Modular Arithmetic

$(A + B) \% n = (A\%n + B\%n) \% n$

$(A - B) \% n = (A\%n - B\%n) \% n$

$(A \times B) \% n = (A\%n \times B\%n) \% n$

Multiplication table modulo 7

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Can see that $2\times4=1$, $3\times5=1$, $4\times2=1$ $5\times3=1$, $6\times6=1$,
Which means that multiplying by 4 is the same as dividing by 2,
multiplying by 5 is the same as dividing by 3, etc.
3 is the "modular inverse" of 5, modulo 7
4 is the "modular inverse" of 2, modulo 7
So modulo 7, division can be done meaningfully.
This always works out when the modulus is prime.

Multiplication table modulo 6

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

Can see that nothing$\times4=1$,
Which means that division modulo 6 can not be done.

Think of a binary number: 1001101. This is 77.
It tells us the 77 = 64 + 8 + 4 + 1.
And makes it easy to work out anything to the power of 77.

$$\begin{aligned} A^{77} &= A^{64+8+4+1} \\ &= A^{64} \times A^{8} \times A^{4} \times A^{1} \\ &= A^{1} \times A^{4} \times A^{8} \times A^{64} \\ &= A \times (A^2)^2 \times ((A^2)^2)^2 \times (((((A^2)^2)^2)^2)^2)^2 \end{aligned}$$

Run a loop, looking at each digit of the exponent in turn, also squaring that value of A each time round. For any 1 in the binary for the exponent, multiply the answer so far by the A so far.

```
int power(int A, int B)
{ int answer=1;
  while (B>0)
  { if (B & 1)
      answer*=A;
    A*=A;
    B>>=1; }
  return answer; }
```

A large number to the power of a large number is a really huge number, and would be very difficult to compute. But if it is all done modulo N, the answer and all intermediate results will be less than N.

$(1263182^{3571532}) \% 1000000$ can be computed very quickly and easily, and only has six digits.

Many encryption algorithms use very very bit integers, many hundreds of bits long. These have to be implemented specially in software, there is not long long long int long enough to give hardware support. The operations performed on these giant numbers are addition, subtraction, multiplication, and to-the-power-of. Taking one giant number to the power of another giant number produces a result whose giantness boggles the mind. Fortunately, we nearly always use modular arithmetic (i.e., we don't just calculate A*B, but instead (A*B)%N), and this restricts the size of the results. Nothing %N can be bigger than N. There are some simple rules for simplifying modular arithmetic:

$$(A+B)\%N = ((A\%N) + (B\%N))\%N$$
$$(A-B)\%N = ((A\%N) - (B\%N))\%N$$
$$(A*B)\%N = ((A\%N) * (B\%N))\%N$$
$$(A^B)\%N = ((A\%N)^B)\%N$$

Note that the to-the-power-of operation does not follow exactly the same pattern as the others: $(A^B)\%N \neq ((A\%N)^{(B\%N)})\%N$. To-the-power-of still requires repeated multiplication, but not as many times as one might expect. For example, $A^{85} = A^{(64+16+4+1)} = A^{64}*A^{16}*A^4*A^1$. Calculating A^{64}, A^{16}, A^4 and A^1 can be done by repeated squaring: $A^4=(A^2)^2$, $A^{16}=((A^4)^2)^2$, $A^{64}=((A^{16})^2)^2$, so the whole value of A^{85} can be computed from A with just 9 multiplications. And of course when calculating $A^{85}\%N$, the %N operation can be performed after every step to keep the numbers small.

Important: The % operator in C/C++ is not required to implement modulus correctly. For positive operands it will be correct, but if either A or B is negative, A%B will usually also be negative. For mathematical and cryptological applications that is incorrect. if A%B comes out negative, it must be replaced by A%B+abs(B). In these notes, I use the % sign everywhere; in all cases it means the mathematical, positive-only operation.

