--Modified Roguelike

Dr. Carol O Sullivan has allowed me to disregard Section B of the report due to the complexity and number of classes of the game that I have explained in greater detail in the report regardless.

--Section A/B--

--Introduction--

This game is based a modified version of Roguelike. Roguelike is a subgenre of role-playing video games characterized by a dungeon crawl through procedurally generated game levels, turn-based gameplay, tile based graphics, and permanent death of the player-character.

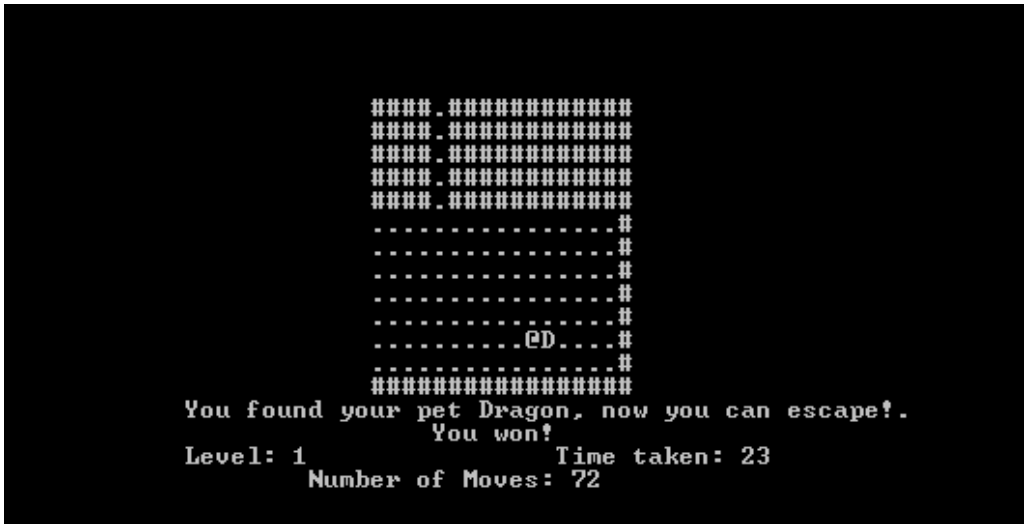
Main objectives for the game to assimilate a Roguelike atmosphere were:

- Load ASCII level from a level file
- Allow player movement using W A S D
- Prevent movement into walls
- Have enemies that randomly move around and allow player to attack them
- Allow enemies to attack player
- Create a high score class that saves after every players turn
- Allow player to retry

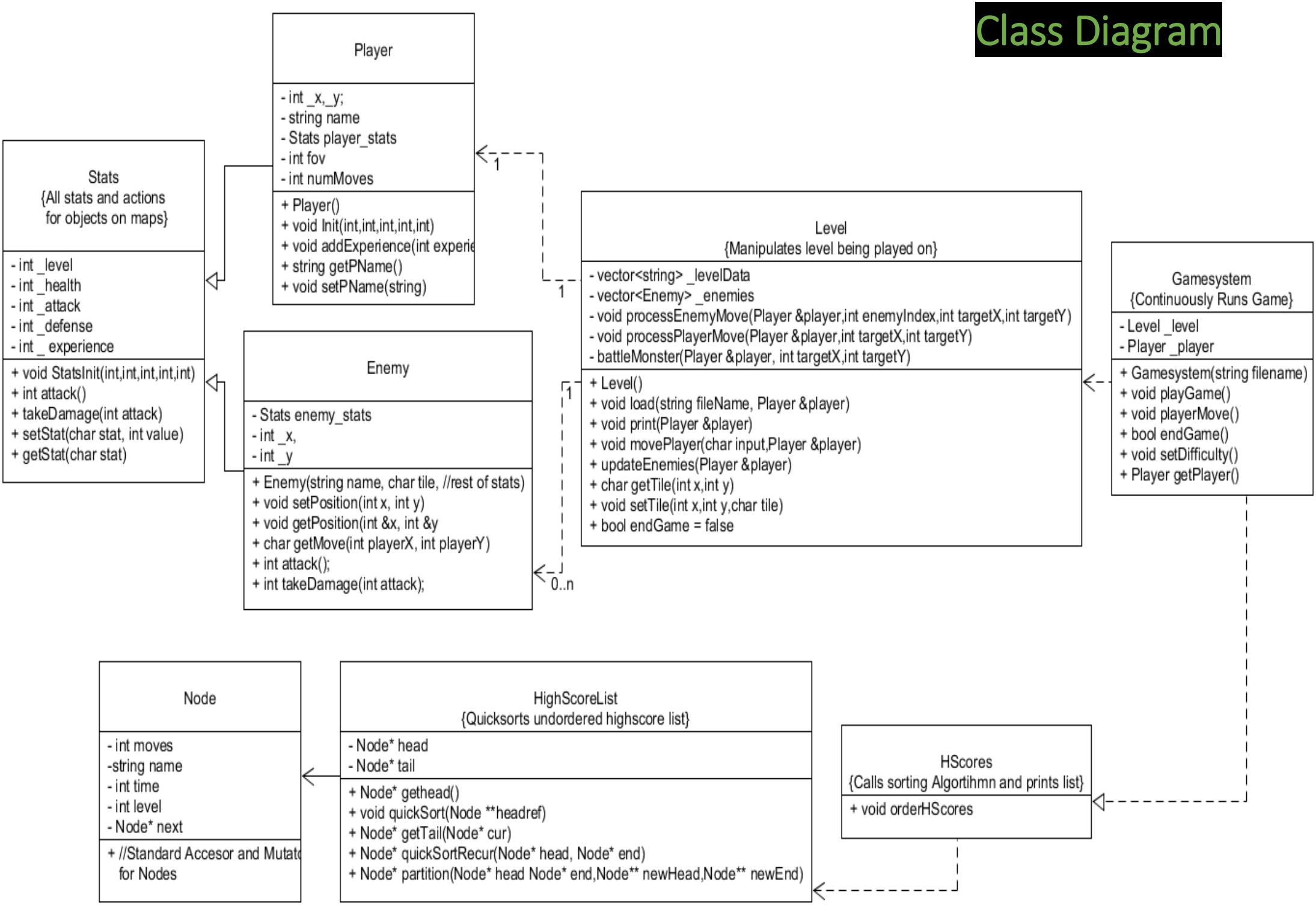
My version of the game has a different/unique story,

