CSCI-1200 Data Structures — Fall 2025
Homework 4 — Autograding Scheduler

In this assignment we will write a program to simulate the scheduling queue that manages Submitty autograding. Historically, the busiest time period on Submitty is Thursday evening from 11pm-midnight. Many different courses in the Computer Science department have a common deadline of 11:59pm, and a significant number of students of all levels are afflicted with *procrastination*. All data for this assignment is synthetic, but it has been created following the typical data patterns we see each semester. *Please carefully read the entire assignment before beginning your implementation.*

We have a number of dedicated high end server machines with multiple processors; however, sometimes a spike of submissions cause a backlog and students will have to wait in a *first in, first out (FIFO)* queue until sufficient processor resources are available. Students may experience a moderate time delay before their autograding results are available. *But do not worry! Due date / late days are always calculated from the upload timestamp, not the timestamp when autograding completes.*

Here is a small example of the input file, `sample_jobs.txt`:

```
23:00:09 csci1100 hw3 jonesp 1 10  9
23:00:11 csci1100 hw1 leek   1 10 10
23:00:14 csci2600 hw3 smitha 1 60 43
23:00:18 csci2800 p1  bakert 4 30 19
23:00:24 csci1200 hw3 hallb  1 25 23
23:00:28 csci1100 hw1 cookr  1 10  9
```

Each line of the input file is a student submission containing the upload timestamp, course name, gradeable name, student username, number of processors needed to run the autograding, the maximum runtime in seconds allowed for the assignment, and the actual runtime for that student's submission (which is $\leq$ the maximum runtime). Here's a sample command line showing how we will run your program for this assignment:

```
./simulate.out sample_jobs.txt 5 --algorithm single_file --visualization vis.txt --log log.txt
```

The visualization output file will contain an ASCII art table showing a simulated schedule for these jobs:

```
timestamp| processor 0         | processor 1         | processor 2         | processor 3         | processor 4         | queue
23:00:00 |                     |                     |                     |                     |                     |    0
23:00:05 |                     |                     |                     |                     |                     |    0
23:00:10 | csci1100_hw3_jonesp |                     |                     |                     |                     |    0
23:00:15 | csci1100_hw3_jonesp |                     |                     |                     |                     |    2
23:00:20 | csci1100_hw1_leek   |                     |                     |                     |                     |    2
23:00:25 | csci1100_hw1_leek   |                     |                     |                     |                     |    3
23:00:30 | csci1100_hw1_leek   |                     |                     |                     |                     |    4
23:00:35 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:00:40 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:00:45 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:00:50 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:00:55 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:01:00 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:01:05 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:01:10 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:01:15 | csci2600_hw3_smitha |                     |                     |                     |                     |    3
23:01:20 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |    2
23:01:25 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |    2
23:01:30 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |    2
23:01:35 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |    2
23:01:40 | csci1200_hw3_hallb  |                     |                     |                     |                     |    1
23:01:45 | csci1200_hw3_hallb  |                     |                     |                     |                     |    1
23:01:50 | csci1200_hw3_hallb  |                     |                     |                     |                     |    1
23:01:55 | csci1200_hw3_hallb  |                     |                     |                     |                     |    1
23:02:00 | csci1200_hw3_hallb  |                     |                     |                     |                     |    1
23:02:05 | csci1100_hw1_cookr  |                     |                     |                     |                     |    0
23:02:10 | csci1100_hw1_cookr  |                     |                     |                     |                     |    0
23:02:15 |                     |                     |                     |                     |                     |    0
```

The visualization contains a vertical column for every processor available on the Submitty system. Each row of the output represents a 5 second time interval. Each cell of the table indicates which assignment is being graded by the processor during that interval, or the cell is empty if the processor is idle.

The command line and visualization above demonstrate the simplest scheduling algorithm you will implement, "*single file*". With this algorithm the system will only grade one student at a time. The autograding will always be assigned to processor 0. The other system processors will only be used if the assignment specifies that it should be run in parallel on multiple processors. For this assignment, we will require that an autograding assignment that uses multiple processors must run on *adjacent processors*. So an autograding job that needs 4 processors must run on processors 0,1,2&3, or 1,2,3&4, or 2,3,4&5, etc.

