НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

КОМП'ЮТЕРНИЙ ПРАКТИКУМ № 2

з дисципліни:

«МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИЙ МЕХАНІЗМІВ»

Реалізація алгоритмів генерації ключів гібридних криптосистем
варіант 1А

Виконала:
Студентка групи ФІ-22мн
Калитюк Дар'я

КИЇВ 2022

Мета роботи: 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.

Постановка задачі: дослідити різні методи генерації випадкових послідовностей для засобів обчислювальної техніки. Дослідити ефективність за часом алгоритми тестування на простоту різних груп – імовірнісних, гипотетичних та детермінованих. Порівняти ймовірність похибки різних імовірнісних тестів (Ферма, Соловея-Штрассена та Мілера-Рабіна з різною кількістю ітерацій) з ймовірністю похибки при виконанні обчислень на ПЕОМ. Розглянути алгоритми генерації простих чисел "Чебишова" та Маурера та провести порівняльний аналіз їх складності. Розробити бібліотеку генерації псевдовипадкової послідовності, тестування простоти чисел та генерації простих чисел для Intel-сумісних ПЕОМ. Розмірність чисел – 768, 1024 біт.

Хід роботи

Для порівняльного аналізу мною було обрано наступні генератори псевдовипадкових чисел:

a) Лінійний конгруентний генератор Лемера (дві модифікації):

$$x_{n+1} = ((2^{16}+1)\cdot x_n + 119) \bmod 2^{32},\ x_0 - \text{випадкове 32-бітне число.}$$

- ***LehmerLow*** в якості *n*-того вихідного значення повертає молодші 8 біт числа x_n;
- ***LehmerHigh*** повертає старші 8 біт числа x_n;

b) Генератори псевдовипадкових двійкових послідовностей ***L20***:

$$x_t = x_{t-3} \oplus x_{t-5} \oplus x_{t-9} \oplus x_{t-20}, x_t \in \{0,1\},$$

та ***L89****:*

$$x_t = x_{t-38} \oplus x_{t-89}, x_t \in \{0,1\}.$$

c) Генератор ***Geffe:***

$$z_i = s_i x_i \oplus (1 + s_i) y_i,$$

де

$$x_{11} = x_0 \oplus x_2,\ y_9 = y_0 \oplus y_1 \oplus y_3 \oplus y_4,\ s_{10} = s_0 \oplus s_3.$$

d) Генератор Вольфрама:

$$x_i = r_i \bmod 2,$$
$$r_{i+1} = (r_i <<< 1) \oplus (r_i \vee (r_i >>> 1)).$$

e) Генератор Блюма-Мікалі ***BM*** (бітовий і байтовий):

$$T_{i+1} = a^{T_i} \bmod p, x_i = \begin{cases} 1, T_i < \dfrac{p-1}{2}, \\ 0, T_i \geq \dfrac{p-1}{2}. \end{cases}, \ 0 \leq T_0 \leq p-1.$$

Для байтової модифікації:

$$x_i = k: \quad \frac{k(p-1)}{256} < T_i \leq \frac{(k+1)(p-1)}{256},$$

де a – генератор групи Z_p^*.

f) Генератор Блюм-Блюма-Шуба ***BBC***(бітовий і байтовий):

$$r_i = r_{i-1}^2 \bmod n,$$
$$x_i = r_i \bmod 2 \ (x_i = r_i \bmod 256),$$

де $n = pq, p$ і q – різні великі прості числа виду $4k+3$.

g) Вбудований в Python генератор, що використовує Вихор Мерсена.

Оцінка якості генераторів

Розглянемо послідовність $\{X_i\}$, де кожна X_i є випадковою величиною, що приймає набір значень із алфавіту Z.

- ✓ Послідовність $\{X_i\}$ задовільняє умові рівноймовірності знаків, якщо кожна X_i розподілена рівноімовірно на Z.
- ✓ Послідовність $\{X_i\}$ задовільняє умові незалежності знаків, якщо імовірність прийняти деяке значення для X_i не залежить від того, які значення прийняли $X_1, X_2, \dots, X_{i-1}$.
- ✓ Послідовність $\{X_i\}$ задовольняє умові однорідності, якщо для довільної реалізації вибірковий розподіл, одержаний на всій послідовності, буде співпадати із вибірковим розподілом, одержаним на довільній її підпослідовності достатньої довжини.

