Worksheet 14: RSA

Note. Use SageMath (or another programming language of your choice). If you haven't installed SageMath on your computer, you can use https://sagecell.sagemath.org/. Here are the SageMath functions from Ste17 for implementing RSA. You can to copy all of this into a cell and then type whatever you want below that.

```
# Turn an ASCII string into a number
def encode(s):
    s = str(s)
    return sum(ord(s[i])*256^i for i in range(len(s)))

# Turn the number computed by encode back into an ASCII string
def decode(n):
    n = Integer(n)
    v = []
    while n != 0:
        v.append(chr(n % 256))
        n //= 256
    return ''.join(v)

# Generate RSA key (n, e, d) where n has specified number of bits
def rsa(bits):
    proof = (bits <= 2048)
    p = next_prime(ZZ.random_element(2^(bits//2 + 1)), proof = proof)
    q = next_prime(ZZ.random_element(2^(bits//2 + 1)), proof = proof)
    n = p * q
    phi_n = (p-1) * (q-1)
    while True:
        e = ZZ.random_element(1,phi_n)
        if gcd(e, phi_n) == 1: break
    d = lift(Mod(e, phi_n)^(-1))
    return n, e, d

# Encrypt message m using the public key (n, e)
def encrypt(m, n, e):
    return lift(Mod(m, n)^e)

# Decrypt ciphertext c using the private key (n, d)
def decrypt(c, n, d):
    return lift(Mod(c, n)^d)
```

Problem 1. Suppose that, while generating an RSA key, you generate the following random primes `p` and `q`, and then you generate a random number `e` which is relatively prime to `euler_phi(p*q)`. What is the decryption key `d`?

```
p = 297561419921063301389273320322260639154593171
q = 511844187540655977055223468767529367046228757
e = 107271416246368254897701692590730156398249043504282573751327682231934074852781347462412901
```

Solution. Use `Mod(e, (p-1)*(q-1))^(-1)`

Problem 2. Your friend Mei's RSA public is the pair `(n, e)` below. You would like to send Mei the secret message `s` below. Use the `encode` and `encrypt` functions to generate ciphertext `c` that you will send to Mei.

```
n = 4770378301525942926141852150101059013119688140460541955063774812409564616233682691018070733
e = 2636610137414576625215964196631567915544285772485959001583461461603649962343913820904122813
s = "Laughing will scare them away!"
```

Solution. Use `encrypt(encode(s), n, e)`

Problem 3. The triple `(n, e, d)` below is your RSA key. Your friend No-Face used your public key `(n, e)` and the functions `encode` and `encrypt` functions to send you the following ciphertext `c`. What is No-Face saying to you?

```
n = 568489058778117811667556499906332631885458651545932206253392348024177339755497981163747371
e = 408209244143415913212148723780593256164806411258321164712872131998881752356569657271537853
d = 341682296564960783963552326562843378382811772669984396407794254711960770933253280504831365
c = 455952070763657012170792869142621699572597329183935699111041738152420980748178206413311246
```

Solution. Use `decode(decrypt(c, n, d))`

Problem 4. Find a partner in the class and go through the entire process of exchanging encrypted messages using RSA with them! Start by generating your own keys using the `rsa` function above and exchanging the public keys `(n, e)`. Then encrypt messages for each other to read.

Notes regarding key size. As of 2015, NIST's recommendation for RSA key size is 2048 bits. As of 2020, the largest RSA key that's publicly been cracked had 829 bits. The SageMath RSA functions implemented above impose a size limit on messages based on the key size: a key with 2048 bits can handle messages that are at most 256 characters long. This is an artificial restriction designed to make the `encode` and `decode` functions simpler. Implementations of RSA that are actually used in practice use smarter `encode` and `decode` functions that don't impose a limit on message size (and they also allow you to use any Unicode characters, not just basic ASCII).

Problem 5. Olutosin's RSA public key is the pair `(n, e)` below. Unfortunately for Olutosin, you're a hacker who conducted a side-channel attack on his computer while it was generating his RSA key for him – and through this attack, you discovered the value of `phi_n = euler_phi(n)` given below.

```
n = 2109841151762277968981500567818902588999395539105240682899199498774945452547
e = 6883549815675650662924100424505085897827419038695814592827304156747962525
phi_n = 2109841151762277968981500567818902588907395150720446724841950946997108339696
```

(a) Find Olutosin's private decryption key `d`.

(b) Suppose $n = pq$ for two distinct primes p and q and let f denote the quadratic polynomial $x^2-(n+1-\varphi(n))x+n$. Show that $f = (x-p)(x-q)$.

(c) Find the two prime factors of the number `n` that Olutosin generated. *Note.* By part (b), it is sufficient to find the roots of the quadratic polynomial

```
f = x^2 - (n + 1 - phi_n)x + n
```

and this can be done using the quadratic formula. Here's a function from Ste17 that does exactly this:

```
# Returns prime factors of n when n is a product of two primes and phi_n = euler_phi(n)
def factor(n, phi_n):
    R.<x> = PolynomialRing(QQ)
    f = x^2 - (n + 1 - phi_n)*x + n
    return [b for b, _ in f.roots()]
```

Solution. For (a), use `Mod(e, phi_n)^(-1)`. For (b), we just expand things out:

$$\begin{aligned}
f &= x^2-(n+1-\varphi(n)x+n \\
&= x^2-(pq+1-(p-1)(q-1))x+pq \\
&= x^2-2(p+q)x+pq \\
&= (x-p)(x-q)
\end{aligned}$$

For (c), use `factor(n, phi_n)`.