COM S 211/ENGRD 211 — April 15, 2003
Prelim 2 — 7:30 PM – 9:00 PM

Information:

Name (clearly *print* last, first, middle): _____

Net ID: _____

CU ID: _____

I have followed the rules of academic integrity on this exam (sign): _____

Instructions:

Failure to follow any instruction may result in a point deduction on your exam:

- Turn off all cell phones, beepers, pagers, and any other devices that will interrupt the exam.
- Remove all calculators, reference sheets, or any other material. This test is closed book.
- Fill out the information at the top of this exam.
- Skim the entire exam before starting any of the problems.
- Read each problem completely before starting it.
- Solve each problem using Java, except where indicated.
- Use only the given code in each problem and follow specifications on whether or not to use the API.
- Write your solutions directly on the test using blue/black pen or pencil. Clearly indicate which problem you are solving. You may write on the back of each sheet. If you need scrap paper, ask a proctor.
- Provide only one statement, expression, value, or comment per blank!
- Do not alter, add, or remove any code that surrounds the blanks and boxes.
- Do not supply multiple answers. If you do so, we will choose which one to grade.
- Follow good style! When possible, keep solutions general, avoid redundant code, use descriptive variables, use named constants, indent substructures, avoid breaking out of loops, and maintain other tenets of programming philosophy.
- Comment each control structure, major variable, method, and class (if used), briefly.
- Do not spend too much time on any single question and budget your time based on the amount of points.
- Do not work on bonus problems until you have thoroughly proofread all required (core-point) problems!
- Figure out any problem yourself before raising your hand so that we can avoid disturbing people in cramped rooms.

Core Points:

1. _____ (15 points) _____

2. _____ (10 points) _____

3. _____ (10 points) _____

4. _____ (20 points) _____

5. _____ (20 points) _____

6. _____ (25 points) _____

Total: _____ /(100 points) _____

Bonus Points:

_____ /(15 points) _____

Problem 1 [15 points] *General Concepts*

Answer the following questions. Be concise and clear. You may use figures in your explanations.

1a [2 points] Distinguish between a *linear* and *hierarchical* data structure.

linear: information connected in a line, no loops, no branching
hierarchical: information connected as a tree, has branching

1b [2 points] What is a *search structure*?

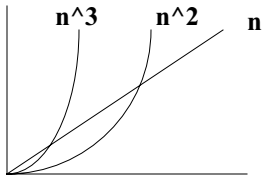
container of data for which searching for an item needs to be quick/efficient

1c [1 point] What is a *stable sort*?

sorting algorithm in which relative positions of equal items stays the same after sorting

1d [2 points] Why is linear search not only $O(n)$, but $O(n^2)$ and $O(n^3)$ as well?

O(n^2) and O(n^3) still bound a linear function



1e [7 points] Determine the approximate running time as a function $T(n)$ for the *body* of the following method. Operations to count are *fetch*, *store*, *operate*, and *return*. Show your work for partial credit.

```
public static int factorial ( int n ) {

   int fact = 1 ;          // 2 (3 is OK: accounts for Java's LHS eval)

   int i = n ;             // 2 (3 is OK)

   while ( i > 1 ) {       // (3n)   3 for ops and (n-1+1) for cond

      fact = fact * i ;    // 4(n-1) (5(n-1) is OK)

      i-- ;                // 4(n-1) (5(n-1) is OK)

   }

   return fact ;           // 2

}
                           // Total: 11n-2 (or 13n-2)
```

1f [1 point] What is the *tightest* asymptotic *time* complexity of the code in Problem 1e? Express your answer in big-O notation. You do not need to justify your answer.

O(n)

Problem 2 [10 points] *Asymptotic Complexity*

Suppose that $f(n) = O(h(n))$ and $g(n) = O(h(n))$. Determine whether or not each of the following relationships is true. If the relationship is true, provide a witness pair to justify your answer. If the relationship is false, provide a counter-example.

2a [4 points] $f(n) + g(n) = O(h(n))$ **true**

```
f(n) <= c1*h(n) for all n > n1
g(n) <= c2*h(n) for all n > n2
```

Let cs=c1+c2 and ns=max(n1,n2) to define witness pair (cs,ns).
Add f1 and f2:
f(n)+g(n) <= (c1+c2)*h(n)
<= cs*h(n)

This relation is true for all n > ns.
So, given our valid witness pair, f(n)+g(n)=O(h(n)).

