Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра «Вычислительная техника»

Дисциплина «Информационная безопасность»

**Лабораторная работа №2**

**Реализация генератора псевдослучайных чисел**

Выполнил

Студент группы ИВТАСбд-41

Ведин В.А.

Проверил:

Старший преподаватель кафедры «ВТ»

Мартынов А.И.

Ульяновск

2025

# Задание

Реализовать приложение, позволяющее выполнять следующие действия:

1. Генерировать последовательность в битах (длина последовательности задается) с помощью выбранного генератора
2. Сохранять полученную последовательность в файл и выводить ее на экран приложения
3. Загружать последовательность из файла
4. Проверять полученную последовательность с помощью тестов, реализованных в предыдущей лабораторной работе. Результат проверки должен отображаться в приложении

# .

# Ход работы

# 1. Квадратичный конгруэнтный генератор

# Квадратичными конгруэнтными генераторами являются генераторы следующей формы:

# В которых - n-й член последовательности, а - предыдущий член последовательности, переменные a, b, c и m - постоянные: a и b – множители, c – инкремент и m – модуль. Ключом или затравкой служит значение .

# Период такого генератора не больше, чем m. Если a, b, c и m подобраны правильно, то генератор будет генератором с максимальным периодом, и его период будет равен m.

# Код, реализующий квадратичный конгруэнтный генератор представлен в листинге 1.

# Листинг 1. Квадратичный конгруэнтный генератор

|  |
| --- |
| def quadratic\_congruential\_generator(self, seq\_len: int = 10000) -> list[int]:"""Квадратичный конгруэнтный генератор псевдослучайной битовой последовательности.Использует рекуррентное соотношение:x\_{n+1} = {a \* x\_n^2 + b \* x\_n + c} mod mArgs:seq\_len: длина генерируемой битовой последовательности (по умолчанию 10000)Returns:List[int]: список битов (0 и 1)"""a = 1664525b = 1c = 1013904223m = 2 \*\* 32 - 1seed = 42bit\_seq = []x\_prev = seedfor \_ in range(seq\_len):x\_next = (a \* x\_prev \*\* 2 + b \* x\_prev + c) % mbit = x\_next & 1bit\_seq.append(bit)x\_prev = x\_nextreturn bit\_seq |

2. Генератор BBS..

Этот генератор основывается на проблеме факторизации больших чисел. Пусть нужно сгенерировать псевдослучайную последовательность длинной m бит.

Шаги алгоритма:

1. Сгенерировать два достаточно больших секретных случайных (и различных) простых числа и вычислить
2. Выбрать случайное стартовое целое число так, что Вычислить
3. Для
   1. Вычислить
   2. Вычислить как самый младший бит двоичного представления числа
4. В результате предыдущего шага формируется выходная псевдослучайная последовательность

Код реализующий данный алгоритм представлен в листинге 2.

Листинг 2. Алгоритм BBS

|  |
| --- |
| def bbs\_generator(self, seq\_len: int = 10000) -> list[int]:  """  Генератор Блюма-Блюма-Шуба (Blum-Blum-Shub).  Алгоритм основан на трудности факторизации больших чисел:  x\_{n+1} = x\_n^2 mod n, где n = p \* q (p и q - большие простые числа)  Args:  seq\_len: длина генерируемой битовой последовательности (по умолчанию 10000)    Returns:  List[int]: список битов (0 и 1)  """  def generate\_prime() -> int:  """  Функция, генерирующая простое число длинной 160 бит.  С помощью функции randprime из библиотеки sympy генерирует  простое число в диапазаоне от 2^159 до 2^160.  Returns:  int: большое простое число, удовлетворяющее условию: `prime ≡ 3 (mod 4)`  """  bit\_len = 160  while True:  prime = sympy.randprime(2\*\*(bit\_len - 1), 2\*\*bit\_len)  if prime % 4 == 3:  return prime    p = generate\_prime()  q = generate\_prime()  n = p \* q    while True:  seed = random.randint(2, n-1)  if math.gcd(seed, n) == 1:  break  x\_prev = pow(seed, 2, n)  bit\_seq = []  for \_ in range(seq\_len):  x\_next = pow(x\_prev, 2, n)  bit = x\_next & 1  bit\_seq.append(bit)  x\_prev = x\_next    return bit\_seq |

3. Алгоритм Yarrow-160

Данный алгоритм разобран Б. Шнайером, Дж. Келси и Н. Фергюсоном в 1999 году. В алгоритме для шифрования (функция ) может использовать алгоритм DES (или 3DES), а в качестве хэш-функции h() – SHA-1.