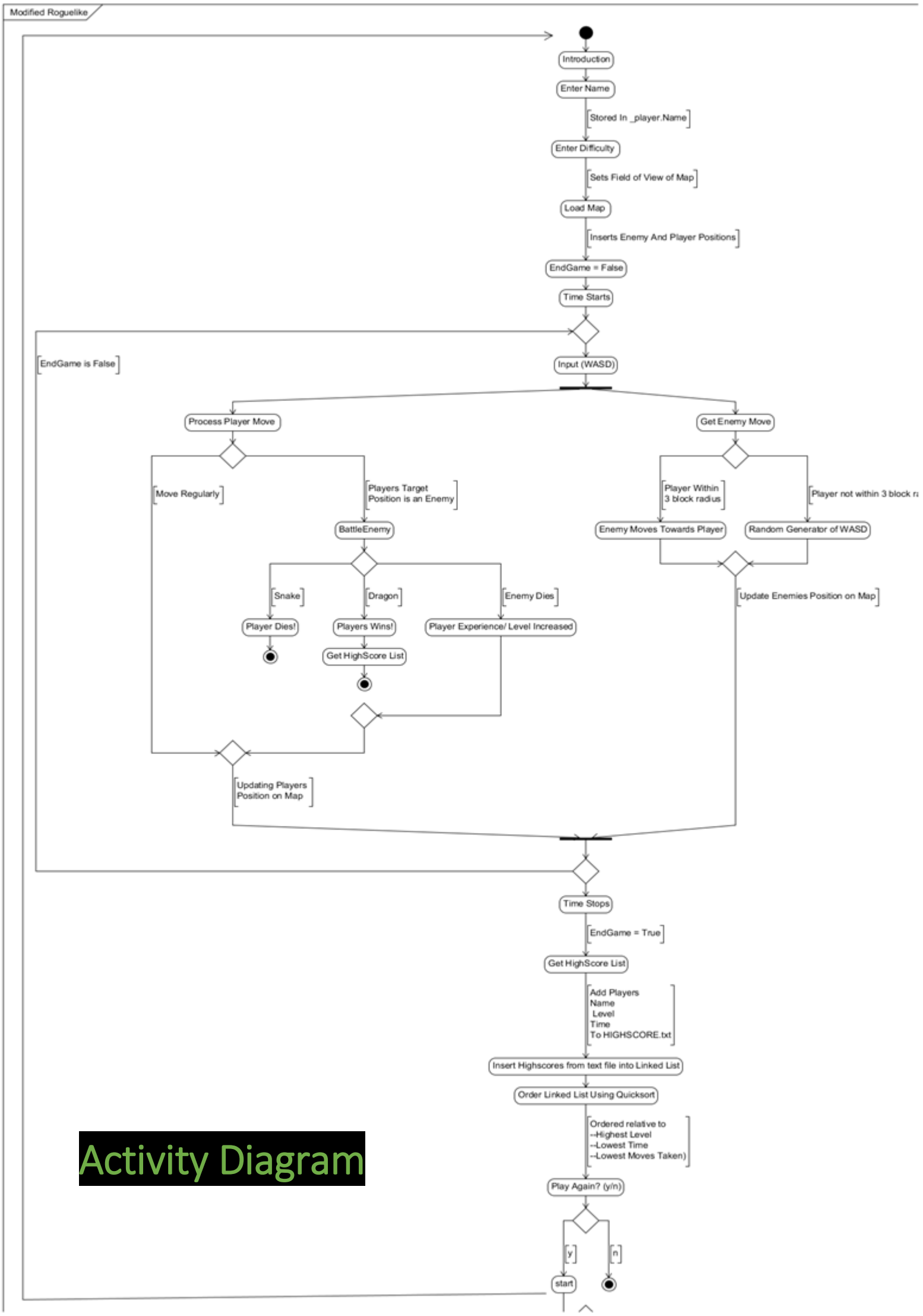
```
OH hek. U lost ur  dragon (D) find him to excape this maze.
Be vary theres nasty monsteros lurking that follow you.
If you dare come close
```

As can be seen from the introduction of the game, the objective of the game is to find your dragon so you may escape from the horrendous maze. The score of the player (Time taken, level, number of moves) are then added and displayed in a High score List.



