Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

**ТЕОРИЯ ПСЕВДОСЛУЧАЙНЫХ ГЕНЕРАТОРОВ**

**ОТЧЕТ ПО ПРАКТИЧЕСКОМУ КУРСУ**

студента 4 курса 431 группы

факультета компьютерных наук и информационных технологий

*Кузнецова Егора Дмитриевича*

фамилия, имя, отчество

Научный руководитель

Ст. преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_ И.И. Слеповичев

подпись, дата

Саратов 2024

**ПРАКТИЧЕСКАЯ РАБОТА**

## Задание 1. Генерация псевдослучайных чисел.

Описание задания: создать программу, генерирующую псевдослучайные числа из заданного диапазона. Входные параметры алгоритмы передаются программе через строку параметров (через файл с параметрами). Выходные значения записываются в файл, указанный в параметре запуска программы.

### Алгоритм 1. Линейный конгруэнтный метод.

**Описание алгоритма.**

Одним из простых и популярных методов сейчас является *линейный конгруэнтный метод* (ЛКМ), предложенный Д.Г. Лехмером в 1949 году. В его основе лежит выбор четырех ключевых чисел:

* , модуль;
* , множитель;
* , приращение (инкремент);
* , начальное значение.

**Определение** Последовательность ПСЧ, получаемая по формуле:

, (3.1)

называется *линейной конгруэнтной последовательностью* (ЛКП). Ключом для неё служит .

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g lc -i 7875 211 1663 347,

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

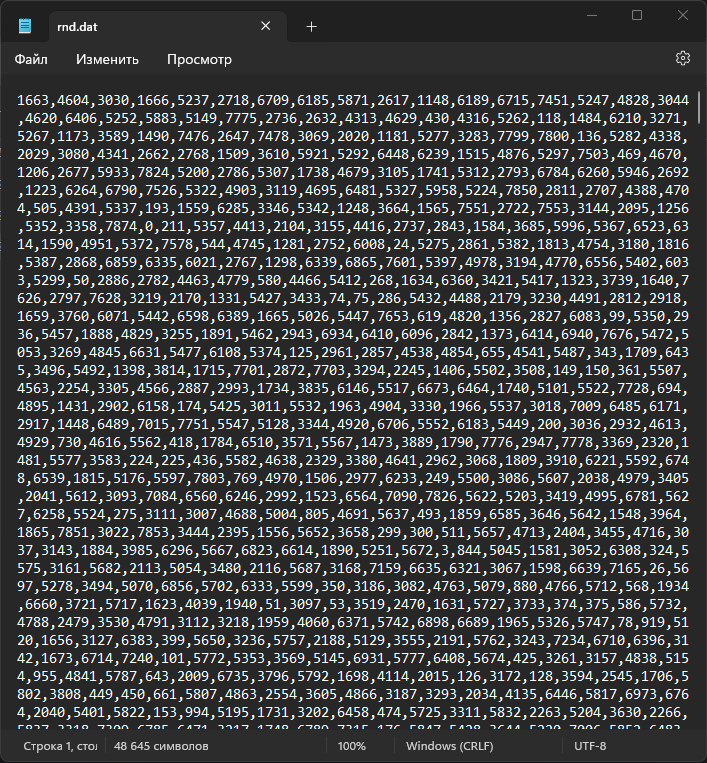


Рис. 1 – Результат генерации линейным конгруэнтным методом.

**Исходный текст программы.**

def lc(m, a, c, x0, n=10000):  
 res = [x0]  
 for i in range(n):  
 res.append((a \* res[i] + c) % m)  
 progress\_bar(i, n)  
  
 return res

### Алгоритм 2. Аддитивный метод.

**Описание алгоритма.**

Идею рекурсивного вычисления значения можно обобщить до формулы, использующей два предыдущих значения последовательности. Например, мы можем рекуррентно вычислять значение как линейную комбинацию значения и . Тогда, максимальная длина последовательности в лучшем случае будет равна *m*2, так как последовательность не будет повторяться, пока не будет получено равенство . Простейшая последовательность, в которой зависит более чем от одного предыдущего значений, это последовательность Фибоначчи

.

Данный генератор рассматривался в начале 1950-х, и обычно он дает длину периода, большую, чем *m*. Однако числа, получаемые с помощью рекуррентного соотношения Фибоначчи, недостаточно случайны. Но можно сделать еще один шаг обобщения и использовать формулу

.

Например, Дж. Ж. Митчел и Д.Ф. Мур в 1958 году предложили последовательность, определенную так:

, (3.4)

где *m* – четное число, а – произвольные целые числа.

Числа *k* и j обычно называют запаздыванием, а последовательность 3.4, – последовательностью Фибоначчи с запаздыванием. Для ПСЧ Фибоначчи с запаздыванием справедлива.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g add -i 1234 19 23 4554 8697 9253 4731 6588 2864 4025 9800 8350 7638 5579 5403 9173 6399 8018 1443 9191 8998 3132 1185 3089 1356 9178 8828 8883 5223 3437 5030 1228 7052 3614 1972 6955 7056 8346 7357 8601 8179 5438 8824 5298

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

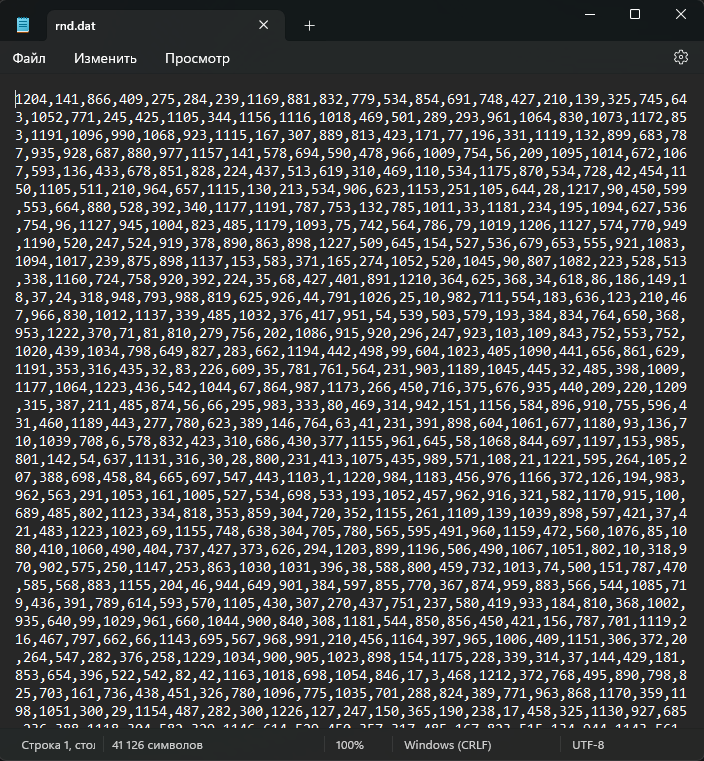


Рис. 2 – Результат генерации аддитивным методом.

**Исходный текст программы.**

def add(m, li, ui, arr, n=10000):  
 res = []  
 tmp = arr.copy()  
 for i in range(n):  
 new\_elem = (tmp[li - 1] + tmp[ui - 1]) % m  
 res.append(new\_elem)  
 tmp.append(new\_elem)  
 tmp.pop(0)  
 progress\_bar(i, n)  
 return res

### Алгоритм 3. Пятипараметрический метод.

**Описание алгоритма.**

Данный метод является частным случаем РСЛОС, использует характеристический многочлен из 5 членов и позволяет генерировать последовательности *w*-битовых двоичных целых чисел в соответствии со следующей рекуррентной формулой:

Параметры и , первоначально задают как начальный вектор.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g 5p -i 107 31 57 82 17 10100101000011011001100001

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

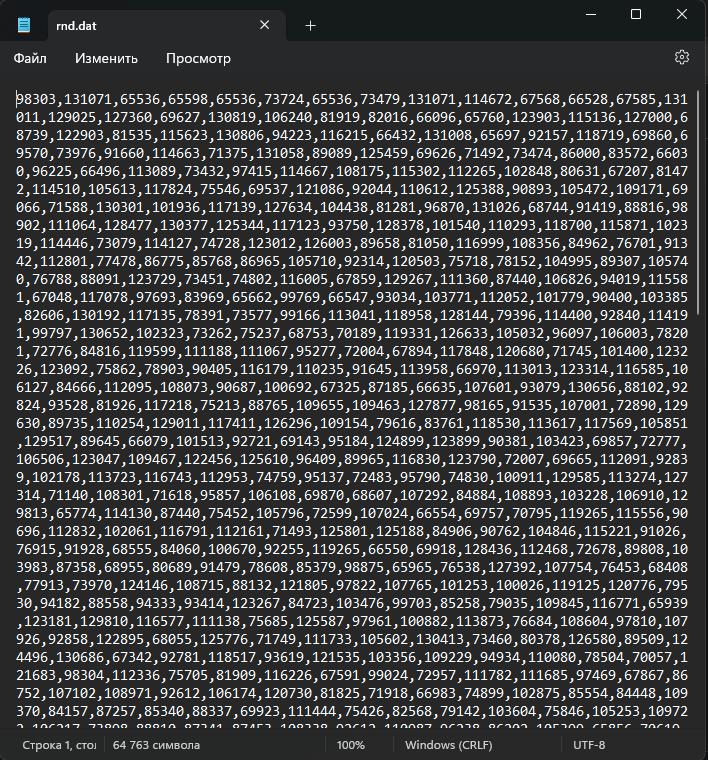


Рис. 3 – Результат генерации пятипараметрическим методом.

**Исходный текст программы.**

def p5(p, q1, q2, q3, w, x0, n=10000):  
 res = []  
 x0 = int(x0, base=2)  
 for i in range(n):  
 new\_elem = 1  
 for \_ in range(w - 1):  
 new\_bit1 = (x0 >> p - q1) & 1  
 new\_bit2 = (x0 >> p - q2) & 1  
 new\_bit3 = (x0 >> p - q3) & 1  
 last\_bit = p & 1  
 xor = new\_bit1 ^ new\_bit2 ^ new\_bit3 ^ last\_bit  
 new\_elem = (new\_elem << 1) | xor  
 x0 = (x0 >> 1) | (xor << p - 1)  
 progress\_bar(i, n)  
 res.append(new\_elem)  
 return res

### Алгоритм 4. Регистр сдвига с обратной связью (РСЛОС).

**Описание алгоритма.**

**Определение** *Регистр сдвига с обратной линейной связью* (РСЛОС) – регистр сдвига битовых слов, у которого входной (вдвигаемый) бит является линейной функцией остальных битов. Вдвигаемый вычисленный бит заносится в ячейку с номером 0. Количество ячеек *p* называют длиной регистра.

Для натурального числа *p* и , принимающих значения 0 или 1, определяют рекуррентную формулу

(3.7)

Как видно из формулы, для РСЛОС функция обратной связи является линейной булевой функцией от состояний всех или некоторых битов регистра.

Одна итерация алгоритма, генерирующего последовательность, состоит из следующих шагов:

1. Содержимое ячейки формирует очередной бит ПСП битов.
2. Содержимое ячейки 0 определяется значением функции обратной связи, являющейся линейной булевой функцией с коэффициентами . Его вычисляют по формуле 3.7.
3. Содержимое каждого *i*-го бита перемещается в -й, .
4. В ячейку 0 записывается новое содержимое, вычисленное на шаге 2.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g lfsr -i 10000000011011 10010010011001

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

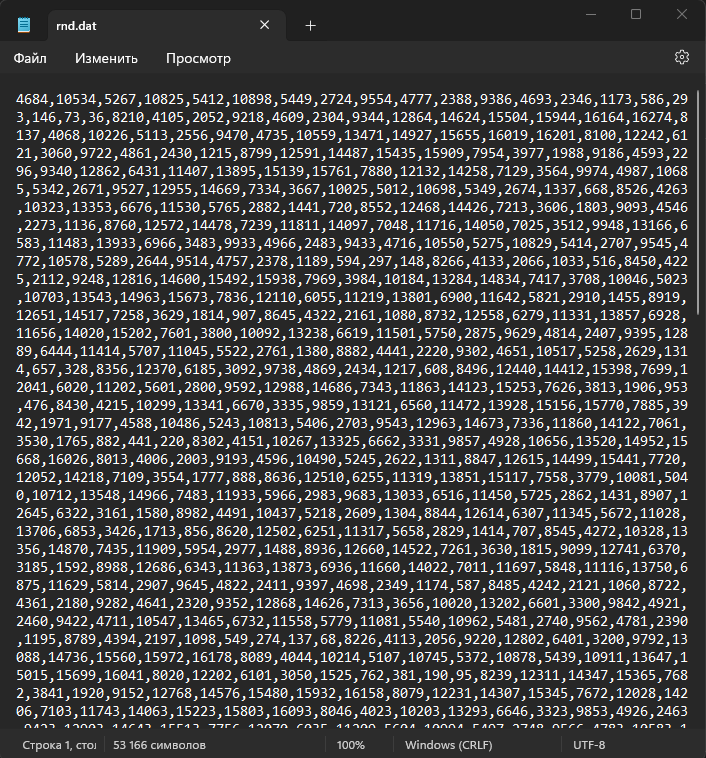


Рис. 4 – Результат генерации РСЛОС.

**Исходный текст программы.**

def lfsr(vec\_of\_cf, register, n=10000):  
 res = []  
 len\_reg = len(register)  
 vec\_of\_cf = int(vec\_of\_cf, 2)  
 register = int(register, 2)  
 for i in range(n):  
 new\_bit = (register ^ ((register ^ vec\_of\_cf) >> 1)) & 1  
 register = (register >> 1) | (new\_bit << (len\_reg - 1))  
 res.append(register)  
 progress\_bar(i, n)  
 return res

### Алгоритм 5. Нелинейная комбинация РСЛОС.

**Описание алгоритма.**

**Определение**. *Алгебраическая нормальная форма функции* – запись функции суммой по модулю 2 произведений порядков *m* независимых переменных . *Нелинейным порядком* функции *f* называется максимальный порядок членов в записи её алгебраической нормальной формы.

Пример. Функция имеет нелинейный порядок 3.

Предположим теперь, что у нас регистров сдвига с линейной обратной связью , их длины попарно различны и больше двух. Подобно комбинированному методу Таусворта, мы можем объединить эти РСЛОС, но уже при помощи нелинейной функции *f:*

.

Тогда *линейная сложность* (понятие линейной сложности описано в п. 4.1) потока ключей равна . Если – попарно взаимно простые числа, то длина периода последовательности равна:

.

Генератор Геффа является примером нелинейной комбинации РСЛОС. В этом генераторе используются три РСЛОС, объединённые нелинейным образом. Длины этих регистров – попарно простые числа. Нелинейная функция генератора:

Длина периода ПСЧ: .  
Линейная сложность: .

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g nfsr -i 10000000011011 100000000101011 1000000000010000000 23 5342 8912 23446

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

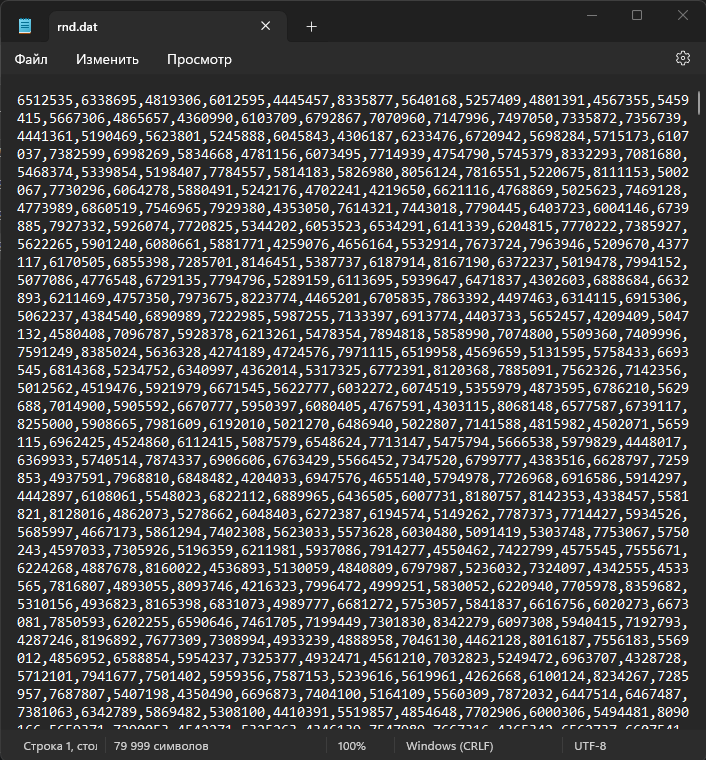


Рис. 5 – Результат генерации нелинейной комбинацией РСЛОС.

