МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

ДРОЖАК Олександр Олександрович

Алгоритми перевірки на простоту для криптографічних перетворень / Simplicity Checking Algorithms for Cryptographic Transformations

спеціальність: 125 – Кібербезпека та захист інформації
освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконав студент групи КБм -21
О. О. Дрожак

Науковий керівник
к.т.н., доцент С.В.Івасьєв

Кваліфікаційну роботу допущено
до захисту:

«____» _____ ___2025 р.

Завідувач кафедри
_____ В.В.Яцків

ТЕРНОПІЛЬ - 2025

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 - Кібербезпека та захист інформації
освітньо-професійна програма –Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ В.В.Яцків
«____» _____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
ДРОЖАКА ОЛЕКСАНДРА ОЛЕКСАНДРОВИЧА
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Алгоритми перевірки на простоту для криптографічних перетворень / Simplicity Checking Algorithms for Cryptographic Transformations
керівник роботи д.т.н., доцент С.В. Івасьєв
затверджені наказом по університету від 20 грудня 2024 року № 938
2. Строк подання студентом закінченої випускної кваліфікаційної роботи 5 грудня 2025 року.
3. Вихідні дані до кваліфікаційної роботи: завдання на випускну кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- дослідити підходи та методи генерації простих чисел;
- проаналізувати наявні програмні бібліотеки для тестування простоти;
- систематизувати існуючі тести простоти;
- розробити функціональні модулі, що реалізують тести простоти;
- реалізувати експериментальний алгоритм пошуку множників числа через перебір залишків у системі залишкових класів;
- створити оптимізований варіант одного з тестів простоти;
- провести верифікацію та оцінку програмного засобу.

5. Перелік графічного матеріалу у роботі:
- порівняння асимптотичної складності тестів простоти залежно від розряності числа
- порівняння асимптотичної складності тестів простоти на логарифмічній шкалі
- схема виклику функцій в процесі тестування алгоритмів
- схема алгоритму факторизації на основі системи залишкових класів
- модифікований алгоритм тесту Міллера–Рабіна
- результати тестування випадкових чисел

6. Консультанти розділів кваліфікаційної роботи

	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 20 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз предметної області	12.2024 р. – 03.2025 р.	
2	Математичні основи тестів простоти	03.2025 р. – 06.2025 р.	
3	Реалізація програмного засобу та алгоритмів	06.2025 р. – 11.2025 р.	

Студент _____ Олександр ДРОЖАК
(підпис)

Керівник роботи _____ к.т.н., доцент Степан ІВАСЬЄВ
(підпис)

АНОТАЦІЯ

Дрожак О. О. Алгоритми перевірки на простоту для криптографічних перетворень – Рукопис.

Дослідження на здобуття освітнього ступеня «магістр» за спеціальністю 125 «Кібербезпека та захист інформації», освітньо-професійна програма «Кібербезпека». – Західноукраїнський національний університет, Тернопіль, 2025.

У ході дослідження було встановлено, що проблема генерації простих чисел великої розрядності та тестів простоти є фундаментальною для сучасної криптографії, оскільки саме такі числа забезпечують високий рівень стійкості криптографічних протоколів до атак, пов’язаних із факторизацією та обчисленням дискретних логарифмів. Аналіз наявних підходів показав, що генерація великих простих чисел ґрунтується на комбінації швидких імовірнісних тестів простоти, методів випадкового вибору кандидатів та використанні чисел спеціального виду, що дозволяє значно зменшити обчислювальне навантаження при створенні криптографічних ключів.

Запропоновано та реалізовано експериментальний алгоритм пошуку множників числа через перебір залишків у системі залишкових класів. Розроблено оптимізований алгоритм на базі тесту Міллера–Рабіна на основі фіксованих баз і попереднього відсіву.

Ключові слова: прості числа; тести простоти; алгоритм Міллера–Рабіна; тест Соловея–Штрассена; тест Ферма; тест Люка; алгоритм AKS; генерація простих чисел; числа Мерсена; числа Ферма.

ABSTRACT

Drozhak O. O. Simplicity Checking Algorithms for Cryptographic Transformations – Manuscript.

Research for the degree of “Master” in specialty 125 “Cybersecurity and information protection”, educational and professional program “Cybersecurity”. – Western Ukrainian National University, Ternopil, 2025.

During the study, it was found that the problem of generating large-digit prime numbers and tests for simplicity is fundamental for modern cryptography, since it is such numbers that provide a high level of resistance of cryptographic protocols to attacks related to factorization and calculation of discrete logarithms. The analysis of existing approaches showed that the generation of large prime numbers is based on a combination of fast probabilistic tests of simplicity, methods of random selection of candidates and the use of numbers of a special type, which allows to significantly reduce the computational load when creating cryptographic keys.

An experimental algorithm for finding the factors of a number by searching for residues in the system of residue classes is proposed and implemented. An optimized algorithm based on the Miller–Rabin test based on fixed bases and preliminary screening is developed.

Keywords: prime numbers; tests of simplicity; Miller–Rabin algorithm; Solovey–Strassena test; Fermat test; Luke test; AKS algorithm; generation of prime numbers; Mersenne numbers; Fermat numbers.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AKS — алгоритм Агравала–Каяла–Саксени для перевірки простоти чисел;

CRT — Chinese Remainder Theorem, китайська теорема про лишки;

DLP — Discrete Logarithm Problem, задача дискретного логарифмування;

ECC — Elliptic Curve Cryptography, криптографія на еліптичних кривих;

GCD — Greatest Common Divisor, найбільший спільний дільник;

GMP — GNU Multiple Precision Arithmetic Library, бібліотека довільної точності.

ВСТУП

Актуальність дослідження зумовлена ключовою роллю простих чисел у сучасних криптографічних системах, зокрема у таких широко застосовуваних протоколах, як RSA, Diffie–Hellman, ElGamal та алгоритми електронного підпису. Ефективність цих протоколів безпосередньо залежить від можливості швидко генерувати великі прості числа та надійно перевіряти їхню простоту. Зі зростанням обчислювальних потужностей, розвитком квантових обчислень та появою нових атак на криптографічні алгоритми питання оптимізації тестів простоти набуває особливого значення, оскільки дозволяє гарантувати безпечне формування криптографічних ключів та забезпечує стійкість інформаційно-комунікаційних систем до загроз. Прості числа спеціального виду, такі як числа Мерсена й Ферма, також відіграють важливу роль у математичному моделюванні та побудові швидких алгоритмів, що додатково посилює потребу в комплексному дослідженні їхніх властивостей та ефективних методів обробки.

Другим аспектом актуальності є необхідність створення інструментів для експериментальної перевірки алгоритмів, що дозволяють оцінити практичну продуктивність різних тестів простоти в умовах реальних обчислень. Попри наявність значної кількості теоретичних робіт, питання практичної оптимізації й порівняльного аналізу тестів простоти залишаються відкритими, оскільки їх ефективність суттєво залежить від внутрішніх алгоритмічних оптимізацій, структури чисел та програмної реалізації.

Мета і завдання дослідження. Метою роботи є дослідження тестів простоти та розробка алгоритму визначення простоти числа.

Досягнення визначеної мети передбачає вирішення таких завдань:

– дослідити підходи та методи генерації простих чисел великої розрядності для криптографічних цілей;

– проаналізувати наявні програмні бібліотеки для тестування простоти та генерації простих чисел і оцінити їхню ефективність;

– систематизувати існуючі тести простоти за принципами роботи, типом алгоритму та рівнем надійності;

– провести порівняльний аналіз тестів простоти за складністю, надійністю та сферою застосування;

– розробити функціональні модулі, що реалізують обробку числових даних та застосування різних тестів простоти;

– реалізувати експериментальний алгоритм пошуку множників числа через перебір залишків у системі залишкових класів;

– створити оптимізований варіант тесту міллера–рабіна на основі фіксованих баз і попереднього відсіву;

– провести верифікацію та експериментальну оцінку точності й продуктивності реалізованого програмного засобу.

Об’єкт дослідження –алгоритми опрацювання багаторозрядних чисел для криптографічних перетворень.

Предмет дослідження – алгоритми та методи пошуку простих чисел.

Методи досліджень. Для розв’язання поставлених задач у даній кваліфікаційній роботі використано: методи теорії чисел.

Наукова новизна одержаних результатів. Запропоновано та реалізовано експериментальний алгоритм пошуку множників числа через перебір залишків у системі залишкових класів. Розроблено оптимізований алгоритм на базі тесту Міллера–Рабіна на основі фіксованих баз і попереднього відсіву.

Практичне значення роботи полягає у створенні та експериментальному дослідженні програмного засобу, що реалізує сучасні тести простоти й демонструє їхню ефективність у контексті криптографічних застосувань.

Публікації та апробація кваліфікаційної роботи.

1. Дрожак О.О. Поліноміальний алгоритм перевірки чисел на простоту: тест Агравала-Каяла-Саксени/ Матеріали науков-практичного симпозіуму «ЗАХИСТ ІНФОРМАЦІЇ», Тернопіль, 2025. – С. 38-43.

2. Дрожак О.О., Аналіз тестів простоти Ферма та Міллера-Рабіна / Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп’ютерно-інтегровані технології»(КБКІТ-2025), Тернопіль, 2025. - С. 96-98.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Значення простих чисел та тестів прости в криптографії

У криптографії тести простоти та визначення простих чисел мають фундаментальне значення, оскільки саме на їхніх властивостях базується безпека більшості сучасних криптосистем. Просте число, що ділиться лише на одиницю та на себе, формує основу для створення великих простих множників, від яких залежить складність обернених обчислень у криптографічних алгоритмах. Неможливість ефективно розкладати великі складені числа на прості множники є ключовим припущенням безпеки таких систем, як RSA, Diffie–Hellman чи алгоритм Ель-Гамаля. Тому здатність швидко та достовірно перевіряти простоту чисел є критичною умовою для побудови стійких до атак криптографічних схем.

У практичних реалізаціях криптографічних протоколів використовуються числа, що мають сотні або навіть тисячі бітів. Їхня генерація передбачає відбір випадкових кандидатів і перевірку їх на простоту. Оскільки повна факторизація таких чисел є обчислювально неможливою, тести простоти виконуються за допомогою спеціалізованих алгоритмів, які забезпечують достатній рівень довіри до отриманого результату. Визначення простоти не лише гарантує криптографічну коректність, але й запобігає можливим вразливостям, що виникають унаслідок використання псевдопростих або складених чисел, які підривають надійність системи. Навіть незначна похибка у виборі простих чисел може призвести до катастрофічних наслідків, зокрема до компрометації секретних ключів або можливості розв’язання рівнянь, на яких базується криптографічна стійкість.

Тести простоти також відіграють ключову роль у формуванні параметрів еліптичних кривих, генерації великих простих модулів для групових обчислень та побудові полів Галуа, що використовуються в симетричних і постквантових алгоритмах. Вони дозволяють забезпечити необхідні алгебраїчні властивості, без яких неможливо досягнути рівноваги між безпекою та ефективністю криптографічних обчислень. У процесі створення ключів системи перевіряють

не лише простоту, а й певні додаткові властивості, наприклад, придатність числа для утворення групи з заданим порядком, що гарантує непередбачуваність математичної структури.

Таким чином, тести простоти є не лише математичним інструментом, а й одним із наріжних каменів криптографічної безпеки. Вони забезпечують математичну основу, на якій будуються асиметричні протоколи шифрування, цифрові підписи, обмін ключами та інші криптографічні механізми. Від точності та ефективності тестів простоти залежить не лише стійкість окремих алгоритмів, але й загальна надійність інформаційної інфраструктури, що оперує з конфіденційними даними. У сучасних умовах, коли обчислювальні потужності зростають, а математичні методи аналізу постійно вдосконалюються, розроблення швидких, детермінованих і статистично достовірних тестів простоти залишається одним із найважливіших завдань теоретичної та прикладної криптографії.

Числа Мерсена та Ферма займають особливе місце в теорії чисел і криптографії, оскільки їхні властивості безпосередньо пов’язані з ефективністю генерації великих простих чисел і побудовою стійких криптографічних систем. Вони належать до класу спеціальних числових послідовностей, для яких простота може бути перевірена значно швидше, ніж для довільних чисел. Це робить їх надзвичайно привабливими в практичних застосуваннях, де необхідно отримувати великі прості числа з гарантованими математичними властивостями.

Числа Мерсена визначаються виразом $M_p = 2^p - 1$, де p — просте число. Такі числа цікавлять математиків і криптографів тим, що для деяких значень p число M_p також є простим. Просте число Мерсена породжує так звані досконалі числа, які мають тісний зв’язок з давньогрецькою теорією пропорційності та гармонії чисел. З криптографічного погляду, їхня значущість полягає у зручній структурі, що дозволяє реалізовувати швидкі операції в двійковій системі, особливо під час виконання модульних обчислень. Простота представлення у вигляді степеня двійки мінус одиниця дає змогу оптимізувати алгоритми множення, ділення та зведення в степінь — ключові операції при реалізації RSA, Diffie–Hellman чи Ель-Гамаля. Крім того, для чисел Мерсена існують спеціальні тести простоти,

наприклад, тест Люка–Лемера, який є значно ефективнішим за універсальні ймовірнісні методи на зразок тесту Міллера–Рабіна. Завдяки цьому відкриття нових простих чисел Мерсена часто супроводжується практичними досягненнями у сфері обчислювальної криптографії.

Числа Ферма мають форму $Fn=2^{2^n}+1$ і були вперше досліджені П'єром Ферма, який припускав, що всі числа цього вигляду є простими. Проте згодом було доведено, що простими є лише перші п'ять із них. Незважаючи на це, числа Ферма зберігають своє значення у криптографії, оскільки вони пов'язані з побудовою полів Галуа та зручними структурами для модульної арифметики. Їхня форма дозволяє ефективно виконувати операції у системах, де модуль має степеневу структуру, що суттєво знижує обчислювальні витрати під час шифрування та дешифрування. Крім того, властивості чисел Ферма використовуються при побудові швидких алгоритмів зведення в степінь, а також у конструкціях, де потрібно забезпечити унікальні періодичні властивості, наприклад, у генераторах псевдовипадкових чисел.

У більш загальному контексті числа Мерсена та Ферма представляють собою приклади спеціальних форм простих чисел, які поєднують елегантність теоретичної математики з практичною доцільністю в обчислювальних застосуваннях. Їхні алгебраїчні властивості дозволяють будувати криптографічні алгоритми, що поєднують високу швидкодію з математичною строгою безпекою. Саме завдяки таким структурам можливо реалізовувати великі прості модулі в обмежених ресурсних середовищах, зберігаючи при цьому високий рівень криптографічної стійкості. Отже, числа Мерсена і Ферма не лише мають історичну та теоретичну цінність, але й відіграють ключову роль у сучасній практиці побудови ефективних і безпечних криптографічних систем.

1.2 Генерація простих чисел

Генерація простих чисел є однією з найважливіших процедур у криптографії, оскільки від її якості залежить рівень стійкості системи до криптоаналітичних атак. Процес створення простих чисел полягає не лише у випадковому виборі числових кандидатів, але й у забезпеченні того, щоб ці числа відповідали певним математичним і статистичним вимогам. Методи генерації простих чисел охоплюють як детерміновані, так і ймовірнісні підходи, що різняться швидкодією, точністю та рівнем довіри до результату.

У сучасній криптографічній практиці найпоширенішим підходом є випадкова генерація кандидатів із подальшою перевіркою їхньої простоти. На початковому етапі створюється випадкове непарне число заданої довжини, що формується генератором криптографічно стійких випадкових чисел. Це число перевіряється на подільність невеликими простими числами, щоб виключити очевидно складені варіанти. Після цього до вибраного кандидата застосовується ймовірнісний тест простоти, зокрема тест Міллера–Рабіна або Соловея–Штрассена. Такі методи не гарантують абсолютної простоти, проте з практичного погляду забезпечують надзвичайно малу ймовірність похибки, яка для декількох ітерацій тесту зменшується до незначних величин. Цей підхід дозволяє швидко отримувати великі прості числа розміром у сотні або тисячі бітів, що використовуються для побудови модулів у RSA або генерації параметрів еліптичних кривих.

Існують також детерміновані методи, які базуються на строгих математичних доведеннях простоти. Одним із таких методів є алгоритм AKS (Agrawal–Kayal–Saxena), що доводить простоту за поліноміальний час і не вимагає ймовірнісних оцінок. Проте через свою високу обчислювальну складність він поки що має обмежене практичне застосування. Іншим прикладом є використання чисел спеціальної форми — наприклад, чисел Мерсена 2^p-1 - або Ферма 2^n+1, для яких існують оптимізовані тести простоти, що суттєво скорочують обчислення. Такі підходи особливо ефективні при створенні великих

простих чисел для наукових обчислень або спеціалізованих криптографічних протоколів.

Окреме місце посідає метод побудови з гарантованою простотою, коли просте число утворюється через математичну залежність від іншого простого. Наприклад, у схемах генерації так званих “безпечних простих” чисел (safe primes) обирається число q, для якого обчислюється $p=2q+1$, і якщо обидва числа прості, то p використовується як надійний модуль. Такі числа часто застосовуються у протоколах обміну ключами Diffie–Hellman, оскільки вони гарантують утворення групи з великим простим порядком, що підвищує рівень криптографічної безпеки. Подібні методи також застосовуються при формуванні параметрів еліптичних кривих або генерації підгруп для дискретного логарифмування.

Загалом, методи генерації простих чисел поєднують стохастичні та аналітичні підходи з метою забезпечення оптимального співвідношення між безпекою, швидкістю та ресурсними витратами. Вони враховують не лише математичну природу простих чисел, але й практичні аспекти їхнього використання в умовах сучасних обчислювальних систем. Від надійності генерації простих чисел залежить стійкість криптографічних ключів, а отже — цілісність і конфіденційність інформаційних потоків у цифровому середовищі.

Методи «решета» становлять клас алгоритмів для побудови переліку простих чисел у відрізку від 2 до заданого верхнього обмеження N. Їхня ідея ґрунтується на послідовному виключенні кратних чисел — тобто на побудові двійкового критерію «ймовірно просте / гарантовано складене» шляхом відмітки елементів послідовності. Найвідомішим представником цього класу є решето Ератосфена: воно починає з масиву булевих значень довжини N, спочатку позначаючи всі числа як потенційно прості, а потім для кожного непозначеного числа ppp виключає (позначає як складені) усі його кратні, починаючи з p^2. Математична ефективність цього підходу випливає з того, що кожне складене число має найменший простий дільник не більший $\sqrt{N}$, тому виключення кратних для $p > \sqrt{N}$ не створює нових виключень. Алгоритмічна складність стандартної реалізації решета Ератосфена оцінюється як $O(N \log \log N)$ з точки

зору часу при використанні простого масиву, а обсяг пам’яті пропорційний N (залежно від представлення — байт на елемент або бітова маска дає економію в 8 разів).

