This exam has 13 questions worth a total of 100 points. You have 180 minutes.

Instructions. This exam is preprocessed by computer. Write neatly, legibly, and darkly. Put all answers (and nothing else) inside the designated spaces. *Fill in* bubbles and checkboxes completely: ● and ■. To change an answer, erase it completely and redo.

Resources. The exam is closed book, except that you are allowed to use a one page reference sheet (8.5-by-11 paper, both sides, in your own handwriting). No electronic devices are permitted.

Honor Code. This exam is governed by Princeton's Honor Code. Discussing the contents of this exam before the solutions are posted is a violation of the Honor Code.

Please complete the following information now.

Name:

NetID:

Exam room: ◯ McCosh 50   ◯ McCosh 60   ◯ Other

Precept:

P01	P02	P03	P04	P05	P06	P07	P08
◯	◯	◯	◯	◯	◯	◯	◯

*"I pledge my honor that I will not violate the Honor Code during this examination."*

Signature

1. **Initialization. (1 point)**

   In the spaces provided on the front of the exam, write your name and NetID; fill in the bubble
   for your exam room and the precept in which you are officially registered; write and sign the
   Honor Code pledge.

2. **Graph search algorithms. (12 points)**

   Run *depth-first search* and *breadth-first search* on the following digraph, starting from vertex 0.
   Assume the adjacency lists are in sorted order: for example, when iterating over the edges
   leaving vertex 4, consider the edge 4→0 before either 4→2, 4→5, 4→6 or 4→8.
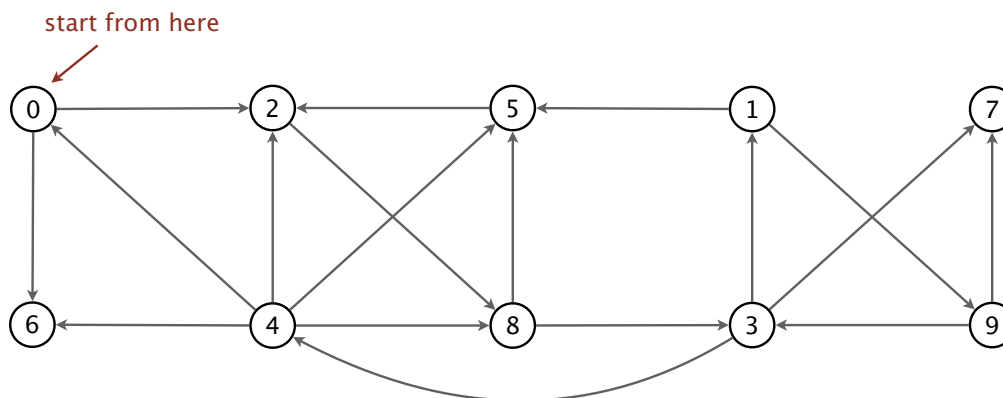
   (a) List the 10 vertices in the order they are *removed from the queue* during the execution
       of BFS.

       0
       ── ── ── ── ── ── ── ── ── ──

   (b) List the 10 vertices in *DFS preorder*.

       0
       ── ── ── ── ── ── ── ── ── ──

(c) List the 10 vertices in *DFS postorder*.

0

_____   _____   _____   _____   _____   _____   _____   _____   _____   _____

(d) Is the *reverse* of the DFS postorder in part (c) a *topological order* for this digraph?

◯   ◯

yes   *no*

3. **Minimum spanning trees. (10 points)**

   Consider the following edge-weighted graph.

start Prim from here

(a) List the weights of the MST edges in the order that *Kruskal's algorithm* adds them to the MST.

(b) List the weights of the MST edges in the order that *Prim's algorithm* adds them to the MST. Start Prim's algorithm from vertex s.

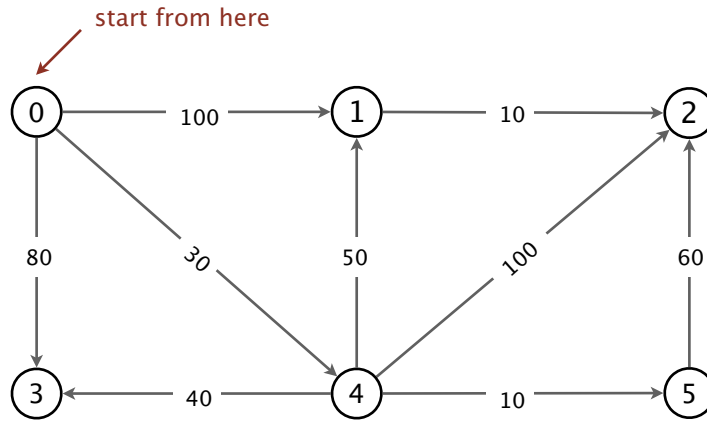|						

4. **Shortest paths. (12 points)**

Consider the following edge-weighted digraph G:

start from here

(a) List the 6 vertices in the order they are *removed from the queue* during *Dijkstra's algorithm* with source vertex $s = 0$.

0

—— —— —— —— —— ——

(b) Consider running the *Bellman–Ford algorithm* on G, with source vertex $s = 0$. Assume that, within a pass, the edges are relaxed in sorted order:

$$0 \rightarrow 1, \ 0 \rightarrow 3, \ 0 \rightarrow 4, \ 1 \rightarrow 2, \ 4 \rightarrow 1, \ 4 \rightarrow 2, \ 4 \rightarrow 3, \ 4 \rightarrow 5, \ 5 \rightarrow 2$$

Immediately *after the first pass*, what are the values of `distTo[v]` for each vertex v? *Write the values in the corresponding boxes.*

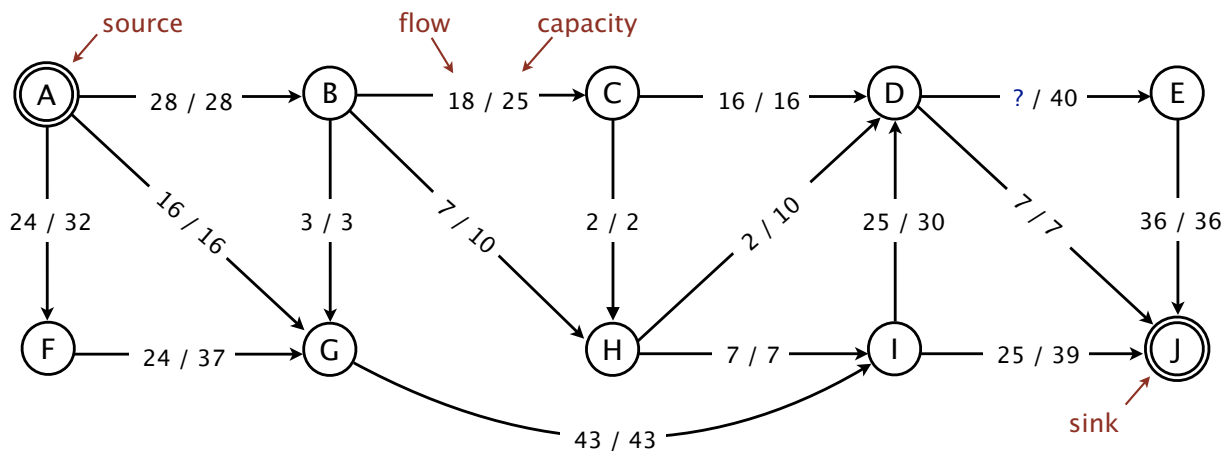| 0					
distTo[0]	distTo[1]	distTo[2]	distTo[3]	distTo[4]	distTo[5]

(c) Immediately after the first pass of *Bellman–Ford*, for which vertices v is `distTo[v]` the length of the *shortest* path from $s = 0$ to v?

Mark all vertices that apply.

■ ☐ ☐ ☐ ☐ ☐
0   1   2   3   4   5

5. **Maxflows and mincuts. (12 points)**

   Consider the following flow network and a *flow f*.

(a) What is the *flow* on the edge $D \to E$?

   ○   ○   ○   ○   ○   ○   ○   ○
   2   16   18   25   27   36   40   43

(b) What is the *capacity* of the cut $\{A, F, G, H, I\}$?

   ○   ○   ○   ○   ○   ○   ○   ○
   50   68   78   80   88   92   107   122

(c) Find an *augmenting path* with respect to f. Write the sequence of vertices in the path.

   $A \to$

(d) What is the value of the *maximum flow*?

   ○   ○   ○   ○   ○   ○   ○
   67   68   71   72   74   75   76

(e) Compute a *minimum A-J cut* in the graph. Which vertices are on the source side of your minimum cut? *Mark all that apply.*

A	B	C	D	E	F	G	H
■	☐	☐	☐	☐	☐	☐	☐

6. **Data structures. (8 points)**

(a) A *linear-probing hash table* of length 10 uses the hash function $h(k) = k \mod 10$ to hash integer keys between 0 and 100. For example, $h(23) = 3$ and $h(30) = 0$.

After inserting 8 keys into an empty hash table, the table is as shown below.

index	0	1	2	3	4	5	6	7	8	9
value	20	19	-	-	14	44	6	74	86	59

Which of the following choices gives a possible order in which the keys could have been inserted into an initially empty hash table?

Assume that the initial length of the array is 10 and that it neither grows nor shrinks.
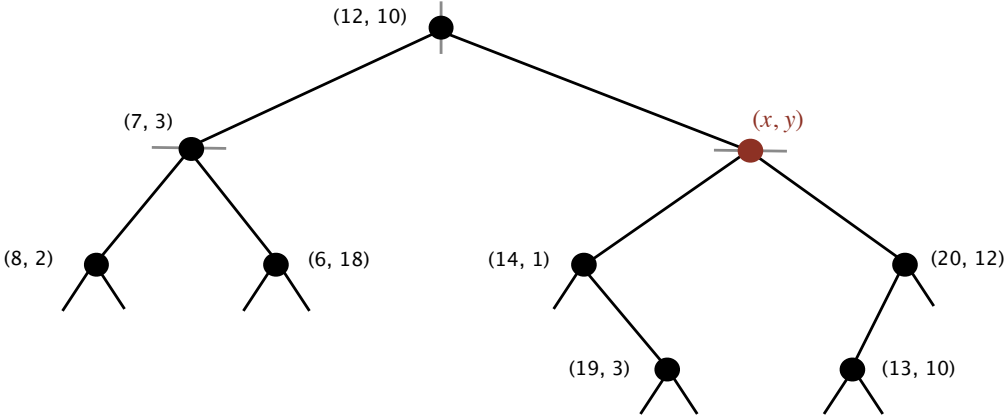
Fill in all checkboxes that apply.

☐ 14, 44, 6, 74, 86, 20, 59, 19

☐ 59, 6, 14, 20, 44, 19, 74, 86

☐ 20, 59, 6, 74, 14, 86, 44, 19

☐ 6, 59, 14, 20, 19, 44, 86, 74

(b) Consider the following *2d-tree*:

Which of the following points could correspond to (x, y)?

Fill in all checkboxes that apply.

☐	☐	☐	☐	☐	☐
$(20, 4)$	$(11, 9)$	$(14, 13)$	$(13, 11)$	$(17, 8)$	$(15, 2)$

7. **Properties of graph algorithms (5 points).**

   Identify each statement as either always true or sometimes/always false.

   true   *false*

   ○   ○   Let G be a weighted graph. If there is a unique minimum weight edge
                          in G, then this edge is in any MST of G.

   ○   ○   Let G be a digraph with non-negative weights and let s and t be two
                          vertices in G. Any shortest path from s to t in G is also a shortest path
                          from s to t in the graph obtained from G by squaring the weight of each
                          edge.

   ○   ○   Let G be a flow network with source s and sink t. Given a minimum
                          weight st-cut in G, the *value* of the maximum flow in G can be
                          computed in time $O(E + V)$.

   ○   ○   Let f be a flow. Assume that there are two augmenting paths with
                          respect to f (different on at least one edge), one with bottleneck
                          capacity 5 and the other with bottleneck capacity 10. Then, the value of
                          the maximum flow is at least 15 more than the value of f.

   ○   ○   Let G be an unweighted graph with a unique global mincut of size 2.
                          Suppose that we execute *Karger's algorithm* on G once, and that in this
                          execution, all the edges are assigned unique weights and the two edges
                          crossing the mincut are assigned the two *largest* weights. Then, Karger's
                          algorithm finds the minimum cut in G.

8. **Dynamic programming. (5 points)**

A store owner has a rod of length n feet. They can cut the rod into pieces of integer length and sell them separately. The value of a piece of rod depends only on its length. The values of pieces of rod of lengths between 0 and n are given in an array of non-negative integers called `values`, where the price of a piece of rod of length t is `value[t]` dollars (and we always have `value[0]=0`). However, *cutting a piece of rod to get two pieces costs $1*.

