Python – Number Theory

R. R. Maiti

GCD

- 7/3 = 2.333
- 7//3 = 2
- def gcd(a,b):
    - If (b ==0):
        - Return(a)
    - Else:
        - return(gcd(b, a%b))
- gcd(409119243, 87780243)
    - 6837
- Find ϕ(26)
    - phi = lambda n: [i for i in range(1,n) if gcd(i,n)== 1]
    - print(phi(26))
- Find inverse(x) mod 26
    - y is inverse of x iff (xy -1) %26 =0
    - inverse = lambda x: [y for y in range(1,26) if (x*y - 1)% 26 == 0]

GCD

```
def gcd(a, b):
    # Return the GCD of a and b using Euclid's Algorithm
    while a != 0:
        a, b = b % a, a
    return b
```

```
def findModInverse(a, m):
# Returns the modular inverse of a % m, which is
# the number x such that a*x % m = 1
    if gcd(a, m) != 1:
        return None # no mod inverse if a & m aren't relatively prime
    # Calculate using the Extended Euclidean Algorithm:
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3 # // is the integer division operator
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return u1 % m
```

package

- import cryptomath
- >>> cryptomath.gcd(24, 32)
- 8
- >>> cryptomath.gcd(37, 41)
- 1
- >>> cryptomath.findModInverse(7, 26)
- 15
- >>> cryptomath.findModInverse(8953851, 26)
- 17

Primitive root and discrete logarithm

- Z_n, n = 26
- Find prime factors of n
- Compute(a,i,n): # a^i mod n
- isPrimitiveRoot(i,n)#powers of i mod n must generate all numbers in Z_n
- Find $dlog_{r,p}(n)$# ans is y such that n = r^ymod p

Prime number (hacking cipher)

- Find the number of prime numbers in every hundred from 1, 100, 200, 300, 400, 500, … , 1000
- 10^{100} – a Googol- find nearest prime number to a googol
- A typical prime number used in our RSA program will have hundreds of digits
- Is this number prime?
  - 112,829,754,900,439,506,175,719,191,782,841,802,172,556,768,253,593,054,977,186,2355,84,979,780,304,652,423,405,148,425,447,063,090,165,759,070,742,102,132,335,103,295,947,000,718,386,333,756,395,799,633,478,227,612,244,071,875,721,006,813,307,628,061,280,861,610,153,485,352,017,238,548,269,452,852,733,818,231,045,171,038,838,387,845,888,589,411,762,622,041,204,120,706,150,518,465,720,862,068,595,814,264,819

```
def rabinMiller(num):
    # Returns True if num is a prime number. 9. 10.
    s = num - 1
    t = 0
    while s % 2 == 0:
        # keep halving s while it is even (and use t
        # to count how many times we halve s)
        s = s // 2
        t += 1
    for trials in range(5):
        # try to falsify num's primality 5 times
        a = random.randrange(2, num - 1)
        v = pow(a, s, num)
        if v != 1:
            # this test does not apply if v is 1.
            i = 0
            while v != (num - 1):
                if i == t - 1:
                    return False
                else:
                    i = i + 1
                    v = (v ** 2) % num
    return True
```

```
def isPrime(num):
	# Return True if num is a prime number. This function does a quicker, prime number check before calling rabinMiller().

	if (num < 2):
		return False
	# 0, 1, and negative numbers are not prime , About 1/3 of the time we can quickly determine if num is not prime
	# by dividing by the first few dozen prime numbers. rabinMiller() is not guaranteed to prove that a number is prime.
	lowPrimes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109,
113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727,
733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887,
907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]
	if num in lowPrimes:
		return True # See if any of the low prime numbers can divide num
	for prime in lowPrimes:
		if (num % prime == 0):
			return False
		# If all else fails, call rabinMiller() to determine if num is a prime.
	return rabinMiller(num)
```

```
def generateLargePrime(keysize=1024):
# Return a random prime number of keysize bits in size.
    while True:
        num = random.randrange(2**(keysize-1), 2**(keysize))
        if isPrime(num):
            return num
```

