In this project I chose to use Python and the gmpy2 library, which is a library for arbitrary precision calculation.

For the first part of the project, I implemented the fundamental functions crucial for elliptic curve calculations: point_addition and scalar_multiplication_double_and_add.

The point_addition function, as the name implies, performs the addition of two points. The scalar_multiplication_double_and_add function utilizes the double-and-add algorithm for scalar multiplication.

When performing point addition on an elliptic curve, there are several special cases to consider.

1) If any one of the point is at infinity, we just return the other point.

```python
# Elliptic curve point addition
def point_addition(x1, y1, x2, y2):

    if (x1, y1) == (mpz(0), mpz(0)):
        return (x2, y2)

    elif (x2, y2) == (mpz(0), mpz(0)):
        return (x1, y1)
```

2) If two points are the same and the y coordinate is 0, we should just return the point at infinity.

3) If two points have the same x coordinate but different y coordinate, we should return the point at infinity as well.

4) Besides these special conditions, we just do the normal point addition on elliptic curve.

```python
if x1 == x2 and y1 == y2:
    if y1 == 0:
        return (mpz(0), mpz(0))
    try:
        m = (3 * x1 * x1 + a) * gmpy2.invert(2 * y1, p)
    except ZeroDivisionError:
        print("ZeroDivisionError")
        exit(0)
elif x1 == x2:
    return (mpz(0), mpz(0))
else:
    # point addition
    m = (y2 - y1) * gmpy2.invert(x2 - x1, p)

m = m % p
x3 = (m * m - x1 - x2) % p
y3 = (m * (x1 - x3) - y1) % p
return x3, y3
```

For point multiplication, we adopt double-and-add algorithm.

First, if the scalar is 0, we just return the point at infinity.

```python
def scalar_multiplication_double_and_add(x, y, scalar):
    # Ensure scalar is of type mpz
    scalar = mpz(scalar)
    if scalar == 0:
        return (mpz(0), mpz(0))
    # Convert the scalar to binary form
    scalar_bin = scalar.digits(2)

    # Initialize the result point as the identity element (0, 0)
    result_x, result_y = mpz(0), mpz(0)
```

Then, we just perform double-and-add procedure to get the final result.

```python
# Iterate through the bits of the scalar
for i, bit in enumerate(scalar_bin):
    if i == 0 and bit == '1':
        # If the first bit is 1, set the result point to the original point
        result_x, result_y = x, y
        continue
    # Double the current point
    result_x, result_y = point_addition(result_x, result_y, result_x, result_y)
    # If the bit is 1, add the original point
    if bit == '1':
        result_x, result_y = point_addition(result_x, result_y, x, y)
    else:
        continue
```

1. Evaluate $4G$.

   result_x:
   10338857399563508035974916425421659830878883530402360147780309523428649499368 3

   result_y:
   37057141145242123013015316630864329550140216928701153669873286428255828810018

2. Evaluate $5G$.

   result_x:
   21505829891763648114329055987619236494102133314575206970830385799158076338148

result_y:
980037086787626212336832405030808601290268873228741388055298849203099635801 18

3. Evaluate $Q = d\ G$.

   (Note: my student number is r11944064, so d is 4064 here)

   result_x:
   576933628783355757501373109146971078483421282094287650643184999901788678416470

   result_y:
   103370482549189659955992029635940713640495380173956914861145077739501085424098

4. With standard Double-and Add algorithm for scalar multiplications, how
   many doubles and additions respectively are required to evaluate dG?

   Since the binary representation of 4064 is: **111111100000**, so the answer is:

   doubles: 11 times.

   Additions: 6 times.

5. Note that it is effortless to find $-P$ from any P on a curve. If the addition of an inverse point is allowed, try your best to evaluate dG as fast as possible.

   Hint: $31P = 2(2(2(2(2P)))) - P$.

   Since the binary representation of 4064 is: **111111100000**, We can first evaluate **(1111111)P = (10000000)P - P**, which requires 7 multiplication and 1 subtraction.

   Then, we get dG by evaluate **(111111100000)P = (1111111)P x 100000**, which requires 5 multiplicatiosns.

