COP 3502C Programming Assignment#5

Topics covered: Heap and Priority Queue

Please check Webcourses for the Due Date

Read all the pages before starting to write your code

Introduction: For this assignment you have to write a c program that will utilize the Heap and Priority Queue.

What should you submit?

Write all the code in a single .c file and upload the .c file.

Please include the following lines in the beginning of your code to declare that the code was written by you:

/* COP 3502C Programming Assignment 5

This program is written by: Your Full Name */

Compliance with Rules: UCF Golden rules apply towards this assignment and submission. Assignment rules mentioned in syllabus, are also applied in this submission. The TA and Instructor can call any students for explaining any part of the code in order to better assess your authorship and for further clarification if needed.

Caution!!!

Sharing this assignment description (fully or partly) as well as your code (fully or partly) to anyone/anywhere is a violation of the policy. I may report such incidence to office of student conduct and an investigation can easily trace the student who shared/posted it. Also, getting a part of code from anywhere will be considered as cheating.

Deadline:

See the deadline in Webcourses. **No late submission will be accepted.** The assignment submission will be locked after the deadline. An assignment submitted by email will not be graded and such emails will not be replied according to the course policy.

Additional notes: The TAs and course instructor can call any students to explain their code for grading.

Problem: Procrastination at its Finest

Between school, your job and your parents, many people are demanding you to do lots of jobs. Part of the reason they ask you to do so much is that they have high expectations of you.

You've recently learned about the priority queue in Computer Science I and have figured out a "fair" way to lower everyone's expectations of you. Every task that someone asks you to do can be broken down into several segments, and each of those segments takes some known amount of time (in seconds). For example, your recitation program may be done in 5 phases, the first taking 500 seconds, the second taking 1800 seconds, the third taking 5000 seconds, the fourth taking 2000 seconds and the last taking 45 seconds. Whenever working on a task, you just work on the next phase of the task to completion. We assume that no one ever interrupts you, so that you are able to complete a phase you are working on in the time listed to complete it.

With so many people piling up tasks on you, it can become hard to choose what to do. In order to lower everyone's expectations, you've decided that you want to come up with a rule to determine which task to work on next so that you don't really complete any task very quickly!

The idea is as follows:

For each task, keep track of the amount of time remaining, you need to complete the task. For example, if you had completed the first two phases of the recitation program previously mentioned, you would have to do an additional 5000 + 2000 + 45 = 7045 seconds of work to complete the task.

Then, of all the remaining unfinished tasks, pick the one which has THE MOST work left to complete, breaking ties by the task number (each task will be given a unique identifier number), with the lower task number being chosen first.

Once you choose a task, complete the next phase of that task. If this was the last phase of that task, you've completed the task. If not, then put this task back into your list of work to do (but now it'll have fewer seconds of work left to complete it.)

Then, repeat the process again - choose the next task (could be the same one you chose last time or a different one), that has the most amount of work left to be done.

Of course, different people come to you with tasks at different times, so while you are in the middle of working on some tasks, new tasks may be added to your list of tasks to complete.

The Problem

Given a list of jobs that people hand you to do, that are broken down into phases, as well as the time people give you the task, using the mechanism described, simulate the process of executing the tasks, calculating the time at which each task will get completed. To show that progress is being made, at each time you complete any phase of any task, print out the time, the task number and the phase number that was completed.

The Input (to be read from in.txt file) - Your Program Will Be Tested on Multiple Files

The first line of the input contains a single positive integer, n ($n \leq 10^6$), representing the number of tasks given to you. Information for each task follows, with information about a single task on one line. The tasks are numbered 1 to n, and ae listed in order by task number.

Each of these lines will have several space separated positive integers.

The i^{th} of these lines starts with a positive integer t_i ($t_i \leq 10^9$), representing the time the i^{th} task is given to you. This is followed by, p_i ($p_i \leq 100$), representing the number of phases for the i^{th} task. This is followed by p_i positive integers, representing the amount of time, in seconds, each phase of the i^{th} task takes, in order. The sum of all the p_i's will not exceed 10^6.

It is guaranteed that the times the tasks are given to you will be in strictly increasing order. (Thus, you'll receive task 1 first, then task 2 at a later time, then task 3 at a time after that, and so forth. Mathematically, $t_i < t_{i+1}$ for all i, $1 \leq i < n$.)

The sum of the time it will take to complete all tasks will not exceed 1,000,000,000 seconds.

The Output (to be printed to out.txt file)

Each time a phase of a task is completed, output a single line with three space separated integers:

The time the phase was completed, the task it was taken from (in between 1 and n, inclusive), and the phase number from that task that was completed (in between 1 and p_i, inclusive).

