Лабораторна робота №2

Скоробагатько Максим, ФБ-31мн

Тема: Реалізація алгоритмів генерації ключів гібридних криптосистем.

Мета: Дослідження алгоритмів генерації псевдовипадкових послідовностей, тестування простоти чисел та генерації простих чисел з точки зору їх ефективності за часом та можливості використання для гененерації ключів асиметричних криптосистем.

Схема генератора ПВЧ для User Endpoint terminal.

Насправді реалізувати генератор простих великих чисел (ПВЧ) надзвичайно просто в тому плані, що нам достатньо вибрати якесь велике число і перевірити його на простоту.

Тож, нехай ми маємо задане число N порядку більше 10^{10} і менше 10^{35}, і ми хочемо отримати просте число $p: p >= N$. Отже маємо 2 основні підходи побудови такого алгоритму:

1. Спочатку застосуємо один із сіткових-алгоритмів для отримання всіх простих чисел до заданого N – решето Ератосфена, решето Аткина, решето Сундарама;

2. Другий підхід - це генерація рандомних непарних чисел більше N і перевірка кожного з них на простоту через перебір дільників, імовірнісні та детерміновані тести простоти (краще ймовірностні тести).

Ми спробуємо інший підхід, а саме скомбінувано максимальну простоту (решето Ератосфена) і критерій Поклінгтона (https://www.rieselprime.de/ziki/Pocklington%27s_theorem), який використовує МТФ (мала теорема Ферма) для отримання однозначно простого числа.

Алгоритм наступний:

1. Будуємо решето Ератосфена до $k = const$. Вибираємо початкове значення s-просте.
2. Якщо $s > N$:

   return s

   інакше 3.
3. Обрати випадкове r: $s <= r <= 2 * 2(2s + 1)$, де $r -$ парне, і отримуємо $n = s * r + 1$
4. Перевіряємо n на подільність простими числами низького порядку, отриманими на 1. - якщо число ділиться на одне з них, воно складене і повертаємося 2. Інакше число може бути простим, тому переходимо до 5.
5. Вибираємо випадкове число a: $1 < a < n$ та перевіряємо для n наступні умови:
   a. $a^{n-1} \equiv 1 (mod n)$
   b. $gcd(a^r - 1, n) = 1$

Якщо обидва виконуються, то за критерієм Поклінгтона число n – просте. Замінюємо $s := n$ і переходимо до 2.Інакше якщо не виконується перша умова, то за малою теоремою Ферма число n не є простим, тому переходимо до вибору но
вого n, тобто 3.
Інакше якщо не виконується друга умова, то ми знайшли дільник d: $1 < d <= n$ для n, оскільки $gcd(a^r - 1, n) == d$.. Якщо $d \neq n$, то d – не простий дільник, отже n не просте. Необхідно знову йти на 3.

Останній випадок, коли $d == n$, що означає $a^r \equiv 1 (mod n)$, а рішень цієї конгруенції існує не більше r. Одне з рішень це $a == 1$, тому на інтервалі $1 < a < n$ існує не більше $r - 1$ рішень, отже при дійсно простому n ми знайдемо (з ймовірністю $1 - O(s^{-1}))$) таке a, яке б задовольняло критерію Поклінгтона, тому переходимо до 5. для повторення вибору a.