**Исходный текст программы.**

def nfsr(R1, R2, R3, w, x1, x2, x3, n=10000):  
 res = []  
 len\_R1 = len(R1)  
 len\_R2 = len(R2)  
 len\_R3 = len(R3)  
 for i in range(n):  
 new\_elem = 1  
 for \_ in range(w - 1):  
 xor\_R1 = (x1 ^ (x1 >> 1))  
 xor\_R2 = (x2 ^ (x2 >> 1))  
 xor\_R3 = (x3 ^ (x3 >> 1))  
 out = ((xor\_R1 ^ xor\_R2) + (xor\_R2 ^ xor\_R3) + xor\_R3) & 1  
 x1 = (x1 >> 1) | (out << len\_R1 - 1)  
 x2 = (x2 >> 1) | (out << len\_R2 - 1)  
 x3 = (x3 >> 1) | (out << len\_R3 - 1)  
 new\_elem = (new\_elem << 1) | out  
 res.append(new\_elem)  
 progress\_bar(i, n)  
 return res

### Алгоритм 6. Вихрь Мерсенна.

**Описание алгоритма.**

Данный ГПСЧ является РСЛОС, состоящим из 624 ячеек по 32 бита. Метод Вихрь Мерсенна позволяет генерировать последовательность двоичных псевдослучайных целых *w*-битовых чисел в соответствии со следующей рекуррентной формулой

где *p,q,r* – целые константы, *p* – степень рекуррентности, ;

– *w*-битовое двоичное целое число;

– двоичное целое число, полученное конкатенацией чисел и , когда первые (*w-r*) битов взяты из , а последние *r* битов из в том же порядке;

*А* – матрица размера *w*×*w,* состоящая из нулей и единиц, определенная посредством *а*;

*XA* – произведение, при вычислении которого сначала выполняют операцию (сдвига битов на одну позицию вправо), если последний бит *X* равен 0, а затем, когда последний бит , вычисляют ,

Алгоритм Вихрь Мерсенна состоит из попеременного выполнения процедур *рекурсивной генерации* и «*закалки*». Рекурсивная генерация представляет из себя РСЛОС с дополнительной рекурсивной функцией для потока выходных битов. Операция «закалки» является процедурой, усиливающей равномерность распределения на больших размерностях битовых векторов.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g mt -i 1739 27541

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

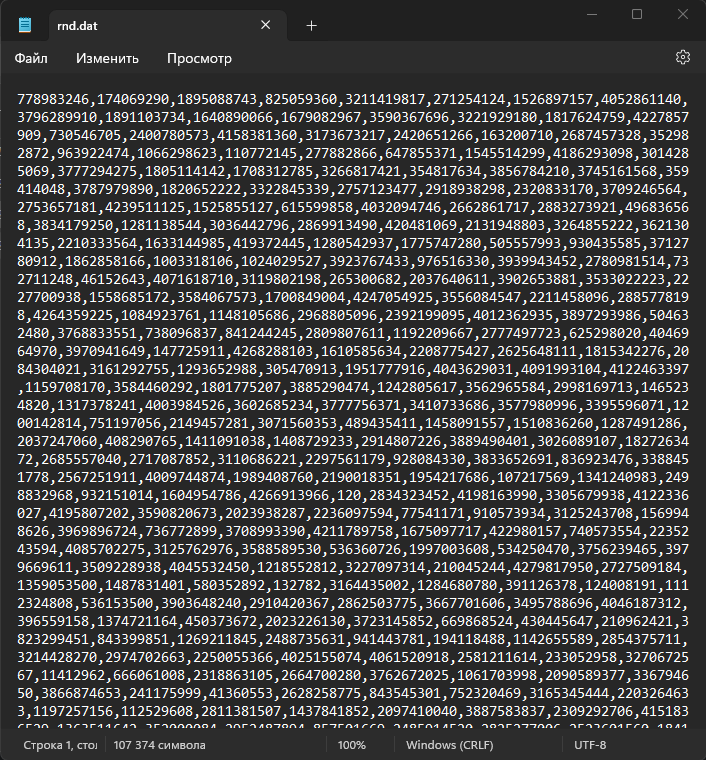


Рис. 6 – Результат генерации вихрем Мерсенна.

**Исходный текст программы.**

def mt(x0, m, n=10000):  
 res = []  
 w = 32  
 start = [0] \* m  
 start[0] = x0  
 for i in range(1, m):  
 start[i] = (start[i - 1] ^ (start[i - 1] >> 30) + i) & ((1 << w) - 1)  
 for i in range(n):  
 index = m  
 u, s, t, l, w = 11, 7, 15, 18, 32  
 b = 0x9D2C5680  
 c = 0xEFC60000  
 if index >= 624:  
 p, r, q = 624, 31, 397  
 a = 0x9908B0DF  
 for j in range(p):  
 rand\_num = (start[j] >> r) + (start[(j + 1) % p] & ((1 << r) - 1))  
 start[j] = start[(j + q) % p] ^ (rand\_num >> 1)  
 if rand\_num % 2 != 0:  
 start[j] ^= a  
 index = 0  
 rand\_num = start[index]  
 rand\_num ^= (rand\_num >> u)  
 rand\_num ^= ((rand\_num << s) & b)  
 rand\_num ^= ((rand\_num << t) & c)  
 rand\_num ^= (rand\_num >> l)  
 index += 1  
 rand\_num = rand\_num & ((1 << w) - 1)  
 res.append(rand\_num)  
 progress\_bar(i, n)  
 return res

### Алгоритм 7. RC4.

**Описание алгоритма.**

Алгоритм RC4 назван по имени его создателя – Ron’s Cipher (Рон Ривест). Алгоритм был разработан компанией RSA Data Security, Inc. в 1987 году. Являясь потоковым шифром, в основе которого генератор псевдослучайных чисел, RC4 широко используется в различных криптографических протоколах. Достоинством алгоритма является высокая скорость работы и переменный размер ключа.

**Описание алгоритма.**

1. Инициализация ,
2. Итерация алгоритма:

Стойкость алгоритма основана на следующем наблюдении: даже если оппонент узнал ключ *К* и номер шага *i*, он может вычислить всего лишь значение , но не всё внутреннее состояние массива. Это следует из того, что оппонент не в состоянии определить значение переменной *t*, не зная *j*, .

Каждый шаг алгоритма усиливает его стойкость:

* шаг a обеспечивает однократное использование элементов массива;
* шаг b обеспечивает нелинейную зависимость выхода от массива;
* шаг c изменяет массив в процессе итераций;
* шаг d скрывает внутреннее состояние массива от анализа.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g rc4 -i 6351 3704 4923 1474 5645 6822 3883 176 9085 4332 6685 9105 6465 7498 7135 1332 5292 3628 652 5450 4067 5662 2147 8960 1277 172 7806 4761 4329 6177 4456 7641 4436 3240 7494 6443 5762 1058 649 5529 5519 9977 9273 1496 7921 166 8305 236 285 8714 2090 9705 8574 2402 6201 3155 5350 6151 4046 1400 6338 3189 4204 3150 4201 2413 8530 7264 6297 8383 1479 2664 1398 7844 782 9460 229 2626 2791 8092 5799 5724 7058 9785 7339 3185 1528 5291 5027 7166 5071 9610 2926 3471 1463 597 3393 1762 3512 7344 6515 1459 5912 6564 6319 6014 8143 1554 522 702 1601 2426 6722 155 938 9488 5847 9555 7628 1155 9722 4224 9394 8444 5295 1425 3543 4528 4817 1242 161 165 279 6401 7959 6240 6887 9219 5568 965 2301 9204 5893 7397 8363 2030 8487 1312 1612 9566 3730 1485 2770 5892 8043 2885 9362 9904 6786 8677 6101 1278 2943 2310 3788 1289 4660 2020 3704 3064 3774 3324 7969 7801 7260 763 1942 3162 500 5842 6723 5967 2212 2083 9458 9729 2421 1909 8299 8313 9682 4887 940 8109 1542 1550 8522 3148 1341 5363 9064 3916 7378 4343 3816 6892 5182 9718 3644 7796 6816 2675 4105 9727 8948 4167 4998 8069 2337 3061 2442 8765 193 2444 3068 4511 7936 3041 5816 6859 7561 8078 7538 9498 4181 8674 4740 4663 6282 615 1275 3779 3937 8286 4726 9381 3979 9259 1348 5102 3434 2104 2880 2195 3237 7727

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

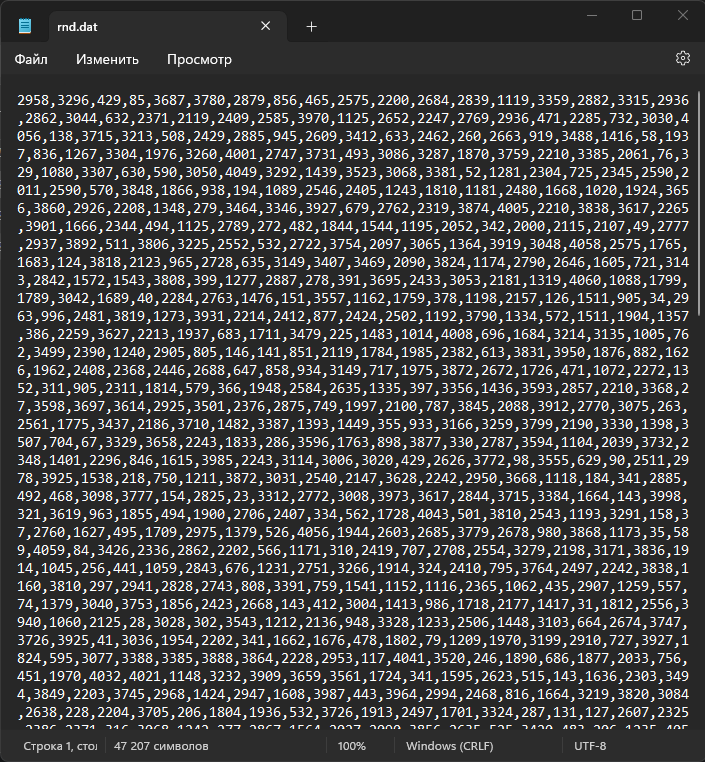


Рис. 7 – Результат генерации RC4.

