Лабораторная работа №2

Реализация генератора псевдослучайных чисел

Цель работы: Научиться генерировать псевдослучайную последовательность с помощью различных алгоритмов

Теоретическая часть

Случайные числа и их генераторы являются неотъемлемыми элементами современных криптосистем. Они используются в криптографии для различных целей:

- сеансовые и другие ключи для симметричных криптосистем, таких как DES, ГОСТ 28147-89, Blowfish;
- стартовые значения для программ генерации ряда математических величин в асимметричных криптосистемах, например больших простых чисел в RSA, ElGamal;
- вектор инициализации для блочных криптосистем, работающих в режиме обратной связи;
- случайные значения параметров для многих систем электронной цифровой подписи, например DSA;
- случайные выборы в протоколах аутентификации, например в протоколе Kerberos;
- случайные параметры протоколов для обеспечения уникальности различных реализаций одного и того же протокола, например в протоколах SET и SSL.

Алгоритмы генерации псевдослучайных последовательностей

Конгруэнтные генераторы

Линейными конгруэнтными генераторами являются генераторы следующей формы:

$$X_n = (aX_{n-1} + b) \bmod m,$$

в которых X_n –*n*-й член последовательности, а X_{n-1} – предыдущий член последовательности. Переменные *a*, *b* и *m* – постоянные: *a* – множитель, *b* – инкремент и *m* – модуль. Ключом или затравкой служит значение X_0.

Период такого генератора не больше, чем *m*. Если *a*, *b* и *m* подобраны правильно, то генератор будет *генератором с максимальным периодом*, и его период будет равен *m*. (Например, для линейного конгруэнтного генератора *b* должно быть взаимно простым с *m*).

Если инкремент *b* равен нулю, то есть генератор имеет вид

$$X_n = (aX_{n-1}) \bmod m,$$

и мы получим самую простую последовательность, которую можно предложить для генератора с равномерным распределением. При соответствующем выборе констант $a = 7^5 = 16\,807$ и $m = 2^{31}-1 = 2\,147\,483\,647$ мы получим генератор с максимальным периодом повторения. Эти константы были предложены учеными Парком и Миллером, поэтому генератор вида

$$X_n = (7^5 \cdot X_{n-1}) \bmod 2^{31} - 1$$

называется *генератором Парка-Миллера*.

Преимуществом линейных конгруэнтных генераторов является их быстрота за счет малого количества операций на байт и простота реализации. К сожалению, такие генераторы в криптографии используются достаточно редко, поскольку являются предсказуемыми.

Иногда используют квадратичные и кубические конгруэнтные генераторы, которые обладают большей стойкостью к взлому.

Квадратичный конгруэнтный генератор имеет вид

$$X_n = (aX_{n-1}^2 + bX_{n-1} + c) \bmod m.$$

Кубический конгруэнтный генератор задается как

$$X_n = (aX_{n-1}^3 + bX_{n-1}^2 + cX_{n-1} + d) \bmod m.$$

Для увеличения размера периода повторения конгруэнтных генераторов часто используют их объединение. При этом криптографическая безопасность не уменьшается, но такие генераторы обладают лучшими характеристиками в некоторых статистических тестах.

Пример такого объединения для 32-битовой архитектуры может быть реализован так:

```
// Long должно быть 32-х битовым целым
static long s1 = 1;
static long s2 = 1;
//MODMULT рассчитывает s*b mod m при условии, что m=a*b+c и 0<=c<m
#define MODMULT(a,b,c,m,s) q = s/a; s = b*(s-a*q)-c*q;
if (s<0) s+=m;
double combinedLCG (void)
```

```
  {
    long q;
    long z;
    MODMULT (53668, 40014, 12211, 2147483563L, s1)
    MODMULT (52774, 40692, 3791, 2147483399L, s2)
z = s1 - s2;
if (z<1)
   z += 2147483562;
return z*4.656613e-10;
 }
void InitLCG (long InitS1, long InitS2)
  {
s1 = InitS1;
s2 = InitS2;
  }
```