Шаги алгоритма:

1. Задание начальных значений
   1. Задать размер шифруемого сообщения , т.к. для шифрования используется алгоритм DES.
   2. Задать размер шифруемого ключа K, используемого при шифровании.
   3. Задать значение обычно определяющее количество бит, после генерации которых нужно обновить значение ключа K.
   4. Задать значение , определяющее количество бит, после генерации которых нужно запустить механизм обновления ключа K и счётчика , используя накопитель энтропии (entropy accumulator).  
      Накопитель энтропии конкатенирует «случайные» данные (текущая дата и время, количество запущенных в система процессов и т.п.). Пусть v – результат конкатенации и накопленных данных.
   5. Задать где t – количество запусков механизма обновления ключа и счётчика.
   6. Задать некоторое начальное значение n-битного счётчика
   7. Присвоить
2. Для выполнить:
   1. Если то:
      1. С помошью функции сгенерировать k бит, которые будут использоваться в качестве нового ключа K;
      2. Присвоить
   2. Если то:
      1. Вычислить ;
      2. Вычислить для
      3. Вычислить
      4. Вычислить
      5. Присвоить
   3. Вычислить которое является следующим блоком выходной последовательности.
   4. Выполнить и
3. В результате предыдущего шага формируется выходная псевдослучайная последовательность из m слов либо двоичная псевдослучайная последовательность из бит:

В этом алгоритме использованы функции G() и H(), определенные следующим образом.

Функция G(i):

1. Вычислить
2. Вернуть как результат вычисления функции.

Функция H(s, K):

1. Вычислить
2. Вычислить для
3. Вернуть первые k бит от конкатенации двоичных слов

Код данного алгоритма представлен в листинге 3.

Листинг 3. Алгоритм Yarrow-160

|  |
| --- |
| class Yarrow160:  """  Реализация криптографического генератора Yarrow-160.  Генератор основан на идее периодического обновления ключа K и счётчика C  с использованием блочного шифра (DES) и хеш-функции (SHA-1)  """  def \_\_init\_\_(  self,  n: int = 64,  k: int = 64,  Pg: int = 10,  Pt: int = 20  ) -> None:  """  Инициализирует параметры генератора Yarrow-160.  Args:  n: размер блока (бит)  k: рзамер ключа (бит)  Pg: порог обновления ключа K  Pt: порог обновления ключа и счётчика  """  self.n = n  self.k = k  self.Pg = Pg  self.Pt = Pt  self.curPg = Pg  self.curPt = Pt  self.C = 0  self.K = os.urandom(8)  self.t = 0    def entropy\_accumulator(self) -> bytes:  """  Имитация наколепния энтропии.  Генерирует псевдослучайные данные на основе системного времени,  PID процесса и случайных байт.  Returns:  bytes: SHA-1 хеш от собранных жнтропийных данных  """  data = f"{time.time\_ns()}\_{os.getpid()}\_{os.urandom(8).hex()}".encode()  return hashlib.sha1(data).digest()    def update\_key(self) -> None:  """  Обновляет внутренний ключ K и счётчик C на сонове новой энтропии.  """  entropy = self.entropy\_accumulator()  self.K = hashlib.sha1(self.K + entropy).digest()[:8]  self.C = (self.C + 1) % (2 \*\* self.n)    def encrypt\_block(self, data: bytes) -> bytes:  """  Шифрует блок данных с помощью DES в режиме ECB.  Args:  data: байты длиной 8 байт.    Returns:  bytes: зашифрованный блок длиной 8 байт  """  cipher = DES.new(self.K, DES.MODE\_ECB)  return cipher.encrypt(data)    def H(self, v: bytes, K: bytes) -> bytes:  """  Хеш-функция H(v, K) = SHA-1(v || K)[:8].  Args:  v: байтовая последовательность  k: ключ    Returns:  bytes: результат хеширования длиной 8 байт  """  return hashlib.sha1(v + K).digest()[:8]    def G(self, i: int) -> bytes:  """  Функция G(i), генерирующая новые данные для ключа.  Args:  i: индекс итерации  Returns:  bytes: результат шифрования счётчика длиной 8 байт  """  Ci = (self.C + i) % (2 \*\* self.n)  Ci\_bytes = Ci.to\_bytes(8, "big")  return self.encrypt\_block(Ci\_bytes)    def generate\_bits(self, seq\_len: int = 10000):  """  Генерирует псевдослучайную битовую последовательность.  Args:  seq\_len: требуемая длина последовательности (по умолчанию 10000)    Returns:  List[int]: список битов (0 и 1)  """  bit\_seq = []  while len(bit\_seq) < seq\_len:  if self.curPg == 0:  self.K = self.G(self.C)  self.curPg = self.Pg    if self.curPt == 0:  v0 = hashlib.sha1(self.entropy\_accumulator() + self.K).digest()[:8]  v = v0  for \_ in range(2):  v = hashlib.sha1(v + v0 + self.K).digest()[:8]  self.K = self.H(v, self.K)  self.update\_key()  self.curPt = self.Pt    xi = self.encrypt\_block(self.C.to\_bytes(8, "big"))  self.C = (self.C + 1) % (2 \*\* self.n)  self.curPg -= 1  self.curPt -= 1  for byte in xi:  for bit in range(8):  bit\_seq.append((byte >> (7 - bit)) & 1)  if len(bit\_seq) >= seq\_len:  break  if len(bit\_seq) >= seq\_len:  break    return bit\_seq    def yarrow160\_generator(self, seq\_len: int = 10000) -> list[int]:  """  Интерфейсная функция для генерации последовательности Yarrow-160.  Создаёт экземпляр внутреннего класса Yarrow160 и вызывает метод generate\_bits.  Args:  seq\_len: длина генерируемой последовательности (по умолчанию 10000)    Returns:  List[int]: список битов (0 и 1)  """  gen = self.Yarrow160()  bit\_seq = gen.generate\_bits(seq\_len)  return bit\_seq |

