RSA problem description:

Our problem we wanted to solve was using a genetic algorithm to find the decryption key in the RSA encryption system. RSA works by generating two large primes, p and q, and multiplying them together to get a public key n. Then, we use p-1 and q-1 to get the value for phi(n). This value is used to generate the public key e. Then to find the private key d, the formula e*d = 1mod(phi(n)). [1] With this in mind, we decided to find the values of p and q using genetic growth in order to calculate the necessary decryption keys.

Initial thoughts/strategy:

To start the project, we both had our existing implementations of RSA using BigInteger in Java. We translated our encrypt and decrypt functions into Python, as well as changed our functions relative to producing the keys into normal integers inside of Java. In translation, we assumed integers would be held in 8 bits, but in bug fixing decryption, we found that our integers were being held in 7 bit sections, and the excess 0 was causing issues in decryption. Using our adjusted Java functions, we generated random keys to develop and test our genetic algorithm in Python. Our initial strategy was to generate two random primes (p and q), and then test those against our public key e. After further analysis of RSA, we decided to go the route of generating two random integers, and testing against the public key n. We decided not to generate random primes because we didn't want the computation time of looping until we hit prime numbers for every genome, so we decided to implement a "primePunishment" in our fitness function in order to balance fitness being close to n as well as making sure factors are prime. We decided that each genome would have the structure of a tuple made up of the two integers, represented as strings. We also theorized that we needed to add leading 0's to the integers, so that they all had the same length for easy crossover. The length we decided on was p/2 because it is the greatest length a prime factor of n could have since p/2 * 2 is the most extreme combination of primes possible.

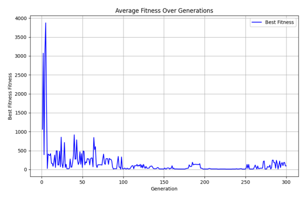
Initial Problems/Solution:

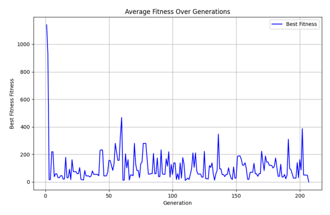
Our first error that we were making was sorting fitness the wrong way, trying to find the highest fitness instead of fitnesses approaching 0. We reversed how we sorted our list of fitnesses in the sortFitness function and we started seeing the decrease that we were looking for. Also, our prime punishment was 50 at first, and we found that our best fitness values were low numbers, i.e < 35, but the best fitness values remained unchanging for the rest of the generations. We concluded that the crossover wasn't changing the numbers anymore and all of the best genes were becoming the same two numbers since there weren't any different digits being added to the gene pool. We solved this issue by increasing the mutation rate by a factor of 10, in order to create more variability within the genomes. (1000 pop, 0.8 cross, 0.01 > 0.1 mut)

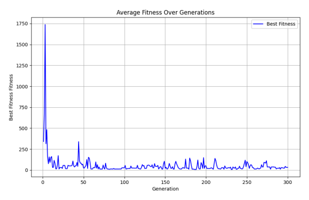
Testing Parameters:

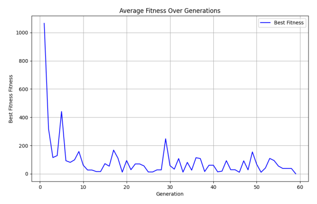
We tested population size, crossover rate, mutation rate, prime punishment, and the size of the primes themselves. Throughout our testing we made a total of 29 text files, with 10 runs per file, condensed down into a graph and displayed averages for each of the 10 runs per file. For population size we tested populations 100 through 1000 in size 100 chunks, using 10 runs

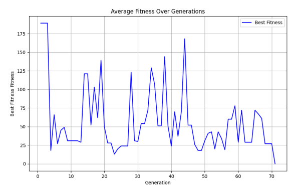
with an unchanging crossover, mutation rate, and n. We recorded the average generation in which our values were found and graphed the last run of each population size to display growth:
Average Generation Found for PopSize100: 272.6
Average Generation Found for PopSize200: 167.3
Average Generation Found for PopSize300: 140.4
Average Generation Found for PopSize400: 107.4
Average Generation Found for PopSize500: 135.6
Average Generation Found for PopSize600: 77.5
Average Generation Found for PopSize700: 51.4
Average Generation Found for PopSize800: 87.5
Average Generation Found for PopSize900: 60.7
Average Generation Found for PopSize1000: 43.6

