НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Лабораторна робота №2
«Реалізація алгоритмів генерації ключів гібридних криптосистем»

Підгрупа 1С

Виконали:
студенти групи ФІ-22мн
Бондаренко Андрій
Гузей Дмитро
Яценко Артем

Перевірила:
Байденко П. В.

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

Нами було обрано наступні генератори псевдовипадкових чисел для порівняльного аналізу:

1) Вбудований в Python генератор, який використовує Вихор Мерсена.
2) Лінійний конгруентний генератор **Lehmer** (**LehmerLow** та **LehmerHigh**).
3) Генератори псевдовипадкових двійкоих послідовностей **L20** та **L89**.
4) Генератор Блюм-Блюма-Шуба **BBC** (бітовий і байтовий).
5) Генератор Блюма-Мікалі **BM** (бітовий і байтовий).
6) Генератор **Джиффі**.
7) Генератор **Вольфрама**.

```
from constants import count_mask, lehmer_const, taps, n,
Bluma_Mikala_const, BBS_const
from random import randint, getrandbits
from geffe import Geffe, GeffeRegister, Lfsr
from lehmer import LehmerLow, LehmerHigh
from wolfram import Wolfram
from BM import BM
from BBS import BBS
```

```
from multiprocessing import Pool
from math import sqrt
from collections import Counter
import numpy as np
from scipy.stats import norm
import time

class Generator:
    def __init__(self, length):
        self.length_of_bit_sequence = length
        self.geffe_ = Geffe(
                GeffeRegister(randint(1, count_mask(n["l11"])), taps["l11"],
n["l11"]),
                GeffeRegister(randint(1, count_mask(n["l9"])), taps["l9"], n["l9"]),
                GeffeRegister(randint(1, count_mask(n["l10"])), taps["l10"],
n["l10"]),
            )
        self.lehmer_low_ = LehmerLow(lehmer_const)
        self.lehmer_high_ = LehmerHigh(lehmer_const)
        self.l20_ = Lfsr(randint(1, count_mask(n['l20'])), taps['l20'], n['l20'])
        self.l89_ = Lfsr(randint(1, count_mask(n['l89'])), taps['l89'], n['l89'])
        self.wolfram_ = Wolfram(32)
        self.embedded_ = 0
        self.BM_bits = BM(Bluma_Mikala_const[0], Bluma_Mikala_const[1],
Bluma_Mikala_const[2], Bluma_Mikala_const[3][0], 'bits')
        self.BM_bytes = BM(Bluma_Mikala_const[0], Bluma_Mikala_const[1],
Bluma_Mikala_const[2], Bluma_Mikala_const[3][1], 'bytes')
        self.BBS_bits = BBS(BBS_const[0], BBS_const[1], BBS_const[2][0],
'bits')
        self.BBS_bytes = BBS(BBS_const[0], BBS_const[1], BBS_const[2][1],
'bytes')
        for txt in ('geffe.txt', 'l20.txt', 'l89.txt', 'lehmerhigh.txt', 'lehmerlow.txt',
'wolfram.txt', 'BBS_byte.txt', 'BM_byte.txt', 'BBS_bit.txt', 'BM_bit.txt',
'embedded_gen.txt'):
            f = open(txt, 'w')
            f.close()

    def geffe(self):
        geffe_ = self.geffe_.generate_bits(self.length_of_bit_sequence)
        geffe_ = '0'*(self.length_of_bit_sequence - len(bin(geffe_)[2:])) +
bin(geffe_)[2:]

        with open("geffe.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
```

```
            for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):

                f.write(str(int(str(geffe_[i:j]), 2))+'\n')
        else:
            for i in range(0, self.length_of_bit_sequence, 8):
                f.write(str((geffe_>>i) & 0xff)+'\n')


    print('Geffe done!')

  def lehmer_low(self):
    lehmer_low_ =
self.lehmer_low_.generate_bits(self.length_of_bit_sequence)
    lehmer_low_ = '0'*(self.length_of_bit_sequence -
len(bin(lehmer_low_)[2:])) + bin(lehmer_low_)[2:]

    with open("lehmerlow.txt", "a") as f:
      if self.length_of_bit_sequence % 8 == 0:
        for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
          f.write(str(int(str(lehmer_low_[i:j]), 2))+'\n')
      else:
        for i in range(0, self.length_of_bit_sequence, 8):
          f.write(str((lehmer_low_>>i) & 0xff)+'\n')
    print('LehmerLow done!')

  def lehmer_high(self):
    lehmer_high_ =
self.lehmer_high_.generate_bits(self.length_of_bit_sequence)
    lehmer_high_ = '0'*(self.length_of_bit_sequence -
len(bin(lehmer_high_)[2:])) + bin(lehmer_high_)[2:]

    with open("lehmerhigh.txt", "a") as f:
      if self.length_of_bit_sequence % 8 == 0:
        for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
          f.write(str(int(str(lehmer_high_[i:j]), 2))+'\n')
      else:
        for i in range(0, self.length_of_bit_sequence, 8):
          f.write(str((lehmer_high_>>i) & 0xff)+'\n')
    print('LehmerHigh done!')

  def l20(self):
```

```
        l20_ = self.l20_.generate_bits(self.length_of_bit_sequence)
        l20_ = '0'*(self.length_of_bit_sequence - len(bin(l20_)[2:])) +
bin(l20_)[2:]

        with open("l20.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(l20_[i:j]), 2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((l20_>>i) & 0xff)+'\n')
        print('L20 done!')

    def l89(self):
        l89_ = self.l89_.generate_bits(self.length_of_bit_sequence)
        l89_ = '0'*(self.length_of_bit_sequence - len(bin(l89_)[2:])) +
bin(l89_)[2:]

        with open("l89.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(l89_[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((l89_>>i) & 0xff)+'\n')
        print('L89 done!')

    def wolfram(self):
        wolfram_ = self.wolfram_.generate_bits(self.length_of_bit_sequence)
        wolfram_ = '0'*(self.length_of_bit_sequence - len(bin(wolfram_)[2:])) +
bin(wolfram_)[2:]

        with open("wolfram.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(wolfram_[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((wolfram_>>i) & 0xff)+'\n')
        print('Wolfram done!')
```

```
    def embedded_gen(self):
        emb_ = getrandbits(self.length_of_bit_sequence)
        self.embedded_ = emb_
        emb = '0'*(self.length_of_bit_sequence - len(bin(emb_)[2:])) +
bin(emb_)[2:]
        with open("embedded_gen.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(emb[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((emb>>i) & 0xff)+'\n')

        print("embedded genetator!")

    def Bluma_Mikala_bits(self):
        bm = ''
        for b in self.BM_bits:
            bm+=str(b)
        bm = '0'*(self.length_of_bit_sequence - len(bm)) + bm
        with open("BM_bit.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bm[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bm>>i) & 0xff)+'\n')

    def Bluma_Mikala_bytes(self):
        bm = self.BM_bytes
        bm_ = ''
