CSCI 2270 Final Exam
Spring 2013

75 points = 100%
10 extra credit points possible.

Write Clearly!

Name: ______________________

9-Digit Student ID: ______________________

TA (circle one):

Shang   Yogesh   Xin   Don't Know

1. Pointers. For parts A and B, use the struct definitions shown at right and the following code:

```
hedge* one = new hedge;
hedge two;
```

```
// vertex
struct vert {
  float x;
  float y;
};

// half-edge
struct hedge {
  hedge* next;
  hedge* pair;
  vert* point;
  face* polygon;
};

// polygon
struct face {
  hedge* edge;
};
```

(a): (4 pts) Write the correct operator in each blank. Possible choices are: **.** **->** ***** **&**

one **->** next = **&** two;

(b) (6 pts) Give the *data type* in each blank.

hedge** x = &one;

hedge y = *one;

hedge* z = &two;

(c) (8 pts) Identify lines that cause compiler errors, and *correct them so the code compiles successfully*.

Note: the struct definitions above are correct, and the function signatures may not change.

There are several ways you can fix this, but erasing everything is not what we're looking for. (even though that would compile...)

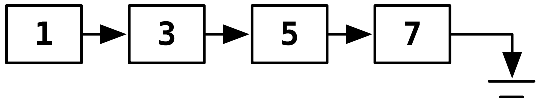
```
void silly() {
  float x = 4;
  float* y = 10;  float y = 10;
  hedge he = new hedge;  hedge* he ...
  he.pair = new hedge;  he->pair = ...
  he.point = mk_vert(&x, &y); he->point = ...
}

vert* mk_vert(float* xptr,
              float* yptr) {
  vert ret = new vert;vert* ret = ...
  ret->x = xptr;  ... = *xptr;
  ret.y = *yptr;  ret->y = ...;
  return ret;
}
```

_____ / 18

2. Recursion and Pointers.

```
struct node {
    int value;
    node* next;
};
node* init_node(int num) {
  node* ret = new node;
  ret-> value = num;
  return ret;
}
```

(a) (10 pts) Write a recursive function called *odds* (use the signature below) that makes a linked list with the first *n* odd numbers (begin with 1). For example, the list returned by *odds(4, NULL)* looks like:



It must return a pointer to the first node, or NULL if a non-positive input parameter is provided. This must be written in proper C++ syntax. *Hints: create the last element first. The nth odd is 2n-1 for n > 0.*

```
node* odds(int n, node* next) {
  if (n < 1) {
    return NULL;
  }
  node* item = init_node(n*2-1);
  item->next = next;
  if (n > 1) {
    node* ret = odds(n-1, item);
    return ret;
  } else {
    return item;
  }
}
```

(b) (6 pts) The math notation at right gives the Fibonacci Sequence. *Do two things*. **One**: translate it into code or pseudocode. Use the space below. *It must be recursive*. **Two**: fill in the table with proper values of fib(n).

$$\text{fib}(n) \begin{cases} 0 & \text{if } n < 1 \\ 1 & \text{if } n \text{ is } 1 \\ \text{fib}(n-2) + \text{fib}(n-1) & \text{if } n > 1 \end{cases}$$

```
int fib(int n) {
  if (n < 1) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return (fib(n-2) + fib(n-1));
  }
}
```

n	fib(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8

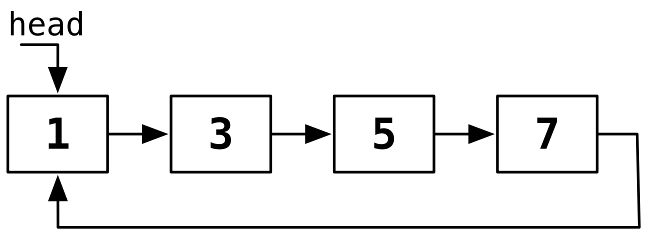
_____ / 16

(c) (6 pts) The function at right scrolls through a linked list and returns the number of times that a node's value is less than the *next* node's value. Write a recursive function that does the same thing. Use the signature given below. It will be called as: **ups(NULL, head)**. *Hint: Gabe's solution was 7 lines long.* Consider using another sheet to experiment and write your final version here.

```
int count_increasing(node* head) {
  node* cursor = head;
  node* prev = NULL;
  int num_increasing = 0;
  while (cursor != NULL) {
    if (prev != NULL &&
    prev->value < cursor->value) {
      num_increasing++;
    }
    prev = cursor;
    cursor = cursor->next;
  }
  return num_increasing;
}
```

```
int ups(node* prev, node* cursor) {
  if (there == NULL) {
    return 0;
  } else if (here != NULL && here->value < there->value) {
    return 1 + ups(there, there->next);
  } else {
    return ups(there, there->next);
  }
}
```

3. Design & Debugging.
A standard linked list ends when a node's *next* member is NULL. (See 2a for a diagram). A *circular* linked list's last element refers to the *head* of the list.

