Chess Puzzles and Programming

By John Cuono

Chess is a game that is quite simple to learn, but extremely difficult to master. As such, it oftentimes presents unique opportunities for puzzles that can be solved with the use of logic or math. Solving these puzzles can be challenging, especially when you are searching for a solution with programming in C, which is the central idea of this project.

In Section I of this document, I will explain some of the rules of chess accompanied by visuals, so that even a beginner who has never played chess can understand at least how the pieces move and become familiar with some terminology. In Section II, I will explain a few puzzles that can require heavy amounts of logic and math and propose some ideas on how to solve them. In Section III, I will show what some of these solutions look like in a C program and explain the logic behind the code itself.

All images are taken from the Chess.com analysis board or my C program, and most of the rules I outline here can be found in Bobby Fischer's book "Bobby Fischer Teaches Chess". I will explain any other terminology or obscure rules that aren't entirely necessary to the game.

Section I – The Rules of Chess

This is a chessboard with all of the pieces set up to start an ordinary game. This is often called the "Opening Position". Each board has 64 squares, made up of 32 "light squares" and 32 "dark squares". At the beginning of the game, the player with the white pieces always moves first.

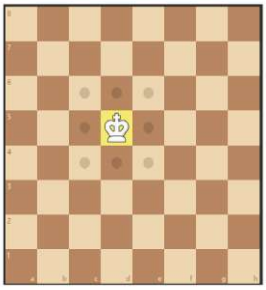
Looking at the top or bottom row of the board, from left to right, there are five unique pieces, being the Rook, Knight, Bishop, Queen, and King. Take note that each player (each player will be referred to as white and black from here on out) has two rooks, two knights, and two bishops.

On the second and seventh rows of the board, there is one piece on all eight squares, called the Pawn. Thus, white and black have a total of 16 pieces each.

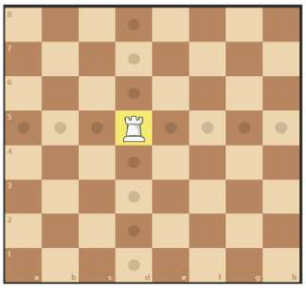
Another important concept to understand is the idea of files and ranks. These are just fancy terms for columns and rows respectively. Files are labelled ‘a’ through ‘h’ and ranks are labeled ‘1’ through ‘8’. For example, the square that white’s king is on in the opening position is “e1”, because it is located on the fifth column of the first row. From black’s perspective, from left to right, the files and ranks will be labeled backwards, from ‘h’ to ‘a’ and '8’ to ‘1’.

Each piece moves differently, and some pieces are more straightforward than others. With one exception which I will discuss later, the king (board 1) can generally move to any of the 8 squares surrounding it. The rook (board 2) has the same exception the king does, but in general it can move any number of squares vertically or horizontally from its current position. The bishop (board 3) can move any number of squares diagonally from its current position, and with this, it should be noted that each player has a “light square bishop” and a “dark square bishop” that can never travel to a square of another color from which they start on. The queen (board 4) combines the movement of the rook and bishop, making it arguably the most valuable piece.

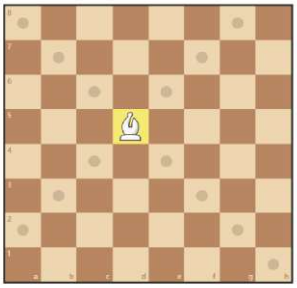
1



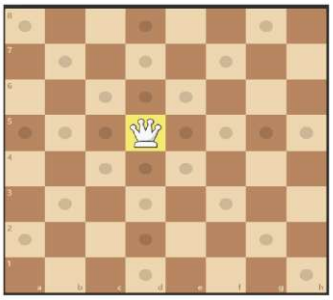
2



3



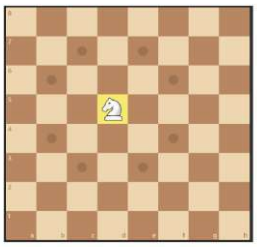
4



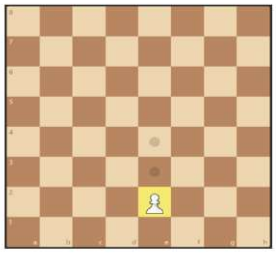
The knight (board 5) can either move two squares vertically and one square horizontally OR two squares horizontally and one square vertically. It is also the only piece that can “jump over” other pieces, meaning that every other piece will be unable to move to a certain square or multiple squares if there is another piece blocking the path, such as on board 8. The knight here can move to c3, ignoring the fact that the path there is taken up by rooks. Conversely, the rook on d4 cannot move anywhere else on the d file because there is a rook directly below it and a knight directly above it. Pawns have the most rules of any piece in chess, including a few rarer

move cases which will be briefly discussed later. Pawns can only move forward and can only move one square at a time (board 7) except for the first move, where they can move one or two squares forward (board 6).

