JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

UNIT CODE: BCT 2314

UNIT NAME: CRYPTOGRAPHY AND COMPUTER SECURITY

ASSIGNMENT I

NAME: THOMAS RAKWACH

REG: CS282-8120/2014

Write ECC code that displays the following curves

1. $y^2 = x^3 + x + 1$
2. $y^2 = x^3 - 25x$
3. $y^2 = x^3 + x + 6$
4. $y^2 = x^3 - 4x$
5. $y^2 = x^3 - 1$

```
import matplotlib.pyplot as plt
import numpy as np

def graph_draw(graph_formula):
    cartesian_size = 8.0
    x_axis = np.linspace(-cartesian_size, cartesian_size, 300)
    y_axis = np.linspace(-cartesian_size, cartesian_size, 300)

    # the X and Y here take their input from the equation values below
    X, Y = np.meshgrid(x_axis, y_axis)

    f = eval(graph_formula)

    plt.contour(X, Y, f, [0])
    plt.title(graph_formula)
    plt.grid()
    plt.show()


def graph_describe():
    graph_draw("X**3 - Y**2 + X + 1")

    graph_draw("X**3 - Y**2 - (25*X)")

    graph_draw("X**3 - Y**2 + X + 6")

    graph_draw("X**3 - Y**2 - (4*X)")

    graph_draw("X**3 - Y**2 - 1")


if __name__ == "__main__":
    graph_describe()
```

Given $\boldsymbol{\beta} = (\mathbf{2}, \mathbf{7})$ $\boldsymbol{p} = \mathbf{11}$ write an ECC program that generates all the points on the curve with $\boldsymbol{p} = \mathbf{11}$ and that the code should be able to perform point addition and doubling

```
class Point(object):
  # Construct a point with two given coordindates.
  def __init__(self, x, y):
    self.x, self.y = x, y
    self.inf = False

  # Construct the point at infinity.
  @classmethod
  def atInfinity(cls):
    P = cls(0, 0)
    P.inf = True
    return P

  def is_infinite(self):
    return self.inf


# Elliptic Curves over any Field ----------------------------------------------------------------

class Curve(object):
  # Set attributes of a general Weierstrass cubic y^2 = x^3 + ax^2 + bx + c over any field.
  def __init__(self, a, b, c, char, exp):
    self.a, self.b, self.c = a, b, c
    self.char, self.exp = char, exp
    print(self)


# Elliptic Curves over Prime Order Fields -------------------------------------------------------

class CurveOverFp(Curve):
  # Construct a Weierstrass cubic y^2 = x^3 + ax^2 + bx + c over Fp.
  def __init__(self, a, b, c, p):
    Curve.__init__(self, a, b, c, p, 1)

  def get_points(self):
    # Start with the point at infinity.
    points = [Point.atInfinity()]

    # Just brute force the rest.
    for x in range(self.char):
      for y in range(self.char):
        P = Point(x, y)
        if (y * y) % self.char == (x * x * x + self.a * x * x + self.b * x + self.c) % self.char:
```

```
            points.append(P)
        return points

    def invert(self, P):
        if P.is_infinite():
            return P
        else:
            return Point(P.x, -P.y % self.char)

    def add(self, P_1, P_2):
        # Adding points over Fp and can be done in exactly the same way as adding over Q,
        # but with of the all arithmetic now happening in Fp.
        y_diff = (P_2.y - P_1.y) % self.char
        x_diff = (P_2.x - P_1.x) % self.char
        if P_1.is_infinite():
            return P_2
        elif P_2.is_infinite():
            return P_1
        elif x_diff == 0 and y_diff != 0:
            return Point.atInfinity()
        elif x_diff == 0 and y_diff == 0:
            if P_1.y == 0:
                return Point.atInfinity()
            else:
                ld = ((3 * P_1.x * P_1.x + 2 * self.a * P_1.x + self.b) * mult_inv(2 * P_1.y, self.char)) % self.char
        else:
            ld = (y_diff * mult_inv(x_diff, self.char)) % self.char
        nu = (P_1.y - ld * P_1.x) % self.char
        x = (ld * ld - self.a - P_1.x - P_2.x) % self.char
        y = (-ld * x - nu) % self.char
        return Point(x, y)


# Extended Euclidean algorithm.
def euclid(sml, big):
    # When the smaller value is zero, it's done, gcd = b = 0*sml + 1*big.
    if sml == 0:
        return big, 0, 1
    else:
        # Repeat with sml and the remainder, big%sml.
        g, y, x = euclid(big % sml, sml)
        # Backtrack through the calculation, rewriting the gcd as we go. From the values just
        # returned above, we have gcd = y*(big%sml) + x*sml, and rewriting big%sml we obtain
        # gcd = y*(big - (big//sml)*sml) + x*sml = (x - (big//sml)*y)*sml + y*big.

        return g, x - (big // sml) * y, y
```

```
# Compute the multiplicative inverse mod n of a with 0 < a < n.
def mult_inv(a, n):
   g, x, y = euclid(a, n)
   # If gcd(a,n) is not one, then a has no multiplicative inverse.
   if g != 1:
      raise ValueError('multiplicative inverse does not exist')
   # If gcd(a,n) = 1, and gcd(a,n) = x*a + y*n, x is the multiplicative inverse of a.
   else:
      return x % n


a = CurveOverFp(1, 0, 0, 11)
a.add(2, 7)
```