Практичні реалізації оптимізують цю базову ідею двома основними шляхами: зниженням числа операцій та зменшенням використаної пам’яті. Перший напрям досягається відкиданням парних чисел (збереженням лише непарних індексів), відміткою починаючи з p^2 замість $2p$, та застосуванням «колеса» (wheel factorization), коли множення на перші кілька простих (наприклад 2, 3, 5, 7) враховуються заздалегідь, і алгоритм опрацьовує лише числа, що не діляться на ці малі прості. Другий напрям реалізують за допомогою компактних бітових масивів (bitset), які зменшують простір пам’яті до приблизно $N/8$ байтів, та сегментуванням: замість утримувати весь масив для $[2..N]$, алгоритм послідовно обробляє відрізки фіксованого розміру, використовуючи попередньо знайдений набір простих до $\sqrt{N}$ для виключення кратних у кожному сегменті. Сегментоване решето є ключовим інструментом при роботі з великими верхніми межами, оскільки воно перетворює пам’яткову залежність від N на залежність від розміру сегмента, що робить можливим генерацію простих у діапазонах, що перевищують доступну оперативну пам’ять.

Існують альтернативні схеми решіт — наприклад, решето Сундарама, яке породжує прості через арифметичні перетворення індексів, та решето Аткіна, яке базується на квадратичних формулах і має кращу асимптотичну ефективність у теорії. Решето Аткіна використовує арифметичні властивості квадратичних рівнянь для первинної відмітки кандидатів і вимагає подальшої перевірки кратностей; у практичних реалізаціях воно часто перевершує Ератосфена лише для дуже великих N і при якісній оптимізації кеш-локальності та побітових операцій. Варто також згадати гібридні підходи: поєднання колового «колеса» з сегментованим решетом та побітовою репрезентацією дає дуже ефективні реалізації для генерації переліків простих до кількох мільярдів у межах сучасного апаратного забезпечення. На рисунку 1.1 приведена реалізація решета Сундарама.

```python
def sieve_sundaram(N: int) -> list[int]:
    """
    Часова складність ~ O(n log n), де n ≈ N/2; пам'ять ~ O(n).
    """
    if N < 2:
        return []
    n = (N - 1) // 2  # максимальний індекс для непарних кандидатів (2k+1) ≤ N
    marked = bytearray(n + 1)  # 0 = можливий простий; 1 = виключено

    # позначаємо числа вигляду i + j + 2ij
    for i in range(1, n + 1):
        j = i
        idx = i + j + 2 * i * j
        while idx <= n:
            marked[idx] = 1
            j += 1
            idx = i + j + 2 * i * j

    primes = [2] if N >= 2 else []
    primes.extend(2 * k + 1 for k in range(1, n + 1) if not marked[k])
    return primes
```

Рисунок 1.1 - Реалізація решета Сундарама.

З практичної точки зору важливо усвідомлювати області застосування і обмеження решіт. Решета чудово підходять для отримання повних списків простих у відрізках середніх і великих розмірів, статистичного вивчення розподілу простих, побудови таблиць малих простих для прискорення подальшої факторизації або пробних діленнь. Водночас вони не призначені для отримання одиничних наддовгих простих чисел криптографічного масштабу (наприклад, 2048- або 4096-бітних простих) напряму: генерація такого окремого простого зазвичай базується на випадковому підборі кандидата і на застосуванні ймовірнісних тестів простоти (Міллер-Рабін тощо). Проте решета ефективно використовують у попередніх етапах цієї процедури: відсів кандидатів по малих простих дільниках значно зменшує кількість повних перевірок і підвищує загальну продуктивність генерації ключів.

Імплементаційні аспекти мають суттєве значення для реальної швидкості. Архітектурно-орієнтовані оптимізації, такі як забезпечення доброго використання кешу процесора, побітові операції та вибір розміру сегмента відповідно до розміру L1/L2 кешу, дають багаторазове прискорення порівняно з наївною реалізацією. Паралельні варіанти сегментованого решета дозволяють розподіляти обробку відрізків між потоками або вузлами кластера: кожен потік може незалежно обробляти свій сегмент, використовуючи однаковий набір

базових простих, що дозволяє масштабувати генерацію на багатоядерних системах або навіть GPU (де перевагу дають масивні побітові операції та SIMD-інструкції). Розподілене виконання вимагає узгодження лише початкової фази (побудова простих до $\sqrt{N}$), після чого робота по сегментах не залежить від інших робочих одиниць. На рисунку 1.2 приведена реалізація решета Ератосфена.

```
from math import isqrt

def sieve_eratosthenes(N: int) -> list[int]:
    """
    Часова складність ~ O(N log log N), пам'ять ~ O(N).
    """
    if N < 2:
        return []
    sieve = bytearray(b"\x01") * (N + 1)
    sieve[0:2] = b"\x00\x00"  # 0 і 1 — не прості
    limit = isqrt(N)
    for p in range(2, limit + 1):
        if sieve[p]:
            start = p * p
            step = p
            sieve[start:N + 1:step] = b"\x00" * (((N - start) // step) + 1)
    return [i for i in range(N + 1) if sieve[i]]
```

Рисунок 1.2 – Решето Ератосфена

З теоретичної перспективи порівняння методів решіт зі зведеними підходами підкреслює важливість вибору методу згідно з завданням: коли необхідно повністю перерахувати прості до середнього N, класичне або сегментоване решето з коловою оптимізацією є найефективнішим вибором; коли ж на меті стоїть пошук одиничного великого простого для криптографії, доцільніший випадковий підхід із попереднім пробним діленням по малих простих (реалізованим через коротке решето) і фінішною перевіркою ймовірнісними або детермінованими тестами. У підсумку решета створюють надійний інструмент для побудови та оптимізації багатьох задач у теорії чисел і практичній криптографії: від підготовки таблиць малих простих і відбору кандидатів до масштабованих паралельних обчислень у високопродуктивних середовищах.

1.3 Сучасні бібліотеки для отримання простих чисел

Процедура перевірки простоти цілих чисел та генерації великих простих є базовою операцією в сучасній криптографії, теорії чисел та обчислювальній математиці. Ефективність реалізації цих операцій значною мірою визначає продуктивність криптографічних протоколів, зокрема алгоритмів шифрування з відкритим ключем та схем електронного підпису. У програмному забезпеченні для наукових та прикладних задач використовуються цілі комплекси оптимізованих бібліотек, що забезпечують швидкі алгоритми для генерування простих чисел різних розрядностей. Найбільш відомими з них є GMP, MPIR, MPZ/Libgmpxx, OpenSSL BN, SymPy, NumPy/Numba (опосередковано), SageMath, а також спеціалізовані бібліотеки на кшталт Flint, PARI/GP та NTL. Кожна з них застосовує власні оптимізації, визначаючи швидкодію та надійність реалізованих методів.

Бібліотека GMP є фактично промисловим стандартом для виконання арифметичних операцій над великими числами. Її модуль mpz містить реалізації тестів простоти Міллера–Рабіна, перевірки на складаність та процедури генерації випадкових простих. У контексті тестів простоти GMP використовує набір оптимізованих баз для алгоритму Міллера–Рабіна, що забезпечує детерміновану коректність для 64-бітного діапазону і надзвичайно малу ймовірність помилки для більших чисел.

Ефективність GMP зумовлена ретельно оптимізованими операціями множення великих цілих чисел, що використовують алгоритми Карацюби, Тума-Кука, FFT-множення та інші методи, адаптовані до конкретної архітектури процесора. Завдяки цьому GMP демонструє одні з найкращих у галузі часових показників при роботі з числами розрядності від сотень до десятків тисяч біт. Бібліотека добре масштабується та підтримує паралельні обчислення.

MPIR є високопродуктивним форком GMP, оптимізованим насамперед для операційних систем Windows. Він містить еквівалентні алгоритми генерації простих чисел, а також реалізацію швидкого ймовірнісного тесту простоти з

використанням фіксованих свідків. Ефективність MPIR зіставна з GMP, але у деяких випадках переважає її на платформах x86/x64 завдяки додатковим оптимізаціям під конкретні набори інструкцій.

Як і GMP, бібліотека використовує високоефективні методи множення та редукції, що робить її придатною для криптографічних застосувань.

Модуль BN у складі бібліотеки OpenSSL реалізує генерацію простих чисел та процедури перевірки простоти для потреб криптографічних протоколів SSL/TLS, RSA та ECC. Алгоритмічна основа – поєднання попереднього відсіву дрібними простими, тесту Міллера–Рабіна з адаптивною кількістю раундів та, за потреби, додаткових перевірок на основі факторизації числа $n-1$.

Ефективність OpenSSL є оптимізованою саме для криптографічних задач: бібліотека забезпечує швидке генерування 1024–4096-бітних простих чисел, і водночас гарантує необхідний рівень криптографічної стійкості. На відміну від суто математичних бібліотек, OpenSSL включає механізми криптографічно коректного генерування випадковості, що підвищує безпечність отримуваних простих.

SymPy є бібліотекою символьної математики на Python, що реалізує детерміновані та ймовірнісні тести простоти, включно з AKS, Міллером–Рабіном та тестами Люка. Вона також містить функцію nextprime() та генератор випадкових простих.

Попри широке математичне покриття, SymPy значно поступається GMP та OpenSSL за швидкодією внаслідок використання Python та суто символьної реалізації. Вона застосовується переважно в освітніх та дослідницьких середовищах, де важлива прозорість алгоритмів більше, ніж продуктивність. Проте для чисел до кількох тисяч біт SymPy залишається досить ефективною завдяки внутрішнім оптимізаціям і використанню швидкого Міллера–Рабіна.

SageMath об’єднує бібліотеки GMP, PARI/GP, NTL та FLINT, забезпечуючи комплексне середовище для роботи з числами великої розрядності. У SageMath реалізовані всі сучасні методи перевірки простоти, включно з ECPP (Elliptic Curve Primality Proving), який є одним із найшвидших детермінованих тестів практичного використання. Ефективність SageMath дуже

висока: поєднання GMP для арифметики великих чисел і PARI/GP для тестів простоти дозволяє отримувати одні з найкращих показників часу серед відкритих математичних систем. Тест ECPP, зокрема, здатний підтвердити простоту чисел у сотні тисяч і навіть мільйони біт, чого не можуть забезпечити традиційні ймовірнісні тести.

PARI/GP спеціалізується на числовій теорії і надає низку високоефективних алгоритмів для перевірки простоти, включно з ECPP, тестами Люка, Міллера–Рабіна та факторизаційними методами. Особливістю PARI/GP є надзвичайно оптимізовані внутрішні числові структури, що дозволяють отримувати точні результати з мінімальними часовими витратами. Бібліотека широко використовується у наукових дослідженнях і залишається одним із найпродуктивніших інструментів для великих простих.

NTL — це високопродуктивна C++ бібліотека для обчислювальної теорії чисел, що включає функції генерації простих, тести простоти та модульні арифметичні структури. Вона використовує GMP як бекенд для великих чисел і містить власні оптимізації для операцій над полями.

Ефективність NTL особливо висока у задачах, що потребують багаторазових операцій над числами однакової розрядності, таких як криптографічні протоколи або пошук великих простих.

Розглянуті бібліотеки демонструють суттєві відмінності як у програмній архітектурі, так і в ефективності реалізованих тестів простоти. Найвищу продуктивність забезпечують GMP, OpenSSL BN та NTL, які використовуються в криптографії завдяки оптимізації операцій над великими числами та застосуванню швидких тести простоти зі спеціально підібраними наборами свідків. Наукові системи на кшталт SageMath та PARI/GP забезпечують максимальну точність і найширші можливості за рахунок еліптичних алгоритмів доведення простоти. Інструменти на базі Python забезпечують більшу доступність та інтерактивність, проте поступаються за швидкодією.

Таким чином, вибір бібліотеки визначається конкретною задачею: для криптографії потрібні GMP/OpenSSL, для теоретичних досліджень — PARI/GP або SageMath, а для освітніх інструментів або прототипів — SymPy.

1.4 Прості числа спеціалного виду

Числа Мерсена визначаються формулою $M_p=2^p-1$, де p є натуральним числом. Особливе значення вони мають у випадку, коли p — просте, оскільки лише за цієї умови число Мерсена може бути простим. Такий зв'язок між простотою експоненти й простотою самого числа є фундаментальною властивістю, що суттєво звужує клас потенційних кандидатів на прості числа Мерсена. Доведено, що якщо M_p є простим, то p обов'язково є простим, але зворотне твердження не виконується: далеко не всі числа виду 2^p-1 при простому p є простими.

Однією з ключових математичних характеристик чисел Мерсена є їхня надзвичайно проста двійкова структура, яка зводиться до послідовності з p одиниць. Така форма дозволяє виконувати модульні операції, множення й порівняння надзвичайно ефективно, що зумовило активне використання чисел Мерсена у прикладних задачах. Також числам Мерсена притаманні специфічні факторизаційні властивості: якщо M_p складене, його дільники мають вигляд $kp+1$,k — парне, що дає змогу будувати цілеспрямовані алгоритми для виявлення складеності й аналізу структури таких чисел.

Окрему важливість числа Мерсена мають у контексті відкриття великих простих чисел. Більшість найбільших відомих простих чисел в історії належать саме до класу простих чисел Мерсена, що пояснюється ефективністю алгоритмів тестування простоти для чисел цього типу, насамперед тесту Люка–Лемера. Оскільки структура 2^p-1 дає змогу значно зменшити обчислювальні витрати, саме цей клас чисел став основним об'єктом пошуку надвеликих простих у рамках проєкту GIMPS.

У практичному контексті числа Мерсена застосовуються в генераторах псевдовипадкових чисел, у побудові швидких алгоритмів для довгих цілочисельних операцій та у криптографії. Відомим прикладом є використання модулів виду 2^p-1 у схемах оптимізації множення або побудові прискорених хеш-функцій. Хоча для криптографічних протоколів безпеки прості числа

Мерсена безпосередньо застосовуються рідше через відкритість їхньої структури, вони широко використовуються в алгоритмах, де необхідні операції над великими числами з мінімальними накладними витратами.

Числа Ферма визначаються формулою $F_n = 2^{2^n} + 1, n \geq 0$.

Це надзвичайно стрімко зростаюча послідовність, що робить кожен її елемент об'єктом окремого теоретичного інтересу. Перші числі Ферма F_0,F_1,F_2,F_3,F_4 є простими, однак починаючи з F_5 для багатьох наступних членів послідовності встановлено їхню складеність. На сьогодні відомо лише п'ять простих чисел Ферма, і жодного нового простого члена послідовності не було знайдено протягом останніх трьох століть.

Однією з головних властивостей чисел Ферма є їхня взаємна простота: будь-які два різні числа Ферма не мають спільних дільників. Це випливає з класичної формули:

$F_0F_1\cdots F_{n-1}=F_{n-2}$, що безпосередньо демонструє взаємну простоту членів послідовності. Дана властивість має концептуальне значення у теорії чисел, зокрема для конструкцій нескінченної множини простих чисел та аналізу поведінки експоненційних послідовностей.

Факторизація чисел Ферма є надзвичайно складною задачею через їхній подвійно експоненційний ріст. Відомо, що всі непарні дільники чисел Ферма мають вигляд

$$k\cdot 2^{n+1}+1,$$

що суттєво впливає на алгоритмічні стратегії їхнього розкладання, але не усуває фундаментальної обчислювальної складності проблеми. Деякі великі числа Ферма залишаються неповністю факторизованими, незважаючи на використання сучасних гібридних методів факторизації великої складності.

Числа Ферма мають багатий спектр теоретичних і практичних застосувань. Їхня історично відома роль — побудова правильних багатокутників, оскільки Ґаус довів, що правильний многокутник можна побудувати циркулем і лінійкою тоді й тільки тоді, коли кількість його сторін має вигляд

$$n = 2^k \prod p_i,$$

де p_i — різні прості числа Ферма.

Цей факт демонструє взаємозв'язок між структурою чисел Ферма та геометричними конструкціями у евклідовому просторі.

Окрім геометричних застосувань, числа Ферма мають важливе значення у криптографії й теорії залишків. Їхня структура дозволяє будувати та аналізувати модулі спеціального вигляду, які використовуються в задачах прискорення операцій піднесення до степеня, генерації спеціалізованих алгоритмів для обчислення дискретного логарифма та вивчення групових структур, що виникають у криптографічних протоколах.

Також числа Ферма застосовуються в алгоритмах тестування простоти. Деякі тести використовують властивості чисел виду $2^{2^n} + 1$ як контрольні випадки або як еталонний матеріал для перевірки поведінки алгоритмів на особливо складних або великих екземплярах.

Числа Мерсена та числа Ферма, попри подібність у визначеннях через експоненційні структури, представляють два важливих і різних класи експоненційних послідовностей із глибокими теоретичними й практичними наслідками. Числа Мерсена вирізняються високою ефективністю тестування простоти та застосовуються в пошуку рекордно великих простих чисел, у генерації псевдовипадкових послідовностей і в оптимізації обчислень. Числа Ферма відзначаються унікальними алгебраїчними властивостями та відіграють ключову роль у геометричних конструкціях, криптографії та аналізі групових структур.

2 МАТЕМАТИЧНІ ОСНОВИ ТЕСТІВ ПРОСТОТИ

2.1 Класифікація тестів простоти

У криптографії тести простоти поділяються на ймовірнісні та детерміновані, залежно від того, чи гарантує метод однозначний висновок про простоту числа або лише оцінює її з певною ймовірністю. Цей поділ визначає не лише точність, але й обчислювальну ефективність тестів, оскільки у практичних криптосистемах важливо досягти компромісу між математичною достовірністю та швидкістю роботи алгоритму.

Детерміновані тести простоти дають однозначну відповідь — число є або простим, або складеним. До них належить, зокрема, алгоритм AKS (Agrawal–Kayal–Saxena), який у 2002 році вперше довів, що простоту можна перевіряти за поліноміальний час без імовірнісних припущень. Ідея цього методу ґрунтується на перевірці рівності $(x-1)^n \equiv x^n - 1 (\mathrm{mod}(x^r - 1, n))$, яка виконується лише для простих чисел. Попри свою теоретичну значущість, AKS залишається малопридатним для практичного використання через велику обчислювальну складність на реальних обсягах даних. До детермінованих також належать класичні алгоритми на кшталт тесту Люка і його узагальнення — тесту Люка–Лемера, який використовується для чисел Мерсена. Ці методи ефективні у випадках, коли числа мають спеціальну форму, що дозволяє скоротити обсяг обчислень і спростити перевірку простоти.

На практиці ж у криптографічних системах домінують ймовірнісні тести простоти, які не гарантують абсолютної впевненості, але забезпечують надзвичайно малу ймовірність помилки, часто менш ніж 2^{-100}. Найпоширенішими є тест Ферма, тест Соловея–Штрассена та тест Міллера–Рабіна. Тест Ферма базується на малому теоремі Ферма: якщо ppp просте, то для будь-якого aaa, взаємно простого з p, виконується рівність $a^{p-1} \equiv 1 (\mathrm{mod}\ p)$. Проте існують складені числа, які задовольняють цю умову — так звані псевдопрості числа Ферма, що обмежує надійність цього методу. Тест Соловея–Штрассена усуває частину цих помилок, використовуючи символ Якобі та квадратичний

закон взаємності, що дозволяє точніше відрізняти складені числа від простих. Найефективнішим з практичного погляду є тест Міллера–Рабіна, який комбінує властивості чисел із модульною експоненціацією. Він перевіряє, чи виконується для числа nnn розклад $n-1=2^s \cdot d$ і чи має залишок при зведенні випадкових баз aaa до степеня d форму, характерну для простих чисел. Якщо певна база не задовольняє умову, nnn вважається складеним. Повторення перевірки для кількох незалежних баз дає змогу довести простоту з ймовірністю, близькою до одиниці.