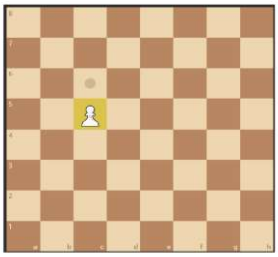
5



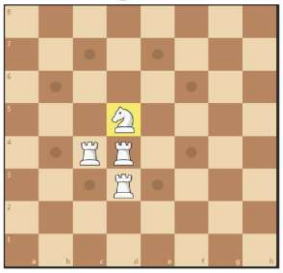
6



7

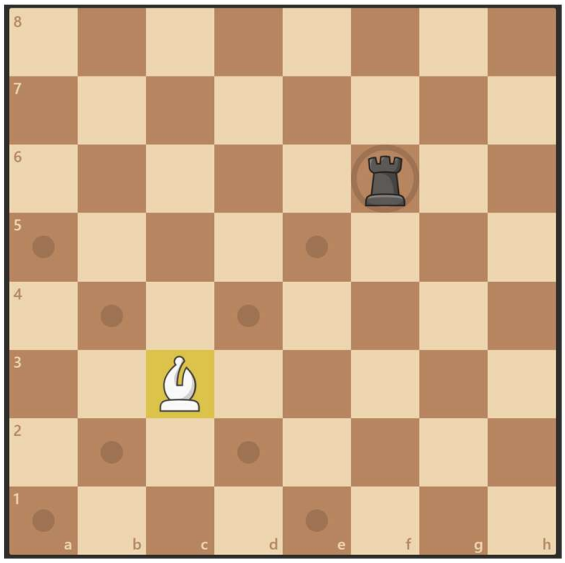


8

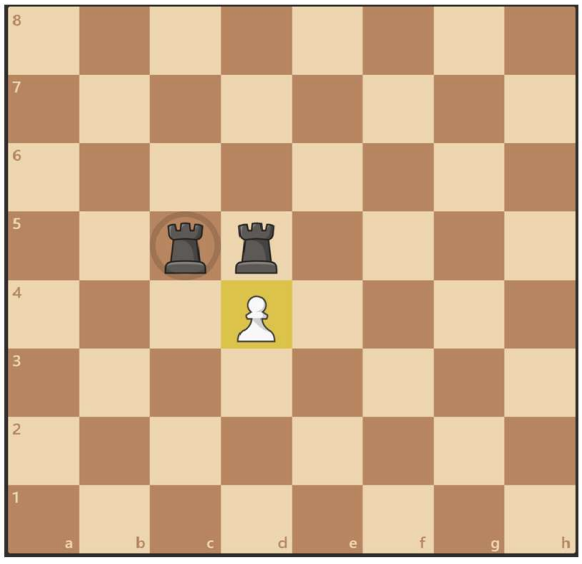


In chess, a "capture" is when a piece of one color takes a piece of another color and removes it from the chessboard. Looking at board 9, you can see that the bishop on c3 can "capture" the rook on f6 if it is currently white's turn because the rook is within the bishop's direct line of sight (the bishop is "attacking" the rook). The pawn is the only piece that captures differently from the way it normally moves. Looking at board 10, you can see that the pawn on d4 can capture the rook on c5, but not the rook on d5, even though this piece is directly ahead of the pawn. From this, it can be gathered that pawns move forward generally but capture diagonally.

9



10



There are a few other key rules of chess that one may want to be familiar with, like pawn

promotion and capturing ”en passant”, conditions for a draw, check and checkmate, and castling. For the sake of this project, I will only further explain castling, since the others don’t really apply to any of the code I’ve written. In chess, castling is the act of moving your king two squares either left or right, and in the process, perform what is essentially a jump over the rook which also moves. (from white’s perspective, the left and right sides of the board are referred to as the ”queenside” and ”kingside” respectively). In board 11, you can see that the king can move both normally and also castle to either side. There are a few conditions for castling which aren’t entirely relevant for this project, but boards 12 and 13 demonstrate what a successful castle to either side looks like.

11

12

13

With all this information in mind, we can now move on to the next section.

Section II – Chess Puzzles and Critical Thinking

The first puzzle I’d like to look at involves a unique form of chess, commonly referred to as Fischer Random. Named after chess grandmaster Bobby Fischer, the game is played as follows:

1. Randomize every back rank piece’s position on only the back rank. White and black’s shuffling should always be a mirror image.
2. If a permutation contains:
   a. Two bishops on the same color square (per player)
   b. A king not placed between the two rooks for castling purposes

   then the permutation is invalid.
3. If a permutation is valid, castling is still allowed, even if it allows the king to move abnormal distances. Queenside castling will always result in the king being on the c file and the castled rook on the d file, and kingside castling will always result in the king being on the g file and the castled rook on the f file.
4. The rest of the game is played as normal.

The main question to answer with this form of chess is: how many different opening permutations / positions are there?

Just as a warmup, let's find the number of possible back rank permutations without the extra restrictions listed in step 2:

1. This is just a combination with repetition. We have eight pieces that need to be randomized, but three of these pieces (rook, knight, bishop) are duplicate pieces.
2. Now, we can simply utilize the combinations with repetition formula: $\frac{n!}{\prod r!}$, where "n" is the total number of pieces to permute (eight) and the denominator is the product of the number of duplicates "r" per piece, factorial.
3. Since there are two rooks, two bishops, and two knights, we find that the number of back rank permutations in general is: $\frac{8!}{2!2!2!} = 7! = 5040$.

Now, let's find the number of possible back rank permutations with the restrictions outlined in Fischer Random:

1. The bishops must be on opposite colored squares, which means that there are four places to place the first bishop and four places to place the second bishop. The number of possible permutations of this subproblem alone is just $4C1 \cdot 4C1 = 16$, since for the first bishop, we choose one square out of four to place it on, then still have four choices for where to place the second bishop.