**Исходный текст программы.**

def rc4(vec, n=10000):  
 vec\_size = len(vec)  
 S = list(range(4096))  
 j = 0  
 for i in range(4096):  
 j = (j + S[i] + vec[i % vec\_size]) % 4096  
 S[i], S[j] = S[j], S[i]  
  
 f = 0  
 s = 0  
 res = []  
 for i in range(n):  
 f = (f + 1) % 4096  
 s = (s + S[f]) % 4096  
 S[f], S[s] = S[s], S[f]  
 new\_elem = S[(S[f] + S[s]) % 4096]  
 res.append(new\_elem)  
 progress\_bar(i, n)  
 return res

### Алгоритм 8. ГПСЧ на основе RSA.

**Описание алгоритма.**

Эди Шамир предложил использовать алгоритм шифрования с открытым ключом RSA для генерации ПСЧ [19]. В его работе показано, что предсказание выхода генератора псевдослучайных чисел равносильно взлому RSA. Очевидным недостатком такого алгоритма является низкая скорость и громоздкость реализации.

Дальнейшей модификацией этого алгоритма является ГПСЧ RSA, который также основан на сложности решения проблемы RSA.

**Описание алгоритма.**

1. Сгенерировать два секретных простых числа *p* и *q*, а также и . Выбрать случайное целое число , такое что
2. Выбрать случайное целое – начальный вектор из интервала
3. .

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g rsa -i 183292009640395590214888670943599889254407673336479468817820022395501420474923891839974832526263583707218142346311235634273205098061509171660517990143484111795063123224329671654802381997553865015352836802354736706323154667318615994555262815691223037591248696879026124585947695957972205849026957552289 721740113940525878464693165169605984115746921202274439947798437099874098831534693267276910373674126915339893910906955771122520186649471327469569713468941776292589145842712981890451759924566220243844660846376820048287571598677174279462665931090123124139379039091295993702208396683165163952858982396937 9 738757207292049099681186263047264483621186264658013413340565144230403446314908771216666695670040613851377887665768240265717648919534745092259554784688980071746560036939207049777629182048114446081198394504024887869085513953906106436793914226972714932196077148543042413224732428045682566262016086615551

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

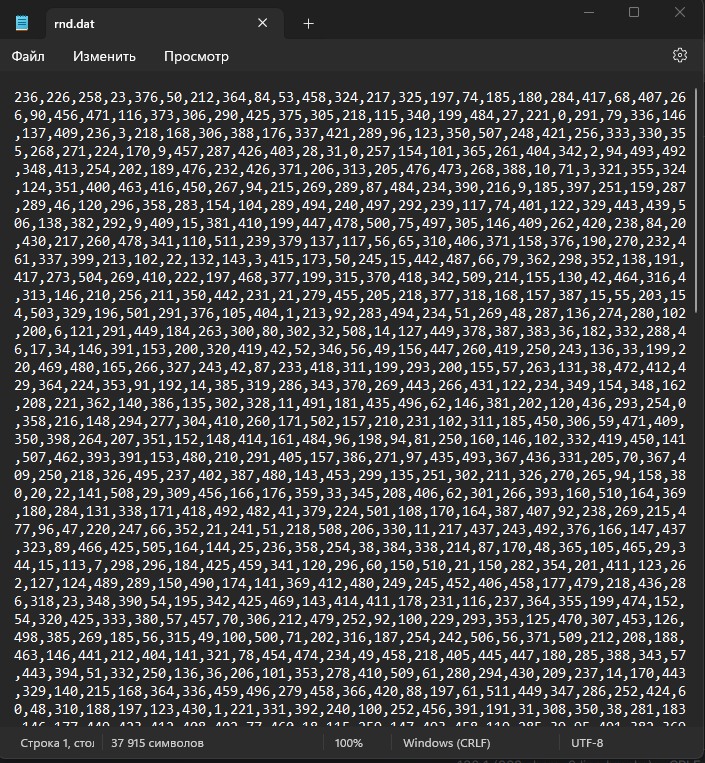


Рис. 8 – Результат генерации ГПСЧ на основе RSA.

**Исходный текст программы.**

def rsa(n, e, w, x, num=10000):  
 res = []  
 for i in range(num):  
 new\_elem = 0  
 for \_ in range(w):  
 x = pow(x, e, n)  
 new\_elem = (new\_elem << 1) | (x & 1)  
 res.append(new\_elem)  
 progress\_bar(i, num)  
 return res

### Алгоритм 9. Алгоритм Блюма-Блюма-Шуба.

**Описание алгоритма.**

Алгоритм был предложен в 1986 году Ленор Блюм, Мануэлем Блюмом и Майклом Шубом [20]. В основе алгоритма – использование квадратичных остатков по модулю *n*. На текущее время это один из самых простых и быстрых алгоритмов ГПСЧ, использующих вычислительно сложные задачи.

**Описание алгоритма [10].**

**На входе:** Длина *l*.

**На выходе:** Последовательность псевдослучайных бит .

1. Сгенерировать два простых числа *p* и *q*, сравнимых с 3 по модулю 4. Это гарантирует, что каждый квадратичный вычет имеет один квадратный корень, который также является квадратичным вычетом. Произведение этих чисел – *n=pq* является целым числом Блюма. Выберем другое случайное целое число *x*, взаимно простое с *n*.
2. Вычислим , которое будет начальным вектором.
3. .

Интересным достоинством этого генератора является то, что для получения *i*-го бита при известных *p* и *q* достаточно воспользоваться формулой

Это дает преимущества данному ГПСЧ при работе с массивами данных с произвольной точкой доступа (random access data).

Безопасность алгоритма основана на сложности разложения *n* на множители. Этот генератор безопасен как для предсказания следующего бита последовательности, так и для определения предыдущего бита. Это свойство следует из свойств задачи разложения *n* на множители.

Недостатком алгоритма является его низкая скорость, что делает его малоприменимым для использования в потоковых шифрах.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

prng.exe -g bbs -i 24 13

где параметр

-g – выбор алгоритма генерации,

-i – входные параметры в алгоритме генерации.

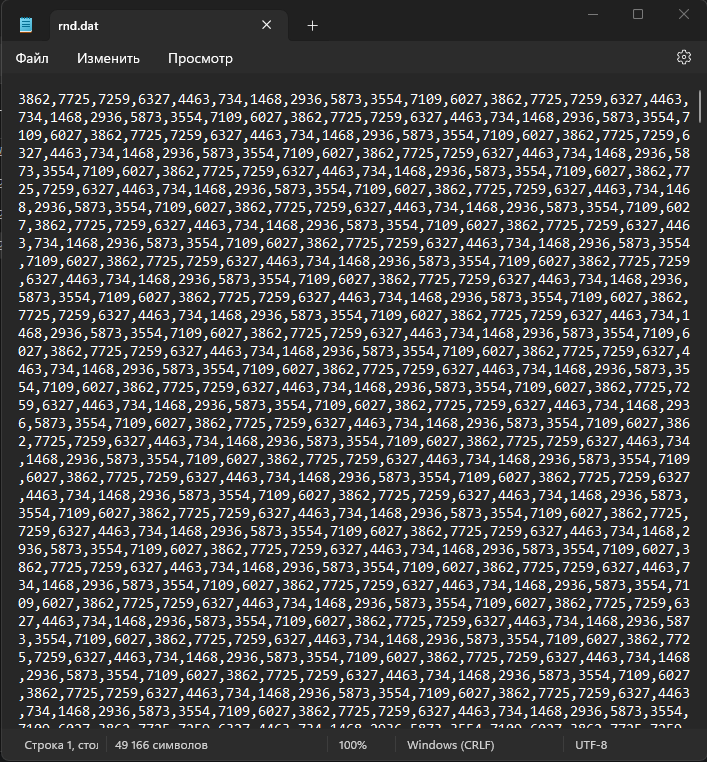


Рис. 9 – Результат генерации алгоритмом Блюма-Блюма-Шуба.

**Исходный текст программы.**

def bbs(x, w, num=10000):  
 p = 127  
 q = 131  
 n = p \* q  
 res = []  
 for i in range(num):  
 new\_elem = 0  
 for \_ in range(w):  
 x = pow(x, 2, n)  
 new\_elem = (new\_elem << 1) | (x & 1)  
 res.append(new\_elem)  
 progress\_bar(i, num)  
 return res

**Задание 2. Преобразование ПСЧ к заданному распределению.**

Создать программу для преобразования последовательности ПСЧ в другую последовательность ПСЧ с заданным распределением:

1. Стандартное равномерное с заданным интервалом;
2. Треугольное распределение;
3. Общее экспоненциальное распределение;
4. Нормальное распределение;
5. Гамма распределение (для параметра *c*=*k*);
6. Логнормальное распределение;
7. Логистическое распределение;
8. Биномиальное распределение.

