

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Звіт за темою

**ДОСЛІДЖЕННЯ АЛГОРИТМІВ
РЕАЛІЗАЦІЇ АРИФМЕТИЧНИХ ОПЕРАЦІЙ
НАД ВЕЛИКИМИ ЧИСЛАМИ**

Виконали студенти
групи ФІ-32мн
Баєвський Константин,
Шифрін Денис,
Кріпака Ілля

Київ — 2023

ЗМІСТ

0.1	Мета практикуму	2
0.1.1	Постановка задачі та варіант	2
0.2	Хід роботи/Опис труднощів	2
1	Результати дослідження.....	3
1.1	C/C++ бібліотеки	3
1.2	Rust.....	4
1.2.1	Обгортки над C/C++	5
1.2.2	Нативні бібліотеки.....	8
1.3	C#	11
1.3.1	Використання GNU GMP бібліотеки із мовою програмування C#	11
1.3.2	Використання System.Numerics.BigInteger бібліотеки із мовою програмування C#	13
1.3.3	Висновки до розділу C#.....	14
1.4	Java	15
1.4.1	Використання бібліотеки BigInteger у мові програмування Java.....	15
1.4.2	Використання GNU GMP бібліотеки у мові програмування Java.....	22
1.4.3	Висновки до розділу Java.....	25
	Висновки	26

0.1 Мета практикуму

Дослідити алгоритми реалізації готових бібліотек над великими числами, скінченними полями та групами із точки зору ефективності за часом та пам'яттю.

0.1.1 Постановка задачі та варіант

Треба виконати	Зроблено
Дослідити бібліотеки на мові Rust	✓
Дослідити бібліотеки на мові C#	✓
Дослідити бібліотеки на мові Java	✓

0.2 Хід роботи/Опис труднощів

На початку роботи гуртом вибрали варіант 1А та мови на яких будемо проводити дослідження – Rust, C#, Java. Також у нас виникала лише одна часова складність під час виконання лабораторної роботи :-).

1 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

На початку дослідження було виявлено, що бібліотеки створюються, як правило, трьома способами:

- 1) реалізуються на мовах C/C++ і використовується там же;
- 2) реалізуються тою мовою, де їх будуть використовувати (до прикладу, крейт `crypto-bigint` у Rust);
- 3) спочатку реалізуються як із пункту 1, а потім пишеться певна обгортка над операціями/типами задля легкості використання.

Спробуємо провести аналіз деяких із представників цих бібліотек.

1.1 C/C++ бібліотеки

Спочатку наведемо короткий опис із бібліотек, що можна було взяти до аналізу прямо із завдання лабораторної роботи. Потрібно одразу попередити, що поки для нас C/C++ дається важко для написання своїх прикладів(. Наведемо доступні бібліотеки.

– PARI/GP – спеціалізована комп'ютерна алгебраїчна система, що у більшості використовується науковцями для різних областей від топології чи числового аналізу до фізики. Як зрозуміли, то GP у ній є такою собі спеціалізованою скриптовою мовою, а PARI, що є безпосередньо `libpari` бібліотекою написаною на C.

```
? factor(20! +20)
%20 =
[      2 2]
[      5 1]
[     71 1]
[1713311273363831 1]
```

Рисунок 1.1 – Приклад факторизації числа $20! + 20$ на мові GP.

– GNU GMP – безкоштовна бібліотека для арифметики довільної точності, яка працює над цілими числами зі знаком, раціональними числами та числами з плаваючою комою. Тут немає жодних практичних обмежень точності, окрім тих, що передбачають доступну пам'ять у машині, на якій працює GMP. GMP має багатий набір функцій, і функції мають звичайний інтерфейс.

```
1  #include <iostream>
2  #include <gmpxx.h>
3
4  int main(int argc, char* argv[]) {
5
6      mpz_t x;
7      mpz_t y;
8      mpz_t z;
9
10     mpz_init_set_ui(x, 0);
11     mpz_init_set_ui(y, 1);
12     mpz_init(z);
13
14     unsigned long long int end = strtoull(argv[1], nullptr, 0);
15
16     std::cout << 0 << "\n" << 1 << "\n";
17     for (unsigned long long int i = 0; i < end; i++) {
18         mpz_add(z, x, y);
19         std::cout << z << "\n";
20
21         mpz_set(x, y);
22         mpz_set(y, z);
23     }
24 }
25
```

Рисунок 1.2 – Приклад виводу у циклі підсумованих великих чисел із використанням типів GNU GMP бібліотеки.

1.2 Rust

У світі сучасної розробки програмного забезпечення мова програмування Rust завоювала своє місце, завдяки вражаючій комбінації ефективності та безпеки. Ця мова програмування може використовуватися для розробки від операційних систем до аналізу даних. Тож розгляньмо ключові бібліотеки Rust, що можуть бути використані для великих обчислень.

1.2.1 Обгортки над C/C++

– Rug – обгортка над C/C++, що написана на Rust, надає цілі числа та числа з плаваючою комою з довільною точністю та округленням разом із операціями над ними. Саме ця бібліотека реалізовує високорівневий інтерфейс над бібліотеками GNU:

- GMP (для цілих чисел та раціональних чисел),
- MPFR (для чисел із плаваючою точкою),
- MPC (для комплексних чисел).