2. Now that we've placed the bishops, there are six squares to place the remaining pieces. First, let's place the queen on any of these six squares. This is just $6C1 = 6$. Similarly, since we now have five remaining squares, we simply choose two of them to contain the knights. This is just $5C2 = 10$.
3. Speaking a bit more theoretically now, there are three squares remaining. Even if two of these squares are in the corners, it is guaranteed that the third square will be between them, and as a result, not in a corner, and so for every single triplet of remaining squares, from left to right, we place the first rook, then the king, then the second rook. There is only one way to do this for every permutation, so this is just $1C1 = 1$.

4. Now, we just multiply all these results from steps 1-3 together. To intuitively explain why we can do this, remember that for all sixteen bishop permutations, there will be six queen permutations each, and for every queen permutation, there will be ten knight permutations each. Then, for every knight permutation, there will be one way to place the remaining rooks and the king each. This means that we are doing repeated addition, better known as multiplication, so finally, multiplying all the results together gives us:$(4C1 \cdot 4C1) \cdot 6C1 \cdot 5C2 \cdot 1C1 = (4 \cdot 4) \cdot 6 \cdot 10 \cdot 1 = 960$

So, after all this, we can conclude that there are 960 unique permutations for the opening position in Fischer Random. In fact, this number became famous enough within the chess community to eventually give Fischer Random a second, nowadays more commonly used name: Chess960. The approach to programming this puzzle is a little different, but that's something I'll discuss later.

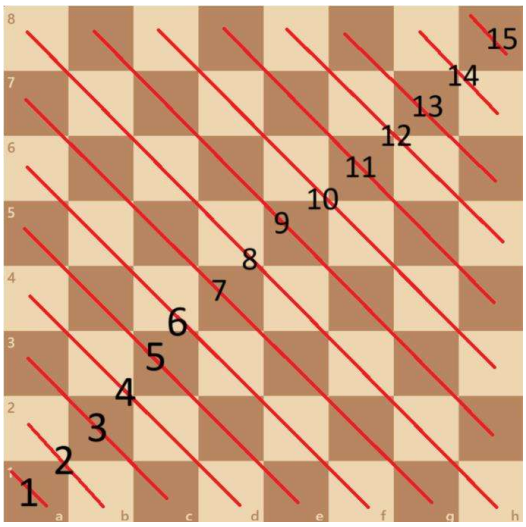
Next, I'd like to look at another puzzle, called the Eight Queens Puzzle. The premise of this puzzle is quite simple: how many ways are there to arrange eight queens on a standard chess board such that no two queens attack each other?

Remembering that a queen can move in any direction vertically, horizontally, or diagonally, let's first find the number of ways to arrange the queens so that no two queens attack each other horizontally and / or vertically.

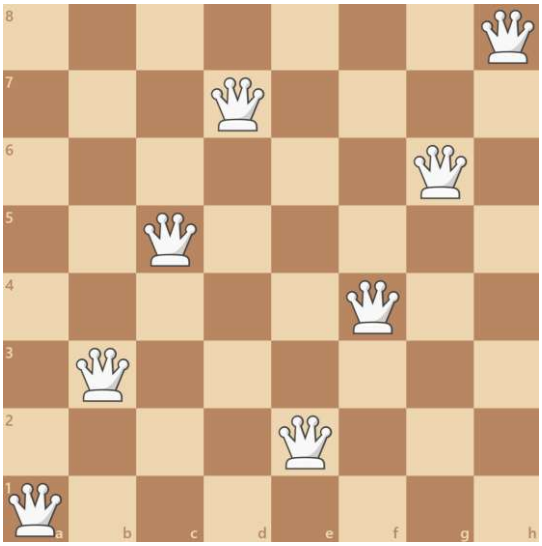
1. On the first file, there are eight squares to choose from, and we place the first queen on any one of these squares. Obviously, since the first queen has now been placed, we cannot possibly place the next queen on the same file, since this would mean they attack each other vertically. We now look to the second file, where we have seven squares to choose from, since one square will be horizontally under attack from the first queen.
2. We can apply this logic all the way down the ladder of placing queens, and this intuitively means there are: $8C1 \cdot 7C1 \cdot 6C1 \cdot 5C1 \cdot 4C1 \cdot 3C1 \cdot 2C1 \cdot 1C1 = 8! = 40320$ ways to place all the queens so that no two queens attack each other horizontally and vertically.

As a quick sidenote, if we apply this exact logic with rooks instead of queens, we find that, since rooks are just queens without the ability to move diagonally, there are exactly 40320 solutions to this so-called "Eight Rooks Puzzle".

Back to the Eight Queens Puzzle, the last roadblock to overcome is accounting for the diagonals. On a standard chessboard, there are exactly 15 diagonals going in the same direction, shown below:



We may try to consider a solution that utilizes these diagonals, but there is unfortunately no known formulaic way that can give the exact number of solutions to this puzzle. The diagonals can still be helpful with analyzing certain patterns in the puzzle though. Take the following solution as an example:



Here, we can see that diagonals 1, 4, 6, 7, 9, 10, 12, 15 have queens on them. If we map these values to an array, we get a nice symmetrical distribution:

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15

Q || || Q || Q Q || Q Q || Q || || Q

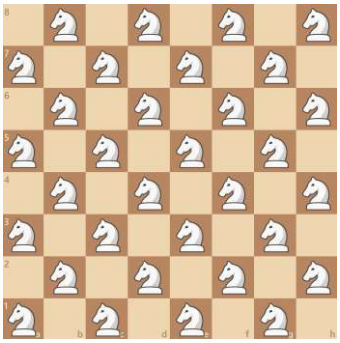
Symmetrical solutions like this are not guaranteed, but there may be some more interesting properties behind the distribution of queens on the diagonals that are beyond the scope of this project.

