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

Звіт за темою:
«Дослідження алгоритмів реалізації арифметичних операцій над великими числами»

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

Київ — 2023

1 Мета практикуму

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

1.1 Постановка задачі та варіант

Треба виконати	**Зроблено**
Дослідити бібліотеки на мові Rust	✓
Дослідити бібліотеки на мові C	✓
Дослідити бібліотеки на мові Python?	✓

2 Хід роботи/Опис труднощів

На початку роботи гуртом вибрали варіант 1C та мови на яких будемо проводити дослідження – Rust, C, Python?. Також у нас виникала лише одна часова складність :-).

3 Результати дослідження

На початку дослідження варто зазначити, що, відповідно до нашого досвіду, можна побачити в основному бібліотеки створюються 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.

3.1 C/C++ бібліотеки

Спочатку наведемо короткий опис із бібіліотек, що можна було взяти до аналізу прямо із завдання лабораторної роботи. Потрібно одразу попередити, що поки для нас C/C++ дається туго(.

- 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.
- 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.

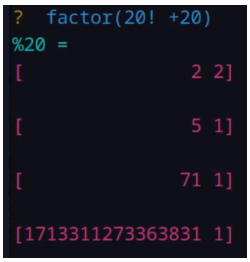


Рис. 1: Приклад факторизації числа $20! + 20$ на мові GP.

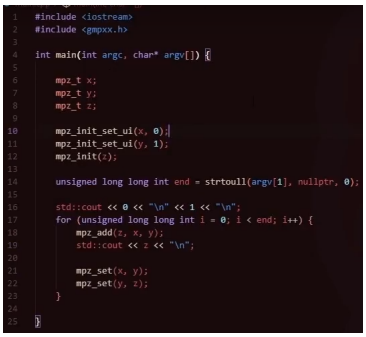


Рис. 2: Приклад виводу у циклі підсумованих великих чисел із використанням типів GNU GMP бібліотеки.

3.2 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 скрішотів та пояснень до коду для цілочисельного типу.

  Почнемо із піднесення до степеня.

```rust
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);
```

Рис. 3: Приклад використання піднесення до степеня за модулем.

Рис. 4: Код який використовує функція 3.

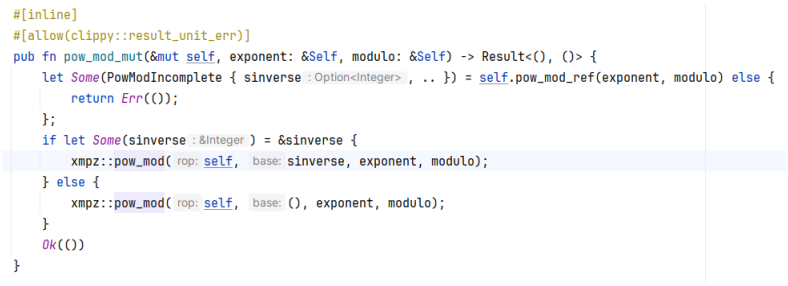


Рис. 5: Далі функція 4 переходить сюди.

Тепер пояснимо код піднесення до степеня. Перший скріншот 3 є прикладом до використання. Потім як передемо по коду, передемо до скріншоту 4 – це є відправною точкою. Далі від неї переходимо у функцію 5. Вона має у собі розвилку, 6 переходить і повертає збережені значення як такі є, а 7 спускається до виклику функції із C. Усе це можливо завдяки FFI(Foreign Funciton 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 без витрат, що зводиться до швидкості виконання коду на C. Наведемо ще один приклад функції.

Цього разу будемо викликати пошук оберненого.

Перший скрішнот 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 обернений елемент. Прейшовши до 9, можна побачити, що все зводиться до виклику іншої функції 10, яка навпаки, замість повернення нового елементу, мутує даний на вхід об'єкт. Далі перейшовши до 11, можна побачити, що знову ж усе зводиться до завраплених функцій від C/C++ бібліотеки.

- C??