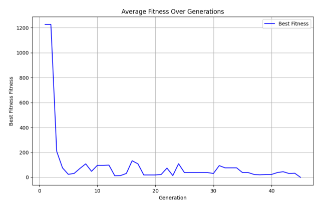


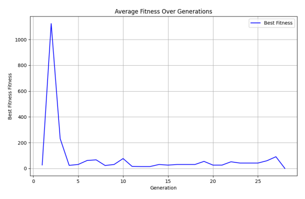


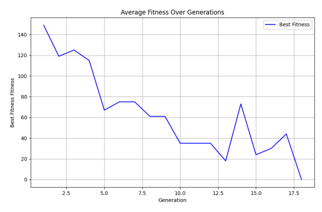


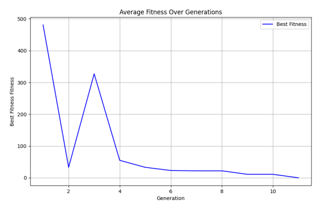


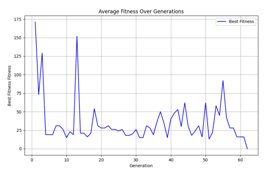








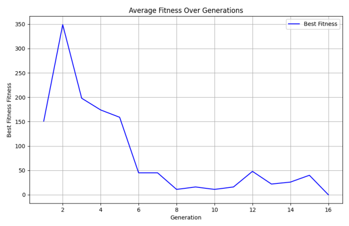


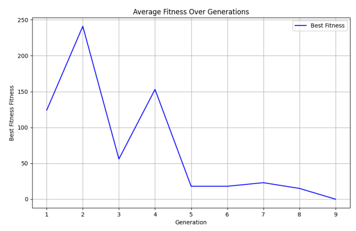


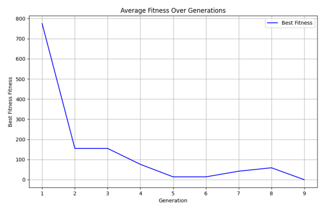
From our data, we found that low population sizes didn't leave enough room to narrow down p and q, finding low values, but not getting all the way to fitness 0. The best performing population size was 1000, but we decided the time it took for that wasn't worth the efficiency in comparison to the still effective sizes like 600 and 700.

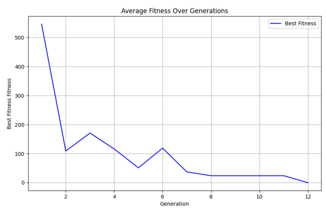
For Crossover, we decided to test from 0.5 through 1.0 with 0.1 increments. In this experiment we used consistent parameters of 700 population size, 0.1 mutation rate, and our same n. This time we printed the graphs from the best runs, which we only did for this run, since we didn't like how it didn't represent the average or "random run", but showed the "luckiest" one.

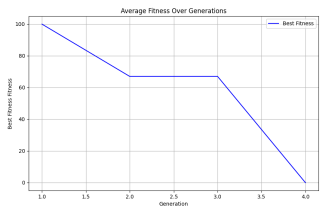
Average Generation Found for Crossover0.5: 85.8
Average Generation Found for Crossover0.6: 55.5
Average Generation Found for Crossover0.7: 63.4
Average Generation Found for Crossover0.8: 51.7
Average Generation Found for Crossover0.9: 65.8
Average Generation Found for Crossover1.0: 55.1

