NP and asymmetry

Jeffrey Goldberg

August 26, 2024

This document is just sort of my workbook for performing some easy and hard computations. It contains math and sketchy partial explantions of that math for things that we have not gotten to yet.

This notebook is using SageMath

[48]:
```
version()
```

[48]:
```
'SageMath version 10.2, Release Date: 2023-12-03'
```

1 Factoring

First we pick two random primes. (I don't want to generate new primes each time I update this document, so I comment those out.)

[1]:
```
dhBits = 220
# p = random_prime(2^(dhBits//2), lbound = 2^(dhBits//2 - 1))
p = 868112830765445632873217196988651
print(f'p is {p}')
# q = random_prime(2^(dhBits//2), lbound = 2^(dhBits//2 - 1))
q = 1180775137873020977354442912336269
print(f'q is {q}')
```

```
p is 868112830765445632873217196988651
q is 1180775137873020977354442912336269
```

The choice of 110 bits for each prime was through trial and error to see what took about a short enough time to actually factor (as it has to happen every time I generate the document), but long enough to illustrate that it does take time.

[2]:
```
time p * q
```

```
CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 5.01 µs
```

[2]:
```
1025046047436407593990706183629376939000352221561805965005888683119
```

[3]:
```
N = p * q
print(f'N: {N}')
```

```
N: 1025046047436407593990706183629376939000352221561805965005888683119
```

1.1 Counting digits

Let's see how many digits that is. Some people might want to save the decimal representation into a string and then count the characters in the string, but I prefer to think of it as a math problem.

It's pretty much the base 10 logarithm rounded up. For example $\log_{10} 99 \approx 1.9956$. If we round up, we get the correct number of digits, 2. Now if we want to know how many digits there are in 102, we do the same. $\log_{10} 102 \approx 2.0086$. We round up and get 3, which is the correct number of digits in 102.

That neat little trick doesn't work when we have a number like 100 which is exactly a power of 10, and $\log_{10}(100) = 2$. Rounding up, would leave us at 2, but one hundred is a three digit number. So what we have to do is add 1 to our logarithm and round down.

We use '$\lfloor a \rfloor$' as the notation for the floor of a, the function of rounding a down.

$$D(x) = \lfloor \log_{10}(x) + 1 \rfloor$$

This can work for any base, not just base 10.

$$D(x, b) = \lfloor \log_b(x) + 1 \rfloor$$

And if you think I am spending too much time on this, do recall (for those who were participating at the time) that I said that the more comfortable you get with dealing with logarithms, the easier the rest of the book will be.

[49]:
```
def digits(x: int, base:int = 10) -> int:
    return floor(log(x, base) + 1)

print(f'Digits in N: {digits(N)}')
```

```
Digits in N: 67
```

[5]:
```
time factor(N)
```

```
CPU times: user 8.47 s, sys: 20 ms, total: 8.49 s
Wall time: 8.61 s
```

[5]:
```
86811283076544563287321719698865 1 * 11807751378730209773544429123362 69
```

When not commented out, that `time factor(N)` produced results like

```
CPU times: user 8.41 s, sys: 168 ms, total: 8.58 s
Wall time: 8.59 s
86811283076544563287321719698865 1 * 11807751378730209773544429123362 69
```

So on my machine, I can factor a 67 digit number in just under 9 seconds. The machine that I am running this on is 2023 MacBook Pro with an M2 Pro chip and 16 gigabytes of universal memory.

1.2 Toy RSA

The RSA algorithm used for these must examples not to be used for actual cryptography, even if the keys were large enough.