Этот генератор работает при условии, что архитектура компьютера позволяет представлять все целые числа между $-2^{31}+85$ и $2^{31}-249$. Переменные *s1* и *s2* глобальные и содержат текущее состояние генератора. Перед первым вызовом их необходимо проинициализировать при помощи функции *InitLCG*. Для переменной *s1* начальное значение должно лежать в диапазоне между *1* и *2 147 483 562*, для переменной *s2* – между *1* и *2147483398*. Период такого генератора близок к 10^{18}. Функция *combinedLCG* возвращает действительное псевдослучайное число в диапазоне *(0,1)*. Она объединяет линейные конгруэнтные генераторы с периодами $2^{15}-405$, $2^{15}-1041$ и $2^{15}-1111$, и ее период равен произведению этих трех простых чисел.

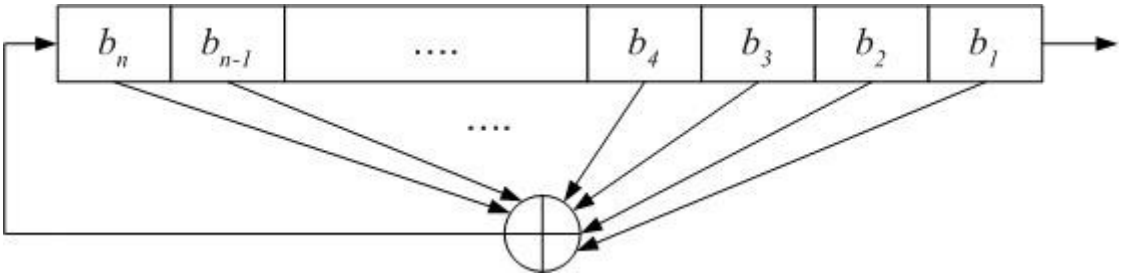
Сдвиговые регистры с обратной связью

Сдвиговый регистр с обратной связью (FSR) состоит из двух частей: *сдвигового регистра* и *функции обратной связи.*

Сдвиговый регистр с обратной связью (FSR)

Сдвиговый регистр представляет собой последовательность битов. Когда нужно извлечь бит, все биты сдвигового регистра сдвигаются вправо на 1 позицию. Новый крайний левый бит является значением функции обратной связи от остальных битов регистра. *Периодом* сдвигового регистра называется длина получаемой последовательности до начала ее повторения.

Простейшим видом сдвигового регистра с обратной связью является *линейный сдвиговый регистр с обратной связью (LFSR – Left Feedback Shift Register)*. Обратная связь представляет собой просто XOR некоторых битов регистра, перечень этих битов называется *отводной*



Сдвиговый регистр с линейной обратной связью (LFSR)

последовательностью.

n-битовый LFSR может находиться в одном из 2^n-1 внутренних состояний. Это означает, что теоретически такой регистр может генерировать псевдослучайную последовательность с периодом 2^n-1 битов. Число внутренних состояний и период равны, потому что заполнение регистра нулями приведет к тому, что он будет выдавать бесконечную последовательность нулей, что абсолютно бесполезно. Только при определенных отводных последовательностях LFSR циклически пройдет через все 2^n-1 внутренних состояний. Такие LFSR называются *LFSR с максимальным периодом*.

Для того чтобы конкретный LFSR имел максимальный период, многочлен, образованный из отводной последовательности и константы *1*, должен быть примитивным по модулю *2*.

Вычисление примитивности многочлена – достаточно сложная математическая задача. Поэтому существуют готовые таблицы, в которых приведены номера отводных последовательностей, обеспечивающих максимальный период генератора. Например, для 32-битового сдвигового регистра можно найти такую запись: *(32,7,5,3,2,1,0)*. Это означает, что для генерации нового бита необходимо с помощью функции XOR просуммировать тридцать второй, седьмой, пятый, третий, второй и первый биты.

