WEB3CLUBS FOUNDATION LIMITED

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

Foundational Mathematics for Web3 Builders

Implemented in RUST ✓

Lecture 24

June 24, 2024

Basic properties of the integers

- Number theory and algebra are the basis for a large portion of contemporary cryptography.
- The study of the integers $\mathbb{Z} = \{\cdots, -4, -3, -2, -1, 0, 1, 2, 3, \cdots\}$ is known as number theory.
- Some of the fundamental characteristics of integers are covered in this chapter, including the concepts of primality and divisibility, unique factorization into primes, greatest common divisors, and least common multiples.
- We will study these properties and implement them in Rust.

Divisibility and primality

Definition 1

An integer $a \neq 0$ is called a divisor (factor) of an integer b (written $a \mid b$) if $b = ac$ for some $c \in \mathbb{Z}$. We also say that b is a multiple of a, or that b is divisible by a. If a does not divide b, then we write $a \nmid b$.

Example 1

a) $2 \mid 8$ because $8 = 2 \cdot 4$
b) $4 \mid -20$ since $-20 = 4(-5)$
c) $3 \nmid 16$ since when we try to divide 16 by 3 we get a remainder.

- To divide big numbers such as $a =$594160336580041203135554245134910186153170211762681749587249714049 13 by $b =$27345537128851 we could use the Rust code below.

$a =$59416033658004120313555424513491018615317021176268174958724971404913 by
$b =$27345537128851 we could use the Rust code below.

```rust
use num_bigint::BigUint;//handles arbitrarily large unsigned integers
use num_integer::Integer;//offers methods for integer operations,
//such as division with remainder.

fn main() {
        // Large number as a string
        let large_number
= "218882428718392752222464057452572750886963111572978236626890378946452262 08587";
        // Divisor as a string
        let divisor = "21727872222088128656872848376251920012568561598304323 62";

// Parse the numbers as BigUint.  //Unwrap() handles any potential mistakes
        // during parsing by halting the application if parsing fails.

        let num = BigUint::parse_bytes(large_number.as_bytes(), 10).unwrap();
        let div = BigUint::parse_bytes(divisor.as_bytes(), 10).unwrap();

        // Perform division and note the remainder.
        //div_rem is used to divide num by div, and returns a tuple
        // containing the quotient and the remainder.
```

```
        let (quotient, remainder) = num.div_rem(&div);

        // Print the result
        println!("Quotient: {}", quotient);
        println!("Remainder: {}", remainder);
}
```

The output is

Quotient: 217278722220881286568728483762519200125685615 9830432363

Remainder: 0 and so $b \mid a$

When we divide

$c =$218882428718392752222464057452572750886963111572978236626 89037894645226208587

by $d =$2172787222208812865687284837625192001256856159830432362

The output is

Quotient: 10073808722783318335649

Remainder: 10073808722783318335649 and so $d \nmid c$.

Definition 2 (Primes and composites)

If n is a positive integer greater than 1 and no other positive integers besides 1 and n divide n then we say n is prime.
If $n > 1$ but n is not prime, then n is said to be composite. That is, $n \in \mathbb{Z}$ is composite if and only if $n = ab$ for some $a, b \in \mathbb{Z}$.
The first few primes are $2, 3, 5, 7, 11, 13, 17, 19, \cdots$.

- To list prime numbers up to a particular number n, we can use the Sieve of Eratosthenes algorithm, which is a fast approach to locate all primes less than or equal to n. Here's the Rust code to do this: 1 120

```
fn main() {
        let start = 1; // Starting point
        let end = 144; // Ending point

        //Now check whether the starting point is greater than the finishing point.
        //If this is the case, the function terminates with an error message.
```

```
        //then calls sieve_of_eratosthenes_in_range to compute the prime
        //numbers in the specified range
        if start > end {
println!("Invalid␣range:␣start␣({})␣is␣greater␣than␣end␣({})", start, end);
                return;
        }

        let primes = sieve_of_eratosthenes_in_range(start, end);

println!("Prime␣numbers␣between␣{}␣and␣{}␣are:␣{:?}", start, end, primes);
}

/// Use the Eratosthenes Sieve to generate prime numbers within a specific range.
//If the end is less than two, this technique produces an empty vector since there
//are no prime numbers fewer than two.
fn sieve_of_eratosthenes_in_range(start: usize, end: usize) -> Vec<usize> {
        if end < 2 {
                return vec![];
        }
        //At initialization, A vector vector is constructed and used to
        //indicate if an integer is prime (true) or not prime.
        //Goes from two to the square root of the end. For each number num,
        // if is_prime[num] is true,it marks all multiples of num as false (non-prime).
```

==

true $\sqrt{144}$

```rust
let mut is_prime = vec![true; end + 1];
        is_prime[0] = false;
        is_prime[1] = false;

        for num in 2..=((end as f64).sqrt() as usize) {
                if is_prime[num] {
                        for multiple in (num * num..=end).step_by(num) {
                                is_prime[multiple] = false;
                        }
                }
        }
        //Collects and returns the prime numbers in the specified range
        is_prime.iter()
        .enumerate()
        .filter(|&(num, &prime)| prime && num >= start)
        .map(|(num, _)| num)
        .collect()
}
```

Using this code prints;

Prime numbers between 1 and 120: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]

- You might use the same Rust code to find prime numbers within a given range.
- For example, the primes between 200000000 and 200000900 are: It prints Prime numbers between 200000000 and 200000900:

[200000033, 200000039, 200000051, 200000069, 200000081, 200000083, 200000089, 200000093, 200000107, 200000117, 200000123, 200000131, 200000161, 200000183, 200000201, 200000209, 200000221, 200000237, 200000239, 200000243, 200000299, 200000321, 200000329, 200000347, 200000377, 200000399, 200000417, 200000431, 200000447, 200000453, 200000477, 200000483, 200000491, 200000513, 200000527, 200000531, 200000543, 200000551, 200000579, 200000677, 200000719, 200000729, 200000777, 200000797, 200000803, 200000819, 200000831, 200000833, 200000863, 200000881, 200000891] ✓

- The Eratosthenes Sieve is excellent for generating primes of up to a few million. Beyond that, memory limits may make it impractical. Sieve of Eratosthenes can be made efficient with optimizations like segmentation and parallelism.
- The following code has been made efficient. Compare it to the first one.

```rust
use num_bigint::BigUint;
use num_traits::{One, Zero};
use num_integer::Integer;
use rayon::prelude::*;
use std::str::FromStr;
use std::time::Instant;

// Function to determine if a number is prime.
fn is_prime(n: &BigUint) -> bool {
        if n <= &BigUint::one() {
                return false;
        }
        let two = BigUint::from(2u32);
        if n == &two {
                return true;
        }
        if n.is_even() {
                return false;
        }
```

```
        let mut i = BigUint::from(3u32);
        let limit = sqrt(n) + &BigUint::one();
        while &i < &limit {
                if n % &i == BigUint::zero() {
                        return false;
                }
                i += &two;
        }
        true
}

// Function for computing the integer square root using the Newton method.
fn sqrt(n: &BigUint) -> BigUint {
        let mut x0: BigUint = n.clone();
        let mut x1: BigUint = (n >> 1) + BigUint::one();
        while x1 < x0 {
                x0 = x1.clone();
                x1 = (n / &x1 + &x1) >> 1;
        }
        x0
}

fn main() {
        let lower_str = "20000000000";
        let upper_str ="20000009000";

        let lower = BigUint::from_str(lower_str).unwrap();
        let upper = BigUint::from_str(upper_str).unwrap();
```

A

```rust

        println!("Finding␣primes␣between␣{}␣and␣{}", lower_str, upper_str);
        let start = Instant::now();

        // Convert the BigUint range to a Vec<BigUint> for parallel processing
        let mut numbers = Vec::new();
        let mut num = lower.clone();
        while num <= upper {
                numbers.push(num.clone());
                num += BigUint::one();
        }

        // Find primes in parallel
        let primes: Vec<BigUint> = numbers
        .into_par_iter()
        .filter(|num| is_prime(num))
        .collect();

        for prime in primes {
                println!("{}", prime);
        }

        let duration = start.elapsed();
        println!("Time␣taken:␣{:?}", duration);
}
```

- For large-scale primality testing, use probabilistic tests such as the Miller-Rabin test or advanced deterministic tests with very big integers.
- For practical applications involving huge numbers (e.g., cryptography), probabilistic testing are often combined with known prime-generating algorithms.
- Such tasks can be efficiently implemented using tools like GMP and Rust libraries like Primal.
- Note that the set of primes is infinite.

Theorem 3 (Fundamental theorem of arithmetic)

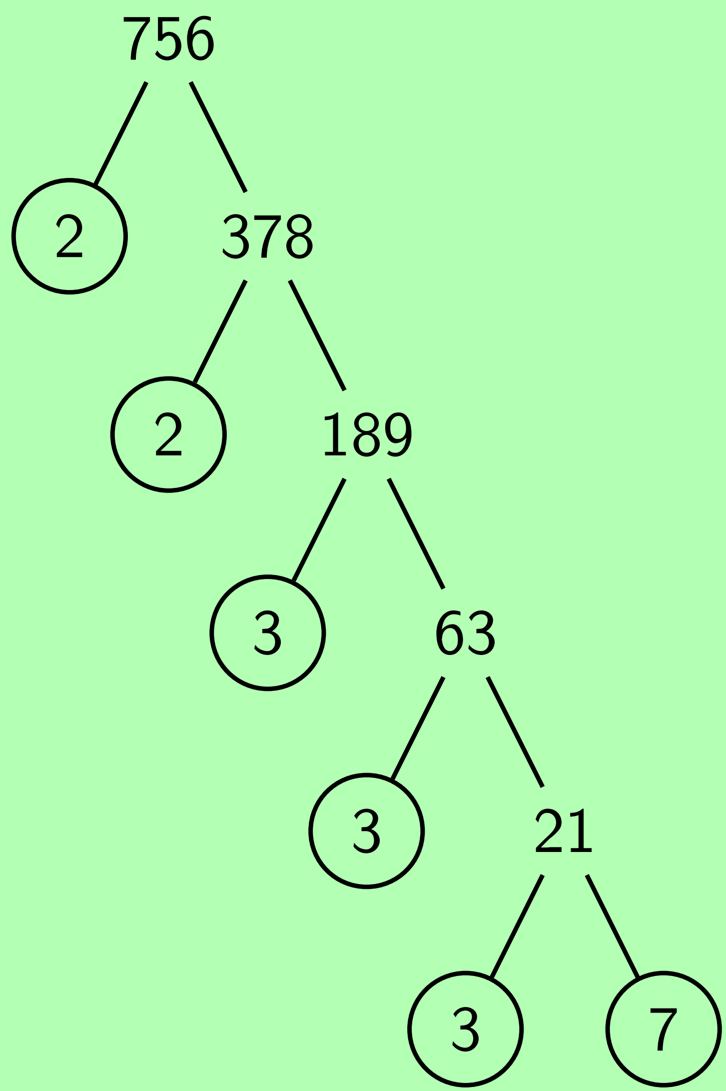
Every integer greater than 1 is either a prime number or it can be factored uniquely into a product of primes.
That is, $n = p_1^{n_1} p_2^{n_2} \cdots p_r^{n_r}$,
where $p_1, \cdots, p_r$ are distinct primes and $n_1, \cdots, n_r$ are positive integers.

This theorem is also called the unique factorization theorem or unique prime factorization theorem

Let us now factorize composites.

Example 2

Find the prime factorization for 3780



Thus, $756 = 2^2 \times 3^3 \times 7$

Example 3

Find the prime factorization for 12600

$$
\begin{array}{r|l}
2 & 12600 \\ \hline
2 & 6300 \\ \hline
2 & 3150 \\ \hline
3 & 1575 \\ \hline
3 & 525 \\ \hline
5 & 175 \\ \hline
5 & 35 \\ \hline
7 & 7 \\ \hline
 & 1
\end{array}
$$

Thus $12600 = 2^3 \times 3^2 \times 5^2 \times 7$

We can check ours findings using Rust code,

```
fn main() {
        let number = 12600; // The number that requires factorization
        let factors = prime_factors(number);
```

```

        println!("Prime␣factors␣of␣{}␣are:␣{:?}", number, factors);
}

fn prime_factors(mut n: u64) -> Vec<u64> {
        let mut factors = Vec::new();

        // Checks for number of 2s that divide n
        while n % 2 == 0 {
                factors.push(2);
                n /= 2;
        }

        // At this stage, n must be odd so we skip even numbers.
        let mut i = 3;
        while i * i <= n {
                // While i divides n, add i and divide n
                while n % i == 0 {
                        factors.push(i);
                        n /= i;
                }
                i += 2;
        }
```

```
            // This condition handles scenarios where n is a prime number
            // bigger than 2
            if n > 2 {
                factors.push(n);
            }

            factors
        }
```

which on compiling prints Prime factors of 12600 are: $[2, 2, 2, 3, 3, 5, 5, 7]$

Note that any algorithm finding prime factorization of integers also answers a simpler question of whether a given integer is prime or composite?

If the number we are factoring is large we could use algorithms like Pollard's ρ Algorithm, Pollard's $P - 1$ algorithm or the Elliptic curve factorization Algorithm. The following is the Rust code for Pollard's ρ algorithm and we use it to factorize