[6]:
```
Z_N = Zmod(N)
# phiN = lcm((p - 1), (q - 1))
phiN = (p - 1) * (q - 1)
print(f'φ(N): {phiN}')
pub_exp = 65537 # 0x10001
d = inverse_mod(pub_exp, phiN)
print(f'secret key (d): {d}')
```

```
φ(N): 10250460474364075939907061836293748901123835830951957373457793582 00
secret key (d):
10212609925288040931019601181024464569615344928544304083188529104 73
```

[7]:
```
# just some checks
if gcd(pub_exp, phiN) == 1:
    print(f'Good: gcd(e, φ(N)) = 1')
else:
    print(f"Bad news: gcd(e, φ(N)) = {gcd(pub_exp, phiN)}")

if gcd(d, phiN) == 1:
    print(f'All is right with the world: gcd(d, φ(N)) = 1')
else:
    print(f"Shouldn't happen: gcd(d, φ(N)) = {gcd(d, phiN)}")
```

```
Good: gcd(e, φ(N)) = 1
All is right with the world: gcd(d, φ(N)) = 1
```

The public exponent, `pub_exp`, is part of the public key, along with N, but it is not unique to the key holder. It is contrived to be prime and have only two 1-bits in its binary representation.

[8]:
```
print(f'pub_exp = {pub_exp} (0b{pub_exp:b})')
print(f'pub_exp {"is" if is_prime(pub_exp) else "is not"} prime')
```

```
pub_exp = 65537 (0b10000000000000001)
pub_exp is prime
```

[9]:
```
# m = randint(0, 2^256)
m = 6183313904903929406795468086412827546420107099893625602925575758 44
m = Z_N(m)
print(f"message (m): {m}")

c = m^pub_exp

print(f'ciphertext (c): {c}')
```

```
message (m): 6183313904903929406795468086412827546420107099893625602925575758 44
ciphertext (c):
```

```
560345466981168561903713542978833062209067740909728572604689354426
```

Decryption is simply $m = c^d \pmod N$

[10]:
```
m1 = c^d
if (m1 == m):
    print("decryption worked")
else:
    print("uh-oh")
```

```
decryption worked
```

There is a thing about totients, $\varphi(N)$ that is crucial to all why this works.

Euler's totient theorem:

$$a^{\varphi(N)+1} \equiv a \pmod N$$

This also holds for 1 plus any integer multiple k of $\varphi(N)$, so

$$a^{k\varphi(N)+1} \equiv a \pmod N$$

Our private key, d has been defined to be the modular inverse of the public exponent, e modulo $\varphi(N)$. So when we multiply d time e we get a result that it one more than a multiple of $\varphi(N)$.

$$ed = 1 \pmod{\varphi(N)}$$
$$ed = 1 + k\varphi(N)$$

Let's just confirm this with our example.

[50]:
```
ed = pub_exp * d
if ( ed % phiN == 1):
    print('ed = 1 (mod φ(N))')
else:
    print('Shucky darn! ed mod φ(N) != 1')

k = (ed - 1) // phiN
# print(f'k = {k}')
if ed ==  (k * phiN) + 1:
    print(f'ed = {k} * φ(N) + 1')
else:
    print('Crap. ed != 1 + kφ(N)')
    print(f'{"k:":>11}  {k}')
    print(f'{"ed:":>11}  {ed}')
    print(f'{"1 + kφ(N):":>11}  {1 + k*phiN}')
    print(f'{"1 + φ(N):":>11}  {1 + phiN}')
    print(f'  gcd(ed, 1 + φ(N)) = {gcd(ed, 1 + phiN)}')
```

```
ed = 1 (mod φ(N))
ed = 65295 * φ(N) + 1
```

Our ciphertext c is defined as $m^e \pmod N$

$$\begin{aligned} c^d &= (m^e)^d \pmod N \\ &= m^{ed} \pmod N \\ &= m^{k\varphi(N)+1} \pmod N \\ &= m \pmod N \end{aligned} \tag{1}$$

2 Discrete Logarithm Problem

As I want to demonstrate the discrete logarithm problem, I will need to create a multiplicative group of a size I have a chance of breaking. These need a prime of the right sort and a generator.

