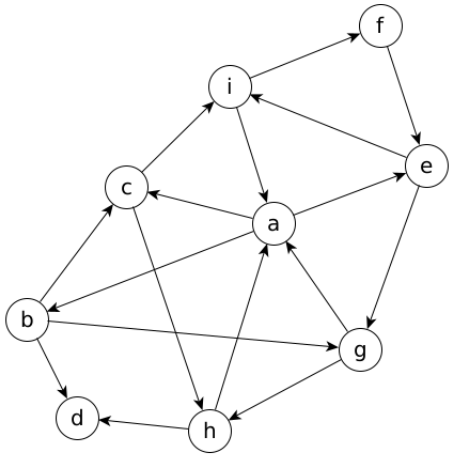
Name: _____

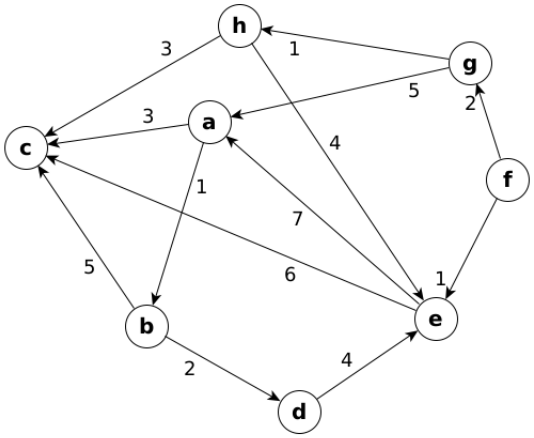
Write all of your responses on these exam pages. If you need extra space please use the backs of the pages.

1. **Algorithms:** (*60 Points*)

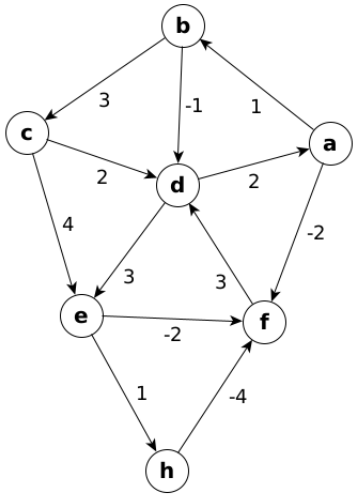
   (a) Given the following directed graph display the spanning tree/forest for a depth-first search/traversal and the spanning tree/forest for a breadth-first search/traversal. As usual, the vertices are to be processed in alphabetical order.



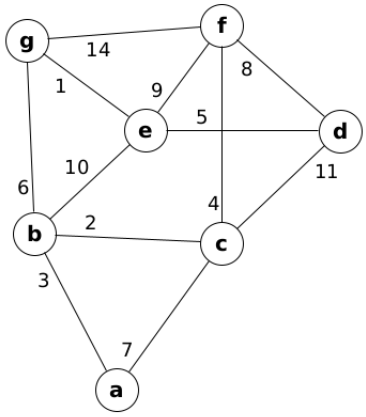
(b) Given the following directed weighted graph, use Dijkstra's algorithm to find the shortest path to all vertices from the starting vertex f. Display each iteration, active vertex, and weight label in chart form as was done in the text and in class.



(c) Given the following directed weighted graph, use Ford's algorithm to find the shortest path to all vertices from the starting vertex b. Display each iteration and sequence of weight label changes in chart form as was done in the text and in class.



(d) Given the following weighted graph, use Kruskal's algorithm to find the minimum spanning tree of the graph. Display the final resulting minimum spanning tree.



2. **Complexities:** (*15 Points*)

   (a) What is the complexity of the depth-first search/traversal?

   (b) What is the complexity of the breadth-first search/traversal?

   (c) What is the complexity of Dijkstra's algorithm for finding the shortest path from one vertex to all the other vertices in a graph?

   (d) What is the complexity of Ford's algorithm for finding the shortest path from one vertex to all the other vertices in a graph?

   (e) What is the complexity of Kruskal's algorithm for finding a minimal spanning tree for a graph?

   (f) What is the complexity of Dijkstra's algorithm for finding a minimal spanning tree for a graph?

   (g) What is the complexity of the Ford-Fulkerson algorithm for finding the maximum flow through a network?

3. **Code:** (*30 Points*)

   (a) Given our graph class structure, write a depth-first search/traversal that will return a list of edges for the traversal order to follow in a depth-first search. The list of edges should be a vector of pairs of templated type, the type that is storing the vertex label.

(b) Given our weighted graph class structure, write a function that will use Kruskal's algorithm to return the minimal spanning tree of the input (parameter) weighted graph. The return type should be a weighted graph, templated of course.

```
template <class T> class Graph {
protected:
  vector<pair<T, vector<T>>> graph;
  bool directed;

public:
  Graph(bool dir = false);
  virtual ~Graph();
  bool isDirected() { return directed; }
  int numVertices();
  int size();
  void clear();
  void addVertex(T);
  void addVertices(vector<T>);
  void addEdge(T, T);
  void addEdge(pair<T, T>);
  void addEdges(T, vector<T>);
  void addEdges(vector<pair<T, T>>);
  void deleteEdge(T, T);
  void deleteEdge(pair<T, T>);
  void deleteEdges(T, vector<T>);
  void deleteEdges(vector<pair<T, T>>);
  void deleteVertex(T);
  int getVertexPos(T);
  int getEdgePos(T, T);
  vector<T> getAdjacentList(T);
  vector<T> getVertexList();
  vector<pair<T, T>> getEdgeList();
  void sortVertexList() { sort(graph.begin(), graph.end()); };
};

template <class T> Graph<T>::Graph(bool dir) { directed = dir; }
template <class T> Graph<T>::~Graph() {}
template <class T> int Graph<T>::numVertices() { return graph.size(); }
template <class T> int Graph<T>::size() { return graph.size(); }
template <class T> void Graph<T>::clear() { graph.clear(); }

template <class T> void Graph<T>::addVertex(T v) {
  bool found = false;
  for (auto vp : graph)
    if (vp.first == v)
      found = true;

  if (!found)
    graph.push_back({v, {}});
}

template <class T> void Graph<T>::addVertices(vector<T> vlist) {
  for (size_t i = 0; i < vlist.size(); i++)
    addVertex(vlist[i]);
}

template <class T> int Graph<T>::getVertexPos(T v) {
  for (size_t i = 0; i < graph.size(); i++)
    if (graph[i].first == v)
      return i;

  return -1;
}

template <class T> int Graph<T>::getEdgePos(T v, T vt) {
  int vpos = getVertexPos(v);
  if (vpos >= 0) {
    vector<T> adjlist = graph[vpos].second;
    for (size_t i = 0; i < adjlist.size(); i++)
      if (adjlist[i] == vt)
        return i;
  }

  return -1;
}
```

```
template <class T> void Graph<T>::addEdge(T v, T vt) {
  addVertex(v);
  addVertex(vt);
  int pos = getVertexPos(v);

  if (getEdgePos(v, vt) == -1)
    graph[pos].second.push_back(vt);

  if (!directed) {
    pos = getVertexPos(vt);

    if (getEdgePos(vt, v) == -1)
      graph[pos].second.push_back(v);
  }
}

template <class T> void Graph<T>::addEdge(pair<T, T> e) {
  addEdge(e.first, e.second);
}

template <class T> void Graph<T>::addEdges(T v, vector<T> vt) {
  for (size_t i = 0; i < vt.size(); i++)
    addEdge(v, vt[i]);
}

template <class T> void Graph<T>::addEdges(vector<pair<T, T>> elist) {
  for (size_t i = 0; i < elist.size(); i++)
    addEdge(elist[i].first, elist[i].second);
}

template <class T> void Graph<T>::deleteEdge(T v1, T v2) {
  int pos = getVertexPos(v1);
  int epos = -1;
  if (pos != -1) {
    epos = getEdgePos(v1, v2);
    if (epos != -1)
      graph[pos].second.erase(graph[pos].second.begin() + epos);
  }

  if (!directed) {
    pos = getVertexPos(v2);
    if (pos != -1) {
      epos = getEdgePos(v2, v1);
      if (epos != -1)
        graph[pos].second.erase(graph[pos].second.begin() + epos);
    }
  }
}

template <class T> void Graph<T>::deleteEdge(pair<T, T> p) {
  deleteEdge(p.first, p.second);
}

template <class T> void Graph<T>::deleteEdges(T v, vector<T> vt) {
  for (size_t i = 0; i < vt.size(); i++)
    deleteEdge(v, vt[i]);
}

template <class T> void Graph<T>::deleteEdges(vector<pair<T, T>> ep) {
  for (size_t i = 0; i < ep.size(); i++)
    deleteEdge(ep[i].first, ep[i].second);
}

template <class T> void Graph<T>::deleteVertex(T v) {
  int pos = getVertexPos(v);
  if (pos == -1)
    return;

  graph.erase(graph.begin() + pos);
  for (size_t i = 0; i < graph.size(); i++) {
    size_t vpos = find(graph[i].second.begin(), graph[i].second.end(), v) -
                  graph[i].second.begin();
```

```
    if (vpos < graph[i].second.size())
      graph[i].second.erase(graph[i].second.begin() + vpos);
  }
}

template <class T> vector<T> Graph<T>::getAdjacentList(T v) {
  int pos = getVertexPos(v);
  if (pos != -1)
    return graph[pos].second;

  vector<T> empty;
  return empty;
}

template <class T> vector<pair<T, T>> Graph<T>::getEdgeList() {
  vector<pair<T, T>> elist;
  for (size_t i = 0; i < graph.size(); i++) {
    T v1 = graph[i].first;
    vector<T> Adj = graph[i].second;
    for (size_t j = 0; j < Adj.size(); j++) {
      T v2 = Adj[j];
      elist.push_back({v1, v2});
    }
  }
  return elist;
}

template <class T> vector<T> Graph<T>::getVertexList() {
  vector<T> lst;
  for (size_t i = 0; i < graph.size(); i++)
    lst.push_back(graph[i].first);

  return lst;
}
```

```
template <class T, class W> class WGraph {
protected:
  vector<pair<T, vector<pair<T, W>>>> graph;
  bool directed;

public:
  WGraph(bool dir = false);
  virtual ~WGraph();
  bool isDirected() { return directed; }
  int numVertices();
  int size();
  void clear();
  void addVertex(T);
  void addVertices(vector<T>);
  void addEdge(T, T, W);
  void addEdge(T, pair<T, W>);
  void addEdge(pair<T, pair<T, W>>);
  void addEdges(T, vector<pair<T, W>>);
  void addEdges(vector<pair<T, pair<T, W>>>);
  void deleteEdge(T, T);
  void deleteEdge(pair<T, T>);
  void deleteEdges(T, vector<T>);
  void deleteEdges(vector<pair<T, T>>);
  void deleteVertex(T);
  vector<pair<T, W>> getAdjacentList(T);
  vector<T> getVertexList();
  vector<pair<T, pair<T, W>>> getEdgeList();
  int getVertexPos(T);
  int getEdgePos(T, T);
  void sortVertexList() { sort(graph.begin(), graph.end()); }
};

template <class T, class W> WGraph<T, W>::WGraph(bool dir) { directed = dir; }
template <class T, class W> WGraph<T, W>::~WGraph() {}
template <class T, class W> int WGraph<T, W>::numVertices() {
  return graph.size();
}
template <class T, class W> int WGraph<T, W>::size() { return graph.size(); }
template <class T, class W> void WGraph<T, W>::clear() { graph.clear(); }

template <class T, class W> void WGraph<T, W>::addVertex(T v) {
  bool found = false;
  for (auto vp : graph)
    if (vp.first == v)
      found = true;

  if (!found)
    graph.push_back({v, {}});
}

template <class T, class W> void WGraph<T, W>::addVertices(vector<T> vlist) {
  for (size_t i = 0; i < vlist.size(); i++)
    addVertex(vlist[i]);
}

template <class T, class W> int WGraph<T, W>::getVertexPos(T v) {