Concepts on Chinese Remainder Theorem

- If you know prime factorization of n, then you can use something called CRT to solve a whole system of equations
- Let the prime factorization of n is p1, p2, …, pt, then
- The system of equations
  - x mod pi = ai where I = 1,2, …, t
- Has unique solution, x, where x is less than n
- Example
  - Primes p1 = 3, p2 = 5,
  - x= 14 = (2,4)
  - x=13 = (1,3)
  - x=12 = (0,2)
  - …..
  - x=1 = (1,1)

Class Assignment 1: CRT

- Write a program for CRT which can solve a set of linear equations with modular arithmetic. Consider following set of equations
  - x = a1 mod p1
  - x = a2 mod p2
  - x = a3 mod p3
  - x = a4 mod p4
- Write three functions: i) for computing gcd(m,n) ii) for computing mod-inverse(m,n) #n is modulus iii) for computing CRT with four well defined steps.

Concept on primitive root Z_n

- If *a* is a primitive of n, then its powers
  - a, a^2, a^3, …, a^phi(n) are distinct (mod n) and relatively prime to n
- Example
  - n = 19
  - Phi(n) = ?
  - 19-1 = 18
  - Ever element in Z_{19} is coprime to 19
  - But, not all of these are primitive roots
- Check for each item and see if powering of an item from 1 to 18 generates all the elements in Z_{19}
- Candidates for primitive root of 19 in Z_{19} are 2,3,10,13,14,15

a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}	a^{17}	a^{18}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
3	9	8	5	15	7	2	6	18	16	10	11	14	4	12	17	13	1
4	16	7	9	17	11	6	5	1	4	16	7	9	17	11	6	5	1
5	6	11	17	9	7	16	4	1	5	6	11	17	9	7	16	4	1
6	17	7	4	5	11	9	16	1	6	17	7	4	5	11	9	16	1
7	11	1	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1
8	7	18	11	12	1	8	7	18	11	12	1	8	7	18	11	12	1
9	5	7	6	16	11	4	17	1	9	5	7	6	16	11	4	17	1
10	5	12	6	3	11	15	17	18	9	14	7	13	16	8	4	2	1
11	7	1	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1
12	11	18	7	8	1	12	11	18	7	8	1	12	11	18	7	8	1
13	17	12	4	14	11	10	16	18	6	2	7	15	5	8	9	3	1
14	6	8	17	10	7	3	4	18	5	13	11	2	9	12	16	15	1
15	16	12	9	2	11	13	5	18	4	3	7	10	17	8	6	14	1
16	9	11	5	4	7	17	6	1	16	9	11	5	4	7	17	6	1
17	4	11	16	6	7	5	9	1	17	4	11	16	6	7	5	9	1
18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1

Discrete logarithm

(a) Discrete logarithms to the base 2, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Ind_{2,19}(a)$	18	1	13	2	16	14	6	3	8	17	12	15	5	7	11	4	10	9

(b) Discrete logarithms to the base 3, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Ind_{3,19}(a)$	18	7	1	14	4	8	6	3	2	11	12	15	17	13	5	10	16	9

(c) Discrete logarithms to the base 10, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Ind_{10,19}(a)$	18	17	5	16	2	4	12	15	10	1	6	3	13	11	7	14	8	9

(d) Discrete logarithms to the base 13, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Ind_{13,19}(a)$	18	11	17	4	14	10	12	15	16	7	6	3	1	5	13	8	2	9

(e) Discrete logarithms to the base 14, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Ind_{14,19}(a)$	18	13	7	8	10	2	6	3	14	5	12	15	11	1	17	16	14	9

(f) Discrete logarithms to the base 15, modulo 19

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$Ind_{15,19}(a)$	18	5	11	10	8	16	12	15	4	13	6	3	7	17	1	2	12	9

1 ≡ 2^x mod 19 ⇒ x = 18

2 ≡ 2^x mod 19 ⇒ x = 1

3 ≡ 2^x mod 19 ⇒ x = 13

4 ≡ 2^x mod 19 ⇒ x = 2

5 ≡ 2^x mod 19 ⇒ x = 16

■

■

18 ≡ 2^x mod 19 ⇒ x = 9

Class Assignment 2: power-mod

- Write function for computing y = x^i mod n using the idea discussed in class.
- Write two functions: i) for converting decimal to binary ii) for computing power-mod

Home assignment 1

- Implement Extended Euclidean algorithm for computing mod-inverse
- You should have one function named ex-Euclidean-mod-inverse(…)