Реалізація алгоритму на Python

```
import random
import math

def getSieve(limit):
    # Generate a list of prime numbers up to a given limit using the Sieve of 
Eratosthenes
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False  # 0 and 1
    for start in range(2, int(math.sqrt(limit)) + 1):
        if sieve[start]:
            for multiple in range(start * start, limit + 1, start):
                sieve[multiple] = False
    return [num for num, is_prime in enumerate(sieve) if is_prime]

def is_probably_prime(n, primes):
    #Check if a number is probably prime using trial division with known primes
    if n < 2:
        return False
    for prime in primes:
        if prime * prime > n:
            break
        if n % prime == 0:
            return False
    return True
```

```
def generatePrime(n: int, primes=None, s=None):
    #Generate a large prime number with n digits
    up_limit = 10**n
    if not primes:
        primes = getSieve(1000)
    if not s:
        s = primes[-1]

    while s < up_limit:
        lo, hi = (s + 1) >> 1, (s << 1) + 1

        while True:
            r = random.randint(lo, hi) << 1
            candidate = s * r + 1
            if not is_probably_prime(candidate, primes):
                continue

            while True:
                a = random.randint(2, candidate - 1)
                if pow(a, candidate - 1, candidate) != 1:
                    break

                d = math.gcd((pow(a, r, candidate) - 1) % candidate, candidate)
                if d != candidate:
                    if d == 1:
                        s = candidate
                    break
            if s == candidate:
```

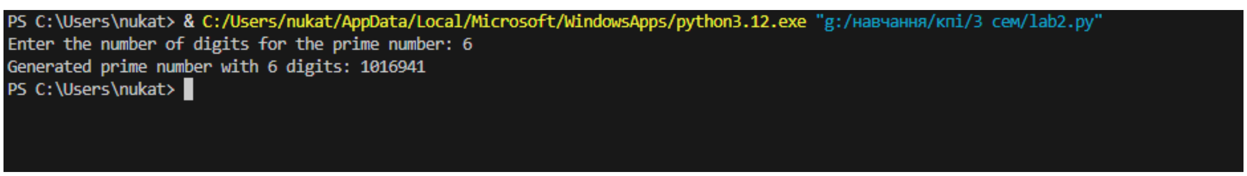
```
            break
    return s

def main():
    n = int(input("Enter the number of digits for the prime number: "))
    prime_number = generatePrime(n)
    print(f"Generated prime number with {n} digits: {prime_number}")

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

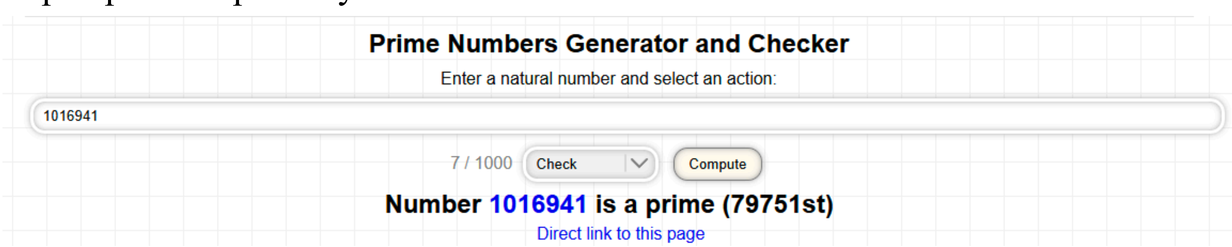
Результати роботи:

Генерація числа

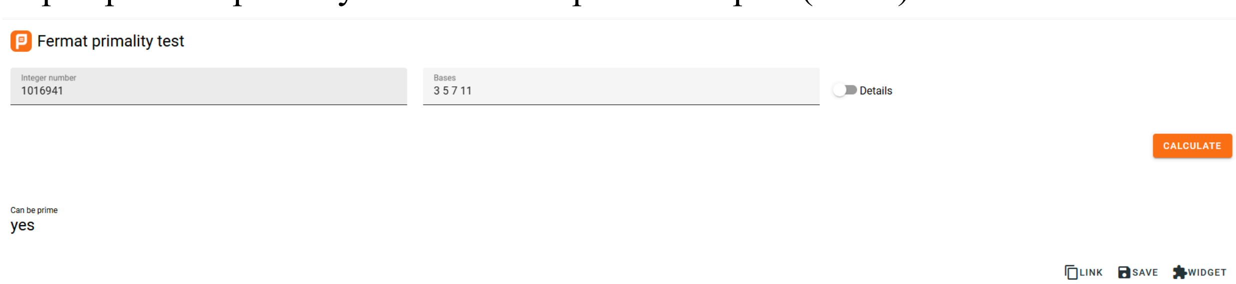


