Henry Lam
12/1/24
CPSC-39-12111

The app:

The game I designed is supposed to resemble the Battleship board game. The basic idea is that two players will have randomly placed ships on a 10x10 grid and must sink the enemy ships before their ships are destroyed. The twist I placed on the game is that rather than the players placing down their ships and being able to see a map, they will be completely blind and must go off of the system's output to be able to tell whether they hit a ship or not. In other words, there is a greater degree of chaos involved, and hopefully, this will make the game a bit more exciting.

The data structures:

The code mostly used two-dimensional arrays to store information because the Battleship board game uses a two-dimensional grid map. Therefore, it was very intuitive to simply try and replicate that in the code. The two-dimensional arrays were mostly used to store board information for each player, including whether a ship was placed, a ship was hit, or if the tile is water. The other type of data structure that was used was a hash map. The hash map was used to track hits and misses.

The algorithms:

The algorithms listed below have the purpose of initializing the board with water in each grid, placing down the ships on the board, and checking to see if either player's ships are all destroyed respectively. When I was designing the code, the algorithm that had the purpose of initializing the board was pretty easy to set up. I simply had to run a nested loop to initialize each index of a two-dimensional array. For the algorithm that placed ships down, I actually needed to figure out everything I needed to consider when placing down ships. These would include whether the ship placement was bending the ship, whether another ship was already placed, or whether the ship was being placed out of bounds. I wrote another set of code that solved this but is not included in this report. The code for placing down ships simply used if-else statements to place the ships based on whether they are vertical or horizontal. Lastly, the algorithm that checked whether the ships on either player's board were destroyed also used a nested loop to check each index of the two-dimensional arrays for an existing ship index.

Algorithm 1: initializeBoard

```
    // Method to initialize the game board
    private void initializeBoard() { // Algorithm: Initialize Board
        for (int i = 0; i < BOARD_SIZE; i++) { // Loop: Iterate over board rows
            for (int j = 0; j < BOARD_SIZE; j++) { // Loop: Iterate over board columns
                playerBoard[i][j] = '~'; // Water
                enemyBoard[i][j] = '~'; // Water
            } // End of Loop: Iterate over board columns
        } // End of Loop: Iterate over board rows
```

```
    } // End of Algorithm: Initialize Board
```

Algorithm 2: markShipOnBoard

```
    // Place the ship on the board
    private void markShipOnBoard(int row, int col, int size, boolean horizontal) { // Algorithm: Mark
Ship
        if (horizontal) { // Conditional: Check if horizontal placement
            for (int i = 0; i < size; i++) { // Loop: Place ship horizontally
                playerBoard[row][col + i] = 'S'; // S for Ship
            } // End of Loop: Place ship horizontally
        } else { // Conditional: Check if vertical placement
            for (int i = 0; i < size; i++) { // Loop: Place ship vertically
                playerBoard[row + i][col] = 'S'; // S for Ship
            } // End of Loop: Place ship vertically
        } // End of Conditional: Check if horizontal placement
    } // End of Algorithm: Mark Ship
```

Algorithm 3: allShipsDestroyed

```
    // Check if all ships are destroyed
    public boolean allShipsDestroyed(char[][] board) { // Algorithm: Check Ships Destroyed
        for (int i = 0; i < BOARD_SIZE; i++) { // Loop: Check board rows
            for (int j = 0; j < BOARD_SIZE; j++) { // Loop: Check board columns
                if (board[i][j] == 'S') return false; // Ship still exists
            } // End of Loop: Check board columns
        } // End of Loop: Check board rows
        return true; // All ships destroyed
    } // End of Algorithm: Check Ships Destroyed
```

Big O time:

Algorithm 1 uses a two-dimensional array for its nested loop, which means that the time complexity is $0(BOARD\_SIZE^2)$. Algorithm 2 simply runs a loop based on the size of the ship, which means that the time complexity is 0(n) where n is the size of the ship. Algorithm 3 uses a two-dimensional array for its nested loop, which means that its time complexity is similar to that of Algorithm 1. In the worst case scenario, the time complexity is $0(BOARD\_SIZE^2)$.

An opportunity:

Something that turned out to be an opportunity was when I was trying to figure out how to represent the information on the board. Initially, I had a 10x10 board created for each type of information I would be storing and for each player. In other words, I was using two-dimensional arrays of true/false for each kind of information I wanted to store. However, when I thought about it a bit more, I realized that I could simply just use a board for each player and instead use different characters or symbols to represent what each grid contained on the board. This simplified things a lot for me as I could simply reuse two-dimensional char arrays whenever I needed to access or change information.

An error:

One issue I ran into during development was figuring out how I could implement a second type of data structure. The Battleship game is a fairly simple game that could easily be represented using just two-dimensional arrays and arrays. However, I wanted something distinct from arrays in order to fulfill the requirement for the project. Fortunately, at that time, I had not yet figured out how I would return hit or miss for shots taken. In the process of figuring that out, I realized that I could use a hash map to track the shots taken. Hash maps are very simple data structures that allow ease of access for information, so it was the perfect data structure to use for representing shots.

Changes I would make:

In the next version of the game, I would like to consider the idea of implementing additional features like printing out the board after a game ends, involving more than two players in the game, allowing ship movement per turn, and creating a universal map for all players. I like the idea of a chaotic game because these are usually the ones that end up being really fun. I might also think about adding in outlandish features such as power-ups or repairs to spice things up. Regarding the actual code, I think using more efficient data structures might be helpful. I did think about using a linked list, but I never really invested into the idea due to time constraints.