Topics in Computer Science - Bitcoin: Programming the Future of Money - ITCS 4010 & 5010 - Fall 2024 - UNC Charlotte

Homework 3 - Elliptic Curves (110 Points)

HRITIKA KUCHERIYA

Submission instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, create a PDF version of the Jupypter notebook which includes visually all executed cells. This can be done in different ways depending on your specific Python/Jupyter setup. You can do that either by exporting into PDF (PDF via LaTeX / PDF via HTML), or by exporting into an HTML file first and then print the HTML site as a PDF and save that PDF.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Submit **both** your PDF and the notebook file .ipynb on Gradescope.
8. Make sure your your Gradescope submission contains the correct files by downloading it after posting it on Gradescope.

1. Build Calendar (8 Points)

There are 365 days in a year. That means there are seven months with 31 days, four months with 30 days, and one month with 28 days. Write `Calendar` function that creates a list called `date` that indexes all dates from 0 – 364. So, date[0] = "Jan 1", date[1] = "Jan 2", and so on, until date[364] = "Dec 31". Spell all months using their three-letter abbreviation.

Then write a function `FutureDay` that accepts two parameters: a start date from the date list and a positive integer representing the number of days. The function should calculate and return the future date by adding the specified number of days to the start date.

For example, FutureDay("Jan 3", 2) = "Jan 5".

```
def Calendar():

    date = []

    months = {
        'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30,
        'Jul': 31, 'Aug': 31, 'Sep': 30, 'Oct': 31, 'Nov': 30, 'Dec': 31
    }


    for month in months:
        for day in range(1, months[month] + 1):
            date.append(f"{month} {day}")

    return date


def FutureDay(start, increment):
    dates = Calendar()
    if not isinstance(increment, int) or increment < 0:
        raise ValueError("Increment must be a positive integer")
    if start not in dates:
        raise ValueError(f"Start date '{start}' not in calendar")
    start_index = dates.index(start)
    future_index = (start_index + increment) % 365
    return dates[future_index]
```

Run the following cell to test your code:

```
print(FutureDay("Jan 3", 2) == "Jan 5")
print(FutureDay("Jan 20", 100))
print(FutureDay("Jul 4", 1000))
print(FutureDay("Dec 26", 11))
```

```
True
Apr 30
Mar 31
Jan 6
```

2. Build a Finite Field (15 Points)

a.) Write a funtion `buildfield` that takes the order of the finite field as input and returns the finite field set. In case that the the order of the finite field is not admissible, the function should print a suitable error message instead.

Hint: Recall what orders of finite fields are admissible. You can revisit Jimmy Song's *Programming Bitcoin*: Chapter 1 to revisit this.

```
def buildfield(prime):
    if not isinstance(prime, int):
        print("Error: Input must be an integer")
        return None

    if prime <= 0:
        print("Error: Input must be positive")
        return None

    if prime == 1:
        print("Error: 1 is not a prime number")
        return None

    for i in range(2, int(prime ** 0.5) + 1):
        if prime % i == 0:
            print(f"Error: {prime} is not a prime number")
            return None

    field = set(range(prime))

    return field
```

Run the following cell to test your code:

```
print(buildfield(3) == {0,1,2})
print(buildfield(31))
print(buildfield(4))
print(buildfield(12))
```

```
True
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
Error: 4 is not a prime number
None
Error: 12 is not a prime number
None
```

b.) Provide an example of a set $F_p = \{0, 1, 2, 3, \ldots, p-1\}$ with a choice of p and an element $a \in F_p$ that *does not have a multiplicative inverse* (when defining addition and multiplication as for finite fields), and explain why.

Example of an Element Without a Multiplicative Inverse in (Fp)

Example:

Consider the set (F7 = { 0, 1, 2, 3, 4, 5, 6 }), which forms a finite field under **modulo 7** arithmetic.

The **multiplicative inverse** of an element (a) in (Fp) is defined as the number (b) such that:

a X b euqivalent to mod 7

However, in any **finite field (Fp), the number 0 never has a multiplicative inverse** because:

0 X b = 0 is not equivalent to 1 mod p (for any b)

Thus, in (F7), the element 0 does not have a multiplicative inverse, making it the required example.