Not any prime will do, either. We first need to fine a Sofie Germain prime, which is a prime q for which $2q + 1$ is also prime. So in

$$p = 2q + 1 \qquad \text{(where both } p \text{ and } q \text{ are prime)}$$

p is called a safe prime, and q is called a Sofie Germain prime. (Germain used these in her important progress on Fermat's *Last* Theorem. Fermat's *Little* Theorem is far more relevant to us, but historical asides weirdly connect things.)

Anyway, I use openssl to create this.

I will use openssl to create DH parameters of an easily breakable size

```
% openssl dhparam -text 112
Generating DH parameters, 112 bit long safe prime, generator 2
This is going to take a long time
.........++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*
    PKCS#3 DH Parameters: (112 bit)
        prime:
            00:9d:b3:ef:0f:f3:d0:1f:4d:36:e4:54:5f:27:4b
        generator: 2 (0x2)
-----BEGIN DH PARAMETERS-----
MBQCDwCds+8P89AfTTbkVF8nSwIBAg==
-----END DH PARAMETERS-----
```

That did not actually take a long time, as I am only asking for a 112 bit safe prime. The line with the `.`, `+`, and `*` is telling me about the progress of primality tests. There are fast tests, which will eliminate most non-primes; and there are slower tests, which will identify non-primes that the fast tests might miss, and there are even slower tests that rule out more. While these are polynomial time, they are slow.

Let's set p to what we got from openssl, and double check that it is a safe prime

[51]:
```
p = 0x009db3ef0ff3d01f4d36e4545f274b
print(f'p = {p}')

if p in Primes():
```

```
    print('p is prime')
    q = (p - 1) // 2
    if q in Primes():
        print('p is a safe prime')
    else:
        print('p is not a safe prime')
else:
    print('p is not prime')
```

```
p = 319859413506597440230382371439597 9
p is prime
p is a safe prime
```

We are going to use this prime to create a Galois Field, F. In everything that follows, there is an implicit "(mod p)" even though it isn't being written out.

(The reasons for using a Galois Field instead of using Zmod(p).unit_group is because I don't know how to work with numbers as members)

[13]:
```
F = GF(p)
```

Now we also have a generator, 2, from openssl.

[14]:
```
g = F(2)
```

While I could just say `g = 2`, I get some really nice magic from the Sage type system by saying `g = F(2)`. It will make g not just the integer 2, but it will make it a member of the field F we created from the safe prime p.

Now that we have our group set up, we will follow the time honored tradition of using a for Alice's secret, and b for Bob's. We will use A for a public key that Alice can share and B for Bob's. Through accidents of history it is conventional to use "little a" and "little b" for the private keys and "big A" and "big B" for the public keys when talking about schemes that are based on Diffie-Hellman key exchange. Do *not* assume that developers are familiar with that convention. So in our SRP implementations we refer to these ephemeral keys with names like "`ephemeralClientSecret`" and "`ephemeralClientPublic`".

[15]:
```
# a = F.random_element()
a = F(172984208477251475262671378417949 9)
print(f"Alice's secret is {a}")

# b =  F.random_element()
b = F(210363586211589175354367618783879 5)
print(f"Bob's secret is {b}")
```

```
Alice's secret is 172984208477251475262671378417949 9
Bob's secret is 210363586211589175354367618783879 5
```

Alice's private key, a, is picked as a random element of our field F. It really is just a random number less than p. Bob's private key, b, is picked the same way.

Again, I don't want to actually generate new random numbers each time I rerun this document, so I have commented those out and just hardcoded values that I got on an earlier run.

Let's time how long it takes to compute $A = g^a \pmod{p}$

[52]:
```
time g^a
```

```
CPU times: user 19 µs, sys: 4 µs, total: 23 µs
Wall time: 24.8 µs
```

[52]:
```
114220353520382243805938148409115 9
```

So that is a few micro seconds.

