# M1508 - Number theory and Cryptography Lab

Lab programs

1. Tile-a-Rectangle-with-Squares

Given an n * m grid (where n,mn,m are integers), we would like to tile it with the minimal number of same size squares. Clearly, it can always be tiled with nm squares of size 1 * 1, but it is not always optimal. For example, a 6 * 10 grid can be tiled by 15 squares of size 2 * 2. My goal in this problem is to implement a function squares(n, m) that returns the minimum number of same size squares required to tile a grid of size n * m

Program:

```
def gcd(n,m) :
  if n == 0:
    return m
  return gcd(m % n, n)
def squares(n, m) :
    return n/gcd(n,m) * m/gcd(n,m)
```

2. Diophantine-Equations

We will use the recursive implementation of the Euclidean algorithm to design a function
Is to solve for solving the Diophantine equation ax + by = c. (1) Such equations are solvable if and only if the gcd(a, b) divides c. First note that if b divides a then we can write down a solution: x = 0, y = c/b. If b does not divide a, write a = bq + r as in the (long) divsion algorithm and then substitute into (1): (bq + r)x + by = c. Rearrange as b(qx + y) + rx = c. Set u = qx + y, v = x and substitute again
to obtain the equation bu + rv = c. (2) We have reduced the equation (1) to the equivalent equation (2) with smaller coefficients. If we can solve the new equation, then we recover a solution of the old equation by the formulas x = v, y = u - qv. 11 The process eventually terminates (by the theory of the Euclidean algorithm) with an equaiton where the second coefficient divides the first. These ideas are embodied in the Python code below. Note the use of the divmod operation to compute the quotient and remainder at the same time. The function divmod returns a pair of numbers, the quotient and remainder, which we store in q, r. Note also the use of lists for the return value.

Program:

```
def diophantine(a, b, c):
  assert c % gcd(a, b) == 0
  q, r = divmod(a,b) #divmod operation to compute the quotient and remainder at the same time
  if r == 0:
    return( [0,c/b] )
  else:
    sol = diophantine( b, r, c )
    u = sol[0]
    v = sol[1]
    return( [ v, u - q*v ] )
```

3 Modular Division

Now that we know how to use extended Euclid's algorithm for finding modular inverses, implement an efficient algorithm for dividing b by a modulo n. Given three integers a, b, and n, such that gcd(a,n)=1 and n > 1, the algorithm should return an integer x such that 0 <= x <= n - 1, and b / a = x (modn) (that is, b = a x (modn)).

```
def gcd(a, b):
  assert a >= 0 and b >= 0 and a + b > 0
  while a > 0 and b > 0:
    if a >= b:
      a = a % b
    else:
      b = b % a
  return max(a, b)
def extended_gcd (a, b):
  # assert a >= b and b >= 0 and a + b > 0
  if b == 0:
    d, x, y = a, 1, 0
  else:
    (d, p, q) = extended_gcd (b, a % b)
    x = q
    y = p - q * (a // b)
  assert a % d == 0 and b % d == 0
  assert d == a * x + b * y
  return (d, x, y)
def divide(a, b, n):
  assert n > 1 and a > 0 and gcd(a, n) == 1
  d, t, s = extended_gcd (a, n)
  if t < 0:
    t = t + n
    return t * b % n
  else:
    return t * b % n
  assert x >= 0 and n - 1 >= x
```

4. Chinese Remainder Theorem

Implement the algorithm to construct the number from the Chinese Remainder Theorem. You need to implement the function Chinese Remainder Theorem(n-1, r-1, n-2, r-2) which takes two coprime numbers n-1 and n-2 and the respective remainders 0 <= r-1 < n-1 and 0 <= r-2 < n-2, and must return the number r such that 0 < r < n-1 n-2 <r r=r1modn1 and r=r2modn2. You have access to the function ExtendedEuclid(a, b)ExtendedEuclid(a,b) which returns pair of numbers (x, y)(x,y) such that ax + by = GCD(a, b)ax+by=GCD(a,b).

```
def ChineseRemainderTheorem(n1, r1, n2, r2):
  (x, y) = ExtendedEuclid(n1, n2)                    #We have access to Extended Euclid funciton
  n = x*n1*r2 + y*n2*r1
  return n % (n1 * n2)
```

