CS 5602 S19 HW 04

Name: Ganesh Sapkota

Student ID: 12584026

Due December 12, 2021 at 11:59 PM

Q.N.1 Write a Python program that given n, computes the size of Sn,the prime factorization of n!, and the number of factors of n!. Output the result in the form (p1 ∗ ∗e1)(p2 ∗ ∗e2)...(pk ∗ ∗ek) where pi are the prime factors of n!. Use the Sieve of Eratosthenes in your code to gure out the prime factors. For example, if n = 4, S4 has 24 elements and the prime factorization should be outputted in the form (2**3)(3**1). Include values for n = 1,...,10 and n = 100.

Steps to solve this problem.

1. Ask n from user.
2. Compute n! which is the size of S_n
3. Compute the prime factors less than n! using Sieve of Eratosthenes Method.
4. Compute the prime factors of n!
5. Print output in the form (P1**e1)(p2**e2)...(pk**ek) where p1 is the prime factor and e is the repeated number of p.

Step 2. Compute n! which is the size of S_n

```
#Step2: Compute n! for given n which computes the size of $S_{n}$
def compute_fact(n):
  if n==0:
    return 1
  fact = n*compute_fact(n-1)
  return fact
```

Above program computes the factorial of a given number n

```
#test code:
n = 10
f = compute_fact(n)
print("Factorial of {} is {}".format(n,f))
```

```
Factorial of 10 is 3628800
```

Step 3. Compute the prime factors less than n! using Sieve of Eratosthenes Method.

```
#compute prime factors of n!
#Sieve of Eratosthenes to find all primes <= n
#Reference: lecture note CS5602Lec21ResiduesFieldExtensions.pdf

def Sieve(f):
  primes = []
  remain = list(range(2,f+1))
  while remain:
    primes.append(remain[0])
    remain = [remain[i] for i in range(1,len(remain)) if 0 != remain[i]%remain[0]]
  return primes
```

Above function generates the prime factors less than or equal to a given number using Sieve of Eratosthenes.

```
#test program
# n is given number
#f is factorial of n
n = 4
f = compute_fact(n)
print(Sieve(f))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Step 4. Compute the prime factors of n!

Below is the function to print all prime factors of the factorial of given number n

```
import math
def get_Prime_Factors(n):
  # n is given number

  # f = n! --> is factorial of n
  f = compute_fact(n)

  #pf is a list of prime factors of f
  pf = []

  # Print the number of two's that divide f
  while f % 2 == 0:
    # print(2)
    pf.append(2)
    f = f / 2

  # if f is odd, skip by 2 ( k = k + 2)
  for k in range(3,int(math.sqrt(f))+1,2):

    # while i divides f , print k and progress
    while f % k == 0:
      # print(k)
      pf.append(k)
      f = f / k
  #if f is a prime number greater than 2
  if f > 2:
    pf.append(int(f))
  return pf




#Program to test above function
n = 5
pf = get_Prime_Factors(n)
print("Prime factors of {}! are:{}".format(n,pf))
```

```
     Prime factors of 5! are:[2, 2, 2, 3, 5]
```

Step 5. Print output in the form (P1**e1)(p2e2)...(pk**ek) where p1 is

```
# format the result in desired type
# eg: (p1 * *e1)(p2 * *e2)...(pk * *ek)
# where p is prime factor and e is number of p in prime factors list
def formatted_output(pf,n):
  #pf is list of prime factors
  # dict -> count the number of same prime factors and store in dict
  dict = {x:pf.count(x) for x in pf}
  # print(dict)

  output = []
  for i in dict:
    # print("({}**{})".format(i,d[i]))
    f = "(" + str(i) + "**" + str(dict[i]) +")"
    output.append(f)
  # print("\nOutput in list format\n",output)
  print("Prime factorization of {}! is:\n".format(n))
  for j in output:
      print(j, end="")
```

Above function generates the output in desired format. It express the a given number as the product of their prime factors that are raised to some powers.

```
#test program
formatted_output(pf,5)
```

```
     Prime factorization of 5! is:

     (2**3)(3**1)(5**1)
```

Tetst list contains number from 1 to 10 and 100

```
t_num = [1,2,3,4,5,6,7,8,9,100]
```

```
for i in t_num:
  # fact = compute_fact(i)
  pf = get_Prime_Factors(i)
  # print(pf)
  if not pf:
    print("{} is not a prime number".format(i))
  else:
    formatted_output(pf,i)
  print("\n")
```

```
# print(fact)


   1 is not a prime number


   Prime factorization of 2! is:

   (2**1)

   Prime factorization of 3! is:

   (2**1)(3**1)

   Prime factorization of 4! is:

   (2**3)(3**1)

   Prime factorization of 5! is:

   (2**3)(3**1)(5**1)

   Prime factorization of 6! is:

   (2**4)(3**2)(5**1)

   Prime factorization of 7! is:

   (2**4)(3**2)(5**1)(7**1)

   Prime factorization of 8! is:

   (2**7)(3**2)(5**1)(7**1)

   Prime factorization of 9! is:

   (2**7)(3**4)(5**1)(7**1)

   Prime factorization of 100! is:

   (2**474)(3**3)(17**1)(4168434910597**1)
```

Q.N 2

For a prime number P and a positive integer n there is an irreducible $\Pi(x)$ of degree n in $F_P[X]$ and $F_P[X]$ / $\Pi(x)$ is a field of order P^n.

Given,

$F_{2^8} = F_2[X]/(x^8 + x^4 + x^3 + X + 1)$

$F_{256} = F_{2^8} = F_{P^n}$

where p = 2 and n = 8

Field F_{2^8} is the finite field of order 2^8 = 256 which means field has 256 finite elements.

we need to find the multiplicative inverse of $x^7 + x + 1$ in finite field F_{2^8} with irreducible polynomial$(x^8 + x^4 + x^3 + x + 1)$

We can find the multiplicative inverse using Extended Euclidean algorithm and Bezout's identity in following steps:

1) Compute GCD(g(x),(px)) and check if it is equal to 1:

> if GCD(g(x),(px)) = 1 then g(x) and p(x) are relatively prime and multiplicative inverse of p(x) exists in F_{2^8}

2) Use Bezout's identity to find the multiplicative inverse.

> if 'a' is relatively prime to n and GCD(a,n) = 1

> then a and and n also satisfy following condition for some X and Y: X*a* + *Y*n = 1

$Y * n$ Mod n = 0

$X = a^{-1}$ which is multiplicative inverse of a modulo n.

So, we need to find X using Bezout Identity.

For out case Bezout identity is:

$a(x) * g(x) + b(x) * p(x) = r(x)$

where r(x) is the GCD(p(x),g(x)) = 1

$a(x).g(x) + b(x).p(x) = 1$

Now we have,

$g(x) = (x8 + x4 + x3 + x + 1)$

and

$p(x) = (x7 + x + 1)$

Step 1: Computing GCD using Extended Euclidean algorithm

$(x8 + x4 + x3 + x + 1) = (x7 + x + 1)(x) + (x4 + x3 + x2 + 1)$ 1

$(x7 + x + 1) = (x4 + x3 + x2 + 1)(x3 + x2 + 1) + (x)$ 2

$(x4 + x3 + x2 + 1) = (x3 + x2 + x)(x) + 1$ 3

It shows that Reminder is 1 which means GCD(g(x),p(x)) = 1 Hence there is an multiplicative inverse of p(x) exists.

We have Bexout's identity:

$a(x)g(x) + b(x)p(x) = d(x)$

where d(x) is the GCD(g(x),p(x)) = 1

Now Lets start from eqn 3 that has non zero reminder

$() = (x^3 + x^2 + x)(x) + 1$

we know that in GF(2) (1 = -1)

So,

$(x^4 + x^3 + x^2 + 1) + (x^3 + x^2 + x)(x) = 1$ 4

and from eqn(2),

we can get,

x = $(x7+x+1) + (x^4+x^3+x^2+1)(x3+x2+1)

x = p(x) + (x^4+x^3+x^2+1)(x3+x2+1) 5

substituting value of x in eqn 4,

we get,

$(x^4 + x^3 + x^2 + 1) + (x^3 + x^2 + x)[p(x) + (x^4 + x^3 + x^2 + 1)(x^3 + x^2 + x)] = 1$

$(x^4 + x^3 + x^2 + 1) + (x^3 + x^2 + x)p(x) + (x^6 + x^2 + 1)(x^4 + x^3 + x^2 + 1) = 1$

$(x^3 + x^2 + x)p(x) + (x^6 + x^2 + 1)(x^4 + x^3 + x^2 + 1) = 1$ 6

from eqn 1,

$x^4 + x^3 + x^2 + 1 = (x8 + x4 + x3 + x + 1) + (x^7 + x + 1)(x)$

$x^4 + x^3 + x^2 + 1 = g(x) + p(x)(x)$ 7

Substituting $x^4 + x^3 + x^2 + 1$ in eqn 6

we get, $(x^3 + x^2 + x)p(x) + (x^6 + x^2 + 1)(g(x) + p(x)(x)) = 1$

$(x^3 + x^2 + x)p(x) + (x^7 + x^3 + x^2 + x)p(x) + (x^6 + x^2 + x + 1)g(x) = 1$

$(x^7)p(x) + (x^6 + x^2 + x)g(x) = 1$ 8

Comparing it with Bezout's identity

$X * a + Y * n = 1$

$X = a^{-1} = x^7$

Hence the inverse of $a = p(x) = x^7 + x + 1$ is x^7

Q.N 3

Fibonacci numbers are defined as $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

Generate Fibonacci sequence upto n

```
def fibo_seq(n):
  f1 = 0
  f2 = 1
  count = -1
  sequence = []
  if n <= 0:
    print("Enter positive integer greater than 0")
  elif n == 1:
    return f1
  else:
    while count <= n:
        if f1 <= n:
            sequence.append(int(f1))
        nxt = f1 + f2
       # update values
        f1 = f2
        f2 = nxt
        count += 1
  return sequence


#test code
fibo_seq(3)
```

```
     [0, 1, 1, 2, 3]
```

```
#test program to generate Fibonacci Sequence upto n
# n is the upper limit of fibonacci sequence
n = 3
n1 = 5
n2 = 6
n3 = 10
print(fibo_seq(n))
print(fibo_seq(n1))
print(fibo_seq(n2))
print(fibo_seq(n3))
```

```
[0, 1, 1, 2, 3]
[0, 1, 1, 2, 3, 5]
[0, 1, 1, 2, 3, 5]
[0, 1, 1, 2, 3, 5, 8]
```

fibo(n) generate Fibonacci Sequence <= n

fibo_seq(5) gives sequence upto 5. ie: [0,1,1,2,3,5]

fibo(6) also gives sequence [0,1,1,2,3,5] since it is not a fibonacci number.

fibo(10) gives sequence [0, 1, 1, 2, 3, 5, 8]

Extended GCD using Euclidean algorithm

```
def eGCD(a, b):
    if a < b:
    #swap a and b
        return eGCD(b, a)
    # print()
    while b != 0:
        (a, b) = (b, a % b)
    return a
```

The above function computes the gcd of two numbers using Extended Eucledean Algorithm

```
#test Program
a,b = 3,17
gcd = eGCD(a,b)
print("GCD of {} and {} is {}".format(a,b,gcd))
```

```
GCD of 3 and 17 is 1
```

Extended GCD calculation Showing all Steps

```
def compute_eGCD(a, b):
  if a < b:
  #swap a and b
    return compute_eGCD(b, a)
  # print()
  c = 1
  while b != 1:
    # if flag: print('{} = {} * {} + {} ----- q{} = {}'.format(a, a//b, b, a % b,c,a//b))
    print('{} = {} * {} + {} -----> q{} = {}'.format(a, a//b, b, a % b,c,a//b))
```

```
    (a, b) = (b, a % b)
    c +=1
```

The above function computes the Extended GCD and shows each steps of the calculation with quotients (q_i) in each step.

Lets select two random pair of a and b. a,b = 7,3 and a,b = 8,5

```
#Test 1:
# for a= 17 and b = 5
a,b = 15,4
compute_eGCD(a,b)
```

```
     15 = 3 * 4 + 3 -----> q1 = 3
     4 = 1 * 3 + 1 -----> q2 = 1
```

In this case all quotient are not equal to 1. ie. q1 = 3 and q2 = 1. This is not the pair of numbers we are interested in.

