RSA

Named after its inventors (Rivest, Shamir, Adelman), another useful application of number theory.

Choose two big prime numbers of about the same size (same number of digits), call them p and q.
Compute n=p×q.
Choose a random number e, making sure that gcd((p-1)×(q-1),e)=1.
Compute d=modinv(e,(p-1)×(q-1)).

Note that (d×e)%((p-1)×(q-1)) = 1
which means that (d×e) = k×(p-1)×(q-1)+1 for some k (that's what % is all about)
Also remember Euler's Generalisation of Fermat's Little Theorem, which says that

$$\begin{aligned}
& A^{-1}\%N = A^{((p-1)\times(q-1))-1}\%N \\
\equiv\ & ((A^{-1}\%N)\times A)\%N = ((A^{((p-1)\times(q-1))-1}\%N)\times A)\%N \\
\equiv\ & 1 = ((A^{((p-1)\times(q-1))-1}\%N)\times A)\%N \\
\equiv\ & 1 = (A^{((p-1)\times(q-1))-1}\times A\%N) \\
\equiv\ & 1 = A^{((p-1)\times(q-1))}\%N
\end{aligned}$$

Now notice that for any message m (a number less than n),
if we "encrypt" it by computing $c=m^e\%n$,
then if we compute $x=c^d\%n$:, then:

$$\begin{aligned}
c^d\%n &= (m^e\%n)^d\%n \\
&= (m^e)^d\%n \\
&= m^{e\times d}\%n \\
&= m^{k\times(p-1)\times(q-1)+1}\%n \\
&= (m\times m^{k\times(p-1)\times(q-1)})\%n \\
&= (m\times(m^{(p-1)\times(q-1)})^k)\%n \\
&= ((m\%n)\times(m^{(p-1)\times(q-1)}\%n)^k)\%n \\
&= ((m\%n)\times 1^k)\%n \\
&= (m\%n)\%n \\
&= m\%n \\
&= m
\end{aligned}$$

So the act of raising something to the power of d modulo n can be reversed by raising the result to the power of e modulo n.
And by the same reasoning, the act of raising something to the power of e modulo n can be reversed by raising the result to the power of d modulo n.
Calculating *Discrete Modular Logarithms* is extremely hard, it can only be done by a brute-force search, which means:

If you know that $c=m^e\%n$,
and you know c, e, and n,
you still can't work out what m was,
unless you know d, in which case it is easy to work out $m=c^d\%n$,
but, even if you know what e and n are, you still can't work out what d is,
unless you know p and q,
and finding p and q from n is exactly as hard as finding the factors of a giant number.

(and the same is true if you swap around d and e)

What this means is that you have a secure* *Public Key* encryption algorithm. (*: it remains secure so long as it is impossibly difficult to find the factors of `n`).

There are two parts to it.

1. First, generate a Public and Private Key Pair:
   Choose two giant prime numbers, called `p` and `q`. (Let's say "giant" means 200 decimal digits)
   Compute `n = p×q`.   (so `n` has 400 digits)
   Choose a random number `e`, making sure that `gcd((p-1)×(q-1),e)=1`.
   Compute `d = modinv(e,(p-1)×(q-1))`.
   Throw away `p` and `q`; destroy all record of them. You'll never need them again, and if anyone else ever finds them all is lost. With current technology it will take 54,000,000,000 years for one computer to work out what `p` and `q` are.
   Designate the pair `(e,n)` as your *Public Key.*
   Designate the pair `(d,n)` as your *Private Key.*

   Publish your Public key in universal directories and databases; security is based on everybody knowing your public key, or at least being able to find it very easily, and it is very important that nobody should be able to pretend that your public key is something other than its true value.

