CSE 332 Final Exam – 03/13/2012

Name _______________________________________

Wait to turn the page until after everyone is told to begin.

Write your full name above. Every other page shares a unique identifier with this one.

Do not write any confidential information on this page.

There are 9 questions worth a total of 90 points. Budget your time to maximize the points you can earn in the allotted 110 minutes. Keep answers brief and to the point.

The exam is closed book and closed notes.

Please keep everything related to a question on the page allocated for that question.
This ensures the questions can be separated from each other for efficient grading.

Two pages of scrap paper are also provided at the end of the exam. If you use these pages, either as scrap or because you need the additional space for a problem, be sure to indicate on which problem you are working.

This page is obviously for instructor use only.

Score ________________ / 90

1.	______ / 10	Radix Sort
2.	______ / 10	Graph Representation
3.	______ / 10	Topological Sort
4.	______ / 10	Single-Source Shortest Paths
5.	______ / 10	Minimum Spanning Tree
6.	______ / 10	Work and Span
7.	______ / 10	MoveToFrontList Concurrency
8.	______ / 10	Heap Concurrency
9.	______ / 10	Fork/Join Pseudocode

1) 10 points : Radix Sort

Perform a radix sort of this list of numbers, using a radix of 10, into ascending order:

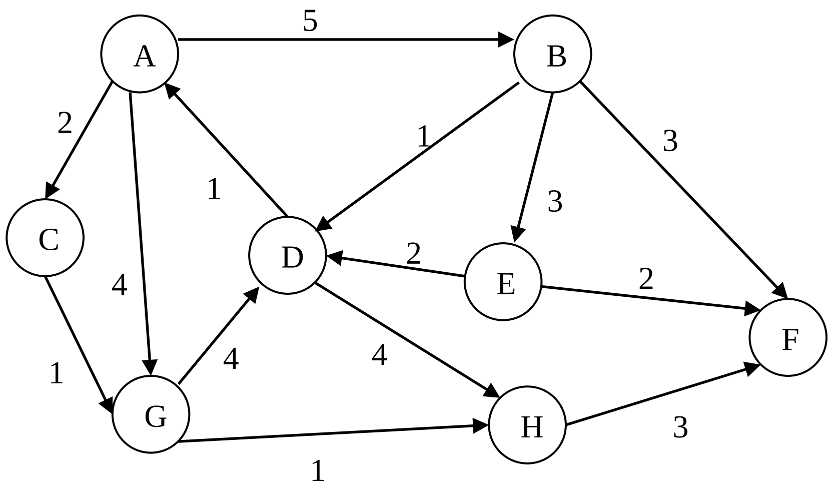
```
620  696  298  395  568  971  29   41   531  21
```

Show the bin/bucket sort conducted in each pass of the radix sort (i.e., as a table).

Write and circle the order of the numbers after each pass of the radix sort.

2) 10 points : Graph Representation

Consider the following directed, weighted graph:



a) Draw an adjacency matrix representation of the above graph.

b) Provide an appropriately tight O (Big-Oh) bound on the time for:

For a given vertex pair (v_1, v_2), testing whether there is an edge from v_1 to v_2:

Computing the in-degree of a given vertex:

Enumerating the vertices adjacent to a given vertex:

c) Draw an adjacency list representation of the above graph.

d) Provide an appropriately tight O (Big-Oh) bound on the time for:

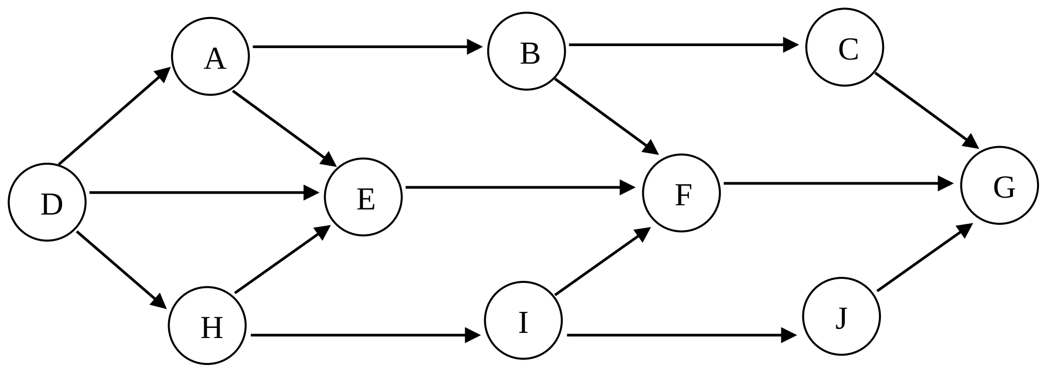
For a given vertex pair (v_1, v_2), testing whether there is an edge from v_1 to v_2:

Computing the in-degree of a given vertex:

Enumerating the vertices adjacent to a given vertex:

3) 10 Points : Topological Sort

Consider the following directed graph:



You will perform two topological sorts on this graph.