З математичного погляду, ймовірнісні тести втілюють практичний компроміс між швидкістю і впевненістю. У сучасних криптографічних протоколах, таких як RSA або Diffie–Hellman, випадкові великі числа проходять кілька раундів ймовірнісних перевірок із різними базами, що забезпечує рівень довіри, достатній для промислових стандартів безпеки. Детерміновані ж тести використовуються переважно у наукових або спеціалізованих задачах, коли необхідно отримати формально доведений результат без жодної похибки.

Таким чином, обидва типи тестів мають власну сферу застосування: детерміновані — у теоретичній математиці та побудові фундаментальних доказів, ймовірнісні — у прикладній криптографії, де швидкість і практична достовірність мають вищий пріоритет. Їхнє поєднання забезпечує баланс між строгістю і продуктивністю, що є критично важливим для надійного функціонування сучасних систем шифрування та генерації криптографічних ключів.

2.2. Тест Ферма

Цей тест базується на малій теоремі Ферма, яка стверджує: якщо p — просте число, тоді для будь-якого a, що не ділиться на p, виконується умова:

$$a^{p-1} \equiv 1 \pmod{p}.$$

Для перевірки випадкового числа n обирають базу a та обчислюють $a^{n-1} \bmod n$. Якщо результат не дорівнює 1, n — складене. Якщо дорівнює 1, число вважають «ймовірно простим».

Метод Ферма ґрунтується на представленні непарного складеного числа NNN у вигляді різниці квадратів:

$$N=a^2-b^2=(a-b)(a+b),$$

де a та b — цілі числа, причому $a>b\geq 0$. Якщо для деякого a вдається отримати, що a^2-N є точним квадратом, тоді:

$$b = \sqrt{a^2 - N},$$

$$p = a - b, q = a + b$$

Такий метод особливо ефективний у випадках, коли шукані множники близькі за значенням. Для прикладу факторизуємо число N=5959:

Крок 1. Перевірка парності і підготовка. Оскільки N є непарним, метод Ферма можна застосовувати.

Крок 2. Обчислюємо початкове значення $a_0=\lceil\sqrt{N}\rceil$. $\sqrt{5959}\approx 77.199\ldots$ Тому початкове значення $a_0=78$.

Крок 3. Обчислюємо $a_0^2 - N$. $a_0^2=78^2-5959=6084-5959=125$. Число 125 не є квадратом, тому переходимо до наступного a.

Крок 4. Збільшуємо a на 1 і повторюємо

a = 79, $79^2-5959=6241-5959=282$, 282 не є квадратом.

a = 80, $80^2-5959=6400-5959=441$.

$441 = 21^2$, тобто є повним квадратом**.**

Таким чином ми знайшли: $b=\sqrt{441}=21$.

Крок 5. Обчислюємо множники

Тепер: $p=a-b=80-21=59$,.$q = a + b = 80 + 21 = 101$.

Отже: $5959=59\cdot 101$

Складність обчислення $a^{n-1} \bmod n$ полягає в тому що виконується методом швидкого піднесення в степінь за $O(\log n)$ множень, кожне з яких коштує $O((\log n)^2)$ при використанні довгої арифметики, а отже, загальна складність становить приблизно $O((\log n)^3)$.

Серед особливостей є те, що існують складені числа (так звані числа Кармайкла), які проходять тест для багатьох баз, тому тест Ферма не є надійним у криптографічному контексті.

2.3 Тест Соловея–Штрассена

Цей алгоритм ґрунтується на законах квадратичної взаємності. Для заданого nnn і випадкового aaa перевіряється рівність

$$a^{(n-1)/2} \equiv (a/n) \pmod{n},$$

де (a/n) — символ Якобі. Якщо вона не виконується — n складене; якщо виконується для кількох баз — n ймовірно просте.

Перевіримо число n=221 на простоту за тестом Соловея–Штрассена. Оскільки число повинно бути непарним і більше 2, умови виконуються.

Крок 1. Вибір бази a. У тесті Solovay–Strassen база aaa вибирається довільно з множини $2 \le a \le n-2$. Візьмемо стандартну першу базу: a=2.

Крок 2. Обчислення gcd(a,n). gcd (2,221)=1, тому тест триває (якщо gcd(a,n)>1, число автоматично складене).

Крок 3. Обчислення символу Якобі $\left(\frac{a}{n}\right)$. Потрібно обчислити символ Якобі: $\left(\frac{2}{221}\right)$. Факторизувати 221 на цьому етапі не потрібно, але для полегшення пояснення зазначимо, що: 221=13·17.

Символ Якобі для складеного знаменника обчислюється як добуток відповідних символів Лежандра:

$$\left(\frac{2}{221}\right) = \left(\frac{2}{13}\right) \cdot \left(\frac{2}{17}\right).$$

Відомо, що:

$$\left(\frac{2}{p}\right) = \begin{cases} 1, p \equiv \pm 1 \pmod 8, \\ -1, p \equiv \pm 3 \pmod 8. \end{cases}$$

Перевіримо:

$$13 \equiv 5 \equiv -3(\mathrm{mod}\ 8) \rightarrow \left(\frac{2}{13}\right) = -1,$$

$$17 \equiv 1(\mathrm{mod}\ 8) \rightarrow \left(\frac{2}{17}\right) = 1.$$

Отже:

$$\left(\frac{2}{221}\right) = (-1) \cdot 1 = -1.$$

Крок 4. Обчислення показникової конгруенції

Тест Соловея–Штрассена перевіряє тотожність: $a^{(n-1)/2} \equiv (\mathrm{mod}\ n)$. Підставляємо значення: a=2,n=221. Обчислюємо показник:

$$\frac{n-1}{2} = \frac{220}{2} = 110.$$

Тепер потрібно знайти: $2^{110}\ \mathrm{mod}\ 221$. Використаємо модульне піднесення до степеня: $2^{110}\ \mathrm{mod}\ 221 = 188$. Тепер порівнюємо: 188 з $\left(\frac{2}{221}\right) \equiv -1(\mathrm{mod}\ 221)$. В інтерпретації по модулю 221: $-1 \equiv 220\ (221)$.

Крок 5. Порівняння результатів: 188≠220. Конгруенція порушена.

Висновок тесту, оскільки:

$$2^{(n-1)/2} \not\equiv \left(\frac{2}{n}\right)\ (\mathrm{mod}\ n),$$

тест Соловея–Штрассена однозначно встановлює: 221 — складене число.

Складність роботи алгоритму базується на сумарній складності кількох операцій кожної ітерації циклу:

– обчислення $a^{(n-1)/2} \mathrm{mod}\ n$ — $O((\log n)^3)$;

– обчислення символу Якобі — $O((\log n)^2)$;

– сумарно $O((\log n)^3)$ на одну ітерацію.

Ймовірність помилки становить не більше 1/2 для кожної бази a.

2.4. Тест Міллера–Рабіна

Це найпоширеніший ймовірнісний тест, який використовується в більшості криптографічних бібліотек (OpenSSL, GnuPG тощо). Він ґрунтується на представленні числа у вигляді $n-1=2^s \cdot d$, де d непарне. Для випадкової бази a перевіряють, чи виконується:

$$a^d \equiv 1 (\mathrm{mod}\ n) \text{ або } a^{2rd} \equiv -1 (\mathrm{mod}\ n).$$

для деякого $0 \leq r < s$. Якщо обидві умови не виконуються, число — складене.

Обчислювальна складність алгоритму полягає в тому, що кожна ітерація тесту: $O((\log n)^3)$,– для k незалежних баз: $O(k(\log n)^3)$.

Ймовірність помилки алгоритму не перевищує $(1/4)^k$, тому при $k=20$ вона менша за 2^{-40}, чого цілком достатньо для промислових систем.

2.5. Тест Люка

Заснований на послідовностях Люка:

$$U_0=0, U_1=1, U_{k+1}=PU_k-QU_{k-1},$$

де параметри P,Q підбираються так, щоб дискримінант $D=P^2-4Q$ задовольняв певні властивості відносно числа n. Якщо виконуються певні конгруентності для U_{n+1} модулю n, число визнається простим.

Нехай $n>2$ — непарне натуральне число і відомий повний розклад числа $n-1$ на прості множники:

$$n-1=\prod_{i=1}^{k} q_i^{e_i},$$

де q_i — прості числа. Тест Люка стверджує, що якщо існує таке ціле число a, що виконується одночасно: $\gcd(a,n)=1$ та $a^{n-1} \equiv 1 (\mathrm{mod}\ n)$, для кожного простого дільника q_i числа $n-1$ має виконуватися вираз :

$$a^{\frac{n-1}{q_i}} \not\equiv 1 \pmod{n},$$

то число n є простим.

Інтуїтивно: ми шукаємо елемент a у $(Z/nZ)^{\times}$, який має точний порядок $n-1$. Це можливо тільки тоді, коли $(Z/nZ)^{\times}$ — циклічна група порядку $n-1$, а так буває лише при простому n.

Перевіримо методом Люка, що число 29 є простим.

Обчислюємо $n-1$ і його факторизацію $n=29, n-1=28$.

Факторизуємо: $28=2^2 \cdot 7$.

З точки зору тесту Люка нас цікавлять прості дільники числа $n-1$, тобто множина $\{2,7\}$.

Вибираємо деяке a, взаємно просте з n. Візьмемо найпростіший варіант: $a=2$. Перевіряємо: gcd(2,29)=1. Перша умова тесту виконується.

Перевірка умови $a^{n-1} \equiv 1 \pmod{n}$. Потрібно обчислити: $2^{28} \bmod 29$.

Обчислюємо покроково, використовуючи модульні степені (тут коротко за рахунок повторного піднесення):

$2^2=4$,

$2^4=16$,

$2^5=32 \equiv 3 \pmod{29}$,

$2^{10}=(2^5)^2 \equiv 3^2=9 \pmod{29}$,

$2^{20}=(2^{10})^2 \equiv 9^2=81 \equiv 81-58=23 \pmod{29}$,

$2^8=(2^4)^2=16^2=256 \equiv 256-232=24 \pmod{29}$,

$2^{28}=2^{20} \cdot 2^8 \equiv 23 \cdot 24 \pmod{29}$.

Обчислюємо добуток: $23 \cdot 24=552$.

Знаходимо залишок: $552 \div 29=19 \cdot 29=551, 552-551=1$.

Отже, $2^{28} \equiv 1 \pmod{29}$. Друга умова тесту Люка виконується.

Перевірка умов для кожного простого дільника $q \mid n-1$. Потрібно для кожного простого $q \in \{2,7\}$ перевірити, що

$$2^{\frac{28}{q}} \not\equiv 1 \pmod{29}.$$

Випадок $q=2$.

$$\frac{n-1}{2} = \frac{28}{2} = 14.$$

Обчислюємо $2^{14} \bmod 29$. Ми знаємо $2^{28} \equiv 1$. А $(2^{14})^2 \equiv 1 (\text{mod } 29)$. Тобто 2^{14} є коренем рівняння:

$$x^2 \equiv 1 (\bmod 29).$$

У полі Z_{29} це рівняння має розв'язки $x \equiv 1$ та $x \equiv -1 \equiv 28 (\text{mod } 29)$. Перевіримо, яке саме. Обчислимо 2^7 і піднесемо до квадрату:

$2^4=16$

$2^5=32\equiv3$,

$2^6=2\cdot3=6$,

$2^7=12$.

Тоді:

$2^{14}=(2^7)^2=12^2=144\equiv144-116=28(\text{mod } 29)$.

Отже: $2^{14} \equiv 28 \equiv -1 \not\equiv 1(\text{mod } 29)$.

Умова для q=2 виконується.

Випадок q=7.

$$\frac{n-1}{7} = \frac{28}{7} = 4.$$

Обчислюємо: $2^4 = 16 \not\equiv 1(mod\ 29)$.

Отже, й для q=7 виконується рівність:

$$2^{\frac{n-1}{7}} \not\equiv 1 \ (\text{mod } 29).$$

Як підсумок за умовами тесту Люка, маємо:

gcd (2,29)=1;

$2^{28} \equiv 1(\text{mod } 29)$;

Для всіх простих $q \mid 28$, тобто $q \in \{2,7\}$, виконується:

$2^{28/2} \not\equiv 1(\text{mod } 29), 2^{\frac{28}{7}} \not\equiv 1(\text{mod } 29)$.

Отже, згідно з критерієм Люка, число 29 є простим**.**

Обчислювальна складність тесту оцінюється близько $O((\log n)^3)$, подібно до Міллера–Рабіна, але з більшими константами через рекурентні обчислення.

Особливість**:** існують комбінації «тестів Люка–Міллера–Рабіна», що дають детермінований результат для певних діапазонів nnn (наприклад, до 2^{64}).

Тест Люка–Лемера (Lucas–Lehmer test) також застосовують, як спеціалізований детермінований алгоритм для чисел Мерсена $M_p=2^p-1$. Він будує послідовність:

$$s_0=4, s_{i+1}=s_i^2-2 \pmod{M_p},$$

і перевіряє, чи $s_{p-2}\equiv 0 \pmod{Mp}$. Якщо так — число просте.

Складність алгоритму обчислення послідовності включає $p-2$ ітерацій, кожна з яких вимагає одного зведення у квадрат за модулем Mp , тому складність становить $O(p^2)$ (або $O((\log M_p)^2)$, що робить його надзвичайно ефективним для великих чисел Мерсена.

2.6 Алгоритм AKS

Перший детермінований поліноміальний тест простоти, що не базується на припущеннях. Суть методу полягає в тому, що перевіряється тотожність:

$$(x-1)^n \equiv x^n-1 \pmod{(x^r-1, n)},$$

де r — параметр, підібраний для забезпечення математичної коректності.

Розглянемо докладно реалізацію кожного кроку алгоритму та розберемо виконання з прикладу числа 47.

На першому кроці слід визначити, чи є число n точним ступенем іншого числа, тобто $n=a^b$. Мабуть, найпростіший спосіб зробити це – перевірити, що $\lfloor n^{1/b} \rfloor^b \neq n$. При цьому достатньо перевіряти лише значення $2 \leq b \leq log\, n$ (оскільки якщо , то b>log n).

Функція для виконання перевірки буде мати наступний вигляд:

Розглянемо приклад :

$\lfloor log\ 47 \rfloor = 5$

Перевіряємо від 2 до 4:

$\lfloor 47^{1/2} \rfloor^2 = 36$

$\lfloor 47^{1/3} \rfloor^3 = 27$

$\lfloor 47^{1/4} \rfloor^4 = 16$

Отже, 47 не є точним ступенем іншого числа.

Відповідне *r* знаходиться перебором. Для кожного *r* потрібно:

Перевірити, що $n^k \neq 1 \pmod r$ для всіх $k \leq \lfloor log^2\ n \rfloor$.

Якщо одне із значень дорівнює одиниці - спробувати наступне *r*.

Якщо всі рівні одиниці — *r* знайдено.

Відомо, що шукане *r* має порядок не більше $O(\log^5 n)$.

Функція знаходження *r* буде складатись з циклу, котрий зупиниться при виконанні умови пошуку.

Приклад:

$\lfloor log\ ^2 47 \rfloor = 30$

Шукане *r* = 41:

$47^1 \pmod{41} = 6$

$47^2 \pmod{41} = 36$

…

$47^{30} \pmod{41} = 9$

Наступні кроки елементарні, потрібно лише реалізувати обчислення НСД двох чисел.

Досягнення теоретичної складності досить використовувати швидкі алгоритми для арифметичних операцій із многочленами. Тут розглянемо одну ефективну практично оптимізацію: перебування залишку від поділу без прямого поділу многочленів, використовуючи лише додавання.

Якщо многочлен ступеня *r*-1 зводиться у квадрат, то результаті виходить многочлен ступеня 2*r*-2:

$$f(x) = c_{2r-2}x^{2r-2} + \ldots + c_r x^r + c_{r-1}x^{r-1} + c_{r-1}x^{r-1} + \ldots + c_1 x + c_0.$$

Розглянемо поділ на x^r-1 $x^r - 1$:

$f(x) = a(x)(x^r - 1) + b(x)$, де

$a(x) = a_{r-2}x^{r-2} + \ldots + a_0$

$b(x) = b_{r-2}x^{r-2}+\ldots+b_0$

Розкриваючи дужки, знаходимо коефіцієнти b:

$b_0 = c_0 + a_0 = c_0 + c_r$

$b_1 = c_1 + a_1 = c_1 + c_{1+r}$

$b_{r-2} = c_{r-2} + a_{r-2} = c_{r-2} + c_{2r-2}$

Приклад: $\left\lfloor\sqrt{\varphi(41)log\ 41}\right\rfloor = 36$

$$a = 1: (x+1)^{47} = x^{47} + 1\ (mod\ x^{41} - 1{,}47) = x^6 + 1$$

$$a = 2: (x+2)^{47} = x^{47} + 2\ (mod\ x^{41} - 1{,}47) = x^6 + 2$$

…

$$a = 36: (x+36)^{47} = x^{47} + 36\ (mod\ x^{41} - 1{,}47) = x^6 + 36$$

Обчислювальна складність для початкового варіанту — $O((\log\ n)^{12})$, оптимізовані версії — $O((\log\ n)^6)$. Попри поліноміальність, великі коефіцієнти роблять його повільним для реальних криптографічних застосувань.

2.7 Порівняння існуючих тестів простоти

Існують і інші тести простоти, такі як: Elliptic Curve Primality Proving (ECPP), Pocklington's criterion та інші. Elliptic Curve Primality Proving (ECPP) — практично детермінований тест із середньою складністю $O((\log n)^5)$. Є одним із найшвидших відомих практичних тестів для дуже великих чисел. Pocklington's criterion — напівдетермінований тест, що використовує часткову факторизацію $n-1$. Складність залежить від розміру факторів і зазвичай становить $O((\log n)^3)$. В таблиці 2.1. приведено порівняння характеристик найрозповсюдженіших тестів простоти.

Таблиця 2.1 – Характеристики тестів прости

Тест	Тип	Імовірність помилки	Складність	Придатність
Ферма	Ймовірнісний	до 1/2	$O((\log n)^3)$	Теоретичний, попередній відсів
Соловей–Штрассен	Ймовірнісний	≤ 1/2	$O((\log n)^3)$	Рідко використовуєтьс я
Міллер–Рабін	Ймовірнісний	≤ (1/4)^k	$O(k(\log n)^3)$	Основний у криптографії
Люка	Детермінований / ймовірнісний	0 / низька	$O((\log n)^3)$	Додаткова перевірка
Люка–Лемера	Детермінований (для Мерсена)	0	$O((\log n)^2)$	Спеціалізований, ефективний
AKS	Детермінований	0	$O((\log n)^6)$	Теоретичний, не практичний
ECPP	Практично детермінований		$O((\log n)^5)$	Для доведення простоти
Pocklington	Напівдетерміновани й	залежить від фактори-зації	$O((\log n)^3)$	Для частково відомих структур

Таким чином, у практичній криптографії використовуються насамперед ймовірнісні методи (Міллер–Рабін, іноді з комбінацією Люка), які забезпечують баланс між швидкістю й надійністю. Детерміновані тести (AKS, ECPP) мають основне значення для математичних доказів простоти, сертифікації простих чисел і досліджень у теорії чисел. Графічно порівняння асимптотичної складності тестів простоти залежно від розряності числа приведено на рисунку 2.1.

