

# Congkak

*Data Structures & Algorithms*

CSC 2103

Group Information			
Name	Student ID	Program	Contribution %
Lishan Abbas	15034614	BCS	30%
Yap Jia Yung	15046824	BCS	30%
Hans Maulloo	16050692	BCS	20%
Daniel Jedidiah Antonio	15068893	BCS	20%

# 1. Introduction

Mancala also known as "congak" is a 2-player traditional game originated from the Tamil Nadus. The game is now played in Malaysia, Indonesia, Maldives, Brunei, Singapore and many other countries as well. In Malaysia, it is commonly known as "Congkak" and is played according to the local rules despite of many other variations in playing the game across the world. The term "Congkak" means mental calculation, which is the main practice in this game where the player who can think a few move in advance including the opponent move is probably the winner of the game. Traditionally, the game consist of a board made up of wood, crafted with mainly 14 holes or 18 holes in two sets of 7 and 9 respectively, and an additional two bigger holes as each player home hole to accumulate their score. Each hole is usually filled with 7 seeds using beans, pebbles, stones, cowrie shells tamarind seeds and marble excluding the home hole. So for this assignment, we as a team were assigned to re-create the traditional Congkak game in a modern way using our chosen programming language to replace the wood crafted board and seeds. On top of that, this assignment has its own variation to play the game according to the specified instruction which we will need to fulfill.

## **Problem Statement**

To be able to create a congkak game using programming language according to the rules of game.

Instruction for playing the game:

1. A player can only select hole from his side and not empty holes
2. The selected hole will be emptied and the seeds obtain from the selected hole will be distribute to each neighbouring holes except for home hole one at a time according to the amount of seeds retrieve starting from the hole next to selected hole.
3. If the distribution stops on a hole with an empty hole next to it, the current player will lose his turn and score are obtained by withdrawing the seeds next to the empty hole and store in current player home hole
4. If the distribution stops on a hole without an empty hole next to it, the seeds from the next hole will be withdrawn and emptied, and distribution of seeds will continue from the hole next to the emptied hole one at a time according to the amount of seeds withdraw. The distribution stops only if the last distributed seeds lands on a hole with an empty hole next to it and scores are obtained. The current player lost his turn
5. The game ends when either player has no valid move to play from their side.
6. If a player has no valid move to play, leftover seeds are distributed to opponent's home
7. Player with the higher home hole score is the winner.

**Assignment scope:**

Usage of proper data structure, complete algorithm that capture the objective of the game.

**Game features**

- Basic game with a fixed board, each player moves, scoring, and able to identify the winner
- An intermediate game with different board and seeds size, each player moves, scoring, identify the winner
- An advance level with the support of player with computer, scoring, advance algorithm, winner, game with interface, and etc.
- Any other feature or variations that are appropriate to the game.

**Limitations**

- Player's name cannot be more than 25 characters long
- Size of board and seed count have their respective limitations to prevent the misalignment of board display in the CLI. In our case, the maximum seeds in each hole are capable of holding not more than 999 seeds. Calculation: Board Size x Seed Count  $\leq$  999.
- Board size display can only hold value of up to 3 digits on each side. Any more would misalign the display.
- Too large number of seeds or board size will require a long time to finish off a move. Consequently, the game will take a long time to end. Test case: Board size of 100 and seed count of 1000 required about 2 hours to finish the game using random generator for random move and time interval set to 0.

## Features for assignment

### Command Line Interface (CLI)

The entire game is carried out through CLI, for faster remote access.

### Main menu browsing

Players can browse through the main menu to different part of the game such as play, about, high scores and exit the game.

### Multiple Game Types (Basic, Intermediate, Advance)

The user can choose different game types at Play, range from basic, intermediate and advance.

Basic level:

- Board size and seed count are fixed. (Board size: 7, seed count: 4)
- Each player is able to make their desired but valid move.
- Scores are obtained for each move depending on position.
- Winner are announced at the end of the game.

Intermediate level:

- Board size and seed count varies depending on player's preference.
- Each player is able to make their desired but valid move.
- Scores are obtained for each move depending on position.
- Winner are announced at the end of the game.

Advance level:

- Board size and seed count varies depending on player's preference.
- Game is display via CLI interface.
- Each player is able to make their desired but valid move.
- Player decision on each move are aided by computer's suggested move.
- Scores are obtained for each move depending on position.
- Winner are announced at the end of the game.

Extras:

- Coin flipping function to decide which player to start the game.
- Highscore file manipulation (View, view sorted, reset, storing inside the object high score)
- Time interval during distribution of each seeds to observe the distribution of each seed to their respective holes.
- Flexible board display support for up to 3 digits' value in a hole and different board size

## 2. Requirement Analysis

### Data Structure and Algorithms:

#### Quick sort implementation in high score system.

Quick sort algorithm is used to sort the score of each player in descending order whereby player with the highest score is display at the top follow by player with lower score. The following code fragments show the implementation of quicksort:

```
// private static void quicksort(ArrayList<Score> arrayList, int from, int to) {
    if (from < to) {
        int pivot = from;
        int left = from + 1;
        int right = to;
        int pivotValue = arrayList.get(pivot).getValue();
        while (left <= right) {
            // left <= to -> limit protection
            while (left <= to && pivotValue >= arrayList.get(left).getValue()) {
                left++;
            }
            // right > from -> limit protection
            while (right > from && pivotValue < arrayList.get(right).getValue()) {
                right--;
            }
            if (left < right) {
                Collections.swap(arrayList, left, right);
            }
        }
        Collections.swap(arrayList, pivot, j: left - 1);
        quicksort(arrayList, from, to: right - 1); // pivot
        quicksort(arrayList, from: right + 1, to); // pivot
    }
}
```

#### Greedy algorithm approach for bot to get best move

Greedy algorithm is achieved by making use of heuristic approach to make the locally optimal choice at every move and be consider as globally optimal choice. In our case, the bot only considers the 1<sup>st</sup> move which grants the player most score without any backtracking. For example, if a move play grants the highest score in first iteration, the move is considered as the best(suggested) move by the bot. The following code fragments show the implementation of greedy algorithm:

```
public static int getBestMove(Hole[] holes, Options options, PlayerBot player) {
    // Create a copy for holes, avoid altering the real game
    holesObject = holesDeepCopy(holes);
    Hole[] holesCopy = holesDeepCopy(holes);

    // Get all the playable moves for the player in an int array
    int[] moves = getPlayableMovesArray(holes, options, player);

    // Initialize: max() function
    int highestScore = Integer.MIN_VALUE;
    int playedMoveScore;
    int bestMove = 1;

    // For every move inside moves array
    for (int i = 0; i < moves.length; i++) {
        holesObject = holesDeepCopy(holes);

        // Create a new BOT player inside the function: playMove, with the passed in player's score
        playedMoveScore = playMove(holesObject, options, new PlayerBot(player.getScore()), moves[i]);

        // Show Moves
        // System.out.format("Move %d: %d\n", getBestMoveForSuggestion(moves[i]), playedMoveScore);

        if (playedMoveScore > highestScore) {
            highestScore = playedMoveScore;
            bestMove = moves[i];
        }
    }

    return bestMove;
}
```

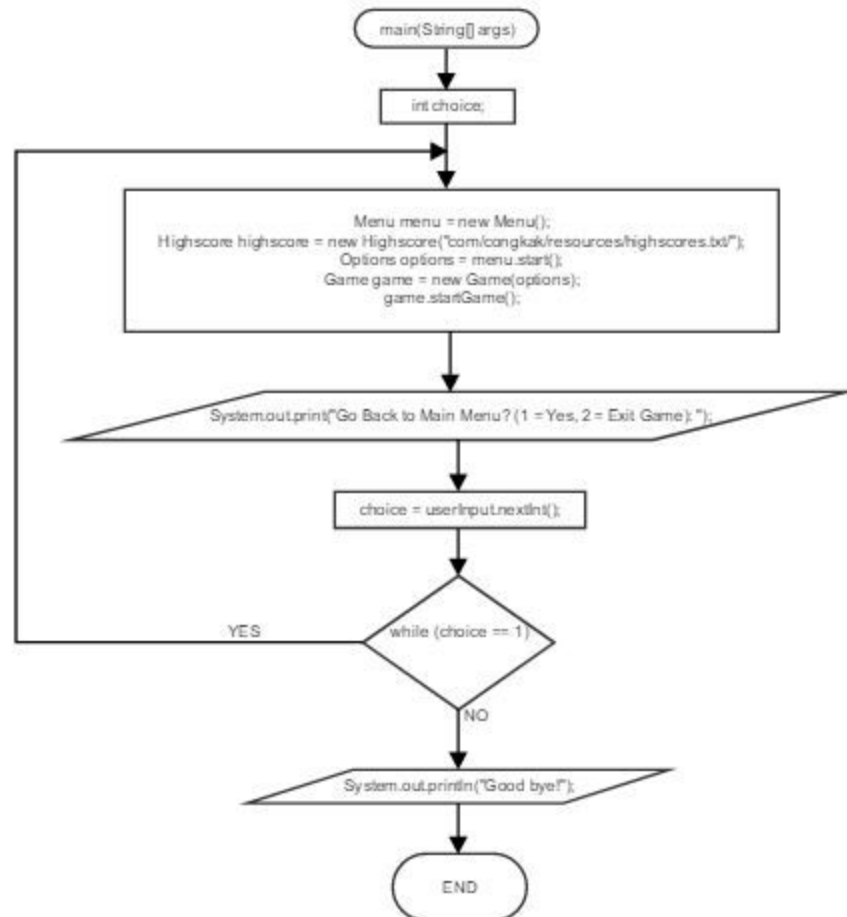
## Deep Copy to get the current state of game for the bot implementation

