1. **4*G***=   
   (103388573995635080359749164254216598308788835304023601477803095234286494993683, 37057141145242123013015316630864329550140216928701153669873286428255828810018)
2. **5*G*** =   
   (21505829891763648114329055987619236494102133314575206970830385799158076338148, 98003708678762621233683240503080860129026887322874138805529884920309963580118)
3. *Q* = *dG* = **942074*G*** =  
   (105071373288702886554749698371318794802666861735086494955172518052502509427025, 73435797439995586110931057112850462637487255569759910087762908567935386067993)
4. First, in binary representation.  
   For every *zero* in , a double is required; and for every *one* in , 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 , a total of  **doubles** and  **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,

Thus, when there are consecutive *ones*, we can calculate the result using doubles and 2 additions (one for the leading *one* and one for adding the inverse) instead of doubles and additions.  
For the case when 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).

1. **Input**
   1. # Definition of secp256k1
   2. F = FiniteField(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F)
   3. CURVE = EllipticCurve([F(0), F(7)])
   4. N = FiniteField(CURVE.order())
   6. # Base point
   7. GX = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
   8. GY = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
   9. G = CURVE(GX, GY)
   11. d = int(942074) # Private key
   12. Q = d \* G       # Public key
   14. # Random transaction from blockchain
   15. z = int(0x3217F8EF32F55DCED1C50F4AB0C35D551C23D2D293264AFDBBB436D8E09CA0E7)
   17. # ECDSA Signing
   18. k = N.random\_element()
   19. kG = int(k) \* G
   20. x1 = kG.xy()[0]
   21. r = N(x1)
   22. s = (1 / k) \* (z + r \* d)
   24. # ECDSA Verifying
   25. w = N(1 / N(s))
   26. u1 = N(z \* w)
   27. u2 = N(r \* w)
   28. x1 = (int(u1) \* G + int(u2) \* Q).xy()[0]
   29. **print**("r == x1:", int(r) == int(x1))

**Output**

1. (‘r == x1:’, True)