CS29003 ALGORITHMS LABORATORY

Last Date of Submission: August 30, 2018

General Instructions:

1. Please do not use any global variable unless you are explicitly instructed so.
2. Please use proper indentation in your code.
3. Please write your name and roll number at the beginning of your program.
4. **In this assignment, you will submit two .c files, each with its own main() function. Please name your files as f1_<roll_no>_<assignment_no> and f2_<roll_no>_<assignment_no>. For example, if your roll number is 14CS10001 then name your file as f1_14CS10001_7.c and f2_14CS10001_7.c (or cpp as applicable).**

Consider that a user runs different programs at different times, each taking some time, which have to be run on a computer with only one processor. We will refer to a program as a *job*. Each job is identified by a 4-tuple <*jobId*, *startTime*, *jobLength*, *remLength*>, where

- *jobId* is a unique integer id for the job, $1 \leq jobId \leq n$, where n is the total number of jobs
- *startTime* is the time (integer) at which the job is submitted, $0 \leq startTime$
- *jobLength* is the time duration for which the job will run on the processor
- *remLength* is the remaining time for which the job will run on the processor. Initially, *remLength = jobLength* for any job. As the job runs, *remLength* is decremented, until it becomes 0, at which point the job is said to have finished.

Since the processor can only run one job at a time, a special scheduler program will schedule the jobs on the processor. Time is broken up into intervals of length 1, with the interval [0,1] referred to as timestep 1, [1,2] referred to as timestep 2 and so on. A *schedule* shows the job that is to be run on the processor at each timestep until all jobs finish.

The scheduler maintains a priority queue of jobs waiting to be scheduled (all jobs that have arrived and have not finished, minus the currently executing job if any). When a job is executed on the processor, its *remLength* value is decremented by 1 after each timestep (giving the remaining time to finish the job). When the currently executing job finishes (*remLength* value 0), the scheduler removes the job with the least *remLength* value from the queue and schedules it to run on the processor (this job now becomes the currently executing job). When a new job x arrives, its job duration is checked with the *remLength* value of the currently executing job y. Two cases are possible:

1. If the job duration of x is less than the *remLength* value of y, y (with its current *remLength* value) is added to the scheduling queue and x becomes the currently executing job (i.e, the processor is taken off from y and given to x; y will finish later when it gets the processor again as per the policy).
2. Otherwise, x is added to the scheduling queue.

If there are two jobs with the same *remLength* value, the one with the lower job id is to be chosen.

The *turnaround time* of a job is the time taken to actually start the job, which is equal to the time at which it starts running on the processor for the first time minus its *startTime* value. The *average turnaround time* of a set of jobs is the average of the turnaround times of all the jobs in the set.

Write the following in the first .c (or .cpp) file to be submitted

Part 1: Implement the scheduling queue

In this part, you will implement the scheduling queue as a priority queue using the array implementation of a heap (start from index 1 in the array). Assume that the maximum number of jobs is 99. The scheduling queue is ordered by the *remLength* value of the jobs.

First define a type to represent a job as follows:

```
#define MAX_SIZE 100
typedef struct _job {
      int jobId;
      int startTime;
      int jobLength;
      int remLength;
} job;

typedef struct _heap {
      job list[MAX_SIZE];
      int numJobs;
} heap;
```

Then, implement the following functions:

void initHeap(heap *H): initializes the heap pointed to by *H* (just sets *numJobs* to 0)
void insertJob(heap *H, job j): inserts the job *j* in the heap pointed to by *H*
int extractMinJob(heap *H, job *j): If the heap is empty, returns -1. Otherwise deletes the minimum element from the heap, sets **j* to it, and returns 0

Part 2: Implement the scheduler

The scheduler is implemented as a function

void scheduler(job jobList[], int n)

The function takes as parameter a list of *n* jobs in an array *jobList*[ ], and schedules them according to the policy specified earlier.