Deep Copy is used to get the current game state to avoid alteration of array value while making calculation to get the best move. The array `holesOriginal` and `holesDeepCopy` refer to different areas of memory, when `holesDeepCopy` is assigned to `holesOriginal`, the values in the memory area which `holesOriginal` points to are copied into the memory area which `holesDeepCopy` points to but later modify to the values of either `holesOriginal` or `holesDeepCopy` are unique to each other thus avoid alteration of original holes while the bot is testing the next best move because the contents are not shared. The following code fragments show the implementation of deep copy:

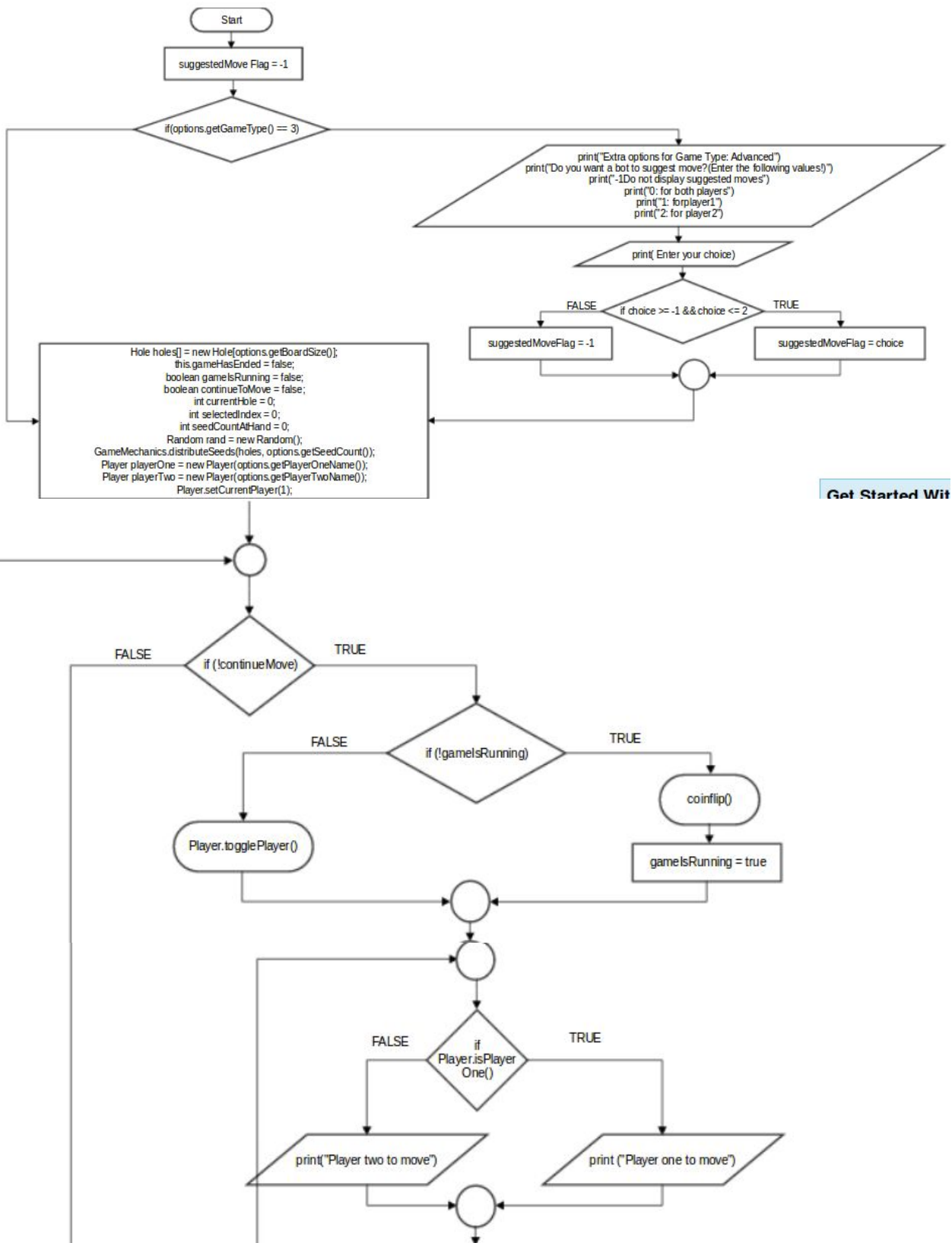
```
private static Hole[] holesDeepCopy(Hole[] holesOriginal) {  
    Hole[] holesCopy = null;  
    try {  
        ByteArrayOutputStream bos = new ByteArrayOutputStream();  
        ObjectOutputStream out = new ObjectOutputStream(bos);  
        out.writeObject(holesOriginal);  
        out.flush();  
        out.close();  
  
        // Create an input stream from the byte array and read a copy of the object back in.  
        ObjectInputStream in = new ObjectInputStream(  
            new ByteArrayInputStream(bos.toByteArray()));  
  
        return (Hole[]) in.readObject();  
    } catch (Exception e) {  
        System.out.println("Deep Copy unsuccessful. ");  
        System.err.println("Exception: " + e.getMessage());  
        return holesOriginal;  
    }  
}
```