# Вывод

В ходе выполнения лабораторной работы было разработано приложение, реализующее три различных генератора псевдослучайных чисел: квадратичный конгруэнтный генератор, генератор Блюма-Блюма-Шуба (BBS) и генератор Yarrow-160.

Квадратичный конгруэнтный генератор показал себя как простой и эффективный метод генерации псевдослучайных последовательностей, основанный на рекуррентном соотношении. Генератор BBS, основанный на сложности факторизации больших чисел, обеспечил более высокий уровень криптостойкости, что делает его применимым в задачах, требующих повышенной надёжности. Алгоритм Yarrow-160, использующий механизмы периодического обновления ключа и накопления энтропии, а также комбинацию хеш-функции SHA-1 и блочного шифра DES, продемонстрировал возможность создания криптостойких последовательностей, устойчивых к различным видам атак.

Важной частью работы стала интеграция ранее реализованных статистических тестов для проверки качества генерируемых последовательностей. Это позволило оценить случайность и равномерность распределения бит, что является ключевым критерием при выборе генератора для практического использования.

В результате проведённой работы было получено приложение, позволяющее генерировать, сохранять, загружать и тестировать псевдослучайные последовательности с использованием различных методов.

**Приложение А. generator.py**

|  |
| --- |
| import random  import hashlib  import time  import os  import sympy  import math  from Crypto.Cipher import DES  class Generator:  """  Класс, объединяющий различные генераторы псевдослучайных битовых последовательностей:  - Квадратичный конгруэнтный генератор (Quadratic Congruential Generator),  - Генератор Блюма-Блюма-Шуба (BBS),  - Генератор Yarrow-160.  """  def \_\_init\_\_(self) -> None:  """  Инициализирует экземпляр генератора без параметров  """  ...    def quadratic\_congruential\_generator(self, seq\_len: int = 10000) -> list[int]:  """  Квадратичный конгруэнтный генератор псевдослучайной битовой последовательности.    Использует рекуррентное соотношение:  x\_{n+1} = {a \* x\_n^2 + b \* x\_n + c} mod m  Args:  seq\_len: длина генерируемой битовой последовательности (по умолчанию 10000)    Returns:  List[int]: список битов (0 и 1)  """  a = 1664525  b = 1  c = 1013904223  m = 2 \*\* 32 - 1  seed = 42    bit\_seq = []  x\_prev = seed  for \_ in range(seq\_len):  x\_next = (a \* x\_prev \*\* 2 + b \* x\_prev + c) % m  bit = x\_next & 1  bit\_seq.append(bit)  x\_prev = x\_next    return bit\_seq    def bbs\_generator(self, seq\_len: int = 10000) -> list[int]:  """  Генератор Блюма-Блюма-Шуба (Blum-Blum-Shub).  Алгоритм основан на трудности факторизации больших чисел:  x\_{n+1} = x\_n^2 mod n, где n = p \* q (p и q - большие простые числа)  Args:  seq\_len: длина генерируемой битовой последовательности (по умолчанию 10000)    Returns:  List[int]: список битов (0 и 1)  """  def generate\_prime() -> int:  """  Функция, генерирующая простое число длинной 160 бит.  С помощью функции randprime из библиотеки sympy генерирует  простое число в диапазаоне от 2^159 до 2^160.  Returns:  int: большое простое число, удовлетворяющее условию: `prime ≡ 3 (mod 4)`  """  bit\_len = 160  while True:  prime = sympy.randprime(2\*\*(bit\_len - 1), 2\*\*bit\_len)  if prime % 4 == 3:  return prime    p = generate\_prime()  q = generate\_prime()  n = p \* q    while True:  seed = random.randint(2, n-1)  if math.gcd(seed, n) == 1:  break  x\_prev = pow(seed, 2, n)  bit\_seq = []  for \_ in range(seq\_len):  x\_next = pow(x\_prev, 2, n)  bit = x\_next & 1  bit\_seq.append(bit)  x\_prev = x\_next    return bit\_seq    class Yarrow160:  """  Реализация криптографического генератора Yarrow-160.  