

WEB3CLUBS FOUNDATION LIMITED

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

Foundational Mathematics for Web3 Builders

Implemented in RUST

Lecture 33

July 15, 2024

Fast powering algorithm

- 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

This algorithm works as follows; To find n^k we do the following;
We start with n and square it repeatedly to find the sequence

$$\underline{n, n^2, n^4, n^8, n^{16}, \dots}$$

then select the terms in the sequence that multiply to give n^k .

Algorithm 1 (Successive Squaring to Compute $a^k \pmod{m}$)

Follow the following steps to compute the value of $a^k \pmod{m}$.

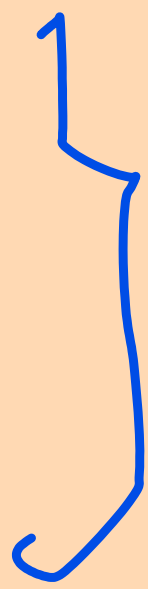
1. Write k as a sum of powers of 2,

$$k = \beta_0 \cdot 2^0 + \beta_1 \cdot 2^1 + \beta_2 \cdot 2^2 + \beta_3 \cdot 2^3 + \beta_4 \cdot 2^4 + \dots + \beta_r \cdot 2^r$$

where each β_i is either 0 or 1. ✓

Algorithm (conti..)

2. Perform successive squaring as follows

$$\begin{array}{lll} a^1 & \equiv A_0(\bmod m) & \\ a^2 \equiv (A_0)^2 & \equiv A_1(\bmod m) & \\ a^4 \equiv (A_1)^2 & \equiv A_2(\bmod m) & \\ a^8 \equiv (A_2)^2 & \equiv A_3(\bmod m) & \\ \vdots & & \vdots \\ a^{2^r} \equiv (A_{r-1})^2 & \equiv A_r(\bmod m) & \end{array}$$


Notice that it is easy to compute the above sequence of values since each number in the sequence is the square of the preceding one. Note also that the table has $r + 1$ lines, where r is the highest exponent of 2 in the binary expansion of k in Step 1.

Algorithm (conti..)

3. Thus, $a^k \equiv A_0^{\beta_0} \cdot A_1^{\beta_1} \cdot A_2^{\beta_2} \cdot A_3^{\beta_3} \cdot \dots \cdot A_r^{\beta_r} \pmod{m}$

Note that all the β_i 's are either 0 or 1. Thus, this number is a product of those A_i 's for which β_i equals 1.

Example 24

Use successive squaring to compute $7^{35} \bmod 27$.

Solution $35_{10} = 100011_2$

$$35 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$7^1 \equiv 7 \bmod 27$$

$$7^2 \equiv 22 \bmod 27$$

$$7^4 \equiv 25 \bmod 27$$

$$7^8 \equiv 4 \bmod 27$$

$$7^{16} \equiv 16 \bmod 27$$

$$7^{32} \equiv 13 \bmod 27 \quad 25^0 = 4^0 = 16^0 = 1$$

$$\text{Thus } 7^{35} = (7^1 \cdot 22^1 \cdot 25^0 \cdot 4^0 \cdot 16^0 \cdot 13^1) \bmod 27$$

$$= (7 \cdot 22 \cdot 13) \bmod 27$$

$$= 4 \bmod 27$$

Example 25

Compute $27^{234} \bmod 313$

Solution

$234_{10} = 11101010_2$. Now,

$$27^1 \equiv 27 \bmod 313$$

$$27^2 \equiv 103 \bmod 313$$

$$27^4 \equiv 280 \bmod 313$$

$$27^8 \equiv 150 \bmod 313$$

$$27^{16} \equiv 277 \bmod 313$$

$$27^{32} \equiv 44 \bmod 313$$

$$27^{64} \equiv 58 \bmod 313$$

$$27^{128} \equiv 234 \bmod 313$$

$$\begin{aligned} \text{Thus } 27^{234} \bmod 313 &= (234 \times 58 \times 44 \times 150 \times 103) \bmod 313 \\ &= 1 \bmod 313 \end{aligned}$$

Let's use the Rust code below to implement the fast powering algorithm:

```

1 fn main() {
2     let base = 27;
3     let exponent = 234;
4     let modulus = 313;
5
6     let result = mod_exp(base, exponent, modulus);
7
8     println!("{}", mod_exp(base, exponent, modulus, result));
9 }
10
11 fn mod_exp(mut base: u64, mut exp: u64, modulus: u64) -> u64 {
12     if modulus == 1 {
13         return 0;
14     }
15     let mut result = 1;
16     base = base % modulus;
17
18     while exp > 0 {
19         if exp % 2 == 1 {
20             result = (result * base) % modulus;
21         }
22         exp = exp >> 1;
23         base = (base * base) % modulus;
24     }
25     result
26 }

```



Understanding the Rust code

```
fn mod_exp(mut base: u64, mut exp: u64, modulus: u64) -> u64 {  
    if modulus == 1 {  
        return 0;  
    }  
    let mut result = 1;  
    base = base % modulus;  
  
    while exp > 0 {  
        if exp % 2 == 1 {  
            result = (result * base) % modulus;  
        }  
        exp = exp >> 1;  
        base = (base * base) % modulus;  
    }  
}
```

- `fn mod_exp(mut base: u64, mut exp: u64, modulus: u64) -> u64 {` in Rust defines a function named `mod_exp` (modular exponentiation) with specific parameters and a return type.
- ✓ `mut base:u64` indicates that `base` is mutable of type `u64`.
- ✓ `mut exp:u64`: indicates that `exponent` is mutable of type `u64`.
- ✓ `modulus: u64`: indicates that `modulus` is type `u64` and is not mutable.


Understanding the Rust code (conti...)

```
if modulus == 1 {  
    return 0;  
}
```



- If the modulus is 1 the function immediately returns 0. Note that any number modulo 1 is 0.

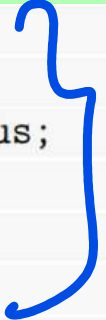
```
let mut result = 1;  
base = base % modulus;
```



- The result variable is initialized to 1
- The base is reduced modulo modulus to prevent overflow and simplify calculations.



Understanding the Rust code (conti...)

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



- The function enters a loop that continues as long as the exponent is greater than 0.

```
    if exp % 2 == 1 {  
        result = (result * base) % modulus;  
    }
```



- If the exponent is odd, multiply the current result by the base and take modulo modulus

```
    exp = exp >> 1;  
    base = (base * base) % modulus;
```

- The exponent is divided by 2 using a bitwise right shift.
- The base is squared and then reduced modulo modulus.

Understanding the Rust code (conti...)

- This process continues until exponent becomes 0, at which point result contains the final value of $\text{base}^{\text{exponent}} \bmod \text{modulus}$.

Example 26 (Showing how code works)

Compute $34^{35} \bmod 27$

Solution

let mut result = 1;

base = base % modulus; that is base = $34 \% 27 = 7$

Main loop, while exp > 0{

First Iteration:

exp = 35, base = 7, result = 1

exp % 2 == 1 is true, so **update result**:

result = (result * base) % modulus; that is, result = $(1 * 7) \% 27 = 7$ ✓

Understanding the Rust code (conti...)

Solution (conti...)

Update exp: $\text{exp} = \text{exp} \gg 1$; That is, $\text{exp} = 35 \gg 1 = 17$

Square base: $\text{base} = (\text{base} * \text{base}) \% \text{modulus}$;

that is, $\text{base} = (7 * 7) \% 27 = 22$

Second Iteration:

$\text{exp} = 17$, $\text{base} = 22$, $\text{result} = 7$

$\text{exp} \% 2 == 1$ is true, so **update result:**

$\text{result} = (\text{result} * \text{base}) \% \text{modulus}$; that is,

$\text{result} = (7 * 22) \% 27 = 19$;

Update exp: $\text{exp} = \text{exp} \gg 1$; that is, $\text{exp} = 17 \gg 1 = 8$

Square base: $\text{base} = (22 * 22) \% 27 = 25$;

Understanding the Rust code (conti...)

Solution (conti...)

Third Iteration:

$\text{exp} = 8$, $\text{base} = 25$, $\text{result} = 19$

$\text{exp} \% 2 == 1$ is false

Update exp: $\text{exp} = \text{exp} \gg 1$; that is, $\text{exp} = 8 \gg 1 = 4$

Square base: $\text{base} = (25 * 25) \% 27 = 4$;

Fourth Iteration:

$\text{exp} = 4$, $\text{base} = 4$, $\text{result} = 19$

$\text{exp} \% 2 == 1$ is false

Update exp: $\text{exp} = \text{exp} \gg 1$; that is, $\text{exp} = 4 \gg 1 = 2$

Square base: $\text{base} = (4 * 4) \% 27 = 16$;

Understanding the Rust code (conti...)

Solution (conti...)

Fifth Iteration:

$\text{exp} = 2, \text{base} = 16, \text{result} = 19$

$\text{exp} \% 2 == 1$ is false

Update exp: $\text{exp} = \text{exp} >> 1$; that is, $\text{exp} = 2 >> 1 = 1$

Square base: $\text{base} = (16 * 16) \% 27 = 13$;

Sixth Iteration:

$\text{exp} = 1, \text{base} = 13, \text{result} = 19$

$\text{exp} \% 2 == 1$ is true

$\text{result} = (\text{result} * \text{base}) \% \text{modulus}$; that is,

$\text{result} = (19 * 13) \% 27 = 4$

Update exp: $\text{exp} = \text{exp} >> 1$; that is, $\text{exp} = 1 >> 1 = 0$

Square base: $\text{base} = (13 * 13) \% 27 = 7$.

Exit Loop: $\text{exp} = 0$, so exit the loop. Thus, $34^{35} \bmod 27 = 4$