Takehome Project 1

Due Thursday, October 15

In this assignment you will implement RSA from the ground up, including generating your own (very large) primes. You will also prove a few things about the algorithms you write, and take a small tour of the Euler φ function and how it relates to RSA. So it will look a lot like homework assignments so far with a few key differences.

- Your work must be your own. For this assignment do not work in groups or share code.
- It is open book: you may use the textbook, your class notes, my class notes and lecture videos, past homework assignments and their solutions, as well as the Sage and Python documentation. Everything else is off limits. I can also be a resource so don't hesitate to reach out!
- Part of the assignment is involves me sending a message to you and you replying. Therefore you must share an RSA public key with me (on Discord or via email if you prefer), **By Tuesday!**

If you have questions, please reach out to me ASAP. Ok, let's get started. Have fun!

Implementation Part

1. First gather your belongings. You will need your fast powering algorithm and the extended euclidean algorithm, as well as your implementations of the ASCII encoding schemes: `textToInt` and `intToText`. It is important that these run correctly and efficiently because they will do much of the hard work of the functions we build, so if you need to make corrections from previous assignments please do, you are welcome to use posted homework solutions as a guide.

2. The problem at the center of RSA is finding the e'th root of the ciphertext c modulo N where $N = pq$ is a product of (distinct) primes. This is hard to do for large N if you don't know the factors p and q, but once you know p and q it becomes easy! Write a finction `findRoot(c,e,p,q)` which solves the equation $x^e \equiv c \mod N$ for x where $N = pq$. That is, it should satisfy the following

Input	Output
An integer c, and two primes $p \neq q$ An exponent e with $\gcd(e,(p-1)(q-1)) = 1$	The eth root of c $\mod N$: $\sqrt[e]{c} \mod N$

3. The security of RSA depends on finding two large primes p and q, and multiplying them together to get a large value N. As demonstrated in problem 2, knowing p and q makes the RSA problem easy, so it is important that you keep p and q secret. The best way to do this is to generate them yourself, so let's do it! We first remind ourselves of the following definition:

   Definition 1 (Miller-Rabin Witness)**.** *Let n be a number, and factor $n-1 = 2^k m$ for m odd. A number $a = 2, ..., n-1$ is called a Miller-Rabin Witness for the compositeness of n if both:*

   - $a^m \not\equiv 1 \mod n$
   - *None of* $a^{2^i m} \equiv -1 \mod n$ *for any* $i = 0, \cdots, k-1$

We proved in class that if such an a exists, n cannot be prime (hence a witnesses that n is composite).

(a) Define a function `millerRabin(a,n)` which takes as input integers $a, n \in \mathbb{Z}$ with $2 \leq a \leq n-1$ and returns `True` if a is a Miller-Rabin witness for n and `False` otherwise. If you get stuck, Table 3.2 in [HPS] outlines some pseudocode you can implement. (**Note**: It is very important that this runs quickly, so you shouldn't compute each $a^{2^i m}$ with fast powering as this will pick up an extra factor of log. Instead use fast powering to compute a^m, and find the rest with successive squaring.)

Finding one Miller-Rabin witness means that n is composite, but if a is not a Miller-Rabin witness that doesn't mean that n is prime. Nevertheless, Miller-Rabin witnesses are very plentiful for composite numbers! We exploit this to write a probabilistic test for primality.

(b) Write a function `probablyPrime(n)` which runs `millerRabin(a,n)` for 20 randomly chosen a between 2 and $n-1$. If any of these a are a witness, the n must be composite, so we return `False`. Otherwise there is a very high probablity that n is prime, so we return `True`. Just how high is the probability? We will address this in the written part of this assignment.

Now we have a way to efficiently test if a given number is (very likely) prime. So how do we find primes in a given range? It turns out that primes are plentiful enough that if we just guess random numbers and see if they are prime we will probably stumble upon one before too long:

(c) Write a function `findPrime(lowerBound,upperBound)` which returns a prime between the two bounds by repeatedly picking a random number n between the bounds and running `probablyPrime(n)`, until it finds an n that is (probably) prime, and returns it.

(d) This works remarkably well! Use it to find a prime number in the following ranges:

i. between 10 and 100.
ii. between 1000 and 10,000.
iii. With 100 digits.
iv. With 500 digits.

Amazing!! Check your work as follows: Sage can (proveably) check if a number n is prime by returning `True` or `False` on `n in Primes()`. Try this on the first 3 primes you just generated (it probably will be too slow for the fourth).

4. Now we can implement RSA.

(a) Write a function `generateRSAKey(b)` which generates an RSA public and private key from primes `b` bits long in the following four steps:

(1) Generate 2 primes p and q of length `b` bits using the functions you wrote in Problem 3 of this assignment.

(2) Choose an encryption exponent $e \in (\mathbb{Z}/(p-1)(q-1)\mathbb{Z})^*$ with $e \neq 1$. (Choosing random numbers and seeing if they are prime to $(p-1)(q-1)$ should actually be a rather quick way to do this).

(3) Compute the decryption exponent d associated to e. (This is essentially the core of the computation in Problem 2, so rather than have to run `findRoot` every time we want to decrypt a message, we can compute this decryption exponent once and for all).

(4) Return a pair of keys `[PublicKey,PrivateKey]` where `PublicKey = [N,e]` for $N = pq$ will be published and `PrivateKey = [N,d]` you will keep to yourself and use to decrypt messages.

(b) Write functions `RSAEncrypt(message,PublicKey)` and `RSADecrypt(cipher,PrivateKey)` encrypt a message or decrypt a cipher. (Note, expect `PublicKey` and `PrivateKey` to be of the form generated in part(a)).

We did it!

5. Let's test this out.

   (a) Generate an RSA key from 16 bit primes. Then encrypt $m = 314159$ and decrypt the given ciphertext. Did it return m?

   (b) Let's use RSA to communicate over a public channel. Generate an RSA key from 512 bit primes. Post your public key on the Discord channel `RSA_Keys` **by TUESDAY**. **Save your private key!** Expect a message from me.

   (c) I have generated the following RSA Key from 512 bit primes:
   $N = 105221311114140839201427133950507696174424336576952551465373118964343111443615210569737200855666356661550817443041847189726504019039974038772379507161582278783519296999687194872224184723574952162166440945091505292215584920700394034422263656616475374115086086511371138567040077974143875815670499682468534466439.$
   $e = 210168362870299867477237598007741018352812758548731801662455430585258873951939630776686025499016539605430325922834829730788909371983531365662652244520555765407005232827058670837160067093166648952976725071881739083691398929472902959389644431397849697545514705621277040862461418920441523752063518556102541155155.$
   You will need this information to reply to my message.

Written Part

6. (a) Prove that `millerRabin(a,n)` runs in $\mathcal{O}(\log n)$ basic operations $(+, -, \times, \div)$.

   (b) Assume that `ZZ.random_element` runs in $\mathcal{O}(1)$. Prove that `probablyPrime(n)` runs in $\mathcal{O}(\log n)$ basic operations.

   It is rather amazing that our probabilistic primality test runs so quickly. It is slightly more subtle to analyze the time complexity of `findPrime` (and therefore anything that calls it), since it relies on randomly guessing numbers before checking if they are prime. Part of the input to this is knowing the distribution of the primes which we haven't discussed yet. Nevertheless, we can discuss some of this.

   (c) Describe how `findPrime` could possibly go on forever.

   (d) Show that it is extremely unlikely that `findPrime` runs forever. (*Hint*: For the second part, let P be proportion of primes in your range. To not pick a nonprime then has probabilitiy $1 - P$. Relate the problem to $\lim_{n\to\infty}(1 - P)^n$).

7. ~~Recall that if n is composite, then 75% of the integers between 2 and $n - 1$ serve as Miller-Rabin witnesses to the compositness of n. Use this fact to compute the probability that n is prime if probablyPrime(n) returns true. Fully justify your answer.~~

8. We finish by proving the general case of Euler's theorem, and using this to show that the generalized RSA problem of solving $x^e \equiv c \mod N$ for x can be solved by factoring N. First let's review the following definition.

Definition 2 (Euler's φ-function)**.** *Let n be a positive integer. Then $\varphi(n)$ is the number of positive integers less than n which are coprime with n. That is:*

$$\varphi(n) = \#\{a = 1, \cdots, n-1 \text{ such that } \gcd(a, n) = 1\}.$$

(a) Show that $(\mathbb{Z}/n\mathbb{Z})^*$ is a finite commutative group under multiplication. What is its order?

(b) Prove Euler's Theorem: Let a be an integer coprime with n, then

$$a^{\varphi(n)} \equiv 1 \mod n.$$

(*Hint*: Recall that we proved Lagrange's theorem for commutative groups. Can this do all the work for you?).

(c) The Euler φ function has nice properties with respect to multiplication. Let's establish them:

i. Let m, n be positive coprime integers. Prove that $\varphi(mn) = \varphi(m)\varphi(n)$.

ii. Let p be a prime number, and j a positive integer. Compute $\varphi(p^j)$. Justify your answer.

iii. Using parts (c)i,ii prove the following formula for φ for a general integer $N > 1$:

$$\varphi(N) = N \cdot \left(\prod_{\substack{\text{primes } p \\ \text{with } p|N}} \left(1 - \frac{1}{p}\right) \right).$$

(d) Euler's theorem is related to several theorems we have already seen.

i. Deduce Fermat's little theorem from Euler's theorem.

ii. Let $N = pq$ be a product of distinct primes. Compare the formula given by Euler's theorem to the version of Euler's theorem for the product of two primes that we proved in class. Which one is stronger? Does one imply the other?

(e) Fermat's little theorem helped us compute roots mod p, and Euler's theorem for the product of two primes helped us compute roots mod pq. Similarly, the general version of Euler's theorem allows us to compute roots mod N for general N. To see this fix N, c, e positive integers such that $\gcd(e, \varphi(N)) = 1$ and such that $\gcd(c, N) = 1$.

i. Show that $x^e \equiv c \mod N$ has a unique solution in $\mathbb{Z}/N\mathbb{Z}$.

ii. Suppose you know the factorization of N. Describe an algorithm to compute the unique solution $x = \sqrt[e]{c} \mod N$ from part (e)i.