Сформулюємо критерії Пірсона для кожної з наведених умов.

❖ *Критерій перевірки рівноімовірності знаків*

- ➢ $\{X_i\}, i = 1, \dots, m. H_0 -$ байти послідовності рівноімовірні.
- ➢ Обчислюємо v_i – кількість байтів i, що спостерігається, $n_i = \frac{m}{256}$ і статистику

$$\chi^2 = \sum_{i=0}^{255} \frac{(v_i - n_i)^2}{n_i}$$

- Обчислюємо граничне значення $\chi^2_{1-\alpha} = \sqrt{2l} Z_{1-\alpha} + l, l = 255, Z_{1-\alpha} -$ $(1-\alpha)$-квантиль стандартного нормального розподілу.
- Якщо $\chi^2 \leq \chi^2_{1-\alpha}$, приймаємо гіпотезу H_0, інакше відхиляємо.

❖ *Критерій перевірки незалежності знаків*

- Розглядаємо пари $(X_{2i-1}, X_{2i}), i = 1, \dots, \left[\frac{m}{2}\right]$. $H_0 -$ байти послідовності не залежать від попереднього значення.
- Обчислюємо v_{ij} – кількість пар (i, j), що спостерігається, $n = \sum_{i,j=0}^{255} v_{ij}, v_i = \sum_{j=0}^{255} v_{ij}, \alpha_j = \sum_{i=0}^{255} v_{ij}$ і статистику

$$\chi^2 = n \sum_{i=0}^{255} \frac{v_{ij}^2}{v_i \alpha_j} - n$$

- $\chi^2_{1-\alpha} = \sqrt{2l} Z_{1-\alpha} + l,\ l = 255^2.$
- Якщо $\chi^2 \leq \chi^2_{1-\alpha}$, приймаємо гіпотезу H_0, інакше відхиляємо.

❖ *Критерій перевірки однорідності*

- Послідовність $\{X_i\}$ розбивається на r відрізків довжиною $m' = \left[\frac{m}{r}\right]$, де m – загальне число байтів.
- Обчислюємо v_{ij} – кількість байтів i, що спостерігається у відрізку j, $v_i = \sum_{j=0}^{r-1} v_{ij}, \alpha_j = \sum_{i=0}^{255} v_{ij}$ і статистику

$$\chi^2 = n \sum_{i=0}^{255} \sum_{j=0}^{r-1} \frac{v_{ij}^2}{v_i \alpha_j} - n$$

- $\chi^2_{1-\alpha} = \sqrt{2l} Z_{1-\alpha} + l,\ l = 255(r-1).$
- Якщо $\chi^2 \leq \chi^2_{1-\alpha}$, приймаємо гіпотезу H_0, інакше відхиляємо.

Реалізація і тестування генераторів

Для реалізації генератора ***BM*** було використано наступні параметри:

```
p =
CEA42B987C44FA642D80AD9F51F10457690DEF10C83D0BC1BCEE12F
C3B6093E3;
a =
5B88C41246790891C095E2878880342E88C79974303BD0400B090FE
38A688356;
q =
675215CC3E227D3216C056CFA8F8822BB486F788641E85E0DE77097
E1DB049F1.
```