From the example board, we can also see that the queens form what one could call a "staircase" in two unique sections. Every queen on the staircase is separated by a knight's move distance, which is the smallest possible distance mathematically speaking that a queen could be without attacking another. This distance is just $\sqrt{2^2 + 1^2} = \sqrt{5}$.

As for the actual number of solutions to the puzzle, this number is known and can be found using a brute force approach, but I'll save this for the programming section of the document.

We've now covered this puzzle type with eight queens and eight rooks. One may want to also consider solutions to a similar problem: what is the maximum number of a specific piece that can be placed on a board so that no two pieces attack each other?

Let's start with knights, which may be the simplest piece to find an answer to. Since knights always attack a square with the opposite color of their current square, we find that a solution is just:

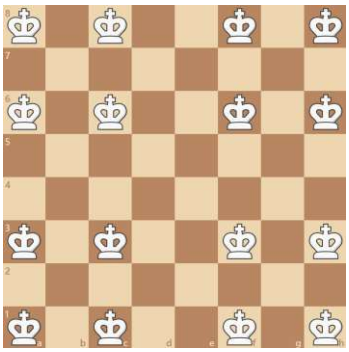


This means that we can have up to 32 knights on the board without any knight attacking the other, and the only other solution to this problem would be to place all the knights on light squares instead of dark squares.

For kings, we must do a little more math.

1. A chessboard has 64 squares, and at the very least, a king will attack 3 other squares if it's placed in the corner.
2. If we place four kings in the corners, we now must consider the next-least number of attacked squares when placing a king. Intuitively, this is the edge of the board, where only five other squares are attacked upon placement.
3. If we place the maximum number of kings on the edge of the board, we are guaranteed to have a 4x4 area in the center of the board where kings can be placed on any of the squares.
4. No matter how we place the kings in the 4x4 area in the center of the board, we will always get exactly four possible king placements. This leaves us with a total of 16 kings that can be placed such that no two kings attack each other.

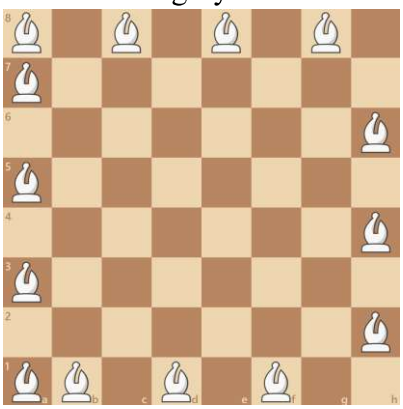
One of the many solutions to this problem looks like this:



For the sake of keeping this document more concise, I won't attempt this problem with bishops methodically, but knowing from earlier that there are 15 diagonals is quite helpful.

In fact, it's impossible to have a bishop on every diagonal, since this guarantees that at least two bishops attack each other on their respective counter-diagonals. For example, diagonals 1 and 15 are both on the same counter diagonal running from the top right to the bottom left of the board.

As a result of this logic, we know that 15 bishops are impossible, but 14 is indeed possible, just from checking by brute force. An example of this is shown below:



Notice how every diagonal except for diagonal 15 is taken, and visually, we can count 14 bishops not attacking each other.

Moving on from this type of problem, there is one more puzzle I'd like to briefly discuss, which is famously known as the Knight's Tour Puzzle. The premise of this puzzle is simple:

Is it possible for a knight to visit every square on a chessboard exactly once? In other words, can we find a path for a knight to move across every square without ever revisiting the same square?

On a standard chessboard, the answer is yes, but finding a working solution can be quite tedious. Let's try to work out some ideas:

1. To start, we know that a knight has at most eight legal moves, and at least two legal moves. This could be helpful in a scenario where a knight's potential next move may trap it, since we aren't allowed to move backwards in the puzzle.
2. Intuitively, one might think that if a knight has more than one legal move, we should choose the square that gives the knight the most legal next moves.

However, there is a principle known as Warnsdorf's Rule, which essentially suggests that every next knight move should be to the square with the least number of next possible moves. This goes completely against what our intuition would tell us, but the reasoning for it in such a complex puzzle makes perfect sense:

If we choose the squares with the lowest numbers of legal next moves, this prioritizes reaching the edges and corners of the board first. This is ideal, since visiting these squares later may prove to be problematic, due to the limited number of ways to access them.

For example, if we enter a corner square late in a simulation of the Knight's Tour, the only exit square may already be blocked off. The faster we check off these squares, the less likely we are to be trapped. Warndorf's Rule is only a heuristic approach and not really algorithmic, but on a standard 8 x 8 chessboard, it is guaranteed to give working solutions from any starting point without the need to backtrack and fix our pathing.

