CS 5000: F24: Theory of Computability
Assignment 09

Vladimir Kulyukin
Department of Computer Science
Utah State University

November 16, 2024

1 Learning Objectives

1. Public-Key Cryptosystems (RSA)
2. Extended Euclid's Algorithm
3. Multiplicative Inverse Modulo n
4. Euler's Totient

Introduction

In this assignent, we'll implement a 2-prime factor version of the RSA system. I broke the implementation into several modular sub-problems and wrote a number of unit tests for each sub-problem in `rsa_uts.py` to assist you in your implementation. You can work on a sub-problem, unit test it, leave the assignment for something else, and then come back to work on the next sub-problem.

You may want to review the PDFs of the lectures on 11/11 and 11/13 and/or your class notes to become comfortable with Extended Euiclid, Multiplicative Inverse Modulo n, Euler's Totient, and RSA.

Problem 1 (5 points)

When you work on the functions and methods in `rsa_aux.py`, `rsa.py`, and `hack_rsa.py`, remember that each of these functions should be no more than 10 lines of Python code. If you find yourself doing something more complex than that, I recommend that you review the relevant slides and brainstorm the problem some more. You may want to comment out my unit tests in `rsa_uts.py` initially and then uncomment them one by one as you work on specific sub-problems.

Subproblem 1.1 ($\frac{1}{2}$ point): Extended Euclid's Algorithm

Implement the Extended Euclid Algorithm in the function $\texttt{xgcd}(a, b)$ in `rsa_aux.py` that uses Extended Euclid to compute d, x, and y such that $d = ax + by$ and $d = \texttt{gcd}(x, y)$.

I wrote `test_xgcd()` in `rsa_uts.py` for you to to test your implementation. This method generates random numbers a and b in $[1, 1\ 000\ 000]$ and tests $\texttt{xgcd}(a, b)$ `ntests` times. Change the defaults of `lwr`, `uppr`, and `ntests` as needed.

Subproblem 1.2 ($\frac{1}{2}$ point): Multiplicative Inverse Modulo n

Use your implementation of `xgcd()` to to mplement the function $\texttt{mult\_inv}(a, n)$ in `rsa_aux.py` that returns the multiplicative inverse of a modulo n (i.e., $a^{-1} \bmod n$). In other words, it solves $ax \equiv 1 \pmod{n}$ for x.

I wrote three unit tests (`test_mult_inv_01()`, `test_mult_inv_02()`, and `test_mult_inv03()`) in the `rsa_uts` class in `rsa_uts.py`. Use them to test your implementation of `mult_inv()`.

Subproblem 1.3 ($\frac{1}{2}$ point): Euler's Totient

Implement the function $\texttt{euler\_phi}(n)$ in `rsa_aux.py` that computes Euler's totient (i.e., $\phi(n)$). Use `test_euler_phi_01()` and `test_euler_phi_02()` in `rsa_uts.py` to test your implementation.

Subproblem 1.4 ($\frac{1}{2}$ point): Choosing e

Implement the static method `rsa.choose_e(eu_phi_n)` in `rsa.py`. This method takes the output of Euler's totient (i.e., `euler_phi(n)`) Specifically, `eu_phi_n` $= \phi(p \cdot q)$, where p and q are two primes such that $p \neq q$. One way to implement this function is to make sure that `eu_phi_n` is sufficiently large (e.g., at least 20), generate all numbers in `[lwr, eu_phi_n - 1]` that are relatively prime to `eu_phi_n`, and then choose one of these numbers randomly. The lower bound of the interval (i.e., `lwr`) should be some prime number at least 2 digits long (e.g., 11). Use `test_choose_e()` in `rsa_uts.py` to test your implementation of `rsa.choose_e()`.

Subproblem 1.5 ($\frac{1}{2}$ point): Key Generation

Implement the static method `rsa.generate_keys_from_pqe(eu_phi_n)` in `rsa.py` that returns the RSA's public and secret keys. Use `test_generate_keys_from_pqe()` in `rsa_uts.py` to test your implementation.

Subproblem 1.6 ($\frac{1}{2}$ point): Encryption and Decryption

Implement the static methods `rsa.encrypt`(m, pk) in `rsa.py` and `rsa.decrypt`(c, sk) that do the RSA encryption and decryption of integer messages and cryptotexts, respectively In `rsa.encrypt(m, pk)`, `m` is a message (i.e., a positive integer) and `pk` is the public key returned by `rsa.generate_keys_from_pqe`(). In `rsa.decrypt(c, sk)`, `c` is a cryptotext (i.e., a positive integer) and `sk` is the secret key returned by `rsa.generate_keys_from_pqe()`. Use `test_encrypt_decrypt_01()` and `test_encrypt_decrypt_02()` in `rsa_uts.py` to test `rsa.encrypt()` and `rsa.decrypt()`.

