1845479-HW05

Carmignani Federico

December 8, 2021

Abstract

The goal of this homework is to provide a working code implementing Shamir Secret Sharing.
The solution has to be sent to cns@diag.uniroma1.it from the institutional email.
The deadline is December 8^{th} at 11:59 pm.

1 What is the challenge

The challenge consists in implementing the Shamir Secret Sharing protocol, in which the goal is to share a secret S with n subjects by consigning some data (fragment) to each of them.
The important thing is that none of them has to know S, they can reconstruct S by joining the fragments they hold. In particular, they can retrieve it just having $k \leq n$ fragments, but not just having less than k fragments. Assuming S as an integer number and a pair (k,n), with $1 < k \leq n$, it is needed to find n data fragments such that the reconstruction is possible only having at least k fragments. The ingredients used are the mod arithmetic and finite fields, but also the polynomial interpolation.

2 How to solve the challenge

2.1 Construction of the fragments

In order to solve the challenge, two are the tasks to be implemented: the construction of the fragments and the reconstruction of the secret. The construction of the fragments is done choosing first a prime number p, greater than S and n, then randomly are chosen k-1 integers in [0,p): a1, a2, a3, a4, a5, ...,ak-1, let be a0 as S. Consider a polynomial P(x) with degree k-1 and these coefficients, s_i = P(i), for i=1, 2, 3, ..., n. So, by construction, P(0)=S.
The fragments are the pairs (i, s_i mod(p)).

2.2 Reconstruction of the secret

The reconstruction is done given k fragments and applying the Lagrange formula to find the degree k-1 polynomial going through these points. Using the Lagrange formula in x=0, it will be possible to retrieve the secret S, since S= L(0), where L is the polynomial given by the Lagrange formula (Figure 1).

3 How to use and run the program

The program performs what described before automatically. It has to be executed using Spider or another IDE to run the program or also in the terminal using Python interpreter with this command: 'python 1845479-HW05.py'.
The secret is never stored more than needed in the program. The user has to insert the parameters n, k, the secret S; whereas the prime number is chosen as the first prime number greater than n and S. The comments in the code describe what each function does. For the construction phase there are functions to handle the calculation of the value of the polynomial in a certain point, to draw the polynomial and to generate the coefficients and the fragments; instead, the reconstruction phase is

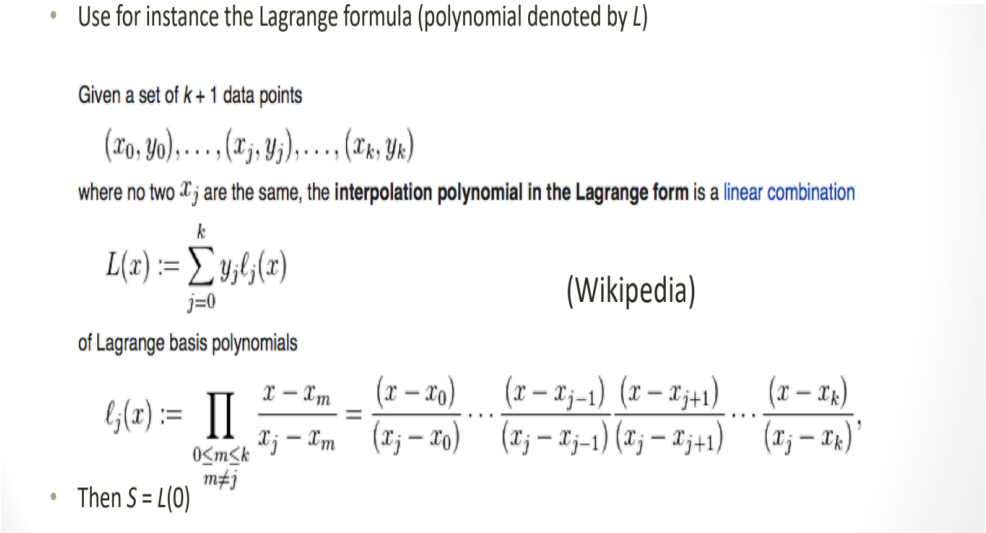

Figure 1: Lagrange formula to retrieve the secret.

done using the Lagrange formula, in mod(p), with multiplicative inverse operation, and calculating the secret as L(0).

