

Mathematical and Algorithmic Background for RSA

The Euclidean Algorithm

The greatest common divisor of two positive integers a and b , $gcd(a,b)$, is the largest integer that divides evenly into both a and b . The algorithm for computing the gcd is based on two properties of the gcd. If a is the larger of the two numbers

$$gcd(a,b) = gcd(a \bmod b, b)$$

$$gcd(a,0) = a$$

The second property is obvious from the definition. You can find a proof of the first property in Theorem 31.9 in the text. These two properties lead to an algorithm for computing the gcd of two positive integers called the *Euclidean Algorithm*.

Euclid(a,b)

1. **if** $b = 0$
2. **then return** a
3. **else return** Euclid($b, a \bmod b$)

The following result is important for understanding why this works, and is also useful in its own right. If $d = gcd(a,b)$ there are integers x and y such that

$$a x + b y = d$$

You can find a proof of this fact in the proof of Theorem 31.2 in the text. This equation is useful for the following reason. Suppose we started with

$$a x + b y$$

and sought a simple mathematical transformation that changed the values of some of these variables while keeping the overall value of the expression the same. A simple example of such a transformation is to add and subtract a factor of $b x$ from both $a x$ and $b y$:

$$a x + b y = (a x - b x) + (b y + b x) = (a - b) x + b (y + x)$$

More generally, if k represents the number of times b can divide evenly into a we can also do

$$\begin{aligned}
a x + b y &= (a x - k b x) + (b y + k b x) \\
&= (a - k b) x + b(y + k x) = (a \bmod b) x + b (y + k x)
\end{aligned}$$

Finally, if we make the second and first factors trade places we have an expression of the form

$$a' x' + b' y'$$

where a' is greater than b' and all the factors are integers. To summarize, we have

$$a x + b y = a' x' + b' y'$$

and

$$\left\{ \begin{array}{l} a' = b \\ b' = a \bmod b \\ x' = y + (a/b) x \\ y' = x \end{array} \right\}$$

If we repeat this process, eventually we will have $a' = d$ and $b' = 0$, and the chain of equalities leading back to the start will give us

$$a x + b y = d x' + 0 y'$$

At this point, we see that the final x' value has to be 1 and the final y' value can be anything. For convenience we can set $y' = 0$. Now that we know what the final x' and y' values are we can work our way backward through the steps of the algorithm to recover the original x and y values.

$$\left\{ \begin{array}{l} x = y' \\ y = x' - (a/b) y' \end{array} \right\} \tag{1}$$

Here is an example. We start by trying to compute the gcd of 147 and 75. On the first pass we compute only what happens to a and b in the algorithm.

a	b	(a/b)
147	75	1
75	72	1
72	3	24
3	0	-

On the next pass we work our way from the bottom back up using the formulas (1).

a	b	(a/b)	x	y
147	75	1	-1	2
75	72	1	1	-1
72	3	24	0	1
3	0	-	1	0

It is easy to verify that these are the correct x and y values.

$$a x + b y = 147(-1) + 75(2) = -147 + 150 = 3 = \gcd(a,b)$$

All of this actually quite easy to implement as an algorithm by taking advantage of a special recursive trick.

Extended-Euclid(a,b)

1. **if** $b = 0$
2. **then return** ($a,1,0$)
3. $(d',x',y') \leftarrow$ Extended-Euclid($b,a \bmod b$)
4. $(d,x,y) \leftarrow (d',y',x'-(a/b) y')$
5. **return** (d,x,y)

This algorithm is also quite efficient. It turns out that the number of recursive calls for a given pair of numbers a and b is $O(\lg b)$.

Congruences

We say that two integers a and b are congruent mod n if there is an integer y such that

$$a + n y = b$$

The notation that is used to express congruence is

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

The extended Euclid algorithm is useful for solving congruence equations of the form

$$a x \equiv g \pmod{b}$$

because solving this congruence is equivalent to solving

$$a x + b y = g \tag{2}$$

This equation has a solution if and only if the gcd of a and b divides evenly into g . If that is the case, we can solve (2) by first using the extended Euclidean algorithm to solve

$$a x + b y = \gcd(a,b)$$

and then multiply both sides of that solution by a factor of $g/\gcd(a,b)$.

Here are a couple of other useful facts about congruences that we will need to implement the RSA algorithm. The first is called the Little Fermat theorem. If p is a prime number and a is any positive integer then

$$a^{p-1} \equiv 1 \pmod{p}$$

The second is a corollary to a theorem called the Chinese remainder theorem. If p and q are prime numbers and

$$x \equiv y \pmod{p}$$

$$x \equiv y \pmod{q}$$

then

$$x \equiv y \pmod{p q}$$

The RSA algorithm

We now have all of the mathematical machinery in place to develop the mathematics needed to implement the RSA algorithm.

