ACCESS 2004 - RSA
Thursday June 17

Our Plan Today:

To help clarify the RSA public-key encryption method that Jim introduced yesterday, we will do the example worked out in the Tom Davis "Cryptography" notes, page 13-14. These are great notes - you can think of them as the Cliff Notes for "The Code Book." Davis' example uses small numbers and a one-letter message, and Maple will do all of our computations. I believe this will make the algorithm more clear to you. The explanation of the Algorithm on page 6-7 of the Rivest-Shamir-Adleman paper is also a very concise outline which will make more and more sense as you play with examples.

After we digest Davis' example we will try somewhat larger prime numbers, to help prepare you for the part of your group project in which you send yourselves (and me) encoded messages . In your actual project, which will be posted later today, you will impliment a medium-sized version of an RSA cipher system (big enough to send short messages, but not really big enough to be secure). You will also incorporate the "secure signature" feature, which we will discuss today. You can read about this in section 8.2 of Davis' notes, or in section IV of the Rivest-Shamir-Adleman paper.

We will stop our lab work by 10:10 today so that we have plenty of time to make our way over to JTB 120 for the lecture by Professor Seger at 10:30; we will back here all of tomorrow morning to tie up loose ends we don't finish today, and to get a good start on your week 1 projects.

The number theory we've talked about, and the relation to RSA cryptography, is usually taught in our number theory course, Math 4400. This is a senior level course, so I would expect that some of what we've talked about has been challenging. Still, students can take Math 4400 fairly early in their undergraduate careers since it does not have very many prerequisites beyond algebra and the ability to reason mathematically. As far as ACCESS goes, we hope that you're enjoying the magic hidden in modular arithmetic, and that you are pleasantly surprised that this "abstract" mathematics turns out to be so practical.

I will make you do a lot of your own typing in Part I below, so that you can begin learning MAPLE and common errors which users make. Therefore, in the file you download, many of the Maple commands which you see in the hardcopy are gone.

Part I

The Davis Example:

In this example Bob is going to send a message to Alice. I will follow Davis' numbering of the steps on pages 13-14. We are also going to use his table on page 9 to convert letters to numbers.

1) **Alice** must create her public key, for Bob to use when he encripts his message to her. So she picks two prime numbers, see page 13: (p=23 and q=41). Define p and q using Maple.

```
> restart: #Just as every time you reopen a file you must
         #re-execute its commands into Maple's memory, you will
         #often want to redo a process in a given file using
         #different data variables.  The restart command clears
         #all of the current memory.
```

```
> p:=23;
  q:=41;
     #Insert prompts with the [> button on the menu bar.
     #use shift-return for multi-line commands.
     #make definitions with the symbols :=, NOT with =
     #comment lines have # as their first character
```

$$p := 23$$

$$q := 41$$

2) **Alice** defines her modulus to be the product of p and q. This will be the first piece of her public key.

```
> N:=p*q;
```

$$N := 943$$

3a) **Alice privately** computes the auxillary modulus N2:=(p-1)*(q-1), which is related to Euler's theorem, with which Alice will pick her encoding and decoding powers. No one else will ever see or use this number. First she finds a number e which is relatively prime to N2; this will be the public encoding power and she will tell it to the world. A good e must be relatively prime to (p-1)*(q-1), so that Alice will be able to find a decoding power d. So we check the gcd:

```
> N2:=(p-1)*(q-1);
  e:=7;
```

$$N2 := 880$$

$$e := 7$$

```
> gcd(e,880);   #greatest common denominator
                #-we want it to be 1
```

$$1$$

```
> ifactor(N2); ifactor(e);
                #for small numbers (only) we can also
                #compare factors to deduce the gcd.
                #How would you find out about these commands if you
                #weren't sure how to use them, or even if they
  existed?
                #Answer: you would use the Help button at the
                #top right of the menu.
```