        for byte in bm:
            b = bin(byte)[2:]
            if len(b) == 8:
                bm_ += bin(byte)[2:]
            else:
                bm_ += '0'*(8 - len(b)) + b
        bm = bm_
        with open("BM_byte.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
```

```
                f.write(str(int(str(bm[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bm>>i) & 0xff)+'\n')

    def bbs_bits(self):
        bbs = ''
        for b in self.BBS_bits:
            bbs += str(b)
        bbs = '0'*(self.length_of_bit_sequence - len(bbs)) + bbs
        with open("BBS_bit.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bbs[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bbs>>i) & 0xff)+'\n')

    def bbs_bytes(self):
        bbs = self.BBS_bytes
        bbs_ = ''
        for byte in bbs:
            b = bin(byte)[2:]
            if len(b) == 8:
                bbs_ += bin(byte)[2:]
            else:
                bbs_ += '0'*(8 - len(b)) + b
        bbs = bbs_
        with open("BBS_byte.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bbs[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bbs>>i) & 0xff)+'\n')


def results_of_generators(generator: Generator,  processes: int):
        funcs = (generator.geffe, generator.lehmer_low, generator.lehmer_high,
generator.l20, generator.l89, generator.Bluma_Mikala_bits,
generator.Bluma_Mikala_bytes, generator.bbs_bits, generator.bbs_bytes,
generator.embedded_gen, generator.wolfram)
```

```
        pool = Pool(processes=processes)
        sub_processes = []
        for func in funcs:
            r = pool.apply_async(func)
            sub_processes.append(r)
        pool.close()
        pool.join()

def list_of_bytes(txt):
        bytes = []
        with open(txt, 'r') as f:
            for line in f.readlines():
                bytes.append(int(line[:-1]))
        return bytes


class Criteria:
    def __init__(self):
        self.txt_names = ('geffe.txt', 'l20.txt', 'l89.txt',  'lehmerhigh.txt',
'lehmerlow.txt',\
            'wolfram.txt', 'BBS_byte.txt', 'BM_byte.txt', 'BBS_bit.txt', 'BM_bit.txt',
'embedded_gen.txt')
        self.alphas = (0.01, 0.05, 0.1)
        self.uniformity_criterion_results = {}
        self.equiprobability_criterion_results = {}
        self.independence_criterion_results = {}


    def uniformityCriterion(self, r = 20):
        for txt in self.txt_names:
            try:
                f = list_of_bytes(txt)
                m2 = len(f) // r
                n = m2*r
                hi2 = 0
                l = 255*(r-1)
                f_ = [f[x:x+m2] for x in range (0, len(f), m2)]
            except ValueError:
                continue
            for i in range(256):
                vi = len([i for x in range(len(f)) if f[x] == i])

                for j in range(r):
                    vi2 = len([i for x in range(len(f_[j]))  if f_[j][x] == i])**2
```

```
                try:
                    hi2 += (vi2 / (vi * m2))
                except ZeroDivisionError:
                    continue
            hi2 = (hi2-1)*n
            print(txt[:-4]+'(uniformity criterion):')
            print(f'\u03C7\u00B2 = {hi2}\n')
            res = []
            hiteor = []
            hi = []
            for alpha in self.alphas:
                hi.append(hi2)
                hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha)+l
                hiteor.append(hi2teor)
                print(f'α = {alpha}')
                print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                if hi2 <= hi2teor:
                    print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(True)
                else:
                    print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(False)
                print()
            self.uniformity_criterion_results[txt[:-4]] = (res, hi, hiteor)
        print(self.uniformity_criterion_results)

    def equiprobabilityCriterion(self, l = 255):
        for txt in self.txt_names:

            bytes = list_of_bytes(txt)
            hi2 = 0
            n = len(bytes) / 256
            for j in range(256):
                v = 0
                for i in bytes:
                    if i == j:
                        v += 1
                try:
                    hi2 += ((v - n)**2) / n
                except ZeroDivisionError:
                    continue
```

```
            print(txt[:-4]+'(equiprobability criterion):')
            print(f'\u03C7\u00B2 = {hi2}\n')
            res = []
            hiteor = []
            hi = []
            for alpha in self.alphas:

                print(f'α = {alpha}')
                hi.append(hi2)
                hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha) + l
                hiteor.append(hi2teor)
                print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                if hi2 <= hi2teor:
                    print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(True)
                else:
                    print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(False)
                print()
            self.equiprobability_criterion_results[txt[:-4]] = (res, hi, hiteor)
        print(self.equiprobability_criterion_results)

    def independenceCriterion(self, l = 255**2):
        for txt in self.txt_names:
            nums = list_of_bytes(txt)
            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)

            print(txt[:-4]+'(independence criterion):')
```

```
            print(f'\u03C7\u00B2 = {hi2}\n')
            res = []
            hiteor = []
            hi = []
            for alpha in self.alphas:

                print(f'α = {alpha}')
                hi.append(hi2)
                hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha) + l
                hiteor.append(hi2teor)
                print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                if hi2 <= hi2teor:
                    print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(True)

                else:
                    print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(False)
                print()
            self.independence_criterion_results[txt[:-4]] = (res, hi, hiteor)
        print(self.independence_criterion_results)
```

