COMP2700 Lab 11: Public Key Cryptography and RSA

The exercises in this tutorial/lab are adapted from selected exercises from Paar & Pelzl's "Understanding Cryptography" (Chapter 6, 7 & 8).

Exercise 1. Assume a (small) company with 120 employees. A new security policy demands encrypted message exchange with a symmetric cipher. How many distinct keys are required, if you are to ensure a secret communication for every possible pair of communicating parties?

Exercise 2. Calculate $\Phi(m)$ for $m = 12, 15$. You may use software to help you with the calculation of GCD in determining $\Phi(m)$. Here's an example using the math library from python.

```
>>> import math
>>> math.gcd(12,3)
3
>>>
```

Exercise 3. Compute the inverse $a^{-1}\ mod\ n$ with Fermat's Theorem (if applicable) or Euler's Theorem:

a) a=4, n=7
b) a=5, n=12
c) a=6, n=13

Exercise 4. For the affine cipher in Chapter 1 the multiplicative inverse of an element modulo 26 can be found as

$$a^{-1} \equiv a^{11}\ mod\ 26$$

Derive this relationship by using Euler's Theorem.

Exercise 5. Determine the order of all elements of the multiplicative group Z_7^*. Which elements are the primitive elements of the group?

Exercise 6. Let the two primes *p* = 41 and *q* = 17 be given as set-up parameters for RSA. Which of the parameters $e_1 = 32, e_2 = 49$ is a valid RSA exponent? Justify your choice.

Exercise 7. Encrypt and decrypt by means of the RSA algorithm with the following system parameters:

a) $p = 3, q = 11, d = 7$. Determine the public key and encrypt $x = 5$.
b) $p = 5, q = 11, e = 3$. Determine the private key and decrypt $y = 9$.

Exercise 8. Show that the multiplicative property holds for RSA, i.e., show that the product of two ciphertexts is equal to the encryption of the product of the two respective plaintexts.

Exercise 9. For this exercise, we will have a quick look at the PyCryptodome RSA library and use it to perform simple encryption and decryption.

First, we need to generate an RSA key, using the "generate()" function, which takes an argument specifying the length of the modulus. The modulus must be at least 1024 bits. Here's an example of a 1024-bit RSA key generation:

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(1024)
```

Here the "key" variable is actually a tuple consisting of the modulus (n), the public exponent (e), the private exponent (d), the pair of primes used to generate the modulus (p, q). Each of these components can be queried by their names, e.g., for the above generated key, the following would display the modulus n, and the primes p and q used to compute n:

```
>>> key.n
>>> key.p
>>> key.q
>>>
```

We can perform a (naïve) encryption using modular exponentiation with key.e, and its corresponding decryption using modular exponentiation with key.d.

a. Use the PyCryptodome's RSA library to generate an RSA key, and use it to perform a naïve encryption of a "plaintext" number 10. Then decrypt the resulting ciphertext, and confirm that the decrypted number matches the plaintext.
   NOTE: you will want to avoid the python exponentiation operator ** for computing modular exponentiations with large numbers. Use the function pow() instead, e.g., to compute $a^b \ mod \ c$, use **pow(a,b,c)** rather than (a ** b) % c. The function pow is implemented using a fast modular exponentiation algorithm that is better suited for computation involving large numbers.
b. Confirm the malleability of RSA encryption using the key you generated above as follows: Given two numbers a and b

   - First compute the ciphertexts x and y of a and b, respectively.

   - Compute $z = x \times y \ mod \ n$.

   - Then show that z decrypts to $a \times b \ mod \ n$.

NOTE: to avoid the malleability of ciphertexts, in practice, paddings are added to the input prior to encryption. One commonly used padding scheme for RSA is the OEAP (Optimal Asymmetric Encryption Padding). See https://www.rfc-editor.org/rfc/rfc8017 for details.