# Flowchart

## Main

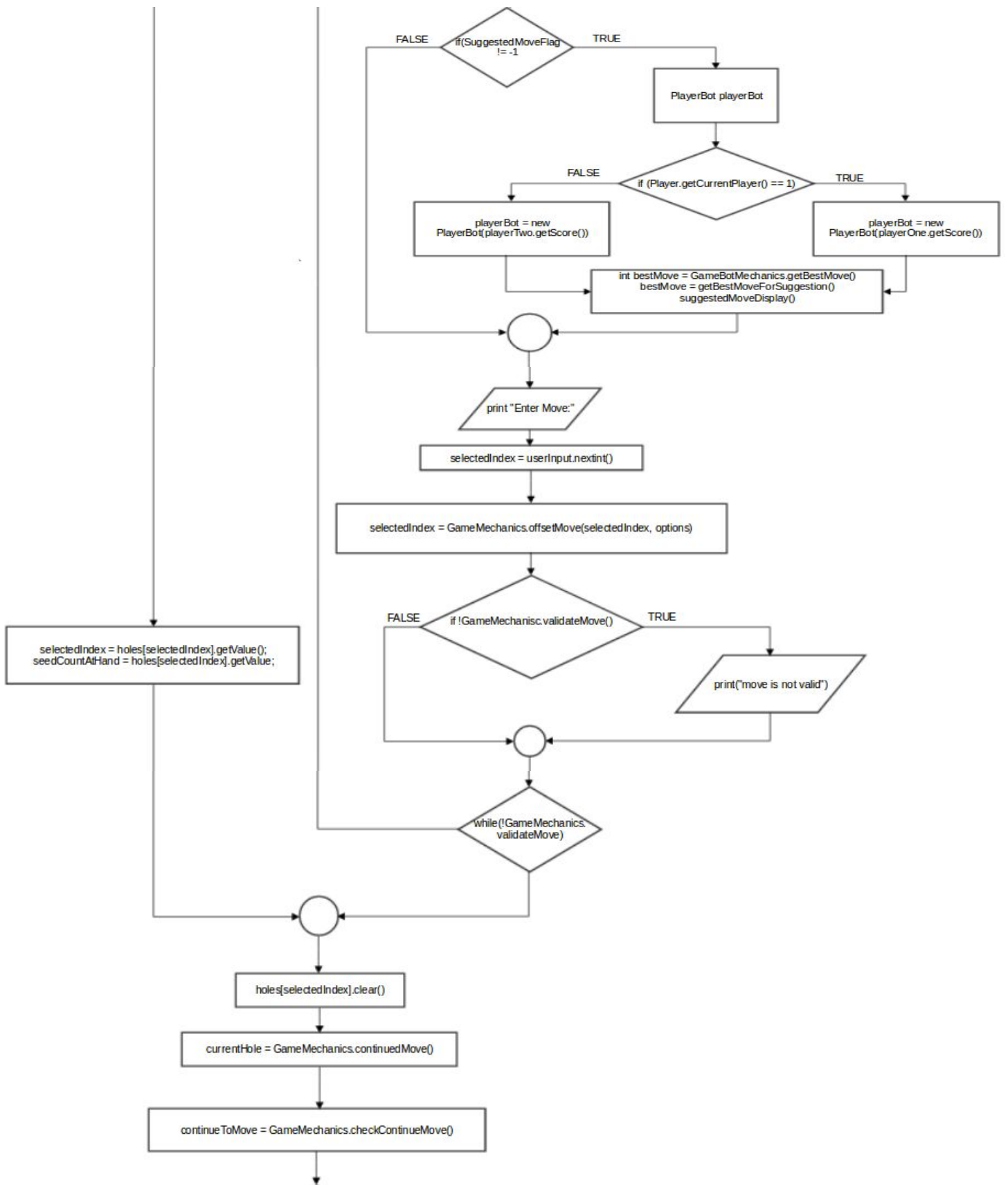


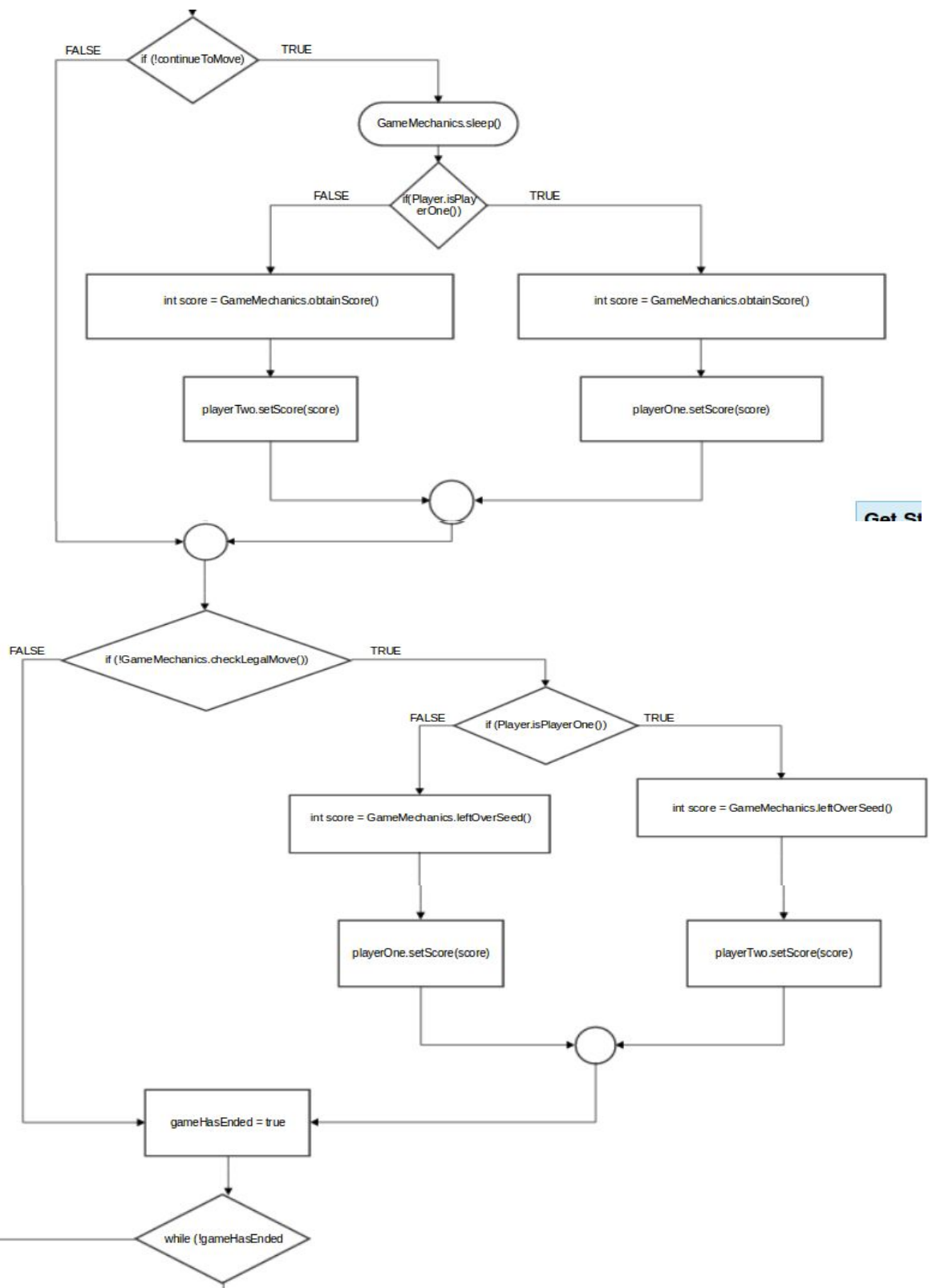
## Game.startGame()

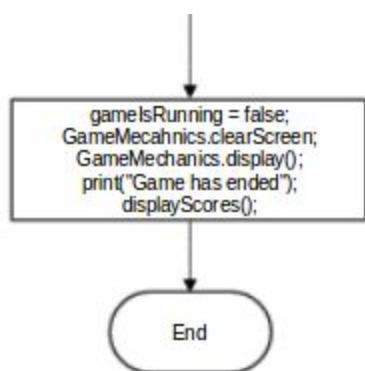


Get Started Wit









**Time complexity:**

Time complexities of the methods in GameMechanics.java					
Methods/BigO	O(1)	O(2)	O(N)	O(2N)	O(7N)
distributeSeeds			•		
validateMove	•				
continuedMove	•				
seedDistribution			•		
offsetMoves	•				
checkEmptyHoles	•				
checkContinueMove		•			
obtainScore	•				
checkLegalMove				•	
leftOverSeed			•		
display					•

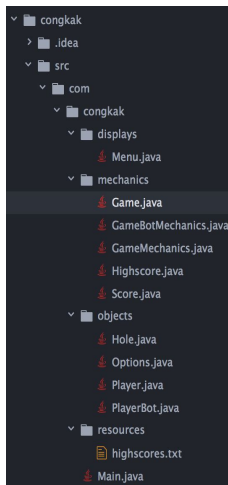
Time complexities of the methods in GameBotMechanics.java			
Methods/BigO	O(1)	O(N)	O(2N)
holesDeepCopy	•		
getBestMove		•	
getPlayableMovesArray		•	
validateBotMove	•		
arrayListToArray		•	
getBestMoveForSuggestion	•		

Time complexities of the methods in Highscore.java					
Methods/BigO	O(1)	O(2N)	O(2N+ $N^2$ )	O(n log n)	O(1 + n log n)
Highscore	•				
add	•				
fillScoresInsideArray	•				
printHighscores			•		
printHighscoresSorted			•		
resetScoresInFile	•				
sortScores					•
quicksort				•	

### 3. Work Done

The game is implemented using the Java programming language. There are many classes, algorithms and methods involved in the development of the game and, it is all discussed on this section.

Overview of the classes:



The files are also organised so that browsing through the code is easy.

Displays: Menu

Mechanics: Game, GameBotMechanics, GameMechanics, Highscore, Score

Objects: Hole, Options, Player, PlayerBot

Resources: highscores.txt

Main

Main.java at path: (com/congak/Main.java)

```
public class Main {  
  
    private static Scanner userInput = new Scanner(System.in);  
  
    public static void main(String[] args) {  
  
        int choice;  
  
        do {  
            Menu menu = new Menu();  
            Highscore highscore = new Highscore("com/congak/resources/highscores.txt/");  
            Options options = menu.start();  
  
            Game game = new Game(options);  
  
            game.startGame();  
  
            System.out.print("Go Back to Main Menu? (1 = Yes, 2 = Exit Game): ");  
            choice = userInput.nextInt();  
        } while (choice == 1);  
  
        System.out.println("Good bye!");  
    }  
}
```

The Main function shows an overview of how the game works in general with no in-depth explanations. Firstly, it creates both Menu and Highscore objects to be later used in the game. Menu class has a method that is called to enter the main menu. After the main menu exits, the method will return the Options object with the options for the game. The Highscore object is created using the path for highscores.txt as a String.

menu.start() will start the Main Menu, ask the user on the options for the game and then finally assign that returned Options object inside the Options object in Main.

The Game object is then created with the options passed in. Then to start the created game, the startGame() method is called - *this method will continue to execute until the game is over*. Once the game is over, the user can either exit or go back to Main Menu.

Menu.java at path: (com/congak/displays/Menu.java)

```
public Options start() {
    Scanner userChoice = new Scanner(System.in);
    Scanner gameT = new Scanner(System.in);
    while (choice!=1){
        if(!back){
            mainMenu();
            System.out.print("Enter choice: ");
            choice = userChoice.nextInt();
            clearScreen(05);
        }
        gameType = 10;
        back = false;

        switch (choice) {
        }
        if(back){
            choice = 99;
        }
    }

    return new Options();
}
```

The main method of Menu.java is start() where the user is presented a Main Menu. The main menu is simply used to get the options as inputs from the user before a game begins.

Here, there are switch statements and if statements involved in order to make the main menu work.

In the worst case scenario, the start() method will return a new Options object with the default options from the default Options constructor.

