ВІСНИК науки та освіти
ISSN 2786-6165 (ONLINE) №4(46) 2026
Bulletin of Science and Education

УДК 004.021

https://doi.org/10.52058/2786-6165-2026-4(46)-4711-4727

Тютюнник Оксана Іванівна кандидат педагогічних наук, доцент, кафедри вищої математики Вінницького національного технічного університету, Вінниця, https://orcid.org/0000-0002-8544-4246

Клєопа Ірина Анатоліївна доктор філософії, доцент, кафедри вищої математики Вінницького національного технічного університету, Вінниця, https://orcid.org/0000-0001-8408-6515

АЛГОРИТМ BABY-STEP GIANT-STEP ЯК ІНСТРУМЕНТ ОБЧИСЛЕННЯ ДИСКРЕТНОГО ЛОГАРИФМА В ТЕОРІЇ ЧИСЕЛ ТА ВИЩІЙ МАТЕМАТИЦІ

Анотація. Задача дискретного логарифмування є однією з фундаментальних математичних проблем, на яких ґрунтується стійкість сучасних систем асиметричної криптографії, зокрема протоколу обміну ключами Діффі-Геллмана, криптосистеми Ель-Гамаля та криптографії на еліптичних кривих. Її математична суть полягає у знаходженні невідомого показника степеня у рівнянні $g^x \equiv h \ (mod\ p)$, де g (основа), h (результат) та p (модуль) є відомими цілими числами.

Для вирішення цієї задачі класичний метод повного перебору (brute-force) є вкрай неефективним і практично неможливим для великих значень модуля p. Алгоритм «baby-step giant-step» (відомий також як алгоритм Шенкса), запропонований Даніелем Шенксом у 1971 році, дозволяє суттєво оптимізувати цей процес. Цей метод базується на парадигмі «зустрічного кроку» (meet-in-the-middle) і дозволяє зменшити часову обчислювальну складність пошуку з $O(p)$ до $O(\sqrt{p})$ вимагаючи при цьому обсягу пам'яті $O(\sqrt{p})$ для збереження проміжних результатів.

У даній роботі розглядається програмна реалізація алгоритму Шенкса з використанням мови програмування Python. Вибір даної мови зумовлений її високорівневою архітектурою, підтримкою довгих цілих чисел (що критично для криптографії) та наявністю ефективних вбудованих структур даних. Зокрема, використання словників (хеш-таблиць) дозволяє забезпечити миттєвий пошук колізій між двома обчисленими множинами значень — «малими» (baby - steps) та «великими» (giant-steps) кроками. Розроблена реалізація є не лише наочним інструментом для глибокого вивчення математичних основ криптоаналізу, а й демонструє практичну вразливість

певних параметрів криптосистем, що є важливим етапом у підготовці фахівців з інформаційної безпеки.

Ключові слова: мова програмування Python, дискретний логарифм, алгоритм Шенкса, baby-step giant-step, асиметрична криптографія, обчислювальна складність.

Tyutyunnyk Oksana Ivanivna PhD in Pedagogical Sciences, Associate Professor, Department of Higher Mathematics, Vinnytsia National Technical University, Vinnytsia, https://orcid.org/0000-0002-8544-4246

Klieopa Iryna Anatoliivna PhD in Philosophy, Associate Professor, Department of Higher Mathematics, Vinnytsia National Technical University, Vinnytsia, https://orcid.org/0000-0001-8408-6515

BABY-STEP GIANT-STEP ALGORITHM AS A TOOL FOR COMPUTING THE DISCRETE LOGARITHM IN NUMBER THEORY AND HIGHER MATHEMATICS

Abstract. The discrete logarithm problem is one of the fundamental mathematical challenges upon which the security of modern asymmetric cryptographic systems is built, including the Diffie-Hellman key exchange protocol, the ElGamal cryptosystem, and elliptic curve cryptography. Its mathematical essence lies in finding an unknown exponent in the equation $g^x \equiv h \ (mod\ p)$, where g (the base), h (the result), and p (the modulus) are known integers.

To solve this problem, the classical brute-force method is extremely inefficient and practically impossible for large values of the modulus p. The "baby-step giant-step" algorithm (also known as Shanks's algorithm), proposed by Daniel Shanks in 1971, allows for significant optimization of this process. This method is based on the "meet-in-the-middle" paradigm and allows for a reduction in computational search complexity from $O(p)$ to $O(\sqrt{p})$while requiring $O(\sqrt{p})$ memory to store intermediate results.

This paper examines a software implementation of Shanks's algorithm using the Python programming language. The choice of this language is driven by its high-level architecture, support for arbitrary-precision integers (which is critical for cryptography), and the availability of efficient built-in data structures. In particular, the use of dictionaries (hash tables) ensures an instantaneous search for collisions between two calculated sets of values - the "baby steps" and "giant steps." The developed implementation serves not only as a visual tool for the in-depth study of the mathematical foundations of cryptanalysis but also

demonstrates the practical vulnerability of certain cryptosystem parameters, which is an important stage in the training of information security specialists.

