WEB3CLUBS FOUNDATION LIMITED

Course Instructor: DR. Cyprian Omukhwaya Sakwa
PHONE: +254723584205 Email: cypriansakwa@gmail.com

Foundational Mathematics for Web3 Builders

Implemented in RUST

Lecture 42

August 12, 2024

Order of an element of a Group

Definition 15

Let G be a group. The order of an element $g \in G$ is the smallest positive integer n such that $g^n = e$ if such n exists. If no such integer n exists then g has infinite order. We denote order of an element g by $|g|$.

From above definition, to determine the order of an element g we repeatedly multiply or (operate) g by itself. That is, we compute $g; g^2; g^3; \cdots$ until we reach the identity element for the first time. If the binary operation on G is addition then the notation $n \cdot g$ or ng is used instead of g^n.

$\frac{7}{\gcd(7,5)} = \frac{7}{1} = \underline{\underline{7}}$

Example 75

Let $G = (\mathbb{Z}_7, +)$. Find $|5|$. $|5| = \underline{\underline{7}}$

Solution

The identity is 0 since G is an additive group. We add 5 to itself until we obtain 0. Thus $1 \cdot 5 = 5$, $2 \cdot 5 = 3$, $3 \cdot 5 = 1$, $4 \cdot 5 = 6$, $5 \cdot 5 = 4$, $6 \cdot 5 = 2$, $7 \cdot 5 = 0$. Thus $|5| = 7$.

Example 76

$\frac{\text{order of group}}{\gcd(\text{order group, element})}$

Consider $G = (\mathbb{Z}_{10}, +)$. Find $|6|$ and $|3|$.

Solution

$\frac{10}{\gcd(10,3)} = \frac{10}{1} = 10$

$1 \cdot 6 = 6$, $2 \cdot 6 = 2$, $3 \cdot 6 = 8$, $4 \cdot 6 = 4$, $5 \cdot 6 = 0$. Thus $|6| = 5$.

And $1 \cdot 3 = 3$, $2 \cdot 3 = 6$, $3 \cdot 3 = 9$, $4 \cdot 3 = 2$, $5 \cdot 3 = 5$, $6 \cdot 3 = 8$, $7 \cdot 3 = 1$, $8 \cdot 3 = 4$, $9 \cdot 3 = 7$, $10 \cdot 3 = 0$. Thus $|3| = 10$.

From above examples, notice that the order of $a \in \mathbb{Z}_n$ is n if a is relatively prime to n.

Example 77

Let $G = (\mathbb{Z}_{10}^*, \times)$. Find:

a) $|9|$.

b) $|7|$.

Solution

a) The identity is 1 since G is a multiplicative group. We multiply 9 by itself until we obtain 1. Thus $9^1 = 9$, $9^2 = 1$. Thus $|9| = 1$.

b) $7^1 = 7$, $7^2 = 9$, $7^3 = 3$, $7^4 = 1$. Thus $|7| = 4$.

Let $G = \mathbb{Z}_n$ be a group of order n and suppose that $m \in G$. Then, the order of m is $\frac{n}{\gcd(n,m)}$.

Example 78

Find the order of element $6 \in \mathbb{Z}_{10}$.

Solution

$|\mathbb{Z}_{10}| = 10$, $\gcd(6, 10) = 2$. Therefore, $|6| = \frac{10}{2} = 5$.

Let us use the following rust code to determine order of elements of a group $\mathbb{Z}_n$.

```rust
use num_bigint::BigUint;
use num_traits::{One, Zero};

fn find_order(n: &BigUint, a: &BigUint) -> BigUint {
        let mut k = BigUint::one();
        let mut sum = a.clone();

        while &sum % n != BigUint::zero() {
                k += BigUint::one();
                sum += a;
        }

        k
}

fn main() {
        let n = BigUint::parse_bytes(b"8", 10).unwrap();
        // Change this value to any positive integer to represent Z_n

        for a in 0..n.to_u32_digits()[0] {
                let a_biguint = BigUint::from(a as u32);
                let order = find_order(&n, &a_biguint);
                println!("Order␣of␣{}␣in␣Z_{}␣is␣{}", a, n, order);
        }
}
```

$sum \% n \neq 0$

$18 \% 8 \neq 0$

let $K = 4$ ✓

$sum = 18 + 6 = 24$

$24 \% 8 = 0$

$|6| = \underline{\underline{4}}$

Understanding the Rust code

This code effectively finds the order of each element in the additive group $\mathbb{Z}_n$ for the given modulus n.The code finds the smallest integer k such that $k \cdot a \bmod n = 0$ for each a in the range $0, \cdots, n-1$

- num_bigint::BigUint: BigUint is here since it can handle extremely large numbers, which is useful in cryptography and mathematical applications.
- num_traits::One, Zero: These traits provide methods to create 1 and 0 values for BigUint.

1. find_order Function

```
fn find_order(n: &BigUint, a: &BigUint) -> BigUint {
        let mut k = BigUint::one();
        let mut sum = a.clone();

        while &sum % n != BigUint::zero() {
                k += BigUint::one();
                sum += a;
        }

        k
}
```

Understanding the Rust code (conti...)

- Inputs: n, the modulus, representing the size of the additive group $\mathbb{Z}_n$ and a The element for which we want to find the order.
- Process:
  - ✓ k is initialized to 1, representing the current multiplier.
  - ✓ sum is initialized as a clone of a. This represents the accumulated sum as we add a repeatedly.
  - ✓ The while loop continues to increment k and add a to sum until sum % n $==$ 0. This means that the accumulated sum is divisible by n, implying that the order has been found.
  - ✓ The order is the smallest integer k such that $k \cdot a \equiv 0 \bmod n$.

