a) Describe the optimal substructure of this problem?

Ans: This is the most appropriate problem which can be solved by implementing greedy algorithm perfectly. Because combining local optimums will be the global optimum on this problem

b) Describe the greedy algorithm that could find an optimal way to schedule the students

Ans: Let's assume I have finished x steps and we should continue from X + 1 step.I simply find the student who can participate in maximum steps continuously from X + 1 step and schedule steps to him. If the number of maximum steps is K, X + K steps will be finished, and I should continue from X + K + 1.Repeating this from when X = 0 to X = n(number of steps) we could find the best schedule with least amount of switching.

(d) What is the runtime complexity of your greedy algorithm? Again, you don't need to factor in the setup of the lookup table, just your scheduling algorithm.

Ans: My maximum runtime complexity is O (n * m)

(e) In your PDF, based on your answer to part b, give a full proof that your greedy algorithm returns an optimal solution.

In this problem finding local optimum solution is the finding global solution!

Ans: Let's assume that I schedule steps from X + 1 to X + K to certain student and the number K is the maximum steps that he can continue. Any other solution exactly scheduling these steps to another student will increase at least 1 switching.

2) (a) Describe an algorithm solution to this problem. Feel free to talk about how you would adapt an algorithm we covered in class.

Ans: First of all, compute the fastest arrival time from S station to I station which is connected directly. It can be computed by finding first train after startTime and add length time takes to travel form S to I.

Code: arrival = ((startTime - first[S][i]) / freq[S][i] + 1) * freq[S][i] + first[S][i] + lengths[S][i];And set the fastest arrival time to array variable fastest_arrival[] at index I

Step 1: Then I compute the fastest arrival time from station I to it’s directly connected stations.

Step 2: And I get arrival information computed already.

If newly computed time is less than old one, we update it.

If not do nothing.

I repeat these steps recursively to find fastest arrival time to all stations.

Then our result will be fastest_arrival[T] – fastest_arrival[S]

That’s all.

(b) What is the complexity of your proposed solution in (a)?

Ans: Maximum complexity is O(V * V) V is number of vertices

(c) See the file FastestRoutePublicTransit.java, the method "shortestTime". Note you can run the file and it'll output the solution from that method. Which algorithm is this implementing?

Ans: It is implementing Dijkstra's algorithm

(d) In the file FastestRoutePublicTransit.java, how would you use the existing code to help you implement your algorithm? The existing code only handles one piece of data per edge, so describe some modifications.

I used none of the existing code.

(e) What's the current complexity of "shortestTime" given V vertices and E edges? How would you make the "shortestTime" implementation faster? Describe any algorithm changes or data structure changes. What's the complexity of the optimal implementation?

Ans: The current complexity of “shortest Time” is O(V * V). We could make “shortest Time” by implementing breadth first search and using Queue data structure.

(f) Code! In the file FastestRoutePublicTransit.java, in the method "myShortestTravelTime", implement the algorithm you described in part (a) using your answers to (d). Don't need to implement the optimal data structure.

(g) Extra credit (15 points): I haven't set up the test cases for "myShortestTravelTime", which takes in 3 matrices. Set up those three matrices (first, freq, length) to make a test case for your myShortestTravelTime method. Make a call to your method from main passing in the test case you set up.

Ans:

```
public class FastestRoutePublicTransit {

 public int myShortestTravelTime(
   int S,
   int T,
   int startTime,
   int[][] lengths,
   int[][] first,
   int[][] freq
  ) {
   // Your code along with comments here. Feel free to borrow code from any
   // of the existing method. You can also make new helper methods.
   return 0;
  }

  public int findNextToProcess(int[] times, Boolean[] processed) {
   int min = Integer.MAX_VALUE;
   int minIndex = -1;

   for (int i = 0; i < times.length; i++) {
     if (processed[i] == false && times[i] <= min) {
       min = times[i];
       minIndex = i;
     }
   }
   return minIndex;
  }

  public void printShortestTimes(int times[]) {
   System.out.println("Vertex Distances (time) from Source");
   for (int i = 0; i < times.length; i++)
       System.out.println(i + ": " + times[i] + " minutes");
  }
  public void shortestTime(int graph[][], int source) {
   int numVertices = graph[0].length;

   // This is the array where we'll store all the final shortest times
   int[] times = new int[numVertices];

   // processed[i] will true if vertex i's shortest time is already finalized
   Boolean[] processed = new Boolean[numVertices];

   // Initialize all distances as INFINITE and processed[] as false
   for (int v = 0; v < numVertices; v++) {
     times[v] = Integer.MAX_VALUE;
     processed[v] = false;
   }

   // Distance of source vertex from itself is always 0
   times[source] = 0;

   // Find shortest path to all the vertices
   for (int count = 0; count < numVertices - 1 ; count++) {
     // Pick the minimum distance vertex from the set of vertices not yet processed.
     // u is always equal to source in first iteration.
     // Mark u as processed.
     int u = findNextToProcess(times, processed);
     processed[u] = true;

     // Update time value of all the adjacent vertices of the picked vertex.
     for (int v = 0; v < numVertices; v++) {
       // Update time[v] only if is not processed yet, there is an edge from u to v,
       // and total weight of path from source to v through u is smaller than current value of time[v]
       if (!processed[v] && graph[u][v]!=0 && times[u] != Integer.MAX_VALUE && times[u]+graph[u][v] <
times[v]) {
          times[v] = times[u] + graph[u][v];
       }
     }
   }

   printShortestTimes(times);
  }

  public static void main (String[] args) {
   /* length(e) */
   int lengthTimeGraph[][] = new int[][]{
     {0, 4, 0, 0, 0, 0, 0, 8, 0},
     {4, 0, 8, 0, 0, 0, 0, 11, 0},
     {0, 8, 0, 7, 0, 4, 0, 0, 2},
     {0, 0, 7, 0, 9, 14, 0, 0, 0},
     {0, 0, 0, 9, 0, 10, 0, 0, 0},
     {0, 0, 4, 14, 10, 0, 2, 0, 0},
     {0, 0, 0, 0, 0, 2, 0, 1, 6},
     {8, 11, 0, 0, 0, 0, 1, 0, 7},
     {0, 0, 2, 0, 0, 0, 6, 7, 0}
   };
   FastestRoutePublicTransit t = new FastestRoutePublicTransit();
   t.shortestTime(lengthTimeGraph, 0);

  }
}
```