Class Diagram




Modified Roguelike
Introduction
Enter Name
Stored In _player.Name
Enter Difficulty
Sets Field of View of Map
Load Map
Inserts Enemy And Player Positions
EndGame = False
Time Starts
EndGame is False
Input (WASD)
Process Player Move
Get Enemy Move
Move Regularly
Players Target Position is an Enemy
Player Within 3 block radius
Player not within 3 block ra
BattleEnemy
Enemy Moves Towards Player
Random Generator of WASD
Snake
Dragon
Enemy Dies
Player Dies!
Players Wins!
Player Experience/ Level Increased
Get HighScore List
Update Enemies Position on Map
Updating Players Position on Map
Time Stops
EndGame = True
Get HighScore List
Add Players
Name
Level
Time
To HIGHSCORE.txt
Insert Highscores from text file into Linked List
Order Linked List Using Quicksort
Ordered relative to
--Highest Level
--Lowest Time
--Lowest Moves Taken)
Play Again? (y/n)
y
n
start
Activity Diagram

--Gamesystem--

```
void Gamesystem::playGame() {

    cout << "\n\n\t\t OH hek. U lost ur  dragon (D) find him t
    system("pause");

    string pname;
    cout << "\t\t What is your nickname, adventurer: ";
    cin >> pname;
    _player.setPName(pname);
    setDifficulty();

    bool isDone = false;

    while (endGame() == false) {
        _level.print(_player);
        playerMove();
        _level.updateEnemies(_player);
    }

    HScores HighScores;
    HighScores.orderHScores(_player);
```

This function is Called in the Main and it runs the Game as long as long as the variable EndGame isnt true.

When the Game is ended the HighScore class is called and the list of highscores printed.

Within the main, the player is asked wether to play again or not and calls playGame over and over again until player doesn't want to play any longer.

```
void Gamesystem::playerMove() {
    char input;
    cout << "\n\n\n" << setw(30) << " Enter a move command (w/a/s/d): ";

    input = _getch();
    _level.movePlayer(input, _player);

}
```

This is another essential looping function that moves the player and updates the player on the map, the players move determines the movement of the enemies, the need for attacks, the number of moves and the change in the orientation of the map.

This function is called within playGame and sets the difficulty. The difficulty determines the field of view of the player and is used in the level class while printing the map.

The while not valid loop prevents an incorrect input to end the game completely and thus maintain stability of the game

```
bool valid = false;
while (!valid) {

    printf("\n\t\tWhat difficulty do you require?: (HARD/MEDIUM/EASY): ");
    cin >> difficulty;
    cout << endl;

    if (difficulty == "HARD" || difficulty == "hard") {
        _player.setfov(3);
        valid = true;
    }
    else if (difficulty == "MEDIUM" || difficulty == "medium") {
        _player.setfov(7);
        valid = true;
    }
    else if (difficulty == "EASY" || difficulty == "easy") {
        _player.setfov(10);
        valid = true;
    }
    else {
        printf("\n\t\tPlease enter appropriate difficulty. \n");
    }
}
```

--Level--

```
while (getline(file, line)) {
    _levelData.push_back(line);
}
```

The while loop stores each line of the map into the vector _levelData as a string even though the map is made of ASII characters.

The level is processed by the proceeding for loop that loops through each character of the string stored in the vector _levelData.

If the character at hand is an '@'

The players position is set for the player class.

The other switch statements find the relevant enemies' ASCII character and create an object of the Class enemy, initialising their respective statistic and setting their initial position.

```
//Process the level
char tile;
for (int i = 0; i < _levelData.size(); i++) {
    for (int j = 0; j < _levelData[i].size(); j++) {
        tile = _levelData[i][j];

        //name, tile, level, attack, defense, health, xp

        switch (tile) {
        case '@': //player
            player.init(1, 100, 100, 100, 0);
            player.setPosition(j, i);

            break;
        case 'S': //snake
            _enemies.push_back(Enemy("Snake", tile, 100, 2000, 2000, 200, 2000)); //
            _enemies.back().setPosition(j, i);
            break;
        case 'g':
            _enemies.push_back(Enemy("Goblin", tile, 2, 4, 4, 4, 150));
            _enemies.back().setPosition(j, i);
            break;
        case 'O' :
            _enemies.push_back(Enemy("Ogre", tile, 2, 10, 10, 10, 500));
            _enemies.back().setPosition(j, i);
            break;
        case 'D':
            _enemies.push_back(Enemy("Dragon", tile,1000, 20000, 20000, 2000, 2000));
            _enemies.back().setPosition(j, i);
            break;
        case 'B':
            _enemies.push_back(Enemy("Bandit", tile,3, 15, 10, 100, 250));
            _enemies.back().setPosition(j, i);
            break;
        }
    }
}
```

```
void Level::processPlayerMove(Player &player, int targetX, int targetY) {

    int playerX, playerY;
    player.getPosition(playerX, playerY);
    char moveTile = getTile(targetX, targetY);

    switch (moveTile) {
        // doesnt move into wall only 1 case allows to move
    case '.':
        player.setPosition(targetX, targetY);
        setTile(playerX, playerY, '.'); // previous position deleted
        setTile(targetX, targetY, '@'); // new position added
        player.numMoves++;
        break;
    case '#':
        break;
    default:
        battleMonster(player, targetX, targetY);
        break;

    }
}
```

The function MovePlayer (not shown) interprets the WASD move command. it calls the function processPlayerMove with the target position (TargetX,TargetY)

W(up), A(left), S(down), D(right) the position is updated and the @ character is moved to the target location, with the previous location being restored to '.' (empty space).