3.3 Нативні бібліотеки

- Одна із бібліотек, про яку б хотіли розповісти – ??crypto-bigint, що реалізована знову ж 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 13, який у свою чергу може бути розписаний у ??

```
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
    }
}
```

Рис. 6: За умови збереженого значення переходимо у цю функцію із 5.

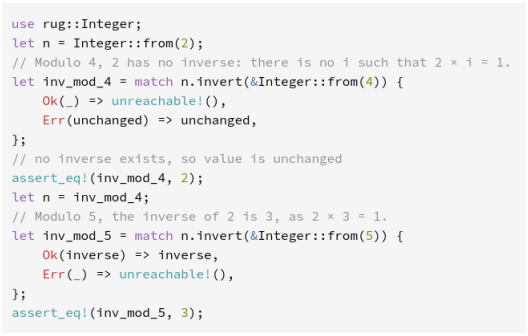
Рис. 7: Задля обчислення числа заново, переходимо сюди із 5.

Як можна побачити, тут була застосована основна родзинка Rust. У тому, що можна написати код, який буде сам себе генерурвати 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 13num-bigint 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 алгортими множення (Карацуби та Toom-3) та на додачу редукцію Монтгомері разом із іншими оптимізованими операціями. Також додам, що приклад використання цієї біліотеки є у папці **./rust**. Там наведена реалізація 3 лабораторної роботи із асиметричної криптографії. Саме тут не бачу сенсу наводити додаткові картинки, так як це усе можна подивитися у документації та і воно буде повторювати попереднє.
-

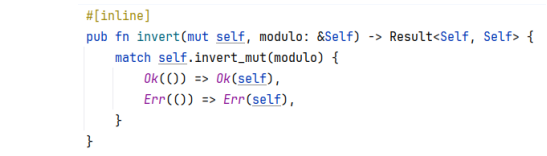
4 Висновки

За допомогою реалізації практикуму "Баєсівський підхід в криптоаналізі: побудова і дослідження детерміністичної та стохастичної вирішуючих функцій" дізналися на практиці як повинен відбуватися баєсівський підхід у криптоаналізі. Також були долучені до створення такого собі «маленького» прикладу із побудови вирішуючих функцій для заданого розподілу повідомлень.



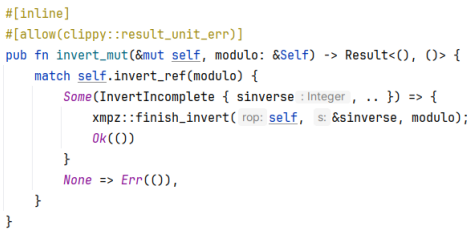
```rust
use rug::Integer;
let n = Integer::from(2);
// Modulo 4, 2 has no inverse: there is no i such that 2 × i = 1.
let inv_mod_4 = match n.invert(&Integer::from(4)) {
    Ok(_) => unreachable!(),
    Err(unchanged) => unchanged,
};
// no inverse exists, so value is unchanged
assert_eq!(inv_mod_4,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

Рис. 8: Приклад використання функції invert().



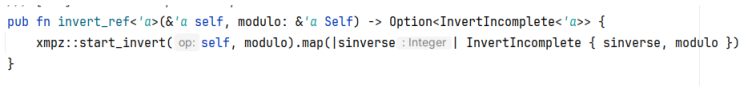
```rust
#[inline]
pub fn invert(mut self, modulo: &Self) -> Result<Self, Self> {
    match self.invert_mut(modulo) {
        Ok(()) => Ok(self),
        Err(()) => Err(self),
    }
}
```

Рис. 9: Сама функція inverse().



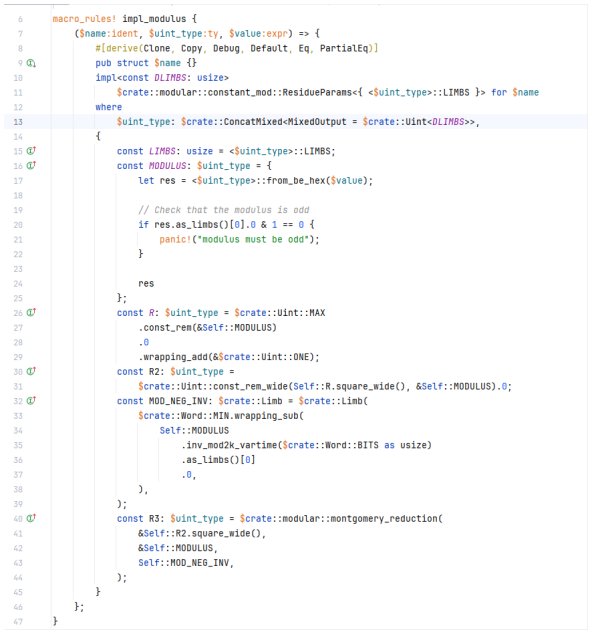
```rust
#[inline]
#[allow(clippy::result_unit_err)]
pub fn invert_mut(&mut self, modulo: &Self) -> Result<(), ()> {
    match self.invert_ref(modulo) {
        Some(InvertIncomplete { sinverse : Integer , .. }) => {
            xmpz::finish_invert( rop: self,  s: &sinverse, modulo);
            Ok(())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

