Name: Nobin Pak

Course: cs323

Assignment: Assignment 2

Question_1:

A. Describe the optimal substructure of this problem?

The optimal substructure of this problem is as follows; • I Calculate consecutive steps for each student separately. • Like if there are 3 students, then I calculate 1st student consecutive steps, then calculate steps of 2nd student right after the ending step of 1st student and so on. • After gathering chunks of every student steps, I calculate optimal solution.

B. Describe the greedy algorithm that could find an optimal way to schedule the students

FUNCTION findlongest(list,k):

    j : k

    while j in list:

    j : j+ 1

    return j-1

    END

START:

n : length_of_signUpTable

maxn : 0

Loop from i to n:

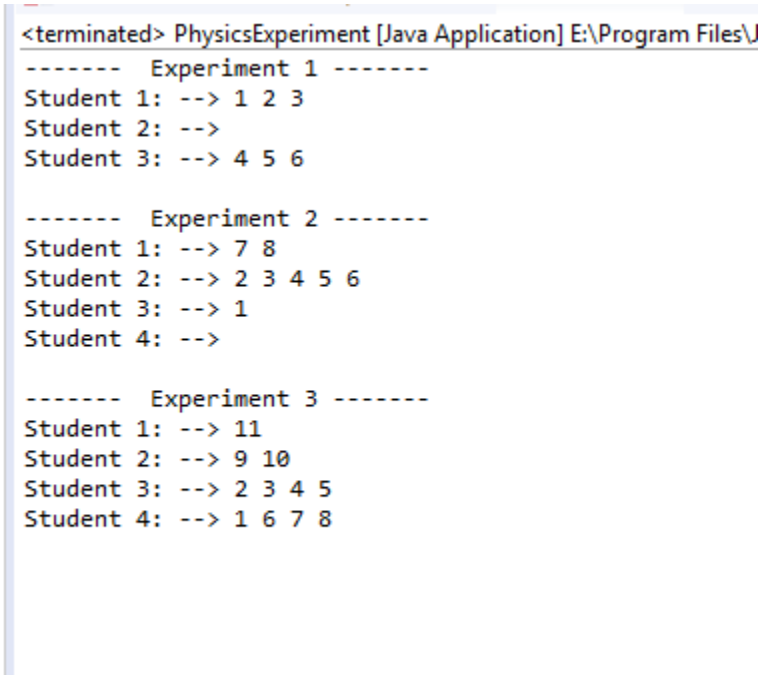
    maxn :- calculate maximum value from signUpTable

```
i  : 1
schedulingTable : [ ]
 k  :-  0
VOID : -1



WHILE i is less than maxn:
   kmax : 0
   j0   :  0
LOOP from j to n:
IF i is in signUpTable[j]:
k : findlongest(signUpTable [j],i)     //call the function findlongest (we declared above)
     IF k is greater than kmax:
        kmax : k
        j0  :-  j
       ENDIF          w
      ENDIF
 FORLOOPEND
IF j0 not equal to VOID:
            schedulingTable.append([j0+1,[i,kmax]]    //where j0+1 is student number and i,kmax are h
                                                                  //is consecutive steps.
             i :- kmax+1
    ENDIF
 WHILEEND
 REMOVE_DUPLICATE_STUDENTS        //see code section.
RETURN schedulingTable
```

