7-2009

Public Key Cryptography and the RSA Cryptosystem

Nuh Aydin
Kenyon College, aydinn@kenyon.edu

Public Key Cryptography and the RSA Cryptosystem

A Computational Science Module

Supported by

National Science Foundation, NSF CCLI Grant (DUE 0618252)

Module Author: Nuh Aydin

Department of Mathematics

Kenyon College

aydinn@kenyon.edu

July 2009

To the Instructor

Ideas from computational complexity play a very fundamental role in cryptography. A formal prerequisite for this module is familiarity with basic concepts of algorithm analysis and computational complexity, including the big-O notation. Students are also assumed to be able to do basic programming in a language such as C, C++ or Java. Necessary material from elementary number theory is covered in the module, though familiarity with modular arithmetic and some very basic logic (such as proof by contradiction) will be useful. Since mostly number theoretical algorithms are discussed in this module, it is very important to emphasize that if the input to such an algorithm is an integer n, then the size of the input is taken to be $\lfloor \log_2(n) \rfloor + 1$, number of bits needed to store n, not the magnitude of n. This is a point that is easy for students to forget, therefore it is worth repeating. While providing necessary background in elementary number theory, students will be given a gentle introduction to some fundamental notions in abstract algebra such as groups, rings, and fields. Students who already have this background may skip that part. (Even if they saw the material before, a quick review would not hurt). If the mathematical maturity of the students is not at the right level, some of the proofs could be optional. The software used in this module is Maple 12, though it should also work with some of the earlier versions of Maple. There are several projects to choose from at the end. Given the nature of the subject some of these projects are research oriented and not completely prescribed. Others are of hands-on type and also not completely prescribed, some of the details need to be determined by the instructor. The exercises inserted throughout the module and the projects at the end can be used as an assessment tool.

Abstract

Cryptography, or cryptology, is a subject that is concerned with privacy or confidentiality of communication over insecure channels, in the presence of adversaries. It seeks to find ways to encrypt messages so that even if an unauthorized party gets a hold of a message, they cannot make sense out of it. The ways to break encryption systems, called cryptanalysis, is also part of the subject. Cryptography is sometimes confused with the related but distinct field of coding theory that deals with reliability of communication over noisy channels. See the author's earlier module titled "An introduction to coding theory via Hamming codes" for an introduction to coding theory.

There are two basic methods in cryptography: classical cryptography and public key cryptography. The latter is a more recent idea and this module will focus on that method through one of its best known and widely used examples: RSA cryptosystem. Proposed in 1977, the RSA cryptosystem has survived many attacks and is still commonly used.

KEYWORDS: cryptography, public key cryptography, classical cryptography, computational complexity, elementary number theory, one-way functions, RSA cryptosystem.

Introduction

The Main Problem

Cryptography is primarily concerned with confidentiality of communication over insecure channels. There are three generic characters that are often referred to in the subject. *Alice* and *Bob* try to communicate confidentially, and *Eve* is the eavesdropper. (Sometimes this character is called “Oscar” or “Marvin”.) Alice and Bob need to find ways to scramble their messages so that even if Eve intercepts some of their messages she should not be able to make sense out of them, at least not in any reasonable amount of time. Ideally, Alice and Bon prefer to have a system with *unconditional security* where Eve cannot gain any knowledge about the messages she may intercept no matter how many she obtains and how much computational power she has. However, in many situations having unconditional security is not possible or practical. In those cases they will be content with *computational security*: breaking the system is not feasible with best known cryptanalytic techniques and available computational resources.

Cryptography has a long history. People have always been interested in keeping some information secret from others. Rulers, emperors, politicians, government and military personnel often need to communicate confidential information and hence need cryptography. There is a well-known cipher named after Julius Caesar. Cryptography and cryptanalysis proved to be a decisive factor in World War II. In today’s world, online and electronic transactions are so common that virtually everyone uses cryptography (knowingly or not) to protect private information (web sites that take sensitive information are set up so that the data is encrypted before being sent).

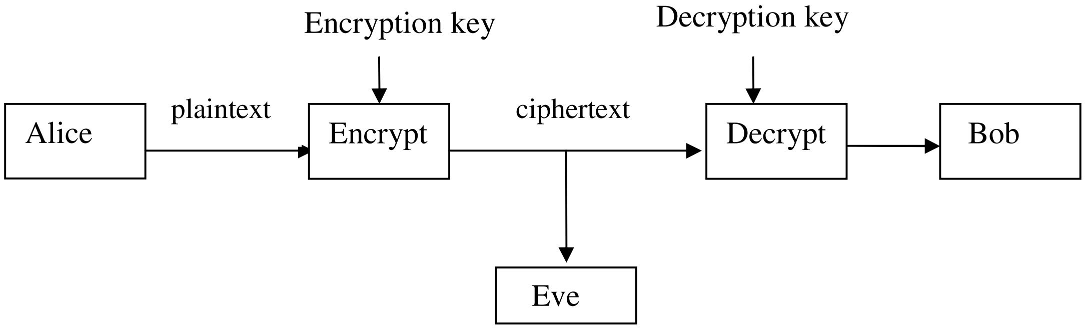
Formal Set Up and Definitions

To study cryptography in a systematic way, we need to set up a mathematical framework and define some terms. First, we need an *alphabet* $\mathcal{A}$ with a finite set of symbols to form messages. The set $\mathcal{M}$ of messages consists of all strings of symbols from $\mathcal{A}$. For a positive integer n $\mathcal{A}^n$ denotes the set of all strings of length n over the alphabet $\mathcal{A}$ and $\mathcal{A}^*$ represents the set of all strings of any finite length over $\mathcal{A}$. In some cryptosystems the messages are required to be of a fixed length. A message m in $\mathcal{M}$ before it is encrypted is called a *plaintext*. Therefore, $\mathcal{M}$ is the set of all plaintexts. A message after encryption is called a *ciphertext*. We shall denote the set of all ciphertexts by $\mathcal{C}$. Ciphertexts are obtained from plaintexts through an encryption function. An encryption function is a function $E_k : \mathcal{M} \to \mathcal{C}$ where k is a key. Given a plaintext m, the encryption function produces the corresponding ciphertext $c = E_k(m)$. Intuitively, an encryption function locks a message, i.e., makes it unreadable, using the key k, like locking a box. We also need a decryption function $D_k : \mathcal{C} \to \mathcal{M}$ for each encryption function. Note that the encryption and decryption functions must be related by $D_k(E_k(m)) = m$ for all m in $\mathcal{M}$. This requires the encryption function to be one-to-one. Intuitively, decryption function unlocks the message, like opening a safe. We can summarize all of this in a definition.

Definition: A cryptosystem is a five-tuple ($\mathcal{M}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D}$) with the following properties.

1. $\mathcal{M}$ is the set of all possible plaintexts.
2. $\mathcal{C}$ is the set of all ciphertexts.
3. $\mathcal{K}$ is the keyspace, a finite set of possible keys.
4. For each k in $\mathcal{K}$ there is an encryption function $E_k : \mathcal{M} \to \mathcal{C}$ and a corresponding decryption function $D_k : \mathcal{C} \to \mathcal{M}$ such that $D_k(E_k(m)) = m$ for all m in $\mathcal{M}$

A general picture of confidential communication in the presence of adversaries can be depicted as follows.



Problem 1 Show that the condition $D_k(E_k(m)) = m$ for all m in $\mathcal{M}$ implies that $E_k : \mathcal{M} \to \mathcal{C}$ is a one-to-one function.

There are two main types of cryptographic systems: symmetric key (or classical) and public key cryptosystems. The distinction comes from how the keys are used. In the former, the encryption and decryption keys are closely related (either the same or one is easily obtained from the other) and they are both secret. In a public key cryptosystem the encryption key is public, and the decryption key is private. Therefore, the keys come in two parts $k = (e, d)$, where e is the public part of the key (the encryption key), and d is the private decryption key. In such systems anyone can lock messages but only the owner of the private key can unlock them.

Adversaries are classified by their capability. A *passive adversary* only listens to the messages intercepted in the unsecured portion of the communication channel between Alice and Bob. An *active adversary* on the other hand may alter messages or replace them with her own, in addition to eavesdropping. A good cryptosystem should be able to detect forged messages. The goal of eavesdropper is to break the cryptosystem which means that she can systematically obtain plaintexts from ciphertexts. We defined two types of security above (unconditional and computational). A third type is called *provable security* which means that breaking the system is as difficult as solving a mathematical problem which is believed to be intractable.

A fundamental assumption in cryptography is *Kerckhoffs' principle* which states that the security of a cryptosystem should rest on the keys alone, not the assumption that the enemy (eavesdropper) does not know the encryption and decryption schemes. Therefore, we always assume that the encryption and decryption systems are known to the adversary, and analyze the security of the system based on the keys. It is very important that encryption and decryption (with the possession of the private key) should be efficient (computationally inexpensive) whereas decryption without the private key should be computationally difficult.