Understanding the Rust code (conti...)

(2) main Function

```
fn main() {
        let n = BigUint::parse_bytes(b"8", 10).unwrap();
        // Change this value to any positive integer to represent Z_n

        for a in 0..n.to_u32_digits()[0] {
                let a_biguint = BigUint::from(a as u32);
                let order = find_order(&n, &a_biguint);
                println!("Order␣of␣{}␣in␣Z_{}␣is␣{}", a, n, order);
        }
}
```

- Sets up the group by defining n, the modulus consisting of elements $0, 1, \cdots, n-1$.
- Loops through elements. The for loop iterates over each integer a from 0 to $n-1$.
- a_biguint converts a from a u32 to a BigUint to be compatible with the find_order function.
- Finds and prints the order. For each a, the find_order function is called to compute its order in $\mathbb{Z}_n$.

Let us use the following rust code to determine order of elements of a group $\mathbb{Z}_n^*$.

```rust
use num_bigint::BigUint;
use num_traits::{One, Zero};

fn gcd(a: &BigUint, b: &BigUint) -> BigUint {
        let mut x = a.clone();
        let mut y = b.clone();
        while y != BigUint::zero() {
                let temp = y.clone();
                y = x.clone() % y.clone();
                x = temp;
        }
        x
}

fn order_of_element(a: &BigUint, n: &BigUint) -> Option<BigUint> {
        if gcd(a, n) != BigUint::one() {
                return None;
// Element ‘a‘ is not coprime with ‘n‘, so it doesn’t have an order in the group.
        }

        let mut k = BigUint::one();
        let mut power = a.clone() % n.clone();
        while power != BigUint::one() {
                power = (power * a.clone()) % n.clone();
                k += BigUint::one();
        }
        Some(k)
}
```

```

fn main() {
        let n = BigUint::parse_bytes(b"77787642", 10).unwrap(); // Modulus

        // Generate a range of elements up to a specified value
        let max_element = 20; // Small value for demonstration

let elements: Vec<BigUint> =
(1..=max_element as u32).map(|x| BigUint::from(x)).collect();

        for element in &elements {
                match order_of_element(&element, &n) {
Some(order) => println!("The␣order␣of␣{}␣in␣Z_{}␣is␣{}", element, n, order),
None => println!("Element␣{}␣is␣not␣coprime␣with␣{}", element, n),
                }
        }
}
```

Understanding the Rust code

This Rust code calculates the order of elements in the multiplicative group $\mathbb{Z}_n^*$ with a given modulus n. It determines a smallest positive integer k such that $a^k \equiv 1 \bmod n$ for each element a that is coprime with n.

- num_bigint::BigUint: A library that provides arbitrary-precision arithmetic on unsigned integers (BigUint).
- num_traits::One, Zero: Traits that provide methods to create 1 and 0 values for BigUint.

1. gcd Function

```
fn gcd(a: &BigUint, b: &BigUint) -> BigUint {
        let mut x = a.clone();
        let mut y = b.clone();
        while y != BigUint::zero() {
                let temp = y.clone();
                y = x.clone() % y.clone();
                x = temp;
        }
        x
}
```

Understanding the Rust code (conti...)

- Inputs a and b, the two BigUint values to calculate the greatest common divisor $\gcd$.
- The function uses the Euclidean algorithm to compute the $\gcd$.
- It iteratively computes the remainder of x divided by y until y equals zero. The last non-zero value of x is the $\gcd$.
- The function returns the $\gcd$ of a and b.

(2) order_of_element Function

```
fn order_of_element(a: &BigUint, n: &BigUint) -> Option<BigUint> {
        if gcd(a, n) != BigUint::one() {
                return None;
// Element ‘a‘ is not coprime with ‘n‘, so it doesn’t have an order in the group.
        }

        let mut k = BigUint::one();
        let mut power = a.clone() % n.clone();
        while power != BigUint::one() {
                power = (power * a.clone()) % n.clone();
                k += BigUint::one();
        }
        Some(k)
}
```

Understanding the Rust code (conti...)

- Inputs a the element to establish the order and n the modulus, representing the group $\mathbb{Z}_n^*$.
- First, it uses the $\gcd$ function to determine whether a is coprime with n. If the $\gcd$ is not 1, a does not have an order in $\mathbb{Z}_n^*$ (i.e., it is not part of the multiplicative group), and the function returns None.
- It initializes k to 1 and computes power $= a \bmod n$.
- It repeatedly multiplies power by a modulo n, incrementing k each time, until power equals 1.
- When power equals 1, the current value of k is the order of a in $\mathbb{Z}_n^*$.
- The function returns Some(k) if an order is found, or None if a is not coprime with n.

Understanding the Rust code (conti...)

(3) main Function

```
fn main() {
        let n = BigUint::parse_bytes(b"77787642", 10).unwrap(); // Modulus

        // Generate a range of elements up to a specified value
        let max_element = 20; // Small value for demonstration

let elements: Vec<BigUint> =
(1..=max_element as u32).map(|x| BigUint::from(x)).collect();

        for element in &elements {
                match order_of_element(&element, &n) {
Some(order) => println!("The␣order␣of␣{}␣in␣Z_{}␣is␣{}", element, n, order),
None => println!("Element␣{}␣is␣not␣coprime␣with␣{}", element, n),
                }
        }
}
```

- Sets up the modulus n.
- max_element is set meaning the code will check the order of all elements from 1 to that maximum element.
- The elements are stored in a vector of BigUint values.

Understanding the Rust code (conti...)

- The code iterates over each element in the vector.
- For each element, it calls order_of_element to determine the order.
- f the order is found, it prints the order. If the element is not coprime with n, it prints a message indicating that.