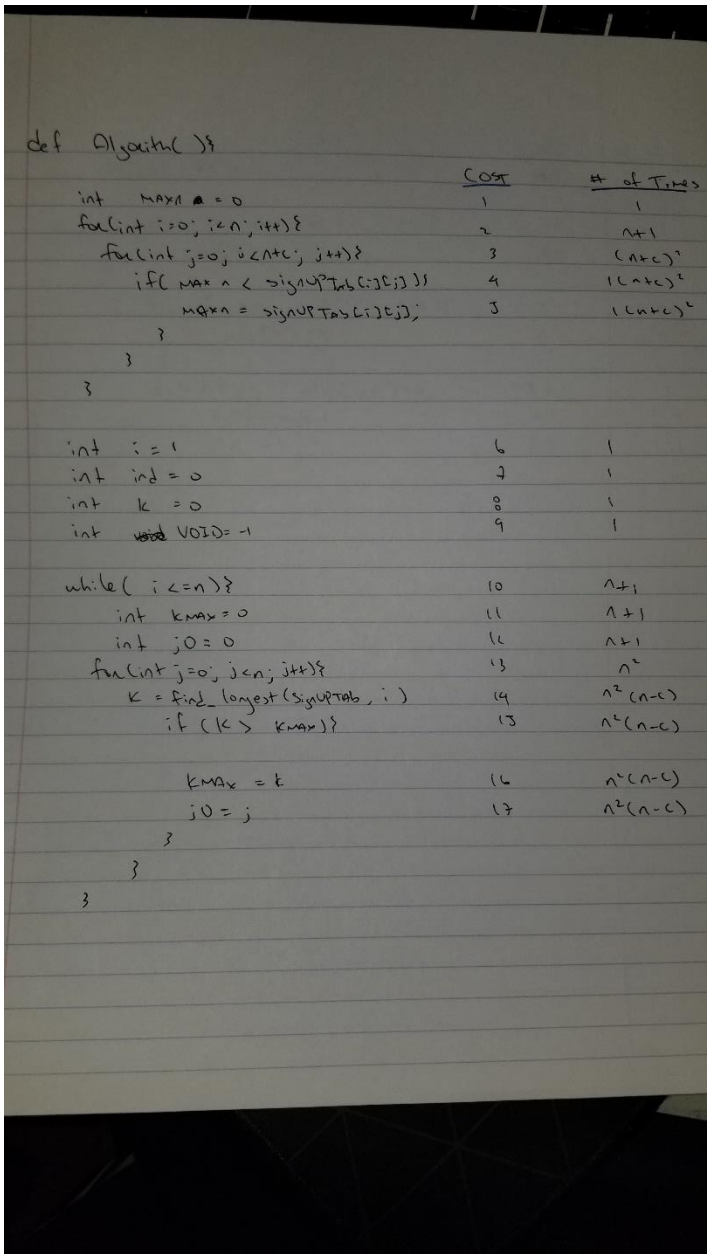
As you can see in the code, I have shown the most optimal way to schedule students, we are using students that can have the most steps in a row. This will later decrease the switching amount that is needed to finish the experiment.

When we find the student that can do the most steps and he can no longer do anymore in a row we will store him as a best option this student will be our best choice and then we check if the other students can do better than the current student. If the new student can do better, then the current we will use that student if not we will stick with the best student and continue. If the new student is better, we will use this one and continue to the next step

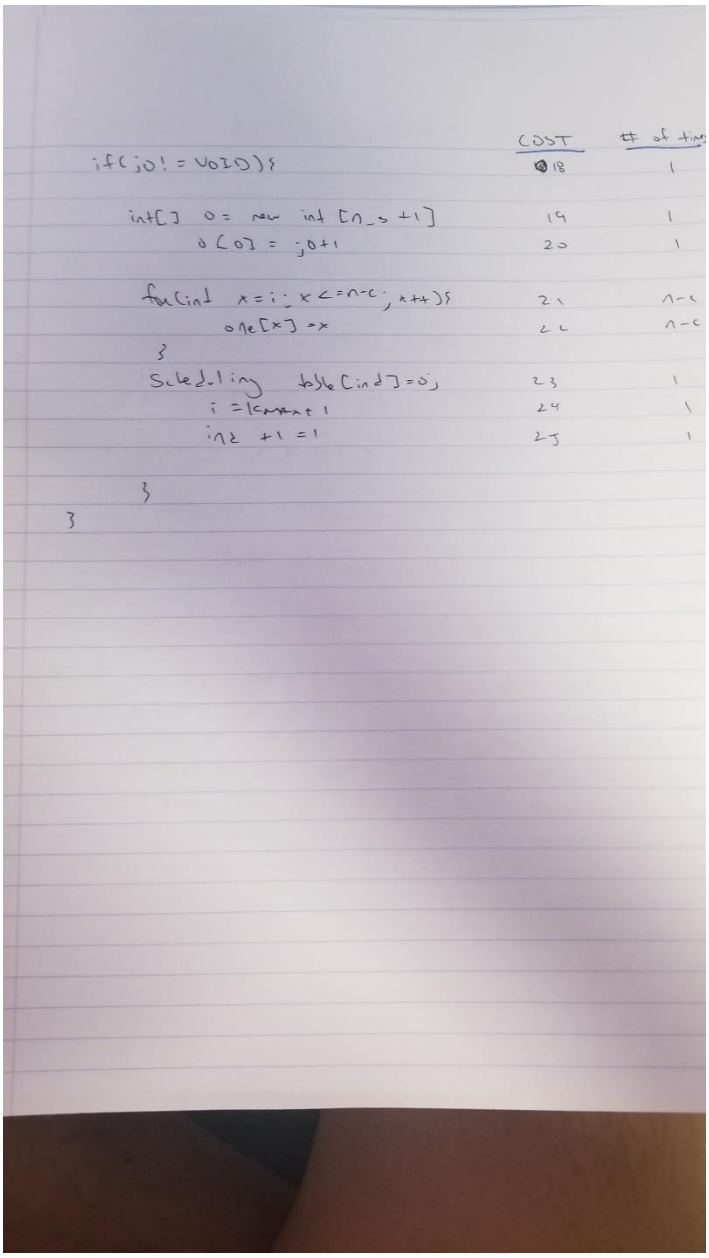
C. Code your greedy algorithm in the file "PhysicsExperiment.java" under the "scheduleExperiments" method where it says "Your code here". Read through the documentation for that method. Note that I've already set up the lookup table automatically for every test case. Do not touch the other methods except 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.



