Activity: RSA Cryptography Lab

Let's implement the ideas behind RSA cryptography to create a public encryption key that you can use to have your friends send you secure messages.

We will use SAGE to compete this activity. You can use the interactive SAGE cells in the online version of this activity. Or use the online SAGE cell at https://sagecell.sagemath.org/. Of course if you have SAGE installed, you can use that.

1. First you will need to select two very large prime numbers. Let's shoot for 20-25 digits. Luckily, SAGE has the command `next_prime()`which will give you the next prime larger than your input. So pick a random input and get two primes. You will want to save these as a constant. For example (but you will need to find a much bigger numbers):

```
p = next_prime(20)
q = next_prime(30)
p, q
```

```
(23, 31)
```

2. Now you can compute $n = pq$ and $m = (p-1)(q-1)$.

```
n = p*q
m = (p-1)*(q-1)
n, m
```

```
(713, 660)
```

Notice that you *could* ask SAGE to find $m = \varphi(n)$ using `euler_phi(n)`, but you should think about why this is a really really really bad idea. If you want, you can try it below, but maybe first with only small values of p and q.

```
euler_phi(n)
```

```
660
```

3. Now we need E and D. Recall that we want $\gcd(E, m) = 1$ and $DE \equiv 1 \pmod{m}$. How are you going to find these?

SAGE has the command `gcd(E, m)`that will compute the gcd of the two inputs. You could try factoring m and looking for a reasonably large value of E, or just guess and check until you find a suitable E.

To find D, there is the command `inverse_mod(E, m)`which will run the Euclidean algorithm forwards and backwards on E and m.

```
# repeatedly pick E until gcd(E,m) = 1.
E = 35
gcd(E,m)
```

```
5
```

```
D = inverse_mod(E,m)
D
```

You can now publish the encryption pair (n, E) so your friends can send you messages. You will need to keep D private. But don't loose it! The whole point is that without p and q, you should not be able to find D again, even if you had n and E. Evaluate the cell below to refer to later:

```
print("Your public key, E =", E,"; n =", n)
print("Your private key, D =", D)
```

If someone has a message x to send you, they would simply need to compute $x^E \pmod{n}$. SAGE can do this with `mod(x^E,n)`, but if x and E are large, this would take a really long time. Luckily, there is a better way: `power_mod(x,E,n)`computes the same thing, but uses the method of repeated squaring to make this much more efficient.

For example:

```
a = mod(12345^35, 713)
b = power_mod(12345,35,713)
a, b
```

```
(470, 470)
```

You will need to use this function to decrypt the message y when you take $y^D \pmod{n}$.

4. When you get an encrypted message, you can assign it to the variable `encrypted`and then decrypt it using the `power_mod`function.

```
encrypted = #paste encrypted message here
decrypted = power_mod(encrypted, D, n)
decrypted
```

5. The last piece of the puzzle is what to do with the number you get out of the decryption. You will find some x, but need that to be translated into text you can read.

This depends on the agreed upon method for translating a string of symbols into numbers. For this lab, you can use the following code do decrypt the message:

```
digits = decrypted.digits(base=128)
letters = [chr(ascii) for ascii in digits]
''.join(letters)
```

Here is why the code above works. First, we need to agree upon how to translate the original message into a number.

Suppose we have a string called `message`. We can create a list of ASCII code values (0 through 127) using `digits = [ord(letter) for letter in message]`. Here `ord()`converts a single letter into its ASCII code, so we do that for each letter in the message.

We must then convert a collection of letters into a single number base 128. SAGE can do this using the `ZZ()`function. So `ZZ(digits,128)`. This will only work if $128^k < n$, where k is the number of digits. So in practice, we would break the longer message into chunks and encrypt each chunk separately.

To undo this coding, you can break down the received message `decrypted`using `digits = decrypted.digits(base=128)`, which makes a list of digits. Then you can create a list of letters using `letters = [chr(ascii) for ascii in digits]`. Finally, put these letters into a string using `''.join(letters)`.

To practice, you can try encoding and decoding messages below. The first set of cells allow you to encrypt and decrypt a very short message. The second pair show a way to break up the message word by word using for loops break up the message into word-long chunks so there is no limit to the length of the message.

```
message = #paste short message here.
digits = [ord(letter) for letter in message]
message_num = ZZ(digits,128)
```

```
encrypted = power_mod(message_num, E, n)
encrypted
```

```
decrypted = power_mod(encrypted, D, n)
digits = decrypted.digits(base=128)
letters = [chr(ascii) for ascii in digits]
''.join(letters)
```

For longer messages:

```
message = #paste message here, enclosed in quotes.
message_array = message.split()
encrypted = []
for word in message_array:
    digits = [ord(letter) for letter in word]
    word_num = ZZ(digits,128)
    encrypted.append(power_mod(word_num, E, n))
encrypted
```

```
dec_message_array = []
for num in encrypted:
    decrypted = power_mod(num, D, n)
    digits = decrypted.digits(base=128)
    letters = [chr(ascii) for ascii in digits]
    dec_message_array.append(''.join(letters))
'␣'.join(dec_message_array)
```

Here is an empty sage cell in case you want to experiment with other commands: