Department of Computer Engineering

BBM103 Assignment 4 Report

Can Ertaş 2220356088

03/01/2023

Contents

- Introduction
- Analysis
- Design
- Programmer’s Catalogue
- User Catalogue

Introduction

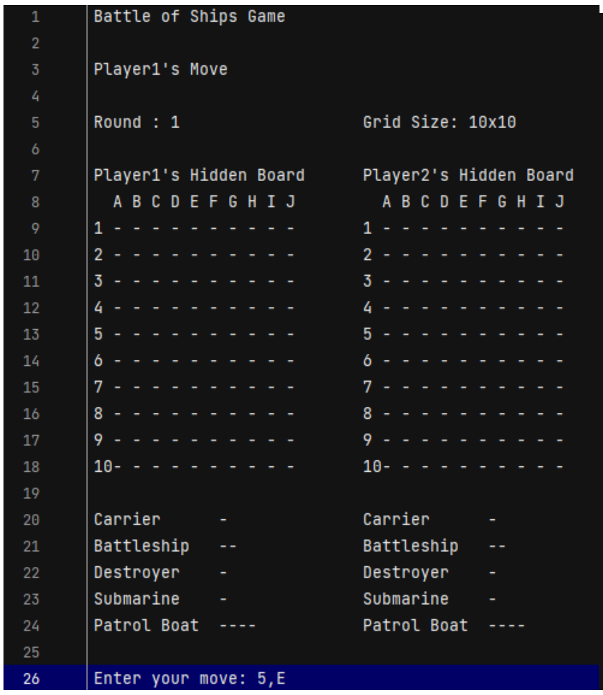
In this report, I will explain the problem and analyze it firstly, then explain my thoughts on the solutions for the problems and my solutions in detail as much as possible. Later I will evaluate myself on the code i produced. Finally, the user will be given information on how to use the program.

Analysis

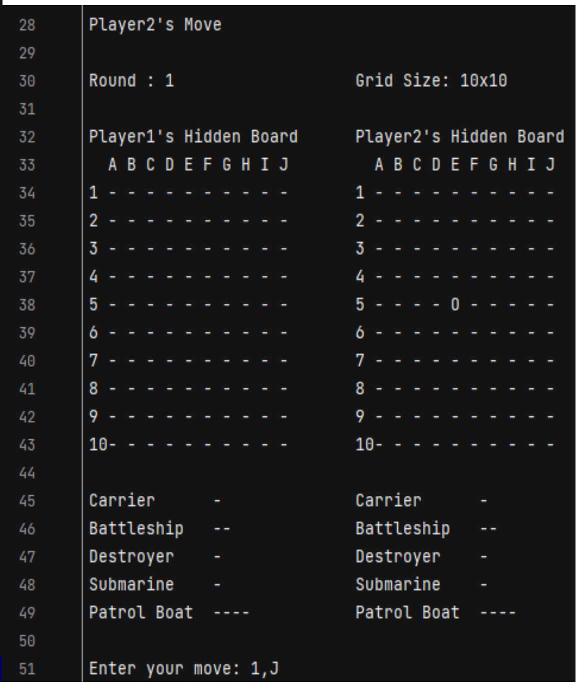
In this chapter the problems of this assignment will be analyzed.

To begin with, the main goal of this assignment is to make a very well known game called “Battleship”. The game is played with 2 players. Each player has 1 carrier with 5 units length, 1 destroyer with 3 units length, 1 submarine with 3 units length, 2 battleships with each having 4 units length, 4 patrol boats with each having 2 units length. Both players place their warships on their 10x10 boards. The inputs for this placing part are taken from 2 “.txt” files (one for each player). Move inputs will be taken from 2 “.in” files (one for each player). Now, the game is ready to be started.