There are several types of attacks on cryptosystems. In the weakest form, called *ciphertext-only attack*, the adversary attempts to obtain plaintexts or the decryption key by accessing some number of ciphertexts. A cryptosystem that is vulnerable to this type of attack is considered to be totally insecure. If an adversary is in possession of a certain number of ciphertexts with the corresponding plaintexts, this is called *known-plaintext attack*. In a *chosen-plaintext attack* the adversary selects a set of plaintexts and obtains the corresponding ciphertexts. This is equivalent to having temporary access to the encryption machine without accessing the key. If the adversary has temporary access to the decryption machine (but not the private key), then she can choose a set of ciphertexts and obtain the corresponding plaintexts. This is called *chosen-ciphertext attack*.

Symmetric-Key Cryptosystems

Although this module will focus on the more modern notion public key cryptosystems, we will have a brief discussion and a few examples of symmetric-key, or classical, cryptosystems as well. As defined earlier, in such systems the encryption and decryption keys are either the same or one is easily obtained from the other. Therefore, both keys must be kept secure. Below are a few examples of symmetric-key systems.

Example Let $\mathcal{A} = \{a_0, a_1, ..., a_{n-1}\}$ be an alphabet of size n ,and let $0 \le k < n$. The encryption is obtained by shifting every symbol in a message by k positions modulo n, i.e. $a_i \rightarrow a_{(i+k) \bmod n}$. This is called a shift cipher. For the English alphabet, $n = 26$ and we can identify each letter with numbers $A \leftrightarrow 0, B \leftrightarrow 1, ..., Z \leftrightarrow 25$ for the sake of convenience. We can then convert a message to a sequence of numbers and do encryption and decryption with integers using addition modulo 26. If for example, $k = 3$ (this is often called the Caesar cipher) the message (plaintext) NEWYORK is converted to the number 13042224141710 (where we write 04 for 4) which is encrypted as 16072502172013 (using addition mod 26) which corresponds to the ciphertext QHZCRUN. Decryption is done by subtracting 3 modulo 26 (equivalently adding 23 mod 26). Note that the encryption and decryption keys are the same for the shift cipher. What is the security of the shift cipher? Clearly, there are only 26 possible keys and it is not difficult to test them all. So a brute force, exhaustive search on the key space is possible because it is so small. Therefore, the system is vulnerable to ciphertext-only attack (remember we assume that the adversary knows the system being used). Also note that, the chosen-plaintext attack breaks the system immediately.

Problem 2: Is there a value of k for the shift cipher on the English alphabet for which the encryption and decryption functions are identical?

The shift cipher is an example of a more general *substitution cipher* in which a permutation on the alphabet is applied to each symbol of a message. A permutation π on $\mathcal{A}$ is a mapping π:

$\mathcal{A} \to \mathcal{A}$ that is one-to-one and onto. Therefore, a message $m = (m_0, m_1, ..., m_{n-1})$ is encrypted as $c = \pi(m) = (\pi(m_0), \pi(m_1), ..., \pi(m_{n-1}))$. A permutation π is the encryption key, and its inverse π^{-1} is the decryption key.

Problem 3: What is the size of the key space for the substitution cipher on the English alphabet?

Although the key space for a substitution cipher may be too large to do an exhaustive search for the key, there is another technique that is usually effective for breaking this system. It makes use of the known frequencies of letters for a given alphabet. The following table gives relative approximate frequencies of single characters in English alphabet in two ways [9], [13].

By Letter		By Frequency	
Letter	Frequency	Letter	Frequency
A	.082	E	.127
B	.015	T	.091
C	.028	A	.082
D	.043	O	.075
E	.127	I	.070
F	.022	N	.067
G	.020	S	.063
H	.061	H	.061
I	.070	R	.060
J	.002	D	.043
K	.008	L	.040
L	.040	C	.028
M	.024	U	.028
N	.067	M	.024
O	.075	W	.023
P	.019	F	.022
Q	.001	G	.020
R	.060	Y	.020
S	.063	P	.019
T	.091	B	.015
U	.028	V	.010
V	.010	K	.008
W	.023	J	.002
X	.001	X	.001
Y	.020	Q	.001
Z	.001	Z	.001

Moreover, it is useful to know the frequencies of the most common bigrams and trigrams. Those frequencies in order are as follows:

Bigrams: th, he, in, en, nt, re, er, an, ti, es, on, at, se, nd, or, ar, al, te, co, de, to, ra, et, ed, it

Trigrams: the, and, tha, ent, ing, ion, tio, for, nde, has, nce, edt, tis, oft, sth, men

Problem 4: Given that the following text is encrypted using a substitution cipher, decrypt the ciphertext and determine the encryption key (permutation). There are no spaces or punctuation marks in the text.

FOEWZKNTZGKSBLNTZJOZZFGMEVNTZEVNHXLTVKYSNTGOATNNTVNHNQGO
DYTVPZVWLGDONZDSFGIKVXNHXVDVIIDHXVNHGFLNTZKLVXKSINGLSLNZEV
FYGNTZKVIIDHXVNHGFLGMFOEWZKNTZGKSIKGPZYNTHLAKZVNEVNTZEVNH
XHVFQKGFA

Example: One-time pad (Vernam cipher). For this cipher the alphabet is the binary alphabet $\mathcal{A}=\{0,1\}$. (Since data of any kind is stored and processed as binary digits in computers, in principal the binary alphabet is sufficient for all situations). The message space, ciphertext space and key space are all equal to $\mathcal{A}^*$, the set of all finite binary sequences. A key k to encrypt a message m is chosen randomly and has the same length as m. The encryption is the bit-wise addition of m and k modulo 2 denoted by $\oplus$. So, the ciphertext is $c = m \oplus k$. For example, if $m = 0011010110$ and $k = 1001000101$ then $c = 1010010011$. When the keys are chosen randomly and they are used only once, this cryptosystem is unconditionally secure against ciphertext-only attack. More specifically, if the length of the messages is n, then the size of the key space for a single message is 2^n and the average complexity of searching the key space is O(2^n). Violation of this requirement (using a key only once) by the Russians during World War II let US National Security Agency crack Soviet messages with VENONA project [9], [15].

Problem 5: Explain what happens if two messages are encrypted with the same key in Vernam cipher.

Although the Vernam cipher has unconditional security, it has a drawback: Key size is very large. To achieve the security a key needs to be used only once, needs to be as large as the total length of all messages to be sent, and needs to be exchanged securely. These requirements make this system not very practical in many situations.

There are many other examples of symmetric key ciphers. You can see more examples in books on cryptography such as [4],[9],[11],[13].

Elementary Number Theory for the RSA Cryptosystem

The RSA cryptosystem is based on some elementary facts from number theory. In this section we introduce and review some material from elementary number theory needed for the RSA cryptosystem. The reason number theory is useful and needed is that all messages will be regarded as numbers modulo some integer. It is not difficult to convert text messages to numbers. We will also review how to do that.

You are familiar with the set of integers, denoted by $\mathbb{Z}$. There are two basic operations on integers addition and multiplication. Note that subtraction is nothing but adding the negative (the additive inverse) i.e., $x-y$ is the same as $x+(-y)$. The ratio of two integers is not always another integer, so division is not a well defined binary operation on (non-zero) integers. Similarly to subtraction, division is a form of multiplication, multiplying with the reciprocal. So, $\frac{x}{y}$ is the same as $x\cdot\frac{1}{y}$ where $y^{-1}=\frac{1}{y}$ is the multiplicative inverse of y. One property that $\mathbb{Z}$ lacks is not every non-zero element of $\mathbb{Z}$ has a multiplicative inverse (reciprocal) in $\mathbb{Z}$. For instance, the inverse of 2 is $\frac{1}{2}$ but it is not an integer. Algebraically, the set of integers has the following properties with the operations of + and $\cdot$.

1. There is an identity element for +, namely 0: $x+0=0+x=x$ for all x in $\mathbb{Z}$.
2. + is associative, i.e., $(x+y)+z=x+(y+z)$ for all integers x, y, z.
3. + is commutative, i.e. $x+y=y+x$ for all x, y in $\mathbb{Z}$.
4. Every x in $\mathbb{Z}$ has an additive inverse: $x+(-x)=0$
5. Multiplication is associative: $(x\cdot y)\cdot z=x\cdot(y\cdot z)$ for all integers x, y, z.
6. There is an identity element for multiplication, namely 1: $x\cdot 1=1\cdot x=x$ for all x in $\mathbb{Z}$.
7. Multiplication is commutative: $x\cdot y=y\cdot x$ for all x, y in $\mathbb{Z}$.
8. Multiplication distributes over addition: $x\cdot(y+z)=x\cdot y+x\cdot z$ for all integers x, y, z.
9. If $x\cdot y=0$ for any x, y in $\mathbb{Z}$, then either $x=0$ or $y=0$ (or both).