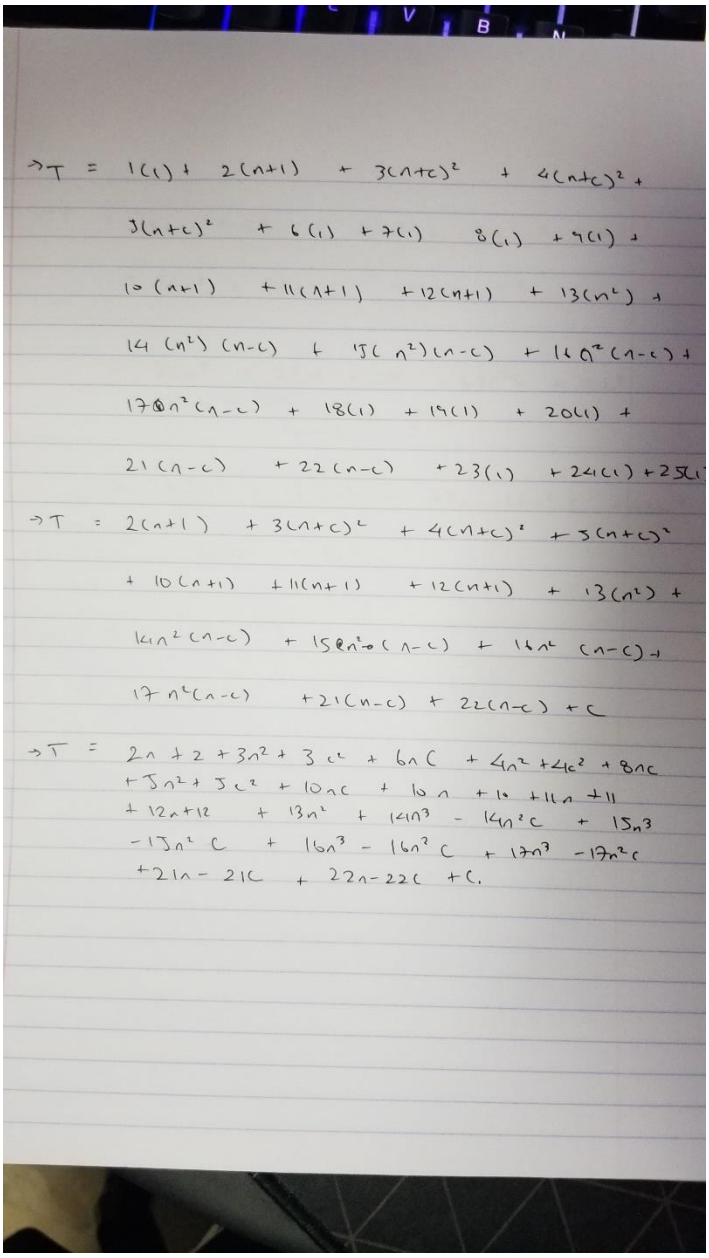
D. What is the runtime complexity of your greedy algorithm? Again, you don't need to factor in the setup of the lookup table, just your scheduling algorithm.



```
def Algorithm( ){                                   COST    # of Times
    int MAXn = 0                                    1       1
    for(int i=0; i<n; i++){                         2       n+1
      for(int j=0; j<n+c; j++){                     3       (n+c)^1
        if( MAXn < signUPTab[i][j] ){               4       1(n+c)^2
            MAXn = signUPTab[i][j];                 5       1(n+c)^2
        }
      }
    }

    int  i = 1                                      6       1
    int  ind = 0                                    7       1
    int  k = 0                                      8       1
    int  void VOID= -1                              9       1

    while( i<=n){                                   10      n+1
      int kMAX = 0                                  11      n+1
      int jO = 0                                    12      n+1
      for(int j=0; j<n; j++){                       13      n^2
        k = find_longest(signUPTab, i)              14      n^2(n-c)
        if (k > kMAX){                              15      n^2(n-c)

          kMAX = k                                  16      n^2(n-c)
          jO = j                                    17      n^2(n-c)
        }
      }
    }
```



