Q1: Unix Shell

The shell has the following files:

- main.cpp - containing the primary flow of the program
- helpers.cpp - functions which do exactly what the respective names say
- helpers.hpp - contains function prototypes in helpers.cpp
- commands - binary file to store commands the shell can execute from
- environment - contains the environment variables of the shell (execution path only as of now as per the assignment requires)
- README.md - readme for the shell

In main.cpp:
main() function primarily decides whether to run the shell in realtime (prompt mode - default shell behaviors) or to take input from an external *binary* file. Accordingly, it calls the run_shell() function which accepts the string entered by the user, pushes it into the histfile (which stores the shell history on disk) , parses it into *args* (argv type array) and executes the respective command.
To keep the robert concise, the function definitions are not explained here as the code is pretty self-explanatory. Still, if needed it can be explained by the submitters.

In helpers.cpp

Algorithm for the simulation:
The code written simulates the following algorithm

1. A student in the odd seat picks up left spoon first, then picks up the right spoon, eats for 20 s, drops the right spoon, drops the left spoon and thinks for 2s and repeats.
2. A student in the even seat picks up the right spoon first, then picks up the left spoon, eats for 20s, drops the left spoon, drops the right spoons and thinks for 2s and repeats.

We've used semaphores to simulate the spoons for this problem (semaphore array of length with all their initial value set to 1). We call sem_wait() for a student to wait for a spoon and sem_signal() to indicate that the spoon is now free. This provides mutual exclusion for any spoon and ensures that only one student can use a particular spoon.

This solution is deadlock free as there will never be a cycle of students waiting for each other to drop the spoons.

As per the output generated file,

Output1:
S1 got to eat 41 times.
S2 got to eat 37 times.
S3 got to eat 36 times.
S4 got to eat 31 times.
S5 got to eat 30 times.

Output2:
S1 got to eat 40 times.
S2 got to eat 38 times.
S3 got to eat 39 times.
S4 got to eat 30 times.
S5 got to eat 29 times.

Output3:
S1 got to eat 42 times.
S2 got to eat 37 times.
S3 got to eat 37 times.
S4 got to eat 31 times.
S5 got to eat 30 times.

Output4:
S1 got to eat 40 times.
S2 got to eat 33 times.
S3 got to eat 32 times.
S4 got to eat 36 times.
S5 got to eat 35 times.

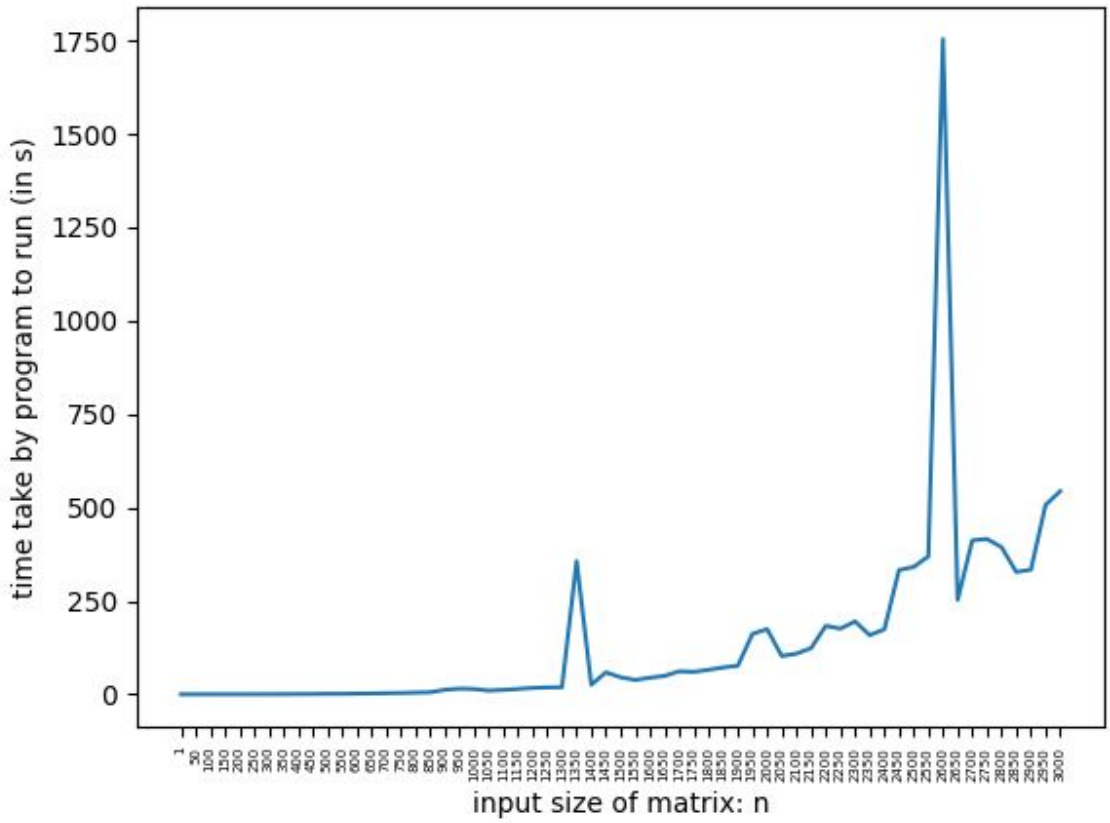
Output5:
S1 got to eat 41 times.
S2 got to eat 37 times.
S3 got to eat 36 times.
S4 got to eat 31 times.
S5 got to eat 31 times.

