

WEB3CLUBS FOUNDATION LIMITED

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

Foundational Mathematics for Web3 Builders

Implemented in RUST

Lecture 29

July 1, 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
 $\mathbb{Z} = \{\dots, -4, -3, -2, -1, 0, 1, 2, 3, \dots\}$ 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 $a = 59416033658004120313555424513491018615317021176268174958724971404913$ by $b = 27345537128851$ we could use the Rust code below.

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

```
1 use num_bigint::BigUint; // handles arbitrarily large unsigned integers
2 use num_integer::Integer; // offers methods for integer operations,
3 // such as division with remainder.
4
5 fn main() {
6     // Large number as a string
7     let large_number
8 = "21888242871839275222246405745257275088696311157297823662689037894645226208587";
9     // Divisor as a string
10    let divisor = "217278722208812865687284837625192001256856159830432362";
11
12    // Parse the numbers as BigUint. //Unwrap() handles any potential mistakes
13    // during parsing by halting the application if parsing fails.
14
15    let num = BigUint::parse_bytes(large_number.as_bytes(), 10).unwrap();
16    let div = BigUint::parse_bytes(divisor.as_bytes(), 10).unwrap();
17
18    // Perform division and note the remainder.
19    // div_rem is used to divide num by div, and returns a tuple
20    // containing the quotient and the remainder.
```

```

21     let (quotient, remainder) = num.div_rem(&div);
22
23     // Print the result
24     println!("Quotient: ␣{}", quotient);
25     println!("Remainder: ␣{}", remainder);
26 }

```

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, \dots .

- 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 fn main() {
2     let start = 1; // Starting point
3     let end = 144; // Ending point
4
5     //Now check whether the starting point is greater than the finishing point.
6     //If this is the case, the function terminates with an error message.

```

```

7     //then calls sieve_of_eratosthenes_in_range to compute the prime
8     //numbers in the specified range
9     if start > end {
10    println!("Invalid range: start({}) is greater than end({})", start, end);
11        return;
12    }
13
14    let primes = sieve_of_eratosthenes_in_range(start, end);
15
16    println!("Prime numbers between {} and {} are: {:?}", start, end, primes);
17 }
18
19 /// Use the Eratosthenes Sieve to generate prime numbers within a specific range.
20 ///If the end is less than two, this technique produces an empty vector since there
21 ///are no prime numbers fewer than two.
22 fn sieve_of_eratosthenes_in_range(start: usize, end: usize) -> Vec<usize> {
23     if end < 2 {
24         return vec![];
25     }
26     //At initialization, A vector vector is constructed and used to
27     //indicate if an integer is prime (true) or not prime.
28     //Goes from two to the square root of the end. For each number num,
29     // if is_prime[num] is true,it marks all multiples of num as false (non-prime).

```



```

30 let mut is_prime = vec![true; end + 1];
31     is_prime[0] = false;
32     is_prime[1] = false;
33
34     for num in 2..=((end as f64).sqrt() as usize) {
35         if is_prime[num] {
36             for multiple in (num * num..=end).step_by(num) {
37                 is_prime[multiple] = false;
38             }
39         }
40     }
41     //Collects and returns the prime numbers in the specified range
42     is_prime.iter()
43     .enumerate()
44     .filter(|&(num, &prime)| prime && num >= start)
45     .map(|(num, _)| num)
46     .collect()
47 }

```

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.

```
1 use num_bigint::BigUint;
2 use num_traits::{One, Zero};
3 use num_integer::Integer;
4 use rayon::prelude::*;
5 use std::str::FromStr;
6 use std::time::Instant;
7
8 // Function to determine if a number is prime.
9 fn is_prime(n: &BigUint) -> bool {
10     if n <= &BigUint::one() {
11         return false;
12     }
13     let two = BigUint::from(2u32);
14     if n == &two {
15         return true;
16     }
17     if n.is_even() {
18         return false;
19     }
```

```

20
21     let mut i = BigUint::from(3u32);
22     let limit = sqrt(n) + &BigUint::one();
23     while &i < &limit {
24         if n % &i == BigUint::zero() {
25             return false;
26         }
27         i += &two;
28     }
29     true
30 }
31
32 // Function for computing the integer square root using the Newton method.
33 fn sqrt(n: &BigUint) -> BigUint {
34     let mut x0: BigUint = n.clone();
35
36     let mut x1: BigUint = (n >> 1) + BigUint::one();
37     while x1 < x0 {
38         x0 = x1.clone();
39         x1 = (n / &x1 + &x1) >> 1;
40     }
41     x0
42 }
43 fn main() {
44     let lower_str = "20000000000";
45     let upper_str = "20000009000";
46
47     let lower = BigUint::from_str(lower_str).unwrap();
48     let upper = BigUint::from_str(upper_str).unwrap();

```

```

49
50     println!("Finding primes between {} and {}", lower_str, upper_str);
51     let start = Instant::now();
52
53     // Convert the BigUint range to a Vec<BigUint> for parallel processing
54     let mut numbers = Vec::new();
55     let mut num = lower.clone();
56     while num <= upper {
57         numbers.push(num.clone());
58         num += BigUint::one();
59     }
60
61     // Find primes in parallel
62     let primes: Vec<BigUint> = numbers
63         .into_par_iter()
64         .filter(|num| is_prime(num))
65         .collect();
66
67     for prime in primes {
68         println!("{}", prime);
69     }
70
71     let duration = start.elapsed();
72     println!("Time taken: {:?}", duration);
73 }

```

- 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.

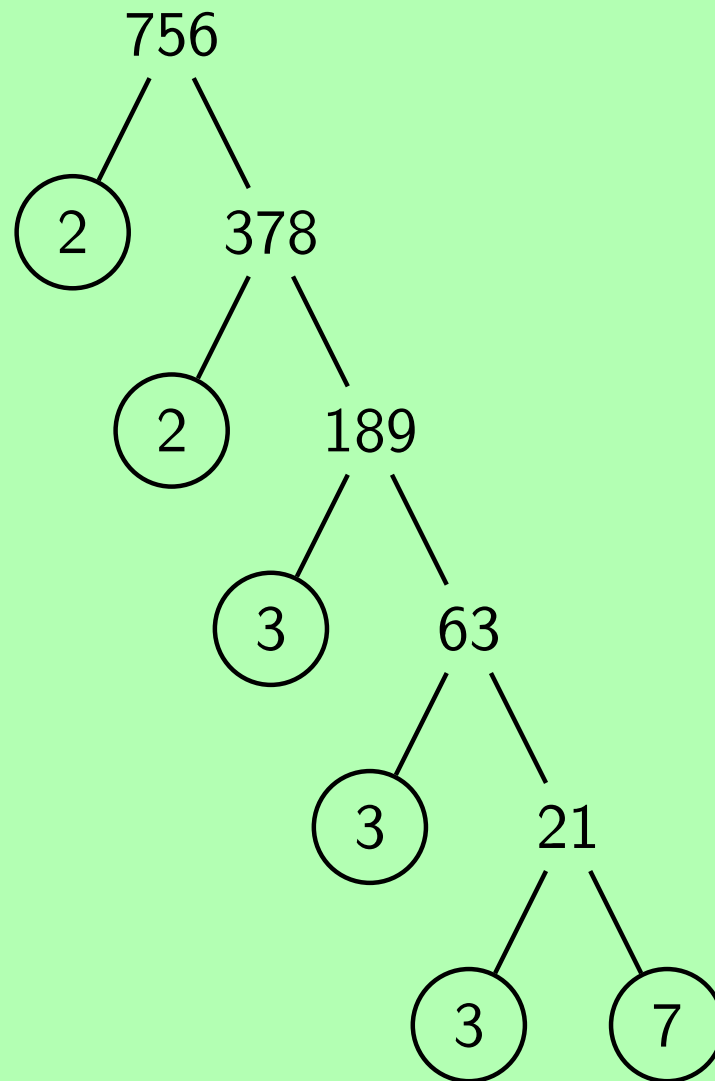
*That is, $n = p_1^{n_1} p_2^{n_2} \cdots p_r^{n_r}$,
where p_1, \cdots, p_r are distinct primes and n_1, \cdots, n_r are positive integers.*

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

2	12600
2	6300
2	3150
3	1575
3	525
5	175
5	35
7	7
	1

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

We can check our findings using Rust code,

