CheckerBot

December 6, 2021

Sean Whitty
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 1-Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Project Plan Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 2-Customer Needs, Requirements, and Specifications</td>
<td>3</td>
</tr>
<tr>
<td>Customer Needs Assessment</td>
<td>3</td>
</tr>
<tr>
<td>Requirements and Specifications</td>
<td>4</td>
</tr>
<tr>
<td>Marketing Requirements</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 3-Functional Decomposition</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 4-Project Planning</td>
<td>7</td>
</tr>
<tr>
<td>Cost Estimates</td>
<td>7</td>
</tr>
<tr>
<td>Chapter 5 - Implementation and Results</td>
<td>8</td>
</tr>
<tr>
<td>Works Referenced</td>
<td>11</td>
</tr>
<tr>
<td>Appendix</td>
<td>12</td>
</tr>
<tr>
<td>Code</td>
<td>14</td>
</tr>
</tbody>
</table>
Abstract

This product is able to play a game of checkers against the customer at three different preset difficulties. It can process the board on its turn, and displays its next turn to its opponent by updating the board state via the LEDs. It consists of a laptop screen display, and a computer chip with the programmed ability to process the checkers board and decide its next move. It provides the customer with an opponent that can be modified in difficulty so it can provide however much of a challenge as the customer desires, as well as allowing for a 2 player experience if the customer choses.

Chapter 1-Introduction

Project Plan Introduction

The idea for this project came from wanting to combine a passion from simple board games and programming. Upon doing research into the topic, I found out that artificial intelligence had been programmed to play checkers as early as 1959 [1]. Researchers are using simple board games like checkers to test out implementations and theories related to Artificial Intelligence and see how they work in a controlled environment [2,6,3,5]. After seeing that A.I. can be programmed on an MCU that I am familiar with [4], I decided that I wanted to try and create my own version. Making the A.I. is only part of the project, however, as the bot needs to have a way to show the board so the customer knows where the pieces are. I built a board using LEDs to show the current board state. I originally had planned to make the project without its own connected board, and instead take a picture of an already existing checkers board, process the move, and then display it to the player. I moved away from this idea as it placed many constraints on how and where the game could be played.

Chapter 2-Customer Needs, Requirements, and Specifications

Customer Needs Assessment

The product is conveniently sized so that it is not too cumbersome to move around and set up. It makes its decisions on its next move fast enough to not slow down the pace of the game. It has enough difficulty settings that it will always provide a sufficient challenge to the customer. Ideally, the product creates a game of checkers that feels like playing against an opponent that matches the customer’s skill level. It also offers a person versus person experience if two people are willing to play.
Requirements and Specifications

To come up with the requirements and specifications for this project, I thought about what a typical person would want in a person vs. CPU experience. The customer would want this to be convenient and as challenging as playing a normal opponent. Convenience is very subjective, but since checkers have been around unchanged for centuries, I assume that people are willing to put up with the standard size and weight of a checkers board. It also must be able to compute its next turn within a certain amount of time to avoid forcing the user to wait for the computer to move and wasting their time. It must also be able to play with at least a few varying difficulties, so as to provide the product with a bit of variety, since playing the same virtual opponent over and over would grow stale.

<table>
<thead>
<tr>
<th>Marketing Reqs.</th>
<th>Engineering Specifications</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>Processes board and update board state within 5 seconds</td>
<td>5 seconds is a rough estimate of the average time for a human opponent to make a move. Any longer and the customer may start to lose interest as they wait.</td>
</tr>
<tr>
<td>1, 2</td>
<td>has at least 3 difficulties (easy, medium, hard)</td>
<td>Providing multiple difficulty levels ensure that the customers will always have a challenge</td>
</tr>
<tr>
<td>3, 4</td>
<td>Weighs less than 5 lbs, and takes up no more than a square foot of space</td>
<td>The final board is roughly (12.5” x 11.5” x 3.5”), which is smaller than a normal checkers board in length and width, but taller by about an inch. These differences are due to the size of the electrical components.</td>
</tr>
<tr>
<td>2, 3</td>
<td>Can be used in normal conditions that a checkers game would be played in.</td>
<td>Must be able to work in normal household conditions, or it cannot be used by a wide audience</td>
</tr>
<tr>
<td>3, 4</td>
<td>Costs no more than $100</td>
<td>being too expensive for a normal leisure item creates disinterest. A typical checkers board costs ~$30, with higher quality boards being closer to ~$50, adding the price of the MCU to this accounts for this price</td>
</tr>
</tbody>
</table>

Table 1-This table lists the 5 engineering specifications of this project, as well as the reason for each, and which marketing requirement it helps satisfy
Marketing Requirements

1. Must process the game fast enough to not slow down the overall pacing
2. Must provide an immersive experience that effectively simulates playing versus a normal opponent
3. Must not be as easy to store as a normal checkers board, despite the fact that it contains more fragile electrical components.
4. Pricing must be competitive with other household entertainment items, such as board games and small computer systems

Chapter 3-Functional Decomposition

Level 0

**Fig. 1**-This image shows the diagram of the level 0 system, with all of its planned inputs and outputs

<table>
<thead>
<tr>
<th>Input</th>
<th>The main source of power for this project will be a 5 volt DC power cable connected to the Arduino board from a laptop. The checkerboard moves will be input into the laptop via the keyboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>The system will output a move by updating the board, and changing whatever LED lights are necessary.</td>
</tr>
<tr>
<td>Functionality</td>
<td>The system will initialize the LED lights, showing the user the board state. When the user inputs their move, the computer will update the board, and make a move of its own.</td>
</tr>
</tbody>
</table>

**Table 3**-This table goes in-depth about each input and output and how they contribute to the system
Level 1

Fig. 2-This figure shows the level 1 of the project system, naming each control system and what it is receiving and producing for the final system output

<table>
<thead>
<tr>
<th>Input</th>
<th>The main source of power for this project will be a 5 volt DC power cable connected to the arduino board from a laptop. The user will input their turn via the laptop keyboard, by pressing the corresponding letters and numbers into the arduino serial display.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Display of board using LEDs.</td>
</tr>
<tr>
<td>Functionality</td>
<td>The system will set up the board with LED lights. Showing the user what the board state is, when the user inputs their move, the computer will then update the board, and then make a move of its own.</td>
</tr>
</tbody>
</table>

Table 4-This table provides more information on the inputs and outputs previously mentioned by the level 0 table and figure

Chapter 4-Project Planning
Cost Estimates

<table>
<thead>
<tr>
<th>Part</th>
<th>Cost Estimate</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arduino Mega Rev3</td>
<td>$40</td>
<td>This board contains the code to run both the game of checkers, and the program used to calculate the computers move.</td>
</tr>
<tr>
<td>Wood (3.5” x 40” x 0.75”)</td>
<td>$10</td>
<td>Used to make the box that holds the board, I purchased a single plank, and then cut it down to fit all the electrical components inside.</td>
</tr>
<tr>
<td>Wires, tape</td>
<td>$22</td>
<td>Wires used to set up the system with its LEDs. The tape was used over the glass so that the physical appearance of a checkers board could be obtained.</td>
</tr>
<tr>
<td>Glass + frost</td>
<td>$18</td>
<td>Glass was put on top of the box so that the LED light can shine through, it was painted over with frost.</td>
</tr>
<tr>
<td>Labor</td>
<td>150 hour minimum $35 dollars per hour $5,250 total</td>
<td>Upon researching the hourly wage of a starting Electrical Engineer, I found it to be $35[9]</td>
</tr>
</tbody>
</table>

Table 5-This table shows the expected costs for all of the materials and parts that will be needed to complete this project, with an explanation of what each item will do

Chapter 5 - Implementation and Results

To go about creating this software, the first step was creating a game of checkers. The first few months were dedicated to creating a way to fit the rules of checkers on an arduino board's memory: first just a simple way to move pieces forward, then on to harder concepts like
kings and double jumps. All communication between the player and the board happens through the Serial interface. The arduino prints out a statement asking the user a question, such as “what is the row of the piece you want to move”. Then the player enters in the given information, and the next question is asked, and so on. It does take some getting used to, since we are typically able to either move pieces with our hands or with a mouse click, which I unfortunately didn’t have enough time to implement. But after playing only a few practice games, I was easily able to make my moves without looking at the labels I had made to remember what each row and column was.

Once I had implemented the game, I began building the board itself. The wiring required for this board is decently complex, since I am using 3 prong LEDs to allow for three different colors, red, green, and yellow(which is just red and green combined). This meant that each LED would require at least 2 wires and 1 connection to ground to work properly. The reason a dual color LED would be needed was because of the fact that every piece would need to be able to display 4 states: off, green, red, and king. Yellow was used as the color of kings, and it will switch back and forth to display the active kings of whoever’s turn it is.

After constructing a basic prototype of the board's wiring, I implemented the computer opponent. When researching different algorithms to use for the computer turn decisions, the one that seemed to work the best for these types of games was the minimax search algorithm [10]. This algorithm is known for being very effective at searching for moves in two player games, and it works by recursively searching through all possible outcomes of each turn, assuming each player will pick the best move they can, and optimizing the outcome in their favor. However, I started running into memory shortages after only a few layers of depth into my searches. Arduino also struggles to return 2d array data types from functions, which is what I was using to store the board states that the functions were searching through. After looking into some other search algorithms like Monte Carlo Tree Search, I concluded that minimax was still the best option, it was simply going to be limited in its power.

When I began implementing the computer difficulties, I made the lowest difficulty just a random move by the opponent. I decided it was a good starting place, especially since this is how most search tree algorithms are initially measured, by how consistently they beat a random opponent. For the next difficulty I implemented a single depth search into a minimax algorithm, so it tries to maximize its most immediate turn. This ensures that it will not miss any obvious jumps, but it can still make bad decisions. The final difficulty was implemented by doing a minimax search with a depth of 3. The coding for this was challenging, because recursively editing a list in arduino code proved to be very inconsistent, with data sometimes being rewritten for seemingly no reason. In the end this forced me to turn away from recursive calling. However, this difficulty is still capable of looking 2 turns ahead, and is able to set up and make good plays.