Для ***BBS***:

```
p = D5BBB96D30086EC484EBA3D7F9CAEB07;
q = 425D2B9BFDB25B9CF6C416CC6E37B59C1F.
```

	DEFAULT GENERATOR	LEHMER LOW	LEHMER HIGH	L20	L89	GEFFE	WOLFRAM	BIUMA-MIKALA	BYTES BIUMA-MIKALA	BBS	BYTES BBS
Equiprobability test											
χ^2	225.248	0.0	46.268	252.137	287.459	284.105	631.062	262.113	258.387	217.965	231.277
$\chi^2_{0.99}$	307.536	307.536	307.536	307.536	307.536	307.536	307.536	307.536	307.536	307.536	307.536
	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓
$\chi^2_{0.95}$	292.146	292.146	292.146	292.146	292.146	292.146	292.146	292.146	292.146	292.146	292.146
	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓
$\chi^2_{0.9}$	283.942	283.942	283.942	283.942	283.942	283.942	283.942	283.942	283.942	283.942	283.942
	✓	✓	✓	✓	-	-	-	✓	✓	✓	✓
Independence test											
χ^2	64612.221	16646144.0	63507.798	61202.556	65046.54	81823.212	152971.196	64858.447	65517.194	64705.833	65676.027
$\chi^2_{0.99}$	65863.938	65863.938	65863.938	65863.938	65863.938	65863.938	65863.938	65863.938	65863.938	65863.938	65863.938
	✓	-	✓	✓	✓	-	-	✓	✓	✓	✓
$\chi^2_{0.95}$	65618.174	65618.174	65618.174	65618.174	65618.174	65618.174	65618.174	65618.174	65618.174	65618.174	65618.174
	✓	-	✓	✓	✓	-	-	✓	✓	✓	-
$\chi^2_{0.9}$	65487.159	65487.159	65487.159	65487.159	65487.159	65487.159	65487.159	65487.159	65487.159	65487.159	65487.159
	✓	-	✓	✓	✓	-	-	✓	-	✓	-
Homogeneity test											
χ^2	4904.65	15.949	4531.211	4916.75	5428.716	4747.734	4598.837	4745.969	4672.141	4897.481	4777.359
$\chi^2_{0.99}$	5074.001	5074.001	5074.001	5074.001	5074.001	5074.001	5074.001	5074.001	5074.001	5074.001	5074.001
	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓
$\chi^2_{0.95}$	5006.916	5006.916	5006.916	5006.916	5006.916	5006.916	5006.916	5006.916	5006.916	5006.916	5006.916
	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓
$\chi^2_{0.9}$	4971.153	4971.153	4971.153	4971.153	4971.153	4971.153	4971.153	4971.153	4971.153	4971.153	4971.153
	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓

Рис 1. Результати тестування генераторів для $m = 2^{21}$.

Оберемо п'ятірку найякісніших генераторів, а саме ***Вихор Мерсена***, ***LehmerHigh***, ***L20***, ***BM*** і ***BBS***, та дослідимо час їх роботи.

	2^8	2^9	2^{10}	2^{12}	2^{21}
DEFAULT GENERATOR	894 ns ± 41.3 ns	1.48 µs ± 22.4 ns	2.33 µs ± 72.9 ns	7.5 µs ± 79.5 ns	5.12 ms ± 103 µs
LEHMER HIGH	24.7 µs ± 1.16 µs	49 µs ± 2.23 µs	98.3 µs ± 4.01 µs	375 µs ± 15.1 µs	203 ms ± 11 ms
L20	323 µs ± 32.1 µs	642 µs ± 45.2 µs	1.24 ms ± 46.3 µs	4.73 ms ± 229 µs	1.26 s ± 99.5 ms
BIUMA-MIKALA	3.42 ms ± 125 µs	6.8 ms ± 82.7 µs	13.7 ms ± 211 µs	54.2 ms ± 909 µs	27.5 s ± 158 ms
BBS	265 µs ± 3.88 µs	544 µs ± 17.3 µs	1.1 ms ± 29.5 µs	4.33 ms ± 186 µs	2.19 s ± 8.39 ms

Насправді, лише два з цих генераторів – ***LEHMER HIGH*** і ***BBS***, затверджені Національним інститутом стандартів і технологій як ті, що є криптографічно застосовними. Очевидно, що з цих двох генераторів для застосування в смарт картці, токені або смартфону краще підійде ***LEHMER HIGH***, адже він є значно швидшим.

Тестування простоти

Розглянемо наступні тести простоти:

1. Імовірнісні
    1.1. Ферма
    1.2. Соловея-Штрассена
    1.3. Мілера-Рабіна
2. Детерміновані
    2.1 Тест Агравала-Каяла-Саксени (не використовується)
    2.2 Тест Адлемана-Померанса-Румелі-Коена-Лейнстри
    2.3 Тест Аткіна-Морейна на еліптичних кривих
3. Гіпотетичні
    3.1 Тест Фібоначчі

Тести Ферма, Соловея-Штрассена, Мілера-Рабіна а також тест Фібоначчі ми імпортуємо з бібліотеки ***gmpy2***, що була розглянута у лабораторній роботі № 1. Реалізацію тесту Адлемана-Померанса-Румелі-Коена-Лейнстри можна знайти за наступним посиланням: https://github.com/wacchoz/APR_CL, тесту на еліптичних кривих: https://github.com/root-z/ECPP.

