# WEB3CLUBS FOUNDATION LIMITED

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

# Foundational Mathematics for Web3 Builders

## Implemented in RUST

## Lecture 45

**August 22, 2024**

# Cyclic Groups

**Definition 18**

A group $G$ is considered cyclic if there exists an element $g \in G$ that generates the entire $G$.

- The definition indicates that a group $G$ is cyclic if there is an element $g \in G$ such that $\langle g \rangle = G$.
- The element $g \in G$ is hence known as a generator.
- Note that a cyclic group can contain several generators.
- If the generator $g$ has order $n$, so does the cyclic group $G = \langle g \rangle$.
- The cyclic group $G = \langle g \rangle$ has infinite order if $|g|$ is infinite.
- Note that if $G$ is generated by $g \in G$, then $G$ is also generated by $g^{-1}$ since $\langle g \rangle = \langle g^{-1} \rangle$.

- Cyclic groups play an important role in modern cryptography, particularly public-key cryptographic protocols.
- Many cryptographic algorithms rely on cyclic group properties to ensure system security. This is because cyclic groups possess well-defined mathematical structures that enable both creating and breaking of cryptographic protocols.
- Cyclic groups are also employed in the design of cyclic codes, which are useful for error detection and correction.

# 22.1 Examples of Cyclic Groups

1) The group of integers $(\mathbb{Z}, +)$ is cyclic. Note that $\mathbb{Z}$ is generated by $1$ and $-1$.

   Generators in the additive group are elements that, when added repeatedly, can generate all of the group's elements.

2) Integers mod $n$ are cyclic. That is, if $n$ is a positive integer then $\mathbb{Z}_n$ is a cyclic group of order $n$ and is generated by $1$.

$\mathbb{Z}_n$

   An element $g^m$ of a finite cyclic group $G$ of order $n$ is a generator of $G$ if $n$ and $m$ are relatively prime. This implies that, for instance, every nonzero element in $\mathbb{Z}_7$ is a generator to $\mathbb{Z}_7$.

   For example, $1 \cdot 1 = 1,\ 2 \cdot 1 = 2,\ 3 \cdot 1 = 3,\ 4 \cdot 1 = 4,\ 5 \cdot 1 = 5,\ 6 \cdot 1 = 6,\ 7 \cdot 1 = 0$. Thus $\langle 1 \rangle = \{0, 1, 2, 3, 4, 5, 6\} = \mathbb{Z}_7$. Therefore $1$ generates $\mathbb{Z}_7$.

$1\cdot2=2,\ 2\cdot2=4,\ 3\cdot2=6,\ 4\cdot2=1,\ 5\cdot2=3,\ 6\cdot2=5,\ 7\cdot2=0.$
Thus $\langle 2\rangle=\{0,1,2,3,4,5,6\}=\mathbb{Z}_7$ meaning that 2 generates $\mathbb{Z}_7$.
$1\cdot3=3,\ 2\cdot3=6,\ 3\cdot3=2,\ 4\cdot3=5,\ 5\cdot3=1,\ 6\cdot3=4,\ 7\cdot3=0.$
Thus $\langle 3\rangle=\{0,1,2,3,4,5,6\}=\mathbb{Z}_7$ and 3 generates $\mathbb{Z}_7$. Similarly, $4,5$ and $6$ generate $\mathbb{Z}_7$.

**Example 89**

List the generators of

a) $\mathbb{Z}_{11}$

b) $\mathbb{Z}_{20}$

**Solution**

a) 11 is prime hence the generators of $\mathbb{Z}_{11}$ are $1,2,3,4,5,6,7,8,9,10$.

## Solution (conti...)

b) Generators of $\mathbb{Z}_{20}$ are $1, 3, 7, 9, 11, 13, 17, 19$. These are elements in
$0, 1, 2, \cdots, 19$ which are relatively prime to $20$. These $8$ elements will generate $\mathbb{Z}_{20}$. For example,

$$
\begin{array}{lll}
1 \cdot 17 = 17 & 8 \cdot 17 = 16 & 15 \cdot 17 = 15 \\
2 \cdot 17 = 14 & 9 \cdot 17 = 13 & 16 \cdot 17 = 12 \\
3 \cdot 17 = 11 & 10 \cdot 17 = 10 & 17 \cdot 17 = 9 \\
4 \cdot 17 = 8 & 11 \cdot 17 = 7 & 18 \cdot 17 = 6 \\
5 \cdot 17 = 5 & 12 \cdot 17 = 4 & 19 \cdot 17 = 3 \\
6 \cdot 17 = 2 & 13 \cdot 17 = 1 & 20 \cdot 17 = 0 \\
7 \cdot 17 = 19 & 14 \cdot 17 = 18 &
\end{array}
$$

3) The set $\mathbb{Z}_n^*$ of residue classes with $\gcd(a, n) = 1$ under multiplication is a cyclic group for most $n$ except for specific cases like $n = 4k$ where $k \geq 2$.

To find generators of the group of units $\mathbb{Z}_n^*$ follow the following steps. $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$

a) Compute Euler's Totient Function $\phi(n)$. This will give the order of $\mathbb{Z}_n^*$. $a^{|n|} = 1$

b) Determine the integers less than $n$ that are coprime to $n$.

c) Find the order of each Element of the above elements coprime to $n$. Note that the order of $g$ is the smallest positive integer $k$ such that $g^k \equiv 1 \bmod n$.

   Note that $g \in \mathbb{Z}_n^*$ is a generator of $\mathbb{Z}_n^*$ if its order equal to the order of $\mathbb{Z}_n^*$ which equals $\phi(n)$.

**Example 90**

Is $\mathbb{Z}_{10}^*$ cyclic?. Show your workings.

**Solution**

We are required to determine if $\mathbb{Z}_{10}^*$ has a generator.

First, $\phi(10) = 4$ so the order of $\mathbb{Z}_{10}^*$ is $4$.

A generator to this group should be a positive element $g$ which is less than $10$ relatively prime to $10$ and of order $4$.

Elements relatively prime to $10$ are $1, 3, 7, 9$.

We disqualify $1$ since its order is $1$.

Checking for $3$ we get,

$3^1 = 3,\ 3^2 = 9,\ 3^3 = 7,\ 3^4 = 1$ and $\langle 3 \rangle = \{1, 3, 7, 9\} = \mathbb{Z}_{10}^*$ meaning that $3$ generates $\mathbb{Z}_{10}^*$.

$\mathbb{Z}_{10}^*$ is therefore cyclic.

## Solution (conti...)

If we wanted to find the other generators we would continue in a similar way.

$7^1 = 7,\ 7^2 = 9,\ 7^3 = 3,\ 7^4 = 1$ and $\langle 7 \rangle = \{1, 3, 7, 9\} = \mathbb{Z}_{10}^*$ meaning that 7 also generates $\mathbb{Z}_{10}^*$.

Let us now check for 9.

$9^1 = 9,\ 9^2 = 1$ and so $\langle 9 \rangle = \{1, 9\}$ thus 9 does not generate $\mathbb{Z}_{10}^*$.

## Example 91

For $k \geq 2$, $\mathbb{Z}_{4k}^* = \mathbb{Z}_8^*, \mathbb{Z}_{16}^*, \mathbb{Z}_{20}^*, \mathbb{Z}_{24}^*, \cdots$ are not cyclic for lack of generators.

## Example 92

Find the generators of $\mathbb{Z}_{22}^*$.

$$\begin{array}{r|l} 2 & 22 \\ \hline 11 & 11 \\ \hline \end{array}$$

$$\frac{1}{2} \times \frac{10}{11} \times 22 = \underline{\underline{10}}$$

## Solution

The order of $\mathbb{Z}_{22}$ is $\phi(22) = 10$.

Numbers less than $22$ and relatively prime to $22$ are $1, 3, 5, 7, 9, 13, 15, 17, 19, 21$.

We disqualify $1$ since its order is $1$ and then test the remaining elements.

Let us find the order of $3$.

$$3^1 = 3$$
$$3^2 = 9$$
$$3^3 = 5$$
$$3^4 = 15$$
$$3^5 = 1$$

$\therefore, |3| = 5$

So, $3$ is not a generator.

Now test for $5$.

$$5^1 = 5$$
$$5^2 = 3$$
$$5^3 = 15$$
$$5^4 = 9$$
$$5^5 = 1$$

$\therefore, |5| = 5$

So, $5$ is not a generator.

## Solution

Now test for 7.

$$7^1 = 7$$

$$7^2 = 5$$

$$7^3 = 13$$

$$7^4 = 3$$

$$7^5 = 21$$

$$7^6 = 15$$

$$7^7 = 17$$

$$7^8 = 9$$

$$7^9 = 19$$

$$7^{10} = 1$$

$\therefore, \mid 7 \mid = 10$

So, 7 is a generator.

Continuing with this gives the generators of $\mathbb{Z}_{22}$ as $7, 13, 17, 19$

The following Rust program finds and lists the generators of the additive group $\mathbb{Z}_n$, where $\mathbb{Z}_n$ represents the set of integers $\{0, 1, 2, \cdots, n-1\}$ under addition modulo $n$. In this regard, a generator is an element that can be used to generate all of the group's elements by repeated addition modulo $n$.

```
        fn main() {
            let n = 19; // Example value for n
            let generators = find_generators(n);

println!("Generators of the additive group Z_{}: {:?}", n, generators);
        }

        /// Finds the generators of the additive group Z_n
        fn find_generators(n: u32) -> Vec<u32> {
            let mut generators = Vec::new();

            for candidate in 1..n {
                if is_generator(candidate, n) {
                    generators.push(candidate);
                }
            }

            generators
        }

```

```
/// Checks if a given number is a generator of the additive group Z_n
        fn is_generator(g: u32, n: u32) -> bool {
                let mut current = g;
                let mut count = 1;

                while current != 0 {
                        current = (current + g) % n;
                        count += 1;
                        if count == n {
                                break;
                        }
                }

                count == n
        }

```

## Understanding the Rust code

1. Main Function: The main function sets the value of $n$, calls the find_generators function, and prints the generators.
2. find_generators function: This function returns a list of generators for the additive group $\mathbb{Z}_n$.

```
fn find_generators(n: u32) -> Vec<u32> {
        let mut generators = Vec::new();

        for candidate in 1..n {
                if is_generator(candidate, n) {
                        generators.push(candidate);
                }
        }

        generators
}
```

## Understanding the Rust code

- It initializes an empty vector generators to store the generator candidates. It iterates from $1$ to $n-1$ and checks if each number is a generator using the is_generator function.
- If a candidate is a generator, it is added to the generators vector. It returns a vector of all generators found.

3. is_generator Function: This function checks if the given number $g$ is a generator for the group $\mathbb{Z}_n$.

```
fn is_generator(g: u32, n: u32) -> bool {
        let mut current = g;
        let mut count = 1;

        while current != 0 {
                current = (current + g) % n;
                count += 1;
                if count == n {
                        break;
                }
        }

        count == n
}
```

## Understanding the Rust code

- It initializes current with the value of $g$ and a counter count starting at $1$.
- It enters a loop that continues until current becomes $0$.
- In each iteration, it adds $g$ to current, takes the result modulo $n$, and increments the counter.
- If count reaches $n$, the loop breaks.
- After the loop, it checks if count is equal to $n$. If true, $g$ is considered a generator; otherwise, it is not.

This Rust program determines the generators of the multiplicative group $Z_n^*$, which consists of all integers modulo $n$ that are coprime with $n$ under multiplication. The code determines these generators by applying Euler's Totient function and prime factorization. The program uses the num-bigint, num-integer, and num-traits crates to handle arbitrary precision integers and mathematical operations.

```rust
use num_bigint::{BigUint};
use num_integer::Integer;
use num_traits::{One};
use std::collections::HashSet;

// Compute Euler's Totient Function phi(n)
fn euler_totient(n: &BigUint) -> BigUint {
        let mut result = n.clone();
        let mut p = BigUint::from(2u32);
        let mut temp_n = n.clone();
        // Create a mutable copy of n for modification
        while &p * &p <= temp_n {
                if temp_n.is_multiple_of(&p) {
                        while temp_n.is_multiple_of(&p) {
                                temp_n /= &p;
                        }
                        result -= &result / &p;
                }
                p += 1u32;
        }
```

```
        if temp_n > BigUint::from(1u32) {
                result -= &result / &temp_n;
        }
        result
}

// Check if a number is a generator of Z_n^*
fn is_generator
(g: &BigUint, n: &BigUint, phi_n: &BigUint, factors: &HashSet<BigUint>)
 -> bool {
        for factor in factors {
                let exponent = phi_n / factor;
                if g.modpow(&exponent, n) == BigUint::one() {
                        return false;
                }
        }
        true
}

// Get the prime factors of a number
fn prime_factors(n: &BigUint) -> HashSet<BigUint> {
        let mut factors = HashSet::new();
        let mut num = n.clone();
        let mut p = BigUint::from(2u32);
        while &p * &p <= num {
```

```
            if num.is_multiple_of(&p) {
                factors.insert(p.clone());
                while num.is_multiple_of(&p) {
                    num /= &p;
                }
            }
            p += 1u32;
        }
        if num > BigUint::from(1u32) {
            factors.insert(num);
        }
        factors
}

fn main() {
        let n = BigUint::parse_bytes(b"22", 10).unwrap();
         // Change this value to test with different n
        let phi_n = euler_totient(&n);
        let factors = prime_factors(&phi_n);

        let mut generators = Vec::new();
        let mut candidate = BigUint::from(2u32);
        while &candidate < &n {
if candidate.gcd(&n) == BigUint::one() && is_generator(&candidate, &n, &phi_n, &factors)
                generators.push(candidate.clone());
            }
            candidate += BigUint::one();
        }

println!("Generators of the multiplicative group Z_{}^*: {:?}", n, generators);
}
```

## Understanding the Rust code

1. Euler's Totient Function $\phi(n)$: Computes the Euler's Totient function $\phi(n)$, which is essential for finding the order of the multiplicative group $\mathbb{Z}_n^*$.

```
fn euler_totient(n: &BigUint) -> BigUint {
        let mut result = n.clone();
        let mut p = BigUint::from(2u32);
        let mut temp_n = n.clone();
        // Create a mutable copy of n for modification
        while &p * &p <= temp_n {
                if temp_n.is_multiple_of(&p) {
                        while temp_n.is_multiple_of(&p) {
                                temp_n /= &p;
                        }
                        result -= &result / &p;
                }
                p += 1u32;
        }
        if temp_n > BigUint::from(1u32) {
                result -= &result / &temp_n;
        }
        result
}
```

## Understanding the Rust code (conti...)

- The function iterates through potential prime factors $p$ of $n$.
- If $n$ is divisible by $p$, it repeatedly divides $n$ by $p$ and updates the result by subtracting $\frac{\text{result}}{p}$.
- If after checking all potential prime factors, $n$ is still greater than 1, it means $n$ itself is prime, and the function adjusts the result accordingly.

2. Check for Generator: Determines if a given number $g$ is a generator of the multiplicative group $\mathbb{Z}_n^*$.

```
fn is_generator
(g: &BigUint, n: &BigUint, phi_n: &BigUint, factors: &HashSet<BigUint>)
 -> bool {
        for factor in factors {
                let exponent = phi_n / factor;
                if g.modpow(&exponent, n) == BigUint::one() {
                        return false;
                }
        }
        true
}
```

## Understanding the Rust code (conti...)

- The function checks if $g^{\frac{\phi(n)}{factor}} \mod n$ is equal to $1$ for any prime factor of $\phi(n)$. If it equals $1$ for any factor, $g$ is not a generator.
- If it never equals $1$, then $g$ is a generator.

3. Prime Factorization: Finds all prime factors of $n$ and returns them in a set.

```rust
fn prime_factors(n: &BigUint) -> HashSet<BigUint> {
        let mut factors = HashSet::new();
        let mut num = n.clone();
        let mut p = BigUint::from(2u32);
        while &p * &p <= num {
                if num.is_multiple_of(&p) {
                        factors.insert(p.clone());
                        while num.is_multiple_of(&p) {
                                num /= &p;
                        }
                }
                p += 1u32;
        }
        if num > BigUint::from(1u32) {
                factors.insert(num);
        }
        factors
}
```

## Understanding the Rust code (conti...)

- The function iteratively divides $n$ by potential prime factors $n$ and adds these factors to a set.
- If $n$ has any remaining factor greater than $1$ after checking up to $\sqrt{n}$, that factor is added as well.

4. Main Function: This is the entry point of the program. It calculates and prints the generators of the multiplicative group $\mathbb{Z}_n^*$ for a given $n$.

- Parses the input $n$.
- Computes $\phi(n)$ using the euler_totient function.
- Finds the prime factors of $\phi(n)$.

## Understanding the Rust code (conti...)

- Iterates through all candidates $g$ from $2$ to $n-1$ to check if they are generators:
  - ▷ A candidate $g$ must be coprime with $n$.
  - ▷ It checks if the candidate is a generator using the is_generator function.
  - ▷ f it is, it adds the candidate to the list of generators.
- Prints the list of generators.