This was another tricky program to try and figure out, from the fitness function, to the selection and down to how to use these in the running of the program itself. I decided to calculate my fitness based on the number of attacking pairs, then subtracting that from the max fitness of which I set to 28. All boards would then have a fitness of 28 or less with 28 being a solution and anything less would indicate there would be pairs of attacking queens. During the selection process, I would normalize the fitness of the population, and then pick the two parents with the best scores overall. These parents would then be used to create a crossover point to create the children, with a 10% chance of a mutation being added to the children.

I found that I had some trouble with the randomization of the population, more than any other portion. When using random.sample, this guaranteed I would not have two queens with the same board states, however, doing so would allow the program to find a solution too quickly in some cases, where generation 0 would find a solution due to the chance that a perfect board was in the population. I tried switching the randomization to be more truly random by using random.randint. In doing so, I found that the board was so random it would lead to poor test results and that I would never find a solution even after 5000 generations and using elitism to maintain the best board states. I decided to go with random.sample for my final solution as that at least guaranteed the algorithm finding a solution.

The final program is in python and runs with a population of 100 for 5000 generations until a solution is found or not found. This program will print out the best board for every 100 generations and will make a quick summary of the solutions found and the success rate as it runs the algorithm 5 times. The graph below shows my best run that I found that really showcases the genetic algorithm as a whole.