WEB3CLUBS FOUNDATION LIMITED

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

Foundational Mathematics for Web3 Builders

Implemented in RUST

Lecture 34

July 18, 2024

Fast powering algorithm Continuation

- The fast powering algorithm, also known as exponentiation by squaring algorithm, is a technique used to efficiently compute the power of a number, especially in modular arithmetic.
- Some texts call it Square-and-Multiply Algorithm.
- This algorithm greatly reduces the number of multiplications needed compared to the straightforward method of multiplying the base by itself repeatedly.
- In some cryptosystems that we will study, for example the RSA we will be required to compute large powers of a number b modulo another number m and so the fast powering algorithm combined with Fermat's Little Theorem or Euler's Theorem will be very vital

Example 32

Compute $(20782727282728287373833^{626777655777666776} + 5546474747476647447647^{991111111111111113334}) \bmod 17892892892892829282$

```
fn mod_exp(base: u128, exp: u128, modulus: u128) -> u128 {
        let mut result = 1;
        let mut base = base % modulus;
        let mut exp = exp;

        while exp > 0 {
                if exp % 2 == 1 {
                        result = (result * base) % modulus;
                }
                exp = exp >> 1;
                base = (base * base) % modulus;
        }

        result
}

fn main() {
        let base1: u128 = 20782727282728287373833;
        let exp1: u128 = 626777655777666776;
        let base2: u128 = 5546474747476647447647;
        let exp2: u128 = 991111111111111113334;
        let modulus: u128 = 17892892892892829282;
```

$a^{m} + b^{n}$

Bigunit

```
        let result1 = mod_exp(base1, exp1, modulus);
        let result2 = mod_exp(base2, exp2, modulus);

        let final_result = (result1 + result2) % modulus;

println!("The␣result␣of␣(20782727282728287373833^626777655777666776␣+␣554647474747664744
}
```

Running this code prints the result as 14411741738844151902.

If you need to work with even larger numbers, you should use the num-bigint crate. The Rust code is below.

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

fn mod_exp(base: &BigUint, exp: &BigUint, modulus: &BigUint) -> BigUint {
        let mut result = BigUint::one();
        let mut base = base % modulus;
        let mut exp = exp.clone();

        while exp > BigUint::zero() {
                if &exp % 2u32 == BigUint::one() {
                        result = (result * &base) % modulus;
                }
                exp >>= 1;
                base = (&base * &base) % modulus;
        }
```

```rust

        result
}

fn main() {
        let base1 = BigUint::parse_bytes(b"207827272827282873733833", 10).unwrap();
        let exp1 = BigUint::parse_bytes(b"626777655777666776", 10).unwrap();
        let base2 = BigUint::parse_bytes(b"554647474747664744764 7", 10).unwrap();
        let exp2 = BigUint::parse_bytes(b"991111111111111113334", 10).unwrap();
        let modulus = BigUint::parse_bytes(b"178928928928928292 82", 10).unwrap();

        let result1 = mod_exp(&base1, &exp1, &modulus);
        let result2 = mod_exp(&base2, &exp2, &modulus);

        let final_result = (result1 + result2) % modulus;