The Final product is able to properly play a person versus person game of checkers, and a computer opponent should the user select it. It requires the user to have a laptop as a power source and a keyboard to input their move. There were several setbacks while implementing this
project, the most notable being the creation of the function to choose the computer players move, which I talked about previously. Despite this I am very happy with the final product. Shown below are the pictures of both the inside and outside of the device, as well as what it looks like when it is powered on.
Works Referenced


Appendix

Summary of Functional Requirements
This project provides the customer with an A.I. capable of playing the game of checkers at a variety of different difficulty levels. It will be able to calculate its next move, and then update the board accordingly.

Primary Constraints
The hardest part of coding this project was trying to do it with the simplicity of the arduino language. Even simple things like returning a list, or using global variables were made extremely difficult by the beginner-friendly nature of this coding language. It is also decently fragile in terms of wiring, since there are so many wires to keep track of, it could be hard to spot a potential burnout.

Economic
This project is designed to be a luxury item, much like the board game it revolves around. It has a low cost to run and maintain, since it can be powered by a simple smartphone charger. As stated in the marketing requirements, it is intended to cost less than 100 dollars to produce. Which is reasonable when considering the cost estimate of all the necessary parts added up to 65 dollars before accounting for labor and manufacturing. The resources necessary for this project are all very cheap and common, with the most expensive part being the MCU that it will run on. Given that all parts of the project are reliant on a constant source of power when operating and
not a battery, there is not a large focus on maintenance and upkeep. The product should typically be able to function properly for at least 4 or 5 years, and the first part that is expected to fail would be the display screen.

**Environmental**
This product is mostly going to be made up of common metals, plastic, and glass. The electronic components make up the majority of the waste created when this product’s life cycle ends, and it will need to be disposed of properly. Luckily these components, as well as the glass used on top of the box, are all recyclable. However there is also a good amount of plastic that is harmful to the environment as it cannot be recycled. Since this is intended to be a luxury item, with a decently small market, its manufacturing impact would be marginal, as it wouldn’t require a large number to be produced consistently. However, the manufacturing that it would require still has an effect on the environment, mostly in the exhaust from the machines used on a typical factory floor.

**Manufacturability**
If this product were to be manufactured, it would need to be modified to be more cost efficient. Since it’s current design relies on buying pre-made parts from other sources and implementing them together. The main part that could probably be modified is the MCU chip, since there is a lot of functionality provided by the chip that is not going to be used for this project. Whether the solution is designing a chip separately or finding one that can perfectly fit the needs of the product, it could significantly lower the production cost.

**Sustainability**
This product requires very little in terms of maintenance, since it runs off of a plugged in 12 V power supply and a MCU and associated parts that can have a lifetime of up to 10 years if they are kept in cool temperatures. It could be made more sustainable by creating a better system for seeing the board, as the one I have currently implemented is decently rudimentary, with a hard to estimate expiration date.

**Ethical**
The intended use of this product has only a few ethical implications. It is a form of entertainment that is enjoyed alone, which can encourage isolation and loneliness. Building a reliance on any form of entertainment is generally unhealthy, and it would be unethical to encourage use of this product to this extreme. The manufacturing of this product would have to ensure it follows all guidelines to ensure that no one involved in the production is operating in unethical working conditions. This product also could help to normalize the idea of A.I. intelligence in the common home, as it is an emerging technology, and will likely soon become a part of everyday life for many first world countries.
Health and Safety
Using this product could positively or negatively affect the consumers mental health. It could refresh them as they enjoy a challenge in a fun and engaging game, or it could encourage isolation, as previously mentioned. Since it is running on 5 V when it is plugged in, there is the risk that it could shock a customer if handled improperly. There could also be cuts and bruises if the product is broken up, and sharp edges from glass or metal could prove to be a safety concern. There are similar safety concerns during the manufacturing of this product. During testing it would need to be powered up to see if it is functioning properly, and a malfunction could electrocute a worker who isn’t careful.

Social and Political
If this product became successful, it could be seen as an advancement in the tech world. It would prove that there is a market for A.I. used for entertainment, and could greatly affect how people view A.I. and policies concerning it. It directly affects the producers of products that can be considered its competition for household entertainment systems, as it is now another product fighting for a share of that market. It was previously mentioned that this product could promote isolation, as you no longer need to search for a partner to entertain yourself. This would have social ramifications if this problem happened on a large enough scale. Tech companies that are researching uses for A.I. and computer visions would benefit the most from this product, as the idea that A.I. is dangerous would be lessened by social groups seeing a more mundane use of its potential.

Development
I enjoyed learning about implementations of A.I., since it is a growing market and most likely will be one of the more important technological advancements in our lifetime. This paired with computer vision are both subjects that I find very interesting, and am happy that I was given the chance to learn more about them while doing my research. I also got more experience coding for a larger scale project, which I hope to use in the future as I would enjoy working in the field of computer engineering in the future.

Code

```c
```
int Board[] =
{1, 1, 1, 1,
1, 1, 1, 1,
1, 1, 1, 1,
0, 0, 0, 0,
0, 0, 0, 0,
2, 2, 2, 2,
2, 2, 2, 2,
2, 2, 2, 2};

int King[] =
{0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0};

int AIMove[] =
{0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0};

int zero[] =
{0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0};

int best_move[] =
{0, 0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
0, 0, 0, 0,
//int *p = best_move;
//int boardCount = 0;
int PorC = 0;
int Diff = 0;
int player1PieceNum = 12;
int player2PieceNum = 12;
int nextPieceNum = 0;
int currentRow = 0;
int currentCol = 0;
char currentDir = 'l';
char currentUorD = 'u';
int Player = 1;
int check = 0;
int PieceList[12];
int JumpList[24];

void printBoard(int board[]){
    int x = 0;
    while (x<32){
        Serial.println(board[x]);
        x++;
    }
}

int findRating(int board[], int king[], int player){
    int opponent;
    if (player == 1){
        opponent = 2;
    }else{
        opponent = 1;
    }
    int P = 0;
    int O = 0;
    int count = 0;
    while(count < 32){
        if (board[count] == player){
            if (count != 0 & count != 7 & count != 8 & count != 15 & count != 16 & count != 23 & count != 24 & count != 31){
                // Add code here
            }
        }
    }
}

```c
P++;
} P+=3;
if(king[count] == player){
P+=7;
}
}
if (board[count] == opponent){
    if (count != 0 & count != 7 & count != 8 & count != 15 &
count != 16 & count != 23 & count != 24 & count != 31){
P++;
}
O+=3;
if(king[count] == opponent){
    O+=7;
}
}
count++;
}
return P-O;
}

void copy(int* src, int* dst, int len){
    for (int i = 0; i < len; i++){
        dst[i] = src[i];
    }
}

int minimax3(int board[], int king[]){
    int bestList[100][4] = {
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
    };
```
```c
int bestList2[10][4] = {{0,0,0,0}, {0,0,0,0}, {0,0,0,0},
                        {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
                        {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
                        {0,0,0,0}, {0,0,0,0}, {0,0,0,0},
                        {0,0,0,0}, {0,0,0,0}, {0,0,0,0}};
int bestList3[4] = {0,0,0,0};
int possibleBoards[10][32];
int possibleBoardsK[10][32];
int possibleBoards2[10][32];
int possibleBoardsK2[10][32];
int possibleBoards3[10][32];
int possibleBoardsK3[10][32];

int count = findPossibleMoves(board, king, possibleBoards,
                              possibleBoardsK, 2);
//  Serial.println(bestList3[2]);
Serial.println("count");
Serial.println(count);
int countsave = count;
int countsave2 = 0;
int countsave3 = 0;
int z = 0;
//printBoard(possibleBoards[3]);
  //int bestList[100][4];
  int bestListCount = 0;
while (count >= 0){
  int count2 = findPossibleMoves2(possibleBoards[count],
  possibleBoardsK[count], possibleBoards2, possibleBoardsK2,
  1);//need to change player
  int countsave2 = count2;
  while (count2 > 0){
  int count3 = findPossibleMoves3(possibleBoards2[count2],
  possibleBoardsK2[count2], possibleBoards3, possibleBoardsK3, 2);
    int countsave3 = count3;
    while (count3 >= 0){
```
int score = findRating(possibleBoards3[count3], possibleBoardsK3[count3], 2);
if(score >= bestList[bestListCount][2]){    
    bestList[bestListCount][0] = 3;
    bestList[bestListCount][1] = count2;
    bestList[bestListCount][2] = score;
    bestList[bestListCount][4] = 2;
}
    bestListCount++;
    count3--;
}
    count2--;
}
Serial.println(bestList3[2]);
int x = 0;
int y = 0;
while (x < countsave2){
    for(int i = 0; i < bestListCount; i++){
        if (bestList[i][1] == x && bestList[i][2] <= bestList2[x][2]){
            bestList2[x][0] = 2;
            bestList2[x][1] = count;
            bestList2[x][2] = bestList[i][2];
            bestList2[x][3] = 1;
        }
    }
    x++;
}
    count--;
}
z = 0;
while (z < savecount){
    Serial.println("start");
    printBoard(possibleBoards[z]);
    z++;
}
int y = 0;
while (y < countsave){
    if (bestList2[y][2] >= bestList3[2]){    
        bestList3[0] = 1;
        bestList3[1] = y;
        bestList3[2] = bestList2[y][2];
    }
bestList3[3] = 2;
}
y++;
}
copy(possibleBoards[bestList3[1]], AIMove, 32);
return bestList3[1];
}

int minimax1(int board[], int king[], int player){
    int possibleBoards[10][32];
    int possibleBoardsK[10][32];
    int count = findPossibleMoves(board, king, possibleBoards,
    possibleBoardsK, player);
    int best[2] = {0, 0}; // score, then index of board with score
    while (count >= 0){
        int score = findRating(possibleBoards[count],
                        possibleBoardsK[count], player);
        if (score >= best[0]){
            best[0] = score;
            best[1] = count;
        }
        count--;
    }
    //Serial.println(best[1]);
    //printBoard(possibleBoards[best[1]]);
    copy(possibleBoards[best[1]], AIMove, 32);
    copy(possibleBoardsK[best[1]], best_move, 32);
    return best[1];
}

