ECE 4122/6122 Lab #3

(100 pts)

Section	Due Date
89313 - ECE 4122 - A	Oct 13th, 2020 by 11:59 PM
89314 - ECE 6122 - A	Oct 13th, 2020 by 11:59 PM
89340 - ECE 6122 – Q, QSZ, Q3	Oct 15th, 2020 by 11:59 PM

ECE 4122 & **ECE 6122** students do all problems.

Note:

You can write, debug and test your code locally on your personal computer. However, the code you submit must compile and run correctly on the PACE-ICE server.

Submitting the Assignment:

The solution to each problem should be contained in a **single zip file** and you must use C\C++ as the programming language. The zip file for each problem needs to contain all the files needed so that your solution can be compiled and run. Each zip file should be labeled Lab3_Problem**#**.zip, where **#** is problem number.

Speed matters in this assignment (see grading rubric)

Grading Rubric

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on his or her homework; however, TA's will **randomly** look through this set of "perfect-output" programs to look for other elements of meeting the lab requirements. The table below shows typical deductions that could occur.

AUTOMATIC GRADING POINT DEDUCTIONS PER PROBLEM:

Element	Percentage Deduction	Details
Files named incorrectly	10%	Per problem.
Execution Time	Up to 8%	0 pts deducted < 5x shortest time pts deducted = (4/5) (your time/shortest time) - 4 (rounded up to whole point value, with 8 pts maximum deduction)
Does Not Compile	30%	Code does not compile on PACE-ICE!
Does Not Match Output	10%-90%	The code compiles but does not produce correct outputs.
Clear Self-Documenting Coding Styles	10%-25%	This can include incorrect indentation, using unclear variable names, unclear/missing comments, or compiling with warnings. (See Appendix A)

LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score - (20/24)*H	H = number of hours (ceiling function) passed deadline note : Sat/Sun count as one day; therefore H = $0.5*H_{weekend}$

Problem 1: Su Doku (50 pts)

(www.projecteuler.net) Su Doku (Japanese meaning *number place*) is the name given to a popular puzzle concept. Its origin is unclear, but credit must be attributed to Leonhard Euler who invented a similar, and much more difficult, puzzle idea called Latin Squares. The objective of Su Doku puzzles, however, is to replace the blanks (or zeros) in a 9 by 9 grid in such that each row, column, and 3 by 3 box contains each of the digits 1 to 9. Below is an example of a typical starting puzzle grid and its solution grid.

0	0	3	0	2	0	6	0	0
9	0	0	3	0	5	0	0	1
0	0	1	8	0	6	4	0	0
0	0	8	1	0	2	9	0	0
7	0	0	0	0	0	0	0	8
0	0	6	7	0	8	2	0	0
0	0	2	6	0	9	5	0	0
8	0	0	2	0	3	0	0	9
0	0	5	0	1	0	3	0	0

4	8	3	9	2	1	6	5	7
9	6	7	3	4	5	8	2	1
2	5	1	8	7	6	4	9	3
5	4	8	1	3	2	9	7	6
7	2	9	5	6	4	1	3	8
1	3	6	7	9	8	2	4	5
3	7	2	6	8	9	5	1	4
8	1	4	2	5	3	7	6	9
6	9	5	4	1	7	3	8	2

Write a program that takes as a **command line argument** that is the path to an input file. There is sample input file called *input_sudoku.txt* included in the assignment.

Your program must include the following:

- Using std::thread to create a multi-threaded program to solve a set of sudoku puzzles.
- Create a class called **SudokuGrid** that is used to hold a single puzzle with a constant 9 x 9 array of **unsigned char** elements.
- Create the following member variables for **SudokuGrid**:
  - std::string m_strGridName;
  - unsigned char gridElement[9][9];
- Create the following member functions for **SudokuGrid**
  - friend fstream& operator>>(fstream& os, const **SudokuGrid** & gridIn);
    - reads a single SudokuGrid object from a fstream file.
  - friend fstream& operator<<(fstream& os, const **SudokuGrid** & gridIn);
    - writes the SudokuGrid object to a file in the same format that is used in reading in the object
  - solve();
- Your **main()** function needs to dynamically determine the maximum number of threads (numThreads) that can run concurrently. Your **main()** function should then spawn (numThreads-1) threads calling the function **solveSudokuPuzzles()**.