Рисунок 2.1 – Порівняння асимптотичної складності тестів простоти залежно від розряності числа

Лінії з показником k = 3 (Ферма, Міллер–Рабін, Соловей–Штрассен, Люка) ростуть помірно і практично збігаються. Це підтверджує, чому вони ефективні навіть для 2048–4096-бітних чисел.

Лінія для тесту Люка–Лемера (k=2) зростає ще повільніше, що відображає його виняткову ефективність для чисел Мерсена.

Методи ECPP (k=5) та особливо AKS (k=6) ростуть набагато стрімкіше — витрати обчислень вибухають для великих бітових довжин. Саме тому AKS практично не застосовується. Для того щоб відобразити різницю між тестами простоти з схожими складностями побудуємо графік на логарифмічній шкалі (рисунок 2.2).

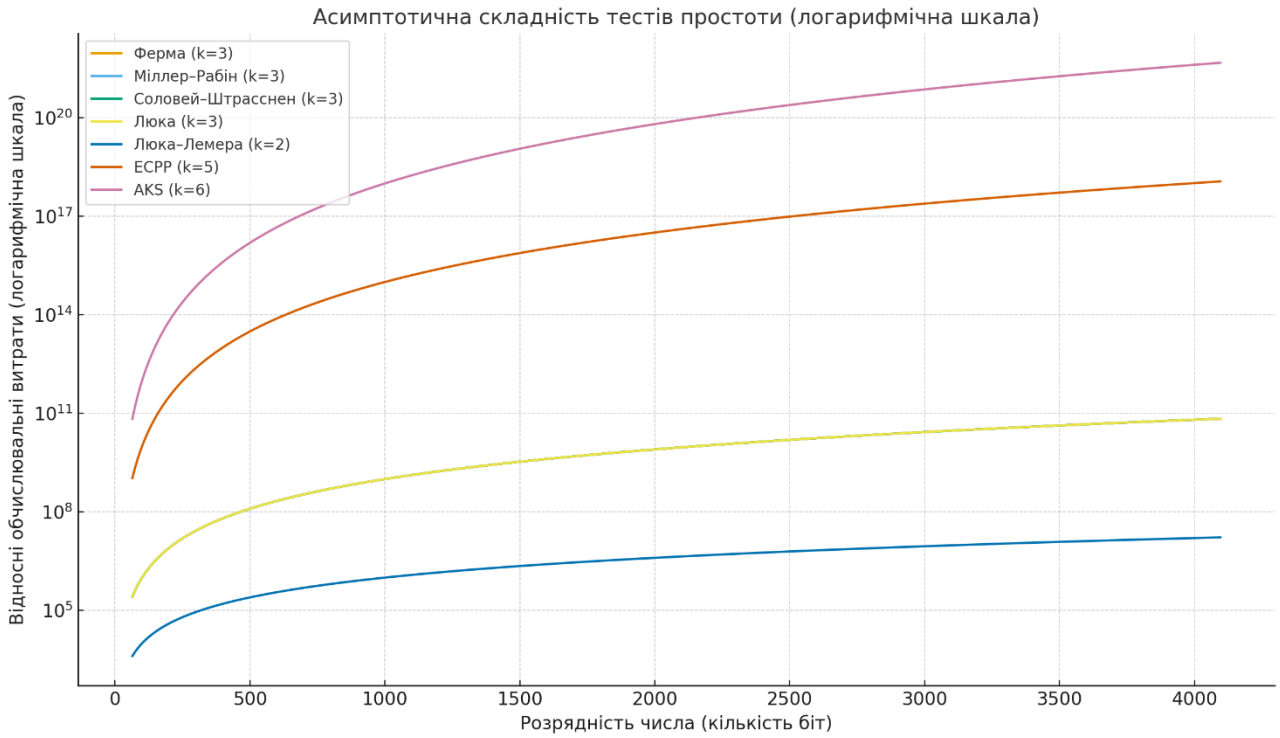

Рисунок 2.2 – Порівняння асимптотичної складності тестів простоти на логарифмічній шкалі

Лінії з k = 3 (Ферма, Міллер–Рабін, Соловей–Штрасснен, Люка) практично зливаються — вони мають однаковий асимптотичний ріст.

Тест Люка–Лемера (k=2) зростає повільніше за всі інші, підтверджуючи свою виняткову ефективність для чисел Мерсена.

ECPP (k=5) та AKS (k=6) демонструють значно крутіший ріст — на логарифмічній шкалі видно, що їхня складність на порядок або два вища.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСОБУ ТА АЛГОРИТМІВ

3.1 Функції обробки та тестування простоти

Програмний засіб реалізований засобами python в вигляді функцій. Функція small_prime_trial реалізує попередній етап перевірки на наявність малих простих дільників. Метою є швидке виявлення тривіально складених чисел без запуску складніших тестів простоти. Алгоритм послідовно перевіряє подільність числа на набір попередньо визначених малих простих чисел. У разі знаходження дільника функція повертає відповідну інформацію, що дозволяє оптимізувати загальний процес і уникнути зайвих витрат часу на подальші високовартісні тести. На рисунку 3.1 приведено схему виклику основних функцій, що створені для тестування ефективності алгоритмів перевірки числа на простоту.

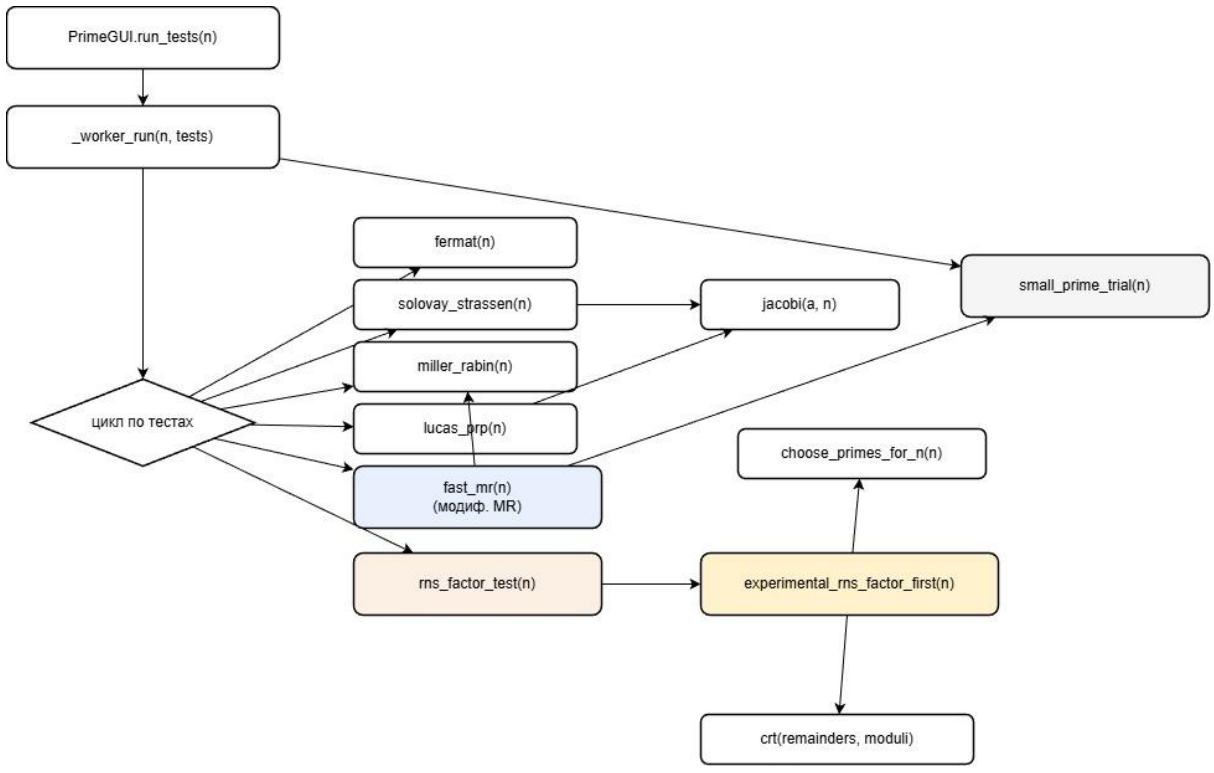

Рисунок 3.1 – Схема виклику функцій в процесі тестування алгоритмів

Функція parse_int відповідає за коректне перетворення текстового вводу у числовий тип. Вона підтримує як стандартний десятковий формат, так і шістнадцяткову нотацію виду 0x.... Це забезпечує універсальність інтерфейсу та

зручність для користувачів, що працюють із великими числами у криптографічних дослідженнях, де шістнадцятковий формат є поширеним.

Функція rand_odd_bits генерує випадкове непарне число заданої бітової довжини. Використання генератора випадкових чисел забезпечує непередбачуваність результату, а встановлення старшого біта гарантує точну відповідність заданій розрядності. Додавання біта молодшого розряду (1) забезпечує непарність числа, що є необхідною передумовою для тестування простоти (оскільки всі парні числа, крім 2, є складеними).

Функція jacobi реалізує обчислення символу Якобі, який є узагальненням символу Лежандра та використовується у тестах простоти, зокрема у тесті Соловея–Штрассена та у тестах Люка. Символ Якобі дозволяє визначити квадратичний характер залишків і є ключовим елементом перевірки у конгруенціях вигляду

$$a^{(n-1)/2} \equiv (\frac{a}{n}) mod\ n.$$

Функція fermat реалізує ймовірнісний тест Ферма, що ґрунтується на малій теоремі Ферма. Метою тесту є визначення, чи задовольняє число ключову властивість простих чисел:

$$a^{n-1} \equiv 1 (mod\ n).$$

Алгоритм виконує перевірку для випадкового набору баз aaa, що робить його швидким, але потенційно вразливим до чисел Кармайкла. Функція використовується як попередній або допоміжний тест.

solovay_strassen – функція, що реалізує покращений ймовірнісний тест Соловея–Штрассена, який поєднує обчислення символу Якобі з піднесенням у степінь. Тест забезпечує значно нижчу ймовірність помилки порівняно з тестом Ферма, а його математичні властивості роблять його надійним методом фільтрації складених чисел. Використання цієї функції особливо доцільне під час роботи з великими розрядностями.

Функція реалізує тест Міллера–Рабіна miller_rabin — один із найпоширеніших та найефективніших ймовірнісних тестів простоти. Його сутність полягає у представленні числа у вигляді

$$n - 1 = 2^s d,$$

і послідовному перевірянні умов, що вказують на простоту або складеність числа. У програмному засобі алгоритм підтримує як використання випадкових баз, так і фіксованих, що дозволяє забезпечити детерміновану коректність для чисел у певних діапазонах.

Функція lucas_prp виконує перевірку на простоту за методом ймовірнісного тесту Люка (Lucas probable prime). На відміну від попередніх тестів, метод Люка використовує рекурентні послідовності та квадратичні форми, що дозволяє виявляти певні категорії псевдопростих чисел, які проходять тест Міллера–Рабіна. Функція служить важливим доповненням у складеніших комбінованих алгоритмах, таких як Baillie–PSW.

PrimeGUI.__init__(...) - конструктор класу відповідає за ініціалізацію всіх параметрів інтерфейсу, зокрема створення прив'язаних змінних, формування структури блока алгоритмів та контролів, ініціалізацію таблиці результатів та налаштування черги для обміну даними між основним та фоновим потоком виконання. Функція гарантує належне відтворення інтерфейсу і правильну логіку відображення результатів.

PrimeGUI._build_ui(self) - функція формує повний графічний інтерфейс користувача: поля введення, кнопки, фрейми, таблиці та елементи взаємодії. Її структура організована так, що компоненти логічно групуються за функціональним призначенням, забезпечуючи інтуїтивність використання та полегшуючи подальше розширення або модифікацію інтерфейсу.

Функція PrimeGUI.generate_candidate(self) відповідає за інтерактивну генерацію випадкового непарного числа заданої розрядності. Це дозволяє користувачу проводити серію експериментів без необхідності вводити значення вручну.

Метод PrimeGUI.run_tests(self) організовує запуск тестів простоти у фоновому потоці. Він відповідає за:

- зчитування параметрів користувача,
- формування переліку активованих алгоритмів,
- початок роботи прогрес-бара,

– створення нового потоку для обчислень,

що забезпечує плавність роботи інтерфейсу і запобігає його блокуванню.

PrimeGUI._worker_run(self, n, tests) - це фоновий обчислювальний процес, який по черзі запускає всі вибрані алгоритми тестування для поданого числа. Функція вимірює час роботи кожного алгоритму та формує підсумковий висновок про простоту або складеність числа. Результати передаються до основного потоку через чергу повідомлень.

PrimeGUI._poll_queue(self) метод що реалізує механізм періодичного опитування черги, у яку фоновий потік надсилає отримані результати. Завдяки цьому вся взаємодія між потоками є безпечною та синхронізованою. Функція автоматично оновлює таблицю результатів та керує зупинкою індикатора прогресу.

PrimeGUI.export_csv(self) - функція забезпечує можливість експорту результатів тестування у формат CSV. Це дозволяє зберігати отримані експериментальні дані, документувати їх у звітах та використовувати для подальшого аналізу.

3.2 Метод факторизації на основі системи залишкових класів

У розробленому програмному засобі реалізовано експериментальний метод пошуку нетривіальних дільників цілого числа на основі системи залишкових класів за простими модулями. Цей підхід не претендує на ефективність порівнянну з класичними алгоритмами факторизації (методами квадратичного решета, числового поля тощо), однак має суттєву дидактичну цінність, оскільки демонструє можливість використання системи залишкових класів для звуження простору пошуку потенційних множників.

В основі методу лежить представлення числа n у вигляді вектора його залишків за простими модулями. Нехай задано перші k простих чисел $p_1,\ldots,p_k$, такі що добуток модулів є істотно меншим за n, а самих модулів достатньо для

відображення корисної частини інформації про структуру числа. Для кожного модуля обчислюється залишок $r_i=n \bmod p_i$. З точки зору теорії чисел, будь-який нетривіальний дільник d числа nnn також однозначно визначається в системі залишкових класів своїм вектором ($d \bmod p_1,\ldots,d \bmod p_k$), причому для кожного i виконується співвідношення

$$n \equiv d \cdot q(mod\ p_i),$$

де q — відповідний співмножник. У запропонованій реалізації ця залежність використовується у спрощеній евристичній формі: для кожного модуля p_i серед можливих залишків кандидата на дільник розглядаються лише такі значення $d_i \in \{1,\ldots,p_i-1\}$, для яких звичайний залишок r_i ділиться на d_i без остачі. Така умова не є необхідною в строгому математичному сенсі, однак дозволяє різко звузити множину потенційних залишків і тим самим зменшити кількість комбінацій, що підлягають перебору.

Після того як для кожного модуля побудовано обмежений набір допустимих залишків d_i , виконується повний перебір усіх можливих векторів ($d_1,\ldots,d_k$). Для кожного такого вектора за допомогою китайської теореми про залишки відновлюється відповідний кандидат d у цілій області. Реалізація КТЗ передбачає обчислення добутку всіх модулів, побудову часткових модулів та обернених елементів у відповідних кільцях, що дозволяє отримати найменший невід’ємний розв’язок системи конгруенцій.

Відновлене число d інтерпретується як потенційний множник. Для нього перевіряється умова нетривіальності $1<d<n$, а також виконується стандартна операція ділення $n \bmod d$. У разі якщо отримано нульовий залишок, знайдено дійсний нетривіальний дільник числа n, після чого перебір комбінацій припиняється. Якщо в межах заданого множини модулів і дозволених залишків жодного дільника виявити не вдалося, робиться висновок про те, що метод не знайшов факторів за заданих параметрів, але це не означає простоту числа загалом. Схема розробленого алгоритму приведена на рисунку 3.2.

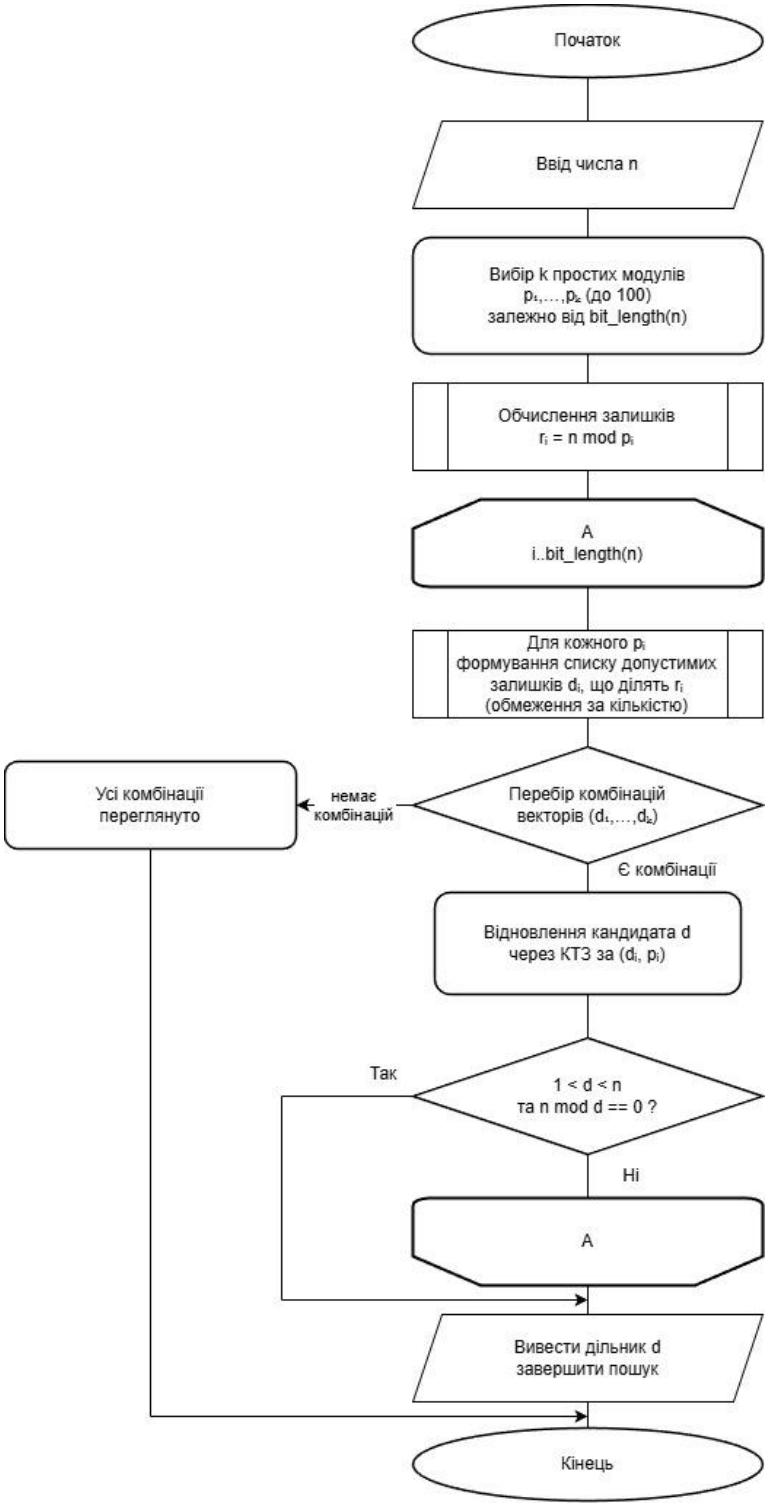

Рисунок 3.2 – Схема алгоритму факторизації на основі системи залишкових класів

У програмному засобі кількість простих модулів для системи залишкових класів обирається адаптивно залежно від бітової довжини числа. Використовується евристичне правило, згідно з яким кількість модулів зростає зі збільшенням розрядності, однак обмежується зверху ста першими простими

числами. Для кожного модуля кількість допустимих залишків дільника обмежується невеликою константою, що запобігає вибуху комбінаторики. Такий підхід забезпечує баланс між наочністю та обчислювальними витратами, дозволяючи використовувати метод як експериментальний інструмент для факторизації чисел середнього розміру.