[53]:
```
A = g^a
print(f"Alice's public A is {A}")
```

```
Alice's public A is 114220353520382243805938148409115 9
```

Now let's look at how long it takes to compute Alice's secret. That is if we know A, g, and p in $A = g^a \pmod{p}$, how long does it take to compute $a = \log_g A \pmod{p}$? This is the opposite of exponentiation, and so it is the logarithm. Because the numbers we are using are all whole numbers, this is a discrete logarithm.

[18]:
```
# time A.log()
```

When I didn't comment out the above, I got something like

```
CPU times: user 9.26 s, sys: 120 ms, total: 9.38 s
Wall time: 9.36 s
172984208477251475262671378417 9499
```

So that took about 10 seconds (in the particular run I am writing about. It varies from trial to trial.)

2.1 Diffie-Hellman key exchange

Now let's see how properties of exponents that everyone learned in high school and have since forgotten allows us to turn all of this into a key exchange mechanism.

We already have a public key A for Alice, which is generated from her secret, a and the group parameters g and p. We need the same for Bob

[54]:
```
B = g^b
print(f"Bob's public B is {B}")
```

```
Bob's public B is 30385099409182021123146495156315 89
```

Now recall (or relearn) that

$$(x^n)^m = x^{mn}$$

Let's look at an example in which we set $x = 5, n = 2, m = 3$. So $(5^2)^3 = 5^{3 \cdot 2} = 5^6$.

$$(5^2)^3 = 25^3 = 15625$$
$$5^{2 \cdot 3} = 5^6 = 15625$$

And keep in mind that $A = g^a$ and $B = g^b$. So if Alice sends Bob A and Bob sends B to Alice, then Bob (who is the only one who knows b) can compute A^b.

[55]:
```
kb = A^b
```

Meanwhile, Alice (who is the only one who knows a) can compute B^a.

[56]:
```
ka = B^a
```

Now we look at those properties of exponents to take a closer look at the key that Bob computes

$$\begin{aligned} k_b &= A^b \\ &= \left(g^a\right)^b \\ &= g^{ba} \\ &= g^{ab} \end{aligned}$$

and the key that Alice computes

$$\begin{aligned} k_a &= B^a \\ &= \left(g^b\right)^a \\ &= g^{ab} \end{aligned}$$

From this we see that k_a and k_b both should equal g^{ab}.

[57]:
```
if kb == ka:
    print(f"Hurrah! Alice and Bob both computed the same key {kb}")
else:
    print("I (JPG) screwed up this demo somewhere")
```

```
Hurrah! Alice and Bob both computed the same key
21629751047973014482000017133319858
```

As a note for later, the randomness properties of numbers computed this way aren't quite right, but we can smooth it out by taking an HMAC of it with a non-secret HMAC key. HKDF was actually invented for the problem. But in the simple case, it just wraps HMAC, and the way I use HMAC here what HKDF would end up doing anyway.

[23]:
```
import hashlib
import hmac
```

In order to hash the integers `ka` or `kb` we need turn them into bytes. You would think that there would be a nice easy way to do that. Perhaps there is, but I haven't found it.

[58]:
```
# There has to be a better way
# convert kb into a form that can be hashed.
ikb = int(kb)
bikb = ikb.to_bytes((ikb.bit_length() + 7) // 8, byteorder="big")
```

We will use HMAC with SHA256 with the key being the bytes of the non-secret key `"Derive from DH key"`

[59]:
```
h = hmac.new(b'Derive from DH key', digestmod=hashlib.sha256)

# hash it
h.update(bikb)
kb_digest = h.digest()

# Convert hash digest to python integer
derived_key = int.from_bytes(kb_digest, byteorder='big', signed = False)

print(f'Key: {derived_key:x}')
```

```
Key: 671d81a11cdf197368766b2a26e48306735b0f8a18635c685dc98cf418192ff9
```

Alice needed her secret to compute the common key, and Bob needed his secret, but someone listening into the communication should not be able to compute either secret from seeing A, B and the group parameters (g and p).

