PS4

A – Hierachy

- I started by trying to run Prim's algorithm to generate a MST then calculate the total weight of the MST. However, I forgot that this graph is directed and MST algorithm only works on undirected graphs
- Now looking back to the problem. Since each employee can be the supervisor of any amount of other employee, that means using greedy algorithm to select the lowest cost supervisor for each employee would not block any future selection, so greedy algorithm is viable. Also since we are going from the person being supervised to the supervisor, we would like to reverse the direction of each edge. This means if u is willing to be the supervisor of v, then we construct the edge v->u. Then out of all the edges[v], we select the one of the lowest cost.
- Then if a node is a root, then it would have no outgoing nodes. Hence we count how many such roots are there. If there only exists one root, then the allocation is valid, otherwise invalid.

B – Useful Decomposition

- If we don't want any edge to be used twice, this means all paths can only have at most 1 common vertex
- The question also states each two paths have at least 1 common vertex.
- Combining the two points above, we can conclude all the paths have exactly one common node.
- In other words, this means there exists only one node that can have more than 2 outgoing edges. If we call this node the root node, and the nodes with only 1 outgoing edge to be the leaf node, then all paths would be from the root node to the leaf node.
- Now we need to check whether such a decomposition exists. The answer is it is only possible when there exists one root. I.e. only one node has more than 2 edges connected to it. We simply count how many such nodes there exists, then if more than one then impossible. Otherwise, the number of paths would be the same as the number of leaf nodes we have, since there exist one node from the root to every leaf node.

C – Bertown Roads

- First reading the question, the statement "one can get from any junction to any other one by the existing roads" reminds me of strongly connected component.
- Then continue reading the question, we are asked to find if it is possible to get from any junction to any other one if all edges become undirected. In other words, we can interpret this as "determine if the removal of any edge disconnects the graph". Now this sounds like the bridge-finding algorithm in the lecture.
- If an edge is a bridge which connects two strongly connected components A and B, then changing this edge to a undirected edge would cause A unable to reach B or B unable to reach A anymore. Hence the question becomes determining if there exists bridge edge in the graph. If yes, then it would be impossible to change all edges to undirected as that would disconnect the strongly connected components.
- If there exists no bridge, then we need to find how to assign direction to edges.
    - We apply the bridge finding algorithm as shown in the lecture slide.

        - We let the “normal edges” go downwards in level, i.e. if there exists normal edge u-v where we started from u and v is visited for the first time, then we change the undirected edge to directed edge u->v
        - We let the “back edges” to up in level. i.e. if there exists backedge u-v where we start from u and v is seen before and preorder[u] >= preorder[v], then we change the undirected edge to directed edge u->v
    - A mistake I made was when checking for backedge from u to v, I compared low[u] with preorder[v], where I should be comparing preorder[u] and preorder[v], which means if preorder[u]>= preorder[v], then the edge from u to v goes from a deeper layer up to a shallower layer, so it is an back edge.

D – Colouring edges

- I first tried to come up with examples where more than 2 colours are needed to complete the colouring scheme but I couldn’t. That gave me the first realization that all colouring schemes can be done with at most 2 colours. We can sort-of provide a rough proof:
    - Assume a graph has more than 3 colouring.
    - If all cycles contain only either 2 of the 3 colouring, then we can eliminate one colour since all cycles only need 2 colours to ensure a successful colouring scheme
    - If there exists a cycle with 3 colouring, then there would be different cases
        - If all edges only exists in this cycle, i.e. they are not common edges in another cycle, then we can simply swap a colour out
        - If an edge is a common edge in another adjacent cycle, then
            - If the other edges in the adjacent cycle all have the same colour, then we can safely swap this edge out with either of the 3 colours that is not present in the adjacent cycle
            - If the other edges in the adjacent cycle have 2 colours already, then we can swap out this edge with any colour
    - Repeat the process for all edges, we can only keep 2 colours for the entire graph
- Now we know that the answer is at most 2, we need to figure out the allocation. A greedy solution would work since we only need one edge in a cycle to be different colour. We let the “last edge”, which is the edge that goes back to an active node to have colour 2, the rest of the nodes to have colour 1. This algorithm also ensure there will not be any cycle that is coloured only in colour 2, as the edge from the root node where we start the cycle detection code, to its adjacent edge has to be coloured in colour 1.
- Since we are not guaranteed that the graph is a strongly connected component, so we need to run the cycle detection algorithm at every node to ensure all edges are visited
- I was stuck on case 3 for a while, then realized that the lecture cycle detection code terminates as soon as any cycle is found, however the behaviour we want is to mark the edge that visits the active node, then keep traversing until all nodes are seen. Hence removing the early stopping fixed the issue.

E - Minimum spanning tree for each edge

- My initial thought would be first include the target edge u-v in a set, then run Kruskal to generate the minimum spanning tree. Then repeat this process for every edge. Obviously this would be too slow

- Then I tried a few examples, realized that the MST generated using the algorithm above is the same as the MST generated by running Kruskal directly, except it has an additional edge u-v and another edge on the original MST removed. Hence the logic should be:
    - 1. Run MST algorithm
    - 2. Add the target edge
    - 3. Remove some edge
- Now the task is to find out which edge to remove. I then tried to add the edge u-v to the MST generated by Kruskal when u-v is not already on the MST. I then found that it would introduce a cycle, and removing any edge on the cycle would generate a new tree. Hence since we are trying to generate the "MST", we would then like to remove the edge of the highest cost within the cycle.
- To find the highest cost edge within the cycle, we would first find the highest cost edge on u-LCA, then find the highest cost edge on v-LCA, then the maximum cost of these 2 would be the maximum cost from u to v0
    - To find the highest cost edge on the path from v to LCA, we first precompute the highest cost from the ith node going up 2^k levels for all nodes for all k less than log_2(n).
        - The highest cost in the upper the 2^j-th levels of v is the max between the highest cost in the 2^(j-1) levels of the 2^(j-1)-th parent and the 2^(j-1) levels of v
    - Then we use the same logic when balancing the depth u and v when computing LCA, but this time we keep computing the maximum cost until the depth of v and LCA are equal.
        - Start the cost with 0, then take the maximum between the cost and the highest cost from v going up 2^k layers where k is the largest integer such that depth of v - 2^k >= depth of LCA. Then we update v to be the 2^kth parent of v. Repeat this process until v and LCA have the same depth.
- It took me a while to get LCA algorithm to work, primary reason being the lec slide uses 0-indexing for nodes and this question uses 1-indexing
- Also I computed log_2(n) wrong, and also made silly mistake such as initializing array with size 1e5 when the input is told to be as large as 2e5