int findPossibleMoves2(int board[], int king[], int
possibleBoards1[][32], int possibleBoardsK1[][32], int player){
    int boardCount = 0;
    findPieces(board, king, player);
    int i = 0;
    boardCount = 0;
    while(i < 12){
        if (PieceList[i] != -1){
            int newBoard1[32];
            int newBoard2[32];
int newBoard3[32];
int newBoard4[32];
int newKBoard1[32];
int newKBoard2[32];
int newKBoard3[32];
int newKBoard4[32];
copy(board, newBoard1, 32);
copy(board, newBoard2, 32);
copy(board, newBoard3, 32);
copy(board, newBoard4, 32);
copy(king, newKBoard1, 32);
copy(king, newKBoard2, 32);
copy(king, newKBoard3, 32);
copy(king, newKBoard4, 32);
int Piece = PieceList[i];
int row = Piece/4;
int col = Piece - 4*row;
if ((row+2)%2 == 0){//even row(includes 0)
    col = col*2;
}else{//odd row
    col = (col+1)*2-1;
}
if (king[Piece] == player){
    if (checkMoveKing(newBoard1, newKBoard1, row, col, player,
        'l', 'u', 0, 0, 1)){
        copy(newBoard1, possibleBoards1[boardCount], 32);
        copy(newKBoard1, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard2, newKBoard2, row, col, player,
        'r', 'u', 0, 0, 1)){
        copy(newBoard2, possibleBoards1[boardCount], 32);
        copy(newKBoard2, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard3, newKBoard3, row, col, player,
        'l', 'd', 0, 0, 1)){
        copy(newBoard3, possibleBoards1[boardCount], 32);
        copy(newKBoard3, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
}
```c
if (checkMoveKing(newBoard4, newKBoard4, row, col, player, 'r', 'd', 0, 0, 1)) {
    copy(newBoard4, possibleBoards1[boardCount], 32);
    copy(newKBoard4, possibleBoardsK1[boardCount], 32);
    boardCount++;
} else {
    if (checkMove(newBoard1, newKBoard1, row, col, player, 'l', 0, 0, 1)) {
        copy(newBoard1, possibleBoards1[boardCount], 32);
        copy(newKBoard1, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMove(newBoard2, newKBoard2, row, col, player, 'r', 0, 0, 1)) {
        copy(newBoard2, possibleBoards1[boardCount], 32);
        copy(newKBoard2, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
}
}
i++;
if (boardCount == 13) {
    i = 20;
}
boardCount--;
return boardCount;
}

int findPossibleMoves3(int board[], int king[], int possibleBoards1[][32], int possibleBoardsK1[][32], int player) {
    int boardCount = 0;
    findPieces(board, king, player);
    int i = 0;
    boardCount = 0;
    while (i < 12) {
        if (PieceList[i] != -1) {
            int newBoard1[32];
            int newBoard2[32];
            int newBoard3[32];
            int newBoard4[32];
            int newKBoard1[32];
```
int newKBoard2[32];
inKBoard3[32];
inKBoard4[32];
copy(board, newBoard1, 32);
copy(board, newBoard2, 32);
copy(board, newBoard3, 32);
copy(board, newBoard4, 32);
copy(king, newKBoard1, 32);
copy(king, newKBoard2, 32);
copy(king, newKBoard3, 32);
copy(king, newKBoard4, 32);
int Piece = PieceList[i];
int row = Piece/4;
inKBoard4[32];
col = Piece - 4*row;
if ((row+2)%2 == 0){//even row(includes 0)
    col = col*2;
} else {//odd row
    col = (col+1)*2-1;
}
if (king[Piece] == player){
    if (checkMoveKing(newBoard1, newKBoard1, row, col, player,
        'l', 'u', 0, 0, 1)){
        copy(newBoard1, possibleBoards1[boardCount], 32);
        copy(newKBoard1, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard2, newKBoard2, row, col, player,
        'r', 'u', 0, 0, 1)){
        copy(newBoard2, possibleBoards1[boardCount], 32);
        copy(newKBoard2, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard3, newKBoard3, row, col, player,
        'l', 'd', 0, 0, 1)){
        copy(newBoard3, possibleBoards1[boardCount], 32);
        copy(newKBoard3, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }if (checkMoveKing(newBoard4, newKBoard4, row, col, player,
        'r', 'd', 0, 0, 1)){
        copy(newBoard4, possibleBoards1[boardCount], 32);
int findPossibleMoves(int board[], int king[], int possibleBoards1[][32], int possibleBoardsK1[][32], int player){
    int boardCount = 0;
    findPieces(board, king, player);
    int i = 0;
    boardCount = 0;
    while(i < 12){
        if (PieceList[i] != -1){
            int newBoard1[32];
            int newBoard2[32];
            int newBoard3[32];
            int newBoard4[32];
            int newKBoard1[32];
            int newKBoard2[32];
            int newKBoard3[32];
            int newKBoard4[32];
            copy(board, newBoard1, 32);
            copy(newKBoard4, possibleBoardsK1[boardCount], 32);
            boardCount++;
        }else{
            if (checkMove(newBoard1, newKBoard1, row, col, player, 'l',
                          0, 0, 1)){
                copy(newBoard1, possibleBoards1[boardCount], 32);
                copy(newKBoard1, possibleBoardsK1[boardCount], 32);
                boardCount++;
            }
            if (checkMove(newBoard2, newKBoard2, row, col, player, 'r',
                          0, 0, 1)){
                copy(newBoard2, possibleBoards1[boardCount], 32);
                copy(newKBoard2, possibleBoardsK1[boardCount], 32);
                boardCount++;
            }
        }
        i++;
        if (boardCount == 13){
            i = 20;}
    }
    boardCount--;  
    return boardCount;
}
copy(board, newBoard2, 32);
copy(board, newBoard3, 32);
copy(board, newBoard4, 32);
copy(king, newKBoard1, 32);
copy(king, newKBoard2, 32);
copy(king, newKBoard3, 32);
copy(king, newKBoard4, 32);
int Piece = PieceList[i];
int row = Piece/4;
int col = Piece - 4*row;
if ((row+2)%2 == 0){//even row(includes 0)
    col = col*2;
}else{//odd row
    col = (col+1)*2-1;
}
if (king[Piece] == player){
    if (checkMoveKing(newBoard1, newKBoard1, row, col, player,
        'l', 'u', 0, 0, 1)){
        copy(newBoard1, possibleBoards1[boardCount], 32);
        copy(newKBoard1, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard2, newKBoard2, row, col, player,
        'r', 'u', 0, 0, 1)){
        copy(newBoard2, possibleBoards1[boardCount], 32);
        copy(newKBoard2, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard3, newKBoard3, row, col, player,
        'l', 'd', 0, 0, 1)){
        copy(newBoard3, possibleBoards1[boardCount], 32);
        copy(newKBoard3, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
    if (checkMoveKing(newBoard4, newKBoard4, row, col, player,
        'r', 'd', 0, 0, 1)){
        copy(newBoard4, possibleBoards1[boardCount], 32);
        copy(newKBoard4, possibleBoardsK1[boardCount], 32);
        boardCount++;
    }
}else{
if (checkMove(newBoard1, newKBoard1, row, col, player, 'l',
0, 0, 1)){
    copy(newBoard1, possibleBoards1[boardCount], 32);
    copy(newKBoard1, possibleBoardsK1[boardCount], 32);
    boardCount++;
}
if (checkMove(newBoard2, newKBoard2, row, col, player, 'r',
0, 0, 1)){
    copy(newBoard2, possibleBoards1[boardCount], 32);
    copy(newKBoard2, possibleBoardsK1[boardCount], 32);
    boardCount++;
}
}
i++;
if (boardCount == 13){
    i = 20;}
boardCount--;
return boardCount;
}