Section III – Programming Chess Puzzles

Of all the problems I discussed earlier, I have programmed three of them for further analysis, those being:

1. The number of permutations of the opening position in Fischer Random / Chess960
2. The number of solutions to the Eight Queens Puzzle, with visuals
3. A Knight's Tour simulator which utilizes Warndorf's Rule to find a solution

Let's first look at the Fischer Random section of the C program:

```
void permutePieces(int piecesArray[], int helperArray[], int counter, int size, char pieces[])
{
    if(counter == size && fischerCheck(piecesArray, size, pieces))
    {
        totalPermutations++;
    }

    for(int i = 0; i < size; i++)
    {
        if(!helperArray[i])
        {
            helperArray[i] = 1;
            piecesArray[counter] = i;
            permutePieces(piecesArray, helperArray, counter + 1, size, pieces);
            helperArray[i] = 0;
        }
    }
}
```

This code is at the center of the Fischer Random puzzle and utilizes a helper array to effectively check all permutations of the back rank. I won't elaborate much on this helper array, since it is essentially standard practice for checking all permutations of anything in C.

The much more interesting aspect of this code lies in the fischerCheck() function, which lies in the top "if" statement of this permutePieces() function. This fischerCheck() function will internally determine whether a permutation of the back rank is valid or not. Let's dive into the code for this function:

```
for(int i = 0; i < size; i++)
{
    if((pieces[piecesArray[i]] == 'B' && bishopTracker) || (pieces[piecesArray[i]] == 'N' && knightTracker) || (pieces[piecesArray[i]] == 'R' && rookTracker))
    {
        if(piecesArray[i] > SIZE / 2)
        {
            return 0; //by returning 0 for every piece with index > size / 2, we are guaranteed to avoid exactly half of the technically valid permutations
        }
        else if(pieces[piecesArray[i]] == 'B')
        {
            bishopTracker = 0; //disable the bishop tracker
            continue;
        }
        else if(pieces[piecesArray[i]] == 'N')
        {
            knightTracker = 0; //disable the knight tracker
            continue;
        }
        else if(pieces[piecesArray[i]] == 'R')
        {
            rookTracker = 0; //disable the rook tracker
            continue;
        }
    }
}
```

In this code segment, we take a much different approach to finding duplicates than we did in Section II. The code runs a few steps to immediately get rid of duplicates:

1. The array piecesArray[] is initialized in the permutePieces() function to hold a permutation of numbers 0-7 which represent the index of the pieces in the array pieces[], which contains pieces in the order {R, N, B, Q, K, B, N, R}.
2. In the first "if" condition of the for loop, we check if each permuted index of pieces[] holds a bishop, knight, or rook. If so, we must check more conditions.
3. In theory, we can just say that if the randomly permuted array of pieces contains a bishop, knight, or rook that is also contained at an index greater than 4 on the right side of a regular chessboard, it will be a duplicate, since in code, we can't distinguish one of these pieces from its counterpart without explicitly excluding one of the two.

4. From here, we just work through the logic of the code. If any bishop, knight, or rook is contained in an index greater than the array size (8) divided by 2 (which equals 4), we immediately return 0 and say the permutation is invalid.
5. If we find one of these same pieces, but it is not contained at an index greater than 4, we simply disable its tracker, so that we know that we found the "correct" piece first as opposed to its counterpart.
6. If we successfully complete this whole loop with all three pieces, we know that we have a "good" solution and can continue in the code.

This code is essential, since without it, we'd be counting the same solutions more than once because each piece that has a duplicate is indistinguishable from its counterpart without checking its index.

In the next section of code, we have two unique cases to check for:

```
if(pieces[piecesArray[0]] == 'K' || pieces[piecesArray[size - 1]] == 'K')
{
    return 0; //king cannot be on the ends because it must be surrounded by rooks
}

for(int i = 0; i < size - 1; i++) //checking bishops
{
    if(pieces[piecesArray[i]] == 'B') //first bishop
    {
        for(int j = i + 1; j < size; j++)
        {
            if(pieces[piecesArray[j]] == 'B') //second bishop
            {
                if(i % 2 == j % 2) //if bishops are on the same color square
                {
                    return 0; //bishops have to be on opposite colored squares
                }
            }
        }
    }
}
```

First, we simply check to see if the king in the current permutation is on a corner of the board, which as previously discussed, is an impossible configuration in Fischer Random. If it is, just return 0, and if not, we proceed to check the bishops.

This section is fairly easy, since if we understand that we can map each dark square to an odd number and each light square to an even number (or vice versa), to make sure that the bishops are on opposite-colored squares, we:

1. Locate the first bishop. From there, start another for loop starting at the next index to find the second bishop.
2. Once we have the indexes of both bishops, check if their indexes % 2 (% meaning modulo) are the same. If they are, that means both bishops were on the same-colored square, since an even number % 2 will always equal 0, whereas an odd number % 2 will always equal 1.

This simple nested for loop will effectively get rid of all bishop permutations that are invalid. To finish off this section of the program, we now check the configurations of the rooks and the king:

```
for(int i = 1; i < size - 1; i++) //dealing with kings and rooks problem
{
    if(pieces[piecesArray[i]] == 'K') //find position of king
    {
        for(int j = 0; j < i + 1; j++) //search for rook before king
        {
            if(j == i)
            {
                return 0;
            }
            else if(pieces[piecesArray[j]] == 'R') //if rook before king is found
            {
                for(int k = i + 1; k < size + 1; k++) //search for rook after king
                {
                    if(k == size) //if we reach the end of the array without success
                    {
                        return 0; //no rook was found after the king
                    }
                    else if(pieces[piecesArray[k]] == 'R')
                    {
                        break; //successful combination found so break
                    }
                }
                break; //finishing breaking
            }
        }
    }
}
```

To start, the first for loop starts at index 1, and only checks up to index 6. This is because we earlier terminated this function if a king was located at index 0 or 7. From here, we check the following:

1. Is the current index the king? If not, increase the index by one and check again.
2. If we find the king, we start checking for the first rook. Since we know that this rook must be to the left of the king for the permutation to be valid, we start checking from index 0. If we reach the king but no rook was found, we know that the permutation is invalid, since both rooks in this instance are located to the right of the king.
3. Otherwise, if we find the first rook before the king, start another for loop at the index of the king plus one. Iterate through the array, and if the second rook is found, the permutation is 100% valid and can now be printed and added to the total. If the second rook isn't found, this means that both rooks were to the left of the king, and so we just return 0.

