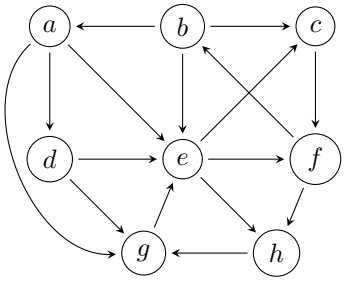
Name: _____

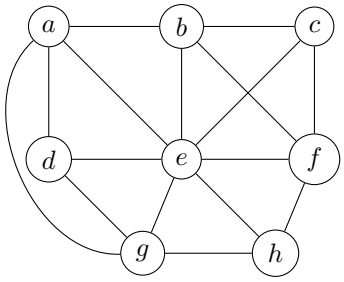
Write all of your responses on these exam pages. If you need extra space please use the backs of the pages.

1 Theory

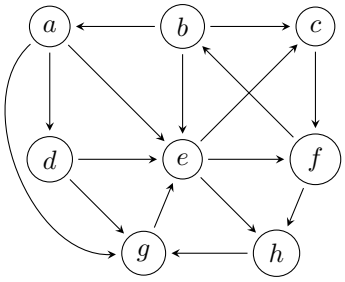
1. (*5 Points*) Draw the representation of this following graph as an adjacency list, as if it were a linked list of linked lists.



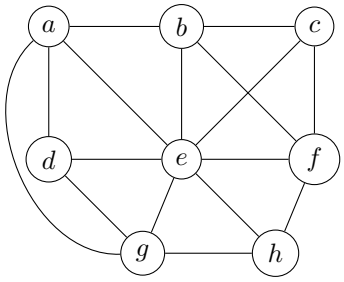
2. (*5 Points*) Draw the representation of this following graph as an adjacency matrix. Assume the row and column designations for the vertices are in alphabetical order.



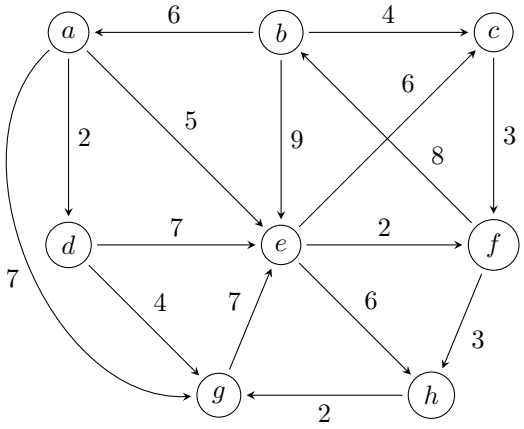
3. (*5 Points*) Create the spanning tree of the following graph using a Depth First Search. As usual, process the nodes and edges in alphabetical order.



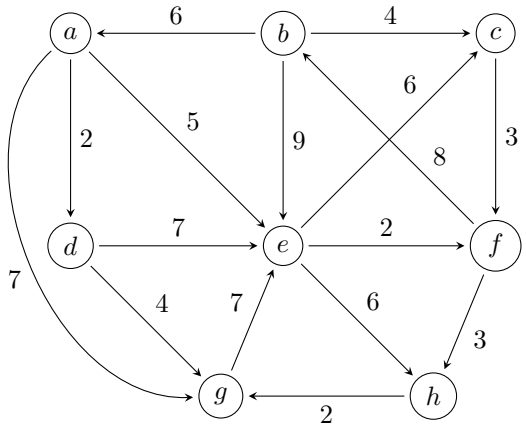
4. (*5 Points*) Create the spanning tree of the following graph using a Breadth First Search. As usual, process the nodes and edges in alphabetical order.



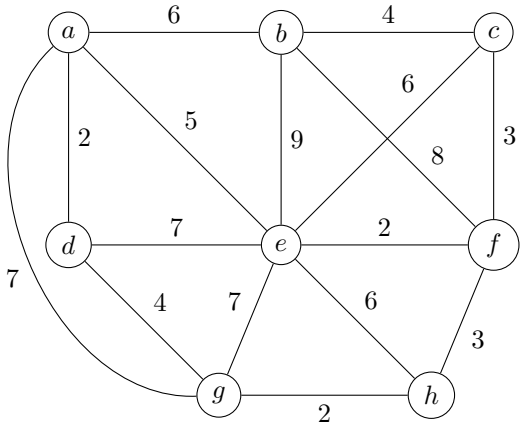
5. (*10 Points*) Go through all the steps of Dijkstra's Algorithm to determine the shortest path from vertex a to all other vertices. Create a chart of the steps as we did in class and is represented in the text.