int* minimax(int board[], int king[], int depth, int maxPlayer){
    if (depth == 0 | player1PieceNum == 0 | player2PieceNum == 0){
        //copy(board, best_move, 32);
        best_move[32] = findRating(board, king, maxPlayer);
        //Serial.println("start");
        //printBoard(best_move);
        return best_move;
    }
    int boardCount = 0;
    findPieces(board, king, maxPlayer);
    int possibleBoards[24][32];
    int i = 0;
    //boardCount = 0;
    while(i < 12){
        if (PieceList[i] != -1){
            int newBoard1[32];
            int newBoard2[32];
            int newBoard3[32];
            int newBoard4[32];
            int newKBoard1[32];
            int newKBoard2[32];
            if (checkMove(newBoard1, newKBoard1, row, col, player, 'l',
                0, 0, 1)){
                copy(newBoard1, possibleBoards1[boardCount], 32);
                copy(newKBoard1, possibleBoardsK1[boardCount], 32);
                boardCount++;
            }
            if (checkMove(newBoard2, newKBoard2, row, col, player, 'r',
                0, 0, 1)){
                copy(newBoard2, possibleBoards1[boardCount], 32);
                copy(newKBoard2, possibleBoardsK1[boardCount], 32);
                boardCount++;
            }
        }
    }
    return best_move;
}
int newKBoard3[32];
int newKBoard4[32];
copy(board, newBoard1, 32);
copy(board, newBoard2, 32);
copy(board, newBoard3, 32);
copy(board, newBoard4, 32);
copy(king, newKBoard1, 32);
copy(king, newKBoard2, 32);
copy(king, newKBoard3, 32);
copy(king, newKBoard4, 32);
int Piece = PieceList[i];
int row = Piece/4;
int col = Piece - 4*row;
if (row+2)%2 == 0){//even row(includes 0)
  col = col*2;
} else {//odd row
  col = (col+1)*2-1;
}
if (king[Piece] == maxPlayer) {
  if (checkMoveKing(newBoard1, newKBoard1, row, col, maxPlayer, 'l', 'u', 0, 0, 1)) {
    copy(newBoard1, possibleBoards[boardCount], 32);
    boardCount++;
  }
  if (checkMoveKing(newBoard2, newKBoard2, row, col, maxPlayer, 'r', 'u', 0, 0, 1)) {
    copy(newBoard2, possibleBoards[boardCount], 32);
    boardCount++;
  }
  if (checkMoveKing(newBoard3, newKBoard3, row, col, maxPlayer, 'l', 'd', 0, 0, 1)) {
    copy(newBoard3, possibleBoards[boardCount], 32);
    boardCount++;
  }
  if (checkMoveKing(newBoard4, newKBoard4, row, col, maxPlayer, 'r', 'd', 0, 0, 1)) {
    copy(newBoard4, possibleBoards[boardCount], 32);
    boardCount++;
  }
} else {
if (checkMove(newBoard1, newKBoard1, row, col, maxPlayer, 'l', 0, 0, 1)){
    copy(newBoard1, possibleBoards[boardCount], 32);
    boardCount++;
}
if (checkMove(newBoard2, newKBoard2, row, col, maxPlayer, 'r', 0, 0, 1)){
    copy(newBoard2, possibleBoards[boardCount], 32);
    boardCount++;
}
)
}
i++;
boardCount--;
if (boardCount == 32){
    Serial.println("to big");
}
//return boardCount;
//int boardCount = findPossibleMoves(board, king, maxPlayer);
if (maxPlayer == 2){//ai player
    int maxEval = -1000;
    copy(zero, best_move, 32);
    while (boardCount >= 0){
        int stor = minimax(possibleBoards[boardCount], king, depth-1, 1)[32];
        best_move[32] = stor;
        if (maxEval < best_move[32]){  
            maxEval = best_move[32];
            copy(possibleBoards[boardCount], best_move, 32);
        }
        boardCount--;
    }
    best_move[32] = maxEval;
    return best_move;
}else{//human player
    int minEval = 1000;
    copy(zero, best_move, 32);
    while (boardCount >= 0){
        int stor = minimax(possibleBoards[boardCount], king, depth-1, 1)[32];
        best_move[32] = stor;
        if (minEval > best_move[32]){  
            minEval = best_move[32];
            copy(possibleBoards[boardCount], best_move, 32);
        }
        boardCount--;
    }
    best_move[32] = minEval;
    return best_move;
}
minEval = best_move[32];
copy(possibleBoards[boardCount], best_move, 32);
}
boardCount--;
}
best_move[32] = minEval;
return best_move;
}

int findPieces(int board[], int king[], int player){
    int count = 0;
    int num = 0;
    memset(PieceList, -1, sizeof(PieceList));
    //int[12] list;
    while (count < 32){
        if (board[count] == player | king[count] == player){
            PieceList[num] = count;
            num++;
        }
        count++;
    }
    return 1;
}

int checkJump(int board[], int king[], int player, int row, int col){
    int Piece = board[row*4 + col];
    int check = 0;
    if (king[Piece] == player){ //if its a king
        if (checkMoveKing(board, king, row, col, player, 'l', 'u', 0, 1, 0)==2 | checkMoveKing(board, king, row, col, player, 'l', 'd', 0, 1, 0)==2 | checkMoveKing(board, king, row, col, player, 'r', 'u', 0, 1, 0)==2 | checkMoveKing(board, king, row, col, player, 'r', 'd', 0, 1, 0)==2){
            check = 1;
        }
    }else{ //normal piece
        if (checkMove(board, king, row, col, player, 'l', 0, 1, 0)==2 | checkMove(board, king, row, col, player, 'r', 0, 1, 0)==2){
            check = 1;
        }
    }
    return check;
}
int findJumps(int board[], int king[], int player, int PieceList[]){
    memset(JumpList, -1, sizeof(JumpList));
    //int[24] jumps;
    int num = 0;
    for (int i = 0; i < 12; i++){
        int check = 0;
        int Piece = PieceList[i];
        int row = Piece/4;
        int col = Piece - 4*row;
        if ((row+2)%2 == 0){//even row(includes 0)
            col = col*2;
        }else{//odd row
            col = (col+1)*2-1;
        }
        if (king[Piece] == player){//if its a king
            if (checkMoveKing(board, king, row, col, player, 'l', 'u', 0, 1, 0)==2 | checkMoveKing(board, king, row, col, player, 'l', 'd', 0, 1, 0)==2 | checkMoveKing(board, king, row, col, player, 'r', 'u', 0, 1, 0)==2 | checkMoveKing(board, king, row, col, player, 'r', 'd', 0, 1, 0)==2){
                check = 1;
            }
        }else{//normal piece
            if (checkMove(board, king, row, col, player, 'l', 0, 1, 0)==2 | checkMove(board, king, row, col, player, 'r', 0, 1, 0)==2){
                check = 1;
            }
        }
        if (check == 1){
            JumpList[num] = Piece;
            num++;
        }
    }
    return check;
}
int checkMove(int board[], int king[], int row, int col, int player, char dir, int jump, int check, int check2){
    int curPieceNum = 4*(row)+col/2;
    int nextPieceNum = 4*(row)+col/2;
    //Serial.println(curPieceNum);
    int nextPiece = 0;
    int nextPieceNum = 0;
    if (player == 1){
        int slant = row%2; //changes how pieces move if row starts
        //with black or white space
        if (slant == 0){ //first space is white, or a playable
            if (dir == 'l'){
                if (col/2 == 0){
                    return 0;
                }else{
                    nextPieceNum = 4*(row)+col/2+3;
                    nextPiece = board[4*(row)+col/2+3]; //+3 seems to work
                }
            }
        }else{ //slant is equal to 1, so first space is black, or
        }
    }
    if (dir == 'r'){
        if (col/2 == 3){
            //try again, up against the wall
            //scratch that, no wall because of slant
            nextPieceNum = 4*(row)+col/2+4;
            nextPiece = board[4*(row)+col/2+4]; //+4 works here
        }else{
            nextPieceNum = 4*(row)+col/2+4;
            nextPiece = board[4*(row)+col/2+4]; //+4 works here
        }
    }
}
else{ //slant is equal to 1, so first space is black, or
not playable
    if (dir == 'l'){
        if (col/2 == 0){
            //try again, up against the wall
            //scratch that, no wall because of slant
            nextPieceNum = 4*(row)+col/2+4;
        }
    }
    return 0;
}
nextPiece = board[4*(row)+col/2+4]; //+4 seems to work for all instances here
} else{
    nextPieceNum = 4*(row)+col/2+4;
    nextPiece = board[4*(row)+col/2+4]; //+4 seems to work for all instances here
}

if (dir == 'r') {
    if (col/2 == 3) {
        return 0;
        //try again, up against the wall
    } else {
        nextPieceNum = 4*(row)+col/2+5;
        nextPiece = board[4*(row)+col/2+5]; //+5 works here, steapest addition possible
    }
}

if (player == 2) {
    int slant = row%2; //changes how pieces move if row starts with black or white space
    if (slant == 0) { //first space is white, or a playable square
        if (dir == 'l') {
            if (col/2 == 3) {
                //try again, up against the wall
                //no wall
                nextPieceNum = 4*(row)+col/2-4;
                nextPiece = board[4*(row)+col/2-4]; //-4 seems to work for all instances here
            } else {
                nextPieceNum = 4*(row)+col/2-4;
                nextPiece = board[4*(row)+col/2-4]; //-4 seems to work for all instances here
            }
        } else {
        }
    } else {
        if (dir == 'r') {
            if (col/2 == 0) {
                return 0;
            } else {
            }
        }
    }
}
//try again, up against the wall
} else{
    nextPieceNum = 4*(row)+col/2-5;
    nextPiece = board[4*(row)+col/2-5]; // -5 works here
}

} else{ // slant is equal to 1, so first space is black, or not playable
    if (dir == 'l'){
        if (col/2 == 3){
            return 0;
            // try again, up against the wall
        } else{
            nextPieceNum = 4*(row)+col/2-3;
            nextPiece = board[4*(row)+col/2-3]; // -3 seems to work for all instances here
        }
    }
    if (dir == 'r'){
        if (col/2 == 0){
            // try again, up against the wall
            // no wall
            nextPieceNum = 4*(row)+col/2-4;
            nextPiece = board[4*(row)+col/2-4]; // -4 works here, steapest addition possible
        } else{
            nextPieceNum = 4*(row)+col/2-4;
            nextPiece = board[4*(row)+col/2-4]; // -4 works here, steapest addition possible
        }
    }
    if (nextPiece == 0){
        if (jump == 0){
            if (player == 1){
                if (check == 0){
                    LEDOff(LEDList, board, king, curPieceNum, check2);
                    LEDGreen(LEDList, board, nextPieceNum, check2);
                    if (nextPieceNum > 27){ // 28-31 for player one makes you king
                        king
                    }
                }
            }
        }
    }
}
LEDKing(LEDList, board, king, nextPieceNum, player, check2);
}
else{return 0;}
}
if (check == 0){
    LEDOff(LEDList, board, king, curPieceNum, check2);
    LEDRed(LEDList, board, nextPieceNum, check2);
    if (nextPieceNum < 4){ //0-3 for player 2 makes you king
        LEDKing(LEDList, board, king, nextPieceNum, player, check2);
    }
    else{return 0;}
}
if (jump == 1){
    if (player == 1){
        if (dir == 'r'){
            //9 to original
            if (check == 0){
                LEDOff(LEDList, board, king, curPieceNum, check2);
                LEDOff(LEDList, board, king, nextPieceNum-9, check2);
                LEDGreen(LEDList, board, nextPieceNum, check2);
                if (check2 == 0){
                    player2PieceNum-=1;
                }
                if (nextPieceNum > 27){
                    LEDKing(LEDList, board, king, nextPieceNum, player, check2);
                }
            }
        }
    }
}
else{
    //7 to original
    if (check == 0){
        LEDOff(LEDList, board, king, curPieceNum, check2);
        LEDOff(LEDList, board, king, nextPieceNum-7, check2);
        LEDGreen(LEDList, board, nextPieceNum, check2);
        if (check2 == 0){
            player2PieceNum-=1;
        }
        if (nextPieceNum > 27){
            LEDKing(LEDList, board, king, nextPieceNum, player, check2);
        }
    }
}
if (dir == 'r') {
    // -9 to original
    if (check == 0) {
        LEDOff(LEDList, board, king, curPieceNum, check2);
        LEDOff(LEDList, board, king, nextPieceNum+9, check2);
        LEDRed(LEDList, board, nextPieceNum, check2);
        if (check2 == 0) {
            player1PieceNum--;}
        if (nextPieceNum < 4) {
            LEDKing(LEDList, board, king, nextPieceNum, player,
        check2);
        } else { return 2; }
    } else {
        // -7 to original
        if (check == 0) {
            LEDOff(LEDList, board, king, curPieceNum, check2);
            LEDOff(LEDList, board, king, nextPieceNum+7, check2);
            LEDRed(LEDList, board, nextPieceNum, check2);
            if (check2 == 0) {
                player1PieceNum--;}
            if (nextPieceNum < 4) {
                LEDKing(LEDList, board, king, nextPieceNum, player,
            check2);
            } else { return 2; }
        }
    }
}
return 1;