Division doesn't make sense in modular arithmetic. Normally we can say that 18÷2=9 because 9*2=18; division simply reverses multiplication. But what is (6÷2)%12? Both (3*2)%12=6 and (9*2)%12=6 are true, so modulo 12, 6÷2 could be 9 or 3.

To avoid confusion, one never talks about division modulo anything. Instead, you try to work out a number's modular inverse. If a number has a modular inverse, then multiplying by it has the effect of reversing a multiplication by the number itself. If a number does not have a modular inverse then there is nothing that division could mean. The inverse of a number A modulo N is written as

$A^{-1}\%N$ or $A^{-1} \bmod N$ or modinv(A,N)

If $A^{-1}\%N$ exists, then

$$(((B*A)\%N)*(A^{-1}\%N))\%N = B\%N$$

or

$$(((B * A * \text{modinv}(A,N))\%N = B\%N$$

and the existence test is simple:

If A and N have no divisors in common, or in other words
if there is nothing that A and N are both divisible by, or in other words
if the greatest common divisor of A and N is 1, i.e.
if gcd(A,N)==1,
then modinv(A,N) exists and has one unique unambiguous value,
otherwise there is no such thing as modinv(A,N).

Modular inverses are very important to some forms of encryption.

Euclid's Algorithm for finding the Greatest Common Divisor has been known for thousands of years:

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

A simple improvement to it provides the Extended GCD algorithm, which not only returns the GCD of two numbers, but also finds two other important values:

```
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; }
```

Although the presence of the loop in `gcd` and the recursion in `extgcd` make it seem otherwise, these two functions are surprisingly fast. No known general-purpose algorithms does the job significantly faster.

If `extgcd(A, B, X, Y) = G`, then `G = X*A + Y*B`, and `G` is also the GCD of `A` and `B`. Because `modinv(A,N)` only exists if `GCD(A,N)==1`, we know that if `extgcd(A,N,X,Y)=1` then

`modinv(A,N)` exists and `X*A+Y*N=1`

therefore `X*A = 1-Y*N`

therefore `(X*A)%N = (1-Y*N)%N`

therefore `(X*A)%N = 1%N - (Y*N)%N`

and because `Y*N` must be a multiple of `N`, `Y*N%N` is zero,

therefore `(X*A)%N = 1`

so `modinv(A,N) = X =` A^{-1}`%N`

Giving this function:

```
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; }
```

and effectively allowing this to be considered true:

`(A/B)%N = ((A%N) * modinv(B,N))%N`

Two Special Cases for Calculating Modular Inverse:

According to *Fermat's Little Theorem*:

if `N` is prime, and `A<N`, then

`modinv(A,N) =` A^{N-2}`%N`

According to Euler's Generalisation of Fermat's Little Theorem:

if `N` is the product of two primes, `N=p×q`, and `A<p`, and `A<q`, then

`modinv(A,N) =` $A^{((p-1)\times(q-1))-1}$`%N`

Finding Prime Numbers

Finding a Big Prime Number is a fairly easy problem, but can not be done quickly unless some small chance of error is acceptable.

Checking that a number is prime, with absolute accuracy:

```
#include <stdio.h>
#include <stdlib.h>

long long int valof(char *s)
{ long long int n=0;
  for (int i=0; 1; i+=1)
  { char c=s[i];
    if (c==0) return n;
    n=n*10+c-'0'; } }

void main(int argc, char *argv[])
{ if (argc!=2)
  { fprintf(stderr, "Need a number on the command line\n");
    exit(1); }
  long long int n=valof(argv[1]);
  int ok=1;
  if (n%2==0 && n!=2) ok=0;
  int max=(int)(sqrt(n)+1);
  for (int i=3; ok && i<max; i+=2)
    if (n%i==0)
      ok=0;
  if (ok)
    printf("%lld is definitely prime\n", n);
  else
    printf("%lld is definitely NOT prime\n", n); }
```

To find a large prime number, the only known method is to pick a random number of the right size, and see if its prime. If it is, the task is over. If it isn't, just try the next number. Of course, you would have enough sense to only test odd numbers, and it would also pay to eliminate multiples of 3, 5, 7, 11, and a few other small primes before calling a primality-testing function.

