Project 6: (Java) Scheduling. You are to implement the dependency graph, and scheduling problem.

As taught in the class, there are four options in the scheduling:
(1) using limited processors where each job takes 1 unit of processing time;
(2) using limited processors where jobs take variable unit of processing time;
(3) using unlimited processors where each job takes 1 unit of processing time;
(4) using unlimited processors where jobs take variable unit of processing time.

In this project we combine the above four options into one. That is, your program will be able to handle all these four options. The number of processors are given within command line arguments (args[]).

You will be given three test data sets where each set includes a file contains the dependency graph and a file contains processing time for each node. Nodes in the graphs represent jobs. One of three graphs contains cycles.

Set1: graph1 and jobTime1
Set2: graph2 and jobTime2
Set3: graph3 and jobTime3

Run your program four times as follows:

- Run your program once on Set1 using 2 processors.
- Run your program once on Set2 using 3 processors.
- Run your program once on Set3 using 3 processors.
- Run your program once on Set3 with (numNodes + 2) processors // meaning, unlimited.

*** Include in your hard copies:

- cover page
- draw the dependency graph for graph1
- draw the dependency graph for graph2
- draw the dependency graph for graph3
- source code
- outFile1 // result of Set1 using 2 processors
- outFile2 // result of Set1 using 2 processors
- outFile1 // result of Set2 using 3 processors
- outFile2 // result of Set2 using 3 processors
- outFile1 // result of set3 using 3 processors
- outFile2 // result of set3 using 3 processors
- outFile1 // result of Set3 using unlimited processors
- outFile2 // result of Set3 using unlimited processors

*** Note: In this specs, nodes and jobs are the same things and used interchangeable.
****

Language: Java
Project points:12 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):

12/12 on time: 4/9/2021 Friday before midnight.
+1 early submission: 4/5/2021 Monday before midnight.
-1 1 day late: 4/10/2021 Saturday before midnight.
-2 2 days late: 4/11/2021 Sunday before midnight.
-12/12 : after 4/11/2021 Sunday after midnight.
-6/12: does not pass compilation.
0/12: program produces no output.
0/12: does not submit hard copy.

*** Follow "Project Submission Requirement" to submit your project.

****

I. Inputs: There are three (3) inputs to the program.
****

1) inFile1 (use args[0]): a text file representing the dependency graph, G=<N, E>.

The first number in inFile1 is the number of nodes in the graph;
then follows by a list of directed edges (dependency) <n_i, n_j>, where n_i, n_j node IDs and nodeID is from 1 to numNodes, 0 is not used.

For example:

```
6       // Graph has 6 nodes
1  2    // 2 is a dependent of 1
1  4    // 4 is a dependent of 1
4  3    // 3 is a dependent of 4
4  2 // 2 is a dependent of 4
:
:
```

2) inFile2 (use args[1]): a text file contains the processing times for nodes.
The first number in inFile2 is the number of nodes in the graph;
then follows by a list of pairs, <n_i, t_i>, where n_i is the node's id and t_i is the unit of processing times for node n_i.

For example: jobs take 1 unit of processing time;

```
6       // Graph has 6 nodes
1  1    // job time for node 1 is 1
2  1    // job time for node 2 is 1
3  1    // job time for node 3 is 1
        :
        :
```

another example: jobs take variable of processing time

```
6       // there are 6 nodes in the input graph
1  3    // job time for node 1 is 3
2  4    // job time for node 2 is 4
3  1    // job time for node 3 is 1
        :
        :
```

3) number processors (use args[2]) // > 0

****
II. Outputs: There are two (2) output files
****

1) outFile1: (use args[3]) for the intermediate and final results of the schedule table, nicely formatted.
For example:

```
==================================
ProcUsed : 3   currentTime: 7

      0   1   2   3   4   5  6   7 . .
     ---------------------------------
P(1) | 1 | 1 | 7 | 3 | 3 | 3 | - | 6 . . .
     ---------------------------------
P(2) | 2 | 4 | 4 | 4 | - | 5 | 5 | -   . . .
     ---------------------------------
P(3) | etc.
```

2) outFile2 (use args[4]): for all debugging outputs <u>to get partial credits if your program does not work completely!!</u>

****
III. Data structure:
****

- A node class
  - (int) jobID
  - (int) jobTime

- (node) next

Method:

- constructor (…)

- A schedule class
  - (int) numNodes // the total number of nodes in the input graph.
  - (int) numProcs // the number of processors can be used.
  - (int) procUsed // number of processors used so far; initialized to 0.
  - (int) currentTime // initialize to 0.
  - (int) totalJobTime // the total job times of nodes in the graph.
  - (int) jobTimeAry[] // an 1D array to store the job time of each node in the graph;
    // to be dynamically allocated, size of numNodes +1; initialied to 0.