Надалі наведу трохи скріншотів та пояснень до коду для цілочисельного типу. Почнемо із піднесення до степеня.

```
use rug::Integer;
// 7 * 143 modulo 1000 = 1, so 7 has an inverse 143.
// 7 ^ -5 modulo 1000 = 143 ^ 5 modulo 1000 = 943.
let n = Integer::from(7);
let e = Integer::from(-5);
let m = Integer::from(1000);
let power = match n.pow_mod(&e, &m) {
    Ok(power) => power,
    Err(_) => unreachable!(),
};
assert_eq!(power, 943);
```

Рисунок 1.3 – Приклад використання піднесення до степеня за модулем.

```
#[inline]
pub fn pow_mod(mut self, exponent: &Self, modulo: &Self) -> Result<Self, Self> {
    match self.pow_mod_mut(exponent, modulo) {
        Ok(()) => Ok(self),
        Err(()) => Err(self),
    }
}
```

Рисунок 1.4 – Код який використовує функція 1.3.

Тепер пояснимо код піднесення до степеня. Перший скріншот 1.3 є прикладом до використання. Потім як передемо по коду, передемо до скріншоту 1.4 – це є відправною точкою. Далі від неї переходимо у функцію 1.5. Вона має у собі

```

#[inline]
#[allow(clippy::result_unit_err)]
pub fn pow_mod_mut(&mut self, exponent: &Self, modulo: &Self) -> Result<(), ()> {
    let Some(PowModIncomplete { sinverse : Option<Integer>, .. }) = self.pow_mod_ref(exponent, modulo) else {
        return Err(());
    };
    if let Some(sinverse : &Integer) = &sinverse {
        xmpz::pow_mod( rop: self, base: sinverse, exponent, modulo);
    } else {
        xmpz::pow_mod( rop: self, base: (), exponent, modulo);
    }
    Ok(())
}

```

Рисунок 1.5 – Далі функція 1.4 переходить сюди.

```

pub fn pow_mod_ref<'a>(
    &'a self,
    exponent: &'a Self,
    modulo: &'a Self,
) -> Option<PowModIncomplete<'a>> {
    if exponent.is_negative() {
        let Some(InvertIncomplete { sinverse : Integer, .. }) = self.invert_ref(modulo) else {
            return None;
        };
        Some(PowModIncomplete {
            ref_self: None,
            sinverse: Some(sinverse),
            exponent,
            modulo,
        })
    } else if !modulo.is_zero() {
        Some(PowModIncomplete {
            ref_self: Some(self),
            sinverse: None,
            exponent,
            modulo,
        })
    } else {
        None
    }
}

```

Рисунок 1.6 – За умови збереженого значення переходимо у цю функцію із 1.5.

розвилку, де 1.6 переходить і повертає збережені значення як такі є, а 1.7 спускається до виклику функції із C. Усе це можливо завдяки FFI(Foreign Function Interface), що є певним механізмом, де програма написана на одній мові програмування може використовувати бібліотеки/сервіси написані на іншій мові програмування. У Rust FFI забезпечує абстракцію та сумісність

```

#[inline]
pub fn pow_mod<0: OptInteger>(rop: &mut Integer, base: 0, exponent: &Integer, modulo: &Integer) {
    if exponent.is_negative() {
        finish_invert(rop, s: base, modulo);
        let rop : *mut mpz_t = rop.as_raw_mut();
        unsafe {
            gmp::mpz_powm(
                rop,
                base: rop.cast_const(),
                exp: exponent.as_neg().as_raw(),
                modu: modulo.as_raw(),
            );
        }
    } else {
        let rop : *mut mpz_t = rop.as_raw_mut();
        let base : *const mpz_t = base.mpz_or( default: rop);
        unsafe {
            gmp::mpz_powm(rop, base, exp: exponent.as_raw(), modu: modulo.as_raw());
        }
    }
}

```

Рисунок 1.7 – Задля обчислення числа заново, переходимо сюди із 1.5.

без витрат, що зводиться до швидкості виконання коду на С.

Наведемо ще один приклад функції. Цього разу будемо викликати пошук оберненого.

```

use rug::Integer;
let n = Integer::from(2);
// Modulo 4, 2 has no inverse: there is no i such that 2 × i = 1.
let inv_mod_4 = match n.invert(&Integer::from(4)) {
    Ok(_) => unreachable!(),
    Err(unchanged) => unchanged,
};
// no inverse exists, so value is unchanged
assert_eq!(inv_mod_4, 2);
let n = inv_mod_4;
// Modulo 5, the inverse of 2 is 3, as 2 × 3 = 1.
let inv_mod_5 = match n.invert(&Integer::from(5)) {
    Ok(inverse) => inverse,
    Err(_) => unreachable!(),
};
assert_eq!(inv_mod_5, 3);

```

Рисунок 1.8 – Приклад використання функції invert().