A set with two binary operations that satisfy these properties is called an *integral domain*. If, in addition to all of these properties, every non-zero element has a multiplicative inverse the system is called a *field*. Some examples of fields are $\mathbb{Q}$, the set of rational numbers; $\mathbb{R}$, the set of real numbers; and $\mathbb{C}$, the set of complex numbers. If all the properties except for the last one are satisfied it is called a *ring* (more accurately, a commutative ring unity). Another way of stating property 9 is to say that $\mathbb{Z}$ does not have any zero divisors. There are algebraic systems in which this property fails, i.e. there exist non-zero elements a and b such that $a\cdot b=0$. Such elements are called *zero divisors*. Examples of rings that are not integral domains (rings with zero divisors) are integers modulo n, for some non-prime integer n. We will discuss these structures in more details in the next few pages.

Computational Complexity of Basic Arithmetic Operations

Since computational complexity is a fundamental notion in cryptography, it will be useful to review running times of basic arithmetic operations. For number theoretical algorithms and operations, the measure of efficiency is usually considered to be the number of bitwise operations involving single-bit arithmetic. For this reason, the size of the input for number theoretical algorithms is taken to be number of bits needed to store (or represent) the number. For an integer n, the number of bits needed to store it in computer memory is $\lfloor \log_2 n \rfloor + 1$. For example, the binary representation of $n=13$ is 1101 and $\lfloor \log_2 13 \rfloor + 1 = 3+1 = 4$. Since the big-O notation is not sensitive to the base of the logarithm, and additive (and multiplicative) constants, we sometimes write the size of the input as $\log(n)$ when analyzing the asymptotic running time of a number theoretical algorithm whose input is an integer n. For two integers

x, y computing their sum takes at most O(n) bit operations, where n is the maximum of the number of bits to represent these numbers, and their product takes at most O(n^2) operations.

Divisibility, Prime Numbers and the Euclidean Algorithm

For $a, b \in \mathbb{Z}$ we say that a *divides* b, denoted $a \mid b$, if there exists $c \in \mathbb{Z}$ such that $b = c \cdot a$. In this case, a is called a factor, or divisor of b. For example, 3|21 since 21=3 $\cdot$7. It is easy to show that for any $n \in \mathbb{Z}$, $n \mid 0$, $n \mid n$, and $1 \mid n$. 1 and n are called trivial factors of n. An integer $p > 1$ that does not have any non-trivial factors is called a *prime*. Note that 1 is not a prime by definition. A basic property of divisibility is the following:

Problem 6: Suppose $a \mid x$ and $a \mid y$. Show that $a \mid (x \cdot n + y \cdot m)$ for any integers n and m.

This means that if a divides both x and y, then it also divides any of their linear combinations.

Prime numbers are very important in number theory because of

Fundamental Theorem of Arithmetic: Every integer $n > 2$ can be written as a product of powers of primes in a unique way (unique up to the ordering of the factors) in the form $n = p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$ where $p_1, p_2, ..., p_k$ are primes and $r_1, r_2, ..., r_k$ are positive integers.

This theorem says prime numbers are building blocks of integers. Any integer is made up of prime numbers. It also says that any integer greater than 2 is either a prime itself, or divisible by a prime. It is well known that there are infinitely many primes. Euclid may have been the first one to prove this result around 300 BC. There are many proofs of this important fact. Even after more than 2000 years, Euclid's original proof remains to be a most elegant argument. In addition to its elegance and pure number theoretical significance, this classical result is crucially important for the security of the RSA cryptosystem.

Problem 7: Show that there are infinitely many primes.

If $a \mid x$ and $a \mid y$ then a is called a common divisor of x and y. For example, 4 is a common divisor of 24 and 36, so are 2, 6 and 12. Among the common divisors of two numbers there is a largest one, called the *greatest common divisor* (gcd). We denote the greatest common divisor of two integers x and y by $\gcd(x, y)$ or simply (x, y). Therefore, (24,36) = 12. If the gcd of two integers is 1, then they are called relatively prime. For example, 4 and 9 are relatively prime integers (note that neither is a prime). Another useful result in elementary number theory is the following theorem, called the Division Algorithm.

The Division Algorithm: Let n and m be integers such that $m \neq 0$. Then there exist unique integers q and r such that $n = mq + r$ where $0 \leq r < |m|$. Here q is called the quotient, and r is called the remainder when n is divided by m.

For example, when 24 is divided by 9 the quotient is 2, and the remainder is 6. We write,

$24=9\cdot 2+6$. If -24 is divided by 9, then the quotient is -3 and the remainder is 3. If 24 is divided by -9, then the quotient is -2 and the remainder is 6. In all cases the remainder is less than the absolute value of the divisor.
Actually, it is a little misleading to call this theorem an "algorithm" because it does not tell us how to find the quotient and remainder, though you are familiar with the long division of integers which gives you those numbers. Nevertheless, this result is a key ingredient for a number of important number theoretical algorithms such as the Euclidean algorithm. When you want to prove that one integer x divides another one, say y, it is a common proof technique to apply the division algorithm to write $y = xq + r$ and try to show that the remainder must be 0.
There is another useful description of the gcd of two numbers. It is the common divisor of the two numbers that is divisible by any of the common divisors. You may want to try to prove that this description of gcd is equivalent to the previous one. Another useful property of gcd is that it can be written as a linear combination of the numbers.

Theorem: Let $d = (x, y)$. Then there exist integers a and b (not necessarily unique) such that $d = ax + by.$
There is a well known procedure called the Euclidean algorithm to find the gcd of two given integers, and to express it as a linear combination. It is described as follows:
Given two integers x and y (without loss of generality we may assume $0 < x < y$), apply the division algorithm successively as follows:

$y = xq_0 + r_0$ (divide y by x and get $0 \le r_0 < x$)
$x = r_0 q_1 + r_1$ (divide x by r_0 and get $0 \le r_1 < r_0$)
$r_0 = r_1 q_2 + r_2$ (divide r_0 by r_1 and get $0 \le r_2 < r_1$)
$\dots$

Continue in this manner each time dividing the divisor from the previous step with the remainder from the previous step, until the remainder is 0. This process must stop at some finite step because the remainders are getting strictly smaller at every step, so the process cannot continue indefinitely. Therefore, we are going to stop at some step, say $r_k = r_{k+1} q_{k+2}$ where the remainder is 0. The previous remainder r_{k+1} is the gcd. By back-substitutions the gcd can be expressed as a linear combination of the integers x and $y.$ We illustrate this algorithms with an example.

Example: Let $x = 28$ and $y = 48.$ Applying the division algorithm successively until the remainder is 0, we get

$48 = 28\cdot 1+20$
$28 = 20\cdot 1+8$
$20 = 8\cdot 2+4$
$8 = 4\cdot 2+0$

Therefore, (28,48) = 4. Next, the following substitutions let us express the gcd as a linear combination of 28 and 48.

$4 = 20+(-2)\cdot 8 = 20+(-2)\cdot(28+(-1)\cdot 20) = 3\cdot 48+(-5)\cdot 28$

Problem 8: Show that if $a \mid bc$ and $(a,b)=1$, then $a \mid c$.
(Hint: Use the fact that 1 can be written as a linear combination of a and b)

What is the complexity of the Euclidean algorithm? The good news is that it is an efficient algorithm, i.e., the worst case running time is a polynomial in the size of the input (number of digits needed to represent the integers x and y). To determine the complexity of the Euclidean algorithm, we need to determine the number of steps it runs in the worst case. Since at every step we are performing the division algorithm, we can multiply the cost of division with the number of steps to obtain the total worst-case running time. It is proven that the number of steps in the Euclidean algorithm is at most $O(\log(x))$ where x is the smaller input number to the algorithm [4]. The worst-case running time of each step (which is a division) is $O(\log(x)\log(y))$. Altogether, the algorithm does not take more than $O(\log^3(y))$. This is a polynomial in the size of the input, hence it is an efficient algorithm. Through a more careful analysis that takes into account the fact that the successive divisions in the algorithm involve smaller and smaller numbers, it is possible to prove that the running time of the Euclidean algorithm is in fact $O(\log(x)\log(y))$ [4]. This, however, does not change the fact that it is a polynomial time algorithm. It is crucial that the Euclidean algorithm is a polynomial time algorithm for the implementation of the RSA cryptosystem.

Problem 9: Implement a version of the Euclidean algorithm in your favorite programming language or computer algebra system. (Most computer algebra systems have this algorithm implemented. Your goal is to do it for yourself).

Integers Modulo n

Modular arithmetic is a fundamental tool for the RSA cryptosystem. Everyone is familiar with modular arithmetic: the clock arithmetic is an example of modular arithmetic with modulo 12. For example, 5 hours after 9 am is 2 pm, because 9 + 5 is 2 modulo 12. In this system 12 is the same as 0. One can do this kind of arithmetic with an arbitrary modulus n, for any positive integer n. The set of integers $\bmod n$ is denoted by $\mathbb{Z}_n$ and it contains the elements $\{0,1,2,\ldots,n-1\}$ (n is the same as 0). Addition and multiplication are done in the usual way, except that the result is reduced $\bmod n$. For an integer x, $x \bmod n$ means the remainder when x is divided by n. For example, $56 \bmod 9$ is 2. So, $7 \cdot 8 \bmod 9 = 2$, $7+8 \bmod 9 = 6$, and $6^2 \bmod 9 = 0$. We also use the notation $a \equiv b \bmod n$ (read a is congruent to b $\bmod n$) to mean a and b has the same remainder when divided by n. For example, $12 \equiv 3 \bmod 9$ and $12 \equiv 21 \bmod 9$.

