1. PROGRAM 1: Experiments Scheduling

a) Optimal substructure

The problem can be divided in to smaller sections based on the number of tasks that a student can complete without getting switched. Therefore, by optimizing those sub sections of the problem separately can bring the whole solution an optimal level.

For example,
Let the tasks <1, 2, 3, 4, 5, 6> be the larger problem.
We can create sub problems by breaking the tasks down.
Let <1,2,3> and <4,5,6> be two sub problems. By optimizing he sub problem separately, we can make the larger problem optimal.

b) Greedy algorithm

Step 1 – Starting from first, find the student who can do maximum number of tasks without getting switched.

Step 2 – Save the maximum tasks the 1st student can do to variables and switch to the next student.

Step 3 – If the next student can do more tasks without switching, save the student and the maximum tasks to variables.

Step 4 – After iterating through all the students, fill the scheduleTable upto the current maximum task.

Step 5 – Start from the current task and do Step 3.

Step 6 – If the current task is equal to number of tasks, then return the scheduleTable.

c) Code is attached.

d) Worst Case – $O(n^2)$

Optimal Case – $O(n)$

e) Proof that your greedy algorithm returns an optimal solution.

Assume that all the tasks <t1, t2, t3, t4> can be done by a single student. Then optimal solution will be, doing the tasks by one student without switching.

Let the above case be the base case. That is, if a student can do consecutive tasks without switching, it gives the optimal solution for that task.

Suppose x be optimal solution for the instance k. Let k be once of the instances of K.

Now we can say that K has the optimal solution when all the instances have optimal solutions.

2. PROGRAM 2: Public, Public Transit

a) The Dijkstra's algorithm for shortest path can be adapted to solve this problem but with some tweaks. Some of the tweaks are,
    a. Depends on the minimum cost of the journey
    b. Has to calculate the waiting time at a station

   The adjacency matrix is used to keep the minimum cost from source to the given destination.
   The equation "first + (i * frequency) – startTime" can be used to find the next train that can be caught to get the next station.

   <u>Algorithm</u>

   Step 1 – Calculate the number of vertices from the given time adjacency matrix.
   Step 2 – Initialize an array to store final shortest times.
   Step 3 – Initialize a Boolean array to process finalized times.
   Step 4 – Initialize all the times to INFINITE and processed times to false.
   Step 4 – Find the next vertex to process and calculate the waiting time.
   Step 5 – Once moved tot the smallest-cost vertex, check each of its neighboring nodes.
   Step 6 – Calculate the cost for the neighboring nodes by summing the cost of the edges leading from start vertex along with waiting time.
   Step 7 – If the cost to a vertex I less than a known cost, update the minimum cost for that vertex.

b) Complexity = $O(V^2)$
c) Dijkstra's algorithm
d) The existing code only handles the time cost from one vertex to another. But there are also another major two data that has to be processed before finding the shortest distance.
    a. Time of the next train
    b. Waiting time

   Since the trains are not always present in the station, we have to wait in the station until a train arrive. This will affect the minimum time to get to the destination. Therefore, we need to pay our attention to that factor in order to find the minimum time to travel.

e) Current complexity = $O(V^2)$

Implementation based on a minimum priority queue implemented by a heap can be used to optimize this algorithm.
This can optimize the algorithm with complexity of O (E + V log V).