In [1]:
```
import math
```

1a)

In [2]:
```
x72 = bin(72)
x136 = bin(136)

print("72 på binærform", x72)
print("136 på binærform", x136)

#Kan også dele 72 eller 136 på 2 med hensyn på rest, og lese det op
pover.
```

```
72 på binærform 0b1001000
136 på binærform 0b10001000
```

72 / 2 = 36 + 0.
36 / 2 = 34 + 0.
34 / 2 = 17 + 0.
17 / 2 = 8 + 1.
8 / 2 = 4 + 0.
4 / 2 = 2 + 0.
2 / 2 = 1 + 0.
1 / 2 = 0 + 1.

72 = 01001000

136 / 2 = 68 + 0.
36 / 2 = 34 + 0.
34 / 2 = 17 + 0.
17 / 2 = 8 + 1.
8 / 2 = 4 + 0.
4 / 2 = 2 + 0.
2 / 2 = 1 + 0.
1 / 2 = 0 + 1.

136 = 10001000

1b)

11^2 = 121 mod 10001
11^4 = 4640 mod 10001
11^8 = 7448 mod 10001
11^16 = 7158 mod 10001
11^32 = 1841 mod 10001
11^64 = 8943 mod 10001
11^128 = 9253 mod 10001

a = 11^72 mod 10001 = 8943*7448 mod 10001 = 804 mod 10001*
*b = 11^136 mod 10001 = 9253*7448 mod 10001 = 9454 mod 10001

1c)

gcd(10001, 804) = 1

10001 = 12 *804* + *353*
804 = *2* 353 + 98
353 = 3 *98* + *59*
98 = *1* 59 + 39
59 = 1 *39* + *20*
39 = *1* 20 + 19
20 = 1 *19* + *1*
19 = *19* 1 + 0

gcd(10001, 9454) = 1

10001 = 1 *9454* + *547*
9454 = *17* 547 + 155
547 = 3 *155* + *82*
155 = *1* 82 + 73
82 = 1 *73* + *9*
73 = *8* 9 + 1
9 = 9 * 1 + 0

1d)

ab mod 10001 = 804*9454 mod 10001 = 256 mod 10001

2)

In [3]:
```
def lcm(a, b):
    return abs(a*b) // math.gcd(a, b)

p = 137
q = 139
n = p*q

phi = lcm(p-1, q-1)


print("p = {}, q = {}".format(p,q))
print("N = ", n)
print("phi(n) = {}".format(phi))
print ("3 < e < {}".format(phi))

e = 5
print("E is =", e)
print("since gcd(5,{}) = 1".format(phi))
```

```
p = 137, q = 139
N =  19043
phi(n) = 9384
3 < e < 9384
E is = 5
since gcd(5,9384) = 1
```

2a)

In [4]:
```
print("public key(n={},e={})".format(n,e))
```

```
public key(n=19043,e=5)
```

2b)

In [5]:
```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        gcd, x, y = egcd(b % a, a)
        return (gcd, y - (b//a) * x, x)

gcd, x, y = egcd(e, phi)


d = x

print("private key(p={},q={},d={})".format(p,q,d))
```

```
private key(p=137,q=139,d=1877)
```

2c)

In [6]:
```
def squareAndMultiply(base, exponent):
    if exponent < 0:
        return squareAndMultiply(1 / base, -exponent);
    elif exponent == 0:
        return  1
    elif exponent == 1  :
        return  base
    elif ((exponent % 2) == 0):
        return squareAndMultiply(base * base,  exponent / 2);
    elif ((exponent % 2) != 0):
        return base * squareAndMultiply(base * base, (exponent - 1)
/ 2);
```

In [7]:
```
def encryptRSA(plaintext, e, n):
    return squareAndMultiply(plaintext, e) % n

cipher = encryptRSA(42, e, n)

print("42 encrypted is = {}".format(cipher))
```

```
42 encrypted is = 18166
```

In [8]:
```
def decryptRSA(cipher, d, n):
    return squareAndMultiply(cipher, d) % n
decrypted = decryptRSA(cipher, d, n)

print("cipher {} is  decrypted to be = {}".format(cipher, decrypte
d))
```

```
cipher 18166 is  decrypted to be = 42
```

3a)

In [9]:
```
def pollard(n, B):
    a = 2
    for j in range(2,B):
        a *= a**j % n
        d = math.gcd(a-1, n)
        if (d > 1 and d < n):
            return d
    return None

n = 1829
B = 5

result = pollard(n, B)
if not result: print("Det finnes ingen primtallsfaktor i N med B so
m 5")
else: print("Result B= {} gives factors {}".format(B,result))
```

```
Result B= 5 gives factors 31
```