$\mathbb{Z}_n$ with addition and multiplication $\bmod n$ always satisfies axioms 1-8 on page 8. Therefore, it is always a ring (more accurately, a commutative ring with unity). However, the last axiom does not always hold true. For example, in $\mathbb{Z}_{10}$ we have $2 \cdot 5 = 0$, but neither 2 nor 5 is 0, so they are zero divisors. Therefore, $\mathbb{Z}_n$ is not always an integral domain. Notice that 2 and 5 do not have multiplicative inverses. In fact, zero divisors never have multiplicative inverses.

Problem 10: Show that a zero divisor in $\mathbb{Z}_n$ does not have a multiplicative inverse.

Some of the non-zero elements of $\mathbb{Z}_{10}$ though do have multiplicative inverses (0 never has a multiplicative inverse, so when we talk about multiplicative inverse we are talking about non-zero elements). For example, $1^{-1} = 1,$ 2 does not have one, $3^{-1} = 7,$ 4,5 and 6 do not have inverses, $7^{-1} = 3,$ 8 does not have one, and $9^{-1} = 9.$ Do you see any pattern? What property do the invertible elements have in common, and what property do non-invertible elements have in common? We have the following complete characterization of the invertible elements in $\mathbb{Z}_n$.

Theorem: A non-zero element a in $\mathbb{Z}_n$ is invertible if and only if $(a,n) = 1.$
Proof: Since this is an "if and only if" statement, we have two directions to prove. In the forward direction we assume that a is invertible, and try to show $(a,n) = 1$. Since a is invertible in $\mathbb{Z}_n$, there exists b in $\mathbb{Z}_n$ such that $ab \equiv 1 \bmod n.$ This means that $ab = nt + 1$ for some integer $t,$ or $ab + n(-t) = 1.$ This equation implies that $(a,n) = 1$ because any common divisor of a and n must also divide their linear combination $ab + n(-t)$ which is 1. Hence a and n cannot have any divisors larger than one. This proves the forward direction.
For the reverse direction, we assume that $(a,n) = 1$ and try to show a is invertible in $\mathbb{Z}_n$. Since $(a,n) = 1$, there exist $x, y \in \mathbb{Z}$ such that $ax + ny = 1$. Reducing this equation $\bmod n$ we obtain $ax \equiv 1 \bmod n$. Hence a is invertible in $\mathbb{Z}_n$.

Euler's Phi Function: From the previous theorem, we know exactly which elements in $\mathbb{Z}_n$ have multiplicative inverses. Is it possible to determine the number of invertible elements in $\mathbb{Z}_n$ for an arbitrary n without having to list them all? It turns out that the answer is yes. There is a well known function that denotes this quantity. It is called *Euler's phi function* and is denoted by $\varphi(n)$ (this function is also called *totient* of n). So,
$\varphi(n) = \#\{x \in \mathbb{Z}_n : x^{-1}$ exists in $\mathbb{Z}_n\},$ number of elements between 1 and $n-1$ that are relatively prime to n (hence invertible in $\mathbb{Z}_n$). For example, $\varphi(9) = 6,$ and $\varphi(10) = 4.$

Problem 11: Find $\varphi(11).$ What is $\varphi(p)$ for a prime number p?

The set of invertible elements in $\mathbb{Z}_n$ is denoted by $\mathbb{Z}_n^*$. Therefore, $\varphi(n)$ is the size of $\mathbb{Z}_n^*$. For example, $\mathbb{Z}_9^* = \{1,2,4,5,7,8\},$ and $\mathbb{Z}_{10}^* = \{1,3,5,7\}.$ The set $\mathbb{Z}_n^*$ is closed under multiplication. That is, if $a,b \in \mathbb{Z}^*_n$ then so is $ab.$ This is not hard to see: if a and b are both relatively prime with n then so is $ab.$ Because, this means a and n do not share any common prime factors, neither do b and n. Then, the product ab does not contain any prime factor common with n. (Here we are using the fact that if $d = (a,n) > 1$ there is a prime dividing d). In the language of abstract algebra, the set $\mathbb{Z}_n^*$ is a *group* with the operation of multiplication ($\bmod n$). It satisfies the following axioms:

1. Multiplication $\bmod n$ is associative: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a,b,c \in \mathbb{Z}_n$
2. There is a multiplicative identity, namely 1: $a \cdot 1 = 1 \cdot a = a$ for all $a \in \mathbb{Z}_n$
3. Multiplication $\bmod n$ is commutative: $a \cdot b = b \cdot a$ for all $a,b \in \mathbb{Z}_n$

4. Every element $a \in \mathbb{Z}_n$ has a multiplicative inverse denoted by a^{-1}: $a \cdot a^{-1} = 1$

Note that these four axioms are the same as the first four axioms on page 8, except that they were for the operation of addition for a ring. Therefore, integers $\bmod n$ $\mathbb{Z}_n$ is a group with the operation of addition. Sometimes we write this as $(\mathbb{Z}_n, +)$ is a group. A ring has two operations, addition and multiplication. A group has only one (which could be called either addition, or multiplication, or even something else). We also write $(\mathbb{Z}_n, +, \cdot)$ is a ring and $(\mathbb{Z}^*_n, \cdot)$ is a group.

Taking exponents $\bmod n$ is an extremely important operation in RSA cryptosystem. Now, we will investigate this operation and its properties. To start off, let us compute powers of 2 $\bmod 9$: $2^1 \equiv 2,\ 2^2 \equiv 4,\ 2^3 \equiv 8 \equiv -1,\ 2^4 \equiv -2 \equiv 7,\ 2^5 \equiv -4 \equiv 5,\ 2^6 \equiv -8 \equiv 1$. We say that the *order* of 2 is 6 because 6 is the smallest positive integer r such that $2^r = 1 \bmod 9$. Also note that powers of 2 give all the elements in this group. We say that 2 is the *generator* of the group $(\mathbb{Z}^*_9, \cdot)$ and we call the group $(\mathbb{Z}^*_9, \cdot)$ a *cyclic group*.

<u>Problem 12:</u> Find the orders of other elements in $(\mathbb{Z}^*_9, \cdot)$. Does $(\mathbb{Z}^*_9, \cdot)$ have any other generators?

From the result of the previous exercise we see that $a^6 = 1$, for all $a \in \mathbb{Z}^*_9$. We also recall that $\varphi(9) = 6$, i.e., $a^{\varphi(9)} = 1 \bmod 9$ for all $a \in \mathbb{Z}^*_9$. It turns out that this is not an accident, rather it is a special case of a famous theorem in elementary number theory, called Fermat's little theorem (more accurately, it is called Euler's generalization of Fermat's little theorem). We state it without a proof since the proof uses some group theory that is beyond the scope of this module).

Theorem: $a^{\varphi(n)} = 1 \bmod n$ for all $a \in \mathbb{Z}^*_n$. Stated slightly more generally, the theorem says that for an integer a that is relatively prime with n, we have $a^{\varphi(n)} = 1 \bmod n$.

When the integer n happens to be a prime in the above theorem, we obtain the special case that is called Fermat's little theorem.

Fermat's Little Theorem: For a prime number p not dividing a, $a^{p-1} = 1 \bmod p$.

Recall that for a prime number p, $\varphi(p) = p - 1$, so this is indeed a special case of the previous theorem. This theorem does have an elementary proof.

<u>Problem 13</u>: Prove the Fermat's little theorem following the hint below:

a) Show that the elements of the set $S = \{a, 2a, ..., (p-1)a\} \subseteq \mathbb{Z}_p$ are non-zero, and there are no repetitions.
b) Part a) shows that $S = \mathbb{Z}^*_p$. Therefore, $1 \cdot 2 \cdots (p-1) \equiv a \cdot (2a) \cdots (p-1)a \bmod p$. Explain how this proves the theorem.

In addition to $\varphi(p) = p-1$ for a prime p, the Euler's phi function has two more properties that make it possible to compute $\varphi(n)$ for any positive integer n.

Theorem:

a) For a prime p, and a positive integer r, $\varphi(p^r) = p^r - p^{r-1} = p^{r-1}(p-1)$

b) If $(m,n)=1$ then $\varphi(mn) = \varphi(m)\varphi(n)$. We say that Euler's phi function is multiplicative.

Part a) can be proven by an elementary counting argument, but part b) is harder to prove.

Problem 14: Prove part a) of the above theorem.