Your program will also produce a log file that contains the scheduling details that correspond to the ASCII art visualization. The log file is ordered by finish time:

```
csci1100_hw3_jonesp   upload: 23:00:09  started grading: 23:00:10  finished grading: 23:00:19  wait_time:    1 sec  grading_time:    9 sec
csci1100_hw1_leek     upload: 23:00:11  started grading: 23:00:20  finished grading: 23:00:30  wait_time:    9 sec  grading_time:   10 sec
csci2600_hw3_smitha   upload: 23:00:14  started grading: 23:00:35  finished grading: 23:01:18  wait_time:   21 sec  grading_time:   43 sec
csci2800_p1_bakert    upload: 23:00:18  started grading: 23:01:20  finished grading: 23:01:39  wait_time:   62 sec  grading_time:   19 sec
csci1200_hw3_hallb    upload: 23:00:24  started grading: 23:01:40  finished grading: 23:02:03  wait_time:   76 sec  grading_time:   23 sec
csci1100_hw1_cookr    upload: 23:00:28  started grading: 23:02:05  finished grading: 23:02:14  wait_time:   97 sec  grading_time:    9 sec
```

Your program will also print simple summary statistics about the simulation to `std::cout`:

```
number of jobs:                     6
scheduling algorithm:     single_file
data structure:             std::list
time to empty queue:        140    sec
average waiting time:         44.33 sec
maximum waiting time:       97     sec
idle percentage:             75.71   %
simulation running time:      0.00 sec
```

Additional Scheduling Algorithms

If the command line instead specifies `--algorithm first_available`, the system will run the first item in the queue on the lowest number processor(s) that are idle. If sufficient resources are not available for the first job in the FIFO queue, the system will wait until other jobs finish and the necessary resources become available. Here is the output of the earlier example using the "*first available*" algorithm:

```
timestamp| processor 0         | processor 1         | processor 2         | processor 3         | processor 4         | queue
23:00:00 |                     |                     |                     |                     |                     |     0
23:00:05 |                     |                     |                     |                     |                     |     0
23:00:10 | csci1100_hw3_jonesp |                     |                     |                     |                     |     0
23:00:15 | csci1100_hw3_jonesp | csci1100_hw1_leek   | csci2600_hw3_smitha |                     |                     |     0
23:00:20 |                     | csci1100_hw1_leek   | csci2600_hw3_smitha |                     |                     |     1
23:00:25 |                     | csci1100_hw1_leek   | csci2600_hw3_smitha |                     |                     |     2
23:00:30 |                     |                     | csci2600_hw3_smitha |                     |                     |     3
23:00:35 |                     |                     | csci2600_hw3_smitha |                     |                     |     3
23:00:40 |                     |                     | csci2600_hw3_smitha |                     |                     |     3
23:00:45 |                     |                     | csci2600_hw3_smitha |                     |                     |     3
23:00:50 |                     |                     | csci2600_hw3_smitha |                     |                     |     3
23:00:55 |                     |                     | csci2600_hw3_smitha |                     |                     |     3
23:01:00 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci1200_hw3_hallb  |     1
23:01:05 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci1200_hw3_hallb  |     1
23:01:10 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci1200_hw3_hallb  |     1
23:01:15 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci1200_hw3_hallb  |     1
23:01:20 | csci1100_hw1_cookr  |                     |                     |                     | csci1200_hw3_hallb  |     0
23:01:25 | csci1100_hw1_cookr  |                     |                     |                     |                     |     0
23:01:30 |                     |                     |                     |                     |                     |     0
```