3b)

In [10]:
```
# 1
n1 = 18779
B = 1
res = pollard(n1,B)

while(res is None):
    B+=1
    res = pollard(n1,B)

pmin1 = res-1
print("N1 =",n1)
print("B = ",B)
print("p-1 =",pmin1)

print("210 faktorisert er = 2 * 3 * 5 * 7")
print("Dermed har vi B = 7 som er garantert å finne en primtallsfak
tor i 18779")


#2

n2 = 42583

B = 1
res = pollard(n2,B)

while(res is None):
    B+=1
    res = pollard(n2,B)

pmin1 = res-1
print("\nN2 =",n2)
print("B = ",B)
print("p-1 =",pmin1)
print("96 faktorisert er = [2,2,2,2,2,3]")
print("2^5 = 32")
print("Dermed kan vi være sikker på at 32 gir oss primtallsfaktoren
97")
```

```
N1 = 18779
B =  7
p-1 = 210
210 faktorisert er = 2 * 3 * 5 * 7
Dermed har vi B = 7 som er garantert å finne en primtallsfaktor i
18779

N2 = 42583
B =  8
p-1 = 96
96 faktorisert er = [2,2,2,2,2,3]
2^5 = 32
Dermed kan vi være sikker på at 32 gir oss primtallsfaktoren 97
```

3c)

In [11]:
```
n = 6319
B = 1
res = pollard(n,B)

while(res is None):
    B+=1
    res = pollard(n,B)


print("Faktor i n({}) er {} med B={}".format(n, res, B))
```

```
Faktor i n(6319) er 71 med B=7
```

4)

In [12]:
```
def pollardRho(n,x,f):
    x = y = x
    d = 1

    i = 0
    while d == 1:
        i += 1
        x  = f(x)
        y  = f(f(y))
        d  = math.gcd(abs(x - y), n)

    if d == n:
        return ("failure", i)
    else:
        return (d, i)

f = lambda x : int(math.pow(x,2)+1)%n
```

4a)

In [13]:
```
x1 = 1
n = 851

resultRHO, i = pollardRho(n,x1,f)
print("Fant faktoren {} med {} iterasjoner".format(resultRHO, i))
```

```
Fant faktoren 37 med 4 iterasjoner
```

4b)

In [14]:
```
x1 = 1
n = 1517

resultRHO, i = pollardRho(n,x1,f)
print("Fant faktoren {} med {} iterasjoner".format(resultRHO, i))
```

```
Fant faktoren 37 med 4 iterasjoner
```

4c)

In [15]:
```
x1 = 1
n = 31861

resultRHO, i = pollardRho(n,x1,f)
print("Fant faktoren {} med {} iterasjoner".format(resultRHO, i))
```

```
Fant faktoren 151 med 5 iterasjoner
```

5a)

$$E(m1) \times E(m2) = E(m1 \times m2)$$

fordi:
$m_1^e \times m_2^e = (m_1 \times m_2)^e \mod n$

RSA har den egenskapen at produktet av to krypteringstekster er lik krypteringen av produktet fra de respektive plaintext-ene.

Eksempel under:

In [16]:
```
m1 = 4
m2 = 5

e = 5
n = 19043

# LEFT SIDE OF CONGRUENCE
m1_res = encryptRSA(m1, e, n)
m2_res = encryptRSA(m2, e, n)
individual_encrypt_Prod = m1_res*m2_res
individual_encrypt_Prod_mod_n = individual_encrypt_Prod % n

print("m1 encrypted = {}, m2 encrypted = {}".format(m1_res, m2_re
s))
print("product of m1^e * m2^e = {}".format(individual_encrypt_Pro
d))
print("m1^e * m2^e mod {} = {}".format(n, individual_encrypt_Prod_m
od_n))

productEncrypt = encryptRSA(m1*m2, e, n)
productEncrypt_mod_n = productEncrypt%n
print("(m1*m2)^e mod {} = {}".format(n, productEncrypt_mod_n))

print("{} congruent {} mod {}".format(individual_encrypt_Prod,produ
ctEncrypt_mod_n, n))
```

```
m1 encrypted = 1024, m2 encrypted = 3125
product of m1^e * m2^e = 3200000
m1^e * m2^e mod 19043 = 776
(m1*m2)^e mod 19043 = 776
3200000 congruent 776 mod 19043
```

5b)

"Because of this multiplicative property a chosen-ciphertext attack is possible. E.g., an attacker who wants to know the decryption of a ciphertext c ≡ me (mod n) may ask the holder of the private key d to decrypt an unsuspicious-looking ciphertext c′ ≡ cre (mod n) for some value r chosen by the attacker.