Subproblem 1.7 (1 point): Encryption and Decryption of Texts

We can now use `rsa.encrypt()` and `rsa.decrypt()` to encrypt and decrypt texts. To keep it simple, we'll encrypt and decrypt character by character. Towards that end, implement the static method `rsa.encrypt_text(text, pub_key)` in `rsa.py` that takes a string `text` and a public key `public_key` and outputs a list of cryptotexts (i.e., positive integers) where each cryptotext is obtained by calling `rsa.encrypt(ord(c), pub_key)` on every character `c` in `text`. The Python function `ord(c)` takes a character and outputs its code.

Implement the static method `rsa.decrypt_cryptotexts(cryptotexts, sec_key)` in `rsa.py` that takes a list of cryptotexts returned by `rsa.encrypt_text()` and a secret key `sec_key` and returns the original text by calling `chr(rsa.decrypt(ctxt, sec_key))` for every cryptotext `ctxt` in `cryptotexts`. The Python function `chr(char_code)` takes a character's code and returns the corresponding character.

Use `test_encrypt_decrypt_text_01()` and `test_encrypt_decrypt_text_02()` in `rsa_uts.py` to test your implementations of `rsa.encrypt_text()` and `rsa.decrypt_cryptotexts()`.

Here's my output from `rsa.test_encrypt_decrypt_text_01()`. You can think of them as Gödel numbers if you want.

```
Cryptotexts:

[333456, 367744, 349962, 377008, 370343, 79149, 256032, 345035, 110330, 73345, 167217,
 345035, 314317, 167217, 129658, 167217, 110330, 294580, 106091, 127737, 349962, 377008,
 304897, 266256, 266256, 167217, 167217, 167217, 167217, 167217, 167217, 167217, 167217,
 167217, 167217, 167217, 167217, 167217, 167217, 167217, 167217, 275374, 370343, 79149,
 256032, 129658, 73345, 106580, 377008, 129658, 314317, 266256]

Original Text:

Everything is a number.

                Pythagoras

Decrypted Text:

Everything is a number.

                Pythagoras
```

Here's my output from `rsa.test_encrypt_decrypt_text_02()`.

```
Cryptotexts:

[400075, 167217, 129658, 106091, 167217, 127737, 370343, 167217, 256032, 349962, 377008,
```

```
 345035, 79149, 129658, 73345, 349962, 167217, 129658, 167217, 142928, 349962, 350866,
 336889, 167217, 127737, 370343, 167217, 76225, 345035, 79149, 345035, 74920, 349962,
 110330, 314317, 256032, 345035, 47229, 167217, 129658, 167217, 63474, 350866, 345035,
 314317, 314317, 336889, 266256, 129658, 110330, 219177, 167217, 127737, 370343, 167217,
 106091, 129658, 346145, 349962, 294580, 47229, 167217, 129658, 167217, 256032, 294580,
 106091, 129658, 110330, 167217, 127737, 349962, 345035, 110330, 73345, 336889, 167217,
 129658, 110330, 219177, 167217, 106580, 110330, 293635, 370343, 167217, 129658, 167217,
 256032, 294580, 106091, 129658, 110330, 167217, 127737, 349962, 345035, 110330, 73345,
 336889, 266256, 350866, 345035, 79149, 256032, 106580, 294580, 79149, 167217, 129658,
 110330, 370343, 167217, 314317, 47229, 349962, 76225, 345035, 129658, 293635, 167217,
 129658, 79149, 79149, 129658, 76225, 256032, 106091, 349962, 110330, 79149, 167217,
 79149, 106580, 167217, 129658, 110330, 370343, 167217, 314317, 79149, 129658, 79149,
 349962, 167217, 106580, 377008, 167217, 110330, 129658, 79149, 345035, 106580, 110330,
 129658, 293635, 266256, 349962, 110330, 79149, 345035, 79149, 370343, 167217, 350866,
 256032, 129658, 79149, 314317, 106580, 349962, 367744, 349962, 377008, 304897, 266256,
 266256, 266256, 167217, 167217, 167217, 167217, 167217, 167217, 167217, 167217, 167217,
 167217, 167217, 167217, 167217, 167217, 167217, 167217, 128700, 293635, 127737, 349962,
 377008, 79149, 167217, 333456, 345035, 110330, 314317, 79149, 349962, 345035, 110330,
 266256]

Original Text:

I am by heritage a Jew, by citizenship a Swiss,
and by makeup a human being, and only a human being,
without any special attachment to any state or national
entity whatsoever.


                Albert Einstein

Decrypted Text:

I am by heritage a Jew, by citizenship a Swiss,
and by makeup a human being, and only a human being,
without any special attachment to any state or national
entity whatsoever.


                Albert Einstein
```