2b [6 points] $\frac{f(n)}{g(n)} = O(1)$. **false**

how to prove false? provide counter example:

```
f(n) <= c1*h(n) for all n > n1
g(n) <= c2*h(n) for all n > n2
```

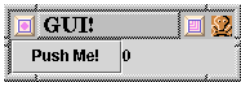
let h(n) = n

Since f(n) = O(h(n)), let f(n)=n
Also, since g(n) = O(h(n)), then we can pick another function, like g(n) = 1, which is also bounded.

f(n)/g(n) = n, which is certainly not bounded by O(1).
So, the assertion that f(n)/g(n) = O(1) is wrong.

Problem 3 [10 points] *GUIs*

Complete the statements in class **`MyGUI`**. This program creates a GUI with a single button and text label, which starts with 0. Each time a user clicks on the button, the label increments by 1, as shown below:



initial state

after user pushes button 3 times

The following page has a copy of the API specifications for every method and constructor that you might need.

```
import javax.swing.*; import java.awt.event.*; import java.awt.*;
public class MyGUI extends JFrame implements ActionListener {
   private int        count;  // count of button pushes
   private Container  CP;     // content pane
   private GridLayout GL;     // grid layout (left is button, right is label)
   private JButton    B;      // button that gets pushed
   private JLabel     JL;     // label that display count

   public MyGUI() {
      setUp()                          ; // set up GUI statics
      CP.setLayout(GL)                 ; // set layout manager
      CP.add(B)                        ; // add button to content pane
      CP.add(JL)                       ; // add label to content pane
      B.addActionListener(this)        ; // register ActionListener for button
   }

   private void setUp() {
      CP = getContentPane();
      B  = new JButton("Push Me!");
      JL = new JLabel(""+count);
      GL = new GridLayout(1,2);
   }

   // If user pushes button, replace updated count in label:
   public void actionPerformed(ActionEvent E) {

      if (E.getSource() == B) // could feasibly skip the "if"

         JL.setText(""+ (++count));


   }

   public static void main(String[] args) {
      MyGUI g = new MyGUI(); g.setTitle("GUI!");
      g.pack(); g.setVisible(true);
      g.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
} // Class MyGUI
```

API Reminders for Problem 3:

From **`javax.swing.JFrame`**:

`JFrame(String title)`
Creates a new, initially invisible **`Frame`** with the specified title.

`public Container getContentPane()`
Returns the **`contentPane`** object for this frame.

`public void setLayout(LayoutManager manager)`
By default the layout of this component may not be set, the layout of its **`contentPane`** should be set instead.
For example: **`thisComponent.getContentPane().setLayout(new GridLayout(1, 2))`**.
An attempt to set the layout of this component will cause an runtime exception to be thrown.

`public Component add(Component comp)`
Appends the specified component to the end of this container.

From **`javax.swing.JButton`**:

`JButton(String text)`
Creates a button with text.

`public void addActionListener(ActionListener l)`
Adds an **`ActionListener`** to the button.

From **`javax.swing.JLabel`**:

`JLabel(String text)`
Create a **`JLabel`** instance with the specified text.

`public void setText(String text)`
Define the single line of text this component will display.

From **`java.awt.event`** and class **`java.awt.ActionEvent`**:

`public Object getSource()`
The object on which the **`Event`** initially occurred. Returns the object on which the **`Event`** initially occurred.

From **`java.awt.event`** and interface **`ActionListener`**:

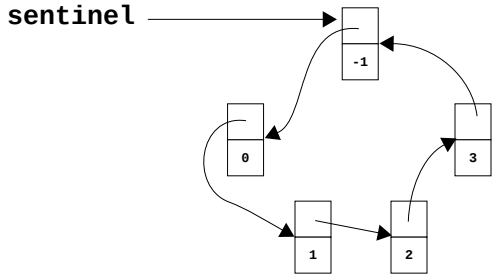
`public void actionPerformed(ActionEvent e)`
Invoked when an action occurs.

From **`java.awt`**, interface **`LayoutManager`**, and class **`GridLayout`**:

`GridLayout(int rows, int cols)`
Creates a grid layout with the specified number of rows and columns.

Problem 4 [20 points] *Inner Classes*, *Iterators*, *Linked Lists*

Background: A ***sentinel node*** can help to manage a linked list. In a circularly linked list, a sentinel node connects the first and last elements of singly-linked list, as shown below. The sentinel provides a way to indicate a beginning and end of the list, which helps for list iteration:



Problem: Complete the inner class **`CircleIterator`** on the following page.

