ACCESS 2010 - RSA
Friday June 18

Our Plan Today:
We'll review the punchlines from yesterday's number theory notes, which explain how to find decryption powers from encryption powers, in modular arithmetic. We'll also discuss the outline of RSA encryption and why it's effective for public key cryptography, as described at the end of yesterday's notes. Then to help explain the RSA public-key encryption method in more detail we will do the example worked out in the Tom Davis "Cryptography" notes, page 13-14. The hand-drawn Alice and Bob diagram posted on our ACCESS page illustrates this example too. 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, as is appendix J of "The Code Book." You could also consult Wikipedia.

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 you will implement 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 using Alice and Bob.

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 junior/senior level course, so I would expect that some of what we've talked about this has been challenging. Still, some students 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, that you are pleasantly surprised that this "abstract" mathematics turns out to be so practical, and that you are appreciating that discovery, experiment and deduction have roles in mathematics just as they do in science.

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

We will continue using Maple 10. If you save your work in one session, do some more work in a later higher number version of Maple, and then try to reopen the most recently saved work in Maple 10 you will likely be out of luck - Maple is not backwards compatible. It's possible your file will open if you save in "classic" mode, i.e. as an .mws rather than an .mw file.

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:  #clears all Maple memory - good to do if you start over.
> p:=23;
  q:=41;  # insert prompts [> with the menu bar.
```

```
            # make definitions with :=, NOT with =.
            # comment lines begin with #
            # if you want more than one line of math commands
            # before executing them, use  <shift><return>
```

$$p := 23$$

$$q := 41$$

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 she needs to find 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(N2,e);  #must be 1 for e to have mult. inv. mod N2
```

$$1$$

```
> ifactor(N2);
  ifactor(e);   #could also check gcd with prime factorizations -
  this
                #is what can't be done in reasonable time for huge
                #composite numbers
```

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

$$(7)$$

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 N2=(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 the Euler-Fermat Theorem, the result of which I told you yesterday (but which we didn't have time to prove), this d will be the decoding power. MAGIC!

```
> d:=e^(-1) mod N2;  #you could've done this with the Euclidean
  algorithm!
```

$$d := 503$$

3b) Now **Alice** is ready to receive messages. She yells her public encryption key 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 mode N.
       #so this is the syntax for defining elementary functions.
```

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

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 works for small modulus
  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;
```

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

```
> decrypt(C);
  C^d mod N;  #either of these work for small modulus
```

$$35$$

$$35$$

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^(30), so that your moduli will be bigger than 10^(60). This is still not big enough to be secure, but you will be able to send messages with up to 60 digits (so 30 letter/punctuation symbols) per packet. (And so that decoding doesn’t get too tedious for all groups, you will be limited to a total message at most 3 packets.)

For now we will pick primes bigger than 10^6, and use message packets of 6 characters. This means our message packets can 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 enter this command
  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’.

                                1276804061
> rand();    #random number generator,
             #default range is between 0 and 12 digits
                               239899980800
> bigger:=rand(10^50..10^51):  #random number between 10^50 and
  10^51
  bigger();
         277995068156650062098263213297258027496408487621572
```

For now, (but not later)

```
> good:=rand(10^6..10^7):
  good();
                                 5155748
> for i from 1 to 100 do   #try 100 times to find a right-sized
  prime
      x1:=good():
     if
      isprime(x1)=true  #check if number is prime
      then print(x1);     #if it is, let’s see it
    end if;
  od:

                                 9823279
```

```
                                  1668509
                                  8919527
                                  5313443
                                  8052211
                                  2087179
                                  5602559
                                  7839089
```

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.) You may chose your p and q using your list! We are repeating the process we worked out in the tiny example.)

```
> p:= 9823279;  #I copied and pasted these with my mouse
  q:= 8919527 ;
  N:= p*q ;          #my modulus
                              p := 9823279
                              q := 8919527
                           N := 87619002269033
```

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); #stands for "integer factorization"
                          (8919527) (9823279)
```

3) Find an encoding power e

```
> N2:=(p-1)*(q-1);  #auxillary modulus
                          N2 := 87618983526228
```

```
> #find encryption power which has a multiplicative inverse
  #mod N2:
  for i from 1 to 10 do
     x2:=good();
     if gcd(x2,N2)=1
       then print(x2);
     fi
  od:
                                  1849627
                                  1559315
                                  4254571
                                  5704541
```

```
> e:=3906431;
  gcd(e,N2);   #check relative prime

                              e := 3906431
                                  1
```

7) Get decoding power:

```
> d:=e^(-1) mod N2;
```

```
  d*e mod N2;
```

$$d := 62789144022551$$

$$1$$

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 smaller steps. The procedure below does the trick. It's analagous to method Davis outlines in his notes, except using powers of 10 rather than powers of 2. We've been using similar trickery in class. Here's the idea: Suppose we want to compute

$$[783]^{565} \text{ mod N}$$

in small steps. We write

$$[783]^{565} = [[783]^{500}]\,[[783]^{60}]\,[[783]^{5}] \text{ mod N}$$
$$= [[783]^{100}]^{5}\,[[783]^{10}]^{6}\,[[783]]^{5} \text{ mod N.}$$

By successive multiplication and reduction to the residue value, the procedure below makes a table of the residue values of 783, $[783]^{10}$, $[783]^{100}$, $[783]^{1000}$, etc., i.e of residue values for the powers of 783, where the power is itself a power of 10. We then multiply these table residue values and reduce mod N, the appropriate number of times, as indicated in the decomposition above. Thus we recover the residue value of $[783]^{565}$ without every having to deal with integers which are greater than N^2. (Actually I was sloppy, and my intermediate numbers could get as large as N^{10}, but for our N-values this won't be a problem.)

Here is the procedure, which uses a subprocedure called "digit" to pick off digits from numbers:

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

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

```
> 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
     #this do loop makes the list of powers of M1 described above:
     L1[1]:=M1  mod N3;
     for i from 2 to 105 do
        L1[i]:=L1[i-1]^10 mod N3;
     od:
     #now multiply table entries to get the residue value of the
     #encryption power function:
     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);
```

$secret := 24801259212994$

```
> encrypt(secret,d,N);
     #decryption is just
     #encryption in modular arithmetic, with the decryption power.
```

12345678910

YES!!!

What would happen above if you started with an M which was larger than your modulus N? (It would not be good - try it!)

Part III
An Actual Message

We Use Davis’ Table on page 9 to encrypt " **I’m Dizzy**". We will need two packets to keep our numbers in the residue range.

```
> M1:=196749101445;  #plaintext (well, plainnumber) packets
  M2:=62626163;
```

$M1 := 196749101445$

$M2 := 62626163$

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

$C1 := 21214521232404$

$C2 := 52079494810590$

```
> encrypt(C1,d,N);  #decryption is encryption with
  encrypt(C2,d,N);  #a different power, should give back originals
```

196749101445

62626163

Part IV
Your work!

For your project you will need the digit and encryption procedures - you can either work in this document or create a new window and copy and paste those procedures into it, into a Maple math field. Think about how you want to organize your work and name your various packets as you go through the process of creating your message packet, your "plainnumber" signature, its decryption and the two packets you make by subdividing this decrypted signature. You will be encrypting these various packets with different functions for the two groups you send them to. You will also be receiving messages from these two groups, each of which will need to be decrypted with your decryption function - and then you will need to glue the last two signature packets back together and encrypt them with the senders' functions to recover each groups' signature. This last step is where the thorniest problems show up - although they actually have arisen much earlier in the process.

```
>
```