Determine the maximum profit obtainable by cutting up the rod and selling the pieces (selling the whole rod as one piece is allowed).

For example, if the length of the rod is $n = 8$ and the values of pieces of rods are given in the following `values` array, then the maximum obtainable profit is $20. This is achieved by cutting the rod to three pieces of lengths 1, 2 and 5. The value of those pieces is $2 + $5 + $15 = $22, but cutting the rod in two locations (to obtain three pieces) costs $2.

length	0	1	2	3	4	5	6	7	8
value	0	2	5	5	8	15	15	16	18

We will solve this problem using *dynamic programming*. Define the following *subproblems*, one for each i with $0 \le i \le n$:

$OPT(i)$ = *max profit obtainable by cutting up a rod of length i and selling its pieces*

Consider the following partial bottom-up implementation:

```
int[] opt = new int[n+1];

for    1    {
    opt[i] =    2    ;
    for    3
        opt[i] = Math.max(    4    ,
                          values[j] +    5    );
}
```

A. (int i = 0; i <= n; i++)

B. (int i = n; i >= 0; i--)

C. (int j = 1; j < n; j++)

D. (int j = 1; j < i; j++)

E. opt[i]

F. opt[j]

G. opt[i] - 1

H. opt[j] - 1

I. opt[i-j] - 1

J. values[0]

K. values[i]

L. values[j]

M. values[i] - 1

N. values[j] - 1

O. values[i-j] - 1

For each numbered oval above, write the letter of the corresponding code fragment on the right in the space provided. You may use each letter once, more than once, or not at all.

1	2	3	4	5

9. **Randomness (5 points).**

Consider the following two methods whose goal is to find a 1 entry in a binary array. Specifically, we are given an integer array `a` of length n with 0 and 1 entries. We assume that $n \geq 100$ and that *at least* $0.1n$ *entries contain a* 1 *value.* We wish to return *any* one index $0 \leq i < n$ such that `a[i] = 1`.

```
static int findOneA(int a[]) {
    int n = a.length;
    int r;
    for (int i = 0; i < 0.1 * n; i++) {
        r = StdRandom.uniformInt(n);
        if (a[r] == 1) return r;
    }
    return -1;
}

static int findOneB(int a[]) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        if (a[i] == 1) return i;
    }
    return -1;
}
```

Fill in all checkboxes that apply.

☐   `findOneA` implements a *randomized* algorithm and `findOneB` implements a *deterministic* (non-randomized) algorithm.

☐   `findOneA` has $O(1)$ *expected* running time, but $\Theta(n)$ *worst case* running time.

☐   `findOneB` has $\sim 0.1n$ *worst case* running time.

☐   `findOneA` always returns a correct answer (*i.e.*, returns i such that `a[i] = 1`).

☐   `findOneB` always returns a correct answer (*i.e.*, returns i such that `a[i] = 1`).

10. **Multiplicative weights (5 points).**

Consider the *experts problem* with $n \geq 2$ experts over a period of T days.

Identify each statement as either always true or sometimes/always false.

true   *false*

◯   ◯   In the *elimination* algorithm, an expert who makes 10 mistakes is eliminated before or on the same day as an expert who makes 5 mistakes.

◯   ◯   Consider the prediction algorithm that, on each day, returns the *minority* prediction. That is, it predicts 0 if and only if less than half of the experts predict 0 (experts are never removed). Then, for any possible expert predictions and outcomes, this algorithm makes at least as many mistakes as the *elimination* algorithm.

◯   ◯   Suppose that one of the experts always predicts correctly and the rest of the $n-1$ experts *always predict incorrectly*. Then, the total number of mistakes made by the *multiplicative weights* algorithm is $O(1)$.

◯   ◯   Suppose that there are $\frac{n}{4}$ experts who each make at most M mistakes. Then, the number of mistakes made by the *multiplicative weights* algorithm is $O(M)$.

◯   ◯   In the *simplified AdaBoost* algorithm, if after the first 5 iterations, one point has 2^5 times the weight as another point, then the first point was mislabeled by each of the decision stumps trained in the first 5 iterations.

11. **Intractability (5 points).**

   Identify each statement as either always true or sometimes/always false.

   true   *false*

   ◯   ◯   If problems X and Y are **NP**-complete, then problem Y poly-time reduces to problem X and problem X poly-time reduces to problem Y.

   ◯   ◯   If BIPARTITE-MATCHING poly-time reduces to a decision problem Y, then Y is in **P**.

   ◯   ◯   Let X, Y and Z be decision problems. If X poly-time reduces to Y and Y poly-time reduces to Z, then X poly-time reduces to Z.

   ◯   ◯   If **P** ≠ **NP** then **SAT** does not have a polynomial time algorithm.

   ◯   ◯   The witness and verification algorithm below prove that the following problem is in **NP**:
   Problem: Given a set of boolean equations and an integer k, decide if there is an assignment that satisfies *exactly* k of the equations.
   Witness: Assignment to variables claimed to satisfy exactly k equations.
   Verification algorithm: For each equation, check if it's satisfied by the assignment. Count the number of satisfied equations and compare to k.

12. **Design: shortest paths through a landmark. (10 points)**

   (a) Design a data structure called `Route`, that allows the client to find the length of the shortest path from a fixed *source* vertex s to a query *destination* vertex v through a fixed "*landmark*" vertex x.

   To construct a `Route` object, the client specifies the weighted *digraph* G (assume that the weights are positive integers), the source vertex s, and the landmark vertex x. The client can then run `pathLen(v)` with any vertex v to obtain the length of the shortest path from s to v through x.

   Implement the following API for `Route`:

   ```
   public class Route
   ```
   _____

   ```
          Route(EdgeWeightedDigraph G, int s, int x)
   ```
   creates a Route data structure

   ```
      int  pathLen(int v)
   ```
   returns the length of the shortest
   path from s through x to v

   Performance requirements. For full credit, the instance variables in your implementation should use space $O(V)$, where V is the number of vertices in G. The constructor should run in time $O(E \log V + V)$, where E is the number of edges in G, and `pathLen()` should run in $\Theta(1)$ time.

   Give a concise English description of your algorithm for implementing the *constructor*. You may use code or pseudocode to improve clarity. You may use any of the data types and algorithms that we considered in this course (either `algs4.jar` or `java.util` versions).

Give a concise English description of your algorithm for implementing `pathLen()`. You may use code or pseudocode to improve clarity.

(b) Design another data type called `RouteXL`, that, as before, allows the client to find the length of the shortest path through a fixed landmark vertex x. But now both the source vertex s, and the destination vertex v, are only specified when calling `pathLen()`.

That is, the new API is:

```
public class RouteXL
```

`RouteXL(EdgeWeightedDigraph G, int x)`	*creates a RouteXL data structure*
`int pathLen(int s, int v)`	*returns the length of the shortest path from s through x to v*

Performance requirements.

Full credit: As before, for full credit, the instance variables in your implementation should use space $O(V)$. The constructor should run in time $O(E \log V + V)$, and `pathLen()` should run in $\Theta(1)$ time.

Partial credit (at least 1/3 of the credit): Same performance guarantees as in the full credit option, but you may assume that the graph is *undirected*.

Are you attempting a full credit solution (directed graph) or a partial credit solution (undirected graph)?
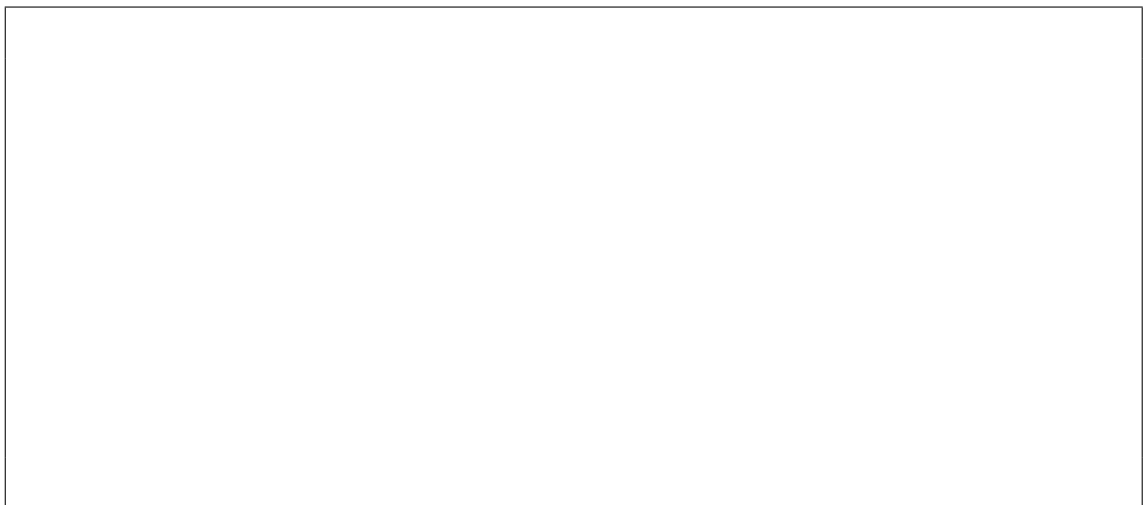
○　　○

