Q1:

We will design the following dynamic programming algorithm:

```
DP-findPath(G, P, total, v, t):
    // G: graph containing E and V.
    // P: current path to v.
    x ← after(v)[0]
    newTotal ← total + w(v, x)
    for u in after(v) do:
        if u is t then
            newTotal ← newTotal + w(v, u)
            P.push(u)
            break
        else if u is not sink then
            P1 ← P
            P1.push(u)
            n ← DP-findPath(G, P1, total + w(v, u), u, t)
            if n ≤ newTotal then:
                newTotal ← n
                P ← P1
        else
            P.pop().
    done
    if P.back() is v then
        report "no" // we cannot find next vertex after v.
    return newTotal

findPath(G, s, t):
    P ← empty vector of optimal path
    topologically sort G // G is DAG
    op ← DP-findPath(G, P, 0, s, t):
    report op, P
```

Let after(v) indicates the set of vertices depends on v. (after v)

The topologically sort for DAG is $O(m+n)$ as the lecture slides say.

Our claim that does not need a proof:

For all vertices u that is after v, there are 3 cases.

① u is target t:

path ends, we get a possible path.

② u is sink:

path does not exist

③ u is on path:

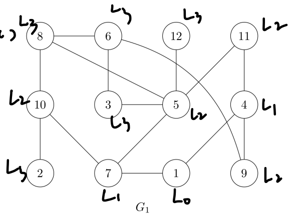
new total will add up $w(v, u) + \min(\text{weight from } u \text{ to } t)$

This is what the loop does in my algorithm. And at the end of each iteration, we will get the minimum weight of path if we follow that u. Therefore, we can update the minimum path so far untill it loopsthrough all neighbors of v. Thus, if such paths exist, we will calculate the weight of each one of them.

Thus, for the worst case, if we start at s, we will go through all paths starting with s, just as ssspDAG. However, the vertices may not only be traversed once, since we know $m \geq n$. The worst case for a single vertex is that it has m edges that direct to it and it directs to the rest $(n-1)$ vertices. Hence, our time complexity $T(n) = O(m+n) + (n-1)\,O(m) \in O(mn)$

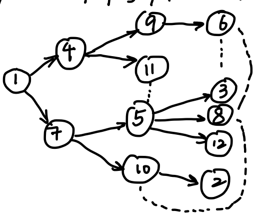
Q2:

(a)



G_1

output: 1 4 7 5 9 10 11 2 3 6 8 12

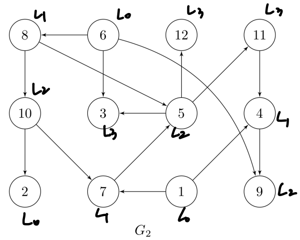


Trace:

- mark 4 and 7 as visited, mark 1 as finished.
- mark 9, 11, 5 and 10 as visited, mark 4 and 7 as finished.
- mark 2, 8, 3, 6 and 12 as visited, mark 9, 11, 5 and 10 as finished
- mark 2, 8, 3, 6 and 12 as finished.

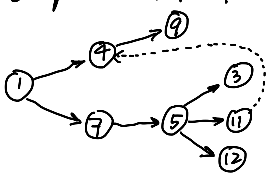
(b) No, it's not bipartite, since there are cross edges in the same level of the graph.

(c)



G_2

Output: 1 4 7 5 9 3 11 12 2 6 8 10



Trace:

- mark 4 and 7 as visited, mark 1 as finished.
- mark 5 and 9 as visited, mark 4 and 7 as finished.
- mark 3, 12 and 11 as visited, mark 5 and 9 as finished
- mark 3, 12 and 11 as finished
- mark 2 as finished
- mark 8 as visited, mark 6 as finished
- mark 10 as visited, mark 8 as finished
- mark 10 as finished

(d) Trace:

in order, mark 1, 4, 9, 6, 3, 5, 8, 10, 2 as visited
mark 2 as finished, mark 7 as visited. mark 7 as finished.
in order, mark 10, 8 as finished, mark 11 as visited
mark 11 as finished, mark 12 as visited. mark 12 as finished
in order, mark 5, 3, 6, 9, 4, 1 as finished.

(e) Trace:

in order, mark 1, 4, 9 as visited. mark 9 as finished, mark 4 as finished
mark 7 as visited, mark 5 as visited, mark 3 as visited.
mark 3 as finished, mark 11 as visited, mark 4 as visited.
mark 4 as finished, mark 11 as finished, mark 12 as visited, mark 12 as finished
mark 5 as finished, mark 7 as finished, mark 1 as finished.
mark 2 as visited, mark 2 as finished.
in order, mark 6, 8, 10 as visited and then mark 10, 8, 6 as finished.

(f) G_2 is not strongly connected, since in the BFS tree in (c), there is no node 6 in the sub-tree of 1, which indicates that there is no path from 1 to 6.

Q3:

(a) We will design the following algorithm:

① First, we find the row # of s in G, denoted as i.

② We check G[i]. Sum up G[i], check if the total is 1. If there are more than 1 neighbors or no neighbor, return "no". else let j be the first non-zero entry in G[i] and move forward to ③.