We are asked to print the hidden boards every time a player makes a move in each round. After one player’s move is played, when it is time for the other player to make his/her move, on the hidden boards the previous move will be played: if it is a miss “-” (meaning never been hitted), will convert to “O”, else “-” will turn into “X”. Underneath each board the status of each player’s fleet is shown, for every sunk ship a “X” appears instead of “-” near the related ship category.



```
Battle of Ships Game

Player1's Move

Round : 1                    Grid Size: 10x10

Player1's Hidden Board       Player2's Hidden Board
  A B C D E F G H I J          A B C D E F G H I J
1 - - - - - - - - - -        1 - - - - - - - - - -
2 - - - - - - - - - -        2 - - - - - - - - - -
3 - - - - - - - - - -        3 - - - - - - - - - -
4 - - - - - - - - - -        4 - - - - - - - - - -
5 - - - - - - - - - -        5 - - - - - - - - - -
6 - - - - - - - - - -        6 - - - - - - - - - -
7 - - - - - - - - - -        7 - - - - - - - - - -
8 - - - - - - - - - -        8 - - - - - - - - - -
9 - - - - - - - - - -        9 - - - - - - - - - -
10- - - - - - - - - -        10- - - - - - - - - -

Carrier      -               Carrier      -
Battleship   --              Battleship   --
Destroyer    -               Destroyer    -
Submarine    -               Submarine    -
Patrol Boat  ----            Patrol Boat  ----

Enter your move: 5,E
```



```
Player2's Move

Round : 1                    Grid Size: 10x10

Player1's Hidden Board       Player2's Hidden Board
  A B C D E F G H I J          A B C D E F G H I J
1 - - - - - - - - - -        1 - - - - - - - - - -
2 - - - - - - - - - -        2 - - - - - - - - - -
3 - - - - - - - - - -        3 - - - - - - - - - -
4 - - - - - - - - - -        4 - - - - - - - - - -
5 - - - - - - - - - -        5 - - - - O - - - - -
6 - - - - - - - - - -        6 - - - - - - - - - -
7 - - - - - - - - - -        7 - - - - - - - - - -
8 - - - - - - - - - -        8 - - - - - - - - - -
9 - - - - - - - - - -        9 - - - - - - - - - -
10- - - - - - - - - -        10- - - - - - - - - -

Carrier      -               Carrier      -
Battleship   --              Battleship   --
Destroyer    -               Destroyer    -
Submarine    -               Submarine    -
Patrol Boat  ----            Patrol Boat  ----

Enter your move: 1,J
```

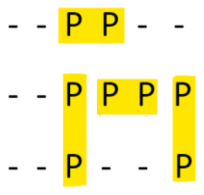
The most challenging problems of the assignment so far are:

1. How the data of points should be kept (if A1 for example is hit, has a ship on it etc.),
2. How to conclude which "P" belongs to which ship, and which "B" belongs to which ship from a given input:

One might place the patrol boats as follows:

```
- - P P - -
- - P P P P
- - P - - P
```

It is not hard to group these boats, after a couple of seconds one might will find the following:



But it may not be this easy to teach a computer to find these groupings.

The same thing applies for battleships surely.

Others are much easier to group of course because each is uniqiue in their category, for example there is only one carrier, only one destroyer, and only one submarine in the game for each player.

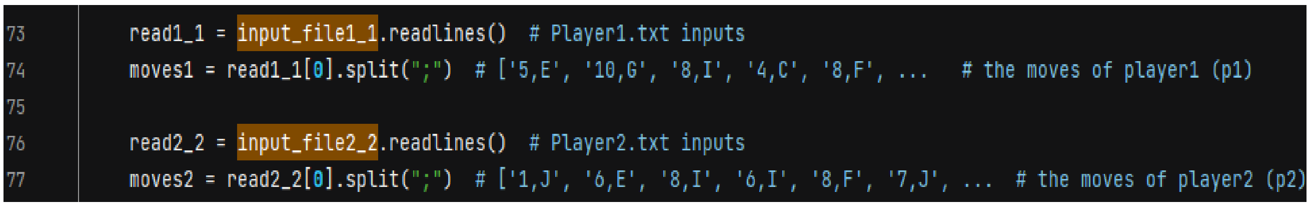
Design

In this section I will talk about the solutions for the problems mentioned above.

