Number Theory and the RSA Public Key Cryptosystem

Minh Van Nguyen

`nguyenminh2@gmail.com`

Version 1.1

2010 October 14

This tutorial uses Sage to study elementary number theory and the RSA public key cryptosystem. A number of Sage commands will be introduced that help us to perform basic number theoretic operations such as greatest common divisor and Euler's phi function. We then introduce the RSA cryptosystem and use Sage's built-in commands to encrypt and decrypt data via the RSA algorithm. Please note that this tutorial on RSA is for pedagogy purposes only. For further details on cryptography or the security of various cryptosystems, please consult specialized texts such as [2, 4, 5].

1 What is Sage?

Sage [3] is an open source computer algebra system that supports teaching, study and research in mathematics. It combines numerous open source packages and provides access to their functionalities via a common interface, namely, a Python based programming language. Sage can be used as a powerful desktop calculator, as a tool to help undergraduate students study mathematics, or as a programming environment for prototyping algorithms and research in algorithmic aspects of mathematics. Sage is available free of charge and can be downloaded from the following website:

<div align="center">

`http://www.sagemath.org`

</div>

The default interface to Sage is command line based, but there is a graphical user interface to the software as well in the form of the Sage notebook. Upon loading Sage from the command line, we are presented with something similar to the following:

```
─────────────────────────── Sage ───────────────────────────
[mvngu@darkstar ~]$ sage
----------------------------------------------------------------
| SAGE Version 3.1.4, Release Date: 2008-10-20                 |
| Type notebook() for the GUI, and license() for information.  |
----------------------------------------------------------------

sage:
```

Plenty of help is provided in the form of the official Sage documentation that is distributed with every release of Sage. From within a Sage session, we can obtain a list of commands matching some pattern. To do so, we type the first few characters and then press the "Tab" key:

```
──────────────────────── Sage ────────────────────────
sage: prime[TAB]
prime_divisors    prime_pi        prime_range      primes
prime_factors     prime_powers    prime_to_m_part  primes_first_n
```

If we know the exact name of a command, we can use the `help` function to obtain further information on that command. For example, the command `help(primes)` provides documentation on the built-in function `primes`, as illustrated in the following help session:

```
──────────────────────── Sage ────────────────────────
Help on function primes in module sage.rings.arith:

primes(start, stop=None)
    Returns an iterator over all primes between start and stop-1,
    inclusive.  This is much slower than \code{prime_range}, but
    potentially uses less memory.

    This command is like the xrange command, except it only iterates
    over primes.  In some cases it is better to use primes than
    prime_range, because primes does not build a list of all primes in
    the range in memory all at once.  However it is potentially much
    slower since it simply calls the \code{next_prime} function
    repeatedly, and \code{next_prime} is slow, partly because it
    proves correctness.

    EXAMPLES:
        sage: for p in primes(5,10):
        ...       print p
        ...
        5
        7
        sage: list(primes(11))
        [2, 3, 5, 7]
        sage: list(primes(10000000000, 10000000100))
        [10000000019, 10000000033, 10000000061, 10000000069, 10000000097]
```

To quit the above help session, we simply press "q" on our keyboard. For further assistance on specific problems, we can also search the archive of the `sage-support` mailing list at

> http://groups.google.com/group/sage-support

or send an email to `sage-support@googlegroups.com`.

2 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 ───────────────────────────────────
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 ───────────────────────────────────
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, that is, $n \mid (a - b)$. 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 ───────────────────────────────────
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 ────────
sage: for n in xrange(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 in Listing 1 to enhance readability. 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.

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

Listing 1: Finding all integers $1 \leq n \leq 20$ such that $\gcd(n, 20) = 1$.

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:

```
──────── Sage ────────
sage: 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

$$1 \quad 3 \quad 7 \quad 9 \quad 11 \quad 13 \quad 17 \quad 19 \tag{1}$$

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 list (1), we proceed as follows:

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

3 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*. Figure 1 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 │
└───────────┘             └───────────┘             └───────────┘
```

Figure 1: A schema for encryption and decryption.

Table 1 provides a very simple method of scrambling a message written in English and using only upper case letters, excluding punctuation characters. Formally, let $\Sigma = \{A, B, C, \ldots, Z\}$ 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 Table 1 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 Table 1 provides, cryptographically speaking, very little to no security at all and we strongly discourage its use in practice.

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

Table 1: ASCII encodings of English capital letters.

4 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. Algorithm 1 [5, p.165] outlines the RSA procedure for encryption and decryption.

1	Choose two primes p and q and let $n = pq$.
2	Let $e \in \mathbb{Z}$ be positive such that $\gcd\big(e, \varphi(n)\big) = 1$.
3	Compute a value for $d \in \mathbb{Z}$ 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 $m \equiv c^d \pmod{n}$.