- Use global variables
    - std::mutex outFileMutex;
    - std::mutex inFileMutex;
    - std::fstream outFile;
        - In the main() function, open the output file **Lab3Prob1.txt** using the command line argument.
    - std::fstream inFile;
        - In the main() function open the file using the command line argument.
- In the function **solveSudokuPuzzles()** use a std::mutex(s) to protect the global variables ***inFile*** and ***outFile.***
    - The function needs to have a do-while loop to continue to read in and solve puzzles and then write out the solution until the end of the file is reached.
    - once the end of the file is reached the function should return.
- After all threads are finished close both the global fstream variables.

You are free to use online solutions for solving the Sudoku problems. Here is one such example: https://www.tutorialspoint.com/sudoku-solver-in-cplusplus. I have not tested this solution so it is up to you to find and/or implement code that works. Make sure you add a comment referencing any online solutions you use.

Problem 2: Another ant on the move problem (25 pts)

(https://brilliant.org/practice/monte-carlo/)

Brilli the ant is trying to get from point AA to point BB in a grid. The coordinates of point AA is (1,1) (this is top left corner), and the coordinates of point BB is (n,n) (this is bottom right corner, nn is the size of the grid).

Once Brilli starts moving, there are four options, it can go **left**, **right**, **up** or **down** (no diagonal movement allowed). Brilli chooses the direction to move randomly. If any of these four options satisfy the following:

- The new point should still be within the boundaries of the n × n grid
- The new point should not be visited previously.

If PP is the probability of Brilli the ant reaching point BB before he gets stuck.

PP = number succeeded/number runs

Your program needs to use OpenMP and the Monte Carlo technique to compute PP. The number of runs for the Monte Carlo simulation (-N) and the size of the grid (-g) need to be input using command line arguments like shown below

./a.out -N 100000 -g 6

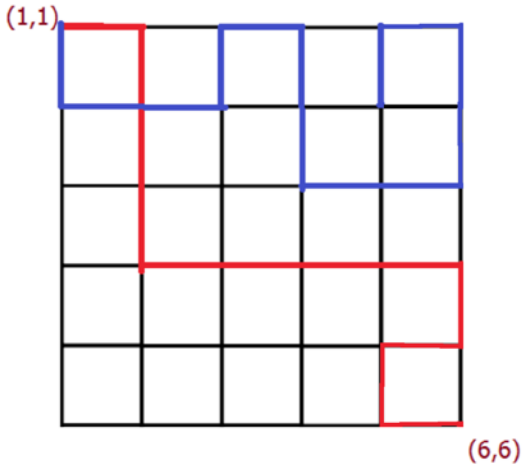
Your program should write out the probability to 7 decimal places (i.e. 0.1234567) to the text file **Lab3Prob2.txt**.

Use the std random number generator library

```
std::default_random_engine generator;
std::uniform_real_distribution<int> distribution(1,4);
```

where ***up => 1, down => 2 , left => 3, right => 4.***

Below is an example of two paths on a 6 x 6 grid.



Problem 3: Using Monte Carlo to solve optimization problem (25 points)

(https://www.acm-sigsim-mskr.org/Courseware/Balci/Slides/BalciSlides-03-MonteCarloSimulation.pdf)

Use OpenMP to create a multi-threaded program that uses the fundamental theory and logic of the Monte Carlo Simulation technique to solve the following optimization problem:

Find the values of X_1, X_2, and X_3 such that the following equation has the maximum value for Z.

$$Z = (e^{X_1} + X_2)^2 + 3(1 - X_3)^2$$

Subject to:

$0 \le X_1 \le 1$

$0 \le X_2 \le 2$

$2 \le X_3 \le 3$

Use

```
std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(start,end);
```

to generate the random values for each run. For each run you need to generate new random numbers for X_1, X_2, and X_3.

