```
import time

debug = True

# display stats on helpful vectors
def helpful_vectors(BB, modulus):
    nothelpful = 0
    for ii in range(BB.dimensions()[0]):
        if BB[ii,ii] >= modulus:
            nothelpful += 1

    print nothelpful, "/", BB.dimensions()[0], " vectors are not helpful"

# 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 boneh_durfee(pol, modulus, mm, tt, XX, YY):
    """
    Boneh and Durfee revisited by Herrmann and May

    finds a solution if:
    * d < N^delta
    * |x| < e^delta
    * |y| < e^0.5
    whenever delta < 1 - sqrt(2)/2 ~ 0.292
    """

    # substitution (Herrman and May)
    PR.<u, x, y> = PolynomialRing(ZZ)
    Q = PR.quotient(x*y + 1 - u) # u = x*y + 1
    polZ = Q(pol).lift()

    UU = XX*YY + 1

    # x-shifts
    gg = []

    for kk in range(mm + 1):
        for ii in range(mm - kk + 1):
            xshift = x^ii * modulus^(mm - kk) * polZ(u, x, y)^kk
            gg.append(xshift)

    gg.sort()

    # x-shifts monomials
    monomials = []

    for polynomial in gg:
        for monomial in polynomial.monomials():
            if monomial not in monomials:
                monomials.append(monomial)

    monomials.sort()

    # y-shifts (selected by Herrman and May)
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            yshift = y^jj * polZ(u, x, y)^kk * modulus^(mm - kk)
            yshift = Q(yshift).lift()
            gg.append(yshift) # substitution

    # y-shifts monomials
    for jj in range(1, tt + 1):
        for kk in range(floor(mm/tt) * jj, mm + 1):
            monomials.append(u^kk * y^jj)
```

```
    # construct lattice B
    nn = len(monomials)

    BB = Matrix(ZZ, nn)

    for ii in range(nn):

        BB[ii, 0] = gg[ii](0, 0, 0)

        for jj in range(1, ii + 1):
            if monomials[jj] in gg[ii].monomials():
                BB[ii, jj] = gg[ii].monomial_coefficient(monomials[jj]) * monomials[
jj](UU,XX,YY)

    # check if vectors are helpful
    if debug:
        helpful_vectors(BB, modulus^mm)

    # check if determinant is correctly bounded
    if debug:
        det = BB.det()
        bound = modulus^(mm*nn)
        if det >= bound:
            print "We do not have det < bound. Solutions might not be found."
            diff = (log(det) - log(bound)) / log(2)
            print "size det(L) - size e^(m*n) = ", floor(diff)
        else:
            print "det(L) < e^(m*n)"

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

    # LLL
    BB = BB.LLL()

    # vectors -> polynomials
    PR.<x,y> = PolynomialRing(ZZ)

    pols = []
    for ii in range(nn):
        pols.append(0)
        for jj in range(nn):
            pols[-1] += monomials[jj](x*y+1,x,y) * BB[ii, jj] / monomials[jj](UU,XX,
YY)
        if pols[-1](xx,yy) != 0:
            pols.pop()
            break

    # find two vectors we can work with
    pol1 = pol2 = 0
    found = False

    for ii, pol in enumerate(pols):
        if found:
            break
        for jj in range(ii + 1, len(pols)):
            if gcd(pol, pols[jj]) == 1:
                print "using vectors", ii, "and", jj
                pol1 = pol
                pol2 = pols[jj]
                # break from that double loop
                found = True
                break

    # failure
    if pol1 == pol2 == 0:
        print "failure"
        return 0, 0

    # resultant
    PR.<x> = PolynomialRing(ZZ)
```

```
    rr = pol1.resultant(pol2)
    rr = rr(x, x)

    # solutions
    soly = rr.roots()[0][0]
    print "found for y_0:", soly

    ss = pol1(x, soly)
    solx = ss.roots()[0][0]
    print "found for x_0:", solx

    #
    return solx, soly


############################################
# Test
##########################################

# RSA gen options (tweakable)
length_N = 2048
length_d = 0.27

# RSA gen (for the demo)
p = next_prime(2^int(round(length_N/2)))
q = next_prime(round(pi.n()*p))
N = p*q
phi = (p-1)*(q-1)
d = int(N^length_d)
if d % 2 == 0: d += 1
while gcd(d, phi) != 1:
    d += 2
e = d.inverse_mod((p-1)*(q-1))

# Problem put in equation (default)
P.<x,y> = PolynomialRing(Zmod(e))
A = int((N+1)/2)
pol = 1 + x * (A + y)

# and the solutions to be found (for the demo)
yy = (-p -q)/2
xx = (e * d - 1) / (A + yy)

#
# Default values
# you should tweak delta and m. X should be OK as well
#
delta = 0.27              # < 0.292 (Boneh & Durfee's bound)
X = 2*floor(N^delta)      # this _might_ be too much
Y = floor(N^(1/2))        # correct if p, q are ~ same size
m = 7                     # bigger is better (but takes longer)
t = int((1-2*delta) * m)  # optimization from Herrmann and May
# Checking bounds (for the demo)
print "=== checking values ==="
print "* |y| < Y:", abs(yy) < Y
print "* |x| < X:", abs(xx) < X
print "* d < N^0.292", d < N^(0.292)
print "* size of d:", int(log(d)/log(2))

# boneh_durfee
print "=== running algorithm ==="
start_time = time.time()
solx, soly = boneh_durfee(pol, e, m, t, X, Y)

# Checking solutions (for the demo)
if xx == solx and yy == soly:
    print "\n=== the solutions are correct ==="
else:
    print "=== FAIL ==="

# Stats
print("=== %s seconds ===" % (time.time() - start_time))
```