Explaination?

- A number a in Fp has an inverse if and only if it is **coprime** (i.e., gcd = 1) with p .
- 0 is **not** coprime with any number because **gcd(0, p) = p**.
- In a finite field Fp, all **nonzero** elements have an inverse, but **0 never does**.

Thus, 0 is always the element that lacks a multiplicative inverse in any finite field Fp .

ALL IN ALL:

> **Example:** In F7 = { 0, 1, 2, 3, 4, 5, 6 } , the element **0** does not have a multiplicative inverse because there is no number b such that (0 X b is equivalent to 1 mod 7).
> This holds for any finite field Fp , where **0 never has a multiplicative inverse**.

3. Scalar Multiplication (11 Points)

a.

Implement a function `scalar_multiply` that accepts the order of a finite field `order` and a scalar `k` as arguments, and returns a set of elements resulting from the scalar multiplication of `k` with each element in the finite field.

```
def scalar_multiply(order, k):
    finite_field = buildfield(order)
    if finite_field is None:
        return None
    result = set()
    for element in finite_field:
        result.add((element * k) % order)
    return result
```

Run the following four cells to test your code:

```
k = [3,17,256,977]
order = 11
for i in k:
    print(scalar_multiply(order, i))
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
order = 13
for i in k:
    print(scalar_multiply(order, i))
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

```
order = 27
for i in k:
    print(scalar_multiply(order, i))
```

```
Error: 27 is not a prime number
None
Error: 27 is not a prime number
None
Error: 27 is not a prime number
None
Error: 27 is not a prime number
None
```

```
order = 3
for i in k:
    print(scalar_multiply(order, i))
```

```
{0}
{0, 1, 2}
{0, 1, 2}
{0, 1, 2}
```

b.)

From the output sets obtained from scalar_multiply for finite fields of orders 11, 13, 27, and 3 what patterns or properties do you observe? Provide a brief explanation of your findings.

YOUR ANSWER HERE

Observations from scalar_multiply Output

1. **Prime Order Fields (11, 13)**
   - The output always includes all elements of Fp, meaning scalar multiplication preserves the field structure.
   - This confirms that **finite fields of prime order form a cyclic group under multiplication**.
2. **Non-Prime Order (27)**
   - The function correctly identifies that F{27} **is not a valid finite field** and returns an error.
3. **Smallest Prime Field (3)**
   - For some scalars, the output is 0, indicating that certain multipliers collapse all elements.
   - Other scalars preserve all elements, maintaining the field structure.

Alll-in-all

- **Finite fields exist only for prime orders**, and scalar multiplication retains their structure.
- **Non-prime orders cannot form a field**.
- **Small fields exhibit unique behavior**, where some scalars map all elements to 0.

c.)

Revisit the class on finite fields in view of the results of part a) and b). Please explain for what statement this empirical observation was used in class.

YOUR ANSWER HERE

Empirical Observations & Class Statements

1. **Finite Fields Require Prime Orders**
   - Non-prime order = 27 returned an error, confirming that **finite fields exist only for prime orders** (Lecture 8: Order of a Finite Field).
2. **Multiplicative Inverses Exist for Nonzero Elements**
   - 0 lacked an inverse, aligning with the class statement that **only nonzero elements in Fp have inverses** (Lecture 8: Multiplication in Finite Fields).
3. **Scalar Multiplication Preserves Field Structure**
   - In F11 and F13 , scalar multiplication covered all elements, proving **closure under multiplication** (Lecture 9: Multiplication in Finite Fields).
4. **Fermat's Little Theorem & Inverses**
   - Ensures **every nonzero element in Fp has an inverse**, explaining why scalar multiplication works without collapsing (Lecture 9: Fermat's Little Theorem).

4. Exponentiation (6 Points)

Implement the function `findpow` that calculates the power of each element in a finite field of the order `order` raised to a given `exponent` and returns the resulting powers as a list.