Game.java at path: (com/congak/mechanics/Game.java)

This is where things get complex, the Game object is the main object where the game operates using all of the methods in GameMechanics.java and other various classes.

The Game object is constructed using two constructors, one is for default games and the other is for games with custom options.

```
public Game() {  
    // DEFAULT GAME  
    this.options = new Options(1, 7, 4, "P1", "P2");  
    this.OS = System.getProperty("os.name");  
    this.userInput = new Scanner(System.in);  
    this.gameHasEnded = false;  
}
```

```
public Game(Options newOptions) {  
    // CUSTOM GAME WITH OPTIONS  
    this.options = newOptions;  
    this.OS = System.getProperty("os.name");  
    this.userInput = new Scanner(System.in);  
    this.gameHasEnded = false;  
}
```

```
private void coinflip(Random rand, Player playerOne, Player playerTwo) {  
    GameMechanics.clearScreen(OS);  
    System.out.print("Please wait, the coin is flipping! ");  
    GameMechanics.sleep(1000);  
    System.out.print("3 ");  
    GameMechanics.sleep(1000);  
    System.out.print("2 ");  
    GameMechanics.sleep(1000);  
    System.out.print("1\n");  
    GameMechanics.sleep(1000);  
  
    int coin = rand.nextInt();  
  
    if (coin > 250) {  
        System.out.printf("%s starts!\n", playerOne.getName());  
    } else {  
        Player.togglePlayer();  
        System.out.printf("%s starts!\n", playerTwo.getName());  
    }  
  
    GameMechanics.sleep(3000);  
    GameMechanics.clearScreen(OS);  
}
```

The coinflip() method does a nifty coin flip to determine which player to start the game. The function uses a random integer, checks whether it is greater than or less than 250, then determines the starting player based on that.

The biggest method inside Game.java is the startGame() method that essentially starts the game and plays until it is over.

```
public int startGame() {  
    int suggestedMoveFlag = -1;  
    if (options.getGameType() == 3) {  
    }  
  
    Hole holes[] = new Hole[options.getBoardSize()];  
  
    this.gameHasEnded = false;  
    boolean gameIsRunning = false;  
    boolean continueToMove = false;  
  
    int currentHole = 0;  
    int selectedIndex = 0;  
    int seedCountAtHand = 0;  
  
    Random rand = new Random();  
    GameMechanics.distributeSeeds(holes, options.getSeedCount());  
  
    Player playerOne = new Player(options.getPlayerOneName());  
    Player playerTwo = new Player(options.getPlayerTwoName());  
  
    Player.setCurrentPlayer(1);  
  
    do {  
    } while (!gameHasEnded);  
  
    gameIsRunning = false;  
  
    GameMechanics.clearScreen(OS);  
    GameMechanics.display(holes, options, playerOne.getScore(), playerTwo.getScore());  
  
    System.out.println("Game Ended!");  
    displayScores(playerOne, playerTwo);  
  
    return 0;  
}
```

The game is first initialized with the options that the Game was constructed on.

Inside the do while loop, each move is played until the game ends. Once it's done, the scores are displayed to the user.

The do while consist of many methods from GameMechanics.java that's involved in the core logic of the game.



The do while loop inside the startGame():

```
if (!gameIsRunning) {
    coinflip(rand, playerOne, playerTwo);
    gameIsRunning = true;
} else Player.togglePlayer();
```

First of all, it will check whether the game is running, if it is then toggle the player, otherwise do a coin flip since it is the start of the game. The starting player is determined by the coin flip.

```
do {
    if (Player.isPlayerOne()) {
        System.out.println(playerOne.getName() + " to move");
    } else {
        System.out.println(playerTwo.getName() + " to move");
    }

    if (suggestedMoveFlag != -1) {
    }

    System.out.print("Enter Move: ");
    selectedIndex = userInput.nextInt() - 1;

    System.out.println();

    selectedIndex = GameMechanics.offsetMoves(selectedIndex, options);

    if (!GameMechanics.validateMove(holes, options, selectedIndex)) {
        System.out.println("Move is not valid");
    }
} while (!GameMechanics.validateMove(holes, options, selectedIndex));

seedCountAtHand = holes[selectedIndex].getValue();
```

Next, the game will announce which player's turn it is, and then ask for the move.

For now, ignore what's under the if statement involving suggestMoveFlag. It will be explained when the bot comes up!

After the user enters the move, it is then offsetted through the method offsetMoves().

Finally, the move is validated through validateMove() method. If it is not, the program will prompt the user again.

The seed count at hand is how many seeds the hole that the user selected has. The game will start distributing the seeds inside seedCountAtHand to the other seeds.

```
// Clear the selected hole's value
holes[selectedIndex].clear();

//seeds from selected hole will be distributed to neighbouring holes and current hole will be increment for each distribution
currentHole = GameMechanics.seedDistribution(holes, options, currentHole, selectedIndex, seedCountAtHand, playerOne, playerTwo);

// check whether the player gets to continue or not
continueToMove = GameMechanics.checkContinueMove(holes, options, currentHole);
```

The game then clears the selected hole's value, since the user is taking seeds from it. Then, the seeds are distributed through the method seedDistribution()

```
public static int seedDistribution(Hole holes[], Options options, int currentHole, Player playerOne, Player playerTwo) {
    for (int i = 1; i <= seedCountAtHand; i++) {
        if (selectedIndex + i > options.getBoardSize() - 1) {
            selectedIndex -= options.getBoardSize();
            holes[selectedIndex + i].incrementValue();
        } else {
            holes[selectedIndex + i].incrementValue();
        }

        currentHole = selectedIndex + i;

        sleep();
        display(holes, options, playerOne.getScore(), playerTwo.getScore());
    }
    return currentHole;
}
```

The method essentially loops from 1 to seedCountAtHand whilst distributing seeds inside the Hole[] array, and then in the end returns the index of the location where it has finished distributing.

Once you have the index where the distribution finished, you use that variable along with the Hole[] array and options to check whether there is a continued move.

```
public static boolean checkContinueMove(Hole holes[], Options options, int currentHole) {  
    int check = checkEmptyHoles(holes, currentHole, options.getBoardSize());  
    if (check == 0) return false;  
    else return true;  
}
```

This method uses another method that returns the value of the next hole. If the value is not 0, then it is a continued move.

```
public static int checkEmptyHoles(Hole holes[], int currentHole, int boardSize) {  
    if (currentHole + 1 > boardSize - 1) {  
        return holes[currentHole + 1 - boardSize].getValue();  
    } else {  
        return holes[currentHole + 1].getValue();  
    }  
}
```

Returns the value of the next hole.

```
if (!continueToMove) {  
    GameMechanics.sleep();  
    if (Player.isPlayerOne()) {  
        int score = GameMechanics.obtainScore(currentHole, options, holes, playerOne.getScore(), playerTwo.getScore());  
        playerOne.setScore(score);  
    } else if (Player.isPlayerTwo()) {  
        int score = GameMechanics.obtainScore(currentHole, options, holes, playerOne.getScore(), playerTwo.getScore());  
        playerTwo.setScore(score);  
    }  
}
```

After checking for the continued move and if the user cannot move, the score is obtained from the hole next to empty hole. The player also loses his turn since it is almost the end of the while loop, and the togglePlayer() method is called at the start.

```
if (!GameMechanics.checkLegalMove(holes, options)) {  
    if (Player.isPlayerOne()) {  
        int score = GameMechanics.leftOverSeed(holes, options, playerOne.getScore(), playerTwo.getScore());  
        playerOne.setScore(score);  
    } else if (Player.isPlayerTwo()) {  
        int score = GameMechanics.leftOverSeed(holes, options, playerOne.getScore(), playerTwo.getScore());  
        playerTwo.setScore(score);  
    }  
    gameHasEnded = true;  
}
```

The last if statement for the while loop is to check for legal moves. In essence, what this method does is- first check whether there are any more legal moves for the player. If not, the leftover seeds are deposited to the opponent's home hole.

```
public static boolean checkLegalMove(Hole holes[], Options options) {
    int i;
    int count = 0;
    int boardBorders = options.getBoardBorders();
    int boardSize = options.getBoardSize();
    boolean flag = true;

    for (i = 0; i < boardBorders; i++) {
        if (holes[i].isEmpty()) {
            count++;
        }
        if (count == boardBorders && Player.isPlayerTwo()) {
            flag = false;

            System.out.println("No more legal move for Player 1.");
            System.out.println("Remaining seeds are deposited to Player 2 Home hole.\n");
        }
    }

    count = 0;
    for (i = boardSize - 1; i >= boardBorders; i--) {
        if (holes[i].isEmpty()) {
            count++;
        }
        if (count == boardBorders && Player.isPlayerOne()) {
            flag = false;

            System.out.println("No more legal move for Player 2.");
            System.out.println("Remaining seeds are deposited to Player 1 Home hole.\n");
        }
    }

    return flag;
}
```

If all the holes from one of the player's side are all empty, it determines that the game has ended.

If its player 1, check between 0 and boardBorders (middle of the board) and if it is player 2, check between the size of the board and boardBorders itself.