```
#Test 2:
#for a = 8, b = 5
a,b = 8,5
compute_eGCD(a,b)
```

```
     8 = 1 * 5 + 3 -----> q1 = 1
     5 = 1 * 3 + 2 -----> q2 = 1
     3 = 1 * 2 + 1 -----> q3 = 1
```

In above calculation of Extended GCD of a and b (a>b), we can see that quotient is equal to 1 in each step. This is the pair of numbers that we are interested in. Now let's check if a and b are consecutive fibonacci number using the function **is_consecutive_fibo(a,b)**

(I) Prove two integers a and b are consecutive fibonacci numbers

```
#check if nummbers are consecutive fibo
def is_consecutive_fibo(a,b):
  #get max and min of two numbers
  large = max(a,b)
  small = min(a,b)

  #generate fibonacci sequence upto largest number
  f_seq = fibo_seq(large)
  print("Fibonacci Sequence:",f_seq)
```

```
  #if numbers are within the sequence then get index of both numbers
  if large in f_seq:
    ind_l =  f_seq.index(large)
    if small in f_seq:
      ind_s = f_seq.index(small)
      #check if the index of large and small number differ by 1
      #to verify numbers are consecutive
      if (ind_l - ind_s) == 1:
        print("{} and {} are consecutive Fibonacci Numbers".format(small,large))
      else:
        # print(1)
        print("{} and {} are not consecutive Fibonacci Numbers.".format(small,large))

    else:
      # if small number is not in fibonacci sequence
      # print(2)
      print("{} is not Fibonacci Number".format(small))

  else:
    #if large number is not in fibonacci sequence
    # print(3)
    print("{} is not Fibonacci Number".format(large))
```

```
#Test program
#a,b (a>b) are two random numbers chosen to commpute extended gcd
#whose all quotients are found to be equal to 1
a,b = 8,5
is_consecutive_fibo(a,b)
```

```
    Fibonacci Sequence: [0, 1, 1, 2, 3, 5, 8]
    5 and 8 are consecutive Fibonacci Numbers
```

Above test program shows that a and b are consecutive Fibonacci numbers. Where a and b any two integers greater than 0 whose all the quotients (q_i) are equal to 1, when we compute their extended GCD.

(II) Converse: Prove that the quotients are all equal to 1 if a and b are consecutive Fibonacci numbers

Consider a test fibonacci sequence below 10: [0,1,1,2,3,5,8,13,21]

Let (3,2),(13,8),(21,13) be three pairs of consecutive fibonacci numbers. Let's test if thes numbers has all the quotients 1 while calculating extended GCD.

```
#Test1 for (1,0)
a,b = 3,2
compute_eGCD(a,b)

     3 = 1 * 2 + 1 -----> q1 = 1


#Test3 for 13,8
a,b = 13,8
compute_eGCD(a,b)

     13 = 1 * 8 + 5 -----> q1 = 1
     8 = 1 * 5 + 3 -----> q2 = 1
     5 = 1 * 3 + 2 -----> q3 = 1
     3 = 1 * 2 + 1 -----> q4 = 1


#Test3 for 21,13
a,b = 21,13
compute_eGCD(a,b)

     21 = 1 * 13 + 8 -----> q1 = 1
     13 = 1 * 8 + 5 -----> q2 = 1
     8 = 1 * 5 + 3 -----> q3 = 1
     5 = 1 * 3 + 2 -----> q4 = 1
     3 = 1 * 2 + 1 -----> q5 = 1
```

All three test cases shows that quotients are equal to 1 in each steps while calculating extended GCD.So it proves that the quotients are all equal to 1 if a and b are consecutive Fibonacci numbers.

Qn. N 4.

Solving Strategy

Step1: Take input list of of odd numbers upto n.

Step 2: Find list of odd numbers n that has k - distinct prime factors.

step 3: Find the number of distinct prime factors (k) that a number has

step 4: Find the number of solutions (m) for $x^2 = 1(mod n)$

Step 5: Check if $m = 2^k$

Step 6: if Step 5 is true then Proved.

Step1: Take input list of of odd numbers upto n.

```
# Step1: Take input list of of odd numbers upto n
def get_odd(n):
  odd = []
  for i in range(0,n):
    if i%2 != 0:
      odd.append(i)
  return odd


#test code
odd = get_odd(10)
print(odd)
```

```
[1, 3, 5, 7, 9]
```

Step 2: Find odd numbers n that has k - distinct prime factors.

```
# Step 2: Find odd numbers n that has k - distinct prime factors.

import math
def get_Prime_Factors(f):
  # n is given number

  # f = n! --> is factorial of n
  # f = compute_fact(n)

  #pf is a list of prime factors of f
  pf = []

  # Print the number of two's that divide f
  while f % 2 == 0:
    # print(2)
    pf.append(2)
    f = f / 2

  # if f is odd, skip by 2 ( k = k + 2)
  for k in range(3,int(math.sqrt(f))+1,2):

    # while i divides f , print k and progress
    while f % k == 0:
      # print(k)
      pf.append(k)
      f = f / k
  #if f is a prime number greater than 2
  if f > 2:
```

```
    pf.append(int(f))
  return pf

odd = get_odd(100)
print(odd)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45,
```

Step 3: Find the number of distinct prime factors (k) that a number has

```
#Find the odd numbers having k distinct prime factors than n
def distinct_prime_factors(odd_list):
  for i in odd:
    pf = get_Prime_Factors(i)
    pf = remove_dup(pf)
    size = len(pf)
    if len(pf)>1:
      print("{}-->{} has {} distinct prime factors".format(i,pf,size))



#removes duplicate prime factors and retuns new list
def remove_dup(pf):
  sNew = []
  for prime in pf:
      if prime not in sNew:
          sNew.append(prime)

  return sNew


# test code
# odd: list of prime odd numbers less than or equal to n
n = 100
odd = get_odd(n)

distinct_prime_factors(odd)
```

```
15-->[3, 5] has 2 distinct prime factors
21-->[3, 7] has 2 distinct prime factors
33-->[3, 11] has 2 distinct prime factors
35-->[5, 7] has 2 distinct prime factors
39-->[3, 13] has 2 distinct prime factors
45-->[3, 5] has 2 distinct prime factors
51-->[3, 17] has 2 distinct prime factors
55-->[5, 11] has 2 distinct prime factors
57-->[3, 19] has 2 distinct prime factors
63-->[3, 7] has 2 distinct prime factors
```

```
     65-->[5, 13] has 2 distinct prime factors
     69-->[3, 23] has 2 distinct prime factors
     75-->[3, 5] has 2 distinct prime factors
     77-->[7, 11] has 2 distinct prime factors
     85-->[5, 17] has 2 distinct prime factors
     87-->[3, 29] has 2 distinct prime factors
     91-->[7, 13] has 2 distinct prime factors
     93-->[3, 31] has 2 distinct prime factors
     95-->[5, 19] has 2 distinct prime factors
     99-->[3, 11] has 2 distinct prime factors
```

```
def size_of_pf(o):
  # for i in odd:
    pf = get_Prime_Factors(o)
    pf = remove_dup(pf)
    k = len(pf)
    if len(pf)>1:
      print("{} has k = {} distinct prime factors".format(o,k))
    # return k


size_of_pf(57)
```

```
     57 has k = 2 distinct prime factors
```

Step 4: Find the number of solutions (m) for $x^2 = 1$ (mod n)

Suppose we have list of odd numbers that are not prime and has k distinct prime factors.