Гіпотетичні тести засновані на теоретичних твердженнях, які вважаються істиними, але не є доведеними. Прикладом такого тесту є тест Фібоначчі:

якщо $p \equiv a \pmod{x^2+4}$, де а – квадратичний нелишок $\mathrm{mod}(\, x^2+4)$, тоді p буде простим при виконанні наступних умов:

$$2^{p-1} \equiv 1 \pmod{p}$$
$$f(1)_{p+1} \equiv 0 \pmod{p},$$

де $f(x)_k$— поліном Фібоначчі. Автори припущення пропонують 620 доларів за контрприклад, який досі не було знайдено.

Найбільш часто вживаними на практиці є імовірнісні тести, адже детерміновані тести часто мають дуже велику просторову складність, як наприклад тест Агравала-Каяла-Саксени, а тест на еліптичних кривих, що вважається найкращим детермінованих алгоритмом, має невідомий час виконання в найгіршому випадку.

Так, наприклад, тест Адлемана-Померанса-Румелі-Коена-Лейнстри для $p = 2^{1279} - 1$ працює 175 секунд, у той час як тест Міллера-Рабіна працює 0.8 секунд з імовірністю помилки 4^{-1000}. Дослідимо час роботи різних імовірнісних тестів для різної кількості ітерацій при $p = 2^{2203} - 1$:

	$k = 10$	$k = 100$	$k = 1000$	**Pr mistake**
Ferma test	38.1 ms ± 91.8 µs	382 ms ± 2.32 ms	3.84 s ± 14.9 ms	2^{-k}
Solovay Strassen test	38.3 ms ± 185 µs	384 ms ± 1.2 ms	3.87 s ± 30.6 ms	2^{-k}
Miller Rabin test	38 ms ± 120 µs	383 ms ± 2.48 ms	3.83 s ± 39.8 ms	4^{-k}

Як видно, час роботи алгоритмів однаковий, проте імовірність похибки в тесті Мілера-Рабіна набагато менша, отже, як правило, майже усюди використовується саме цей тест. Наведемо його теоретичний опис:

i. $p - 1 = d \cdot 2^s, iter = 0$.

ii. $x \in_R (1, p)$. Якщо $\gcd(x, p) > 1$: p – складене.

iii.

iii.1. Якщо $x^d mod\, p = \pm 1$, то p – сильно псевдопросте за основою x, інакше

iii.2. Для $r = \overline{1, s-1}$: $x_r = x^{d \cdot 2^r} mod\, p$. Якщо $x_r = -1$, то p – сильно псевдопросте за основою x; zкщо $x_r = 1$, то p – не сильно псевдопросте за основою x, завершуємо роботу; Інакше продовжуємо цикл.

iii.3. Якщо за кроки iii.1., iii.2. не було встановлено сильну псевдопростоту: p – складене. Інакше $iter = iter + 1$.

iv. $iter < k$: переходимо до кроку ii. Інакше p – просте.

Генерація простих чисел

Згідно тереми Чебишева про розподіл простих чисел кількість простих чисел на інтервалі $[1, N]$ дорівнює приблизно $\pi(N) = \frac{N}{lnN}$. Для генерації простого числа треба вибрати довільно непарне T і перевірити його на простоту: якщо воно виявиться складеним, покласти $T = T + 2$, повернутися на попередній крок. В середньому, через $\log_2 T$ кроків просте число буде знайдено.

Іншим вибором може бути алгоритм Маурера, що заснований на критерії Поклингтона, ще один детермінованим тестом на простоту, який, однак, вимагає пошук простого дільника $p - 1$, що може призвести до зведення до задачі факторизації.

У цій роботі для реалізації генератора простих чисел нам стане у нагоді функція `next_prime(...)` з модуля ***gmpy2*** .

