КОМП'ЮТЕРНИЙ ПРАКТИКУМ КРЕДИТНОГО МОДУЛЯ

"МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ"

Виконали: студенти групи ФІ-93

Ємець Єлизавета, Звєрєв Сергій, Коваленко Дар'я

Лабораторна робота № 1

Тема: «Вибір та реалізація базових фреймворків та бібліотек».

Варіант 1А. Дослідити бібліотеки багатослівної арифметики.

Бібліотеки багаторозрядної арифметики, вбудовані в програмні платформи C++/ C# (BigInteger), Java (BigInt) та Python (обрати одну з них) для процесорів із 32-розрядною архітектурою та обсягом оперативної пам'яті до 8 ГБ (робочі станції).

Java (BigInt)

Незмінні цілі числа довільної точності. Всі операції поводяться так, ніби BigIntegers представлено у нотації з двома доповненнями (як примітивні цілі типи Java). BigInteger надає аналоги всіх примітивних цілочисельних операторів Java та всіх відповідних методів з java.lang.Math. Крім того, BigInteger надає операції для модулярної арифметики, обчислення GCD, тестування первинності, генерації простих чисел, маніпуляції з бітами та деякі інші різноманітні операції.

Семантика арифметичних операцій точно імітує семантику операторів цілочисельної арифметики Java, як визначено у специфікації мови Java. Наприклад, ділення на нуль генерує виключення ArithmeticException, а ділення від'ємного на додатне дає від'ємний (або нульовий) залишок. Всі деталі специфікації щодо переповнення ігноруються, оскільки BigIntegers створюються настільки великими, наскільки це необхідно для того, щоб вмістити результати операції.

Семантика операцій зсуву розширює семантику операторів зсуву Java, щоб дозволити від'ємні відстані зсуву. Зсув вправо з від'ємною відстанню зсуву призводить до зсуву вліво, і навпаки. Оператор зсуву вправо без знаку (>>>) опущено, оскільки ця операція не має сенсу у поєднанні з абстракцією "нескінченного розміру слова", що надається цим класом.

Операції порівняння виконують цілочисельні порівняння зі знаком, аналогічні тим, що виконуються операторами відношення та рівності Java.

Операції модулярної арифметики передбачено для обчислення залишків, піднесення до степеня та обчислення обернених мультиплікативних чисел. Ці методи завжди повертають невід'ємний результат, від 0 до (по модулю - 1) включно.

Бітові операції оперують з одним бітом двійкового представлення операнда. Якщо необхідно, операнд розширюється за знаком так, щоб він містив вказаний біт. Жодна з однобітових операцій не може створити BigInteger зі знаком, відмінним від знаку BigInteger, над яким виконується операція, оскільки вони впливають лише на один біт, а абстракція "нескінченного розміру слова", що надається цим класом, гарантує, що перед кожним BigInteger існує нескінченна кількість "віртуальних знакових бітів".

Усі методи та конструктори цього класу генерують виключення NullPointerException, якщо в якості вхідного параметра передається нульове посилання на об'єкт.

У Java всі байти представляються в системі з двійковим доповненням з використанням запису з прямим порядком байтів . Він зберігає найстарший байт слова за найменшою адресою пам'яті (найнижчий індекс). Більше того, перший біт байта також є знаковим бітом.

- *1000 0000* представляет *-128*
- *0111 1111* представляет собой 127
- *1111 1111* представляет *-1*

Вся магія класу BigInteger починається з властивості mag. Вона зберігає задане значення в масиві, використовуючи двійкове представлення, що дозволяє обійти обмеженняпримітивних типів даних.

Більше того, BigInteger групує їх 32-бітними порціями - наборами по чотири байти. Завдяки цьому величина всередині визначення класу оголошується як масив типу int: **int[] mag;**

Цей масив зберігає величину заданого значення у великій системі числення. Нульовий елемент цього масиву є старшим за значенням int величини.

abs()	Повертає BigInteger, значенням якого є абсолютне значення цього BigInteger.
add()	Цей метод повертає BigInteger, просто обчислюючи значення 'this + val'.