③ We check G[j]. Sum up G[j], check if the total is 2. If there are exactly 2 neighbors, then let k be the first non-zero entry and not equal to i, proceed ④. Else return "no".

④ We check G[k] to see if there is only one zero except from G[k][k]. Let # of zero entries be l. If $l \neq i$, return "no", else return "yes".

Analysis:

We first check if sting has degree 1 and then check if tail has degree 2. After this, we check if all other vertices of body connect to tail or sting. Thus, the graphs that satisfy the definition of scorpion graph should return "yes".

Time complexity analysis:

Checking a row of $n \times n$ matrix requires $O(n)$. We have 3 checks, thus $T(n) = 3 \cdot O(n) \in O(n)$

(b) We will design the following algorithm:

① First, find the row # of b, denoted as i.

② Check # of zeros of each entry of $G[i]$, if $\# \neq 1$, return "no".
During iterations, let m be the first zero entry except for $G[i][i]$.

③ use the algorithm in (a) to check if m is a sting, return the result from (a).

Analysis:

For ① and ②, time complexity is $O(n)$ since we will iterate through n entries.

Thus, $T(n) = O(n) + O(n) \in O(n)$

(c) We will design the following algorithm:

We will use AlgoA to refer to algorithm in (a), and AlgoB for (b)

① We pick a vertex v_i, calculate the degree of v_i by summing up $G[i]$ (start with v_0) $O(n)$

② Now, there are 4 cases:

1) if degree is 1, then we use AlgoA on v_i, if the result is true, return true. $O(n)$
Else use AlgoB on v_i's neighbor. return the result.

2) if degree is 2, then v_i is tail or body.
Then, let m be v_i's first neighbor and n be v_i's second neighbor. $2O(n)$
If v_i is tail, AlgoA(m) and AlgoB(n) should be XOR, return true

3) if degree is $n-2$, then return AlgoB(v_i) $O(n)$

4) v_i can only possible be feet.

③ if 4), we will divide $G[i]$ into two groups.
For every entry in $G[i]$, if it is 1, it goes to G_1 else it goes to G_2 $O(n)$

Then, we know for sure that body should be in G_1 and sting should be in G_2.

④ We will then do the following iteration to make G_1 and G_2 smaller,

1) $a \leftarrow G_1$.back() $b \leftarrow G_2$.back()
2) if $G[a][b]$ is 0, G_1.popback, G_2.popback(), repeat 1)
3) else G_2.popback(), $c \leftarrow G_2$.back()
4) if $G[a][c] = G[b][c] = 1$, repeat 3)
5) elseif $G[a][c]=1$, $G[b][c]=0$, G_1.pushfront(a), G_2.pushfront(c), repeat 1)
6) else if $G[a][c]=0$, $G[b][c]=1$, G_1.pushfront(b), G_2.pushfront(c), repeat 2)
7) else G_2.pushfront(c), repeat 3)
8) if 7) is repeated 3 times. At the end of the 3-th time, repeat 1).

⑤ Now, if G_1.size() = 1, AlgoB($G_1[0]$)
if G_2.size() = 1, AlgoA($G_2[0]$).

By this algorithm, we can make sure that all cases are handled and vertices are checked. It will end since if it goes to case 4), it will eventually terminate since G_1 and G_2 will be reduced to 1.

Time complexity:

For ④ iteration, since we will eventually reduce at least one set to 1. Thus, we can be sure that there will be at most $n-2$ iterations. Thus, it is $O(n)$

Therefore, $T(n) \in O(n)$.

Q4:

(a) For the purpose of contradiction, assume the MST of G is T and T is not an MHEST. Let H be an MHEST.

Let $e \in T$ be the heaviest edge in T. Thus, $e \notin H$.

If we insert e into S. Assume $(v_1, v_2) = e$. Then, this would create a cycle $v_1 \cdots v_2 v_1$.

We can see that e is the heaviest edge in the cycle, since e is the heaviest edge in T, for $\forall H$, e has greater weight than any edge in H.

Now, if MST contains the heaviest edge e in the cycle $C = v_1 \cdots v_2 v_1$ that connects (v_1, v_2)

Since we also know no cycles are allowed in MST.

Then, this indicates that some edge in C cannot exist in MST.

If we remove e from MST, then MST would be cut into 2 parts. $v_1, \cdots v_i$, $v_j, \cdots, v_2$

Since there are two paths now, we can restructure them by connecting v_i and v_j to make a new tree.

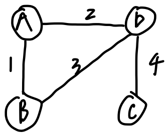
The new tree is definitely with less weight than MST, since e is the heaviest edge in C but is removed.

Therefore, this leads to the contradiction.

Therefore, the heaviest edge e in a cycle cannot be contained in a MST. Thus, MST can not be in a MHEST.

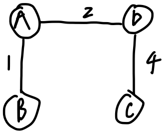
Therefore, MST is an MHEST.

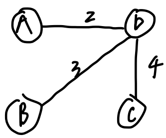
(b) Consider the following graph:



we have

MST: and an MHEST:





weight = 7 weight = 9

minimum heaviest edge = 4 minimum heaviest edge = 4

Thus, an MHEST is not always an MST.