We now see that the first 3 students (`jonesp`, `leek`, and `smitha`) can run at the same time on different processors. Since the 4th student (`bakert`) needs 4 adjacent processors, this job cannot be started until the job running on processor 2 finishes. Here is the log file from this run:

```
csci1100_hw3_jonesp   upload: 23:00:09  started grading: 23:00:10  finished grading: 23:00:19  wait_time:    1 sec  grading_time:    9 sec
csci1100_hw1_leek     upload: 23:00:11  started grading: 23:00:15  finished grading: 23:00:25  wait_time:    4 sec  grading_time:   10 sec
csci2600_hw3_smitha   upload: 23:00:14  started grading: 23:00:15  finished grading: 23:00:58  wait_time:    1 sec  grading_time:   43 sec
csci2800_p1_bakert    upload: 23:00:18  started grading: 23:01:00  finished grading: 23:01:19  wait_time:   42 sec  grading_time:   19 sec
csci1200_hw3_hallb    upload: 23:00:24  started grading: 23:01:00  finished grading: 23:01:23  wait_time:   36 sec  grading_time:   23 sec
csci1100_hw1_cookr    upload: 23:00:28  started grading: 23:01:20  finished grading: 23:01:29  wait_time:   52 sec  grading_time:    9 sec
```

The corresponding statistics `std::cout` output shows that the simulation empties the queue earlier, and the waiting time is reduced and the idle percentage of the server is decreased:

```
number of jobs:                         6
scheduling algorithm:    first_available
data structure:                std::list
time to empty queue:           95    sec
average waiting time:          22.67 sec
maximum waiting time:          52    sec
idle percentage:               64.21   %
simulation running time:        0.00 sec
```

In the previous example, you may have noticed that the fifth and sixth students were delayed even though their resource needs were minimal and processors were idle. With the third scheduling algorithm you will implement, `--algorithm keep_busy`, the system will try to keep more/all of the processors busy. If there are idle processors, the algorithm will look into the queue of waiting jobs – *beyond the first item* – for a job that can immediately use the available resources. Here is the output of the same example using the "*keep busy*" algorithm:

```
timestamp| processor 0         | processor 1         | processor 2         | processor 3         | processor 4         | queue
23:00:00 |                     |                     |                     |                     |                     |     0
23:00:05 |                     |                     |                     |                     |                     |     0
23:00:10 | csci1100_hw3_jonesp |                     |                     |                     |                     |     0
23:00:15 | csci1100_hw3_jonesp | csci1100_hw1_leek   | csci2600_hw3_smitha |                     |                     |     0
23:00:20 |                     | csci1100_hw1_leek   | csci2600_hw3_smitha |                     |                     |     1
23:00:25 | csci1200_hw3_hallb  | csci1100_hw1_leek   | csci2600_hw3_smitha |                     |                     |     1
23:00:30 | csci1200_hw3_hallb  | csci1100_hw1_cookr  | csci2600_hw3_smitha |                     |                     |     1
23:00:35 | csci1200_hw3_hallb  | csci1100_hw1_cookr  | csci2600_hw3_smitha |                     |                     |     1
23:00:40 | csci1200_hw3_hallb  |                     | csci2600_hw3_smitha |                     |                     |     1
23:00:45 | csci1200_hw3_hallb  |                     | csci2600_hw3_smitha |                     |                     |     1
23:00:50 |                     |                     | csci2600_hw3_smitha |                     |                     |     1
23:00:55 |                     |                     | csci2600_hw3_smitha |                     |                     |     1
23:01:00 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |     0
23:01:05 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |     0
23:01:10 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |     0
23:01:15 | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  | csci2800_p1_bakert  |                     |     0
23:01:20 |                     |                     |                     |                     |                     |     0
```

We can see that the 5th and 6th students `hallb` and `cookr` are now scheduled ahead of the 4th student `bakert`, who needs more resources for autograding. Here is the log file for the *keep busy* algorithm:

```
csci1100_hw3_jonesp   upload: 23:00:09  started grading: 23:00:10  finished grading: 23:00:19  wait_time:    1 sec  grading_time:    9 sec
csci1100_hw1_leek     upload: 23:00:11  started grading: 23:00:15  finished grading: 23:00:25  wait_time:    4 sec  grading_time:   10 sec
csci1100_hw1_cookr    upload: 23:00:28  started grading: 23:00:30  finished grading: 23:00:39  wait_time:    2 sec  grading_time:    9 sec
csci1200_hw3_hallb    upload: 23:00:24  started grading: 23:00:25  finished grading: 23:00:48  wait_time:    1 sec  grading_time:   23 sec
csci2600_hw3_smitha   upload: 23:00:14  started grading: 23:00:15  finished grading: 23:00:58  wait_time:    1 sec  grading_time:   43 sec
csci2800_p1_bakert    upload: 23:00:18  started grading: 23:01:00  finished grading: 23:01:19  wait_time:   42 sec  grading_time:   19 sec
```

And the expected improvements are also shown in the statistics:

```
number of jobs:                         6
scheduling algorithm:          keep_busy
data structure:                std::list
time to empty queue:           85    sec
average waiting time:           8.50 sec
maximum waiting time:          42    sec
idle percentage:               60.00   %
simulation running time:        0.00 sec
```

These sample input and output files and additional, larger input files are posted on the course website.

After you have implemented and debugged these scheduling algorithms, you will test and analyze the performance of these three algorithms using the provided larger synthetic input datasets on larger simulated Submitty systems with proportionally more processors. What works well? What can be improved? Consider how you might use the maximum runtime for each upcoming job in the queue. *NOTE:* Our scheduling algorithm should not use the *actual* runtime when deciding which job to run on which processor, since that future knowledge is not known during real-world autograding!

You will design and implement your own scheduling algorithm (run with `--algorithm custom`). In your README.txt file you will describe the goals of your scheduling algorithm, how you implemented it, and how it performs compared to the 3 required algorithms. We will create a leaderboard for this assignment so you can see how well your custom algorithm does compared to your classmates. A small amount of extra credit will be awarded to students with interesting algorithms that perform well relative to their classmates.

Queue Implementation Requirements and Performance Analysis

In lecture we discussed the similarities and differences between STL's two primary sequential data structures, the STL `vector` and the STL `list`. We said that the STL `list` was preferable for applications that involve frequent insertions (and/or removals) at the front or middle of the structure. Thus the primary/default data structure you will use for the *first-in, first-out (FIFO) queue* used in the 4 scheduling algorithms (3 required and 1 custom) will be the STL `list`.

To confirm that `list` has better performance than `vector` for this application, we will use *preprocessor directives* in the code to switch from a `list` to a `vector`. In your implementation, anywhere the code for using the two structures is different you will surround the code with a preprocessor `#if`, `#else`, `#endif` block. For example:

```
#ifdef VECTOR
  std::vector<Job> todo;
#else
  std::list<Job> todo;
#endif
```

By default your code will use STL `list`, but if the `-DVECTOR` option is added to your `g++/clang++` compilation line the compiled executable will instead use the code in the `VECTOR` section of the preprocessor directives.

```
g++ -Wall -Wextra timestamp.cpp simulator.cpp main.cpp          -o simulate.out
g++ -Wall -Wextra timestamp.cpp simulator.cpp main.cpp -DVECTOR -o simulate_vector.out
```

Test each of the scheduling algorithms with both `vector` and `list` on autograding scheduling simulations of increasing size. Record this data and summarize the results of this testing in your `README.txt`. Discuss whether this matches your expectations given the Big O Notation of key functions for STL `list` vs. `vector`. The final line of the statistics output measures the wall clock running time – *you'll need to use larger input files to see a difference in the running time!*

Provided Code and Additional Instructions

We have provided the `main.cpp` file which parses the input file and the `Timestamp` and `Job` classes. You should not need to modify anything in these files. Your task is to complete the scheduling algorithms in the `Simulator` class and complete the code to collect and output statistics about the simulation. Be sure to study all of the provided code before you get started. The code to format the output data should be helpful to ensure that your assignment can be correctly autograded – *Yes, isn't it ironic?*

In your README.txt file, provide a Big O Notation complexity analysis for the simulation using each of the 4 algorithms. Focus first on the running time using an STL `list`, then you can discuss how the answer is different when you use a `vector`. Use j to represent the total number of student autograding submissions/jobs, use p for the total number of processors on the Submitty system, use n for the average or maximum number of processors needed by a single job, use r to be the average or maximum runtime of a single job, use q as the maximum number of jobs in the queue at any time, and use t for the number of timesteps to empty the queue and complete all autograding in the simulation. Discuss how the variables j, p, n, r, q, and t are related to each other.

In your README.txt file collect and organize the results of your testing with different input files and command line arguments. Discuss how the simulation results compare to your Big O Notation analysis. And as always, be sure to list your collaborators and all resources used in completing the assignment in your README.txt.