Given the last theorem and the fundamental theorem of arithmetic, we can now compute $\varphi(n)$ for any positive integer n. First, obtain the prime factorization of n, say $n = p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$. Then, $\varphi(n) = \varphi(p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}) = \varphi(p_1^{r_1}) \cdots \varphi(p_k^{r_k})$, and the each term in the last product can be computed using part a) of the above theorem. For example,

$\varphi(10) = \varphi(2\cdot 5) = \varphi(2)\cdot\varphi(5) = (2-1)(5-1) = 4$, and

$\varphi(4725) = \varphi(3^3 \cdot 5^2 \cdot 7) = \varphi(3^3)\cdot\varphi(5^2)\cdot\varphi(7) = (3^3 - 3^2)\cdot(5^2-5)\cdot(7-1) = 2160.$

Modular Exponentiation

Securely implementing the RSA cryptosystem requires modular exponentiation with extremely large bases, modulus and exponents, i.e., computing numbers of the form $a^e \bmod n$ where $a, e,$ and n are very large numbers. On the one hand these numbers need to be large for security reasons, on the other hand it should be easy (efficient) to compute such exponents for the practicality of the system. We now discuss how expensive it is to compute an expression of the form $a^e \bmod n$. Of course a^e means $a \cdot a \cdots a$ where a is multiplied with itself $e-1$ times. As discussed earlier, multiplication of two k-digit numbers takes O(k^2) bit operations (where $k = \log_2(n)$). Since we do this e times, the total cost will be O(ek^2). Is this a polynomial in the size of the input? Unfortunately, it isn't. Since the input size for the number e is $\log_2(e)$, the term e is exponential, not polynomial, in the size of the input: $e = 2^{\log_2(e)}$. Therefore, the straightforward method of multiplication is not an efficient way for modular exponentiation with large numbers.

Fortunately, there is a more clever way of computing such exponents. It is called *square and multiply* method. The idea for this method comes from that fact that if we use the binary representation of the exponent e we may reduce the number of operations significantly. Let $e = \sum_{i=0}^{t} e_i 2^i$ be the binary expansion of e where $t = \lfloor \log_2(e) \rfloor$ and each e_i is a 0 or 1. Then,

$$a^e = a^{\sum_{i=0}^{t} e_i 2^i} = \prod_{i=0}^{t} a^{e_i 2^i} = \prod_{i=0}^{t} (a^{2^i})^{e_i} = \prod_{0 \le i \le t, e_i = 1} a^{2^i}$$

Since $a^{2^i} = (a^{2^{i-1}})^2$, this product can be calculated using at most t modular squarings and t modular multiplications. Therefore, the following algorithm computes $a^e \bmod n$.

Square and Multiply Algorithm:

Input: $0 \neq a \in \mathbb{Z}_n$, and integer $0 \leq e < n$ with binary representation $e = \sum_{i=0}^{t} e_i 2^i$

Output: $a^e \bmod n$

```
1. Set $P \leftarrow 1$, and s $\leftarrow$ a
2. For $i$ =0 to t do
      2.1 If $i > 0$, set $s \leftarrow s^2 \bmod n$
      2.2 If $e_i = 1$, set $P \leftarrow P \cdot s \bmod n$
3. Return $P$.
```

As an example, let us compute $4^{45} \bmod 55$ using this algorithm. First, the binary representation of 45 is 101101. Next, compute successive squares of 4 mod 55:

$$4^2 = 16,\ 4^{2^2} = 16^2 = 36,\ 4^{2^3} = 36^2 = 31,\ 4^{2^4} = 31^2 = 26,\ 4^{2^5} = 26^2 = 16.$$

Finally, multiply together those powers that correspond to a 1 in the binary representation of 45:

$$4^{45} \bmod 55 \equiv 4^{2^5} \cdot 4^{2^3} 4^{2^2} 4^{2^0} \equiv 16 \cdot 31 \cdot 36 \cdot 4 \bmod 55 \equiv 34$$

Notice that we did 5 squarings mod 55 and 3 multiplications mod 55. The naïve method would have performed 44 multiplications. Also, reducing results mod n at every step of the computation, as opposed to reducing the final result at the end, helps us save memory considerably. In fact, memory overflow is likely with large numbers if we do not reduce the results at every step because exponential functions grow very fast. The computations in this example could be organized in a tabular form as follows:

i	0	1	2	3	4	5
e_i	1	0	1	1	0	1
s	4	$4^2 \equiv 16$	$16^2 \equiv 36$	$36^2 \equiv 31$	$31^2 \equiv 26$	$26^2 \equiv 16$
P	4	4	$36 \cdot 4 \equiv 34$	$31 \cdot 34 \equiv 9$	9	$16 \cdot 9 \equiv 34$

Now, let us formally determine the complexity of the square and multiply algorithm. As we already observed, there are at most t squaring and t multiplications. Since each of these operations take at most O(t^2) bit-operations, the total cost will be proportional to t^3, hence the total cost is at most O(t^3), a polynomial in the size of the input $t = \lfloor \log_2(e) \rfloor$. Therefore, this algorithm is indeed an efficient algorithm. The difference between the two methods in number of operations needed gets more dramatic as the exponent increases. For example, if $e = 10^7$, the straightforward method requires ten million multiplications (more accurately $10^7 - 1$), whereas the efficient method requires at most $2 \cdot \lfloor \log_2(10^7) \rfloor = 46$ squaring or multiplications.

Many computer algebra systems have both methods implemented for modular exponentiation. To see the difference we execute examples to compute $a^e \bmod n$ for various values of the parameters with each method in the computer algebra systems Maple and Magma. Magma has an online calculator that is free to use for computations up to 20 seconds The web address of the Magma calculator is http://magma.maths.usyd.edu.au/calc/. Try the following on Maple and Magma and see the results for yourself. Below is what I obtained.

Modular Exponentiation in Maple:

The naïve and inefficient way of computing $a^e \bmod n$ in Maple has the same syntax. We enter the values and time the operation as follows:

```
> a := 3567; e := 24556677888111; n := 10^9 + 27;
> a^e mod n;
Error, numeric exception: overflow
```

Here, the exponent is so big that the computation causes a memory overflow, meaning the numbers are get so large that they cannot be stored in the memory. However, the efficient square and multiply method easily finishes the execution. The Maple command for this method is Power(a,e) mod n.

```
> Power(a, e) mod n;
                    371667798
```

Now, let's try examples where both methods can finish and compare the times they take. Study the Maple code below to see how to time a computation.

```
> e := 10^7 + 5·10^6 + 50067;
> st := time( ); a^e mod n; time( ) − st;
st := 34.796
111182625
11.313
```

Here, we used the same values of the parameters except that we changed the value of e that caused problems. The result shows that this computation took about 11.31 seconds of CPU time (and the result of the computation is 111182625).
Now, the same computation using the Power function:

```
> st := time( ); Power(a, e) mod n; time( ) − st;
st := 34.671
111182625
0.
```

The same result but the time it took shows 0, which probably means it took so little time that the time( ) function's precision is not big enough to capture a non-zero value.

Modular Exponentiation in Magma

Now, we perform the exact same computations in Magma. Magma's command to do the naïve exponentiation is a^e mod n; and the efficient one is by Modexp(a,e,n); The online magma calculator automatically gives the amount of time the computation takes. If you happen to have Magma installed on a computer you can access, to time a computation simply put time in front of the computation, like

```
        > time a^e mod n;
```

The input

```
        a:=3567;  e:=24556677888111;  n:=10^9+27;
        a^e mod n;
```

gives the output

```
        >> a^e mod n;
        Runtime error in '^': Argument 2 (24556677888111) is too large
        Total time: 0.140 seconds, Total memory usage: 7.19MB
```

Meaning the numbers get too large to be handled by the program.
The Modexp command on the other hand has no trouble computing this quantity. The command

```
        Modexp(a,e,n);
```

returns

```
        371667798
        Total time: 0.140 seconds, Total memory usage: 7.19MB
```

The same answer as Maple, and it took 0.14 seconds. (Magma also tells you how much memory is used). The input

```
        a:=3567;  e:=10^7+5*10^6+50067; n:=10^9+27;
        a^e mod n;
```

produces the output

```
        111182625
        Total time: 4.059 seconds, Total memory usage: 137.02MB
```

The same answer as Maple, produced in 4.059 seconds.
Finally, the same exponential with the efficient method is as follows:

```
Input:    Modexp(a,e,n); (with the above values of a, e, n)
Output:  111182625
          Total time: 0.140 seconds, Total memory usage: 7.19MB
```

This time it only took 0.14 seconds.

Problem 15:
Implement the square and multiply algorithm in your favorite programming language. Be sure to test it.

Public Key Cryptography and the RSA Cryptosystem

A major drawback of symmetric key cryptosystems is that the keys must be exchanged first between the two parties over a secure channel before they can start using an encryption scheme. This is often not practical. The problem becomes even more difficult if many different parties want to communicate securely, as is the case with online transactions that are so common today. The idea of public key cryptography is first introduced by W. Diffie and M. E. Hellman in their seminal paper [5]. It eliminates the need for advance key exchange. In a public key cryptosystem only the decryption key needs to be kept secret. The encryption key is made public. Unlike symmetric key cryptosystems, it is (should be) infeasible to obtain the decryption key (the private key) from the encryption key (public key). Another requirement is that although the public key is not secret, its authenticity needs to be protected. An adversary should not be able to alter or replace the publicly announced encryption key, otherwise she would be able to decrypt messages not intended for her.