	COST	# of times
if(j0 != VOID){	18	1
int[] 0 = new int [n_ + 1]	19	1
0[0] = j0+1	20	1
for(int x = i; x <= n-c; x++){	21	n-c
one[x] = x	22	n-c
}		
Scheduling table[ind] = 0;	23	1
i = [illegible] + 1	24	1
in2 +1 = 1	25	1
}		
}		



$$\rightarrow T = 1(1) + 2(n+1) + 3(n+c)^2 + 4(n+c)^2 +$$

$$3(n+c)^2 + 6(1) + 7(1) \quad 8(1) + 9(1) +$$

$$10(n+1) + 11(n+1) + 12(n+1) + 13(n^2) +$$

$$14(n^2)(n-c) + 15(n^2)(n-c) + 16n^2(n-c) +$$

$$17n^2(n-c) + 18(1) + 19(1) + 20(1) +$$

$$21(n-c) + 22(n-c) + 23(1) + 24(1) + 25(1$$

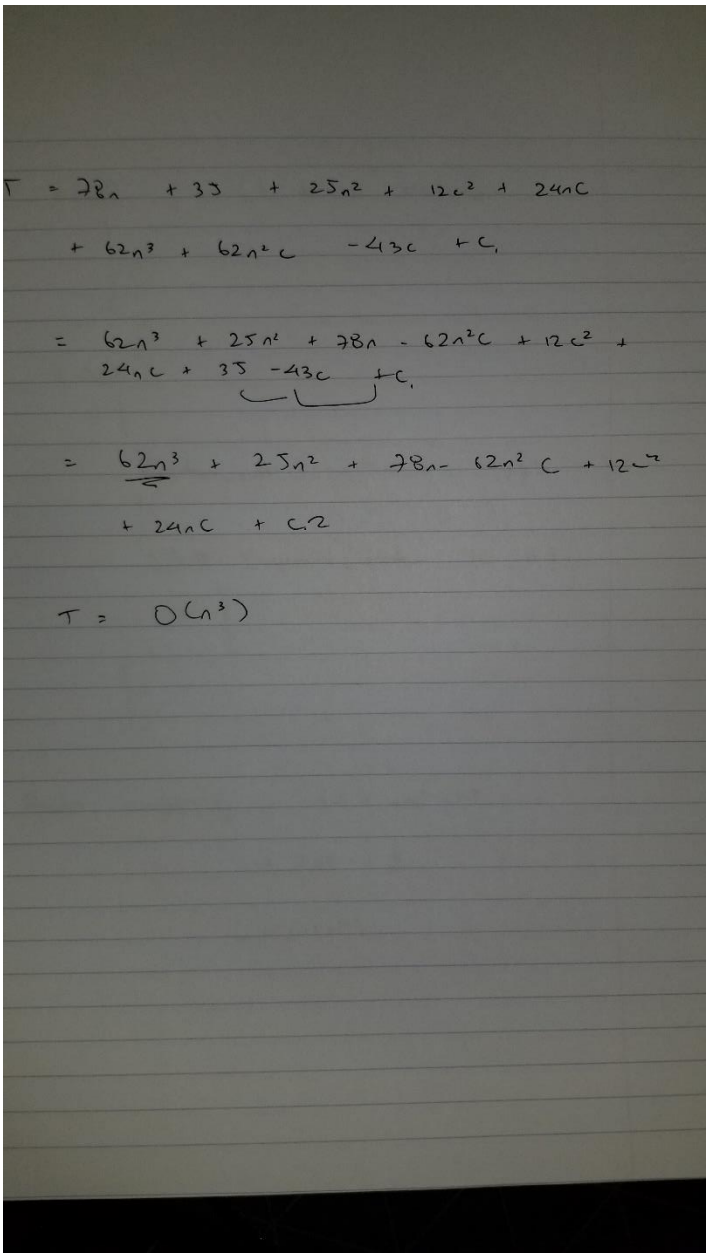
$$\rightarrow T = 2(n+1) + 3(n+c)^2 + 4(n+c)^2 + 5(n+c)^2$$

