My project was designed to solve three different problems related to the game 2048. The first was that I had a problem with the current rule set of 2048 only allowing one undo. The second was how to handle a data set that was constantly needing to be sorted, such that a user could give their high score and be given a rank. The third was how to find the best solutions to the current game state. My motivation came from a bit of competitiveness with my dad as we liked to try to one up each other's scores on the game and send screen shots to each other. Solving the second problem would allow me to not only compare my score against his but others as well. When we played, there was an issue we both had with the game, and it was that they only allowed you to undo once. Creating this game myself would allow me to give the exact rules I wanted such as being able to undo as many as we liked. Solving the third problem, which is finding the best moves to use, would give me quite a competitive advantage against him though I would not include that functionality in his version. In all sincerity, the third problem was more for my own pedagogical reasons as I wanted an opportunity to play with and create very basic heuristics.

Solving the first problem of creating 2048 was simple. A 2D array to handle the matrix of the game board and a stack to take in the game board or to pop a game board during an undo command. The logic of 2048 was a doozy though through trail and error of doing data swapping or trying to combine all like numbers at once I eventually landed on a working solution. First filling moving data based on zeros, combining same cell numbers, then moving the data into the newly created zero positions.

The second problem of handling our leader boards was done with an open hash table and binary search trees. When they want to see their own rank and score, they simply need to search with their username and city which is then hashed for the open hash table. If they want to see a ranking of a city, a buffer binary search tree is filled with the given City by looping through the entire open hash table and inserting any record with the given City. When they were done with the game and wanted to add their score to the leaderboards their data was stored into the hash table. Then when saving their score for the leaderboards file it was then the job of the binary search tree to collect from the hash table to then store the sorted data into the file. This solution for the leadboards was messy. During the creating of the data structures, I was thinking about the easiest solutions of implementation rather than taking the time to think about the whole problem and analyzing the strengths and weaknesses of other data structures interacting with each other. I chose a hash table to be the main holding of our data with two assumptions. One that the user would always be searching their name in the leaderboards or inserting their score in therefore I wanted an O(1) for them. The second was that it would work well with binary search tree considering the binary search tree works best with random order of data being given to it. The first mistake was using an open hash table as it was more complicated than a closed hash table in collecting all the data from it in a function. The next mistake was when implementing it I hadn't taken the time to analyze the potential time complexity aside from the user's interaction with the leader board leading to an understanding that there were potentially better solutions for handling the storing and sorting based on efficiency. From trying to solve these issues and encountering these mistakes Ive learned that before I start to work on a large project, I should take a step back and analyze what I plan on using, how they will interact with each other, and what problems might occur like overall time complexity.

The third problem was handling the heuristics of 2048. I attempted to solve this problem by creating a graph of all possible solutions leading to an exponentially large time complexity. Thinking myself a genius, I ignored this, and simply attempted to have the recursion solve the board until a win state was reached. A quick google search states a perfect game of 2048 will be at minimum 520 moves. While it might have worked eventually, I would probably be several eons late in turning in this project. Right now there is a hard stop to the quadtree height, allowing only the best solution to be chosen and for the rest to be pruned before the user can use functionality of it again, bringing down time to find solution a few centuries. A score error function decides on which game board is the best state to be chosen. Through trial and error, I was able to find some variable weights that made this work for the most part. It was a lot of fun working on trying to find an effective solution but now I fully understand why it’s better to use machine learning to create those values.