If Alice and Bob want to communicate securely using a public key cryptosystem, one of them, say Bob, generates a pair of keys (e,d) where e is the public key, and d is the private key. Bob then announces the public portion of the key (e) (for example, he publishes it on his web site). Alice then obtains Bob's public key and uses it to encrypt her message to Bob, i.e., she locks her box with that key. Even if Eve intercept Alice's encrypted message she won't be able to decrypt it because she does not have access to Bob's private key, neither can she determine the private key from the public key. Upon receiving Alice's encrypted message, Bob decrypts it (unlocks the box) using his private key.

RSA Cryptosystem

Shortly after the seminal paper on public key cryptography [5] which introduced the idea but did not give any specific, practical implementation, the RSA cryptosystem was proposed by Rivest, Shamir, and Adleman (hence the acronym) in [12]. It is still a widely used cryptosystem. It uses most of the facts from elementary number theory reviewed in this module. First, messages are converted to numbers (integers), then the numbers are manipulated according to the prescribed encryption scheme. Here is the description of the RSA cryptosystem.

The RSA Algorithm
1. Bob chooses two large distinct primes p and q, and computes $n = pq$.
2. Bob chooses a random integer e such that $(e, \varphi(n)) = 1$.
3. Bob finds d such that $ed \equiv 1 \bmod \varphi(n)$.
4. Bob makes n and e public, and keeps p, q, d private.
5. Alice obtains public information revealed by Bob.
6. To encrypt her message m, Alice computes $c = m^e \bmod n$ and sends c to Bob.
7. Bob decrypts c (i.e. obtains m) by computing $c^d \bmod n$.

In this set up, the integer n is called the *RSA modulus*, e is called the *encryption exponent* and d is called the *decryption exponent.*

Let us now analyze this system and justify that it works and it is a feasible and secure system. First, note that all computations are done $\bmod n$, and a message to be encrypted is converted to a number $m < n$. (If an original message is larger than what can fit into single number $\bmod n$, it is divided into smaller blocks.) Since n is a product of two distinct primes, $\varphi(n) = \varphi(pq) = \varphi(p)\varphi(q) = (p-1)(q-1)$. Also, since $(e, \varphi(n)) = 1$, e is invertible $\bmod \varphi(n)$, hence the number d in step 3 exits (and is unique). Both m and c and are numbers $\bmod n$, so the ciphertext does not necessarily get bigger than the plaintext (it may actually get smaller). It is just converted to another number $\bmod n$. A final and important step to justify the algorithm is to show that decryption works, i.e., $c^d \equiv m \bmod n$, equivalently, $m^{ed} \equiv m \bmod n$.

It turns out that we need to consider two cases to prove this fact: $(m,n)=1$ and $(m,n)\neq 1$.
Let us suppose that $(m,n)=1$. Then by Fermat's little theorem we have $m^{\varphi(n)} \equiv 1 \bmod n$. On the other hand, since $ed \equiv 1 \bmod \varphi(n)$, there exists $s \in \mathbb{Z}$ such that $ed = 1 + s\cdot\varphi(n)$. Putting all this together we obtain: $m^{ed} \equiv m^{1+s\varphi(n)} \equiv m\cdot(m^{\varphi(n)})^s \equiv m\cdot(1)^s \equiv m \bmod n.$
The same result can also be obtained in the case $(m,n)\neq 1$. with some more work. However, let us observe that for large prime primes p and q, it is much more likely that $(m,n)=1$ rather than $(m,n)>1$ for an arbitrary number $m<n$. Therefore, we skip the proof for that case.

Problem 16: Give a justification for the last claim, i.e., show that for large primes p and q, an arbitrary integer m mod n, $n=pq$, is much more likely to be relatively prime with n than not.

Example:
To set up an RSA encryption system to communicate with Alice securely (for that matter with anybody else who might be interested) Bob chooses the following primes:
$p=$ 9776698225631087 and $q=$ 33267710468717971
He then computes
$n = pq = 325248365910323724697873293164477$
and $\varphi(n) = (p-1)(q-1) = 325248365910323681653464598815420$. Next, he chooses the encryption exponent $e = 173830646339602015143190932502207$ and computes the corresponding decryption exponent $d = e^{-1} \bmod n = 185024181617369988056414344042783$.
Finally, he publishes the values of e and n, on his web site, and keeps the rest of the values private.

Alice downloads those values from Bob's web site, and she has an important message to communicate to Bob. She wants Bob to be the first person to hear the big news that Fermat's last theorem has just been proven (by Alice). She does not want to make this exciting information public yet before she is absolutely sure. So she wants to send the message encrypted. Here is what she does:

Her message in text form is "Big news: Fermat's last theorem is proven!". First she converts her message into a numerical form. Alice and Bob use Maple and its method of converting messages back and forth between text and numerical forms. (This method is explained next when we show the details of the implementation using Maple). Since her message in numerical form occupies more than 33 digits, she breaks it up into blocks of length 33 or less. She gets 4 blocks of numerical data as follows:
$m_1 = 712134940784389781103961729821 70$, $m_2 =$ 151614105658550004853057724619273
$m_3 =$ 998908611304870242878718395666 28, and $m_4 =$ 531.
She then encrypts each piece of her message using Bob's public encryption key and obtains:

$c_1 = m_1^e \bmod n =$ 274550104966733060872501653048215

$c_2 = m_2^e \bmod n =$ 115962549954362347388038785672652

$c_3 = m_3^e \bmod n =$ 163931895700108966814292922877385

$c_4 = m_4^e \bmod n =$ 121158951233957499726842760892188

Upon receiving these four pieces of ciphertext, Bob decrypts them using his private key:

$m_1 = c_1^d \bmod n = 712134940784389781103961729821 70$

$m_2 = c_2^d \bmod n = 151614105658550004853057724619273$

$m_3 = c_3^d \bmod n = 998908611304870242878718395666 28$

$m_4 = c_4^d \bmod n = 531$

We see that he correctly obtains the plaintext in numerical form. Finally, he converts it to the corresponding text message to read the secret message: "Big news: Fermat's last theorem is proven!"

Remarks and Issues on RSA Cryptosystem

Before we show the details of how to implement the RSA system on Maple (with the above example) there are several important questions we need to ask about its practicality and security. Notice that the very first task Bob needs to take care of is generating "large" primes. Earlier we learned that there are infinitely many primes. So, there is a sufficient supply of those. Still, from the implementation point of view we need a practical (efficient) way of finding them. The prime number theorem (PNT) says that they are relatively dense among the integers. The theorem was first proven by Hadamard [8], but many improvements, refinements and simplifications followed over the next century. See [7] for an excellent survey on the subject. Roughly speaking PNT says that if a number m is randomly selected in the order of N, then the probability of m being a prime is about $1/\ln(N)$. For example, around 10^6 approximately one out of every 14 numbers is prime. Therefore, one can generate large random numbers, and check whether they are in fact primes. PNT ensures that before too long we should hit a prime. But how easy or expensive is it to check whether a given number is actually a prime? There are many fast probabilistic algorithms that decide with any prescribed level of accuracy. See [11, chapter 4] on prime number generation and primality tests. In 2002, a deterministic polynomial time algorithm was found to test whether a given number is a prime [1]. This was a major breakthrough mostly for theoretical reasons. The algorithm was discovered by two undergraduates. A nice survey article on the subject is [3]. So, the prime number generation is completely solved for both theoretical and practical purposes.

Problem 17: Given an integer n, a simple method to test whether it is a prime is to check all integers between 2 and n-1 to see if they are divisors of n. As soon as you find a divisor you can stop and declare the number to be composite (non-prime). Show that this is not an efficient algorithm, i.e. show that the number of steps is exponential in the size of the input. An observation that improves the running time of this algorithm is it suffices to check the numbers up to $\sqrt{n}$ (rather n-1) since if n has a nontrivial factor at all, it must have one that is less than or equal to $\sqrt{n}$ (since factors come in pairs). Show that, this improved algorithm is not efficient either. Finally, implement this algorithm in your favorite programming language.

The next question is, how large should those primes be? There is no fixed answer to this question, the answer has been changing by the advances in integer factorization algorithms, and the computing resources. Currently, 512-bit primes (about 155 digits for primes p and q) are recommended for long term security. See [11, page 290] and [9, page 285] for more detailed recommendations. There has been a series of RSA challenges to factor some proposed large integers starting with the original authors (see [14], page 154). Although no longer active, the RSA challenge web site gives a history of the challenge for those who are interested. It is located at http://www.rsa.com/rsalabs/node.asp?id=2093 and http://www.rsa.com/rsalabs/node.asp?id=2094

Next, Bob computes $n = pq$ and $\varphi(n) = (p-1)(q-1)$. Multiplication of two integers, even if they are large, is computationally efficient (polynomial time in the size of the input), so no problems computing those. He also needs to find an integer e that is relatively prime with $\varphi(n)$. He can do so by choosing a random integer e and checking whether it is relatively prime with $\varphi(n)$. If not, he repeats the process. There are efficient methods for random number generation. How about finding gcd of two numbers? Recall that the Euclidean algorithm is an efficient algorithm to find gcd of two numbers, so that is practical as well. Then, Bob needs to compute the inverse e of mod $\varphi(n)$. Can this be done efficiently? Again recall that the gcd of e and $\varphi(n)$ can be written as a linear combination in the form $ex + \varphi(n)y = 1$ and the Euclidean algorithm tells us how to obtain the numbers x and y. If we reduce that equation mod $\varphi(n)$ we obtain $ex \equiv 1 \bmod \varphi(n)$, hence $x \bmod \varphi(n)$ the inverse of e. So, everything Bob needs to compute so far is efficient.