Problem 1

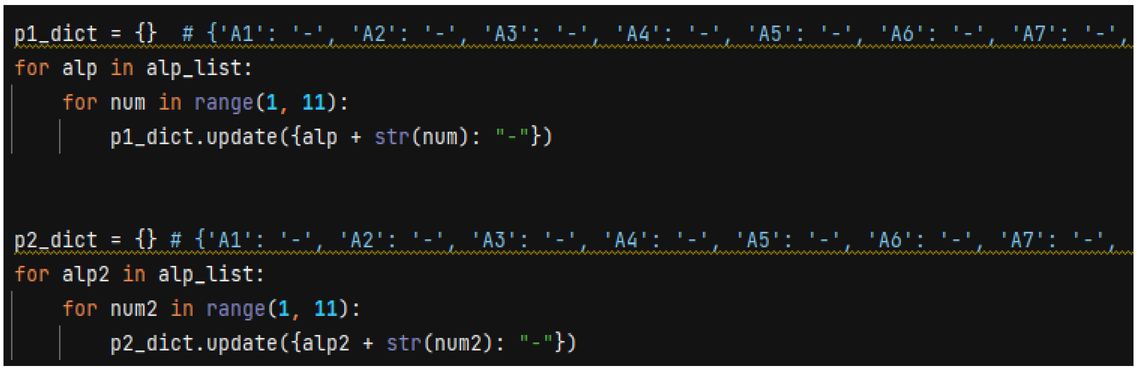
For the first problem, I thought of using dictionary and I did so. Lists would be unnecessarily complicated for me.

Firstly, after the 2 txt input files are read, moves are sent into two lists:



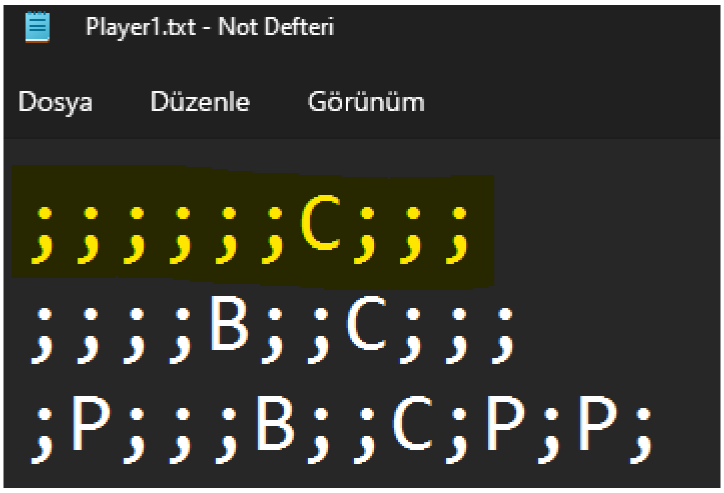
```
read1_1 = input_file1_1.readlines()  # Player1.txt inputs
moves1 = read1_1[0].split(";")  # ['5,E', '10,G', '8,I', '4,C', '8,F', ...   # the moves of player1 (p1)

read2_2 = input_file2_2.readlines()  # Player2.txt inputs
moves2 = read2_2[0].split(";")  # ['1,J', '6,E', '8,I', '6,I', '8,F', '7,J', ...  # the moves of player2 (p2)
```

Then, two dictionaries (where the positions of ships will be stored) are created for each player:



```
p1_dict = {}  # {'A1': '-', 'A2': '-', 'A3': '-', 'A4': '-', 'A5': '-', 'A6': '-', 'A7': '-',
for alp in alp_list:
    for num in range(1, 11):
        p1_dict.update({alp + str(num): "-"})


p2_dict = {} # {'A1': '-', 'A2': '-', 'A3': '-', 'A4': '-', 'A5': '-', 'A6': '-', 'A7': '-',
for alp2 in alp_list:
    for num2 in range(1, 11):
        p2_dict.update({alp2 + str(num2): "-"})
```

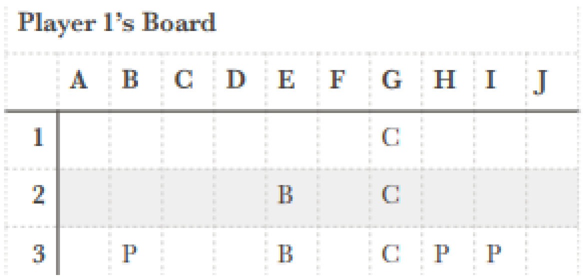
Now, I will use the “.in” files by using the fact that in each line, the number of semicolons before the number of interest indicate how many spaces there must be between the row number and the related ship part. For example,



```
;;;;;;C;;;
;;;;B;;C;;;
;P;;;B;;C;P;P;
```

In the first line of the “Player1.txt” there are 6 semicolons before “C” (Carrier part), so there must be 6 spaces between the row number sqaure at the very left and square in which “C” lies; meaning the column of C must be 7.

This is what is shown in the pdf:

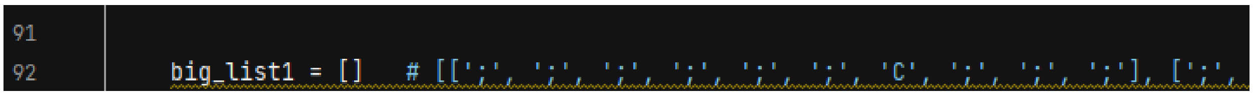


Player 1’s Board

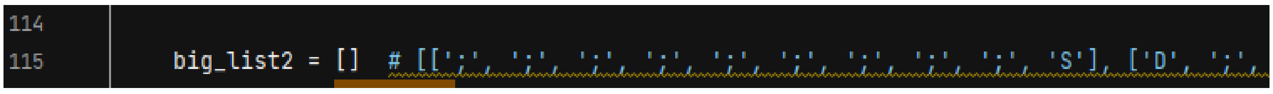
	A	B	C	D	E	F	G	H	I	J
1							C			
2					B		C			
3		P			B		C	P	P	

There are exactly 6 spaces before “C” at the first row of Player1’s board as expected.

The info from these txt are stored in two multidimensional lists which are called big_list1 and big_list2 (one for each player):

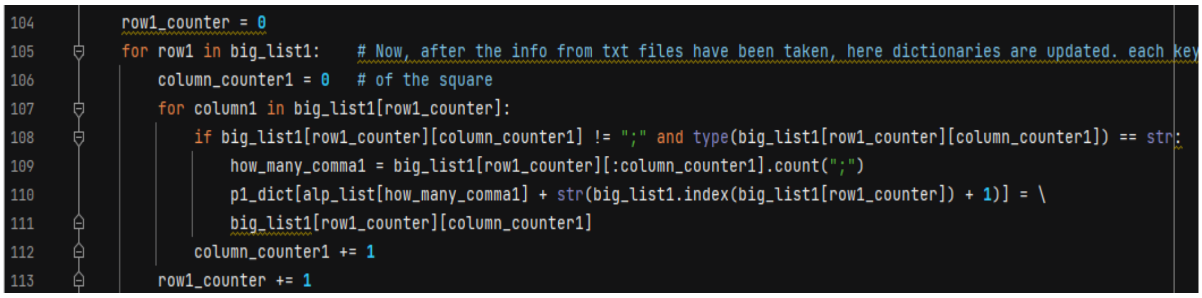


```
big_list1 = []   # [[';', ';', ';', ';', ';', ';', 'C', ';', ';', ';'], [';',
```



```
big_list2 = []  # [[';', ';', ';', ';', ';', ';', ';', ';', ';', 'S'], ['D', ';',
```

The information from these lists are used as follows:



```
104     row1_counter = 0
105     for row1 in big_list1:    # Now, after the info from txt files have been taken, here dictionaries are updated. each key
106         column_counter1 = 0   # of the square
107         for column1 in big_list1[row1_counter]:
108             if big_list1[row1_counter][column_counter1] != ";" and type(big_list1[row1_counter][column_counter1]) == str:
109                 how_many_comma1 = big_list1[row1_counter][:column_counter1].count(";")
110                 p1_dict[alp_list[how_many_comma1] + str(big_list1.index(big_list1[row1_counter]) + 1)] = \
111                 big_list1[row1_counter][column_counter1]
112             column_counter1 += 1
113         row1_counter += 1
```

The program keeps searching elements the lists inside the multidimensional list unless it finds an element that is not “;”, when it finds such an element, it firstly counts how many semicolons it passed to find this element. Then necessary data changing process begins; at lines 110-111 “p1dict[alp_..]” is equal to the coordinate of the square that is to be changed, right side (line 111) is what the value of these squares (A1, B8) should be and these values are “C”, “D”, “B”, “S”, “P”.

The same thing is applied for the player2’s dictionary as well with a copy paste.