and()	Цей метод повертає BigInteger, обчислюючи значення 'this & val'.
andNot()	Цей метод повертає BigInteger, обчислюючи значення 'this & ~val '.
bitCount()	Цей метод повертає кількість бітів у представленні двійкового доповнення цього BigInteger, які відрізняються від знакового біта.
bitLength()	Цей метод повертає кількість бітів у представленні мінімального двійкового доповнення цього знакового біта без урахування знакового біта.
clearBit()	Цей метод повертає BigInteger, значення якого дорівнює цьому BigInteger, біт якого очищено.
compareTo()	Цей метод порівнює це BigInteger з заданим BigInteger.
divide()	Цей метод повертає BigInteger, обчислюючи значення 'this /~val'.
divideAndRemainder()	Цей метод повертає BigInteger, обчислюючи значення 'this & ~val ', за яким слідує 'this%value'.
doubleValue()	Цей метод перетворює BigInteger до double.
equals()	Цей метод порівнює дане BigInteger з заданим Object на рівність.
flipBit()	Цей метод повертає BigInteger, значення якого дорівнює цьому BigInteger з перевернутим бітом.
floatValue()	Цей метод перетворює BigInteger у число з плаваючою комою.
gcd()	Цей метод повертає BigInteger, значення якого є найбільшим спільним дільником між abs(this) і abs(val).
getLowestSetBit()	Цей метод повертає індекс крайнього правого біта (найнижчого порядку) у цьому BigInteger (кількість нульових бітів праворуч від крайнього правого біта).
hashCode()	Цей метод повертає хеш-код для цього BigInteger.

intValue()	Цей метод перетворює BigInteger в int.
isProbablePrime()	Цей метод повертає булеве значення true тоді і тільки тоді, коли BigInteger є простим, інакше для складених значень він повертає false.
longValue()	Цей метод перетворює BigInteger в long.
max()	Цей метод повертає максимальне значення між цим BigInteger та val.
min()	Цей метод повертає мінімальне значення між цим BigInteger та val.
mod()	Цей метод повертає значення BigInteger для даної моди m.
modInverse()	Цей метод повертає BigInteger, значенням якого є "цей зворотний mod m".
modPow()	Цей метод повертає BigInteger, значення якого дорівнює 'thisexponent mod m'.
multiply()	Цей метод повертає BigInteger шляхом обчислення значення 'this *val'.
negate()	Цей метод повертає BigInteger, значенням якого є '-this'.
nextProbablePrime()	Цей метод повертає наступне просте число, яке є більшим за дане BigInteger.
not()	Цей метод повертає BigInteger, значенням якого є '~this'.
or()	Цей метод повертає BigInteger, значенням якого є 'this | val'
pow()	Цей метод повертає BigInteger, значенням якого є 'this^{exponent}'.
probablePrime()	Цей метод повертає додатне просте ціле число BigInteger із заданою довжиною bitLength.
remainder()	Цей метод повертає BigInteger, значенням якого є 'this % val'.
setBit()	Цей метод повертає BigInteger, значення якого дорівнює цьому BigInteger з встановленим бітом.

shiftLeft()	Цей метод повертає BigInteger зі значенням 'this << val'.
shiftRight()	Цей метод повертає BigInteger, значенням якого є 'this >> val'.
signum()	Цей метод повертає функцію signum цього BigInteger.
subtract()	Цей метод повертає BigInteger, значенням якого є 'this - val'.
testbit()	Цей метод повертає булеве значення true, якщо вказаний біт встановлено.
toByteArray()	Цей метод повертає байтовий масив, що містить двійкове представлення цього BigInteger.
toString()	Цей метод повертає десяткове рядкове представлення цього BigInteger.
valueOf()	Цей метод повертає BigInteger, значення якого еквівалентне значенню вказаного long.
xor()	xor() Цей метод повертає BigInteger шляхом обчислення значення 'this ^ val '.

У Java клас BigInteger надає можливість виконувати математичні операції з цілими числами довільної точності. Для процесорів із 32-бітною архітектурою та об'ємом оперативної пам'яті до 8 ГБ є кілька ключових обмежень і аспектів, які слід враховувати:

1.Обмеження Пам'яті:

Обмеження Адресації: Навіть якщо у вас є до 8 ГБ оперативної пам'яті, 32-бітна система обмежує максимальний розмір адресованої пам'яті для кожного процесу (зазвичай, до 2-4 ГБ, залежно від ОС і конфігурації). Отже, розмір і кількість об'єктів BigInteger, з якими ви можете працювати, обмежуються цим обсягом пам'яті.

GC (Garbage Collector) і продуктивність: Робота з великими об'єктами BigInteger може призвести до підвищеної активності збирача сміття, що, своєю чергою, може вплинути на продуктивність програми.

