Problem 1:

(a) Describe the optimal substructure of this problem

The optimal substructure of this problem is to looking for the student who can do the most consecutive experiments from the experiments we have left with.

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

1)Use while loop to keep track how many experiments we have left

```
int exprToDo = 1;//update after found what expriments are Done
int currExprStrt = 1;
int currExprEnd = 0;
while(exprToDo<=numSteps)
```

2)Use a nested for loop to check each student in the signUp table, and keep track of the consecutive experiments each student is able to do from the current experiment we're checking

```
for(int stud = 1; stud<=numStudents; stud++)
        int currCount = 0;//update after start to check each student
        for(int expr = exprToDo ; expr<= numSteps; expr++)
```

3)select the student who's able to do the most consecutive experiments from current experiment we're checking

```
if(currCount > maxCount){
   maxCount = currCount;
    goodStud = stud;
```

4)output the student we've selected and the experiments numbers that we assigned to them

```
for(int i = currExprStrt; i <= currExprEnd; i++) {
        scheduleTable[goodStud][i] = signUpTable[goodStud][i];
```

(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.

The runtime complexity of my greedy algorithm is n*m*n where n is numOfSteps and m is the numOfStudents, therefore O(m*n^2)

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

optimal solution.

In order to prove that my greedy algorithm is the optimal solution, we first need to assume there is one optimal solution and this solution is better than mine:

1)assume the optimal solution has less switch than my greedy solution:

opt solu:S1,S2,S3, ... ,Sn

my solu:S'1,S'2,S'3, ... , S'n,...,S'm where n<m

(where S is the num of switches)

Suppose two solution are same up until Sn-1 = S'n-1, where S1 = S'1, S2 = S'2, ... Sn = S'n

2)prove that opt soul is no better than my greedy solution by showing the contradiction to the assumption:

By design, since each step in the optimal solution and my greedy solution are all the same before step (S'n) +1 and in each of these steps we're both looking for the students whose fitting the solution we're pursuit by any mean, then if at step Sn optimal solution has solved the problem, my greedy solution should also have the same solution at the time, therefore we can substitute S'1, S'2,…S'n in the rest of the way. Which contradicts to n<m.

Problem 2:

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

Since we've covered a greedy algorithm called Dijkstra's shortest path algorithm in class, so I decided to adapt this algorithm in my solution:

1) Create a set shortest path tree set that keeps track of stations included in shortest path tree. Initially, this set is empty.

```
// 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++) {
       the length between each station becomes infinity
       processed[v] = false;
}
```

2)Call a helper function waitingTime which calculate the waiting time and updates the length between each vertices (add the waiting time into the length)

```
Initialize waitingTime = 0;
initialize  arrivalTime = times[u] + startTime;

if(first[u][v] >= arrivalTime) waitingTime = first[u][v] - arrivalTime;

else{
        if((arrivalTime - first[u][v])% freq[u][v] == 0){
                waitingTime = 0;
        }
        else{
        waitingTime = freq[u][v] - (arrivalTime - first[u][v])% freq[u][v];
        }
}
return waitingTime;
```

3) Assign a length value to all stations in the input graph. Initialize all length values as INFINITE between each station. Assign distance value as 0 for the source vertex so that it is picked first.

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

   // Distance of source vertex from itself is always 0
           shortestTravelTimes[S] = 0;
```

4) While shortest path tree set doesn't include all vertices
….a) Pick a vertex u which is not there in shortest path tree set and has minimum length value.

```
// 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(shortestTravelTimes, processed);
```

```
      processed[u] = true;
```

….b) Include station u to shortest path tree set.
….c) Update length value of all adjacent vertices(A.K.A stations) of u. To update the length values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and length u-v, is less than the length value of v, then update the length value of v.

```
 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] && lengths[u][v]!=0 && shortestTravelTimes[u] != Integer.MAX_VALUE) {

        int waitingT = waitingTime(startTime,shortestTravelTimes,u,first,freq,v);

        if( shortestTravelTimes[u]+ lengths[u][v] + waitingT< shortestTravelTimes[v]) {
                shortestTravelTimes[v] = shortestTravelTimes[u] + lengths[u][v] + waitingT;
        }
        }
    }
            if(processed[T] == true)   break;
```

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

The complexity of my solution is O(n^2).

(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?

The method “shortestTime” is implementing same algorithm I used in my solution, called “Dijkstra’s shortest path 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.

The existing code is basically the way to implement Dijkstra’s shortest path algorithm **,** the only thing I need to modify is the variable called “length”, since in this problem we also need to involve “waiting time” as one of most important considerations. Therefore, my modification to the existing code is just add another helper function that calculate the waiting time and add that into the length, so that I can use the existing code but use the updated length instead.

(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?

The current complexity of “shortestTime” is O(V^2), I cannot make this "shortestTime" implementation anytime faster because this is optimal solution to this problem as far as I understand. Because no matter what, we must go through the signUpTable matrix in order to find the solution. However, if I can modify the input such as using the adjacency list as the representation of the graph, we then can change the data structure to read in the list, and use binary heap as algorithm change. Therefore, this could be the optimal implementation. The complexity of the optimal implementation can be reduced to O(E log V).

Work Cited

1. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
2. Worked with Timothy Lei, Wei Ning, Lin Hui