Cpsc 448: Problem Solving in Computer Science
Midterm 1: February 27, 2003.

Name: ______________________________

Student Number: ______________________

Instructions, Rules & Regulations.

This examination booklet has 10 pages. Please ensure that your copy is complete. Do NOT open the booklets until prompted to do so. This examination will last for 80 minutes, and there are a total of 70 marks available on the test. There are easier and more difficult questions on the exam. The amount of marks allocated to each question is marked besides the question; use them to guide your time management.

To aid you in writing the exam, you are permitted to bring *notes* to the exam. The following items, and only these items, are acceptable as *notes*: Any hand-written notes from this (or any other) class; Any handouts of from this (or any other) class; Print-outs of any of your code; and finally any print-outs of material posted to the course web page. Note that in particular, the (optional) textbook of the course is NOT permitted.

In those questions where you are asked to write an algorithm, pseudo code is acceptable. Unless explicitly instructed otherwise, you can use any of the algorithms introduced in class as part of your solutions. Assume that your programs will be given 30 seconds on a reasonable 20th century computer to solve the problem (this means that you have several billion operations, but not more).

NOTE: You must answer questions 1, 2 and 3. Pick 4 more questions to answer from questions 4 through 9. Indicate your choice *clearly* in the table. We will mark only the questions marked with a YES in the table, no points are given for work on questions that were not marked. Please turn off any cell-phones, pagers or alarm clocks. Smoking is not permitted during the exam. Good luck!

Question	Answered	Marks
1	YES	/10
2	YES	/8
3	YES	/12
4		/10
5		/10
6		/10
7		/10
8		/10
9		/10
	Total	/70

Student Signature:

____

Question 1. True/False Questions [10 marks].
For each statement below, indicate whether it is true or false.

T F BFS is useful for finding shortest paths in unweighted graphs.

> *True: BFS works very well here.*

T F The Bellman-Ford algorithm does not work on graphs that have negative weight cycles.

> *False: Bellman-Ford detects whether there are negative weight cycles. In contrast, Floyd-Warshall will not notice such a cycle.*

T F If a problem can be solved recursively then it can be solved asymptotically more quickly using dynamic programming.

> *False: For example, insertion into a binary search tree can be implemented recursively in O(log n) time, which is optimal (no asymptotically faster algorithm exists).*

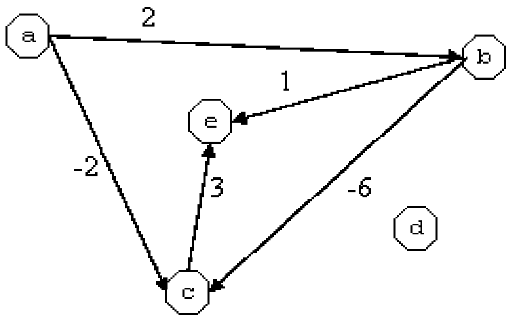
T F Given an undirected unweighted graph $G = <V, E>$ with $|V|=|E|=10\ 000$, representing this graph as an adjacency matrix takes up less space than as an adjacency list.

> *False: An adjacency list will need about 20,000 memory locations, depending on the exact implementation. An adjacency matrix will need 100,000,000 memory locations.*

T F We wish to provide $v > 10$ dollars in change. If we can do so using an unlimited supply of coins of values 2, 4, and 9, then we can also provide change with an unlimited supply of coins of values 2, and 5.

> *True: Note that 4 = 2 + 2 and 9 = 2 + 2 + 5. So we can make 2, 4 and 9 out of 2 and 5 using addition.*

Consider the following directed graph to answer the remaining True/False questions.



T F **Disregarding the weights, the graph can be represented as *G = <V, E> with V = {a, b, c, d, e}*, and**

E = {*(a,b), (b,c), (b,e), (c,e), (a,e), (a,c)*}.

False: There is no (a,e) edge.

T F There exists at least one negative weight cycle in the graph.

False

T F Length of the shortest path from *a* to *e* is 3.

False: The path a->c->e has length -1.

T F Suppose we run DFS on this graph, starting at vertex *a*. The algorithm will visit *e* at most once.

True: DFS, as described in class, always visits every (accessible) vertex exactly once.

T F The length of the longest path between any pair of vertices in the graph is 3.

True: The path c->e is the longest (length 3). Non-existant paths (like the one from a to d) do not technically have a length. However, since we have not made this clear, the answer "False" was also accepted.

Question 2. Edge Counting [8 marks].
In the table below, rows and columns describe different attributes that a graph can have. For example, cell 1,1 represents a graph that is directed, and may contain *self-loops* i.e., a vertex A may have an edge to itself. Fill in the table, where each cell contains the maximum number of edges that a graph of that particular type can have. Given answers in terms of *n*, the number of vertices in the graph.

	Directed	**Undirected**
Self-loops allowed	n^2	$n\frac{(n+1)}{2}$
Self-loops not allowed	$n(n-1)$	$n\frac{(n-1)}{2}$

Question 3. Short Answer Questions [12 marks].