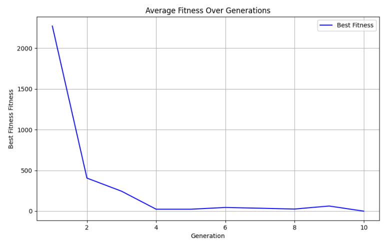








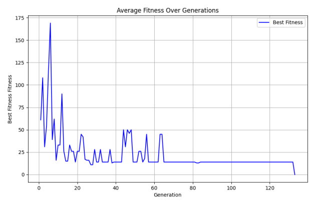


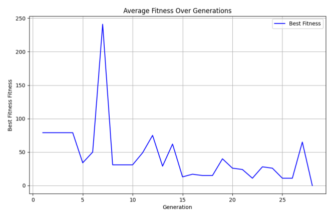


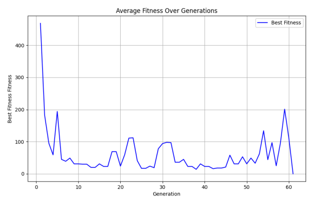
With this data, we concluded that differences in crossover values weren't the most important factor in the algorithm, as long as more than half the time it was happening. We found much more convincing changes in population size and mutation rate.

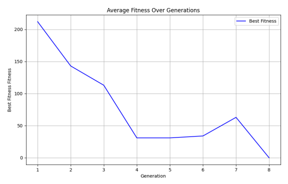
For mutation rate, we tested from 0.05 through 0.2 with increments of 0.05. We kept the parameters population size 700, crossover rate 0.8, and the same n for this experiment. We again graphed the results from the last run of each.

Average Generation Found for Mutation0.05: 93.9
Average Generation Found for Mutation0.1: 49.3
Average Generation Found for Mutation0.15: 110.5
Average Generation Found for Mutation0.2: 246.1





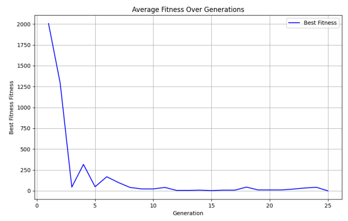


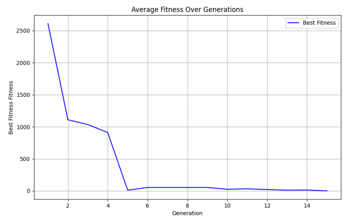


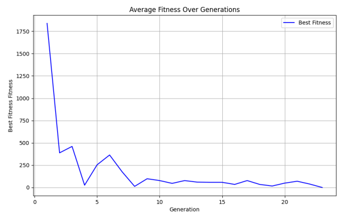
In mutation rate, we found far more convincing data that this parameter was affecting our growth in a major way. If we didn't crossover enough, i.e. 0.05, then our p and q were too static, and if there was too much mutation, our best genome statistics would spike all over the place and not be able to show any consistent growth towards 0.

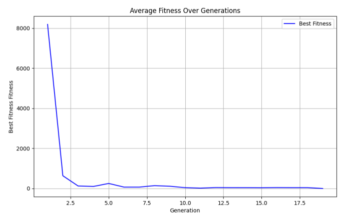
The next experiment we ran was for our prime punishment value. We decided to generalize the punishment to around .1% of p through manual testing, so for actual testing, we chose a range between 0% and .25%, with increments of 0.5% rounded down using the floor function. For the default parameters, we used population size 700, crossover rate 0.8, mutation rate 0.1, and the same n value.

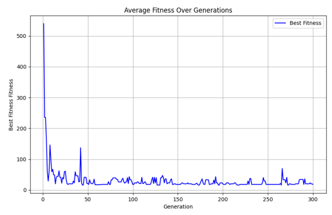
Average Generation Found for primePunishment0: 126.6
Average Generation Found for primePunishment3: 112.6
Average Generation Found for primePunishment7: 109.9
Average Generation Found for primePunishment11: 93.4
Average Generation Found for primePunishment15: 145.6
Average Generation Found for primePunishment18: 164.1

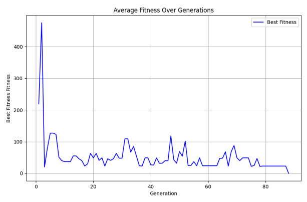












For this data, we also found a very clear range of effectiveness for our prime punishment attribute for fitness which was anywhere between 0.05% and 0.2% with .15% being the most effective.

