CS 5000: F21: Theory of Computability
Assignment 12

Vladimir Kulyukin
Department of Computer Science
Utah State University

December 1, 2021

1 Learning Objectives

1. Public-Key Cryptosystems
2. Extended Euclid's Algorithm
3. Multiplicative Inverse Modulo N
4. Modular Exponentiation
5. Euler's Totient
6. RSA Key Generation
7. RSA

Introduction

The last theme of CS5000: F21 is number theory (NT) and cryptography. The Lecture 25 and 26 PDFs in Canvas that contain the materials on Extended Euclid, Multiplicative Inverse Modulo N, Modular Exponentiation, Euler's Totient and RSA Key Generation. You can start working on these modules. We'll do Euler's Totient, RSA Key Generation on Monday, Dec. 6, 2021.

In this assignent, we'll implement the version of the RSA system (named after Rivest, Shamir, and Adleman, the three mathematicians who invented it). I've broken the implementation into several modular sub-problems and written 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. Technically (tongue in cheek!), the last sub-problem on hacking RSA is not a component of RSA. But, you should be aware of how RSA can, in theory, be broken even though it is not that easy in practice.

I did my best to make the slides as self-sufficient as possible and skip marginal technical details from number theory (NT) and algorithms. When a theorem has a straightfoward proof, I sketch it. When a theorem requires a long and technical proof, I skip it, state the theorem, and give examples of how the theorem is applied.

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 (Slide 13 in CS5000_F21_Cryptography_Part02.pdf in Canvas). Specifically, implement the function $\texttt{xgcd}(a, b)$ in `rsa_aux.py` that uses the 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

Review slides 14 – 17 in CS5000_F21_Cryptography_Part02.pdf (and/or your class notes) and implement the function `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've written 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): Modular Exponentiation

Review slides 19 – 21 in CS5000_F21_Cryptography_Part02.pdf (and/or your class notes) and implement the function `mod_exp`(a, b, n) in `rsa_aux.py` that uses modular exponentiation to compute $a^b \bmod n$. The function uses the binary representation of b. The Python function `bin()` provides a straightforward way to obtain binary representations of integers. Here's how.

```
>>> bin(11)
'0b1011'
>>> bin(135)
'0b10000111'
```

As you can see from the above interaction, given an integer n, the function `bin`(n) returns strings with the prefix `'0b'` indicating that the subsequent characters are the binary representation of n. Since we don't need the prefix, we can chop it off with string slicing. Here's how.

```
>>> bin(132)[2:]
'10000100'
```

Use `test_mod_exp_01()` and `test_mod_exp_02()` in `rsa_uts.py` to test your implementation of `mod_exp()`.

Subproblem 1.4 ($\frac{1}{2}$ point): Euler's Totient

Review slides 22–36 in CS5000_F21_Cryptography_Part02.pdf (and/or your notes) and implement the function `euler_phi`(n) in `rsa_aux.py` that computes Euler's totient (i.e., $\phi(n)$). You don't have to understand or go deep into group theory if you don't know it. I've added these slides to give you a broader conceptual background for RSA. Use `test_euler_phi_01()` and `test_euler_phi_02()` in `rsa_uts.py` to test your implementation.

Subproblem 1.5 ($\frac{1}{2}$ point): Choosing e

Review slide 38 in CS5000_F21_Cryptography_Part02.pdf and implement the static method `rsa.choose_e(eu_phi_n)` in `rsa.py`. This is step 3 on the slide. We'll talk about it more on Dec. 6, 2021. 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 prime numbers. 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 1.6 ($\frac{1}{2}$ 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 (i.e., implements steps 4, 5, and 6 on slide 38 in CS5000_F21_Cryptography_Part02.pdf). Use `test_generate_keys_from_pqe()` in `rsa_uts.py` to test your implementation.

Subproblem 1.7 (1 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 (i.e., implement the equations on slide 39 in CS5000_F21_Cryptography_Part02.pdf). 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 1.8 ($\frac{1}{2}$ 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.

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 1.9 ($\frac{1}{2}$ point): Hacking RSA

The security of RSA rests, in large part, on the difficulty of factoring large integers. If the eavesdropper Eve can factor the modulos 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*.

Let's assume that Eve has C and $P = (e, n)$ (remember that P is publicly available!) and has factored 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 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

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. What's relevant is that 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. Gödel then goes even further and uses Euclid's gift to show limitations 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 mathematician, 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.

Quantum entaglement may well work not just across space but across time (if the latter exists independently of space, that is). Euler is entagled with Gödel and Euler and Sun-Tsu are entangled with Rivest, Shamir, and Adleman.

What to Submit?

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

Happy Hacking!