The probability that an arbitrarily chosen number N is prime is $1/\log_e(N)$, so not many numbers will have to be tested before a prime is found. Even for 200-digit numbers, 1 in $\log_e(10^{200})$, or 1 in $200\times\log_e(10)$, or 1 in 461 will be prime.

When testing the number N, the loop goes round $\frac{1}{2}\sqrt{N}$ times. That may seem fast, but secure encryption uses very big numbers. If you want to check a 50 digit number for primeness, the loop will be executed 5,000,000,000,000,000,000,000,000 times.

Fortunately there is a faster trick: *Probabilistic Prime Checking*. Some rather complicated theory tells us that for any random number R between 2 and N-2, calculate $x=R^{((n-1)/2)}\%n$. If N is prime, then x can not possibly be equal to either N or to N-1. If N is <u>not</u> prime, the probability of x being equal to either N or to N-1, is ½. So just pick lots of random Rs. If ever the value of x comes out to N or N-1 you instantly know that N is not prime. If you survive K random selections, you now the probability of N not being prime is $\frac{1}{2}^{K}$.

Finding Prime Numbers

Finding a Big Prime Number is a fairly easy problem, but can not be done quickly unless some small chance of error is acceptable.

Checking that a number is prime, with absolute accuracy:

```
#include <stdio.h>
#include <stdlib.h>

long long int valof(char *s)
{ long long int n=0;
  for (int i=0; 1; i+=1)
  { char c=s[i];
    if (c==0) return n;
    n=n*10+c-'0'; } }

void main(int argc, char *argv[])
{ if (argc!=2)
  { fprintf(stderr, "Need a number on the command line\n");
    exit(1); }
  long long int n=valof(argv[1]);
  int ok=1;
  if (n%2==0 && n!=2) ok=0;
  int max=(int)(sqrt(n)+1);
  for (int i=3; ok && i<max; i+=2)
    if (n%i==0)
      ok=0;
  if (ok)
    printf("%lld is definitely prime\n", n);
  else
    printf("%lld is definitely NOT prime\n", n); }
```

To find a large prime number, the only known method is to pick a random number of the right size, and see if its prime. If it is, the task is over. If it isn't, just try the next number. Of course, you would have enough sense to only test odd numbers, and it would also pay to eliminate multiples of 3, 5, 7, 11, and a few other small primes before calling a primality-testing function.

The probability that an arbitrarily chosen number N is prime is $1/\log_e(N)$, so not many numbers will have to be tested before a prime is found. Even for 200-digit numbers, 1 in $\log_e(10^{200})$, or 1 in $200\times\log_e(10)$, or 1 in 461 will be prime.

When testing the number N, the loop goes round $\frac{1}{2}\sqrt{N}$ times. That may seem fast, but secure encryption uses very big numbers. If you want to check a 50 digit number for primeness, the loop will be executed 5,000,000,000,000,000,000,000,000 times.

Fortunately there is a faster trick: *Probabilistic Prime Checking*. Some rather complicated theory tells us that for any random number R between 2 and N-2, calculate $x=R^{((n-1)/2)}\%n$. If N is prime, then x can not possibly be equal to either N or to N-1. If N is <u>not</u> prime, the probability of x being equal to either N or to N-1, is ½. So just pick lots of random Rs. If ever the value of x comes out to N or N-1 you instantly know that N is not prime. If you survive K random selections, you now the probability of N not being prime is $\frac{1}{2}^K$.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