But the target position is checked if it's

- a wall '#'
- an empty space '.'
- or a monster (default since only other option)

The enemies have similar functions for interpreting movement called Update Enemies and ProcessEnemyMove (analogous to movePlayer and ProcessPlayerMove).

The for loop obtains an AI move that is a random move command generated by the Enemies getMove function. This move is then processed by the processEnemyMove function.

```
void Level::updateEnemies(Player &player) {
    char aiMove;

    int playerX, playerY;
    int enemyX, enemyY;

    player.getPosition(playerX, playerY);

    for (int i = 0; i < _enemies.size(); i++) {
        aiMove = _enemies[i].getMove(playerX,playerY);
        _enemies[i].getPosition(enemyX, enemyY);

        switch (aiMove) {
        case 'w': //up
            processEnemyMove(player,i, enemyX, enemyY - 1);
            break;
        case 's': // down
            processEnemyMove(player,i, enemyX, enemyY + 1);
            break;
        case 'a': // left
            processEnemyMove(player,i, enemyX - 1, enemyY);
            break;
        case 'd': //right
            processEnemyMove(player,i, enemyX + 1, enemyY);
            break;

        }
    }
}
```

```
    bool valid = false; // prevent terminal closing instantly
    while (!valid) {
        char input = _getch();
        switch (input) {
        case 'w':
        case 'a':
        case 's':
        case 'd':
            system("pause");
            break;
        default:
            valid = true;
            break;
        }
    }

    endGame = true;
}
system("pause");
return;
}
```

```
                                        ..................#
                                      ####################
                    You found your pet Dragon, now yo
                                            You won!
                    Level: 1                        Time take
                                Number of Moves: 72


Press any key to continue . . .
Press any key to continue . . .
Press any key to continue . . .
Press any key to continue . . .
```

This small while loop within BattleMonsters function in level prevents the player from exiting the game instantaneously when he/she was pressing either of the WASD keys repeatedly.

```
void Level::battleMonster(Player &player, int targetX, int targetY) {

    int enemyX, enemyY, attackRoll, attackResult, playerX, playerY;
    string enemyName;

    player.getPosition(playerX, playerY);

    for (int i = 0; i < _enemies.size(); i++) {

        _enemies[i].getPosition(enemyX, enemyY);
        enemyName = _enemies[i].getName();

        if (targetX == enemyX && targetY == enemyY) {
            //Battle
            attackRoll = player.attack();
            printf(" \n \t\t Player attacked %s with a roll of %d \n", enemyName.c_str(), attackRoll);
            attackResult = _enemies[i].takeDamage(attackRoll);

            if (attackResult != 0) {
                setTile(targetX,targetY, '.'); // removes enemy
                print(player);
                player.addExperience(_enemies[i].getStat('e'));
                printf("\t\t Monster died! \n");
                _enemies[i] = _enemies.back(); // removes enemy
                _enemies.pop_back();
                i--;
                system("pause");

                return;
            }

            //Enemy turn
            attackRoll = _enemies[i].attack();
            printf(" \n \t\t%s attacked player with a roll of %d \n", enemyName.c_str(), attackRoll);
            attackResult = player.takeDamage(attackRoll);

            if (attackResult != 0) {
                if (_enemies[i].getTile() == 'D') {
                    int elapsed = time(0) - timetaken;
                    //setTile(playerX, playerY, 'X');
                    print(player);
                    printf("\t\t You found your pet Dragon, now you can escape!. \n\t\t\t\t\t You won! \n");
                    cout << "\t\t Level: " << player.getStat('l');
                    cout  << "\t\t Time taken: " << elapsed << endl;
                    cout << "\t\t\t Number of Moves: " << player.numMoves << endl << endl << endl;

                    ofstream outfile;
                    outfile.open("highscores.txt", ios::app);
                    outfile << player.numMoves << "\t" << player.getPName() << "\t" << elapsed << "\t" << p
                    outfile.close();
                }

                else {
                    setTile(playerX, playerY, 'X');
                    print(player);
                    printf("\t\t You lost! \n");
                    cout << "\t\t Level: " << player.getStat('l') << endl;
                    printf("\t\t %s killed you \n", enemyName.c_str());
                    int elapsed = time(0) - timetaken;
                    cout << "\t\t Time taken: " << elapsed << endl << endl << endl;;
                }
```

The loop goes through the enemies to find the target enemy. The IF statement doesn't initiate for the other enemies.