2. Продуктивність:

Швидкість Обчислень: Операції з великими числами зазвичай повільніші, ніж із примітивними типами, оскільки вони реалізовані через алгоритми програмного рівня, а не через апаратні інструкції процесора.

Операції: Деякі операції, наприклад, ділення або обчислення квадратного кореня, можуть бути значно менш ефективними для великих чисел порівняно з базовими операціями, такими як додавання або множення.

3. Алгоритми та Розробка:

Оптимізація Алгоритмів: Залежно від вашого додатка, вам, можливо, доведеться оптимізувати ваші алгоритми для мінімізації використання операцій BigInteger або для використання більш ефективних алгоритмів для зменшення часу обчислень і використання пам'яті.

Керування Пам'яттю: Особливу увагу слід приділяти керуванню пам'яттю, щоб мінімізувати необхідність виділення та звільнення великих блоків пам'яті.

Приблизний Розрахунок:
Припустимо, у вас є 3 ГБ пам'яті, доступної для зберігання об'єктів BigInteger. Припустимо, що середній об'єкт BigInteger займає 100 байт пам'яті (що відповідає числу розміром приблизно в 25-30 десяткових цифр, включно з метаданими об'єкта).

3 ГБ = 3 * 1024 МБ = 3 * 1024 * 1024 КБ = 3 * 1024 * 1024 * 1024 * 1024 байт
Кількість об'єктів = [Загальна пам'ять] / [Розмір одного об'єкта]
Кількість об'єктів = (3 * 1024 * 1024 * 1024 байт) / 100 байт ≈ 31,457,280 об’єктів

Це дуже грубий розрахунок, і реальна кількість об'єктів та їхній максимальний розмір значно варіюватимуться залежно від фактичного використання пам'яті, наявності інших об'єктів і даних у пам'яті, роботи збирача сміття, структури купи та стека, і безлічі інших чинників.

C#(BigInteger)

Тип BigInteger є незмінним типом, що представляє довільно велике ціле число, значення якого теоретично не має верхньої або нижньої межі. Члени типу тісно відповідають елементам BigInteger інших цілочисельних типів (Byte-типи , Int16, Int32, Int64SByte, UInt16, , UInt32i UInt64). Цей тип відрізняється від інших цілочисельних типів у платформі .NET Framework, які мають діапазон, вказаний їхніми MinValue властивостями та MaxValue .

Примірник можна використовувати так само, як і будь-який BigInteger інший цілочисельний тип. BigInteger перевантажує стандартні числові оператори, що дає змогу виконувати основні математичні операції, такі як додавання, віднімання, ділення, множення та унарне заперечення. Можна також використовувати стандартні числові оператори для порівняння двох BigInteger значень одне з одним. Як і інші цілочисельні типи, BigInteger також підтримує побітові Andоператори , Or, XOr, зсув вліво і зсув вправо. Для мов, які не підтримують користувацькі оператори, структура BigInteger також надає

еквівалентні методи для виконання математичних операцій. До них відносяться Add, Divide, Multiply, NegateSubtract, і кілька інших.

Змінність і структура BigInteger
У наступному прикладі створюється екземпляр об'єкта BigInteger , а потім збільшується його значення на одиницю.

```
BigInteger number = BigInteger.Multiply(Int64.MaxValue, 3);
number++;
Console.WriteLine(number);
```

Хоча здається, що в цьому прикладі змінюється значення наявного об'єкта, це не так. Об'єкти BigInteger є незмінними, а це означає, що всередині середовища CLR фактично створюється новий об'єкт BigInteger і присвоюється йому значення, на одиницю більше, ніж його попереднє значення. Цей новий об'єкт потім повертається об'єкту, що викликає.

Робота з байтовими масивами та шістнадцятковими рядками
При перетворенні значень типу BigInteger у байтовий масив або при перетворенні байтових масивів у значення типу BigInteger необхідно враховувати порядок байт. Структура BigInteger очікує, що окремі байти у байтовому масиві з'являтимуться у малому порядку (тобто молодші байти значення передують старшим байтам). Ви можете округлити значення типу BigInteger, викликавши метод ToByteArray, а потім передати отриманий байтовий масив конструктору BigInteger(Byte[]).
Щоб отримати значення типу BigInteger з байтового масиву, який представляє значення іншого інтегрального типу, можна передати значення інтегралу методу BitConverter.GetBytes, а потім передати отриманий байтовий масив конструктору BigInteger(Byte[]).