The **`CircleIterator`** returns each node's **`data`** in a circular singly-linked list by moving a **`cursor`**. The **`cursor`** starts at **`sentinel`**'s **`next`** and moves to another node until reaching the node whose **`next`** is **`sentinel`**. So, calling **`CircleIterator`**'s method **`next()`** *never* returns **`sentinel`**'s **`data`**. At each node to which **`cursor`** points, the **`next()`** method returns the node's **`data`** and moves the cursor to the next node. Method **`main`**, below, demonstrates the use of **`CircleIterator`** for the example in the figure.

```
import java.util.*;

public class TestCircle {

   public static void main(String[] args) {

      // Create list with sentinel as first node:
         Circle c = new Circle(-1); // create Circle with sentinel
         Node s = c.sentinel;       // set reference s to sentinel node

      // Add nodes to list:
         Node n0 = new Node(0); Node n1 = new Node(1);
         Node n2 = new Node(2); Node n3 = new Node(3);
         s.next = n0; n0.next = n1; n1.next = n2; n2.next = n3; n3.next = s;

      // Enumerate Circle nodes, but skip sentinel:
         Iterator i = c.new CircleIterator();
         while( i.hasNext() ) System.out.print( i.next() ); // outputs 0123
   }

} // Class TestCircle

class Node {
   public Node next;
   public Object data;
   public Node(int d) { data = new Integer(d); }
   public String toString() { return ""+data; }
} // Class Node
```

```
class Circle {

   public Node sentinel; // ref to sentinel node in circularly linked list

   // Create Circle with sentinel node that contains dummy value:
      Circle(int d) {
         sentinel = new Node(d);    // create sentinel with arbitrary (dummy) value
         sentinel.next = sentinel;  // create default circular list
      }

   // CircleIterator enumerates the elements of the list, which sentinel points to.
   // Iteration starts at sentinel's next and continues to sentinel's previous node:
      public class CircleIterator implements Iterator {

         private Node cursor; // finger into list

            // Create a CircleIterator and set cursor to sentinel's next:
            public CircleIterator( ) {

               cursor = sentinel.next;

            }

            public boolean hasNext( ) {

               return cursor != sentinel;

            }

            public Object next( ) {

               Object result = cursor.data;
               cursor = cursor.next;
               return result;

            }

            // Do not implement Iterator's remove method:
            public void remove( ) { }

      } // Class CircleIterator
} // Class Circle
```

Problem 5 [20 points] *Trees*, *Recursion*

Background: A ***Binary Search Tree*** (BST) is a binary tree composed of nodes such that for each node,

- All nodes in the left subtree have data smaller than the node's data, and
- All nodes in the right subtree have data larger than the node's data.

We are assuming that our BSTs do not contain duplicates.

Problem: Refer to the code below and on the next page. The intrepid programmer Louis Sypher creates a binary tree **`bt`** in **`main`**, using classes **`BinaryNode`** and **`BinaryTree`**. Complete the recursive method **`isBST(BinaryNode n)`** in class **`BinaryTree`** to help Louis determine whether or not the binary tree is a BST. You may not write any helper methods, but you *might* need to use the other methods we have defined in **`BinaryTree`**.

Specifications:

- Assume that **`bt`** refers to a valid binary tree.
- Do not use any interfaces or classes from **`java.util`** in your solution.
- Use **`Comparable`** to compare nodal **`data`**. Remember that **`Comparable`** objects implement method **`compareTo`**.

Reminder: `int compareTo(Object other)`

- returns positive integer if current object > **`other`**
- returns 0 if current object == **`other`**
- returns negative integer if current object < **`other`**

```
public class TestBST {
   public static void main(String[] args) {
      BinaryTree bt = new BinaryTree();   // build binary tree
      /* example code to build the tree not shown */
      System.out.println( bt.isBST() );   // report if bt is a BST
   }
}

class BinaryNode {
   public BinaryNode left;
   public BinaryNode right;
   public Object     data;
   public BinaryNode(Object data) { this.data=data; }
}

class BinaryTree {
   public BinaryNode root; // root of binary tree
   public BinaryTree( ) { root = null; }
   public BinaryTree(Object data) { this.root = new BinaryNode(data); }

   private Object findRightMost(BinaryNode n) {
      if (n==null) return null;
      if (n.right==null) return n.data;
      else return findRightMost(n.right);
   }

   private Object findLeftMost(BinaryNode n) {
      if (n==null) return null;
      if (n.left==null) return n.data;
      else return findLeftMost(n.left);
   }

   // problem continues on next page
```

```
   // Is current tree a BST starting from root?
      public boolean isBST( ) { return isBST( root ) ; }

   // Is binary subtree (starting from node n) a BST?
      private boolean isBST ( BinaryNode n ) {

       if (n == null) return true;
       if (
           isBST(n.left)
             &&
           isBST(n.right)
             &&
           ((n.left == null)  ||
             ((Comparable) (n.data)).compareTo(findRightMost(n.left)) >= 0)
             &&
           ((n.right == null) ||
             ((Comparable) (n.data)).compareTo(findLeftMost(n.right)) <= 0)
          )
             return true;
       return false;

      }

} // Class BinaryTree
```

