7.6 Elliptic Curves

The fundamental algorithms that we described in Chapter 6 are arithmetic of points on elliptic curve, the Pollard $(p-1)$ and elliptic curve integer factorization methods, and the the ElGamal elliptic curve cryptosystem. In this section we implement each of these algorithms for elliptic curves over $\mathbf{Z}/p\mathbf{Z}$, and finish with an investigation of the associative law on an elliptic curve.

7.6.1 Arithmetic

Each elliptic curve function takes as first input an elliptic curve $y^2 = x^3 + ax + b$ over $\mathbf{Z}/p\mathbf{Z}$, which we represent by a triple `(a,b,p)`. We represent points on an elliptic curve in Python as a pair `(x,y)`, with $0 \leq x, y < p$ or as the string `"Identity"`. The functions in Listings 7.6.1 and 7.6.2 implement the group law (Algorithm 6.2.1) and computation of mP for possibly large m.

Listing 7.6.1 (Elliptic Curve Group Law).

```
def ellcurve_add(E, P1, P2):
    """
    Returns the sum of P1 and P2 on the elliptic
    curve E.
    Input:
         E -- an elliptic curve over Z/pZ, given by a
              triple of integers (a, b, p), with p odd.
         P1 --a pair of integers (x, y) or the
              string "Identity".
         P2 -- same type as P1
    Output:
         R -- same type as P1
    Examples:
    >>> E = (1, 0, 7)   # y**2 = x**3 + x over Z/7Z
    >>> P1 = (1, 3); P2 = (3, 3)
    >>> ellcurve_add(E, P1, P2)
    (3, 4)
    >>> ellcurve_add(E, P1, (1, 4))
    'Identity'
    >>> ellcurve_add(E, "Identity", P2)
    (3, 3)
    """
    a, b, p = E
    assert p > 2, "p must be odd."
    if P1 == "Identity": return P2
    if P2 == "Identity": return P1
```

```
    x1, y1 = P1; x2, y2 = P2
    x1 %= p; y1 %= p; x2 %= p; y2 %= p
    if x1 == x2 and y1 == p-y2: return "Identity"
    if P1 == P2:
        if y1 == 0: return "Identity"
        lam = (3*x1**2+a) * inversemod(2*y1,p)
    else:
        lam = (y1 - y2) * inversemod(x1 - x2, p)
    x3 = lam**2 - x1 - x2
    y3 = -lam*x3 - y1 + lam*x1
    return (x3%p, y3%p)
```

Listing 7.6.2 (Computing a Multiple of a Point).

```
def ellcurve_mul(E, m, P):
    """
    Returns the multiple m*P of the point P on
    the elliptic curve E.
    Input:
        E -- an elliptic curve over Z/pZ, given by a
             triple (a, b, p).
        m -- an integer
        P -- a pair of integers (x, y) or the
             string "Identity"
    Output:
        A pair of integers or the string "Identity".
    Examples:
    >>> E = (1, 0, 7)
    >>> P = (1, 3)
    >>> ellcurve_mul(E, 5, P)
    (1, 3)
    >>> ellcurve_mul(E, 9999, P)
    (1, 4)
    """
    assert m >= 0, "m must be nonnegative."
    power = P
    mP = "Identity"
    while m != 0:
        if m%2 != 0: mP = ellcurve_add(E, mP, power)
        power = ellcurve_add(E, power, power)
        m /= 2
    return mP
```

7.6.2 Integer Factorization

In Listing 7.6.3 we implement Algorithm 6.3.2 for computing the least common multiple of all integers up to some bound.

Listing 7.6.3 (Least Common Multiple of Numbers).

```
def lcm_to(B):
    """
    Returns the least common multiple of all
    integers up to B.
    Input:
        B -- an integer
    Output:
        an integer
    Examples:
    >>> lcm_to(5)
    60
    >>> lcm_to(20)
    232792560
    >>> lcm_to(100)
    69720375229712477164533808935312303556800L
    """
    ans = 1
    logB = log(B)
    for p in primes(B):
        ans *= p**int(logB/log(p))
    return ans
```

Next we implement Pollard's $p-1$ method, as in Algorithm 6.3.3. We use only the bases $a = 2, 3$, but you could change this to use more bases by modifying the `for` loop in Listing 7.6.4.

Listing 7.6.4 (Pollard).

```
def pollard(N, m):
    """
    Use Pollard's (p-1)-method to try to find a
    nontrivial divisor of N.
    Input:
        N -- a positive integer
        m -- a positive integer, the least common
             multiple of the integers up to some
             bound, computed using lcm_to.
    Output:
        int -- an integer divisor of n
    Examples:
```

```
    >>> pollard(5917, lcm_to(5))
    61
    >>> pollard(779167, lcm_to(5))
    779167
    >>> pollard(779167, lcm_to(15))
    2003L
    >>> pollard(187, lcm_to(15))
    11
    >>> n = random_prime(5)*random_prime(5)*random_prime(5)
    >>> pollard(n, lcm_to(100))
    315873129119929L     #rand
    >>> pollard(n, lcm_to(1000))
    3672986071L          #rand
    """
    for a in [2, 3]:
        x = powermod(a, m, N) - 1
        g = gcd(x, N)
        if g != 1 and g != N:
            return g
    return N
```