The first battle is the players attack. The roll determines the damage taken by the target enemy. If the enemies' takeDamage function returns a 1 it means the health's below zero, hence the enemy is dead

If the players' takeDamage returns a 1 it means the player is dead.

The player is only dead in 2 cases:

- Killed by enemy
- Killed by Dragon which means we found our dragon and may escape

When the player is killed by the dragon 'D' the player wins and their moves, name, level and time are added to the HighScore List.

Otherwise the player is killed and replaced with an 'X'.

```
void Player::addExperience(int experience) {
    player_stats.setStat('e',player_stats.getStat('e') + experience);

    // Level up

    while (player_stats.getStat('e') > 140) {
        printf("\t\t Leveled up! \n");

        player_stats.setStat('e', player_stats.getStat('e') - 140);
        player_stats.setStat('a', player_stats.getStat('a') + 10);
        player_stats.setStat('d', player_stats.getStat('d') + 5);
        player_stats.setStat('h', player_stats.getStat('h') + 10);
        player_stats.setStat('l', player_stats.getStat('l') + 1);

        cout << "\t\t Level  " << player_stats.getStat('l') << endl;
    }
}
```

This is the players' mechanism of adding experience. The stat is retrieved and the relevant experience gained by killing the enemy is added on, if the total experience exceeds 140.

The experience is recalibrated and other stats of the player are increased including the level.

This Function is what gives the enemies Artificial Movement.

A random engine creates a number ranging from 1 to 6.

The distance is the difference between the enemy and the player position. If it is less than 3 then the enemy is persuaded to follow/ come closer to the player.

This is done by minimising the greatest distance. If dx is greater. Then the enemy moves either A(left) or D(right).

If dy is greater the enemy moves either W(up) or S(down).

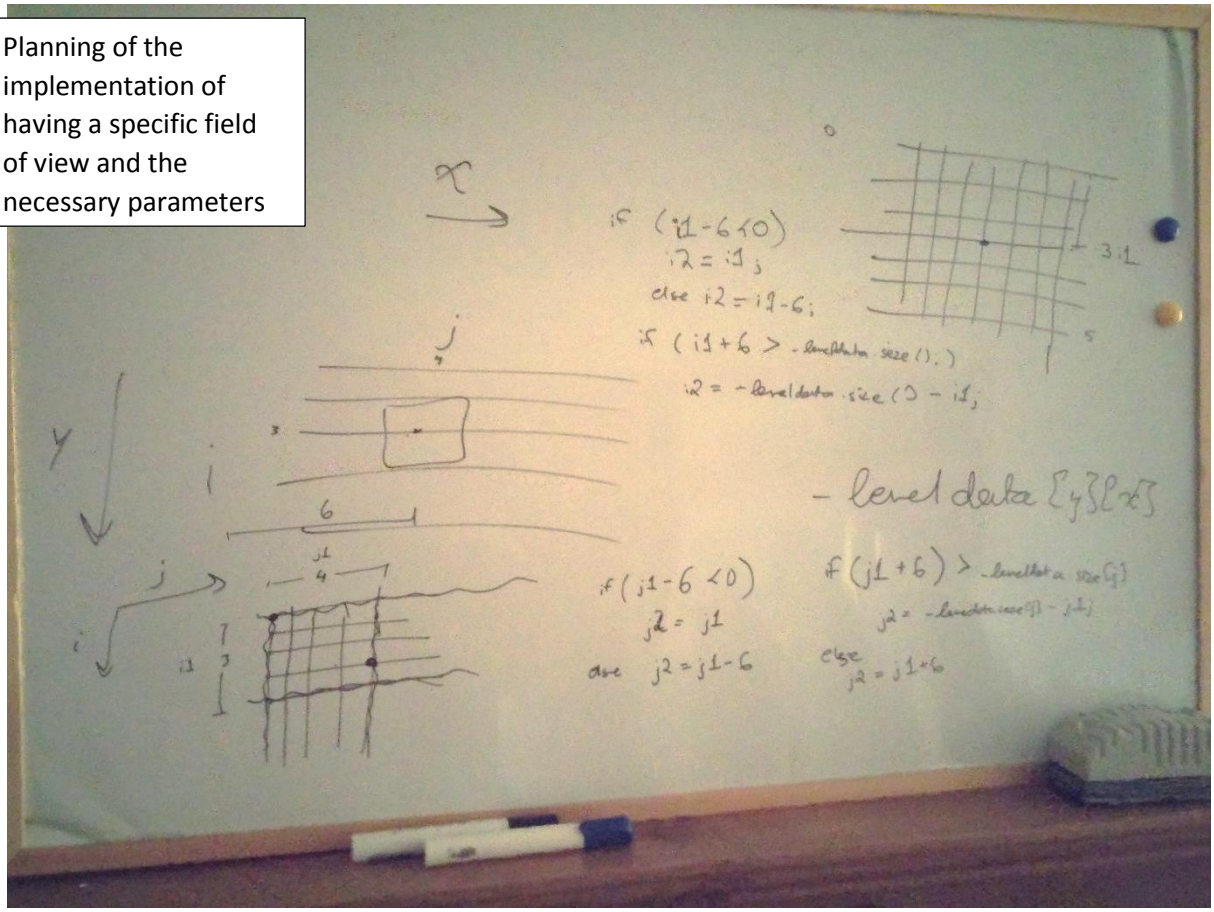
IF the player is move than 3 blocks radius away the getmove returns a rando WASD move depending on the random number generated.