Код для такого LFSR на языке C++ будет таким:

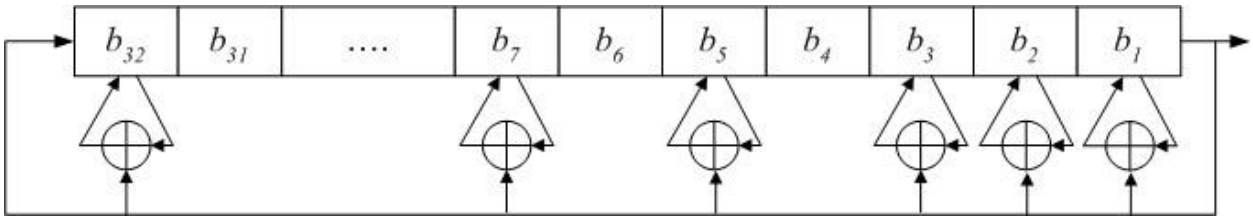
```
int LFSR (void)
{
  static unsigned long ShiftRegister = 1;
  // Любое значение, кроме нуля
  ShiftRegister = ((((ShiftRegister >> 31)
      ^ (ShiftRegister >> 6)
      ^ (ShiftRegister >> 4)
      ^ (ShiftRegister >> 2)
      ^ (ShiftRegister >> 1)
      ^ ShiftRegister)& 0x00000001) <<31)
      | (ShiftRegister >> 1);
  return ShiftRegister & 0x00000001;  }
```

Программные реализации LFSR генераторов достаточно медленны и быстрее работают, если они написаны на ассемблере, а не на языке С. Одним из решений является использование параллельно 16-ти LFSR (или 32 в зависимости от длины слова в архитектуре конкретного компьютера). В такой схеме используется массив слов, размер которого равен длине LFSR, а каждый бит слова массива относится к своему LFSR. При условии, что используются одинаковые

номера отводных последовательностей, то это может дать заметный выигрыш в производительности.

Схему обратной связи также можно модифицировать. При этом генератор не будет обладать большей криптостойкостью, но его будет легче реализовать программно. Вместо использования для генерации нового крайнего левого бита битов отводной последовательности выполняется XOR каждого бита отводной последовательности с выходом генератора и замена его результатом этого действия, затем результат генератора становится новым левым крайним битом.

Эту модификацию называют *конфигурацией Галуа*. На языке С это выглядит следующим образом:



LFSR Галуа

```
static unsigned long ShiftRegister = 1;
void seed_LFSR (unsigned long seed)
{
  if (seed ==0)
    seed = 1;
  ShiftRegister = seed;
}

int Galua_LFSR (void)
{
  if (ShiftRegister & 0x00000001) {
    ShiftRegister = (ShiftRegister ^ mask >> 1) | 0x8000000;
    return 1;
  } else {
    ShiftRegister >>= 1;
    return 0;
  }
}
```

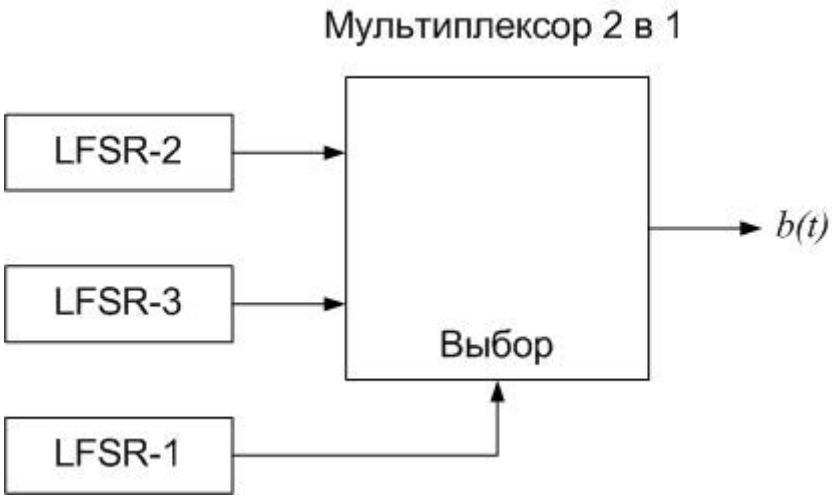
Выигрыш состоит том, что все XOR выполняются за одну операцию. Эта схема также может быть распараллелена.