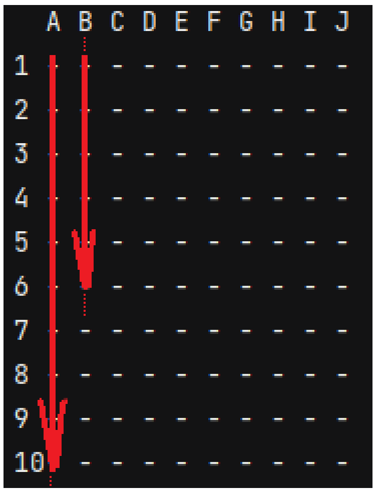
Now, my dictionaries(p1_dict, p2_dict) have the coordinates of the ships.

Problem 2

The second one was the most challenging and time-taking one for me. I didn’t want to use optional inputs and wanted my code to be ready for every type of input and solve it if it has only one solution

To handle this, I defined a function called “ship_finder”. What this function does is basically execute the following algorithm:

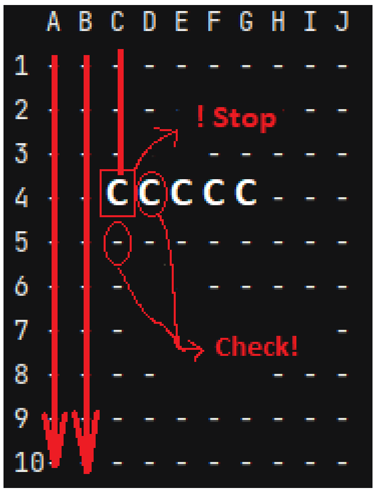
Step 1: Start searching all 100 squares stop until finding a square that doesn’t consist “-” as value in the dictionary of each player (p1_dict, p2_dict). The searching direction will be from A1 to A10 Then from B1 to B10 and so on. The visualisation is as follows:



```
   A B C D E F G H I J
1  - - - - - - - - - -
2  - - - - - - - - - -
3  - - - - - - - - - -
4  - - - - - - - - - -
5  - - - - - - - - - -
6  - - - - - - - - - -
7  - - - - - - - - - -
8  - - - - - - - - - -
9  - - - - - - - - - -
10 - - - - - - - - - -
```

Step 2: if the square that doesn't consist of "-" has either "C", "D", "S" as element move to next, if it is "B" move to step 4 , else move to step 5

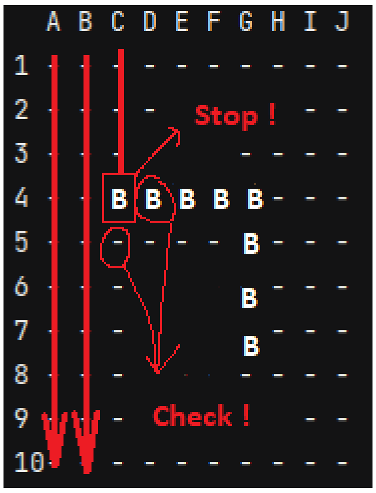
Step 3: With the fact that the squares just above this square and left square don't have the same element that stopped the program (say "C" for example), check the square just below, and the neighbour square that is on the right side.,



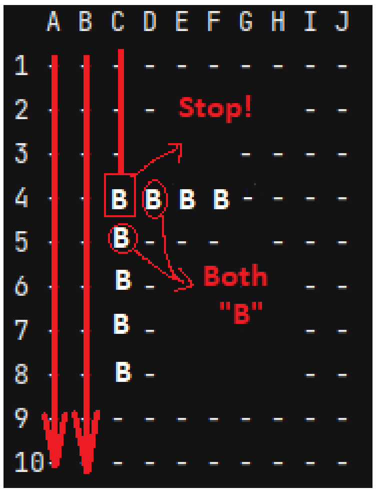
```
   A B C D E F G H I J
1  - - - - - - - - - -
2  - - - - - - - - - -
3  - - - - - - - - - -
4  - - C C C C C - - -
5  - - - - - - - - - -
6  - - - - - - - - - -
7  - - - - - - - - - -
8  - - - - - - - - - -
9  - - - - - - - - - -
10 - - - - - - - - - -
```

If the square on the right side is not "C" conclude the ship is downwards (C4,C4,C6,C7,C8) are the coordinates for this example), else conclude the ship is rightwards (which is the case in this example). Apply this to the other 2 types of ships then store these coordinates of ships.