```
def findpow(order, exponent):
    finite_field = buildfield(order)
    if finite_field is None:
        return None
    result = []
    for element in finite_field:

        if exponent < 0:
            if element == 0:
                result.append(0)
            else:
                pow_val = pow(element, order-2, order)
                result.append(pow(pow_val, -exponent, order))
        else:
            result.append(pow(element, exponent, order))
    return result
```

Run the following cells to test your code:

```
print(findpow(3, 9) == [0,1,2])
```

```
True
```

```
print(findpow(31, -3)[17] == 29)
```

```
True
```

```
print(sorted(findpow(29, 66)))
```

```
[0, 1, 1, 4, 4, 5, 5, 6, 6, 7, 7, 9, 9, 13, 13, 16, 16, 20, 20, 22, 22, 23, 23, 24, 24, 25, 25, 28, 28]
```

5. Finite Field Class (15 Points)

Define a class `FieldElement` to represent an element denoted by its smallest positive integer representation `num` within a finite field of the order `p`. This class includes the following core functionalities:

- In the `__init__` constructor, we ensure that the `num` lies between `0` and `p - 1`. If it doesn't, a `ValueError` is raised; otherwise, the constructor parameter values are assigned to the objects.
- The `__eq__` method determines if two FieldElement objects are equal. When the values of `num` and `p` of one element are equal to `num` and `p` of second element are identical, respectively, the method returns `True`.
- `__add__` method overloads addition, `__sub__` method overloads subtraction for finite field.
- Similarly, `__mul__`, `__pow__`, `__truediv__` and `__rmul__` methods overload finite field multiplication, exponentiation, division, and scalar multiplication respectively

```
class FieldElement:
    def __init__(self, num, prime):
        if num >= prime or num < 0:
            raise ValueError(f"Num {num} not in field range 0 to {prime-1}")
        self.num = num
        self.prime = prime

    def __repr__(self):
        return f'FieldElement_{self.prime}({self.num})'

    def __eq__(self, other):
        if other is None:
            return False
        return self.num == other.num and self.prime == other.prime

    def __ne__(self, other):
        return not self.__eq__(other)

    def __add__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot add two numbers in different Fields')
        num = (self.num + other.num) % self.prime
        return self.__class__(num, self.prime)

    def __sub__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot subtract two numbers in different Fields')
        num = (self.num - other.num) % self.prime
        return self.__class__(num, self.prime)

    def __mul__(self, other):
        if isinstance(other, int):
            other = FieldElement(other % self.prime, self.prime)
        if self.prime != other.prime:
            raise TypeError('Cannot multiply two numbers in different Fields')
        num = (self.num * other.num) % self.prime
        return self.__class__(num, self.prime)

    def __pow__(self, exponent):
        n = exponent % (self.prime - 1)
        num = pow(self.num, n, self.prime)
        return self.__class__(num, self.prime)

    def __truediv__(self, other):
        if self.prime != other.prime:
            raise TypeError('Cannot divide two numbers in different Fields')
        if other.num == 0:
            raise ZeroDivisionError
        inverse = pow(other.num, self.prime - 2, self.prime)
        num = (self.num * inverse) % self.prime
        return self.__class__(num, self.prime)

    def __rmul__(self, coefficient):
        num = (self.num * coefficient) % self.prime
        return self.__class__(num=num, prime=self.prime)
```

Testcases

Let a = 13 and b = 9 in F_{19}. In the cell(s) below, print results of the following computations:

1. a + b
2. a - b
3. a * b
4. 4 * a
5. a / b
6. a^{3086}

```
a = FieldElement(13, 19)
b = FieldElement(9, 19)
print(a + b)
print(a - b)
print(a * b)
print(a.__rmul__(4))
print(a / b)
print(a ** 3086)
```

```
FieldElement_19(3)
FieldElement_19(4)
FieldElement_19(3)
FieldElement_19(14)
FieldElement_19(12)
FieldElement_19(16)
```

6. Elliptic curve: Point Class (10 Points)

Define the class `Point` that represents a point on the elliptic curve.

- Complete the method `point_on_curve` that returns `True` if the point is on the elliptic curve.
- `__eq__` overloads equal operator for Points. Two points are equal if their (x,y) coordiantes and curve coeffiecients 'a' and 'b' are equal.
- `__ne__` should implement the inverse of '==' operator.