This gives the enemy a 4/7 chance of moving.

```
char Enemy::getMove(int playerX, int playerY) {

    static default_random_engine randomEngine(time(NULL));
    uniform_int_distribution<int> moveRoll(0,6);

    int dx = _x - playerX;
    int dy = _y - playerY;
    int adx = abs(dx); //abs slow function therfore stored in a variable
    int ady = abs(dy);
    int distance = adx + ady;

    if (distance <= 3) {
        //Moving along x axis
        if (adx > ady) {
            if (dx > 0) {
                return 'a';
            }
            else
                return 'd';
        }
        else {
            //Move along y axis
            if (dy > 0) {
                return 'w';
            }
            else
                return 's';
        }
    }
    //when distance isnt less than 5, move randomly
    int  randMove = moveRoll(randomEngine);

    switch (randMove) {
    case 0 :
        return 'a';
    case 1 :
        return 'w';
    case 2:
        return 's';
    case 3:
        return 'd';
    default:
        return '.'; // dont move 4/7 chance in moving, 2/7 chance staying still
    }
}
```

--Setting Field Of View--

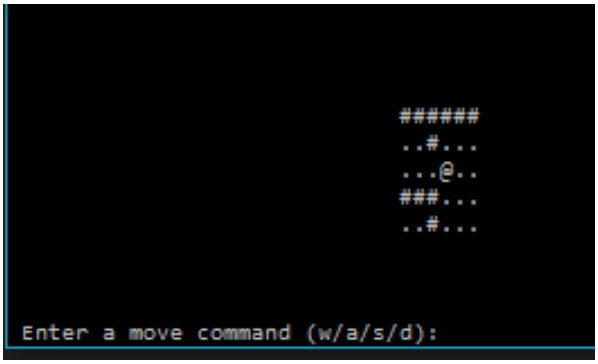
Planning of the implementation of having a specific field of view and the necessary parameters



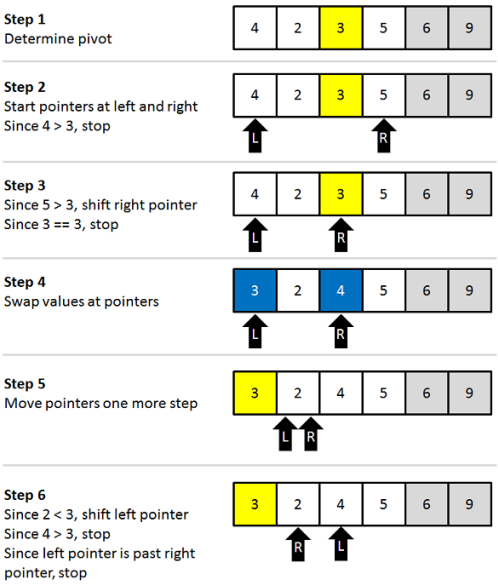

```
void Level::print(Player &player) {
    int i, j, i1, j1, minj2, maxj2, mini2, maxi2, fov;

    fov = player.getfov();

    player.getPosition(j1, i1); // position of player

    if (j1 - fov < 2) // ensuring fov isnt outside text file
        minj2 = 0;
    else minj2 = j1 - fov;

    if (j1 + fov > _levelData[i1].size())
        maxj2 = _levelData[i1].size();
    else maxj2 = j1 + fov;

    if (i1 - fov < 2)
        mini2 = 0;
    else mini2 = i1 - fov;

    if (i1 + fov > _levelData.size())
        maxi2 = _levelData.size();
    else maxi2 = i1 + fov;

    cout << string(100, '\n');

    for ( i = mini2; i < maxi2; i++) {
        cout << setw(30);
        for (j = minj2; j < maxj2; j++) {
            cout << _levelData[i][j];
        }
        cout << endl;
    }

}
```

The field of view is set by the difficulty selected at the start of the game.

Instead of the print function printing the whole updated map after each input command. The function only outputs the fov (field of view). By obtaining the current position of the player and creating the minimum and maximums of the [i] and [j] while printing the map. At the same time preventing the [i] or [j] min/max to be outside map limitations.



--Quicksorting The HighScore list--



Initially the unordered linked list is created including the recent players' stats.

