## Problem 1:

*Purpose: Learn about Euler tours.* Please solve Problem 22-3 a (6 points) on page 623.

Euler tours (22-3): An Euler tour of a strongly connected, directed graph G = (V, E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Euler tour <=> in-degree(v) = out-degree(v) for each vertex v ∈ V

   Ans:

   P1: Euler tour => in-degree(v) = out-degree(v) for each vertex v ∈ V

   Euler tour can be represented by a set of simple cycles (edge=disjoint). For each simple cycle, every vertex will have one edge in and one edge out. Hence, for the simple cycle, the in-degree(v) = out-degree(v). Then the set of simple cycles form the Euler tour. We treat simple cycle as a vertex, and Euler tour as a cycle. Then each simple cycle will have one edge in and one edge out. Based on this, we get the in-degree(v) = out-degree(v) for all vertex in Euler tour.

   P2: in-degree(v) = out-degree(v) for each vertex v ∈ V => Euler tour

   In order to prove this, we need to do construction. First, we start from one edge and from one vertex to another. By this way, we follow the unvisited edge until we get to the first vertex and form a cycle. Then we find another unvisited edge as a new start. Do the same steps to get another cycle. At the end, if all the edges are visited and since the graph if fully connected, we can get the result that this is a Euler tour.

b. Describe O(E) time algorithm to find an Euler tour if exists

   Ans: here we use BFS to solve, here for each edge we just visited once

   Pseudo code:

```
Input: edgeList [[v1,v2], [v0, v4]]        // for verList[i], it means vertex vi is connect to vj
tour = []
curVer = edgeList[0][0]
tour.append(startVer)
While not empty(edgeList):
    for edge in edgeList:
        if curVer in edge:
            if edge[0] == curVer: curVer = edge[1]          // get next vertex
            elif edge[1] == curVer: curVer = edge[0]
        graph.remove(edge)
        tour.append(curVer)
        break
    else: return False        // no tour

return tour
```

## Problem 2

*Reinforce your understanding of minimum spanning trees* (9 points). Please sole problem 23-4 on page 641. You do NOT have to describe the most efficient implementation of each algorithm.

Ans:

a. T is the set of edges. And edge time we remove one most weighed edge of T when T – {e} is still a connected graph. Hence, at the end T is a spanning tree. Also, since the most weighted edges are removed, we can ensure the total weight of spanning tree is minimum.
b. The loop takes edges in arbitrary order which will lead the higher weight been added first, the condition T ∪ {e} only ensure there is no cycle in this set. Hence, this algorithm is incorrect.
c. This algorithm is different from the second one, because when there's a cycle, we do remove the larger edge in the cycle graph which ensure the weight of subproblem to be minimal. Then at the end, we not only ensure there's no cycle in this tree, but also we ensure the total weight is minimal.

## Problem 3

*Purpose: Reinforce your understanding of minimum spanning trees, practice algorithm design*. Suppose you are given a connected graph G=(V,E), with edge costs that you may assume are all distinct. A particular edge e*=(u,v)∈E of G is specified.

a) (6 points) Show that e* does not belong to any MST of G if and only if v and u can be joined by a path consisting entirely of edges that are cheaper than e*.

b) (6 points) Give an algorithm with running time O(|V|+|E|) to decide whether e* is contained in any minimum spanning tree of G. Tip: use a).

Ans:

a. Firstly, suppose we have a path P which connect v – w with edges cheaper than e, then when e is added to this path, there will be a cycle, and e is the greatest edge which we can delete. Hence, when we have a MST which contains e, we can always delete e by a cheaper edge.
b. Here we use MST cycle property. It said that "or any cycle C in the graph, if the weight of an edge e of C is larger than the weights of all other edges of C, then this edge cannot belong to an MST." So first, we run DFS from one of the end-point (either u or v) of edge E, and only go to the edge which has less weight than E. If at the end, u and v get connected, it means E cannot be part of MST. Because there exists a cycle with E. And the previous connection is smaller then E. Based on we describe in part a. but if at the end of dfs, u and v stay disconnected, this mean E could be part of MST. We run the DFS algorithm which takes O(|V| + |E|)

## Problem 4

*Purpose: Reinforce your understanding of Dijkstra's shortest path algorithm, and practice algorithm design (6 points).* Suppose you have a weighted, undirected graph $G$ with positive edge weights and a start vertex $s$. Describe a modification of Dijkstra's algorithm that runs (asymptotically) as fast as the original algorithm and assigns a binary value usp[$u$] to every vertex $u$ in $G$, so that usp[$u$]=1 if and only if there is **a unique shortest path from $s$ to $u$**. We set usp[$s$]=1. In addition to your modification, be sure to provide arguments for both the correctness and time bound of your algorithm, and an example.

Ans:

Because when we start from s and each time, we use shortest path to reach some vertexes, we set usp to be 1. So we would use BFS to solve this problem.

pseudo code:

```
Input:   graph = (v, e) undirected, positive edge weight W = [We]
usp = [0] * len(v)
usp[s] = 1
for all u in V:
        dist[u] = MAX_VALUE
        prev[v] = None
dist[s] = 0
H = heap(V)
while not empty(H):
        u = H.popmin()
        for all edges (u, v) in E:
                if dist(v) > dist(u) + w(u, v):
                        dist(v) = dist(u) + w(u, v)
                        prev(v) = u
                        decreaseKey(H,v)
                        usp[v] = 1
```

The time complexity: O(|E| + |V|log(V))    E is for edge, V is for Vertex, VlogV is for heap

The example would be:

graph: [[A, B, 4], [A, C, 2], [B, C, 1], [C, E, 5], [B, E, 3], [B, D, 2], [D, E, 1]]

suppose we start from A, all other node dist is MAX_VALUE

| dist | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 | 0 |
| B | MX | 4 | 3 | 3 | 3 |
| C | MX | 2 | 2 | 2 | 2 |
| D | MX | MX | MX | 6 | 6 |
| E | MX | MX | 7 | 7 | 7 |

At the end, all ele in usp is set to be 1. Because every one has a unique path from start to end

## Problem 5

*Purpose: Reinforce your understanding of Dijkstra's shortest path algorithm, learn about multiple solutions, and practice algorithm design and Implementation (12 points).* In the usual formulation of Dijkstra's algorithm, the number of edges in the shortest (= lightest) path is not a consideration. Here, we assume that there might be multiple shortest paths. Implement an algorithm that takes as input an undirected graph **G = (V, E)**, a nonnegative cost function **w** on **E**, a source vertex **s** and a destination vertex **t**, and produces a path with the fewest edges amongst all shortest paths from **s** to **t**. If there are multiple such shortest paths with the fewest edges, your algorithm should output the *unique* path with the lexicographically smallest sequence of vertices amongst all such paths. Please describe (6 points) and implement your algorithm (6 points). **Your implementation should follow the description of Dijkstra's algorithm given in class.**

Ans:

Describe:

Here we first parse input and build a graph use adjacent list. And then we do Dijkstra algorithm to find the shortest path. Each time when we find multiple shortest path, we add it to candidate path. And when the Dijkstra hit to the end Vertex. We would do the BFS to find the path with lexicographically smallest sequence of vertices.