In each sort, maintain a "bag" of "pending" vertices. When the processing of a vertex creates more than one new "pending" vertex, add the new "pending" vertices to the "bag" in alphabetical order (e.g., `push(x), push(y), push(z)`).

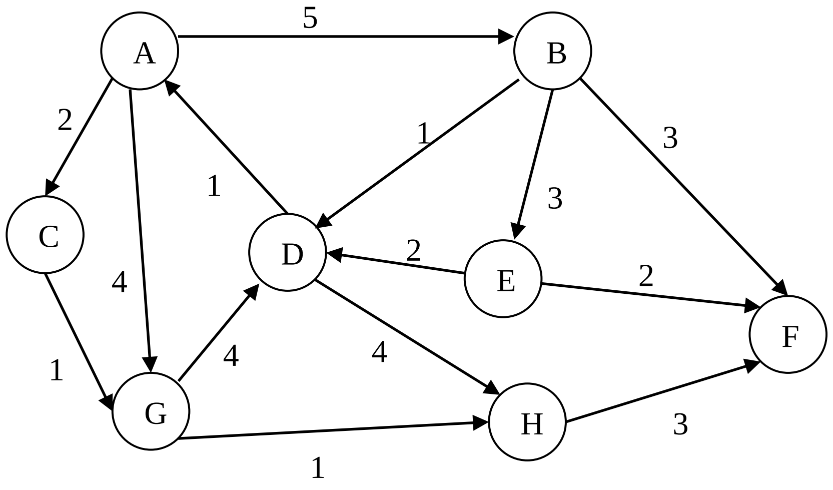
For each topological sort, write and circle the order the graph's vertices are processed. Show your work to allow partial credit (e.g., show adding and removing from the "bag").

a) Perform a topological sort using a **queue** to maintain the set of "pending" vertices:

b) Perform a topological sort using a **stack** to maintain the set of "pending" vertices:

4) 10 points : Single-Source Shortest Paths

Consider the following directed, weighted graph:



a) Step through Dijkstra's algorithm to calculate the single-source shortest paths from vertex *A* to every other vertex. Show your steps in the table below. Cross out old values and write in new ones, from left to right in each cell, as the algorithm proceeds. Also list the vertices in the order which Dijkstra's algorithm marks them known:

Order vertices marked as known: ___ ___ ___ ___ ___ ___ ___ ___

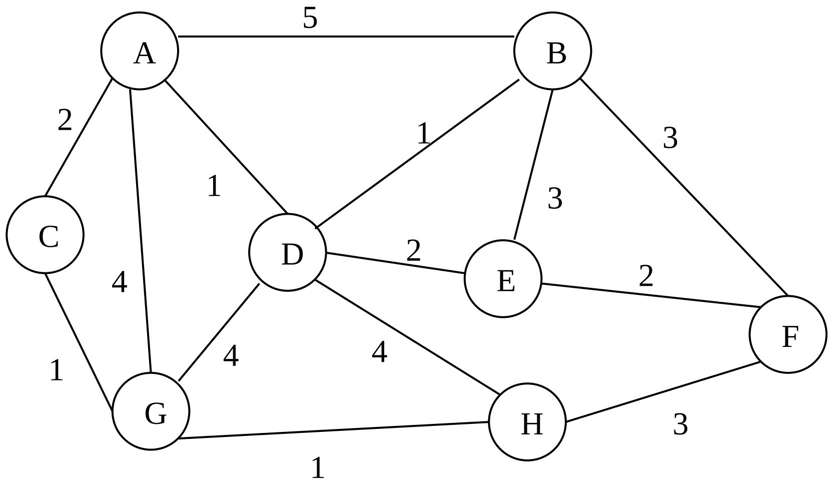
Vertex	Known	Distance	Path
A			
B			
C			
D			
E			
F			
G			
H			

b) What is the lowest-cost path from *A* to *F* in the graph, as computed above?

c) To guarantee correctness of Dijkstra's algorithm, all edge costs must be non-negative. Imagine the edge from A to B had cost -3. Why would this make it impossible for any algorithm to provide a correct answer for single-source shortest paths?