Keywords: Python programming language, discrete logarithm, Shanks's algorithm, baby-step giant-step, asymmetric cryptography, computational complexity.

Постановка проблеми. Задача обчислення дискретного логарифма є однією з фундаментальних проблем теорії чисел та вищої математики, що має важливе значення у сучасній криптографії та системах захисту інформації. Незважаючи на існування різноманітних алгоритмів, задача дискретного логарифмування залишається обчислювально складною, особливо для великих значень параметрів, що забезпечує надійність багатьох криптографічних протоколів.

В освітньому процесі вивчення дискретного логарифма часто обмежується теоретичними аспектами без належної уваги до алгоритмічної реалізації та обчислювального аналізу. Це призводить до розриву між абстрактними математичними поняттями та їх практичним застосуванням, особливо у підготовці студентів технічних та ІТ-спеціальностей.

У зв’язку з цим виникає необхідність дослідження ефективних алгоритмічних підходів до розв’язування задачі дискретного логарифма, зокрема алгоритму baby-step giant-step, а також його інтеграції у процес навчання вищої математики. Особливу увагу доцільно приділити програмній реалізації таких алгоритмів із використанням сучасних засобів програмування та оцінюванню їх обчислювальної складності й ефективності на основі експериментальних досліджень. Незважаючи на наявність грунтовної теоретичної бази щодо математичних принципів задачі дискретного логарифмування, питанням практичної імплементації алгоритму Шенкса із застосуванням вбудованих структур даних сучасних мов програмування приділяється недостатньо уваги. Зокрема, потребує глибшого практичного дослідження ефективність використання словників Python як хеш-таблиць, що дозволяють забезпечити пошук колізій за константний час $O(1)$.

Аналіз останніх досліджень та публікацій. Проблема дискретного логарифмування є однією з фундаментальних математичних проблем, що лежать в основі стійкості сучасних систем асиметричної криптографії. Теоретичним та практичним аспектам цієї задачі присвячено праці багатьох вітчизняних та закордонних науковців.

Фундаментальні основи прикладної криптології та застосування криптографічних протоколів, таких як протокол обміну ключами Діффі-Геллмана, криптосистема Ель-Гамаля та криптографія на еліптичних

кривих, детально висвітлені у працях І. Д. Горбенка та Ю. І. Горбенка [1]. Розширений аналіз теоретичної бази цих алгоритмів також представлений у зарубіжних дослідженнях D. R. Stinson та M. Paterson [2]. У цих роботах доводиться, що класифікаційний метод повного перебору (brute-force) для знаходження невідомого показника степеня є практично неможливим для великих значень модуля P.

Основоположним для вирішення задачі прискореного пошуку дискретного логарифма є дослідження Даніеля Шенкса (1971 р.), який запропонував алгоритм «baby-step giant-step» [3]. Цей метод, що базується на парадигмі «зустрічного кроку» (meet-in-the-middle), дозволив якісно змінити підхід до криптоаналізу. Загальні аспекти алгоритмічної складності, оптимізації структур даних для подібних задач та пошуку колізій грунтовно розглядаються у класичній праці Т. Кормена щодо побудови алгоритмів [4].

Сучасні методи дискретного логарифмування в криптографії та оцінка їхньої обчислювальної складності є предметом актуальних досліджень О. М. Жданова.

Разом з тим, для ефективної програмної реалізації алгоритму Шенкса особливого значення набуває використання сучасних високорівневих мов програмування, інструментарій яких дозволяє оптимізувати роботу з хеш-таблицями та довгою арифметикою [5]. Зокрема, стандартні бібліотеки мови Python надають широкі можливості.

Метою статті є комплексне дослідження математичних принципів задачі дискретного логарифмування та обґрунтування ефективності її практичного вирішення за допомогою алгоритму «baby-step giant-step» (алгоритму Шенкса), реалізованого на базі оптимізованих вбудованих структур даних мови програмування Python.

Для досягнення поставленої мети у дослідженні сформульовано та вирішено такі **основні завдання**:

Формалізувати математичну модель задачі дискретного логарифмування та теоретичну базу парадигми «зустрічного кроку» (meet-in-the-middle) у контексті криптоаналізу.

Розробити та описати програмну імплементацію алгоритму Шенкса мовою Python, деталізувавши механізм швидкого пошуку колізій між множинами «малих» та «великих» кроків за допомогою хеш-таблиць (словників).

Здійснити порівняльний емпіричний аналіз обчислювальної складності розробленого методу $O(\sqrt{p})$, із класичним алгоритмом повного перебор $(O(p))$, на прикладі великих простих чисел модулів.

Оцінити баланс між витратами процесорного часу та обсягом необхідної оперативної пам'яті (time-memory trade-off) визначивши

практичну цінність алгоритму для оцінки надійності асиметричних криптосистем

Виклад основного матеріалу дослідження.