$$+ 10(n+1) + 11(n+1) + 12(n+1) + 13(n^2) +$$

$$14n^2(n-c) + 15n^2(n-c) + 16n^2(n-c) +$$

$$17n^2(n-c) + 21(n-c) + 22(n-c) + C$$

$$\rightarrow T = 2n + 2 + 3n^2 + 3c^2 + 6nc + 4n^2 + 4c^2 + 8nc$$
$$+ 5n^2 + 5c^2 + 10nc + 10n + 10 + 11n + 11$$
$$+ 12n + 12 + 13n^2 + 14n^3 - 14n^2c + 15n^3$$
$$- 15n^2c + 16n^3 - 16n^2c + 17n^3 - 17n^2c$$
$$+ 21n - 21c + 22n - 22c + C.$$


T = 78n + 35 + 25n² + 12c² + 24nc
+ 62n³ + 62n²c - 43c + c₁
= 62n³ + 25n² + 78n - 62n²c + 12c² +
24nc + 35 - 43c + c₁
= 62n³ + 25n² + 78n - 62n²c + 12c²
+ 24nc + c.2
T = O(n³)

T = O(n^3)

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

PROOF: If a student is doing step a and is qualified for all the steps a,a+1,…,b−1,b, there is no reason to switch away from the student at any time before step b. So we can assume that once a student starts doing a step, they keep working until they reach a step they're not qualified to do.

Let [0],[1], . . ., [n] denote the possible states we can be in: state [k] denotes that the first k steps have been completed. A student qualified to do steps a through b but not b+1 can get us from state [a−1] to state [b], or from any of the states [a],[a+1],…,[b−1] to state [b] without switches, after which we need to switch to another student.

Consider Experiment 1 in which cases are as follows:

• Student 1: <1, 2, 3, 5> • Student 2: <2, 3, 4> • Student 3: <1, 4, 5, 6>

Firstly, the consecutive steps are 1,2,3 (done by Student 1) from here upward algorithm looks for consecutive steps after 3, and that are 4,5,6 (done by student 3). Hence it gives optimal solution with a smaller number of switches. Solution is;