Because of the multiplicative property c′ is the encryption of mr (mod n). Hence, if the attacker is successful with the attack, they will learn mr (mod n) from which they can derive the message m by multiplying mr with the modular inverse of r modulo n."

-wikipedia

Altså på grunn av den multiplikative egenskapen så kan man lett legge til en egenlagd melding, og få et offer til å dekryptere den meldingen. Med den dekrypterte meldingen, så kan vi dekryptere andre meldinger med.

Se eksempel under:

In [17]:
```
#Public and private keys
e = 5
n = 19043
d=1877


c_a = 2

#our own cipher that we add into the mix
Ca = encryptRSA(c_a, e, n) % n

# C is the already encrypted cipher
C = encryptRSA(4, e, n) % n
print("Cipher text we want to decode is: ", C)

#product of two ciphers mod n
Cb = (Ca * C) % n

#Decrypt product of two ciphers mod n

Cb_decrypted = decryptRSA(Cb, d, n)

print("C_b decrypted is:", Cb_decrypted)
print("(C_b)^d = t * 2 mod n")
print("therefore ((C_b)^d)/2 = plaintext for other message")
```

```
Cipher text we want to decode is:  1024
C_b decrypted is: 8
(C_b)^d = t * 2 mod n
therefore ((C_b)^d)/2 = plaintext for other message
```

$C \equiv x^e$ og $x \equiv C^d$
Så $(x^e)^d \equiv x \mod n$
og derfor er $(C_b)^d = t * 2 \mod n$

Man kan altså dekryptere den originale meldingen C med å dele på 2

6a)

Forklar hvorfor vi kan skrive q−p= 2d, hvor d er et heltall.

oddetall - oddetall = partall

$p = 2c + 1$
$q = 2b + 1$

$p - q = 2c + 1 - (2b + 1) = 2(c - b)$

Dermed er:
$p - q = 2(c - b) = 2d$

6b)

Vis at n + d^2 er et kvadrattall

$n = p * q$

$q - p = 2d$

utleder d

$d = \frac{q-p}{2}$

$n + d^2$

=> $pq + (\frac{q-p}{2})^2$

=> $pq + \frac{(q-p)^2}{4}$

=> $\frac{4pq}{4} + \frac{(q-p)^2}{4}$

=> $\frac{(q-p)^2+4pq}{4}$

=> $\frac{p^2+2pq+q^2}{4}$

faktoriserer dette og får =>

$(\frac{p+q}{2})^2$

6c)

Vis hvordan vi kan faktorisere n hvis n+ d^2 er et kvadrattall.

Vi antar her at at d^2 er “liten nok”.

$n + d^2 = m^2$

$n = m^2 - d^2$

Bruker konjugatsetningen

$n = konjugatsetningen(m^2 - d^2)$
$n = (m + d)(m - d)$

Siden q > p, vil p og q se sånn her ut:

q = m + d
p = m - d

6d)

Faktoriser n= 152416580095517 med denne metoden.

n = 152416580095517

$$x^2 = \sqrt{n + d^2}$$
$$x = \sqrt{n} + d$$

Finner x ved å ta kvadratroten av N og runde til nærmeste hele tall

$$x = \sqrt{n} = \sqrt{152416580095517} = 12345710.999999838 \approx 12345711$$

prøver med d = 2

$$n = q \times p$$
$$n = (x + d) \times (x - d)$$
$$n = (12345711 + 2) \times (12345710 - 2)$$
$$n = 12345711 \times 12345709$$
$$n = 152416580095517$$

Vi finner tilbake til n, dermed stemmer p og q.

$$n = 12345711 \times 12345709$$

In [18]:
```
def factorizeN(n):
    x = round(math.sqrt(n))
    i = 1
    while True:
        res = (x*x) - (i*i)
        if(res == n):
            p = x-d
            q = x+d
            return (p, q)
        i += 1
        if(i == math.sqrt(n)+1): #not found
            break

n = 152416580095517
res = factorizeN(n)
print("Factors of N={} is {}".format(n,res))
```

```
Factors of N=152416580095517 is (12343834, 12347588)
```

In [ ]: