Intro to Cryptography

Mark Anderson
Homework 4

April 20, 2019

1. Write a Python program that given n, computes the size of S_n, the prime factorization of n!, and the number of factors of n!.

```
#!/usr/bin/env python3
import math
# <summary>
# @summary:
#  - Sieve or Eratosthenes as shown in class
# @params: n
#  - integer of which we want to calculate all primes < n
# </summary>
def sieve(n):
    if n < 2:
        return []

    s = list(range(n+1))
    s[0] = s[1] = -1
    for i in range(2, n + 1):
        if s[i] > 0:
            for k in range(2 * i, n + 1, i):
                s[k] = -1

    os = []
    for i in range(2, n+1):
        if s[i] > 0:
            os.append(i)
    return os

# <summary>
# Takes in an integer input, and calculates the prime factorization of the
    input factorial
# @params:
#  - n: is the integer of which we want to calculate the prime
    factorization of factorial (n!)
# @return:
```

```
#  - result: this is a string representation of the prime factorization of
    n! for example (9! = (2**7)(3**4)(5**1)(7**1))
#  - calcVal: this is the value of calculating our prime factorization of
    n!, in the example of 9! this is the result of
#           2^7 * 3^4 * 5^1 * 7^1, this value is used to compare to the
    math.factorial() value from python to ensure
#           we calculated the correct roots
def primeFactorization(n):
    #our return values
    result = ""
    calcVal = 1
    #All primes < our input n, (our prime factors)
    primes = sieve(n)
    #A List representing the future exponents that will correspond to our
        prime factors
    # e.g:
    #   n = 6
    #   primes        = [2, 3, 5]
    #   primeExponents = [4, 2, 1]
    #   which corresponds to (2**4)(3**2)(5**1)
    primeExponents = [0 for _ in range(len(primes))]

    #loop through all of our prime numbers to determine their exponents
    for i, num in enumerate(primes):
        exp = 1
        #the first computed exponent determines how many times this factor
            shows up in n!
        first = math.floor(n / (num**exp))
        #add the exponent to the corresponding position in the exponents list
        primeExponents[i] += first
        while(exp < first):
            exp += 1
            val = math.floor(n / (num**exp))
            #add the computed value to the corresponding position in the
                exponents list
            primeExponents[i] += val

    #This is used to format the results we calculated above into the
        specified format in the homework
    #as well as calculating the result of n! from our factored primes to
        compare to math.factorial() later
    for i in range(0, len(primes)):
        calcVal *= primes[i]**primeExponents[i]
        result += "(" + str(primes[i]) + "**" + str(primeExponents[i]) + ")"

    return result, calcVal
# <summary>
```

```
# Takes in an integer input and calculatres the primeFactorization of that
    number factorial, then compares it to the result
# of that computation and pythons math.factorial(), prints out the number
    of test cases passed
# @params:
#  - maxnum: the amount of test cases / highest value integer we would like
    to compute the prime factorization of
def test(maxNum):
    numPassed = 0
    for i in range(1,maxNum):
        resString, resVal = primeFactorization(i)
        print("primeFactorization(%d) %s" % (i, resString))
        factorRes = math.factorial(i)
        if(factorRes == resVal):
            numPassed+=1
    print("%d out of %d cases passed" % (numPassed, maxNum - 1))

def main():
    NUM_CASES = 100
    test(NUM_CASES)

if __name__ == "__main__":
    main()
```

Some results from this function for $1 \leq n \leq 100$, many results omitted to save space