## Алгоритм 1. Стандартное равномерное с заданным интервалом.

Если максимальное значение равномерного целого случайного числа равно , для генерации стандартных равномерных случайных чисел необходимо применять следующую формулу:

Если стандартное равномерное случайное число получено методом, установленным в предыдущем параграфе, то равномерное случайное число должно быть получено в соответствии со следующей формулой:

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d st -p1 119 -p2 2147

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2 – входные параметры в алгоритме преобразования.

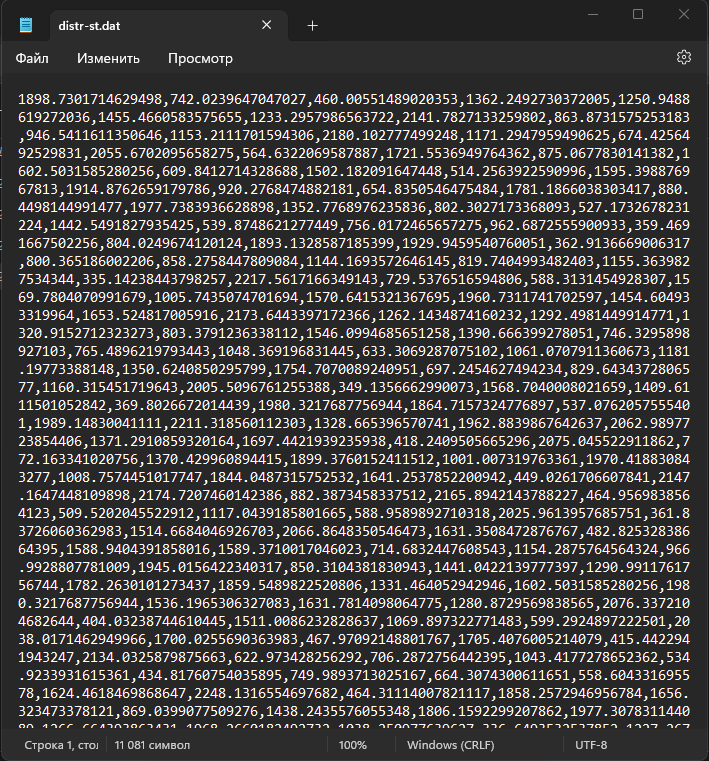
****

Рис. 10 – Результат преобразования.

**Исходный текст программы.**

def st(a, b, n):  
 m = max(n) + 1  
 return [x / m \* b + a for x in n]

## Алгоритм 2. Треугольное распределение.

Если стандартные случайные числа и независимо получены методом генерации стандартного равномерного числа, то случайное число *Y*, подчиняющееся треугольному распределению, определяют по формуле .

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d tr -p1 1279 -p2 1293

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2 – входные параметры в алгоритме преобразования.

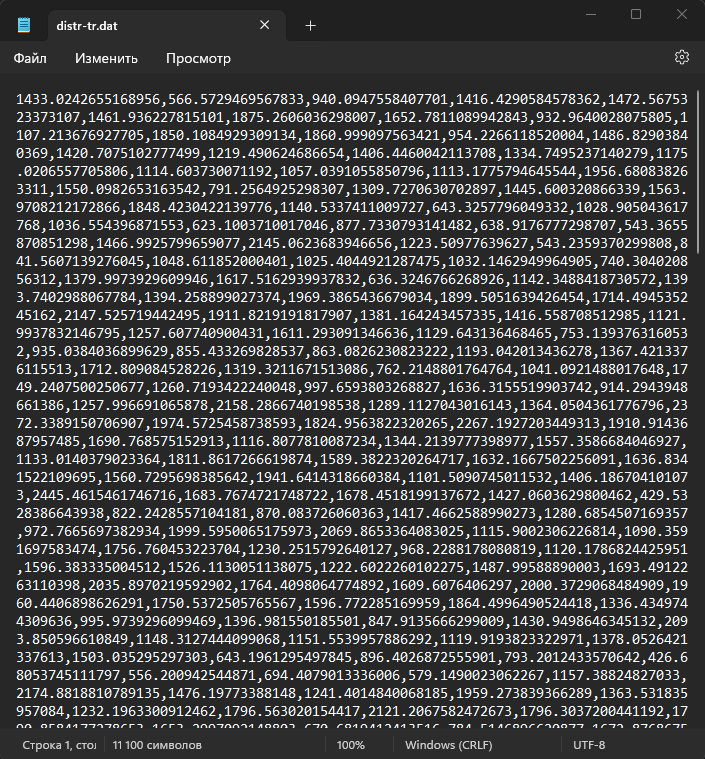
****

Рис. 11 – Результат преобразования.

**Исходный текст программы.**

def tr(a, b, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 return [a + b \* (n[i] + n[i + 1] - 1) for i in range(len(n) - 1)]

## Алгоритм 3. Общее экспоненциальное распределение.

Если стандартное равномерное случайное число *U* генерировано одним из методов, установленным в разделе 2, то случайное число, соответствующее экспоненциальному распределению, получают по формуле

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d ex -p1 1293 -p2 1279

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2 – входные параметры в алгоритме преобразования.

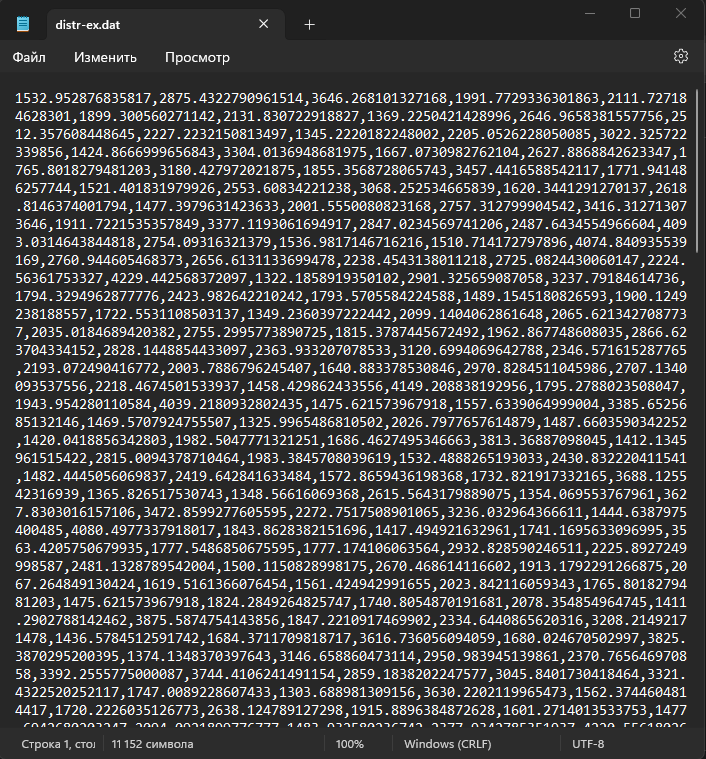
****

Рис. 12 – Результат преобразования.

**Исходный текст программы.**

def ex(a, b, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 return [-b \* math.log(u) + a for u in n]

## Алгоритм 4. Нормальное распределение.

Если стандартные равномерные случайные числа и независимо сгенерированы методом, установленным в разделе 2, то два независимых нормальных случайных числа получают в соответствии со следующей процедурой

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d nr -p1 1293 -p2 277

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2 – входные параметры в алгоритме преобразования.

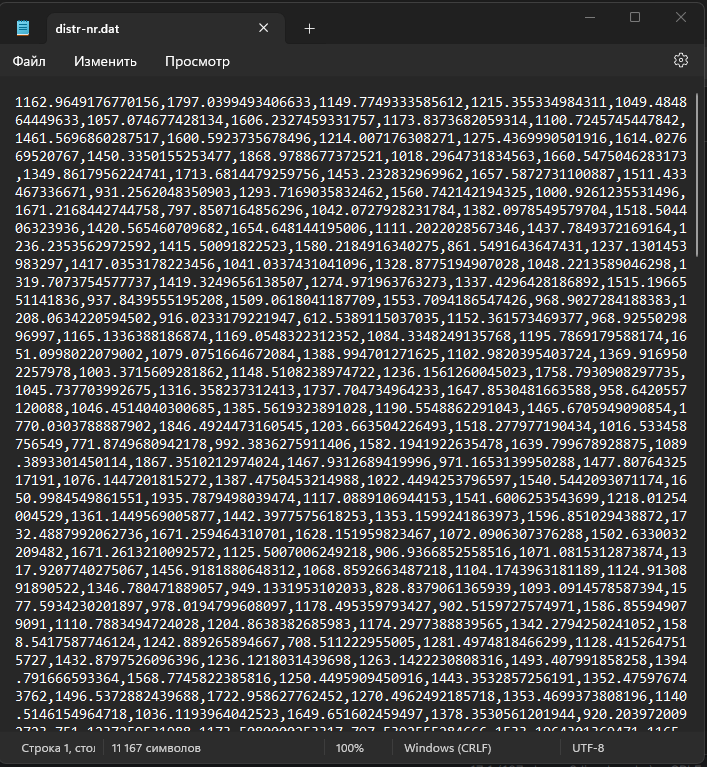
****

Рис. 13 – Результат преобразования.