In order to implement the elliptic curve method and also in our upcoming elliptic curve cryptography implementation, it will be useful to define the function `randcurve` of Listing 7.6.5, which computes a random elliptic curve over $\mathbf{Z}/p\mathbf{Z}$ and a point on it. For simplicity, `randcurve` always returns a curve of the form $y^2 = x^3 + ax + 1$, and the point $P = (0, 1)$. As an exercise you could change this function to return a more general curve, and find a random point by choosing a random x, then incrementing it until $x^3 + ax + 1$ is a perfect square.

Listing 7.6.5 (Random Elliptic Curve).

```
def randcurve(p):
    """
    Construct a somewhat random elliptic curve
    over Z/pZ and a random point on that curve.
    Input:
        p -- a positive integer
    Output:
        tuple -- a triple E = (a, b, p)
        P -- a tuple (x,y) on E
    Examples:
    >>> p = random_prime(20); p
    17758176404715800329L    #rand
    >>> E, P = randcurve(p)
    >>> print E
```

```
    (15299007531923218813L, 1, 17758176404715800329L)  #rand
    >>> print P
    (0, 1)
    """
    assert p > 2, "p must be > 2."
    a = randrange(p)
    while gcd(4*a**3 + 27, p) != 1:
        a = randrange(p)
    return (a, 1, p), (0,1)
```

In Listing 7.6.6, we implement the elliptic curve factorization method.

Listing 7.6.6 (Elliptic Curve Factorization Method).

```
def elliptic_curve_method(N, m, tries=5):
    """
    Use the elliptic curve method to try to find a
    nontrivial divisor of N.
    Input:
        N -- a positive integer
        m -- a positive integer, the least common
             multiple of the integers up to some
             bound, computed using lcm_to.
        tries -- a positive integer, the number of
             different elliptic curves to try
    Output:
        int -- a divisor of n
    Examples:
    >>> elliptic_curve_method(5959, lcm_to(20))
    59L      #rand
    >>> elliptic_curve_method(10007*20011, lcm_to(100))
    10007L   #rand
    >>> p = random_prime(9); q = random_prime(9)
    >>> n = p*q; n
    117775675640754751L   #rand
    >>> elliptic_curve_method(n, lcm_to(100))
    117775675640754751L   #rand
    >>> elliptic_curve_method(n, lcm_to(500))
    117775675640754751L   #rand
    """
    for _ in range(tries):                     # (1)
        E, P = randcurve(N)                    # (2)
        try:                                   # (3)
            Q = ellcurve_mul(E, m, P)          # (4)
        except ZeroDivisionError, x:           # (5)
            g = gcd(x[0],N)                    # (6)
```

```
            if g != 1 or g != N: return g      # (7)
    return N
```

In line (1) the underscore means that the for loop iterates `tries` times, but that no variable is "wasted" recording which iteration we are in. In line (2) we compute a random elliptic curve and point on it. The elliptic curve method works by assuming N is prime, doing a certain computation, on an elliptic curve over $\mathbf{Z}/N\mathbf{Z}$, and detecting if something goes wrong. Python contains a mechanism called exception handling, which leads to a very simple implementation of the elliptic curve method, that uses the elliptic curve functions that we have already defined. The `try` statement in line (3) means that the code in line (4) should be executed, and if the `ZeroDivisionError` exception is raised, then the code in lines (6) and (7) should be executed, but not otherwise. Recall that in the definition of `inversemod` from Listing 7.2.2, when the inverse could not be computed, we raised a `ZeroDivisionError`, which included the offending pair (a, n). Thus when computing mP, if at any point it is not possible to invert a number modulo N, we jump to line (6), compute a gcd with N, and hopefully split N.

7.6.3 ElGamal Elliptic Curve Cryptosystem

Listing 7.6.7 defines a function that creates an ElGamal cryptosystem over $\mathbf{Z}/p\mathbf{Z}$. This is simplified from what one would do in actual practice. One would use a more general random elliptic curve and point than we do in `elgamal_init`, and count the number of points on it using the Schoof-Elkies-Atkin algorithm, then repeat this procedure if the number of points is not a prime or a prime times a small number, or is p, $p-1$, or $p+1$. Since implementing Schoof-Elkies-Atkin is beyond the scope of this book, we have not included this crucial step.

Listing 7.6.7 (Initialize ElGamal).

```
def elgamal_init(p):
    """
    Constructs an ElGamal cryptosystem over Z/pZ, by
    choosing a random elliptic curve E over Z/pZ, a
    point B in E(Z/pZ), and a random integer n.  This
    function returns the public key as a 4-tuple
    (E, B, n*B) and the private key n.
    Input:
        p -- a prime number
    Output:
        tuple -- the public key as a 3-tuple
                 (E, B, n*B), where E = (a, b, p) is an
```