        println!("The result of (207827272827282873733833^626777655777666776 + 5546474747
}
```

Exercise 3

1. Use Rust program to compute the following quantities:

   (i) $2^{1000} \pmod{2379}$ (ii) $567^{1234} \pmod{4321}$

   (iii) $47^{258008} \pmod{1315171}$

2. Compute $7^{7386} \pmod{7387}$ by the method of successive squaring. Is 7387 prime?
3. Compute $7^{7392} \pmod{7393}$ by the method of successive squaring. Is 7393 prime?
4. Compute $2^{9990} (\bmod\ 9991)$ by successive squaring and use your answer to say whether you believe that 9991 is prime.

Linear Congruence

- Consider the linear congruence
$$ax \equiv b \pmod m \qquad (1)$$
where a, b, m are integers with $m > 0$.
- By a solution of equation (1) we mean an integer $x = x_1$ for which $m \mid (ax_1 - b)$.
- Note that if x_1 is a solution of equation (1) then $x_1 + km$ for $k \in \mathbb{Z}$ is another solution of equation (1).
- **Note:** An equation $ax \equiv b(\mod m)$ has a solution if $\gcd(a, m)$ divides b.
- In this case, if $d = \gcd(a, m)$ and $d \mid b$ then the congruence equation has d solutions.
- This congruence equation has no solution if $d \nmid b$.

Example 33 $\gcd(5,8) = 1$

Solve the following linear congruence equations.

$0, 1, 2, \ldots, 7$ ✓

a) $5x \equiv 3 \pmod 8$
b) $6x \equiv 4 \pmod 9$
c) $6x \equiv 8 \pmod{10}$
d) $3x + 2 \equiv 8 \pmod{10}$
e) $6x - 3 \equiv 5 + 2x \pmod{10}$
f) $\frac{2}{3}x \equiv 4 \pmod 7$

Solution

Since the moduli is relatively small, we will find solutions by testing. Later on we will see how to use extended Euclid's Algorithm to find solutions to such congruence equations.

Solution (conti...)

a) Here $\gcd(5, 8) = 1$ and 1 divides 3 hence the equation has a unique solution. Let us test $0, 1, 2, 3, \cdots 7$ to find the solution.

$5x \equiv 3 \bmod 8$

$5(0) = 0 \not\equiv 3 \pmod 8$	$5(4) = 4 \not\equiv 3 \pmod 8$
$5(1) = 5 \not\equiv 3 \pmod 8$	$5(5) = 1 \not\equiv 3 \pmod 8$
$5(2) = 2 \not\equiv 3 \pmod 8$	$5(6) = 6 \not\equiv 3 \pmod 8$
$5(3) = 7 \not\equiv 3 \pmod 8$	$5(7) = 3 \equiv 3 \pmod 8$

Thus the unique solution is $x = 7$

b) The $\gcd(6, 9) = 3$ but $3 \nmid 4$ and so the congruence equation has no solution.

$6x \equiv 4 \bmod 9$

Solution (conti...)

$6x \equiv 8 \mod 10$

c) Here $\gcd(6, 10) = 2$ and $2 \mid 8$ hence the equation has two solutions. Let us test $0, 1, 2, 3, \cdots 9$ to find the solutions.

$$6(0) = 0 \not\equiv 8(\mathrm{mod}\ 10) \qquad 6(5) = 0 \not\equiv 8(\mathrm{mod}\ 10)$$

$$6(1) = 6 \not\equiv 8(\mathrm{mod}\ 10) \qquad 6(6) = 6 \not\equiv 8(\mathrm{mod}\ 10)$$

$$6(2) = 2 \not\equiv 8(\mathrm{mod}\ 10) \qquad 6(7) = 2 \not\equiv 8(\mathrm{mod}\ 10)$$

$$6(3) = 8 \equiv 8(\mathrm{mod}\ 10) \qquad 6(8) = 8 \equiv 8(\mathrm{mod}\ 10)$$

$$6(4) = 4 \not\equiv 8(\mathrm{mod}\ 10) \qquad 6(9) = 4 \not\equiv 8(\mathrm{mod}\ 10)$$

Thus the two solutions are $x_1 = 3$ and $x_2 = 8$

We work with congruence relations modulo m much as with ordinary equalities. That is, we add to, subtract from, or multiply both sides of a congruence modulo m by the same integer; also, if b is congruent to a modulo m we may

Solution (conti...)

substitute b for a in any simple arithmetic expression (involving addition, subtraction, and multiplication) appearing in a congruence modulo m.

