<u>How to Run:</u>
The Swift project should run on any ios device. However, I only tested it on my iphone 6s and the iphone 7 emulator in XCode. When the program is run, it will automatically produce a 3x3 grid and display all the unique solution chains in a scrollable TableView below the grid. There is a generate button at the bottom of the screen that will produce new random values for the grid and the new solutions will immediately be displayed in the TableView.

The solutions displayed in the scrollable TableView are organized depending on their starting cell. Cells are numbered from left to right starting in the top left corner. For a 3x3 Grid this would mean the cells are numbered as:
1 2 3
4 5 6
7 8 9

The computation time is usually instantaneous, but in can take 10 seconds or more in rare cases. The computational progress can always be monitored in the Xcode debugging console. I have a print statement in the code that will allow you to see how many more “validations” need to be made to produce the solution. This monitoring is especially useful when running the program and setting the WIDTH constant greater than 3.

To run the program and change the width to be greater or smaller than the default value of 3, simply change the value of the global constant WIDTH at the top of the ViewController.swift file.

Once the program is done running a list of tuples will be displayed in the debugging console that are associated with the cells of the valid chains. This is particularly useful to check when there are solutions that use different cells, but use the same sequence of numbers (1,1,3 and 1,1,3 could be unique). For a 3x3 grid the tuples would represent cells like so:
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)

Finally, if you want to test a specific case, you can do this in the file ThreeByThreeView.swift. The comment block before the function randomizeNxN explains how to do this.

UPDATE: I BELIEVE THIS FIRST PROBLEM IS FIXED, BUT I WILL KEEP THE SOLUTION HERE IN CASE THE PROBLEM IS ENCOUNTERED. I did noticed one unusual issue when downloading this project from the git repository and then running them. My Swift solution to problem 3 seems to fail because a .xml extension is added to two files and it shouldn't be there. They consequently "can't be found."

These files are found in ThreeByThreeChains/ThreeByThreeChains/Base.Iproj
The two files are LauchScreen.storyboard and Main.storyBoard. If you get an error saying the storyboards can't be found, it is possible the .xml extensions have been added to these files. The extensions simple need to be deleted to get rid of them. I'm not sure why the extensions are being added.

I also found that when the project is first opened it says the scheme is missing. But this can be easily fixed by clicking No Scheme->Manage Schemes -> AutoCreate Schemes Now.

How it works:

The display is mostly created dynamically.

Random numbers are set to display in the grid of labels in the top half of the screen. These random values are stored into an array and sent to an instance of the ThreeByThreeChainsCalculator class to calculate all the valid chain solutions.

These valid chains can then be accessed by ViewController. The ViewController sets up the scrollable tableView so the user can view these solutions organized in a table by their starting cell.

When the generate button is pushed a new set of random values are displayed and the new data is once again sent to the ThreeByThreeChainsCalculator object.

Algorithm:

Finding all valid chains while ignoring duplicates was quite straightforward. I simply checked every adjacent direction from each cell until the chain added up to the area the grid, was at least the grid width - 1 long, and there were no more valid adjacent cells. Then I went backwards one cell and repeated the same test.

The duplicates that were not ignored fall into the specific case of having a final cell that is adjacent to the final cell of the duplicate, both the final cells are also adjacent to the second to last cell in the chain, and they have the same starting cell.

All the valid chains that pass this duplicate test are placed into an array and thoroughly checked for duplicates.

Checking for Duplicates

I assigned a value to every chain that depended on what cells they used and the starting cell they had. This value seemed to almost always be unique. This allowed me to check the values two chains have before checking if they are duplicates. If the values were not the same, it was impossible for the two chains to be duplicates. So, I could immediately move on from them.

I also checked that any chain did not end in a cell that was before it. This was a valid check because of the way I constructed the chains. I constructed every chain starting from cell 1, then every chain from cell 2, then every chain from cell 3 etc. Thus, if a chain ended in a cell that came before it then it was a duplicate of a chain already produced that started at the cell the duplicate chain ends in.

A trait that was unique to how I constructed each chains value was that if:
1. The chains had the same value
2. The chains started at the same cell
3. The chains are the same length

Then the chains are duplicates.

Finally, if a chain passed these special checks, but they happened to still have the same chain "value" then I did a brute force check of the two chains to determine if they are duplicates. I had an array of tuples for every chain. If the tuples were just re-ordered in the second chain, then it was a duplicate.

I removed all the invalid chains from the array of chains. I then stored how many chains start at each cell in an array, and I put the list of valid chains into a public array. These were to be used by the ViewController to display the solutions.

Making a Version with a Larger Grid/ Scaling problems:

The program is not specialized for a 3x3 grid. As mentioned before if the global constant WIDTH is changed at the top of the ViewController.swift file the program will run for that WIDTH.

When WIDTH > 3 the computation time quickly increases, but solutions to WIDTH = 4 usually have computation times that aren't too long (seconds or a few minutes). WIDTH = 5 tends to run significantly slower and usually has impractical computation times.

I have seen the number of comparisons get up to over 13,000,000,000 when WIDTH is set to 5 compared to the typical 4,000-70,000 for a three by three grid. 13,000,000,000 is significantly larger for my algorithm since my algorithm does not even consistently take less than a few seconds to find the solution at smaller sized grids.

It is possible these computation times can be greatly decreased by altering the data structures or by optimizing the algorithm. The significantly larger number of comparisons might be completely due to a poor algorithm choice. I focused primarily on creating a solution that works. If my focus was to optimize the efficiency of the solution it would be important for me to analyze every algorithm and how data structures are being used in the program with respect to memory.