Problem 1

(a) Describe the optimal substructure of this problem

Optimal substructure:

n = the number of steps in the experiment

We can split the n steps to a set of sub steps S = {k_1, k_2, k_3....k_j}

n = $k_1 + k_2 + k_3 + ... + k_j$

Each element in the set S has the **longest continuous** signed up value, that means no switch in any k.

So we can make sure we have the least switches.

For any k_1, k_2, k_3....k_j , it only associate with one student.

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

At the beginning of scheduling, we have 3 inputs: numStudents, numSteps, signUpTable (lookup table).

Our goal is to find the longest continuous 1's.

For example 1, signUpTable is :

{

{0, 0, 0, 0, 0, 0, 0},

{0, **1, 1, 1,** 0, 1, 0},

{0, 0, 1, 1, 1, 0, 0},

{0, 1, 0, 0, **1, 1, 1**}

}

Step 1: create another 2-d array to store our schedule results called scheduleTable.

Step 2: from step 1, student 1, use iteration to find the longest continuous 1's in look up table. Get the steps number maxSubStep.

```
for i from 1to numStudents
for j from currentStep to numSteps
if signUpTable [i] [j] equals 1
tempStep++
```

else break

if maxSubStep < tempStep

maxSubStep = tempStep;

Step 3: find the current step, currentStep = currentStep + maxSubStep, currentStep from 1.

From current step to next maxSubStep,

repeat Step2.

Step 4: After n Steps, we get our results.

(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 is $O(m*n^2)$, n is numSteps , m is numStudent, in my function I have 1 while loop nest with 2 nested for loop.

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

Assume we have an optimal solution has the least switch times.

Optimal solution: $S_1, S_2, S_3, \ldots S_m$

Our solution: $K_1, K_2, K_3, \ldots K_n$

And $m < n$

Suppose before i^{th} step :

$S_1, S_2, S_3, \ldots S_{i-1} = K_1, K_2, K_3, \ldots K_{i-1}$

By my algorithm design, max number of the current continuous steps in K_i , so K_i has the least switch number, then we can replace K_i by S_i . Now we can implement the same prove to i +1 step, until the end of our solution K_n .So we get at least $m = n$, conflict with $m < n$.

In a word, our solution is at least as good as one of the optimal solutions.

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.

I checked *shortestTime* function, this function Dijkstra algorithm, and found I still can use Dijkstra algorithm for myShortestTravelTime function with just adding waiting time.

Because for each step, waiting time is related to frequency table freq [u] [v] , arriveTime , firstTrainTime, these are all given from a specific station u to v.

So that means in one iteration, we both can calculate train's running time and customer's waiting time.

Step1: given parameters: S, T, startTime, lengths [] [], first [] [], freq [] [],

set new parameters: times [] , processed [],

initial all distance in times as INFINITE, processed value as false.

Step2: find shortest path from S to T:

find next vertex u to process, u = S(source) in first iteration.

for 0 to numVertices - 1

u = findNextToProcess(times, processed).

Step3: find shortest time value (times[]) of all the adjacent vertices of the picked vertex using an iteration

for v = 0; v < numVertices; v++

First, check processed, lengths and times table:

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

Then we use mod to calculate waiting time, but first we need compare first train start time at this station and arrive time.

arriveTime = times[u] + startTime

if (firstTrainTime >= arriveTime)

waitingTime = firstTrainTime - arriveTime

else if ((arriveTime - firstTrainTime) % freq [u] [v] == 0)

waitingTime = 0;

else

waitingTime = freq[u] [v] - (arriveTime - firstTrainTime) % freq[u] [v]

Then check times value is minimum or not:

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

times[v] = times[u] + lengths[u] [v] + waitingTime;

Step4: break these 2 for loops when T (termination station) has been included into processed table.

if (processed[T] == true)

break;

Finally, return times[T]. Because we only need the shortest time from S to T .

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

The complexity is O(V^2) , because we only have 2 nested for loops , and use mod to calculate waiting time, so we do not add complexity on original 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?

This is Dijkstra algorithm, because the source vertex (station) is given, all the shortest is from this given source vertex.

(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 add **waiting time** to those comparisons of the shortest path's calculation.

arriveTime is the time we arrive one station,

arriveTime = times[u] + startTime

if (firstTrainTime >= arriveTime)

waitingTime = firstTrainTime - arriveTime

else if ((arriveTime - firstTrainTime) % freq [u] [v] == 0)

waitingTime = 0;

else **waitingTime** = freq[u] [v] - (arriveTime - firstTrainTime) % freq[u] [v]

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?

The current complexity of “shortestTime” is O(V^2).

Data structure changes: change the adjacency 2-d array which present the graph to binary heap.

Complexity can be reduced to O(ElogV).

References

1. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
2. Professor Yao's lecture notes.

3. Worked with Timothy Lei, Robby Zhang.