Сами по себе LFSR являются хорошими генераторами псевдослучайных последовательностей, но они обладают некоторыми нежелательными неслучайными свойствами. Последовательные биты линейны, что делает их бесполезными для шифрования. Для LFSR длины *n* внутреннее состояние представляет собой предыдущие *n* выходных битов генератора. Даже если схема обратной связи хранится в секрете, то она может быть определена по *2n* выходным битам генератора при помощи специальных алгоритмов. Кроме того, большие случайные числа, генерируемые с использованием идущих подряд битов этой последовательности, сильно коррелированны и для некоторых типов приложений не являются случайными. Несмотря на это, LFSR часто используются для создания алгоритмов шифрования. Для этого используются несколько LFSR, обычно с различными длинами и номерами отводных последовательностей. Ключ является начальным состоянием регистров. Каждый раз, когда необходим новый бит, все регистры сдвигаются. Эта операция называется *тактированием*. Бит выхода представляет собой функцию,

желательно нелинейную, некоторых битов LFSR. Эта функция называется *комбинирующей*, а генератор в целом – *комбинирующим генератором*. Многие из таких генераторов безопасны до сих пор.

Генератор Геффа

Одним из комбинирующих генераторов является генератор Геффа. В нем используются три LFSR, объединенные нелинейным способом. LFSR-2 и LFSR-3 являются входами мультиплексора *(рабочие регистры)*, а третий управляет входом мультиплексора. Если длины LFSR равны n_1, n_2, n_3 соответственно, то линейная сложность генератора равна $(n_1 + 1) \cdot n_2 + n_1 \cdot n_3$.

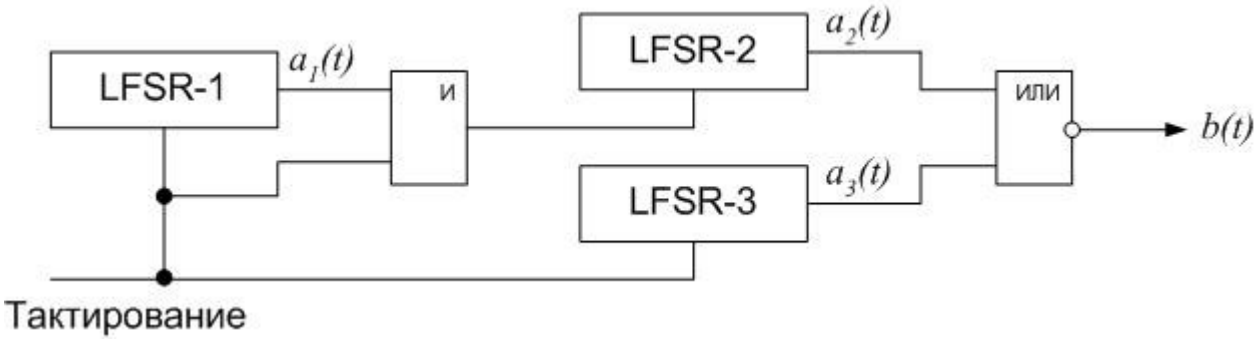


Период такого генератора будет равен наименьшему общему делителю периодов трех генераторов. При условии, что размеры регистров взаимно просты, то период этого генератора будет равен произведению периодов трех LFSR. В обобщенной схеме генератора Геффа используются несколько рабочих LFSR.

