# Overview

When approaching how to write this method, which may be the most important method of this game, I realized I did not want to write a solution that would be O(n2) or worse, which would result from a natural approach to this problem.

My first “breakthrough,” of sorts, is that I did not have to write a general solution to a connect four game, but rather only one for this grid size.

One of the first lines of my isWin() method checks if at least seven chips have been played. Regardless of the orientation of the chips, you cannot have a win unless one player has played at least 4 chips, meaning the game will have at least 7 chips on the board.

```
//Checks if atleast 7 chips have been played
if (ViewController.chipsPlayed < 6) { return false }
```

Next I decided to check the row and column *respective to the placed Chip*. Notice how my isWin() method takes in a row, column, and Chip. By obtaining the row and column I am able to only loop through the Chip’s respective row and column and therefore reduce the need to iterate over the entire board.

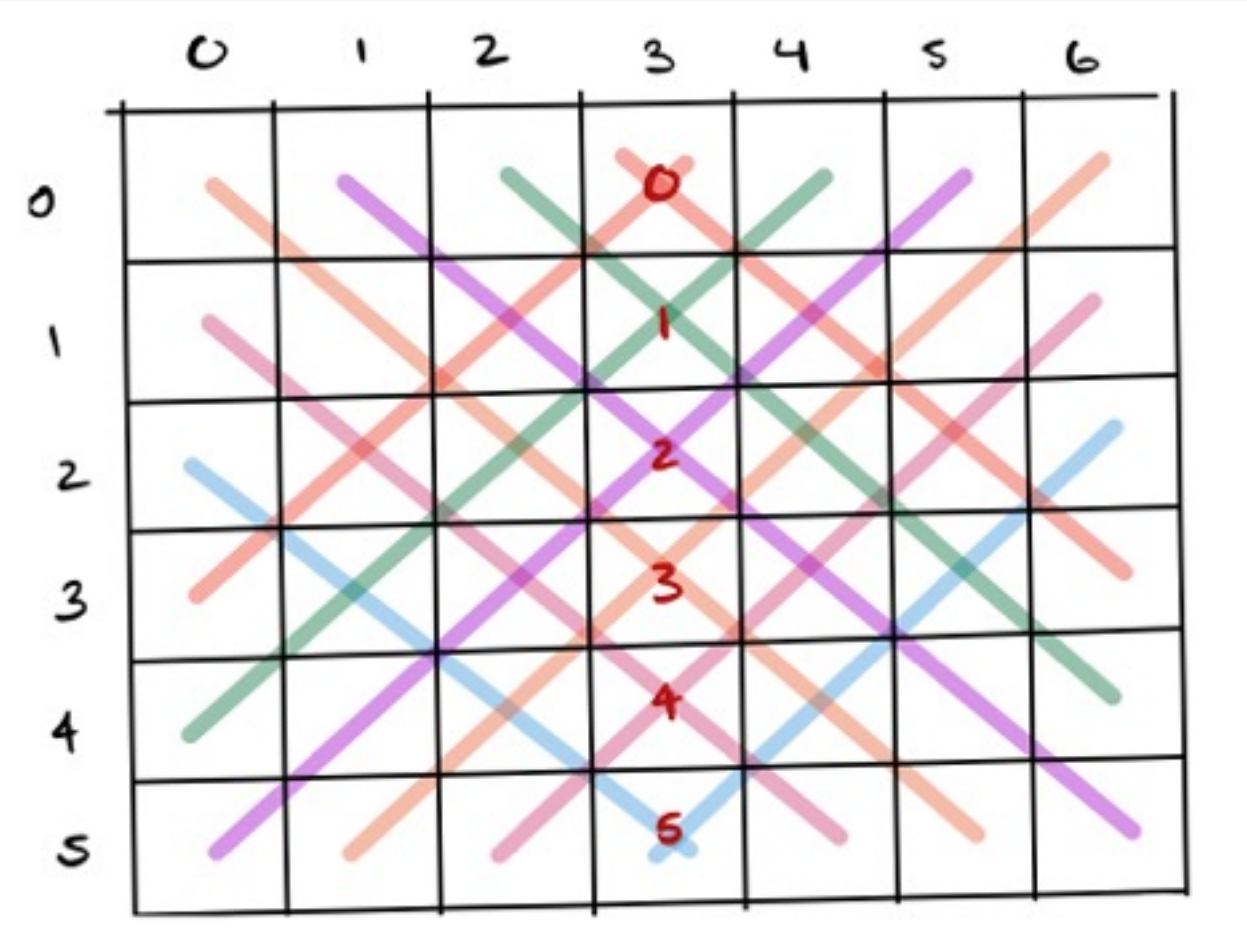
```
//Checks the column
for i in (0..<3) {
    if ((spaces[column][i] == ChipColor) && (spaces[column][i + 1] == ChipColor) && (spaces[column][i + 2] ==
        ChipColor) && (spaces[column][i + 3] == ChipColor)) {
        return true
    }
}

//Checks the row
for i in (0..<4) {
    if ((spaces[i][row] == ChipColor) && (spaces[i + 1][row] == ChipColor) && (spaces[i + 2][row] == ChipColor) &&
        (spaces[i + 3][row] == ChipColor)) {
        return true
    }
}
```

**Ari Jain**

Now it's time for the hard part, checking the diagonals. I took a couple different approaches to this, and although my final version does have more lines of code, it is the most efficient solution I could think of.