Рис. 10: Переходимо сюди із 9.



```rust
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 })
}
```

Рис. 11: Остання функція для переходу із 10.

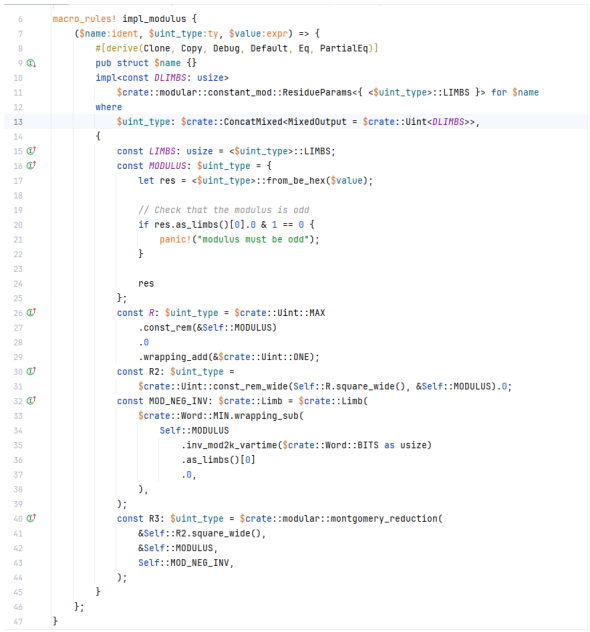


```
macro_rules! impl_modulus {
    ($name:ident, $uint_type:ty, $value:expr) => {
        #[derive(Clone, Copy, Debug, Default, Eq, PartialEq)]
        pub struct $name {}
        impl<const DLIMBS: usize>
            $crate::modular::constant_mod::ResidueParams<{ <$uint_type>::LIMBS }> for $name
        where
            $uint_type: $crate::ConcatMixed<MixedOutput = $crate::Uint<DLIMBS>>,
        {
            const LIMBS: usize = <$uint_type>::LIMBS;
            const MODULUS: $uint_type = {
                let res = <$uint_type>::from_be_hex($value);

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

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

Рис. 12: Приклад синтаксису макросу у crypto-bigint.



```
macro_rules! impl_modulus {
    ($name:ident, $uint_type:ty, $value:expr) => {
        #[derive(Clone, Copy, Debug, Default, Eq, PartialEq)]
        pub struct $name {}
        impl<const DLIMBS: usize>
            $crate::modular::constant_mod::ResidueParams<{ <$uint_type>::LIMBS }> for $name
        where
            $uint_type: $crate::ConcatMixed<MixedOutput = $crate::Uint<DLIMBS>>,
        {
            const LIMBS: usize = <$uint_type>::LIMBS;
            const MODULUS: $uint_type = {
                let res = <$uint_type>::from_be_hex($value);

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

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

Рис. 13: Приклад розширення макросу для типу U256.