Генератор основан на идее периодического обновления ключа K и счётчика C  с использованием блочного шифра (DES) и хеш-функции (SHA-1)  """  def \_\_init\_\_(  self,  n: int = 64,  k: int = 64,  Pg: int = 10,  Pt: int = 20  ) -> None:  """  Инициализирует параметры генератора Yarrow-160.  Args:  n: размер блока (бит)  k: рзамер ключа (бит)  Pg: порог обновления ключа K  Pt: порог обновления ключа и счётчика  """  self.n = n  self.k = k  self.Pg = Pg  self.Pt = Pt  self.curPg = Pg  self.curPt = Pt  self.C = 0  self.K = os.urandom(8)  self.t = 0    def entropy\_accumulator(self) -> bytes:  """  Имитация наколепния энтропии.  Генерирует псевдослучайные данные на основе системного времени,  PID процесса и случайных байт.  Returns:  bytes: SHA-1 хеш от собранных жнтропийных данных  """  data = f"{time.time\_ns()}\_{os.getpid()}\_{os.urandom(8).hex()}".encode()  return hashlib.sha1(data).digest()    def update\_key(self) -> None:  """  Обновляет внутренний ключ K и счётчик C на сонове новой энтропии.  """  entropy = self.entropy\_accumulator()  self.K = hashlib.sha1(self.K + entropy).digest()[:8]  self.C = (self.C + 1) % (2 \*\* self.n)    def encrypt\_block(self, data: bytes) -> bytes:  """  Шифрует блок данных с помощью DES в режиме ECB.  Args:  data: байты длиной 8 байт.    Returns:  bytes: зашифрованный блок длиной 8 байт  """  cipher = DES.new(self.K, DES.MODE\_ECB)  return cipher.encrypt(data)    def H(self, v: bytes, K: bytes) -> bytes:  """  Хеш-функция H(v, K) = SHA-1(v || K)[:8].  Args:  v: байтовая последовательность  k: ключ  Returns:  bytes: результат хеширования длиной 8 байт  """  return hashlib.sha1(v + K).digest()[:8]    def G(self, i: int) -> bytes:  """  Функция G(i), генерирующая новые данные для ключа.  Args:  i: индекс итерации  Returns:  bytes: результат шифрования счётчика длиной 8 байт  """  Ci = (self.C + i) % (2 \*\* self.n)  Ci\_bytes = Ci.to\_bytes(8, "big")  return self.encrypt\_block(Ci\_bytes)    def generate\_bits(self, seq\_len: int = 10000):  """  Генерирует псевдослучайную битовую последовательность.  Args:  seq\_len: требуемая длина последовательности (по умолчанию 10000)    Returns:  List[int]: список битов (0 и 1)  """  bit\_seq = []  while len(bit\_seq) < seq\_len:  if self.curPg == 0:  self.K = self.G(self.C)  self.curPg = self.Pg    if self.curPt == 0:  v0 = hashlib.sha1(self.entropy\_accumulator() + self.K).digest()[:8]  v = v0  for \_ in range(2):  v = hashlib.sha1(v + v0 + self.K).digest()[:8]  self.K = self.H(v, self.K)  self.update\_key()  self.curPt = self.Pt    xi = self.encrypt\_block(self.C.to\_bytes(8, "big"))  self.C = (self.C + 1) % (2 \*\* self.n)  self.curPg -= 1  self.curPt -= 1  for byte in xi:  for bit in range(8):  bit\_seq.append((byte >> (7 - bit)) & 1)  if len(bit\_seq) >= seq\_len:  break  if len(bit\_seq) >= seq\_len:  break  return bit\_seq    def yarrow160\_generator(self, seq\_len: int = 10000) -> list[int]:  """  Интерфейсная функция для генерации последовательности Yarrow-160.  Создаёт экземпляр внутреннего класса Yarrow160 и вызывает метод generate\_bits.  Args:  seq\_len: длина генерируемой последовательности (по умолчанию 10000)  Returns:  List[int]: список битов (0 и 1)  """  gen = self.Yarrow160()  bit\_seq = gen.generate\_bits(seq\_len)  return bit\_seq |