full   partial
credit   credit

Give a concise English description of your algorithm for implementing the *constructor.* You may use code or pseudocode to improve clarity.

Give a concise English description of your algorithm for implementing `pathLen()`. You may use code or pseudocode to improve clarity.

13. **Design: shortest path with a reverse edge. (10 points)**

Let G be an (unweighted) digraph. Given a source vertex s and a target vertex t, we wish to find the *shortest "almost-path"* from s to t.

An *almost-path* in G is a path where *exactly* one of the edges on the path is used in the *reverse* direction. Formally, an almost-path of length m in G is a sequence of vertices $v_0, v_1, v_2, \ldots, v_m$ such that there exists $0 \le j < m$ for which the following holds: for all $0 \le i < m$, $i \ne j$, it holds that $v_i \to v_{i+1}$ is a directed edge in G. Additionally, $v_{j+1} \to v_j$ is a directed edge in G.

Example. In the following digraph G, the shortest path from 0 to 7 is $0, 1, 2, 3, 7$ (length 4); the shortest almost-path from 0 to 7 is $0, 6, 7$ (length 2, the edge $7 \to 6$ is used in the reverse direction).
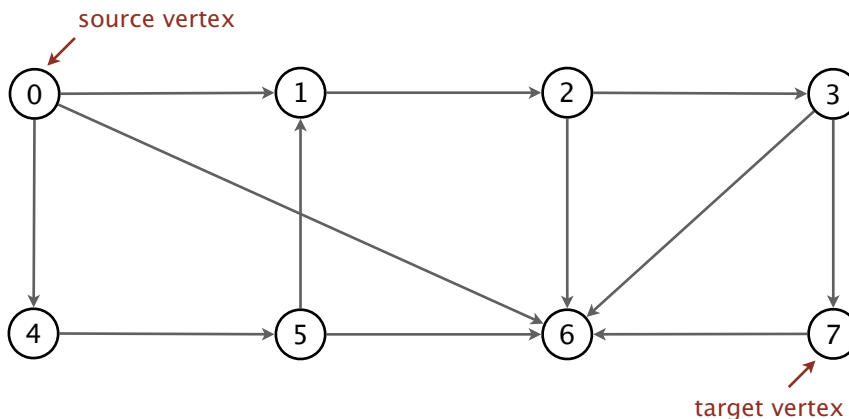
(a) *Is the following claim correct?*

For any digraph G and any two vertices s and t, no shortest almost-path from s to t in G uses an edge in both directions (that is, if an almost-path uses the edge $v \to w$, it does not use the reverse edge $w \to v$).

○   ○

yes   *no*

(b) Design an algorithm that finds the shortest almost-path from s to t.

   Full credit: Your algorithm should run in time $O(E + V)$ in the worst case, where V is the number of vertices in G and E is the number of edges.

   Partial credit (at least half credit): Your algorithm should run in time $O((E + V)E)$ in the worst case.

   Give a concise English description of your algorithm. Feel free to use code or pseudocode to improve clarity. *You may also add a sketch to illustrate your algorithm.*

   The running time of your solution is

   $\Theta(\qquad\qquad)$

(c) Assume we were to change the definition of an almost-path to say that *at most one* (instead of exactly one) of the edges is used in the reverse direction.

Design an algorithm for finding the shortest almost-path from s to t using this new definition. The full credit and partial credit performance requirements are as in part (b). *You may use the algorithm you designed in part (b) and explain the changes that should be made.*

This page is intentionally blank. You may use this page for scratch work.