Required Imports

```
import numpy as np
import os
import math
import binascii
import gmpy2
import secrets
import sympy
```

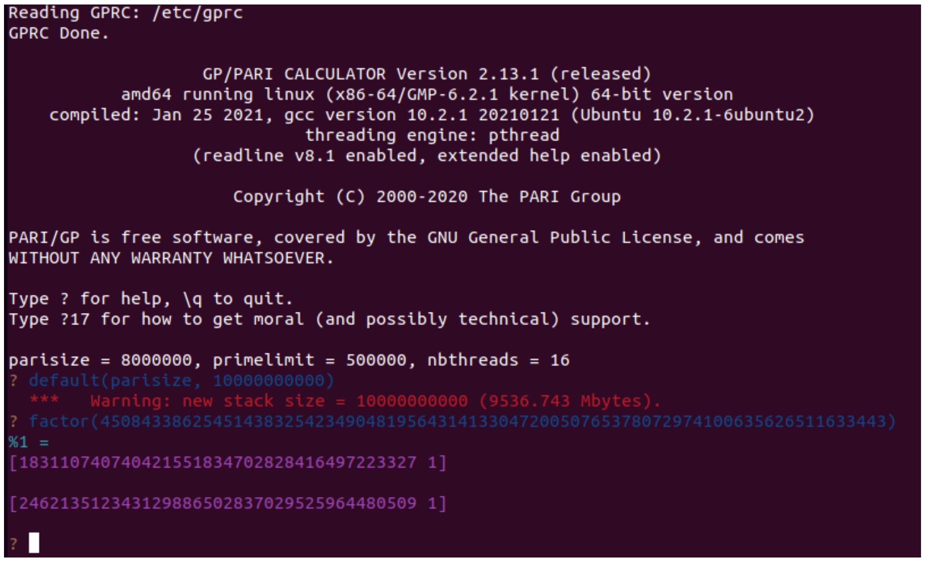
Challenge #1 - Simple factorization

We have a 256-bit number that we need to factorize.

Implement the solution in any chosen programming language!

`N=45084338625451438325423490481956431413304720050765378072974100635626511633443`

I used PARI/GP as a programming language. It has a built in function factor(N) which can factorise a 256-bit number in about 5 minutes.



```
Reading GPRC: /etc/gprc
GPRC Done.

                   GP/PARI CALCULATOR Version 2.13.1 (released)
           amd64 running linux (x86-64/GMP-6.2.1 kernel) 64-bit version
    compiled: Jan 25 2021, gcc version 10.2.1 20210121 (Ubuntu 10.2.1-6ubuntu2)
                             threading engine: pthread
                  (readline v8.1 enabled, extended help enabled)

                      Copyright (C) 2000-2020 The PARI Group

PARI/GP is free software, covered by the GNU General Public License, and comes
WITHOUT ANY WARRANTY WHATSOEVER.

Type ? for help, \q to quit.
Type ?17 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000, nbthreads = 16
? default(parisize, 10000000000)
  ***   Warning: new stack size = 10000000000 (9536.743 Mbytes).
? factor(45084338625451438325423490481956431413304720050765378072974100635626511633443)
%1 =
[183110740740421551834702828416497223327 1]

[246213512343129886502837029525964480509 1]

? 
```

Double-check the result in python

```
p = 183110740740421551834702828416497223327
q = 246213512343129886502837029525964480509

print(f"N = {N}\n")
print(f"First prime p:\n{p}\n")
print(f"Second prime q:\n{q}\n")
print(f"N == (p * q) : {N == (p * q)}")
```

```
N = 45084338625451438325423490481956431413304720050765378072974100635626511633443

First prime p:
183110740740421551834702828416497223327

Second prime q:
246213512343129886502837029525964480509

N == (p * q) : True
```

Challenge #2 - Special prime numbers

We have a 2048-bit number (N) that we need to factorize.

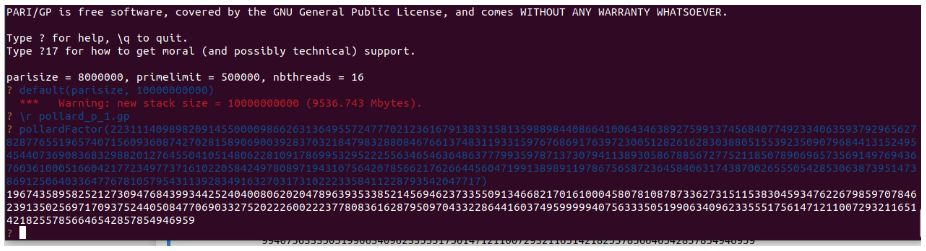
Implement the solution in any chosen programming language!

Why is it possible to factorize N?

N=22311140989820914550000986626313649557247770212361679138331581359889844086641006434638927

I used PARI/GP as a programming language. I wrote the Pollard's p-1 function in a .gp file (using the sample codes provided during the semester for help) and used it to factor N.

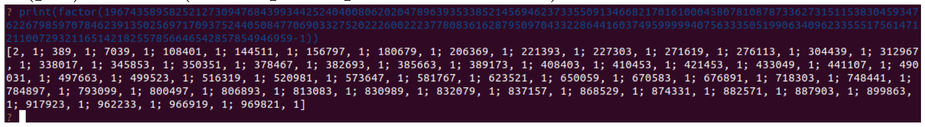
```
pollardFactor(n) = {
a = vector(1000003, i, i+1); \\works if p-1 factors < 1000000 (p is a prime factor of n)
k = 2;
while((gcd(n, a[k-1]-1) % n) == 1,
a[k]= Mod(a[k-1], n)^k; k++;);
print(lift(gcd(n,a[k-1]-1)));
}
```



```
PARI/GP is free software, covered by the GNU General Public License, and comes WITHOUT ANY WARRANTY WHATSOEVER.

Type ? for help, \q to quit.
Type ?17 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000, nbthreads = 16
? default(parisize, 10000000000)
  ***   Warning: new stack size = 10000000000 (9536.743 Mbytes).
? \r pollard_p_1.gp
? pollardFactor(2231114098982091455000098662631364955724777021236167913833158135988984408664100643463892759913745684077492334063593792965627
82877655196574071560936087427028158906900392837032184798328808467661374831193315976768691763972300512826162830388051553923509079684413152495
45440736908368329882012764550410514806228109178699532952225563465463648637779935978713730794113893058678856727752118507890696573569149769436
76036100051660421772349773716102205842497808971943107564207856621762664456047199138989119786756587236458406317438700265550542853063873951473
8691250640336477678105795431139283491632703173102223358411228793542047717)
196743589582521273094768439934425240400806202047896393533852145694623733550913466821701610004580781087873362731511538304593476222679859707846
23913502569717093752440508477069033275202226002223778083616287950970433228644160374959999940756333505199063409623355517561471211007293211651
42182557856646542857854946959
? 
```

We can factorize N because (p-1) factors (where p is a prime factor of N) are "small" (less than 1000000). 

```
? print(factor(19674358958252127309476843993442524040080620204789639353385214569462373355091346682170161000458078108787336273151153830459347
622267985970784623913502569717093752440508477069033275202226002223778083616287950970433228644160374959999940756333505199063409623355517561471
21100729321165142182557856646542857854946959-1))
[2, 1; 389, 1; 7039, 1; 108401, 1; 144511, 1; 156797, 1; 180679, 1; 206369, 1; 221393, 1; 227303, 1; 271619, 1; 276113, 1; 304439, 1; 312967
, 1; 338017, 1; 345853, 1; 350351, 1; 378467, 1; 382693, 1; 385663, 1; 389173, 1; 408403, 1; 410453, 1; 421453, 1; 433049, 1; 441107, 1; 490
031, 1; 497663, 1; 499523, 1; 516319, 1; 520981, 1; 573647, 1; 581767, 1; 623521, 1; 650059, 1; 670583, 1; 676891, 1; 718303, 1; 748441, 1;
784897, 1; 793099, 1; 800497, 1; 806893, 1; 813083, 1; 830989, 1; 832079, 1; 837157, 1; 868529, 1; 874331, 1; 882571, 1; 887903, 1; 899863,
1; 917923, 1; 962233, 1; 966919, 1; 969821, 1]
? 
```

Double-check the result in python

```
p = 1967435895825212730947684399344252404008062020478963935338521456946237335509134668217016
q = N // p
print(f"N:\n{N}\n")
print(f"First prime p:\n{p}\n")
print(f"Second prime q:\n{q}\n")
print(f"N == (p * q) : {N == (p * q)}")
```

```
N:
22311140989820914550000986626313649557247770212361679138331581359889844086641006434638927599

First prime p:
19674358958252127309476843993442524040080620204789639353385214569462373355091346682170161000

Second prime q:
11340212424284770348682740989489556903535930770077280769579291594213942989527651847507988186

N == (p * q) : True
```

Challenge #3 - RSA factorization of a 2048-bit N modulus

```
N = 1553558122535294204146382160754477729177064335988234493840142817233685295530583281678982
```

```
#Original source code to generate N

from Crypto.Util import number
# Generate 1024-bit P prime
x = number.getRandomNumber(1024)
while True:
    x=x+2
    if (number.isPrime(x)==True):
        P=x
        break
# Generate 1024-bit Q prime
while True:
    x=x+2
    if (number.isPrime(x)==True):
        Q=x
        break
N=P*Q
print(N)
```

In the first case, the problem is that P and Q will be consecutive primes.

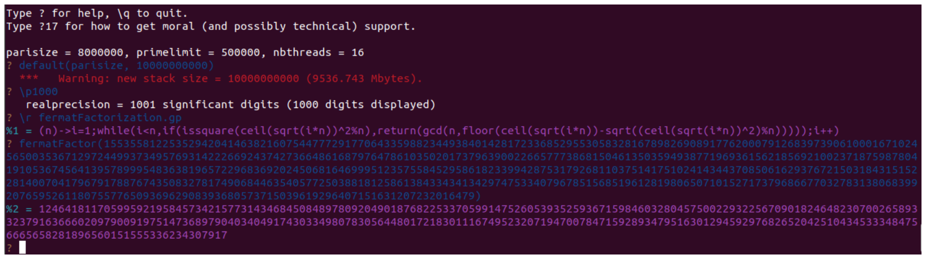
```
#Correct source code to generate a safe N
```

```
from Crypto.Util import number
# Generate 1024-bit P prime
x = number.getRandomNumber(1024)
while True:
    x += 1
    if (number.isPrime(x)==True):
        P=x
        break
# Generate 1024-bit Q prime
x = number.getRandomNumber(1024)
while True:
    x += 1
    if (number.isPrime(x)==True):
        Q=x
        break
N=P*Q
print(N)
```

In the second case, P and Q are independent of each other. I used another modification too because I don't know if the random number generated with number.getRandomNumber(1024) is odd or even. So I used x + = 1 in the iteration.

For the factorization I used the Fermat factorization algorithm in a PARI/GP function.

```
fermatFactor(n) = {
i = 1;
while(i < n, if(issquare(ceil(sqrt(i*n))^2 % n), return(gcd(n, floor(ceil(sqrt(i * n)) -
sqrt((ceil(sqrt(i*n))^2) % n)))));i++)
}
```



```
Type ? for help, \q to quit.
Type ?17 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000, nbthreads = 16
? default(parisize, 10000000000)
  ***   Warning: new stack size = 10000000000 (9536.743 Mbytes).
? \p1000
   realprecision = 1001 significant digits (1000 digits displayed)
? \r fermatFactorization.gp
%1 = (n)->i=1;while(i<n,if(issquare(ceil(sqrt(i*n))^2%n),return(gcd(n,floor(ceil(sqrt(i*n))-sqrt((ceil(sqrt(i*n))^2)%n)))));i++)
? fermatFactor(155355812253529420414638216075447772917706433598823449384014281723368529553058328167898269089177620007912683973906100016710245650035367129724499373495769314222669243742736648616879764786103502017379639002266577738681504613503594938771969361562185692100237187598780419105367456413957899954836381965722968369202450681646999512357558452958618233994287531792681103751417510241434437085061629376721503184315152281400704179679178876743508327817490684463540577250388181258613843343413429747533407967851568519612819806507101527173796866770327831380683992076595261180755776509369629083936805737150396192964071516312072320164 79)
%2 = 12464181170599592195845734215773143468450848978092049018768225337059914752605393525936715984603280457500229322567090182464823070026589332379163666020979009197514736897904034049174303349807830564480172183011167495232071947007847159289347951630129459297682652042510434533348475666565828189656015155533623430791 7
? 
```

Double-check the result in python

```
p = 1246418117059959219584573421577314346845084897809204901876822533705991475260539352593 67
q = N // p

print(f"N:\n{N}\n")
print(f"First prime p:\n{p}\n")
print(f"Second prime q:\n{q}\n")
print(f"N == (p * q) : {N == (p * q)}")
```

```
N:
15535581225352942041463821607544777291770643359882344938401428172336852955305832816789826908

First prime p:
12464181170599592195845734215773143468450848978092049018768225337059914752605393525936715984

Second prime q:
12464181170599592195845734215773143468450848978092049018768225337059914752605393525936715984

N == (p * q) : True
```

Challenge #4 - Encrypted text

Recover the original cleartext message from textEnc!

