CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 9 (Heaps)
Date: 5th November, 2020

Important Instructions

1. **Files to be submitted:** ROLLNO_GroupNO_A9.c/.cpp and ROLLNO_GroupNO_A9_p3.c/.cpp
2. You are to **stick to the output formats strictly** as per the instructions.
3. Submission through **.zip files are not allowed**.
4. Write your name and roll number at the beginning of your program.
5. Do not use any global variable unless you are explicitly instructed so.
6. Use proper indentation in your code.
7. Please follow all the guidelines (function prototypes, etc.). Failing to do so will cause you to lose marks.
8. **There will be part marking.**

Let's revisit the refrigerator servicing problem from your last worksheet. The servicing facility can handle just one refrigerator at a time. Suppose that on a particular day there are n refrigerators awaiting repair. Arrival of refrigerator i is specified by a start time s_i, and a processing time p_i. We consider that at any time repairing of a refrigerator can be suspended and then completed later.
We will refer to each refrigerator servicing as a *job*. A *schedule* specifies for each unit time interval, the unique job that is run during that time interval. In a feasible schedule, every job J_i has to be run for exactly p_i time units (*jobLength*) after time s_i (*startTime*). The completion time C_i for job J_i is the earliest time when J_i has been run for p_i time units. In the worksheet, you were asked to give a schedule that minimizes

$$\sum_{j=1}^{n} C_j$$

We want to schedule in an online fashion and it would look as follows:

$t = 0$
while there are jobs left not completely scheduled

- Among those jobs J_i such that $s_i \leq t$
- pick a job J_m to schedule at time t according to some rule;
- increment t

The greedy algorithm that works in this case is as follows:
Let $y_{i,t}$ be the total time that job J_i has been run before time t. Pick J_m to be a job that has minimal remaining processing time, that it, that has minimal $p_i - y_{i,t}$ (*remLength*). Ties may be broken based on some convention. We will use a unique *jobId* so that in the case of ties, lower jobId will be preferred.
An efficient implementation of this greedy algorithm can be achieved by the scheduler maintaining 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). 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 (s_i in description), $0 \leq startTime$
- *jobLength* is the time duration for which the job will run on the processor (p_i is description)
- *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.

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

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:

- 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).
- 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.

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.
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.

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.

Main function

Your main function should do the following:

- Read the number of jobs n
- Dynamically allocate an array of n *job* structures
- Read the *jobId*, *startTime*, and *jobLength* values of each job (exactly in this order) one by one in the array
- 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 3 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
```

Note that the individual turnaround times for the jobs are 0, 0, 4 and 1, respectively. The jobs need not be sorted in any order. If you need to sort in any order, you should use counting sort. Assume all ranges are of the order of the input.

Part 3: Implement a modified scheduler

Now suppose that all the jobs are not independent, and there exist 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%, if it has not started yet. 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 may 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, thus remLength 5 is changed to 2) 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:

- Read the number of jobs n
- Dynamically allocate an array of n *job* structures
- Read the *jobId*, *startTime*, and *jobLength* values of each job (exactly in this order) one by one
- Read the number of dependency-pairs m
- Dynamically allocate an array of m *jobpair* structure to store the m job-dependency pairs
- Read the job-dependency pairs one by one
- 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
Enter the number of dependency pairs
1
Enter the dependency pairs
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
Average Turnaround Time is 1.25
```

Note here that since job 2 is finished before job 3 starts, the job 3 is run only for 3 timesteps. However, the turnaround times still remain the same.