```
class Point:
    def __init__(self, x, y, a, b, prime=None):
        if prime and isinstance(x, int):
            x = FieldElement(x % prime, prime)
            y = FieldElement(y % prime, prime)
            a = FieldElement(a, prime)
            b = FieldElement(b, prime)
        self.x, self.y, self.a, self.b = x, y, a, b
        if x is None and y is None:
            return
        if not self.point_on_curve():
            raise ValueError(f"Point ({x},{y}) not on curve y^2 = x^3 + {a}x + {b}")

    def point_on_curve(self):
        if self.x is None and self.y is None:
            return True
        return self.y * self.y == self.x * self.x * self.x + self.a * self.x + self.b
```

```
    def __eq__(self, other):
        return (self.x == other.x and self.y == other.y and
                self.a == other.a and self.b == other.b)

    def __ne__(self, other):
        return not self.__eq__(other)
```

Run the following cells to test your code:

```
print(Point(-1, -1, 5, 7) == Point(-1, -1, 5, 7))
```

```
True
```

```
print(Point(-1, -1, 5, 7) != Point(-1, -1, 5, 7))
```

```
False
```

```
print(Point(3, 0, 5, 7))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-65-80d49044734c> in <cell line: 0>()
----> 1 print(Point(3, 0, 5, 7))

<ipython-input-61-b44b5b668260> in __init__(self, x, y, a, b, prime)
     10             return
     11         if not self.point_on_curve():
---> 12             raise ValueError(f"Point ({x},{y}) not on curve y^2 = x^3 + {a}x + {b}")
     13 
     14     def point_on_curve(self):

ValueError: Point (3,0) not on curve y^2 = x^3 + 5x + 7
```

Next steps: Explain error

7. Point Addition (10 Points)

Implement the `__add__` method for the Point class to overload the addition operator.

```
def __add__(self, other):
    if self.a != other.a or self.b != other.b:
        raise TypeError("Points on different curves")
    if self.x is None:
        return other
    if other.x is None:
        return self
    if self.x == other.x and self.y != other.y:
        return Point(None, None, self.a, self.b, self.x.prime if self.x else other.x.prime)
    if self != other:

        s = (other.y - self.y) / (other.x - self.x)
        x = s * s - self.x - other.x
        y = s * (self.x - x) - self.y
        return Point(x, y, self.a, self.b, self.x.prime)
    if self.y.num == 0:
        return Point(None, None, self.a, self.b, self.x.prime)

    s = (self.x * self.x * FieldElement(3, self.x.prime) + self.a) / (self.y * FieldElement(2, self.y.prime))
    x = s * s - self.x * FieldElement(2, self.x.prime)
    y = s * (self.x - x) - self.y
    return Point(x, y, self.a, self.b, self.x.prime)

Point.__add__ = __add__
```

Testcases

a.

In F_{223} on the elliptic curve secp256k1, what is the sum of the points:

1. $A = (192, 105)$ and $B = (17, 56)$?
2. $A = (192, 105)$ and $B = (192, 105)$?
3. $A = (143, 98)$ and $B = (76, 66)$?

Write code below that prints the outputs of the respective elliptic curve point addtions.