Well, of course they could because I didn't pick a large enough p, but if we pick a p of around 3072 bits, then it should take very roughly about as much work to for an attacker to compute a or b as it would take to brute force a 112 bit symmetric key.

I need to note that this is toy DHKE meant only to illustrate how a (presumed) NP problem can be used in cryptography. Turning this into a secure construction takes a great deal of other work. And most notably, the scheme illustrated above may protect against someone who can eavesdrop on the communication between Alice and Bob, but this is insecure against an attacker who could interfer with the communication between Alice and Bob.

2.2 DLP over elliptic curves

The integer group that we set up with our 112 bit safe prime is not the only cyclic group that can be used to make use of the Discrete Logarithm Problem. There are other suitable groups. I will create a toy elliptic curve that has 40-bit keys and offers at most 20-bit security. How I picked the particular parameters is described much later.

The group is defined by a curve (which is a set of points) and a special point that will act as a generator. But to make things a bit more complicated, the x and y values for the points on the curve are defined over an integer field. So we will be taking a prime, which I will call `p40` from the standards to define a finite field, `FF` as our first step.

[61]:
```
# Finite field prime
# p256 = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
# p112 = 0xdb7c2abf62e35e668076bead208b
```

```
p40 = 872078115079
FF = GF(p40)
```

An elliptic curve is of the form

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

p is our p40 chosen above. We need a and b to be able to define the curve.

[62]:
```
# a112 = FF(0x6127c24c05f38a0aaaf65c0ef02c)
# b112 = FF(0x51def1815db5ed74fcc34c85d709)

a40 = FF(522510265620)
b40 = FF(189830553521)
```

We can now define the curve.

[63]:
```
EC = EllipticCurve([a40, b40])
```

There is also a standard point, G, on the curve that acts as our generator. It has an x coordinate and a y coordinate.

[64]:
```
# Base point (x, y)
gx = FF(842965367165)
gy = FF(470264993162)

G = EC(gx, gy)
```

Curve "order" is talked about in different ways The order of a point member of a group is not the same as the order of the group as a whole. Different statements of the standards use the term in different ways.

So following some versions, I will use n as the order of the base-point, and h for the cofactor. These can be computed from what we already have, but they are useful numbers to have around, and they are typically included in standards.

[66]:
```
h40 = 2
n40 = 436039332233
```

We run a few consistency checks that group parameters we are told about actually conform to the group.

[67]:
```
if EC.order() == n40 * h40: print("|EC| = n * h")
else:
    print('I thought "order" and "cardinality" mean the same thing in this␣
 ↪context')

if G.order() == n40: print('|G| = n')
else:
    print(f' Out of order. |G| ({G.order()}) != n40 ({n40})')
```

```
|EC| = n * h
|G| = n
```

2.3 Generalized Discrete Logarithm Problem (GDLP)

Groups have a single operation that have to have certain properties. It doesn't really matter if we call the operation multiplication or addition. The group operation that we used in $\mathbb{Z}_p$ was ordinary multiplication except that it was all on modulo p. And so it was very natural to just use the notation for ordinary multiplication and exponentiation when looking at those.

For historical reasons, the group operation in elliptic curves is called "point addition." And there is a special way of adding points (its a fairly complicated computation), and so that is the way these things are characterized in Elliptic curves. But really, in both cases we have an operation and a set of things (integers in the first case, points of an elliptic curve in the latter case) and we have an operation over those things that give us a finite cyclical group of the right sort.

Another convention is that points on an elliptic curve are typically given variable names like P, Q, R, etc.

Operation	Integer group	Elliptic curve group
Group operation	modular multiplication ab	point addition $P+Q$
repeated	exponentiation x^n	scalar multiplication dP
from generator	$A = g^a$	$P = pG$
logarithm	Find a given A, g $\log_g A$	find p given P, G `discrete_log`(P, G)

All of the computations except for the logarithm one are easy. But the logarithm problems are (presumed) hard.