Задача дискретного логарифмування полягає у знаходженні такого цілого числа x, для якого виконується рівність:

$$a^x \equiv b(\mathrm{mod} p),$$

де a— генератор мультиплікативної групи за модулем p, b— заданий елемент цієї групи, p— просте число. Дана задача є оберненою до задачі піднесення до степеня за модулем і широко використовується у криптографічних протоколах.

Одним із ефективних алгоритмів розв'язування цієї задачі є алгоритм baby-step giant-step, запропонований Д. Шенксом. Його ідея базується на представленні шуканого показника xу вигляді:

$$x = i \cdot m + j,$$

де $m \approx \lceil\sqrt{p}\rceil$, $i, j \in \mathbb{Z}$.

З метою оцінки ефективності алгоритму baby-step giant-step було проведено обчислювальний експеримент, у якому порівнювався час виконання цього алгоритму з методом повного перебору (brute-force) при розв'язуванні задачі дискретного логарифмування.

У межах дослідження розглядається один із класичних підходів до обчислення дискретного логарифма — алгоритм «зустріч посередині» Baby-step Giant-step (BSGS), який також відомий як алгоритм Крока немовляти — Гігантського кроку.

Його ефективність базується на комбінуванні попереднього обчислення значень (baby steps) та подальшого ітераційного пошуку (giant steps), що дозволяє суттєво зменшити обчислювальну складність задачі дискретного логарифмування.

Для кращого розуміння принципу роботи алгоритму baby-step giant-step доцільно подати його покрокову візуалізацію. Алгоритм базується на ідеї поділу задачі дискретного логарифмування на два етапи: попереднє обчислення множини «малих кроків» та послідовний перебір «гігантських кроків» до моменту знаходження співпадіння.

На рисунку 1 схематично показано процес знаходження розв'язку для прикладу $2^x \equiv 8(\mathrm{mod} 11)$.

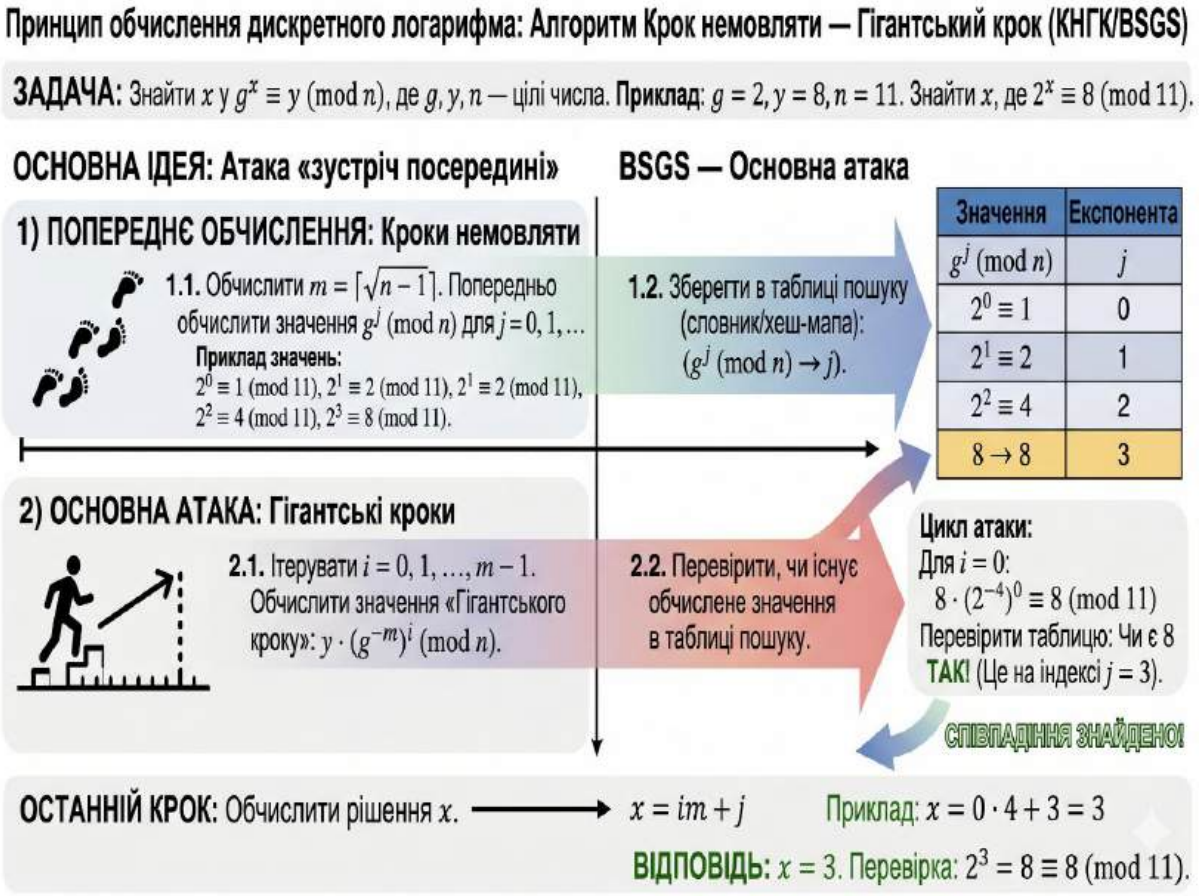

Рисунок 1. Принцип роботи алгоритму baby-step giant-step для знаходження дискретного логарифма

Як видно з рисунка 1, на першому етапі формуються значення $g^j(\mathrm{mod}n)$, які зберігаються у таблиці пошуку. На другому етапі обчислюються значення виду:

$$y \cdot (g^{-m})^i(\mathrm{mod}n)$$

де здійснюється перевірка наявності співпадіння з попередньо сформованою таблицею. Після виявлення однакових значень шуканий показник визначається за формулою Д. Шенксом.

Такий підхід дозволяє суттєво зменшити складність задачі порівняно з повним перебором.

Для проведення дослідження алгоритм було реалізовано мовою програмування **Python**, яка є зручною для математичного моделювання, експериментальних досліджень та швидкого створення програмних прототипів. Завдяки вбудованій підтримці роботи з цілими числами великої розрядності Python ефективно підходить для реалізації задач криптографічного спрямування [6].

Алгоритм складається з двох основних етапів:

1. **Baby-step (малий крок):** обчислення та збереження значень:

$$a^j \bmod p, j = 0,1, \dots, m-1.$$

Giant-step (великий крок): обчислення значень :

$$b \cdot (a^{-m})^i \bmod p, i = 0,1, \dots, m-1,$$

та пошук співпадіння з попередньо обчисленими значеннями.

У разі знаходження збігу визначається значення $x = i \cdot m + j$.

Алгоритм має часову складність $O(\sqrt{p})$та просторову складність $O(\sqrt{p})$, що значно ефективніше порівняно з повним перебором.

З метою дослідження ефективності алгоритму було реалізовано програмну модель мовою Python. Для обчислень використовувалися стандартні арифметичні операції та вбудовані функції роботи з великими цілими числами.

Розроблена програма дозволяє:

- знаходити дискретний логарифм для заданих параметрів;
- порівнювати швидкодію алгоритму baby-step giant-step та brute-force;
- оцінювати вплив розміру модуля на складність обчислень;
- проводити експериментальні дослідження у сфері криптоаналізу.

Нижче наведено фрагмент реалізації алгоритму [7-9]:

1. *Підключення бібліотек*

```
import math
import time
```

math - надає доступ до базових математичних функцій.

time - стандартний модуль Python для роботи з часом.

2. *Підготовка інструментів для обчислення (Алгоритм Шенкса)*

```
def baby_step_giant_step(g, h, p):
    m = math.ceil(math.sqrt(p))
```

Визначення параметра m: Алгоритм починається з обчислення оптимального розміру кроку m, який дорівнює $\lceil \sqrt{p} \rceil$. Це забезпечує оптимальний баланс між кількістю ітерацій та обсягом необхідної оперативної пам'яті.

```
    # 1. Baby steps
    baby_steps = {}
    for j in range(m):
    val = pow(g, j, p)
    baby_steps[val] = j
```

Фаза "Малих кроків" (Baby steps): Створюється порожній словник baby_steps, який виконує роль хеш-таблиці. У циклі обчислюються

значення $g^j \ (mod\ p)$ для j від 0 до m-1 . Отримане значення стає ключем у словнику, а показник j — його значенням. Завдяки внутрішній архітектурі словників у Python, пошук елементів у такій структурі в подальшому відбуватиметься за константний час $O(1)$.

```
c = pow(g, (p - 2) * m, p)
```

Обчислення множника для великих кроків: Знаходиться обернене значення $g^{-m}(mod\ p)$. Для уникнення складних алгоритмів пошуку оберненого елемента застосовується Мала теорема Ферма: оскільки p — просте число, $g^{-1} \equiv g^{p-2}\ (mod\ p)$. Відповідно, $g^{-m} \equiv (g^{p-2})^m\ (mod\ p)$. Вбудована функція pow(base, exp, mod) у Python дозволяє виконати це модульне піднесення до степеня високоефективно.

```
# 2. Giant steps та пошук колізій
for i in range(m):
 # Обчислення h * (g^-m)^i mod p
gamma = (h * pow(c, i, p)) % p
 # Перевірка наявності колізії у хеш-таблиці
 if gamma in baby_steps:
return i * m + baby_steps[gamma]
 return None
```

Фаза "Великих кроків" (Giant steps) та пошук колізій: Програма ітерує змінну i від 0 до m-1 , щоразу обчислюючи нове значення gamma за формулою $h \cdot (g^{-m})^i\ (mod\ p)$. Далі відбувається миттєва перевірка: якщо обчислене значення вже існує як ключ у словнику baby_steps (тобто знайдено колізію), алгоритм успішно завершується. Знайдений дискретний логарифм повертається за формулою $x = i \cdot m + j$.