Subproblem 1.8 (1 point): Hacking RSA

Is RSA hackable? Two different branches of mathmematics will give you two different responses. From the point of view of number theory the answer is, yes, because the unique factorization theorem states that any natural number can be factored into a unique product of primes. Computability theory will add to this: not only is it breakable, it is primitive recursively breakable, because all operations in breaking RSA are primitive recursive functions. Probability theory, on the other hand, will state, well, that may be so, but, in practice, breaking RSA is difficult, because finding factorization is expensive and guessing the right prime factors is highly unlikely. The security of RSA rests, in large part, on the difficulty of factoring large integers. If the eavesdropper Eve can factor n in a public key, then she can obtain the secret key S from the public key P. How? Suppose Eve has managed (has enough computational power) to factor n into p and q and now has the cyphertext C of some message M.

Obtaining cyphertexts is much easier than computing prime factorizations, especially if cyphertexts are transferred wirelessly (e.g., Wi-Fi). Read up on *wardriving* (e.g., `en.wikipedia.org/wiki/Wardriving`).

Let's assume that Eve has C and $P = (e, n)$ (remember that P is publicly available!) and has managed to factor n into p and q. All Eve has to do is to compute d as the multiplicative inverse of e modulo $\phi(n)$. And, (drum roll!) Eve has the secret key $S = (d, n)$. The cryptosystem is now broken, because $C = P(M)$, for some message M, and $M = S(P(M))$. Since Eve now has both S and P, she can decrypt any captured cyphertext C.

Implement the static method `get_sec_key(message, cryptotext, pub_key)` in `hack_rsa.py`. This method takes a message (a positive integer), the message's cryptotext (another positive integer), and a public key and attempts to break the RSA encryption by obtaining the secret key as outlined above. Use `test_hack_rsa_01()`, `test_hack_rsa_02()`, and `test_hack_rsa_03()` to test your implementation.

The test `test_hack_rsa_01()` uses 2-digit primes for p and q so breaking it is easy. The test `test_hack_rsa_02()` uses

3-digit primes for p and q, which makes it slightly harder to break, but not that hard. The test `test_hack_rsa_03()` uses 4-digit primes for p and q. When you run it, you should notice that it takes significantly more time to break. Imagine the difficulty of breaking it if p and q contain 100 or 200 digits each.

In general, the more digits we add to p and q, the harder it becomes to break our encryption even if the evesdropper Eve knows the message, its cryptotext, and our public key. Just imagine what a gargantuan task it would be for her if both p and q contained 10,000 digits. My brain starts to hurt when I think of those BIG numbers and they actually exist.

Parting Thoughts

An inspiration I always draw from public-key cryptosystems is that mathematics works in beautiful and mysterious ways across times, languages, and cultures. Think about it! In the 4-th century BCE, Master Euclid proves that there are infinitely many primes and writes the proof down in his famous *Book of Elements*. It could've been his disciples that wrote the proof, but it's irrelevant, because that knowledge was passed on. What's relevant and fascinating is this: the proof lies dormant for centuries (centuries!) until, in the early 20-th centery, another Master, Kurt Gödel, uses Euclid's theorem to design an ingenious technique to map arbitrary formal statetements into natural numbers so that one can use various properties of those numbers to reason about the properties of the statements the numbers encode. This technique was later called Gödel numbering. Gödel then goes even further and uses Euclid's gift to show limitations of a specific type of formal reasoning.

In his *Book of Elements*, Euclid proves another theorem on how to compute the greatest common divisor of two numbers. In 100 AD, Sun-Tsu, a Chinese Master, proves a theorem on the correspondence between a system of equations modulo a set of pairwise relatively prime numbers and an equation modulo the product of those numbers. The theorem later comes to be known as the Chinese Remainder Theorem. Again, both theorems (Euclid's and Sun-Tsu's) hibernate for centuries until, in the second half of the 20-th century, Drs. Rivest, Shamir, and Adleman use them to design the RSA system and to prove its correctness.

What to Submit?

Save your implementations in `rsa_aux.py`, `rsa.py`, and `hack_rsa.py` and submit these three files in Canvas.

Enjoy RSA Hacking!