if (nextPiece == 1 & player == 1) {
    return 0;
}
if (nextPiece == 2 & player == 2) {
    return 0;
}
if (nextPiece == 2 & player == 1) {
    if (jump == 1) {
        return 0;
    } else {
        if (dir == 'r') {
int checkMoveKing(int board[], int king[], int row, int col, int player, char dir, char UorD, int jump, int check, int check2)
{
    int currentPiece = board[4*(row)+col/2];
    int curPieceNum = 4*(row)+col/2;
    //Serial.println(curPieceNum);
    int nextPiece = 0;
    int nextPieceNum = 0;
    if (player == 1)
    {
        int slant = row%2; //changes how pieces move if row starts
        with black or white space
        if (slant == 0){ //first space is white, or a playable
            square
            if (UorD == 'u'){//like normal
                if (dir == 'l'){
                    if (col/2 == 0){
                        return 0;
                    }
                }
            }
        }
    }
    if (nextPiece == 1 & player == 2){
        if (jump == 1){
            return 0;
        }else{
            if (dir == 'r'){
                return checkMove(board, king, row-1, col-1, player, dir,
                1, check, check2);
            }
        }
    }
}

int checkMoveKing(int board[], int king[], int row, int col, int player, char dir, char UorD, int jump, int check, int check2)
{
    int currentPiece = board[4*(row)+col/2];
    int curPieceNum = 4*(row)+col/2;
    //Serial.println(curPieceNum);
    int nextPiece = 0;
    int nextPieceNum = 0;
    if (player == 1)
    {
        int slant = row%2; //changes how pieces move if row starts
        with black or white space
        if (slant == 0){ //first space is white, or a playable
            square
            if (UorD == 'u'){//like normal
                if (dir == 'l'){
                    if (col/2 == 0){
                        return 0;
                    }
                }
            }
        }
    }
    if (nextPiece == 1 & player == 2){
        if (jump == 1){
            return 0;
        }else{
            if (dir == 'r'){
                return checkMove(board, king, row-1, col-1, player, dir,
                1, check, check2);
            }
        }
    }
    if (nextPiece == 1 & player == 2){
        if (jump == 1){
            return 0;
        }else{
            if (dir == 'r'){
                return checkMove(board, king, row-1, col-1, player, dir,
                1, check, check2);
            }
        }
    }
}
if (dir == 'r') {
    if (col/2 == 3) {
        // try again, up against the wall
    } else {
        nextPieceNum = 4*(row)+col/2+3;
        nextPiece = board[4*(row)+col/2+3]; // +3 seems to work for all instances here
    }
}
if (dir == 'l') {
    if (col/2 == 0) {
        return 0; // try again, up against the wall
    } else {
        nextPieceNum = 4*(row)+col/2-4;
        nextPiece = board[4*(row)+col/2-4]; // -4 works here
    }
}
}
if (dir == 'r') {
    if (col/2 == 3) {
        // try again, up against the wall
    } else {
        nextPieceNum = 4*(row)+col/2-4;
        nextPiece = board[4*(row)+col/2-4]; // -4 works here
    }
}
}
} else { //slant is equal to 1, so first space is black, or not playable
    if (UorD == 'u') {//stays the same
        if (dir == 'l') {
            if (col/2 == 0) {
                //try again, up against the wall
                //scratch that, no wall because of slant
                nextPieceNum = 4*(row)+col/2+4;
                nextPiece = board[4*(row)+col/2+4];  //+4 seems to work
            } else {
                nextPieceNum = 4*(row)+col/2+4;
                nextPiece = board[4*(row)+col/2+4];  //+4 seems to work
            }
        }
        if (dir == 'r') {
            if (col/2 == 3) {
                return 0;
                //try again, up against the wall
            } else {
                nextPieceNum = 4*(row)+col/2+5;
                nextPiece = board[4*(row)+col/2+5];  //+5 works here, steapest addition possible
            }
        }
    }
    else {//down direction
        if (dir == 'l') {
            if (col/2 == 0) {
                //try again, up against the wall
                //scratch that, no wall because of slant
                nextPieceNum = 4*(row)+col/2-4;
                nextPiece = board[4*(row)+col/2-4];  //-4 seems to work
            } else {
                nextPieceNum = 4*(row)+col/2-4;
                nextPiece = board[4*(row)+col/2-4];  //-4 seems to work
            }
        }
        if (dir == 'r') {
            //
if (col/2 == 3){
    return 0;
    //try again, up against the wall
}else{
    nextPieceNum = 4*(row)+col/2-3;
    nextPiece = board[4*(row)+col/2-3]; // -3 works here, steapest addition possible
}
}
}
}
if (player == 2){
    int slant = row%2; //changes how pieces move if row starts with black or white space
    if (slant == 0){ //first space is white, or a playable square
        if (UorD == 'u'){ //stays the same
            if (dir == 'l'){
                if (col/2 == 3){ //try again, up against the wall
                    //no wall
                    nextPieceNum = 4*(row)+col/2-4;
                    nextPiece = board[4*(row)+col/2-4]; // -4 seems to work for all instances here
                }else{
                    nextPieceNum = 4*(row)+col/2-4;
                    nextPiece = board[4*(row)+col/2-4]; // -4 seems to work for all instances here
                }
            }else{ //down direction
                if (dir == 'l'){
                    if (col/2 == 3){
                        return 0;
                        //try again, up against the wall
                    }else{
                        nextPieceNum = 4*(row)+col/2-5;
                        nextPiece = board[4*(row)+col/2-5]; // -5 works here
                    }
                }else{ //down direction
                    if (dir == 'r'){
                        if (col/2 == 0){
                            return 0;
                            //try again, up against the wall
                        }else{
                            nextPieceNum = 4*(row)+col/2-5;
                            nextPiece = board[4*(row)+col/2-5]; // -5 works here
                        }
                    }
                }
            }
        }else{ //down direction
            if (dir == 'r'){
                if (col/2 == 3){
                    if (col/2 == 3){
//try again, up against the wall
//no wall
nextPieceNum = 4*(row)+col/2+4;
nextPiece = board[4*(row)+col/2+4]; //+4 seems to work for all instances here
} else {
    nextPieceNum = 4*(row)+col/2+4;
    nextPiece = board[4*(row)+col/2+4]; //+4 seems to work for all instances here
}
}
if (dir == 'r') {
    if (col/2 == 0) {
        return 0;
        //try again, up against the wall
    } else {
        nextPieceNum = 4*(row)+col/2+3;
        nextPiece = board[4*(row)+col/2+3]; //+3 works here
    }
}
if (UorD == 'u') { //stays the same
    if (dir == 'l') {
        if (col/2 == 3) {
            return 0;
            //try again, up against the wall
        } else {
            nextPieceNum = 4*(row)+col/2-3;
            nextPiece = board[4*(row)+col/2-3]; //-3 seems to work for all instances here
        }
    }
}
else { //slant is equal to 1, so first space is black, or not playable
    if (UorD == 'u') { //stays the same
        if (dir == 'l') {
            if (col/2 == 3) {
                return 0;
                //try again, up against the wall
            } else {
                nextPieceNum = 4*(row)+col/2-3;
                nextPiece = board[4*(row)+col/2-3]; //-3 seems to work for all instances here
            }
        }
    }
    if (dir == 'r') {
        if (col/2 == 0) {
            //try again, up against the wall
            //no wall
            nextPieceNum = 4*(row)+col/2-4;
            nextPiece = board[4*(row)+col/2-4]; //-4 works here, steapest addition possible
if (nextPiece == 0){
    if (jump == 0){
        if (player == 1){
            if (check == 0){
                LEDOff(LEDList, board, king, curPieceNum, check2);
                LEDGreen(LEDList, board, nextPieceNum, check2);
                LEDKing(LEDList, board, king, nextPieceNum, player,
                check2);
            } else{return 0;}
        }
    }
}
}
if (check == 0){
    LEDOff(LEDList, board, king, curPieceNum, check2);
    LEDRed(LEDList, board, nextPieceNum, check2);
    LEDKing(LEDList, board, king, nextPieceNum, player, check2);
}
else{return 0;}
}
}
if (jump == 1){
    if (player == 1){
        if (dir == 'r'){
            //-9 to original
            if (check == 0){
                LEDOff(LEDList, board, king, curPieceNum, check2);
                LEDOff(LEDList, board, king, nextPieceNum-9, check2);
                LEDGreen(LEDList, board, nextPieceNum, check2);
                if (check2 == 0){
                    player2PieceNum-=1;
                }
                LEDKing(LEDList, board, king, nextPieceNum, player, check2);
            }
        }else{
            //-7 to original
            if (check == 0){
                LEDOff(LEDList, board, king, curPieceNum, check2);
                LEDOff(LEDList, board, king, nextPieceNum-7, check2);
                LEDGreen(LEDList, board, nextPieceNum, check2);
                if (check2 == 0){
                    player2PieceNum-=1;
                }
                LEDKing(LEDList, board, king, nextPieceNum, player, check2);
            }else{return 2;}//if check is on
        }
    }else{
        if (dir == 'r'){
            //-9 to original
            if (check == 0){
                LEDOff(LEDList, board, king, curPieceNum, check2);
                LEDOff(LEDList, board, king, nextPieceNum+9, check2);
                LEDRed(LEDList, board, nextPieceNum, check2);
            }else{return 2;}//if check is on
        }
    }
}
if (check2 == 0){
    player1PieceNum-=1;
    LEDKing(LEDList, board, king, nextPieceNum, player, check2);
} else {return 2;}
}