It also prints on the screen to let the user know that there are no more legal moves available.

```
    } while (!gameHasEnded);

    gameIsRunning = false;

    GameMechanics.clearScreen(OS);
    GameMechanics.display(holes, options, playerOne.getScore(), playerTwo.getScore());

    System.out.println("Game Ended!");

    displayScores(playerOne, playerTwo);

    return 0;
}
```

Once the method exits the while loop, indicating that the game has ended, it will set gameIsRunning to false, display the game state and finally print out the scores for the users to see.

```
private void displayScores(Player playerOne, Player playerTwo) {
    System.out.printf("%s's score: %d\n", playerOne.getName(), playerOne.getScore());
    System.out.printf("%s's score: %d\n", playerTwo.getName(), playerTwo.getScore());

    if (playerOne.getScore() > playerTwo.getScore()) {
        System.out.printf("%s Wins!", playerOne.getName());
        Highscore.add(playerOne.getName(), playerOne.getScore());
    } else if (playerTwo.getScore() > playerOne.getScore()) {
        System.out.printf("%s Wins!", playerTwo.getName());
        Highscore.add(playerTwo.getName(), playerTwo.getScore());
    } else {
        System.out.println("Draw!");
    }
    System.out.println();
}
```

The displayScores() method takes player one & player two, displays their scores and then finally add the winner's score inside the highscores.txt file through

the static method add() in the Highscore object.

Time to backtrack a little bit, what does the player do if the player gets a continued move?

```
if (!continueToMove) {  
    if (!gameIsRunning) {  
    } else Player.togglePlayer();  
    GameMechanics.clearScreen(OS);  
    GameMechanics.display(holes, options, playerOne.getScore(), playerTwo.getScore());  
    do {  
    } while (!GameMechanics.validateMove(holes, options, selectedIndex));  
    seedCountAtHand = holes[selectedIndex].getValue();  
} else {  
    selectedIndex = GameMechanics.continuedMove(options, currentHole);  
    seedCountAtHand = holes[selectedIndex].getValue();  
}
```

It will then get the index of where the next move starts and set it inside selectedIndex. Finally, it adds the seeds inside the selectedIndex to seedCountAtHand. Either this happens, or the program asks the user for a new move.

This concludes the Game object. There still are many methods involved in the development of the class, they will be discarded as they are not much of an importance to the game logic compared to the ones mentioned.

Highscore.java at path: (com/congak/mechanics/Highscore.java)

The Highscore object is one of the core-objects of the game that is used for manipulating the highscores.txt file located at path: (com/congak/resources/highscores.txt). The class is able to reset, view, view sorted and finally, append to the highscores.txt file through the methods developed.

```
public static void printHighscoresSorted() {
    sortScores();

    for (int i = 0; i < 23; i++) {
        if (i % 2 == 0) System.out.print("+ ");
        else System.out.print("- ");
    }
    System.out.println();

    System.out.println("|                Highscores!                |");

    for (int i = 0; i < 23; i++) {
        if (i % 2 == 0) System.out.print("+ ");
        else System.out.print("- ");
    }
    System.out.println();

    int count = 1;

    for (int c = scores.size() - 1; c >= 0; c--) {
        System.out.format("| %3d. %-25s :    %-4d  |\n", count, scores.get(c).getPlayerName(), scores.get(c).getValue());

        for (int i = 0; i < 23; i++) {
            if (i % 2 == 0) System.out.print("+ ");
            else System.out.print("- ");
        }
        System.out.println();
        count++;
    }
}
```

This method prints out what's inside the txt file sorted, it does not change to the file itself leaving it untouched. The printing part is self-explanatory, however the sortScores() method is not.

```
private static void sortScores() {
    fillScoresInsideArray();
    quicksort(scores, 0, scores.size() - 1);
}
```

```
private static ArrayList<Score> scores;
```

It will fill the scores inside the Highscore object, and then do a quicksort!

The scores are stored inside the arraylist scores, and can be accessed in a static context.



```

public static void fillScoresInsideArray() {
    scores.clear();
    try {
        File file = highscoreFile;

        Scanner scanner = new Scanner(file);

        ArrayList<String> names = new ArrayList<String>();
        ArrayList<Integer> values = new ArrayList<Integer>();

        String temp = "";
        while (scanner.hasNextLine()) {
            if (!scanner.hasNextInt()) {
                temp += scanner.next() + " ";
            } else {
                scores.add(new Score(temp, scanner.nextInt()));
                temp = "";
            }
        }

        for (int i = 0; i < names.size(); i++) {
            scores.add(new Score(names.get(i), values.get(i)));
        }
    } catch (Exception e) {
        System.err.println("Exception: " + e.getMessage());
    }
}

```

The fillScoresInsideArray() is a fun method that goes through the highscores.txt, then stores the names and the values accordingly to the arraylist.

For every line in the text file, until you find an integer, all the characters are added to a temporary string. Then once you find an integer, the integer is then added to the scores location. Every value is added to both arraylists respectively.

Once that's done, all the names and values are added inside the scores arraylist of the Object for later use.

```

private static void quicksort(ArrayList<Score> arrayList, int from, int to) {
    if (from < to) {
        int pivot = from;
        int left = from + 1;
        int right = to;
        int pivotValue = arrayList.get(pivot).getValue();
        while (left <= right) {
            // left <= to -> limit protection
            while (left <= to && pivotValue >= arrayList.get(left).getValue()) {
                left++;
            }
            // right > from -> limit protection
            while (right > from && pivotValue < arrayList.get(right).getValue()) {
                right--;
            }
            if (left < right) {
                Collections.swap(arrayList, left, right);
            }
        }
        Collections.swap(arrayList, pivot, left - 1);
        quicksort(arrayList, from, left - 1); // pivot
        quicksort(arrayList, left + 1, to); // pivot
    }
}

```

This is a quicksort method for sorting the scores arrayList as efficiently as possible. The worst case for time complexity is  $O(N^2)$ .

In essence, the algorithm will divide the array, and each partition will be sorted recursively by also dividing recursively whilst choosing pivot points.

```

public static void resetScoresInFile() {
    try {
        PrintWriter writer = new PrintWriter(highscoreFile);
        writer.print("Testscore -999");
        writer.close();
        System.out.println("Highscore file is resetted!");
        GameMechanics.sleep(3000);
    } catch (Exception e) {
        System.out.println("Cannot reset the scores");
        System.err.println("Exception: " + e.getMessage());
    }
}

```

This method resets the values inside highscores.txt and then appends the value "Testscore -999" to it, for indication.

```

public static void add(String name, int score) {
    try {
        String filename = highscoreFilePath;
        FileWriter fileWriter = new FileWriter(filename, true);
        fileWriter.write("\n" + name + " " + score);
        fileWriter.close();
    } catch (Exception e) {
        System.err.println("IOException: " + e.getMessage());
    }
}

```

This method adds a score to the text file. The score is made up of two values, name and score, that is differed by their data types. The scores are always integers whereas the names are Strings.

There is also a Score object to keep them saved.

GameBotMechanics.java at path: (com/congak/mechanics/GameBotMechanics.java)

The part that was ignored before, regarding the suggestedMoveFlag, is about the bot that suggests the move for the player. The bot will essentially play every possible move, and from those moves it will determine which one is the greatest and then show it to the user (if the user chose the advanced game).

```

public static int getBestMove(Hole[] holes, Options options, PlayerBot player) {
    holesObject = holesDeepCopy(holes);

    int[] moves = getPlayableMovesArray(holes, options, player);

    int highestScore = Integer.MIN_VALUE;
    int playedMoveScore;
    int bestMove = 1;

    for (int i = 0; i < moves.length; i++) {
        holesObject = holesDeepCopy(holes);

        // Create a new BOT player inside the function: playMove, with the passed in player's score
        playedMoveScore = playMove(holesObject, options, new PlayerBot(player.getScore()), moves[i]);

        if (playedMoveScore > highestScore) {
            highestScore = playedMoveScore;
            bestMove = moves[i];
        }
    }

    return bestMove;
}

```

This is the main method that gets the best move uses the status of the Hole[] holes array and the options to get the current status of the game. The PlayerBot player is used to play the move.