**Исходный текст программы.**

def nr(a, b, n):  
 result = []  
 m = max(n) + 1  
 n = [x / m for x in n]  
 for i in range(0, len(n) - 1, 2):  
 z1 = a + b \* math.sqrt(-2 \* math.log(1 - n[i])) \* math.cos(2 \* math.pi \* n[i + 1])  
 z2 = a + b \* math.sqrt(-2 \* math.log(1 - n[i])) \* math.sin(2 \* math.pi \* n[i + 1])  
 result.append(z1)  
 result.append(z2)  
 return result

## Алгоритм 5. Гамма распределение.

где a,b,c – параметры положения, масштаба и формы соответственно.

Используя независимые равномерные случайные числа , применяют формулу

.

Примечание. Этим методом для и может быть получено распределение Хи-квадрат с четным числом степеней свободы.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d gm -p1 1293 -p2 1279 -p3 14

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2, -p3 – входные параметры в алгоритме преобразования.

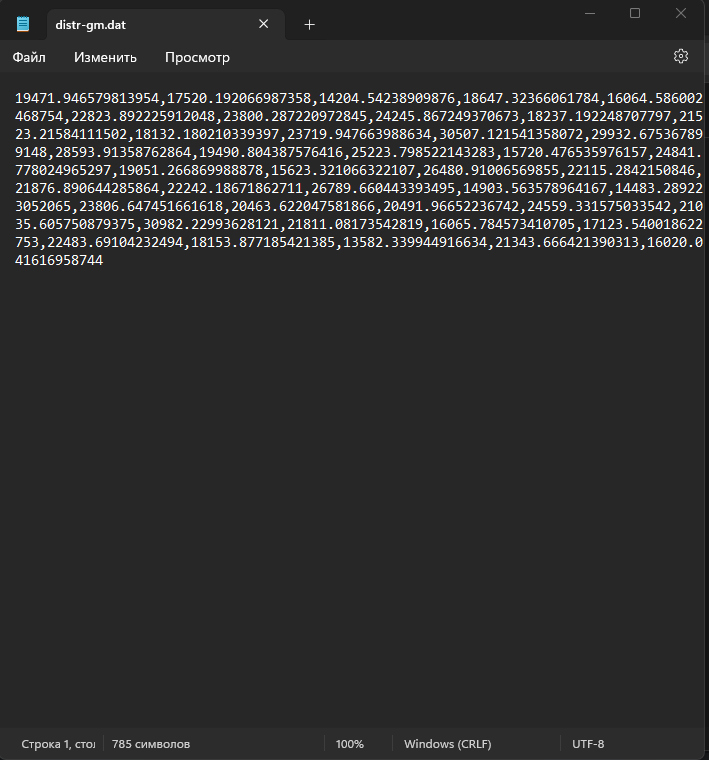
****

Рис. 14 – Результат преобразования.

**Исходный текст программы.**

def gm(a, b, k, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 list = [n[i: i + k] for i in range(0, len(n), k)]  
 if len(list[-1]) != k:  
 list.pop()  
 return [a - b \* math.log(reduce(lambda x, y: x \* (1 - y), first, 1)) for first in list]

## Алгоритм 6. Логнормальное распределение.

где a и b – параметры положения и масштаба, соответствующего нормального распределения.

Используя стандартные нормальные случайные числа Z, применяют формулу для получения случайных чисел, соответствующих логнормальному распределению.

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d ln -p1 1263 -p2 14

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2 – входные параметры в алгоритме преобразования.

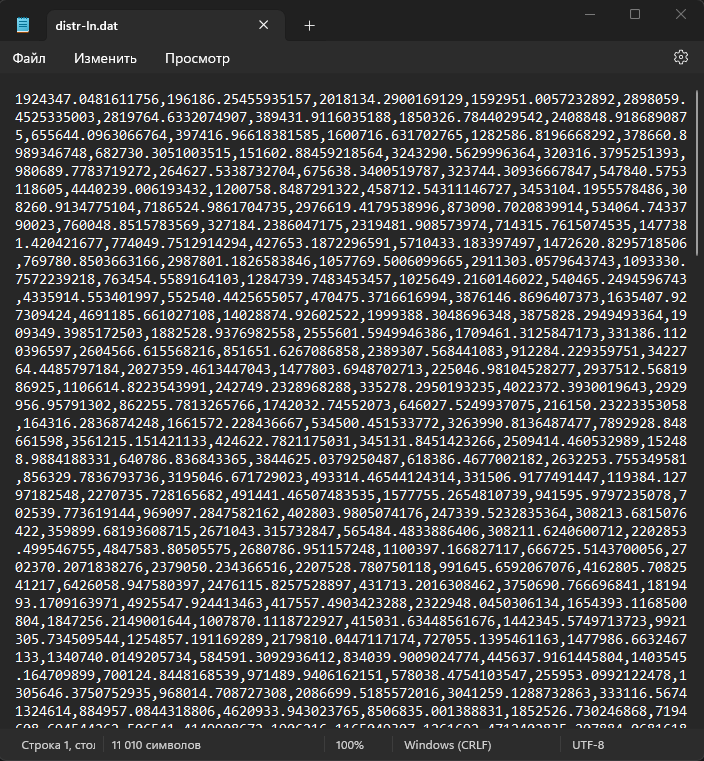
****

Рис. 15 – Результат преобразования.

**Исходный текст программы.**

def ln(a, b, n):  
 n = nr(0, 1, n)  
 return [a + math.exp(b - z) for z in n]

## Алгоритм 7. Логистическое распределение.

где *a* и *b* – параметры положения и масштаба соответственно.

Если стандартные равномерные случайные числа U генерированы методом, изложенным выше, то случайные числа, соответствующие логистическому распределению, получают по формуле

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d ls -p1 1287 -p2 1255

где параметр

-d – выбор алгоритма преобразования,

-p1, -p2 – входные параметры в алгоритме преобразования.

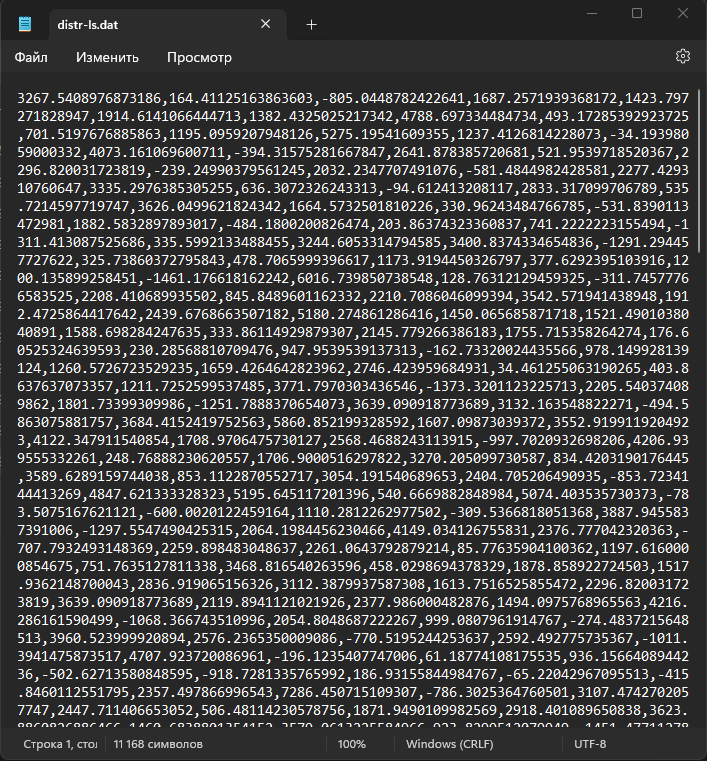
****

Рис. 16 – Результат преобразования.

**Исходный текст программы.**

def ls(a, b, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 return [a + b \* math.log(u / (1 - u)) for u in n]

## Алгоритм 8. Биномиальное распределение.

Если вероятность появления события при каждом испытании равна p, то вероятность того, что это событие произойдет y раз за n испытаний, определяют по формуле

где .

Вычисляют функцию распределения

Для получения случайного числа Y генерируют стандартное равномерное случайное число U. Случайное число Y является наименьшим значением y, для которого

**Параметры запуска программы.**

Для запуска программы необходимо выполнить следующую команду:

rnc.exe -d bi -p1 1271

где параметр

-d – выбор алгоритма преобразования,

-p1 – входные параметры в алгоритме преобразования.