5. Modular Exponentiation
How to compute $b_e$ mod? There is no need to compute the giant number $b_e$ and divide by m. We can start with 1, then multiply by b and immediately take the result modulo m, repeat e times.

```
def modExponentiation(b, e, m):
 c = 1
 i = 0
 while i < e:
  c = (c * b) % m
  i += 1
 return c
```

6. Fast Modular Exponentiation
(1) Implement the function Fast Modular Exponentiation (b, k, m) which computes ($b_2k$) modm using only around 2k modular multiplications. You are not allowed to use Python built-in exponentiation functions.
(2) Implement the function FastModular Exponentiation (b, e, m) which computes ($b_e$) modm using around 2log2(e) modular multiplications. You are not allowed to use Python built-in exponentiation functions.

```
def FastModularExponentiation_(b, k, m):
  for i in range(k):
    b = (b * b) % m
 # your code here
  return b
def FastModularExponentiation(b, e, m):
   n = ""
   answer = 1
   while e != 0:
      n = n + str(e % 2)
      e = int(e // 2)
   for i in range(len(n)):
      if n[i] == '1':
         answer*=FastModularExponentiation_(b,i,m)
      answer = answer % m
   return answer
# With this code all of the Modular Exponentiation problems could been solved.
print(FastModularExponentiation(7,128,11)
```

7. RSA Quiz: Code Question 1
Implement RSA encryption with the given public key modulo, exponent modulo, exponent.
You have access to the function PowMod(a, n, modulo) which computes anmodmodulo
using the fast modular exponentiation algorithm from the previous module. You also
have access to the function ConvertToInt(message) which converts a text message to
an integer.
You need to fix the implementation of the function Encrypt(message, modulo, exponent) to return the integer ciphertext according to RSA encryption algorithm.

```
import sys, threading
sys.setrecursionlimit(10**7)
threading.stack_size(2**27)
def ConvertToInt(message_str):  # Converts message to integers
  res = 0
  for i in range(len(message_str)):
    res = res * 256 + ord(message_str[i])
  return res
def PowMod(a, n, mod): # Converts message(int), exponent, public key to ciphertext
    if n == 0:
        return 1 % mod
    elif n == 1:
        return a % mod
    else:
        b = PowMod(a, n // 2, mod)
        b = b * b % mod
        if n % 2 == 0:
          return b
        else:
          return b * a % mod
```

RSA Quiz: Code Question 3
Secret agent Alice has sent one of the following messages to the center: attack don’t attack wait
Alice has ciphered her message using public key modulo, exponent that is available to you, and you have intercepted her ciphertext. You want to know what was the content of her message. You have access to the function Encrypt(message, modulo, exponent) which takes in a message as a string and returns a big integer as a ciphertext. It uses RSA encryption with public key modulo, exponent. In the starter code, you have an example usage of the function Encrypt. You also have function DecipherSimple
(ciphertext, modulo, exponent, potential-messages) implemented in the starter code. You need to fix this implementation to solve the problem. It should take the ciphertext sent from Alice to the center, the public key modulo, exponent and the set of potential messages that Alice could have sent, and return the message that Alice encrypted and sent as a string. For example, if Alice took message "wait", encrypted it with the given modulo and exponent, and got number 139763215 as the ciphertext, you will need to return the string "wait" given the ciphertext = 139763215, modulo, exponent and potential-messages ="attack","don’t attack","wait

```
def PowMod(a, n, mod):
    if n == 0:
        return 1 % mod
    elif n == 1:
        return a % mod
    else:
        b = PowMod(a, n // 2, mod)
        b = b * b % mod
        if n % 2 == 0:
```

```
     return b
    else:
     return b * a % mod
def ConvertToInt(message_str):
 res = 0
 for i in range(len(message_str)):
  res = res * 256 + ord(message_str[i])
 return res
def Encrypt(message, modulo, exponent):
 # Substitute this implementation with your code from question 1 of the "RSA Quiz".
 return PowMod(ConvertToInt(message), exponent, modulo)
def DecipherSimple(ciphertext, modulo, exponent, potential_messages): #Main function to attack
the ciphertext
 # Fix this implementation
 for i in range(len(potential_messages)):
  if ciphertext == Encrypt(potential_messages[i], modulo, exponent):
   return potential_messages[i]
 return "don't know"
modulo = 101
exponent = 12
ciphertext = Encrypt("attack", modulo, exponent)
print(ciphertext)
print(DecipherSimple(ciphertext, modulo, exponent, ["attack", "don't attack", "wait"]))
```