First of all, the object holes is deep copied into the holesObject (which is a field of GameBotMechanics) through the holesDeepCopy method.

```
private static Hole[] holesDeepCopy(Hole[] holesOriginal) {  
    Hole[] holesCopy = null;  
  
    try {  
        ByteArrayOutputStream bos = new ByteArrayOutputStream();  
        ObjectOutputStream out = new ObjectOutputStream(bos);  
        out.writeObject(holesOriginal);  
        out.flush();  
        out.close();  
  
        // Create an input stream from the byte array and read a copy of the object back in.  
        ObjectInputStream in = new ObjectInputStream(  
            new ByteArrayInputStream(bos.toByteArray()));  
  
        return (Hole[]) in.readObject();  
    } catch (Exception e) {  
        System.out.println("Deep Copy unsuccessful. ");  
        System.err.println("Exception: " + e.getMessage());  
  
        return holesOriginal;  
    }  
}
```

The above method does a deep copy of the array, so that arrays don't get reused where they are not supposed to and alter the real game. It first writes the original array inside the output stream, flushes it and then reads it back in using input stream to later be returned by the method after casting. If there is an Exception error, it lets the user know through the command line and returns the passed in object.

```
private static int[] getPlayableMovesArray(Hole holes[], Options options, PlayerBot player) {  
    if (!player.isBot()) {  
        player.setId(Player.getCurrentPlayer());  
    }  
    int move = 1;  
    ArrayList<Integer> playableMoves = new ArrayList<Integer>();  
  
    for (int i = 0; i < options.getBoardBorders(); i++) {  
        if (player.getId() == 2) {  
            move = GameMechanics.offsetMoves(move, options);  
        }  
  
        if (validateBotMove(holes, options, move, player)) {  
            playableMoves.add(i + 1);  
        }  
        move = i+1;  
        move++;  
    }  
  
    return arrayListToArray(playableMoves);  
}
```

After the deep copy of holes, getBestMove() then gets all the playable moves inside an integer array as integers. It uses the method getPlayableMovesArray() to do this.



First, if the playerBot is not initialized, it will set the id to the current player id. Move is a variable that will have the value i+1 to check for every move.

The arrayList playableMoves will, in the end, have all the playable moves. In the for loop, variable "i" will start at 0 and increment until it is greater than or equal to the board borders, which is basically all the count of all holes at one side of the board. The possible moves are from 1 until the value of boardBorders. If the bot is player 2, the moves will be offsetted. After that, it is validated through the method validateBotMove(). It is almost the same method as validateMove() from GameMechanics, however it is made for the bot moves. If the move is valid, the value i+1 is added to the arrayList (that value is the move itself). This is done for every possible valid move made by the bot player.

After all the playable moves are obtained, the moves are then converted into an int array, and returned to the playBestMove() method.

```
private static int[] arrayListToArray(ArrayList<Integer> arrayList) {  
    int[] array = new int[arrayList.size()];  
  
    for (int i = 0; i < arrayList.size(); i++) {  
        array[i] = arrayList.get(i);  
    }  
    return array;  
}
```

Converts the arrayList to an int array by copying all the values directly.

```
for (int i = 0; i < moves.length; i++) {  
    holesObject = holesDeepCopy(holes);  
  
    // Create a new BOT player inside the function: playMove, with the passed in player's score  
    playedMoveScore = playMove(holesObject, options, new PlayerBot(player.getScore()), moves[i]);  
  
    if (playedMoveScore > highestScore) {  
        highestScore = playedMoveScore;  
        bestMove = moves[i];  
    }  
}  
  
return bestMove;
```

So now that all the playable moves are obtained, the method then plays every move and gets the score from that move. The score is compared to the highest score so far, and if it is bigger, it will get replaced. The bestMove is also set to the move that was played with the highest score.

The playMove() method is a copy of startGame() inside Game.java. Instead of playing an entire game, it will only play one move. The methods inside are made fit for the bot player. The method also returns the score made from the move played for comparison later.

```

if (suggestedMoveFlag != -1) {
    PlayerBot playerBOT;
    if (Player.getCurrentPlayer() == 1) {
        playerBOT = new PlayerBot(playerOne.getScore());
    } else {
        playerBOT = new PlayerBot(playerTwo.getScore());
    }
    int bestMove = GameBotMechanics.getBestMove(holes, options, playerBOT);
    bestMove = getBestMoveForSuggestion(bestMove, options.getHoleCount());
    suggestedMoveDisplay(suggestedMoveFlag, bestMove);
}

```

Coming all the way back to startGame() of Game.java, this is where the best move is suggested to the user.

getBestMove() is passed along with the current holes status, options and the newly created bot player that is constructed using the current player's score. The bestMove is generated and it is later converted using getBestMoveForSuggestion() so that the player can see it properly.

```

private void suggestedMoveDisplay(int suggestedMoveFlag, int bestMove) {
    switch(suggestedMoveFlag) {
        // -1 : Do not display
        case -1:
            break;

        // 0 : For both players
        case 0:
            System.out.println("Suggested Move: " + bestMove);
            break;

        // 1 : For Player 1
        case 1:
            if (Player.getCurrentPlayer() == 1) {
                System.out.println("Suggested Move: " + bestMove);
            }
            break;

        // 2 : For Player 2
        case 2:
            if (Player.getCurrentPlayer() == 2) {
                System.out.println("Suggested Move: " + bestMove);
            }
            break;

        // 3 :
        case 3:
            break;
    }
}

```

Lastly, the move is shown through this method. The suggestedMoveFlag is manipulated by the user after the user selections the advanced game type.

## 4. Installation & Testing

### Installation

The installation process is fairly simple.

1. Make sure you have JDK installed to compile the code.
2. Add Java to PATH.
3. Open command prompt and locate to congak/src/
4. Then on the command line, enter the command

```
'javac com/congak/Main.java && java com/congak/Main'
```

That's it, you'll then be greeted by the main menu.

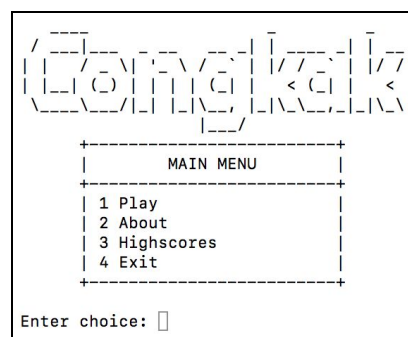
### Program Testing

Testing will be categorized into 5 major parts:


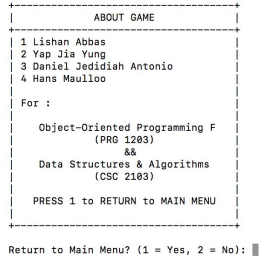
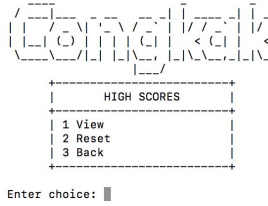
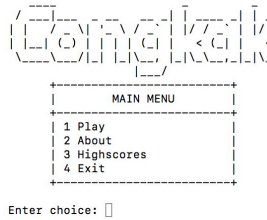
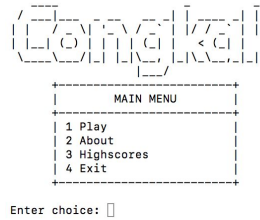
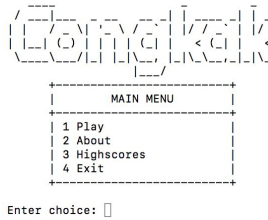
- Main Menu testing
- Highscore file manipulation
- Different game types
- Different board sizes and seed counts
- Suggested Move testing
- Game states testing

### Main Menu Testing

As soon as Main.java is compiled and executed, the user is presented with the MAIN MENU of the game. There are multiple pages inside the Main Menu that will guide the user throughout the game.



There are four options, and the output for choosing them respectively are:

1	2	3	4
			<p>Good bye! \$ █</p>
All inputs work, next is going back with the respective inputs.			
4	1	3	
			
All inputs work for this segment of the main menu. Tests passed.			

## Highscore file manipulation

Next up is the highscore file manipulation. The algorithms explained before will now be tested.

First, we will locate to the highscores.txt file using the terminal, and read the file using vim.

```
$ ls
highscores.txt
$ vim highscores.txt █
```

Then, we will manually enter highscores inside the file.

```
1 Hans 77
2 Lishan 95
3 Daniel 102
4 Jia Yung 44 █
```

These values can be viewed from the game as well with the use of the Highscore Object.

In the HIGH SCORES menu, there three options; views, reset and back. Views will call the printHighscoresSorted() function. Here we will be able to see the results of the quicksorting algorithm implemented, as well as the joint methods.

HIGH SCORES &gt; View

[illegible]

Even though in the file, the highscores weren't in order, the quicksort algorithm managed to sort it inside the static `ArrayList<Score> scores`, after getting the values inside it.

Test passed.

HIGH SCORES > Reset

```
Enter choice: 2
Are you sure? (1 = Yes, 2 = No):
```

```
Highscore file is resetted!
```

# Google

```

+-----+
|           HIGH SCORES           |
+-----+
| 1 View                           |
| 2 Reset                         |
| 3 Back                         |
+-----+

```

Enter choice:

The highscores have been reset! Now lets view it once again to see the contents.

```

+ - + - + - + - + - + - + - + - + - +
|                                     |
|               Highscores!         |
| - + - + - + - + - + - + - + - + |
|   1. Testscore                   : -999 |
+ - + - + - + - + - + - + - + - + - +

```

Testscore is a value that is added after the resetting of scores for testing purposes.

That concludes the highscore file manipulation section.

## Different game types testing

The game is made up of three game types; basic, intermediate and advanced.

Basic will only ask for the player names, because the board setup will be DEFAULT.

Intermediate will ask for the board setup and the player names.