---

```
primeFactorization(2) (2**1)
primeFactorization(3) (2**1)(3**1)
primeFactorization(4) (2**3)(3**1)
primeFactorization(5) (2**3)(3**1)(5**1)
primeFactorization(6) (2**4)(3**2)(5**1)
primeFactorization(7) (2**4)(3**2)(5**1)(7**1)
primeFactorization(8) (2**7)(3**2)(5**1)(7**1)
primeFactorization(9) (2**7)(3**4)(5**1)(7**1)
primeFactorization(10) (2**8)(3**4)(5**2)(7**1)
primeFactorization(11) (2**8)(3**4)(5**2)(7**1)(11**1)
.
.
primeFactorization(36)
    (2**34)(3**17)(5**8)(7**5)(11**3)(13**2)(17**2)(19**1)(23**1)(29**1)(31**1)
primeFactorization(37)
    (2**34)(3**17)(5**8)(7**5)(11**3)(13**2)(17**2)(19**1)(23**1)(29**1)(31**1)(37**1)
primeFactorization(38)
    (2**35)(3**17)(5**8)(7**5)(11**3)(13**2)(17**2)(19**2)(23**1)(29**1)(31**1)(37**1)
.
.
imeFactorization(98)
    (2**95)(3**46)(5**22)(7**16)(11**8)(13**7)(17**5)(19**5)(23**4)(29**3)(31**3)(37**2)
```

```
    (41**2)(43**2)(47**2)(53**1)(59**1)(61**1)(67**1)(71**1)(73**1)(79**1)(83**1)(89**1)
    (97**1)
primeFactorization(99)
    (2**95)(3**48)(5**22)(7**16)(11**9)(13**7)(17**5)(19**5)(23**4)(29**3)(31**3)(37**2)
    (41**2)(43**2)(47**2)(53**1)(59**1)(61**1)(67**1)(71**1)(73**1)(79**1)(83**1)(89**1)
    (97**1)
99 out of 99 cases passed
```

---

2. Compute the inverse of x^7+x+1 in the finite field $F_{2^8} = \frac{F_2}{x^8+x^4+x^3+x+1}$ Since we are given that F_{2^8} is a finite field, we know that $x^8+x^4+x^3+x+1$ must be irreducible in the field. Using this, we want to find a polynomial g^{-1} such that $q(x) * f(x) + g(x) * r(x) = 1$ where $g(x) = x^7 + x + 1$ and $f(x) = x^8 + x^4 + x^3 + x + 1$ and $r(x) \in F_2[x]$ so using repeated division with exetended euclidean algorithm we find our polynomials $r(x) = x^6 + x^2 + x + 1$ and $q(x) = x^7$. So the inverse of our $g(x)(modf) = x^7(modf)$

3. Prove that a and b must be consecutive Fibonacci numbers if all the quotients of the GCD(a,b) are equal to 1. Let a,b be coprime with eachother, then the $GCD(a, b) = 1*b+(b-a)$ because $a > b, b-a$ is a negative number, resulting in a lower value than the first, take another iteration of $GCD(b, b-a) = 1*b+b-(b-a)$. This sequence defines the fibonacci in reverse, where the remainder of each iteration of the GCD algorithm with the 2 coprime elements, is the larger element (a) + (the smaller(b) - the larger(a)). Using the definiton that $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ we can show this using the extended euclidean algorithm. $GCD(F_n, F_{n-1}) = 1 * F_{n-1} + (F_{n-1} - F_n)$, this result can be proven to be F_{n-2} from the formula above, $F_{n-1} + F_{n-2} = F_n$ subtract over F_{n-1} to get $F_{n-2} = F_{n-1} - F_n$ which results in our new GCD being $GCD(F_{n-1}, F_{n-2})$ which defines the fibonacci sequence as stated above.

4. Prove the converse, that if a and b are consecutive fibonacci numbers then the quotients are all equal to 1. (each consecutive element of the sequence is coprime with eachother). We will work in reverse from above, using the first 2 elements of the fibonacci sequence $[gcd(F_1, F_2) = 1] == [gcd(1, 2) = 1 * 1 + 1 = 2]$ we must show that the $gcd(F_{n+1}, F_{n+2}) = 1 \forall n$ terms in the fibonacci sequence. from the formula above, we know that the $F_n = F_{n-1} + F_{n-2}$ so we can write $gcd(F_{n+1}, F_{n+2})$ as $gcd([F_{n+1-1} - F_{n+1-2}], [F_{n+2-1} + F_{n+2-2}]) == gcd(F_{n+1}, F_n) = 1$

5. Let p be a prime. Show that if n is odd and has k distinct prime factors then the number of solutions of $x^2 = 1(modn)$ is equal to 2^k.

