1. Babyfaces vs. Heels. [10 marks total]

There are two types of professional wrestlers: "babyfaces" (the good guys) and "heels" (the bad guys). Between any pair of wrestlers there may or may not be a rivalry. Suppose there are n wrestlers and we have a list of r pairs of wrestlers for which there are rivalries.

(a) Give an algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

This problem can be solved by representing it as a graph. Each vertex is a wrestler, and each edge is a rivalry.

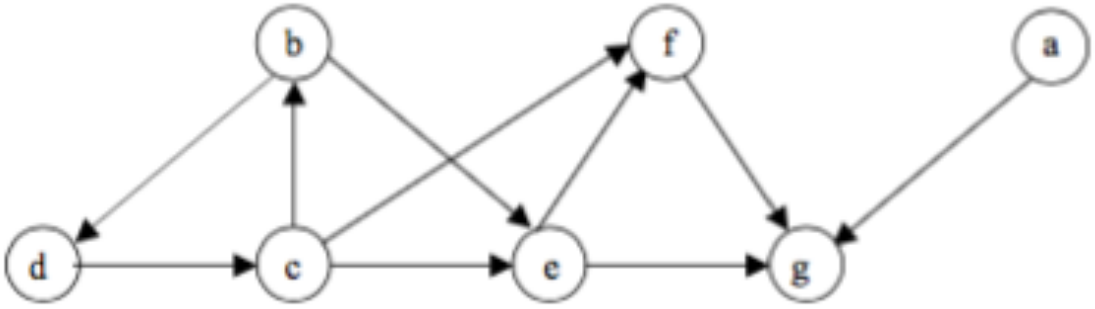
```
vertex wrestlers[n];
edge rivalries[r];
boolean vertexVisited[n] = { false };

// Visit all n vertices
while (vertexVisited[0…n] == false) {
   breadthFirstSearch(wrestlers);
}

// Assign wrestlers[0] = BabyFaces
// Assign wrestlers[0].neighbors = Heels

// If any heel has a heel neighbor, designation is not possible
if (wrestlers[n] == Heels && wreslters[n].neighbor == Heels) {
   cout << "Designation not possible" << endl;
   break;
}
```

(b) What is the worst-case run time of your algorithm?

Same as breadth-first search, *O(n+r)*, where n is the number of vertices and r is the number of edges.

2. DFS of a Directed Graph. [10 marks total]

Given the following directed graph, traverse the graph by depth-first search (DFS) starting at vertex a and visiting vertices in alphabetical order.



(a) Label each vertex with its discovery and finish time

No other vertices are discoverable, besides a-g.

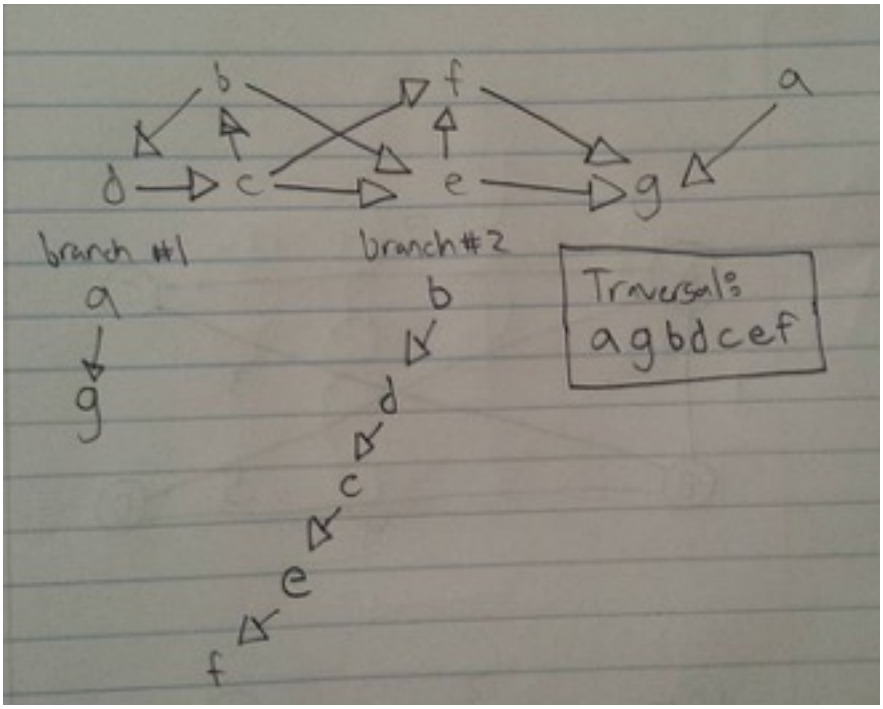
Vertex	Discovery	Finish Time
a	0	2
b	∞	∞
d	∞	∞
c	∞	∞
e	∞	∞
f	∞	∞
g	1	1