3. *Алгоритм повного перебору (для порівняльного аналізу)*

```
def brute_force(g, h, p):
 Знаходить x методом повного перебору (brute-force).
for x in range(p):
 if pow(g, x, p) == h:
 return x
return None
```

Ця допоміжна функція реалізує тривіальний лінійний пошук (brute-force). Вона інтегрована в програму виключно для емпіричного підтвердження теоретичної складності $O_{(p)}$і наочної демонстрації переваг оптимізованого методу Шенкса.

4. *Функція main (Ініціалізація та тестування)*

```
def main():
 # 1. Ініціалізація параметрів
p = xxx # Велике просте число
```

```
g = xxx # Основа (відоме ціле число)
x_secret = xxx
h = pow(g, x_secret, p)
print(f"Пошук x у рівнянні: {g}^x ≡ {h} (mod {p})")
# 2. Виконання алгоритму Шенкса та аналіз часу
start_time = time.time()
result_bsgs = baby_step_giant_step(g, h, p)
end_time = time.time()
```

Збір статистики: Програма фіксує точний час системи до та після виклику функції baby_step_giant_step, що дозволяє розрахувати реальні витрати процесорного часу. Аналогічний блок коду застосовується і для виклику функції brute_force, після чого отримані результати виводяться у консоль для проведення порівняльного аналізу.

```
print(f"\n[Baby-step Giant-step]")
print(f"Знайдений x: {result_bsgs}")
print(f"Час виконання: {end_time - start_time:.6f} секунд")
3. Виконання алгоритму повного перебору (опціонально для порівняння)
print(f"\n[Brute-force (Повний перебір)]")
start_time_bf = time.time()
result_bf = brute_force(g, h, p)
end_time_bf = time.time()
print(f"Знайдений x: {result_bf}")
print(f"Час виконання: {end_time_bf - start_time_bf:.6f} секунд")
if __name__ == "__main__":
main()
```

Для практичної перевірки ефективності алгоритму baby-step giant-step було проведено серію обчислювальних експериментів та порівняно час виконання з методом повного перебору. Результат роботи програми можемо побачити в табл. 1.

Таблиця 1

Порівняння часу виконання алгоритмів baby-step giant-step та brute-force

Пошук x у рівнянні: 16807^x ≡ 584118435 (mod 2147483647) [Baby-step Giant-step] Знайдений x: 15000000 Час виконання: 0.079509 секунд [Brute-force (Повний перебір)] Знайдений x: 15000000 Час виконання: 46.784104 секунд	Пошук x у рівнянні: 2^x ≡ 81 (mod 997) [Baby-step Giant-step] Знайдений x: 124 Час виконання: 0.000019 секунд [Brute-force (Повний перебір)] Знайдений x: 124 Час виконання: 0.000024 секунд
Пошук x у рівнянні: 6320430^x ≡ 963936121928 (mod 999999999989) [Baby-step Giant-step] Знайдений x: 9808358 Час виконання: 3.258091 секунд [Brute-force (Повний перебір)] Знайдений x: 9808358 Час виконання: 72.603084 секунд	Пошук x у рівнянні: 546198^x ≡ 4695203303 (mod 10000000019) [Baby-step Giant-step] Знайдений x: 874977 Час виконання: 0.255769 секунд [Brute-force (Повний перебір)] Знайдений x: 874977 Час виконання: 2.535251 секунд
Пошук x у рівнянні: 7^x ≡ 510444705 (mod 2147483647) [Baby-step Giant-step] Знайдений x: 123456789 Час виконання: 0.073093 секунд [Brute-force (Повний перебір)] Знайдений x: 123456789 Час виконання: 403.586611 секунд	Пошук x у рівнянні: 2^x ≡ 3283003 (mod 10000019) [Baby-step Giant-step] Знайдений x: 854321 Час виконання: 0.001648 секунд [Brute-force (Повний перебір)] Знайдений x: 854321 Час виконання: 0.572470 секунд
Пошук x у рівнянні: 5^x ≡ 36437896 (mod 40000003) [Baby-step Giant-step] Знайдений x: 35000000 Час виконання: 0.006542 секунд [Brute-force (Повний перебір)] Знайдений x: 35000000 Час виконання: 29.881109 секунд	Пошук x у рівнянні: 85735^x ≡ 342150 (mod 524827) [Baby-step Giant-step] Знайдений x: 154960 Час виконання: 0.000631 секунд [Brute-force (Повний перебір)] Знайдений x: 154960 Час виконання: 0.137538 секунд

Використання алгоритму BSGS у межах роботи демонструє практичну реалізацію задачі дискретного логарифмування, яка є базовою для сучасної криптографії з відкритим ключем. Його застосування дозволяє проілюструвати експоненціальне зменшення обчислювальної складності порівняно з наївним перебором та є важливим елементом аналізу криптографічної стійкості [10].

У ході експерименту для різних значень модуля p та відповідних параметрів aі b обчислювався дискретний логарифм двома способами:

- за допомогою алгоритму baby-step giant-step;

- методом повного перебору можливих значень показника.

