1. Physics Experiment

a. The optimal substructure should be the student who's able to make most steps for the physics experiment. For example, the experiment 1 requires four steps, and student 1= {1,2,3}, student 2 = {3}, student 3 = {4}, the optimal substructure here is student 1.

b. Before the experiment steps are all completed, the Greedy Algorithm would continually search for the student who does most steps that the experiment needs. From the example in part (a), after student 1 completes his steps, the Greedy Algorithm would rather choose student 3 instead of student 2, because student 3 has 1 step that it needs whereas student 2 has 0 step that it needs.

d. The runtime complexity is m*n (m is numStudents, n is numSteps)

e. Claim: let S be the solution output by the greedy algorithm in (b) and O be the optimum solution. If S is different from O, then we can tweak O to get another solution O∗ that is different from O and strictly better than O.

First, assume that S is the solution vector that contains (S1…. Sn) and O is the optimal solution vector that contains (O1…. On). If S is different from O, then there must exist some index *i*, where Si != Oi. By changing ith choice to the one chosen by the greedy algorithm. So, we get O* to be something like this:

O* = (O1,O2…Si, Oi+1,Oi+2…On)

In (b), since the greedy algorithm always pick the best optimal substructure for the experiment, it would not have more switches than any optimal solutions. If another student doesn't do the step that current student does which is chosen by greedy algorithm, then the substructure found by greedy algorithm is still counted as one switch no matter what choices that optimal solution would make. So, by contradiction, the greedy algorithm in this case is correct

2. Public transit

(a) Dijkstra's shortest path algorithm. The only difference thing between current method to the previous one is that I need to consider the time cost of the train coming as the fact to the total traveling time.

(b) runtime complexity is Big O(numberVertices(or stations) ^ 2)

(c) it implements Dijkstra's shortest path algorithm.

(d) For example: the modification

Case 1. if the time cost from the starting place to adjacent station is less than the time that first train come to the station, set the time that the first train come as the time cost.

Case 2: if the time cost is larger than the time that first train come, we can calculate the time cost by time[v] = freq[u][v] - ((time[u]-first[u][v]) % freq[u][v]) + time[u] + length[u][v] (if we still wait for train at that moment)

If the train just comes when we arrive at the station:

Time [v] = time[u] + length[u][v] (still in case 2)

For rest of code, I can just simply implement the code from shortest time function.

(e) the runtime complexity for shortest time is big O(numberVertices^2), but if we store the data in the binary heap, the runtime complexity will become Big O(Vertice lg(Vertice) + E lg(Vertice) ) which is better than the previous one.