if (check == 0){
    LEDOff(LEDList, board, king, curPieceNum, check2);
    LEDOff(LEDList, board, king, nextPieceNum+7, check2);
    LEDRed(LEDList, board, nextPieceNum, check2);
    if (check2 == 0){
        player1PieceNum-=1;
    }
    LEDKing(LEDList, board, king, nextPieceNum, player, check2);
} else {return 2;}

if (nextPiece == 1 & player == 1){
    return 0;
}
if (nextPiece == 2 & player == 2){
    return 0;
}
if (nextPiece == 2 & player == 1){
    if (jump == 1){
        return 0;
    } else {
        if (UorD == 'u'){
            if (dir == 'r'){
                return checkMoveKing(board, king, row+1, col+1, player, dir, UorD, 1, check, check2);
            }
            if (dir == 'l'){
                return checkMoveKing(board, king, row+1, col-1, player, dir, UorD, 1, check, check2);
            }
        } else {//down
            if (dir == 'r'){
                return checkMoveKing(board, king, row+1, col+1, player, dir, UorD, 1, check, check2);
            }
            if (dir == 'l'){
                return checkMoveKing(board, king, row+1, col-1, player, dir, UorD, 1, check, check2);
            }
        }
    }
} else {//down
    if (dir == 'r'){
return checkMoveKing(board, king, row-1, col+1, player, dir, UorD, 1, check, check2);
}
if (dir == 'l'){
    return checkMoveKing(board, king, row-1, col-1, player, dir, UorD, 1, check, check2);
}
}
}

if (nextPiece == 1 & player == 2){
if (jump == 1){
    return 0;
} else{
    if (UorD == 'u'){
        if (dir == 'r'){
            return checkMoveKing(board, king, row-1, col-1, player, dir, UorD, 1, check, check2);
        }
        if (dir == 'l'){
            return checkMoveKing(board, king, row-1, col+1, player, dir, UorD, 1, check, check2);
        }
    }
    if (dir == 'l'){
        return checkMoveKing(board, king, row-1, col-1, player, dir, UorD, 1, check, check2);
    }
} else{
    if (dir == 'r'){
        return checkMoveKing(board, king, row+1, col-1, player, dir, UorD, 1, check, check2);
    }
    if (dir == 'l'){
        return checkMoveKing(board, king, row+1, col+1, player, dir, UorD, 1, check, check2);
    }
}
}

void LEDOff(int LEDList[32][2], int board[], int king[], int LEDnum, int check){
    board[LEDnum] = 0;
    if (check == 0){
        digitalWrite(LEDList[LEDnum][1], LOW);
```c
void LEDRed(int LEDList[32][2], int board[], int LEDnum, int check){
    board[LEDnum] = 2;
    if (check == 0){
        digitalWrite(LEDList[LEDnum][1], LOW);
        digitalWrite(LEDList[LEDnum][1]+2, HIGH);
    }
}

void LEDGreen(int LEDList[32][2], int board[], int LEDnum, int check){
    board[LEDnum] = 1;
    if (check == 0){
        digitalWrite(LEDList[LEDnum][1], HIGH);
        digitalWrite(LEDList[LEDnum][1]+2, LOW);
    }
}

void LEDKing(int LEDList[32][2], int board[], int king[], int LEDnum, int player, int check){
    king[LEDnum] = player;
    board[LEDnum] = player;
    if (check == 0){
        digitalWrite(LEDList[LEDnum][1], HIGH);
        digitalWrite(LEDList[LEDnum][1]+2, HIGH);
    }
}

void changeKing(int LEDList[32][2], int board[], int king[], int player, int Oplayer) {
    int num = 0;
    while (num < 32){
        if (king[num] == player){
            LEDKing(LEDList, board, king, num, player, 0);
        }
        if (king[num] == Oplayer){
            if (Oplayer == 1){
                LEDGreen(LEDList, board, num, 0);
            } else{
                LEDRed(LEDList, board, num, 0);
            }
        }
        num++;
    }
}
```
void setup() {
    Serial.begin(9600);
    pinMode(54, OUTPUT); //0
    pinMode(56, OUTPUT);
    pinMode(55, OUTPUT); //1
    pinMode(57, OUTPUT);
    pinMode(58, OUTPUT); //2
    pinMode(60, OUTPUT);
    pinMode(59, OUTPUT); //3
    pinMode(61, OUTPUT);
    pinMode(62, OUTPUT); //4
    pinMode(64, OUTPUT);
    pinMode(63, OUTPUT); //5
    pinMode(65, OUTPUT);
    pinMode(66, OUTPUT); //6
    pinMode(68, OUTPUT);
    pinMode(67, OUTPUT); //7
    pinMode(69, OUTPUT);
    pinMode(22, OUTPUT); //8
    pinMode(24, OUTPUT);
    pinMode(23, OUTPUT); //9
    pinMode(25, OUTPUT);
    pinMode(26, OUTPUT); //10
    pinMode(28, OUTPUT);
    pinMode(27, OUTPUT); //11
    pinMode(29, OUTPUT);
    pinMode(30, OUTPUT); //12
    pinMode(32, OUTPUT);
    pinMode(31, OUTPUT); //13
    pinMode(33, OUTPUT);
    pinMode(34, OUTPUT); //14
    pinMode(36, OUTPUT);
    pinMode(35, OUTPUT); //15
pinMode(37, OUTPUT);
pinMode(38, OUTPUT); //16
pinMode(40, OUTPUT);
pinMode(39, OUTPUT); //17
pinMode(41, OUTPUT);
pinMode(42, OUTPUT); //18
pinMode(44, OUTPUT);
pinMode(43, OUTPUT); //19
pinMode(45, OUTPUT);
pinMode(46, OUTPUT); //20
pinMode(48, OUTPUT);
pinMode(47, OUTPUT); //21
pinMode(49, OUTPUT);
pinMode(50, OUTPUT); //22
pinMode(52, OUTPUT);
pinMode(51, OUTPUT); //23
pinMode(53, OUTPUT);
pinMode(2, OUTPUT); //24
pinMode(4, OUTPUT);
pinMode(3, OUTPUT); //25
pinMode(5, OUTPUT);
pinMode(6, OUTPUT); //26
pinMode(8, OUTPUT);
pinMode(7, OUTPUT); //27
pinMode(9, OUTPUT);
pinMode(10, OUTPUT); //28
pinMode(12, OUTPUT);
pinMode(11, OUTPUT); //29
pinMode(13, OUTPUT);
pinMode(14, OUTPUT); //30
pinMode(16, OUTPUT);
pinMode(15, OUTPUT); //31
pinMode(17, OUTPUT);
LEDGreen(LEDList, Board, 0, 0);
LEDGreen(LEDList, Board, 1, 0);
LEDGreen(LEDList, Board, 2, 0);
LEDGreen(LEDList, Board, 3, 0);
LEDGreen(LEDList, Board, 4, 0);
LEDGreen(LEDList, Board, 5, 0);
LEDGreen(LEDList, Board, 6, 0);
LEDGreen(LEDList, Board, 7, 0);
LEDGreen(LEDList, Board, 8, 0);
LEDGreen(LEDList, Board, 9, 0);
LEDGreen(LEDList, Board, 10, 0);
LEDGreen(LEDList, Board, 11, 0);
// LEDKing(LEDList, board, king, 20, 2);
// LEDKing(LEDList, board, king, 21, 2);
// LEDKing(LEDList, board, king, 22, 2);
// LEDKing(LEDList, board, king, 23, 2);
LEDRed(LEDList, Board, 20, 0);
LEDRed(LEDList, Board, 21, 0);
LEDRed(LEDList, Board, 22, 0);
LEDRed(LEDList, Board, 23, 0);
LEDRed(LEDList, Board, 24, 0);
LEDRed(LEDList, Board, 25, 0);
LEDRed(LEDList, Board, 26, 0);
LEDRed(LEDList, Board, 27, 0);
LEDRed(LEDList, Board, 28, 0);
LEDRed(LEDList, Board, 29, 0);
LEDRed(LEDList, Board, 30, 0);
LEDRed(LEDList, Board, 31, 0);
int incomingByte = 0;
Serial.println("Input the number of players you have, 1 = player vs computer, 2 = player vs player");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 49: //1
        PorC = 1;
        break;
    case 50: //2
        PorC = 2;
        break;
    default:
        Serial.println("unable to recognize your input, please re-enter");
        return;
        //try again
        break;
if(PorC == 1){
    Serial.println("Input the number of the difficulty you would like, 1 = easy, 2 = medium, 3 = hard");
    while (Serial.available() == 0){
        //incomingByte = Serial.read();
    }
    incomingByte = Serial.read();
    Serial.println(incomingByte);
    switch (incomingByte) {
    case 49: //1
        Diff = 1;
        break;
    case 50: //2
        Diff = 2;
        break;
    case 51: //3
        Diff = 3;
        default:
            Serial.println("unable to recognize your input, please re-enter");
            return;
            //try again
            break;
    }
}

