HACETTEPE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

BBM103 ASSIGNMENT 4 REPORT

SALİH EREN YÜZBAŞIOĞLU-2220456040

4.01.2023

1 Introduction

In this section I talk about the assignment that was given to us and it’s properties.

We were given to make a game named Battleship with a description like this: Battleship is a strategy-based guessing game for two players. It is played on a grid where each player has a fleet of ships that are concealed from the other player. Players take turns calling "shots" at the other player's ships, with the objective being to destroy the opposing player's fleet. The game is played on four grids of 10x10 squares, with two grids for each player. Each player arranges their ships on their own grid, with the ships occupying consecutive squares either horizontally or vertically. The ships are not allowed to overlap. The types and numbers of ships allowed are the same for each player. The game is a discovery game, with players trying to discover the positions of their opponent's ships. At the start of the game, each player secretly arranges their ships on their hidden grid. The game proceeds in rounds, with each player announcing a target square on their opponent's grid and the computer announcing whether or not the square is occupied by a ship. If all the squares of a ship have been hit, the computer announces the sinking of that ship. If all of a player's ships have been sunk, the game is over and their opponent wins. If all ships of both players are sunk by the end of the round, the game is a draw.

For this homework, I was required to read in two files for player’s ship positions, Player1.txt and Player2.txt, which contain the initial ship positions for each player. I was then required to convert the contents of these files into a list data structure in your program. This list would represent the grid for each player, with the ships occupying specific positions on the grid according to the data in the input files.

Each 'C', 'B', 'D', 'S', 'P', or ';' character represents a square on the grid, with 'C' representing a carrier, 'B' representing a battleship, 'D' representing a destroyer, 'S' representing a submarine, 'P' representing a patrol boat, and ';' representing the line between squares. I was required to use this data to create a list representation of the grid for each player and use it to play the game of Battleship. I was also required to implement necessary printing operations to display the state of the game board to the user.

2 Design

My solution includes 3 parts input reading, perfectly locating of the ships and the part that applies every round iteratively. In my input reading part, I did not just read the input file, but I also modified it to be used in the other parts too. I also created global variables that were required in other parts so input reading was also used to make other parts easier. Here is the code for input reading:

```
def inputreading():
    import sys
    global player1list,player2list,player1moves,player2moves,output,wronginput
    errorstring ,output="IOError: input file(s) ",open("Battleship.out", "w")
    try:
      player1=open(sys.argv[1]).readlines()
    except IOError:
        errorstring +="Player1.txt, "
    try:
      player2=open(sys.argv[2]).readlines()
    except IOError:
        errorstring +="Player2.txt, "
    try:
      player1moves=open(sys.argv[3]).readline().strip(";").split(";")
    except IOError:
        errorstring +="Player1.in, "
    try:
      player2moves=open(sys.argv[4]).readline().strip(";").split(";")
    except IOError:
        errorstring +="Player2.in, "
    if len(errorstring)>23:
        print_and_output(errorstring[:-2]+" is/are not reachable.")
        raise IOError
    player1list,player2list,wronginput,shipcount=[],[],[0,0],0
    def Playerlist(player,playerlist):
        for row in player:
            values = row.split(";")  # Split the row by the semicolon character and store the resulting list in a new variable
            values = [value[0] if (value and value!="\n") else "-" for value in values]  # Replace any empty values with dashes
            playerlist.append(values)  # Add the list to the playerlist
    Playerlist(player1,player1list),Playerlist(player2,player2list)
    print_and_output("Battle of Ships Game\n\n")
```

This code is part of a function called inputreading() that reads in input for the Battleship game from four files: Player1.txt, Player2.txt, Player1.in, and Player2.in. The function starts by importing the sys module and defining several global variables, including player1list, player2list, player1moves, player2moves, output, and wronginput.

The function then attempts to open and read each of the four input files, catching any IOError exceptions that may be raised if a file is not found. If an IOError exception is caught, the function adds the name of the file that caused the error to the errorstring variable.

If the errorstring variable is longer than 23 characters, it means that at least one input file could not be found, so the function prints an error message to the screen and raises an IOError exception.

The function then initializes the player1list, player2list, and wronginput variables to empty lists, and the shipcount variable to 0. It defines a nested function called Playerlist() that takes in a player's input data and a list to store the data in. The Playerlist() function splits each row of the player's data by the semicolon character and replaces any empty values with dashes. It then appends the modified list of values to the playerlist provided as an argument.