   Let k = the number of distinct prime factors of our odd number n. The prime factors of n are all relatively prime with each other because $gcd(p_1, p_2) = 1$ for any 2 prime numbers. Because our factors are relatively prime with each other, we can use the Chinese Remainder Theorem to express the equation $x^2 \equiv 1(modn)$ as $x^2 \equiv 1(modn_1 * n_2 * n_k)$ where n_k is the kth prime factor of n. This can be further broken down into k seperate equations $x^2 \equiv 1(modn_1)$, $x^2 \equiv 1(modn_2)$, and $x^2 \equiv 1(modn_k)$ where each of these are guaranteed 1 solution $(n_k^{k-1})^2 \equiv 1(modn_k)$ by Fermat's little theorem, and because each of these solutions is squared from our x being square, they have 2

solutions each, $-\sqrt{n_k} and \sqrt{n_k}$. So each distinct factor k has 2 solutions, resulting in $x^2 \equiv 1(mod n_1 * n_2 * n_k)$ having 2^k solutions, and we proved earlier that this equation is equal to our original n.

6. Let p be a prime. Show that g is a generator of $F_p^* \iff g^{p-1} = 1(modp)$ and $g^q \neq 1(modp) \forall$ divors q of $p-1$ We are given that p is a prime, and by Fermat's Little Theorem we know that $g^{p-1} \equiv 1(modp)\ \forall a \neq 0(modp)$ and since we know $g^q \neq 1(modp) \forall$ divisors q of p - 1. We know that each divisor g^q is coprime with $p-1$ meaning $g^q is a generator \forall gcd(q, p-1) = 1$ and we know that all q divisors are coprime meaning that each divisor coprime with p-1 is a generator because the group of any finite field f_{p-1} is a cyclic group with $\phi(p-1)$ generators.

7. Determine a primitive element for each of the finite fields F_{2^n} for $1 \leq n \leq 8$

   - n = 2: primitive element = $x^2 + x + 1$
   - n = 3: primitive element = $x^3 + x + 1$
   - n = 4: primitive element = $(x^2 + x + 1)^2 == x^4 + x^2 + 1$
   - n = 5: primitive element = $x^5 + x^2 + 1$
   - n = 6: primitive element = $x^6 + x^5 + x^2 + x + 1$
   - n = 7: primitive element = $x^7 + x^3 + 1$
   - n = 8: primitive element = $x^8 + x^4 + x^3 + x + 1$

8. Write a python program that computes the Legendre symbol and the Jacobi Symbol for any legal input.

```
#!/usr/bin/env python3
import math
from p_factor import sieve
# <summary>
# Computes the gcd of 2 given integers
# @params:
#   - a,b are 2 integers
# @return:
#   - returns the GCD of the two integers a and b
def gcd(a,b):
    while b != 0:
        (a, b) = (b, a % b)
    return a


# <summary>
# Computes the quadratic residue of 2 given integers a and p
# @params:
#   - a and p are two integers with p assumed to be prime and odd
# @return:
```

```
#   - returns 1 if the two integers form a quadratic residue, or -1 if they
    form a quadratic
#     non-residue
#   - 0 if the two numbers are divisible
def quadratic_residue(a, p):
    if a % p == 0:
        return 0
    for x in range(0,p - 1):
        if (x ** 2) % p == a:
            return 1
    return -1


# <summary>
# Wrapper for the quad_residue function, takes in the same params and
    returns the legendre_symbol
def legendre_symbol(a, p):
    qR = quadratic_residue(a,p)
    return qR


# <summary>
# A function to determine if a given number is prime, loops through the
    range of the number
# only returning False if the number has been divided evenly, returns True
    otherwise
# @params:
#   - num is an integer > 2
# @return:
#   - True/False whether or not the number is a prime
def isPrime(num):
    for i in range(2, num):
        if num % i == 0:
            return False

    return True

# <summary>
# A function that prints the Legendre_symbol table from wikipedia to easily
    check whether
# the computed legendre symbols are correct
# @params:
#   - symbol_table is a 2d list corresponding to the (a | p) values
    respectively for the legedre symbol
def print_table(symbol_table):
    for elem in range(1,31):
        if elem < 10:
```

```
            print(str(elem) + " | ", end="")
        else:
            print(str(elem) + " | ", end="")

    print()
    for row in symbol_table:
        for elem in row:
            if elem is not -1:
                print(str(elem) + " | ", end="")
            else:
                print(str(elem) + " | ", end="")
        print()


# <summary>
# Function to compute the same test cases as given on the wikipedia to
    determine if the legendre_symbol is functioning
# correctly, The given range for legendre(a | p) is a = 30, p = 127
def testLegendre():
    symbol_table = [[0 for _ in range(30)] for _ in range(30)]
    for a in range(1, 31):
        c=-1
        for p in range(3,128):
            if isPrime(p) and (p % 2 == 1):
                c+=1
                if(gcd(a,p)) == 1:
                    symbol = legendre_symbol(a,p)
                    symbol_table[c][a-1] = symbol
                else:
                    symbol_table[c][a-1] = 0
    print_table(symbol_table)

def main():
    testLegendre()

if __name__ == "__main__":
    main()
```