If you'd like us to continue on with a new branch, starting with the next alphabetical letter 'b', the following would be the answer:

Vertex	Discovery	Finish Time
a	0	2
b	0	8
d	1	7
c	2	6
e	3	5
f	4	4

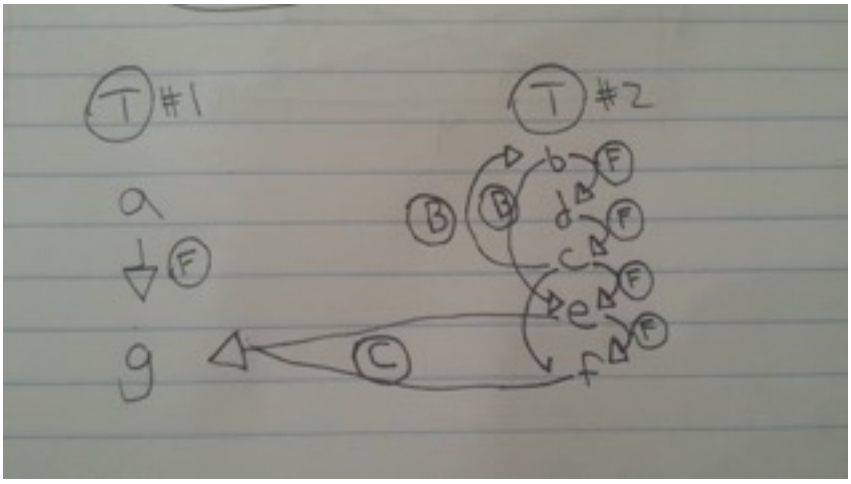
Vertex	Discovery	Finish Time
g	1	1

(b) Construct the corresponding DFS forest in the standard form.



(c) Classify each edge as a tree (T), forward (F), back (B), or cross (C) edge.

Labeled by circles:



3. Connected Components. [10 marks total]

Show how to modify depth-first search (DFS) of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify DFS so that it assigns to each vertex v and integer label between 1 and k, where k is the number of connected components of G, such that the labels of two vertices u and v are equal if and only if u and v are in the same connected component.

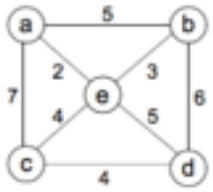
4. Minimum Spanning Tree (MST) Algorithms. [20 marks total]

All answers in question 4 will use the following pseudo-code API:

vertex From(vertex To, int Weight)

if Weight is blank, no valid path is present in graph / answer not yet available

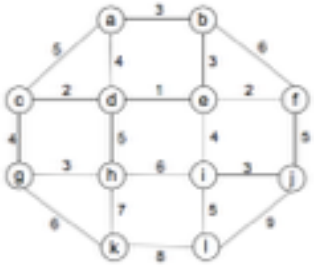
(a) Apply Prim's algorithm to the following graph. Include in the priority queue all the vertices not already in the tree.



Minimum spanning tree: a(e, 2), e(b, 3), e(c, 4), c(d, 4).

Vertices	Priority Queue - Vertices not in tree
a(,)	b(a, 5), c(a, 7), d(a,), e(a, 2)
e(a, 2)	b(e, 3), c(e, 4), d(e, 5)
b(e, 3)	c(e, 4), d(e, 5)
c(e, 4)	d(c, 4)
d(c, 4)	

(b) Now, apply Prim's algorithm to the following graph. Include in the priority queue only the fringe vertices (the vertices not in the current tree which are adjacent to at least one tree vertex).

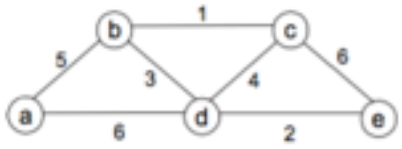


Minimum spanning tree: a(b, 3), b(e, 3), e(d, 1), d(c, 2), e(f, 2), e(i, 4), i(j, 3), c(g, 4), g(h, 3), i(l, 5), g(k, 6).