The function then calls the Playerlist() function twice, once for each player, passing in the appropriate input data and list to store the data in. Finally, the function prints a message to the screen.

As you can see there is function named print_and_output which basically prints and writes to Battleship.out file given string.

```
def print_and_output(strng):
    global output
    print(strng, end="") output.write(strng)
```

Now the second part perfectly locating of the ships is done by a function named shipFinder that works by finding alone ship letters here is the code and detailed explanation:

```
def shipfinder(playerlist):
    for toFind in ["B14", "B24", "P42", "P32" "P12" "P22"]:
        for i in range(10):
            for j in range(10):
                m=0
                if playerlist[i][j]==toFind[0]:
                    for k,b in [(1,0),(0,1),(-1,0),(0,-1)]:
                        try:
                            if playerlist[i+k][j+b]==toFind[0] and (playerlist[i+k*2][j+b*2]==toFind[0] or toFind[0]=="P"):
                                m+=1
                                r=(k,b)
                        except:
                            pass
                    if m==1:
                        playerlist[i][j]=toFind[:2]
                        for t in range(1,int(toFind[-1])):
                            playerlist[i+t*r[0]][j+t*r[1]]=toFind[:2]
                        break
            if m==1:
                break
    global ships,letters
    ships,letters={},{}
    ships["B1_1"],ships["B2_1"],ships["B2_2"],ships["B1_2"],ships["P3_1"],ships["P4_1"],ships["P3_2"],ships["P4_2"],ships["P1_1"],ships["P2_1"],ships["P1_2"],ship
    letters["A"],letters["B"],letters["C"],letters["D"],letters["E"],letters["F"],letters["G"],letters["H"],letters["I"],letters["J"]=0,1,2,3,4,5,6,7,8,9
    shipcount=0
    for line in playerlist:
        for element in line:
            if element!="-":
                shipcount+=1
    if shipcount!=27:
        assert AssertionError
```

You can't see the dictionary assignments but they are basically assigning ships to amount of sqares they hold ex: ships[B2_2]=4 since B ships have 4 squares also "_2" says that it is player 2's ship. You might also realize it iterates over strings with 3 characters first character indicates ship second indicates index B has indexes 1 to 2 since there are 2 B ships and third indicates length of the ships. The shipfinder() function takes in a list representing the grid for one of the players in the game. The function iterates through the list, searching for a specific pattern of cells that corresponds to one of the ships in the game (Battleship, Patrol Boat, etc.). When it

finds a match for one of these patterns, it replaces the letters in the pattern with the corresponding ship type (e.g. "B1" for a Battleship).

The function also initializes a dictionary called ships, which maps the names of the ships to the number of squares they occupy. It also initializes a dictionary called letters, which maps the letters A through J to the corresponding column indices in the grid.

The function then counts the number of cells in the grid that are occupied by ships (i.e. not dashes), and checks if this number is equal to 27 (the total number of squares occupied by all of the ships in the game). If the number of occupied squares is not equal to 27, the function raises an AssertionError.

Lastly we have a function n that iterates every round here is the code:

```
def gameOutput(Turn):
    try:
        def player1or2(z):
            global wronginput
            if z==2:
                player="_1"
                global player2moves
                turnmoves=player2moves
                global player1list
                notturnlist=player1list
            else:
                player="_2"
                global player2list
                notturnlist=player2list

                global player1moves
                turnmoves=player1moves
            print_and_output(f"Player{z}'s Move\n\n")
            print_and_output(f"Round : {Turn}\t\t\t\t\tGrid Size: 10x10\n\n")
            print_and_output("Player1's Hidden Board\t\tPlayer2's Hidden Board\n")
            print_and_output("  A B C D E F G H I J\t\t  A B C D E F G H I J\n")
            for i in range(10):
                print_and_output(f"{i+1}")
                for j in range(10):
                    toprint=(player1list[i][j][0]=="O")*"O" or (player1list[i][j][0]=="X")*"X" or "-"
                    print_and_output(" "*(i!=9 or j!=0)+f"{toprint}")
                print_and_output("\t\t")
                print_and_output(f"{i+1}")
                for j in range(10):
```

The gameOutput() function is used to display the game state to the user at the beginning of each round. It takes in an integer parameter Turn, which indicates the current round number.

It starts by calling player1or2 function The player1or2() function is a nested function within the gameOutput() function. It is used to print round information and to set up the variables and data structures needed for either player 1 or player 2, depending on the value of the z argument passed to the function.