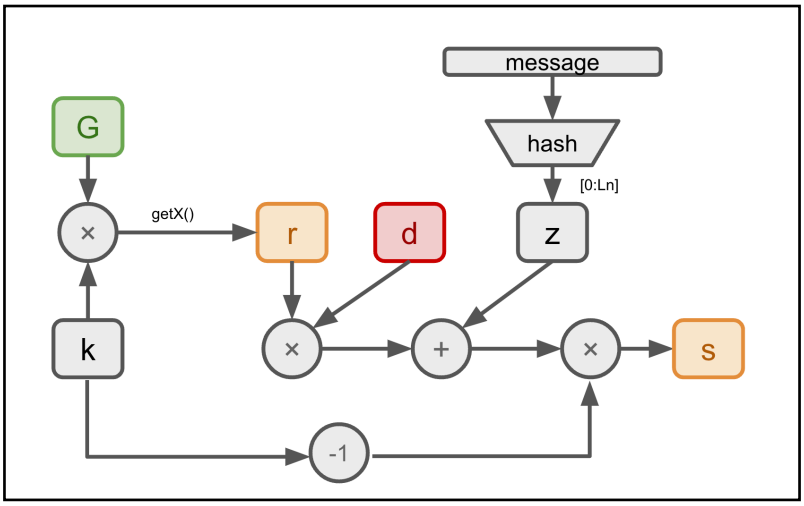
Overall, we only needs:

doubles: 12 times.

Subtraction: 1 time.

to evaluate dG if the addition of an inverse point is allowed.

6. Take a Bitcoin transaction as you wish. Sign the transaction with a random number *k* and your private key *d*.



The figure above illustrates the workings of the ECDSA algorithm. In this example, 'd' represents the private key, and we've chosen the value 4064. 'G' is the generating point defined by the secp256k1 standard. 'K' is a random number selected from the range 0 to n-1; in this case, it's generated using a random number generator. 'Z' refers to the top Ln (n in this scenario) bits of the message hash. We're using the SHA-256 hash function implemented in Python's 'hashlib' library. For illustrative purposes, the message being hashed is a simple "hello world".

First,k is generated using a random bits generator from the gmpy library.

```
# Generate a random mpz number of 256 bits
random_mpz = gmpy2.mpz_rrandomb(rand_state, 256)

k = random_mpz % (n-2) + 1 # k in [1, n-1]
```

Secondly, the hash is generated using the SHA-256 algorithm.

```
message_bytes = message.encode('utf-8')
message_hash = hashlib.sha256(message_bytes).digest()
z = message_hash.hex().upper()
z = gmpy2.mpz('0x' + z)
```

Thirdly, The value r is derived by multiplying the point 'G' with the random number k.

```
# Calculate the curve point (x1, y1) = k * G
(x1, y1) = scalar_multiplication_double_and_add(Gx, Gy, k)

# Calculate r = x1 mod n
r = x1 % n
if r == 0:
    return ecdsa_sign(message)  # Start over if r is 0
```

Finally we can calculate s as follows.

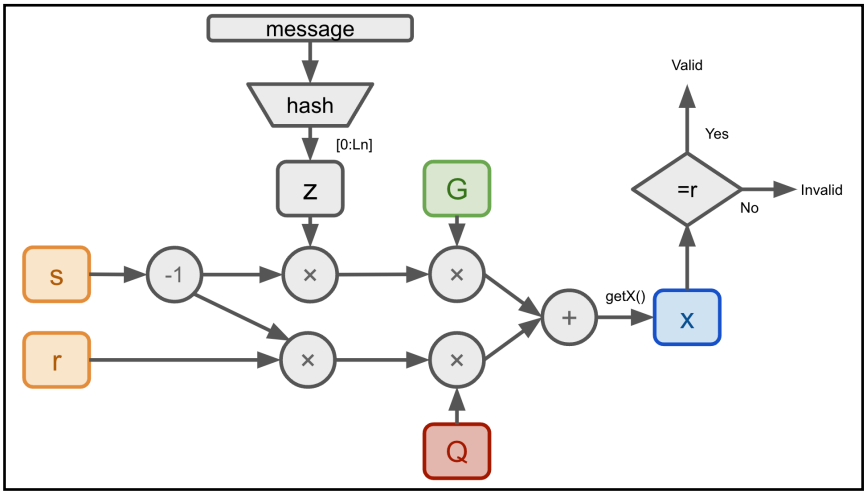
```
# Calculate s = k^-1(z + r*private_key_d) mod n
s = (gmpy2.invert(k, n) * (z + r * d)) % n
if s == 0:
    return ecdsa_sign(message) # Start over if s is 0
```

