Number Theory and the RSA Public Key Cryptosystem

Author: Minh Van Nguyen <nguyenminh2@gmail.com>

This tutorial uses Sage to study elementary number theory and the RSA public key cryptosystem. A number of Sage commands will be presented that help us to perform basic number theoretic operations such as greatest common divisor and Euler's phi function. We then present the RSA cryptosystem and use Sage's built-in commands to encrypt and decrypt data via the RSA algorithm. Note that this tutorial on RSA is for pedagogy purposes only. For further details on cryptography or the security of various cryptosystems, consult specialized texts such as [MenezesEtAl1996], [Stinson2006], and [TrappeWashington2006].

Elementary number theory

We first review basic concepts from elementary number theory, including the notion of primes, greatest common divisors, congruences and Euler's phi function. The number theoretic concepts and Sage commands introduced will be referred to in later sections when we present the RSA algorithm.

Prime numbers

Public key cryptography uses many fundamental concepts from number theory, such as prime numbers and greatest common divisors. A positive integer $n > 1$ is said to be *prime* if its factors are exclusively 1 and itself. In Sage, we can obtain the first 20 prime numbers using the command `primes_first_n`:

```
sage: primes_first_n(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

Greatest common divisors

Let a and b be integers, not both zero. Then the greatest common divisor (GCD) of a and b is the largest positive integer which is a factor of both a and b. We use $\gcd(a, b)$ to denote this largest positive factor. One can extend this definition by setting $\gcd(0, 0) = 0$. Sage uses `gcd(a, b)` to denote the GCD of a and b. The GCD of any two distinct primes is 1, and the GCD of 18 and 27 is 9.

```
sage: gcd(3, 59)
1
sage: gcd(18, 27)
9
```

If $\gcd(a, b) = 1$, we say that a is *coprime* (or relatively prime) to b. In particular, $\gcd(3, 59) = 1$ so 3 is coprime to 59 and vice versa.

Congruences

When one integer is divided by a non-zero integer, we usually get a remainder. For example, upon dividing 23 by 5, we get a remainder of 3; when 8 is divided by 5, the remainder is again 3. The notion of congruence helps us to describe the situation in which two integers have the same remainder upon division by a non-zero integer. Let $a, b, n \in \mathbb{Z}$ such that $n \neq 0$. If a and b have the same remainder upon division by n, then we say that a is *congruent* to b modulo n and denote this relationship by

$$a \equiv b \pmod{n}$$

This definition is equivalent to saying that n divides the difference of a and b, i.e. $n \mid (a - b)$. Thus $23 \equiv 8 \pmod{5}$ because when both 23 and 8 are divided by 5, we end up with a remainder of 3. The command `mod` allows us to compute such a remainder:

```
sage: mod(23, 5)
3
sage: mod(8, 5)
3
```

Euler's phi function

Consider all the integers from 1 to 20, inclusive. List all those integers that are coprime to 20. In other words, we want to find those integers n, where $1 \leq n \leq 20$, such that $\gcd(n, 20) = 1$. The latter task can be easily accomplished with a little bit of Sage programming:

```
sage: for n in range(1, 21):
...       if gcd(n, 20) == 1:
...           print n,
...
1 3 7 9 11 13 17 19
```

The above programming statements can be saved to a text file called, say, `/home/mvngu/totient.sage`, organizing it as follows to enhance readability.

```
for n in xrange(1, 21):
    if gcd(n, 20) == 1:
        print n,
```

We refer to `totient.sage` as a Sage script, just as one would refer to a file containing Python code as a Python script. We use 4 space indentations, which is a coding convention in Sage as well as Python programming, instead of tabs.

The command `load` can be used to read the file containing our programming statements into Sage and, upon loading the content of the file, have Sage execute those statements:

```
load("/home/mvngu/totient.sage")
1 3 7 9 11 13 17 19
```

From the latter list, there are 8 integers in the closed interval $[1, 20]$ that are coprime to 20. Without explicitly generating the list