Implement the solution in any chosen programming language!

```
textEnc = 2563892093825260556889554674591586443003260144924638910200390559559028153561449408
N = 3047991548766932693015493838096876643183394999334699113269414972318781793533952896285554
d = 1165349714473549788499461817007086654077030204286579423549929520139641964090212516932834
e = 65537

#import binascii

class RSA:

    def __init__(self, p=0, q=0, e = 2**16 + 1, N=0, Phi=0, d=0):
        self.p = p
        self.q = q
        self.e = e
        self.Phi = Phi
        if (p and q and e):
            self.N = p * q
            self.Phi = (p - 1) * (q - 1)
            self.d = gmpy2.invert(e,self.Phi)
        elif (Phi):
            self.N = N
            self.Phi = Phi
            self.d = gmpy.invert(e, Phi)
        else:
            self.N = N
            self.Phi = Phi
            self.d = d
    def encodeRSA(self,m):
        """Encode m plain text with e and N"""
        mInt = int(binascii.hexlify(bytes(m, "utf-8")).decode(),16)
        return pow(mInt, self.e, self.N)

    #import binascii
    def decodeRSA(self,c):
        """Recover the original text from cipher text (c)
           using N (N = p * q, the product of two prime numbers)
           and d (e * d = k * Phi(N) + 1 (where k is a whole number)
           and e is a chosen number,
           which is coprime with Phi(N) = (p-1) * (q-1): (Phi(N), e) = 1)."""

        textPlain = pow(c, self.d, self.N)
        return binascii.unhexlify(hex(textPlain)[2:]).decode()


rsa = RSA(N=N, d=d, e=e)
print(rsa.decodeRSA(textEnc))

print(f"""\nReverse the cipher text from plain text:
      \n{rsa.encodeRSA(rsa.decodeRSA(textEnc))}\n""")
print(f"""textEnc == rsa.encodeRSA(rsa.decodeRSA(textEnc))
      : {textEnc == rsa.encodeRSA(rsa.decodeRSA(textEnc))}""")
```

```
RSA is a very simple but efficient encryption alghorithm!

Reverse the cipher text from plain text:
256389209382526055688955467459158644300326014492463891020039055955902815356144940831152854

textEnc == rsa.encodeRSA(rsa.decodeRSA(textEnc)) : True
```

Challenge #5 - Creating the RSA private key

Recreate the RSA public and private key from P and Q numbers!

Implement the solution in any chosen programming language!

```
def gcd(a, b):
    while (b != 0):
        a, b = b, a % b
    return a
```

```
P = 47107077831526529631313930390625355687928115212735348527388428825777111998627
Q = 21536887994154870131965390890995885766722023702428541798692456954162584328961
N = P * Q
Phi = (P - 1) * (Q - 1)
```

```
e = 2**16 + 1
d = int(gmpy2.invert(e,Phi))

print(f"gcd(Phi_N, e) == 1 : {gcd(Phi, e) == 1}")
print(f"(e * d) % Phi == 1: {(e * d) % Phi == 1}")
```

```
gcd(Phi_N, e) == 1 : True
(e * d) % Phi == 1: True
```

The public and private keys are (e,N) and (d,N) respectively.

```
rsa = RSA(p=P, q=Q, e=e)
print(f"e = {rsa.e}\n")
print(f"d:\n{rsa.d}\n")
print(f"N:\n{rsa.N}")
```

```
e = 65537

d:
2637552850674197570179056561335764094249016060591033586573528721720676048804388163701968860

N:
1014539858989522750069402687288777857992709036054434605958852869089141602362395132312673511
```

Challenge #6 - Creating RSA private key from Phi(n)

Recreate the RSA public and private key from Phi(n)!

```
Phi = 3072714441320937436364108905029797568594870671848898979161725511500698914627748388986
N = 931075978510432194799476591235433185764752330230979037809128697780185907953438206157508
e = 2**16 + 1
d = gmpy2.invert(e,Phi)

print(f"gcd(Phi_N, e) == 1 : {gcd(Phi, e) == 1}")
print(f"(e * d) % Phi == 1: {(e * d) % Phi == 1}")
```

```
gcd(Phi_N, e) == 1 : True
(e * d) % Phi == 1: True
```

The public and private keys are (e,N) and (d,N) respectively.

```
print(f"e = {e}\n")
print(f"d:\n{d}\n")
print(f"N:\n{N}")
```

```
e = 65537

d:
```

```
546587499533385547601092232095417551225704598813105029205905000428386223762611818038739919
```

```
N:
931075978510432194799476591235433185764752330230979037809128697780185907953438206157508453
```

Challenge #7 - Diffie-Hellman key exchange

Implement and demonstrate the Diffie-Hellman key exchange in python!