Генератор Геффа

Генератор «Стоп-пошел»

Этот генератор использует выход одного LFSR для управления тактовой частотой другого LFSR. Тактовый выход LFSR-2 управляется выходом LFSR-1, так что LFSR-2 может изменять свое состояние в момент времени *t* только, если выход LFSR-1 в момент времени *t–1* был равен *1*.



Генератор «Стоп-пошел»

Аддитивные генераторы

Аддитивные генераторы (называемые иногда запаздывающими генераторами Фиббоначи) очень эффективны, так как их результатом являются случайные слова, а не биты.

Начальное состояние генератора представляет собой массив *n*-битовых слов $X_1, X_2, X_3, \ldots, X_m$. Это первоначальное состояние и является ключом . *i*-е слово генератора получается как

$$X_i = \left(X_{i-a} + X_{i-b} + X_{i-c} + \ldots + X_{i-m}\right) \bmod 2^n .$$

При правильном выборе коэффициентов *a,b,c, …,m* период этого генератора не меньше $2^n - 1$. Для этого должно выполнятся условие взаимной простоты коэффициентов *a,b,c, …,m*. Например, если *a = 55*, а *b = 24*, то мы получим аддитивные генератор с максимальным периодом повторения вида:

$$X_i = \left(X_{i-55} + X_{i-24}\right) \bmod 2^n .$$

Существует несколько модификаций аддитивных генераторов. Самые известные из них – ish, Pike, Mush.

Алгоритм FIPS–186

Этот алгоритм является национальным стандартом США и предназначен для генерации конфиденциальных параметров и ключевой информации для национального стандартного алгоритма электронной цифровой подписи *DSA* [3]. В качестве односторонней функции G может использоваться алгоритм шифрования *DES* или алгоритм хэширования *SHA*.

Входные данные: количество генерируемых псевдослучайных чисел m и 160-битное простое число q.

Выходные данные: последовательность m псевдослучайных чисел $x_i \in \{0, 1, \dots, q-1\}$.

Шаги алгоритма:

1. Задать произвольное число b: $160 \le b \le 512$.
2. Сгенерировать некоторым образом случайное (и секретное) b-битное начальное значение s.
3. Задать вспомогательное 160-битное слово t (в шестнадцатеричной записи):

```
t = 67452301 efcdab89 98badcfe 10325476 c3d2e1f0.
```

4. Для $i = \overline{1, m}$:
   1) произвольно задать b-битное число y_i либо положить его равным нулю;
   2) вычислить $z_i = (s + y_i) \bmod 2^b$;
   3) вычислить $x_i = G(t, z_i) \bmod q$;
   4) вычислить $s = (1 + s + x_i) \bmod 2^b$.
5. Как результат выполнения предыдущего шага формируется псевдослучайная последовательность $x_1, x_2, \dots, x_m$, которую можно использовать в качестве потока псевдослучайных бит.

Алгоритм вычисления значений функции $G(t, c)$:

1. Разбить слово t на пять 32-битных слов:

$$t = H_0 \,||\, H_1 \,||\, H_2 \,||\, H_3 \,||\, H_4.$$

2. К слову c дописать справа $512 - b$ нулей так, чтобы получить 512-битное слово:

$$M = c \,||\, 0^{512-b}.$$

3. Разбить слово U на шестнадцать 32-битных слов $M = M_0 \,||\, M_1 \,||\, \dots \,||\, M_{15}$.
4. Выполнить 1 раз шаг 4 (основной цикл) алгоритма SHA–1, изменяющий H_i (т.е. для $N = 1$).
5. Выходное слово является конкатенацией:

$$G(t, c) = H_0 \,||\, H_1 \,||\, H_2 \,||\, H_3 \,||\, H_4.$$

Примечание: можно использовать стороннюю библиотеку для работы с большими числами (из 160 и более бит).

Алгоритм ANSI X9.17

Этот алгоритм является национальным стандартом США для генерации двоичной псевдослучайной последовательности [2]. Используется в приложениях, обеспечивающих безопасность финансовых платежей, и *PGP*.