Для кожного набору вхідних даних фіксувався час виконання обох алгоритмів. Отримані результати було представлено у вигляді діаграми (рисунок), де:

- зелений колір відповідає алгоритму baby-step giant-step;
- синій — методу повного перебору.

З огляду на значний розкид значень часу виконання, графік побудовано в логарифмічній шкалі, що дозволяє наочно порівняти ефективність алгоритмів навіть при великих значеннях параметрів.

Рисунок 2. Порівняння часу виконання алгоритмів baby-step giant-step та brute-force для тестового прикладу

Результати експерименту показали, що алгоритм baby-step giant-step працює значно швидше за метод повного перебору. Зі зростанням розміру вхідних даних різниця у часі виконання стає більш суттєвою: якщо для малих значень параметрів обидва методи демонструють відносно близькі результати, то для великих значень pчас виконання повного перебору зростає експоненційно, тоді як для алгоритму baby-step giant-step — пропорційно квадратному кореню від p.

Проведений експеримент підтверджує теоретичні оцінки складності алгоритму та демонструє його доцільність для практичного застосування

при розв'язуванні задач дискретного логарифмування. Крім того, використання програмної реалізації сприяє кращому розумінню студентами алгоритмічних аспектів вищої математики та їх застосування у криптографії.

Для узагальнення отриманих результатів було сформовано таблицю, у якій наведено час виконання алгоритму baby-step giant-step, методу повного перебору (brute-force), а також різницю між ними.

Таблиця 2

Зведена різниця у часі

Baby-step Giant-step (c)	Brute-force (c)	Різниця (c)
0,079509	46,784104	46,704595
0,000019	0,000024	0,000005
3,258091	72,603004	69,344913
0,255769	2,535251	2,279482
0,073093	403,586611	403,513518
0,001648	0,57247	0,570822
0,006542	29,881109	29,874567
0,000631	0,137538	0,136907
0,011751	153,22101	153,209259
0,100711	25,468046	25,367335

Аналіз даних показує, що в усіх проведених експериментах алгоритм baby-step giant-step демонструє значно менший час виконання порівняно з методом повного перебору. Різниця у часі є суттєвою і в окремих випадках досягає сотень секунд. Зокрема, найбільше перевищення часу спостерігається для великих значень параметрів, де метод повного перебору потребує понад 400 секунд, тоді як алгоритм baby-step giant-step виконується менш ніж за 0,1 секунди.

Навіть для відносно невеликих значень вхідних даних перевага алгоритму baby-step giant-step залишається помітною, хоча різниця у часі є меншою. Це пояснюється тим, що метод повного перебору має лінійну залежність від величини показника, тоді як алгоритм baby-step giant-step працює значно ефективніше завдяки використанню підходу «розбиття задачі», що забезпечує складність порядку $O(\sqrt{p})$.

Отримані результати підтверджують теоретичні оцінки ефективності алгоритму та свідчать про його практичну доцільність для розв'язування задач дискретного логарифмування. Крім того, значна різниця у часі виконання підкреслює обмеженість використання методу повного перебору для задач із великими параметрами.

Обговорення результатів експерименту та практичні наслідки.

Отримані в експериментальній частині результати та побудована логарифмічно зведена діаграма порівняння свідчать про ефективність досліджуваного підходу в межах заданих параметрів. Проте для повноцінної інтерпретації результатів необхідно розглянути їх у контексті реальних криптографічних систем та практичних сценаріїв застосування [11].

Практичний вектор атак: слабкі ключі та роль OSINT. Сучасні системи асиметричної криптографії з довжиною ключа 2048 біт і більше забезпечують високий рівень стійкості до класичних математичних атак. Водночас у реальних умовах безпека часто знижується не через теоретичні обмеження алгоритмів, а через помилки реалізації, застарілі протоколи та слабкі криптографічні параметри.

У таких умовах алгоритм baby-step giant-step (алгоритм Шенкса) набуває практичного значення як інструмент криптоаналізу для випадків із малими значеннями модуля p.

Вразливі системи та IoT-середовище. Найбільш уразливим сегментом є пристрої Інтернету речей (IoT): маршрутизатори, IP-камери, датчики розумного дому та промислові контролери попередніх поколінь. Через обмежені апаратні ресурси виробники таких систем часто змушені використовувати спрощені підходи до криптографічного захисту, зокрема:

- застосування застарілих криптографічних бібліотек із обмеженою підтримкою великих чисел;
- використання застарілих «експортних» наборів шифрів із малими параметрами (наприклад, 512-бітні модулі);
- використання однакових або жорстко закодованих параметрів протоколу Діффі–Геллмана для масових пристроїв.

Такі особливості створюють передумови для практичного застосування алгоритмів дискретного логарифмування.

OSINT-аналіз і пошук вразливих пристроїв. Виявлення подібних систем у глобальній мережі здійснюється за допомогою методів OSINT (Open Source Intelligence). Спеціалізовані сервіси, такі як Shodan, дозволяють автоматично сканувати інтернет-простір та аналізувати:

- відкриті порти та сервіси;
- сертифікати безпеки;
- параметри криптографічних handshake-процедур.