Результати

В результаті проведених досліджень було створено бібліотеку, в якій реалізовано алгоритм генерації псевдопростих послідовностей, алгоритм генерації простих чисел і алгоритм перевірки чисел на простоту. Для 1024х біт маємо наступний приклад роботи алгоритмів:

```
  Generate pseudo random sequence:
  [176, 226, 21, 72, 123, 175, 228, 25, 78, 132, 186, 241, 40,
96, 152, 208, 10, 67, 125, 184, 243, 46, 106, 166, 227, 32, 94,
156, 219, 26, 90, 154, 218, 27, 93, 159, 225, 36, 103, 171, 239,
52, 121, 191, 5, 75, 146, 218, 34, 106, 179, 253, 70, 145, 219,
39, 114, 190, 11, 88, 166, 244, 66, 145, 225, 48, 129, 210, 35,
117, 199, 25, 108, 192, 20, 105, 190, 19, 105, 191, 22, 110,
197, 30, 118, 207, 41, 131, 222, 57, 148, 240, 77, 169, 7, 101,
195, 34, 129, 224, 65, 161, 2, 100, 198, 40, 139, 239, 82, 183,
28, 129, 231, 77, 179, 26, 130, 234, 83, 188, 37, 143, 249, 100,
207, 59, 167, 20]
  Generate prime number:
  163094488016872970995812521509254242402864528791340098014588
058343505095045945781385433768893956435314606436573990365208123
792162700668784511368085721360013514872541147546735246930480835
975698720886163312384784915744409561768212902976235161721421
917647621272084768488967301591594311029348316354374629469945091
  Check it's primality:
  True
```

Висновки: мною було досліджено одинадцять алгоритмів генерації випадкових послідовностей, для кожного з якого було застосовано три тести перевірки якості генераторів. Також було оглянуто і досліджено різні групи тестів простоти, реалізовано імовірнісні тести простоти засобами бібліотеки ***gmpy2***. Було розглянуто алгоритми генерації простих чисел “Чебишова” та Маурера, розроблено бібліотеку генерації псевдовипадкової послідовності, тестування простоти чисел та генерації простих чисел для 1024 біт.