Let's start by looking at the layout of the board. Here is every possible diagonal that could contain 4 in a row. Notice how for column three, there are two diagonals that intersect. Here is where I got the naming convention for my diagonal checker methods ("diagonalRow5"… "diagonalRow0").



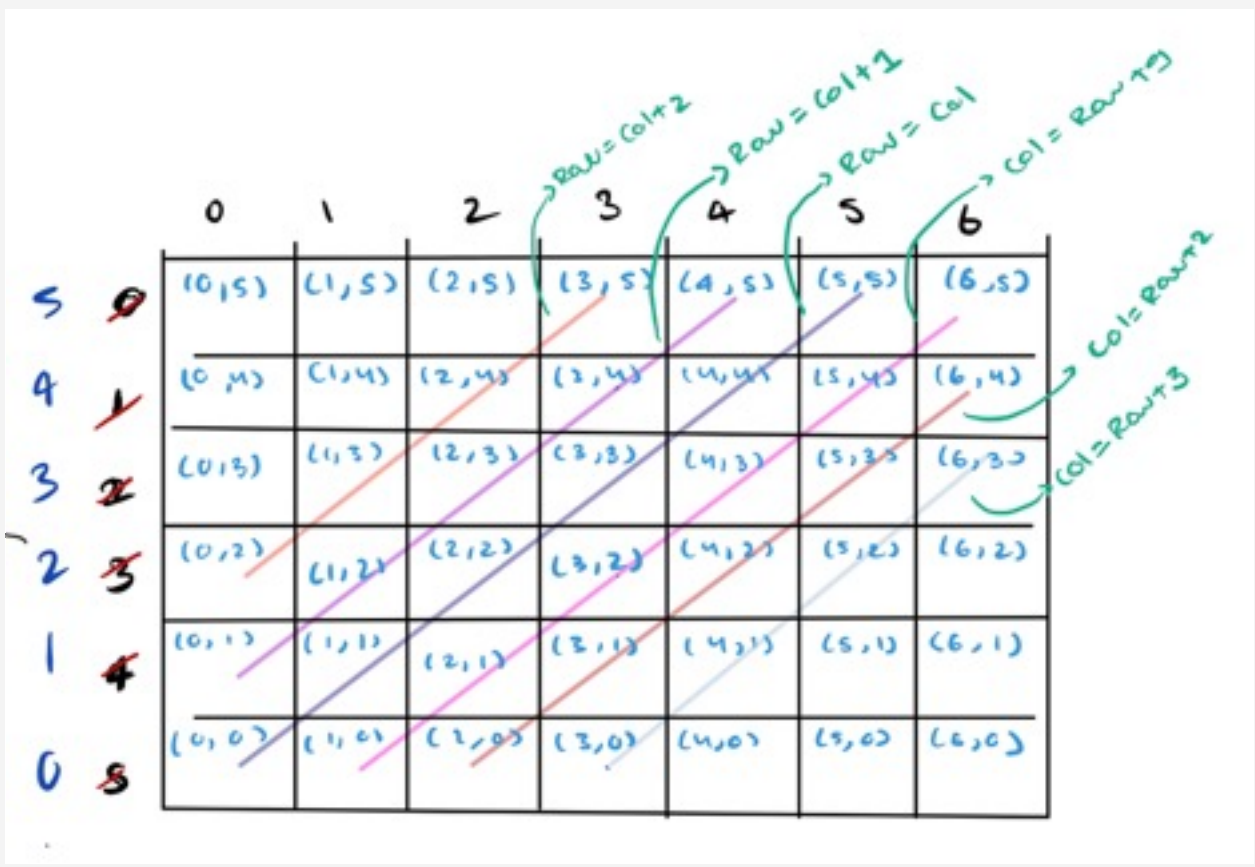

Next, we create a function that checks a Chip's respective diagonal(s) based upon where the chips were placed. You don't want to have to check every single diagonal, but instead only the diagonals that the chip lies in, just like how we only check the Chip's respective row and column.

But how do we know what diagonals to check for? We need to figure out a relationship between the row and column of each diagonal, and then we can write a function that will check if the diagonal has a win.

**Ari Jain**


Here we have shown the relationship between the column and row coordinates of the diagonals. Now that we know what we are checking for, we can call the correct ‘diagonal row win’ function for the respective point. This allows us to not have to iterate over all the diagonals, and only the one needed.

As I’m sure you noticed, the above picture only depicts have the diagonals, we still need to figure out relationships between the points of the other half.




**Ari Jain**

Notice how there is no relationship between the points on each respective diagonal. In order for me to continue to use the logic I used above, I had to get creative.

I saw that if I flipped the y-axis there would be a relationship between the row and column coordinates again. Checking for this would be easy as well, as I could just replace (column, row) with (column, heightOfBoard – 1- row).

Now I have successfully created conditions to filter each point so that I only have to check the diagonals that the Chip is currently in. Thereby guaranteeing an **O(n)** solution to checking if there is a win for this game of Connect Four! I hope this walk through was clear as can be!

**Ari Jain**