3.1 The code

```
#libraries
import random
import numpy as np
import matplotlib.pyplot as plt

#1)construction of the fragments

#evaluating the polynomial in a point i, using mod(p) operation
def poly_with_value(coefficients,i,mod):
    cnt = 0
    for coeff in (coefficients):
        cnt *= i
        cnt+= coeff
        cnt %= mod
    return cnt

#drawing the polynomial
def draw(coefficients):
    x = np.linspace(-20, 20, num=100)
    fx = []
    for j in range(len(x)):
        res = 0
        #calculating in some points the polynomial
        for el in coefficients:
            res = x[j] * res + el
        fx.append(res)
    #giving x and f(x)
    plt.plot(x,fx)
    plt.grid()
    plt.axvline()
    plt.axhline()
```

```
    plt.show()
    return

#generating the coefficients of the polynomial in [0.p)
def create_coefficients(n, k, secret):
    big = max(n,secret)    #p>S, p>n
    p = big + 1
    #generation of p, prime number
    cnt = 0
    found = 0
    while cnt == 0:
        for i in range(2, p-1):
            if p % i == 0:
                found = 1
                break
        if (found == 1):
            p += 1
            found = 0
        else:
            cnt += 1
    print("\nPrime number P: ",p)

    coefficients = []
    for j in range(0, k-1):
        #appending k-1 integers as coefficients
        coefficients.append(random.randrange(0, p))
    coefficients.append(secret)
    return [coefficients,p]

#generating the fragments
def create_fragments(couple, secret):
    k = couple[0]
    n = couple[1]
    c= create_coefficients(n, k, secret)
    coefficients = c[0]
    prime = c[1]
    print(f'\nCoefficients: {coefficients}')
    draw(coefficients)
    #generating the shares as (i,s(i)), with i from 1 to n,
    #and the s(i) as the polynomial calculated in i
    shares = []
    for j in range(1,n+1):
        shares.append((j, poly_with_value(coefficients, j, prime)))
    generated = [shares, prime]
    shares = generated[0]
    print(f'\nShares: {", ".join(str(share) for share in shares)}')
    return [shares, generated[1]]

################################################################################

#2)reconstruction of the secret

#multiplicative inverse to be consistent with the mod(p) operation
def multiplicative_inverse(num, p):
    #operation taken from:
    #http://en.wikipedia.org/wiki/Modular_multiplicative_inverse#Computation
```

```
    x = 0
    last_x = 1
    y = 1
    last_y = 0
    while p != 0:
        quot = num // p
        num, p = p, num % p
        x, last_x = last_x - quot * x, x
        y, last_y = last_y - quot * y, y
    return last_x

#function to recover the secret from a set of fragments
def retrieve_secret(fragments,p):
    #split fragment into two list: ordinates and abscissas
    x=[]
    y=[]
    for i in range(len(fragments)):
        x.append(fragments[i][0])
        y.append(fragments[i][1])
    k=len(x) #k is the number of fragments
    #calculating the list of numerators and denominators (Lagrange formula)
    nums=[]
    dens=[]
    #calculating all factors of Lagrange formula with x=0 in mod(p)
    ll_0=[]
    for i in range(k):
        num=1
        den=1
        l=1
        for j in range(k):
            if(i!=j):
                num*=(0-x[j])
                den*=(x[i]-x[j])
        nums.append(num)
        dens.append(den)
        #need multiplicative inverse to be consisten with the generation in mod(p)
        inv_den=multiplicative_inverse(den,p)
        l=(num*inv_den)%p
        ll_0.append(l)
    #linear interpolation: calculate the polynomial in 0
    poly_in_0=0
    for i in range(k):
        #adding the factors previously computed
        poly_in_0+=(ll_0[i]*y[i]) % p
    return poly_in_0 % p

################################################################################

#main
k = int(input("K: "))
n = int(input("N: "))

#construction of fragments
ret = create_fragments([k,n] , int(input("Secret: ")))
shares = ret[0]
prime = ret[1]
```

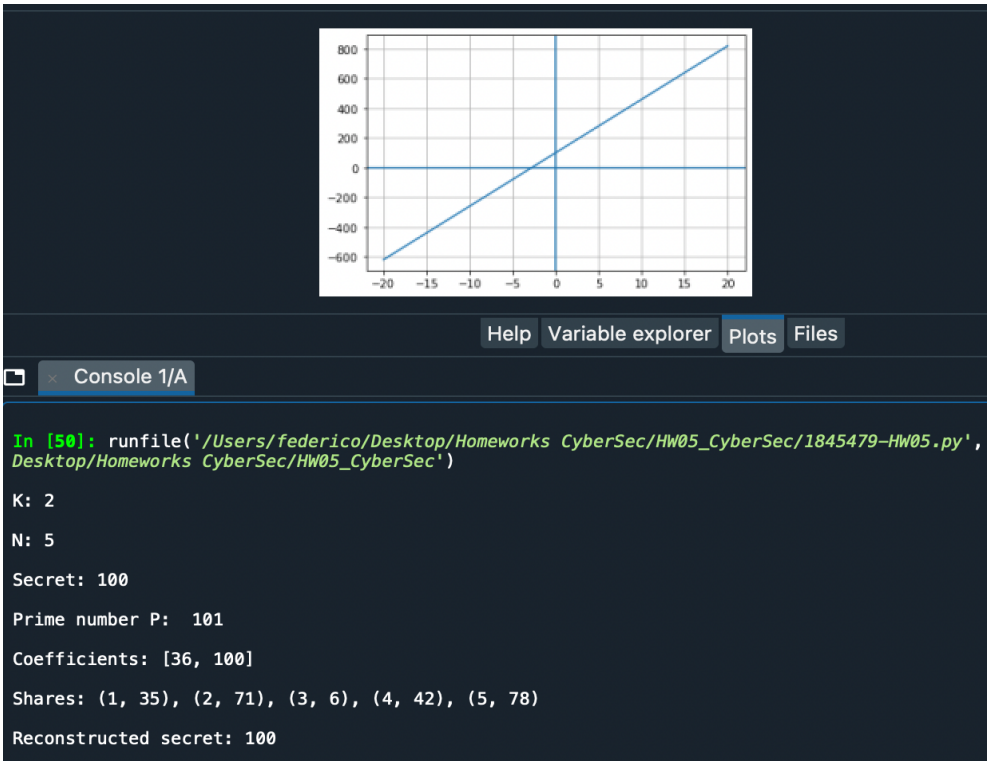

Figure 2: Demo 1.

```
#reconstructing secret from h>=k shares (the first k shares for example)
print(f'\nReconstructed secret: {retrieve_secret(shares[:k], prime)}')
```

3.2 The program in use

The execution of the program with k=2, n=5 and secret=100 is shown in Figure 2, including the drawing of the polynomial in the plots' space. In the program, the reconstruction is done using the first k fragments generated, but it could be done with any other k fragments.