В этом генераторе в качестве односторонней функции используется алгоритм шифрования *TripleDES* (*3DES*, *«тройной DES»*). Этот алгоритм использует два 64-битных ключа K_1 и K_2 следующим образом:

$$F_K(M) = E_{K_1}\left(D_{K_2}\left(E_{K_1}(M)\right)\right),$$

где $K = K_1 \,||\, K_2$ – составной 128-битный ключ, $E_{K_1}(M)$ – шифрование сообщения M алгоритмом *DES* с ключом K_1, $D_{K_2}(M)$ – дешифрование сообщения M алгоритмом *DES* с ключом $K_2 \neq K_1$.

Входные данные: некоторое случайное (и секретное) 64-битное начальное значение s_0, 128-битный составной ключ K и m – количество генерируемых 64-битных двоичных слов.

Выходные данные: последовательность m 64-битных двоичных слов $x_1, x_2, \dots, x_m$.

Схематично алгоритм представлен на рисунке 1 следующим образом:

Рисунок 1 – Алгоритм ANSI X9.17

Шаги алгоритма:

1. Зафиксировать 64-битное представление d даты и времени в момент обращения к программе генерации и вычислить вспомогательное 64-битное двоичное слово $Temp = F_K(d)$.
2. Для $i = \overline{1, m}$:
   1) вычислить значение i-го выходного слова $x_i = F_K(Temp \oplus s_{i-1})$;
   2) вычислить новое значение параметра s_i: $s_i = F_K(x_i \oplus Temp)$.
3. В результате предыдущего шага формируется выходная псевдослучайная последовательность из m слов $x_1, x_2, \dots, x_m$ либо двоичная псевдослучайная последовательность из $64 \cdot m$ бит: $X = x_1 \,||\, x_2 \,||\, \dots \,||\, x_m$.

Примечание: алгоритм шифрования *DES* можно не реализовывать самостоятельно, а использовать готовую реализацию.

Алгоритм BBS (Blum-Blum-Shub)

Этот алгоритм основывается на проблеме факторизации больших чисел [2].
Пусть нужно сгенерировать псевдослучайную последовательность длиной m бит.

Шаги алгоритма:

1. Сгенерировать два достаточно больших секретных случайных (и различных) простых числа $p, q \equiv 3 \pmod 4$ и вычислить $N = p \cdot q$.
2. Выбрать случайное стартовое целое число s $(1 \leq s \leq N-1)$ так, что $\text{НОД}(s, N) = 1$. Вычислить $u_0 = s^2 \bmod N$.
3. Для $i = \overline{1, m}$:
   1) вычислить $u_i = u_{i-1}^2 \bmod N$;
   2) вычислить x_i как самый младший бит двоичного представления числа u_i.
4. В результате предыдущего шага формируется выходная псевдослучайная последовательность $x_1, x_2, \dots, x_m$.

Примечание: можно использовать стороннюю библиотеку для работы с большими числами (из 160 и более бит).

Алгоритм RSA (Шамира)

Этот алгоритм основывается на проблеме факторизации больших чисел [2].
Пусть нужно сгенерировать псевдослучайную последовательность длиной m бит.

Шаги алгоритма:

1. Сгенерировать два различных больших простых числа p и q. Вычислить $N = p \cdot q$ и $\phi(N) = (p-1) \cdot (q-1)$.
2. Выбрать случайное целое число k $(1 < k < \phi(N))$, взаимно простое с $\phi(N)$, т.е. $\text{НОД}\big(k, \phi(N)\big) = 1$.
3. Выбрать случайное целое стартовое значение u_0: $1 < u_0 < N - 1$.
4. Для $i = \overline{1, m}$:
   1) вычислить $u_i = u_{i-1}^k \bmod N$;
   2) вычислить $x_i \in \{0,\ 1\}$ – самый младший бит числа u_i в двоичном представлении.

5. В результате предыдущего шага формируется выходная псевдослучайная последовательность $x_1, x_2, \dots, x_m$.