// the loop function runs over and over again forever
void loop() {
    int incomingByte = 0;
    if (PorC==1){
        if (Diff == 3){
            if (Player == 1){
                Player = 1;
                Serial.println("Player 1's turn");
            }else{
                Player = 2;
                Serial.println("Player 2's turn");
            }
        }
        if (Player == 2){//cpu turn
            //AIMove = minimax(Board, King, 2, 2);
            int score;
        }
    }
}
score = minimax3(Board, King);
//printBoard(possibleBoards[bestList3[1]]);
Serial.println("index");
Serial.print(score);
//Serial.println(bestList3[1]);
//printBoard(possibleBoards[bestList3[1]]);
copy(AIMove, Board, 32);
//copy(possibleBoards[bestList3[1]], Board, 32);//minimax3 stuff
//copy(possibleBoardsK[bestList3[1]], King, 32);
//int num = minimax1(Board, King, 2);//minimax1 stuff
//printBoard(possibleBoardsK[num]);
//copy(minimax(Board, King, 2, 2), AIMove, 32);
//copy(possibleBoards[num], Board, 32);//minimax1 stuff
//copy(possibleBoardsK[num], King, 32);
//copy(AIMove, Board, 32);
//printBoard(Board);
Player = 1;
int x = 0;
while (x < 32){
    if (Board[x] == 0){
        LEDOff(LEDList, Board, King, x, 0);
    } else{
        findPieces(Board, King, Player);
        findJumps(Board, King, Player, PieceList);
        int count = 0;
        while (JumpList[count] != -1){
            Serial.println("possible jump, you must move one of the following pieces");
            int row = JumpList[count]/4;
            int col = JumpList[count] - row*4;
            if ((row+2)%2 == 0){//even row(includes 0)
            }
        }
    }
    if (Board[x] == 1){
        LEDGreen(LEDList, Board, x, 0);
    } else{
        findPieces(Board, King, Player);
        findJumps(Board, King, Player, PieceList);
        int count = 0;
        while (JumpList[count] != -1){
            Serial.println("possible jump, you must move one of the following pieces");
            int row = JumpList[count]/4;
            int col = JumpList[count] - row*4;
            if ((row+2)%2 == 0){//even row(includes 0)
            }
        }
    }
    if (Board[x] == 2){
        LEDRed(LEDList, Board, x, 0);
    } else{
        findPieces(Board, King, Player);
        findJumps(Board, King, Player, PieceList);
        int count = 0;
        while (JumpList[count] != -1){
            Serial.println("possible jump, you must move one of the following pieces");
            int row = JumpList[count]/4;
            int col = JumpList[count] - row*4;
            if ((row+2)%2 == 0){//even row(includes 0)
            }
        }
    }
    x++;
}
}
col = col*2;
} else { // odd row
    col = (col+1)*2-1;
}
Serial.print((char)(row+97)); // in Ascii
Serial.print(",");
Serial.println(col+1);
count++;
}
Serial.println("Input the row of the piece you wish to move");
while (Serial.available() == 0) {
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
case 97: //a
    currentRow = 0;
    break;
case 98: //b
    currentRow = 1;
    break;
case 99: //c
    currentRow = 2;
    break;
case 100: //d
    currentRow = 3;
    break;
case 101: //e
    currentRow = 4;
    break;
case 102: //f
    currentRow = 5;
    break;
case 103: //g
    currentRow = 6;
    break;
case 104: //h
    currentRow = 7;
    break;
default:
Serial.println("unable to recognize your input, please re-enter your turn");
  return;
  //try again
  break;
}
delay(1000);

Serial.println("Input the column of the piece you wish to move");
while (Serial.available() == 0){
  //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
case 49: //1
  currentCol = 0;
  break;
case 50: //2
  currentCol = 1;
  break;
case 51: //3
  currentCol = 2;
  break;
case 52: //4
  currentCol = 3;
  break;
case 53: //5
  currentCol = 4;
  break;
case 54: //6
  currentCol = 5;
  break;
case 55: //7
  currentCol = 6;
  break;
case 56: //8
  currentCol = 7;
break;
default:
    Serial.println("unable to recognize your input, please re-enter your turn");
    return;
    //try again
    break;
}
delay(1000);

if(King[4*(currentRow)+currentCol/2] == Player){
    Serial.println("Input 'u' if you wish to move the piece up, and 'd' if you wish to move the piece down");
    while (Serial.available() == 0){
        //incomingByte = Serial.read();
    }
    incomingByte = Serial.read();
    Serial.println(incomingByte);
    switch (incomingByte) {
        case 117: //u
            currentUorD = 'u';
            break;
        case 100: //d
            currentUorD = 'd';
            break;
        default:
            Serial.println("unable to recognize your input, please re-enter your turn");
            return;
            //try again
            break;
    }
    delay(1000);
}

Serial.println("Input 'l' if you wish to move the piece to the left, and 'r' if you wish to move the piece to the right");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 108: //l
        currentDir = 'l';
        break;
    case 114: //r
        currentDir = 'r';
        break;
    default:
        Serial.println("unable to recognize your input, please re-enter your turn");
        return;
        //try again
        break;
}
delay(1000);

if(King[4*(currentRow)+currentCol/2] == Player){
    check = checkMoveKing(Board, King, currentRow, currentCol, Player, currentDir, currentUorD, 0, 0, 0);
} else {
    check = checkMove(Board, King, currentRow, currentCol, Player, currentDir, 0, 0, 0);
}
if (check == 1){
    Serial.println("move successful");
    int nextRow = nextPieceNum/4;
    int nextCol = nextPieceNum - 4*nextRow;
    if (((nextRow+2)%2 == 0) //even row(includes 0)
        nextCol = nextCol*2;
    } else { //odd row
        nextCol = (nextCol+1)*2-1;
    }
if (checkJump(Board, King, Player, nextRow, nextCol)){
    Serial.println("double jump available");
    //must double jump
} else {
    if (Player == 1){
        Player = 2; //next players turn
    }
changeKing(LEDList, Board, King, 2, 1);
Serial.println("Player 2's turn");
}else{
  Player = 1;
  changeKing(LEDList, Board, King, 1, 2);
  Serial.println("Player 1's turn");
}
}else{
  Serial.println("this move is not possible, please try again");
}
if (player1PieceNum == 0){
  Serial.println("Player 2 has won!");
  //player 2 wins
}
if (player2PieceNum == 0){
  Serial.println("Player 1 has won!");
  //player 1 wins
}
delay(1000);
//Serial.println(incomingByte);
}
if (Diff == 2){
  if (Player == 1){
    Player = 1;
    Serial.println("Player 1's turn");
  }else{
    Player = 2;
    Serial.println("Player 2's turn");
  }
  if (Player == 2){//cpu turn
    int count = minimax1(Board, King, Player);
    copy(AIMove, Board, 32);//minimax1 stuff
    copy(best_move, King, 32);
    //copy(AIMove, Board,32);
    //printBoard(Board);
    Player = 1;
    int x = 0;
    while (x < 32){
      if (Board[x] == 0){
        LEDOff(LEDList, Board, King, x, 0);
      }
    }
  }
}
if (Board[x] == 1) {
    LEDGreen(LEDList, Board, x, 0);
}

if (Board[x] == 2) {
    LEDRed(LEDList, Board, x, 0);
}
x++;
}

else{
    findPieces(Board, King, Player);
    findJumps(Board, King, Player, PieceList);
    int count = 0;
    while (JumpList[count] != -1) {
        Serial.println("possible jump, you must move one of the following pieces");
        int row = JumpList[count] / 4;
        int col = JumpList[count] - row * 4;
        if ((row + 2) % 2 == 0) {// even row (includes 0)
            col = col * 2;
        } else {// odd row
            col = (col + 1) * 2 - 1;
        }
        Serial.print((char)(row + 97)); // in Ascii
        Serial.print(",");
        Serial.println(col + 1);
        count++;
    }
    Serial.println("Input the row of the piece you wish to move");
    while (Serial.available() == 0) {
        // incomingByte = Serial.read();
    }
    incomingByte = Serial.read();
    Serial.println(incomingByte);
    switch (incomingByte) {
    case 97: // a
        currentRow = 0;
        break;
    case 98: // b
        currentRow = 1;
        break;
    }
case 99: //c
    currentRow = 2;
    break;
case 100: //d
    currentRow = 3;
    break;
case 101: //e
    currentRow = 4;
    break;
case 102: //f
    currentRow = 5;
    break;
case 103: //g
    currentRow = 6;
    break;
case 104: //h
    currentRow = 7;
    break;
default:
    Serial.println("unable to recognize your input, please re-enter your turn");
    return;
    //try again
    break;
}

delay(1000);

Serial.println("Input the column of the piece you wish to move");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 49: //1
        currentCol = 0;
        break;
    case 50: //2
currentCol = 1;
break;
case 51: //3
    currentCol = 2;
    break;
case 52: //4
    currentCol = 3;
    break;
case 53: //5
    currentCol = 4;
    break;
case 54: //6
    currentCol = 5;
    break;
case 55: //7
    currentCol = 6;
    break;
case 56: //8
    currentCol = 7;
    break;
default:
    Serial.println("unable to recognize your input, please re-enter your turn");
    return;
    //try again
    break;
}
delay(1000);

if(King[4*(currentRow)+currentCol/2] == Player){
    Serial.println("Input 'u' if you wish to move the piece up, and 'd' if you wish to move the piece down");
    while (Serial.available() == 0){
        //incomingByte = Serial.read();
    }
    incomingByte = Serial.read();
    Serial.println(incomingByte);
    switch (incomingByte) {
        case 117: //u
            currentUorD = 'u';
            break;
break;
case 100: //d
    currentUorD = 'd';
    break;
default:
    Serial.println("unable to recognize your input, please re-enter your turn");
    return;
    //try again
    break;
}
delay(1000);
}