   Keep your Private key Completely Secret and very securely protected. Don't lose it, don't let anybody see it. If you lose it nobody will be able to talk to you; if someone else finds it, they will be able to become you.
2. Use it.
   Let $E_{pub}(x)$ denote encrypting x with your public key, i.e. $E_{pub}(x)=x^{e}\%n$
   Let $E_{prv}(x)$ denote encrypting x with your private key, i.e. $E_{prv}(x)=x^{d}\%n$
   X must be less than n to encrypt it in one go, but larger messages are just split up into smaller blocks that are encrypted one-by-one. If n is a 400 decimal digit number, that is equivalent to 1327 bits, so you can split a message up into convenient 128-byte chunks for encryption.
   Remember that $E_{pub}(E_{prv}(x))=x$ and $E_{prv}(E_{pub}(x))=x$
   Everyone in the world, including you, can perform $E_{pub}(x)$
   You and only you can perform $E_{prv}(x)$

To make some data *X* secret, replace it with $X=E_{pub}(X)$. Only you can recover it with $X=E_{prv}(X)$.

To be sure your data hasn't been changed, produce a one-way-hash *H* of it, and encrypt the one way hash with $H'=E_{prv}(H)$. Anybody can decrypt *H'* to find out what the one-way-hash value was, but so what? If anybody changes your data, the value of the one-way-hash will change; they can compute the new one-way-hash value, but they can't fool you, because they can't produce a new valid *H'*. (To "Sign and Seal" a document encrypt a one-way-hash of it with your private key).

If somebody wants to send a message *M* that only you can read, they perform $C=E_{pub}(M)$, and send *C* to you by totally insecure means. They can even publish it in the newspaper, knowing that only you will be able to work out what *M* was. (A private message to A is encrypted with A's public key).

If you want to send a message to somebody, so that they can be absolutely certain that it really was you who sent it, and nobody has modified it in any way, you perform $C=E_{prv}(M)$, and send *C* to them. Anyone who sees *C* can decrypt it with $M=E_{pub}(C)$, but anybody who finds that $E_{pub}(C)$ produces a valid message knows that it must have been created by $C=E_{prv}(M)$, and therefore knows that you must have created it. (A signed message from A is encrypted with A's private key).

These last two can be combined: If A wants to send a message to B, so that only B can read it, and B can be sure that it really came from A and hasn't been altered, A encrypts it with his own private key and then again with B's public key.

To prove your identity to a stranger, send them a personalised, timed and dated, greeting encrypted with your own private key. E_{prv}(*"Howdy Mr. Smith at 12.15 on 27th November 2002 from X. Jones"*). By retrieving X. Jones' public key from the universal directory/database, Mr. Smith can decrypt the greeting and know it was written by X. Jones, the person you claim to be. The time and date simply ensure that you aren't just passing on a greeting that the real X. Jones gave to you in the past.

It is very important that public keys really are public. Not just non-secret, but actively publicised and universally accessible. It must be impossible for anyone to get away with falsifying the record of someone's public key.

How? The system comes to its own rescue. If there is one (or just a few) central public database for public keys (like directory enquiries for the phone company (before deregulation)), that central registry can have its own public-private key pair. The one and only public key for the one and only central registry can be built into the hardware of communication devices, so they can never be tricked about it, and the central registry simply encrypts all responses to public key enquiries with its own private key.

R.S.A. is only as secure as it is difficult to factorise very big numbers. If somebody discovers a new factoring method all could be lost. But then, if somebody discovers a flaw in the permutations of DES or RC4, all is equally lost. With a public key system, each person only needs to keep one encryption key ever (but that one key is very valuable). For symmetric systems (e.g. DES, RC4) there must be a separate pair for each pair of people who ever want to communicate. Your RSA private key provides you with a secure digital identity, symmetric encryption keys can't do anything of the sort.

With current technology, a computer would have to devote 48,000,000 years to crack a 1024-bit RSA key. you would only need a 73-bit DES key to require the same amount of cracking effort. But then, how would you make a secure 73-bit version of DES? 56-bit DES is nearly 1,000,000 times less secure than 73-bit. And how important is key length? People are not expected to memorise them.

DES encryption is approximately 1,000 times faster than RSA encryption; RC4 is 10 times faster than that.

Don't just pick one and use it for everything. It is quite reasonable to pick a 2048-bit RSA key pair (which would take 42,000,000,000,000,000 years of modern computing to crack) and only use it to encrypt small things (digital signatures, one-way-hashes, etc.) so it doesn't matter how slow it is. Then when you need secure communications for something big, make up a *Session Key*, an encryption key that will only ever be used once, then destroyed; use 2048-bit RSA to encrypt it and securely send it to your friend, then use that session key with something fast like RC4 for this one long communication.

```
$ cat rsa.cpp
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void usage(void)
{ fprintf(stderr, "either: rsa gen <keylength-up-to-31>\n");
  fprintf(stderr, "    or: rsa enc <key>\n");
  exit(1); }

int isprime(int n)
{ if (n%2==0) return 0;
  int max=(int)(sqrt(n)+1);
  for (int i=3; i<max; i+=2)
    if (n%i==0) return 0;
  return 1; }

int firstprimefrom(int n)
{ if (n%2==0) n+=1;
  while (1)
  { if (isprime(n)) return n;
    n+=2; } }

int gcd(int a, int b)
{ while (b!=0)
  { int t=a%b;
    a=b;
    b=t; }
  return a; }

int extgcd(int a, int b, int & m, int & n)
{ int m1, n1, g;
  if (b==0)
  { g=a; m=1; n=0;
    return g; }
  g=extgcd(b, a%b, m1, n1);
  int quo=a/b;
  m=n1;
  n=m1-quo*n1;
  return g; }

int modinv(int a, int n)
{ int g, x, y;
  g=extgcd(a, n, x, y);
  if (g!=1)
  { fprintf(stderr, "Error: impossible modinv(%d,%d)\n", a, n);
    exit(1); }
  return x; }

int modpower(int base, int expo, int modu)
{ long long int b=base;
  long long int m=modu;
  long long int a=1;
  while (expo>0)
  { if (expo&1)
    { a*=b;
      a%=m; }
    b*=b;
    b%=m;
    expo>>=1; }
  return a; }

void main(int argc, char *argv[])
{ if (argc!=3)
    usage();
  if (strcasecmp(argv[1], "gen")==0)
  { int keylen=atol(argv[2]);
    if (keylen<=0 || keylen>30)
    { fprintf(stderr, "keylength of %d bits no good\n", keylen);
      exit(1); }
    srandomdev();
    int keymin=1<<(keylen/2);
    int keyrange=keymin/2;
    int p=firstprimefrom(keymin+random()%keyrange);
    int q=firstprimefrom(p+random()%keyrange);
    int n=p*q;
    printf("p = %d\n", p);
    printf("q = %d\n", q);
    printf("n = %d\n", n);
    int e, d;
    do
    { e=random()%keylen+2;
      while (gcd(e, (p-1)*(q-1))!=1)
        e+=1;
      d=modinv(e, (p-1)*(q-1)); }
        while (d<0 || e<0);
    printf("d = %d\n", d);
    printf("e = %d\n", e);
    printf("your  public key is %d:%d\n", e, n);
    printf("your private key is %d:%d\n", d, n); }
  else if (strcasecmp(argv[1], "enc")==0)
  { char *s=strtok(argv[2], ":");
    int d=atol(s);
    s=strtok(NULL, ":");
    int n=atol(s);
    printf("d = %d\n", d);
    printf("n = %d\n", n);
    while (1)
    { printf("? ");
      int m, k=scanf("%d", &m);
      if (k!=1) break;
      int c=modpower(m, d, n);
      printf(" -> %d\n", c); } }
  else
    usage(); }
```

```
$ rsa gen 20
p = 1523
q = 1931
n = 2940913
d = 652769
e = 9
your  public key is 9:2940913
your private key is 652769:2940913
```

```
$ rsa enc 9:2940913
d = 9
n = 2940913
? 12
 -> 1418950
? 123
 -> 787644
? 111
 -> 1424022
? ctrl-D
```

(The numbers 12, 123, 111 are encrypted as 2940913, 1418950, 1424022)

```
$ rsa enc 652769:2940913
d = 652769
n = 2940913
? 1418950
 -> 12
? 787644
 -> 123
? 1424022
 -> 111
? ctrl-D
```

(The numbers 2940913, 1418950, 1424022 are decrypted as 12, 123, 111)

Table 7.1
Average Time Estimates for a Hardware Brute-Force Attack in 1995

	Length of Key in Bits					
Cost	40	56	64	80	112	128
\$100 K	2 seconds	35 hours	1 year	70,000 years	10^{14} years	10^{19} years
\$1 M	.2 seconds	3.5 hours	37 days	7000 years	10^{13} years	10^{18} years
\$10 M	.02 seconds	21 minutes	4 days	700 years	10^{12} years	10^{17} years
\$100 M	2 milliseconds	2 minutes	9 hours	70 years	10^{11} years	10^{16} years
\$1 G	.2 milliseconds	13 seconds	1 hour	7 years	10^{10} years	10^{15} years
\$10 G	.02 milliseconds	1 second	5.4 minutes	245 days	10^{9} years	10^{14} years
\$100 G	2 microseconds	.1 second	32 seconds	24 days	10^{8} years	10^{13} years
\$1 T	.2 microseconds	.01 second	3 seconds	2.4 days	10^{7} years	10^{12} years
\$10 T	.02 microseconds	1 millisecond	.3 second	6 hours	10^{6} years	10^{11} years

64 bits = 10^{19}
$\frac{1}{2}\,10^{19}$ tried/year $\approx$ 10^{11} tried/second
at 10^{7} attempts/second
= 10,000 in parallel

Table 7.2
Brute-Force Cracking Estimates for Chinese Lottery

			Time to Break	
Country	Population	# of Televisions/Radios	56-bit	64-bit
China	1,190,431,000	257,000,000	280 seconds	20 hours
U.S.	260,714,000	739,000,000	97 seconds	6.9 hours
Iraq	19,890,000	4,730,000	4.2 hours	44 days
Israel	5,051,000	3,640,000	5.5 hours	58 days
Wyoming	470,000	1,330,000	15 hours	160 days
Winnemucca, NV	6,100	17,300	48 days	34 years

(All data is from the *1995 World Almanac and Book of Facts.*)

lengths.

Factoring large numbers is hard. Unfortunately for algorithm designers, it is getting easier. Even worse, it is getting easier faster than mathematicians expected. In 1976 Richard Guy wrote: "I shall be surprised if anyone regularly factors numbers of size 10^{80} without special form during the present century" [680]. In 1977 Ron Rivest said that factoring a 125-digit number would take 40 quadrillion years [599]. In 1994 a 129-digit number was factored [66]. If there is any lesson in all this, it is that making predictions is foolish.

Table 7.3 shows factoring records over the past dozen years. The fastest factoring algorithm during the time was the quadratic sieve (see Section 11.3).

These numbers are pretty frightening. Today it is not uncommon to see 512-bit numbers used in operational systems. Factoring them, and thereby completely compromising their security, is well in the range of possibility: A weekend-long worm on the Internet could do it.

Computing power is generally measured in mips-years: a one-million-instruction-per-second (mips) computer running for one year, or about $3*10^{13}$ instructions. By convention, a 1-mips machine is equivalent to the DEC VAX 11/780. Hence, a mips-year is a VAX 11/780 running for a year, or the equivalent. (A 100 MHz Pentium is about a 50 mips machine; a 1800-node Intel Paragon is about 50,000.)

The 1983 factorization of a 71-digit number required 0.1 mips-years; the 1994 fac-

Table 7.3
Factoring Using the Quadratic Sieve

Year	# of decimal digits factored	How many times harder to factor a 512-bit number
1983	71	>20 million
1985	80	>2 million
1988	90	250,000
1989	100	30,000
1993	120	500
1994	129	100