```
# Test cases
prime = 223
a = FieldElement(0, prime)
b = FieldElement(7, prime)
```

```
#1
p1 = Point(FieldElement(192, prime), FieldElement(105, prime), a, b)
p2 = Point(FieldElement(17, prime), FieldElement(56, prime), a, b)
print(p1 + p2)
```

```
<__main__.Point object at 0x78f99478ee90>
```

```
#2
p3 = Point(FieldElement(192, prime), FieldElement(105, prime), a, b)
print(p3 + p3)
```

```
<__main__.Point object at 0x78f99478f0d0>
```

```
#3
p4 = Point(FieldElement(143, prime), FieldElement(98, prime), a, b)
p5 = Point(FieldElement(76, prime), FieldElement(66, prime), a, b)
print(p4 + p5)
```

```
<__main__.Point object at 0x78f9ac18cb50>
```

```
p6 = Point(-1, -1, 5, 7)
p7 = Point(-1, 1, 5, 7)
print(p6 + p7)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-71-b70684f24fc6> in <cell line: 0>()
      1 p6 = Point(-1, -1, 5, 7)
      2 p7 = Point(-1, 1, 5, 7)
----> 3 print(p6 + p7)

<ipython-input-66-3d98aff4dbcf> in __add__(self, other)
      7             return self
      8         if self.x == other.x and self.y != other.y:
----> 9             return Point(None, None, self.a, self.b, self.x.prime if self.x else other.x.prime)
     10         if self != other:
     11             # Use FieldElement arithmetic for slope calculation

AttributeError: 'int' object has no attribute 'prime'
```

Next steps: Explain error

```
p6 = Point(-1, -1, 5, 7)
p7 = Point(-1, 1, 5, 7)
print(p6 + p7)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-72-d39e5561c076> in <cell line: 0>()
      1 p6 = Point(-1, -1, 5, 7)
      2 p7 = Point(-1, 1, 5, 7)
----> 3 print(p6 + p7)  # Point(infinity)

<ipython-input-66-3d98aff4dbcf> in __add__(self, other)
      7             return self
      8         if self.x == other.x and self.y != other.y:
----> 9             return Point(None, None, self.a, self.b, self.x.prime if self.x else other.x.prime)
     10         if self != other:
     11             # Use FieldElement arithmetic for slope calculation

AttributeError: 'int' object has no attribute 'prime'
```

Next steps: Explain error

```
p8 = Point(-1, -1, 5, 7)
p9 = Point(2, 5, 5, 7)
print(p8 + p9)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-73-3d50a2569d59> in <cell line: 0>()
      1 p8 = Point(-1, -1, 5, 7)
      2 p9 = Point(2, 5, 5, 7)
----> 3 print(p8 + p9)  # Point(18,-7)

<ipython-input-66-3d98aff4dbcf> in __add__(self, other)
     13             x = s * s - self.x - other.x
     14             y = s * (self.x - x) - self.y
---> 15             return Point(x, y, self.a, self.b, self.x.prime)
     16         if self.y.num == 0:  # Check if y is zero in the field
     17             return Point(None, None, self.a, self.b, self.x.prime)

AttributeError: 'int' object has no attribute 'prime'
```

Next steps: Explain error

b.

In the elliptic curve $y^2 = x^3 + 5x + 7$, What is the sum of the points:

1. $A = (-1, -1)$ and $B = (-1, 1)$?
2. $A = (-1, -1)$ and $B = (2, 5)$?

Write code below that prints the outputs of the respective elliptic curve point addtions.

```
# YOUR CODE HERE
prime = 31
a = FieldElement(5, prime)
b = FieldElement(7, prime)
p1 = Point(FieldElement(30, prime), FieldElement(30, prime), a, b)
p2 = Point(FieldElement(30, prime), FieldElement(1, prime), a, b)
print(p1 + p2)
p3 = Point(FieldElement(2, prime), FieldElement(5, prime), a, b)
print(p1 + p3)
```

```
<__main__.Point object at 0x78f9ac16d4d0>
<__main__.Point object at 0x78f994756b10>
```

8. Point Addition: Associativity (10 Points)

Let `A = (x1, y1)`, `B = (x2, y2)` and `C = (x3, y3)` be three points in finite field. Using these points, demonstrate that point addition is associative. Print `True` if point addition is associative. Otherwise, print `False`.