```
n = 100
odd = get_odd(n)
distinct_prime_factors(odd)
```

```
     15-->[3, 5] has 2 distinct prime factors
     21-->[3, 7] has 2 distinct prime factors
     33-->[3, 11] has 2 distinct prime factors
     35-->[5, 7] has 2 distinct prime factors
     39-->[3, 13] has 2 distinct prime factors
     45-->[3, 5] has 2 distinct prime factors
     51-->[3, 17] has 2 distinct prime factors
     55-->[5, 11] has 2 distinct prime factors
     57-->[3, 19] has 2 distinct prime factors
     63-->[3, 7] has 2 distinct prime factors
     65-->[5, 13] has 2 distinct prime factors
     69-->[3, 23] has 2 distinct prime factors
     75-->[3, 5] has 2 distinct prime factors
     77-->[7, 11] has 2 distinct prime factors
     85-->[5, 17] has 2 distinct prime factors
     87-->[3, 29] has 2 distinct prime factors
     91-->[7, 13] has 2 distinct prime factors
```

```
93-->[3, 31] has 2 distinct prime factors
95-->[5, 19] has 2 distinct prime factors
99-->[3, 11] has 2 distinct prime factors
```

Lets take any two odd numbers which are odd but has k distince prime factors: 39, 91.

```
o1 = 39
o2 = 75
size_of_pf(o1)
size_of_pf(o2)
```

```
39 has k = 2 distinct prime factors
75 has k = 2 distinct prime factors
```

So, 39 has 2 prime factors: 3,13 and 75 has 2 prime factors 3, 5

Now Solving $x^2 = 1$ Mod 39

we have,

$x^2 - 1$ = 0 Mod 3

and

$x^2 - 1$ = 0 Mod 13

Solving $x^2 - 1$ = 0 Mod 3 we get,

x = 1 mod 3

and

x = -1 mod 3

or, x = 2 mod 3

so, x = 1,2

Again Solving $x^2 - 1$ = 0 Mod 13 we get,

x = 1 mod 13

x = -1 mod 13

or x = 12 (mod 13)

so x = 1,12

So, there are total 4 solution of x for $x^2 = 1$ (Mod 39) which is $2^k = 2^2 = 4$.

$x^2 = 1$ Mod 75

we have

$x^2 = 1$ Mod 3

and

$x^2 = 1$ Mod 5

for $x^2 = 1$ Mod 3 we have 2 solutions:

1 and 2

and for $x^2 = 1$ Mod 5 we have 2 solutions:

1 and 4

So, $x^2 = 1$ Mod 75 has total of 4 solution

Step 5:

so, Number of solutions of $x^2 = 1$ (mod n)are m = 4

and n has k = 2 distinct prime factors.

So, $2^k = 2^2 = 4$

Thus,

m = 2^k

Hence, If n is odd and has k distinct prime factors then the number of solutions of $x^2 = 1$ (mod n) is equal to 2^k

QN 5

For arithmetic modulo n, Let Z_n be the set of reminders/residues. where n is any integer. so Z_n ={0,1,2,3,4...n-1}}

if n is prime number then Z_p is a finite field F_p

Also Z_p^* represent the field whose elements are co-prime of p

i.e Z_p^* = {a ϵ Z_p : gcd(a,p) = 1}

if p = 7 then,

$Z_7^* = F_7^*$ = {1,2,3,4,5,6}

Generator of Finite Field

Generator g of a finite field is an element/member of a group (G,*) which can represent all the elements in the group with its power.

From the definition of Generator, Following python program checks which elements in the GF(7) are generator.

```
# Generate (Z7,+) and check if the element is a generator.
for i in range(1,7):
  print("for i = {}".format(i))
  print([(i**x)%7 for x in range(1,13)])
```

```
for i = 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
for i = 2
[2, 4, 1, 2, 4, 1, 2, 4, 1, 2, 4, 1]
for i = 3
[3, 2, 6, 4, 5, 1, 3, 2, 6, 4, 5, 1]
for i = 4
[4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1]
for i = 5
[5, 4, 6, 2, 3, 1, 5, 4, 6, 2, 3, 1]
for i = 6
[6, 1, 6, 1, 6, 1, 6, 1, 6, 1, 6, 1]
```

From the above output it can be seen that only 3 and 5 are able to represent all the members in the group using their powers upto 10. So 3 and 5 are generater of the group Z7.

If 3 and 5 are generators they should also satisfy the condition $g^{p-1} = 1$ (mod p) and g^q != 1 (mod p) for all prime divisors q of p-1

So Lets check if 3 and 5 satisfy the above conditions.

p = 7

p-1 = 6

q = primer factors of p-1 = {2,3}

For 3:

> 1. Checking if $g^{p-1} = 1$ (mod p)
>
> > $3^6 = 729 = 1$ (mod 7) which is TRUE

and

> 2.Checking if g^q != 1 (mod p) is True:
>
> > $3^2 = 9 = 2! = 1$ (mod 7) which is True
> >
> > and

> > $3^3 = 27 = 6! = 1$ (mod 7) which is also True

Hence 3 is a generator.

For 5:

> 1. Checking if $g^{p-1} = 1$ (mod p)

> > $5^6 = 15625 = 1$ (mod 7) which is TRUE

and

> 2.Checking if g^q != 1 (mod p) is TRUE:

> > $5^2 = 25 = 4! = 1$ (mod 7) which is TRUE

> > and

> > $5^3 = 125 = 6! = 1$ (mod 7) which is also TRUE

5 Satisfies all two conditions. Hence 5 is a generator.

Now,let's check the condition for an element 2 which is not a generator.

> > 2^6 = 64 = 1 mod 7 which is true

and

> > 2^2 = 4 != 1 mod 7 which is TRUE

> > 2^3 = 8 = 1 mod 7 which is FALSE

So 2 does not satisfies both condition since it is not a generator.

Hence it is proved that if and only if $g^{p-1} = 1$ (mod p) and g^q != 1 (mod p) for all prime divisors q of p-1, then g is a generator.

Q.N 6

Field $GF(p^n)^* = F_{p^n}^* = GF(p^n)$ – {0} is a cyclic group under multiplication, and the generators of this group are considered as primitive elements of the field.

Order of Element in the cyclic sub group

Order of a modulo n is the smallest positive integer m such that $a^m \cong 1$ (mod n)

```
#return the order of an element
#cycle: is an list of cyclic subgroup
def get_order(cycle):
  for i in cycle:
    if i==1:
      order = cycle.index(i) + 1
      return order
```

Above function returns the order of field element by checking the index of 1 in the cyclic subgroup of generated from the power of that element.