Примечание: можно использовать стороннюю библиотеку для работы с большими числами (из 160 и более бит).

Алгоритм Yarrow–160

Данный алгоритм разработан Б. Шнайером, Дж. Келси и Н. Фергюсоном в 1999 году [6]. В алгоритме для шифрования (функция $E_K()$) может использоваться алгоритм *DES* (или *3DES*), а в качестве хэш-функции $h()$ – *SHA–1*.

Шаги алгоритма:

1. Задание начальных значений:
   1) Задать размер шифруемого сообщения $n = 64$, т.к. для шифрования используется алгоритм *DES*.
   2) Задать $k = 64$ – размер ключа K, используемого при шифровании.
   3) Задать значение P_g ($0 < P_g < 2^{n/3}$, обычно $P_g = 10$), определяющее количество бит, после генерации которых нужно обновить значение ключа K.
   4) Задать значение P_t ($P_t > P_g > 0$), определяющее количество бит, после генерации которых нужно запустить механизм обновления ключа K и счётчика C_i, используя накопитель энтропии (*entropy accumulator*).
      Накопитель энтропии конкатенирует «случайные» данные (текущая дата и время, количество запущенных в системе процессов и т.п.). Пусть v – результат конкатенации накопленных данных.
   5) Задать $t = 0$, где t – количество запусков механизма обновления ключа и счётчика.
   6) Задать некоторое начальное значение n–битного счётчика C_0.
   7) Присвоить $curP_g = P_g$, $curP_t = P_t$.
2. Для $i = \overline{1,m}$ выполнить:
   1) Если $curP_g = 0$, то:
      а) с помощью функции $G(i)$ сгенерировать k бит, которые будут использоваться в качестве нового ключа K;
      б) присвоить $curP_g = P_g$.
   2) Если $curP_t = 0$, то:
      а) вычислить $v_0 = h(v \,||\, t)$;
      б) вычислить $v_i = h(v_{i-1} \,||\, v_0 \,||\, i)$ для $i = 1, \dots, t$;
      в) вычислить $K = H(h(v_t \,||\, K), k)$;
      г) вычислить $C_i = E_K(0)$;
      д) присвоить $curP_t = P_t$, $curP_g = P_g$, $t = t + 1$.
   3) Вычислить $x_i = G(i)$, которое является следующим блоком выходной последовательности.
   4) Выполнить $curP_g = curP_g - 1$ и $curP_t = curP_t - 1$.
3. В результате предыдущего шага формируется выходная псевдослучайная последовательность из m слов $x_1, x_2, \dots, x_m$ либо двоичная псевдослучайная последовательность из $n \cdot m$ бит: $X = x_1 \,||\, x_2 \,||\, \dots \,||\, x_m$.

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

Функция $G(i)$:

1. Вычислить $C_i = (C_{i-1} + 1) \bmod 2^n$.
2. Вернуть $E_K(C_i)$ как результат вычисления функции.

Функция $H(s, k)$:

1. Вычислить $s_0 = s$.
2. Вычислить $s_i = h(s_0 \,||\, \dots \,||\, s_{i-1})$ для $i = 1, 2, \dots$
3. Вернуть первые k бит от конкатенации двоичных слов $s_0 \,||\, s_1 \,||\, \dots$

Примечание: алгоритмы шифрования *DES* и вычисления значения хэш-функции можно не реализовывать самостоятельно, а использовать готовые реализации.

Задание

Реализовать приложение интерфейсом, 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

Дополнительные требования к приложению

- Программа должна быть оформлена в виде удобной утилиты
- Текст программы оформляется прилично (удобочитаемо, с описанием ВСЕХ функций, переменных и критических мест).
- В процессе работы программа ОБЯЗАТЕЛЬНО выдает информацию о состоянии процесса генерации / тестирования.
- Интерфейс программы может быть произвольным, но удобным и понятным (разрешается использование библиотек GUI
- Среда разработки и язык программирования могут быть произвольными.