The resulting Legendre symbol table as computed by the testLegendre() function, some results omitted for size constraints

```
    1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
3| 1  | -1 | 0  | -1 | -1 | 0  | -1 | -1 | 0  | 0  |
5| 1  | -1 | -1 | 1  | 0  | -1 | -1 | -1 | -1 | 0  |
7| 1  | 1  | -1 | 1  | -1 | -1 | 0  | -1 | -1 | 0  |
11| 1  | -1 | 1  | 1  | 1  | -1 | -1 | -1 | 1  | 0  |
13| 1  | -1 | 1  | 1  | -1 | -1 | -1 | -1 | 1  | 0  |
```

```
17| 1  | 1  | -1 | 1 | -1 | -1 | -1 | 1 | 1 | 0 |
19| 1  | -1 | -1 | 1 | 1  | 1  | 1  | -1| 1 | 0 |
23| 1  | 1  | 1  | 1 | -1 | 1  | -1 | 1 | 1 | 0 |
29| 1  | -1 | -1 | 1 | 1  | 1  | 1  | -1| 1 | 0 |
31| 1  | 1  | -1 | 1 | 1  | -1 | 1  | 1 | 1 | 0 |
37| 1  | -1 | 1  | 1 | -1 | -1 | 1  | -1| 1 | 0 |
41| 1  | 1  | -1 | 1 | 1  | -1 | -1 | 1 | 1 | 0 |
43| 1  | -1 | -1 | 1 | -1 | 1  | -1 | -1| 1 | 0 |
47| 1  | 1  | 1  | 1 | -1 | 1  | 1  | 1 | 1 | 0 |
53| 1  | -1 | -1 | 1 | -1 | 1  | 1  | -1| 1 | 0 |
59| 1  | -1 | 1  | 1 | 1  | -1 | 1  | -1| 1 | 0 |
61| 1  | -1 | 1  | 1 | 1  | -1 | -1 | -1| 1 | 0 |
67| 1  | -1 | -1 | 1 | -1 | 1  | -1 | -1| 1 | 0 |
71| 1  | 1  | 1  | 1 | 1  | 1  | -1 | 1 | 1 | 0 |
73| 1  | 1  | 1  | 1 | -1 | 1  | -1 | 1 | 1 | 0 |
79| 1  | 1  | -1 | 1 | 1  | -1 | -1 | 1 | 1 | 0 |
83| 1  | -1 | 1  | 1 | -1 | -1 | 1  | -1| 1 | 0 |
89| 1  | 1  | -1 | 1 | 1  | -1 | -1 | 1 | 1 | 0 |
97| 1  | 1  | 1  | 1 | -1 | 1  | -1 | 1 | 1 | 0 |
101| 1 | -1 | -1 | 1 | 1 | 1  | -1 | -1| 1 | 0 |
103| 1 | 1  | -1 | 1 | -1 | -1 | 1  | 1 | 1 | 0 |
107| 1 | -1 | 1  | 1 | -1 | -1 | -1 | -1| 1 | 0 |
109| 1 | -1 | 1  | 1 | 1  | -1 | 1  | -1| 1 | 0 |
113| 1 | 1  | -1 | 1 | -1 | -1 | 1  | 1 | 1 | 0 |
127| 1 | 1  | -1 | 1 | -1 | -1 | -1 | 1 | 1 | 0 |
```