```
#test program
# cyclic sub group generated from the power of 3 in GF(2^n)
cy = [3, 1, 3, 1, 3, 1]
get_order(cy)
```

```
2
```

Generate cyclic-Sub group using the power of element mod 2^n and check if the elements are generators of the group. Then generate the list of Primitive elements if they exist in the group.

```
#range of n in GF(2^n): 1<=n<=8
def cyclic_subgroup(start,end):
  for n in range(start,end):
    a = 2**n
    print("----------------------------------------")
    print("for GF(2^{})".format(n))
    print("\nElements are:{}".format([i for i in range(a)]))
    # print([i for i in range(n)])
    print("----------------------------------------")
    # i represents elements in field f
    #gen is the list of generaters present in the group
    gen = []
    for i in range(1,a):
      print("for i = {}".format(i))
      #cycle is the list of cyclic sub-group
      #x represent the power of an element upto 20
      cycle = [(i*x)%a for x in range(1,20)]
      # order: order of element
      order = get_order(cycle)
      print(cycle)
```

```
      # phi: phi gives number of totatives from Euler's totient
      # phi = n-1
      phi = a-1
      if(order == phi):
        gen.append(i)
        print("\n{} is a primitive element".format(i))
    # if (len(gen)!= 0):
    #   print("generators are:",gen)
    # else:
    #   print("There are no generators in this group")

#range of n in GF(2^n): 1<=n<=8
start,end = 1,8
cyclic_subgroup(start,end)
```

```
[97, 66, 35, 4, 101, 70, 39, 8, 105, 74, 43, 12, 109, 78, 47, 16, 113, 82, 51]
for i = 98
[98, 68, 38, 8, 106, 76, 46, 16, 114, 84, 54, 24, 122, 92, 62, 32, 2, 100, 70]
for i = 99
[99, 70, 41, 12, 111, 82, 53, 24, 123, 94, 65, 36, 7, 106, 77, 48, 19, 118, 89]
for i = 100
[100, 72, 44, 16, 116, 88, 60, 32, 4, 104, 76, 48, 20, 120, 92, 64, 36, 8, 108]
for i = 101
[101, 74, 47, 20, 121, 94, 67, 40, 13, 114, 87, 60, 33, 6, 107, 80, 53, 26, 127]
for i = 102
[102, 76, 50, 24, 126, 100, 74, 48, 22, 124, 98, 72, 46, 20, 122, 96, 70, 44, 18]
for i = 103
[103, 78, 53, 28, 3, 106, 81, 56, 31, 6, 109, 84, 59, 34, 9, 112, 87, 62, 37]
for i = 104
[104, 80, 56, 32, 8, 112, 88, 64, 40, 16, 120, 96, 72, 48, 24, 0, 104, 80, 56]
for i = 105
[105, 82, 59, 36, 13, 118, 95, 72, 49, 26, 3, 108, 85, 62, 39, 16, 121, 98, 75]
for i = 106
[106, 84, 62, 40, 18, 124, 102, 80, 58, 36, 14, 120, 98, 76, 54, 32, 10, 116, 94]
for i = 107
[107, 86, 65, 44, 23, 2, 109, 88, 67, 46, 25, 4, 111, 90, 69, 48, 27, 6, 113]
for i = 108
[108, 88, 68, 48, 28, 8, 116, 96, 76, 56, 36, 16, 124, 104, 84, 64, 44, 24, 4]
for i = 109
[109, 90, 71, 52, 33, 14, 123, 104, 85, 66, 47, 28, 9, 118, 99, 80, 61, 42, 23]
for i = 110
[110, 92, 74, 56, 38, 20, 2, 112, 94, 76, 58, 40, 22, 4, 114, 96, 78, 60, 42]
for i = 111
[111, 94, 77, 60, 43, 26, 9, 120, 103, 86, 69, 52, 35, 18, 1, 112, 95, 78, 61]
for i = 112
[112, 96, 80, 64, 48, 32, 16, 0, 112, 96, 80, 64, 48, 32, 16, 0, 112, 96, 80]
for i = 113
[113, 98, 83, 68, 53, 38, 23, 8, 121, 106, 91, 76, 61, 46, 31, 16, 1, 114, 99]
for i = 114
[114, 100, 86, 72, 58, 44, 30, 16, 2, 116, 102, 88, 74, 60, 46, 32, 18, 4, 118]
for i = 115
[115, 102, 89, 76, 63, 50, 37, 24, 11, 126, 113, 100, 87, 74, 61, 48, 35, 22, 9]
for i = 116
[116, 104, 92, 80, 68, 56, 44, 32, 20, 8, 124, 112, 100, 88, 76, 64, 52, 40, 28]
for i = 117
```

```
[117, 106, 95, 84, 73, 62, 51, 40, 29, 18, 7, 124, 113, 102, 91, 80, 69, 58, 47]
for i = 118
[118, 108, 98, 88, 78, 68, 58, 48, 38, 28, 18, 8, 126, 116, 106, 96, 86, 76, 66]
for i = 119
[119, 110, 101, 92, 83, 74, 65, 56, 47, 38, 29, 20, 11, 2, 121, 112, 103, 94, 85]
for i = 120
[120, 112, 104, 96, 88, 80, 72, 64, 56, 48, 40, 32, 24, 16, 8, 0, 120, 112, 104]
for i = 121
[121, 114, 107, 100, 93, 86, 79, 72, 65, 58, 51, 44, 37, 30, 23, 16, 9, 2, 123]
for i = 122
[122, 116, 110, 104, 98, 92, 86, 80, 74, 68, 62, 56, 50, 44, 38, 32, 26, 20, 14]
for i = 123
[123, 118, 113, 108, 103, 98, 93, 88, 83, 78, 73, 68, 63, 58, 53, 48, 43, 38, 33]
for i = 124
[124, 120, 116, 112, 108, 104, 100, 96, 92, 88, 84, 80, 76, 72, 68, 64, 60, 56, 52]
for i = 125
[125, 122, 119, 116, 113, 110, 107, 104, 101, 98, 95, 92, 89, 86, 83, 80, 77, 74, 71]
```

Above Program check if the element of a field is a primitive element and return the list of generators from the field F_{2^n} where 1<=n<=8. The program also generates the cyclic subgroup by using the power of elements.

The program computes order of element and no of totatives from Euler's totient function $\varphi(2^n)$ which is equal to 2^n-1.

If the order of element is equal to $\varphi(2^n)$ then the element is primitive element.

Based on this principle above program generates the primitive elements from finite field F_{2^n} where 1<=n<=8.

Q.N 7

Get Prime Factor of a Number n