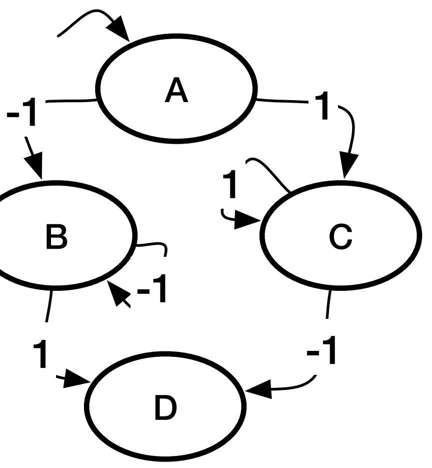


circular linked list

The code from question 2, Part C (the box above) runs forever when J. Random Student's code calls it with a pointer to a circular linked list. *In English (with code allowed)*, tell the student **(a)** (2 pts) what's the problem with their approach, and **(b)** (3 pts) how to re-write the iterative count_increasing function so it accommodates a circular linked list (it must still work for normal NULL-terminated lists).

The count_increasing function expects a null-terminated list, but this one never terminates. That's why it runs forever. Fix by having it test if the cursor is NULL *or* if it is the same as the head pointer. Just need to be careful that you let it run one time (don't stop the first time it is head, only stop once it gets to head the 2nd time).

4. Finite State Machines. (3 pts)



Say we have a FSM that obeys the diagram at right. If it receives a signal other than 1 or -1, nothing happens. Say the FSM starts in initial state A at the beginning of each batch of signals. What is the FSM's state at the end of each batch? Valid states are A, B, C, or D.

Batch 1: 1, 1, 0, 1. Result state: C

Batch 2: -1, 1, -1, -1. Result state: D

Batch 3: 0, 0, 0, 0. Result state: A

Batch 4: 1, 0, 0, 1. Result state: C

5. Complexity. (2 pts per blank) These questions use the struct definitions on page 1, and the code at the right.

```
// get distance from a to b
float dist(vert* a, vert* b) {
  float dx = a->x - b->x;
  float dy = a->y - b->y;
  float xx = dx * dx;
  float yy = dy * dy;
  return sqrt(xx + yy);
}
```

(a) Runtime complexity of *dist*:

O(1) *aka constant*

(b) Runtime complexity of using *dist* to find the distance between a single point and a set of *n* other vertices:

O(n) *aka linear*

(c) Say you have a list of *n* vertices. What is the runtime complexity of creating two binary trees, one that sorts on x, the other on y?

O(n log n) *aka loglinear*

(d) Complexity of using *dist* to compute and return the distance from each vertex to all other vertices in a set of *n* other vertices:

Runtime complexity: $O(n^2)$ (how many times is dist called?)

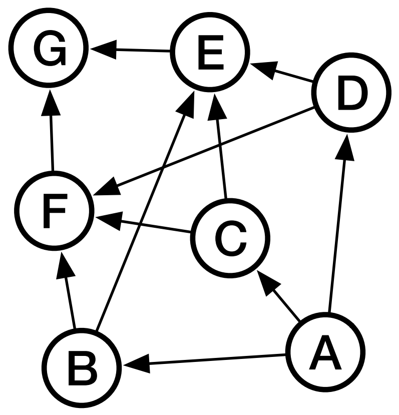
aka quadratic

Space complexity: $O(n^2)$ (how many results must we store?)

(e) Given a binary tree with height *h*, what size array is required to store all (non-NULL) node values? Assume a one-node tree has h=1.

minimum: h maximum: $2^h - 1$

6. Graphs. These questions relate to the graph shown at right involving nodes A through G.



(a) (2 pts) Is this graph a DAG? [✓ y] / n

(b) (4 pts) Say we perform a depth-first search beginning with node A. Fill in the table by indicating the discovery and finish time for each node. *Note: there are several possible solutions;* you only need to provide one. Assume node A is discovered at time step 0.

