МИНОБРНАУКИ РОССИИ
ФГБОУ ВО «СГУ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

ТЕОРИЯ ПСЕВДОСЛУЧАЙНЫХ ГЕНЕРАТОРОВ

ЛАБОРАТОРНАЯ РАБОТА

студента 4 курса 431 группы
специальности 100501 — Компьютерная безопасность
факультета КНиИТ
Ивановой Ксении

Проверил
доцент _____ И. И. Слеповичев

Саратов 2024

СОДЕРЖАНИЕ

1 Генераторы псевдослучайных чисел 3
1.1 Линейный конгруэнтный метод 3
1.2 Аддитивный метод 3
1.3 Пятипараметрический метод 4
1.4 Регистр сдвига с обратной связью (РСЛОС) 5
1.5 Нелинейная комбинация РСЛОС 5
1.6 Вихрь Мерсенна 6
1.7 RC4 8
1.8 ГПСЧ на основе RSA 9
1.9 Алгоритм Блюм-Блюма-Шуба 9
2 Преобразование ПСЧ к заданному распределению 11
2.1 Стандартное равномерное с заданным интервалом 11
2.2 Треугольное распределение 11
2.3 Общее экспоненциальное распределение 11
2.4 Нормальное распределение 12
2.5 Гамма распределение 12
2.6 Логнормальное распределение 13
2.7 Логистическое распределение 13
2.8 Биномиальное распределение 13
Приложение А Файлы заданий 15
1.1 Аргументы для генерации 15
1.2 Реализация генераторов 15
1.3 Реализация распределений 18
1.4 Дополнительные функции 19
1.5 main задания 1 19
1.6 main задания 2 20

1 Генераторы псевдослучайных чисел

1.1 Линейный конгруэнтный метод

Линейной конгруэнтной последовательностью (ЛКП) называется последовательность ПСЧ, получаемая по формуле:

$$X_{n+1} = (aX_n + c) \mod m,\ n \geq 1, \tag{1}$$

Параметры запуска программы: Модуль, множитель, приращение, начальное значение.

Пример запуска:

```
python3 main.py -g lc -i 1024,1664525,1013904223,1 -n 10000 -f lc.dat
```

Алгоритм:

```
def lc(args=args_for_gen["lc"]):
  a = int(args[0])
  c = int(args[1])
  mod = int(args[2])
  x_n = int(args[3])
  while(True):
      x_n = (x_n * a + c) % mod
      yield x_n
```

1.2 Аддитивный метод

Последовательность определяется следующим образом:

$$X_n = (X_{n-k} + X_{n-j}) \mod m,\ j > k \geq 1. \tag{2}$$

Параметры запуска программы: Модуль, младший индекс, старший индекс, последовательность начальных значений.

Пример запуска:

```
python3 main.py -g add -i 30000,24,55,79,134,213,347,560,907,1467,2374,3...,
17506 -n 10000 -f add.dat
```

Алгоритм:

```
def add(args=args_for_gen["add"]):
  mod, k, j = int(args[0]), int(args[1]), int(args[2])
  x = list(map(int, args[3:]))
```

```
  i = 55
  while(True):
      val = (x[i-k] + x[i-j]) % mod
      x.append(val)
      i += 1
      yield val
```

1.3 Пятипараметрический метод

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:

$$X_{n+p} = X_{n+q_1} + X_{n+q_2} + X_{n+q_3} + X_n,\ n = 1, 2, 3, \ldots \tag{3}$$

Параметры запуска программы: p, q1, q2, q3, w, начальное значение.
Пример запуска:

```
  python3 main.py -g 5p -i 89,7,13,24,10,234122131 -n 10000 -f 5p.dat
```

Алгоритм:

```
def fp(args=args_for_gen["fp"]):
  p, w, x = int(args[0]), int(args[4]), int(args[5])
  mask_w, mask_p = 0, 0
  list_par = list(map(int, args[1:4]))
  list_par.append(0)

  for i in range(w):
      mask_w = set_bit(mask_w, i, 1)
  for i in range(p):
      mask_p = set_bit(mask_p, i, 1)

  while(True):
      cur_bit = 0
      for i in list_par:
          cur_bit ^= get_bit(x, i)
      x = shift(x, p)
      x = set_bit(x, 0, cur_bit)
      x = x & mask_p
      yield x & mask_w
```

1.4 Регистр сдвига с обратной связью (РСЛОС)

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.

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

1. Содержимое ячейки $p-1$ формирует очередной бит ПСП битов.
2. Содержимое ячейки 0 определяется значением функции обратной связи, являющейся линейной булевой функцией с коэффициентами $a_1, \ldots, a_{p-1}$.
3. Содержимое каждого i-го бита перемещается в $(i+1)$-й, $0 \leq i < p-1$.
4. В ячейку 0 записывается новое содержимое, вычисленное на шаге 2.