Program for calculating the Jacobi symbol of any two given inputs, some functions from the legendre program are reused, but omitted due to space

---

```
#!/usr/bin/env python3
import math
from collections import defaultdict
#calculate the prime factors for any given number using integer division
def prime_factors(n):
    pFac = defaultdict(int)
    factor = 2
    while n > 1:
        while n % factor == 0:
            pFac[factor] += 1
            n //= factor
        factor += 1

    return dict(pFac)


def jacobi(a, n):
    #grab the prime factors of the denominator
```

```
    pFacs = prime_factors(n)
    symbol = 1
    #for each prime factor and prime factor exponent pair calculate the
        product of their
    #legendre symbols
    for key, val in pFacs.items():
        #multiple the resulting symbol for that prime factor
        symbol *= legendre_symbol(a, key)
        #raise the result of that symbol to the corresponding power of the
            prime factor
        symbol = symbol**val
    return symbol


#Tests the jacobi symbols from a = 1..10 and p = 1.60 where p is an odd
    integer
def testJacobi():
    for a in range(1, 10):
        for p in range(1, 60):
            if p % 2 == 1:
                if gcd(a,p) == 1:
                    print("Jacobi (%d | %d): %d" % (a, p, jacobi(a,p)))
                else:
                    print("Jacobi (%d | %d): %d" % (a, p, 0))


def main():
    testJacobi()

if __name__ == "__main__":
    main()
```

Some results from the jacobi test suite, many omitted due to space

```
   Jacobi (1 | 1): 1
  Jacobi (1 | 3): 1
  Jacobi (1 | 5): 1
  jacobi (3 | 25): 1
  Jacobi (3 | 27): 0
  Jacobi (3 | 29): -1
  Jacobi (3 | 33): 0
  Jacobi (3 | 35): 1
  Jacobi (3 | 37): 1
  Jacobi (3 | 39): 0
  Jacobi (3 | 41): -1
  Jacobi (3 | 43): -1
  acobi (4 | 47): 1
  Jacobi (4 | 49): 1
```

```
Jacobi (4 | 53): 1
Jacobi (4 | 55): 1
Jacobi (9 | 9): 0
Jacobi (9 | 11): 1
Jacobi (9 | 13): 1
Jacobi (9 | 15): 0
Jacobi (9 | 17): 1
Jacobi (9 | 19): 1
acobi (9 | 49): 1
Jacobi (9 | 51): 0
Jacobi (9 | 53): 1
Jacobi (9 | 57): 0
Jacobi (9 | 59): 1
```

---

9. Compute the following Jacobi Symbols by hand $\binom{311}{653}$ and $\binom{666}{777}$

   - $\binom{311}{653}$ we can take $653 mod 311 = 31$ and write that as $\binom{31}{311}$ because both the numerator and the denominator are odd we can apply the law of quadratic reciprocity (rule 6) and write it as $\binom{311}{31}$ but because $311 \equiv 3(mod 4)$ and $31 \equiv 3(mod 4)$ we write it as $-\binom{311}{31}$ from following the rules of quadratic reciprocity (rule 6). We can then apply rule 2 again because $311 \equiv 1(mod 31)$ so we write $-\binom{1}{31}$ and $\binom{1}{31} = -(1)$ so our Jacobi symbol is -1.
   - $\binom{666}{777}$ we can take the fact that the numerator is even and apply $\binom{2m}{n} = \binom{m}{n}$ if $n \equiv +-1 mod 8$ to get $\binom{333}{777}$ and then apply rule 2 $777 mod 333 = 111$ to get $\binom{111}{333}$ and apply quadratic reciprocity and rule 2 again to achieve $\binom{0}{111}$ which is equal to 0.