Node	disco time	finish time
A	0	13
B	1	8
C	9	10
D	11	12
E	6	7
F	2	5
G	3	4

7. More Graphs. These questions relate to the grid-like graph below involving nodes H through P. Edges have distance values.

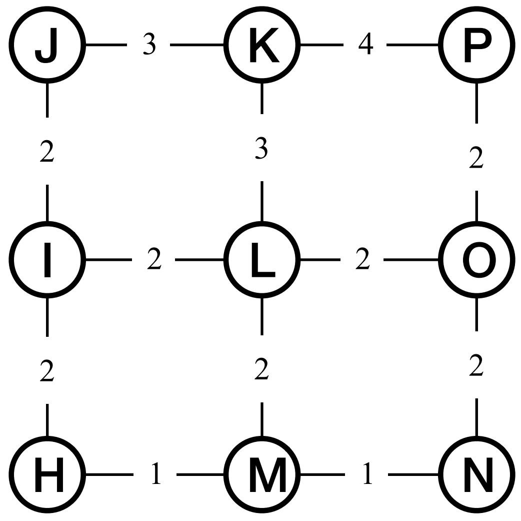
(a) (2 pts) What is the minimum distance from node H to node P? (a number)

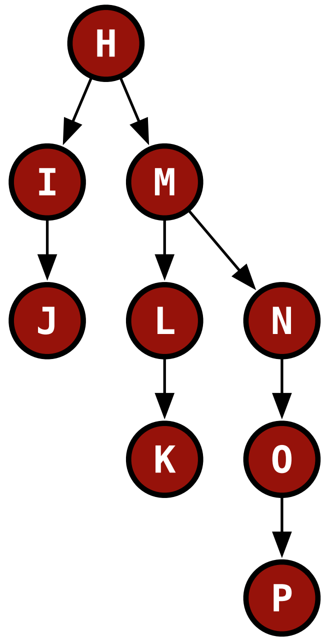
6

(b) (2 pts) What is the sequence of nodes that minimizes the distance from node H to node P?

H, M, N, O, P

(c) (3 pts) In the blank spot below, draw the complete minimum-spanning tree produced by Dijkstra's algorithm beginning at node H. You might want to use a blank page to work this out and re-draw the final version here. Only draw the tree. You don't need to give path totals.





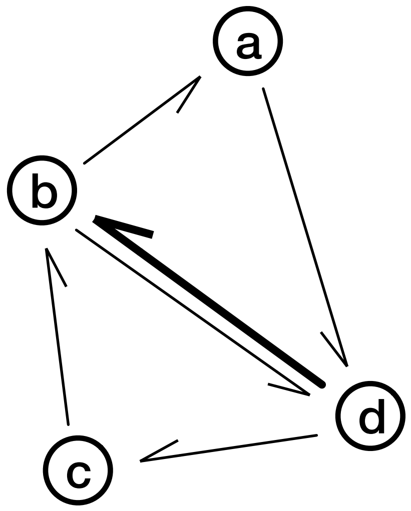
_____ / 13

8. Apply CS 2270 skills to new situations. *AKA the Evil Question.*

A *half-edge* is a common data structure in computer graphics to model surfaces. It combines vertices, directed edges between vertices, and faces (polygons) formed by closed loops of edges.

A half-edge stores the vertex where it *ends*, but not where it *begins*. Instead, all half-edges retain a pointer to its *pair*, which connects the same two points, but in the opposite direction. If there is no pair (e.g. the polygon is at the edge of the surface) this value is NULL. Each half-edge also points to the *next* half-edge on the same polygon.

A half-edge keeps a pointer to its face, and each face keeps a pointer to just *one* of the half-edges that surround it.



For example, say we have a reference to *db*---the half-edge that starts at d and ends at b. In the diagram this is shown with a thicker line. Its 'next' pointer is to *ba*, and *ba's* next pointer is to *ad*, which in turn points to *db* (where we started).

For this question, assume that all half-edges have a valid *next* pointer (no NULL detection needed).

(a) (4 pts) Write code that tells you if a polygon is a triangle:

```
bool is_triangle(face* poly) {
  hedge* start = poly->edge;
  return (start->next->next->next == start);
}
```

(b) (6 pts) Write code that tells you if a polygon has exactly n sides:

```
bool is_n_gon(face* poly, int n) {
  int count = 1;
  hedge* cursor = poly->start->next;
  while (cursor != poly->start) {
    count++;
    cursor = cursor->next;
  }
  return (count == n);
}
```

_____ / 10