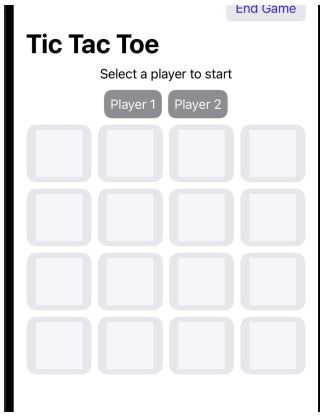
Q1a

To change the board to a 4x4 board, the first thing I thought about was changing the size of the square so that the screen can fit four to a row. Next, I modified the reset function in the GameSquare file, to make sure it initializes 1 to 16 inclusively for the gameboard.




```
static var reset:[GameSquare]{
    var squares = [GameSquare]()
        for index in 1...16 {
            squares.append(GameSquare(id: index))
        }
        return squares


}
```

To ensure that the screen will display the squareView in a 4 by 4 grid, I then edited the VStack in the GameView file. Each row would have 1 more square and an extra row of 4 squares is added. The indices were modified to match the labeling of squares that go from left to right, up to down.

```
//create game board here
VStack{
    HStack{
        ForEach(0...3, id: \.self){
            index in SquareView(index: index)
        }
    }

    HStack{
        ForEach(4...7, id: \.self){
            index in SquareView(index: index)
        }
    }

    HStack{
        ForEach(8...11, id: \.self){
            index in SquareView(index: index)
        }
    }
    HStack{
        ForEach(12...15, id: \.self){
            index in SquareView(index: index)
        }
    }
}
//
```

```
func calcWinningMoves() ->[[Int]] {
    var ans:[[Int]] = []
    // loop through and add all row combinations
    for i in 0..<4 {
        let row1 = [i*4+1, i*4+2, i*4+3]
        let row2 = [i*4+2, i*4+3, i*4+4]
        ans.append(row1)
        ans.append(row2)
    }

    // columns
    for i in 0..<4 {
        let col1 = [i+1, i+5, i+9]
        let col2 = [i+5, i+9, i+13]
        ans.append(col1)
        ans.append(col2)
    }

    let diag1 = [1, 2, 5, 6]
    let diag2 = [3, 4, 7, 8]

    // upper left to bottom right diagonals
    for i in diag1 {
        ans.append([i, i+5, i + 10])
    }

    // upper right to bottom left diagonals
    for i in diag2 {
        ans.append([i, i+3, i+6])
    }

    return ans
}
```

Next, the gameModel was edited so that all 3-in-a-row combinations were accounted for. Instead of writing out all 24 combinations of winning moves, I wrote out a function that could simplify the process of writing the array of arrays of all combinations. For the horizontal 3-in-a-row combinations, each row would have two, so the first for loop iterates through the four rows and adds the combinations to the ans nested array. The columns also have two combinations each, so the second for loop iterates through the rows and adds the combinations for each column. For the diagonals, I wrote out the indices of each square of the grid, and found that there are four 3-in-a-row diagonals in each direction. The diagonals that go from upper left corner to bottom right corner all have indices that increment by 5, while the other direction have indices that increment by 3. I saved the smallest index of each diagonal in an array and used a for loop to get the combinations. The function returns an array of arrays which is then called for the winningMoves variable that already existed in the original code.

```
91  enum Moves{
92      static var all = [1,2,3,4,5,6,7,8,9]
93
94  //    static var winningMoves = [[1,2,3],[4,5,6],[7,8,9],
95  //    [1,4,7],[2,5,8],[3,6,9],
96  //    [1,5,9], [3,5,7]]
97
98      static var winningMoves = calcWinningMoves()
99  }
00
```

After all these changes, this modified game functions with the standard rules of a 3x3.

Q1b - in order of thought process, candyCrush.playground

For a candy crush game, each square can have a different type of candy, so the GameSquare struct has to be modified. Instead of a player variable, there will be a candy variable that calls the struct Candy. Since each square should be filled in the candy crush game, there will not be the '?' operator since it cannot be nil.

```
struct GameSquare{
    var id:Int //index of the tile in grid
    var candy:Int // number to represent which candy in the square

    var image:Image{                                   Cannot find type
        // returns the image associated to the candy
    }                                                  Missing return in accessor expected

    static var reset:[GameSquare]{
        var squares = [GameSquare]()
            for index in 1...16 {
                squares.append(GameSquare(id: index))
            }
            return squares
    }
}
```

The Player and gamePiece in the GameModels file will be changed to the following lines. This is assuming that there are five types of candy in the game.

```
52  struct Candy{
53      let candyPiece:CandyPiece
54      var name:String
55      var moves:[Int] = []
56  }
57
58  enum CandyPiece: String{
59      case a, b, c, d, e
60      var image:Image{
61          Image(self.rawValue)
62      }
63  }
```

In the GameService file, there would also be a variable tracking every single candy and the squares that have that type of candy. The way it will track will be similar to how we have been tracking the players in the lines of code below.

```
11  class GameService: ObservableObject{
12      @Published var player1 = Player(gamePiece: .x, name: "Player 1")
13      @Published var player2 = Player(gamePiece: .o, name: "Player 2")
```

The reset function will be modified so that it sets up random pieces of candy on the screen. A major difference in a candy crush and tic-tac-toe game is how the game ends. Since there will not be any players and winners, there are two actions to consider. The first is that to make a move, the player will be swiping between two GameSquares instead of tapping on one. The second is when a candy "pops," assuming that the situation will only happen if there is a candy that is three-in-a-row horizontally and vertically. The calcWinningMoves() function and winningMoves variable in the GameModels file will be changed accordingly.