В інтерактивному інтерфейсі даний алгоритм представлено як окремий режим «RNS factor search». Результатом його роботи є повідомлення про успішне знаходження дільника з явним зазначенням значення фактора або інформація про відсутність знайденого множника в межах здійсненого перебору. Отримані результати можуть бути експортовані у формат CSV для подальшого аналізу, порівняння із класичними алгоритмами або використання у навчальних цілях.

3.3 Модифікований тест Міллера–Рабіна

Тест Міллера–Рабіна є одним із найефективніших ймовірнісних алгоритмів перевірки простоти цілих чисел, який ґрунтується на аналізі структури мультиплікативної групи цілих чисел за модулем перевірюваного значення nnn. Основним джерелом його популярності є поєднання високої швидкодії, простої реалізації та надзвичайно малої ймовірності хибнопозитивного результату. Проте класична форма алгоритму передбачає використання випадкових баз, що приводить до певної варіабельності часу виконання та потреби у виборі достатньої кількості раундів, аби гарантувати надійність для чисел довільної розрядності.

Запропонована модифікація тесту Міллера–Рабіна полягає у використанні фіксованих наборів баз, які забезпечують детерміновану коректність у чітко визначених діапазонах значень nnn. Цей підхід базується на відомих теоретичних результатах про поведінку псевдопростих чисел відносно певних баз, а також на численних емпіричних дослідженнях, що встановлюють найменші множини

свідків, достатніх для покриття всього проміжку натільних чисел до заданої верхньої межі.

Класичний тест Міллера–Рабіна для певної бази a перевіряє конгруенції, що випливають із слабкої форми малої теореми Ферма: $a^{n-1} \equiv 1(mod\ n)$, а також із розкладу $n-1 = 2^s d$, d непарне.

За умови, що n — просте, піднесення a^d та послідовних квадратів має призводити до визначеної структури елементів у групі $Z_n^\times$. Виявлення відхилення від цієї структури означає, що число є складеним.

Теоретично доведено, що для будь-якого складеного числа існує "свідок складеності" — така база ааа, що тест Міллера–Рабіна виявить непрохідність. Однак частка баз, які не є свідками, тобто помилково засвідчують простоту псевдопростих чисел, є мізерно малою. Це дозволяє використовувати невелику кількість баз.

Сутність модифікації полягає у переході від випадкового вибору баз до детермінованого набору, оптимального для чисел у конкретних діапазонах. Дослідження останніх десятиліть (Jaeschke, Sorenson, Damgård, Pomerance, Sinclair та ін.) встановили мінімальні набори баз, що гарантують відсутність псевдопростих чисел для всіх n у межах, наприклад, 10^{15}, та навіть для всього діапазону 64-бітних чисел.

У межах 64-бітного діапазону відомо, що множина баз:

$$a \in \{2,325,9375,28178,450775,9780504,1795265022\}$$

є повною, тобто жодне складене число менше за 2^{64} не здатне пройти тест Міллера–Рабіна одночасно для всіх цих баз. Це дозволяє оголосити метод детермінованим на практиці для цього діапазону.

Таким чином, модифікація передбачає наступне:

- у межах певних діапазонів застосовуються фіксовані таблиці баз, які замінюють випадковий вибір;
- кількість баз у таких наборах є мінімальною, що забезпечує найкоротший час виконання;
- для більших чисел використовується стандартний ймовірнісний варіант з обмеженою кількістю раундів.

На рисунку 3.3 приведено модифікований алгоритм тесту Міллера–Рабіна, що дещо спрощує класичну реалізацію і в багатьох випадках його швидкодія значно вища.

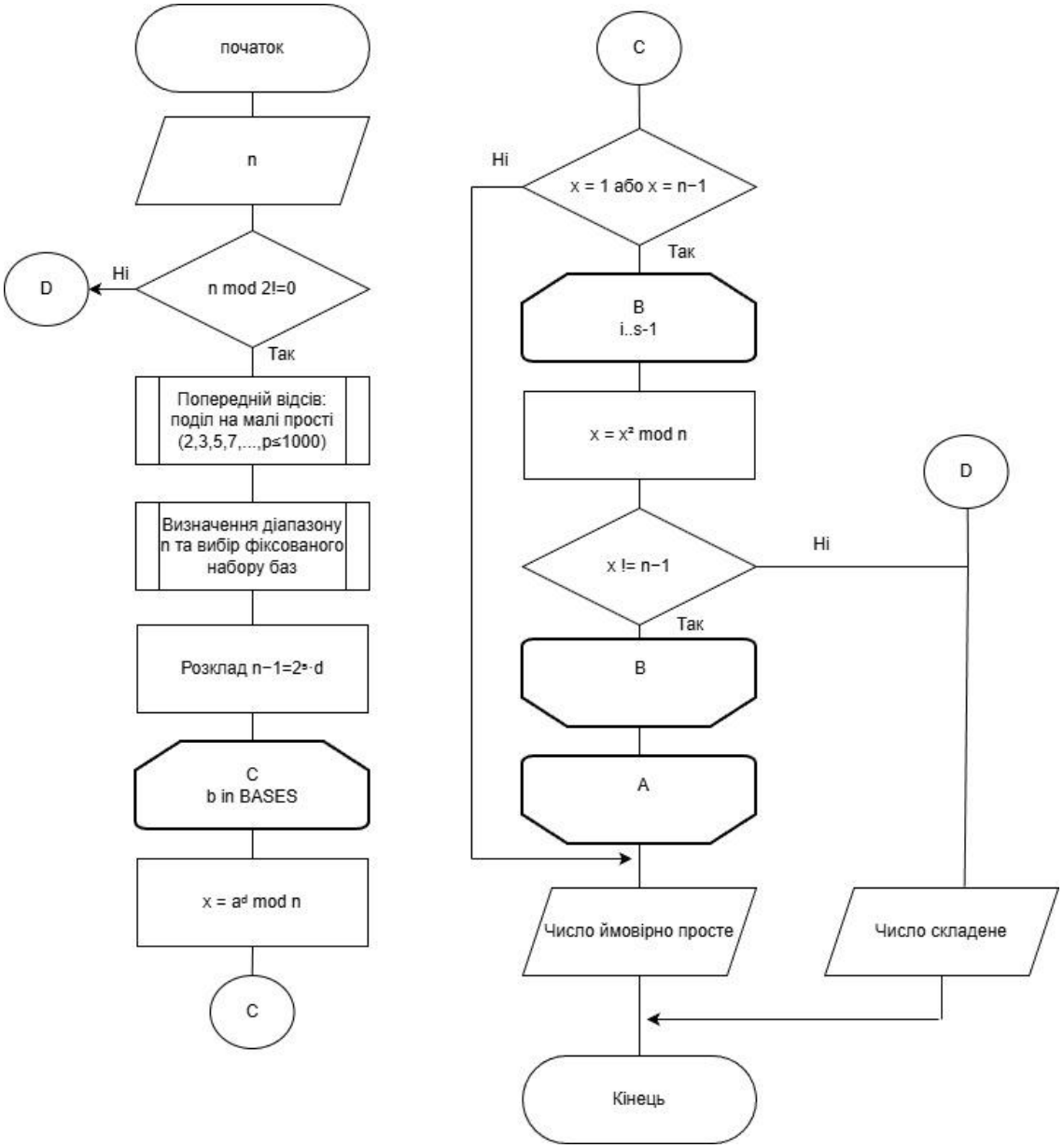

Рисунок 3.3 – Модифікований алгоритм тесту Міллера–Рабіна

Завдяки використанню фіксованих наборів баз час роботи алгоритму скорочується з двох причин:

1. Скорочується кількість степеневих обчислень, оскільки замість 8–12 випадкових баз використовується 3–7 фіксованих.

2. Зникає потреба у генерації випадкових чисел, що особливо важливо в контексті криптографії, де випадкове число має відповідати певним вимогам щодо ентропії.

3. Не виконується зайва перевірка, оскільки фіксовані бази підібрано так, що вони усувають спеціальні класи складених чисел (наприклад, псевдопрості числа до певних значень).

Додатково був інтегрований попередній відсів на основі ділення на малі прості числа, що дозволяє відкидати значну частину складених значень ще до виконання основного тесту.

Запровадження такої модифікації значно підвищує ефективність тесту в умовах, коли необхідно виконувати масові перевірки чисел або оцінювати велику кількість кандидатів, наприклад, під час генерації простих чисел для криптографічних протоколів.

Модифікація дає такі переваги:

- детермінована коректність для всього 64-бітного діапазону;
- висока швидкодія завдяки мінімальним наборам баз;
- простота реалізації, оскільки модифікація не змінює структури алгоритму;
- зниження ймовірності хибнопозитивних результатів через використання оптимальних свідків;
- покращена стабільність часу виконання — відсутність варіацій, притаманних варіантам з випадковими базами.

Модифікація тесту Міллера–Рабіна, реалізована у програмному засобі, представляє собою компроміс між строгістю детерміністичного методу та швидкодією ймовірнісного алгоритму. Використання фіксованих наборів баз у визначених діапазонах дозволяє досягти практично оптимальної ефективності без втрати достовірності для основних класів задач, де необхідні оцінки простоти чисел.

У поєднанні з попереднім відсівом на малих простих та використанням мінімальної кількості степеневих операцій модифікований алгоритм поєднує високу швидкість із теоретично обґрунтованою надійністю. Саме це робить його одним із найефективніших інструментів для реалізації перевірки простоти у розробленому програмному забезпеченні.

3.4 Тестування програмного забезпечення

Графічний інтерфейс розробленого програмного засобу побудований на основі бібліотеки Tkinter та має структуру, орієнтовану на забезпечення зручності взаємодії користувача з основними функціональними можливостями системи. Він містить декілька логічно відокремлених областей, кожна з яких відповідає за введення параметрів, вибір алгоритмів тестування, ініціацію обчислень та перегляд результатів. На рисунку 3.4 приведено головне вікно програмного засобу з результатами тестування 32-бітного випадкового числа.

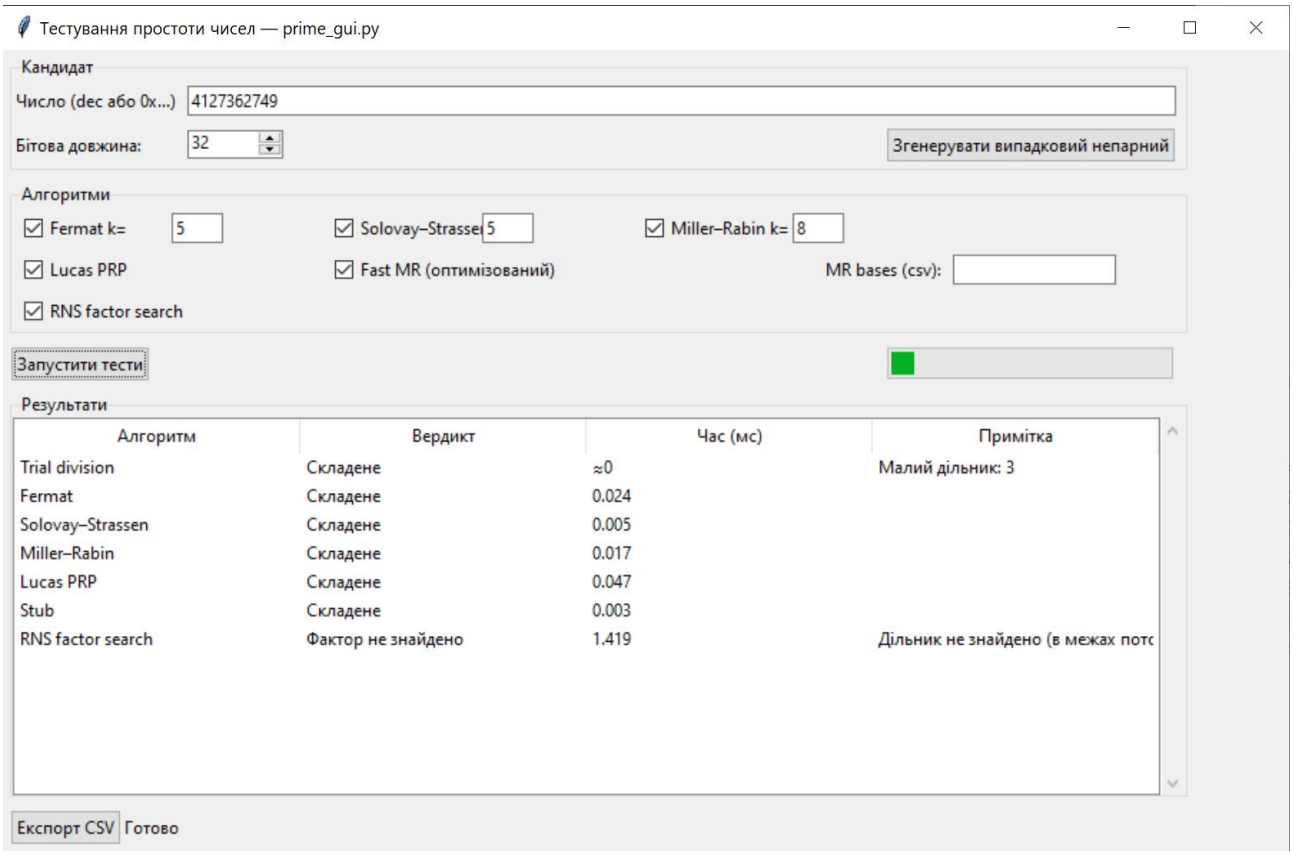

Рисунок 3.4 - Результати тестування 32-бітного випадкового числа

Основна область введення даних призначена для задання числа, що підлягає перевірці. Користувач має можливість вводити його у десятковому чи шістнадцятковому форматі, що забезпечує універсальність застосування системи під час роботи з великими криптографічними параметрами. Додатково

передбачено засіб генерації випадкових кандидатів заданої бітової довжини, що полегшує проведення експериментальних досліджень з формуванням наборів тестових значень. Для тесту розроблених алгоритмів проведено ряд експериментів з різними розрядностями чисел, що тестуються. На рисунку 3.5 приведено результати тестування для 64-бітного випадкового числа.

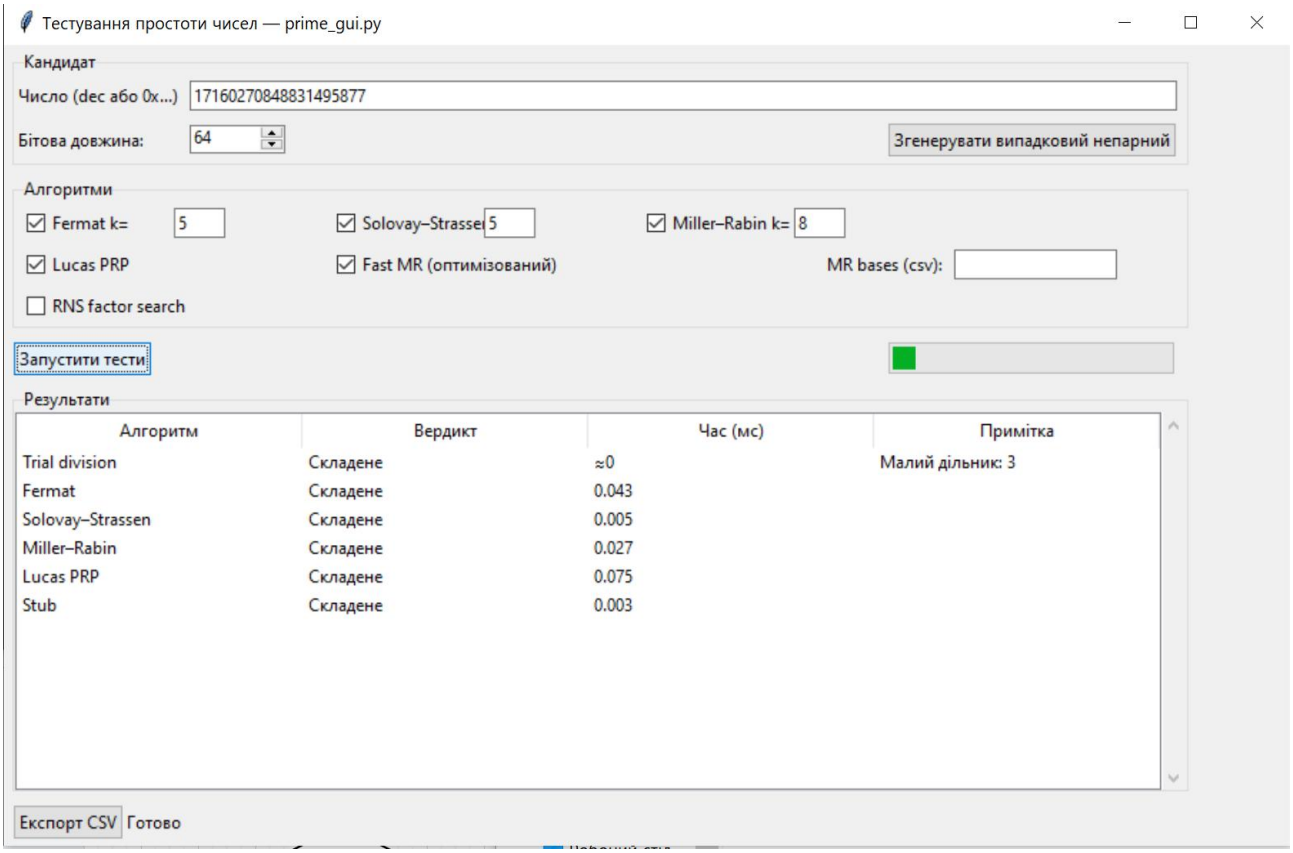

Рисунок 3.5 - Результати тестування 65-бітного випадкового числа

В програмному засобі реалізовано обидва запропонованих алгоритми. Починаючи з 64 біт, тестування методу перевірки простоти, що базується на СЗК припинено, оскільки часовий проміжок для цього методу занадто великий. Проте запропонований модифікований тест Міллера–Рабіна показує кращі результати за його класичну реалізацію. Центральним елементом інтерфейсу є керуюча кнопка, що ініціює процес тестування. Її активація запускає окремий фоновий потік виконання, завдяки чому графічна оболонка залишається інтерактивною навіть під час виконання тривалих обчислень. Індикація стану процесу реалізована через текстове поле та динамічне оновлення таблиці результатів. На рисунку 3.6 приведено тестування для 128-бітного числа.

