Name: Yinsheng Dong
Student Number: 11148648
NSID: yid164
Assignment 1

1. Kleinberg and Tardos p. 109 #8

   Proof by contradiction:

   Assume to the contrary that there exists a positive natural number c so that for all connected graphs G, it is the case that diam(G)/apd(G) > c

   We have that diam(G) = max{dist(u, v) : u, v $\in$V} and apd(G) <= max{ dist(u, v) : u, v $\in$V} = diam(G).

   Since we've known that diam (G) != apd(G), diam(G)/apd(G) !=1.

   We got that (diam(G))/(apd(G)) > 1 which c = 1. Which contradict the problem given which is diam(G)/apd(G) <= c.

   We now proved that the statement is false.

2. Kleinberg and Tardos p. 110 #10

   a. Solution:

   To solve this problem, we need to compute the number of shortest paths from v to w.

   First, we can perform a BFS from v, the algorithm designed:

```
BFS(u)
    Set Discovered[u] = true and Discovered[v] = false for the node we need to find
    Set the number of shortest path n = 0
    Initialize L[0] to consist of the single element u
    Set the layer counter i = 0;

    While L[i] is not empty
        for each node u is in L[i]
            if Discovered[v] = false then
                i+1
            else if Discovered[v] = true
                n = cout L[i-1]
            endif
        end for
        i++
    end while
    return n
```

   From his algorithm, we got that for each node v in L1, the complexity is 1, and for the at most the degree of Ln, we have the sum of the degrees in the O(m), so the overall running time complexity is O(n+m).

   b. For all n = 3q+1 where q is a positive integer, there exists a graph with n vertices and two named vertices, u and v, that has $2^{(n-1)/3}$ shortest paths from u to v.

Direct Prove:
If we want to have multiple shortest paths from u to v in a graph G, G at least has 4 vertices which contains u and v, and it has at least 4 edges, and the shortest paths has 2.

We assume p is the number of shortest paths, so we have known $n = 3q+1$, $p = 2^{(n-1)/3}$ and q is the positive integer. We got $n>=4$ and $p>=2$.

Then using induction to prove.

Basic case: q = 1, then n = 4, p = 2, that shows the statement is true.

Inductive step: q1 = q+1 = 2. Then n1 = n+3 = 7, and p1 = 4 which is also true.

To conclude, we got the statement is true.

3. In a directed graph G = (V, E), a vertex v is called middle if and only if for every vertex x in V either there exists a directed path from v to x, or there exists a directed path from x to v.
   a. Given a Directed acyclic graph G and a vertex u in V, provide an O(|V|+|E|) time algorithm for determining whether or not vertex u is middle in G.

```
Determine Algorithm for (a)

Set edge(u,v) = false
     //(which v is the node in the loop, and u is the given node)

For each node v in the G
    if u != v && (edge(u,v) == false || edge (v,u) == false)
        return false
    endif
Endfor
return true
```

   Explanations: The edge(u,v) function is using to determine u and v has edge. The algorithm provides O(|V|+|E|)

   b. Given an acyclic directed graph G = (V, E) provide an O(|V|+|E|) time algorithm for computing all the middle vertices of G.

```
Algorithm for (b)
edge(u, v) is the function to determine if u and v has edge
i = 0
j = 0
for(vi in G)
    if edge (vi, vj) || edge (vj, vi) || i == j
        if (vj is last vertex of G)
            return i
            i++
        endif

        j++
    else
        i++
    endif
endfor
```

This algorithm is using a for-loop to control the vertex, to compare if vi and vj have edge or not. The max complexity is O(|V|+|E|), since we only need to know if the vertex v does have edges with all other vertices, if it does, it must be a middle point, skip it otherwise. Since we the graph is a DAG, so we need determine which edge(u, v) or edge (v, u).

c. Given a general directed graph G = (V, E) provide an O(|V|+|E|) time algorithm for computing all the middle vertices of G.

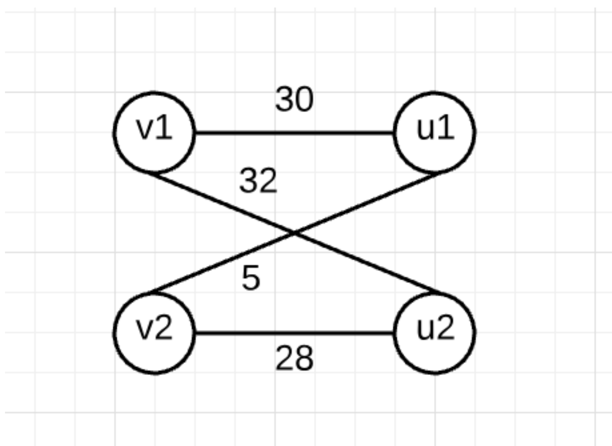
```
Algorithm for (c)
edge(u, v) is the function to determine if u and v has edge
i = 0
j = 0
for(vi in G)
    if edge (vi, vj) || edge (vj, vi) || i == j
        if (vj is last vertex of G)
            return i
            i++
        endif

        j++
    else
        i++
    endif
endfor
```

This algorithm is similar like (b), it is using a for-loop to control the vertex, to determine if they have edges or not. Time complexity is O(|E| + |V|)

4. Let G = (V∪U, E) be a bipartite graph such that each edge e ∈ E has an associated weight w(e). A matching for G is a subset M ⊆ E such that no two edges in M share a common vertex. The weight of M is w(M) = ∑e⊆M w(e).
A greedy algorithm for bipartite matching could start with an empty matching M, and then repeatedly add the largest weight edge that does not share a vertex with an edge already included in M

a. Given an example edge weighted bipartite graph for which the above greedy algorithm will fail to find the maximum weight matching



Explain: w(v1, u1) = 30, w(v1, u2) = 32, w(v2, u1) = 5, w(v2, u2) = 28
In the situation above will make the greedy algorithm fail to find the maximum weight matching
When the greedy algorithm starts at w(v1, u2) = 32, then w(v2, u1) = 5,
We got the maximum weight from greedy =37.

However, the best match should be w(v1, u1) = 30, w(v2, u2) = 28.
The max weight = 30+28=58.

b. For bipartite graphs in which all the edge weights are distance and each is power of w, prove that the above greedy algorithm always produces the maximum weight match.
Inductive Proof : Since the edge weights are distinct and each is a power of 2, and the greedy algorithm repletely add the largest weight edge.

Basic case: We assume there has 4 vertices in the graph, the largest weight edge is 2^n, and all other weight edge power i<n.
We know that 2^n+2^1 > 2^(n-1)+2^(n-2).

Inductive step: we add one more vertex and the largest weight edge is 2^(n+1), then we found that 2^(n+1)+2^(1)+2(2) > 2^n+2(n-1)+2(n-2).

To conclude, we can get that the greedy algorithm picked the largest one.