**Приложение Б. bit\_tests.py**

|  |
| --- |
| from typing import Optional, Callable  from generator import Generator  import random  import math  import json  CONST = 1.82138636 # Критическое значение статистики для всех тестов  def generate\_bit\_seq(  generator\_fn: Callable,  seq\_len: Optional[int] = 10000,  input\_file\_path: Optional[str] = "",  output\_file\_path: Optional[str] = "",  seed: int = 42  ) -> list[int]:  """  Вызывает функцию выбранного генератора или загружает последовательность битов (0 и 1).  Args:  generator\_fn: Функция выбранного генератора  seq\_len: Длина генерируемой последовательности (по умолчанию 10000 бит)  input\_file\_path: Путь к файлу для загрузки последовательности  output\_file\_path: Путь для сохранения последовательности  seed: Seed для генератора случайных чисел  Returns:  list[int]: последовательность битов (0 и 1).    Note:  Если указан input\_file\_path, последовательность загружается из файла.  Иначе генерируется случайная последовательность заданной длины.  """  bit\_seq = None  if input\_file\_path != "":  with open(input\_file\_path, "r", encoding="utf-8") as f:  bit\_seq = list(f.read())  bit\_seq = [int(bit) for bit in bit\_seq]  bit\_seq = generator\_fn(seq\_len) if bit\_seq is None else bit\_seq  if output\_file\_path != "":  with open(output\_file\_path, "w", encoding="utf-8") as f:  f.write("".join(map(str, bit\_seq)))    return bit\_seq  def frequency\_test(bit\_seq: list[int]) -> bool:  """  Частотный тест (Frequency Test).  Проверяет пропорцию нулей и единиц в последовательности.  Определяет, является ли кол-во нулей и единиц приблизительно таким же,  как в истинно случайно последовательности.  Args:  bit\_seq: Последовательность битов (0 и 1)  Returns:  bool: True если тест пройден (последовательность случайна), False в противном  случае  Algorithm:  1. Преобразование битов в значения -1 и 1  2. Вычисление суммы S\_n  3. Вычисление статистики S = |S\_n| / sqrt(n)  4. Сравнение с критическим значением CONST  """  fixed\_bit\_seq = [2 \* bit - 1 for bit in bit\_seq]  S\_n = sum(fixed\_bit\_seq)  if (abs(S\_n) / (math.sqrt(len(bit\_seq)))) <= CONST:  return True  return False  def r(bit\_seq: list[int], k: int):  """  Вспомогательная функция для теста на последовательность одинаковых бит.  Определяет, является ли k-тый бит началом новой цепочки.  Args:  bit\_seq: Последовательность битов  k: Индекс текущего бита    Returns:  int: 1 если бит k и k+1 разные (начало новой цепочки) 0 если одинаковые  """  if bit\_seq[k] == bit\_seq[k + 1]:  return 0  return 1  def identical\_bit\_seq\_test(bit\_seq: list[int]) -> bool:  """  Тест на последовательность одинаковых бит (Rusn Test).  