The main computation Alice needs to do is exponentiation mod n. We have seen that this can be done efficiently by the square and multiply method, so she is okay. Finally, to decrypt a ciphertext Bob computes an exponential mod n, again that can be done efficiently.

What about Eve? If she intercepts a message (ciphertext) what can she do to decrypt it? She needs d to decrypt. If she could factor n, then she obtains p and q, and calculates $\varphi(n)$. She can then compute d efficiently from e (remember e is public). The problem she faces is factorization of large integers is believed to be a hard problem: No polynomial time algorithm is known for that problem despite much effort to find one (though that does not mean that one can never be found). However, the prime factors p and q making up n need to be large enough to ensure security against currently best known attacks and algorithms. Again, 512- bit primes with certain properties are currently considered to be safe.

An Implementation of RSA on Maple

We now show step by step details of how to implement RSA cryptosystem on Maple using the previous example. First, generate two random numbers in the order of 10^{16} and 10^{17} (this is much smaller than what should be used in practice. This is just a toy example to illustrate the method. A separate Maple file containing all the Maple code in this module is attached.)

```
> N := rand(10^15..10^16 -1)( ) : M := rand(10^16..10^17 -1)( ) :
```

Then find the next prime after each one of these random numbers.

```
> p := nextprime(N); q := nextprime(M);
              p := 9776698225631087, q := 33267710468717971
```

To double check whether they are indeed primes, do

```
> isprime(p); isprime(q);
```

```
true

true
```

Compute n and $\varphi(n)$.

```
> n := p·q; ϕ := (p − 1)·(q − 1);

n := 32524836591032372469787329316477

ϕ := 32524836591032368165346459881520
```

Let’s see how many digits they have

```
> length(n); length(phi);
33

33
```

Generate a random integer that is relatively prime with $\varphi(n)$. This may take several trials.

```
> e := rand(0..n − 1)( );
e := 17383064633960201514319093250207

> gcd(e, phi);
1
```

Compute the inverse of e, and double check it

```
> d := 1/e mod phi;
d := 18502418161736998805641434404283

> e·d mod phi;
1
```

Define the plaintext as a text.

```
> plaintext := "Big news: Fermat's last theorem is proven!";
```

Then convert it to numerical data. Here the method we use is to convert each character (including space and punctuation marks) to the corresponding integer. Every character has a numerical representation in the computer memory in binary (called ASCII code). If desired, that binary number can be converted to the corresponding integer. For example, the ASCII code for the letter “A” is 1000001 which can be converted to the integer 65. Maple’s convert command gives the numerical value for each letter. For example,

```
> convert("A", bytes);
[65]
```

It also does the reverse conversion such as

```
> convert([65, 66, 67], 'bytes')
"ABC"
```

The following command converts each character in the plaintext to the corresponding integer. Each of these digits is considered to be an integer mod 256 by Maple.

```
> plaintextNumber := convert(plaintext, bytes);
```

```
        plaintextNumber := [66, 105, 103, 32, 110, 101, 119, 115, 58, 32, 70, 101,
            114, 109, 97, 116, 39, 115, 32, 108, 97, 115, 116, 32, 116, 104, 101,
            111, 114, 101, 109, 32, 105, 115, 32, 112, 114, 111, 118, 101, 110, 33]
```

Next, we need to break this list into groups so that each group is a number mod *n* (hence not more than 33 digits long). The following Maple command does this nicely:

```
>  m := convert(plaintextNumber, base, 256, n);
        m := [712134940784389781103961729821,70
            151614105658550004853057724619,273
            998908611304870242878718395666,25831]
```

Now, we want to apply the encryption function (exponentiation mod n with exponent e) to each block of the ciphertext. To do this we first define the function, then use Maple's map command to let the function apply on each component of the input.

```
>  f := (x, e, n) → Power(x, e) mod n;
        f := (x, e, n) → Power(x, e)  mod  n

>  cipherText := map(f, m, e, n);
        cipherText := [274550104966733060872501653048,215
            115962549954362347388038785672,652
            163931895700108966814292922877,385
            1211589512339574997268427608921]88
```

This is the ciphertext with each component being a number mod n. Out of curiosity you may wonder what text it corresponds to. Let's find out. This will be done in two steps. First convert each number to a sequence of base 256 numbers, then find the corresponding characters.

```
>  WrongDecipherMod256 := convert(cipherText, base, n, 256);
 WrongDecipherMod256 := [208, 3, 214, 158, 236, 216, 232, 185, 52,
    231, 118, 136, 106, 50, 154, 191, 164, 147, 39, 24, 44, 250, 170, 245,
    123, 54, 148, 148, 183, 214, 220, 211, 67, 38, 252, 48, 76, 194, 129,
    148, 233, 251, 11, 247, 150, 62, 130, 71, 43, 107, 1, 78, 57, 96]

>  WrongDecipherText := convert(WrongDecipherMod256 , bytes);
 WrongDecipherText :=
    "Ð□ÖžìØè¹4çv^j2š¿¤“'□,úªõ{6”"•ÖÜÓC&ü0LÂ□”éû□
    ÷−>,G+k□N9"
```

As you see, the ciphertext is not anything intelligible. To decrypt it correctly, we need to use the decryption key first then apply the previous two commands.

```
>  decipherModn := map(f, cipherText, d, n);
 decipherModn := [712134940784389781103961729821,70
    151614105658550004853057724619,273
    998908611304870242878718395666,25831]
```

```
> decipherMod256 := convert(decipherModn, base, n, 256);
decipherMod256 := [66, 105, 103, 32, 110, 101, 119, 115, 58, 32, 70,
    101, 114, 109, 97, 116, 39, 115, 32, 108, 97, 115, 116, 32, 116, 104,
    101, 111, 114, 101, 109, 32, 105, 115, 32, 112, 114, 111, 118, 101,
    110, 33]

> decipherText := convert(decipherMod256, bytes);
decipherText := "Big news: Fermat's last theorem is proven!
```

It works!

One-way Functions: A fundamental notion in public key cryptography is that of a *one-way trapdoor function.* A function $f : X \to Y$ is called a one-way function if $f(x)$ is easy to compute for all $x \in X$, and for all (or most) $y \in Y$, it is computationally infeasible to find $x \in X$ such that $f(x) = y$. A one-way function f is called *trapdoor* if given some extra information it becomes feasible to find $x \in X$ such that $f(x) = y$. Multiplication of large integers is believed to be a one-way function (when the number is large and does not contain small prime factors), and that assumption is at the heart of the security of the RSA cryptosystem. We saw that multiplying two integers is easy, i.e. there exists an efficient algorithm to do it, however no efficient algorithm is known for factoring a given (large) integer. One can also ask whether it is possible to decrypt messages without having to factor n in RSA. Clearly, it suffices to compute $\varphi(n)$ to break the system. The authors of RSA claim that determining $\varphi(n)$ is no easier than factoring n [12]. Yet another question is whether computing e-th roots mod n can be done efficiently. If yes, then it would be possible to break RSA without having to find $\varphi(n)$. However, no efficient algorithms are known to do that either (except for some special cases). The RSA function $f : \mathbb{Z}_n \to \mathbb{Z}_n$ defined by $f(x) = x^e \bmod n$ is believed to be a trapdoor one-way function. Given an arbitrary element of $\mathbb{Z}_n$, no efficient algorithms are known to find its e-th root in general. With the additional information of d however it becomes easy to do so. Whether computing e-th roots mod n is equivalent to integer factoring is not known either.

Another problem that is believed to be a one-way function is the discrete logarithm problem. There is an encryption scheme called ElGamal that is based on the difficulty of this problem. See [13], [14] more on the discrete log problem and the ElGamal cryptosystem.

Now try factoring some large integers on Maple and Magma to get a sense of how much time it may take with numbers of different magnitudes.