d) First we subtract 2 on both sides to get $3x \equiv 6 (\mathrm{mod}\ 10)$. Here, $\gcd(3, 10) = 1$ and 1 divides 6 and so the equation has one solution. Now test $0, 1, 2, \cdots 9$ to find the solution.

$$3(0) = 0 \not\equiv 6 (\mathrm{mod}\ 10) \qquad 3(5) = 5 \not\equiv 6 (\mathrm{mod}\ 10)$$

$$3(1) = 3 \not\equiv 6 (\mathrm{mod}\ 10) \qquad 3(6) = 8 \not\equiv 6 (\mathrm{mod}\ 10)$$

$$3(2) = 6 \equiv 6 (\mathrm{mod}\ 10) \qquad 3(7) = 1 \not\equiv 6 (\mathrm{mod}\ 10)$$

$$3(3) = 9 \not\equiv 6 (\mathrm{mod}\ 10) \qquad 3(8) = 4 \not\equiv 6 (\mathrm{mod}\ 10)$$

$$3(4) = 2 \not\equiv 6 (\mathrm{mod}\ 10) \qquad 3(9) = 7 \not\equiv 6 (\mathrm{mod}\ 10)$$

Thus the solution is $x = 2$

Solution (conti...)

e) Collect like terms. $6x - 2x \equiv 5 + 3 \pmod{10}$ which becomes $4x \equiv 8 \pmod{10}$. The $\gcd(4, 10) = 2$ and $2 \mid 8$ hence equation has 2 solutions. Now test $0, 1, 2, \cdots 9$ to find the solution.

$$4(0) = 0 \not\equiv 8 \pmod{10} \qquad 4(5) = 0 \not\equiv 8 \pmod{10}$$
$$4(1) = 4 \not\equiv 8 \pmod{10} \qquad 4(6) = 4 \not\equiv 8 \pmod{10}$$
$$4(2) = 8 \equiv 8 \pmod{10} \qquad 4(7) = 8 \equiv 8 \pmod{10}$$
$$4(3) = 2 \not\equiv 8 \pmod{10} \qquad 4(8) = 2 \not\equiv 8 \pmod{10}$$
$$4(4) = 6 \not\equiv 8 \pmod{10} \qquad 4(9) = 6 \not\equiv 8 \pmod{10}$$

The solutions are $x_1 = 2$ and $x_2 = 7$

Solution (conti...)

f) To work with fractions a/d modulo m the denominator must be relatively prime to m.
Simplify $\frac{2}{3}x \equiv 4 \pmod 7$ to get $2x \equiv 12 \pmod 7$ or $2x \equiv 5 \pmod 7$. Now, $\gcd(2,7) = 1$ and so there is one solution. Test $0, 1, 2, \cdots 6$ to find the solution.

$$2(0) = 0 \not\equiv 5 \pmod 7$$
$$2(1) = 2 \not\equiv 5 \pmod 7$$
$$2(2) = 4 \not\equiv 5 \pmod 7$$
$$2(3) = 6 \not\equiv 5 \pmod 7$$
$$2(4) = 1 \not\equiv 5 \pmod 7$$
$$2(5) = 3 \not\equiv 5 \pmod 7$$
$$2(6) = 5 \equiv 5 \pmod 7$$

Thus the solution is $x = 6$.

Solution of Linear Congruences Using Euclid's Algorithm

Example 34

Find the least positive integer x for which

$$53 \equiv \overset{b}{1} \mod 93$$

gcd | b

Solution

First, $\gcd(93, 53) = 1$ and 1 divides 1 and the equation has 1 solution.

By Euclid's algorithm algorithm we have

$$93 = 53(1) + 40$$

$$53 = 40(1) + 13$$

$$40 = 13(3) + 1$$

$$13 = 1(13) + 0$$

Solution (conti...)

Now solve for the $\gcd$.

$$
\begin{aligned}
1 &= 40 - 13(3) = 40 - [53 - 40(1)](3) = 40 - 53(3) + 40(3)\\
&= -53(3) + 40(4) = -53(3) + [93 - 53(1)](4)\\
&= -53(3) + 93(4) - 53(4)\\
&= 93(4) + 53(-7)
\end{aligned}
$$

$x_1 = x$
$x_2 = x_1 + m$
$x_3 = x_2 + m$
$x_4 = x_3 + m$

Thus $1 = 93(4) + 53(-7)$ and therefore modulo 93 gives
$53(-7) \equiv 1 \mod 93$.

$(x + m) \% \bmod m$

Thus $x = -7$ is a solution. We could also give this answer as $x = 86$ since 86 is the least positive number congruent to $-7 \mod 93$. So, $x = 86$ is the required answer.

$x + m$

The extended Euclidean technique is used in Rust code to determine all solutions to a linear congruence. The Rust code checks for solutions and computes them using properties of modular arithmetic and the GCD. This code generates and returns every possible solution.

≡ \u(2261)

$53x \equiv 1 \bmod 93$