Розглянемо послідовність Y_j, $j = 1, \ldots, m$, де кожна Y_j є випадковою величиною, що приймає набір значень із алфавіту A. Всі величини Y_j мають однаковий розподіл та розглядаються як вихідні значення деякого генератору.

Оцінювання якості генераторів

Послідовність Y_j задовольняє умові рівноімовірності знаків, якщо кожна Y_j розподілена рівноімовірно на A. Таким чином, кожне значення із повинно зустрічатись у довільній реалізації даної послідовності однакову кількість разів.

Тест на виконання умови рівноімовірності не відрізняється великою чутливістю, однак він доволі швидкий. В практичних задачах його рекомендується застосовувати в першу чергу, оскільки якщо послідовність не пройде цей тест, то немає рації застосовувати до неї інші. Також в якості підсилення можна розглядати умову рівноімовірності серій знаків, коли рівноімовірними в послідовності повинні бути пари, трійки, четвірки знаків тощо.

Послідовність Y_j задовольняє умові незалежності знаків, якщо імовірність рийняти деяке значення для Y_j не залежить від того, які значення прийняли $Y_1, Y_2, \ldots, Y_{j-1}$. Однак перевірка такої умови зазвичай вкрай важка, тому часто розглядають більш послаблені вимоги – наприклад, значення Y_j не повинно залежати від значення Y_{j-1} (незалежність від попереднього знаку).

Послідовність Y_j 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; іншими словами, на довільному фрагменті послідовність веде себе однаково. Зауважимо, що для виконання умови однорідності не важливо, який саме розподіл будуть мати Y_j. Зокрема, цей розподіл не обов'язково повинен бути рівноімовірним.

Перевірка умови однорідності в повному обсязі також доволі складна, тому на практиці перевіряють більш слабкі форми даної умови – наприклад, розбивають послідовність на окремі інтервали та перевіряють, чи співпадають вибіркові розподіли на цих інтервалах.

Програмно реалізуємо критерії Пірсона для кожної з наведених умов.

```
class Criteria:
   def __init__(self):
      self.txt_names = ('geffe.txt', 'l20.txt', 'l89.txt',  'lehmerhigh.txt', 'lehmerlow.txt',\
         'wolfram.txt', 'BBS_byte.txt', 'BM_byte.txt', 'BBS_bit.txt', 'BM_bit.txt',
'embedded_gen.txt')
      self.alphas = (0.01, 0.05, 0.1)
      self.uniformity_criterion_results = {}
      self.equiprobability_criterion_results = {}
      self.independence_criterion_results = {}
```

```
    def uniformityCriterion(self, r = 20):
        for txt in self.txt_names:
            try:
                f = list_of_bytes(txt)
                m2 = len(f) // r
                n = m2*r
                hi2 = 0
                l = 255*(r-1)
                f_ = [f[x:x+m2] for x in range (0, len(f), m2)]
            except ValueError:
                continue
            for i in range(256):
                vi = len([i for x in range(len(f)) if f[x] == i])

                for j in range(r):
                    vi2 = len([i for x in range(len(f_[j]))  if f_[j][x] == i])**2
                    try:
                        hi2 += (vi2 / (vi * m2))
                    except ZeroDivisionError:
                        continue
            hi2 = (hi2-1)*n
            print(txt[:-4]+'(uniformity criterion):')
            print(f'\u03C7\u00B2 = {hi2}\n')
            res = []
            hiteor = []
            hi = []
            for alpha in self.alphas:
                hi.append(hi2)
                hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha)+l
                hiteor.append(hi2teor)
                print(f'α = {alpha}')
                print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                if hi2 <= hi2teor:
                    print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(True)
                else:
                    print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(False)
                print()
            self.uniformity_criterion_results[txt[:-4]] = (res, hi, hiteor)
        print(self.uniformity_criterion_results)
```

```
    def equiprobabilityCriterion(self, l = 255):
        for txt in self.txt_names:

            bytes = list_of_bytes(txt)
            hi2 = 0
            n = len(bytes) / 256
            for j in range(256):
                v = 0
                for i in bytes:
                    if i == j:
                        v += 1
                try:
                    hi2 += ((v - n)**2) / n
                except ZeroDivisionError:
                    continue

            print(txt[:-4]+'(equiprobability criterion):')
            print(f'\u03C7\u00B2 = {hi2}\n')
            res = []
            hiteor = []
            hi = []
            for alpha in self.alphas:

                print(f'α = {alpha}')
                hi.append(hi2)
                hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha) + l
                hiteor.append(hi2teor)
                print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                if hi2 <= hi2teor:
                    print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(True)
                else:
                    print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(False)
                print()
            self.equiprobability_criterion_results[txt[:-4]] = (res, hi, hiteor)
        print(self.equiprobability_criterion_results)

    def independenceCriterion(self, l = 255**2):
        for txt in self.txt_names:
            nums = list_of_bytes(txt)
```

```
        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)

        print(txt[:-4]+'(independence criterion):')
        print(f'\u03C7\u00B2 = {hi2}\n')
        res = []
        hiteor = []
        hi = []
        for alpha in self.alphas:

            print(f'α = {alpha}')
            hi.append(hi2)
            hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha) + l
            hiteor.append(hi2teor)
            print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
            if hi2 <= hi2teor:
                print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                res.append(True)

            else:
                print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                res.append(False)
            print()
        self.independence_criterion_results[txt[:-4]] = (res, hi, hiteor)
    print(self.independence_criterion_results)
```

За результатами тестування, можемо сказати, що Вихор Мерсена, L20, LehmerHigh, BM і BBS найякісніші генератори. Посилаючись на NIST, LehmerHigh і BBS є криптографічно застосовними. З двох цих генераторів ми оберемо LehmerHigh

для застосування у системі клієнт-сервер (серверна частина), адже він є значно швидшим.

Генерація простих чисел

За теоремою Чебишева про розподіл простих чисел, кількість простих чисел на проміжку $[1,N]$ приблизно дорівнює $\pi(N)=N/lnN$. Щоб згенерувати просте число, виберіть будь-яке непарне число T і перевірте його на просте: якщо воно виявиться складеним, покладіть $T=T+2$ і поверніться до попереднього кроку. В середньому, після $\log_2 T$ кроків буде знайдено просте число.

Іншою альтернативою може бути алгоритм Маурера, заснований на критерії Поклінгтона, ще одному детермінованому тесті на простоту, який, однак, вимагає знаходження простого дільника p-1, що може призвести до зведення до задачі факторизації.

Тестування простоти

Для тестування простоти існують, такі види тестів:

1. Гіпотетичні
    a. Тест Фібоначчі
2. Імовірнісні
    a. Ферма
    b. Соловея-Штрассена
    c. Мілера-Рабіна
3. Детерміновані
    a. Тест Адлемана-Померанса-Румелі-Коена-Лейнстри
    b. Тест Аткіна-Морейна на еліптичних кривих

З з бібліотеки *gmpy2*, яку ми розглянули в лабораторній роботі № 1, імпортуємо тести Фібоначчі, Соловея-Штрассена, Мілера-Рабіна, Ферма, а реалізацію тесту Адлемана-Померанса-Румелі-Коена-Лейнстри та тесту Аткіна-Морейна на еліптичних кривих візьмемо з *github*.