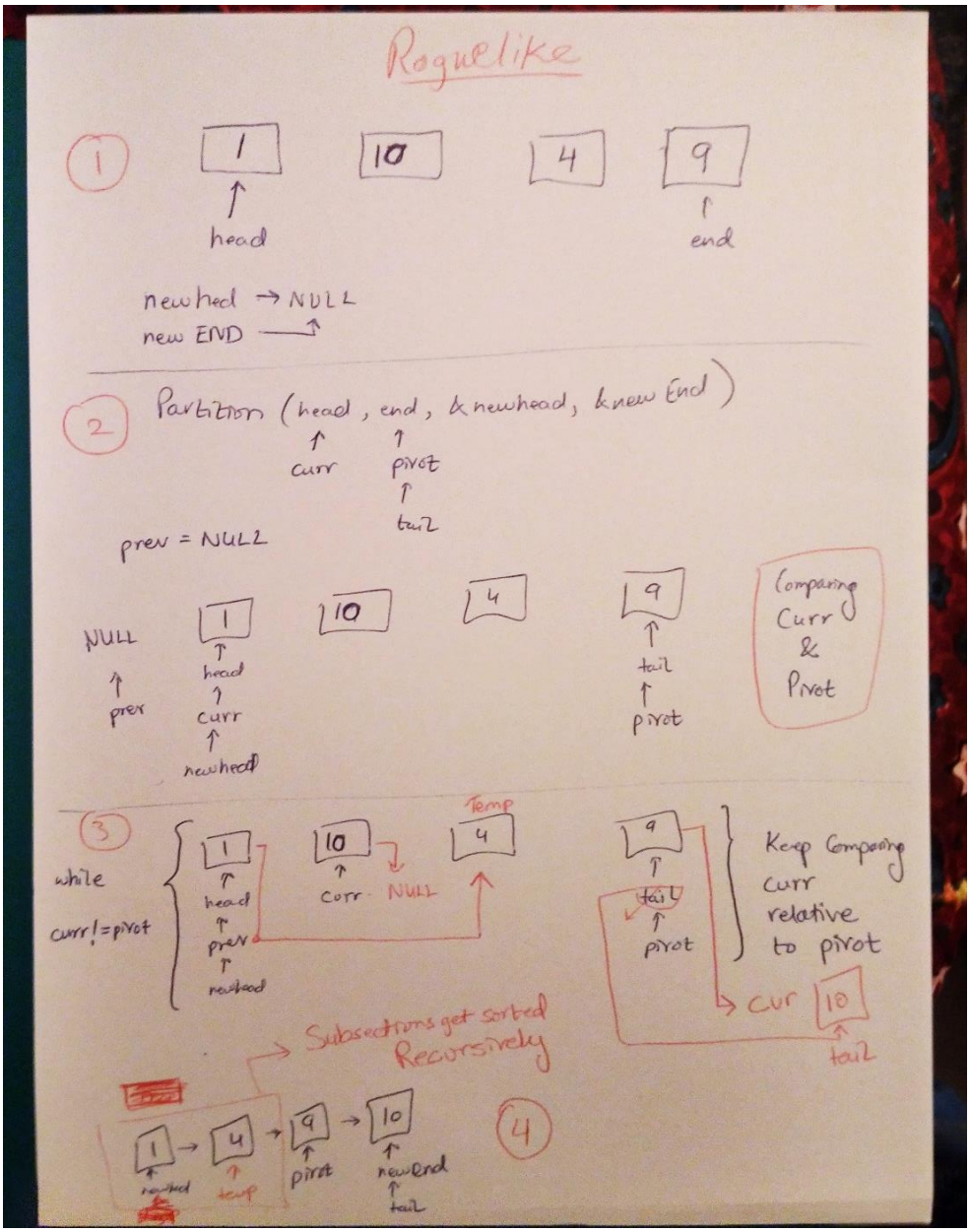
The quick-sorting mechanism consists of 3 main functions:

QuickSort -

Starts the process by calling the recurring function QuickSortRecur and passing the head and tail of the unordered list.

QuickSortRecur –

Provides the section of the list to be partitioned.