To account for the swiping motion, the game will have to track which two squares the player wants to swap. The makeMove function that is called when a gameSquare is pressed will be modified to something similar to the following. This function will only be called if the two squares have different types of candy.

Parameters: sq1 and sq2 represent the index of the two squares swapped. This void function will update the candy instances tracked.

```
func swap(_ sq1: Int, _sq2:Int) {
        candy1 = the type of candy in square 1
        candy2 = type of candy in square 2

        Remove sq1 from the moves array of candy1 and add sq2
        Remove sq2 from the moves array of candy2 and add sq1

        Change candy accordingly for the two squares, which are GameSquare instances

        arr1 = checkPop(candy1)
        arr2 = checkPop(candy2)

        if arr1 != empty array{
                pop(arr1, candy1)
        }

        // do the same with candy2
}
```

The following functions will be used to check whether a candy can "pop," and having everything shift down.

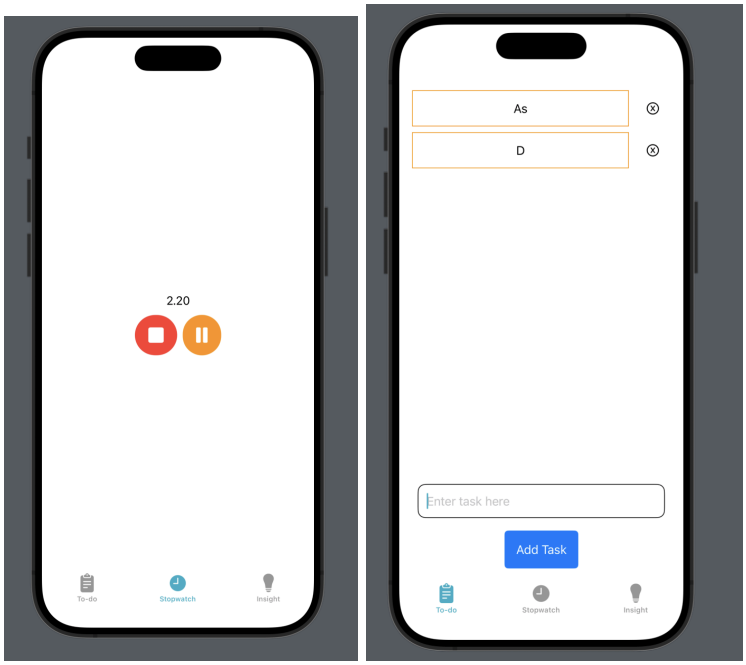
```
This variable will be added to the Candy struct
var checkPop:Int[] {
        // Similar to how isWinner works

        // if there are three in a row, return an array with those three indices
        For moves in Moves.winningMoves {
                If moves.allSatisfy(self.moves.contains) {
```

```
                    Return moves
                }
        }

        // if none in a row, return empty array
}

This function updates the moves array of the candy and calls for change in the visual grid
func pop(_ arr: [Int], _ candy:Candy) {
        Remove all numbers in arr from candy.moves

        If arr is horizontal:
                For num in arr:
                        shiftDown(getColNum(num), num, 1)
        Else: // arr is vertical:
                // call shiftDown with the bottom square in the combination
                // which would be arr[2] due to how we order the squares
                shiftDown(getColNum(arr[2]), arr[2], 3)
}

This function returns the column number of the square
func getColNum(_ num: Int) -> Int {
        Returns the column number of the passed in index using modulo
}

This function will update the visual grid
func shiftDown(_colNum: Int, _ curr: Int,  _shift: Int) {
        for square in column colNum of gameBoard, starting at curr {
                If “shift” squares above square exists in grid:
                        Change candy type to the candy in the “shift” squares above
                Else: // if out of bounds
                        Replace with random candy
        }

        Check whether each candy can “pop” and repeat process until nothing pops
}
```

Q2

I added three views and buttons at the bottom that would switch to each view. The buttons are added onto each view so that you can switch between each view.

For the stopwatch view, there is a running stop watch in which you can pause or stop. The to-do view has an add task button at the bottom and all the added tasks will show at the top of the screen.



The app would use the database in storing when the stopwatches get pressed to start or pause, along with the task associated with the stopwatch, which will be prompted with a drop down list from the todo list. The database information will then be used to generate the insights page, which will be a breakdown of how the user spent their day. The database data will also be analyzed to give users an average number for how much time they spend on different categories of tasks.