НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ

Комп'ютерний практикум № 1

Вибір та реалізація базових фреймворків та бібліотек

Підгрупа 1А.

Виконали:
Волинець Сергій ФІ-42мн
Сковрон Роман ФІ-42мн
Радомир Беш ФІ-42мн

Київ — 2025

1 Мета

Вибір базових бібліотек або сервісів для подальшої реалізації. криптосистеми.

2 Завдання на лабораторну роботу

Дослідити бібліотеки багаторозрядної арифметики, вбудовані в програмні платформи C++/C# (BigInteger), Java (BigInt), Python або Crypto++ (обрати одну з них) для процесорів із 32-розрядною архітектурою та обсягом оперативної пам'яті до 16 ГБ (user Endpoints terminal).

3 Дослідження

Для нашого дослідження ми вибрали мову програмування Java, а точніше, бібліотеку BigInteger. Крім того, ми розглянемо мови Rust та Python. У середовищі Java є кілька бібліотек для роботи з багаторозрядною арифметикою, і однією з основних є клас BigInteger з пакету java.math. Цей клас дозволяє здійснювати арифметичні операції з великими числами, розміри яких виходять за межі стандартних типів даних, таких як int або long. Розглянемо основні особливості використання BigInteger для процесорів із 32-розрядною архітектурою і обсягом оперативної пам'яті до 16 ГБ.

Ключ до розуміння того, як працює BigInteger, лежить у його внутрішньому представленні. BigInteger використовує внутрішній масив для зберігання значення числа у представленні 32-бітних слів. Тобто в пам'яті зберігається розклад числа за основою 2^{32}. Ця основа це називається базисом або радіксом. BigInteger також зберігає знак числа (додатний чи від'ємний). Для цього він використовує маркер, який вказує, чи є це число додатним чи від'ємним. Числа в представленні BigInteger, якщо потрібно, можуть динамічно виділяти пам'ять, саме тому вони не мають фіксованого розміру, і майже нічим не обмежені.

Алгоритми в BigInteger працюють з числами довільної довжини. Операції виконуються над масивами 32-бітних слів.

Додавання працює доволі просто. Функція проходиться по двох масивах, додає відповідні значення при цьому зберігаючи та додаючи значення переносу. Алгоритм схожий до того алгоритму, який ми використовували в школі.

Множення великих чисел вимагає розбиття числа на менші частини, перемноження цих частин, а потім додавання отриманих добутків разом. Для швидшої роботи з дуже великими числами використовуються більш ефективні алгоритми, такі як множення Карацуби [KO62] або множення Тома-Кука. Алгоритм Карацуби - це алгоритм швидкого множення. Це алгоритм "розділяй і володарюй", який зводить множення двох n-цифрових чисел до трьох множень $n/2$-цифрових чисел, і повторюючи цю редукцію, до щонайбільше $n^{\log_2 3} \approx n^{1.58}$ одноцифрових множень. Очевидно, що цей алгоритм асимптотично швидший за традиційний алгоритм, який виконує n^2 одноцифрових добутків. Алгоритм Тома-Кука розбиває задані числа a і b на k менших частин, кожна з яких має довжину l, і виконує операції над частинами. Зі збільшенням k можна об'єднати багато під операцій множення, таким чином

зменшуючи загальну обчислювальну складність алгоритму. Крім того, під операції множення можна обчислювати рекурсивно. Складність цього алгоритму рівна $\Theta(n^{\log 5/\log 3}) \approx \Theta(n^{1.46})$. Java BigInteger.multiply використовує стандартне множення для множення для малих чисел алгоритм Карацуба для чисел середньої довжини, та алгоритм Тома-Кука, з $k = 3$, для чисел великої довжини.

Для ділення двох числел, також використовуються різні алгоритми. Якщо довжина числа, або різниця довжин чисел менша певного порогу, то використовується алгоритм Кнута, зі складністю $O(n^2)$, інакше, застосовується алгоритм Бурникеля - Циглера зі складністю $O(n \log n)$ [BZ98]. Алгоритм Кнута передбачає ефективне обчислення частки та остачі за допомогою зсувів, віднімань та іноді мультиплікативних обернених, замість прямого використання оператора ділення, що робить ділення швидшим. Для великих чисел використовується алгоритм Бурникеля - Циглера, що є рекурсивним алгоритмом. Для ділення A/B, він використовує алгоритм $D_{2n/1n}$ для ділення $2n$-цифрового числа на n-цифрове число з n-цифровою остачею. Якщо числа не того розміру, якого потрібно, то можна провести нормалізацію. Припускаючи, що n парне і достатньо велике, щоб вийти з рекурсії, ми розбиваємо A на чотири частини $[A_1; A_2; A_3; A_4]$ довжиною $n = 2$ кожна, а B на дві частини $[B_1; B_2]$. Спочатку обчислимо верхню частину Q_1 частки $\lfloor A/B \rfloor$, використовуючи частини A_1; A_2; A_3 з A і верхню частину B_1 з B за відповідним алгоритмом з назвою $D_{3n/2n}$, який рекурсивно викликає $D_{2n/1n}$ і по суті є розширенням алгоритму $D_{3n/2n}$. Використовуючи залишок що відповідає Q_1 і нижній частині A_4 від A, ми отримуємо аналогічно нижню частину Q_2 використовуючи алгоритм $D_{3n/2n}$ знову.

