CIS 110 — Introduction to Computer Programming

7 May 2012 — Practice Final

Name: ____________________

Recitation # (e.g. 201): ____________________

Pennkey (e.g. bjbrown): ____________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

____________________ ________________

Signature Date

Scores:

1		1
2		3
3		8
4		6
5		15
6		10
7		8
8		15
9		34
Total:		100

CIS 110 Final Instructions

- **This practice final is designed to give a range of questions that cover material since the midterm. You should review the midterm and practice midterm as well, since questions similar to those may appear on the final as well. However you can expect the questions on the final to focus more heavily on the second half of the semester.**
- You have 110 minutes to finish this exam. Time will begin when called by a proctor and end precisely 110 minutes after that time. If you continue writing after the time is called, you will receive a zero for the exam.
- This exam is *closed-book, closed-notes, and closed-computational devices.* Except where noted, you can assume that code included in the question is correct and use it as a reference for Java syntax.
- This exam is long. If you get stuck part way through a problem, it may be to your advantage to go on to another problem and come back later if you have time.
- When writing code, the only abbreviations you may use are for `System.out.println`, `System.out.print`, and `System.out.printf` as follows:

  System.out.println $\longrightarrow$ S.O.PLN
  System.out.print $\longrightarrow$ S.O.P
  System.out.printf $\longrightarrow$ S.O.PF

  Otherwise all code must be written out as normal, including all curly braces and semicolons.
- Please do not separate the pages of the exam. If a page becomes loose, write your name on it and use the provided staplers to reattach the sheet when you turn in your exam so that we don't lose it.
- If you require extra paper, please use the backs of the exam pages or the extra sheet(s) of paper provided at the end of the exam. Clearly indicate on the question page where the graders can find the remainder of your work (e.g. "back of page" or "on extra sheet"). Staple an extra sheets you use to the back of your exam when you turn it in using the provided staplers.
- If you have any questions, please raise your hand and an exam proctor will come to answer them.
- When you turn in your exam, you will be required to show ID. If you forgot to bring your ID, please talk to an exam proctor immediately.

Good luck, have fun!

TOY Reference Card

```
INSTRUCTION FORMATS

             | . . . . | . . . . | . . . . | . . . .|
  Format 1:  | opcode  |    d    |    s    |    t   |  (0-6, A-B)
  Format 2:  | opcode  |    d    |       addr       |  (7-9, C-F)


ARITHMETIC and LOGICAL operations
    1: add              R[d] <- R[s] +  R[t]
    2: subtract         R[d] <- R[s] -  R[t]
    3: and              R[d] <- R[s] &  R[t]
    4: xor              R[d] <- R[s] ^  R[t]
    5: shift left       R[d] <- R[s] << R[t]
    6: shift right      R[d] <- R[s] >> R[t]

TRANSFER between registers and memory
    7: load address     R[d] <- addr
    8: load             R[d] <- mem[addr]
    9: store            mem[addr] <- R[d]
    A: load indirect    R[d] <- mem[R[t]]
    B: store indirect   mem[R[t]] <- R[d]

CONTROL
    0: halt             halt
    C: branch zero      if (R[d] == 0) pc <- addr
    D: branch positive  if (R[d] >  0) pc <- addr
    E: jump register    pc <- R[d]
    F: jump and link    R[d] <- pc; pc <- addr


Register 0 always reads 0.
Loads from mem[FF] come from stdin.
Stores to mem[FF] go to stdout.
```

Miscellaneous

1. (1 points)

(a) Write your name, recitation number, and PennKey (username) on the front of the exam.

(b) Sign the certification that you comply with the Penn Academic Integrity Code

Number Systems

2. (3 points) The following hexadecimal numbers represent 16-bit 2's complement numbers. Put them in order from lowest (most negative) to highest (most positive).