  - (int) adjMatrix[][] // a 2-D integer array, size numNodes+1 by numNodes+1,
    // To represent the dependency graph; to be dynamically allocated, initialized to zero.

    // We use this matrix for a lot of ways as follows:

    // adjMatrix[0][0] to store number of nodes remain in the graph; initialize to numNodes;
    // decreases by 1, when a node is deleted; So, to check if the graph is empty,
    // just check if AdjMatrix[0][0] == 0, in O(1).

    // adjMatrix[i][j] >= 1, means node i is a parent of node j; and node j is a dependent of node i.
    // get it from inFile1.

    // adjMatrix[0][j], we use row 0 to store the parent counts of node j.
    //The parent counts of node j is the total count of none zero rows in j-th column.
    // So if we want to know the parent counts of node j, we check AdjMatrix[0][j], in O(1).

    // adjMatrix[i][0], we use column 0 to store the dependent counts of node i.
    //The dependent counts of node i is the total count of none zero columns in i-th row.
    // So if we want to know the dependent counts of node i, we check AdjMatrix[i][0] , in O(1)..

    // adjMatrix[i][i], the diagonal, [i][i] to indicate the status of the node, where i = 1 to numNodes
    // AdjMatrix[i][i] == 0 means node i is not in the graph.;
    //AdjMatrix[i][i] == 1 means node i is in the graph;
    //AdjMatrix[i][i] == 2 means node i is marked.

  - (int) Table[][] // a 2-D integer array, size of (numProcs +1) by (totalJobTime +1) for scheduling;
    // to be dynamically allocated, and initialized to 0.
    // Table row indices represent processor's id, and column indices represent time slots.
    // Table[i][T] > 0 means a job, jobID, is scheduled on the processor i for time slot T.
    // Table[i][T] <= 0 means the processor i is idled (available) at time slot T.
    // where i = 0 to numProcs and T = 0 to totalJobTime.

  - (node) OPEN // OPEN acts as the list head of a linked list with a dummy node.
    // Nodes in OPEN are sorted in <u>descending order</u> by the # of dependents of orphan nodes.
    // i.e., nodes with more dependents will be in the front of nodes with less dependents.
    // A node i's dependent count will be found in adjMatrix[i][0]

Methods:

- constructor (…) // take care all member allocations, initialization, etc.
- loadMatrix (inFile1) // read an edge <n_i, n_j> from inFile1and load to adjMatrix.
- int loadJobTimeAry (inFile2) // read each pair <jobID, time> from inFile2 and load to jobTimeAry;
  // set jobTimeAry[jobID] ← time; need to keep track of total job times;
  // returns totalJobTime
- setMatrix (…) // Compute parent counts and store in adjMatrix[0][j],

// computes dependent counts and store in adjMatrix[i][0];
//set diagonal entries AdjMatrix[i][i] to 1;
//set AdjMatrix[0][0] to numNodes.

- printMatrix (outFile2) // Print the entire content of adjMatrix with row and column indices in readable format.

- int findOrphan (…)
// Check AdjMatrix[0][j] to find the next un-marked orphan node, j, i.e., AdjMatrix[0][j] == 0 &&
//AdjMatrix[j][j] == 1. If found, mark the orphan, i.e., set AdjMatrix[j][j] ← 2, then returns j;
// if no such j, returns -1.

- OpenInsert (node) // on your own. Perform a linked list insertion;
// insert node into OPEN in the descending order by the # of dependents.

- printOPEN (outFile2) // print to outFile2, nodes in OPEN linked list
// Re-use codes similar to the printList method in your earlier projects.

- int getNextProc (currentTime) // on your own
// check Table [i][currentTime] to find the first i where Table [i][currentTime] == 0
// if found returns i, else returns -1, means no available processor.

- putJobOnTable (availProc, currentTime, jobID, jobTime) // see algorithm below.

- printTable (outFile1, currentTime) // print the scheduleTable upto the currentTime slot to outFile1,
// On your own, see the format description given in the above.