From here, we just print each solution and count how many valid permutations we find. Here is the final result of this code, along with a few examples of valid array permutations:

```
Array 957: Q B B N R K N R
Array 957: 3 2 5 1 0 4 6 7

Array 958: Q B B N R K R N
Array 958: 3 2 5 1 0 4 7 6

Array 959: Q B B N R N K R
Array 959: 3 2 5 1 0 6 4 7

Array 960: Q B B N N R K R
Array 960: 3 2 5 1 6 0 4 7

Total Permutations In Fischer Random: 960
```

As you can see, the program does indeed find the correct number of solutions and prints all of them. Just looking at the last few here, the sequence of letters represents a valid orientation of pieces for Fischer Random, and the number under each piece represents the original index of the piece in the array pieces[].

Next, let's look at the code for the Eight Queens Puzzle. Similarly to the Fischer Random code, we start this puzzle with a helper array to recursive iterate through all permutations of queens:

```
void eightQueens(int queensArray[], int helperArray[], int counter, int size)
{
    if(counter == size)
    {
        totalSolutions++;
        printf("Solution %d:\n", totalSolutions); //update and print solutions
        for(int i = 0; i < size; i++)
        {
            for(int j = 0; j < size; j++)
            {
                if(queensArray[j] == i)
                {
                    printf("Q ");
                }
                else
                {
                    printf("/ ");
                }
            }
            printf("\n");
        }
        printf("\n");
    }

    for(int row = 0; row < size; row++) //perform recursion
    {
        if(!helperArray[row] && queensValidity(queensArray, size, row, counter))
        {
            helperArray[row] = 1;
            queensArray[counter] = row;
            eightQueens(queensArray, helperArray, counter + 1, size);
            helperArray[row] = 0;
        }
    }
}
```

The main differences here are that we print the solutions within this eightQueens() function since we are printing slightly different output, and that we are checking if a permutation is valid this time with the queensValidity() function.

Now, let's take a look at this queensValidity() function, which is surprisingly brief:

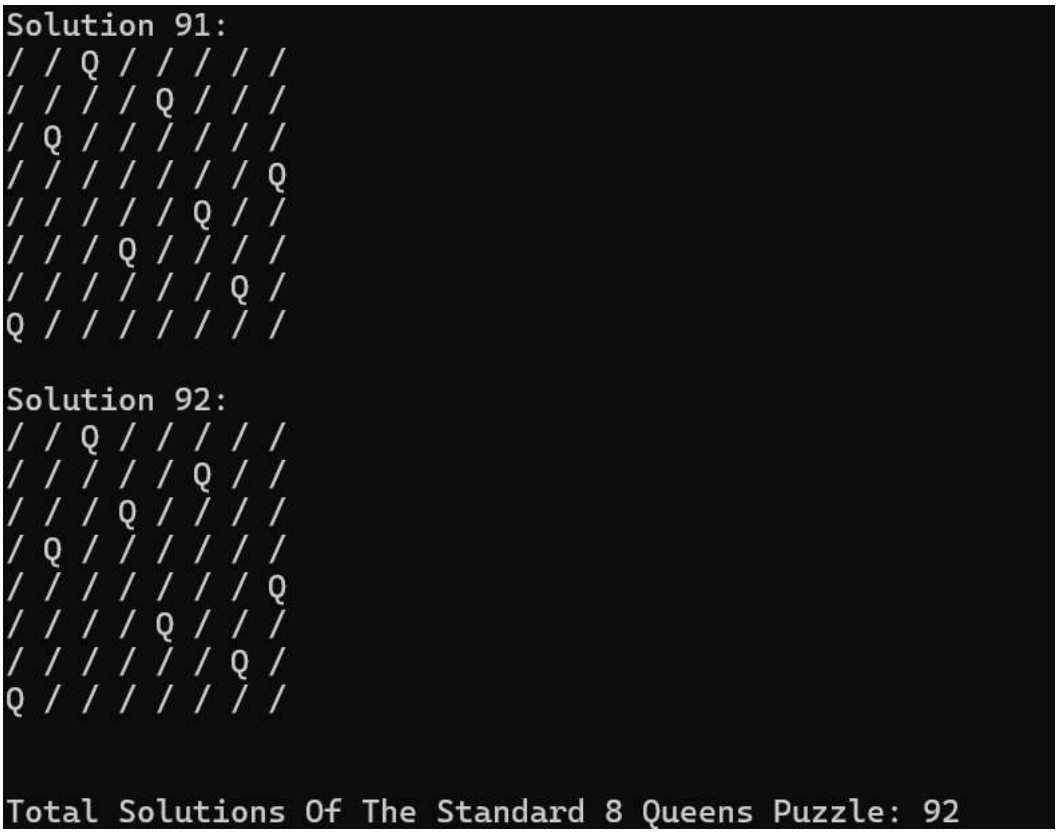
```
int queensValidity(int queensArray[], int size, int row, int counter)
{
    for(int column = 0; column < counter; column++)
    {
        if(abs(counter - column) == abs(queensArray[column] - row))
        {
            return 0; //if the distance between two points has the same x and y, it must be diagonal so it can't be a solution
        }
    }

    return 1;
}
```