Advanced will also ask for the board setup and the player names, it will also further ask the user whether the user wants the bot to suggest moves. The user will have four options, it will be explained hereinafter.

Play > Basic

```
Game Type: Basic
```

```
Player 1's Name: Hans
```

```
Player 2's Name: Jia Yung
```

Entered "Hans" and "Jia Yung" as player names.

```
Please wait, the coin is flipping! 3 2 1
Jia Yung starts!
```

The game then does a coin flip as mentioned before.

```
+-----+
|               |
|  Player2      | 7   6   5   4   3   2   1   Player1 | | | | | | |
| + - + | 004 | 004 | 004 | 004 | 004 | 004 | + - + |
| | 000 | + - + | - + - + - + - + - + - + | | 000 | |
| + - + | 004 | 004 | 004 | 004 | 004 | 004 | + - + |
|   Score      | 1   2   3   4   5   6   7   Score  |
|               |
+-----+

Jia Yung to move
Enter Move: 
```

The game began with DEFAULT options:

- Board size: 7 holes per side
- Seed count: 4 seeds per hole

Test Passed.

Play > Intermediate

Game Type: Intermediate

Enter Board Size: 4

Enter Seed Count: 225

Player 1's Name: Lishan

Player 2's Name: Daniel

Board Size is 4, seed count is 225 and the two player names are "Lishan" and "Daniel".

Please wait, the coin is flipping! 3 2 1  
Lishan starts!

Once again, the coin flip determines the starting player.

+-----+														
		4		3		2		1						
Player2	+	-	+		225		225		225		225	Player1		
+	-	+		225		225		225		225		+	-	+
	000		+	-	+	-	+	-	+	-	+		000	
+	-	+		225		225		225		225		+	-	+
Score	+	-	+	-	+	-	+	-	+	-	+	Score		
		1		2		3		4						
+-----+														
Lishan to move														
Enter Move: <input type="text"/>														

The game began with the custom options entered by the user. Test passed.

Play > Advanced

Game Type: Advanced

Enter Board Size: 10

Enter Seed Count: 45

Player 1's Name: Hans

Player 2's Name: Daniel

Board size is 10, seed count is 45 and the two player names are now "Hans" and "Daniel".

Extra options for Game Type: Advanced  
Do you want a bot to suggest moves for you? (Enter the following values!)

- 1 : Do not display suggested moves
- 0 : For both players
- 1 : For Player 1
- 2 : For Player 2

Your choice:

Instead of the usual coin flip, this time we get a prompt with more options. This is the feature built into advanced game, where the user gets to select how the suggested moves should display. Next we will test all four values.

To test the suggested moves displays, we will use a board size of 2 and seed count of 1.

Player 1 is “P1”

Player 2 is “P2”

-1	0	1	2
<i>Player 2 Starts</i>	<i>Player 2 Starts</i>	<i>Player 1 Starts</i>	<i>Player 1 Starts</i>
<div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  001   001  </div><div>Player1</div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>  001   001  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P2 to move Enter Move: <input type="text"/></div><div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  000   000  </div><div>Player1</div><div>+ - +</div><div>  002  </div><div>+ - +</div><div>  000   002  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P1 to move Enter Move: <input type="text"/></div></div></div>	<div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  001   001  </div><div>Player1</div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>  001   001  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P2 to move Suggested Move: 2 Enter Move: <input type="text"/></div><div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  000   000  </div><div>Player1</div><div>+ - +</div><div>  002  </div><div>+ - +</div><div>  000   002  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P1 to move Suggested Move: 2 Enter Move: <input type="text"/></div></div></div>	<div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  001   001  </div><div>Player1</div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>  001   001  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P1 to move Suggested Move: 2 Enter Move: <input type="text"/></div><div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  002   000  </div><div>Player1</div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>  000   000  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P2 to move Enter Move: <input type="text"/></div></div></div>	<div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  001   001  </div><div>Player1</div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P1 to move Enter Move: <input type="text"/></div><div><div><div><div><div>2</div><div>1</div></div><div>Player2</div><div>+ - +</div><div>  002   000  </div><div>Player1</div><div>+ - +</div><div>  000  </div><div>+ - +</div><div>  000   000  </div><div>+ - +</div><div>Score</div><div>+ - +</div><div>1 2</div><div>Score</div></div></div><div>P2 to move Suggested Move: 2 Enter Move: <input type="text"/></div></div></div>
The suggested move is turned off for both players.	The suggested move is turned on for both players.	The suggested move is turned on for Player 1.	The suggested move is turned on for Player 2.
All tests passed.			



## Game states testing

This section will test moves and check whether game states after the entered moves are correct. The testing will be done on an advanced game with board size 5 and seed count 4.

Tests: Game Logic (1 entire move, Seed distribution, Checking for continued move, Checking for a legal move, end-game) and Suggested Move (Best move calculation).

Game Setup	
<p><b>Game Type: Advanced</b></p> <p><b>Enter Board Size: 5</b></p> <p><b>Enter Seed Count: 4</b></p> <p><b>Player 1's Name: P1</b></p> <p><b>Player 2's Name: P2</b></p>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <p>Extra options for Game Type: Advanced</p> <p>Do you want a bot to suggest moves for you? (Enter the following values!)</p> <p>-1 : Do not display suggested moves</p> <p>0 : For both players</p> <p>1 : For Player 1</p> <p>2 : For Player 2</p> <p>Your choice: 1</p> </div> <div style="border: 1px solid black; padding: 5px;"> <p>Please wait, the coin is flipping! 3 2 1</p> <p>P2 starts!</p> </div>

Game States		
#	Screenshot	Description
1	<pre> +-----+        5      4      3      2      1          Player2 + - + - + - + - + - + Player1     + - +   004   004   004   004   004   + - +       000   + - + - + - + - + - +   000     + - +   004   004   004   004   004   + - +     Score + - + - + - + - + - + Score          1      2      3      4      5        +-----+  P2 to move Enter Move: 1 </pre>	Since the coin flip announced that P2 starts, the game starts with P2 to move. P2 is going to enter the move: 1.
2	<pre> +-----+        5      4      3      2      1          Player2 + - + - + - + - + - + Player1     + - +   005   005   005   000   000   + - +       005   + - + - + - + - + - +   000     + - +   000   005   005   005   005   + - +     Score + - + - + - + - + - + Score          1      2      3      4      5        +-----+  P1 to move Suggested Move: 2 Enter Move: 2 </pre>	Player 2 made the move 1 and got 5 score from that move.

## The Game Logic

Let's take a look at the move that was made by P2 at the start.

The game state was at first:

4 4 4 4 4  
4 4 4 4 4

Since P2 played the move 1, the game will first empty the selected hole, and then distribute that seed amongst the next holes.

Therefore:

<b>4 4 4 4 0</b> <b>4 4 4 4 4</b>	<b>4 4 4 5 0</b> <b>4 4 4 4 4</b>	<b>4 4 5 5 0</b> <b>4 4 4 4 4</b>	<b>4 5 5 5 0</b> <b>4 4 4 4 4</b>	<b>5 5 5 5 0</b> <b>4 4 4 4 4</b>
--------------------------------------	--------------------------------------	--------------------------------------	--------------------------------------	--------------------------------------

Now that all 4 seeds are distributed, the game is going to check whether the player has a continued move. In this case, the player can move because the next hole is not empty, it has the value of four.

Distribute the 4 seeds:

<b>5 5 5 5 0</b> <b>0 4 4 4 4</b>	<b>5 5 5 5 0</b> <b>0 5 4 4 4</b>	<b>5 5 5 5 0</b> <b>0 5 5 4 4</b>	<b>5 5 5 5 0</b> <b>0 5 5 5 4</b>	<b>5 5 5 5 0</b> <b>0 5 5 5 5</b>
--------------------------------------	--------------------------------------	--------------------------------------	--------------------------------------	--------------------------------------

The 4 seeds are once again distributed amongst the next holes. Now, the game will check again whether the player has a continued move. In this case, the player does not have a continued move because the next hole is empty. Therefore, the game will then take the values inside the hole next to the empty hole (in this case the hole with value 5), and then deposit that inside the home hole of P2 which is located in the left.

Then the game would look like this:

5 5 5 5 0 0  
0 5 5 5 5 0

That's how the game will continue to play until the special condition is met: Whether the player has a legal move left (end of the game). The next screenshot will be the 13th move of the game, where P2 is to move.

(The game logic is further explained in the next pages)

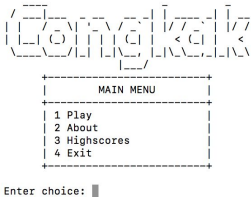
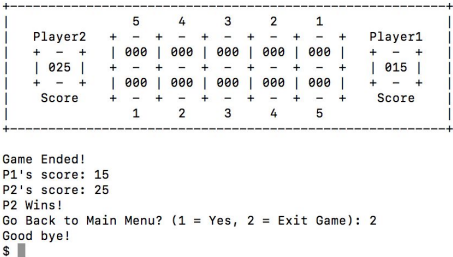
13	<div><div><div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div></div></div><div><div></div><div></div><div>&lt;/</div></div></div></div></div>
----	--

While loop at Main.java

```
public static void main(String[] args) {  
  
    int choice;  
  
    do {  
        Menu menu = new Menu();  
        Highscore highscore = new Highscore("com/congak/resources/highscores.txt/");  
        Options options = menu.start();  
  
        Game game = new Game(options);  
  
        game.startGame();  
  
        System.out.print("Go Back to Main Menu? (1 = Yes, 2 = Exit Game): ");  
        choice = userInput.nextInt();  
    } while (choice == 1);  
  
    System.out.println("Good bye!");  
}
```

Recalling back to the main method in Main.java, the last prompt, “Go back to Main Menu..” is presented to the user. As mentioned, if the user chooses 1, the while loop will continue to run, otherwise the main method will finish running.

The values are tested below.

1	2
	
Test Passed.	

Suggested Move Calculation

The suggested move feature that is available in an Advance game works like a greedy algorithm where the method returns the move with the best score. The algorithm is deeply explained previously in this document, therefore we will just test the algorithm using multiple tracing tables to prove that the algorithm works in this section.

To test, we will use the same game we used before:

```
+-----+  
| Player2 | 5 | 4 | 3 | 2 | 1 | Player1 |  
| + - + | 007 | 001 | 000 | 004 | 001 | + - + |  
| 020 | + - + | - + - + - + - + - + | 003 |  
| + - + | 000 | 000 | 000 | 001 | 003 | + - + |  
| Score | + - + | - + - + - + - + - + | Score |  
| 1 | 2 | 3 | 4 | 5 |  
+-----+  
  
P1 to move  
Suggested Move: 4  
Enter Move: 4
```

This is a random moment in the game where Player 1 has a score of 3, and Player 2 has a score of 20. Here, P1 is to move with the suggested move: 4. Since the available moves are 4 and 5, let's run the algorithm on both moves and check to see which one scores the highest.

Move	Game States	Score					
4	<table><tr><td>7 1 0 4 1 0 0 0 1 3</td><td>7 1 0 4 1 0 0 0 0 4</td><td>7 1 0 5 0 0 0 0 0 4</td><td>7 0 0 5 0 0 0 0 0 4</td></tr></table> <p>After the distribution, there is no continued move. The hole next to the empty hole has a value of 1, therefore the score is 1 from this move.</p>	7 1 0 4 1 0 0 0 1 3	7 1 0 4 1 0 0 0 0 4	7 1 0 5 0 0 0 0 0 4	7 0 0 5 0 0 0 0 0 4	1	
7 1 0 4 1 0 0 0 1 3	7 1 0 4 1 0 0 0 0 4	7 1 0 5 0 0 0 0 0 4	7 0 0 5 0 0 0 0 0 4				
5	<table><tr><td>7 1 0 4 1 0 0 0 1 3</td><td>7 1 0 4 2 0 0 0 1 0</td><td>7 1 0 5 2 0 0 0 1 0</td><td>7 1 1 5 2 0 0 0 1 0</td><td>8 0 1 5 2 0 0 0 1 0</td></tr></table> <p>After the distribution once again, the player is left with no continued move. The hole next to the empty hole is also empty, therefore the player obtains no score from this move.</p>	7 1 0 4 1 0 0 0 1 3	7 1 0 4 2 0 0 0 1 0	7 1 0 5 2 0 0 0 1 0	7 1 1 5 2 0 0 0 1 0	8 0 1 5 2 0 0 0 1 0	0
7 1 0 4 1 0 0 0 1 3	7 1 0 4 2 0 0 0 1 0	7 1 0 5 2 0 0 0 1 0	7 1 1 5 2 0 0 0 1 0	8 0 1 5 2 0 0 0 1 0			

Test Passed for this move. This shows that the algorithm returns the next best move.

## 5. Conclusion

In conclusion, this project has been a great lesson overall for all the members of the group. It has taught us about data structures, algorithms and even the Java programming language. Prior to taking this course, none of us had experience in Java programming. But now, we all have a deep understanding of what Java is along with countless programming concepts that we have learnt on the way. The assignment overall was not easy- it took its turns, but in the end, we had finished the entire assignment right on time! For a group project like this, teamwork and communication are both two crucial elements for a successful project. Our group managed to work hard together with great communication that always kept everyone updated and organised.

# 6. Reflections

## Lishan Abbas

### Reflection

For this project, I acted as a group leader to make sure that we as a group gets the job done on time. Jia Yung was assigned to work on the game logic, Hans and Daniel were assigned to work on the classes and the implementations. I worked on developing the core-classes of the project and, kept the code organised and heavily commented.

### Contributions

Game, GameBotMechanics, Highscore, PlayerBot, Options, Object and Report.

### Problems encountered

There were many instances where I almost gave up because of the tough problems I encountered. The hardest problem of them all was the issue with passing the game array inside bot mechanics to generate the best move. The array object kept altering outside the real game, and to solve this, Jia Yung and I worked on this method to create a deep copy of the array. The method worked and fixed the crazy problem of the game being altered outside the Game object. There were many other problems I had to solve as time went by. Regardless, it's great to have all those problems solved with the help of my group members.

### Strength and Weakness

Strength: Setting up the game, classes and solving problems. There were obstacles, however I managed to overcome them in the end.

Weakness: Java, still very new to the language. Setting up the bot was quite tough too- it was reimplemented from scratch three times.

### Group member contributions:

1. Lishan Abbas - 30%
2. Yap Jia Yung - 30%
3. Hans Maulloo - 20%
4. Daniel J. Antonio - 20%

## **Yap Jia Yung**

### **Reflections**

For this assignment, my main focus is to work on the game logic. Upon getting the assignment, I am assigned to get the game logic right and I am glad to do so because I find it the most challenging part for the entire assignment. The game logic is like the back bone of the entire assignment because without a functioning game, there is no purpose of implementing other aspects. Thus, I carry a heavy responsibility to make sure what my group mates have done won't put into a waste.

### **Contributions**

Game mechanic class, Game,Game Bot Mechanics, Report, flowchart

### **Problems encountered**

Along the work, I encounter numerous problem both on the coding side and user side. One is to make the code bug free and organise so my group mates are able to understand my code and the other part is to make the code user friendly by making them flexible and knowing what's going on during the gameplay. Despite the problem I face and number of changes I had made for the code, I never gave up because I am determined to finish the task given.

### **Strength and Weakness**

Strength: Game logic is not too hard for me, although I face some problem while coding, but I am able to debug them easily.

Weakness: Implement OOP practice into my code.

### **Group member contributions:**

1. Lishan Abbas - 30%
2. Yap Jia Yung - 30%
3. Hans Maulloo - 20%
4. Daniel J. Antonio - 20%

## **Hans Maulloo**

### **Reflections**

My main focus concerning the project, at first, was to build a GUI for the game, which I actually started, but couldn't finish. Later on, I was only assigned to design the main menu, since I'm not really good at implement logics inside the game. So I focused a bit more on the main menu for CLI. Also, I explained on the work done for the OOP documentation, and helped in some parts while writing the documentation.

### **Contributions**

MainMenu design along with Lishan

Work done for OOP documentation

Flowchart along with Yap Jia Yung

### **Problems encountered**

While designing the GUI, I had issues in implementing the game board and it became very complex. Later while designing the main, I faced some issues on the option to back from one Menu to previous one. I was able to do it later on with help on groupmates but did not implement it on the game because it is very complex and a lot more research need to be done for other part.

### **Strength and weakness**

I can personally say I didn't work as much as I had to on the project, since my group mates worked a lot. I was not always up to date with my team, and it was by the last two weeks that I started designing the menu for the game.

### **Group member contributions percentage**

1. Lishaan Abbas: 30%
2. Daniel J. Antonio: 30%
3. Jia Yung Yap: 30%
4. Hans Maulloo: 10%



## **Daniel Jedidiah Antonio**

### **Reflection**

My contribution to the group work this time isn't one to be proud of, honestly i personally am quite ashamed of my own contribution. For this assignment there are a lot of things that i could have done to speed up the progress, however i let my fellow teammates handle it without checking up on it.

### **Contribution**

Flowchart, documentation, Hole, and testings

### **Problems encountered**

Problems i encounter in this assignment comes into 2 parts, one which is my lack of motivation, laziness, and bad time-management, and second is communication. As for the first one, i am constantly trying to improve by day, and for the second one, as time goes Lishan talked to us on how we should talk to each other and do the work more on google docs as we can all monitor each other's work.

### **Strength and Weakness**

My evaluation of our work is that it is strong in terms of user interaction, however there could still be a lot more improvements done towards the end result. We could have added more exception handling for the inputs as wrong inputs would crash the code.

According to my evaluation, each group member's contribution is as so:

- |                           |     |
|---------------------------|-----|
| 1. Lishan Abbas           | 30% |
| 2. Yap Jia Yung           | 30% |
| 3. Hans Maulloo           | 20% |
| 4. Daniel J. Antonio (Me) | 20% |