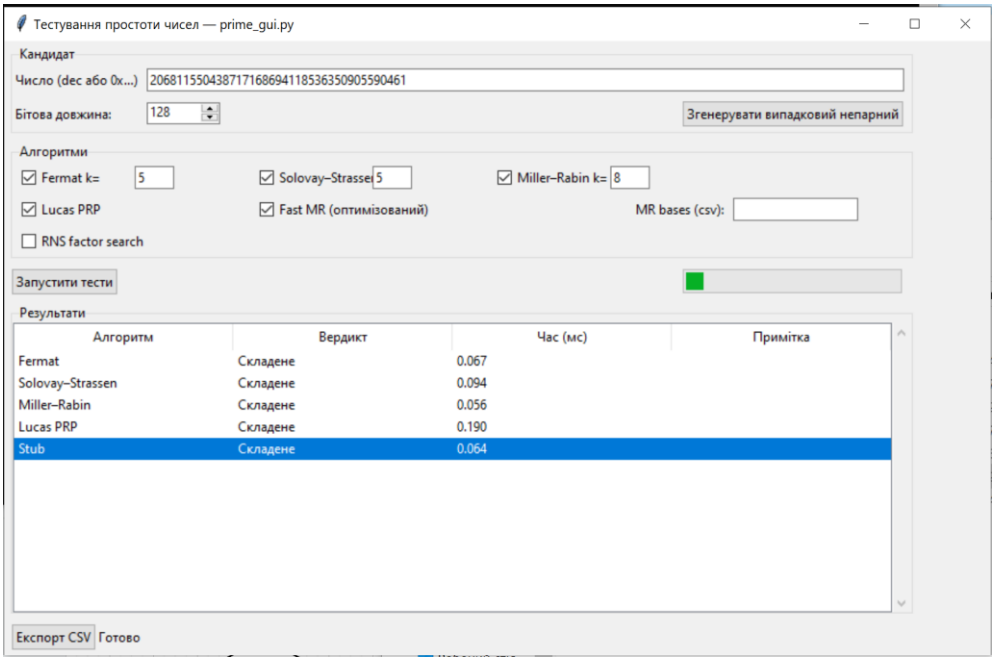

Рисунок 3.6 - Результати тестування 128-бітного випадкового числа

Проведено також тестування для чисел розрядністю 512 біт (рисунок 3.7). Запропонована модифікація дозволяє ефективно перевіряти багаторозрядні числа на простоту.

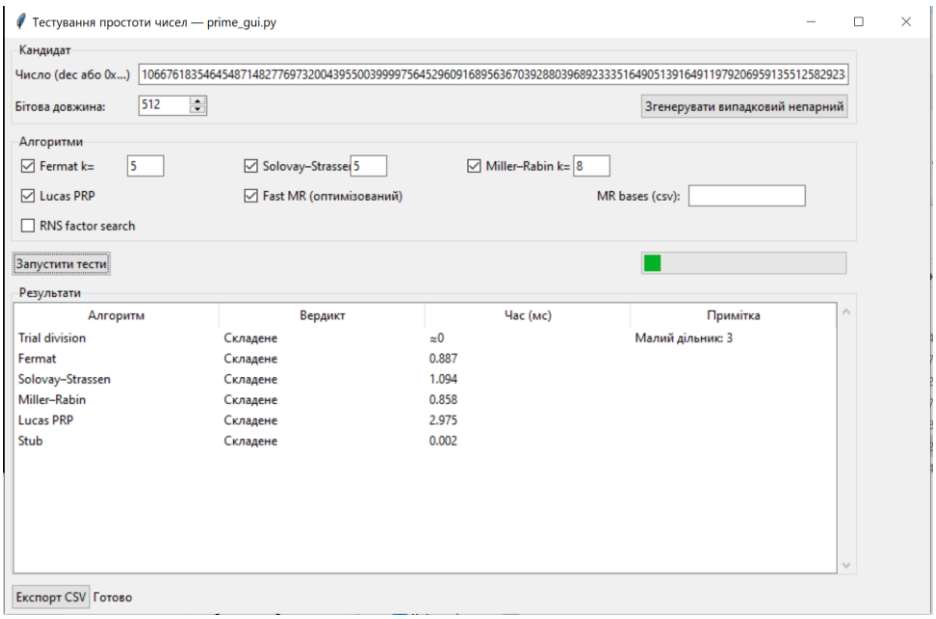

Рисунок 3.7 - Результати тестування 512-бітного випадкового числа

Розроблений програмний засіб дозволяє ефективно оцінити часові характеристики роботи класичних тестів простоти та запропонованих алгоритмів перевірки числа на простоту.

ВИСНОВКИ

У ході дослідження було встановлено, що проблема генерації простих чисел великої розрядності є фундаментальною для сучасної криптографії, оскільки саме такі числа забезпечують високий рівень стійкості криптографічних протоколів до атак, пов’язаних із факторизацією та обчисленням дискретних логарифмів. Аналіз наявних підходів показав, що генерація великих простих чисел ґрунтується на комбінації швидких імовірнісних тестів простоти, методів випадкового вибору кандидатів та використанні чисел спеціального виду, що дозволяє значно зменшити обчислювальне навантаження при створенні криптографічних ключів.

Проведене дослідження програмних бібліотек для тестування та генерації простих чисел дало змогу оцінити їхню ефективність з точки зору швидкодії, надійності та придатності для криптографічних застосувань. Встановлено, що найвищу продуктивність забезпечують бібліотеки, побудовані на основі високоефективної арифметики великих чисел (наприклад GMP, NTL або OpenSSL BN), тоді як символьні математичні пакети на кшталт SymPy є менш продуктивними, але зручними для дослідницьких цілей. Систематизація тестів простоти підтвердила доцільність їх поділу на детерміновані, імовірнісні та спеціалізовані алгоритми, що відрізняються за теоретичними властивостями, ступенем надійності та сферами практичного застосування.

Порівняльний аналіз тестів простоти засвідчив, що імовірнісні методи, такі як Міллер–Рабін та Соловей–Штрассен, демонструють найкраще співвідношення між точністю й швидкодією, тоді як детерміновані алгоритми, зокрема AKS, мають переважно теоретичне значення через свою високу обчислювальну складність. Розроблені у ході роботи функціональні модулі для тестування простоти чисел забезпечили можливість інтегрованого застосування цих алгоритмів та створили основу для проведення експериментальної перевірки їхньої ефективності.

У роботі реалізовано експериментальний алгоритм пошуку множників

числа на основі перебору залишків у системі залишкових класів, що дало змогу оцінити можливості такого підходу у контексті факторизаційних задач та перевірити його придатність для пошуку нетривіальних дільників. Крім того, створено оптимізований варіант тесту Міллера–Рабіна, який використовує фіксовані бази та попередній відсів малими простими числами, що забезпечило зменшення часу перевірки та підвищення надійності результатів для чисел середньої розрядності.

На основі розробленого програмного засобу проведено верифікацію коректності реалізованих алгоритмів і виконано експериментальну оцінку їх продуктивності. Результати тестування підтвердили відповідність реалізації теоретичним властивостям методів і продемонстрували достовірність отриманих результатів як з точки зору швидкодії, так і з точки зору надійності оцінки простоти. Таким чином, поставлені завдання виконано повною мірою, а отримані результати мають як теоретичну, так і практичну цінність для подальших досліджень у галузі криптографії та обчислювальної теорії чисел.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Andrijchuk V.A. Modern Algorithms and Methods of the Person Biometric Identification. Proceedings / V.A. Andrijchuk, I.P. Kuritnyk, M.M. Kasyanchuk, M.P. Karpinski // Third IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS–2005) – Sofia, Bulgaria. – 2005. – P.403–406.

2. D. Hankerson, A. Menezes, V. Scott. “Guide to Elliptic Curve Cryptography”. Springer-Verlag New York, USA, 2004, 311 p.

3. D. Kozaczko, I.Yakymenko, M. Kasianchuk, S. Ivasiev, “Vector Module Exponential in the Remaining Classes System”, Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS–2015), Warsaw, Poland. Vol. 1. pp.161–163, 2015.

4. D.Kozaczko, M.Kasianchuk, I.Yakymenko, S.Ivasiev. “Vector Module Exponential in the Remaining Classes System”. Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS–2015), Warsaw, Poland, V.1, September – 2015, p.161–163.

5. Fischer W. Note on fast computation of secret RSA exponents / W. Fischer, J.-P. Seifert // Information Security and Privacy (ACISP 2002), Vol. 2384 of Lecture Notes in Computer Science, Springer-Verlag, 2002, p. 136–143.

6. Hu Zhengbing. V. Yatskiv, A. Sachenko. “Increasing the Data Transmission Robustness in WSN Using the Modified Error Correction Codes on Residue Number System”. Elektronika ir Elektrotechnika. Vol 21, No 1, 2015, p. 76–81.

7. K. Gjosteen. “A new security proof for Damgard’s ElGamal”. In D. Pointcheval, editor, CT-RSA –2006, Vol. 3860, of Lecture Notes in Computer Science, Springer, p. 150–158

8. Karpiński M. Advanced method of factorization of multi-bit numbers based on Fermat's theorem in the system of residual classes / M. Karpiński, S. Ivasiev,

I. Yakymenko, M. Kasianchuk, T. Gancarczyk // Proc. of 16th International Conference on Control, Automation and Systems (ICCAS–2016), Gyeongju, Korea, V.1, October, 2016, p.1484–1486.

9. Kasianchuk M.N. Theory and Methods of Constructing of Modules System of the Perfect Modified Form of the System of Residual Classes / M.N. Kasianchuk, Ya.N. Nykolaychuk, I.Z. Yakymenko // Journal of Automation and Information Sciences. 2016, Vol.48, №8, p.56-63.

10. Kozaczko D. Vector Module Exponential in the Remaining Classes System / D.Kozaczko, M.Kasianchuk, I.Yakymenko, S.Ivasiev // Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS–2015). - Warsaw, Poland. - V.1, September – 2015. - P.161–163.

11. M. Joye, P. Pascal. “GCD-Free Algorithms for Computing Modular Inverses”. CHES 2003: Cologne, Germany, 2003, p. 243-253.

12. M. Karpiński, S. Ivasiev, I. Yakymenko, M. Kasianchuk, T. Gancarczyk. “Advanced method of factorization of multi-bit numbers based on Fermat's theorem in the system of residual classes”. Proc. of 16th International Conference on Control, Automation and Systems (ICCAS–2016), Gyeongju, Korea, V.1, October, 2016, p.1484–1486.

13. M. N Kasianchuk, Ya. N Nykolaychuk, I.Z. Yakymenko. “Theory and Methods of Constructing of Modules System of the Perfect Modified Form of the System of Residual Classes”. Journal of Automation and Information Sciences. 2016, Vol.48, №8, p.56-63.

14. M.M. Kasianchuk, Ya. M. Nykolaychuk, I. Z. Yakymenko “Theoretical Foundations of the Modified Perfect form of Residue Number System”. Cybernetics and Systems Analysis, 2016, p. 219-223.

15. Ma Q. An integrated framework for information security management / Q. Ma, B.M. Schmidt, J.M. Pearson // Review of Business. Retrieved – 26 October 2013.– pp.58–69.

16. N. Koblitz, A. Menezes. “Another look at non-standard discrete log and Diffie-Hellman problems”. Cryptology ePrint Archive. Report 2007/442, 2007. http://eprint.iacr.org/.

17. Nykolaychuk Ya. M. Theoretical Foundations of the Modified Perfect form of Residue Number System / Ya.M. Nykolaychuk, M.M. Kasianchuk, I.Z. Yakymenko // Cybernetics and Systems Analysis, 2016, p. 219-223.

18. Omondi, B. Premkumar. “Residue Number System: Theory and Implementation”. Imperial College Press, 2007,Vol. 2, 296 p.

19. Q. Ma, B. M. Schmidt, J. M. Pearson. “An integrated framework for information security management”.// Review of Business. Retrieved – 26 October 2013.– pp.58–69.

20. R. Lorencz. “New Algorithm for Classical Modular Inverse”. 4 th Cryptographic Hardware and Embedded Systems. International Workshop, 2002, p. 57-70.

21. Rajba T. Research of Time Characteristics of Search Methods of Inverse Element by the Module / T. Rajba, A. Klos-Witkowska, S. Ivasiev, I. Yakymenko, M. Kasianchuk // Proceedings of the 2017 IEEE 9th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS–2017) – Bucharest, Romania. – V.1. – September, 2017. – P.82–85.

22. S. Dasgupta, C. Papadimitriou, U. Vazirani “Algorithms”. McGraw-Hill Science, Engineering, Math; 1 edition, 13 September, 2006, 336 p.

23. Sachenko, V. Yatskiv, R. Krepych, A. Karachka. “Data Encoding in Residue Number System”. Proceeding of the International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: IDAACS‘2009, 2009, p. 679 – 681

24. Stallings W. Cryptography and Network Security: Principles and Practice / W. Stallings /. 5th Prentice Hall Press Upper Saddle River, NJ, USA.– 2010.–719 p.

25. Tsmots I. Basic Components of Neuronetworks with Parallel Vertical Group Data Real-Time Processing / I. Tsmots, V. Teslyuk, T. Teslyuk, I. Ihnatyev //

Advances in Intelligent Systems and Computing II. CSIT 2017. Advances in Intelligent Systems and Computing, vol. 689, Springer, Cham. – pp. 558 – 576.

26. V.K. Zadiraka, A.M. Kudin, V.O. Lyudvichenko, A.S. Oleksyuk, Computer technologies of cryptographic protection of information on the specific digital carriers, Textbooks and manuals, Kyiv - Ternopil, Ukraine, 2007.

27. V.K. Zadiraka, A.S. Oleksyuk Computer cryptology, Tanha, Ternopil, Ukraine, 2002.

28. W. Fischer, J.-P. Seifert. “Note on fast computation of secret RSA exponents”. Information Security and Privacy (ACISP 2002), Vol. 2384 of Lecture Notes in Computer Science, Springer-Verlag, 2002, p. 136–143.

29. W. Stallings. “Cryptography and Network Security: Principles and Practice”. 5th Prentice Hall Press Upper Saddle River, NJ, USA.– 2010.–719 p.

30. Ya. M. Nykolaychuk, M. M. Kasianchuk, I. Z. Yakymenko, “Theoretical Foundations of the Modified Perfect form of Residue Number System”, Cybernetics and Systems Analysis, Vol 52, No 2, pp.219-223.

31. Yakymenko I. Matrix Algorithms of Processing of the Information Flow in Computer Systems Based on Theoretical and Numerical Krestenson’s Basis / I.Yakymenko, M.Kasyanchuk, Ya. Nykolajchuk. // Proceedings of the X–th International Conference ”Modern Problems of Radio Engineering, Telecommunications and Computer Science” (TCSET–2010).–L’viv–Slavske.– 2010. – P.241.

Додаток А
Копії публікацій

ЗАХІДНОУКРАЇНСЬКИЙНАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КІБЕРБЕЗПЕКИ
ГРОМАДСЬКА ОРГАНІАЦІЯ «КІБЕРБЕЗПЕКА І АВТОМАТИЗАЦІЯ»

Матеріали
науково-практичного симпозіуму
"ЗАХИСТ'ІНФОРМАЦІЇ'2025"

28 листопада 2025
Тернопіль

Збірник матеріалів науково-практичного симпозіуму «Захист інформації'2025», Тернопіль, 2025. – 118с.

Адреса редакції:
Громадська організація «Кібербезпека і автоматизація»
м. Тернопіль
Контактний телефон: (066)043-42-10
e-mail: conferencekb@gmail.com

ЗМІСТ

УДК 004.056.5

Олександр ДРОЖАК

Західноукраїнський національний університет

ПОЛІНОМІАЛЬНИЙ АЛГОРИТМ ПЕРЕВІРКИ ЧИСЕЛ НА ПРОСТОТУ: ТЕСТ АГРАВАЛА–КАЯЛА–САКСЕНИ

Вступ. Тест простоти AKS є першим детермінованим поліноміальним алгоритмом, який доводить простоту натуральних чисел без використання додаткових гіпотез або випадковості. Його поява стала фундаментальним проривом у теорії чисел та алгоритмічній криптографії, оскільки забезпечила строгий математичний підхід до перевірки простоти. Алгоритм базується на властивостях многочленів у кільці $Z_n[x]$ та дозволяє гарантовано класифікувати число як просте чи складене. Попри значну теоретичну цінність, практичне застосування AKS обмежується його високою обчислювальною складністю порівняно з імовірнісними тестами. Дослідження алгоритму є важливим для глибшого розуміння меж детермінованих методів у криптографічних системах та сучасних обчислювальних моделях.

Метою є дослідження ефективності поліноміального алгоритму перевірки чисел на простоту, зокрема тесту Агралила–Каяла–Саксени. У роботі розглядаються теоретичні основи алгоритму, його математична обґрунтованість та порівняння з іншими методами перевірки простоти чисел, що дозволяє оцінити швидкість та точність його виконання на великих числах. Також вивчаються практичні аспекти застосування цього алгоритму для криптографічних задач та обчислювальних обмежень.

1. Всластивості AKS тесту простоти

Індійськими математиками Агравалом, Кайялом та Саксеною у роботі «PRIMES is in P» було вперше запропоновано алгоритм перевірки простоти чисел, який одночасно є поліноміальним, універсальним, детермінованим та безумовним. До цього були відомі алгоритми, які мали максимум три із чотирьох властивостей.

Поліноміальність означає, що складність алгоритму обмежена поліномом від кількості біт у числі. Прикладом тесту, який не задовольняє властивості поліноміальності, є тест Адлемана–Померанца–Румелі.

Універсальність має на увазі, що алгоритм застосовний до будь–яких чисел, а не лише до чисел спеціального вигляду. Прикладом неуніверсального алгоритму є тест Люка–Лемера, який можна застосовувати лише для чисел Мерсенна.

Детермінованість означає, що алгоритм завжди видає ту саму відповідь на одних і тих самих вхідних даних. Прикладом недетермінованого (імовірнісного) тесту може бути тест Міллера–Рабіна.

Безумовність – це незалежність від недоведених гіпотез. Чи не задовольняє властивості безумовності, наприклад, тест Міллера, який спирається на узагальнену гіпотезу Рімана.

Введемо позначення:

Zn – кільце відрахувань по модулю n.

Запис $(x) = h(x)(mod\ n)$ означає, що многочлени, коефіцієнти яких сприймаються як відрахування з Z_n , рівні.

Запис $g(x)=h(x)$ (mod x^r-1,n) означає, що багаточлени з коефіцієнтами Z_n рівні по модулю многочлена x^r-1.

$ord_r(n)$ – порядок числа n за модулем r. Мінімальна кількість k, таке що $n^k=1$ (mod r).

$\varphi(n)$ – функція Ейлера. Кількість натуральних чисел, менших n і взаємно простих із нею.

Log(n) – логарифм за основою два числа n.