For each question provide a short answer. For questions asking about time, or space complexity, no proof is required.

1.[2 marks] What is the asymptotic running time of the Bellman-Ford single-source shortest path algorithm, as a function of |V| and |E|? how about the Floyd-Warshall all-pairs shortest path algorithm?

Bellman-Ford: $O(|V||E|)$ *, Floyd-Warshall:* $O(|V^3|)$

2.[2 mark] Which algorithm is asymptotically faster for searching a graph, looking for a given vertex, DFS or BFS?

Neither. They are both linear in $|V|+|E|$ *.*

3.[2 marks] What is the output from the following code fragment?

```
char s[10] = "ABERCDZYX";
next_permutation(s[0], s[9]);
cout << s << endl;
```

ABERCXDYZ: The next string after ABERCDZYX in lexicographic order.

4.[2 marks] Both shortest path algorithms we have encountered produce a 'predecessor array' as one of their outputs. Describe what information is contained in this array.

For each vertex $v \in V$ *, pred[v] is the vertex u visited immediately before v in the shortest path.*

5.[2 marks] Given a graph and two vertices in the graph, describe one algorithm that can be used to detect if there exists a path between the two vertices.

DFS or BFS are the best choices (linear time).

6.[2 mark] We introduced an algorithm in class for finding exactly which amounts of change could be created given a (finite) set of coins. What will be the time and space complexity of the algorithm if we run it on a set containing *n* coins, where the total value of the coins present is *S*?

Time is $O(nS)$ *, space is* $O(S)$ *.*

Pick 4 more questions to answer from questions 4 through 9. Indicate your choice *clearly* in the table provided on the cover of the exam.

Question 4. Data Structures [10 marks]

You are required to implement a collection class that supports the following operations, all in constant time.

```
    - void Insert( int element)
    - Collection Combine( Collection C1, Collection C2 )
    - void Print()
```

Elements are inserted into the collection using the *Insert* operation. Two collections can be combined using the *Combine* operation to obtain a *new* collection with the elements of both collections. The collection **MAY** contain duplicate elements. Print() is the only access to the elements in the collection, and is **NOT** required to print the elements in sorted order. It is not even required to group duplicate objects together when printing.

Which data structure(s) would you use to implement this class? Give a short description of how each of the 3 operations will be implemented to conform to the constant time restriction.

*A **linked list** would do the job. The list would be kept in unsorted order. To insert an element, add it to the front (or the back) of the list. To combine two lists, splice them by connecting the last element of the first list to the first element of the second list. To print, traverse the list.*

Question 5. *k*-Paths [10 marks]

You are given

- a graph *G = (V, E)*, with |*V*| <= *10,*
- two vertices, *s* and *t*, in *V*, and
- an integer *k (0 < k <* |*V*|*)*.

Give an algorithm that will print a path from *s* to *t* in *G* of length exactly *k* if one exists. If not, print "No Way". Please specify the data structure(s) you use to represent the graph.

> *One possible solution involves taking the adjacency matrix of G to the k'th power. This is tricky to do because we would need to keep track of a predecessor array while taking powers.*

> *A better solution uses a modified BFS. If we omit the* **seen** *set and do exactly k steps of BFS, starting at s, the queue will contain all the vertices reachable in excatly k steps. If t is on the queue at that moment, follow the* **pred** *array back to s to find the path. Otherwise, print "No Way". By "a step of BFS" here I mean processing all the vertices on the queue that have the same depth. So, the first step would be adding all of s's neighbours to the queue. The second step will add all the neighbours of s's neighbours. Since we do not keep a* **seen** *set, the second step will add s to the queue again. Here is the C++-like pseudocode:*

```
void getPath( G, int s, int t, int k ) {
     int pred[k][NUM_VERTICES(G)];
     foreach (i, j) pred[i][j] = -1;

     queue< int > q;
     q.push( s );
     int count = 1;
     for( int i = 0; i < k; i++ ) {
          while( count-- ) {
               int u = q.top(); q.pop();
               foreach neighbour v of u in G {
                    pred[i][v] = u;
                    q.push( v );
               }
          }
          count = q.size();
     }
     if( q.count( t ) ) {
          for( int v = t; v >= 0; v = pred[--k][v] ) {
               printf( "%d <- ", v );
          }
          printf( "\n" );
     }
     else printf( "No Way\n" );
}
```

> *For a similar problem, take a look at assignment 4: problem P.*

Question 6. Useless Computations [10 marks]
We need a way to quickly compute *f(a,b)* defined below, given integers a and b in range [0, 100]. The recursive solution takes in excess of 2^100 iterations; we can not afford that much time. Provide pseudo code for this task. Your algorithm need to take in two integers *a, b*, and return another integer, that of value *f(a,b)*. Hint: Do not try to solve the recurrence for a closed form.