`0FFF      F000      1ABC      8888      FFFF`

lowest ____________

____

____

____

highest ____________

TOY

3. (8 points) Consider what happens when the following TOY program is executed by pressing RUN with the program counter set to `10`:

```
10: 8110   R[1] <- Mem[10]
11: 7203   R[2] <- 03
12: 1212   R[2] <- R[1] + R[2]
13: 9214   Mem[14] <- R[2]
14: 0000   Halt
15: 0000   Halt
```

(a) What is the value (in hex) of `R[1]` after the instruction at location `10` completes?

(b) What is the value of `R[2]` after the instruction at location `12` completes?

(c) What is the value of `Mem[14]` after the instruction at location `13` completes?

(d) What is the value of `R[1]` when the program halts?

Sorting and Analysis

4. (6 points) You are provided a method called `pennSort` that takes an array of `int`s, and sorts it in ascending order using one of the algorithms we covered this semester. You called this method with some sample arrays and recorded the execution times in this table:

Array size	Time (sec)
50000	5
100000	22
200000	90
800000	1451

(a) How long would you expect it to take to sort an array of 1.5 million items? Circle your answer:

   i. Under half an hour
   ii. 1 − 3 hours
   iii. 4 − 6 hours
   iv. ∼ half a day

(b) Based on your timing analysis, which sort could `pennSort()` be? Circle your answer:

   i. Insertion sort
   ii. Mergesort
   iii. Insertion sort or mergesort
   iv. Neither insertion sort nor mergesort

(c) Given an unsorted array of size `N`, what is the order of growth you would expect to find in the execution time of the *second* (and ***only** the second*) call to `pennSort()` below? Circle your answer.

```
pennSort(randomUnsortedArray);
pennSort(randomUnsortedArray);
```

   i. Constant
   ii. Linear
   iii. Logarithmic
   iv. Quadratic

Linked Lists

5. (15 points) Consider the following class, which implements a linked list data structure:

```
class Node {
    public int key;
    public Node next;
    public Node(int key) { this.key = key; this.next = null; }
}

public class LinkedList {
    private Node foo;
    private Node bar;
    public LinkedList() { foo = null; bar = null; }
    // instance methods A(), B(), C(), D(), E()
}
```

Answer the questions on the following page about these five instance methods for this class

```
public void A(int key) {
    Node x = new Node(key);
    if (foo == null) foo = x;
    else bar.next = x;
    bar = x;
}

public int B() { return bar.key; }

public void C() {
    Node t = foo;
    while (t != null) {
        System.out.print(t.key + " ");
        t = t.next;
    }
    System.out.println();
}
```

```
public void D() {
    if (foo == bar) {
        foo = null;
        bar = null;
    } else {
        Node t = foo;
        while (t.next != bar) t = t.next;
        t.next = null;
        bar = t;
    }
}

public LinkedList E() {
    LinkedList ll = new LinkedList();
    while (foo != null) {
        ll.A(this.B());
        D();
    }

    return ll;
}
```

(a) Match each intance method with one of the descriptions below by writing its letter in the blank to the left of the corresponding description. Since there are twice as many descriptions as instance methods, you must leave five options blank.

_____ Prints the list in order

_____ Prints the list in reverse order

_____ Adds a key at the beginning of the list

_____ Adds a key at the end of the list

_____ Copies the list in order

_____ Copies the list in reverse order

_____ Returns the key in the first element of the list

_____ Returns the key in the last element of the list

_____ Removes the first element of the list

_____ Removes the last element of the list

(b) In the space provided, write the output produced by the following code:

```
LinkedList ll = new LinkedList();
ll.A(1); ll.A(7); ll.A(4); ll.A(9);
ll.C();
ll.D();
ll.E().C();
```

first line of output: ______________________________

second line of output: ______________________________

Analysis of Algorithms

6. (10 points)