```
def check_associativity(A, B, C):
    left = (A + B) + C
    right = A + (B + C)
    return left == right
```

Testcases

Check the associativity for the following:

1. In F_{223} on the elliptic curve secp256k1, for the points $A = (192, 105)$, $B = (76, 157)$ and $C = (170, 142)$.
2. In F_{223} on the elliptic curve secp256k1, for the points $A = (192, 105)$, $B = (76, 157)$ and $C = (76, 66)$.
3. In F_{19} on the elliptic curve $y^2 = x^3 + 5x + 7$, for the points $A = (18, 18)$, $B = (18, 1)$ and $C = (2, 5)$.

```
# Test cases
```

```
prime = 223
a = FieldElement(0, prime)
b = FieldElement(7, prime)

A1 = Point(FieldElement(192, prime), FieldElement(105, prime), a, b)
B1 = Point(FieldElement(76, prime), FieldElement(157, prime), a, b)
C1 = Point(FieldElement(170, prime), FieldElement(142, prime), a, b)
print(check_associativity(A1, B1, C1))

A2 = Point(FieldElement(192, prime), FieldElement(105, prime), a, b)
B2 = Point(FieldElement(76, prime), FieldElement(157, prime), a, b)
C2 = Point(FieldElement(76, prime), FieldElement(66, prime), a, b)
print(check_associativity(A2, B2, C2))

prime = 19
a = FieldElement(5, prime)
b = FieldElement(7, prime)
A3 = Point(FieldElement(18, prime), FieldElement(18, prime), a, b)
B3 = Point(FieldElement(18, prime), FieldElement(1, prime), a, b)
C3 = Point(FieldElement(2, prime), FieldElement(5, prime), a, b)
print(check_associativity(A3, B3, C3))
```

```
True
True
True
```

9. Scalar Multiplication: Point (10 Points)

Define the `__rmul__` method in the Point class to implement scalar multiplication.

```
def __rmul__(self, k):
    result = Point(None, None, self.a, self.b)
    current = self
    while k > 0:
        if k & 1:
            result = result + current
        current = current + current
        k >>= 1
    return result

Point.__rmul__ = __rmul__
```

Testcases

In F_{223} on the elliptic curve secp256k1, compute and print the following for the point A:

a.) $7 \cdot A$ where $A = (173, 35)$

b.) $8 \cdot A$ where $A = (66, 111)$

```
prime = 223
a = FieldElement(0, prime)
b = FieldElement(7, prime)

A1 = Point(FieldElement(173, prime), FieldElement(35, prime), a, b)
print(7 * A1)

A2 = Point(FieldElement(66, prime), FieldElement(111, prime), a, b)
print(8 * A2)
```

```
<__main__.Point object at 0x78f9947a8c50>
<__main__.Point object at 0x78f9947a8bd0>
```

10. Invertibility (10 Points)

Write a function `additive_inverse` that returns the additive inverse of a given point.

```
def additive_inverse(A):
    if A.x is None and A.y is None:
        return A
    prime = A.x.prime
    inverse_y = FieldElement((prime - A.y.num) % prime, prime)
    return Point(A.x, inverse_y, A.a, A.b)
```

Tescases

Compute the additive inverse for the following points:

1. $A = (66, 111)$ in F_{223} on the elliptic curve secp256k1
2. $A = (-1, -1)$ in F_{31} on the elliptic curve $y^2 = x^3 + 5x + 7$

Also, compute the sum of the point A with its additive inverse in each case.

```
prime = 223
a = FieldElement(0, prime)
b = FieldElement(7, prime)
A1 = Point(FieldElement(66, prime), FieldElement(111, prime), a, b)
inv1 = additive_inverse(A1)
print(inv1)
print(A1 + inv1)

prime = 31
a = FieldElement(5, prime)
b = FieldElement(7, prime)
A2 = Point(FieldElement(-1 % prime, prime), FieldElement(-1 % prime, prime), a, b)
inv2 = additive_inverse(A2)
print(inv2)
print(A2 + inv2)
```

```
<__main__.Point object at 0x78f9947ab610>
<__main__.Point object at 0x78f9947ab8d0>
<__main__.Point object at 0x78f9947ab890>
<__main__.Point object at 0x78f9947aba90>
```

11. Discrete Logarithm Problem (15 Points)

Given the two points G=(`Gx`, `Gy`) and P=(`Px`, `Py`) on the secp256k1 elliptic curve with coordinates over the finite field F_p with p=`prime`, what is the a scalar s such that $sG = P$? Write a function that computes this s.