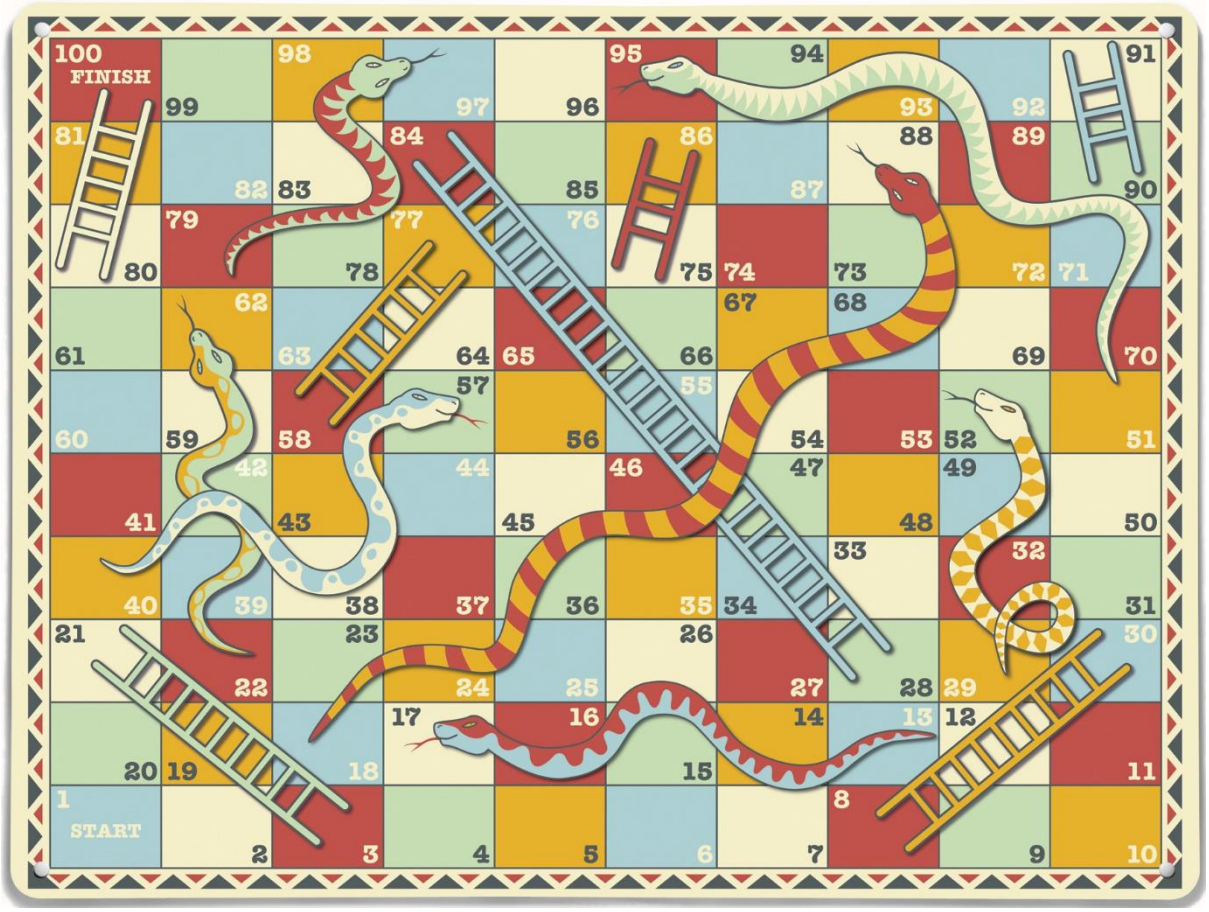
Partition –

Orders the player Nodes relative to the pivot node and provides the newhead and newend to QuickSortRecur. Which is the new subsection to be partitioned and quicksorted further.

Aim was to order from shortest to longest time. Therefore to incorporate a higher level as part of a better score the magnitudes of the number of moves and the inverse of the level (1/level) was used to order the list as a higher level results in a lower inverse.

--Section C--

The new game I have created is snakes and ladders (not snake, to your relief), this game is a childhood classic and its activity is similar to my modified Roguelike since it will contain a map which I can read and assign the different objects (snakes, ladders, and players) their relative characteristics.



Picture take from ipoem.files.wordpress

Snakes and Ladders can be played by two or more players on a game board having numbered, gridded squares. A number of “ladders” and “snakes” are dispersed across the board squares. The objective of the game is to navigate through the board according to die rolls, from start to finish helped by ladders or hindered by snakes.

The game is a simple race based on sheer luck, it had roots with morality lessons, where progression represented a life journey accompanied by virtues (ladders) and vices (snakes).

Game

The Game class can have the methods of the level in my previous game as well, so it stores the map in a vector of strings, which can be used to access each character to process and initialise the ladders, snakes, the first and the last square. The map can be stored into a dynamic 2-d array as well and using it would aid in the arithmetic side of computing the proceeding position (after die roll).

The Game constructor must only process the level and Players. Hence the main only needs to create a game. The parameters required would be the string of names (Players), the number of players on the map and the array to be manipulated into a map.

Play

This method in Game allows the game to continuously run and has the ability to use the notOver variable and return it to main when player has won and the Game is over. The player won can be added to the Winner.

Not possible in Constructor since constructors can't be used for return.

Dice

Produces a random number ranging from 1 to 6. Implemented using the same function used for the AI movement of the enemies (Previous Game). Instead the engine has the parameter (1,6) instead of (0,6) to mimic the outcome of a die roll.

```
static default_random_engine randomEngine(time(NULL));
uniform_int_distribution<int> moveRoll(0,6);
```

Player

The game can have multiple players that will be determined by their name. The game can ask the user the amount of players playing and we can create a vector of strings that can store all the players. Each player has a position which is manipulated by the Square class. Only one player can win.

Square

Each square can be empty, a ladder, a snake, First Square, or the Last Square. Given a dice roll this class should be able to move the player and assesses and set the next position of the player. And allows 2 or more players to have the same position.

First Square

Knows it can have zero, one or more players. Is identified by a certain character on map text file to determine the start of the map.

Last Square

Used to determine the end of the GameBoard. It has the Boolean function isOccupied() to checkWin() and endgame to pass on to the Game class.

Ladder/Snake

Knows that a player landed on it has to be moved to a new position. Contains a start and end position. Each object will have to main positons, the start and end. The start of the ladder is the bottom part and the end is the top part. Whereas for the snake, the start is the head (top) and bottom is the tail (bottom).

To find the relative part of the ladder we can search for any neighbouring **#** around an already found **#.** If one exists ONLY at a n [i] and/or[j] positon lower than the current **#** being looked at, we consider it a top part,

If one exists ONLY at an [i] and/or[j] positon higher than the current # being looked at, we consider it a bottom part.

Similar logic can be applies for snake represented as **S** on map.

The variables transport can be the boost or hindrance the respective object inflict on the Player. The sum of the transports can be displayed for all the players at the end of the Game to demonstrate the luck of each person.

Class Diagram II