5) 10 points : Minimum Spanning Tree

Consider the following undirected, weighted graph:



a) Step through Prim's algorithm to calculate a minimum spanning tree, starting from vertex *A*. Show your steps in the table below. Cross out old values and write in new ones, from left to right in each cell, as the algorithm proceeds. Also list the vertices in the order which Prim's algorithm marks them known:

Order vertices marked as known: ___ ___ ___ ___ ___ ___ ___ ___

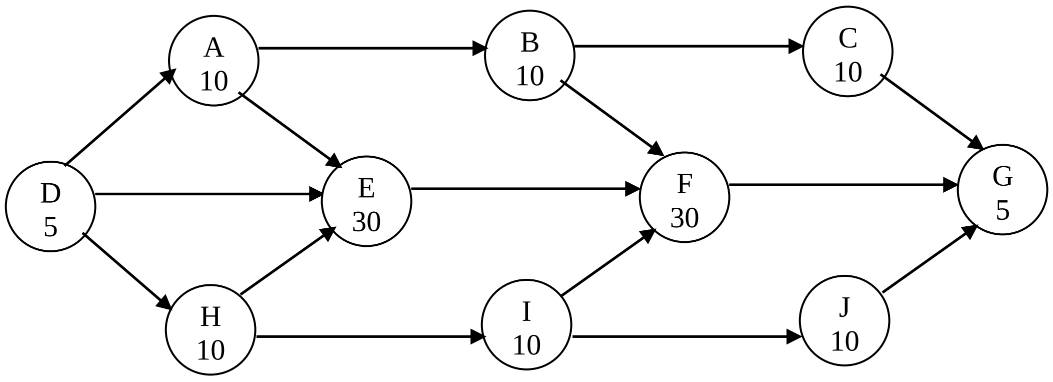
Vertex	Known	Distance	Path
A			
B			
C			
D			
E			
F			
G			
H			

b) What are the edges in the minimum spanning tree, as computed above?

6) 10 Points : Work and Span

Consider the following directed acyclic graph representing the dependencies in a parallel computation implemented using a fork / join technique.

Each vertex is annotated with the cost of performing its work.



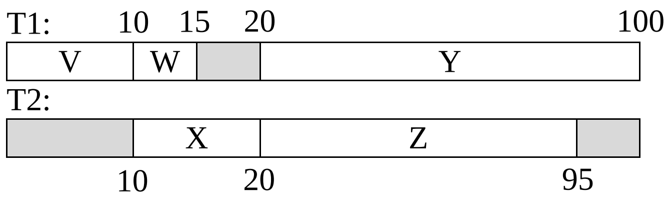
a) What is the work of this computation (i.e., a number)?

b) More generally, what is the work of a computation represented in this manner (i.e., described in terms of the graph and the cost of each vertex)?

c) What is the span of this computation (i.e., a number)?

d) More generally, what is the span of a computation represented in this manner (i.e., described in terms of the graph and the cost of each vertex)?

e) Assume **two threads** are executing a computation. We can illustrate the parallel work of multiple threads by drawing timelines of the work they execute. For example:



This pair of timelines illustrates a hypothetical computation in which:
T1: V for 10 units, W for 5 units, idle for 5 units, Y for 80 units
T2: idle for 10 units, X for 10 units, Z for 75 units, idle for 5 units

Draw a timeline using **two threads** to execute the graph from the previous page as quickly as possible. Be sure your timeline illustrates the start and stop time of each task on each thread.

f) Would additional threads be able to perform the computation more quickly? Why?

7) 10 points : MoveToFrontList Concurrency

Consider this pseudocode for a MoveToFrontList, which is correct in a sequential context. It does not map keys to data items, but instead just tests whether a key is in the list.

```
01: class Node {
02:   Key  key;
03:   Node next;
04:
05:   Node(...) { // Constructor that stores these 2 fields }
06: }
07:
08: class MoveToFrontList {
09:   Node front = null;
10:
11:   void insert(Key key) {
12:     front = new Node(key, front);
13:   }
14:
15:   Boolean contains(Key find) {
16:     if(front == null) { return false; }
17:     if(front.key == find) { return true; }
18:
19:     Node prev = front;
20:     Node current = front.next;
21:     while(current != null) {
22:       if(current.key == find) {
23:         prev.next = current.next;
24:         current.next = front;
25:         front = current;
26:         return true;
27:       }
28:       current = current.next;
29:     }
30:     return false;
31:   }
32: }
```

We have numbered the lines of code so that you can reference them in your answers. Please do this, as it will be faster and will keep your answer more concise.

