1 if it does not exist */
    public int indexOf(String str)

    /* Returns the length of this string. */
    int length()

    /* Returns a new string that is a substring of this
       string. Begins at the specified beginIndex and
       extends to the character at index endIndex - 1 */
    String substring(int beginIndex, int endIndex)
 }
```

TreeSet/HashSet

```
public class TreeSet {
    // Constructs a new, empty set
    public TreeSet()

    // Returns an iterator over the elements in
    // this set.  The elements are visited in
    // ascending order.
    public Iterator iterator()

    // Returns the number of elements in this set.
    public int size()

    // Returns true if this set contains o
    public boolean contains(Object o)

    // Adds the specified element to this set
    // if it is not already present.
    public boolean add(Object o)
}
```

Queue

```
public class Queue
{
    /* A new element <code>o</code> is added to the queue */
    public boolean add(Object o)
    /* retrieves and removes head of queue */
    public Object remove()
    // Retrieves, but does not remove, the head of this queue, or r
    public Object peeky()
}
```

Stack

```
public class Stack
{
  /* Pushes an item onto the top of this stack. */
  public Object push(Object o)
  /* Removes the object at the top of this stack and returns that o
  public Object pop()
  /* Tests if this stack is empty */
  public boolean empty()
}
```

PriorityQueue

```
public class PriorityQueue
{
   // Constructs an empty Priority Queue
    public PriorityQueue()
    // Inserts the specified element into
    // this priority queue.
    public boolean add(Object o)
    // Retrieves, but does not remove the head of this queue
    public Object peek()
    // Retrieves and removes the head of this queue
    public Object poll()
    // Returns the number of elements
    public int size()
}
```