```

1 fn main() {
2     let number = 31500; // The number to be factored
3     let factors = prime_factors(number);
4
5     println!("Prime factors of {} are: {:?}", number, factors);
6 }
7
8 fn prime_factors(mut n: u64) -> Vec<u64> {
9     let mut factors = Vec::new();
10
11     //A while loop tests if n is divisible by two.
12     //If n is divisible by two, it adds two to the factor vector and divides
13     //it by two.
14     //This loop repeats until n is no longer divisible by two.
15     while n % 2 == 0 {
16         factors.push(2);
17         n /= 2;
18     }
19
20     // n must be odd at this point, so we can skip even numbers
21     //The code then proceeds to check odd factors beginning with 3.
22     //A while loop iterates with i beginning at 3 and increasing by
23     //2 with each iteration (skipping even values).
24     //A while loop determines if n is divisible by i for each i.
25     //If so, n is divided by i and i is added to the factors vector.
26     //This keeps happening until n can no longer be divided by i.

```

```

27     let mut i = 3;
28     while i * i <= n {
29         while n % i == 0 {
30             factors.push(i);
31             n /= i;
32         }
33         i += 2;
34     }
35
36     // In the event that n remains more than 2 after all, it indicates
37     //that n is a prime number that is added to the factors vector.
38     if n > 2 {
39         factors.push(n);
40     }
41
42     factors
43 }

```

which on compiling prints Prime factors of 31,500 are: [2, 2, 3, 3, 5, 5, 5, 7]

Understanding the Rust code

Let us break down each part of the code.



```
fn main() {  
    let number = 31500; // The number to be factored  
    let factors = prime_factors(number);  
  
    println!("Prime factors of {} are: {:?}", number, factors);  
}
```

- ✓ In Rust, the fn main() {...} block specifies the main function, which serves as the program's entry point.
- ✓ let number = 31500;: This declares a variable named number and assigns it the value 31500.
- ✓ let: A keyword for declaring a variable.

Understanding the Rust code

- ✓ Number: Variable name.
- ✓ let factors = prime_factors(number);: This invokes the prime_factors function with number as an argument and assigns the result to the variable factors.
- ✓ Let factors: Defines a new variable named factors.
- ✓ prime_factors(number): Calls the prime_factors function with a number as the argument.
- ✓ =: Assigns the result of the function call to factors.
- ✓ factors: The variable that hold carry the vector of the number's prime factors.

Understanding the Rust code

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

- ✓ println!: A macro that prints text to the console.
- ✓ "Prime factors of {} are: {:?}", number, factors);": The format string containing placeholders {} and {:?}", number, factors);".
- ✓ number: is the first argument, replacing the placeholder {}.
- ✓ factors: is the second argument which substitutes the placeholder {:?}", number, factors); with a vector of prime factors.

Understanding the Rust code

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

This function is part of the function definition for prime_factors

- ✓ fn Prime Factors: This declares a function called prime_factors.
- ✓ (mut n = u64): indicates that the function accepts a single parameter n of type u64, and mut indicates that this parameter is mutable, which means that it can be changed inside the function.
- ✓ - > Vec < u64 >: The function returns a value of type Vec < u64 >, a dynamic array of 64-bit unsigned integers.

Definition 4

An unsigned integer is a form of integer in computer science and programming that can only represent non-negative whole values. Unsigned integers, as opposed to signed integers which represent both positive and negative values, can only express positive and zero values.

u8 i32

The range of an unsigned integer depends on its size (number of bits)

For instance, $\underline{2^8 - 1} = \underline{255}$ $\underline{2^{16} - 1}$

8-bit unsigned integer (u8): Can represent values from 0 to 255.

16-bit unsigned integer (u16): Can represent values from 0 to 65,535.

32-bit unsigned integer (u32): Can represent values from 0 to 4,294,967,295.

64-bit unsigned integer (u64): Can represent values from 0 to 18,446,744,073,709,551,615.

Signed integers can express positive and negative values. Negative numbers use half of the range, whereas positive numbers use the other half (minus one for zero).

For instance, A 32-bit signed integer (i32) can have values ranging from -2,147,483,648 to 2,147,483,647.

In Rust, unsigned numbers are denoted by the u prefix followed by the bit size.

Unsigned integers cannot express negative values, making them unsuitable for applications requiring negative values.

Understanding the Rust code