Sample Input (in.txt file)

```
5
10 3 180 3000 2000
200 5 1800 5000 7000 1000 60
1000 2 20000 1
2000 8 1000 1000 1000 1000 1000 1000 1000 1000
1000000 6 10 20 30 30 20 10
```

Sample Output (out.txt file)

```
190 1 1
3190 1 2
23190 3 1
24990 2 1
29990 2 2
36990 2 3
37990 4 1
38990 4 2
39990 4 3
40990 4 4
41990 4 5
42990 4 6
44990 1 3
45990 4 7
46990 2 4
47990 4 8
48050 2 5
48051 3 2
1000010 5 1
1000030 5 2
1000060 5 3
1000090 5 4
1000110 5 5
1000120 5 6
```

Sample Explanation (Also see the additional explanation file uploaded in the assignment 4 folder)

At first, task 1 is the only one to do. This is true for the first two phases of it. But, after the second phase of task 1 is completed, tasks 2, 3 and 4 have already been added to the priority queue. At this point, task 3 is the furthest from being completed, so we spend 20,000 seconds on its first phase. After this, task 2 is furthers from being completed and continues to be so as we do phases 1, 2 and 3 of task 2. After completing these 3 phases, task 4, with 8,000 seconds left, takes priority. It keeps the priority for 6 consecutive phases until t = 42990. At this time, task 1 has 2000 seconds left, task 2 has 1060 seconds left, task 3 has 1 second left, and task 4 has 200 seconds left. In this case, the tie-break goes to task 1, which finally completes at time t = 44,990. After this task though, we go back to task 4 phase 7, completing that at t = 46,990. At this point in time, task 2 has 1060 seconds left, task 3 has 1 second left and task 4 has 1000 seconds left. Task 2 is chosen. Then, in

succession, task 4 completes, followed by task 2 and lastly by task 3. After much idle time, task 5 runs from start to finish.

Implementation Restrictions, Assignment Details

1) You must create a struct to store information about a task:

a) Its ID Number

b) Time Assigned

c) Number of Phases

d) Integer array, dynamically allocated storing the lengths of each phase

e) Total Time Left to Complete the Task

f) 0-based number of which phase to complete next.

2) Write a compareTo function that takes in two pointers to the struct you created, and returns a negative integer if the first struct "comes first", a positive integer if the first struct "comes second", and 0 if they are identical. One struct comes before another for the purposes of this program if the total time left to complete the task is longer, or if the total time is the same and its ID number is lower.

3) Create another struct, which is a binary heap of the struct previously mentioned. This struct should store the heap in a dynamically allocated array, that gets resized as needed. Start with a default size of 10.

4) In your implementation, use a single binary heap of tasks. When a new task is given to you, the task is inserted into the binary heap. When you choose a task to work on, you remove the task from the heap. When you complete a phase of a task, if the task isn't complete, you reinsert the task in its new state (with less time left to complete it), back into the heap. Note that while you are "working" on a single phase of a task, several new tasks may be added to your heap. To make things a bit easier, I recommend reading in all of the tasks into a regular array at first, before starting the simulation, and then indexing into that array during the simulation.

5) Though all tasks are assigned at unique times, it's possible that you may complete a phase of a task (but not finish the task) at the exact same time that a new task is given to you. In this situation, add BOTH the new task AND the task with the phase you just completed to the binary heap. AFTER that, select which task to work on next. (Thus, it's possible that either the task you were just working on, OR the new task you just got, or another task will be selected at this point.)

Additional Requirement:

You must have to use read data from in.txt file and write the result to out.txt file. The output must have to match with the sample output format. Do not add additional space, extra characters or words with the output as we will use diff command to check whether your result matches with our result. Next, you must have to write a well structure code. ***There will be a deduction of 10% points if the code is not well indented and 5% for not commenting important blocks.***

- As always, all the programming submission will be graded base on the result from **Eustis**. If your code does not work in Eustis we will conclude that your code has an error even if it works in your computer.
- Your code should contain the memory leak detector like programming assignment 1 (otherwise 10% penalty)
- Again, your program must have to compile in Eustis to get points. Note that you can use <math.h> library if you need. In that case you have to use –lm option while compiling your code.

For example: $*gcc filename.c -lm*

Steps to check your output AUTOMATICALLY in Eustis or repl.it:

You can run the following commands to check whether your output is exactly matching with the sample output or not.

Step1: Copy the sample output into sample_out.txt file and move it to the server (you can make your own sample_out.txt file)

Step2: compile and run your code using typical gcc and other commands. Your code should produce out.txt file.

Step3: Run the following command to compare your out.txt file with the sample output file

```
$diff -i out.txt sample_out.txt
```

The command will not produce any output if the files contain exactly same data. Otherwise, it will tell you the lines with mismatches.

Incase if your code does not match, you can use the following command to see the result in side by side:

```
$diff -y out.txt sample_out.txt
```

Rubric:

1) A code not compiling or creating seg fault without producing any result can get zero. There may or may not be any partial credit. But at least they will not get more than 50% even if it is a small reason. Because, we cannot test those code at all against our test cases to see whether it produces the correct result or not.
2) There is no grade for just writing the required functions. However, there will be 20% penalty for not CompareTo function.
3) Inserting into heap properly : 25%
4) Deleting from heap properly: 25%
5) Matching test cases : 50%. Your code will be tested against a set of test files and will be grade based on the match.
6) There is no point for well structured, commented and well indented code. ***There will be a deduction of 10% points if the code is not well indented and 5% for not commenting important blocks.***

Hints:

- Start programming assignment as soon as possible.
- Read the entire assignment and get a very good idea what is expected.
- Use the sample input/output to understand it better
- Draw and simulate the sample input/output step by step by drawing the example heap partially to get better idea about implementation.
- You have learned heap implementation with integer. In the assignment, you have to store structure in the heap. In order to make a decision which structure is bigger or smaller, use the compareTo function to make a decision.