$$d = 942074$$

1. $\boldsymbol{4G} =$
(103388573995635080359749164254216598308788835304023601477803095234286494993683,
370571411452421230130153166308643295501402169287011536698732864282558288810018)

2. $\boldsymbol{5G} =$
(215058298917636481143290559876192364941021333145752069708303857991580763338148,
980037086787626212336832405030808601290268873228741388055298849203099635801118)

3. $Q = dG = \boldsymbol{942074G} =$
(105071373288702886554749698371318794802666861735086494955172518052502509427025,
734357974399955861109310571128504626374872555697599100877629085679353860679993)

4. First, $d = 942074 = (11100101111111111010)_2$ in binary representation.
For every *zero* in d, a double is required; and for every *one* in d, a double and an addition are required (excluding the first 1, which is the initial setting). Since there are 15 *ones* and 5 *zeros* in d, a total of $(15-1)+5 = \mathbf{19}$ **doubles** and $(15-1) = \mathbf{14}$ **additions** are required.

5. When observing multiple consecutive *ones* in the binary representation of a number (with a preceding *zero*), we can view them as multiple consecutive *zeros* (with a preceding *one*) subtracted by 1. For example,
$$(\cdots 0111)_2 = (\cdots 1000)_2 - 1.$$
Thus, when there are n consecutive *ones*, we can calculate the result using $(n+1)$ doubles and 2 additions (one for the leading *one* and one for adding the inverse) instead of $(n+1)$ doubles and $(n+1)$ additions.
For the case when $d = 942074 = (11100101111111111010)_2$ where the consecutive *ones* are highlighted in red:
   - 111: 2 doubles, 2 additions,
   - 001: 3 doubles, 1 addition,
   - 01111111111: 11 doubles, 2 additions,
   - 010: 3 doubles, 1 addition.

   The result can be computed using a total of **19 doubles** and **6 additions** (one of which is an inverse addition).

6. **Input**

```
# Definition of secp256k1
F = FiniteField(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F)
CURVE = EllipticCurve([F(0), F(7)])
N = FiniteField(CURVE.order())

# Base point
GX = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
GY = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
G = CURVE(GX, GY)

d = int(942074) # Private key
Q = d * G       # Public key

# Random transaction from blockchain
z = int(0x3217F8EF32F55DCED1C50F4AB0C35D551C23D2D293264AFDBBB436D8E09CA0E7)

# ECDSA Signing
k = N.random_element()
kG = int(k) * G
x1 = kG.xy()[0]
r = N(x1)
s = (1 / k) * (z + r * d)

# ECDSA Verifying
w = N(1 / N(s))
u1 = N(z * w)
u2 = N(r * w)
x1 = (int(u1) * G + int(u2) * Q).xy()[0]
print("r == x1:", int(r) == int(x1))
```

Output

```
(‘r == x1:’, True)
```