Піднесення до степеня в Java відбувається з використанням піднесень до квадрата, при цьому виконуючи різні пришвидшення. Наприклад, піднесення в степені двійки можна швидко реалізувати бітовим зсувом.

Знаходження модуля числа виконується так само, як і цілочисельне ділення, але повертається не результат, а частка.

Піднесення до степеня за модулем для непарного модуля виконується з використанням віконного алгоритму. Ідея полягає у тому, щоб зберігати біжучий добуток $b1 = n^{(\text{старші біти експоненти})}$, і додавати до нього біти експоненти. Залежно від того, які біти наступні, до 3-бітового вікна буде застосовано певний патерн додавання. Наприклад, якщо додається 000, то піднести до квадрату, якщо 001, то помножити на n, і так далі. Для парного модуля, задача зводиться до попередньої (в двох екземплярах), після чого результат знаходиться з використанням китайської теореми про лишки.

4 Аналіз швидкодії алгоритмів багаторозрядної арифметики на прикладі бібліотеки BigInteger в Java, бібліотеки BigInt в Rust, та мови Python

Для аналізу швидкодії алгоритмів, було проведено практичні заміри часу виконання алгоритмів, з усередненням часу. Деякі алгоритми – це вбудовані алгоритми, а деякі, які можуть виявитися корисними для практичного застосування, було реалізовано власноруч. В результаті дослідження, було отримано наступні дані.

Табл. 1: Середній час виконання алгоритмів бібліотеки BigInteger для чисел довжиною в 1024, 2048, та 4096 біт в мілісекундах

Операція	**1024 біт**	**2048 біт**	**4096 біт**
Додавання	0.149038	0.179353	0.662192
Віднімання	0.068709	0.057730	0.189843
Множення	0.085653	0.121011	0.452259
Ділення	0.041874	0.066320	0.157867
Піднесення до малого степеня	1161.704453	2866.606040	8386.252683
Піднесення до малого степеня за схемою Горнера	1359.685523	4110.971217	10422.068631
Піднесення до великого степеня за схемою Горнера	782.345711	3805.727940	4146.865804
Додавання за модулем	0.030558	0.068837	0.085706
Віднімання за модулем	0.027482	0.054883	0.086755
Множення за модулем	0.040295	0.121007	0.181788
Ділення за модулем	0.027815	0.049418	0.029841
Піднесення до малого степеня з подальшим взяттям модуля	929.211160	4681.523990	38921.345963
Вбудоване піднесення до малого степеня за модулем	0.127951	0.223819	0.845944
Піднесення до малого степеня за схемою Горнера	777.293278	3126.443064	22727.894394
Піднесення до великого степеня за схемою Горнера	373.118151	1075.967598	10279.038735

Табл. 2: Середній час виконання алгоритмів бібліотеки мови Rust для чисел довжиною в 1024, 2048, та 4096 біт в мілісекундах

Операція	**1024 біт**	**2048 біт**	**4096 біт**

Операція	**1024 біт**	**2048 біт**	**4096 біт**
Додавання	0.001124	0.003008	0.003949
Віднімання	0.001197	0.002503	0.003908
Множення	0.001657	0.003472	0.007197
Ділення	0.001536	0.002404	0.004243
Піднесення до малого степеня	613.825313	1613.378775	5960.086977
Піднесення до малого степеня за схемою Горнера	1013.622439	3417.817883	12087.457466
Піднесення до великого степеня за схемою Горнера	7924.398150	27688.302389	38687.316548
Додавання за модулем	0.002664	0.005063	0.009654
Віднімання за модулем	0.002418	0.00478	0.009279
Множення за модулем	0.004421	0.009778	0.029100
Ділення за модулем	0.002536	0.004735	0.009799
Піднесення до малого степеня з подальшим взяттям модуля	0.066701	0.197391	0.742242
Вбудоване піднесення до малого степеня за модулем	0.076596	0.20646	0.753670
Піднесення до малого степеня за схемою Горнера	661.020491	1843.227677	7734.883335
Піднесення до великого степеня за схемою Горнера	5402.644940	13333.364138	18368.100775