The Diffie-Hellman protocol:

- Alice chooses prime P at random and finds a generator g.
- Alice chooses $X \leftarrow_R \{0, 1, \ldots, P-2\}$ and sends P, g and $\hat{X} = g^X \pmod{P}$ to Bob.
- Bob chooses $Y \leftarrow_R \{0, 1, \ldots, P-2\}$ and sends $\hat{Y} = g^Y \pmod{P}$ to Alice.
- Alice and Bob both compute $k = (g^X)^Y = (g^Y)^X \pmod{P}$. Alice does that by computing $\hat{Y}^X$ and Bob does this by computing $\hat{X}^Y$.
- They then use k as a key to exchange messages using a private key encryption scheme.

```python
#import secrets
#import sympy

P = '0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A
P = int(P,16)

class User:
    #Standards for choosing P prime and g generator,
    #all participants have this dictionary in their system.
    #in this example I will use the RFC 3526 standard
    #(the other "standards" only demonstrates the option of choose)
    #RFC 3526 now consist of a 2048 bit P prime and the generator g = 2
    standardsMap = {"RFC_3526" : (P,2), 'P7g3' : (7,3), 'P353g3' : (353,3)}

    def chooseStandard(self):
        self.standardName = "RFC_3526"
        self.P = self.standardsMap["RFC_3526"][0]
        self.g = self.standardsMap["RFC_3526"][1]

    def receiveStandard(self, name):
        self.StandardName = name
        self.P = self.standardsMap[name][0]
        self.g = self.standardsMap[name][1]
    def generatePrivateKey(self):
        self.privateKey = secrets.randbits(2048) % (self.P - 1) #{0,1,...,P-2}

    def generatePublicKey(self):
        self.publicKey = pow(self.g, self.privateKey, self.P)

    def receivePublicKey(self, key):
        self.receivedPublicKey = key

    def createCommonKey(self):
        self.commonKey = pow(self.receivedPublicKey, self.privateKey, self.P)


def diffieHellmanAB(A, B):
    #Share P and g
    A.chooseStandard()
    B.receiveStandard(A.standardName)

    #Create private keys
    A.generatePrivateKey()
    B.generatePrivateKey()

    #Create public keys
    A.generatePublicKey()
    B.generatePublicKey()

    #Share public keys
    A.receivePublicKey(B.publicKey)
    B.receivePublicKey(A.publicKey)

    #Create shared secret (or common key)
    A.createCommonKey()
    B.createCommonKey()


def testDiffieHellman(times):
    for i in range(times):
        A = User()
        B = User()
        diffieHellmanAB(A, B)
        print(f"""keyA:\n{A.commonKey}\nkeyB:\n{B.commonKey}\n\t\t\t\t\t
        keyA == keyB : {A.commonKey == B.commonKey}\n\n""")

testDiffieHellman(5)
```

```
keyA:
6813558378495060920993017228276398163883358878924437225574176922845613159271465322901251234
keyB:
6813558378495060920993017228276398163883358878924437225574176922845613159271465322901251234

        keyA == keyB : True


keyA:
1491370417231133294019224852537026416570272300628932570932838295012714808777690048084777774
keyB:
1491370417231133294019224852537026416570272300628932570932838295012714808777690048084777774

        keyA == keyB : True


keyA:
2975376384472492875327038559310756101945766345059942321663836236425790369819433107060616744
keyB:
2975376384472492875327038559310756101945766345059942321663836236425790369819433107060616744

        keyA == keyB : True


keyA:
9433817214694380570486638489978116666223948987228364811369656893142595025562306706925643841
keyB:
9433817214694380570486638489978116666223948987228364811369656893142595025562306706925643841

        keyA == keyB : True


keyA:
8409513448774699845038564095921177203165905649657616720807348636837313479826773842038999090
keyB:
8409513448774699845038564095921177203165905649657616720807348636837313479826773842038999090

        keyA == keyB : True
```