На основі цих даних можна ідентифікувати пристрої, які використовують слабкі криптографічні параметри або застарілі протоколи обміну ключами.

Сценарій експлуатації вразливості. Після виявлення цільового пристрою можливе здійснення атаки типу «людина посередині» (Man-in-

the-Middle), під час якої перехоплюються параметри обміну ключами: генератор g, модуль pта публічний ключ h.

Далі застосовується алгоритм baby-step giant-step для розв'язання задачі дискретного логарифмування. У випадку малих значень pобчислювальна складність $O(\sqrt{p})$дозволяє знайти секретний показник xза практично прийнятний час — від кількох хвилин до кількох днів на стандартному обладнанні.

Отримавши значення x, атакувальник відновлює спільний секретний ключ, що дозволяє повністю розшифровувати та модифікувати подальший трафік. Це демонструє критичну важливість правильного вибору криптографічних параметрів навіть для периферійних пристроїв.

Перехід до аналізу обчислювальних меж алгоритму. Розглянуті сценарії показують, що ефективність алгоритму baby-step giant-step суттєво залежить від розміру модуля p. У практичних умовах він може бути ефективним лише для слабких або застарілих систем.

Водночас виникає фундаментальне питання: чи можливо застосувати цей підхід до сучасних криптографічних систем із великими розмірами ключів, таких як 256-бітні еліптичні криві або 1024-бітні RSA/Діффі–Геллман параметри?

Для відповіді на це питання необхідно розглянути теоретичні обчислювальні межі алгоритму та оцінити вимоги до пам'яті й часу його виконання при зростанні розміру ключа.

Сценарій 1: Злам 256-бітного ключа (наприклад, еліптичні криві)

Якщо модуль $p \approx 2^{256}$розрахуємо необхідну кількість кроків і пам'яті:

Кількість елементів у словнику (m):

$$m = \sqrt{2^{256}} = 2^{128}$$

У десятковій системі це приблизно3.4×10^{40} записів.

Обсяг оперативної пам'яті: Припустимо, ми дуже зекономили і один запис займає всього 100 байт.

$V = 3.4\times10^{38} \times 100 = 3.4\times10^{40}$ байт

У даному Python-скрипту знадобилося б 3.4×10^{25} Петабайтів, або 34000000000000000 Йотабайтів оперативної пам'яті.

Для порівняння: загальний обсяг усіх цифрових даних, що існують на планеті Земля у 2024 році, оцінюється приблизно у 150 Зетабайтів (це 0.15 Йотабайта).

Сценарій 2: Злам 1024-бітного ключа (наприклад, класичний Diffie-Hellman або RSA)

Якщо модуль $p \approx 2^{1024}$

Кількість елементів (m):

$$m = \sqrt{2^{1024}} = 2^{512}$$

У десятковій системі це неймовірні 1.34×10^{154} записів.

Обсяг оперативної пам'яті: $V = 1{,}34 \times 10^{156}$ байт

Цей обсяг неможливо виразити у терабайтах, щоб це мало хоч якийсь сенс. Навіть якби кожен байт інформації важив як один атом, оперативна пам'ять для алгоритму перевищила б кількість усіх атомів у видимому Всесвіті (яких лише близько 10^{80}).

Висновки. У статті проведено комплексне дослідження проблеми дискретного логарифмування, яка є фундаментальною основою стійкості сучасних систем асиметричної криптографії, таких як протокол Діффі-Геллмана, криптосистема Ель-Гамаля та криптографія на еліптичних кривих. Класичний метод повного перебору (brute-force) для вирішення цієї задачі виявляється вкрай неефективним і практично неможливим при використанні великих значень модуля.

На противагу цьому, застосування алгоритму «baby-step giant-step» (алгоритму Шенкса) дозволяє суттєво оптимізувати процес пошуку завдяки парадигмі «зустрічного кроку».

Програмна реалізація алгоритму за допомогою мови програмування Python повністю виправдала себе завдяки високорівневій архітектурі, вбудованій підтримці довгої арифметики та ефективній роботі зі структурами даних. Використання словників мови Python як хеш-таблиць дозволило забезпечити миттєвий пошук колізій між масивами «малих» та «великих» кроків за константний час $O(1)$. Емпіричний порівняльний аналіз беззаперечно підтвердив теоретичні розрахунки: на прикладі різних великих простих чисел модулів оптимізований метод Шенкса продемонстрував колосальну перевагу у швидкодії порівняно з лінійним перебором.

Розглянутий алгоритм виходить за межі суто теоретичної математики і стає дієвим практичним інструментом криптоаналітика при роботі зі слабкими або неправильно сконфігурованими інфраструктурами. Зокрема, головною мішенню для подібних атак є сегмент пристроїв Інтернету речей (IoT) та застарілі системи, де через жорсткі апаратні обмеження часто використовуються спрощені криптографічні бібліотеки, застарілі набори шифрів з малим значенням модуля або жорстко закодовані параметри обміну ключами.