Serial.println("Input 'l' if you wish to move the piece to the left, and 'r' if you wish to move the piece to the right");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 108: //l
        currentDir = 'l';
        break;
    case 114: //r
        currentDir = 'r';
        break;
    default:
        Serial.println("unable to recognize your input, please re-enter your turn");
        return;
        //try again
        break;
}
delay(1000);

if(King[4*(currentRow)+currentCol/2] == Player){
check = checkMoveKing(Board, King, currentRow, currentCol, Player, currentDir, currentUorD, 0, 0, 0);
}
else{
    check = checkMove(Board, King, currentRow, currentCol, Player, currentDir, 0, 0, 0);
}
if (check == 1){
    Serial.println("move successful");
    int nextRow = nextPieceNum/4;
    int nextCol = nextPieceNum - 4*nextRow;
    if ((nextRow+2)%2 == 0){//even row(includes 0)
        nextCol = nextCol*2;
    }else{//odd row
        nextCol = (nextCol+1)*2-1;
    }
    if (checkJump(Board, King, Player, nextRow, nextCol)){
        Serial.println("double jump available");
        //must double jump
    }else{
        if (Player == 1){
            Player = 2; //next players turn
            changeKing(LEDList, Board, King, 2, 1);
            Serial.println("Player 2's turn");
        }else{
            Player = 1;
            changeKing(LEDList, Board, King, 1, 2);
            Serial.println("Player 1's turn");
        }
    }
}
else{
    Serial.println("this move is not possible, please try again");
}
if (player1PieceNum == 0){
    Serial.println("Player 2 has won!");
    //player 2 wins
}
if (player2PieceNum == 0){
    Serial.println("Player 1 has won!");
    //player 1 wins
}
delay(1000);
//Serial.println(incomingByte);
if (Diff == 1) {//random
    if (Player == 1) {
        Player = 1;
        Serial.println("Player 1's turn");
    } else {
        Player = 2;
        Serial.println("Player 2's turn");
    }

    if (Player == 2) {//cpu turn
        int count = minimax1(Board, King, Player);
        int possibleBoards[10][32];
        int possibleBoardsK[10][32];
        int count = findPossibleMoves(Board, King, possibleBoards,
                                      possibleBoardsK, Player);
        int num = random(0, count);
        copy(possibleBoards[num], Board, 32);//minimax1 stuff
        copy(possibleBoardsK[num], King, 32);
        Player = 1;
        int x = 0;
        while (x < 32) {
            if (Board[x] == 0) {
                LEDOff(LEDList, Board, King, x, 0);
            } else if (Board[x] == 1) {
                LEDGreen(LEDList, Board, x, 0);
            } else if (Board[x] == 2) {
                LEDRed(LEDList, Board, x, 0);
            }
            x++;
        }
    } else {
        findPieces(Board, King, Player);
        findJumps(Board, King, Player, PieceList);
        int count = 0;
        while (JumpList[count] != -1) {
            Serial.println("possible jump, you must move one of the
                          following pieces");
            int row = JumpList[count]/4;
            int column = JumpList[count]%4;
            // do something with the jump
        }
    }
}
int col = JumpList[count] - row*4;
if ((row+2)%2 == 0){//even row(includes 0)
    col = col*2;
} else {//odd row
    col = (col+1)*2-1;
}
Serial.print((char)(row+97)); //in Ascii
Serial.print(',');
Serial.println(col+1);
count++;

Serial.println("Input the row of the piece you wish to move");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 97: //a
        currentRow = 0;
        break;
    case 98: //b
        currentRow = 1;
        break;
    case 99: //c
        currentRow = 2;
        break;
    case 100: //d
        currentRow = 3;
        break;
    case 101: //e
        currentRow = 4;
        break;
    case 102: //f
        currentRow = 5;
        break;
    case 103: //g
        currentRow = 6;
        break;
    case 104: //h
        currentRow = 7;
break;
default:
    Serial.println("unable to recognize your input, please re-enter your turn");
    return;
    //try again
    break;
}
delay(1000);

Serial.println("Input the column of the piece you wish to move");
while (Serial.available() == 0)
{
    //incomingByte = Serial.read();
}
ingoingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
case 49: //1
    currentCol = 0;
    break;
case 50: //2
    currentCol = 1;
    break;
case 51: //3
    currentCol = 2;
    break;
case 52: //4
    currentCol = 3;
    break;
case 53: //5
    currentCol = 4;
    break;
case 54: //6
    currentCol = 5;
    break;
case 55: //7
    currentCol = 6;
    break;
case 56: //8
    currentCol = 7;
    break;
default:
    Serial.println("unable to recognize your input, please
re-enter your turn");
    return;
    //try again
    break;
}
delay(1000);

if(King[4*(currentRow)+currentCol/2] == Player){
    Serial.println("Input 'u' if you wish to move the piece up,
and 'd' if you wish to move the piece down");
    while (Serial.available() == 0){
        //incomingByte = Serial.read();
    }
    incomingByte = Serial.read();
    Serial.println(incomingByte);
    switch (incomingByte) {
    case 117: //u
        currentUorD = 'u';
        break;
    case 100: //d
        currentUorD = 'd';
        break;
    default:
        Serial.println("unable to recognize your input, please
re-enter your turn");
        return;
        //try again
        break;
    }
delay(1000);
}

Serial.println("Input 'l' if you wish to move the piece to the
left, and 'r' if you wish to move the piece to the right");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 108: //l
        currentDir = 'l';
        break;
    case 114: //r
        currentDir = 'r';
        break;
    default:
        Serial.println("unable to recognize your input, please re-enter your turn");
        return;
        //try again
        break;
}
delay(1000);
if (Player == 1) {
    Player = 2; // next players turn
    changeKing(LEDList, Board, King, 2, 1);
    Serial.println("Player 2's turn");
} else {
    Player = 1;
    changeKing(LEDList, Board, King, 1, 2);
    Serial.println("Player 1's turn");
}
}
Serial.println("this move is not possible, please try again");
}
if (player1PieceNum == 0) {
    Serial.println("Player 2 has won!");
    // player 2 wins
} else if (player2PieceNum == 0) {
    Serial.println("Player 1 has won!");
    // player 1 wins
} else {// TWO PLAYER
    if (Player == 1) {
        Player = 1;
        Serial.println("Player 1's turn");
    } else {
        Player = 2;
        Serial.println("Player 2's turn");
    }

    findPieces(Board, King, Player);
    findJumps(Board, King, Player, PieceList);
    int count = 0;
    while (JumpList[count] != -1) {
        Serial.println("possible jump, you must move one of the following pieces");
        int row = JumpList[count]/4;
        int col = JumpList[count] - row*4;
if ((row+2)%2 == 0){//even row (includes 0)
    col = col*2;
} else{//odd row
    col = (col+1)*2-1;
}
Serial.print((char)(row+97)); // in ASCII
Serial.print(",");
Serial.println(col+1);
count++;
}
Serial.println("Input the row of the piece you wish to move");
while (Serial.available() == 0){
    // incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 97: // a
        currentRow = 0;
        break;
    case 98: // b
        currentRow = 1;
        break;
    case 99: // c
        currentRow = 2;
        break;
    case 100: // d
        currentRow = 3;
        break;
    case 101: // e
        currentRow = 4;
        break;
    case 102: // f
        currentRow = 5;
        break;
    case 103: // g
        currentRow = 6;
        break;
    case 104: // h
        currentRow = 7;
        break;
default:
    Serial.println("unable to recognize your input, please re-enter your turn");
    return;
    //try again
    break;
}

delay(1000);

Serial.println("Input the column of the piece you wish to move");
while (Serial.available() == 0){
    //incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 49: //1
        currentCol = 0;
        break;
    case 50: //2
        currentCol = 1;
        break;
    case 51: //3
        currentCol = 2;
        break;
    case 52: //4
        currentCol = 3;
        break;
    case 53: //5
        currentCol = 4;
        break;
    case 54: //6
        currentCol = 5;
        break;
    case 55: //7
        currentCol = 6;
        break;
    case 56: //8
        currentCol = 7;
        break;
    default:
        Serial.println("unable to recognize your input, please re-enter your turn");
        return;
        //try again
        break;
    }
    delay(1000);

    if(King[4*(currentRow)+currentCol/2] == Player){
        Serial.println("Input 'u' if you wish to move the piece up, and 'd' if you wish to move the piece down");
        while (Serial.available() == 0){
            incomingByte = Serial.read();
        }
        incomingByte = Serial.read();
        Serial.println(incomingByte);
        switch (incomingByte) {
            case 117: //u
                currentUorD = 'u';
                break;
            case 100: //d
                currentUorD = 'd';
                break;
            default:
                Serial.println("unable to recognize your input, please re-enter your turn");
                return;
                //try again
                break;
        }
        delay(1000);
    }

    Serial.println("Input 'l' if you wish to move the piece to the left, and 'r' if you wish to move the piece to the right");
    while (Serial.available() == 0){
incomingByte = Serial.read();
}
incomingByte = Serial.read();
Serial.println(incomingByte);
switch (incomingByte) {
    case 108: //l
        currentDir = 'l';
        break;
    case 114: //r
        currentDir = 'r';
        break;
    default:
        Serial.println("unable to recognize your input, please re-enter your turn");
        return;
        //try again
        break;
}

delay(1000);

if(King[4*(currentRow)+currentCol/2] == Player){
    check = checkMoveKing(Board, King, currentRow, currentCol, Player, currentDir, currentUorD, 0, 0, 0);
} else{
    check = checkMove(Board, King, currentRow, currentCol, Player, currentDir, 0, 0, 0);
}
if (check == 1){
    Serial.println("move successful");
    int nextRow = nextPieceNum/4;
    int nextCol = nextPieceNum - 4*nextRow;
    if ((nextRow+2)%2 == 0){//even row(includes 0)
        nextCol = nextCol*2;
    } else{//odd row
        nextCol = (nextCol+1)*2-1;
    }
    if (checkJump(Board, King, Player, nextRow, nextCol)){
        Serial.println("double jump available");
        //must double jump
    } else{
        if (Player == 1){
Player = 2; //next players turn
changeKing(LEDList, Board, King, 2, 1);
Serial.println("Player 2's turn");
}
else{
    Player = 1;
    changeKing(LEDList, Board, King, 1, 2);
    Serial.println("Player 1's turn");
}
}
else{
    Serial.println("this move is not possible, please try again");
}
}
if (player1PieceNum == 0){
    Serial.println("Player 2 has won!");
    //player 2 wins
}
if (player2PieceNum == 0){
    Serial.println("Player 1 has won!");
    //player 1 wins
}
delay(1000);
//Serial.println(incomingByte);