Your program needs to take in the number of runs as a command line argument and output the values of X_1, X_2, and X_3 in a text file called **Lab3Prob3.txt** to **15** decimal places. Like the following:

X1: 0.123456789012345
X2: 0.123456789012345
X3: 0.123456789012345

Make sure to use **doubles** in your program.

Appendix A: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:

Every file should have the following header at the top

```
/*
Author: <your name>
Class: ECE4122 or ECE6122
Last Date Modified: <date>

Description:

What is the purpose of this file?

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.

Appendix B: Accessing PACE-ICE Instructions

ACCESSING LINUX PACE-ICE CLUSTER (SERVER)

To access the PACE-ICE cluster you need certain software on your laptop or desktop system, as described below.

Windows Users:

Option 0 (Using SecureCRT)- THIS IS THE EASIEST OPTION!

The Georgia Tech Office of Information Technology (*OIT)* maintains a web page of software that can be downloaded and installed by students and faculty. That web page is:

http://software.oit.gatech.edu

From that page you will need to install SecureCRT.

To access this software, you will first have to log in with your Georgia Tech user name and password, then answer a series of questions regarding export controls.

Connecting using SecureCRT should be easy.

- Open SecureCRT, you'll be presented with the "Quick Connect" screen.
- Choose protocol "ssh2".
- Enter the name of the PACE machine you wish to connect to in the "HostName" box (i.e. *pace-ice.pace.gatech.edu*)
- Type your username in the "Username" box.
- Click "Connect".
- A new window will open, and you'll be prompted for your password.

Option 1 (Using Ubuntu for Windows 10):

Option 1 uses the Ubuntu on Windows program. This can only be downloaded if you are running Windows 10 or above. If using Windows 8 or below, use Options 2 or 3. It also requires the use of simple bash commands.

1. Install Ubuntu for Windows 10 by following the guide from the following link:

   https://msdn.microsoft.com/en-us/commandline/wsl/install-win10

2. Once Ubuntu for Windows 10 is installed, open it and type the following into the command line:

   *ssh **YourGTUsername**@pace-ice.pace.gatech.edu* where **YourGTUsername** is replaced with

   your alphanumeric GT login. Ex: bkim334

3. When it asks if you're sure you want to connect, type in:
   yes

   and type in your password when prompted (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:

vi filename.cc OR *nano vilename.cpp*

For a list of vim commands, use the following link:

https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

Option 2 (Using PuTTY):

Option 2 uses a program called PuTTY to ssh into the PACE-ICE cluster. It is easier to set up, but doesn't allow you to access any local files from the command line. It also requires the use of simple bash commands.

1. Download and install PuTTY from the following link: www.putty.org

2. Once installed, open PuTTY and for the Host Name, type in:
   pace-ice.pace.gatech.edu and for the port,

   leave it as 22.

3. Click Open and a window will pop up asking if you trust the host. Click Yes and it will then ask you for your username and password. (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in: *vim filename.cc* OR
   nano vilename.cpp

   For a list of vim commands, use the following link:

   https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

MacOS Users:.

Option 0 (Using the Terminal to SSH into PACE-ICE):

This option uses the built-in terminal in MacOS to ssh into PACE-ICE and use a command line text editor to edit your code.

1. Open Terminal from the Launchpad or Spotlight Search.

2. Once you're in the terminal, ssh into PACE-ICE by typing:

*ssh **YourGTUsername**@ pace-ice.pace.gatech.edu* where **YourGTUsername** is replaced with your alphanumeric GT login. Ex: bkim334

3. When it asks if you're sure you want to connect, type in:
   yes

   and type in your password when prompted (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:

   vi filename.cc OR *nano filename.cpp*

   For a list of vim commands, use the following link: https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

Linux Users:

If you're using Linux, follow Option 0 for MacOS users.