2. Принцип роботи алгоритму

Основна ідея алгоритму спирається наступну теорему: якщо числа a і n взаємно прості, то тотожність $(x+a)^n=x^n+a$ (mod n) виконується тоді й лише тоді, коли n – просте.

Цю тотожність вже можна використовувати для перевірки простоти. Але алгоритм не буде поліноміальним, тому що потрібно зробити перевірок – за кількістю коефіцієнтів у поліномах. Ідея полягає в тому, щоб розглядати рівність поліномів за модулем іншого полінома, ступенем меншим, ніж n. З теореми випливає слідство:

Як наслідок: якщо n просте, то для будь–якого натурального r та будь–якого натурального $0 < a < n$ виконується $(x+a)^n=x^n+a$ (mod x^r-1,n).

Зворотне твердження у випадку неправильно. У статті індійських математиків доведено, що для деякої r і деякої множини значень a зворотне твердження буде вірним:

Якщо існує r, таке що $ord_r(n) > \log^2 n$, при якому тотожність

$$(x+a)^n=x^n+a \text{ (mod } x^r-1, n).$$

Виконано для всіх $1 \le a \le \lfloor \sqrt{\varphi(n)} \log n \rfloor$, то n або просто, або ступінь простого.

Це дозволяє побудувати поліноміальний алгоритм перевірки чисел на простоту, що є основним результатом роботи.

3. Алгоритм роботи тесту простоти AKS

На вхід алгоритму подається натуральне число $n > 1$.

Вихід: ПРОСТЕ число, якщо n – просте число. СКЛАДНЕ в іншому випадку. Якщо n – точний степінь деякого числа $(n = a^b, a, b \in N, b > 1)$, повернути СКЛАДНЕ.

Знайти найменше r, що $ord_r(n) > \log^2 n$. Якщо $1 <$ НСД(a, n) $< n$ для деякого $a \le r$, повернути СКЛАДНЕ. Якщо $n \le r$ повернути СКЛАДНЕ.

Для кожного $1 \le a \le \lfloor \sqrt{\varphi(n)} \log n \rfloor$: Якщо $(x+a)^n \ne x^n+a$ (mod x^r-1, n) повернути СКЛАДНЕ.

Повернути ПРОСТЕ.

Розглянемо докладно реалізацію кожного кроку алгоритму та розберемо виконання з прикладу числа 47.

На першому кроці слід визначити, чи є число n точним ступенем іншого числа, тобто $n=a^b$. Мабуть, найпростіший спосіб зробити це – перевірити, що

$\lfloor n^{1/b} \rfloor^b \neq n$. При цьому достатньо перевіряти лише значення $2 \leq b \leq log\, n$ (оскільки якщо , то b>log n).

Функція для виконання перевірки буде мати наступний вигляд:

```
function isPerfectPower(n)
{
   for (b = 2; b < ceil(log(n)); b++) {
      if (floor(n ** (1.0 / b)) ** b) == n:
         return true;
   }
   return false;
}
```

Розглянемо приклад :

$$\lfloor log\ 47 \rfloor = 5.$$

Перевіряємо від 2 до 4:

$$\lfloor 47^{1/2} \rfloor^2 = 36;$$
$$\lfloor 47^{1/3} \rfloor^3 = 27;$$
$$\lfloor 47^{1/4} \rfloor^4 = 16.$$

Отже, 47 не є точним ступенем іншого числа.

Відповідне *r* знаходиться перебором. Для кожного *r* потрібно:

Перевірити, що $n^k \neq 1 \pmod r$ для всіх $k \leq \lfloor log^2\ n \rfloor$.

Якщо одне із значень дорівнює одиниці – спробувати наступне *r*.

Якщо всі рівні одиниці – *r* знайдено.

Відомо, що шукане *r* має порядок не більше $O(\log^5 n)$.

Функція знаходження *r* буде складатись з циклу, котрий зупиниться при виконанні умови пошуку.

```
function findR(n) {
   r = 2;
   while (true) {
      find = true;
      for (k = 1; k <= ceil(log(n) ** 2); k++) {
         if (n ** k % r == 1) {
            find = false;
            break
         }
      }

      if (find) {
         return r;
      } else {
         r++;
      }
   }
}
```

Приклад:

$$\lfloor log\ ^2 47 \rfloor = 30.$$

Шукане $r = 41$:

$47^1 (\text{mod } 41)=6$

$47^2 (\text{mod } 41)=36$

…

$47^{30} (\text{mod } 41)=9$

Наступні кроки елементарні, потрібно лише реалізувати обчислення НСД двох чисел.

Досягнення теоретичної складності досить використовувати швидкі алгоритми для арифметичних операцій із многочленами. Тут розглянемо одну ефективну практично оптимізацію: перебування залишку від поділу без прямого поділу многочленів, використовуючи лише додавання.

Якщо многочлен ступеня r–1 зводиться у квадрат, то результаті виходить многочлен ступеня 2r–2:

$$f(x) = c_{2r-2}x^{2r-2}+\ldots+c_r x^r + c_{r-1}x^{r-1} + c_{r-1}x^{r-1}+\ldots+c_1 x + c_0.$$

Розглянемо поділ на x^r–1 $x^r - 1$:

$f(x) = a(x)(x^r - 1) + b(x)$, де

$a(x) = a_{r-2}x^{r-2}+\ldots+a_0$

$b(x) = b_{r-2}x^{r-2}+\ldots+b_0$

Розкриваючи дужки, знаходимо коефіцієнти b:

$b_0 = c_0 + a_0 = c_0 + c_r$

$b_1 = c_1 + a_1 = c_1 + c_{1+r}$

$b_{r-2} = c_{r-2} + a_{r-2} = c_{r-2} + c_{2r-2}$

```
function mod (p) {
  for (i = p.degree(); i >= r; i—){
    p[i – r] = p[i – r] + p[i];
    p[i] = 0;
  }
}
```

Приклад: $\left\lfloor \sqrt{\varphi(41) log\ 41} \right\rfloor = 36$

$$a = 1: (x+1)^{47} = x^{47} + 1\ (mod\ x^{41} - 1,47) = x^6 + 1$$

$$a = 2: (x+2)^{47} = x^{47} + 2\ (mod\ x^{41} - 1,47) = x^6 + 2$$

…

$$a = 36: (x+36)^{47} = x^{47} + 36\ (mod\ x^{41} - 1,47) = x^6 + 36$$

4. Оцінка складності

Введемо позначення: $\tilde{O}(t(n)) = O(t(n)) + poly(log(t(n)))$, де t(n) – Деяка функція від n.

При оцінках спиратимемося те що, що множення і розподіл m–бітних чисел мають складність $\tilde{O}(m)$. Відповідно, операції над поліномами ступеня d мають складність $\tilde{O}(dm)$. НСД двох m–бітних чисел можна знайти за $O(log\ n)$.

На першому етапі визначення, чи є число точним ступенем, має складність $\tilde{O}(log^3 n)$.

На другому кроці слід знайти r. Це робиться перебором можливих значень r

та перевіркою для всіх. Для кожного r це вимагає множення по модулю r, отже для кожного r складність $\check{O}(log^2 n\, log\, r)$. Враховуючи, що r має порядок $O(log^5 n)$, складність всього кроку дорівнює $\check{O}(log^7 n)$.

Третій крок вимагає обчислення НСД для r чисел. Загальна складність кроку $O(r\ log\ \ n) = O(\log^6 n)$.

На п'ятому кроці слід перевірити рівності $\lfloor\sqrt{\varphi(n)}\log n\rfloor$. Кожна рівність включає $O(log\ \ n)$ множення поліномів ступеня r, які мають коефіцієнти порядку $O(log\ \ n)$. Кожна рівність може бути перевірена за $\check{O}(log^2 n)$. Разом складність кроку $\check{O}(r\sqrt{\varphi(r)}\ log^3 n = \check{O}(r^{3/2}\ log^3 n) = \check{O}(log^{21/2} n)$. Цю оцінку можна покращити. Найкраща оцінка на даний момент $\check{O}(log^6 n)$. Оцінка складностей тестів простоти приведена в таблиці 1.

Таблиця 1 – Обчислювальні складності відомих тестів простоти

Тест	Тип	Ймовірність помилки	Орієнтовна складність	Придатність
Ферма	Ймовірнісний	до 1/2	O((log n)^3)	Теоретичний, попередній відсів
Соловей–Штрассен	Ймовірнісний	≤ 1/2	O((log n)^3)	Рідко використовується
Міллер–Рабін	Ймовірнісний	≤ (1/4)^k	O(k(log n)^3)	Основний у криптографії
Люка	Детермінований / ймовірнісний	0 / низька	O((log n)^3)	Додаткова перевірка
Люка–Лемера	Детермінований (для Мерсена)	0	O((log n)^2)	Спеціалізований, ефективний
AKS	Детермінований	0	O((log n)^6)	Теоретичний, не практичний

Порівняння обчислювальних складностей на логарифмічній шкалі можна зобразити на графіку як показано на рисунку 1.

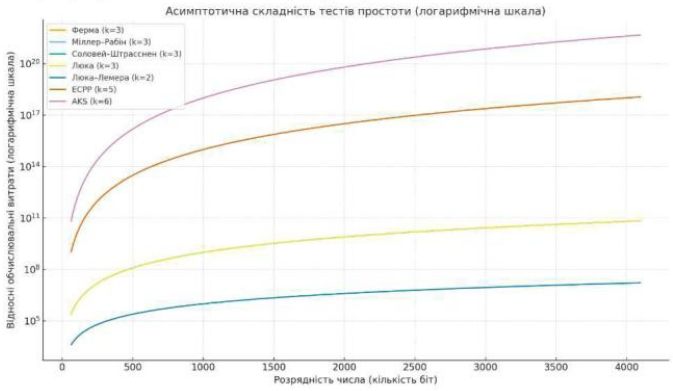

Рисунок 1 – Порівняння обчислювальних складностей тестів простоти на логарифмічній шкалі

Перспективи вдосконалення тесту простоти AKS доцільно розглядати у двох взаємопов'язаних площинах: теоретичній, що стосується асимптотичної складності алгоритму, та прикладній, пов'язаній із його практичною придатністю у криптографічних системах. Попри те, що AKS став фундаментальним результатом, який уперше довів можливість детермінованої перевірки простоти в поліноміальному часі без додаткових гіпотез, його базова форма залишається істотно повільнішою за ймовірнісні тести та спеціалізовані детерміновані методи для чисел певних класів. Саме тому подальші дослідження зосереджуються не стільки на зміні структури алгоритму, скільки на оптимізації його ключових компонентів, зокрема вибору параметра r, організації обчислень у кільці многочленів та скороченні кількості перевірок конгруенцій.

Висновок. Проведене дослідження дозволило встановити, що тест простоти AKS є ключовим теоретичним результатом у сучасній алгоритмічній теорії чисел, оскільки вперше продемонстрував можливість детермінованої поліноміальної перевірки простоти. Алгоритм AKS має важливе теоретичне значення, проте не застосовується практично. По–перше, оцінка складності містить поліном високого ступеня. По–друге, алгоритм вимогливий з пам'яті. Навіть при оцінці параметра для перевірки числа розміром 1024 біт знадобиться близько 1 гігабайта пам'яті. Його математична строгість і відсутність залежності від імовірнісних припущень роблять алгоритм фундаментальним елементом у вивченні меж обчислюваності та побудові надійних криптографічних методів. Разом із тим, експериментальна оцінка показує, що через високу практичну складність AKS поступається сучасним стохастичним тестам за швидкодією та ефективністю, що обмежує його застосування в реальних системах. Незважаючи на це, він залишається важливим еталоном для розробки нових детермінованих методів і поглибленого аналізу алгоритмів перевірки простоти. Отримані результати підтверджують значення AKS як теоретичної основи для подальших досліджень у сфері криптографії та обчислювальної теорії чисел.

Перелік використаних джерел.

1. J.P. Buhler Algorithmic Number Theory: Proc. ANTS–III – Portland, OR, v.1423, Lect.Not.Comp.Sci. Springer–Verlag, 1998, 640 p.

2. D. Venturi Lecture Notes on Algorithmic Number Theory. – Springer–Verlag, New–York, Berlin, 2009, 217 p.

3. Sh.T. Ishmukhametov Methods of factorization of natural numbers: a tutorial.– Kazan, Kazan University, 2011, 190 p.

4. Ya.M.Nikolaichuk, Kasianchuk M.M., Yakymenko I.Z., Ivasiev S.V Vector and modular method of multiplication of multidigit numbers in RademacherKrestenson basis. Herald of the National University "Lviv Polytechnic" "Computer systems and networks", no. 694, 2014, pp. 118–125.

5. M.Kasyanchuk, I. Yakymenko, Y.Nykolajchuk. Matrix Algorithm of Processing of the Information Flow in Computer Systems Based on Theoretical and Numerical Krestenson's Based. Proceedings of the Integrational Conference TCSET'2010, February 23–27, 2010, p. – C: 241

ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
ГАЛИЦЬКИЙ ФАХОВИЙ КОЛЕДЖ ІМ. В'ЯЧЕСЛАВА ЧОРНОВОЛА

КІБЕРБЕЗПЕКА ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ (КБКІТ – 2025)

науково-практична конференція
молодих вчених, аспірантів та студентів

28–29 серпня 2025
Тернопіль

Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп'ютерно-інтегровані технології» (КБКІТ - 2025), Тернопіль, 2025. - 154 с.

Адреса редакції:
Західноукраїнський національний університет, кафедра кібербезпеки,
вул. Олени Теліги 8, м. Тернопіль 46003
Контакти:
*e-mail:*conferencekb@gmail.com

УДК 004.056.5

Олександр ДРОЖАК

Західноукраїнський національний університет

АНАЛІЗ ТЕСТІВ ПРОСТОТИ ФЕРМА ТА МІЛЛЕРА-РАБІНА

Вступ. Аналіз тестів простоти числа є важливим аспектом у теорії чисел та криптографії, оскільки ефективне визначення простоти числа має ключове значення для безпеки сучасних криптографічних алгоритмів. Розвиток швидких і точних методів тестування простоти чисел дозволяє знижувати обчислювальні витрати при роботі з великими числами, що є критичним для практичних застосувань, таких як генерація ключів у криптографії.

Актуальність таких тестів також зростає з урахуванням потреб у безпечному зберіганні даних та захисті інформації в цифрову епоху.

Метою аналізу тестів простоти Ферма та Тесту Міллера-Рабіна є оцінка їх ефективності, точності та надійності при визначенні простоти великих чисел.

1. Тест Ферма

Тест Ферма дозволяє швидко виявити складні числа, але може давати хибнопозитивні результати, що робить його менш надійним для великих чисел.

За теоремою Ферма, якщо n – просте число, тоді будь-якого a справедливо така рівність

$$a^{n-1}=1 \pmod{n}.$$

Звідси ми можемо вивести правило тесту Ферма на перевірку простоти числа: візьмемо випадкове

$$a \in \{1, ..., n-1\}$$

і перевіримо чи дотримуватиметься рівність

$$a^{n-1}=1 \pmod{n}.$$

Якщо рівність недотримується, отже швидше за все n – складове.

Проте умова рівності може бути дотримано, навіть якщо n – не просте. Наприклад, візьмемо $n = 561 = 3 \times 11 \times 17$. Відповідно до Китайської теореми про залишки:

$$Q_{561} = Q_3 \times Q_{11} \times Q_{17},$$

де кожне $a \in Q^*_{561}$ відповідає наступному:

$$(x,y,z) \in Q^*_3 \times Q^*_{11}11 \times Q^*_{17}.$$

По теоремі Ферма

$$x^2=1,\ y^{10}=1 \text{ і } z^{16}=1.$$

Оскільки 2, 10 і 16 всі є дільниками 560, це означає, що $(x, y, z)^{560} = (1, 1, 1)$, тобто $a^{560} = 1$ для будь-якого $a \in Q^*_{561}$.

Не має значення яке ми виберемо, 561 завжди буде проходити тест Ферма незважаючи на те, що воно складене, доки a є взаємно простим з n. Такі числа називаються числами Кармайкла і встановлено, що їх існує безліч.

Якщо a не взаємно просте з n, воно тест Ферма не проходить, але в цьому випадку ми можемо відмовитися від тестів і продовжити шукати дільники n,

обчислюючи НСД(a,n).

2. Тест Міллера-Рабіна

Тест Міллера-Рабіна, у свою чергу, є більш точним і менш схильний до помилок, що робить його одним з найбільш використовуваних методів для перевірки простоти в криптографії. Аналіз цих тестів сприяє вибору оптимального алгоритму для застосувань, де важлива швидкість та точність перевірки простоти чисел.

Можна вдосконалити тест, сказавши, що n - просте тоді і тільки тоді, коли рішеннями

$$x^2 = 1 \pmod{n} \text{ є } x = \pm 1.$$

Таким чином, якщо n проходить тест Ферма, тобто

$$a^n - 1 = 1,$$

тоді ми ще перевіряємо щоб

$$a^{(n-1)/2} = \pm 1,$$

оскільки

$$a^{(n-1)/2}$$

це квадратний корінь 1.

На жаль, такі числа, як, наприклад 1729 - третє число Кармайкла, досі можуть обдурити цей покращений тест. Можливим вдосконаленням буде проведення ітерацій. Тобто поки це буде можливо, зменшуватимемо експоненту вдвічі, доки не дійдемо до якогось числа, крім 1. Якщо ми отримаємо в результаті щось, крім -1, тоді n буде складним. Якщо говорити формальніше, то нехай 2^S буде найбільшим ступенем 2, що ділиться на n-1, тобто

$$n-1=2^S q$$

для якогось непарного числа q.

Кожне число із послідовності

$$a^{n-1} = a^{(2\text{^}S)q}, a^{(2\text{^}S\text{-}1)q},\ldots, aq.$$

Це квадратний корінь попереднього члена послідовності.

Тоді якщо n – просте число, то послідовність повинна починатися з 1 і кожне наступне число теж має бути 1, або перший член послідовність може бути не дорівнює 1, але тоді він дорівнює -1.

Тест Міллера-Рабін бере випадкове $a \in Z_n$. Якщо вищезазначена послідовність не починається з 1, або перший член послідовності не дорівнює 1 або -1, тоді n - не просте.

Виявляється, що для будь-якого складеного n, включаючи числа Кармайкла, можливість пройти тест Міллера-Рабіна дорівнює приблизно 1/4. (У середньому значно менше.) Таким чином, ймовірність того, що n пройде декілька прогонів тесту, зменшується експонентно.

Якщо n не проходить тест Міллера-Рабіна з послідовністю, що починається з 1, тоді у нас з'являється нетривіальний квадратний корінь з 1 по модулю n, і ми можемо ефективно знаходити дільники n. Тому числа Кармайкла завжди зручно розкладати на множники.

Коли тест застосовується до чисел виду pq, де p і q - великі прості числа,

вони не проходять тест Міллера-Рабіна практично у всіх випадках, оскільки послідовність не починається з 1.

На практиці тест Міллера-Рабіна реалізується так:

Дано n, потрібно знайти s, що

$$n - 1 = 2^S q$$

для деякого непарного q.