```
1  3  7  9  11  13  17  19
```

how can we compute the number of integers in $[1, 20]$ that are coprime to 20? This is where Euler's phi function comes in handy. Let $n \in \mathbb{Z}$ be positive. Then *Euler's phi function* counts the number of integers a, with $1 \leq a \leq n$, such that $\gcd(a, n) = 1$. This number is denoted by $\varphi(n)$. Euler's phi function is sometimes referred to as Euler's totient function, hence the name `totient.sage` for the above Sage script. The command `euler_phi` implements Euler's phi function. To compute $\varphi(20)$ without explicitly generating the above list, we proceed as follows:

```
sage: euler_phi(20)
8
```

How to keep a secret?

Cryptography is the science (some might say art) of concealing data. Imagine that we are composing a confidential email to someone. Having written the email, we can send it in one of two ways. The first, and usually convenient, way is to simply press the send button and not care about how our email will be delivered. Sending an email in this manner is similar to writing our confidential message on a postcard and post it without enclosing our postcard inside an envelope. Anyone who can access our postcard can see our message. On the other hand, before sending our email, we can scramble the confidential message and then press the send button. Scrambling our message is similar to enclosing our postcard inside an envelope. While not 100% secure, at least we know that anyone wanting to read our postcard has to open the envelope.

In cryptography parlance, our message is called *plaintext*. The process of scrambling our message is referred to as *encryption*. After encrypting our message, the scrambled version is called *ciphertext*. From the ciphertext, we can recover our original unscrambled message via *decryption*. The following figure illustrates the processes of encryption and decryption. A *cryptosystem* is comprised of a pair of related encryption and decryption processes.

```
+ ---------+   encrypt    +------------+   decrypt    +-----------+
| plaintext| -----------> | ciphertext | -----------> | plaintext |
+----------+              +------------+              +-----------+
```

The following table provides a very simple method of scrambling a message written in English and using only upper case letters, excluding punctuation characters.

```
+-------------------------------------------------+
| A   B   C   D   E   F   G   H   I   J   K   L   M  |
| 65  66  67  68  69  70  71  72  73  74  75  76  77 |
+-------------------------------------------------+
| N   O   P   Q   R   S   T   U   V   W   X   Y   Z  |
| 78  79  80  81  82  83  84  85  86  87  88  89  90 |
+-------------------------------------------------+
```

Formally, let

$$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{Z}\}$$

be the set of capital letters of the English alphabet. Furthermore, let

$$\Phi = \{65, 66, 67, \ldots, 90\}$$

be the American Standard Code for Information Interchange (ASCII) encodings of the upper case English letters. Then the above table explicitly describes the mapping $f : \Sigma \longrightarrow \Phi$. (For those familiar with ASCII, f is actually a common process for *encoding* elements of Σ, rather than a cryptographic "scrambling" process *per se*.) To scramble a message written using the alphabet Σ, we simply replace each capital letter of the message with its corresponding ASCII encoding. However, the scrambling process described in the above table provides, cryptographically speaking, very little to no security at all and we strongly discourage its use in practice.

Keeping a secret with two keys

The Rivest, Shamir, Adleman (RSA) cryptosystem is an example of a *public key cryptosystem*. RSA uses a *public key* to encrypt messages and decryption is performed using a corresponding *private key*. We can distribute our public keys, but for security reasons we should keep our private keys to ourselves. The encryption and decryption processes draw upon techniques from elementary number theory. The algorithm below is adapted from page 165 of [TrappeWashington2006]. It outlines the RSA procedure for encryption and decryption.

1. Choose two primes p and q and let $n = pq$.
2. Let $e \in \mathbf{Z}$ be positive such that $\gcd\left(e, \varphi(n)\right) = 1$
3. Compute a value for $d \in \mathbf{Z}$ such that $de \equiv 1 \pmod{\varphi(n)}$.
4. Our public key is the pair (n, e) and our private key is the triple (p, q, d).
5. For any non-zero integer $m < n$, encrypt m using $c \equiv m^e \pmod{n}$.
6. Decrypt c using $m \equiv c^d \pmod{n}$.

The next two sections will step through the RSA algorithm, using Sage to generate public and private keys, and perform encryption and decryption based on those keys.

Generating public and private keys

Positive integers of the form $M_m = 2^m - 1$ are called *Mersenne numbers*. If p is prime and $M_p = 2^p - 1$ is also prime, then M_p is called a *Mersenne prime*. For example, 31 is prime and $M_{31} = 2^{31} - 1$ is a Mersenne prime, as can be verified using the command `is_prime(p)`. This command returns `True` if its argument `p` is precisely a prime number; otherwise it returns `False`. By definition, a prime must be a positive integer, hence `is_prime(-2)` returns `False` although we know that 2 is prime. Indeed, the number $M_{61} = 2^{61} - 1$ is also a Mersenne prime. We can use M_{31} and M_{61} to work through step 1 in the RSA algorithm:

```
sage: p = (2^31) - 1
sage: is_prime(p)
True
sage: q = (2^61) - 1
sage: is_prime(q)
True
sage: n = p * q ; n
4951760154835678088235319297
```

A word of warning is in order here. In the above code example, the choice of p and q as Mersenne primes, and with so many digits far apart from each other, is a very bad choice in terms of cryptographic security. However, we shall use the above chosen numeric values for p

and q for the remainder of this tutorial, always bearing in mind that they have been chosen for pedagogy purposes only. Refer to [MenezesEtAl1996], [Stinson2006], and [TrappeWashington2006] for in-depth discussions on the security of RSA, or consult other specialized texts.

For step 2, we need to find a positive integer that is coprime to $\varphi(n)$. The set of integers is implemented within the Sage module `sage.rings.integer_ring`. Various operations on integers can be accessed via the `ZZ.*` family of functions. For instance, the command `ZZ.random_element(n)` returns a pseudo-random integer uniformly distributed within the closed interval $[0, n-1]$.

We can compute the value $\varphi(n)$ by calling the sage function `euler_phi(n)`, but for arbitrarily large prime numbers p and q, this can take an enormous amount of time. Indeed, the private key can be quickly deduced from the public key once you know $\varphi(n)$, so it is an important part of the security of the RSA cryptosystem that $\varphi(n)$ cannot be computed in a short time, if only n is known. On the other hand, if the private key is available, we can compute $\varphi(n) = (p-1)(q-1)$ in a very short time.

Using a simple programming loop, we can compute the required value of e as follows:

```
sage: p = (2^31) - 1
sage: q = (2^61) - 1
sage: n = p * q
sage: phi = (p - 1)*(q - 1); phi
4951760152529835076874141700
sage: e = ZZ.random_element(phi)
sage: while gcd(e, phi) != 1:
...       e = ZZ.random_element(phi)
...
sage: e  # random
185056762330061596630395 4877
sage: e < n
True
```

As `e` is a pseudo-random integer, its numeric value changes after each execution of `e = ZZ.random_element(phi)`.

To calculate a value for `d` in step 3 of the RSA algorithm, we use the extended Euclidean algorithm. By definition of congruence, $de \equiv 1 \pmod{\varphi(n)}$ is equivalent to

$$de - k \cdot \varphi(n) = 1$$

where $k \in \mathbb{Z}$. From steps 1 and 2, we already know the numeric values of e and $\varphi(n)$. The extended Euclidean algorithm allows us to compute d and $-k$. In Sage, this can be accomplished via the command `xgcd`. Given two integers x and y, `xgcd(x, y)` returns a 3-tuple `(g, s, t)` that satisfies the Bézout identity $g = \gcd(x, y) = sx + ty$. Having computed a value for `d`, we then use the command `mod(d*e, phi)` to check that `d*e` is indeed congruent to 1 modulo `phi`.

```
sage: n = 4951760154835678088235319297
sage: e = 185056762330061596630395 4877
sage: phi = 4951760152529835076874141700
sage: bezout = xgcd(e, phi); bezout  # random
(1, 4460824882019967172592779313, -16670957085153779250870330 35)
sage: d = Integer(mod(bezout[1], phi)) ; d  # random
```

```
4460824882019967172592779313
sage: mod(d * e, phi)
1
```

Thus, our RSA public key is

$$(n, e) = (4951760154835678088235319297, 185056762330061596630395487 7)$$

and our corresponding private key is

$$(p, q, d) = (2147483647, 2305843009213693951, 4460824882019967172592779313)$$

Encryption and decryption

Suppose we want to scramble the message `HELLOWORLD` using RSA encryption. From the above ASCII table, our message maps to integers of the ASCII encodings as given below.