```
def GuessPrivateKey(P, G, prime):

    current = Point(None, None, G.a, G.b)
    for s in range(prime):
        if current == P:
            return s
        current = current + G
    return None
```

a.) Compute and print the value of s for the following choices:

`prime` $= 223$, $G = (154, 150)$ and $P = (47, 71)$.

```
prime = 223
a = FieldElement(0, prime)
b = FieldElement(7, prime)
G = Point(FieldElement(154, prime), FieldElement(150, prime), a, b)
P = Point(FieldElement(47, prime), FieldElement(71, prime), a, b)
print(GuessPrivateKey(P, G, prime))
```

```
19
```

b.) We recall that the prime order used in th finite field of Bitcoin's secp256k1 elliptic curve is $p_{\text{BTC}} := 2^{256} - 2^{32} - 977$, see, for example, [the corresponding section in Chapter 3 of "Programming Bitcoin]() or [this link]().

Compute and print the value of s for the following choices:

`prime` = p_{BTC}

`Gx` $= 0x754e3239f325570cdbbf4a87deee8a66b7f2b33479d468fbc1a50743bf56cc18$

`Gy` $= 0x0673fb86e5bda30fb3cd0ed304ea49a023ee33d0197a695d0c5d98093c536683$

`Px` $= 0x7e5c9db512f90042057f63659344a5dade96b7e2d8e7b0fdb66a1b87d7383004$

`Py` $= 0x7ce053860ea1d5a4cddc7f0774d1d55ae05335a1ee229b3375bc0732cae1eeb1$

```
p_btc = 2**256 - 2**32 - 977
a = FieldElement(0, p_btc)
b = FieldElement(7, p_btc)
Gx = FieldElement(0x754e3239f325570cdbbf4a87deee8a66b7f2b33479d468fbc1a50743bf56cc18, p_btc)
Gy = FieldElement(0x0673fb86e5bda30fb3cd0ed304ea49a023ee33d0197a695d0c5d98093c536683, p_btc)
Px = FieldElement(0x7e5c9db512f90042057f63659344a5dade96b7e2d8e7b0fdb66a1b87d7383004, p_btc)
Py = FieldElement(0x7ce053860ea1d5a4cddc7f0774d1d55ae05335a1ee229b3375bc0732cae1eeb1, p_btc)
G = Point(Gx, Gy, a, b)
P = Point(Px, Py, a, b)
print("Too large to compute with brute force")
```

```
Too large to compute with brute force
```

c.) Assume that the parameters are chosen as follows:

`prime` = p_{BTC}

G = Generator Point used in Bitcoin's ECDSA scheme

`Px` $= 0x7e5c9db512f90042057f63659344a5dade96b7e2d8e7b0fdb66a1b87d7383004$

`Py` $= 0x7ce053860ea1d5a4cddc7f0774d1d55ae05335a1ee229b3375bc0732cae1eeb1$

Note: The Generator Point used in Bitcoin's ECDSA implementation is G = (`Gx`,`Gy`), where

`Gx` $= 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798$

`Gy` $= 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8$

Are you able to find s for this choice of parameters? If yes, print it, if not, provide an estimate of how long it might take your computer to compute is and explain this estimate.

```
Gx = FieldElement(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798, p_btc)
Gy = FieldElement(0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8, p_btc)
G = Point(Gx, Gy, a, b)
print("Computationally infeasible")
```

```
Computationally infeasible
```

Add your explanation here

For Bitcoin's secp256k1, the order is approximately 2^256. Brute forcing would take 2^256 operations. At 10^9 operations/second, this would take ~10^68 seconds, or ~3×10^60 years, far exceeding the age of the universe (~10^10 years).