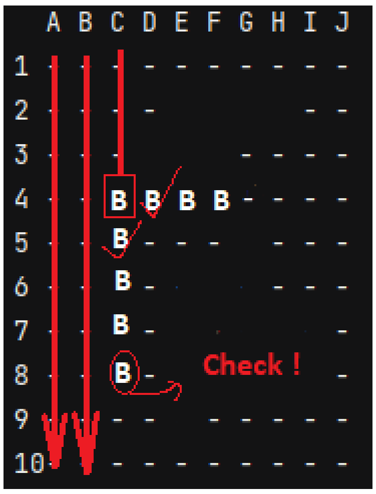
Step 4: If this square is stored pass, else check the neighbouring right and down squares, if the right one is “B” and down one is not, then conclude the ship of this square is rightwards (like this example C4, D4,E4,F4 will be store). If the down one is “B” and the right one is not, then conclude the ship of this square is downwards(C4,C5,C6,C7 would be stored in this example). If a ship is found, store the coordinates.



If both squares have “B” :



Then check the square that is 4 squares below the square that stopped the code:



If it is “B” conclude that the square which stopped the code belongs to a ship that is rightwards, else conclude that the square which stopped the code belongs to a ship that is downwards.

Step 5: (This step differs from others: when this is step is reached it groups all of the “P”s at once with a while loop): If this square is stored then pass, else

Check all the squares around the square that is *under investigation,* keep searching squares until you find a “P” square which is neighbouring only one “P” square

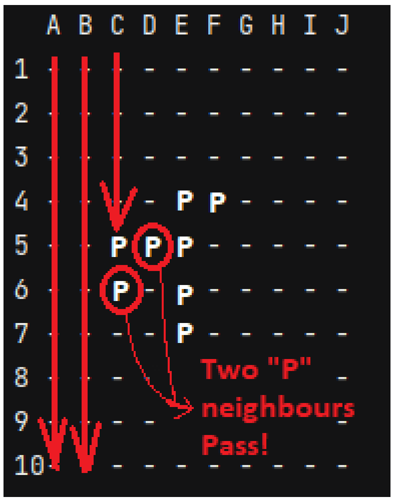
if such a square is found store it and the neighbouring “P” square and move to the next square

else investigate the next square

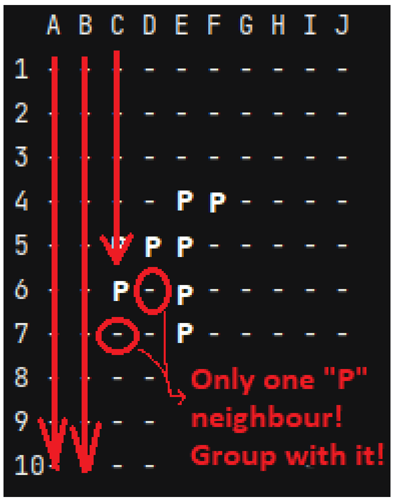
When you reach the last square (J1) go back to the square which caused the first “P” square interaction.

Repeat Step 5 until the number of non-store "P” squares equals 0.

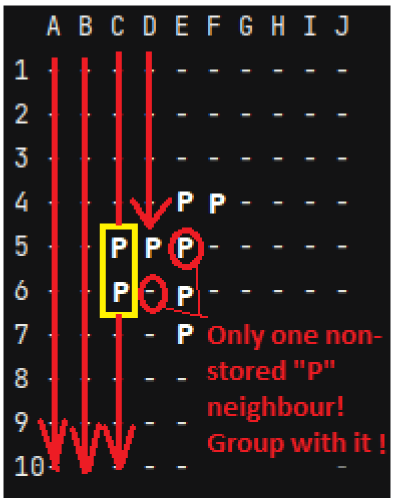
(One hard example is visualized below.