Анализирует количество цепочек (непрерывных последовательностей одинаковых бит)  в проверяемой последовательности  Args:  bit\_seq: Последовательность битов (0 и 1)    Returns:  bool: True если тест пройден, False в противном случае    Algorithm:  1. Вычисление частоты единиц (pi)  2. Вычисление количества переходов V\_n  3. Вычисление статистики S  4. Сравнение с критическим значением CONST  """  pi = 1 / (len(bit\_seq)) \* sum(bit\_seq)  V\_n = sum([r(bit\_seq, k) for k in range(len(bit\_seq) - 1)]) + 1  S = abs(V\_n - 2 \* len(bit\_seq) \* pi \* (1 - pi))  S /= 2 \* math.sqrt(2 \* len(bit\_seq) \* pi \* (1 - pi))  if S <= CONST:  return True  return False  def extended\_random\_deviation\_test(bit\_seq: list[int]) -> bool:  """  Расширенный тест на произвольные отклонения.  Оценивает общее число посещений определенного состояния при произвольном обходе  кумулятивный суммы. Состоит из 18 подтестов для состояний из -9 до 9 (кроме 0).  Args:  bit\_seq: Последовательность битов (0 и 1)    Returns:  bool: True если все 18 тестов пройдены, False если хотя бы один не пройден    Algorithm:  1. Преобразование битов в -1 и 1  2. Вычисление кумулятивных сумм  3. Фомирование расширенной последовательности S'  4. Подсчёт количества нулей L  5. Вычисление количества посещений каждого состояния  6. Вычисление статистик Y\_j для каждого состояния  7. Проверка всех статистик на превышение CONST  """  states = [str(i) for i in range(-9, 10) if i != 0]  fixed\_bit\_seq = [2 \* bit - 1 for bit in bit\_seq]  S = [0, fixed\_bit\_seq[0]]  fixed\_bit\_seq = fixed\_bit\_seq[1:]  S.extend([S.append(S[-1] + bit) for bit in fixed\_bit\_seq])  S.append(0)  L = S.count(0)  theta = {state: S.count(int(state)) for state in states}  Y = {  j: (abs(theta[j]) - L) / math.sqrt(2 \* L \* (4 \* abs(int(j)) - 2))  for j in states  }  if all([y < CONST for y in list(Y.values())]):  return True  return False    def run\_tests(seq\_len: int, config\_path: str, generator\_fn: Callable) -> tuple[list[int], dict]:  """  Основная функция для запуска всех тестов псевдослучайных последовательностей.  Args:  seq\_len: Длина последовательности для генерации    Returns:  tuple: (строковое представление последовательности, словарь с результатами тестов)    Note:  Загружает конфигурацию из config.json, генерирует/загружает последовательность  выполняет все три теста и возвращает результаты  """  with open(config\_path, "r", encoding="utf-8") as f:  config = json.load(f)    bit\_seq = generate\_bit\_seq(  generator\_fn,  seq\_len, config["input\_file\_path"],  config["output\_file\_path"]  )    str\_bit\_seq = "".join(str(bit) for bit in bit\_seq)  test\_results = {  "Частотный тест": frequency\_test(bit\_seq),  "Тест на последовательность одинаковых бит": identical\_bit\_seq\_test(bit\_seq),  "Расширенный тест на произвольные отклонения": extended\_random\_deviation\_test(bit\_seq)  }  return str\_bit\_seq, test\_results |