The *Generalized* discrete logarithm problem (GDLP) covers both cases (and more). These days, now that elliptic curve cryptography has pretty much replaced integer field DLP, the term "DLP" is used for both. But back when elliptic curve cryptography was first becoming a thing, people distinguished more between the two terms.

Now that we have the machinery up, we will talk about Penelope and Quintin.

[69]:
```
# p = randint(2, p40)
p = 266038276613
print(f"Penelope's private scaler is {p}")
```

```
Penelope's private scaler is 266038276613
```

Remember that G is a point and p is a number. pG just means $G + G + G + \cdots + G$ with p as the number of _G_s in that addition.

While point addition is a weird things, there are ways of speeding up repeated point addition that means that computing `p * G` is really fast.

[33]:
```
time p * G
```

```
CPU times: user 597 µs, sys: 267 µs, total: 864 µs
Wall time: 2.1 ms
```

[33]:
```
(714049738717 : 803151079992 : 1)
```

Now we continue setting up public and private keys.

[70]:
```
P = p * G # Penelope's public point
print(f"Penelope's public point:\n\tx: {P[0]}\n\ty: {P[1]}")

# q = randint(2, p40)
q = 856017483482
print(f"Quintin's private scaler is {q}")

Q = q * G
print(f"Quintin's public point:\n\tx: {Q[0]}\n\ty: {Q[1]}")
```

```
Penelope's public point:
        x: 714049738717
        y: 803151079992
Quintin's private scaler is 856017483482
Quintin's public point:
        x: 724364265403
        y: 770301686565
```

Penelope can use her secret p along with Quintin's public key Q to compute a point. Quintin can compute the same key using his secret and Penelope's public key. Again, these computations are fast.

[71]:
```
K_p = p * Q
K_q = q * P

if K_q == K_p:
    print("Penelope and Quintin computed the same point:")
    print(f'\tx: {K_q[0]}\n\ty: {K_q[1]}')
else:
    print("uh-oh. EC demo did not work.")
```

```
Penelope and Quintin computed the same point:
        x: 412011888668
        y: 543936543559
```

And this all works because the scaler (integer) multiplication behaves as expected. Even though Penelope nor any eavesdropper never learn Quintin's private integer, p, when Penelope computes $K_p = pQ$ that is the same as $K_p = pqG$ because $Q = qG$.

$$K_p = pQ = p(qG) = (pq)G$$
$$K_q = qP = q(pG) = (qp)G$$

And so the keys that they each compute are the same, even though no secret information needed to be transmitted.

An attacker with knowledge of one of the public keys, say P and the (public) group parameters (curve definition and base point G) would be able to learn the shared K if they could solve

$$P = pG$$

for little p.

The 20-bit security I have for this curve is within easy breaking on this machine, but it still illustrates that finding little p given G and P is much harder than computing P given p and G.

[36]:
```
# time discrete_log(P, base = G, ord = n40, operation = '+')
```

Running

```
time discrete_log(P, base = G, ord = n40, operation = '+')
```

Yielded

```
CPU times: user 24.8 s, sys: 536 ms, total: 25.4 s
Wall time: 27.4 s
266038276613
```

Contrast that 25 seconds with few milliseconds to compute `p*G`.

As with the integer field stuff. There is a lot more to actual turn this into a secure system and protocol. This all is only used to illustrate how a (presumed) non-deterministic polynomial-time (NP) can be turned into (what can be further developed as) a public key system. But mostly what this is to illustrate is that we can look at things like cyclic finite fields and see that properties that hold of one (integer fields with the right sort of primes) can hold of another (elliptic curves of the right sort.)

There are many reasons to prefer elliptic curve cryptography over integer groups for this stuff. One of them is that to get a security at around the level of 128-bits, we need keys of 3072 bits with integers, but only 256 bits for ECC.