(a) Professor Quring proposes the following recursive algrotihms for sorting an array of real numbers in ascending order:

```
public static void sort(double[] a) {
    sort(a, 0, a.length - 1);
}

public static void sort(double[] a, int lo, int hi) {
    if (lo >= hi) return;
    sort(a, lo, hi - i);
    if (a[hi] < a[hi - 1]) {
        double temp = a[hi];
        a[hi] = a[hi - i];
        a[hi - i] = temp;
        sort(a, lo, hi - 1);
    }
}
```

To analyze its running time, the professor performs the following experiments on arrays of random numbers:

N	time (sec)
1,000	0.75
2,000	6.00
3,000	20.25
4,000	48.00
5,000	93.75
6,000	162,00

Predict (within 10%) holw long it will take (in seconds) to sort 10,000 random numbers:

(b) Give two compelling reasons why you might prefer to implement an algorithm that run in time proportion to N^2 rather than $N \log N$.

- Reason 1:

- Reason 2:

Divide and Concquer

7. (8 points) Consider the following class:

```
public calss Foo {
    public static String bar(String s) {
        if (s.length() < 2) return s;
        int m = s.length() / 2;
        String lh = bar(s.substring(0, m));
        String rh = bar(s.substring(m, s.length()));
        s = rh + lh;
        System.out.println(s);
        return s;
    }

    public static void main(String[ args) {
        String s = args[0];
        s = bar(s);
    }
}
```

Recall that `s.substring(i, j)` returns the substring of `s` from indicies `i` to $j-1$. For example, if `s` is the string `"stressed"`, then `s.substring(0, 6)` is `"stress"`.

(a) Give the one line of output produced by the command

```
% java Foo ab
```

(b) Give the seven lines of output produced by the command

```
% java Foo stressed
```

Queue and Stack

8. (15 points) Given the following APIs for a queue and a stack of integers:

```
public class QueueOfInts
-------------------------------------------------------------
          QueueOfInts()         creates an empty queue
  boolean isEmpty()             true if queue is empty
     void enqueue(int item)     enqueues one int
      int dequeue()             dequeues on int

public class StackOfInts
-------------------------------------------------------------
          StackOfInts()         creates an empty stack
  boolean isEmpty()             true if stack is empty
     void push(int item)        pushes one int
      int pop()                 pops one int
```

Write a static function `reverse` that reverses a `QueueOfInts` using an intermediate `StackOfInts`. `reverse` should not return anything, and should not crash if its argument is `null`.

Graphs

9. (34 points) In the Traveling Salesman homework, we assumed that the distance between any pair of cities was the Euclidean, or straight-line, distance between them. But in reality, cities are connected by roads that may be indirect; there may not even be direct roads between certain cities. Moreover, cities may be connected by one-way streets so that the distance from city A to city B is different than the distance from B to A. We can represent this situation using a *graph* data type that stores not only the cities, but also the lengths of the paths that connect them. In the standard graph terminology, the cities are called *vertices*, and the paths are called *edges*. Computing a tour on a graph is no different than on the set of points in the homework except that the graph data type will need to include a method

```
public double distance(Vertex a, Vertex b)
```

that takes the place of the `Point.distanceTo()` method.

The question naturally arises how the edges should be stored. Two of the most common approaches are the *adjacency matrix*, and the *adjacency list*. An adjacency matrix is a $n \times n$ array (assuming there are n vertices), where entry (i, j) stores the distance from vertex i to vertex j. If there is no edge from vertex i to vertex j, then entry (i, j) in the array is set to `Double.POSITIVE_INFINITY`. The alternative, an adjacency list, stores an array of edges. Each edge records the starting and ending vertices, and the distance. If no edge exists that goes from i to j then there is no way to travel directly from i to j. This approach is slightly more complex than an adjacency matrix, but avoids storing lots of `Double.POSITIVE_INFINITY`'s if there are relatively few edges.