**Приложение В. interface.py**

|  |
| --- |
| from rich.console import Console  from rich.panel import Panel  from rich.prompt import Prompt  from rich import print  from bits\_tests import run\_tests  from generator import Generator  import json  import os  import time  class Interface:  """  Графический интерфейс для тестирования псведослучайных последовательностей.  Предоставляет интерактивное меню для работы с программой тестирования битовых последовательностей.  Включает главное меню, меню запуска тестов, справку и настройки.  Attributes:  console: Объект для работы с консольным выводом (библиотека rich)  config\_path: Путь к файлу конфигурации  config: Загруженная конфигурация программы  """  def \_\_init\_\_(self) -> None:  """  Инициализирет интерфейс, загружает конфигурацию из JSON-файла  """  self.console = Console()  self.config\_path = "/home/v\_vedin/university/labs/FouthCourse/FirstTerm/Information-Security/lab1/config.json"  with open(self.config\_path, "r", encoding="utf-8") as f:  self.config = json.load(f)  self.generator = Generator()  self.generators\_fns = {  "1": ("Квадратичный Конгруэтный Генератор", self.generator.quadratic\_congruential\_generator),  "2": ("Blum-Blum-Shub", self.generator.bbs\_generator),  "3": ("Yarrow-160", self.generator.yarrow160\_generator)  }    def print\_title(self) -> None:  """  Отображает заголовок программы с информацеий об университете и студенте.  Очищает экран и выводит:  - Шапку с названием университета  - Название лабораторной работы и дисциплины  - Информацию о студенте  """  self.console.clear()  self.console.rule('Федеральное государственное бюджетное образовательное ' \  'учреждение высшего образования "Ульяновский ' \  'государственный технический университет"', style="blue")  self.console.print(  "\nЛабораторная Работа №2\n" \  "Дисциплина: Информационная Безопасность\n" \  "Работа выполнена студентом группы ИВТАСбд-41 Ведином Владимиром Александровичом\n",  justify="center"  )  self.console.rule(style="blue")    def print\_menu(self) -> None:  """  Отображает главное меню программы и обрабатывает выбор пользователя.  Пункты меню:  1. Старт программы - запуск тестирования последовательности  2. Инструкция - отображание справки по программе  3. Настройки - изменение путей к файлам  4. Выход - завершение программы  Вызывает соответствующие методы в зависимости от выбора пользователя  """  print(Panel.fit(  "[bold cyan] Главное Меню [/bold cyan]",  style="cyan",  ))  print("1. Старт программы")  print("2. Инструкция")  print("3. Настройки")  print("4. Выход")  self.console.rule(style="blue")  choice = Prompt.ask(  "[bold yellow]Выберите пункт[/bold yellow]",  choices = ["1", "2", "3", "4"],  default="1"  )  if choice == "1":  self.run\_menu()  elif choice == "2":  self.help\_menu()  elif choice == "3":  self.settings\_menu()  elif choice == "4":  print("Завершение программыы...")  time.sleep(0.5)  self.console.clear()  exit(0)    def run\_menu(self) -> None:  """  Запускает процесс тестирования последовательности битов.  Если не указан входной файл, запрашивает длину генерируемой последовательности.  Затем зарашивает тип генератора из предложенных вариантов на выбор  Выполняет тесты и отображает результаты:  - Выводит саму последовательность  - Показывает результаты каждого теста (пройден/не пройден)  - Прекращает тестирование при первом неудачном тесте    После завершения тестов возвращает управление в главное меню  """  seq\_len = 10000  if self.config["input\_file\_path"] == "":  self.console.clear()  self.console.rule(style="blue")  print(Panel.fit(  "[bold green] Запуск программы [/bold green]",  style="green"  ))  self.console.rule(style="blue")  seq\_len = int(Prompt.ask(  "[bold yellow]Укажите длину генерируемой последовательности[/bold yellow]",  default="10000"  ))  self.console.clear()  self.console.rule(style="blue")  print(Panel.fit(  "[bold cyan] Выберите генератор [/bold cyan]",  style="cyan",  ))  self.console.rule(style="blue")  print(f"1. {self.generators\_fns["1"][0]}")  print(f"2. {self.generators\_fns["2"][0]}")  print(f"3. {self.generators\_fns["3"][0]}")  generator = (Prompt.ask(  "[bold yellow]Выберите пункт[/bold yellow]",  choices=["1", "2", "3"],  default="1"  ))    self.console.clear()  self.console.rule(f"Результаты тестов {self.generators\_fns[generator][0]}", style="blue")  bit\_seq, test\_data = run\_tests(seq\_len, self.config\_path, self.generators\_fns[generator][1])  print("Вывод битов...")  