$$f(x, y) = \begin{cases} 0 & \text{if } x = 0 \\ x & \text{if } y = 0 \\ y + f(x-1, y) - f(x, y-1) & \text{otherwise} \end{cases}$$

All we need to do to make the recursive solution fast enough is use memoization:

```
bool seen[128][128];
int table[128][128];

int f( int x, int y )
{
     if( !seen[x][y] )
     {
          table[x][y] = (
               x == 0 ? 0 :
               y == 0 ? x :
               y + f( x – 1, y ) - f( x, y – 1 ) );
          seen[x][y] = true;
     }
     return table[x][y];
}

void main()
{
     memset( seen, 0, sizeof( table ) );

     // Get the values of x and y from the input

     cout << f( x, y ) << endl;
}
```

Question 7. Puddles [10 marks]

The street is full of circular puddles. You are a 5-year-old kid standing in the center of puddle number 1. You want to get to puddle number n, but you want to minimize your dry land walking distance.

Your input is the number n, followed by n triples of integers *(X_i, Y_i, R_i)*, one for each puddle. X_i, and Y_i are the Cartesian coordinates of puddle *i;* R_i is its radius.

Write an algorithm that will print the minimum distance you have to walk on dry land.

This is a standard single-source shortest path problem. Here, the puddles are the vertices. The source vertex is puddle number 1. The destination vertex is puddle number n. Think of edges as being the straight-line segments, connecting the centers of a pair of puddles. Draw an edge between every pair of puddles. The weight of an edge in this case is the dry-land distance between the two puddles, i and j, given by the formula

$$d(i,j)=\sqrt{(X_i-X_j)^2+(Y_i-Y_j)^2-R_i-R_j}$$

With this in mind, run a shortest-path algorithm (for example, Bellman-Ford) and print the answer.

Question 8. Transitive Closures [10 marks]
Given a graph *G=(V,E)*, produce a graph *H=(V,E*)* with the same vertex set, but such that E* will contain an edge *(u,v)* if and only if there is any path in G from *u* to *v*. Assume both graphs are directed. *H* is called the "transitive closure" of G.

Give pseudo code for computing the transitive closure of a given graph *G*. Do not worry about details of reading the graph from input or outputting the transitive closure after you compute it. Concentrate on computing its transitive closure. Please specify explicitly the data structure(s) you use to represent the two graphs.

> *Consider the adjacency matrix M of G, represented as an array of* ***bool****s – M[i][j] is* ***true*** *if there is an edge (i,j) in G and* ***false*** *if there is not. Now run this variant of the Floyd-Warshall algorithm:*

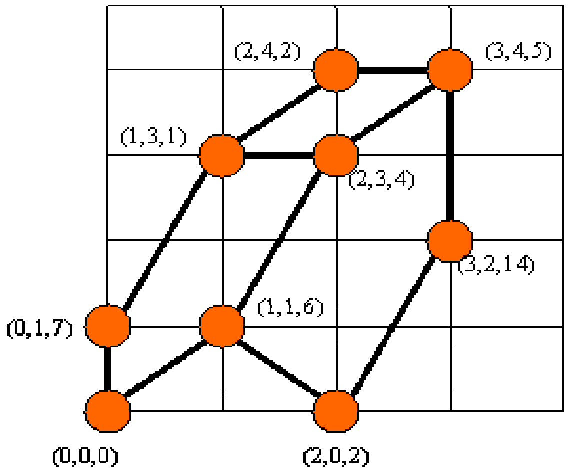
```
for( int k = 0; k < n; k++ )
     for( int i = 0; i < n; i++ )
          for( int j = 0; j < n; j++ )
               M[i][j] = M[i][j] || M[i][k] && M[k][j];
```

> *Now M contains the adjacency matrix of the desired transitive closure.*

Question 9. Height Advantage [10 marks]

The following algorithm is a straight forward modification of BFS for searching through a map for a given location. Each vertex is represented by a triple (x, y, z) where z is the height. At each step, instead of picking the first node in the queue (as BFS typically does) we pick the *vertex in the queue with largest height*. Given the undirected graph below, the source and destination, carry out the algorithm and show the vertices visited by the algorithm in the order visited, starting from Source until arriving at Destination. Hint: Destination will be found in finite time.

We wish to start from Source (0,0,0), and end up at Destination (3,4,5).



The vertices will be visited in this order:

```
(0,0,0)
      adds (0,1,7) and (1,1,6) to the queue.
(0,1,7)
      adds (1,3,1) to the queue.
(1,1,6)
      adds (2,0,2) and (2,3,4) to the queue.
(2,3,4)
      adds (3,4,5) to the queue (1,3,1) is already on the queue.
(3,4,5)
      Done.
```

The key idea here is that at every step, the algorithm has a queue (set) of ***frontier*** *vertices. Then, among these vertices, it picks the one with the largest z-value and adds all of its neighbours to the frontier.*