Структура BigInteger має специфічну конфігурацію для зберігання від'ємних значень у форматі доповнення двійки. Конструктор BigInteger(Byte[]) інтерпретує старший біт останнього байту як біт знаку, тож важливо враховувати цей факт при створенні чисел. Додатні значення, які зазвичай мають старший біт останнього байту у масиві, повинні мати додатковий байт із значенням 0, щоб вони не були помилково інтерпретовані як від'ємні числа. Наприклад, масив байтів 0xC0 0xBD 0xF0 0xFF може бути як -1,000,000, так і 4,293,967,296, але оскільки старший біт останнього байту дорівнює одиниці, конструктор BigInteger(Byte[]) інтерпретує його як -1,000,000. Для створення додатного BigInteger необхідно передати конструктору масив байтів з елементами 0xC0 0xBD 0xF0 0xFF 0x00.

```
int negativeNumber = -1000000;
uint positiveNumber = 4293967296;

byte[] negativeBytes = BitConverter.GetBytes(negativeNumber);
BigInteger negativeBigInt = new BigInteger(negativeBytes);
Console.WriteLine(negativeBigInt.ToString("N0"));

byte[] tempPosBytes = BitConverter.GetBytes(positiveNumber);
byte[] positiveBytes = new byte[tempPosBytes.Length + 1];
Array.Copy(tempPosBytes, positiveBytes, tempPosBytes.Length);
BigInteger positiveBigInt = new BigInteger(positiveBytes);
Console.WriteLine(positiveBigInt.ToString("N0"));
// The example displays the following output:
//    -1,000,000
//    4,293,967,296
```

Constructors

BigInteger(Byte[])	Ініціалізує новий екземпляр структури BigInteger, використовуючи значення з байтового масиву.
BigInteger(Decimal)	Ініціалізує новий екземпляр структури BigInteger з використанням десяткового значення.
BigInteger(Double)	Ініціалізує новий екземпляр структури BigInteger значенням з плаваючою комою подвійної точності.
BigInteger(Int32)	Ініціалізує новий екземпляр структури BigInteger 32-бітним знаковим цілим значенням.
BigInteger(Int64)	Ініціалізує новий екземпляр структури BigInteger з допомогою 64-бітового цілого зі знаком.
BigInteger(ReadOnlySpan<Byte>, Boolean, Boolean)	Ініціалізує новий екземпляр структури BigInteger, використовуючи значення в діапазоні байт, доступному тільки для читання, і необов'язково вказуючи кодування знаку та порядок байтів ендіанності.
BigInteger(Single)	Ініціалізує новий екземпляр структури BigInteger значенням з плаваючою комою одинарної точності.
BigInteger(UInt32)	Ініціалізує новий екземпляр структури BigInteger беззнаковим 32-бітним цілим значенням.
BigInteger(UInt64)	Ініціалізує новий екземпляр структури BigInteger беззнаковим 64-бітним цілим значенням.

Існують певні обмеження та аспекти, які слід враховувати під час роботи з великими числами (BigInteger) у C# на 32-розрядній архітектурі.

1. Обмеження Пам'яті:
Обмежений Адресний Простір: 32-розрядні додатки зазвичай обмежені приблизно 2-3 ГБ доступної пам'яті, навіть якщо на машині встановлено більше оперативної пам'яті.
Управління Пам'яттю: Особливо важливо стежити за управлінням пам'яттю, оскільки операції з великими числами можуть швидко використати доступну пам'ять, якщо не контролювати розподіл і звільнення ресурсів.

2. Продуктивність:
Процесорний Час: Операції з великими числами можуть вимагати значних обчислювальних ресурсів, особливо на 32-розрядній архітектурі, де операції з 64-розрядними числами менш ефективні порівняно з 64-розрядною архітектурою.
Операції Додавання/Множення: Додавання і множення великих чисел можуть бути особливо обчислювально складними і витратними в плані продуктивності.

3. Розмір і Представлення:
Розмір Чисел: BigInteger у C# не має верхньої межі, за винятком доступної пам'яті. Однак що більше число, то більше пам'яті та процесорного часу знадобиться для роботи з ним.
Подання: Хоча BigInteger дозволяє працювати з дуже великими числами, внутрішньо він використовує масив байтів для зберігання, що може впливати на ефективність операцій, особливо на 32-розрядних архітектурах.

4. Операційна Система та Платформа:
Платформа: Під 32-розрядні системи, можливо, доведеться особливо уважно ставитися до оптимізації та управління пам'яттю.
Сумісність: У деяких випадках, для оптимізації продуктивності, може бути раціонально розглянути використання нативних бібліотек або алгоритмів, спеціально адаптованих для 32-розрядних систем.