In totality:

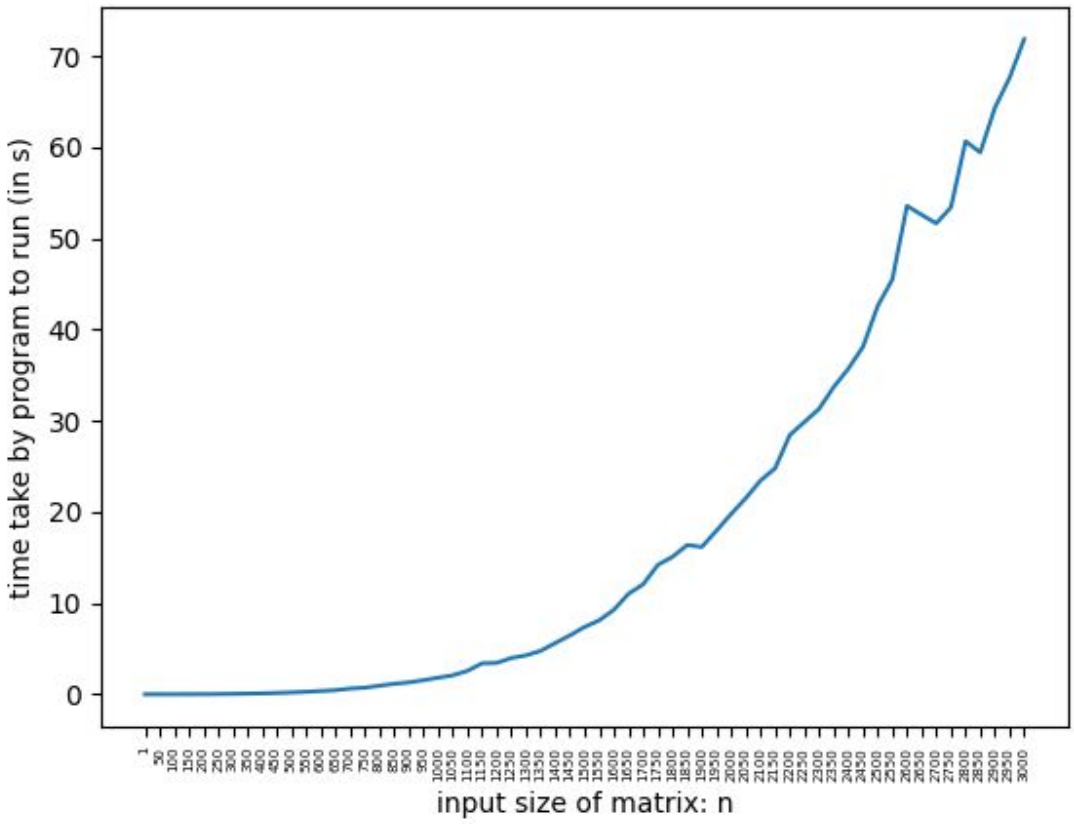
S1 got to eat 204 times.
S2 got to eat 182 times.
S3 got to eat 180 times.
S4 got to eat 159 times.
S5 got to eat 155 times.

Hence, fair allocation is being given to each student and thus he/she have somewhat equal chances of eating.

Graph for the sequential program:



Graph for program with threads:



We can see a huge difference in the runtime of the program for larger values of n (Very less for parallel programs than sequential) . Due to the multi core processor in the computer we experience better performance owing to parallelism.

In the parallel program, the matrix is computed for every 12 consecutive rows. The complexity hence reduces to $O(kn^2)$ where k denotes the number of rows per thread. In the sequential program, the complexity is $O(n^3)$ hence it takes longer to execute for large inputs of n.