• Student 1 → 1, 2, 3 • Student 2 → No steps • Student 3 → 4, 5, 6

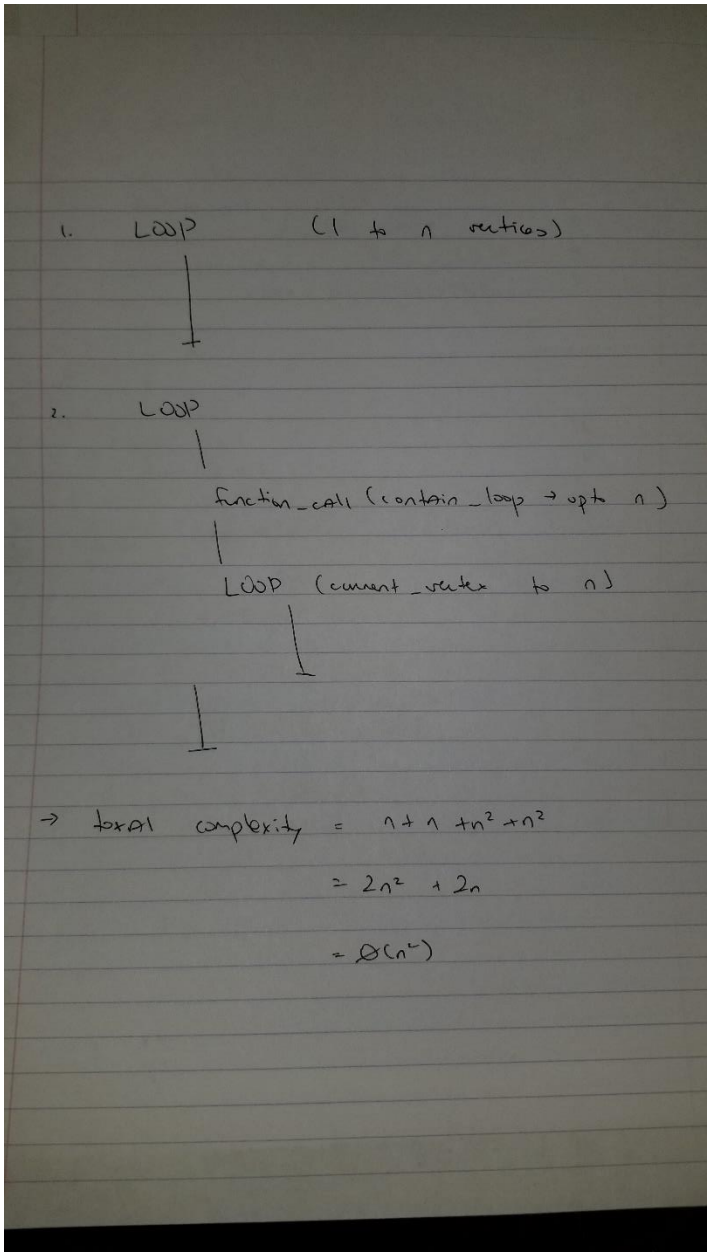
Similarly, if we do this job by not getting consecutive steps done by a student than the complexity and possible combinations of steps for each student got much increases.

Question_2:

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

Algorithm: I. Create a set time (contain minimum time for each verticex) that keeps track of vertices included in minimum time, i.e., whose minimum time from source is calculated and finalized. Initially, this set is empty. II. Assign a time value to all vertices in the input graph. Initialize all time values as INFINITE. Assign time value as 0 for the source vertex so that it is picked first. III. While time set doesn't include all vertices. a. Pick a vertex u which is not there in time set and has minimum time value. b. Include u to time set. c. Update time value of all adjacent vertices of u. To update the time values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of time value of u (from source) and weight of edge u-v, is less than the time value of v, then update the time value of v.

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



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?

➔ 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 need some modification in the inner loop of ShortestTime method, and the changing is that, for each time we add frequency of current edge and first of edge (first(e) +f*freq(e))

where f is simply a counter.

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(n^2). I am not making it faster because of restriction of optimal substructures but we can make it complexity less to O(ElogV) and this is possible by using 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.

Done. See .java file.