Consider the primes p=1991864036785658356909640066309 and q=207311008982089108727256416579693. They are indeed primes, as the following computation shows. It also shows that it takes practically no time to multiply these two integers, but takes more than 44 seconds to factor them on Maple.

```
> a := rand(10^30..10^31 - 1)( ) : b := rand(10^32..10^33 - 1)( ) :
> p := nextprime(a); q := nextprime(b);
        p := 1991864036785658356909640066309
        q := 207311008982089108727256416579693

>  st := time( ); n := p·q; time( ) − st;
st := 10.00

 n :=
     412935343221171890507263551700539374717408982737905 66
     63137


0.

>  st := time( ); ifactor(n); time( ) − st;
st := 10.00

 (1991864036785658356909640066309)
     (207311008982089108727256416579693)


44.171
```

Now, working with larger numbers we get the following. I had to stop the execution after 940.01 seconds and 39.68 MB of memory.

```
>  a := rand(10^50..10^51 −1)( ) : b := rand(10^52..10^53 −1)( ) :
>  p := nextprime(a); q := nextprime(b);
        p := 834668062591566554168170422306573003701249072 98

        q := 114895618128697575013842061823012720505918011400 8

>  st := time( ); n := p·q; time( ) − st;
        st := 3.15

         n :=
             958997029837404764263187744968710815011978162065949 38
             490637673332225056447494180264857295133877903


0.

>  st := time( ); ifactor(n); time( ) − st;
        st := 3.234

        Warning,  computation interrupted
        936.65
```

The same calculations in Magma (http://magma.maths.usyd.edu.au/calc/) gives the following results. The input

```
p:=1991864036785658356909640066309;
q:=207311008982089108727256416579693;
IsPrime(p);
IsPrime(q);
n:=p*q;
```

produces the output

```
true
true
```

```
        Total time: 0.170 seconds, Total memory usage: 9.57MB
```

Factorization of this integer in Magma however takes more than 20 second, therefore the online calculator stops without giving the answer. The Magma command to factor an integer is Factorization(n). On my computer Magma took 137.63 seconds to finish this factorization with output:

```
        > time Factorization(n);
[ <1991864036785658356909640066309, 1>, <207311008982089108727256416579693, 1> ]
        Time: 137.630
```

Problem 18: Suppose your enemy is exchanging messages using RSA. You learned that their modulus is n =1119782317124294788535000054247349 and encryption exponent is e =10033995567893. You also obtained the following ciphertext [102384236890322070151018468555072,70105509788103331217503087684013]
Decipher this message. What makes you able to decipher this message so easily?

RSA Signature Scheme

In addition to confidential communication, cryptographic protocols can also be used to sign electronic documents. The RSA algorithm can be modified to sign documents. Suppose Bob wants to sign a document m. He chooses numbers p,q,n,e, and d exactly as in the RSA encryption scheme. Again, (e,n) is made public, p,q, and d are kept private. Bob's signature on m is s = m^d mod n. He sends the document along with the signature to Alice. Upon receiving the pair (m,s), Alice verifies the signature by computing s^e mod n and checking that it is equal to m.

Example: Bob chooses p = 1753, q=1907, and computes n= p*q=3342971, $\varphi(n) = (p-1)(q-1) = 3339312$. He also chooses e=15073, and finds d=1231777. So his public key is (n,e)=(3342971,15073) and his private key is d=1231777. He wants to withdraw $120 from an ATM. He signs 120 by s=120^d = 3023701. The machine computes s^{e}mod n = 120, and knows that it is Bob who wants to withdraw $120.

Suppose Eve intercepts the signed message (m,s) and wants to attach Bob's signature on another message y (of her choosing). She cannot use the pair (y,s) as s^e≠y. She needs s$_1$ such that s_1^e = y. However, this is as hard as breaking the RSA system.

Problem 19: Suppose Eve chooses s first and lets the message be m=s^e mod n. Then, it is indeed the case that the signature s on m is verified (satisfies the requirement). This is called *existential forgery*. Explain what the problem is with this idea.

Further Remarks on RSA

Besides choosing large enough primes in setting up the RSA system, there are a number of issues one needs to pay attention to when implementing the system in the real world. Although the RSA system is widely used and is secure when properly implemented, it can be vulnerable to various attacks if some of the implementation aspects are not properly addressed. RSA has been the subject of intensive research and many attacks are devised. One has to be aware of existing attacks and pitfalls. There is an excellent survey article on attacks on RSA system [2]. Boneh states that "although twenty years of research have led to a number of fascinating attacks, none of them is devastating. They mostly illustrate the dangers of improper use of RSA. Indeed, securely implementing RSA is a nontrivial task". Some of the important points for a real world application of RSA are as follows. Messages should be padded with long enough random bits before encryption so that multiple encryptions of the same message will give different ciphertexts (against chosen-ciphertext attacks). It is totally unsafe that different users use the same RSA modulus. Small decryption exponents should not be used. Low encryption exponent is not always safe (not as devastating as low decryption exponent). There are also requirements on the prime factors used. The prime factors used for the RSA modulus should be so called *strong primes* [11]. For more on attacks on RSA and issues to consider for practical implementation see [2],[6],[11],[13],[14].

Possible Projects Related to the RSA Cryptosystem

A Cryptographic Scavenger Hunt

To help students master the RSA cryptosystem and at the same time have them some fun, a "scavenger hunt" type of project similar to the one described in [10] can be designed. Such a project needs to be individually designed for each instructor tailored to his or her students and campus environment. An award for the winning team makes the project more attractive and exciting for the students.

Integer Factoring Algorithms

Since the difficulty of integer factorization is at the heart of the RSA cryptosystem, students can study various integer factorization algorithms and write up a paper describing, summarizing and perhaps implementing some of them. Some methods students can study are: Pollard's rho factoring algorithm, p-1 method, random squares, and quadratic sieve. Some resources on these algorithms are [4], [11],[13],[14].

Prime Number Generation and Primality Tests

Generating large primes and verifying that they are indeed primes is another essential problem for the RSA system. Students can study various algorithms for these tasks. There are deterministic and probabilistic tests for primality. Some topics to consider are: Trial division, Fermat's test, Carmichael numbers, Miller-Rabin test, random generation of primes, Solovay-Strassen test. Some resources on these topics are [4],[6],[11],[13].

Attacks on RSA system

Studying various attacks on the RSA system would give student a better understanding of the system and important points to pay attention to when implementing the system in practice. Some topics to consider in this project would be the size of the encryption exponent, the size of the decryption exponent, forward search attack, timing attack, common modulus attack, cycling attacks, and unconcealed messages. Some resources are [2],[11],[13],[14].

Equivalence of Factorization of n to Determining the Decryption Key

We saw that factorization of the RSA modulus n enables us to determine the private key d. It turns out that the converse of this statement is also true: if you can somehow acquire the

private key d (in addition to public key (n,e)), then you can factor n efficiently. Your project is to find and implement an algorithm to do that. Therefore, the following result is obtained:
Theorem: The problem of computing the RSA decryption exponent d from the public key (n,e) and the problem of factoring n are computationally equivalent.
Show how your method works with the following example: n=176059003, e=357, and d=116368493.

Using RSA in the Classroom
As an immediate application of the RSA system, the students in the classroom decide to communicate with each other using the RSA system implemented in Maple. Each student generates the necessary information to use the RSA system and submits it to the instructor. The instructor keeps the private data and posts the public portion on a web site. Each student must send and receive a prescribed number of messages to and from other students in the class. Then, everyone decrypts the messages received. Finally, the decrypted messages are verified with the sender.

The Best Attacker
The instructor can design another hands-on activity where the class is divided into attacker groups. The instructor posts a number of RSA moduli of various sizes together with public keys and an encrypted message for each RSA modulus. The goal of the groups is to decrypt as many of these messages as possible within the given time. The group with largest number of decrypted messages is the winner. Students should be encouraged to apply attacking strategies from the literature.

REFERENCES

1. M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. Of Math* 160 (2): 781-793, 2004.

2. D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices Amer. Math. Soc.* 46(2), 203-213, 1999. Available at: http://www.ams.org/notices/199902/boneh.pdf

3. F. Bornemann, PRIMES Is in P: A breakthrough for “everyman”, *Notices Amer. Math. Soc*. 50 (5): 545-552, 2003. Available at: http://www.ams.org/notices/200305/fea-bornemann.pdf

4. J. A. Buchmann. *Introduction to Cryptography*, New York: Springer, 2002.

5. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*. 22 (6): 644-654, 1976.

6. N. Ferguson and B. Schneider. *Practical Cryptography*. Wiley Publishing, 2003.

7. D. Goldfeld. *The elementary proof of the prime number theorem: an historical perspective*, available at http://www.math.columbia.edu/~goldfeld/ErdosSelbergDispute.pdf

8. Hadamard, Etude sur les propriet'es des fonctions enti'eres et en particulier d'une function consid'er'ee par Riemann, J. *de Math. Pures Appl.* (4) **9**, 171-215 (1893) ; reprinted in Oeuvres de Jacques Hadamard, C. N. R. S., Paris, vol 1. 103-147, 1968.

9. D. R. Hankerson, D. G. Hoffman, D. A. Leonard D. A, C. C. Lindner, K. Y. Phelps, C. A. Rodger, and J. R. Wall. *Coding Theory and Cryptography, The essentials*, 2nd edition, New York: Marcel Dekker, 2000.

10. J. A. Holdener and E. J. Holdener. A cryptographic scavenger hunt. *Cryptologia*, 31: 316-323, 2007.

11. A.J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. New York: CRC Press, 1997.

12. R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM* (2) **21**, 120-126, 1978.

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

14. W. Trape and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Upper Saddle River NJ: Prentice Hall, 2002.

15. P. Wright. *Spycatcher: The candid autobiography of a senior intelligence officer*. Viking, New York, 1987.