```
use std::vec::Vec;

fn main() {
        let a = 53;
        let b = 1;
        let n = 93;

        match solve_linear_congruence(a, b, n) {
                Some(solutions) => {
println!("Solutions␣to␣{}x␣\u{2261}␣␣{}␣(mod␣{}):␣{:?}", a, b, n, solutions);
                }
                None => {
println!("The␣congruence␣{}x␣\u{2261}␣␣{}␣(mod␣{})␣has␣no␣solutions", a, b, n);
                }
        }
}

fn gcd_extended(a: i64, b: i64) -> (i64, i64, i64) {
        if a == 0 {
                return (b, 0, 1);
        }
        let (g, x1, y1) = gcd_extended(b % a, a);
        let x = y1 - (b / a) * x1;
        let y = x1;
        (g, x, y)
}
```

a b n

a, b

$d = ax + by$

$d = ax + by$

$\gcd(0, b) = b$

```rust
fn mod_inverse(a: i64, n: i64) -> Option<i64> {
        let (g, x, _) = gcd_extended(a, n);
        if g != 1 {
                None
        } else {
                Some((x % n + n) % n)
        }
}

fn solve_linear_congruence(a: i64, b: i64, n: i64) -> Option<Vec<i64>> {
        let (g, _x, _) = gcd_extended(a, n); // Prefix '_x' to suppress warning

        if b % g != 0 {
                return None; // No solutions
        }

        let a_prime = a / g;
        let b_prime = b / g;
        let n_prime = n / g;

        let x0 = (b_prime * mod_inverse(a_prime, n_prime)?) % n_prime;
        let mut solutions = Vec::new();

        for i in 0..g {
                solutions.push((x0 + i * n_prime) % n);
        }

        Some(solutions)
}
```

When the code is compiled, it out puts "Solutions to $53x \equiv 1 \pmod{93}$: [86]"

Understanding the Rust code

- The main function initializes the values for a, b, and n, and then calls solve_linear_congruence to find the solutions.
- The gcd_extended function computes the greatest common divisor (GCD) of two numbers and also finds the coefficients (x and y) such that $ax + by = \gcd(a, b)$. This is done using the extended Euclidean algorithm.
- The mod_inverse function uses the result of the gcd_extended function to find the modular inverse of a modulo n, if it exists. The modular inverse exists if and only if the GCD of a and n is 1.

Understanding the Rust code (conti...)

- The solve_linear_congruence function solves the congruence $ax \equiv b \mod n$. It first finds the GCD of a and n and checks if b is divisible by this GCD. If not, there are no solutions. If b is divisible, it reduces the problem to a simpler form and finds the modular inverse of the reduced coefficient. It then calculates all solutions based on this inverse.

Example 35

Find integer x for which $7x \equiv 13 \bmod 19$

Solution

$\gcd(19,7) = 1$ and so equation has 1 solution.

By Euclid's algorithm algorithm we have

$$19 = 7(2) + 5$$

$$7 = 5(1) + 2$$

$$5 = 2(2) + 1$$

$$2 = 1(2) + 0$$

We solve for $\gcd$

$$1 = 5 - 2(2) = 5 - [7 - 5(1)](2) = 5 - 7(2) + 5(2) = -7(2) + 5(3)$$

$$= -7(2) + [19 - 7(2)](3) = -7(2) + 19(3) - 7(6)$$

$$= 19(3) + 7(-8)$$

That is, $1 = 19(3) + 7(-8)$.

Solution (conti...)

Since we require $13 = 19(n) + 7(x)$ for some $n \in \mathbb{Z}$, we multiply $1 = 19(3) + 7(-8)$ by 13 to get

$13 = 19(3 \times 13) + 7(-8 \times 13)$ which we compute $\bmod 19$ to get

$13 = 19(1) + 7(10)$

Thus, $x = 10$

Example 36

Solve $4043n \equiv 27 (\bmod 166361)$

Solution

Here, $\gcd(166361, 4043) = 13$ but $13 \nmid 27$. Hence the congruence has no solution.

Example 37

Find

$$507444444444339383938392343327777777744444 2x \equiv 1838383833838383837899999999999999999998 \pmod{71338888888888828282828287777777777777774}$$

For larger integers, we use a big integer library such as num-bigint in Rust. The BigInt type is used to handle large integers, and the num-bigint crate provides the necessary functionality for working with these big integers. We also import ToPrimitive trait. The Rust code for this is here included.

```
use std::vec::Vec;
use num_bigint::BigInt;
use num_traits::{Zero, One, ToPrimitive};

fn main() {
let a: BigInt = "5074444444443393839383923433277777777444442".parse().unwrap();
let b: BigInt = "1838383833838383837899999999999999999998".parse().unwrap();
let n: BigInt = "71338888888888828282828287777777777777774".parse().unwrap();

        match solve_linear_congruence(&a, &b, &n) {
                Some(solutions) => {
println!("Solutions␣to␣{}x␣␣␣{}␣(mod␣{}):␣{:?}", a, b, n, solutions);
                }
                None => {
println!("The␣congruence␣{}x␣␣␣{}␣(mod␣{})␣has␣no␣solutions", a, b, n);
                }
        }
}

fn gcd_extended(a: &BigInt, b: &BigInt) -> (BigInt, BigInt, BigInt) {
        if a.is_zero() {
                return (b.clone(), BigInt::zero(), BigInt::one());
        }
        let (g, x1, y1) = gcd_extended(&(b % a), a);
        let x = y1 - (b / a) * &x1;
        let y = x1;
        (g, x, y)
}

```

```rust
fn mod_inverse(a: &BigInt, n: &BigInt) -> Option<BigInt> {
        let (g, x, _) = gcd_extended(a, n);
        if !g.is_one() {
                None
        } else {
                Some((x % n + n) % n)
        }
}

fn solve_linear_congruence(a: &BigInt, b: &BigInt, n: &BigInt)
-> Option<Vec<BigInt>> {
        let (g, _x, _) = gcd_extended(a, n);
        // Prefix '_x' to suppress warning

        if b % &g != BigInt::zero() {
                return None; // No solutions
        }

        let a_prime = a / &g;
        let b_prime = b / &g;
        let n_prime = n / &g;

        let x0 = (b_prime * mod_inverse(&a_prime, &n_prime)?) % &n_prime;
        let mut solutions = Vec::new();

        for i in 0..g.to_usize().unwrap() {
                solutions.push((x0.clone() + i * &n_prime) % n);
        }

        Some(solutions)
}
```

Compiling this code prints

"The congruence $5074444444443393839383923433277777777444442x \equiv 18383838338383838378999999999999999999998 \pmod{71338888888888828282828287777777777777774}$ has no solutions