Гіпотетичні тести ґрунтуються на теоретичних твердженнях, які вважаються істинними, але не доведені.
Тест Фібоначчі:

Якщо $p \equiv a \pmod{x^2+4}$, де a – квадратичний нелишок mod(x^2+4), тоді p буде простим при виконанні наступних умов: $f(1)_{p+1} \equiv 0 \pmod{p}$ і $2^{p-1} \equiv 1 \pmod{p}$, де $f(x)_k$— поліном Фібоначчі.

Імовірнісні тести найчастіше використовуються на практиці, оскільки

детерміновані тести часто мають дуже високу просторову складність.

```
import time
from gmpy2 import gcd, is_euler_prp, is_fermat_prp, is_strong_prp
from random import randint
from APR_CL import APRtest as test

k = (10, 100, 1000)

def SolovayStrassen(n, k):
    for i in range(k):
        a = randint(2, n)
        if not is_euler_prp(n, a):
            return False
    return True

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

def MillerRabin(n, k):
    for i in range(k):
        a = randint(2, n)
        if not is_strong_prp(n, a):
            return False
    return True

if __name__ == "__main__":
    start_time = time.time()
    print(test(2**2203 - 1))
    print(f'Time in seconds: {time.time() - start_time} sec')
    for i in k:
        print(f'\tk = {i}:')
        print('Ferma')
        start_time = time.time()
        print(Ferma(2**2203 - 1, i))
        print(f'Time in seconds: {time.time() - start_time} sec')
        print('Miller Rabin:')
        start_time = time.time()
```

```
print(MillerRabin(2**2203 - 1, i))
print(f'Time in seconds: {time.time() - start_time} sec')
print('Solovay Strassen:')
start_time = time.time()
print(SolovayStrassen(2**2203 - 1, i))
print(f'Time in seconds: {time.time() - start_time} sec')
```

Час роботи алгоритмів однаковий, але ймовірність помилки в тесті Міллера-Рабіна набагато нижча, тому цей тест зазвичай використовується майже скрізь.

Висновок: ми розглянули одинадцять алгоритмів генерації випадкових послідовностей, для кожного з яких ми застосували три тести для перевірки якості роботи генераторів. Ми також розглянули і вивчив різні групи тестів на простоту і реалізував імовірнісні тести на простоту за допомогою бібліотеки *gmpy2*. Були розглянуті алгоритми *Чебишева* і *Маурера* для генерації простих чисел, а також розроблена бібліотека для генерації псевдовипадкової послідовності, перевірки простого числа і генерації простих чисел для 1024 біт.

Код:

gen.py

```
from constants import count_mask, lehmer_const, taps, n, Bluma_Mikala_const,
```

```
BBS_const
from random import randint, getrandbits
from geffe import Geffe, GeffeRegister, Lfsr
from lehmer import LehmerLow, LehmerHigh
from wolfram import Wolfram
from BM import BM
from BBS import BBS
from multiprocessing import Pool
from math import sqrt
from collections import Counter
import numpy as np
from scipy.stats import norm
import time
from gmpy2 import is_prime, next_prime

class Generator:
    def __init__(self, length):
        self.length_of_bit_sequence = length
        self.geffe_ = Geffe(
                GeffeRegister(randint(1, count_mask(n["l11"])), taps["l11"], n["l11"]),
                GeffeRegister(randint(1, count_mask(n["l9"])), taps["l9"], n["l9"]),
                GeffeRegister(randint(1, count_mask(n["l10"])), taps["l10"], n["l10"]),
            )
        self.lehmer_low_ = LehmerLow(lehmer_const)
        self.lehmer_high_ = LehmerHigh(lehmer_const)
        self.l20_ = Lfsr(randint(1, count_mask(n['l20'])), taps['l20'], n['l20'])
        self.l89_ = Lfsr(randint(1, count_mask(n['l89'])), taps['l89'], n['l89'])
        self.wolfram_ = Wolfram(32)
        self.embedded_ = 0
        self.BM_bits = BM(Bluma_Mikala_const[0], Bluma_Mikala_const[1],
Bluma_Mikala_const[2], Bluma_Mikala_const[3][0], 'bits')
        self.BM_bytes = BM(Bluma_Mikala_const[0], Bluma_Mikala_const[1],
Bluma_Mikala_const[2], Bluma_Mikala_const[3][1], 'bytes')
        self.BBS_bits = BBS(BBS_const[0], BBS_const[1], BBS_const[2][0], 'bits')
        self.BBS_bytes = BBS(BBS_const[0], BBS_const[1], BBS_const[2][1], 'bytes')
        for txt in ('geffe.txt', 'l20.txt', 'l89.txt', 'lehmerhigh.txt', 'lehmerlow.txt',
'wolfram.txt', 'BBS_byte.txt', 'BM_byte.txt', 'BBS_bit.txt', 'BM_bit.txt',
'embedded_gen.txt'):
            f = open(txt, 'w')
            f.close()

    def geffe(self):
        geffe_ = self.geffe_.generate_bits(self.length_of_bit_sequence)
        geffe_ = '0'*(self.length_of_bit_sequence - len(bin(geffe_)[2:])) + bin(geffe_)[2:]
```

```
        with open("geffe.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):

                    f.write(str(int(str(geffe_[i:j]), 2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((geffe_>>i) & 0xff)+'\n')


        print('Geffe done!')

    def lehmer_low(self):
        lehmer_low_ = self.lehmer_low_.generate_bits(self.length_of_bit_sequence)
        lehmer_low_ = '0'*(self.length_of_bit_sequence - len(bin(lehmer_low_)[2:])) +
bin(lehmer_low_)[2:]

        with open("lehmerlow.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(lehmer_low_[i:j]), 2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((lehmer_low_>>i) & 0xff)+'\n')
        print('LehmerLow done!')

    def lehmer_high(self):
        lehmer_high_ = self.lehmer_high_.generate_bits(self.length_of_bit_sequence)
        lehmer_high_ = '0'*(self.length_of_bit_sequence - len(bin(lehmer_high_)[2:])) +
bin(lehmer_high_)[2:]

        with open("lehmerhigh.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(lehmer_high_[i:j]), 2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((lehmer_high_>>i) & 0xff)+'\n')
        print('LehmerHigh done!')
```

```
    def l20(self):
        l20_ = self.l20_.generate_bits(self.length_of_bit_sequence)
        l20_ = '0'*(self.length_of_bit_sequence - len(bin(l20_)[2:])) + bin(l20_)[2:]

        with open("l20.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(l20_[i:j]), 2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((l20_>>i) & 0xff)+'\n')
        print('L20 done!')

    def l89(self):
        l89_ = self.l89_.generate_bits(self.length_of_bit_sequence)
        l89_ = '0'*(self.length_of_bit_sequence - len(bin(l89_)[2:])) + bin(l89_)[2:]

        with open("l89.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(l89_[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((l89_>>i) & 0xff)+'\n')
        print('L89 done!')

    def wolfram(self):
        wolfram_ = self.wolfram_.generate_bits(self.length_of_bit_sequence)
        wolfram_ = '0'*(self.length_of_bit_sequence - len(bin(wolfram_)[2:])) +
bin(wolfram_)[2:]

        with open("wolfram.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(wolfram_[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((wolfram_>>i) & 0xff)+'\n')
        print('Wolfram done!')

    def embedded_gen(self):
```

```
        emb_ = getrandbits(self.length_of_bit_sequence)
        self.embedded_ = emb_
        emb = '0'*(self.length_of_bit_sequence - len(bin(emb_)[2:])) + bin(emb_)[2:]
        with open("embedded_gen.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(emb[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((emb>>i) & 0xff)+'\n')

        print("embedded genetator!")

    def Bluma_Mikala_bits(self):
        bm = ''
        for b in self.BM_bits:
            bm+=str(b)
        bm = '0'*(self.length_of_bit_sequence - len(bm)) + bm
        with open("BM_bit.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bm[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bm>>i) & 0xff)+'\n')

    def Bluma_Mikala_bytes(self):
        bm = self.BM_bytes
        bm_ = ''
        for byte in bm:
            b = bin(byte)[2:]
            if len(b) == 8:
                bm_ += bin(byte)[2:]
            else:
                bm_ += '0'*(8 - len(b)) + b
        bm = bm_
        with open("BM_byte.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bm[i:j]),2))+'\n')
            else:
```

```
            for i in range(0, self.length_of_bit_sequence, 8):
                f.write(str((bm>>i) & 0xff)+'\n')

    def bbs_bits(self):
        bbs = ''
        for b in self.BBS_bits:
            bbs += str(b)
        bbs = '0'*(self.length_of_bit_sequence - len(bbs)) + bbs
        with open("BBS_bit.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bbs[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bbs>>i) & 0xff)+'\n')