3 Appendix: Making my demo curve

I really want to run the `discrete_log()` operation on my machine with its mere 8 GB of memory. So I wish to have a curve with about 20 bit security. Because I would like the curve to have the other properties of curves of this sort (at least to the standards of 1998) I am (mostly) following Annex A.3.2 of ANSI X9.62-1998. (I will skip some of the checks)

I need a 40 bit prime for the integer field that the curve is defined over. I pick one at random, and define the field. (Don't worry about the difference between a field and a group, but this is the first time we need a field. All fields are groups, but not all groups are fields.)

[72]:
```
# p40 = random_prime(2^40, lbound = 2^39)
p40 = 872078115079

F40 = GF(p40)
```

The standards parameterize creation in terms of strength using something called '$r_{\min}$'. Things like the size of the prime are supposed to be defined in terms of it. I am going the other way around.

Because we are working with such small numbers, we can eyeball things instead of looping through paramater creations to see if we get a group with a large subgroup

[73]:
```
r_min = 4 * sqrt(p40)

l_max = 255 # for "nearly prime" computation

# just pick a and b at random

# a40 = F40.random_element()
a40 = F40(522510265620)
# b40 = F40.random_element()
b40 = F40(189830553521)

print(f'a: {a40}')
print(f'b: {b40}')

# check the 4a3 + 27b^2 condition
if F(4*a^3 + 27*b^2) == 0:
    print("we failed that condition")

# Create a candidate curve
EC = EllipticCurve([a40, b40])
```

```
a: 522510265620
b: 189830553521
```

We now need see if the group defined by our a, b, and p has a large subgroup. There are fancy algorithms for that (and finding the largest subgroup), but because the numbers we have here are no more than 40 bits, I can just factor the cardinality of the group.

I could have written a loop to generate a and b repeatedly and tested for the sizes of subgroups (which is where $l_{\min}$ would have come in), but as I found one after just a couple of trials, I just manually reran this until I got numbers that I liked.

[74]:
```
u40 = EC.cardinality()

# instead of writing the code for finding "nearly prime" orders,
```

```
# I will just factor and eyeball the results until we get one
# with a small _h_
print(f'|EC| = {u40} = {factor(u40)}')

# we have a winner with the a and b that are now hardcoded above.
h40 = 2
n40 = 436039332233
```

```
|EC| = 872078664466 = 2 * 436039332233
```

Curves that fail to satisfy the MOV condition (Menezes, Okamoto and Vanstone) reduce the DLP to a smaller field. It, and the test for it, is described in §A.1.1 of ANSI X9.62-1998.

[75]:
```
def satisfies_mov(B, p, n):
    F = GF(n)
    p = F(p)
    t = F(1)
    for _ in range(1, B):
        t = t * p
        if t == 1:
            return(False)
    return(True)

satisfies_mov(20, p40, n40)
```

[75]:
```
True
```

Another condition to check is whether the curve is "anomalous". A very simple test. I am not sure why to run this, as any anomalous curve would instantly fail MOV.

[76]:
```
if p40 == u40:
    print("Uh-oh. Curve is anomalous. Pick a different a and b")
else:
    print("Curve is not anomalous.")
```

```
Curve is not anomalous.
```

We now have our curve

[77]:
```
print(EC)
```

```
Elliptic Curve defined by y^2 = x^3 + 522510265620*x + 189830553521 over Finite
Field of size 872078115079
```

3.1 Generating the generator

Now on to base point generation. We need to find a point that is in the big subgroup. Here we have the algorithm from §A.3.1, though I could have just generated random G until `G.order() == n40`, but showing this algorithm helps communicate what is meant by order of an element in a cyclic group.

[80]:
```
def gen_base(E, n, h):
    G = E(0) # because Python doesn't have a do-while
    while G == E(0):
        R = E.random_element()
        G = h * R
    if n * G != E(0):
        raise ValueError("wrong order")
    return G

# G = gen_base(EC, n40, h40)
G = EC([842965367165, 470264993162])
```

[81]:
```
print(f'Base point:\n\tx: {G[0]}\n\ty: {G[1]}')
print(f'order of G: {G.order()}')
```

```
Base point:
        x: 842965367165
        y: 470264993162
order of G: 436039332233
```

We now have a curve and a base point.