The player1or2() function starts by setting the value of the player variable to "_1" if z is equal to 2, and to "_2" if z is not equal to 2. player variable will be used to decrease the number in the ships dictionary if a ship has been hit.

Next, the function sets the value of the turnmoves variable to the global player2moves variable if z is equal to 2, and to the global player1moves variable if z is not equal to 2. Which will be used so we know the right player to make moves. Finally, the function sets the value of the notturnlist variable to the global player1list variable if z is equal to 2, and to the global player2list variable if z is not equal to 2. Which will be used to know list of the player who's not playing the turn.

```
        for j in range(10):
            toprint=(player2list[i][j][0]=="O")*"O" or (player2list[i][j][0]=="X")*"X" or "-"
            print_and_output(" "*(i!=9 or j!=0)+f"{toprint}")
        print_and_output("\n")
    print_and_output("\n")
    for j in [["C"],["B1","B2"],["D"],["S"],["P1","P2","P3","P4"]]:
        ship_1,ship_2="",""

        for k in j:
            ship_1=(ships[k+"_1"]==0)*("X "+ship_1) or ship_1+"- "
            ship_2=(ships[k+"_2"]==0)*("X "+ship_2) or ship_2+"- "

        ship="Carrier"*(k[0]=="C") or "Battleship"*(k[0]=="B") or "Destroyer"*(k[0]=="D") or "Submarine"*(k[0]=="S") or "Patrol Boat"*(k[0]=="P")
        tabNumber="\t"*2*(k[0]=="C") or "\t"
        tabNumber2="\t"*3*(k[0]=="P") or "\t\t\t\t"
        print_and_output(f"{ship}{tabNumber}{ship_1[:-1]}{tabNumber2}{ship}{tabNumber}{ship_2[:-1]}\n")
    print_and_output("\n")

    while True:
        move1 = turnmoves[Turn - 1+wronginput[z-1]]
        print_and_output(f"Enter your move: {move1}\n\n")


        move=move1.split(",")
        if len(move)==1:
            print_and_output(f"IndexError: move should have ',' in it.\n\n")
            wronginput[z-1]+=1
            continue

        if len(move)!=2:
```

Here is the unseen part: notturnlist[int(move[0])-1][letters[move[1]]]=(notturnlist[int(move[0])-1][letters[move[1]]]=="-")*"O" or "X"+notturnlist[int(move[0])-1][letters[move[1]]]

This part starting with the upper image's print function first prints the name of the current player's move and the round number. Then it prints the hidden boards of both players, which are represented as 10x10 grids with letters on the top and numbers on the side to label each cell.

For each cell in the grid, the code looks at the value stored in the corresponding cell in the player1list or player2list list. If the value is "O", it prints an "O" to represent a hit. If the value is "X", it prints an "X" to represent a miss. Otherwise, it prints a "-".

After printing the grids, the code then prints the status of each player's ships. It goes through a list of ship names (Carrier, Battleship, etc.) and for each ship, it looks at the corresponding value in the ships dictionary. If the value is 0, it prints an "X" to represent that the

ship has been sunk. Otherwise, it prints a "-". The code also adds some tab characters to make the output look properly aligned.

```
        if len(move)!=2:
            print_and_output(f"ValueError: {move1} should have single ',' in it.\n\n")
            wronginput[z - 1] += 1
            continue
        if move[0]=="" or move[1]=="":
            print_and_output(f"IndexError: {move1} should have a number between 1 and 10 and a letter in it.\n\n")
            wronginput[z - 1] += 1
            continue
        try:
            if int(move[0]):
                pass
        except:
            print_and_output(f"IndexError: {move1} should start with a number between 1 and 10.\n\n")
            wronginput[z - 1] += 1
            continue
        if (int(move[0]) < 1 or int(move[0]) > 10):
            print_and_output("AssertionError: Invalid Operation.\n\n")
            wronginput[z - 1] += 1
            continue
        if move[1] not in letters:
            print_and_output("AssertionError: Invalid Operation.\n\n")
            wronginput[z - 1] += 1
            continue
        if notturnlist[int(move[0])-1][letters[move[1]]][0]=="X" or notturnlist[int(move[0])-1][letters[move[1]]][0]=="O":
            print_and_output("AssertionError: Invalid Operation.\n\n")
            wronginput[z - 1] += 1
            continue

        break
```

As you can see there is a long while True part in the code which basically makes sure the given move is correct if not gives the proper error message and looks at the next move until the move is correct it uses a list named wronginput the store amount of times a player has given wrong input 0^{th} index for player 1 1th index for player 2 so that when finding the move from the turnmoves list it does’not look at the same move twice.

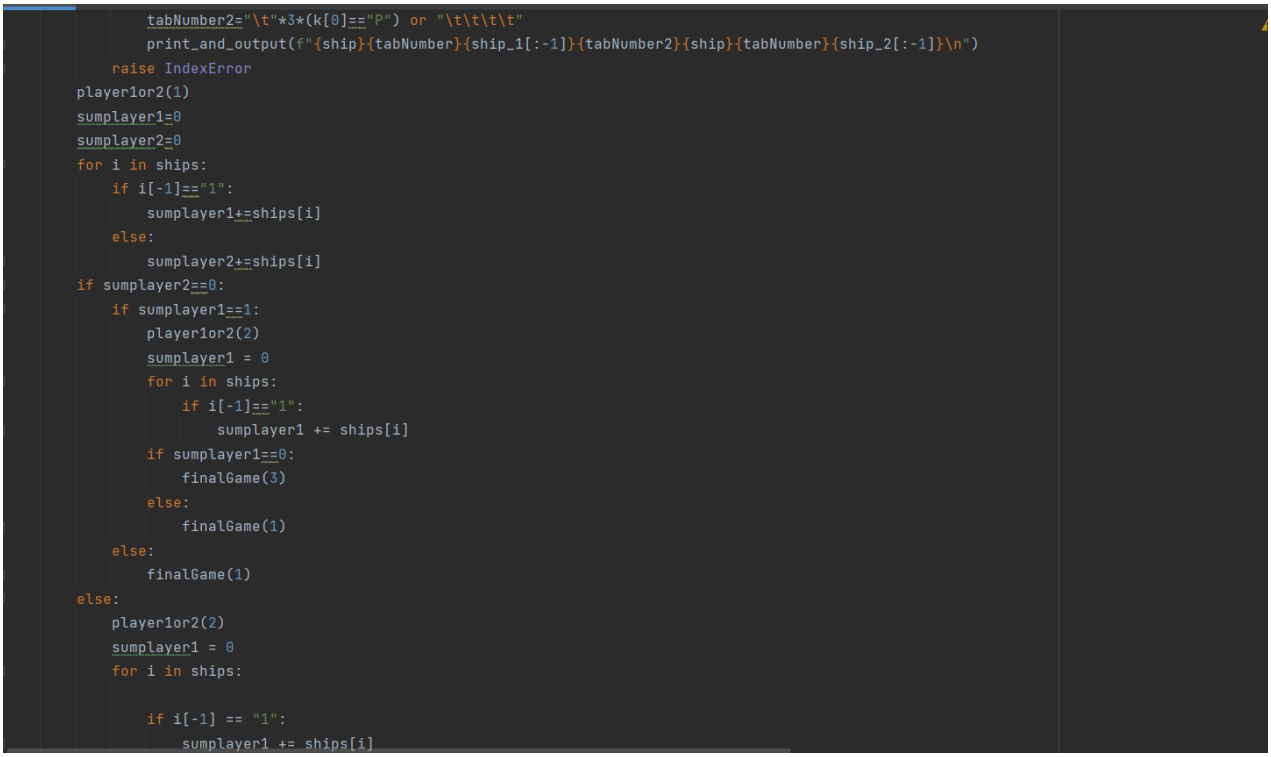
```
    if notturnlist[int(move[0])-1][letters[move[1]]]!="-":
        ships[notturnlist[int(move[0]) - 1][letters[move[1]]]+player]-=1


    notturnlist[int(move[0])-1][letters[move[1]]]=(notturnlist[int(move[0])-1][letters[move[1]]]=="-")*"O" or "X"+notturnlist[int(move[0])-1][letters[m
def finalGame(z):
    print_and_output((z==3)*"It is a Draw!\n\n" or f"Player{z} Wins!\n\n")
    print_and_output("Final Information\n\n")
    print_and_output("Player1's Board\t\t\t\tPlayer2's Board\n")
    print_and_output("  A B C D E F G H I J\t\t  A B C D E F G H I J\n")
    for i in range(10):
        print_and_output(f"{i+1}")
        for j in range(10):
            toprint=player1list[i][j][0]
            print_and_output(" "*(i!=9 or j!=0)+f"{toprint}")
        print_and_output("\t\t")
        print_and_output(f"{i+1}")
        for j in range(10):
            toprint=player2list[i][j][0]
            print_and_output(" "*(i!=9 or j!=0)+f"{toprint}")
        print_and_output("\n")
    print_and_output("\n")
    for j in [["C",["B1","B2"],["D"],["S"],["P1","P1","P3","P4"]]:
        ship_1,ship_2="",""
        for k in j:
            ship_1=(ships[k+"_1"]==0)*("X "+ship_1) or ship_1+"- "
            ship_2=(ships[k+"_2"]==0)*("X "+ship_2) or ship_2+"- "
        ship="Carrier"*(k[0]=="C") or "Battleship"*(k[0]=="B") or "Destroyer"*(k[0]=="D") or "Submarine"*(k[0]=="S") or "Patrol Boat"*(k[0]=="P")
        tabNumber="\t"*2*(k[0]=="C") or "\t"
        tabNumber2="\t"*3*(k[0]=="P") or "\t\t\t\t"
```