Vertices	Priority Queue - Fringe Vertices
a(,)	b(a, 3), c(a, 5), d(a, 4)

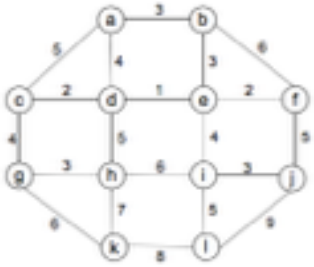
Vertices	Priority Queue - Fringe Vertices
b(a, 3)	c(a, 5), d(a, 4), e(b, 3), f(b, 6)
e(b, 3)	c(a, 5), d(e, 1), f(e, 2), i(e, 4)
d(e, 1)	c(d, 2), f(e, 2), h(d, 5), i(e, 4)
c(d, 2)	f(e, 2), g(c, 4), h(d, 5), i(e, 4)
f(e, 2)	g(c, 4), h(d, 5), i(e, 4), j(f, 5)
i(e, 4)	g(c, 4), h(d, 5), l(i, 5), j(i, 3)
j(i, 3)	g(c, 4), h(d, 5), l(i, 5)
g(c, 4)	h(g, 3), l(i, 5), k(g, 6)
h(g, 3)	l(i, 5), k(g, 6)
l(i, 5)	k(g, 6)
k(g, 6)	

(c) Apply Kruskal's algorithms to find a MST in the graph below and also to the graph in part (b).



Minimum spanning tree: b(c, 1), d(e, 2), b(d, 3), a(b, 5).

———————



Minimum spanning tree: d(e, 1), d(c, 2), e(f, 2), a(b, 3), e(b, 3), g(h, 3), i(j, 3), c(g, 4), e(i, 4), i(l, 5), g(k, 6).

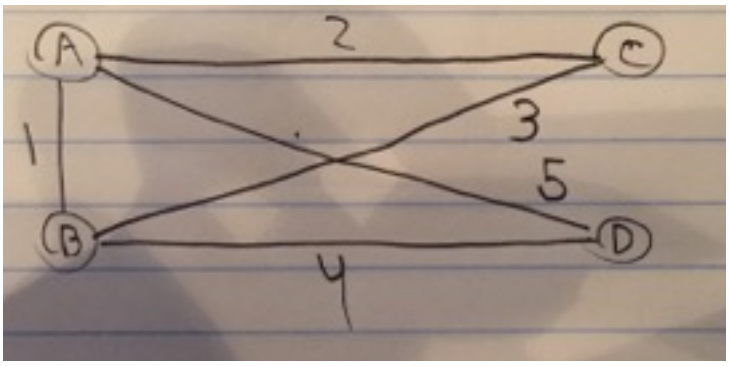
5. Second-best Minimum Spanning Tree (MST). [15 marks]

Let G = (V, E) be an undirected, connected graph whose weight function is w : E → R, and suppose that |E| ≥ |V | and all edge weights are distinct.
We define the second-best minimum spanning tree as follows. Let T be the set of all spanning trees of G, and let T be a MST of G. Then a *second-best minimum spanning tree* is a spanning tree T such that $w(T) = \min_{T'' \in (T - T')}\{w(T'')\}$.

(a) Show that the MST is unique, but that the second-best MST need not be unique.

We can see by example that a graph G with a unique minimum spanning tree may have two (or more) second-best minimum spanning trees, which is perfectly ok.

Heres an example, which a unique minimum spanning tree, but with two second-best minimum spanning trees:



(b) Let T be the MST of G. Prove that G contains edges (u,v) ∈ T and (x,y) ∈ T such that T – {(u, v)} ∪ {(x, y)} is a second-best MST of G.

Our second-best spanning tree contains a cycle and one of the edges in the cycle is not a part of the minimum spanning tree. We know that (x, y) is part of the minimum spanning tree, and has a higher weight, otherwise (u, v) would be part of the cycle to give the minimum spanning tree. Our minimum spanning tree without (x, y), but with (u, v) is also a spanning tree since (u, v) and (x, y) are in the same cycle. Let this edge be (x,y). By uniqueness of a minimum spanning tree, we can conclude this try because only one edge differs, which is (x,y) and (u, v).

(c) Given an efficient algorithm to compute the second-best MST of G.

```
int secondBestMSTWeight = infinity; // Store second best MST weight
secondBestMST = null;

MST = minimumSpanningTree(G); // compute MST of graph G
// for each edge in MST, calculate MST for graph G to edge e
foreach (edge e : MST) {
        partialMST = minimumSpanningTree(e);
        // If weight of partial MST (G to e) < weight of MST, remember
        if (partialMST.weight < secondBestMSTWeight) {
                secondBestMSTWeight=partialMST.weight;
                secondBestMST = partialMST;
        }
}
return secondBestMST;
```

6. Variations of Dijkstra's Algorithm. [15 marks]

Explain what adjustments if any need to be made in Dijkstra's algorithm and/or in an underlying graph to solve the following problems.

(a) Solve the single-source shortest-paths problem for directed weighted graphs.

Dijkstra's Algorithm can solve single-source shortest-path problems for directed weighted graphs without modifications under the assumption that all weighted edges are non-negative. Negative numbers do not work with Dijkstra's Algorithm. You can normalize the weights in order to counter this negative weight limitation.

(b) Find a shortest path between two given vertices of a weighted graph of digraph. (This variation is called the *single-pair shortest-path* problem.)