We can then obtain a signature (r, s) with

r:
11747203460228494780299306713586482340548324644079267799193845498057748934662

s:
24109619879480872715406210110089102915898434357101945108964767957511361403484

7. Verify the digital signature with your public key Q.



First, we need to verify the following conditions.

1) Point Q must be on the elliptic curve.
2) Point Q should not be at infinity.
3) The result of multiplying Q by the order of the field, denoted as n, must be the point at infinity.
4) Both components of the signature, r and s, must fall within the range [1, n−1].

```
# Check that Q_A is not equal to the identity element O, a
if Q == (0, 0) or not is_point_on_curve(*Q):
    print('not on curve')
    return False

# p * Q should be equal to O
if scalar_multiplication_double_and_add(*Q, n) != (0, 0):
    print('n * Q != O')
    return False

# Verify that r and s are integers in [1, n - 1].
if not (1 <= r < n) or not (1 <= s < n):
    print('r or s not in [1, n-1]')
    return False
```

If these checks are successful, we then proceed with a series of calculations to determine whether the derived signature matched the original signature.

First, we calculate z as was done in the signing stage.

```
# Calculate z
message_bytes = message.encode('utf-8')
message_hash = hashlib.sha256(message_bytes).digest()
z = message_hash.hex().upper()
z = gmpy2.mpz('0x' + z)
```

Secondly, we compute u1 and u2 by multiplying z and r with the inverse of s.

```
# Calculate u1 = z * s^-1 mod n and u2 = r * s^-1 mod n.
s_inv = gmpy2.invert(s, n)
u1 = (z * s_inv) % n
u2 = (r * s_inv) % n
```

Thirdly, we calculate two points: the first by multiplying the generator point G by u1, and the second by multiplying the public key point Q by u2. We then add these two points together and take the x-coordinate of the resulting point.

Finally, we check if this x-coordinate matches r to verify the signature.

```
# Calculate the curve point (x1, y1) = u1 * G + u2 * Q_A.
point_u1G = scalar_multiplication_double_and_add(Gx, Gy, u1)
point_u2Q = scalar_multiplication_double_and_add(*Q, u2)
point_x1y1 = point_addition(point_u1G[0], point_u1G[1], point_u2Q[0], point_u2Q[1])

# If the resulting point is at infinity, the signature is invalid.
if point_x1y1 == (0, 0):
    print('point at infinity')
    return False


# The signature is valid if r ≡ x1 (mod n), invalid otherwise.
return r%n == point_x1y1[0] % n
```

8. Over Z_{10007}, construct the quadratic polynomial $p(x)$ with $p(1) = 10$, $p(2) = 20$, and $p(3) = d$.

For d = 4064, we have

$$p(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)}10 + \frac{(x-1)(x-3)}{(2-1)(2-3)}20 + \frac{(x-1)(x-2)}{(3-1)(3-2)}4064$$

$$p(x) = \frac{(x-2)(x-3)}{2}10 + \frac{(x-1)(x-3)}{-1}20 + \frac{(x-1)(x-2)}{2}4064$$

Since

$(2)^{-1} \equiv 5004 \ (mod\ 10007)$,
$(-1)^{-1} \equiv 10006 \ (mod\ 10007)$.

And

$(5004 \times 10) \equiv 5 \ (mod\ 10007)$,
$(10006 \times 20) \equiv 9987 \ (mod\ 10007)$,
$(5004 \times 4064) \equiv 2032 \ (mod\ 10007)$.

Thus,

$$p(x) = 5(x-2)(x-3) + 9987(x-1)(x-3) + 2032(x-1)(x-2).$$