Примечание: Задание является дифференцированным по сложности. На оценку «Удовлетворительно» достаточно реализовать генератор, приведенный в таблице вариантов в первом столбце, на оценку «хорошо» - в первом и втором столбце, на оценку «отлично» необходимо реализовать все три генератора

Требования для сдачи лабораторной работы:

- Демонстрация работы реализованной вами системы.
- АВТОРСТВО
- Теория (ориентирование по алгоритмам и теоретическим аспектам методов тестирования)
- Оформление и представление письменного отчета по лабораторной работе, который содержит:
    - Титульный лист
    - Задание на лабораторную работу
    - Описание используемых алгоритмов шифрования
    - Листинг программы

Варианты задания

№	На оценку «Удовлетворительно»	На оценку «Хорошо»	На оценку «Отлично»
1	Линейный конгруэнтный генератор	Генератор Геффа	FIPS-186
2	Квадратичный конгруэнтный генератор	Генератор Стоп-Пошел	ANSI X9.17
3	Кубический конгруэнтный генератор	Аддитивный генератор	RSA
4	Генератор Парка-Миллера	Генератор BBS	Yarrow-160
5	Линейный конгруэнтный генератор	Генератор Стоп-Пошел	ANSI X9.17
6	Квадратичный конгруэнтный генератор	Аддитивный генератор	RSA
7	Кубический конгруэнтный генератор	Генератор BBS	Yarrow-160
8	Генератор Парка-Миллера	Генератор Геффа	FIPS-186
9	Линейный конгруэнтный генератор	Аддитивный генератор	RSA
10	Квадратичный конгруэнтный генератор	Генератор BBS	Yarrow-160
11	Кубический конгруэнтный генератор	Генератор Геффа	ANSI X9.17
12	Генератор Парка-Миллера	Генератор Стоп-Пошел	FIPS-186
13	Линейный конгруэнтный генератор	Генератор BBS	RSA
14	Квадратичный конгруэнтный генератор	Генератор Геффа	Yarrow-160
15	Кубический конгруэнтный генератор	Генератор Стоп-Пошел	ANSI X9.17
16	Генератор Парка-Миллера	Генератор BBS	FIPS-186
17	Линейный конгруэнтный генератор	Генератор Геффа	RSA

Приложение №1. Таблица коэффициентов для линейного конгруэнтного генератора

Переполняется при	a	B	m
2^{20}	106	1283	6075
2^{21}	211	1663	7875
2^{22}	421	1663	7875
2^{23}	430	2531	11979
2^{23}	936	1399	6655
2^{23}	1366	1283	6075
2^{24}	171	11213	53125
2^{24}	859	2531	11979
2^{24}	419	6173	29282
2^{24}	967	3041	14406
2^{25}	141	28411	134456
2^{25}	625	6571	31104
2^{25}	1541	2957	14000
2^{25}	1741	2731	12960
2^{25}	1291	4621	21870
2^{25}	205	29573	139968
2^{26}	421	17117	81000
2^{26}	1255	6173	29282
2^{26}	281	28411	134456
2^{27}	1093	18257	86436
2^{27}	421	54773	259200
2^{27}	1021	24631	116640
2^{27}	1021	25673	121500
2^{28}	1277	24749	117128
2^{28}	741	66037	312500
2^{28}	2041	25673	121500
2^{29}	2311	25367	120050
2^{29}	1807	45289	214326
2^{29}	1597	51749	244944
2^{29}	1861	49297	233280
2^{29}	2661	36979	175000
2^{29}	4081	25673	121500
2^{29}	3661	30809	145800
2^{30}	3877	29573	139968
2^{30}	3613	45289	214326
2^{30}	1366	150889	714025
2^{31}	8121	28411	134456
2^{30}	4561	51349	243000
2^{30}	7141	54773	259200
2^{32}	9301	49297	233280
2^{32}	4096	150889	714025
2^{33}	2416	374441	1771875
2^{34}	17221	107839	510300