- bool checkCycle () // on your own.
Check the followings:
(1) OPEN is empty.
(2) Graph is not empty. // you should know where to check
(3) all processors are available. // you should know where to check

if all 3 conditions in the above are true,
returns true
else returns false

- bool isGraphEmpty () // on your own
//if AdjMatrix[0][0] == 0 returns true, else returns false. When you printMatrix, pay attention to
// the content of AdjMatrix[0][0].

- deleteJob (jobID) // see algorithm steps below.
// When a job is done, we delete the job and its outgoing edges from the graph,as follows.
// 1) To delete a job in the graph is to set adjMatrix[jobID][jobID] to 0 and decrease the
number of node in graph by 1, i.e., adjMatrix [0][0] --
// 2) To delete a job's outgoing edges is to decrease all its dependents the parent counts by 1.
Note: the job's dependents are those none zero adjMatrix[jobID][j] > 0, on jobID row
For example, if adjMatrix[jobID][j] > 0, then decrease adjMatrix[0][j] by 1; adjMatrix[0][j]--

****

IV. main(..)

****

step 0: inFile1, inFile2, outFile1, outFile2 ← open
numNodes ← read from inFile1.
adjMatrix [][] ← dynamically allocate, size of numNodes+1 by numNodes+1, initialize to zero
numProcs ← get from args[2]
if (numProcs <= 0) exit with error message “need 1 or more processors”.
else if (numProcs > numNodes)
numProcs ← numNodes // means unlimited processors.

```
        OPEN ← get a dummy node for OPEN to point to
        currentTime ←0 // at the beginning of scheduling
        procUsed ← 0 // 0 processor is used at the beginning
Step 1: loadMatrix (inFile1)
        totalJobTimes ← loadJobTimeAry (inFile2)
        Table [][] ← dynamically allocate, size of numProcs by totalJobTimes,  initialize to zero
        printTable (outFile1, currentTime)
Step 2: setMatrix (…)
        printMatrix(outFile2) // check for your self to make sure the matrix is corrected set.
Step 3: jobID ← findOrphan (…)
Step 4: if jobID > 0
            {newNode ← call node constructor with (jobID, jobTimeAry[jobID], null)
             OpenInsert (newNode)
             printOPEN(outFile2) // debug print
            }
Step 5: repeat step3 to step 4 until no more unmarked orphan
Step 6: availProc ← getNextProc(currentTime)
        if availProc >  0 // means there is a processor available
          { procUsed ++
            newJob ← remove the front node of OPEN after dummy node // newJob is a node!
            putJobOnTable (availProc, currentTime, newJob.jobID, newJob.jobTime)
          }
step 7: repeat step 6 while availProc > 0 && OPEN is not empty && ProcUsed < numProcs
Step 8: printTable (outFile1, currentTime)
step 9: hasCycle ← checkCycle ()
            if hasCycle == true
                output error message to console: “there is cycle in the graph!!!” and exit the program
Step 10: currentTime ++
Step 11: proc← 0
Step 12: if Table[proc][ currentTime] <=  0 && Table[proc][ currentTime - 1] >  0
                    // the processor, proc, just finished a job in the // previous time cycle.
                    {jobID ← Table[proc][ currentTime - 1]
                      deleteJob(jobID) // see algorithm steps below.
                    }
Step 13: printMatrix (outFile2)
step 14: proc ++
step 15: repeat step 12 to step 14 while proc <= procUsed
step 16: repeat step 3 to step 15 until isGraphEmpty ()
step 17: printTable (outFile1) // The final schedule table.
step 18: close all files
```

****

V. putJobOnTable (availProc, currentTime, jobId, jobTime)

****

```
Step 0:  Time ← currentTime
         EndTime ← Time + jobTime

Step 1: Table[availProc][Time] ← jobId
Step 2: Time ++
Step 3: repeat step 1 to step 2 while Time < EndTime
```

****

VI. deleteJob (jobID) // When a job is done, we delete the job and its outgoing edges.

****

Step 1: adjMatrix [jobID][jobID] ← 0 // delete jobID from the graph
adjMatrix [0][0] -- // one less node in the graph

Step 2: j ← 1
Step 3: if adjMatrix [jobID][j] > 0 // means j is a dependent of jobID
AdjMatrix [0][j] -- // decrease j's parent count by 1
Step 4: j ++
Step 5: repeat Step 3 to Step 4 while j <= numNodes