```
#get prime factors of a number f
import math
def get_Prime_Factors(f):
  # n is given number

  # f = n! --> is factorial of n
  # f = compute_fact(n)

  #pf is a list of prime factors of f
  pf = []

  # Print the number of two's that divide f
```

```
  while f % 2 == 0:
    # print(2)
    pf.append(2)
    f = f / 2

  # if f is odd, skip by 2 ( k = k + 2)
  for k in range(3,int(math.sqrt(f))+1,2):

    # while i divides f , print k and progress
    while f % k == 0:
      # print(k)
      pf.append(k)
      f = f / k
  #if f is a prime number greater than 2
  if f > 2:
    pf.append(int(f))
  return pf


# test code
#get prime factors of n
n = 10001
get_Prime_Factors(n)
```

```
[73, 137]
```

Get Prime numbers less then or equal to a number n

```
def Sieve(f):
  primes = []
  remain = list(range(2,f+1))
  while remain:
    primes.append(remain[0])
    remain = [remain[i] for i in range(1,len(remain)) if 0 != remain[i]%remain[0]]
  return primes[1:]


#list of prime numbers <=n except 2
# n = 10
# all_prime = Sieve(n)
# all_prime[1:]
```

check if a number is a prime

```
def isPrime(n):
    return all(n % i for i in range(2, n))
```

```
#test code
n = 777
isPrime(n)
```

```
     False
```

Calculate Legendre Symbol

The function returns 0 if a is a multiple of p, 1 if a has a square root mod p, and -1 otherwise.

```
# calculate Legendre Symbol
# a is positive integer
# p is an odd prime
def Legendre(a, p):
    if a >= p or a < 0:
        return Legendre(a % p, p)
    elif a == 0 or a == 1:
        return a
    elif a == 2:
        if p % 8 == 1 or p % 8 == 7:
            return 1
        else:
            return -1
    elif a == p - 1:
        if p % 4 == 1:
            return 1
        else:
            return -1
    elif not isPrime(a):
        factors = get_Prime_Factors(a)
        product = 1
        for pi in factors:
            product *= Legendre(pi, p)
        return product
    else:
        if ((p - 1) / 2) % 2 == 0 or ((a - 1) / 2) % 2 == 0:
            return Legendre(p, a)
        else:
            return (-1) * Legendre(p, a)
```

```
# test code
a = 311 # positive integer
p = 653 # odd Prime
Legendre(a, p)
```

```
     -1
```

```
#testing Legendre symbol for 20 legal inputs
row = []
n = 20
odd_primes = Sieve(n)
#print a in row
for i in range(10):
  row.append(" " +str(i)+ " ")
print("p\\a" + str(row))
# print symbols

# for i in range(len(odd_primes)):
for p in odd_primes:
  l = []
  for a in range(1,n):
  # print("For a = ",a)
    # print(calculateLegendre(a, p))
    ls = Legendre(a, p)
    l.append(ls)
    # for i in odd_primes:
  print(p,l)
```

```
p\a[' 0 ', ' 1 ', ' 2 ', ' 3 ', ' 4 ', ' 5 ', ' 6 ', ' 7 ', ' 8 ', ' 9 ']
3 [1, -1, 0, 1, -1, 0, 1, -1, 0, 1, -1, 0, 1, -1, 0, 1, -1, 0, 1]
5 [1, -1, -1, 1, 0, 1, -1, -1, 1, 0, 1, -1, -1, 1, 0, 1, -1, -1, 1]
7 [1, 1, -1, 1, -1, -1, 0, 1, 1, -1, 1, -1, -1, 0, 1, 1, -1, 1, -1]
11 [1, -1, 1, 1, 1, -1, -1, -1, 1, -1, 0, 1, -1, 1, 1, 1, -1, -1, -1]
13 [1, -1, 1, 1, -1, -1, -1, -1, 1, 1, -1, 1, 0, 1, -1, 1, 1, -1, -1]
17 [1, 1, -1, 1, -1, -1, -1, 1, 1, -1, -1, -1, 1, -1, 1, 1, 0, 1, 1]
19 [1, -1, -1, 1, 1, 1, 1, -1, 1, -1, 1, -1, -1, -1, -1, 1, 1, -1, 0]
```

Above function calculate legendre symbol for any legal (a,p) where a is any positive integer and p is an odd Prime. The function returns 0 if a is a multiple of p, 1 if a has a square root mod p, and -1 otherwise.

Calculate Jacobi Symbol

The Jacobi symbol is a generalization of the Legendre symbol and uses the same notation. Unlike in legendre symbol it does not require p to be prime and only requires that p is odd.

So in Jacobi symbol(a,n), a represent positive integer and n represents positive odd number.

So, legendre symbol is an special case of the jacobi symbol. if an odd number n comes to be the Prime number, jacobi symbol and Legendre symbols gices the same notation.