```
let mut factors = Vec::new();
```

- ✓ let mut factors: Defines a mutable variable called factors.
- ✓ Let is a keyword used to declare variables.
- ✓ Mut: Indicates that the variable factors are mutable and can be changed after initial declaration.
- ✓ Factors: The variable's name.
- ✓ = Vec::new();: Initializes factors as a new, empty vector of type u64.
- ✓ Vec::new() invokes the new associated function on the Vec type, resulting in an empty vector.

Understanding the Rust code

```
• while n % 2 == 0 {  
    factors.push(2);  
    n /= 2;  
}
```

if
n /= 2
/=

- ✓ The while loop determines whether n is divisible by 2 ($n \% 2 == 0$). It enters the loop if it is true.
 - ✓ factors.push(2); - This means “add number 2 to the factors vector”.
- In Rust, an element is added to the end of a vector using the push method. Since 2 is a prime factor of the number n , it is appended to the factors vector in this case.
- ✓ $n /= 2$; - This line updates n with the result of dividing the value n by 2.



In Rust, the operator $/=$ stands for division followed by assignment. Hence, $n /= 2$ is the same as $n = n / 2$.

Understanding the Rust code

Thus, `factors.push(2);` and `n /= 2;` has a combined effect of continuously dividing `n` by 2 until `n` is no longer divisible by 2 and adding 2 to the vectors each time.

Take `n = 31500` for instance,

First Iteration:

 `'n % 2 == 0'` (true) 
`'factors.push(2)'` adds 2 to factors (now factors = [2]) ✓
`n /= 2` updates `n` to 15750 *`n = 15750`*

Second Iteration:

`'n % 2 == 0'` (true) ✓
`'factors.push(2)'` adds 2 to factors (now factors = [2,2]) ✓
`n /= 2` updates `n` to 7875 *`n = 7875`* ↑

Third Iteration:

`'n % 2 == 0'` (false)

Since `n` is no longer divisible by 2, the loop exits.

Understanding the Rust code

After this loop is finished, factors will contain [2, 2], and n will be 7875, which means that all of the 2 factors have been removed from n. Next, the algorithm looks for further prime factors.