The two-line code at the top is used to modifie ships dictionary here is how it works:

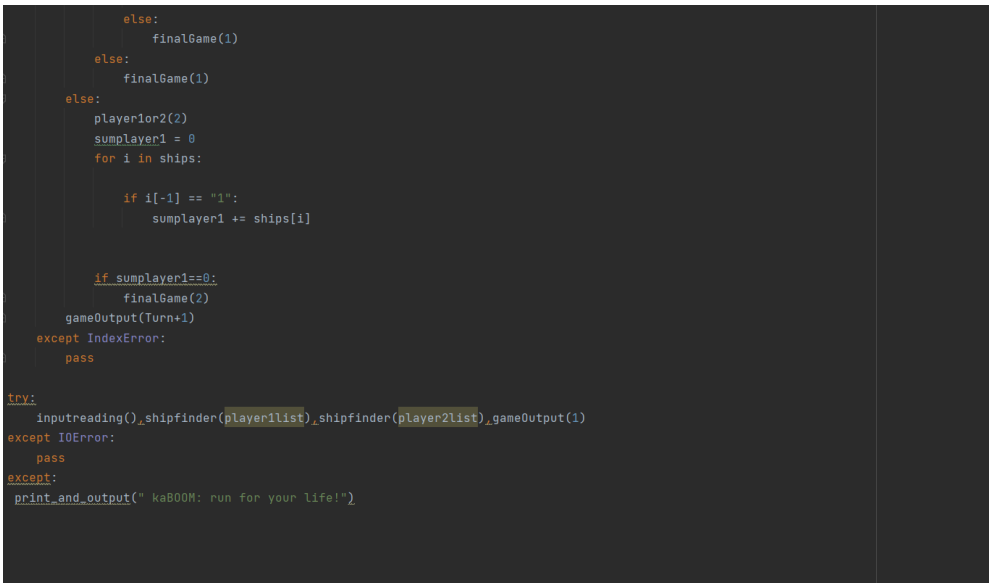
Code is checking if a player's move is a hit or a miss. notturnlist is a list that represents the hidden board of the player who is not currently making a move. int(move[0])-1 is the index in the notturnlist of the row corresponding to the row number specified in the player's move (the move is a string in the form "ROWCOLUMN", so move[0] is the row and move[1] is the column). letters[move[1]] is the index in the row of the column specified in the player's move.

The code is checking if the value at the specified row and column in the notturnlist is not "-". If it is not, then this means that the player's move is a hit. In this case, the code decreases the value in the ships dictionary corresponding to the ship that was hit. The key for the ship in the dictionary is constructed by concatenating the value at the specified cell in the notturnlist with the string in the player variable (either "_1" or "_2"). The value of player is either "1" or "2" and indicates which player's ship was hit. The code decreases the value in the dictionary by 1 to keep track of how many times the ship has been hit.



```
            tabNumber2="\t"*3*(k[0]=="P") or "\t\t\t\t"
            print_and_output(f"{ship}{tabNumber}{ship_1[:-1]}{tabNumber2}{ship}{tabNumber}{ship_2[:-1]}\n")
        raise IndexError
    player1or2(1)
    sumplayer1=0
    sumplayer2=0
    for i in ships:
        if i[-1]=="1":
            sumplayer1+=ships[i]
        else:
            sumplayer2+=ships[i]
    if sumplayer2==0:
        if sumplayer1==1:
            player1or2(2)
            sumplayer1 = 0
            for i in ships:
                if i[-1]=="1":
                    sumplayer1 += ships[i]
            if sumplayer1==0:
                finalGame(3)
            else:
                finalGame(1)
        else:
            finalGame(1)
    else:
        player1or2(2)
        sumplayer1 = 0
        for i in ships:

            if i[-1] == "1":
                sumplayer1 += ships[i]
```



```
            else:
                finalGame(1)
        else:
            finalGame(1)
    else:
        player1or2(2)
        sumplayer1 = 0
        for i in ships:

            if i[-1] == "1":
                sumplayer1 += ships[i]


        if sumplayer1==0:
            finalGame(2)
    gameOutput(Turn+1)
except IndexError:
    pass

try:
    inputreading(),shipfinder(player1list),shipfinder(player2list),gameOutput(1)
except IOError:
    pass
except:
 print_and_output(" kaBOOM: run for your life!")
```

