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 2×4=1, 3×5=1, 4×2=1 5×3=1, 6×6=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×4=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.

$(126318^{2^{3571532}}) \% 1000000$ can be computed very quickly and easily, and only has six digits.

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 $\frac{1}{2}^K$.

```
#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`.