In describing an interleaving, you might write:

Thread 1 runs `contains(…)`, *stopping between lines 16 and 17.*

In describing a modification of the code, you might write:

Insert additional code after line 9:

```
09a:   Node middle = null;
09b:   Node back = null;
```

Replace line 28:

```
30:     return true;
```

Now consider using our MoveToFrontList in a multi-threaded context:

a) Describe an interleaving of `insert("a")` and `contains("b")` that results in the `insert("a")` being "missed" (i.e., `contains("a")` will return `false`).

b) Describe an interleaving of `insert("a")` and `insert("b")` that results in the `insert("a")` being "missed" (i.e., `contains("a")` will return `false`).

c) Describe an interleaving of `contains("a")` and `contains("a")` that results in them returning different values (i.e., one returns `true` and one returns `false`).

d) Using any of the mutual exclusion mechanisms discussed in lecture (including those unique to Java), describe how to fix this class so that it is correct in concurrent usage. Your only concern is correctness (i.e., performance is not a concern).

8) 10 points : Heap Concurrency

You and a partner are implementing an array-based binary heap. Your partner wants to use fine-grained locking to simultaneously allow multiple concurrent operations in the heap. They propose the following strategy for implementing the locking:

1) Guard each location in the array with a lock (i.e., guard each node in the heap with a lock). Always obtain the lock before reading from or writing to the array location (i.e., the node).
2) Implement `percolateUp` and `percolateDown` such that they lock nodes in the course of the percolation. Before comparing the keys of two nodes, for example, they will lock both nodes. To ensure nothing else is reading or manipulating the portion of the heap affected by percolation, they will hold locks they obtain until the percolation completes.

Your partner claims this is a good strategy and that you can work out the details in the course of the implementation. But you already see two major problems.

a) As described, this approach includes a data race. A critical variable for implementing the heap's `insert` and `deleteMin` operations is unguarded. What is that critical variable? Give an example of a bad behavior might result from it being unprotected.

b) Why will it be extremely difficult to guard the variable from (a) while also preserving your partner's desire to allow multiple concurrent operations in the heap?

c) In additional to potential race conditions, the proposed approach has another serious concurrency problem. What is the name for that problem? How could the problem occur with this strategy for implementing the locking?

9) 10 points : Fork/Join Pseudocode

In an internship at a major mobile phone network provider, you are tasked with writing a parallel program to generate personalized letters informing customers they will be automatically migrated to new data plans for "improved service".

Another intern has provided methods:

```
   Boolean isProfitableToMigrate(Customer c)
   Letter generateMigrationLetter(Customer c)
```

Given a large array of `Customer` objects, your job is to (1) determine which should be migrated, (2) generate a letter for each, and (3) determine the total number of migrations.

You immediately recognize this can be computed as a set of maps and reductions:

1) A map given input `Customer[] customers`
   that outputs `Boolean[] migrate`
   where each entry contains a Boolean for whether a customer will be migrated
2) A map given `Customer[] customers` and `Boolean[] migrate`
   that outputs `Letter[] letters`
   where each entry contains the letter for the customer at the same array index
3) A reduction given `Boolean[] migrate`
   that outputs the number of accounts that will be migrated (i.e., as an integer)

Instead of three separate computations, you decide to do everything in a single ForkJoin computation (as you hope the computational savings will give you time to look for a job with a better company). The computation is invoked by this pseudocode:

```
class InternProject {
  static final ForkJoinPool fjPool = new ForkJoinPool();

  static Boolean isProfitableToMigrate(Customer c) { ... }
  static Letter generateLetter(Customer c) { ... }

  InternResult doEvilInParallel(Customer[] customers) {
    InternTask task = new InternTask(
      customers,
      new Boolean[customers.length],
      new Letter[customers.length],
      0, customers.length
    );

    return fjPool.invoke(task);
  }
}
```

a) Provide pseudocode for the class `InternTask`. We provide its member declarations and its constructor. You just need to implement the `compute()` method. Do not use a sequential cutoff: the base case should process a single `Customer`. Your implementation should perform the computation in *O*(n) work and *O*(log n) span.

```
class InternResult {
  Customer[] customers;
  Boolean[]  migrate;
  Letter[]   letters;
  int        numMigrate;

  InternResult(...) { // Constructor that stores these 4 fields }
}

class InternTask extends RecursiveTask<InternResult> {
  Customer[] customers;
  Boolean[]  migrate;
  Letter[]   letters;
  int        low;
  int        high;

  InternTask(...) { // Constructor that stores these 5 fields }

  InternResult compute() {
    }
```

```
    }
}
```

Extra Credit) 2 Points for the Best Response in the Class : Make a Funny

Write or draw something funny. Any G-rated joke will be considered. Possible lead-ins include:

Why did the professor go to Spokane?	*So I was coding P2 at 3am, when …*
A stack and a queue walk into a bar …	*Your code is so parallel, it …*

This page provided as scrap. Please indicate on which problem you are working.

This page provided as scrap. Please indicate on which problem you are working.