Problem 6 [25 points] *Algorithms & Recursion*

DIS has devised an inefficient, but interesting way to reverse the elements in an array. As shown below and on the next page, method **`flip`** reverses a general 1-D array of integers via a recursive method called **`flip2`** that has the signature **`flip2(int[] x, int a, int b)`**. Indices **`a`** and **`b`** represent the first and last indices of the input array **`x`**, respectively.

You need to complete the implementation of method **`flip2`**. *Do not use extra memory by creating arrays in* **`flip2`**!

To recursively reverse the array, you must follow this pattern:

- If the array length is zero or one, stop recursing.
- If the array length is greater than one, swap the left and right halves of that array between indices **`a`** and **`b`** (inclusive) without creating a new array. For example, for **`a`**=0 and **`b`**=3, flipping **`{1,2,3,4}`** *once* rearranges the array into **`{3,4,1,2}`** by swapping elements **`3`** and **`4`** with **`1`** and **`2`**. If the array length is odd, swap the elements around the middle element. Continue by flipping both of those halves recursively.

For example, reversing the array **`{2,3,1,4}`** would have this pattern:
`{2,3,1,4}` → **`{1,4,2,3}`** → **`{4,1,3,2}`**

An example of an odd-length array **`{1,2,3,4,5}`** has this pattern:
`{1,2,3,4,5}` → **`{4,5,3,1,2}`** → **`{5,4,3,2,1}`**

[code appears on next page; use the following space to refine your algorithm; Hint: Work out formulas for figuring out the index for each half of the array between **`a`** and **`b`**.]

```
public class ReverseArray {

   public static void main(String[] args) {
      int[] x1 = {2,3,1,4};
      int[] x2 = {1,2,3,4,5};
      print(flip(x1)); // outputs {4,1,3,2}
      print(flip(x2)); // outputs {5,4,3,2,1}
   }

   // Reverse the elements in x and return that array:
      public static int[] flip(int[] x) {
         flip2(x, 0, x.length-1);
         return x;
      }

   // Reverse the elements in x in place, so do not create new arrays:
      private static void flip2( int[]x , int a, int b ) {

          if (a >= b) return;
          else {
             int L = b-a+1;
             int L1 = a;
             int R1 = a+L/2-1;
             int L2 = a+L/2+L%2;
             int R2 = b;
             int adj = R1-L1+1+L%2;

             for (int i = a; i <= R1; i++) {
                int tmp = x[i];
                x[i] = x[i+adj];
                x[i+adj]=tmp;
             }

             flip2(x,L1,R1);
             flip2(x,L2,R2);
          }

      }

   public static void print(int[] x) { /* code not shown */ }
} // Class ReverseArray
```

Bonus: Do not work on these problems until you have *thoroughly* finished all core-point (required) problems!

B0) [0 bonus points] We're curious if anyone remembers the answer to the bonus question from Prelim 1 that asked if you knew the answer to *that* question. What was that answer? **17.2 (I think)**

B1) [1 bonus point] What is the approximate number of words in the English language?
1 million

B2) [1 bonus point] What is the longest word in the English language?
Antidisestablishmentarianism (actually, there are better ones…)

B3) [1 bonus point] What is *Rilly*?
a song written by primary author of Perl In A Nutshell

B4) [1 bonus point] Why is the notation $f(n) \in O(g(n))$ more appropriate than $f(n) = O(g(n))$?
O(g(n)) represents a set of functions to f(n) belongs
To be bounded, f(n) must be inside that set.

B5) [2 bonus points] Explain the *height of a node* in terms of a **`getHeight`** method that might appear in a **`Tree`** class.

To determine the height of a tree in a recursive fashion, each subtree at a node is inspected and the height is determined. Since we feel comfortable finding the height of a subtree whose root is a node in the main tree, then we are really finding the height of a node.

B6) [3 bonus points] What do you think is the asymptotic time complexity of the program in Problem 6? Informally justify your answer.

O(nlogn): cutting array in half each step (log n) and swapping elements in a linear fashion (n)

B7) [6 bonus points] Write an iterative version of **`flip2`** from Problem 6 on the back of this page.
left as an exercise…