```
+----------------------------------------+
| H   E   L   L   O   W   O   R   L   D  |
| 72  69  76  76  79  87  79  82  76  68 |
+----------------------------------------+
```

Concatenating all the integers in the last table, our message can be represented by the integer

$$m = 72697676798779827668$$

There are other more cryptographically secure means for representing our message as an integer. The above process is used for demonstration purposes only and we strongly discourage its use in practice. In Sage, we can obtain an integer representation of our message as follows:

```
sage: m = "HELLOWORLD"
sage: m = map(ord, m); m
[72, 69, 76, 76, 79, 87, 79, 82, 76, 68]
sage: m = ZZ(list(reversed(m)), 100) ; m
72697676798779827668
```

To encrypt our message, we raise m to the power of e and reduce the result modulo n. The command `mod(a^b, n)` first computes `a^b` and then reduces the result modulo `n`. If the exponent `b` is a “large” integer, say with more than 20 digits, then performing modular exponentiation in this naive manner takes quite some time. Brute force (or naive) modular exponentiation is inefficient and, when performed using a computer, can quickly consume a huge quantity of the computer’s memory or result in overflow messages. For instance, if we perform naive modular exponentiation using the command `mod(m^e, n)`, where `m`, `n` and `e` are as given above, we would get an error message similar to the following:

```
mod(m^e, n)
Traceback (most recent call last)
/home/mvngu/<ipython console> in <module>()
/home/mvngu/usr/bin/sage-3.1.4/local/lib/python2.5/site-packages/sage/rings/integer.so
in sage.rings.integer.Integer.__pow__ (sage/rings/integer.c:9650)()
RuntimeError: exponent must be at most 2147483647
```

There is a trick to efficiently perform modular exponentiation, called the method of repeated squaring, cf. page 879 of [CormenEtAl2001]. Suppose we want to compute $a^b \mod n$. First, let $d := 1$ and obtain the binary representation of b, say $(b_1, b_2, \ldots, b_k)$ where each $b_i \in \mathbf{Z}/2\mathbf{Z}$. For $i := 1, \ldots, k$, let $d := d^2 \mod n$ and if $b_i = 1$ then let $d := da \mod n$. This algorithm is implemented in the function `power_mod`. We now use the function `power_mod` to encrypt our message:

```
sage: m = 72697676798779827668
sage: e = 185056762330061596630395 4877
sage: n = 49517601548356780882353192 97
sage: c = power_mod(m, e, n); c
63091363257752005841552109 0
```

Thus $c = 630913632577520058415521090$ is the ciphertext. To recover our plaintext, we raise `c` to the power of `d` and reduce the result modulo `n`. Again, we use modular exponentiation via repeated squaring in the decryption process:

```
sage: m = 72697676798779827668
sage: c = 63091363257752005841552109 0
sage: d = 44608248820199671725927793 13
sage: n = 49517601548356780882353192 97
sage: power_mod(c, d, n)
72697676798779827668
sage: power_mod(c, d, n) == m
True
```

Notice in the last output that the value 72697676798779827668 is the same as the integer that represents our original message. Hence we have recovered our plaintext.

Acknowledgements

1. 2009-07-25: Ron Evans (Department of Mathematics, UCSD) reported a typo in the definition of greatest common divisors. The revised definition incorporates his suggestions.
2. 2008-11-04: Martin Albrecht (Information Security Group, Royal Holloway, University of London), John Cremona (Mathematics Institute, University of Warwick) and William Stein (Department of Mathematics, University of Washington) reviewed this tutorial. Many of their invaluable suggestions have been incorporated into this document.

Bibliography

[CormenEtAl2001] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, USA, 2nd edition, 2001.

[MenezesEtAl1996] *(1, 2)* A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, USA, 1996.

[Stinson2006] *(1, 2)* D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, Boca Raton, USA, 3rd edition, 2006.

[TrappeWashington2006] *(1, 2, 3)* W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Pearson Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 2006.