```
PS C:\Users\nukat> & C:/Users/nukat/AppData/Local/Microsoft/WindowsApps/python3.12.exe "g:/навчання/кпі/3 сем/lab2.py"
Enter the number of digits for the prime number: 6
Generated prime number with 6 digits: 1016941
PS C:\Users\nukat>
```

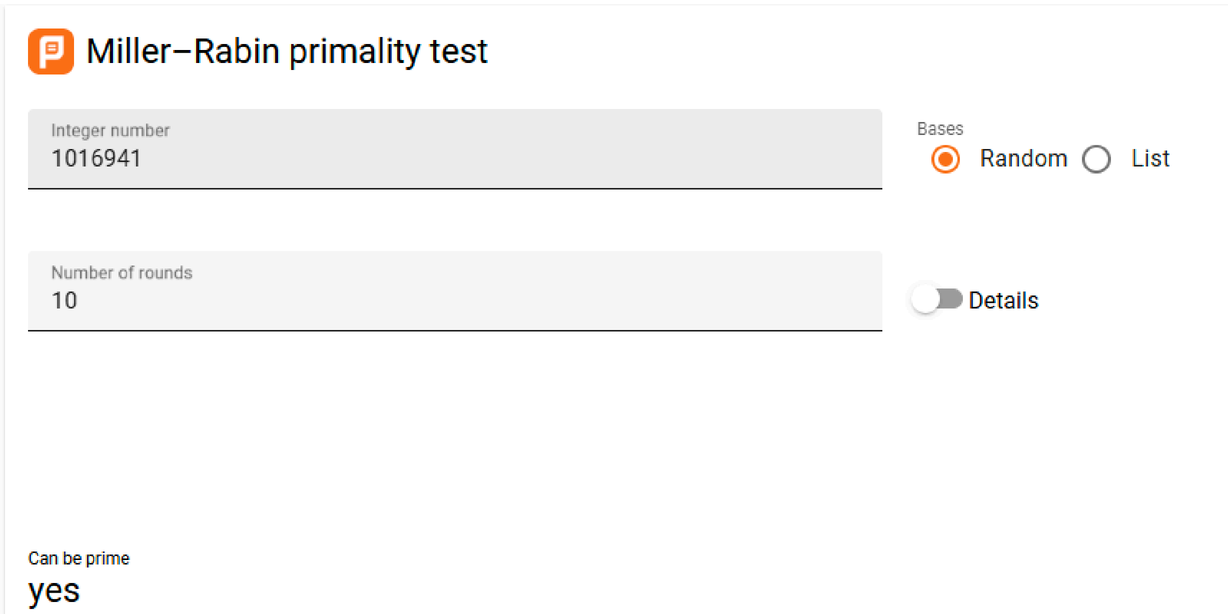
Перевірка на простоту в базі:



Перевірка на простоту за малою теоремою Ферма (МТФ):



Перевірка на простоту імовірнісним алгоритмом Міллера-Рабіна:



Також є наявні реалізації імовірністних тестів простоти, а саме алгоритм Міллера-Рабіна та Соловея-Штрасена, які наведені на Github: https://github.com/user3719431/tna_lab1/

Висновок:

В даній роботі було реалізовано алгоритм генерації великих простих чисел з використанням комплексного підходу. Також було продемонстровано імовірнісні алгоритми перевірки чисел на простоту (алгоритм Міллера-Рабіна та алгоритм Соловея-Штрасена), які були раніше реалізовані в рамках іншої дисципліни.

Комбінований підхід реалізації алгоритму генерації забезпечив швидшу швидкість роботи завдяки використанню математичного підґрунтя до роботи з простими числами, а використання підходу алгоритму "сито Ератосфена" забезпечило більшу ймовірність генерації не складеного числа.