The part that starts with player1or2(1) and ends with expect IndexError is the part of the code that controls functions in the finalGame function it also controls if the game is draw or a win here is how it works: The code first calls player1or2(1) so player 1 plays his/her move then initializes the variables sumplayer1 and sumplayer2 to 0. It then iterates through the keys in the

ships dictionary and adds the values corresponding to player 1's ships to sumplayer1 and the values corresponding to player 2's ships to sumplayer2.

Next, the code checks if sumplayer2 is 0. If it is, this means that all of player 2's ships have been sunk and player 1 has won. In this case, if player 2 has one square of ship left it calls player1or2(2) so he/she plays his/her last round if he successfully hits the last square this means it is a draw and calls finalGame with argument equal to 3 meaning it is a draw if player 2 can't hit the last square the code calls the finalGame function with 1 as an argument to declare player 1 as the winner.

If sumplayer2 is not 0, the code calls the player1or2 function to allow player 2 to make a move. It then checks if sumplayer1 is 0. If it is, this means that all of player 1's ships have been sunk and player 2 has won. In this case, the code calls the finalGame function with 2 as an argument to declare player 2 as the winner.

If neither sumplayer1 nor sumplayer2 is 0, this means that the game is not over and the code will repeat this process until the game is over.

Finalgame function is used for the last round here is deliberate explanation:

The finalGame function is called when the game is over to print the final state of the game to the screen and declare a winner or a draw.

The function takes an integer as an argument, which is either 1, 2, or 3. If the argument is 1, this means that player 1 has won. If the argument is 2, this means that player 2 has won. If the argument is 3, this means that the game is a draw.

The function begins by printing the result of the game. It does this by using a ternary operator to check if the argument is 3. If it is, it prints "It is a Draw!\n\n". Otherwise, it prints "Player{z} Wins!\n\n", where {z} is the number of the winning player.

Next, the function prints "Final Information\n\n" and then the revealed boards of both players. The boards are represented as 10x10 grids with letters on the top and numbers on the side to label each cell. The function iterates through the rows and columns of each board and prints the value stored in the corresponding cell in the player1list or player2list list.

After printing the boards, the function then prints the status of each player's ships. It goes through a list of ship names (Carrier, Battleship, etc.) and for each ship, it looks at the corresponding value in the ships dictionary. If the value is 0, it prints an "X" to represent that the ship has been sunk. Otherwise, it prints a "-". The function also adds some tab characters to make the output look properly aligned.

Finally, the function raises an IndexError exception. This is done to end the game and prevent the code from continuing to execute.

3 Programmer's Catalogue

I spent a lot of time on shipFinder since I needed to find every ship, so it took me around an hour to find that solution and another hour to implement it. The other functions were not easy either, but they were straightforward, so they took a lot of time but was not so hard another hard part was debugging since this is 200+ lines of code so to understand an error I spent a lot of time probably around 3-4 in total spent examining and solving errors. The most used function in my code was the first one (printing and writing the given string) I will probablty use it in other codes too. Also, shipfinder code was brilliant too the same idea I used can be used in lot of finding functions with minimal amount of modification.

4 User Catalogue

I spent a huge amount of time trying to make user my code won't go kaboom and will explain the error in any sort of input to the user and keep going so it is quite user friendly. The biggest restriction on the program is it takes all input at the start so there is no way to modifie it after the game starts. It could be better if the moves were given with input function.

Evaluation	Points	Evaluate Yourself / Guess Grading
Using Explanatory Comments	5	… 5
Efficiency (avoiding unnecessary actions)	5	…5
Function Usage	15	… 15
Correctness, File I/O	30	… 30
Exceptions	20	20
Report	20	… 20
There are several negative evaluations	…	…