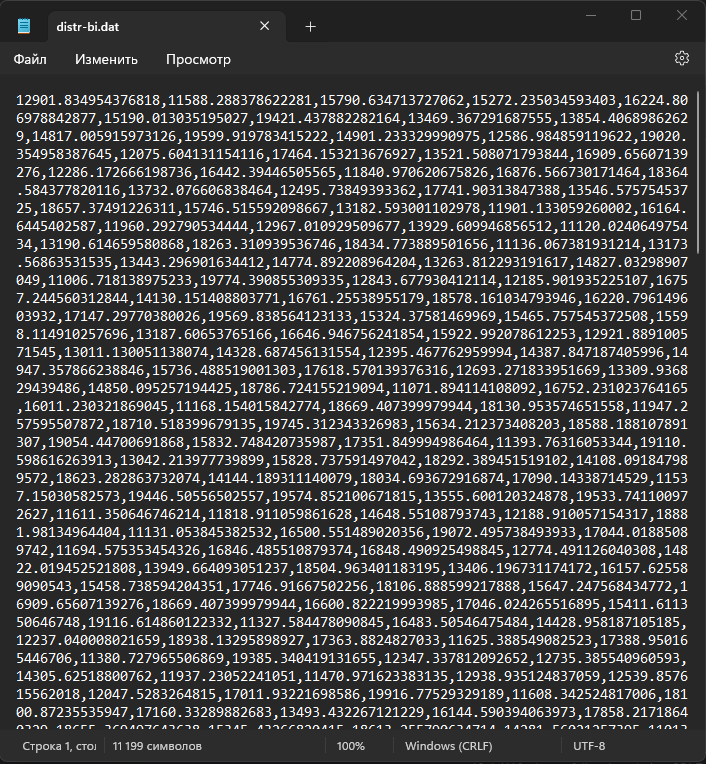
****

Рис. 17 – Результат преобразования.

**Исходный текст программы.**

def bi(a, num):  
 n = len(num)  
 m = max(num) + 1  
 num = [x / m for x in num]  
 result = []  
 for i in range(1, n):  
 temp = 0  
 y = num[i]  
 while temp < num[i]:  
 k = 0  
 while y > k:  
 temp += math.comb(n, k) \* pow(a, k) \* pow(1 - a, n - k)  
 k += 1  
 y += 1  
 result.append(y \* 10000)  
 return result

ПРИЛОЖЕНИЕ А

Листинг программы prng.exe

import argparse  
  
  
def progress\_bar(now, all):  
 if ((now + 1) / all \* 100) % 20 == 0:  
 print(f"\nГенерация: {(now + 1) / all \* 100}%")  
  
  
def lc(m, a, c, x0, n=10000):  
 res = [x0]  
 for i in range(n):  
 res.append((a \* res[i] + c) % m)  
 progress\_bar(i, n)  
  
 return res  
  
  
def add(m, li, ui, arr, n=10000):  
 res = []  
 tmp = arr.copy()  
 for i in range(n):  
 new\_elem = (tmp[li - 1] + tmp[ui - 1]) % m  
 res.append(new\_elem)  
 tmp.append(new\_elem)  
 tmp.pop(0)  
 progress\_bar(i, n)  
 return res  
  
  
def p5(p, q1, q2, q3, w, x0, n=10000):  
 res = []  
 x0 = int(x0, base=2)  
 for i in range(n):  
 new\_elem = 1  
 for \_ in range(w - 1):  
 new\_bit1 = (x0 >> p - q1) & 1  
 new\_bit2 = (x0 >> p - q2) & 1  
 new\_bit3 = (x0 >> p - q3) & 1  
 last\_bit = p & 1  
 xor = new\_bit1 ^ new\_bit2 ^ new\_bit3 ^ last\_bit  
 new\_elem = (new\_elem << 1) | xor  
 x0 = (x0 >> 1) | (xor << p - 1)  
 progress\_bar(i, n)  
 res.append(new\_elem)  
 return res  
  
  
def lfsr(vec\_of\_cf, register, n=10000):  
 res = []  
 len\_reg = len(register)  
 vec\_of\_cf = int(vec\_of\_cf, 2)  
 register = int(register, 2)  
 for i in range(n):  
 new\_bit = (register ^ ((register ^ vec\_of\_cf) >> 1)) & 1  
 register = (register >> 1) | (new\_bit << (len\_reg - 1))  
 res.append(register)  
 progress\_bar(i, n)  
 return res  
  
  
def nfsr(R1, R2, R3, w, x1, x2, x3, n=10000):  
 res = []  
 len\_R1 = len(R1)  
 len\_R2 = len(R2)  
 len\_R3 = len(R3)  
 for i in range(n):  
 new\_elem = 1  
 for \_ in range(w - 1):  
 xor\_R1 = (x1 ^ (x1 >> 1))  
 xor\_R2 = (x2 ^ (x2 >> 1))  
 xor\_R3 = (x3 ^ (x3 >> 1))  
 out = ((xor\_R1 ^ xor\_R2) + (xor\_R2 ^ xor\_R3) + xor\_R3) & 1  
 x1 = (x1 >> 1) | (out << len\_R1 - 1)  
 x2 = (x2 >> 1) | (out << len\_R2 - 1)  
 x3 = (x3 >> 1) | (out << len\_R3 - 1)  
 new\_elem = (new\_elem << 1) | out  
 res.append(new\_elem)  
 progress\_bar(i, n)  
 return res  
  
  
def mt(x0, m, n=10000):  
 res = []  
 w = 32  
 start = [0] \* m  
 start[0] = x0  
 for i in range(1, m):  
 start[i] = (start[i - 1] ^ (start[i - 1] >> 30) + i) & ((1 << w) - 1)  
 for i in range(n):  
 index = m  
 u, s, t, l, w = 11, 7, 15, 18, 32  
 b = 0x9D2C5680  
 c = 0xEFC60000  
 if index >= 624:  
 p, r, q = 624, 31, 397  
 a = 0x9908B0DF  
 for j in range(p):  
 rand\_num = (start[j] >> r) + (start[(j + 1) % p] & ((1 << r) - 1))  
 start[j] = start[(j + q) % p] ^ (rand\_num >> 1)  
 if rand\_num % 2 != 0:  
 start[j] ^= a  
 index = 0  
 rand\_num = start[index]  
 rand\_num ^= (rand\_num >> u)  
 rand\_num ^= ((rand\_num << s) & b)  
 rand\_num ^= ((rand\_num << t) & c)  
 rand\_num ^= (rand\_num >> l)  
 index += 1  
 rand\_num = rand\_num & ((1 << w) - 1)  
 res.append(rand\_num)  
 progress\_bar(i, n)  
 return res  
  
  
def rc4(vec, n=10000):  
 vec\_size = len(vec)  
 S = list(range(4096))  
 j = 0  
 for i in range(4096):  
 j = (j + S[i] + vec[i % vec\_size]) % 4096  
 S[i], S[j] = S[j], S[i]  
  
 f = 0  
 s = 0  
 res = []  
 for i in range(n):  
 f = (f + 1) % 4096  
 s = (s + S[f]) % 4096  
 S[f], S[s] = S[s], S[f]  
 new\_elem = S[(S[f] + S[s]) % 4096]  
 res.append(new\_elem)  
 progress\_bar(i, n)  
 return res  
  
  
def rsa(n, e, w, x, num=10000):  
 res = []  
 for i in range(num):  
 new\_elem = 0  
 for \_ in range(w):  
 x = pow(x, e, n)  
 new\_elem = (new\_elem << 1) | (x & 1)  
 res.append(new\_elem)  
 progress\_bar(i, num)  
 return res  
  
  
def bbs(x, w, num=10000):  
 p = 127  
 q = 131  
 n = p \* q  
 res = []  
 for i in range(num):  
 new\_elem = 0  
 for \_ in range(w):  
 x = pow(x, 2, n)  
 new\_elem = (new\_elem << 1) | (x & 1)  
 res.append(new\_elem)  
 progress\_bar(i, num)  
 return res  
  
  
def write\_to\_file(data, filepath="rnd.dat"):  
 with open(filepath, "w", encoding="UTF-8") as f:  
 f.write(data)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 parser = argparse.ArgumentParser(  
 prog="prng.exe",  
 description="Генерация последовательности псевдослучайных чисел")  
 help1 = """ - метод генерации:\n   
 • lc – линейный конгруэнтный метод;\n  
 • add – аддитивный метод;\n  
 • 5p – пятипараметрический метод;\n  
 • lfsr – регистр сдвига с обратной связью (РСЛОС);\n  
 • nfsr – нелинейная комбинация РСЛОС;\n  
 • mt – вихрь Мерсенна;\n  
 • rc4 – RC4;\n  
 • rsa – ГПСЧ на основе RSA;\n  
 • bbs – алгоритм Блюма-Блюма-Шуба;\n  
 """  
 help2 = "- количество генерируемых чисел. По умолчанию - 10000."  
 help3 = "- имя файла, в который будут записываться данные.\n По умолчанию - rnd.dat."  
 help4 = "Перечисление параметров для выбранного генератора"  
 parser.add\_argument("-g", help=help1, required=True,  
 choices=["lc", "add", "5p", "lfsr", "nfsr", "mt", "rc4", "rsa", "bbs"], nargs=1)  
 parser.add\_argument("-i", nargs="\*", help=help4, )  
 parser.add\_argument("-n", nargs=1, type=int, default=[10000], help=help2)  
 parser.add\_argument("-f", nargs=1, default=["rnd.dat"], help=help3, )  
 args = parser.parse\_args()  
  
