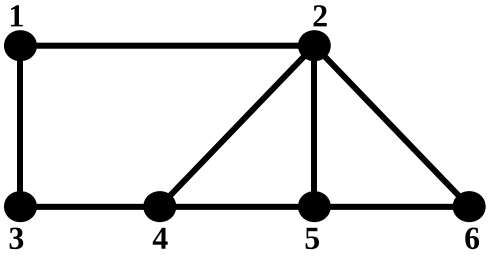
Programming Assignment 5

- This assignment contains 3 problems: A , B and C
- Prepare and submit 3 source files (A, B, C) using your favorite programming language, and put them into a zip file and then submit in Canvas before the deadline.
- You are responsible for correctly submitting each programming assignment on Canvas.
- PLEASE DO NOT SUBMIT ANY EXECUTABLE FILES THROUGH CANVAS.
- Before submitting make sure that the source codes can be compiled and successfully executed on the CSEGRID. No points will be awarded for programs that do not compile in the server. Your program may run on your machine, but still crash on omega. Programs that crash on CSEGRID at any point in their execution receive a penalty of 20%-100% depending on the posted requirements in the assignments.
- We will test your submitted programs with the data provided in the "Sample Inputs" section in each of the assignment statements **and also with other test files while following the input and output specifications**. YOU ARE RESPONSIBLE TO TEST YOUR PROGRAMS THOROUGHLY BASED ON THE INPUT/OUTPUT SPECIFICATIONS WRITTEN IN THE ASSIGNMENT.

Problem A: Find shortest path

In this problem you are going to implement the graph data structure and associated algorithm *(hint:BFS)*. Just to restate, the adjacency list representation of a graph consists of a sequence of lists. Each list corresponds to a vertex in the graph and gives the neighbors of that vertex. For example, the graph



has adjacency list representation

```
1: 2 → 3
2: 1 → 4 →  5 →  6
3: 1 → 4
4: 2 → 3 → 5
5: 2 → 4 → 6
6: 2 → 5
```

You will create a Graph data structure which represents an undirected graph as an array of Lists. Each vertex will be associated with an integer label in the range 1 to *n*, where *n* is the number of vertices in the graph. Your program will use this Graph data structure to find shortest paths (i.e. paths with the fewest edges) between pairs of vertices.

Input Specifications

First line of the input will contain a single integer N (N<=20), denotes number of test cases (i.e., example graphs) you need process.

Following this there will be N input graphs. Each of the input has two parts. The first part begins with a line consisting of a single integer *n*which is the number of vertices in the graph. Each subsequent line will represent a single edge in the graph by a pair of distinct numbers in the range 1 to *n*, separated by a space. These numbers are the end vertices of the corresponding edge. The first part of the input file defines the graph, and will be terminated by a dummy line containing "`0 0`". After these lines are read your program will print the adjacency list representation of the graph to the output file. For instance, the lines below define the graph pictured above, and cause the above adjacency list representation to be printed.

```
6
1 2
1 3
2 4
2 5
2 6
3 4
4 5
5 6
0 0
```

The second part of the input will consist of a number of lines, each consisting of a pair of integers in the range 1 to *n*, separated by a space. Each line specifies a pair of vertices in the graph; a starting point (or source) and a destination. The second part of the input will also be terminated by the dummy line "`0 0`". For each source-destination pair your program will do the following:

- Perform a Breadth First Search (BFS) on the given source to assign a parent vertex to each vertex in the graph. The BFS algorithm will be discussed in class and is described in general terms below.
- Use the results of BFS to print out the distance from the source vertex to the destination vertex, then use parent pointers to print out a shortest path from source to destination.

Output Specifications

For each of the N test cases, first print the case number, and then your program's operation can be broken down into two basic steps, corresponding to the two groups of input data.

1. Read and store the graph and print out its adjacency list representation.
2. Enter a loop which processes the second part of the input. Each iteration of the loop should read in one pair of vertices (source, destination), run BFS on the source, print the distance, then find and print the resulting shortest path, if it exists, or print a message if no path from source to destination exists.

If there is no path from source to destination (which may happen if the graph is disconnected), then your program should print a message to that effect. Note that there may be more than one shortest path. The particular path discovered by BFS depends on the order in which it steps through the vertices in your adjacency lists.

Sample Input	**Sample output**
2	
6 1 2 1 3 2 4 2 5 2 6 3 4 4 5 5 6 0 0 1 5 3 6 2 3 4 4 0 0	Graph #1: 1: 2 3 2: 1 4 5 6 3: 1 4 4: 2 3 5 5: 2 4 6 6: 2 5 The distance from 1 to 5 is 2. A shortest path from 1 to 5 is: 1 -> 2 -> 5 The distance from 3 to 6 is 3. A shortest path from 3 to 6 is: 3 -> 1 -> 2 -> 6 The distance from 2 to 3 is 2. A shortest path from 2 to 3 is: 2 -> 1 -> 3 The distance from 4 to 4 is 0. A shortest path from 4 to 4 is: 4

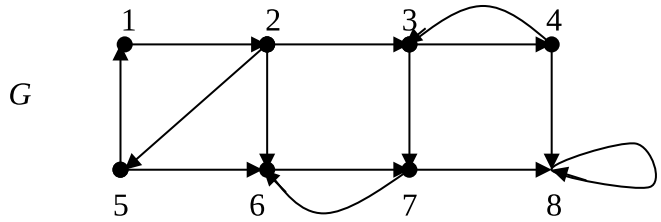
7 1 4 1 5 4 5 2 3 2 6 3 7 6 7 0 0 2 7 3 6 1 7 0 0	Graph #2: 1: 4 5 2: 3 6 3: 2 7 4: 1 5 5: 1 4 6: 2 7 7: 3 6 The distance from 2 to 7 is 2. A shortest path from 2 to 7 is: 2 -> 3 -> 7 The distance from 3 to 6 is 2. A shortest path from 3 to 6 is: 3 -> 2 -> 6 The distance from 1 to 7 is infinity. No path from 1 to 7 exists

Problem B: Strongly Connected Components

In this assignment you will build a Graph module in C, implement Depth First Search (DFS), and use your Graph module to find the strongly connected components of a directed graph. Begin by reading sections 22.3-22.5 in the text.

A directed graph (or digraph) $G = (V, E)$is said to be *strongly connected* if for every pair of vertices $u, v \in V$, u is reachable from v and v is reachable from u. Most directed graphs are not strongly connected. In general we say a subset $X \subseteq V$is *strongly connected* if every vertex in X is reachable from every other vertex in X. A subset which is maximal with respect to this property is called a *strongly connected component* of G. In other words, $X \subseteq V$is a strongly connected component of G if and only if X is strongly connected, and the addition of one more vertex to X would create a subset which is not strongly connected.

Example



It's easy to see that there are 4 strong components of G: $C_1 = \{1,2,5\}$, $C_2 = \{3,4\}$, $C_3 = \{6,7\}$, and $C_4 = \{8\}$.

To find the strong components of a digraph G first call DFS(G), and as vertices are finished, place them on a stack. When this is complete the stack stores the vertices sorted by decreasing finish times. Next compute the transpose G^T of G, which is obtained by reversing the directions on all edges of G. Finally, run DFS(G^T), but in the main loop (lines 5-7) process vertices by decreasing finish times. (Here we mean finish times from the first call to DFS.) This is accomplished by simply taking vertices off the stack in loop 5-7 of DFS. When this process is complete the trees in the resulting DFS forest constitute the strong components of G. (Note the strong components of G are identical to the strong components of G^T.) See the algorithm (Strongly-Connected-Components) and proof of correctness in section 22.5 of the text.

Input Specifications

Observe that the input file format is very similar to that of problem A.
First line of input will contain an integer N (N<=20), number of test cases. Following this line, there will be N test graphs. The first line of each graph will be given by the number of vertices in the graph, subsequent lines specify directed edges, and input is terminated by the 'dummy' line 0 0.

Output Specifications

For each of the N graphs, you will be doing the following:

- Read the input graph.
- Store the graph using the adjacency list representation.
- Print the adjacency list representation G to the output file.

- Run DFS on G and G^T, processing the vertices in the second call by decreasing finish times from the first call.
- Determine the strong components of G.
- Print the strong components of G to the output file.