```
#Jacobi
#a is any positive integer
```

```
#n is odd number
def jacobi(a, n):
  if(((n > a) and (a>0))and(n%2 == 1)):
    # assert(n > a > 0 and n%2 == 1)
    t = 1
    while a != 0:
        while a % 2 == 0:
            a /= 2
            r = n % 8
            if r == 3 or r == 5:
                t = -t
        a, n = n, a
        if a % 4 == n % 4 == 3:
            t = -t
        a %= n
    if n == 1:
        return t
    else:
        return 0
  # else:
  #   return ""


#get odd numbers
def odd(n):
  o = []
  for k in range(1,n):
    if k%2 == 1:
      o.append(k)
  return o
```

Above function returns list of odd numbers upto n

```
#generate odd number upto n
n = 10
odd(10)
```

```
     [1, 3, 5, 7, 9]
```

```
#get odd prime numbers upto n


#Testing Jacobi symbol for 20 legal inputs
row = []
n = 20
odds = odd(10)
#print a in row
for i in range(10):
  row.append(" " +str(i)+ " ")
```

```
print("p\\a" + str(row))
# print symbols

# for i in range(len(odd_primes)):
for o in odds:
  l = []
  for a in range(10):
    ls = jacobi(a,o)
    l.append(ls)
  print(o,l)
```

```
   p\a[' 0 ', ' 1 ', ' 2 ', ' 3 ', ' 4 ', ' 5 ', ' 6 ', ' 7 ', ' 8 ', ' 9 ']
   1 [None, None, None, None, None, None, None, None, None, None]
   3 [None, 1, -1, None, None, None, None, None, None, None]
   5 [None, 1, -1, -1, 1, None, None, None, None, None]
   7 [None, 1, 1, -1, 1, -1, -1, None, None, None]
   9 [None, 1, 1, 0, 1, 1, 0, 1, 1, None]
```

Testing if the inputs are legal for for Legendre and jacobi symbol

Hand Written solution for (311/653) aand (666/777)

For any integer a and any positive odd integer n, the Jacobi symbol (a/n) is defined as the product of the Legendre symbols corresponding to the prime factors of n.

For (311/653) we have a = 311 and b = 653. here both a and b are prime nummber but a<b

we can only apply jacobi for a>b

Also,

(a/b) = -(b/a)

So we can swap a = 653 and b = 311 and apply jacobi

so 653 mod 311 = 31 mod 311

653 $\cong$ 31 mod 311

Again 31 and 311 are both prime and 31 is less than 311 so we swap them and perform modulo arithmatic.

311 mod 31 $\cong$ 1 mod 31

so $(1)^{(31-1)/2}$ mod 31

1^{15} mod 31

= 1

Thus, (653/311) = 1

So,

(311/653) = -(653/311) = -1

```
a1 = 311
p1 = n1 = 653
j = jacobi(a1,n1)
print("Jacobi Symbol",j)
l = Legendre(a1, p1)
print("Legendre sybmol",l)
```

```
     Jacobi Symbol -1
     Legendre sybmol -1
```

Hence, the hand written result and output of the program for jacobi symbol of (311/653) are both -1.

Similarly,

For (666/777)

we have a = 666 and b = 777. here a is an even number and b is an odd number which has 3 prime factors 3, 7 and 37.

so 666 mod 777 = (666/3).(666/7) (666/37) mod 777

Now,

666 mod 3 $\cong$ 0 mod 3

> (666/3)= 0

And,

> 666 mod 7 $\cong$ 1 mod 7 $(1)^{(7-1)/2}$ mod 7 $\cong 1^3$ mod 7 $\cong$ 1 mod 7
>
> (666/7) = 1

And,

> 666 mod 37 $\cong$ 0 mod 37
>
> (666/37) = 0

Thus,

(666/777) = (666/3).(666/7) (666/37) mod 777

So,

$(666/777) = 0 * 1 * 0 = 0$

```
a2 = 666
p2 = n2 = 777
j = jacobi(a2,n2)
print("Jacobi Symbol",j)
l = Legendre(a2, p2)
print("Legendre sybmol",l)
```

```
     Jacobi Symbol 0
     Legendre sybmol 0
```

Hence, the hand written result and output of the program for jacobi symbol of (666/777) are both 0.

Q.N 8

a) List of generators of C9

Find the order of element in cyclic sub group under Addition

```
#return the order of an element
#cycle: is an list of cyclic subgroup
def get_order(cycle):
  for i in cycle:
    if i==1:
      order = cycle.index(i) + 1
      return order



#test code
cycle = [2, 4, 6, 8, 1, 3, 5, 7, 0]
get_order(cycle)
```

```
     5
```

Cyclic sub-group of Z_9 and Generators

```
#find the generator of the group
# for n in range(1,9):
a = 9
print("----------------------------------------")
# print("for 2^{}" format(n))
```

```
# print(" for Z^{} .format(n))
print("\nElements are of Z9 are:{}".format([i for i in range(a)]))
# print([i for i in range(n)])
print("----------------------------------------")
# i represents elements in field f
#gen: is the list of generaters present in the group
gen = []
for i in range(1,a+1):
  print("for i = {}".format(i))
  #cycle is the list of cyclic sub-group under an group operation
  #addition operation
  cycle = [(i*x)%a for x in range(1,20)]
  #multiplication operation
  # cycle = [(i**x)%a for x in range(1,20)] ()

  #order: order of element
  order = get_order(cycle)
  print("..................")
  print("order is:",order)
  print("..................")
  print("Cycle:")
  print(cycle)
  # phi: phi gives number of totatives from Euler's totient (n-1)
#   phi = a-1
#   if(order == phi):
#     gen.append(i)
#     # print("\n{} is a primitive element".format(i))
# if (len(gen)!= 0):
#   print("generators are:",gen)
# else:
#   print("There are no generators in this group")
```

```
----------------------------------------

Elements are of Z9 are:[0, 1, 2, 3, 4, 5, 6, 7, 8]
----------------------------------------
for i = 1
..................
order is: 1
..................
Cycle:
[1, 2, 3, 4, 5, 6, 7, 8, 0, 1, 2, 3, 4, 5, 6, 7, 8, 0, 1]
for i = 2
..................
order is: 5
..................
Cycle:
[2, 4, 6, 8, 1, 3, 5, 7, 0, 2, 4, 6, 8, 1, 3, 5, 7, 0, 2]
for i = 3
..................
order is: None
..................
Cycle:
[3, 6, 0, 3, 6, 0, 3, 6, 0, 3, 6, 0, 3, 6, 0, 3, 6, 0, 3]
for i = 4
```

```
for i = 4
..................
order is: 7
..................
Cycle:
[4, 8, 3, 7, 2, 6, 1, 5, 0, 4, 8, 3, 7, 2, 6, 1, 5, 0, 4]
for i = 5
..................
order is: 2
..................
Cycle:
[5, 1, 6, 2, 7, 3, 8, 4, 0, 5, 1, 6, 2, 7, 3, 8, 4, 0, 5]
for i = 6
..................
order is: None
..................
Cycle:
[6, 3, 0, 6, 3, 0, 6, 3, 0, 6, 3, 0, 6, 3, 0, 6, 3, 0, 6]
for i = 7
..................
order is: 4
..................
Cycle:
[7, 5, 3, 1, 8, 6, 4, 2, 0, 7, 5, 3, 1, 8, 6, 4, 2, 0, 7]
for i = 8
..................
order is: 8
..................
Cycle:
[8, 7, 6, 5, 4, 3, 2, 1, 0, 8, 7, 6, 5, 4, 3, 2, 1, 0, 8]
for i = 9
..................
order is: None
..................
Cycle:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

if C_m is a cyclic group of order m generated by x, then the generators of C_m are the elements x^k with k coprime with m.

