```
import time

debug = True

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print a

def coppersmith_howgrave_univariate(pol, modulus, beta, mm, tt, XX):
    """
    Coppersmith revisited by Howgrave-Graham

    finds a solution if:
    * b|modulus, b >= modulus^beta , 0 < beta <= 1
    * |x| < XX
    """
    #
    # init
    #
    dd = pol.degree()
    nn = dd * mm + tt

    #
    # checks
    #
    if not 0 < beta <= 1:
        raise ValueError("beta should belongs in (0, 1]")

    if not pol.is_monic():
        raise ArithmeticError("Polynomial must be monic.")

    #
    # calculate bounds and display them
    #
    """
    * we want to find g(x) such that ||g(xX)|| <= b^m / sqrt(n)

    * we know LLL will give us a short vector v such that:
    ||v|| <= 2^((n - 1)/4) * det(L)^(1/n)

    * we will use that vector as a coefficient vector for our g(x)

    * so we want to satisfy:
    2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n)

    so we can obtain ||v|| < N^(beta*m) / sqrt(n) <= b^m / sqrt(n)
    (it's important to use N because we might not know b)
    """
    if debug:
        # t optimized?
        print "\n# Optimized t?\n"
        print "we want X^(n-1) < N^(beta*m) so that each vector is helpful"
        cond1 = RR(XX^(nn-1))
        print "* X^(n-1) = ", cond1
        cond2 = pow(modulus, beta*mm)
        print "* N^(beta*m) = ", cond2
        print "* X^(n-1) < N^(beta*m) \n-> GOOD" if cond1 < cond2 else "* X^(n-1) >=
 N^(beta*m) \n-> NOT GOOD"

        # bound for X
        print "\n# X bound respected?\n"
        print "we want X <= N^(((2*beta*m)/(n-1)) - ((delta*m*(m+1))/(n*(n-1)))) / 2
 = M"
        print "* X =", XX
        cond2 = RR(modulus^(((2*beta*mm)/(nn-1)) - ((dd*mm*(mm+1))/(nn*(nn-1)))) / 2
)
```

```
        print "* M =", cond2
        print "* X <= M \n-> GOOD" if XX <= cond2 else "* X > M \n-> NOT GOOD"

        # solution possible?
        print "\n# Solutions possible?\n"
        detL = RR(modulus^(dd * mm * (mm + 1) / 2) * XX^(nn * (nn - 1) / 2))
        print "we can find a solution if 2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n)"
        cond1 = RR(2^((nn - 1)/4) * detL^(1/nn))
        print "* 2^((n - 1)/4) * det(L)^(1/n) = ", cond1
        cond2 = RR(modulus^(beta*mm) / sqrt(nn))
        print "* N^(beta*m) / sqrt(n) = ", cond2
        print "* 2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n) \n-> SOLUTION WILL BE FOUND" if cond1 < cond2 else "* 2^((n - 1)/4) * det(L)^(1/n) >= N^(beta*m) / sqroot(n) \n-> NO SOLUTIONS MIGHT BE FOUND (but we never know)"

        # warning about X
        print "\n# Note that no solutions will be found _for sure_ if you don't respect:\n* |root| < X \n* b >= modulus^beta\n"

    #
    # Coppersmith revisited algo for univariate
    #

    # change ring of pol and x
    polZ = pol.change_ring(ZZ)
    x = polZ.parent().gen()

    # compute polynomials
    gg = []
    for ii in range(mm):
        for jj in range(dd):
            gg.append((x * XX)**jj * modulus**(mm - ii) * polZ(x * XX)**ii)
    for ii in range(tt):
        gg.append((x * XX)**ii * polZ(x * XX)**mm)

    # construct lattice B
    BB = Matrix(ZZ, nn)

    for ii in range(nn):
        for jj in range(ii+1):
            BB[ii, jj] = gg[ii][jj]

    # display basis matrix
    if debug:
        matrix_overview(BB, modulus^mm)

    # LLL
    BB = BB.LLL()

    # transform shortest vector in polynomial
    new_pol = 0
    for ii in range(nn):
        new_pol += x**ii * BB[0, ii] / XX**ii

    # factor polynomial
    potential_roots = new_pol.roots()
    print "potential roots:", potential_roots

    # test roots
    roots = []
    for root in potential_roots:
        if root[0].is_integer():
            result = polZ(ZZ(root[0]))
            if gcd(modulus, result) >= modulus^beta:
                roots.append(ZZ(root[0]))

    #
    return roots

############################################
# Test on Stereotyped Messages
##########################################
```

```
print "///////////////////////////////////"
print "// TEST 1"
print "/////////////////////////////////"

# RSA gen options (for the demo)
length_N = 1024  # size of the modulus
Kbits = 200      # size of the root
e = 3

# RSA gen (for the demo)
p = next_prime(2^int(round(length_N/2)))
q = next_prime(p)
N = p*q
ZmodN = Zmod(N);

# Create problem (for the demo)
K = ZZ.random_element(0, 2^Kbits)
Kdigits = K.digits(2)
M = [0]*Kbits + [1]*(length_N-Kbits);
for i in range(len(Kdigits)):
    M[i] = Kdigits[i]
M = ZZ(M, 2)
C = ZmodN(M)^e

# Problem to equation (default)
P.<x> = PolynomialRing(ZmodN) #, implementation='NTL')
pol = (2^length_N - 2^Kbits + x)^e - C
dd = pol.degree()

# Tweak those
beta = 1                                # b = N
epsilon = beta / 7                      # <= beta / 7
mm = ceil(beta**2 / (dd * epsilon))     # optimized value
tt = floor(dd * mm * ((1/beta) - 1))    # optimized value
XX = ceil(N**((beta**2/dd) - epsilon))  # optimized value

# Coppersmith
start_time = time.time()
roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)

# output
print "\n# Solutions"
print "we want to find:",str(K)
print "we found:", str(roots)
print("in: %s seconds " % (time.time() - start_time))
print "\n"

############################################
# Test on Factoring with High Bits Known
##########################################
print "///////////////////////////////////"
print "// TEST 2"
print "/////////////////////////////////"

# RSA gen
length_N = 1024;
p = next_prime(2^int(round(length_N/2)));
q = next_prime( round(pi.n()*p) );
N = p*q;

# qbar is q + [hidden_size_random]
hidden = 200;
diff = ZZ.random_element(0, 2^hidden-1)
qbar = q + diff;

F.<x> = PolynomialRing(Zmod(N), implementation='NTL');
pol = x - qbar
dd = pol.degree()

# PLAY WITH THOSE:
beta = 0.5                             # we should have q >= N^beta
epsilon = beta / 7                     # <= beta/7
```

```
mm = ceil(beta**2 / (dd * epsilon))    # optimized
tt = floor(dd * mm * ((1/beta) - 1))   # optimized
XX = ceil(N**((beta**2/dd) - epsilon)) # we should have |diff| < X

# Coppersmith
start_time = time.time()
roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)

# output
print "\n# Solutions"
print "we want to find:", qbar - q
print "we found:", roots
print("in: %s seconds " % (time.time() - start_time))
```