The algorithm begins with the selection of two large primes p and q . From p and q we construct

$$n = p q$$

$$\phi(n) = (p - 1)(q - 1)$$

The next step is to select a small odd integer e such that $\gcd(e, \phi(n)) = 1$. Next, we use the extended Euclidean algorithm to compute a d such that

$$d e \equiv 1 \pmod{\phi(n)}$$

This is equivalent to saying that

$$d e - k \phi(n) = 1$$

for some positive k or

$$d e = 1 + k (p - 1)(q - 1)$$

If we now take any integer M we have

$$M^{d e} = M^{1 + k (p - 1)(q - 1)} = M M^{k(p-1)(q-1)}$$

Using the Little Fermat theorem we have

$$M^{p-1} \equiv 1 \pmod{p}$$

so that

$$M^{d e} \equiv M (M^{p-1})^{k(q-1)} \equiv M (1)^{k(q-1)} \equiv M \pmod{p}$$

and similarly

$$M^{d e} \equiv M (M^{q-1})^{k(p-1)} \equiv M (1)^{k(p-1)} \equiv M \pmod{q}$$

The corollary to the Chinese remainder theorem now gives us

$$M^{d e} \equiv M \pmod{n}$$

Here then are the steps in the actual RSA algorithm needed to send a message securely from A to B .

1. B selects two large primes p and q and computes $n = p q$ and $\phi(n) = (p-1)(q-1)$
2. B selects a small odd number e such that $\gcd(e, \phi(n)) = 1$.
3. B uses Extended-Euclid to compute d such that $d e \equiv 1 \pmod{\phi(n)}$.
4. B publishes the pair (e, n) as a public key.
5. A breaks their message into blocks and converts each block into a large integer M .

6. A computes $M^e \pmod n$ and sends that to B .

7. B uses the pair (d, n) as a private key to compute $(M^e)^d \equiv M \pmod n$.
This recovers the original message M .

Additional Algorithmic Ingredients

As a final practical consideration, we need to briefly discuss a few other useful algorithms needed to implement parts of the RSA algorithm efficiently.

The first additional thing we need is an efficient exponentiation algorithm to compute things like $M^e \pmod n$. We actually saw such an algorithm way back at the beginning of the course in the handout on loop invariants. Here is the code for that algorithm:

```
z = x;
s = x;
t = y - 1;
while(t > 0)
  if(t % 2 == 0)
    {
      t = t / 2;
      s = s * s;
    }
  else
    {
      z = z * s;
      t = t - 1;
    }
```

This algorithm computes x^y by maintaining an invariant $x^y = z * s^t$. Not only is this a very efficient algorithm, but it is also trivial to modify into an algorithm that does all of its arithmetic mod n :

```
z = x;
s = x;
t = y - 1;
while(t > 0)
  if(t % 2 == 0)
    {
      t = t / 2;
      s = (s * s) % n;
    }
  else
    {
      z = (z * s) % n;
      t = t - 1;
    }
```

```

else
{
z = (z * s) % n;
t = t - 1;
}

```

If you need to squeeze even better performance out of this algorithm you can use an efficient multiplication scheme (such as FFT multiplication) to implement the multiplications.

A second ingredient needed to implement RSA efficiently is a way to construct the large random primes p and q . The method used is based on two results from number theory. The first result says that among large numbers primes occur with a somewhat predictable frequency. The result is that if you pick a random number in the vicinity of a large number n the odds that you will select a prime at random are $1/\ln n$. For most practical applications of RSA people try to find primes in the neighborhood of $n = 2^{1024}$. In that region primes occur with frequency

$$\frac{1}{\ln 2^{1024}} \approx \frac{1}{710}$$

In other words, if you generate approximately 1500 random 1024 bit numbers chances are pretty good that one of them will be prime.

The final thing we need is some efficient way to test candidate random numbers to see if they are prime. Here again number theory comes to the rescue with an idea called pseudoprimality testing. The idea is based on the Little Fermat theorem. If n is prime number then

$$a^{n-1} \equiv 1 \pmod{n}$$

for all numbers a . What is useful about this fact is that if n is not a prime then very often a^{n-1} will not equal $1 \pmod{n}$ for many choices of a . In fact, if we compute 2^{n-1} for a 1024 bit composite number there is only a one in 10^{41} chance of that accidentally coming out equal to $1 \pmod{n}$. We call numbers for which $2^{n-1} \equiv 1 \pmod{n}$ *pseudoprimes*. Pseudoprimes are almost always prime numbers and only very rarely turn to not be prime. Using an idea based on this observation we can quickly filter through our 1500 random candidates to find one or two pseudoprimes and then conduct some slower, more accurate tests to verify that they are in fact prime.