Sample Input	**Sample output**
1	
8 1 2 2 3 2 5 2 6 3 4 3 7 4 3 4 8 5 1 5 6 6 7 7 6 7 8 8 8 0 0	Graph #1: Adjacency list representation of G: 1: 2 2: 3 5 6 3: 4 7 4: 3 8 5: 1 6 6: 7 7: 6 8 8: 8 G contains 4 strongly connected components: Component 1: 1 5 2 Component 2: 3 4 Component 3: 7 6 Component 4: 8

Problem C: Minimum Spanning Tree

In this assignment you will implement Prim's algorithm for finding a minimum weight spanning tree in a weighted (undirected) graph.

One significant difference between this problem and problem B is that you will be working with undirected graphs. It will therefore be necessary to create a function called addEdge(u, v) (instead of addDirectedEdge) which establishes an undirected edge joining vertices u and v. Another difference is that each edge will posses a weight, which should be stored as type `double`. These weights can be maintained by including a two dimensional array of doubles as part of your Graph struct. If vertex u is adjacent to vertex v, then the u^{th} row v^{th} column this array contains the weight of the corresponding edge. If u is not adjacent to v, then the u^{th} row v^{th} column contains infinity. As usual it is recommended that you avoid index 0, and therefore this array will be of size $(n + 1) \times (n + 1)$, where n is the number of vertices in the graph.

As in the last two projects vertices will be labeled 1 through n. Prim's algorithm (page 634 of the text) requires that vertices posses the attributes *parent* and *key*. It is recommended that these attributes be included in your Graph as a pair of parallel arrays of types `int` and `double` respectively.

Your project will be tested on connected graphs with no more that 1,000 vertices and 100,000 edges in which no edge weight exceeds 1,000. Therefore an adequate value to represent infinity is 10,000. It is recommended that you #define macros for infinity and nil. It may seem that Prim should have as precondition that the input graph G is connected, since if this is false, G does not contain any spanning tree. However a glance at MST-Prim on page 634 of the text reveals that nothing bad happens when the algorithm is run on a disconnected graph. What does Prim return in this case? In any case, we avoid this question by guaranteeing that the input graph will be connected. Therefore, you don't need to worry about that.

Input and Output Specifications

There will be N test cases which will be given in the first line of the input.
Following this, for each of the N test graphs, first line of input gives the number of vertices in the graph, and the second line gives the number of edges. (Thus it is not necessary to terminate the input by a dummy line 0 0, as was done in problem A or B.) Each of the remaining lines gives the end vertices and the weight of an edge.

So, in a nutshell, your program must be doing the following for each of the test graph:

- Read the input file.
- Store the graph using the list representation.
- Print the adjacency list representation of G to the output file.
- Run Prim on G (using any root r) to determine a minimum weight spanning tree in G, encoded by the parent of each vertex.
- Print a listing of the edges in that tree, as well as it's total weight, to the output file. Include the end vertices and weight of each edge.

Sample Input	Sample Output
1	
10 20 1 2 1.0 1 4 5.0 2 3 3.0 2 4 1.0 2 5 5.0 3 5 5.0 3 6 2.0 3 7 2.0 4 5 5.0 4 8 4.0 4 9 3.0 5 6 5.0 5 8 2.0 5 9 5.0 5 10 1.0 6 7 1.0 6 10 4.0 7 10 4.0 8 9 4.0 9 10 4.0	Graph #1: Adjacency list: 1: 2 4 2: 1 3 4 5 3: 2 5 6 7 4: 1 2 5 8 9 5: 2 3 4 6 8 9 10 6: 3 5 7 10 7: 3 6 10 8: 4 5 9 9: 4 5 8 10 10: 5 6 7 9 A minimum weight spanning tree consists of: Edge (1, 2) of weight 1.0 Edge (2, 3) of weight 3.0 Edge (2, 4) of weight 1.0 Edge (10, 5) of weight 1.0 Edge (3, 6) of weight 2.0 Edge (6, 7) of weight 1.0 Edge (5, 8) of weight 2.0 Edge (4, 9) of weight 3.0 Edge (6, 10) of weight 4.0 Total weight = 18.0