    def bbs_bytes(self):
        bbs = self.BBS_bytes
        bbs_ = ''
        for byte in bbs:
            b = bin(byte)[2:]
            if len(b) == 8:
                bbs_ += bin(byte)[2:]
            else:
                bbs_ += '0'*(8 - len(b)) + b
        bbs = bbs_
        with open("BBS_byte.txt", "a") as f:
            if self.length_of_bit_sequence % 8 == 0:
                for i, j in zip(range(0, self.length_of_bit_sequence+8, 8), range(8,
self.length_of_bit_sequence+8, 8)):
                    f.write(str(int(str(bbs[i:j]),2))+'\n')
            else:
                for i in range(0, self.length_of_bit_sequence, 8):
                    f.write(str((bbs>>i) & 0xff)+'\n')


def results_of_generators(generator: Generator,  processes: int):
        funcs = (generator.geffe, generator.lehmer_low, generator.lehmer_high,
generator.l20, generator.l89, generator.Bluma_Mikala_bits,
generator.Bluma_Mikala_bytes, generator.bbs_bits, generator.bbs_bytes,
generator.embedded_gen, generator.wolfram)
        pool = Pool(processes=processes)
        sub_processes = []
```

```
        for func in funcs:
            r = pool.apply_async(func)
            sub_processes.append(r)
        pool.close()
        pool.join()

def list_of_bytes(txt):
        bytes = []
        with open(txt, 'r') as f:
            for line in f.readlines():
                bytes.append(int(line[:-1]))
        return bytes


class Criteria:
    def __init__(self):
        self.txt_names = ('geffe.txt', 'l20.txt', 'l89.txt',  'lehmerhigh.txt', 'lehmerlow.txt',\
            'wolfram.txt', 'BBS_byte.txt', 'BM_byte.txt', 'BBS_bit.txt', 'BM_bit.txt',
'embedded_gen.txt')
        self.alphas = (0.01, 0.05, 0.1)
        self.uniformity_criterion_results = {}
        self.equiprobability_criterion_results = {}
        self.independence_criterion_results = {}


    def uniformityCriterion(self, r = 20):
        for txt in self.txt_names:
            try:
                f = list_of_bytes(txt)
                m2 = len(f) // r
                n = m2*r
                hi2 = 0
                l = 255*(r-1)
                f_ = [f[x:x+m2] for x in range (0, len(f), m2)]
            except ValueError:
                continue
            for i in range(256):
                vi = len([i for x in range(len(f)) if f[x] == i])

                for j in range(r):
                    vi2 = len([i for x in range(len(f_[j]))  if f_[j][x] == i])**2
                    try:
                        hi2 += (vi2 / (vi * m2))
                    except ZeroDivisionError:
```

```
                continue
        hi2 = (hi2-1)*n
        print(txt[:-4]+'(uniformity criterion):')
        print(f'\u03C7\u00B2 = {hi2}\n')
        res = []
        hiteor = []
        hi = []
        for alpha in self.alphas:
            hi.append(hi2)
            hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha)+l
            hiteor.append(hi2teor)
            print(f'α = {alpha}')
            print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
            if hi2 <= hi2teor:
                print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                res.append(True)
            else:
                print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                res.append(False)
            print()
        self.uniformity_criterion_results[txt[:-4]] = (res, hi, hiteor)
    print(self.uniformity_criterion_results)

def equiprobabilityCriterion(self, l = 255):
    for txt in self.txt_names:

        bytes = list_of_bytes(txt)
        hi2 = 0
        n = len(bytes) / 256
        for j in range(256):
            v = 0
            for i in bytes:
                if i == j:
                    v += 1
            try:
                hi2 += ((v - n)**2) / n
            except ZeroDivisionError:
                continue

        print(txt[:-4]+'(equiprobability criterion):')
        print(f'\u03C7\u00B2 = {hi2}\n')
        res = []
```

```
        hiteor = []
        hi = []
        for alpha in self.alphas:

            print(f'α = {alpha}')
            hi.append(hi2)
            hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha) + l
            hiteor.append(hi2teor)
            print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
            if hi2 <= hi2teor:
                print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                res.append(True)
            else:
                print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                res.append(False)
            print()
        self.equiprobability_criterion_results[txt[:-4]] = (res, hi, hiteor)
    print(self.equiprobability_criterion_results)

  def independenceCriterion(self, l = 255**2):
    for txt in self.txt_names:
        nums = list_of_bytes(txt)
        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)

        print(txt[:-4]+'(independence criterion):')
        print(f'\u03C7\u00B2 = {hi2}\n')
        res = []
        hiteor = []
```

```
            hi = []
            for alpha in self.alphas:

                print(f'α = {alpha}')
                hi.append(hi2)
                hi2teor = sqrt(2*l)*norm(loc=0, scale=1).ppf(1-alpha) + l
                hiteor.append(hi2teor)
                print(f'\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                if hi2 <= hi2teor:
                    print(f'criterion passed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(True)

                else:
                    print(f'criterion failed with α = {alpha},
\u03C7\u00B2\u2081\u208B\u2090 = {hi2teor}')
                    res.append(False)
                print()
            self.independence_criterion_results[txt[:-4]] = (res, hi, hiteor)
        print(self.independence_criterion_results)


def Chebyshev(l):
    n = getrandbits(l)
    if is_prime(n):
        return n
    return next_prime(n)
```