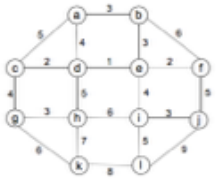
Dijkstra's Algorithm can solve this single-pair shortest-path problem by adding one conditional. If the single-pair shortest-path we want to find is from U to V in graph G, we can run Dijkstra's Algorithm on graph G, starting at vertex U. Once vertex V is found (our conditional), we can exit Dijkstra's Algorithm and our single-pair shortest-path problem is solved.

(c) Find the shortest paths to a given vertex from each other vertex of a weighted graph or digraph. (This variation is called the single-destination shortest-paths problem.)

Dijkstra's Algorithm can solve this single-destination shortest-paths problem by adding in two *steps.* The first step is to reverse the direction of all edges of graph G. Next, Dijkstra's Algorithm must be performed as usual, but this time starting at the destination V, rather than the source U. To keep track of the shortest-path from all sources to a single-destination, we simply keep a log of the current shortest path from U to V, V being the starting source of our reversed graph G (or rather, the destination in theory) once each shortest path to U is found.

Summarized: reverse direction of edges and run Dijkstra's Algorithm from destination to source.

7. Dijkstra's Algorithm graph. [10 marks]
Trace the Dijkstra's algorithm on the graph in Question 4(b) with vertex a as the source.



L(v) depicts the minimum path length of edges from source to v.

Shortest Path {Set}	L(a)	L(b)	L(c)	L(d)	L(e)	L(f)	L(g)	L(h)	L(i)	L(j)	L(k)	L(l)
∅	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{ a }		3	5	4	∞	∞	∞	∞	∞	∞	∞	∞
{ a, b }			5	4	6	9	∞	∞	∞	∞	∞	∞
{ a, b, d }			5		5	9	∞	9	∞	∞	∞	∞
{ a, b, d, c }					5	9	9	9	∞	∞	∞	∞
{ a, b, d, c, e }						7	9	9	9	∞	∞	∞
{ a, b, d, c, e, f }							9	9	9	12	∞	∞
{ a, b, d, c, e, f, g }								9	9	12	15	∞
{ a, b, d, c, e, f, g, h }									9	12	15	∞
{ a, b, d, c, e, f, g, h, i }										12	15	14
{ a, b, d, c, e, f, g, h, i, j }											15	14
{ a, b, d, c, e, f, g, h, i, j, l }											15	
{ a, b, d, c, e, f, g, h, i, j, l, k }												

8. Disjoint Sets. [10 marks]

Professor Gump suggests that it might be possible to keep just one pointer in each set object, rather than two (head and tail), while keeping the number of pointers in each list element at two. Show that the professor's suggestion is well-founded by describing how to represent each set by a linked list such that each operation Make-Set, Union, and Find-Set has the same running time as the operations using the standard linked list representation of disjoint sets. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect.

Implementing Professor Grump's suggestion of a single pointer set, and double pointer elements results in a set object whose pointer points to the list representative linked list head. Each element will be in a linked list, with one pointer pointing to the list representative of the set, and a second pointer pointing to the next element of the set.

When using this linked list representation of a disjoint set we obtain a running time complexity of O(1) for both Make-Set( ) and Find-Set( ).

Make-Set(Element e): O(1), linear time

Makes a new linked list whose only element is < e >. e.next is initialized to null.

(Weighted-union heuristic: initialize <int size> to 1)

Find-Set(Element e): O(1), linear time

Returns the list representative pointer of element < e >.

Union(Set s, Set t): O(L), where L is the length of the set appended

Set last element of Set < s > (s.next) to list representative of Set < t >. This simply means that we are appending Set < t > to Set < s >.

We must be sure to update all list representative pointers of the elements originally in Set < t > to the list representative of Set < s >

This takes linear time O(L), where L is the length of Set < t >.

(*Weighted-union heuristic: append Set whose <int size> variable is smaller. Update merged Set's <size> variable to a sum of t.size and s.size*)

(See Below For Details On Special Consideration Which Must Be Taken To Ensure A Weighted-Union Heuristic)

Weighted-union heuristic:

The weighted-union heuristic means to keep track of the number of objects in our set, and to append the shorter of the two lists during our Union(Set s, Set t) algorithm.

This is favorable, because as you can see from our O(L) running time, the running time is determined solely based on the length of the set we are appending. If we append the smaller set, our running time takes the smallest possible impact.

In order to implement weighted-union heuristic, we must simply create an <int size> property for our Set object and increment it by L when performing Union(Set s, Set t), where L is the length (size) of the Set which we are appending. Obviously, we can simply set our new Sets size variable by first adding the previous size variables of our merged sets.

Make-Set(Element e) must also initialize our new <int size> property to 1.