if self.config["input\_file\_path"] != "":  print(  "Последовательность прочитанная из файла:\n"  f"{bit\_seq}"  )  else:  print(  "Сгенерированная последовательность\n"  f"{bit\_seq}"  )      for name, res in test\_data.items():  print(f"Результаты {name}: ", end="")  if res == True:  text = "пройден"  style = "green"  else:  text = "непройден"  style = "red"  self.console.print(text, style=style)  if res == False:  print("Последовательность не случайна")  break  else:  print("Все тесты пройдены, последовательность случайна")  Prompt.ask(  "[bold yellow]Нажмите любую кнопку, чтобы вернуться в главное меню[/bold yellow]"  )      def help\_menu(self) -> None:  """  Отображает инструкцию по работе с программой.  Содержит информацию о:  - Назначении программы  - Доступных тестах и их описании  - Возможностях генерации и загрузки последовательностей  - Логике выполнения тестов  """  self.console.clear()  self.console.rule(style="blue")  print(Panel.fit(  "[bold blue] Инструкция [/bold blue]"  ))  print(  "Это программа по тестированию последовательности бит на равномерность и случайность \n" \  "В ней представлены такие тесты как:\n" \  "\t1. Частотный тест (Оценка пропорции нулей и единиц в последовательности)\n" \  "\t2. Тест на последовательность одинаковых бит (Анализ кол-ва непрерывных последовательностей одинаковых бит)\n" \  "\t3. Расширенный тест на произвольные отклонения (Оцнека общего числа посещения состояния при произвольном обходе кумулятивной суммы)\n" \  "\t\tСостояния - последовательность чисел вида [-9, -8, ..., -1, 1, 2, ..., 9]\n\n"  )  print(  "Саму последовательность любой длинны (на выбор пользователя) можно как случайно генерировать" \  " с помощью заданных на выбор генераторов так и считывать из файла, путь до кторого пользователь может указать\n" \  "Также, саму последовательность можно и сохранить в файл, снова указав до него путь\n" \  "Указать путь до входного и выходного файла можно в настройках\n\n"  )  print(  "Тесты проводятся по очереди (от 1 до 3, как указано выше). Если акой-то из тестов не проходит, то остальные не проводятся\n"  )  print(  "Генераторы данные на выбор:\n" \  "\t1. Квадратичный Конгруэнтный генератор\n" \  "\t2. BBS (Blum-Blum-Shub) генератор\n" \  "\t3. Yarrow160 генератор\n"  )  self.console.rule(style="blue")  Prompt.ask(  "[bold yellow]Нажмите любую кнопку чтобы выйти[/bold yellow]",  )  return  def settings\_menu(self) -> None:  """  Позволяет настроить пути к входному и выходному фалам.  Предоставляет возможности:  1. Указать путь до входного файла с последовательностью  2. Указать путь до выходного файла  3. Вернуться в главное меню  Проверяет существование указанных файлов и сохраняет настройки в конфигурационный файл  """  self.console.clear()  self.console.rule(style="blue")  print(Panel.fit(  "[bold magenta] Настройки [/bold magenta]"  ))  print(  "1. Указать путь до входного файла\n" \  "2. Указать путь до выходного файла\n" \  "3. Выход\n" \  )  self.console.rule(style="blue")  choice = Prompt.ask(  "[bold yellow]Выберите пункт[/bold yellow]",  choices = ["1", "2", "3"],  default="1"  )  if choice == "1" or choice == "2":  choices\_str\_dict = {  "1": "входного",  "2": "выходного"  }  choices\_keys\_dict = {  "1": "input\_file\_path",  "2": "output\_file\_path"  }    self.console.clear()  self.console.rule(style="blue")  print(  f"Текущий путь до {choices\_str\_dict[choice]} файла: " \  f"{self.config[choices\_keys\_dict[choice]] if self.config[choices\_keys\_dict[choice]] != '' else 'не указан'}"  )  self.console.rule(style="blue")    while True:  file\_path = Prompt.ask(  f"[bold yellow]Введите путь до {choices\_str\_dict[choice]} файла или введите 0 для сброса[/bold yellow]",  default="Нажмите enter, чтобы вернуться назад"  )  if file\_path == "0":  self.config[choices\_keys\_dict[choice]] = ""  with open(self.config\_path, "w", encoding="utf-8") as f:  json.dump(self.config, f, indent=4)  print("Путь до файла сброшен")  break    if file\_path == "Нажмите enter, чтобы вернуться назад":  break  if os.path.exists(file\_path) and os.path.isfile(file\_path):  self.config[choices\_keys\_dict[choice]] = file\_path  with open(self.config\_path, "w", encoding="utf-8") as f:  json.dump(self.config, f, indent=4)  print("Путь до файла успешно сохранён")  break  print("Такого файла не существует или путь указан неверно попробуйте ещё раз")    def main\_loop(self) -> None:  """  Основный цикл работы интерфейса.  Бесконечно отображает заголовок и главное меню, обеспечивая нерперывную работу интерфейса  до явного выхода пользователя через пункт меню "Выход"  """  while True:  self.console.clear()  self.print\_title()  self.print\_menu() |

**Приложение Г. main.py**

|  |
| --- |
| from interface import Interface  if \_\_name\_\_ == "\_\_main\_\_":  interface = Interface()  interface.main\_loop() |