$$(2)^4\ (5)\ (11)$$

$$(7)$$

7) **Alice privately** finds her "secret" decoding power. Since she does this step sooner than Davis says, we will too. Since e is relatively prime to (p-1)*(q-1) it has a multiplicative inverse d, mod (p-1)*(q-1). (We talked about how to find the multiplicative inverse using the Euclidean algorithm. Luckily for us, Maple has a subroutine which does this step for us.) By Euler's Theorem, which Jim talked about yesterday (very briefly), this d will be the decoding power. I wouldn't be surprised if you're still amazed and confused by this fact, but like I tell my students in every class, confusion is the first step to understanding. I'll be happy to talk with anyone who wants to understand this part of the math better....

```
> isolve(e*z+y*N2=1);  #isolve stands for "integer solve",
       #i.e. find integer solutions.  Here the unknowns
```

```
        #are z and y, since we already set e=7
        #and N2=880  We don't really care about the y-value,
        #we just want z.  this is because when we interpret
        #e*z+y*N2=1 in N2 clock arithmetic, N2 is congruent to 0, so
        #so ez=1 mod N2
```

$$\{y = 3 + 7\ \_Z1, z = -377 - 880\ \_Z1\}$$

```
> d:=-377 mod N2;
        #using mod N2 will get d back in usual residue range.
        #We get DAvis' solution from step 7!  For "fun" you could
        #try getting this number from the Euclidean algorithm.
```

$$d := 503$$

3b) Now **Alice** is ready to receive messages. She yells from the rooftops: My modulus is N=943. My encryption power is e=7. If you want to send me a message use the encrypt function:

```
> encrypt:=x->x^e mod N;
    #if you defined e and N above,
    #then their values will be used for the encrypt function
    #this command is Maplese for encrypt(x)=x^e mod N.
    #So this is the syntax for defining
    #elementary functions.
```

$$encrypt := x \to x^e \bmod N$$

Note: In class and in the picture notes I made for you we use the letter E for the encrypt function.

4) The message which **Bob** wishes to send Alice is the letter Y. He consults Davis' table on page 9. The number that corresponds to Y is 35.

```
> M:=35;
```

$$M := 35$$

5) **Bob** encrypts the message using Alice's public key:

```
> C:=encrypt(M);  #either of these should work
  C:=M^e mod N;
```

$$C := 545$$

$$C := 545$$

6) **Bob** sends the number 545 to Alice.

8) **Alice** decodes the message using her decoding power d, which she found in step 7, a while ago.

```
> decrypt:=y->y^d mod N;
```

$$decrypt := y \to y^d \bmod N$$

```
> decrypt(C);  #both of these should work
  C^d mod N;
```

$$35$$

$$35$$

Alice consults the table, sees that 35 corresponds to Y, and understands what Bob has sent. WE DID IT! Except with Alice's puny primes our message pieces can only be one letter long, so what we've got is really no better than a substitution cipher.

Part II:
A more practical size.

In your project everyone will pick primes bigger than 10^50, so that your moduli will be bigger than 10^(100). This is still not big enough to be secure, but you will be able to send messages with up to 50 characters per packet. (And so that decoding doesn't get too tedious for all groups, you will be limited to a total message at most 2 packets.)

For now we will pick primes bigger than 10^6, and use message packets of 6 characters. This means our message packets will have up to 12 digits per packet, which will be less than our modulus N, since N will be greater than 10^12.

```
> restart:   #this will clear all old definitions.
             #It's a good idea to restart when you begin
             #new work - of course you might need to
             #go back and re-enter some old commands that
             #you need again.  (Repetition is good pedagogy.)
> randomize();#this will tell the "random" number generator
             #where to start. the "seed" it generates is based
             #on the system clock, so if you all hit this at
             #the same time you might get the same "random" numbers.
             #That would be bad.  See help windows to see how to
             #make your random numbers unlike your classmates'.

                                  1055994951
> rand();    #random number generator,
             #default range is between 0 and 12 digits
                                  512591693163
> bigger:=rand(1..10^51):  #much bigger
  bigger();
             740824634526537088768083436587070915633696711864653
```

For now,

```
> good:=rand(1..10^7):
  good();
                                  3125820
> for i from 1 to 100 do
    x1:=good():
     if (x1>10^6         #check number is big enough
        and
      isprime(x1)=true)  #and check if number is prime
      then print(x1);     #if it is, let's see it
    fi;
  od:

                                  6755989
                                  4590941
                                  3351683
```

