Algorithms for Data Science
Lab 3
RSA Cracking

In this lab, you will gain a better understanding for how quickly running times increase for exponential, O(2^n), algorithms. We will explore this by looking at cracking RSA encryption.

In 1977, Ronald Rivest, Adi Shamir, and Leonard Adleman came up with the now famous RSA encryption algorithm (based on the work of mathematical greats like Euler and Fermat). This algorithm is one of the cornerstones of public key encryption (PKE). In fact, this algorithm is so important that in 2002, the authors were awarded the highest honor one can receive in the field of computer science: the Turing Award.

The underlying principle of RSA is that a one-way computation is performed—that is, something that is easy to do in one direction but impossible (or very, very difficult) to reverse without the proper key. The computation used is the multiplication of prime numbers. For example, take the two prime numbers P=13 and Q=23. PQ=13*23=299. This is a simple calculation to perform. A computer can do it with ease, and a human could even do it with nothing more than paper and pencil. However, the reverse operation of finding the P and Q given only PQ is very difficult, even for a computer. This is called finding a prime factorization of PQ. Try it: Given a PQ of 203, what were the P and Q? Give up? How about P=7 and Q=29? How did I know this? Because I picked P and Q in the first place and cheated! OK, so a human has a hard time finding P and Q from PQ. But a computer can do it easily, right? Well, a computer *can* do it, but it takes a lot of time. And the time it takes is proportional to the length (in bits) of PQ. So, a computer could crack the example given above because P and Q are small numbers. But if a PQ were picked so that it contained 1024 bits or more (e.g., a 1024-bit number can have a range of [0…2^1024], or up to 309 decimal digits!), it would take a computer longer than the estimated age of the universe to crack the code. 1024 is actually a typical key length value for many encryption systems.

In addition to finding primes P and Q, as well as PQ, the RSA algorithm also has two other values it needs to compute: E and D. Our public key will be the pair (E, PQ), and the private key will be the pair (D, PQ). P and Q are used to generate all three of these numbers but are then thrown away after PQ, E, and D are found. E and PQ are public, and anyone can use them to encrypt a message to send to the person holding the matching private key. D is private and is used along with PQ to decrypt a message encrypted with E and PQ. So, you never want to give out D. And the only way the encryption code can be cracked is if someone can recreate the factors P and Q from their product, PQ. The details of E and D are not important for this lab.

For this lab, you are going to time how long it takes to crack RSA encryption keys. The first step will be to create the key in the first place. To do so, you need to create two functions. The first is

```
def isPrime(p):
```

This function should take a number, p, and return true if it is prime and false otherwise. You should do this by simply trying to divide p by all the numbers between 2 and p (or if you want to get fancy, the sqrt(p)). If any of them evenly divide it (p%i == 0), then it is clearly not prime. If you make it through all the numbers, then it is prime because the only numbers that evenly divide p are 1 and p itself.

The second function is

```
def nBitPrime(n):
```

This function should generate a random prime number that is up to *n* bits long. An *n*-bit number can range from 0…2^n. For example, if *n* is 8, then the number can be from 0…256. To do this, first generate a random float using random.random(). This generates a number between 0.0 and 1.0. Then multiply it by 2***n* to get a random number in the range we are interested in. If this number is >= 2 and prime, then return it. Otherwise repeat and generate another potential prime random number. Eventually it will generate and return one.

After you can generate large *n*-bit length prime numbers, generate two of them (P and Q), and multiply them together to form PQ. For example, to generate PQ from two 20-bit primes: `pq = nBitPrime(20) * nBitPrime(20)`

Up to this point, all the code you have written would have been done when creating the public/private key pairs. P and Q would then be thrown away, and PQ would be shipped out as a public key that everyone had access to. Thus, in order to “crack” RSA encryption, all one needs to do is to factor PQ into the P and Q that formed it. It turns out that PQ has exactly two factors, P and Q, because of how it was formed from primes. So, all you need to do to factor PQ is to try to divide it by every number between 2 and PQ and find the one that evenly divides it. That will be P, and Q would then be PQ/P (or P and Q could be reversed, but that doesn’t matter). I want you to write that function. Have it return both P and Q.

```
def factor(pq):
```

After the factor is working, you should now time how long it takes to perform just the factor step. The factor step would be the “cracking” step. Use the timing setup from the previous lab to time how long this function takes to run in milliseconds. I then want you to create and factor keys for increasing bit lengths. Start with something small like 15 bits and work your way up until it starts taking longer than around 5 minutes to factor. You should see that the times increase rapidly. In fact, this is an exponential algorithm, O(2^n).

Most people don’t fully understand just how quickly exponential curves increase. To demonstrate this, I want you to place your timing results into a program that can fit curves and provide predictions. Excel will do this as well many free and online curve fitters. You will have pairs like

```
15   0.2
16   1.1
17   1.4
18   10.6
19   30.2
20   46.6
…
```

Put them into the curve fitter, and fit an exponential curve to the data. For Excel, creating a scatterchart is often the best choice for this type of data. When it plots it will show only the collected data points so you will want to add a trend line that connects the points together (right-click the grid points to add a trend line). Note that Excel is picky about its exponential fitting algorithm. If your data has zeros and perhaps other duplicate values, it won't offer you exponential as a choice. If this happens, then the problem is most likely your data at the start of your run. Simply make the chart without this starting data so you can fit an exponential curve to the remaining data. Once you have the trend line, you can estimate how long it would take to crack a key of any length (the forecast option of trend lines). I want you to give me an estimate for how long it would take to crack a 1024-bit key (a typical length for many encryption systems). I want this estimate in both milliseconds and in years. Please include this number as a comment in your code.

Keep in mind that what you are really plotting is the relative time it takes to do each crack—that is, as the bit length goes up, the cracking time goes up by some percentage. Because of this, you *must* run all your timing tests on the same computer system. They will be meaningless if you run half of them on your desktop and half on your laptop. It doesn't matter what machine they are run on, it just needs to be the same machine for all the tests.