6. (*10 Points*) Go through all the steps of Ford's Algorithm to determine the shortest path from vertex a to all other vertices. Create a chart of the steps as we did in class and is represented in the text. As usual, process the nodes and edges in alphabetical order.



7. (*10 Points*) Go through all the steps of Kruskal's Algorithm to determine the minimal spanning tree of the following graph.



2 Coding

1. (*25 Points*) Code any two sorts from the following list.

   (a) Merge Sort
   (b) Quick Sort
   (c) Comb Sort
   (d) Shell Sort
   (e) Radix Sort for non-negative integer data.
   (f) Count Sort for positive integer data.
   (g) Bucket Sort for floating point data in the range $[0, 1)$.

---

2. (*25 Points*) Given the following code framework, code any two of the following. All implementations are to be templated.

   (a) Depth first search of the given graph. This will print out a list of edges to the console in order of the edge traversal for this algorithm. The graph is stored as a ListOfLists object in adjacency list form (not an adjacency matrix).

   `void depthFirstSearch(ListOfLists<T> G)`

   (b) Breadth first search of the given graph. This will print out a list of edges to the console in order of the edge traversal for this algorithm. The graph is stored as a ListOfLists object in adjacency list form (not an adjacency matrix).

   `void breadthFirstSearch(ListOfLists<T> G)`

   (c) Dijkstra's algorithm to find the shortest path. This will return a vector of weighted nodes that will store the node name and the minimal distance from the start node to the node itself. The graph is stored as a list (vector) of weighted edge objects. The parameter start is the initial vertex to start from.

   `vector<wnode<T>> DijkstraAlgorithm(vector<wedge<T>> G, T start)`

   (d) Ford's algorithm to find the shortest path. This will return a vector of weighted nodes that will store the node name and the minimal distance from the start node to the node itself. The graph is stored as a list (vector) of weighted edge objects. The parameter start is the initial vertex to start from.

   `vector<wnode<T>> FordAlgorithm(vector<wedge<T>> G, T start)`

   (e) Kruskal's algorithm to find the minimal spanning tree. This will return a vector of weighted edges that will store the edge list for the construction of the minimal spanning tree. The graph is stored as a list (vector) of weighted edge objects.

   `vector<wedge<T>> KruskalAlgorithm(const vector<wedge<T>> &G)`

```
template<class T>
class edge {
public:
    T f, t;

    edge(T from, T to) {
        f = from;
        t = to;
    }
};

template<class T>
class wnode {
public:
    T name;
    double weight;

    wnode(T t, double w = 0) {
        name = t;
        weight = w;
    }

    friend ostream& operator <<(ostream &strm, const wnode &obj) {
        strm << obj.name << " : " << obj.weight;
        return strm;
    }

};

template<class T>
```

```
class wedge {
public:
    T from, to;
    double weight;

    wedge(T f, T t, double w = 0) {
        from = f;
        to = t;
        weight = w;
    }

    bool operator<(const wedge &rhs) {
        return weight < rhs.weight;
    }

    bool operator>(const wedge &rhs) {
        return weight > rhs.weight;
    }

    bool operator==(const wedge &rhs) {
        return (weight == rhs.weight) && (from == rhs.from) &&
        (to == rhs.to);
    }

    friend ostream& operator <<(ostream &strm, const wedge &obj) {
        strm << obj.from << " -> " << obj.to << " : " << obj.weight;
        return strm;
    }
};
```

You may also assume that you have the `ListOfLists` structure we used in class, specification is below.

```
template<class T>
class ListOfLists {
protected:
    vector<vector<T>> list;

public:
    ListOfLists(int rows = 0, int cols = 0);
    virtual ~ListOfLists();

    int size();
    void addRow();
    void addRows(int rows = 1, int cols = 0);
    void push_back(vector<T>);
    vector<T>& operator[](const int&);
};
```

You may also assume that you have the following functions at your disposal without creating them.

- `int findVertexPos(ListOfLists<T> G, T v)` returns the position of the vertex v in the ListOfLists structure for graph G.
- `int getWnodePos(const vector<wnode<T>> &nodes, T node)` returns the position of the weighted node *node* in the list of weighted nodes *nodes*.
- `int findMinWnodePos(const vector<wnode<T>> &nodes)` returns the position of the minimum weighted node in the list of weighted nodes *nodes*.
- `bool detectCycles(ListOfLists<T> G)` that will return true if there is a cycle in the graph and false otherwise.