4 Appendix: Figure 12–2

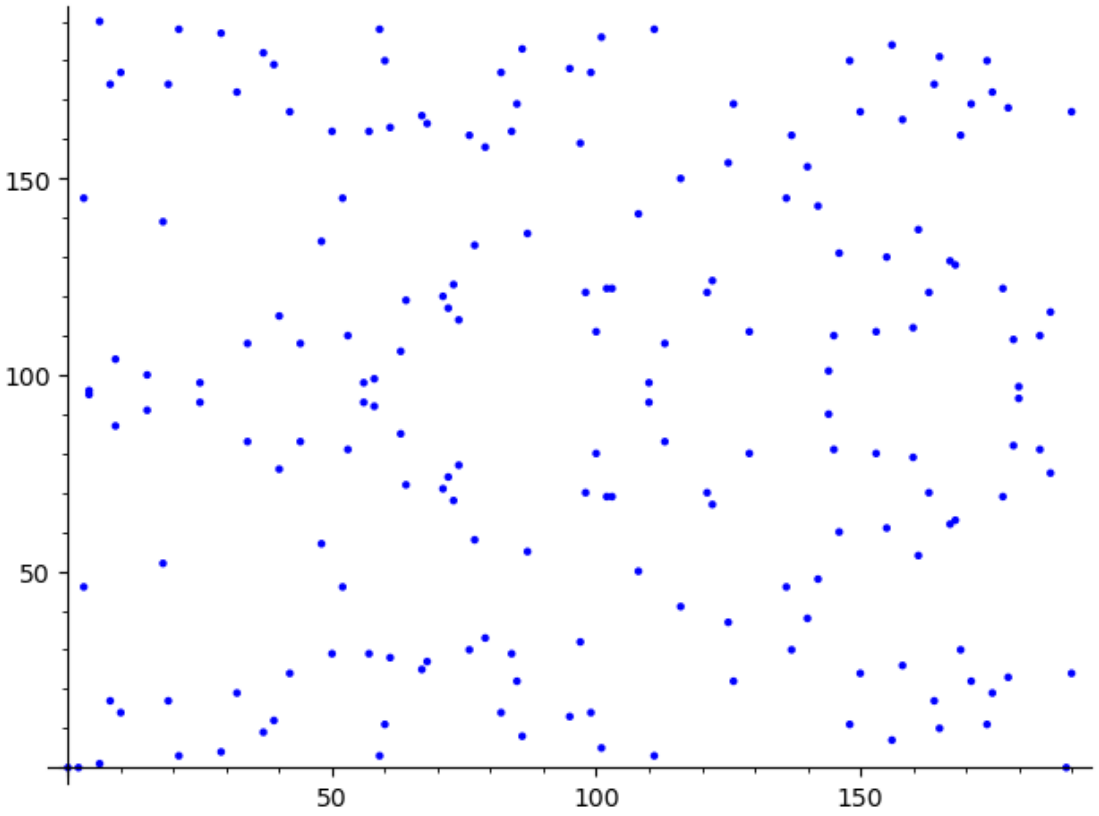
This is an attempt to replicate the examples in chapter 12

[82]:
```
FF = GF(191)
EC = EllipticCurve(FF, [-4, 0])
print(EC)
```

```
Elliptic Curve defined by y^2 = x^3 + 187*x over Finite Field of size 191
```

[83]:
```
plot(EC)
```

[83]:



The left hand side of the question is y^2 so if we were dealing with real numbers, there would be positive and negative values for each x value. But in our integer field, everything is modulo 191, so the points are reflected around 96.

[84]:
```
G = EC.gens()[0]
print(f"generator G is {G.xy()} with order {G.order()}")
```

```
generator G is (121, 70) with order 96
```