Сучасні стандарти шифрування гарантують надійний захист: для теоретичного зламу 256-бітного ключа еліптичних кривих або 1024-бітного ключа класичного алгоритму RSA обсяги необхідної оперативної пам'яті для збереження проміжних кроків значно перевищують кількість усієї цифрової інформації на планеті або навіть кількість атомів у видимому Всесвіті.

Таким чином, розроблена реалізація алгоритму «baby-step giant-step» є блискучим освітнім та аналітичним інструментом, що наочно демонструє практичну вразливість певних параметрів криптосистем, проте сучасна асиметрична криптографія з довжиною ключа від 2048 біт залишається фундаментально стійкою до цього вектору атак

Отже, алгоритм baby-step giant-step є ефективним інструментом лише для криптографічних систем із малими або компрометованими параметрами. Для сучасних стандартів із великими ключами його застосування стає практично неможливим через експоненційне зростання обчислювальних і пам'яттєвих вимог.

Література:

1. Горбенко І. Д., Горбенко Ю. І. Прикладна криптологія. Теорія. Практика. Застосування : монографія. Харків : Форт, 2012. 880 с. ISBN 978-966-8599-91-8.

2. Stinson D. R., Paterson M. B. Cryptography: Theory and Practice. 4th ed. Boca Raton : CRC Press, 2018. 604 p. ISBN 978-1138042276.

3. Shanks D. Class number, a theory of factorization and genera // Proceedings of Symposia in Pure Mathematics. 1971. Vol. 20. P. 415–440. DOI: 10.1090/pspum/020/0316385

4. Кормен Т. Г., Лейзерсон Ч. Е., Рівест Р. Л., Стайн К. Вступ до алгоритмів / пер. з англ. Київ : К.І.С., 2019. 1288 с. ISBN 978-617-684-239-2.

5. Жданов О. М. Сучасні методи дискретного логарифмування в криптографії та оцінка їх обчислювальної складності.

6. Python Software Foundation. Built-in Functions (pow()). The Python Standard Library. URL: https://docs.python.org/3/library/functions.html#pow

7. Python Software Foundation. math — Mathematical functions. The Python Standard Library. URL: https://docs.python.org/3/library/math.html

8. Python Software Foundation. time — Time access and conversions. The Python Standard Library. URL: https://docs.python.org/3/library/time.html

9. Python Software Foundation. Dictionaries. Python Tutorial: Data Structures. URL: https://docs.python.org/3/tutorial/datastructures.html#dictionaries

10. Shodan Help Center. What is Shodan? URL: https://help.shodan.io/the-basics/what-is-shodan

11. Villanueva J. C. How many atoms are there in the universe? Universe Today. URL: https://www.universetoday.com/36302/atoms-in-the-universe/

References:

1. Horbenko, I. D., & Horbenko, Yu. I. (2012). *Prykladna kryptolohiia. Teoriia. Praktyka. Zastosuvannia* [Applied Cryptology. Theory. Practice. Application]. Kharkiv: Fort. ISBN 978-966-8599-91-8 (in Ukrainian).

2. Stinson, D. R., & Paterson, M. B. (2018). *Cryptography: Theory and Practice* (4th ed.). Boca Raton: CRC Press. ISBN 978-1138042276.

3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2019). *Vstup do alhorytmiv* [Introduction to Algorithms]. Kyiv: K.I.S. ISBN 978-617-684-239-2 (in Ukrainian).

4. Shanks, D. (1971). Class number, a theory of factorization and genera. *Proceedings of Symposia in Pure Mathematics*, 20, pp. 415–440. https://doi.org/10.1090/pspum/020/0316385

5. Zhdanov, O. M. (n.d.). *Suchasni metody dyskretnoho loharyfmuvannia v kryptohrafii ta otsinka yikh obchysliuvalnoi skladnosti* [Modern Methods of Discrete Logarithm in Cryptography and Evaluation of Their Computational Complexity] (in Ukrainian).

6. Python Software Foundation. (n.d.). Built-in Functions (pow()). *The Python Standard Library*. https://docs.python.org/3/library/functions.html#pow

7. Python Software Foundation. (n.d.). math — Mathematical functions. *The Python Standard Library*. https://docs.python.org/3/library/math.html

8. Python Software Foundation. (n.d.). time — Time access and conversions. *The Python Standard Library*. https://docs.python.org/3/library/time.html

9. Python Software Foundation. (n.d.). Dictionaries. *Python Tutorial: Data Structures*. https://docs.python.org/3/tutorial/datastructures.html#dictionaries

10. Shodan Help Center. (n.d.). What is Shodan? https://help.shodan.io/the-basics/what-is-shodan

11. Villanueva, J. C. (n.d.). How many atoms are there in the universe? *Universe Today*. https://www.universetoday.com/36302/atoms-in-the-universe/

Дата першого надходження статті до видання: 09.04.2026
Дата прийняття статті до друку після рецензування: 23.04.2026