5. Розробка та Налагодження:
Тестування: Ретельне тестування вкрай важливе для забезпечення точності та продуктивності під час роботи з великими числами.
Оптимізація: Особливо важливо оптимізувати алгоритми і забезпечувати ефективне використання ресурсів, щоб мінімізувати ймовірність вичерпання пам'яті або перевантаження процесора.

Робота з великими числами завжди передбачає певні обчислювальні складнощі, і врахування цих аспектів критично важливе для розроблення ефективних і надійних застосунків, особливо на 32-розрядних системах.

Python(int)

Хоча бібліотеки багаторозрядної арифметики, які інкорпоровані в популярні програмні мови, здебільшого є незалежними від конкретної архітектури процесора і обсягу оперативної пам’яті, деякі особливості роботи цих бібліотек на 32-розрядних системах із обмеженою оперативною пам'яттю (до 8 ГБ) можуть бути корисними для розгляду.
Давайте розглянемо Python і вбудовану бібліотеку для роботи з довгими числами int
У Python 3 вбудований тип int може представляти цілі числа будь-якої довжини, обмежені тільки доступною пам’яттю системи.

Обробка великих чисел:

- Python динамічно виділяє пам'ять під цілі числа, тому ви можете працювати із дуже великими числами навіть на 32-розрядній системі, при умові, що у вас достатньо RAM.
- Операції над великими числами можуть бути повільнішими, ніж операції над фіксованою розрядністю, які оптимізовані на апаратному рівні.

Оптимізація пам’яті:

- Python автоматично управляє пам’яттю, але у випадку роботи з великими об’ємами даних потрібно бути уважними, щоб уникнути витікання пам'яті або великої затратності пам'яті.
- Використання бібліотек, як-от NumPy, може забезпечити більш ефективне використання пам'яті та прискорення деяких операцій завдяки векторизації, але це не застосовується до довільної точності.

Integer у Python це структура C, визначена таким чином:

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};
```

PyObject_VAR_HEAD - це макрос, він розкривається в PyVarObject, який має таку структуру:

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

Це означає, що ціле число, подібно до кортежу або списку, має змінну довжину, і це перший крок до розуміння того, як Python може підтримувати роботу з гігантськими числами. Після розкриття макроса _longobject можна буде розглядати як:

```
struct _longobject {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
    digit ob_digit[1];
};
```

ob_digit - це статично алоційований масив одиничної довжини типу digit (typedef для uint32_t). Оскільки це масив, ob_digit насамперед є покажчиком на число, і, отже, за необхідності він може бути збільшений за допомогою функції malloc до будь-якої довжини. Таким чином python може представляти й обробляти дуже довгі числа.

ob_size зберігає кількість елементів в ob_digit. Python перевизначає і потім використовує значення ob_size для визначення фактичної кількості елементів, що містяться в масиві, щоб підвищити ефективність виділення пам'яті масиву ob_digit.

Зберігання:
Замість того, щоб зберігати тільки одну десяткову цифру в кожному елементі масиву ob_digit, Python перетворює числа із системи числення з основою 10 на числа в системі з основою 230 і викликає кожний елемент, як цифру, значення якої коливається від 0 до 230 - 1.

У шістнадцятковій системі числення, основа 16 ~ 24 означає, що кожна "цифра" шістнадцяткового числа коливається від 0 до 15 у десятковій системі числення. У Python аналогічно, "число" з основою 230, що означає, що число коливатиметься від 0 до 230 - 1 = 1073741823 у десятковій системі числення.

Таким чином, Python ефективно використовує майже весь виділений простір у 32 біти на одну цифру, економить ресурси і все ще виконує прості операції, як-от додавання і віднімання на рівні математики початкової школи.

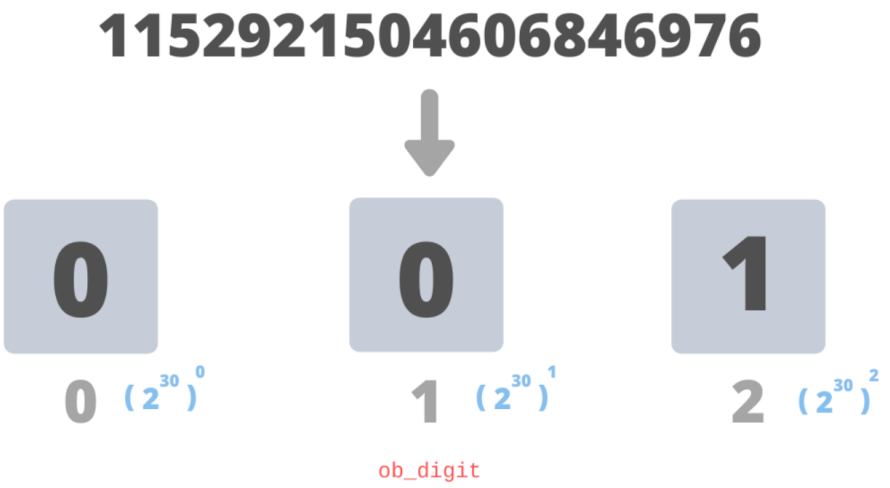
Як уже згадувалося, для Python числа представлені в системі з основою 2^30, тобто якщо ви конвертуєте 1152921504606846976 у систему числення з основою 2^30, ви отримаєте 100.

1152921504606846976 = 1 * (2^30)^2 + 0 * (2^30)^1 + 0 * (2^30)^0

Оскільки ob_digit першим зберігає найменш значущу цифру, воно зберігається як 001 у вигляді трьох цифр.

Структура _longobject для цього значення буде містити:

- `ob_size` как 3
- `ob_digit` как [0, 0, 1]



Python 2

Максимальне значення типу Int у Python 2 за замовчуванням - 65535, все, що перевищує це значення, буде довгим

Наприклад:

>> print type(65535)
<тип 'int'>
>>> print type(65536*65536)
<тип 'long'>>.

У Python 3 тип даних long було вилучено, і всі цілочисельні значення обробляються класом Int. Розмір Int за замовчуванням буде залежати від архітектури вашого процесора.

Наприклад:

32-розрядні системи, тип даних за замовчуванням для цілих чисел буде 'Int32'
64-розрядні системи: тип даних за замовчуванням для цілих чисел буде 'Int64'.
Мінімальні/максимальні значення для кожного типу можна знайти нижче:

Int8: [-128,127]
Int16: [-32768,32767]

Int32: [-2147483648,2147483647]
Int64: [-9223372036854775808,9223372036854775807]
Int128:
[-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
UInt8: [0,255]
UInt16: [0,65535]
UInt32: [0,4294967295]
UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]

Якщо розмір вашого типу Int перевищує вказані вище межі, python автоматично змінить його тип і виділить більше пам'яті, щоб впоратися зі збільшенням мінімального/максимального значення. Якщо у Python 2 він перетворювався на "long", то тепер він просто перетворюється на наступний розмір Int.

Приклад: Якщо ви використовуєте 32-бітну операційну систему, максимальне значення типу Int за замовчуванням буде 2147483647. Якщо присвоєно значення 2147483648 або більше, тип буде змінено на Int64.

Ви можете використовувати вбудовану функцію bin(), щоб конвертувати ціле число (включно з BigInteger) у рядок, що представляє двійкове число:
big_int = 2**100
binary_representation = bin(big_int)

Для конвертації рядка, що представляє двійкове число, назад у ціле число використовується функція int() із зазначенням другого аргументу як 2:
original_int = int(binary_representation, 2)

Бітові операції, такі як І (&), АБО (|), виключне АБО (^), заперечення (~), зсув уліво (<<) і зсув управо (>>) можна виконувати прямо на цілих числах.

Щоб інвертувати двійкове представлення великого числа (bigint) у Python, ви можете використати бітову операцію NOT, яка позначається тильдою ~.

Однак, важливо пам'ятати, що в Python числа представляються в додатковому коді, і тому інверсія бітів числа x призведе до отримання числа -x-1. Якщо x позитивний, результат буде негативним, і навпаки.

```
big_int = 2**10  # 1024 в десятичной системе
print(bin(big_int))  # 0b10000000000 в двоичной системе

inverse_big_int = ~big_int
print(bin(inverse_big_int))  # -0b10000000001 в двоичной системе
```

Якщо ви хочете інвертувати певні біти числа, ви можете використати XOR (^) з бітовою маскою, в якій встановлено в 1 ті біти, які ви хочете інвертувати. Цей метод дасть вам змогу інвертувати біти без зміни знака результату.

```
big_int = 2**10  # Пример большого числа
mask = 2**10 - 1  # Маска с 10 битами установленными в 1: 0b1111111111
inverse_big_int = big_int ^ mask
```

Не могла не добавить это сюда)