And, although brief, the code is bit hard to decipher off the bat. Let's try to break this down:

1. The value of row is dependent on the iteration of the for loop in the eightQueens() function, but essentially acts as exactly that, the value of the current rank on the chessboard.
2. The column variable in the for loop is used to check an index in the array queensArray[]. It's important to note that we won't be checking for any columns past the current counter value (where the counter is representative of the current column in the main eightQueens() function).
3. The actual comparison made in the "if" statement is checking the diagonals of the board, just not in a very intuitive way. If the current column (represented with the counter variable) minus the iterative column variable is equal to queensArray[column] minus the current row, the solution is invalid.

After this, we just return whether a successful permutation was found or not, using 0 and 1. If the permutation is valid, we continue iterating through the recursion until the column counter is equal to the length / height of the board. This implies that a solution has been found, since all eight columns were successfully traversed with recursion.

Here is what some of the solutions look like, along with a final answer to the mystery of how many solutions there are to this Eight Queens Puzzle:



As you can see, the permutations being printed are valid, and it turns out there are 92 unique solutions to the Eight Queens Puzzle on a standard 8 x 8 chessboard. Although I didn't mention it earlier, this code can be applied to find the number of solutions to the more general "n Queens Puzzle", where the "n" represents the length or height of the n x n chessboard, which shows the power of brute force recursion whereas we couldn't even deduce 92 solutions formulaically.

Finally, let's discuss the code for the Knight's Tour Puzzle:

```
void knightsTour()
{
    int knightsBoard[SIZE][SIZE]; //board to store move order
    for(int i = 0; i < SIZE; i++) //initialize the board
    {
        for(int j = 0; j < SIZE; j++)
        {
           knightsBoard[i][j] = -1;
        }
    }
    int moveCounter = 0;

    int xCord;
    int yCord;

    printf("Enter a square to begin on using coordinate notation: ");
    scanf("%d %d", &yCord, &xCord);

    while(yCord < 0 || yCord > 7 || xCord < 0 || xCord > 7)
    {
        printf("Enter a square to begin on using coordinate notation: ");
        scanf("%d %d", &yCord, &xCord);
    }

    knightsBoard[xCord][yCord] = moveCounter; //starting square
    printChessBoardKnights(knightsBoard);
    moveCounter++;

    solveKnightsRec(xCord, yCord, moveCounter, knightsBoard);
}
```

Here, we can see the initial code for the setup of this puzzle. We initialize the chessboard to all negative 1s, gather input from the user, and check to make sure the input is valid. Specifically, this input will be a starting (x,y) coordinate which is inputted in the form "x y". This coordinate pair will be where the knight begins its so-called "tour".

We also set the starting square equal to 00, representing the total number of moves that have currently been made. For example, wherever the knight moves next will be marked as 01, to signify that the knight has made one move thus far.

From here, we call the solveKnightRec() function to perform the actual tour. Here is the code for a section of this function:

```
int solveKnightsRec(int xCord, int yCord, int moveCounter, int knightsBoard[SIZE][SIZE])
{
    printChessBoardKnights(knightsBoard);
    if(moveCounter == SIZE * SIZE) //if all squares have been visited
    {
        printChessBoardKnights(knightsBoard);
        return 1;
    }

    int maxPossibleMoves = 9; //This utilizes Warndorf's rule. Since a knight can't have 9 options ever, we initialize to 9 for comparison later

    for(int i = 0; i < 8; i++) //tries to find the next square with the least number of moves, so that backtracking can be more efficient
    {
        int nextXCord = xCord + movesArray[i][0]; //temp coordinates to check each next square
        int nextYCord = yCord + movesArray[i][1];

        if(knightsValidity(nextXCord, nextYCord, knightsBoard) == 1) //if we can move to the square
        {
            int lowestPossibleMoves = knightsPossibleSquares(nextXCord, nextYCord, knightsBoard); //set to the number of possible moves on a particular check
            if(lowestPossibleMoves < maxPossibleMoves) //if this new number is lower than anything we've found thus far
            {
                maxPossibleMoves = lowestPossibleMoves; //update the new max possible moves for later
            }
        }
    }
```

This part of the function utilizes Warndorf's Rule to find the best next move. In essence, we check the base case (have we travelled the entire board?). If the base case is false, we initialize a

variable called maxPossibleMoves to nine. A knight can only have a maximum of eight different legal moves, meaning that this value can be used later for comparison purposes.