The nice thing is that we can implement TSP for both types of graph data structure by using an interface, because `Tour` only needs a method to compute the distance between cities. We will therefore define the following `Vertex` class and `GraphDistance` interface, and modify the `Tour` API as follows:

```
public class Vertex {
    public Point p;
    public int  id;

    public Vertex(double x, double y, int id) {
        p = new Point(x, y);;

        // record the number of this vertex
        // (0, 1, 2, etc.)
        this.id = id;
    }
}

public Interface GraphDistance {
    // Return the distance between a and b according
    //   to the edges of the graph.
    // If there is no path from vertex a to vertex b,
    //   return Double.POSITIVE_INFINITY
    public double distance(Vertex a, Vertex b);
}

public class Tour {
    // create a Tour for the graph g.  g can be any
    // type that implements the GraphDistance interface
    // since we only need the distance() method to compute tours
    public Tour(GraphDistance g) { ... }

    public void show() { ... }
    public void draw() { ... }
    public int  size() { ... }
    public double distance() { ... }
    public void insertNearest(Vertex v) { ... }
    public void insertSmallest(Vertex v) { ... }
}
```

Complete the `distance()` and `readGraph()` methods for the `AdjacencyMatrixGraph` data type below. `distance()` should use the `id` instance variable in the two vertices to figure out which entries of the adjacency matrix to access. It does not need to do any error checking. `readGraph()` should read two integers n and e from standard input using the `StdIn` library; n gives the number of vertices in the graph, and e gives the number of edges. The next n lines contain the x and y coordinates of the vertices, in order. Following that are e lines, each containing two integers and a double which are the start and end vertices of an edge, and the distance along that edge. You may assume the input will always be valid and that at most one edge will be listed from vertex i to vertex j. However it is possible there will be no edge from i to j, in which case that distance should be set to `Double.POSITIVE_INFINITY`. Also, remember that the distances from i to j and from j to i are not necessarily the same; if there is an edge from i to j but not from j to i, then the distance from j to i is still `Double.POSITIVE_INFINITY`. For example:

```
3 4
0.2 0.3
0.1 0.7
0.4 0.9
0 1 5.0
1 0 3.6
1 2 4.2
2 0 1.3
```

is a graph with three vertices at locations $(0.2, 0.3)$, $(0.1, 0.7)$, and $(0.4, 0.9)$. The distance from vertex 0 to vertex 1 is 5, the distance in the other direction (from vertex 1 to vertex 0) is 3.6, the distance from vertex 1 to vertex 2 is 4.2, and the distance from vertex 2 to vertex 0 is 1.3. The remaining distances (from 0 to 2, from 2 to 1, and from any vertex to itself) are all `Double.POSITIVE_INFINITY`.

```
public class AdjacencyMatrixGraph implements GraphDistance {
    private double[][] adjacencyMatrix;
    private Vertex[]   vertices;

    // create a graph with n vertices
    public AdjacencyMatrixGraph(int n) {
      adjacencyMatrix = new double[n][n];
      vertices = new Vertex[n];
    }

    public static void main(String[] args) {
        // read in the graph
        AdjacencyMatrixGraph g = readGraph();

        // compute a Tour
        Tour t = new Tour(g);
        for (int i = 0; i < g.vertices.length; i++)
            t.insertSmallest(g.vertices[i]);
        t.show();
    }

    // return the distance between vertices a and b
    // or Double.POSITIVE_INFINITY if there is no
    // edge between them
    public double distance(Vertex a, Vertex b) {
        // COMPLETE THIS










    }
```

Complete the `readGraph()` method on the next page...

```
    public static AdjacencyMatrixGraph readGraph() {
        int n = StdIn.readInt(); // get number of vertices
        int e = StdIn.readInt(); // get number of edges
        AdjacencyMatrixGraph g = new AdjacencyMatrixGraph(n);

        // read in the vertices; set the id of the first vertex
        // to 0, of the second vertex to 1, etc.
        for (int i = 0; i < n; i++) {
            // COMPLETE THIS











        }

        // COMPLETE THIS
        // read in the edges and set the distances in the adjacency
        // matrix accordingly, or to Double.POSITIVE_INFINITY for any
        // edges that don't exist

















    }
}
```

Postscript (extra paper)