So, for Cyclic group order 9 is denoted by: C_9 = {1,2,3,4,5,6,7,8,9}

Generators are the elements in $C^9 which are Relatively prime to 9.

So generators are 1,2,4,5,7,8.

Examples: 1 generates C9

1 = 1 mod 9

1 + 1 = 2 mod 9

1 + 1 + 1 = 3 mod 9

1 + 1 + 1 + 1 = 4 mod 9

1 + 1 + 1 + 1 + 1 = 5 mod 9

1 + 1 + 1 + 1 + 1 + 1 = 6 mod 9

1 + 1 + 1 + 1 + 1 + 1 + 1 = 7 mod 9

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8 mod 9

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 0 mod 9

So, 1 is the generator of C9

Similarly for 2:

2 = 2 mod 9

2+2= 4 mod 9

2+2+2 = 6 mod 9

2+2+2+2 = 8 mod 9

2+2+2+2+2 = 1 mod 9

12 = 3 mod 9

14 = 5 mod 9

16 = 7 mod 9

18 = 0 mod 9

b Group of order 9 that G!= C9 is the Z3 X Z3.

Suppose we have two group of order 3 which are abelian. $C3$ = {1,a,a^2} and $C3$ = {1,b,b^2} then C3 X C3 gives the group of order 9 which is also abelian.

A group G = $Z_3 * Z_3$ is a group of order 9 which is not cyclic. ie. G != C9.

The group is G = $Z_p * Z_q$ is cyclic if only GCD(p,q) = 1.

In G = $Z_3 * Z_3$, GCD(3,3) != 1. Hence $Z_3 * Z_3$ is not a cyclic group.

But any group which is the order of p^2 is always abelian. Hence G is also abelian.

In Z3 we have element of order either 1 or 3. Element of 1 order is identity and the remaining elements are of order 3.

So in case of $Z_3 * Z_3$ we can find the order of the elements by calculating the LCM of the order of elements in two seprate groups.

for eg: LCM(1,1) = 1

Element of order 1 is obviously the identity element.

LCM(1,3) = 3

Total number of elements is given by = $φ(1)*φ(3)* = *1*2 = 2

LCM(3,1) = 3

Total number of elements is given by = $φ(3)*φ(1)* = *2*1 = 2

LCM(3,3) = 3 Total number of elements is given by = $φ(3)*φ(3)* = *2*2 = 4

So there are total 8 elements of order 3 and one element of order 1 in G = $Z_3 * Z_3$

Hence it is proved that G has some elements of order 3.

e)

Given two groups G1 and G2, the Cartesian product $G_1 X G_2$ is the set of ordered pairs (g1, g2) with g1 ϵ G1, g2 ϵ G2: G1 X G2 = {(g1, g2) | g1 ϵ G1, g2 ϵ G2} which is aslo a group.

let G1 = g1 · g1' and G2 = g1 · g2' Direct cartician product:
$G_1 X G_2 = (g1, g1')(g2, g2') = (g1g2, g1'g2')$

Identity is (e1, e2) where e1 is the identity of G1 nad e2 is the identity of G2.

Also Inverse is:

$(g1, g2)^{-1}$ = (g_1^{-1}, g_2^{-1})

(g) and (h)

Cayley Table:

Associativity depends on a 3 term equation, (a b) c = a (b c) but the Cayley table shows only 2-term products. Thats why we cannot simply assume that operation is associative looking into the result of Cayley table.

K)

There are only two group of order 9 and both of them are abelian. The groups are: C_9 and C_3 X C_3 . C_9 is cyclic group because some elements in group has the order equal to the group order 9. But $C_3 X C_3$ is only abelian and not cyclic because no elements in this group has order 9.

Q.N 9

(a)

(i)Take your 8-digit student ID and break it into two 4-digit num bers by taking the top four digits as one number and the bottom four digits as the other number. For example, if your student ID was 87654321, your top number would be 8765 and your bottom number would be 4321. Ignore leading zeroes.

Double-click (or enter) to edit

```
my_sid = '12584026'
top_number = my_sid[:4]
bottom_number = my_sid[4:]
print(top_number)
print(bottom_number)
```

```
1258
4026
```

ii. Write the code to compute (bottom number)topnumber mod 10001 to get a four digit number. Pad with leading zeroes to make sure it is a 4-digit string.

```
x = int(bottom_number)**int(top_number)
y = x%10001
print("class id =",y)
```

```
class id = 1133
```

> My New Class id is 1133

(b) Explain why it is better to use mod 10001 than mod 10000 in the calculation above.

```
#check if a number is prime
def isPrime(n):
  if type(n) != int:
    return False
  n = abs(n)
  if n < 2:
    return False
  if n == 2:
    return True
  if 0 == n%2:
    return False
  d = 3
  while d*d <= n:
    if 0 == n%d:
      return False
    d += 2
  return True


# test code
n = 1133
if isPrime(n):
  print("Number {} is prime".format(n))
else:
  print("Number {} is not prime".format(n))
```

```
Number 1133 is not prime
```

```
x = int(bottom_number)**int(top_number)
y = x%10001
z = x%10000
print("class id =",y)
print("class id =",z)
```

```
class id = 1133
class id = 6576
```

```
get_Prime_Factors(10001)
```

```
[73, 137]
```

```
get_Prime_Factors(10000)
```

```
[2, 2, 2, 2, 5, 5, 5, 5]
```

10001 is an odd number which is the product of two prime factors 73 and 137.

while 10000 is an even and the product of 2^4 and 5^4. its prime factors are 2 and 5.

So, it is more difficult to reverse the process when n has bigger prime factors.

c) How hard it is to reverse engineer the process?

Given only the class id,it would be very difficult to reverse engineer the the process of finding original student id. As we have, class_id = $bottom^{top}$ mod 10001

we need to find bottom and top number.

The problem can be solved only by finding top number. But Top number can be found by taking log_{bottom} of class_id only if the bottom number and class id is known. This is an discrete logarithm problem.

To solve this problem, we need to calculate all the powers of (base nummber) modulo (73*137) which requires alot of computational time and space.