From here, we follow a somewhat easy-to-follow procedure:

1. Enter a for loop which will run eight times, representing the number of squares that must be checked for the knight's next move.
2. We initialize two variables which represent a temporary coordinate that must be checked for the knight. We know this coordinate needs to be checked because we are adding the current coordinate pair to a coordinate pair in the array movesArray[][], pictured here:

```
int movesArray[8][2] = {{2,1}, {1, 2}, {-1, 2}, {-2, 1}, {-2, -1}, {-1, -2}, {1, -2}, {2, -1}}; //possible knight moves
```

   This 2D array contains all eight different possible knight moves relative to where a knight is currently located, and so by adding the values of this array to the knight's current coordinates, we are guaranteed to get a valid knight move (unless we are out of bounds, but this will be checked later).
3. After this, we check if the square we're checking is a valid move option. That is done in the knightsValidity() function, shown here:

```
int knightsValidity(int xCord, int yCord, int knightsBoard[SIZE][SIZE])
{
   if(xCord >= 0 && xCord <= 7 && yCord >= 0 && yCord <= 7 && knightsBoard[xCord][yCord] == -1)
   {
       return 1; //if in bounds and square hasn't already been visited
   }
   return 0;
}
```

   As long as the move we're checking is in bounds and has a value of −1 (meaning it hasn't yet been visited), we are okay to proceed further in the solveKnightsRec() function.
4. Since we know the square we're checking is valid, the next task is to check how many possible next moves we have from this square, which is where Warnsdorf's Rule comes into play. This knightsPossibleSquares() function looks like this:

```
int knightsPossibleSquares(int xCord, int yCord, int knightsBoard[SIZE][SIZE])
{
    int possibleCounter = 0;

    for(int i = 0; i < 8; i++) //check how many of the next moves possible are valid
    {
        int nextXCord = xCord + movesArray[i][0];
        int nextYCord = yCord + movesArray[i][1];
        if(knightsValidity(nextXCord, nextYCord, knightsBoard) == 1)
        {
            possibleCounter++;
        }
    }
    return possibleCounter;
}
```

   This function just checks how many possible moves we have from the current square we're checking, then returns a number representing the total possible moves.
5. Since Warndorf's Rule says that we want to travel to the square with the least amount of possible next moves, we make a final comparison, which updates the aforementioned

maxPossibleMoves variable to the lowest number we could find between each next square's possible moves.

This code is easy to read but contains a lot of logical thinking. The next part of the solveKnightsRec() function just performs the actual recursion. Here is the code:

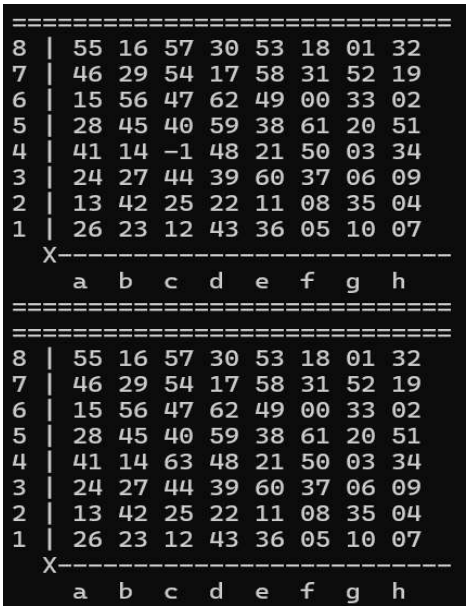
```
for(int i = 0; i < 8; i++) //this actually performs the recursion
{
    int nextXCord = xCord + movesArray[i][0]; //temp coordinates to check each next square
    int nextYCord = yCord + movesArray[i][1];

    if(knightsValidity(nextXCord, nextYCord, knightsBoard) == 1 && knightsPossibleSquares(nextXCord, nextYCord, knightsBoard) == maxPossibleMoves)
    {
        knightsBoard[nextXCord][nextYCord] = moveCounter; //this move is guaranteed to be successful
        if(solveKnightsRec(nextXCord, nextYCord, moveCounter + 1, knightsBoard) == 1) //recursion
        {
            return 1;
        }
    }
}

return 0;
```

This final code segment starts a for loop, once again running eight times, which is the maximum number of legal knight moves. The rest of the code runs as follows:

1. Initialize a few variables, similar to what we did earlier, which just represent the squares that we're going to be checking.
2. Now, inside the "if" statement, we make sure once again that the square we're checking is valid, but also just as importantly, whether the number of potential next moves from this next square is equal to the value we found from Warndorf's rule earlier.
3. If a move is a match, we finally update the chessboard on the new square we're moving to, which represents the move number.
4. Then we just call the solveKnightsRec() function again recursively, which just performs the same tasks for every single square until the tour is finished (by reaching the base case).

From here, the code is all done. You may have noticed in the earlier code segment that we were printing the chessboard after every move was completed. Here's what a final board from a knight

starting at the coordinate (5,5) (or f6 in chess notation) would look like:



As you can see, the top board represents the penultimate move of the knight, where there is only one –1 left on the board. You can also see that up two spaces and right one space from this –1 is move 62. It takes 63 moves from the starting coordinate to finish the Knight's Tour, so ultimately on the bottom board, you can see that the final move is made and the –1 is replaced with a 63, finishing the tour.

A final, somewhat interesting observation one could make is that since we started on (5,5) (or f6) on move 00 (which is even), every index representing a dark square will contain an even number, and every index representing a light square will contain an odd number, since knights alternate which color square they are on every move. Had we started on (5,6) (or f7), every dark square would contain an odd number, and every light square would contain an even number.

Final Summary:

Section I: Went through the basic rules and terminology of chess needed to understand the puzzles and problems that would come in the later sections.

Section II: Looked at several different chess puzzles and found mathematical or brute force ways for tackling most of them, utilizing logic and critical thinking skills.

Section III: Reviewed the C program I wrote which shows solutions to some of the problems we faced in the previous section, including a few problems which had unobvious or unknown answers.