7363409

5332843

It's unlikely your numbers agree with mine. (Well, in truth if you all start the generator at the same place, you'll all get the same so-called random numbers.) But let's all use two of mine. We are repeating the process we worked out in the tiny example.)

```
> p:=6755989;  #I got these with my mouse, by
               #highlighting with left mouse,
               #clicking cursor, and pasting with
               #middle mouse (at least on our system)
  q:=4590941;
  N:=p*q;           #our modulus
```

$$p := 6755989$$

$$q := 4590941$$

$$N := 31016346895649$$

To see that a system of this size is not secure, try the following command. This is the command that would fail if we had chosen primes of length 200 instead of 12, and that's the reason RSA is secure when you use huge primes.

```
> ifactor(N);
```

$$(4590941)\ (6755989)$$

3) Find an encoding power e

```
> N2:=(p-1)*(q-1);
```

$$N2 := 31016335548720$$

```
> for i from 1 to 10 do
     x2:=good();
     if gcd(x2,N2)=1
       then print(x2);
     fi
  od:
```

1683301

2760301

```
> e:=1683301;
  gcd(e,N2);   #check relative prime
```

$$e := 1683301$$

$$1$$

7) Get decoding power

```
> isolve(e*z + y*N2 =1);
     #find decryption power
```

$$\{z = -466433227379 - 31016335548720\ \_Z1,\ y = 25314 + 1683301\ \_Z1\}$$

```
> d:= -466433227379 mod N2;
     #use the mouse to copy and paste big numbers
```

$$d := 30549902321341$$

Technical Point: When we get to step 6, or certainly step 8 Maple will complain when we try to compute large powers of large numbers, so we have to lead it through this modular computation in

compute large powers of large numbers, so we have to lead it through this modular computation in smaller steps. The procedure below does an analgous computation to what Jim did yesterday, and what Davis also outlines, except using powers of 10 rather than powers of 2. It's fine with me if you just use this procedure, but it's even finer if you can understand it. The encryption algorithm makes use of the digit procedure which picks off the coefficients of powers of 10 in the decimal expansion of a number. So make sure to load digit before you load encrypt.

```
> digit:=(x,n)->trunc(x/10^n)-10*trunc(x/10^(n+1));
```

$$digit := (x, n) \to \mathrm{trunc}\left(\frac{x}{10^{n}}\right) - 10\,\mathrm{trunc}\left(\frac{x}{10^{(1+n)}}\right)$$

```
> digit(123.56,-1);
  digit(123.56,2);
  digit(123.56,0);
     #check how digit picks off the digits corresponding
     #to powers of 10
```

5

1

3

```
> encrypt:= proc(M1,E,N3)  #message, encipher power,modulus
                  #we assume all M1's, E's have at most 105 digits
      local i,j,  #indices
            L1,   #list of succesive 10th powers of M1
            ans;  #answer
    L1[1]:=M1  mod N3;
    for i from 2 to 105 do
       L1[i]:=L1[i-1]^10 mod N3;
    od:
    ans:=1:    #initialize answer
    for j from 1 to 105 do
       ans:=ans*(L1[j]^digit(E,j-1)) mod N3;
     od:

  RETURN(ans);
  end:


>
```

Let's check!!!

```
> M:=12345678910;
```

$$M := 12345678910$$

```
> secret:=encrypt(M,e,N);
```

$$secret := 30774808215266$$

```
> encrypt(secret,d,N);
    #decryption is just encryption with a different power
```

12345678910

YES!!!

Part III
An Actual Message

We Use Davis' Table on page 9 to encrypt " **I'm Dizzy**". We will need two chunks

```
> M1:=196749101445;
  M2:=62626163;
```

$$M1 := 196749101445$$

$$M2 := 62626163$$

```
> C1:=encrypt(M1,e,N);
  C2:=encrypt(M2,e,N);
```

$$C1 := 27689001541179$$

$$C2 := 28946550909513$$

```
> encrypt(C1,d,N);
  encrypt(C2,d,N);
```

$$196749101445$$

$$62626163$$

```
>
```