Schnorr

In [ ]:
```
from sage.all import *
```

In [ ]:
```
n = 100
bits = 64
c = 3
N = random_prime(2^bits-1,lbound=2^(bits-1)) * random_prime(2^(bits-1)-1,lbound=2^(bits-2))
L = N^c
Q = Primes()[:n]
```

In [ ]:
```
lnQ   = lambda n : QQ(log(RDF(n),2))
sqlnQ = lambda n : QQ(sqrt(log(RR(n),2)))
pZ    = lambda z : prod([q^e for (e,q) in zip(z,Q)])

vq    = [lnQ(q) for q in Q]
svq   = [sqlnQ(q) for q in Q]
```

In [ ]:
```
mQ = matrix(QQ,n,1,vq)
mZ = matrix(QQ,1,n,[0]*n)
mz = matrix(QQ,n,1,[0]*n)
mI = identity_matrix(QQ,n)
mS = diagonal_matrix(QQ,n,svq)

mt = matrix(QQ,1,1,[-lnQ(N)])
mT = matrix(QQ,1,n,[0]*n).augment(mt)

um = matrix(QQ,1,1,[1])
```

In [ ]:
```
#G = block_matrix(QQ,2,2,[mI,L*mQ,mZ,L*mt])
```

In [ ]:
```
#Gr = G.LLL()
```

In [ ]:
```
#for i in range(n):
#    l = Gr[i][:n]; s = round(sqrt(sum([a^2 for a in l]))); e = RDF(Gr[i][-1]/L)
#    print(l,s,e)
```

In [ ]:
```
#z = Gr[0][:n]
#e = 2^(RDF(Gr[0][-1]/L))
#print(z,e)
```

In [ ]:
```
def u_v(z):
    u = [0]*n ; v = [0]*n
    for k in range(n):
        if z[k] >= 0:
            u[k] = z[k]
        else:
            v[k] = -z[k]
    return (u,v)

#(uz,vz) = u_v(z)
#u = pZ(uz)
#v = pZ(vz)
#err = abs(u-N*v)
```

In [ ]:
```
#G_ = block_matrix(QQ,1,2,[mS , mQ])
```

```
G_ = block_matrix(QQ,1,2,[mI , L*mQ])
GG = block_matrix(QQ,2,2,[G_,mz,L*mT,L*um])
```

In [ ]:
```
GG.parent()
```

In [ ]:
```
GGr = GG.LLL()
```

In [ ]:
```
last_line = GGr[n]
```

In [ ]:
```
u = last_line[-1]/L
e = RDF(last_line[-2]/L)
z = last_line[:n]
print("erro =" , e, "\nz = ", z) if u == 1 else "Fail"
```

In [ ]:
```
(uz,vz) = u_v(z)
u = pZ(uz)
v = pZ(vz)
err = abs(u-N*v)
```

In [ ]:
```
print(RDF(u/(N*v)))
```

In [ ]:
```
print(err)
```

Implementação: https://github.com/lducas/SchnorrGate

In [ ]:
```
from sage.all import *

from fpylll import IntegerMatrix, SVP
import sys

def svp(B):
        A = IntegerMatrix.from_matrix(B)
        return SVP.shortest_vector(A)

def first_primes(n):
        p = 1
        P = []
        while len(P) < n:
                p = next_prime(p)
                P += [p]
        return P

def is_smooth(x, P):
        y = x
        for p in P:
                while p.divides(y):
                        y /= p
        return abs(y) == 1



# Test if a factoring relation was indeed found.
def test_Schnorr(N, n, prec=1000):
        P = first_primes(n)
        f = list(range(1, n+1))
        shuffle(f)
```

```
        # Scale up and round
        def sr(x):
                return round(x * 2**prec)

        diag = [sr(N*f[i]) for i in range(n)] + [sr(N*ln(N))]
        B = diagonal_matrix(diag, sparse=False)
        for i in range(n):
                B[i, n] = sr(N*ln(P[i]))


        b = svp(B)
        e = [b[i] / sr(N*f[i]) for i in range(n)]

        u = 1
        v = 1
        for i in range(n):
                assert e[i] in ZZ
                if e[i] > 0:
                        u *= P[i]**e[i]
                if e[i] < 0:
                        v *= P[i]**(-e[i])

        return is_smooth(u - v*N, P)

try:
        bits = int(sys.argv[1])
except:
        bits = 400

try:
        n = int(sys.argv[2])
except:
        n = 47

try:
        trials = int(sys.argv[3])
except:
        trials = 100


print("Testing Schnorr's relation finding algorithm with n=%d on RSA-moduli of %d bits, %d trials"%(n, bits, trials))

successes = 0
for i in range(trials):
        p = random_prime(2**(bits/2), false, 2**(bits/2-1))
        q = random_prime(2**(bits/2), false, 2**(bits/2-1))
        N = p*q
        success = test_Schnorr(N, n)
        successes += success
        print(success, end="\t")
        sys.stdout.flush()

print("\n%d Factoring Relation found out of %d trials"%(successes, trials))
```