Перший скрішнот 1.8 показує приклад використання оберненого за модулем, де за будь-яких умов повертається число. Тільки у негативному випадку всеодно отримаємо число для якого повинно було б бути обчислено


```
#[inline]
pub fn invert(mut self, modulo: &Self) -> Result<Self, Self> {
    match self.invert_mut(modulo) {
        Ok(()) => Ok(self),
        Err(()) => Err(self),
    }
}
```

Рисунок 1.9 – Сама функція `inverse()`.

```
#[inline]
#[allow(clippy::result_unit_err)]
pub fn invert_mut(&mut self, modulo: &Self) -> Result<(), ()> {
    match self.invert_ref(modulo) {
        Some(InvertIncomplete { sinverse : Integer, .. }) => {
            xmpz::finish_invert(rop: self, s: &sinverse, modulo);
            Ok(())
        }
        None => Err(()),
    }
}
```

Рисунок 1.10 – Переходимо сюди із 1.9.

```
pub fn invert_ref<'a>(&'a self, modulo: &'a Self) -> Option<InvertIncomplete<'a>> {
    xmpz::start_invert(op: self, modulo).map(|sinverse : Integer| InvertIncomplete { sinverse, modulo })
}
```

Рисунок 1.11 – Остання функція для переходу із 1.10.

обернений елемент. Прейшовши до 1.9, можна побачити, що все зводиться до виклику іншої функції 1.10, яка навпаки, замість повернення нового елементу, мутує даний на вхід об'єкт. Далі перейшовши до 1.11, можна побачити, що знову ж усе зводиться до обгорнутих функцій від C/C++ бібліотеки.

1.2.2 Нативні бібліотеки

– Одна із бібліотек, про яку б хотіли розповісти – `crypto-bigint`, що реалізована знову на Rust. Її особливістю є те, що вона підтримується спільнотою, що робить криптологічні крейти для Rust. Також можна додати, що вона використовує у собі багато макросів, що рекурсивно генерують код для певних значень.

Сама бібліотека реалізована ефективно, зокрема використовує такий собі підхід константних значень до яких може бути реалізовано код за потреби макросами. Ось, наприклад, є приклад макросу 1.12, який у свою чергу може бути розписаний у 1.13

```

6 macro_rules! impl_modulus {
7     ($name:ident, $uint_type:ty, $value:expr) => {
8         #[derive(Clone, Copy, Debug, Default, Eq, PartialEq)]
9         pub struct $name {}
10        impl<const DLIMBS: usize>
11            $crate::modular::constant_mod::ResidueParams<{ <$uint_type>::LIMBS }> for $name
12        where
13            $uint_type: $crate::ConcatMixed<MixedOutput = $crate::Uint<DLIMBS>,>,
14        {
15            const LIMBS: usize = <$uint_type>::LIMBS;
16            const MODULUS: $uint_type = {
17                let res = <$uint_type>::from_be_hex($value);
18
19                // Check that the modulus is odd
20                if res.as_limbs()[0].0 & 1 == 0 {
21                    panic!("modulus must be odd");
22                }
23
24                res
25            };
26            const R: $uint_type = $crate::Uint::MAX
27                .const_rem(&Self::MODULUS)
28                .0
29                .wrapping_add(&$crate::Uint::ONE);
30            const R2: $uint_type =
31                $crate::Uint::const_rem_wide(Self::R.square_wide(), &Self::MODULUS).0;
32            const MOD_NEG_INV: $crate::Limb = $crate::Limb(
33                $crate::Word::MIN.wrapping_sub(
34                    Self::MODULUS
35                        .inv_mod2k_vartime($crate::Word::BITS as usize)
36                        .as_limbs()[0]
37                        .0,
38                ),
39            );
40            const R3: $uint_type = $crate::modular::montgomery_reduction(
41                &Self::R2.square_wide(),
42                &Self::MODULUS,
43                Self::MOD_NEG_INV,
44            );
45        }
46    };
47 }

```

Рисунок 1.12 – Приклад синтаксису макросу у crypto-bigint.

Як можна побачити, тут була застосована основна родзинка Rust. У тому, що можна написати код, який буде сам себе генерувати для нових подібних типів або буде скорочувати роботу розробнику.

– Також розповім про останню, просто досить популярну бібліотеку, яку, зокрема, використовував для лабораторних робіт. Це є num-bigint

```

#[derive(Clone, Copy, Debug, Default, Eq, PartialEq)]
pub struct Modulus {}
impl<const DLIMBS: usize>
::crypto_bigint::modular::constant_mod::ResidueParams<{ <U256>::LIMBS }> for Modulus
where
    U256: ::crypto_bigint::ConcatMixed<MixedOutput=::crypto_bigint::Uint<DLIMBS>>,
{
    const LIMBS: usize = <U256>::LIMBS;
    const MODULUS: U256 = {
        let res = <U256>::from_be_hex("ffffffff00000000ffffffffffffbce6faada7179e84f3b9cac2fc632551"
        );

        if res.as_limbs()[0].0 & 1 == 0 {
            panic!("modulus must be odd");
        }

        res
    };
    const R: U256 = ::crypto_bigint::Uint::MAX
        .const_rem(&Self::MODULUS)
        .0
        .wrapping_add(&::crypto_bigint::Uint::ONE);
    const R2: U256 =
        ::crypto_bigint::Uint::const_rem_wide(Self::R.square_wide(), &Self::MODULUS).0;
    const MOD_NEG_INV: ::crypto_bigint::Limb = ::crypto_bigint::Limb(
        ::crypto_bigint::Word::MIN.wrapping_sub(
            Self::MODULUS
                .inv_mod2k_vartime(::crypto_bigint::Word::BITS as usize)
                .as_limbs()[0]
                .0,
        ),
    );
    const R3: U256 = ::crypto_bigint::modular::montgomery_reduction(
        &Self::R2.square_wide(),
        &Self::MODULUS,
        Self::MOD_NEG_INV,
    );
}

```

Рисунок 1.13 – Приклад розширення макросу для типу U256.

простою ефективною бібліотекою, де реалізовані знакові та беззнакові числа, використання різних баз для створення великих чисел (32, 64, 128 бітні числа). Зокрема у ній реалізовані ефективні алгоритми множення (Карацуби та Тоом-3) та на додачу редукцію Монтгомері разом із іншими оптимізованими операціями. Також додаю, що приклад використання цієї бібліотеки є у папці `./rust`. Там наведена реалізація 3 лабораторної роботи із асиметричної криптографії.

1.3 C#

C# — це сучасна об'єктно-орієнтована мова програмування загального призначення, розроблена корпорацією Microsoft у рамках її ініціативи .NET і пізніше затверджена як стандарт Європейською асоціацією виробників комп'ютерів (ЕСМА) і міжнародними стандартами (ISO). В основному C# використовується для розробки Web додатків, ігор та застосунків для Windows. Розгляньмо деякі із бібліотек.

1.3.1 Використання GNU GMP бібліотеки із мовою програмування C#

Як вже було згадано вище, GNU GMP допомагає виконувати будь які арифметичні операції із довільною точністю, що обмежена лише пам'яттю машини, на якій виконується програма. Для того, щоб використовувати цю бібліотеку в мові програмування CSharp слід встановити до проєкту Nuget пакет із назвою "Math.Gmp.Native.NET".

Бібліотека "Math.Gmp.Native.NET" автоматично завантажує на виконання 32-бітну або 64-бітну бібліотеку GNU MP, яка відповідає поточній архітектурі центрального процесора, тим самим дозволяючи будувати проєкти Visual Studio для режимів Any CPU, x86 або x64. Вона базується на розширеній збірці GNU MP, яка автоматично визначає тип поточного процесора і вибирає оптимізацію коду на мові асемблера для даного процесора, тим самим забезпечуючи найкращу продуктивність.

Клас Math.Gmp.Native.gmp_lib містить статичний метод для кожної з функцій GNU MP. Інші типи визначені для відтворення структур і псевдонімів типів GNU MP і мов C, а також мовних конструкцій C.

Заради зручності цю довідку було створено з офіційного посібника GNU MP версії 6.1.2. Вона демонструє, з прикладами, як кожна функція GNU MP викликається в .NET.

Для початку порівняємо код мовою програмування C# із кодом на Rust.(рис. 1)

```
private static void Check_mpz_powm()
{
    // Create, initialize, and set the value of base to 2.
    mpz_t @base = new mpz_t();
    gmp_lib.mpz_init_set_ui(@base, 2U);

    // Create, initialize, and set the value of exp to 4.
    mpz_t exp = new mpz_t();
    gmp_lib.mpz_init_set_ui(exp, 4U);

    // Create, initialize, and set the value of mod to 3.
    mpz_t mod = new mpz_t();
    gmp_lib.mpz_init_set_ui(mod, 3U);

    // Create, initialize, and set the value of rop to 0.
    mpz_t rop = new mpz_t();
    gmp_lib.mpz_init(rop);


    // Set rop = base^exp mod mod.
    gmp_lib.mpz_powm(rop, @base, exp, mod);

    // Assert that rop is 1.
    Console.WriteLine(gmp_lib.mpz_get_si(rop) == 1 ? "Result: 1. Method works correctly." : $"Result: not 1. Method does not works c

    // Release unmanaged memory allocated for rop, base, exp, and mod.
    gmp_lib.mpz_clears(rop, @base, exp, mod, null);
}
```

Рисунок 1.14 – Приклад піднесення до степеня.

Видно, що для використання GMP треба використати змінні класу `mpz_t` і задавати певні значення чисел у функцію `mpz_powm`. Якщо метод відпрацював успішно, то в консолі можна буде побачити повідомлення про коректну роботу. (рис. 2)



```
Result (mpz_powm): 1. Method works correctly.
Result (BigInteger): 1. Method works correctly.
```

Рисунок 1.15 – Приклад для перевірки коректності роботи алгоритму.

Розглянемо для остаточного розуміння, як саме використовувати написану мовою програмування C бібліотеку у поєднанні із CSharp оболонкою для знаходження зворотнього елементу. (рис. 3)

```

1 reference
public static void Check_mpz_invert()
{
    // Create, initialize, and set the value of op1 to 3.
    mpz_t op1 = new mpz_t();
    gmp_lib.mpz_init_set_ui(op1, 3U);

    // Create, initialize, and set the value of op2 to 11.
    mpz_t op2 = new mpz_t();
    gmp_lib.mpz_init_set_ui(op2, 11U);

    // Create, initialize, and set the value of rop to 0.
    mpz_t rop = new mpz_t();
    gmp_lib.mpz_init(rop);

    // Set rop to the modular inverse of op1 mod op2, i.e. b, where op1 * b mod op1 = 1.
    gmp_lib.mpz_invert(rop, op1, op2);

    // Assert that rop is 4,
    Console.WriteLine(gmp_lib.mpz_get_si(rop) == 4
        ? "Result: 4. Method works correctly."
        : $"Result: not 4. Method does not works correctly.");

    // Release unmanaged memory allocated for rop, op1, and op2.
    gmp_lib.mpz_clears(rop, op1, op2, null);
}

```

Рисунок 1.16 – Приклад знаходження зворотнього елементу.

1.3.2 Використання System.Numerics.BigInteger бібліотеки із мовою програмування C#

BigInteger - це тип даних в програмуванні, який дозволяє представляти та виконувати операції з великими цілими числами, які виходять за межі можливостей стандартних числових типів. В інших мовах програмування цей тип може мати різні назви, але концепція залишається однаковою - він дозволяє працювати з числами довільної довжини і не обмежується розміром пам'яті, яку може займати число.

У .NET BigInteger знаходиться в просторі імен System.Numerics і забезпечує можливість виконувати арифметичні операції, порівнювати та взаємодіяти з великими цілими числами, незалежно від їх розміру, забезпечуючи точність та надійність операцій. Це особливо корисно при виконанні обчислень, де великі числа можуть виникати при розв'язанні математичних задач, криптографії, обробці великих даних тощо.

Таким чином, маємо реалізацію піднесення до степеню, використовуючи System.Numerics.BigInteger.ModPow.

Після виконання такого шматочку коду в результаті побачимо

```

private static void Check_ModPow()
{
    // Base, exponent, and modulus values
    BigInteger baseValue = 2;
    BigInteger exponent = 4;
    BigInteger modulus = 3;

    // Calculate (base^exponent) % modulus using BigInteger.ModPow method
    BigInteger result = BigInteger.ModPow(baseValue, exponent, modulus);

    // Assert that rop is 1.
    Console.WriteLine(result == 1 ? "Result (BigInteger): 1. Method works correctly." : $"Result (BigInteger): not 1. Method does not work correctly.");
}

```

Рисунок 1.17 – Приклад піднесення до степеня BigInteger.

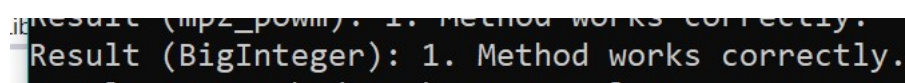


Рисунок 1.18 – Результат піднесення до степеня BigInteger.

Таким чином, бачимо, що обидві функції коректно працюють.

1.3.3 Висновки до розділу C#

Мова програмування C# є відмінним вибором для роботи із різними арифметичними операціями, оскільки вона підтримує різні бібліотеки для роботи з великими числами та арифметикою високої точності. Перш за все, вбудований клас `System.Numerics.BigInteger` надає можливість оперувати дуже великими цілими числами без втрати точності, що робить його ідеальним вибором для обчислень, де великі числа важливі.

Додатково, наявність зовнішніх бібліотек, таких як `Math.Gmp.Native` або `System.Numerics.MPFR`, розширює можливості C#. Ці бібліотеки забезпечують швидку та ефективну роботу з великими числами, що особливо корисно для вимогливих до продуктивності застосувань. Наприклад, вони можуть бути використані в задачах криптографії, математичних моделях або статистичних розрахунках, де точність та продуктивність є ключовими.

Крім того, можливість використання різних бібліотек дає можливість вибору відповідно до конкретних потреб проекту. Наприклад, `System.Numerics.BigInteger` надає зручний та простий інтерфейс для роботи

з великими числами, тоді як `Math.Gmp.Native` або `System.Numerics.MPFR` можуть бути використані для оптимізації продуктивності у вимірюваних додатках.

Загалом, здатність `C#` підтримувати і вбудовані, і сторонні бібліотеки для роботи з арифметичними операціями робить її потужним і гнучким інструментом для вирішення широкого спектру математичних завдань у сучасному програмуванні.

1.4 Java

Java – високорівнева, об’єктно орієнтована мова програмування, що має на меті дозволити розробникам писати один раз і запускати програми будь-де. Це означає, що скомпільований код Java може працювати на всіх платформах, які підтримують Java, без необхідності повторної компіляції. Розгляньмо бібліотеки і для цієї мови.

1.4.1 Використання бібліотеки `BigInteger` у мові програмування Java

В мови програмування Java є декілька бібліотек, які можуть працювати з багаторозрядною арифметикою:

- `BigInteger` та `BigDecimal`. Ці класи є вбудованими в Java і дозволяють вам працювати з цілими числами та десятковими числами великої точності. Вони підтримують арифметичні операції з високою точністю.

- `Arffloat`. Ця Java бібліотека надає інструменти для роботи з десятковою арифметикою з довільною точністю. Вона може бути використана для виконання обчислень з високою точністю, схожих на `MPFR`.

- `JScience`. Ця бібліотека має підтримку для роботи з великими

цілими числами, десятковою арифметикою та багатою функціональністю для числових обчислень.

– **Apache Commons Math**. Ця бібліотека містить різноманітні математичні функції та алгоритми для Java. Вона також містить класи для роботи з десятковою арифметикою та раціональними числами.

Але ми зупинимось на вбудованій і достатньо багатій бібліотеці для роботи з багаторозрядною арифметикою – **BigInteger**.

BigInteger – це потужний інструмент для виконання будь-яких арифметичних операцій з довільною точністю, і його обмеження обумовлене лише ресурсами пам'яті вашої системи.

Бібліотека **BigInteger** є частиною стандартної бібліотеки Java і входить в пакет `java.math`. Вам не потрібно встановлювати його додатково, він доступний за замовчуванням при використанні будь-якого стандартного дистрибутива Java. Ви можете імпортувати даний пакет та використовувати для роботи з великими цілими числами у своїх Java програмах без будь-яких додаткових дій.

BigInteger надає аналоги всім примітивним цілочисельним операторам Java та всім відповідним методам із `java.lang.Math`. Крім того, **BigInteger** надає операції для модульної арифметики, обчислення GCD, перевірки простоти, генерації простих значень, обробки бітів і кількох інших різних операцій. Семантика арифметичних операцій точно імітує цілочисельні арифметичні оператори Java. Семантика порозрядних логічних операцій точно імітує побітові цілочисельні оператори Java. Також надаються модульні арифметичні операції для обчислення залишків, піднесення до степеня та обчислення мультиплікативних обернених. Ці методи завжди повертають невід'ємний результат від 0 до (modulus - 1) включно.

BigInteger має 3 основних поля:

- `static BigInteger ONE` // Константа **BigInteger**.
- `static BigInteger TEN` // Константа **BigInteger** десять.
- `static BigInteger ZERO` // Константа **BigInteger** нуль.

Тепер розглянемо декілька методів, які нас цікавлять.

Метод `java.math.BigInteger.pow(int exponent)` використовується для обчислення збільшення `BigInteger` до степеня деякого іншого числа, переданого як експонента, значення якого дорівнює ($this^{exponent}$). Цей метод виконує операцію над поточним `BigInteger`, за допомогою якого цей метод викликається, а експонента передається як параметр.

```
public BigInteger pow(int exponent)
```

Повертає велике ціле число, значення якого дорівнює ($this^{exponent}$).
Зауважимо, що експонента є цілим числом, а не `BigInteger`.

Параметри: *exponent* – експонента, до якої має бути піднесено це *BigInteger*.

Повернення: $this^{exponent}$

Виняток: *ArithmeticException* – експонента від’ємна. (Це призведе до того, що операція дасть неціле значення.)

Приклад використання методу `BigInteger.pow(int exponent)`:

```
import java.math.BigInteger;

public class BigIntegerExample {
    public static void main(String[] args) {

        //Create 2 numbers of type BigInteger
        BigInteger base = new BigInteger( val: "2");

        //Power to which we raise the number
        BigInteger exponent = new BigInteger( val: "10");

        //Raise the number to the specified power
        BigInteger result = base.pow(exponent.intValue());

        //Show the result
        System.out.println(base + " raised to a power " + exponent + " is equal to " + result);
    }
}
```

Рисунок 1.19 – Приклад піднесення до степеня `BigInteger`.

2 raised to a power 10 is equal to 1024

Рисунок 1.20 – Результат піднесення до степеня `BigInteger`.

```

© java.math.BigInteger
@Contract(pure = true) *
@NotNull *
public BigInteger pow(
    @Range(from = 0, to = Integer.MAX_VALUE) * int exponent
)

```

Рисунок 1.21 – Функція, яка використовується при піднесенні до степеня.

Метод **Java.math.BigInteger.modPow()** повертає `BigInteger`, значення якого дорівнює $(this^{exponent} \bmod m)$. Якщо експонента $= 1$, повертається значення (це $\bmod m$), а якщо експонента < 0 , повертається модульне мультиплікативне значення $(this^{-exponent})$. Метод видає `ArithmeticException`, якщо $m \leq 0$.

```
public BigInteger modPow(BigInteger exponent, BigInteger m)
```

Параметри: метод приймає два параметри: параметр експоненти, параметр модуля.

Повернення: метод повертає об'єкт `BigInteger`, значення якого дорівнює $(this^{exponent} \bmod m)$.

Виняток: *ArithmeticException*: Якщо $(m \leq 0)$ або експонента від'ємна, і це велике ціле число не є взаємно простим до m .

Приклад використання методу **BigInteger.modPow()**:

```

public static void modPow(){
    // Create 3 BigInteger objects
    BigInteger bigint1, bigint2, result;

    // Initializing all BigInteger Objects
    bigint1 = new BigInteger( val: "23895");
    bigint2 = new BigInteger( val: "14189");
    BigInteger exp = new BigInteger( val: "15");

    // Perform modPow operation on the objects and exponent
    result = bigint1.modPow(exp, bigint2);
    String expression = bigint1 + "^" + exp + " mod " + bigint2 + " equal is " + result;

    // Displaying the result
    System.out.println(expression);
}

```

Рисунок 1.22 – Приклад піднесення до степеня `BigInteger` за `mod`.

$23895^{15} \bmod 14189$ equal is 344

Рисунок 1.23 – Результат піднесення до степеня BigInteger за mod.

```
@Contract(pure = true) @NotNull
public BigInteger modPow(BigInteger exponent, BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("BigInteger: modulus not positive");

    // Trivial cases
    if (exponent.signum == 0)
        return (m.equals(ONE) ? ZERO : ONE);

    if (this.equals(ONE))
        return (m.equals(ONE) ? ZERO : ONE);

    if (this.equals(ZERO) && exponent.signum >= 0)
        return ZERO;

    if (this.equals(negConst[1]) && (!exponent.testBit(0)))
        return (m.equals(ONE) ? ZERO : ONE);

    boolean invertResult;
    if ((invertResult = (exponent.signum < 0)))
        exponent = exponent.negate();

    BigInteger base = (this.signum < 0 || this.compareTo(m) >= 0
        ? this.mod(m) : this);
    BigInteger result;
    if (m.testBit(0)) { // odd modulus
        result = base.oddModPow(exponent, m);
    } else {
```

```

int p = m.getLowestSetBit(); // Max pow of 2 that divides m

BigInteger m1 = m.shiftRight(p); // m/2**p
BigInteger m2 = ONE.shiftLeft(p); // 2**p

// Calculate new base from m1
BigInteger base2 = (this.signum < 0 || this.compareTo(m1) >= 0
    ? this.mod(m1) : this);

// Calculate (base ** exponent) mod m1.
BigInteger a1 = (m1.equals(ONE) ? ZERO :
    base2.oddModPow(exponent, m1));

// Calculate (this ** exponent) mod m2
BigInteger a2 = base.modPow2(exponent, p);

// Combine results using Chinese Remainder Theorem
BigInteger y1 = m2.modInverse(m1);
BigInteger y2 = m1.modInverse(m2);

if (m.mag.length < MAX_MAG_LENGTH / 2) {
    result = a1.multiply(m2).multiply(y1).add(a2.multiply(m1).multiply(y2)).mod(m);
} else {
    MutableBigInteger t1 = new MutableBigInteger();
    new MutableBigInteger(a1.multiply(m2)).multiply(new MutableBigInteger(y1, t1);
    MutableBigInteger t2 = new MutableBigInteger();
    new MutableBigInteger(a2.multiply(m1)).multiply(new MutableBigInteger(y2, t2);
    t1.add(t2);
    MutableBigInteger q = new MutableBigInteger();
    result = t1.divide(new MutableBigInteger(m), q).toBigInteger();
}
}

return (invertResult ? result.modInverse(m) : result);
}

```

Рисунок 1.24 – Функція, яка використовується при піднесенні до степеня за mod.

Метод **`Java.math.BigInteger.modInverse()`** повертає модульну мультиплікативну інверсію `this` за mod `m`. Цей метод створює `ArithmeticException`, якщо $m \leq 0$ або `this` не має мультиплікативного зворотного за модулем `m` (тобто $\gcd(this, m) \neq 1$).

public BigInteger modInverse(BigInteger m)

Параметри: `m` – модуль.

Повернення: метод повертає об'єкт `BigInteger`, значення якого дорівнює $this^{-1} \bmod m$.

Виняток: `ArithmeticException` – $m \leq 0$, або цей `BigInteger` не має мультиплікативного зворотного за модулем `m` (тобто цей `BigInteger` не є взаємно простим до `m`).

Приклад використання методу `BigInteger.Inverse()`:

```
public static void Inverse(){
    BigInteger a = new BigInteger( val: "7"); // The number for which we are looking for the return element
    BigInteger n = new BigInteger( val: "10"); // Module

    BigInteger inverse = a.modInverse(n); // Find the return element

    System.out.println("The inverse of a " + a + " mod " + n + " equal is " + inverse);
}
}
```

Рисунок 1.25 – Приклад знаходження оберненого `BigInteger` за `mod`.

The inverse of a 7 mod 10 equal is 3

Рисунок 1.26 – Результат знаходження оберненого `BigInteger` за `mod`.

```
@Contract(pure = true) @NotNull
public BigInteger modInverse(BigInteger m) {
    if (m.signum != 1)
        throw new ArithmeticException("BigInteger: modulus not positive");

    if (m.equals(ONE))
        return ZERO;

    // Calculate (this mod m)
    BigInteger modVal = this;
    if (signum < 0 || (this.compareMagnitude(m) >= 0))
        modVal = this.mod(m);

    if (modVal.equals(ONE))
        return ONE;

    MutableBigInteger a = new MutableBigInteger(modVal);
    MutableBigInteger b = new MutableBigInteger(m);

    MutableBigInteger result = a.mutableModInverse(b);
    return result.toBigInteger( sign: 1);
}
```

Рисунок 1.27 – Функція, яка використовується при знаходженні оберненого за `mod`.

1.4.2 Використання GNU GMP бібліотеки у мові програмування Java

GNU GMP (GNU Multiple Precision Arithmetic Library) – це бібліотека на мові програмування C для виконання арифметики з довільною точністю. Щоб використовувати GMP в Java, можна скористатися JNI (Java Native Interface) або використовувати інші засоби для інтеграції коду на C в Java.

Операція піднесення до ступеня за модулем за допомогою бібліотеки GNU GMP в Java використовує функцію *'mpz_powm'*. Також потрібно буде створити 2 файли – GMPExample.java та GMPExample.c.

Приклад використання GMP в Java з допомогою JNI для піднесення числа до степеня за mod:

```
public class GMPExample {
    static {
        System.loadLibrary( libname: "gmp"); // Load the GMP library
    }

    1 usage
    public native static String powerMod(String base, String exp, String mod); // External method for exponentiation

    public static void main(String[] args) {
        String base = "2";
        String exp = "10";
        String mod = "1000";

        String result = powerMod(base, exp, mod); // Example of calling the exponentiation method
        System.out.println("GMPpow: " + base + "^" + exp + " mod " + mod + " equal is " + result);
    }
}
```

Рисунок 1.28 – Приклад коду у файлі GMPExample.java.

```

#include <jni.h>
#include "GMPEExample.h"
#include <gmp.h>

JNIEXPORT jstring JNICALL Java_GMPEExample_powerMod(JNIEnv *env, jclass cls, jstring base, jstring exponent, jstring modulus) {
    const char *baseStr = (*env)->GetStringUTFChars(env, base, 0);
    const char *expStr = (*env)->GetStringUTFChars(env, exponent, 0);
    const char *modStr = (*env)->GetStringUTFChars(env, modulus, 0);

    mpz_t baseInt, expInt, modInt, result;
    mpz_init_set_str(baseInt, baseStr, 10);
    mpz_init_set_str(expInt, expStr, 10);
    mpz_init_set_str(modInt, modStr, 10);
    mpz_init(result);

    mpz_powm(result, baseInt, expInt, modInt);

    char *resultStr = mpz_get_str(NULL, 10, result);
    jstring resultString = (*env)->NewStringUTF(env, resultStr);

    mpz_clear(baseInt);
    mpz_clear(expInt);
    mpz_clear(modInt);
    mpz_clear(result);
    free(resultStr);

    (*env)->ReleaseStringUTFChars(env, base, baseStr);
    (*env)->ReleaseStringUTFChars(env, exponent, expStr);
    (*env)->ReleaseStringUTFChars(env, modulus, modStr);

    return resultString;
}

```

Рисунок 1.29 – Приклад коду у файлі GMPEExample.c.

Варто зазначити, що цей код треба компілювати з використанням gcc (GNU Compiler Collection) або іншого компілятора.

Операція інверсії (знаходження зворотного елемента) для багаторозрядної арифметики в GNU GMP в Java використовує функцію '*mpz invert*'. Ця функція знаходить обернений елемент по модулю, тобто, якщо c – обернений елемент числа a за модулем b , то $a * c \equiv 1(mod b)$. Також потрібно буде створити 2 файли – GMPEExample.java та GMPEExample.c.

Приклад використання GMP в Java з допомогою JNI для знаходження зворотного числа за mod:

```
public class GMPEXample {
    static {
        System.loadLibrary("gmp"); // Load the GMP library
    }

    1 usage
    public native static String invert(String base, String modulus);

    public static void main(String[] args) {
        String base = "7"; // The value we're looking for the inverted element for
        String modulus = "11"; // The module by which we look for the inverted element

        String result = invert(base, modulus);
        System.out.println("The inverse of " + base + " mod " + modulus + " is: " + result);
    }
}
```

Рисунок 1.30 – Приклад коду у файлі GMPEXample.java.

```
#include <jni.h>
#include <gmp.h>

JNIEXPORT jstring JNICALL Java_GMPEXample_invert(JNIEnv *env, jclass cls, jstring base, jstring modulus) {
    const char *baseStr = (*env)->GetStringUTFChars(env, base, 0);
    const char *modStr = (*env)->GetStringUTFChars(env, modulus, 0);

    mpz_t baseInt, modInt, result;
    mpz_init_set_str(baseInt, baseStr, 10);
    mpz_init_set_str(modInt, modStr, 10);
    mpz_init(result);

    // Виклик функції інверсії
    int success = mpz_invert(result, baseInt, modInt);

    (*env)->ReleaseStringUTFChars(env, base, baseStr);
    (*env)->ReleaseStringUTFChars(env, modulus, modStr);

    if (success == 0) {
        // Якщо інверсія не вдалася (наприклад, якщо числа не взаємно прості)
        return (*env)->NewStringUTF(env, "No inverse exists");
    } else {
        // Якщо інверсія успішна, повертаємо результат у вигляді рядка
        char *resultStr = mpz_get_str(NULL, 10, result);
        jstring resultString = (*env)->NewStringUTF(env, resultStr);

        mpz_clear(baseInt);
        mpz_clear(modInt);
        mpz_clear(result);
        free(resultStr);

        return resultString;
    }
}
```

Рисунок 1.31 – Приклад коду у файлі GMPEXample.c.

Обернений елемент числа a за модулем b існує, якщо a і b взаємно прості.

1.4.3 Висновки до розділу Java

Мова програмування Java є високофункціональною та гнучкою для роботи з багаторозрядною арифметикою. Вбудована бібліотека BigInteger надає зручний інтерфейс для операцій над дуже великими цілими числами, забезпечуючи високу точність без втрати продуктивності.

Крім цього, існує можливість використання сторонніх обгортки та інтерфейсів, таких як Java GMP (jGMP) та Java MPFR (jMPFR) для бібліотек GMP (GNU Multiple Precision Arithmetic Library) та MPFR (Multiple Precision Floating-Point Reliable Library) відповідно. Оскільки завдяки ним розширюються можливості Java у сфері багаторозрядної арифметики. Використання цих обгортки дозволяє розробникам Java використовувати потужні функціональності GMP та MPFR у своїх проектах, розширюючи можливості багаторозрядної арифметики та операцій з плаваючою комою в Java.

Усі ці можливості дають розробникам можливість обирати інструмент, який найкраще відповідає конкретним вимогам їх проекту. Зручний інтерфейс BigInteger та висока продуктивність сторонніх бібліотек роблять Java ефективним інструментом для вирішення різноманітних математичних завдань у світі програмування.

ВИСНОВКИ

У ході дослідження було проведено порівняльний аналіз наявних бібліотек над великими числами для трьох різних мов програмування: Rust, C#, Java. Також було виділено основні методи реалізації подібних бібліотек. Для підсумку, варто сказати, що наявні бібліотеки ефективно реалізують арифметику над великими числами.