  for (size_t i = 0; i < graph.size(); i++)
    if (graph[i].first == v)
      return i;

  return -1;
}

template <class T, class W> int WGraph<T, W>::getEdgePos(T v, T vt) {
  int vpos = getVertexPos(v);
  if (vpos >= 0) {
    vector<pair<T, W>> adjlist = graph[vpos].second;
    for (size_t i = 0; i < adjlist.size(); i++)
      if (adjlist[i].first == vt)
        return i;
  }
```

```
  return -1;
}

template <class T, class W> void WGraph<T, W>::addEdge(T v, T vt, W w) {
  addVertex(v);
  addVertex(vt);
  int pos = getVertexPos(v);

  if (getEdgePos(v, vt) == -1)
    graph[pos].second.push_back({vt, w});

  if (!directed) {
    pos = getVertexPos(vt);

    if (getEdgePos(vt, v) == -1)
      graph[pos].second.push_back({v, w});
  }
}

template <class T, class W> void WGraph<T, W>::addEdge(T v, pair<T, W> p) {
  addEdge(v, p.first, p.second);
}

template <class T, class W> void WGraph<T, W>::addEdge(pair<T, pair<T, W>> p) {
  addEdge(p.first, p.second);
}

template <class T, class W>
void WGraph<T, W>::addEdges(T v, vector<pair<T, W>> vt) {
  for (size_t i = 0; i < vt.size(); i++)
    addEdge(v, vt[i]);
}

template <class T, class W>
void WGraph<T, W>::addEdges(vector<pair<T, pair<T, W>>> elist) {
  for (size_t i = 0; i < elist.size(); i++)
    addEdge(elist[i].first, elist[i].second);
}

template <class T, class W> void WGraph<T, W>::deleteEdge(T v1, T v2) {
  int pos = getVertexPos(v1);
  int epos = -1;
  if (pos != -1) {
    epos = getEdgePos(v1, v2);
    if (epos != -1)
      graph[pos].second.erase(graph[pos].second.begin() + epos);
  }

  if (!directed) {
    pos = getVertexPos(v2);
    if (pos != -1) {
      epos = getEdgePos(v2, v1);
      if (epos != -1)
        graph[pos].second.erase(graph[pos].second.begin() + epos);
    }
  }
}

template <class T, class W> void WGraph<T, W>::deleteEdge(pair<T, T> p) {
  deleteEdge(p.first, p.second);
}

template <class T, class W> void WGraph<T, W>::deleteEdges(T v, vector<T> vt) {
  for (size_t i = 0; i < vt.size(); i++)
    deleteEdge(v, vt[i]);
}

template <class T, class W>
void WGraph<T, W>::deleteEdges(vector<pair<T, T>> ep) {
  for (size_t i = 0; i < ep.size(); i++)
    deleteEdge(ep[i].first, ep[i].second);
}
```

```
template <class T, class W> void WGraph<T, W>::deleteVertex(T v) {
  int pos = getVertexPos(v);
  if (pos == -1)
    return;

  graph.erase(graph.begin() + pos);
  for (size_t i = 0; i < graph.size(); i++) {
    int vpos = -1;
    for (size_t j = 0; j < graph[i].second.size(); j++)
      if (graph[i].second[j].first == v)
        vpos = j;
    if (vpos != -1)
      graph[i].second.erase(graph[i].second.begin() + vpos);
  }
}

template <class T, class W>
vector<pair<T, W>> WGraph<T, W>::getAdjacentList(T v) {
  int pos = getVertexPos(v);
  if (pos != -1)
    return graph[pos].second;

  vector<pair<T, W>> empty;
  return empty;
}

template <class T, class W>
vector<pair<T, pair<T, W>>> WGraph<T, W>::getEdgeList() {
  vector<pair<T, pair<T, W>>> elist;
  for (size_t i = 0; i < graph.size(); i++) {
    T v1 = graph[i].first;
    vector<pair<T, W>> Adj = graph[i].second;
    for (size_t j = 0; j < Adj.size(); j++) {
      pair<T, W> v2 = Adj[j];
      elist.push_back({v1, v2});
    }
  }
  return elist;
}

template <class T, class W> vector<T> WGraph<T, W>::getVertexList() {
  vector<T> lst;
  for (size_t i = 0; i < graph.size(); i++)
    lst.push_back(graph[i].first);

  return lst;
}
```