Note that in real life, the jobs will be coming one by one at their respective start times, and will not all be known in advance in an array. However, in this program, we are just simulating what will happen if they come at those start times, and the array of jobs gives the required information to simulate this.

The function prints the job ids of the jobs running on the processor for every timestep till all jobs are finished. It also prints the average turnaround time of the jobs at the end.

Your function should run in O(*nlgn* + *T*) time, where *T* is the sum of the job durations of all the jobs (Note that it can be done also in O(*nlgn*) time independent of *T*, but not required for this assignment; ok if you do it though).

main( ) function:

Your main function should do the following:

1. Read the number of jobs *n*
2. Dynamically allocate an array of *n job* structures
3. Read the *jobId*, *startTime*, and *jobLength* values of each job (exactly in this order) one by one in the array
4. Call **scheduler()** to schedule the jobs and print out the schedule for each timestep and the average turnaround time

Sample output:

```
Enter no. of jobs (n): 4
Enter the jobs:
1 0 21
2 1 3
3 2 6
4 3 2
Jobs scheduled at each timestep are:
1 2 2 2 4 4 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Average Turnaround Time is 1.25
```

Write the following in the second .c (or .cpp) file to be submitted

(For this part, start with a copy of the first file and modify it. Part 3 is only incremental changes to Parts 1 and 2)

Part 3: Implement a modified scheduler

Now suppose that all the jobs are not independent, and there exists some pairs of jobs (*x*,*y*) such that if job *x* finishes **before** *y* starts, it can reduce the running time (*remLength* value) of *y* by 50% . We call such a pair a *job-dependency pair*. You can assume that there can be O(*n*) such pairs given. Note that the relationship is one way only, y depends on x does not imply x depends on y, so (*x*,*y*) and (*y*,*x*) are different pairs.

You can represent each such job-dependency pair by the structure

```
typedef struct _jobPair {
     int jobid_from;
     int jobid_to;
} jobpair;
```

Your job is to schedule the jobs using the same policy, subject to the change mentioned above. However, note that in this case, the duration of a job already in the heap can change in the middle if another job that it depends on finishes. Your function should still run in O(*nlogn* + *T*) time. To do this, you will have to change the definition of the heap and the associated functions, and add a new function. Once again, the changes are incremental, so copy the file for Part 2 into a new file, and then modify it.

- Define a new type `newheap`. This should have the same fields as the `heap` type, plus any other field you may want to add
- Write the functions
    - **void initHeap(newheap *H):** initializes the heap
    - **void insertJob(newheap *H, job j):** same as earlier
    - **int extractMinJob(newheap *H, job *j):** same as earlier
    - **void decreaseKey(newheap *H, int jid):** decreases the value of the *remLength* field of the job with job id *j* in heap H by 50% (take floor if resultant value is not an integer) only if the job *j* has not started yet; if *j* has already started, nothing is done. Note that changing the value may violate the heap property so you will need to restore the heap property if so. You may in addition change the *jobLength* value also if it helps you.
- Finally write the new scheduler function

  void newScheduler(job jobList[], int n, jobpair pairList[], int m): takes a list of *n* jobs in the array *jobList* and a list of *m* job-dependency pairs in the array *pairList* as parameter, and schedules the jobs.

main( ) function:

Your main function should do the following:

5. Read the number of jobs *n*
6. Dynamically allocate an array of *n job* structures
7. Read the *jobId*, *startTime*, and *jobLength* values of each job (exactly in this order) one by one
8. Read the number of dependency-pairs *m*
9. Dynamically allocate an array of *m jobpair* structure to store the *m* job-dependency pairs
10. Read the job-dependency pairs one by one
11. Call **newScheduler() to** schedule the jobs and print out the schedule for each timestep and the average turnaround time

Sample output:

```
Enter no. of jobs (n): 4
Enter the jobs:
1 0 21
2 1 3
3 2 6
4 3 2
1
2 3
Jobs scheduled at each timestep are:
1 2 2 2 4 4 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Average Turnaround Time is 1.25
```