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

Кафедра математичних методів захисту інформації

Звіт

з комп’ютерного практикуму

«Вибір та реалізація базових фреймворків та бібліотек»

з кредитного модуля

«Сучасні алгебраїчні криптосистеми»

Варіант 1А

Виконала студентка

Групи ФІ-52мн

Остаповець Олеся

Київ 2025

1. Всут

Багаторозрядна арифметика – це розширення стандартних арифметичних операцій (додавання, віднімання, множення, ділення) для роботи з числами, що перевищують максимальну довжину слова (кількість бітів, які процесор може обробити за один раз), що дозволяє програмувати виконання цих операцій над дуже великими числами які обмежені лише доступною пам'яттю комп'ютера, а не апаратними обмеженнями. Це ключова техніка в криптографії, наукових обчисленнях та інших галузях, де потрібна висока точність.

Для дослідження було обрано вбудовану багаторозрядну арифметику Python: тип int (arbitrary-precision integer)

2. Опис функцій

Додавання: c = a + b
Вхід: a, b ∈ ℤ (int)
Вихід: c ∈ ℤ (int)
Алгоритм: класичний алгоритм – посимвольне додавання з урахуванням переносу
Складність**:** O(n), де n – кількість розрядів.

Віднімання: c = a – b
Вхід: a, b ∈ ℤ (int)
Вихід: c ∈ ℤ (int)
Алгоритм: класичний алгоритм – посимвольне віднімання з урахуванням позики
Складність**:** O(n).

Множення: c = a * b
Вхід: a, b ∈ ℤ (int)
Вихід: c ∈ ℤ (int)
Алгоритм: Karatsuba (для великих чисел) або класичний алгоритм довгого множення
Складність**:** для Karatsuba O(n^1.585), класичний алгоритм – $O(n^2)$

Ділення: q = a // b, r = a % b
Вхід: a, b ∈ ℤ (int)
Вихід: q, r ∈ ℤ (int), причому a = q·b + r
Алгоритм: Knuth D(для великих чисел) або класичний алгоритм довгого ділення
Складність: для Knuth D $O(n^2)$, класичний алгоритм – O(n).

Модульне піднесення до степеня : pow(a, e, m)
Вхід: a, e, m ∈ ℤ, m ≠ 0
Вихід: a^e mod m
Алгоритм: Square-and-Multiply (метод бінарного піднесення)

1. Представити показник b у двійковій системі
2. Ініціалізувати result = 1
3. Для кожного біта (справа наліво):

- Якщо біт = 1: result = (result × base) mod m
- base = ($base^2$) mod m

```python
def modular_exponentiation(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        exp = exp >> 1
        base = (base * base) % mod
    return result
```

Складність: O(log b × M(log m)), де M(log m) – складність множення

Знаходження НСД: gcd(a, b)
Вхід: a, b ∈ ℤ
Вихід: g = gcd(a, b)
Алгоритм: алгоритм Евкліда
Складність: O(log n).

Знаходження оберненого елемента по модулю: inv = pow(a, -1, m)
Вхід: a, m ∈ ℤ (int), m ≠ 0, gcd(a, m) = 1
Вихід: inv ∈ ℤ (int), причому (a·inv) mod m = 1
Алгоритм: розширений алгоритм Евкліда
Складність: O(log n)

Алгоритм Karatsuba.
Алгоритм рекурсивно ділить числа на дві половини й рекурсивно перемножує ці частини за допомогою спеціальних формул, що обчислюють проміжні значення. Кінцевий результат отримують шляхом додавання цих проміжних величин.

$$(a + bx)(c + dx) = ac + ((a + b)(c + d) - ac - bd)x + bdx^2.$$

Knuth D – це узагальнення довгого ділення для багаторозрядних чисел, поданих як масив “слів” у основі $B = 2^k$. Спочатку виконується **нормалізація**: ділене й дільник масштабуються (або зсуваються), щоб старше слово дільника було достатньо великим, що підвищує точність оцінки. Далі по черзі обчислюються цифри частки: на кожному кроці оцінюється поточна цифра частки за старшими словами діленого і дільника, після чого виконується віднімання добутку дільника на цю цифру. Якщо оцінка виявилася завищеною, цифра частки коригується (зменшується), а проміжний результат виправляється додаванням дільника. Після завершення обчислень виконується денормалізація остачі.

3. Результати тестування

Приклад 1: Базові операції
a = 12345678901234567890123456789O
b = 98765432109876543210987654321O
Додавання: 1111111110111111111011111111100
Віднімання: -864197532086419753208641975320
Множення:
121932631137021795226185032590122949089020085301974335557920900

Приклад 2: Модульне піднесення до степеня
base = 5
exp = 13
mod = 19
Результат: $5^{13} \bmod 19 = 11$
Перевірка: $5^{13} = 1220703125$
$1220703125 \bmod 19 = 11$

Приклад 3: Алгоритм Евкліда
a = 48
b = 18
НСД(48, 18) = 6
Розширений алгоритм Евкліда:
$48 \times (-1) + 18 \times 3 = 6$

Приклад 4: Обернений елемент
a = 17
m = 3120
$17^{-1} \bmod 3120 = 2753$
Перевірка: $(17 \times 2753) \bmod 3120 = 1$

Таблиця 1: Час виконання базових операцій (мс)

Розмір (біт)	Додавання	Множення	Ділення	НСД
1024	0.0002	0.0018	0.0002	0.0058
2048	0.0002	0.0061	0.0002	0.0146
4096	0.0005	0.0199	0.0005	0.0526

Таблиця 2: Модульне піднесення до степеня (exp=65537)

Розмір модуля (біт)	Час (мс)
1024	0.0917
2048	0.2294
4096	0.8514

Висновки

У ході виконання лабораторної роботи було досліджено вбудовану багаторозрядну арифметику Python (тип int), яка забезпечує виконання базових операцій над цілими числами довільної точності. Було розглянуто основні операції (додавання, віднімання, множення, ділення, модульне піднесення до степеня, НСД та знаходження оберненого елемента) й наведено приклади їх застосування, що підтвердило коректність роботи алгоритмів на контрольних даних.

За результатами вимірювання продуктивності встановлено, що час виконання операцій зростає зі збільшенням розрядності чисел. Додавання та ділення для 1024, 2048, 4096 біт виконуються дуже швидко і збільшуються незначно (від 0.0002 до 0.0005 мс), що відповідає їх лінійній природі в реалізації. Натомість множення демонструє суттєвіший ріст часу (0.0018 → 0.0199 мс), що узгоджується з використанням ефективніших методів для великих чисел (зокрема Karatsuba) та загальною вищою складністю множення порівняно з додаванням/відніманням. Обчислення НСД також помітно сповільнюється зі зростанням розміру операндів (0.0058 → 0.0526 мс), що є очікуваним через багаторазові ітерації алгоритму Евкліда. Найбільш ресурсомісткою операцією виявилось модульне піднесення до степеня: час зріс з 0.0917 мс для 1024 біт до 0.8514 мс для 4096 біт, оскільки воно виконує серію модульних множень за методом бінарного піднесення. Отже, Python int є придатним і ефективним інструментом для задач, де потрібні операції над великими числами, зокрема для криптографічних застосувань.