Візьмемо випадкове

$$a \in \{1,...,n-1\}$$

Якщо $a^q = 1$, n проходить тест і припиняємо виконання. Для $i = 0, \ldots, s-1$ перевірити рівність

$$a^{(2\wedge i)q} = -1.$$

Якщо рівність виконується, то n проходить тест (припиняємо виконання).Якщо жодна з вищенаведених умов не виконана, то n – складене.

Перед виконанням тесту Міллера-Рабін варто провести ще кілька тривіальних поділів на маленькі прості числа. Строго кажучи ці тести є тестами на те чи вважається число складеним, оскільки вони не доводять по суті, що число просте, що перевіряється, але точно доводять, що воно може виявитися складовим.

Існують ще детерміновані алгоритми, які працюють за поліноміальний час для визначення простоти (Agrawal, Kayal і Saxena), проте на сьогоднішній день вони вважаються непрактичними.

Висновок. Аналіз тестів простоти Ферма та Міллера-Рабіна показує, що кожен з них має свої переваги та обмеження. Тест Ферма є швидким, але може давати хибнопозитивні результати для псевдопростих чисел, що обмежує його надійність при перевірці великих чисел. Тест Міллера-Рабіна, в свою чергу, є більш точним і менш схильним до помилок, тому його часто використовують як основний метод у криптографії. Враховуючи це, оптимальним є використання тесту Міллера-Рабіна в поєднанні з іншими методами для досягнення високої точності при визначенні простоти чисел.

Перелік використаних джерел.

1. J.P. Buhler Algorithmic Number Theory: Proc. ANTS-III – Portland, OR, v.1423, Lect.Not.Comp.Sci. Springer–Verlag, 1998, 640 p.

2. D. Venturi Lecture Notes on Algorithmic Number Theory. – Springer-Verlag, New-York, Berlin, 2009, 217 p.

3. Sh.T. Ishmukhametov Methods of factorization of natural numbers: a tutorial.– Kazan, Kazan University, 2011, 190 p.

4. Ya.M.Nikolaichuk, Kasianchuk M.M., Yakymenko I.Z., Ivasiev S.V Vector and modular method of multiplication of multidigit numbers in RademacherKrestenson basis. Herald of the National University "Lviv Polytechnic" "Computer systems and networks", no. 694, 2014, pp. 118–125.

5. M.Kasyanchuk, I. Yakymenko, Y.Nykolajchuk. Matrix Algorithm of Processing of the Information Flow in Computer Systems Based on Theoretical and Numerical Krestenson's Based. Proceedings of the Integrational Conference TCSET'2010, February 23-27, 2010, p. – C: 241

Додаток Б.

Код програмного засобу

```
from __future__ import annotations

import csv
import queue
import random
import threading
import time
from math import gcd, prod
from tkinter import (
    Tk, StringVar, IntVar, BooleanVar,
    END
)
from tkinter import ttk, filedialog, messagebox



SMALL_PRIMES = [
    3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
    37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
    79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
    131, 137, 139, 149, 151, 157, 163, 167, 173, 179,
    181, 191, 193, 197, 199
]


FIRST_100_PRIMES = [
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
    127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
    233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
    283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
    353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
    419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
    467, 479, 487, 491, 499, 503, 509, 521, 523, 541
]




def small_prime_trial(n: int):
    """
    Швидкий відсів по малих простих:
    повертає (is_ok, divisor):
      - якщо знайдено малий дільник -> (False, p)
      - якщо малих дільників немає -> (True, None)
    """
    if n < 2:
```

```
        return False, None
    if n in (2,):
        return True, None
    if n % 2 == 0:
        return (n == 2), (2 if n != 2 else None)
    for p in SMALL_PRIMES:
        if n == p:
            return True, None
        if n % p == 0:
            return False, p
    return True, None


def parse_int(text: str) -> int:

    t = text.strip().lower().replace('_', '')
    if not t:
        raise ValueError("порожній рядок")
    if t.startswith('0x'):
        return int(t, 16)
    return int(t, 10)


def rand_odd_bits(bits: int, rng: random.Random) -> int:

    if bits < 2:
        bits = 2
    n = rng.getrandbits(bits)
    n |= (1 << (bits - 1))
    n |= 1
    return n




def jacobi(a: int, n: int) -> int:

    if n <= 0 or n % 2 == 0:
        raise ValueError("n must be a positive odd integer")
    a %= n
    result = 1
    while a != 0:
        while a % 2 == 0:
            a //= 2
            if n % 8 in (3, 5):
                result = -result
        a, n = n, a
        if a % 4 == 3 and n % 4 == 3:
            result = -result
        a %= n
    return result if n == 1 else 0


def crt(remainders, moduli):
```

```
    M = prod(moduli)
    x = 0
    for a_i, m_i in zip(remainders, moduli):
        M_i = M // m_i
        inv = pow(M_i, -1, m_i)
        x += a_i * M_i * inv
    return x % M


def choose_primes_for_n(n: int, max_primes: int = 100):

    bits = max(1, n.bit_length())

    k = max(8, bits // 4)
    k = min(max_primes, k)
    return FIRST_100_PRIMES[:k]



def fermat(n: int, k: int = 5, rng: random.Random | None = None) -> bool:

    if n < 4:
        return n in (2, 3)
    if n % 2 == 0:
        return False
    r = rng or random
    for _ in range(k):
        a = r.randrange(2, n - 1)
        if gcd(a, n) != 1:
            return False
        if pow(a, n - 1, n) != 1:
            return False
    return True


def solovay_strassen(n: int, k: int = 5, rng: random.Random | None = None) -> bool:

    if n < 4:
        return n in (2, 3)
    if n % 2 == 0:
        return False
    r = rng or random
    for _ in range(k):
        a = r.randrange(2, n - 1)
        if gcd(a, n) != 1:
            return False
        x = jacobi(a, n)
        if x == 0:
            return False
        if pow(a, (n - 1) // 2, n) != x % n:
            return False
    return True
```

```
    def miller_rabin(n: int, k: int = 8, rng: random.Random | None = None, bases=None) ->
bool:

        if n < 4:
            return n in (2, 3)
        if n % 2 == 0:
            return False


        d = n - 1
        s = 0
        while d % 2 == 0:
            d //= 2
            s += 1

        def check(a: int) -> bool:
            x = pow(a, d, n)
            if x == 1 or x == n - 1:
                return True
            for _ in range(s - 1):
                x = (x * x) % n
                if x == n - 1:
                    return True
            return False

        if bases is not None:
            for a in bases:
                if a % n == 0:
                    continue
                if not check(a % n):
                    return False
            return True

        r = rng or random
        for _ in range(k):
            a = r.randrange(2, n - 2)
            if not check(a):
                return False
        return True


    def lucas_prp(n: int) -> bool:

        if n < 2:
            return False
        if n in (2, 3):
            return True
        if n % 2 == 0:
            return False


        D = 5
```

```
    sign = 1
    while True:
        j = jacobi(D, n)
        if j == -1:
            break
        D = (abs(D) + 2) * (-1 if sign > 0 else 1)
        sign *= -1

    P = 1
    Q = (1 - D) // 4

    def lucas_double_add(k: int, P: int, Q: int, mod: int):

        U, V = 0, 2
        U1, V1 = 1, P
        kbin = bin(k)[2:]
        for bit in kbin:

            U2 = (U1 * V1) % mod
            V2 = (V1 * V1 - 2 * Q) % mod
            U1, V1 = U2, V2
            if bit == '1':
                U, V = (U * V1 + U1 * V) % mod, (V * V1 + P * U * U1) % mod
        return U % mod, V % mod

    U, V = lucas_double_add(n + 1, P, Q, n)
    return U == 0


def stub_method(n: int) -> bool:


    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0:
        return False


    ok, div = small_prime_trial(n)
    if not ok:

        return False




    if n < 2_152_302_898_747:

        bases = [2, 3, 5, 7, 11]
        return miller_rabin(n, bases=bases)
```

```
    if n < 3_474_749_660_383:

        bases = [2, 7, 61]
        return miller_rabin(n, bases=bases)

    if n < 2**64:

        bases = [2, 325, 9375, 28178, 450775, 9780504, 1795265022]
        return miller_rabin(n, bases=bases)



    return miller_rabin(n, k=5)



def experimental_rns_factor_first(n: int,
                                  max_candidates_per_mod: int = 5) -> int | None:

    if n <= 3:
        return None

    primes = choose_primes_for_n(n, max_primes=100)
    residues = [n % p for p in primes]

    allowed_residues_per_mod = []
    for r_i, p_i in zip(residues, primes):
        candidates = []
        for d_i in range(1, p_i):
            if r_i % d_i == 0:
                candidates.append(d_i)
            if len(candidates) >= max_candidates_per_mod:
                break
        if not candidates:
            candidates = [1]
        allowed_residues_per_mod.append(candidates)

    found = [None]

    def backtrack(idx, current_vector):
        if found[0] is not None:
            return
        if idx == len(primes):
            d = crt(current_vector, primes)
            if 1 < d < n and n % d == 0:
                found[0] = d
            return
        for d_i in allowed_residues_per_mod[idx]:
            current_vector.append(d_i)
            backtrack(idx + 1, current_vector)
            current_vector.pop()

    backtrack(0, [])
```

```
    return found[0]


def rns_factor_test(n: int) -> tuple[bool, str]:

    factor = experimental_rns_factor_first(n)
    if factor is not None:
        return False, f"Знайдено дільник: {factor}"
    else:
        return True, "Дільник не знайдено (в межах поточного перебору)"




class PrimeGUI:
    def __init__(self, root: Tk):
        self.root = root
        self.root.title("Тестування простоти чисел — prime_gui.py")


        self.number_var = StringVar()
        self.bits_var = IntVar(value=256)


        self.fermat_k = IntVar(value=5)
        self.ss_k = IntVar(value=5)
        self.mr_k = IntVar(value=8)
        self.mr_bases_var = StringVar(value="")


        self.use_fermat = BooleanVar(value=True)
        self.use_ss = BooleanVar(value=False)
        self.use_mr = BooleanVar(value=True)
        self.use_lucas = BooleanVar(value=False)
        self.use_stub = BooleanVar(value=False)
        self.use_rns = BooleanVar(value=False)

        self._build_ui()

        self.queue = queue.Queue()
        self.worker = None

    def _build_ui(self):
        pad = {"padx": 6, "pady": 4}


        frm_top = ttk.LabelFrame(self.root, text="Кандидат")
        frm_top.grid(row=0, column=0, sticky="ew", **pad)
        frm_top.columnconfigure(1, weight=1)

        ttk.Label(frm_top, text="Число (dec або 0x...)").grid(row=0, column=0, sticky="w")
        ttk.Entry(frm_top, textvariable=self.number_var, width=70).grid(
            row=0, column=1, sticky="ew", **pad
        )
```

```
        ttk.Label(frm_top, text="Бітова довжина:").grid(row=1, column=0, sticky="w")
        ttk.Spinbox(frm_top, from_=8, to=4096, textvariable=self.bits_var, width=8).grid(
            row=1, column=1, sticky="w", **pad
        )
        ttk.Button(
            frm_top,
            text="Згенерувати випадковий непарний",
            command=self.generate_candidate
        ).grid(row=1, column=1, sticky="e", **pad)


        frm_alg = ttk.LabelFrame(self.root, text="Алгоритми")
        frm_alg.grid(row=1, column=0, sticky="ew", **pad)
        for i in range(4):
            frm_alg.columnconfigure(i, weight=1)

        ttk.Checkbutton(frm_alg, text="Fermat k=", variable=self.use_fermat).grid(
            row=0, column=0, sticky="w", **pad
        )
        ttk.Entry(frm_alg, textvariable=self.fermat_k, width=5).grid(
            row=0, column=0, sticky="e", padx=70
        )

        ttk.Checkbutton(frm_alg, text="Solovay–Strassen k=", variable=self.use_ss).grid(
            row=0, column=1, sticky="w", **pad
        )
        ttk.Entry(frm_alg, textvariable=self.ss_k, width=5).grid(
            row=0, column=1, sticky="e", padx=70
        )

        ttk.Checkbutton(frm_alg, text="Miller–Rabin k=", variable=self.use_mr).grid(
            row=0, column=2, sticky="w", **pad
        )
        ttk.Entry(frm_alg, textvariable=self.mr_k, width=5).grid(
            row=0, column=2, sticky="e", padx=70
        )

        ttk.Label(frm_alg, text="MR bases (csv):").grid(row=1, column=2, sticky="e")
        ttk.Entry(frm_alg, textvariable=self.mr_bases_var, width=18).grid(
            row=1, column=3, sticky="w", **pad
        )

        ttk.Checkbutton(frm_alg, text="Lucas PRP", variable=self.use_lucas).grid(
            row=1, column=0, sticky="w", **pad
        )
        ttk.Checkbutton(frm_alg, text="Fast MR (оптимізований)",
variable=self.use_stub).grid(
            row=1, column=1, sticky="w", **pad
        )
        ttk.Checkbutton(frm_alg, text="RNS factor search", variable=self.use_rns).grid(
            row=2, column=0, sticky="w", **pad
        )
```

```
    frm_ctrl = ttk.Frame(self.root)
    frm_ctrl.grid(row=2, column=0, sticky="ew", **pad)
    frm_ctrl.columnconfigure(0, weight=1)

    ttk.Button(frm_ctrl, text="Запустити тести", command=self.run_tests).grid(
        row=0, column=0, sticky="w"
    )
    self.progress = ttk.Progressbar(frm_ctrl, mode="indeterminate", length=200)
    self.progress.grid(row=0, column=1, sticky="e", padx=10)


    frm_tbl = ttk.LabelFrame(self.root, text="Результати")
    frm_tbl.grid(row=3, column=0, sticky="nsew", **pad)
    self.root.rowconfigure(3, weight=1)
    frm_tbl.rowconfigure(0, weight=1)
    frm_tbl.columnconfigure(0, weight=1)

    self.tree = ttk.Treeview(
        frm_tbl,
        columns=("alg", "verdict", "time", "note"),
        show="headings"
    )
    self.tree.heading("alg", text="Алгоритм")
    self.tree.heading("verdict", text="Вердикт")
    self.tree.heading("time", text="Час (мс)")
    self.tree.heading("note", text="Примітка")
    self.tree.grid(row=0, column=0, sticky="nsew")

    sb = ttk.Scrollbar(frm_tbl, orient="vertical", command=self.tree.yview)
    self.tree.configure(yscrollcommand=sb.set)
    sb.grid(row=0, column=1, sticky="ns")


    frm_bottom = ttk.Frame(self.root)
    frm_bottom.grid(row=4, column=0, sticky="ew", **pad)
    ttk.Button(frm_bottom, text="Експорт CSV", command=self.export_csv).grid(
        row=0, column=0, sticky="w"
    )
    self.status = ttk.Label(frm_bottom, text="Готово")
    self.status.grid(row=0, column=1, sticky="e")



def generate_candidate(self):

    try:
        bits = int(self.bits_var.get())
        n = rand_odd_bits(bits, random.Random())
        self.number_var.set(str(n))
    except Exception as e:
        messagebox.showerror("Помилка", f"Не вдалося згенерувати число: {e}")

def run_tests(self):
```

```
        for i in self.tree.get_children():
            self.tree.delete(i)
        self.status.configure(text="")


        try:
            n = parse_int(self.number_var.get())
        except Exception:
            messagebox.showerror(
                "Помилка вводу",
                "Введіть коректне число (десяткове або 0x...)."
            )
            return


        ok, div = small_prime_trial(n)
        if not ok and div is not None:
            self.tree.insert(
                "",
                END,
                values=("Trial division", "Складене", "≈0", f"Малий дільник: {div}")
            )
            self.status.configure(
                text="Знайдено малий дільник — інші тести можна все одно виконати."
            )


        tests = []

        if self.use_fermat.get():
            k = int(self.fermat_k.get())
            tests.append(("Fermat", lambda x, kk=k: fermat(x, k=kk)))

        if self.use_ss.get():
            k = int(self.ss_k.get())
            tests.append(("Solovay–Strassen", lambda x, kk=k: solovay_strassen(x, k=kk)))

        if self.use_mr.get():
            k = int(self.mr_k.get())
            bases_txt = self.mr_bases_var.get().strip()
            if bases_txt:
                try:
                    bases = [int(a) for a in bases_txt.split(",") if a.strip()]
                except Exception:
                    messagebox.showerror("Помилка", "Невірний список баз для MR.")
                    return
                tests.append(("Miller–Rabin[bases]", lambda x, b=bases: miller_rabin(x,
bases=b)))
            else:
                tests.append(("Miller–Rabin", lambda x, kk=k: miller_rabin(x, k=kk)))

        if self.use_lucas.get():
```

```
            tests.append(("Lucas PRP", lambda x: lucas_prp(x)))

        if self.use_stub.get():
            tests.append(("Stub", lambda x: stub_method(x)))

        if self.use_rns.get():
            tests.append(("RNS factor search", lambda x: rns_factor_test(x)))

        if not tests:
            messagebox.showwarning("Немає тестів", "Оберіть хоча б один алгоритм.")
            return


        self.progress.start(12)
        self.worker = threading.Thread(
            target=self._worker_run,
            args=(n, tests),
            daemon=True
        )
        self.worker.start()
        self.root.after(100, self._poll_queue)

    def _worker_run(self, n: int, tests):

        for name, fn in tests:
            t0 = time.perf_counter()
            try:
                res = fn(n)
                note = ""


                if isinstance(res, tuple):
                    base_res, note = res
                else:
                    base_res = res


                if name.startswith("Lucas"):
                    verdict = "PRP" if base_res else "Складене"
                elif name.startswith("RNS factor"):
                    verdict = "Фактор не знайдено" if base_res else "Складене"
                else:
                    verdict = "Ймовірно просте" if base_res else "Складене"

                ms = (time.perf_counter() - t0) * 1000.0
                self.queue.put(("row", (name, verdict, f"{ms:.3f}", note)))
            except Exception as e:
                self.queue.put(("row", (name, "Помилка", "—", str(e))))
        self.queue.put(("done", None))

    def _poll_queue(self):

        try:
            while True:
```

```
                msg, payload = self.queue.get_nowait()
                if msg == "row":
                    self.tree.insert("", END, values=payload)
                elif msg == "done":
                    self.progress.stop()
                    self.status.configure(text="Готово")
            except queue.Empty:
                pass

            if self.worker and self.worker.is_alive():
                self.root.after(100, self._poll_queue)

        def export_csv(self):

            rows = [self.tree.item(i, "values") for i in self.tree.get_children()]
            if not rows:
                messagebox.showinfo("Експорт CSV", "Немає даних для експорту.")
                return

            fname = filedialog.asksaveasfilename(
                title="Зберегти результати",
                defaultextension=".csv",
                filetypes=[("CSV files", "*.csv"), ("All files", "*.*")],
                initialfile="prime_gui_results.csv",
            )
            if not fname:
                return

            with open(fname, "w", newline="", encoding="utf-8") as f:
                w = csv.writer(f)
                w.writerow(["Algorithm", "Verdict", "Time_ms", "Note"])
                for r in rows:
                    w.writerow(list(r))
            messagebox.showinfo("Експорт CSV", f"Збережено: {fname}")


    def main():
        root = Tk()
        gui = PrimeGUI(root)
        root.geometry("900x560")
        root.minsize(760, 420)
        root.mainloop()


    if __name__ == "__main__":
        main()
```