Додаток А
Лістинг програм

PRNGs.py

```
from collections import Counter
import numpy as np
import random
import gmpy2
import time

alphas = [0.01, 0.05, 0.1]
quantiles = {0.01: 2.326347874, 0.05: 1.644853627, 0.1: 1.281551566}

L20 = [0, 11, 15, 17]
L89 = [37, 88]
#Bluma-Mikala
A = 0x5B88C41246790891C095E2878880342E88C79974303BD0400B090FE38A688356
Q = 0x675215CC3E227D3216C056CFA8F8822BB486F788641E85E0DE77097E1DB049F1
P = 0xCEA42B987C44FA642D80AD9F51F10457690DEF10C83D0BC1BCEE12FC3B6093E3
#BBS
p = 0xD5BBB96D30086EC484EBA3D7F9CAEB07
q = 0x425D2B9BFDB25B9CF6C416CC6E37B59C1F
#Lehmer
A_L = 2**16 + 1
M_L = 2**32
C_L = 119

BITS = 2**21
BYTES = 2**18

def python_generator(k):
    return random.getrandbits(k)

def generate_state(k):
    result = []
    for i in range(k):
        result.append(random.choice([0, 1]))
    if result == k*[0]:
        result = generate_state(k)
    return result

def convert_to_int(vec):
    return int(''.join(map(str, vec)), 2)

def LehmerLow(a, m ,c, lim = BYTES):
    x0 = random.getrandbits(32)
    res = []
    for i in range(lim):
        x1 = (a*x0 + c)%m
        b = bin(x1)[2:]
        res.append(('0'*(32 - len(b)) + b)[-8:])
        x0 = x1
    return [int(i, 2) for i in res]

def LehmerHigh(a, m, c, lim = BYTES):
    x0 = random.getrandbits(32)
```

```
    res = []
    for i in range(lim):
        x1 = (a*x0 + c)%m
        b = bin(x1)[2:]
        res.append(('0'*(32 - len(b)) + b)[:8])
        x0 = x1
    return [int(i, 2) for i in res]

def lfsr(state, taps, n, lim = BITS):
    res = []
    it = 0
    state0 = state
    while True:
        it += 1
        res += [state[0]]
        state = state[1:] + [sum(state[i] for i in taps)%2]
        if state == state0 or it == lim:
            break
    return res

def Geffe():
    res=[]
    x0 = generate_state(11)
    y0 = generate_state(9)
    s0 = generate_state(10)
    x=lfsr(x0, [0, 2], 11)
    y=lfsr(y0, [0, 1, 3, 4], 9)
    s=lfsr(s0, [0, 3], 10)
    for i in range(len(y)):
        res=res+[x[i] if s[i]==1 else y[i]]
    return res

def Wolfram(lim = BITS):
    r0 = generate_state(32)
    res = []
    for i in range(lim):
        res.append(r0[-1])
        f = r0[1:] + r0[:1]
        s = r0[-1:] + r0[:-1]
        sf = [0 if s[i]==r0[i]==0 else 1 for i in range(32)]
        r = [(sf[i]+f[i])%2 for i in range(32)]
        r0 = r
    return res

def Blum_Mikal(a, q, p, lim = BITS):
    x = []
    T0 = random.randint(0, p-1)
    for i in range(lim):
        T1 = gmpy2.powmod(a, T0, p)
        x.append(1 if T1 < (p-1)/2 else 0)
        T0 = T1
    return x

def Blum_Mikal_bytes(a, q, p, lim = BYTES):
    x = []
    T0 = random.randint(0, p-1)
    b = gmpy2.div((p-1), 256)
    for i in range(lim):
```

```
        T1 = gmpy2.powmod(a, T0, p)
        k = 0
        while gmpy2.mul(k, b) >= T1 or gmpy2.mul(k + 1, b) < T1:
            k+=1
        k = bin(k)[2:]
        x.append('0'*(8 - len(k)) + k)
        T0 = T1
    return [int(i, 2) for i in x]

def BBS(p, q, lim = BITS):
    n = p*q
    r0 = random.randint(2, n)
    x = []
    for i in range(lim):
        r1 = (r0**2)%n
        x.append(r1%2)
        r0 = r1
    return x

def BBS_bytes(p, q, lim = BYTES):
    n = p*q
    r0 = random.randint(2, n)
    x = []
    for i in range(lim):
        r1 = (r0**2)%n
        b = bin(r1%256)[2:]
        x.append('0'*(8 - len(b))+b)
        r0 = r1
    return [int(i, 2) for i in x]

def Chebyshev_prime(length):
    T = random.getrandbits(length)
    if gmpy2.is_prime(T):
        return T
    return gmpy2.next_prime(T)

def check_hypothesis(hi, l):
    print('   χ2 = ', hi)
    for alpha in alphas:
        hi2_alpha = np.sqrt(2*l)*quantiles[alpha] + l
        if hi <= hi2_alpha:
            print(f'   Test passed with α = {alpha}, χ2_(1-α) = {hi2_alpha}')
        else:
            print(f'   Test failed with α = {alpha}, χ2_(1-α) = {hi2_alpha}')

def equiprobability_test(nums, l = 255):
    n = len(nums)/256
    vi = []
    for j in range(256):
        vi.append(nums.count(j))
    hi2 = sum(((vi[i] - n)**2)/n for i in range(256))
    check_hypothesis(hi2, l)

def independence_test(nums, l = 65025):
    n = int(len(nums)/2)
    pairs = [(nums[2*i], nums[2*i - 1]) for i in range(n)]
    v = np.zeros((256, 256))
    unique_pairs = Counter(pairs)
```

```
    for pair in unique_pairs:
        v[pair[0]][pair[1]] = unique_pairs[pair]
    vi = [sum(v[i][j] for j in range(256)) for i in range(256)]
    alpha = [sum(v[i][j] for i in range(256)) for j in range(256)]
    hi2 = 0
    for i in range(256):
        for j in range(256):
            if vi[i]*alpha[j]!=0:
                hi2 += ((v[i][j]**2)/(vi[i]*alpha[j]))
    hi2 = n*(hi2 - 1)
    check_hypothesis(hi2, l)

def homogeneity_test(nums, r = 20):
    m_ = int(len(nums)/r)
    n = m_*r
    l = 255*(r - 1)
    strings = []
    for i in range(0, len(nums), m_):
        strings.append(nums[i: i + m_])
    v = np.zeros((256, r))
    for i in range(256):
        for j in range(r):
            v[i][j] = strings[j].count(i)
    vi = [sum(v[i][j] for j in range(r)) for i in range(256)]
    alpha = m_
    hi2 = 0
    for i in range(256):
        for j in range(r):
            if vi[i] != 0 :
                hi2 += (v[i][j]**2)/(vi[i]*alpha)
    hi2 = n*(hi2 - 1)
    check_hypothesis(hi2, l)

def group_to_bytes(vect):
    if len(vect)%8!=0:
        vect = '0'*(8 - len(vect)%8) + vect
    result = []
    for i in range(0, len(vect), 8):
        result.append(int(vect[i:i+8], 2))
    return result

def conv(lst):
    return group_to_bytes(''.join([str(i) for i in lst]))

def main():
    start_time = time.time()

    geffe = Geffe()
    while len(geffe) < BITS:
        geffe += Geffe()

    gens = {'DEFAULT GENERATOR' :
group_to_bytes(bin(python_generator(BITS))[2:]),
            'LEHMER LOW' : LehmerLow(A_L, M_L, C_L),
            'LEHMER HIGH' : LehmerHigh(A_L, M_L, C_L),
            'L20' : conv(lfsr(generate_state(20), L20, 20)),
            'L89' : conv(lfsr(generate_state(89), L89, 89)),
            'GEFFE' : conv(geffe),
```

```
            'WOLFRAM' :  conv(Wolfram()),
            'BlUMA-MIKALA' : conv(Blum_Mikal(A, Q, P)),
            'BYTES BlUMA-MIKALA' : Blum_Mikal_bytes(A, Q, P),
            'BBS' : conv(BBS(p, q)),
            'BYTES BBS' : BBS_bytes(p, q)
            }

    print('Generation time: %s seconds ' % (time.time() - start_time))
    start_time = time.time()

    for gen in gens:
        print(f'\n           {gen}\n')
        print('1. Equiprobability test: ')
        equiprobability_test(gens[gen])
        print('2. Independence test: ')
        independence_test(gens[gen])
        print('3. Homogeneity test: ')
        homogeneity_test(gens[gen])

    print('Testing time: %s seconds ' % (time.time() - start_time))

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

primality_tests.py

