It is not an easy task to "randomly generate" cryptographically secure elliptic curves plus suitable base point. That is a reason why pre-selected elliptic curves are of practical importance.

The following are a few examples of specific elliptic curves that are widely used in practice.

http://blog.bjrn.se/2015/07/lets-construct-elliptic-curve.html

Example 224. For signing transactions, Bitcoin uses the elliptic curve

$$y^2 = x^3 + 7 \pmod{p}, \quad p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1,$$

together with the base point $P = (P_x, P_y)$ such that, in hexadecimal,

$$P_x = \texttt{79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798}.$$

(The y-coordinate P_y can be "lifted" from this; see the Sage code below.)

Where is this coming from? This is one of several vetted choices of elliptic curves compiled by the Standards for Efficient Cryptography Group (SECG), an industry consortium, in SEC 2: http://www.secg.org/

The particular curve above is secp256k1 in that document. While the equation of the curve and the prime p are clearly chosen to be "nice" (and so that the curve has nice properties; for instance its order is again a prime q; consequently, all regular points have order q themselves and, thus, generate all other points), it is much more mysterious how the point P was chosen:

https://crypto.stackexchange.com/questions/60420/

On the other hand, the choice of point is believed to not make much of a difference; in particular, it is not hard to see that the discrete logarithm problem is equally difficulty for all points.

Here is how to compute with that elliptic curve in Sage:

```
>>> p = 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
>>> E = EllipticCurve(GF(p), [0,7])
>>> P = E.lift_x(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)
>>> P
```

(55066263022277343669578718895168534326250603453777594175500187360389116729240 : 3267051\
0020758816978083085130507043184471273380659243275938904335757337482424 : 1)

```
>>> E.order()
```

115792089237316195423570985008687907852837564279074904382605163141518161494337

```
>>> is_prime(E.order())
```

1

```
>>> P.order()
```

115792089237316195423570985008687907852837564279074904382605163141518161494337

```
>>> 100*P
```

(107303582290733097924842193972465022053148211775194373671539518313500194639752 : 103795\
966108782717446806684023742168462365449272639790795591544606836007446638 : 1)

Armin Straub
straub@southalabama.edu

Example 225. A few years ago, more than 90% of webservers used one specific, NIST specified, elliptic curve referred to as P-256:

$$y^2 = x^3 - 3x + 41058363725152142129326129780047268409114441015993725554835256314039467401291,$$

taken modulo $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ (the fact that $p \approx 2^{256}$ makes the computations on the elliptic curve much faster in practice). The initial point $P = (x, y)$ on the curve has huge coordinates as well.

Using this single curve is sometimes considered to be problematic, especially following the concerns that the NSA may have implemented a backdoor into Dual_EC_DRBG, which was a previous NIST standard (2006–2014).

https://en.wikipedia.org/wiki/Dual_EC_DRBG

Example 226. A popular alternative is the curve Curve25519. In addition to some desirable theoretical advantages, its parameters are small ("nothing-up-my-sleeve numbers") and therefore not of similarly mysterious origin as the ones for P-256:

$$y^2 = x^3 + 486662x^2 + x, \quad p = 2^{255} - 19, \quad x = 9.$$

[Instead of points with (x, y) coordinates, one can actually work with just the x-coordinates for an additional speed-up.]

https://en.wikipedia.org/wiki/Curve25519

```
>>> E = EllipticCurve(GF(2^255-19), [0,486662,0,1,0])
>>> E
```

$y^2 = x^3 + 486662\,x^2 + x$

```
>>> E.order()
```

57896044618658097711785492504343953926856930875039260848015607506283634007912

```
>>> log(E.order(),2).n()
```

255.000000000000

```
>>> P = E.lift_x(9)
>>> P
```

(9: 43114425171068552920764898935933967039370386198203806730763910166200978582548: 1)

```
>>> 100*P
```

(44032819295671302737126221960004779200206561247519912509082330344845040669336: 8626006\
39244757237163427806001665957575078127166632317389150490196167274334 4: 1)

```
>>> P.order()
```

7237005577332262213973186563042994240857116359379907606001950938285454250989

```
>>> log(P.order(),2).n()
```

252.000000000000

```
>>> E.order() / P.order()
```

8

```
>>> 5*(20*P) == 20*(5*P)
```

1