Problem 1

Find: An optimal way to schedule students to steps such that there is the least amount of switching as possible.

(a) Describe the optimal substructure of this problem

Answer: Optimal solution of problem is based on optimal solution of sub problems. In each switch we search for the most consecutive number of steps from the signUpTable add this student's step to the optimal solution. Then we subtract the total number of steps from current steps a student provided. Updated the next starting step.

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

Answer:

1) **Use while loop to keep track of how many steps left in the experiment.**
2) **Use nested for loops to keep track of consecutive steps in the signUpTable each student is able to do from current experiment we are checking.**
3) **We select the student gives max consecutive steps from current experiment, which a student provided.**
4) **Output the students we've selected and the steps they provided on the experiment.**
5) **Update variable steps Repeat step 2 to 4 while(steps <= numSteps)**

Below is pseudocode of the greedy algorithm

```
step = 1, begin = 1, end = 0;

while step is still less than or equal to numSteps

     student = 0, maxConsecutive1 = 0;
     for r from 1 to numStudents
          consecutive1 = 0;

         for c from begin to numSteps
              if (signUpTable[r][c] == 1)
                   consecutive1++;
```

```
                    else
                         break;

               if (consecutive1 > maxConsecutive1)
                    maxConsecutive1 = consecutive1;
                    student = r;

          step += maxConsecutive1;
          end = end + maxConsecutive1;

          for i = begin to end
               scheduleTable[student][i] = signUpTable[student][i];
          begin = end + 1;
     return scheduleTable;
```

(c) Code your greedy algorithm in the file "PhysicsExperiment.java" under the "scheduleExperiments" method where it says "Your code here". Read through the documentation for that method. Note that I've already set up the lookup table automatically for every test case. Do not touch the other methods except possibly the main method to build your own test cases (but delete/comment out your own test cases in the submission). Run the code without your implementation first and you should see this:

Answer: code in github

(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.

Answer: $O(m*n^2)$ where m is number of students, n is number of steps

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

Answer: prove by contradiction. Assume the optimal solution required few switches than my greedy algorithm solution.

My algorithm solution: $Switch_1$, $Switch_2$, … $Switch_k$… $Switch_L$

Optimal solution: $Switch_1'$, $Switch_2'$, … $Switch_k'$

Where K < L

Suppose at i is the first place where my algorithm and optimal solution is differ in which students gives switch, $Switch_1$, $Switch_2$, ... $Switch_{i-1}$ = $Switch_1'$, $Switch_2'$... $switch_{i-1}$ '. By design, max number of steps in our algorithm $switch_i$ provide by student i so we can replace $switch_i'$ with our $switch_i$. Now let's move onto switch i+1. We can substitute in the rest all the way to switch k, our algorithm ends when we finish all the steps on the experiment. Which contradict K < L. Thus my algorithm is at least as good as the optimal solution.

Problem 2. Public, Public Transit

Goal: Find the shortest time it would take to get from S to T.

Note: If at Y trains arrive at 5:59a, 6:07a, 6:15a, 6:23a, etc. If I make it to Y at 6:21a I'll have to wait 2 minutes but if I make it at 6:23a I can get on right away.

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

Answer: The algorithm I used in this case is Dijkstra's algorithm to find shortest time between stations in a map. In this algorithm i add waiting time on each station from waiting the train with travel time to next station. Because waiting time is related with when is first train pass by, how often train pass by and arrival time of the customer on each station.

Step 1: create two new array variable processed and times.

```
for v = 0   to  numVertices {
   times[v] = Integer.MAX_VALUE;
   processed[v] = false;
 }
```

step 2: Assign length value as 0 for the starting station so that it is picked first.

```
times[S] = 0;
```

step3: While proceesed doesn't include all station

a) Pick a station u which is not there in *processed* and has minimum length value.

```
for  count = 0   to  numVertices - 1
     u = findNextToProcess(times, processed);
     processed[u] = true;
```

b) Update length value of all adjacent stations of u. To update the length values, iterate through all adjacent stations. For every adjacent station v, if sum of length value of u (from starting) + waiting time and weight of edge u-v, is less than the length value of v, then update the length value of v.

```
if (!processed[v] && lengths[u][v]!=0 && times[u] != Integer.MAX_VALUE )
        waitingTime = 0;
        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];


                        if( times[u]+ lengths[u][v] + waitingTime< times[v])
                                times[v] = times[u] + lengths[u][v] + waitingTime;
```

step 4: break out of loops if (processed[T] == true) because we have reach our final destination, finally we just need to return times[T] to output result.

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

Answer: O(v^2), I use only 2 nested for loops. I use module to calculate waiting time, didn't add complexity to the algorithm.

(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?

Answer: The "shortestTime" method is implemented Dijkstra's shortest path algorithm because all the distance between each connected station are given.

(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.

Answer: I add the waiting time on each station for those comparison of shortest path calculation. Here is how I modify the existing code.

if (!processed[v] && lengths[u][v]!=0 && times[u] != Integer.MAX_VALUE) is true

I will create a waiting time variable

```
arrivalTime = times[u] + startTime
if first train pass time > 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]
finally, update new length as follow
if( times[u]+ lengths[u][v] + waitingTime< times[v])
times[v] = times[u] + lengths[u][v] + waitingTime;
```

(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?

Answer: The current complexity of "shortestTime" is O(v^2). We can make the "shortestTime" faster if we change the data structure of the input graph is represented using adjacency list, which can be reduced to O(ElogV) with help of binary heap.

(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.

Answer: github code.

(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.

Answer: github code.

Reference:

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

worked with Feng Yu Zhang, Wei Ning