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

void main(int argc, char *argv[])
{ if (argc!=3)
  { fprintf(stderr, "Need a number and a probability logarithm on the command line\n");
    exit(1); }
  srandomdev();
  int n=atol(argv[1]);
  if (n%2==0 && n!=2)
  { printf("%d is definitely NOT prime\n", n);
    exit(1); }
  int pp=atol(argv[2]);
  double p=exp(pp*log(10.0));
  double psofar=1.0;
  int ok=1;
  while (psofar>p)
  { int r=random() % (n-2) + 2;
    int x=modpower(r, (n-1)/2, n);
    if (x!=1 && x!=n-1) { ok=0; break; }
    psofar*=0.5; }
  if (ok)
    printf("%d is prime with probability better than %.15f\n", n, 1.0-p);
  else
    printf("%d is definitely NOT prime\n", n); }
```

This algorithm has two parameters: N, the number we wish to check for primeness, and P the acceptable probability of a wrong answer (actually P's logarithm is provided, so an input of -6 means that a probability of error of $P=10^{-6}$ (one in a million) is acceptable. Usually a *much* lower probability is required.

If this program says that a number is NOT prime, it is definitely correct.
If it says that a number IS prime, there is still a 10^{-6} probability that it actually isn't.

The number of times around the loop is $-\log_2(P)$, which does not depend on the size of the number. For a one-in-a-million chance of error, 20 times round the loop. For one-in-a-million-million, 40 times round the loop, etc.

This program uses "long long" ints for intermediate calculations, but is still restricted to single precision (32 bit) ints for the value of N.

Finding Factors.

Once you know that a number is not prime, there is no quick way to find out what its factors are. If a number `N` is the product of two similarly-sized primes, there in nothing much better that just trying out all possible factors (up to √`N`) to see if they divide into `N` or not. That's nearly ½√`N` divisions, so to find the factors of a 200 digit number could require about 10^{99} trial divisions, and they would be slow `BigInt` divisions, not simple integer divisions supported by hardware.

Rabbit takes 3mS to divide a 200 digit number by a 100 digit number. The fastest PC available today, with a program very specially optimised for speed most probably couldn't do it as quickly as 100μS, so finding the factors of a 200-digit number by this method should take about 3,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 years (that is 3×10^{87} years).

In 1999, a group of people using a very complex method called a *Number Field Sieve* succeeded in factorising a 155 digit number (which was the product of two 78 digit primes), and in so doing set the record for the biggest difficult number ever factorised at that time. The computational effort was roughly equivalent to a 3GHz Pentium running continuously for 8 years, and required 2GB of RAM.

They won $100 for that achievement, but it was a lot more significant than that measly prize would suggest. The particular number is known as RSA-155, and was part of the "RSA challenge". What they found was that

```
  1094173864157052742180970732204035761200373294544920599091384213147634998428893478471799725789126733249762575289978183379707653724402714674353159335433897
=
  102639592829741105772054196573991675900716567808038066803341933521790711307779
*
  106603488380168454820927220360012878679207958575989291522270608237193062808643
```

The security of the very famous and popular RSA encryption algorithm depends upon it being difficult to find the factors of very big numbers that are in fact the product of two same-sized primes. A favourite key-length for RSA is 512 bits, and 512 bits is equivalent to 155 decimal digits.

The time taken by a Number Field Sieve to factorise N is approximately proportional to

$$e^{1.9\times(\log_e(N))^{1/3}\times(\log_e(\log_e(N)))^{2/3}}$$

Given the time taken for RSA-155, we can produce the following table, showing the number of digits in a hard-to-factorise number, and the number of years it would take the fastest of modern PCs to perform that factorisation using the best *currently known* algorithm.

bits	digits in number	years required for one fast 2003-vintage PC to factorise
512	155	8
640	194	770
661	200	1,500
768	232	44,000
896	271	1,700,000
993	300	22,000,000
1024	309	48,000,000
1536	463	4,200,000,000,000
1658	500	43,000,000,000,000
2048	618	42,000,000,000,000,000
3319	1000	3,000,000,000,000,000,000,000,000
4096	1234	25,000,000,000,000,000,000,000,000,000