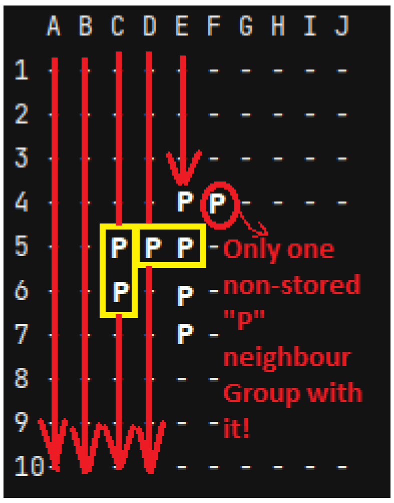
Picture 1



Picture 2

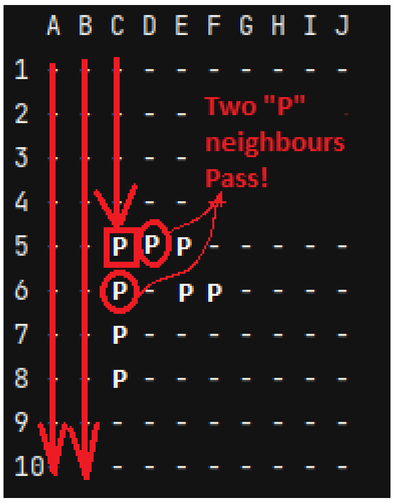


Picture 3

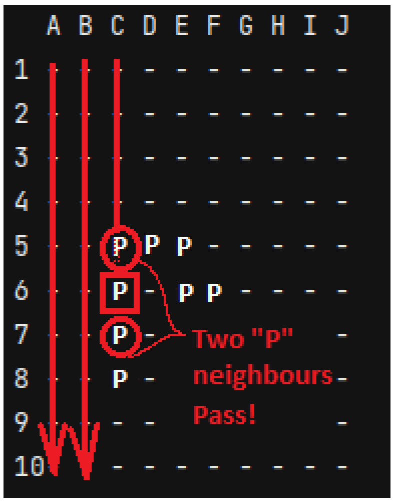


Picture 4

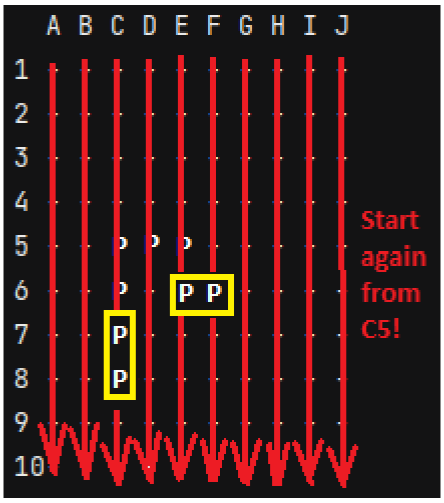
And so on. This searching process can fall into a loop if it is not completed at first searching process, one such example is this:



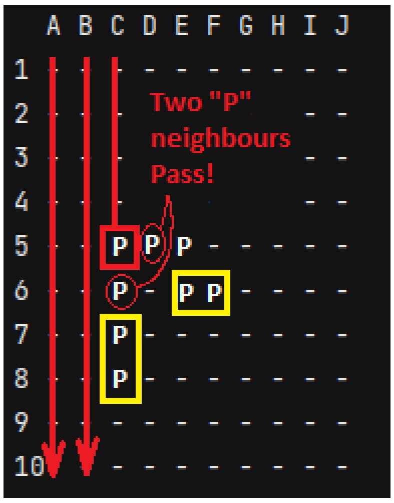
Picture 1



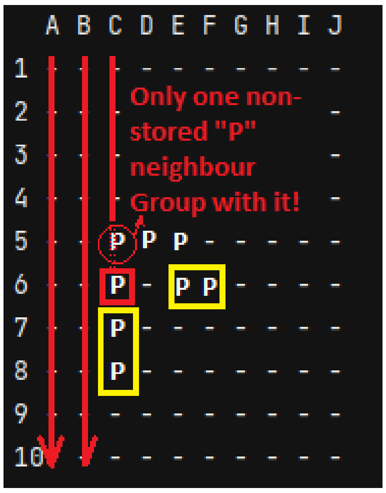
Picture 2



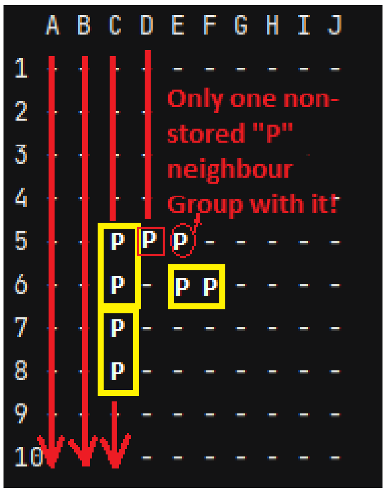
Picture 3



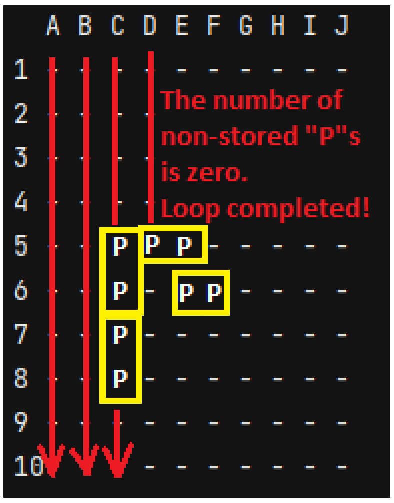
Picture 4



Picture 5



Picture 6



Picture 7)

Step 6: You are done!

Programmer's Catalogue

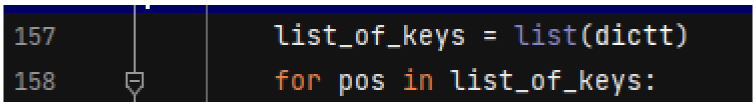
In total this code took about 40-50 hours of my time in total. I defined 4 functions in total. Here are brief explanations for them:

1- ship_finder(dictt)

Half of the time I spent was for defining this function.

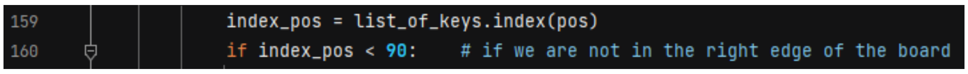
This is the function for locating ships. We need to print an "X" for every sunk ship. This funciton is essential for this purpose.

Inside it keys of the parameter dictionary is listed:



```
    list_of_keys = list(dictt)
    for pos in list_of_keys:
```

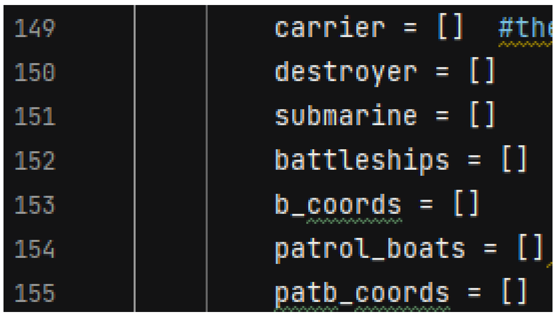
Then a for loop is started for searching every square.



```
        index_pos = list_of_keys.index(pos)
        if index_pos < 90:    # if we are not in the right edge of the board
```

This if statement is created because the function always tries to check the right neighbouring square. It could have been handled in more clever way actually. I think I used too many conditional statements which made the code so long.

These empty lists below are created to store the coordinates of ships.



```
        carrier = []  #the
        destroyer = []
        submarine = []
        battleships = []
        b_coords = []
        patrol_boats = []
        patb_coords = []
```

The lists *carrier*, *destroyer*, *submarine* are filled with coordinates of the related ships. For every empty list above a “X” will appear next to the related ship category.

battleships is filled with 2 lists (one for each battleship), *patrol_boats* is filled with 4 lists (one for each patrol boat). For every empty list in the multimensional list *battleship*, an “X” will appear next to battleship category. The same logic applies for patrol boats also.

2- *show_board2(p1_dict, p2_dict)*

This function is for showing two boards next to each other

With the experience from Assignment 3, it wasn’t hard to print a board. The most challenging part of this printing 2 boards next to each other. However, this also didn’t take much time about 1-1.5 hour was spent for function. This is called in the fourth function.

3- *show_ship_c(p1_ships, p2_ships)*

The parameters are dictionaries and each contains keys as “carrier”, “battleships”, “destroyer”, “submarine”, “patrol boats”, and the values are the relevant lists that are shown in the previous image in the detail of second function. This function prints the sunk boat information which is under the hidden boards. This is called in the fourth function.

4- *move_maker(n)*

The paramater is the round this function will play. This function takes the moves, check the coordinates to see if there is a ship on the fired location, call *show_board2(p1_dict, p2_dict)* and *show_ship_c(p1_ships, p2_ships).*

User Catalogue

It is assumed in the code that the user in the in files put as many more correct moves as he/she put incorrect moves.