Lenstra's elliptic curve factorization algorithm

Repeated duplication method. To compute nP the obvious method is to apply n times the group law, i.e. $nP = P + \overset{n \text{ times}}{\cdots} + P$

```
def easy_mult(n,P):
    result = '0'
    for i in range(n):
        result = g_l( P, result )
    return result
```

But this is useless whn n is very large, say hundred of digits.

The following program applies the analog of the repeated squaring alogirthm in $\mathbb{F}_p$. It is actually the same algorithm changing the multiplicative notation by the additive notation.

```
def mult_2(n,P):
    result = '0'
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l( pow_2P, result )
        n //=2
        pow_2P = g_l( pow_2P, pow_2P )
    return result
```

Comparing both algorithm one have to reject the first one even for not very high values of n

```
E = EllipticCurve(GF(1000000007),[-6,5])
P = E([2,1])

a = Mod(-6,5)
b = Mod(5,5)

P=[2,1]
time easy_mult(10^6,P)
time mult_2(10^6,P)
```

gives

```
Time: CPU 8.97 s, Wall: 9.25 s
Time: CPU 0.00 s, Wall: 0.00 s
```

Factorization. In principle it is not possible to define an elliptic curve over a ring if we can save the group law. In fact the following line in Sage

```
E = EllipticCurve(GF(10403),[-6,5])
```

(note that 10403 is not prime) raises the error

```
ValueError: the order of a finite field must be a prime power
```

Let us see in an example what happens when we apply our function to add points in the ring $\mathbb{Z}/10403\mathbb{Z}$.

Example:

```
#
# Example P=(0,1) y^2= x^3+x+1, n = 10403
#

P= [0,1]
a= Mod(1, 10403)
print mult_2( factorial(7), P )

def mult_2(n,P):
    result = 'O'
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l( pow_2P, result )
        n //=2
        pow_2P = g_l( pow_2P, pow_2P )
    return result

def g_l( P, Q ):
    if P == 'O':
        return Q
    if Q == 'O':
        return P

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return 'O'

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        m = (3*P[0]^2+a)/2/P[1]
    else:
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1]]
```

Introducing at the beginning of the definition of the function the sentnece

```
print P,Q
```

we learn that the error appears when adding

```
[9696, 506] [7878, 10200]
```

The reason is that when computing the slope m we have to invert `Q[0]-P[0]` $= -1818$ and this is not possible because 1818 and $n = 10403$ are not coprime.

Lenstra elliptic curve factorization It is the analog of Pollard's $p-1$ method. It consists in computing $1!P, 2!P, \ldots, B!P$ in an elliptic curve $E \pmod n$. If an error arises in the group law then it can be employed to get a factor of n. The power of the method is based on the fact that if the the factor is trivial or no error appear one can easily change the elliptic curve. In some sense is like a Pollard's $p-1$ method with varying abelian gropus.

Firstly we have to hack the group law to detect the cases in which the group law is not well-defined. We employ the following notation for the points on E. The last case is the output corresponding to an error in the group law.

```
#
# "Normal" points [x,y,1]
# Point at infinity [0,1,0]
# Fake points [0,0,d] with d not coprime to the modulus.
#
```

The modified group law function is:

```
#
# Group law
#
def g_l_l( P, Q, a ):
    if P[2] != 1:
        if P[1]==1:
            return Q
        return P
    if Q[2] != 1:
        if Q[1]==1:
            return P
        return Q

    if (P[0] == Q[0]) and (P[1] == -Q[1]):
        return [0,1,0]

    if (P[0] == Q[0]) and (P[1] == Q[1]):
        if P[1].is_unit()==False:
            return [0,0,P[1]]
        m = (3*P[0]^2+a)/2/P[1]
    else:
        if (Q[0]-P[0]).is_unit()==False:
            return [0,0,Q[0]-P[0]]
        m = (Q[1]-P[1])/(Q[0]-P[0])
    x3 = m^2-P[0]-Q[0]
    return [x3,m*(P[0]-x3)-P[1],1]
```

and the modified multiplication function is:

```
#
# Same multiplication routine
# changing 0 and g_l by g_l_l
#
def mult_2_l(n,P,a):
    result = [0,1,0]
    pow_2P = P
    while n!=0:
        if (n%2)==1:
            result = g_l_l( pow_2P, result, a )
        n //=2
        pow_2P = g_l_l( pow_2P, pow_2P, a )
    return result
```

For instance, the result of

```
g_l_l([9696, 506,1],[7878, 10200,1],1)
```

is now

```
[0, 0, -1818]
```

We integrate this functions in Lenstra's algorithm. We use $y^2 = x^3 + ax + 1$ as vaying elliptic curve, because for any value of a the point $P = (0, 1)$ (that we take as starting point) is on it.

```
#
# Lenstra's algorithm
#
def lenstra(n,bound_a,bound_b):
    if is_prime(n):
        print n,'is prime'
        return n
    if n%2==0:
        return 2
    if n%3==0:
        return 3

    for a in range(bound_a):
        # Consider only elliptic curves
        if Mod(4*a^3+27,n)==0:
            continue

        f_point = [Mod(0,n),Mod(1,n),1]
        for b in range(bound_b):
            # compute factorial
            f_point = mult_2_l(b+1,f_point,a)
            if f_point[2]==0:
                break
            if f_point[2]>1:
                print a,b
                return gcd( f_point[2], n)
    print 'Increase the values of bound_a and bound_b'
```

The parameters `bound_a` and `bound_b` in the function `lenstra` give the upper bound for a, the number of elliptic curves, and B, the number of multiplications $n!P$. Of course the algorithm is stronger but slower taking large values of `bound_a` and `bound_b`.

For example

```
# therea are twenty-one 3's in the first example
time print lenstra(133333333333333333333,200,100)
time print lenstra( (10^8+7)*(9*10^8+11),200,100)
time print lenstra( 10^20+699,200,100)
time print lenstra( 10^30+427,300,1000)
```

In a standard computer the results have been

```
43 78 -> 4363363
Time: CPU 1.02 s, Wall: 1.04 s
74 30 -> 900000011
Time: CPU 1.63 s, Wall: 1.66 s
156 20 -> 32935987639
Time: CPU 3.70 s, Wall: 3.74 s
223 756 -> 852759062050499
Time: CPU 89.23 s, Wall: 90.27 s
```

The algorithm has its limitations,

```
time print lenstra( next_prime(10^18)*next_prime(10^19),300,3000)
```

produces

```
Increase the values of bound_a and bound_b
None
Time: CPU 421.03 s, Wall: 423.92 s
```

Appendix. It is possible to program Lenstra's algorithm using only Sage functions but it requires some deeper knowledge of Python and Sage. Essentially one cheats Sage forcing it to consider $\mathbb{Z}/n\mathbb{Z}$ as a field and one employs the exception handling in Python to redirect the flow after an error.

The following program was written by Professor W. Stein, the lead developer of Sage, and included in his book *Elementary Number Theory: Primes, Congruences, and Secrets*.

Note that it introduces a twist with respect to our previous program, now the elliptic curve is chose at random.

```
def ecm(N, B=10^3, trials=10):
    m = prod([p^int(math.log(B)/math.log(p))
              for p in prime_range(B+1)])
    R = Integers(N)
    # Make Sage think that R is a field:
    R.is_field = lambda : True
    for _ in range(trials):
        while True:
            a = R.random_element()
            if gcd(4*a.lift()^3 + 27, N) == 1: break
        try:
            m * EllipticCurve([a, 1])([0,1])
        except ZeroDivisionError, msg:
            # msg: "Inverse of <int> does not exist"
            return gcd(Integer(str(msg).split()[2]), N)
    return 1
```

Now

```
time print ecm(133333333333333333333,200,100)
time print ecm( (10^8+7)*(9*10^8+11),200,100)
time print ecm( 10^20+699,200,100)
time print ecm( 10^30+427,300,1000)
```

gives

```
4363363
Time: CPU 0.79 s, Wall: 0.89 s
100000007
Time: CPU 0.90 s, Wall: 1.00 s
3036192541
Time: CPU 0.96 s, Wall: 1.06 s
1
Time: CPU 47.94 s, Wall: 49.03 s
```

The random choice of the elliptic curve can give different results when running the program several times.