```
let mut i = 3;
while i * i <= n {
    while n % i == 0 {
        factors.push(i);
        n /= i;
    }
    i += 2;
}
```

Handwritten annotations: A blue circle around the `<=` operator in the first while loop. A blue arrow points from the `i` in `i += 2;` to the `i` in the inner while loop. A blue circle around the `+=` operator in `i += 2;`. A blue checkmark next to `i += 2;`. The text `+= 2` is written below `i += 2;`.

- ✓ `let mut i = 3;` This declares a mutable variable `i` and initializes it to 3. 'let': Introduces a new variable. mut: Indicates that `i` is mutable, meaning its value can change. `i = 3`: Initializes `i` to 3, starting the search for odd prime factors from 3.

Understanding the Rust code

- ✓ while $i * i \leq n$ - This outer loop runs as long as $i * i$ is less than or equal to n .
- ✓ The loop evaluates all possible factors up to the square root of n . If a number n has a factor greater than its square root, the associated co-factor will be less than the square root, ensuring that all factors are considered. This is because we only need to check up to the square root of n since if n is divisible by a number higher than its square root, the quotient must be less than the square root and has already been checked.
- ✓ Inside loop, while $n \% i == 0$, this loop runs as long as n is divisible by i (i.e., $n \% i == 0$)
- ✓ `factors.push(i);`: Adds i to the factors vector.
- ✓ `n /= i;`: Divides n by i to remove one instance of the factor i from n .

Understanding the Rust code

- ✓ $i += 2$; To proceed to the next odd number, increment i by 2. That is, because integers bigger than two are not prime, we skip them. Starting at 3 and increasing by 2 ensures that only odd numbers (potentially odd prime factors) are examined.

Consider our example above. We were left with $n=7875$ and the factors were $[2,2]$ ✓

Initialization: $i=3$

First Iteration of Outer Loop ($i = 3$).

- Condition: $3 * 3 \leq 7875$ (true) ✓
- Inner Loop:

- $7875 \% 3 == 0$ (true) ✓

✓ $\text{factors.push}(3) \rightarrow \text{factors} = [2,2,3]$ ✓

✓ $7875 / 3$ \rightarrow $n = 2625$

Understanding the Rust code

$$3 \times 3 \leq 875 \quad (1-2)$$

- $2625 \% 3 == 0$ (true)
- ✓ factors.push(3) \rightarrow factors = [2,2,3,3]
- ✓ 2625 /= 3 \rightarrow n = 875 ✓
- 875 % 3 == 0 (false) ✓
- Increment: i += 2 \rightarrow i = 5

Second Iteration of Outer Loop (i = 5):

- Condition: 5 * 5 \leq 875 (true)
- Inner Loop:

- 875 % 5 == 0 (true)
- ✓ factors.push(5) \rightarrow factors = [2,2,3, 3, 5] ✓
- ✓ 875 /= 5 \rightarrow n = 175 ✓

$$n /= i$$

Understanding the Rust code

35 <= 35

- $175 \% 5 == 0$ (true)

✓ factors.push(5) - > factors = [2,2,3, 3, 5,5] ✓

n /= 5

✓ 175 / = 5 - > n = 35

- 35 % 5 == 0 (true)

✓ factors.push(5) - > factors = [2,2,3, 3, 5,5,5] ✓

✓ 35 / = 5 - > n = 7

- 7 % 5 == 0 (false)

600

- Increment: i += 2 - > i = 7

- Condition: 7 * 7 <= 7 (false): Exit Loop:

27 27

At this point, the outer loop is terminated, and the function properly recognized the prime factors as [2,2,3, 3, 5,5,5] and n=7.

Understanding the Rust code

```
if n > 2 {  
    factors.push(n);  
}
```

This line is used to handle the situation in which the residual value of n is a prime number higher than two after all smaller prime factors have been handled. This section of the function ensures that such a prime number appears in the list of factors.

- Condition: $n > 2$ Checks whether the remaining value of n is larger than 2.
- `factors.push(n);` Adds the remaining prime number, n , to the factors vector.

Consider our example above. Our remaining $n = 7$ which cannot be divided by a number less than its square root except for one.

Understanding the Rust code

After the loop:

- n is now 7, which is greater than 2.
- The condition if $n > 2$ is true.
- 7 is added to the factors vector.

The resulting factors vector will be [2,2,3, 3, 5,5,5,7]

Exercise 1

Write a Rust code to add any two unsigned 32-bit integers