  
 def check\_params(count\_params, parametrs):  
 if len(parametrs) != count\_params:  
 raise Exception("Передано неверное количество аргументов")  
 for par in parametrs:  
 if not par.isdigit():  
 raise Exception("Переданы неподходящие параметры")  
 return True  
  
  
 type = args.g[0]  
 parametrs = args.i  
 path\_file = args.f[0]  
 count = args.n[0]  
 try:  
 match type:  
 case 'lc':  
 if check\_params(4, parametrs):  
 m = int(parametrs[0])  
 a = int(parametrs[1])  
 c = int(parametrs[1])  
 x0 = int(parametrs[2])  
 result = lc(m, a, c, x0, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'add':  
 if len(parametrs) < 4:  
 raise Exception("Неверное количество аргументов")  
 for par in parametrs:  
 if not par.isdigit():  
 raise Exception("Неподходящие параметры")  
 m = int(parametrs[0])  
 li = int(parametrs[1])  
 ui = int(parametrs[2])  
 arr = list(map(int, parametrs[3:]))  
 result = add(m, li, ui, arr, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case '5p':  
 if check\_params(6, parametrs):  
 p = int(parametrs[0])  
 q1 = int(parametrs[1])  
 q2 = int(parametrs[2])  
 q3 = int(parametrs[3])  
 w = int(parametrs[4])  
 x0 = parametrs[5]  
 result = p5(p, q1, q2, q3, w, x0, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'lfsr':  
 if check\_params(2, parametrs):  
 vec\_of\_cf = parametrs[0]  
 registr = parametrs[1]  
 result = lfsr(vec\_of\_cf, registr, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'nfsr':  
 if check\_params(7, parametrs):  
 R1 = parametrs[0]  
 R2 = parametrs[1]  
 R3 = parametrs[2]  
 w = int(parametrs[3])  
 x1 = int(parametrs[4])  
 x2 = int(parametrs[5])  
 x3 = int(parametrs[6])  
 result = nfsr(R1, R2, R3, w, x1, x2, x3, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'mt':  
 if check\_params(2, parametrs):  
 x0 = int(parametrs[1])  
 m = int(parametrs[0])  
 result = mt(x0, m, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'rc4':  
 if check\_params(256, parametrs):  
 vec = list(map(int, parametrs))  
 result = rc4(vec, n=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'rsa':  
 if 4 == len(parametrs):  
 n = int(parametrs[0])  
 e = int(parametrs[1])  
 w = int(parametrs[2])  
 x = int(float(parametrs[3]))  
 result = rsa(n, e, w, x, num=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 case 'bbs':  
 if check\_params(2, parametrs):  
 x = int(parametrs[0])  
 w = int(parametrs[1])  
 result = bbs(x, w, num=count)  
 data = ",".join(map(str, result))  
 write\_to\_file(data, filepath=path\_file)  
 except Exception as err:  
 print("Произошла ошибка!")  
 print(" • " + str(err))

ПРИЛОЖЕНИЕ Б

Листинг программы rnc.exe

import math  
import argparse  
from functools import reduce  
  
  
def st(a, b, n):  
 m = max(n) + 1  
 return [x / m \* b + a for x in n]  
  
  
def tr(a, b, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 return [a + b \* (n[i] + n[i + 1] - 1) for i in range(len(n) - 1)]  
  
  
def ex(a, b, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 return [-b \* math.log(u) + a for u in n]  
  
  
def nr(a, b, n):  
 result = []  
 m = max(n) + 1  
 n = [x / m for x in n]  
 for i in range(0, len(n) - 1, 2):  
 z1 = a + b \* math.sqrt(-2 \* math.log(1 - n[i])) \* math.cos(2 \* math.pi \* n[i + 1])  
 z2 = a + b \* math.sqrt(-2 \* math.log(1 - n[i])) \* math.sin(2 \* math.pi \* n[i + 1])  
 result.append(z1)  
 result.append(z2)  
 return result  
  
  
def gm(a, b, k, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 list = [n[i: i + k] for i in range(0, len(n), k)]  
 if len(list[-1]) != k:  
 list.pop()  
 return [a - b \* math.log(reduce(lambda x, y: x \* (1 - y), first, 1)) for first in list]  
  
  
def ln(a, b, n):  
 n = nr(0, 1, n)  
 return [a + math.exp(b - z) for z in n]  
  
  
def ls(a, b, n):  
 m = max(n) + 1  
 n = [x / m for x in n]  
 return [a + b \* math.log(u / (1 - u)) for u in n]  
  
  
def bi(a, num):  
 n = len(num)  
 m = max(num) + 1  
 num = [x / m for x in num]  
 result = []  
 for i in range(1, n):  
 temp = 0  
 y = num[i]  
 while temp < num[i]:  
 k = 0  
 while y > k:  
 temp += math.comb(n, k) \* pow(a, k) \* pow(1 - a, n - k)  
 k += 1  
 y += 1  
 result.append(y \* 10000)  
 return result  
  
  
def write\_to\_file(data, filepath):  
 with open(filepath, "w", encoding="UTF-8") as f:  
 f.write(data)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 parser = argparse.ArgumentParser(  
 prog="rnc.exe",  
 description="Генерация последовательности псевдослучайных чисел по выбранному распределению")  
 help1 = """Указывает тип распределения:   
 st – стандартное равномерное с заданным интервалом,  
 tr – треугольное распределение,  
 ex – общее экспоненциальное распределение,   
 nr – нормальное распределение,  
 gm – гамма распределение,  
 ln – логнормальное распределение,   
 ls – логистическое распределение,  
 bi – биномиальное распределение"""  
 help2 = "Указывает путь до файла из которого берется входная последовательность чисел"  
 help3 = "1-й параметр, необходимый, для генерации ПСЧ заданного распределения"  
 help4 = "2-й параметр, необходимый, для генерации ПСЧ заданного распределения (в случае bi указывает на количество разрядов в генерируемом чиле до 10^6)"  
 help5 = "3-й параметр, необходимый, для генерации ПСЧ гамма-распределением."  
 parser.add\_argument("-d", help=help1, required=True, choices=["st", "tr", "ex", "nr", "gm", "ln", "ls", "bi"],  
 nargs=1)  
 parser.add\_argument("-f", nargs=1, default=["rnd.txt"], help=help2)  
 parser.add\_argument("-p1", nargs=1, type=int, required=True, help=help3)  
 parser.add\_argument("-p2", nargs=1, type=int, help=help4, default=[None])  
 parser.add\_argument("-p3", nargs=1, type=int, help=help5, default=[None])  
 args = parser.parse\_args()  
  
  
 def check\_params(count\_params, parametrs):  
 if len(parametrs) != count\_params:  
 raise Exception("Передано неверное количество аргументов")  
 for par in parametrs:  
 if not par.isdigit():  
 raise Exception("Переданы неподходящие параметры")  
 return True  
  
  
 type = args.d[0]  
 path\_file = args.f[0]  
 p1 = args.p1[0]  
 p2 = args.p2[0]  
 p3 = args.p3[0]  
 try:  
 with open(path\_file, "r") as f:  
 line = f.readline()  
 list = list(map(int, line.split(",")))  
 match type:  
 case 'st':  
 a = p1  
 b = p2  
 file = "distr-st.dat"  
 numbers = st(a, b, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'tr':  
 a = p1  
 b = p2  
 file = "distr-tr.dat"  
 numbers = tr(a, b, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'ex':  
 a = p1  
 b = p2  
 file = "distr-ex.dat"  
 numbers = ex(a, b, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'nr':  
 a = p1  
 b = p2  
 file = "distr-nr.dat"  
 numbers = nr(a, b, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'gm':  
 a = p1  
 b = p2  
 k = p3  
 file = "distr-gm.dat"  
 numbers = gm(a, b, k, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'ln':  
 a = p1  
 b = p2  
 file = "distr-ln.dat"  
 numbers = ln(a, b, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'ls':  
 a = p1  
 b = p2  
 file = "distr-ls.dat"  
 numbers = ls(a, b, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 case 'bi':  
 a = p1  
 file = "distr-bi.dat"  
 numbers = bi(a, list)  
 data = ",".join(map(str, numbers))  
 write\_to\_file(data, filepath=file)  
 except Exception as err:  
 print("В процессе генерации произошла ошибка!")  
 print(" • " + str(err))