Табл. 3: Середній час виконання алгоритмів бібліотеки мови Python для чисел довжиною в 1024, 2048, та 4096 біт в мілісекундах

Операція	**1024 біт**	**2048 біт**	**4096 біт**
Додавання	0.00163	0.00332	0.00319
Віднімання	0.00131	0.0025	0.00319
Множення	0.00296	0.01134	0.01788

Операція	**1024 біт**	**2048 біт**	**4096 біт**
Ділення	0.00143	0.002320	0.00348
Піднесення до малого степеня	2014.005333	5856.792	18797.044333
Піднесення до малого степеня за схемою Горнера	5355.753333	16704.563333	48603.789666
Піднесення до великого степеня за схемою Горнера	0.002333	0.002666	0.002333
Додавання за модулем	0.0016	0.00258	0.00554
Віднімання за модулем	0.00146	0.00308	0.00802
Множення за модулем	0.00631	0.01494	0.04175
Ділення за модулем	0.00202	0.00293	0.00449
Піднесення до малого степеня з подальшим взяттям модуля	0.085333	0.214	0.704333
Вбудоване піднесення до малого степеня за модулем	0.105666	0.205	0.688
Піднесення до малого степеня за схемою Горнера	0.113333	0.223333	0.744666
Піднесення до великого степеня за схемою Горнера	0.01	0.001333	0.001666

За даними результатами можна побачити, що операції для чисел довжини 1024 та 2048, виконуються в рази швидше ніж для чисел довжини 4096. Крім того, можна побачити, що такі операції як додавання, віднімання, множення ділення та їхні відповідні операції за модулем, відбуваються приблизно однаково швидко, але піднесення до степеня, відбувається в тисячі разів довше. Аналізуючи дані для мови Rust, можна побачити, що Rust демонструє значно кращу продуктивність порівняно з Java для всіх основних операцій з великими числами. Це свідчить про високу оптимізацію бібліотеки мови Rust для роботи з великими числами. Python також має гарні швидкість виконання але лише вбудованих операцій. Операції, що були реалізовані вручну, в рази повільніші. Отже, важливо не тільки мати гарний алгоритми, а й добре їх оптимізувати.

5 Висновки

Бібліотека BigInteger мови програмування Java забезпечує підтримку багаторозрядної арифметики, що дозволяє працювати з великими числами, що перевищують межі стандартних типів даних, таких як int або long. Бібліотека BigInteger підтримує ефективні алгоритми для виконання основних арифметичних операцій з великими числами. Додавання, віднімання та множення великих чисел реалізуються через оптимізовані алгоритми, як-от множення Карацуби та Тома-Кука, що значно пришвидшують ці операції порівняно з наївними методами. Проведений аналіз швидкодії показав, що операції додавання, віднімання, множення і ділення мають подібну швидкість, а от піднесення до степеня займає значно більше часу, особливо для великих чисел. Використання бібліотеки BigInteger у Java є доцільним для задач, що потребують роботи з великими числами, таких як криптографія, де важлива точність та швидкість виконання операцій над великими цілими числами.

За результатами аналізу даних можна зробити кілька важливих висновків. По-перше, видно, що операції з числами довжиною 1024 та 2048 біт виконуються значно швидше, ніж з числами довжиною 4096 біт. Зі збільшенням довжини числа час виконання операцій зростає експоненційно. Це особливо помітно для таких складних операцій, як піднесення до степеня, яке займає тисячі разів більше часу в порівнянні з базовими операціями (додавання, віднімання, множення, ділення) та їхніми варіантами за модулем.

Аналізуючи дані для бібліотек BigInt в Rust, BigInteger в Java та стандартний int в Pythnon, можна зробити ще одне важливе спостереження: Rust демонструє значно кращу продуктивність порівняно з Java та Python для всіх основних операцій з великими числами. Наприклад, базові операції в Rust виконуються в рази швидше, а операції з піднесенням до степеня та взяттям модуля — в десятки разів. Це вказує на високу оптимізацію бібліотеки мови Rust для роботи з великими цілими числами.

Отже, цей аналіз підтверджує важливість не лише правильного вибору алгоритму, а й його ефективної оптимізації. Погана оптимізація навіть найкращого алгоритму може погіршити його продуктивність, особливо при роботі з великими данними.

Література

[BZ98] Christoph Burnikel and Joachim Ziegler. Fast recursive division, 1998.

[KO62] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 12 1962.