For our last experiment, we tried using larger keys to test timing and effectiveness for our algorithm. Our default n we had been testing on was a combination of p and q both below 200. We then tested a random key for values < 500 and values < 1000. Our algorithm solved values < 500 sometimes, and it was not able to solve < 1000 within 300 generations. Below are the statistics for all three of those runs.

Average Generation Found for p,q < 200: 64.3
Average Time per Run: 41.6
Total Time for 10 Runs: 416 - 6 minutes 56 seconds

Average Generation Found for p,q < 500: 240.4
Average Time per Run: 158.2
Total Time for 10 Runs: 1582

Average Generation Found for p,q < 1000: 300.0
Average Time per Run: 205.4
Total Time for 10 Runs: 2054

As we can see from this timing, our algorithm becomes obsolete very quickly when the integers get larger. The largest integer we solved in 300 generations was 17 bits, which is quite far from RSA standards.

After this, we tried creating the initial population with genomes following basic rules of prime factorization such as only using odd numbers, and having p be less than sqrt(n) and q be larger than sqrt(n).[3] This addition actually hurt our algorithm:

Average Generation Found for p,q < 200: 254.2
Average Time per Run: 131.2
Total Time for 10 Runs: 1312

We theorized that this is because the initial population was missing too many needed digits for our algorithm to have enough room to find the p and q values.

Conclusions:

Overall, we would say that our algorithm was semi-successful since it does consistently crack very, very low levels of RSA. We were also able to use these values to decrypt a message. In our case, the message had to be one letter since the n value was so small, and the value of the message in bits cannot exceed the value of n. Where we were less successful was scaling our algorithm to larger values of n, and our efficiency compared to classic prime factorization strategies was significantly worse. Our first value of n was 13 bits, so theoretically, if the genetic algorithm was equally effective for the 2048-bit integers used for RSA, then our algorithm would take a little over 18 hours to run, however, it is not directly scaleable nor effective for that large of values. What we learned from developing and experimenting with this algorithm was that RSA is the current encryption practice for a good reason. We do not think a genetic algorithm is the answer to RSA because of computation time, and with examples like Mobin and Kamrujjaman using a genetic algorithm with a more complicated fitness function that uses a “multi-threaded bound changing chaotic firefly technique, for the integer factorization problem… [and] a fitness function to lure fireflies closer to each other according to their light levels”[2]. Even with this technique they only successfully factored a prime of 14 digits. The more time efficient strategies involve quantum computing such as Shor’s algorithm which, “showed how a hypothetical computer that exploited the quirks of quantum physics could break large numbers into their prime factors far faster than any ordinary classical machine”[4]. We believe that a more complicated fitness function and a more powerful computing base could bring a genetic algorithm closer to solving RSA decryption, but it is impossible with our current computing power and other strategies show more promise in this area.

Scaling time for real RSA values (theorizing same effectiveness and computation size)
2048 bits (the average length for real RSA values)
2048/13 = 157.5385
157.5385*416 = 65536 seconds
65536/60 = 1092.2667 minutes
1092.2667/60 = 18.2044

Works Cited

[1] tone, liu faan. “Protocol for RSA: Mathematical Cryptography - COMP 3705/4705.” Accessed November 18, 2024. https://canvas.du.edu/courses/175564/pages/protocol-for-rsa?module_item_id=4067752.

[2] Al Mobin, Mahadee, and Kamrujjaman Md. “Cryptanalysis of RSA Cryptosystem: Prime Factorization Using Genetic Algorithm.” ar5iv. Accessed November 18, 2024. https://ar5iv.labs.arxiv.org/html/2407.05944.

[3] Burns, Carol. “The Prime Factorization Theorem.” Accessed November 18, 2024. https://www.onemathematicalcat.org/algebra_book/online_problems/primeFactorization.htm.

[4] Brubaker, Ben. “Thirty Years Later, a Speed Boost for Quantum Factoring.” Quanta Magazine, October 17, 2023. https://www.quantamagazine.org/thirty-years-later-a-speed-boost-for-quantum-factoring-20231017/.