

Computer Tutorial 6

Start MAGMA, type `SetLogFile("ctut6.txt");` and type `load "tut6data.txt";`.

1. You can get MAGMA to tell you how long a command takes by preceding the command with `time`. For example, try

```
time Factorial(10000);
```

In fact, most of the time here was taken up printing the value of 10000!; so if you want to know how long it took just to compute 10000! it is better to use

```
time x:=Factorial(10000);
```

since this gets MAGMA to compute it without printing it.

Now let us choose some random 50 digit numbers and see how long it takes MAGMA to factorize them. Repeat the following commands a dozen or so times:

```
n:=Random(10^49,10^50);
```

```
time Factorization(n);
```

and observe that numbers with two large prime factors seem to be the hardest.

2. For the RSA cryptosystem (and also for Elgamal, which we will do later) it is necessary to be able to find very large prime numbers. So it is quite fortunate that testing whether or not a very large number is prime is, in fact, not as hard a computation as you might expect. We will make use of the MAGMA function `NextPrime`, which returns the next prime number greater than the input value. Do the commands

```
n:=Random(10^49,10^50);
```

```
time NextPrime(n);
```

a few times, observing that MAGMA does it quickly.

3. Choose two 25 digit primes and let n be their product:

```
p:=NextPrime(Random(10^24,10^25));
```

```
q:=NextPrime(Random(10^24,10^25));
```

```
n:=p*q;
```

Now do

```
time Factorization(n);
```

and observe (by comparison with Exercise 1) that a 50 digit number that is the product of two large primes is usually harder to factorize than a randomly chosen 50 digit number. Now repeat the above sequence of commands with the numbers 24 and 25 replaced by 25 and 26, then by 26 and 27, and so on. You will probably find that it gets significantly slower with each increase.

4. Why is primality testing easier than factorizing? All factorizing techniques require a certain amount of random hunting for factors, and when the number is large there are a lot of possibilities to deal with. But there are primality tests that do not involve any hunting, just computation. For example, Fermat's Little Theorem tells us that if n is a prime number and a any positive integer less than n then a^{n-1} is congruent to 1 mod n . Computing the residue of $a^{n-1} \bmod n$ can be done very efficiently, and the MAGMA command `Modexp(a,n-1,n)` does so. If we can find an integer a less than n such that `Modexp(a,n-1,n)` does not give 1 then n is not prime. This almost always provides a fast way of proving that a composite number is composite without having to factorize it. Furthermore, if `Modexp(a,n-1,n)` equals 1 for even one value of a (except 1) then it is rather likely that n is prime. The following procedure chooses random numbers until a number is found that fails this test for compositeness:

```

testrandom:=procedure(~n,N)
i:=0;
repeat
i,"random numbers tested and found to be composite";
n:=Random(N);
i:=i+1;
a:=2+Random(n-3);
until Modexp(a,n-1,n) eq 1;
end procedure;

```

Enter the above, and then type `testrandom(~n,10^100)`; . When it stops, type `n`; to see the number that failed this compositeness test. Get MAGMA to print it (`n`;) and check that it really is prime via `IsPrime(n)`; . (It is *possible* that your `n` will turn out not to be prime, but I'll bet this doesn't happen.) Repeat the test a few times.

- In this exercise we illustrate one possible implementation of the RSA cryptosystem. Choose a random number of 101 digits via the command `a:=Random(10^100,10^101)`; and then define p via `p:=NextPrime(a)`; . Type `p`; to see the prime you have got. Choose another 101 digit prime q similarly. Define `n:=p*q`; , `e:=65537`; , and `d:=InverseMod(e,(p-1)*(q-1))`; . Type `d`;

```

e*d mod ((p-1)*(q-1));

```

RSA encryption and decryption relies on the fact that if $ef \equiv 1 \pmod{(p-1)(q-1)}$ then $b \equiv a^e \pmod{n}$ if and only if $a \equiv b^d \pmod{n}$. To encrypt a message we first convert the message into a sequence of numbers that are all less than n . For example, we could split our text into blocks of length 66 characters each, and represent each character by some number between 100 and 999. Then just concatenate these numbers to make a number 198 digits long. (Note that our `n` has more than 198 digits.) The startup file `MagmaProcedures.txt` contains a function `NaiveEncoding` to do this. Type (or copy and paste)

```

usydmaths:=NaiveEncoding("Computer Tutorial 6,
MATH2068/2988 (Number Theory and Cryptography),
School of Mathematics and Statistics,
The University of Sydney,
NSW 2006,
Australia.");
usydmaths;
NaiveDecoding(usydmaths);

```

Now encipher the plaintext `usydmaths` via

```

ct:=[Modexp(m,e,n): m in usydmaths];

```

and type `ct`; to see the ciphertext. Then decipher it via

```

pt:=[Modexp(m,d,n): m in ct];
NaiveDecoding(pt);

```

- Two modern fables by James Thurber have been enciphered using exactly the same method as in Exercise 5. To see the first ciphertext you have to decipher, type `ctext1`; . Type `n1`; and `e1`; to see the modulus and encryption exponent used. (These correspond to `n` and `e` in Exercise 5.) The prime factors of `n1` are given: type `p1`; and `q1`; to see them. Find the decryption exponent `d1`, and decipher the message. Then do the same for `ctext2`: you are given `n2`, `e2`, `p2` and `q2`, and must find `d2` to decipher the message.
- The numbers 1, 11, 111, 1111, ... (composed entirely of 1's) are called *repunits*. You should be able to see that if $m|n$ then the m -th repunit is a factor of the n -th repunit. There are only five prime repunits known. See if you can find them. (Note: The largest of them has more than 1000 but fewer than 2000 digits. You need to use a loop to generate successive repunits – e.g. start with `r:=1`; and do `r:=10*r+1`; with each successive iteration of the loop. Use `IsProbablyPrime(r)`; rather than `IsPrime(r)`; since it is much faster. You only need to test those repunits that have a prime number of digits.)