COMP-10205 – Data Structure and Algorithms
Lab # 6 – BattleShip Strategy
7% of course grade (BONUS AVAILABLE)

Submission Requirements
Complete the following exercise and submit electronically in the dropbox on eLearn. Please refer the course Calendar for the exact date and time of the submission. **This lab may be done in a team of up to three students or can be done individually. Students must be in the same CRN if done in a team.**

Background
You are to write a program that will play a limited game of battleship. The game of battleship is typically played with two players, each of which place 5 ships of various sizes on a 10 x 10 grid. Each player on a turn by turn basis attempts to place a shot where the opponent has placed a ship. Of course, your opponent can not see where you have placed your ships and you cannot see where the opponent has placed their ships. You typically call out shots to try and find and sink each of your ships. The player that can sink all of the opponent ships first is the winner. For a more complete description see Battleship.
In this instance of BattleShip the computer will randomly place 5 ships (lengths 2,3,3,4,5) on the board. A total of 17 spaces on the board out of 100 will have a ship. Once you obtain 17 hits you have solved the game. Your goal is to achieve the lowest average number of shots to do this. There are many different strategies to solve this problem.
The starting project provided includes the BattleShip API. The Battleship API is described below:

- **`public BattleShip()`** - you need to call the constructor once in your program to create an instance of the battleship game.
- **`public boolean shoot(Point shot)`** - you need to call this function to make each shot. See the sample source code for an example use.
- **`public int numberOfShipsSunk()`** - returns the total number of ships sunk at any point during the game. It is a good idea to use this method to determine when a ship has been sunk.
- **`public boolean allSunk()`** - returns a boolean value that indicates whether all the ships have been sunk.
- **`public int totalShotsTaken()`** - returns the total number of shots taken. Your code needs to be responsible for ensuring the same shot is not taken more than once.
- **`public int[] shipSizes()`** - returns an array of all of the ship sizes. The length of the array indicates how many ships are present. This array is fixed for the game. It does not update when a ship is sunk. You must try to write logic in your code to determine which ship has been sunk.
- **`public enum CellState`** - this enum object is very useful for marking cells has either Empty, Hit or Miss. It also has a convenience toString method so that can be used for printing purposes. You may also create your own Enum / Class for this in your code, but it is suggested that you use this instead of integers / characters to mark a Cell state

Steps

1. Download the starting code for the project from the eLearn dropbox. Rename the project to **Comp10152_Lab6**
2. Create a class for your battleship Logic - A starter solution is in the starting code for the lab.
3. Add some logic to your class to eliminate duplicate cell selection (i.e. - don't fire on the same cell twice)
4. Timing code is in the starting solution. You need to leave that code in your solution. It is best to play 10000 games to get enough games in order to calculate the average. Do not use fewer games.
5. Some Ideas for solving the problem:
    - Consider reducing the number of shots you need to fire to find a ship.
    - Create a map that tracks where you have placed your shots so that you don't fire on a cell more than once.
    - After you have hit a ship, you may want to change how you select your next shot.
    - The probability of a ship being on a particular square is not the same for each square. This probability changes after each shot as well. This technique for shot selection is a little more advanced but yields better ship targeting and when used correctly can greatly reduce the number of shots required.

Useful classes for this lab

All of the data structures that have been covered in the course could be used to solve the problem. Some classes are better suited than others so spend some time thinking about which classes to use.

Best Scores (based on a run of 10000 games)

- Student best score 2014 class = **51.61**
- Student best score 2015 class = **46.86**
- Student best score 2016 class = **47.70**
- Professor best score all time = **46.10**

Marking Scheme (A 100% grade is 20/20 - A maximum grade is 25/20)

Code quality, modularity, documentation, comments, naming conventions – 25%(5)
Appropriate Data Structures/Algorithms used – 30% (6)
Efficiency (Only given if shot selection/mapping used) – Bot must complete 10000 games in less than 2 s – 10% (2)
Shot Performance Score – 35% (7 marks)
Performance Score (based on 10 x 10 grid) - 10000 games

- >= 97.0 (0)
- < 97.0 (1)
- < 90.0 (2)
- < 80.0 (3)
- < 73.0 (4)

- < 68.0 (5)
- < 62.0 (7)
- **< 58 (9) (2 Bonus marks)**
- **< 53 (11)(5 Bonus Marks)**