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.

```
//YOUR TEST CASSES ARE HERE

//Test Cases Experiments.
FastestRoutePublicTransit experiment_no_1 = new FastestRoutePublicTransit();
FastestRoutePublicTransit experiment_no_2 = new FastestRoutePublicTransit();
FastestRoutePublicTransit experiment_no_3 = new FastestRoutePublicTransit();

System.out.println("\n#######################################");
System.out.println("### SELF MADE TEST CASESES RESULT ####");
System.out.println("#######################################\n");


//Create Own test cases.
System.out.println("#### Test Case # 1 ###");
//test case 1

/*length(e)*/
lengths = new int [][]{
        {0, 0,9, 5, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 3, 0, 7, 0},
        {9, 0, 0, 1, 5, 0, 0, 0, 4},
        {5, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 5, 5, 0, 4, 0, 0, 0},
        {0, 3, 0, 0, 0, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 2, 0},
        {0, 7, 0, 0, 0, 0, 2, 0, 5},
        {1, 0, 4, 0, 0, 0, 0, 5, 0}
};
/*first(e)*/
first = new int [][]{
        {0, 0, 4, 5, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 12, 0, 17, 0},
        {9, 0, 0, 11, 15, 0, 0, 0, 14},
        {6, 0, 10, 0, 0, 0, 0, 0, 0},
        {0, 0, 14, 15, 0, 24, 0, 0, 0},
        {0, 6, 0, 0, 0, 0, 12, 0, 0},
        {0, 0, 0, 0, 0, 9, 0, 2, 0},
        {0, 6, 0, 0, 0, 0, 8, 0, 15},
        {1, 0, 4, 0, 0, 0, 0, 8, 0}
};
/*freq(e)*/
freq = new int [][]{
        {0, 0, 19,18, 0, 0, 0, 0, 17},
        {0, 0, 0, 0, 0, 3, 0, 7, 0},
        {9, 0, 0, 8, 7, 0, 0, 0, 6},
        {5, 0, 4, 0, 0, 0, 0, 0, 0},
        {0, 0, 5, 4, 0, 3, 0, 0, 0},
        {0, 3, 0, 0, 0, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 0},
        {0, 17, 0, 0, 0, 0, 12, 0, 15},
        {11, 0, 14, 0, 0, 0, 0, 12, 0}};
experiment_no_1.myShortestTravelTime(2,7,10,lengths,first,freq);
```

```
System.out.println("\n#### Test Case # 2 ###");
//test case 2

/*length(e)*/
lengths = new int[][]{
        {0, 9, 0, 1, 10, 0, 12},
        {9, 0, 0, 9, 1, 5, 1},
        {0, 0, 0, 6, 9, 7, 5},
        {1, 9, 6, 0, 1, 1, 0},
        {10, 1, 9, 1, 0, 0, 0},
        {0, 5, 7, 1, 0, 0, 11},
        {12, 1, 5, 0, 0, 11, 0}
};

/*first(e)*/
first = new int[][]{
        {0, 6, 0, 2, 5, 0, 10},
        {6, 0, 0, 10, 10, 5, 10},
        {0, 0, 0, 8, 14, 15, 6},
        {2, 3, 4, 0, 4, 4, 0},
        {6, 7, 4, 10, 0, 0, 0},
        {0, 15, 4, 1, 0, 0, 6},
        {9, 4, 3, 0, 0, 5, 0}
};

/*freq(e)*/
freq = new int[][]{
        {0, 15, 0, 20, 15, 0, 10},
        {6, 0, 0, 5, 7, 5, 9},
        {0, 0, 0, 6, 7, 9, 12},
        {10, 9, 8, 0, 7, 6, 0},
        {5, 4, 3, 10, 0, 0, 0},
        {0, 11, 20, 11, 0, 0, 16},
        {19, 14, 13, 0, 0, 2, 0}
};
experiment_no_2.myShortestTravelTime(3,5,15,lengths,first,freq);
```

```
    };
    experiment_no_2.myShortestTravelTime(3,5,15,lengths,first,freq);


    System.out.println("\n#### Test Case # 3 ###");
    //test case 3
    lengths = new int[][]{
            {0, 10, 20, 15, 0, 0, 9},
            {10, 0, 6, 8, 0, 0, 1},
            {20, 6, 0, 4, 4, 0, 0},
            {15, 8, 4, 0, 9, 1, 2},
            {0, 0, 4, 9, 0, 0, 0},
            {0, 0, 0, 1, 0, 0, 9},
            {9, 1, 0, 2, 0, 9, 0}
    };
    first = new int[][]{
            {0, 5, 10, 5, 0, 0, 9},
            {10, 0, 9, 8, 0, 0, 7},
            {5, 4, 0, 14, 3, 0, 0},
            {9, 8, 7, 0, 6, 5, 4},
            {0, 0, 14, 19, 0, 0, 0},
            {0, 0, 0, 11, 0, 0, 4},
            {5, 11, 0, 12, 0, 3, 0}
    };
    freq = new int[][]{
            {0, 10, 15, 20, 0, 0, 25},
            {1, 0, 2, 3, 0, 0, 4},
            {4, 3, 0, 2, 1, 0, 0},
            {10, 8, 6, 0, 4, 2, 11},
            {0, 0, 14, 15, 0, 0, 0},
            {0, 0, 0, 6, 0, 0, 11},
            {11, 2, 0, 10, 0, 5, 0}
    };
    experiment_no_3.myShortestTravelTime(2,6,10,lengths,first,freq);
}
```