wolfram.py

```
from constants import count_mask
from random import randint

class Wolfram:

    def __init__(self, bits_size, r_init = None):
        if r_init == None:
            self.r = randint(1, count_mask(bits_size))
        else:
            self.r = r_init
        self.bits_size = bits_size
        self.mask = count_mask(bits_size)

    def get_bit(self):
        x = self.r % 2
```

```
        self.r = ((self.r >> (self.bits_size - 1)) | ((self.r << 1) & self.mask)) ^ (self.r |
((self.r & 1 << (self.bits_size - 1)) | (self.r >> 1)))
        return x

    def generate_bits(self, size):
        result = 0
        for i in range(size):
            result = (self.get_bit() << i) | result
        return result
```

lehmer.py

```
from math import ceil

class Lehmer:

    def __init__(self, const):
        self.m = const[0]
        self.a = const[1]
        self.c = const[2]
        self.x = [const[3]]

    def calc(self, n):
        if n % 8 == 0 and n != 8:
            while n > 0 and n != 0 and n != 8:
                self.x.append((self.a*self.x[-1]+self.c)%self.m)
                n-=8
        else:
            n = ceil(n/8)*8
            while n > 0 and n != 0 and n != 8:
                self.x.append((self.a*self.x[-1]+self.c)%self.m)
                n-=8
        return self.x

class LehmerLow (Lehmer):

    def __init__(self, const):
        Lehmer.__init__(self, const)
        self.mask = 0xff

    def generate_bits(self, n):
        Lehmer.calc(self, n)
        res = 0
        for i in self.x:
            res = (res << 8) | (int(i) & self.mask)
```

```
        return res

class LehmerHigh(Lehmer):

    def __init__(self, const):
        Lehmer.__init__(self, const)
        self.mask = 0xff000000

    def generate_bits(self, n):
        Lehmer.calc(self, n)
        res = 0
        for i in self.x:
            res = (res << 8) | ((int(i) & self.mask) >> 24)
        return res
```

geffe.py

```
from constants import count_mask
class GeffeRegister:

    def __init__(self, register, taps, size):
        self.register = register
        self.taps = taps
        self.size = size


class Geffe:
    def __init__(self, l1, l2, l3):
        self.reg1 = Lfsr(l1.register, l1.taps, l1.size)
        self.reg2 = Lfsr(l2.register, l2.taps, l2.size)
        self.reg3 = Lfsr(l3.register, l3.taps, l3.size)
        self.x = None
        self.y = None
        self.z = None
        self.s = None

    def get_bit(self):
        self.x = self.reg1.get_bit()
        self.y = self.reg2.get_bit()
        self.s = self.reg3.get_bit()
        self.z = (self.s * self.x) ^ ((1 ^ self.s) * self.y)

        return self.z

    def generate_bits(self, size):
```

```
        result = 0
        for i in range(size):
            result = (self.get_bit() << i) + result
        return result


class Lfsr:
    def __init__(self, register_init, taps, register_size, mask=None):
        self.register = register_init
        self.taps = taps
        self.register_size = register_size
        if mask:
            self.mask = mask
        else:
            self.mask = count_mask(register_size)

    def get_bit(self):
        xor = 0
        for t in self.taps:
            xor ^= (self.register >> t) & 1
        result = self.register & 1
        self.register = (xor << self.register_size - 1) | (
            (self.register >> 1) & self.mask
        )
        return result

    def generate_bits(self, size):
        result = 0
        for i in range(size):
            result = (self.get_bit() << i) + result
        return result
```

constants.py

```
from random import randint

def count_mask(n):
    res = 0
    for i in range(n):
        res += 2 ** i
    return res

n = {
    "l11": 11,
    "l9": 9,
```

```
    "l10": 10,
    "l20" : 20,
    "l89" : 89
    }

taps = {
    "l11": (0, 2),
    "l9": (0, 1, 3, 4),
    "l10": (0, 3),
    "l20" : (0, 11, 15, 17),
    "l89" : (0, 51)
    }

lehmer_const = (2**32, 2**16+1, 119, randint(1, count_mask(32)))


Bluma_Mikala_const =
(0x5B88C41246790891C095E2878880342E88C79974303BD0400B090FE38A688356
,
0x675215CC3E227D3216C056CFA8F8822BB486F788641E85E0DE77097E1DB049
F1,
0xCEA42B987C44FA642D80AD9F51F10457690DEF10C83D0BC1BCEE12FC3B60
93E3, (2**21, 2**18))


BBS_const = (0xD5BBB96D30086EC484EBA3D7F9CAEB07,
0x425D2B9BFDB25B9CF6C416CC6E37B59C1F, (2**21, 2**18))
```

BM.py

```
from random import randint
from gmpy2 import powmod, mul, div

def BM(a, q, p, lim, type = 'bit'):
    x = []
    T0 = randint(0, p-1)
    if type == 'bit' or type == 'bits':
        for i in range(lim):
            T1 = powmod(a, T0, p)
            x.append(1 if T1 < (p-1)/2 else 0)
            T0 = T1
        return x
    if type == 'byte' or type == 'bytes':
        b = div((p-1), 256)
```

```
        for i in range(lim):
            T1 = powmod(a, T0, p)
            k = 0
            while mul(k, b) >= T1 or 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]
```

BBS.py

```
from random import randint

def BBS(p, q, lim, type = 'bit'):
    n = p*q
    r0 = randint(2, n)
    x = []
    if type == 'bit' or type == 'bits':
        for i in range(lim):
            r1 = (r0**2)%n
            x.append(r1%2)
            r0 = r1
        return x
    if type == 'byte' or type == 'bytes':
        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]
```