Algorithm 1: The RSA algorithm for encryption and decryption.

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.

5 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 Algorithm 1:

```
——————————————————————————— Sage ———————————————————————————
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. Please refer to [2, 4, 5] 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 $[0, n-1]$. Using a simple programming loop, we can compute the required value of e as follows:

```
                          ─── Sage ───
sage: e = ZZ.random_element(euler_phi(n))
sage: while gcd(e, euler_phi(n)) != 1:
....:      e = ZZ.random_element(euler_phi(n))
....:
sage: e
1850567623300615966303954877
sage: e < n
True
```

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

To calculate a value for d in step 3 of Algorithm 1, 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 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, euler_phi(n))` to check that `d*e` is indeed congruent to 1 modulo `euler_phi(n)`:

```
                          ─── Sage ───
sage: bezout = xgcd(e, euler_phi(n)) ; bezout
(1, 4460824882019967172592779313, -1667095708515377925087033035)
sage: d = Integer(mod(bezout[1], euler_phi(n))) ; d
4460824882019967172592779313
sage: mod(d * e, euler_phi(n))
1
```

Thus, our RSA public key is

$$(n, e) = (4951760154835678088235319297,\ 1850567623300615966303954877) \quad (2)$$

and our corresponding private key is

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

H	E	L	L	O	W	O	R	L	D
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
72	69	76	76	79	87	79	82	76	68

Table 2: ASCII encodings of `HELLOWORLD`.

6 Encryption and decryption

Suppose we want to scramble the message "`HELLOWORLD`" using RSA encryption. From Table 1, our message maps to integers of the ASCII encodings as given in Table 2. Concatenating all the integers in Table 2, our message can be represented by the integer

$$m = 72697676798779827668 \tag{3}$$

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 —————————————————————————————
sage: m = "HELLOWORLD"
sage: m = [ord(c) for c in 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` is per (3), and `n` and `e` are as in (2), we would get an error message similar to the following:

```
————————————————————————————— Sage —————————————————————————————
sage: mod(m^e, n)
--------------------------------------------------------------------
RuntimeError                        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 [1, p.879]. 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 $b_i \in \mathbb{Z}_2$. 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 Listing 2, which defines a Sage function called `modexp` to perform modular exponentiation using repeated squaring.

```
1  def modexp(a, b, n):
2      """
3      Modular exponentiation using repeated squaring.
4      That is, we want to compute a^b mod n.
5
6      INPUT
7      a -- a positive integer
8      b -- a positive integer
9      n -- a positive integer
10
11     OUTPUT
12     a^b mod n
13     """
14     d = 1
15     for i in list(Integer.binary(b)):
16         d = mod(d * d, n)
17         if Integer(i) == 1:
18             d = mod(d * a, n)
19     return Integer(d)
```

Listing 2: Sage function for modular exponentiation via repeated squaring.

The code in Listing 2 can be saved to a file called, say, `/home/mvngu/modexp.sage`. The file can then be read into Sage using the command `load`. Let us read the above file into Sage and encrypt our message:

```
———————————— Sage ————————————
sage: load "/home/mvngu/modexp.sage"
sage: c = modexp(m, e, n) ; c
63091363257752005841552109 0
```

Thus $c = 63091363257752005841552109 0$ 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 ————————————
sage: modexp(c, d, n)
72697676798779827668
sage: m
72697676798779827668
```

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.

7 Further discussion

This tutorial has introduced some basic programming from within a Sage session, in particular the code Listings 1 and 2. The official tutorial on Sage can be found online at `http://www.sagemath.org/doc/tut/tut.html` or in your local installation of Sage at the directory `SAGE_ROOT/devel/sage/doc/output/html/en/tutorial/index.html`. In order to make effective use of Sage and extend its functionalities, you are encouraged to consult the official Python documentation at `http://docs.python.org`.

Some code examples presented in this tutorial are said to be un-Pythonic, in the sense that they do not make effective use of various features of the Python programming language. For instance, the function `modexp` in Listing 2 can also be implemented as follows:

```
1  def modexp(a, b, n):
2      d = 1
3      for i in reversed(b.digits(base=2)):
4          d = mod(d * d, n)
5          if i == 1:
6              d = mod(d * a, n)
7      return Integer(d)
```

Listing 3: A Pythonic version of `modexp`.

Sage modules in the directory `SAGE_ROOT/devel/sage-main/sage/crypto` can be used to complement a course in cryptography. David Kohel's notes at

$$\texttt{http://www.sagemath.org/library/crypto.pdf}$$

uses Sage to teach a course in cryptography. Now it is over to you to explore what you can do using Sage.

Acknowledgements

References

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

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

[3] W. Stein. *Sage: Open Source Mathematics Software (version 3.1.4).* The Sage Group, 03 November 2008.
http://www.sagemath.org.

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

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