```
import time
import gmpy2
import random
from APR_CL import APRtest

N = 2**2203 - 1
k = [10, 100, 1000]

def Ferma_test(n, k = 1000):
    for i in range(k):
        a = random.randint(2, n)
        if gmpy2.gcd(n, a) > 1:
            return False
        if not gmpy2.is_fermat_prp(n, a):
            return False
    return True

def Solovay_Strassen_test(n, k = 1000):
    for i in range(k):
        a = random.randint(2, n)
        if not gmpy2.is_euler_prp(n, a):
            return False
    return True

def Miller_Rabin_test(n, k = 1000):
    for i in range(k):
        a = random.randint(2, n)
        if not gmpy2.is_strong_prp(n, a):
            return False
    return True

if __name__ == "__main__":
```

```
    start_time = time.time()
    print(APRtest(N))
    print(time.time() - start_time, "sec")
    for k_i in k:
        print(f'______k = {k_i}______')
        print('Ferma test')
        start_time = time.time()
        print(Ferma_test(N, k_i))
        print(time.time() - start_time, "sec")
        print('Solovay Strassen test')
        start_time = time.time()
        print(Solovay_Strassen_test(N, k_i))
        print(time.time() - start_time, "sec")
        print('Miller Rabin test')
        start_time = time.time()
        print(Miller_Rabin_test(N, k_i))
        print(time.time() - start_time, "sec")
```

laba2.py

```
from primality_tests import Miller_Rabin_test
from PRNGs import LehmerHigh, Chebyshev_prime, A_L, M_L, C_L

if __name__ == '__main__':
    N = 1024
    print('Generate pseudo random sequence:')
    print(LehmerHigh(A_L, M_L, C_L, N//8))
    print('Generate prime number:')
    prime = Chebyshev_prime(N)
    print(prime)
    print("Check it's primality:")
    print(Miller_Rabin_test(prime))
```