Alex Urbanski and Mengchen Gong

HW #8

Approaches Tried

Initially, we started with a brute force approach to see how fast we could solve for some of the factors of the numbers in the composite list. We did this will a trial division approach and it didn't take very long to solve most of the first values. But, you can see from the list that as the numbers get four digits larger, you would have to do 1,000 more comparisons, so the code would take about 1,000 times as long to run. This clearly doesn't work for most of the values in the list because the end values are a lot more than four digits larger than the initial values, so as you go through values in the list, the time complexity blows up.

Another idea to make this process faster was to only divide by the prime numbers for the trial division. The problem with this was array size. We wrote code to output the declaration of an array with only prime numbers up to a certain point. This way we could just declare an array of primes at the beginning of the trial division code and the array initialization would be a constant time operation. The problem with this idea is the size of an array of prime numbers up to a very large value is so large that the computer that it is running on doesn't have enough space to hold the array.

At this point it was clear that any brute force method probably wouldn't work and we needed to turn to some sort of algorithm. We went with the Pollard Rho algorithm because it seemed easier to understand when compared to some of the other factoring algorithms like the quadratic sieve. This algorithm also gets the job done well. It can factor up to about the 4500^{th} number in the composite list in a reasonable amount of time. And I'm sure the performance would increase if the code wasn't written in python.

Explanation of Code

The code starts by selecting a random integer between 2 and the square root of the number passed in. In the code this value is named first. Then the code generates a random value to add. This value is called adder in our code. The code then runs these values through a randomizing function. The function used is first^2 + adder. There is also a second value named second that is set equal to first initially but is run through the randomizing function above twice. The code then runs the Euclidean algorithm function that we wrote to see what the greatest common denominator of (second – first) and the value passed in, modder, is. You use (second – first) because of the birthday trick/paradox. When you are trying to get a random value in a range, if you take the difference of two random values it greatly increases the odds of getting the value that you are looking for. Consider a case where you have two random values i and j, if you are trying to pick one random number and have it equal 33, the odds that you randomly pick a value i that is 33 are slim, but if pick two value and take the difference, the odds of j-i equaling 33 are greatly improved because there are a lot of possibilities that give you j-i equals 33. So after (first-second) and modder get passed to the Euclidean algorithm and the response is returned, we run a function called checkValue. This checks for certain conditions. If the result of the Euclidean algorithm is 1, this indicates that the values passed in are relatively prime and the code can just be cycled through again. You just take the old first and second values and run them through the randomizing function again to get new values and then run the difference through the Euclidean

algorithm again. Another case is where the greatest common factor is the value of modder. In this case, you must start the whole process over again and pick new random starting values. The gcd being equal to modder indicates that this group of starting values does not have a solution. This case means that first-second is equal to the modder so their greatest common factor is modder. The last option is that you end up with some value that is not modder or 1. In this case, (first-second) is a prime factor of modder. The only way that a value other than modder and 1 could be returned is if it is a factor of modder. We then return this value and the other factor as an array.

Results

The code runs well. As mentioned before, the code can run up to about the 4500^{th} number in the composite list that was given. When it gets to numbers in this range it usually takes about 30 seconds to run. After this point, the time that the code takes to run really starts to increase. If the code was written in a lower level language it might be able to factor numbers past this point more quickly. The whole process of writing the code shows that the brute force approach doesn't always work and you will often need to come up with or use algorithms because they are usually a lot more efficient.