PS5

A – Jumping on walls

- The algorithm for this question is really straight forward, but the implementation detail was the part that I spent a while on
- Treat each area on the wall as a node. 3 moves can be performed, which are climbing one unit up, one unit down, or jump k units to the opposite wall. Hence, we add edge to all these nodes.
    - For converting each area on the walls to node index, I assigned each are on the left wall of indices 0, 2, 4, 6, 8, … etc, and on the areas on the right wall have indices 1, 3, 5, 7, 9, … etc. So that when we want to compute the index of a node from its height, we simply multiply it by 2, then add 1 if it's on the right wall.
- Since out goal is to jump out of the canyon, then it would not matter how many areas are we above the top of the canyon, we only care about if the final step is above the canyon or not. Following this logic, we should create extra terminating node representing the top of the canyon. If the ninja's height is above the top of the canyon, then we construct an edge from the current cell that the ninja is at to the terminating node.
    - Also since the ninja can exit the canyon from both the left wall or the right wall, we create two such termination nodes, one for the left wall exit one for the right wall exit.
- To find the shortest path to exit the canyon, since this graph is undirected, we simply run bfs. However, we also need to ensure that the ninja will not drown in water, since for every step the ninja takes, the height of the water raises by 1, we need to ensure we can only travel to area with height greater than or equal to the steps required to reach that area
    - To keep account for how many steps are required to reach each area, we start with 0 at the bottom left, then for each edge, the number of steps required = the number of steps required to reach the previous area +1.
- The result of BFS is then the solution to this problem

B – Jzzhu and Cities.

- Define edge with attribute "type" being either train and route, and the comparator would always prioritize route over train. This means when we run Dijkstra, when there exists multiple edges of the same weight in the priority queue, the car route will be popped out first
- Initialize counter c, insert all edges then run Dijkstra
    - When running Dijkstra:
        - If the current vertex popped from the priority queue is seen before, and it is a train route, then this edge can be removed, so c++.
        - When checking the outgoing edges that is connected to the current vertex, if the outgoing edge is seen before and it is a train route, then this edge can be removed, so c++.
- However, there is an issue with the algorithm above, which is when we find an outgoing edge that goes to an unseen vertex, we need to add a new edge to the priority queue with the accumulated distance from the root. So what should be the type of the new edge?
    - Clearly if a car route extends upon another car routes results in another car route, but what if a car route extends upon a train route? In that case, we cannot remove this edge since it has a car route as a segment, hence if a path has both car route and train route, then the new edge we create should also be a car route.

    - To start off with Dijkstra, we also need a pseudo-edge in the priority queue as the very first edge. This edge should be a Train edge, so that any edge starts from the root, if it is a Train edge, then Train + Train = Train, if it is a Car edge, then Train + Car = Car.

C – Greg and Graph

- First observation is that the question asks for "shortest paths between all pairs", which immediately got me thinking about Floyd-Warshall algorithm
- Then the process of deleting edges one by one until non left can be interpreted as adding edges one by one from scratch in reversed order.
- Hence, my initial approach is to add edges one by one, then run Floyd-Warhshall algorithm everytime I add an edge, then find the sum of shortest path between all pairs.
- However, that did not work out as the time limit was exceeded.
- Then I'm thinking about how can I increase the efficiency of my algorithm. Funning F-W algorithm multiple times seems to be the main bottleneck since it has O(V^3) time complexity, so I started thinking how to solve the question without running F-W multiple times.
- Then I realized, F-W works essentially by limiting the intermediate nodes that can be used, hence I added the condition check so that the intermediate node used has to be after the ith deleted node in the ith round in F-W, so now when calculating all pairs of shortest path, the algorithm can only use the nodes that is already added to the graph, so running F-W once would yield the correct answer.