Параметры запуска программы: двоичное представление вектора коэффициентов, начальное значение регистра

Пример запуска:

```
python3 main.py -g lfsr -i 10011011010011010,000101000111 -n 10000 -f lfsr.dat
```

Алгоритм:

```
def lfsr(args=args_for_gen["lfsr"]):
  s, x, y = len(args[1]), int(args[0], 2), int(args[1], 2)
  while(True):
      current = 0
      for i in range(s):
          current ^= get_bit(x, i) * get_bit(y, i)
      y = shift(y, s)
      y = set_bit(y, 0, current)
      yield y
```

1.5 Нелинейная комбинация РСЛОС

Последовательность получается из нелинейной комбинации трёх РСЛОС следующим образом:

$$f(x_1, x_2, x_3) = x_1 x_2 \oplus x_2 x_3 \oplus x_3 \tag{4}$$

Параметры запуска программы: двоичное представление векторов коэффициентов для R1, R2, R3, w, x1, x2, x3. w – длина слова, x1, x2, x3 – десятичное представление начальных состояний регистров R1, R2, R3.

Пример запуска:

```
python3 main.py -g nfsr -i 00000001010101,01011100000111101,010101001100000,
0001001,10000010010,000000001,1001 -f nfsr.dat -n 10000
```

Алгоритм:

```
def nfsr(args=args_for_gen["nfsr"]):
  r1 = lfsr([args[0], args[3]])
  r2 = lfsr([args[1], args[4]])
  r3 = lfsr([args[2], args[5]])
  w1 = 0
  for i in range(int(args[6])):
      w1 = set_bit(w1, i, 1)
  while True:
      a, b, c = next(r1), next(r2), next(r3)
      yield (a ^ b + b ^ c + c) & w1
```

1.6 Вихрь Мерсенна

Метод основан на свойствах простых чисел Мерсенна и обладает рядом достоинств относительно многих других ГПСЧ. «Вихрь» – это преобразование, которое обеспечивает равномерное распределение ПСЧ.

Числом Мерсенна называется натуральное число, определяемое формулой

$$M_n = 2^n - 1. \tag{5}$$

Алгоритм состоит из следующих шагов:

1. Инициализируются значения u, h, a по формуле:
   $u := (1, 0, \ldots, 0)$ - всего $w - r$ бит,
   $h := (0, 1, \ldots, 1)$ - всего r бит,
   $a := (a_{w-1}, a_{w-2}, \ldots, a_0)$ - последняя строка матрицы .
2. $X_0, X_1, \ldots, X_{p-1}$ заполняются начальными значениями.
3. Вычисляется $Y := (y_0, y_1, \ldots, y_{w-1}) := (X_n^r | X_{n+1}^l)$.
4. Вычисляется новое значение X_i:
   $X_n := X_{(n+q) \bmod p} \oplus (Y >> 1) \oplus a$, если младший бит $y_0 = 1$;
   $X_n := X_{(n+q) \bmod p} \oplus (Y >> 1) \oplus 0$, если младший бит $y_0 = 0$;
5. Вычисляется $X_i T$:
   $Y := X_n$,
   $Y := Y \oplus (Y >> u)$,

$Y := Y \oplus ((Y << s) \cdot b)$,

$Y := Y \oplus ((Y << t) \cdot c)$,

$Z := Y \oplus (Y >> l)$.

Z подается на выход, как результат.

6. $n := (n + 1) \text{mod } p$. Переход на шаг 3.

Параметры запуска программы: модуль, начальное значение x.
Пример запуска:

```
python3 main.py -g mt -i 1000,1234 -n 10000 -f mt.dat
```

Алгоритм:

```
def mt(args=args_for_gen["mt"]):
  p, w, r, q, a, u = 624, 32, 31, 397, 2567483615, 11
  s, t, l, b, c = 7, 15, 18, 2636928640, 4022730752
  mod = int(args[0])
  MT = [int(args[1])]
  lower_mask = (1 << r) - 1
  upper_mask = (~lower_mask * -1) & w1
  w1 = 0
  ind = p

  for i in range(w):
      w1 = set_bit(w1, i, 1)

  for i in range(1, p):
      MT.append((MT[i - 1] ^ (MT[i - 1] >> 30)) + i)

  while(True):
      if (ind >= p):
          for i in range(p):
              x = (MT[i] & upper_mask) + (MT[(i + 1) % p] & lower_mask)
              xA = x >> 1
              if (x & 1):
                  xA ^= a
              MT[i] = MT[(i + q) % p] ^ xA
          ind = 0
      y = MT[ind]
      ind += 1
      y ^= (y >> u)
      y ^= (y << s) & b
```

```
        y ^= (y << t) & c
        y ^= (y >> l)
        yield y % mod
```

1.7 RC4

Алгоритм состоит из следующих шагов:

1. Инициализация S_i.
2. $i = 0, j = 0$.
3. Итерация алгоритма:
   а) $ans = 0$;
   б) $i = (i + 1) \mod 256$;
   в) $j = (j + S_i) \mod 256$;
   г) $Swap(S_i, S_j)$;
   д) $t = (S_i + S_j) \mod 256$;
   е) $K = S_t$.
   ж) $ans = ans << 1 + K\&1$
   з) Если количество бит $< w$, перейти к шагу б.

Параметры запуска программы: 256 начальных значений.
Пример запуска:

```
python3 main.py -g rc4 -i 213,968,838,64,355,214,212,...,659,180,10 -f rc.dat
-n 10000
```

Алгоритм:

```
def rc4(args=args_for_gen["rc4"]):
    k = list(map(int, args[:-2]))
    w = int(args[-1])
    l = len(k)
    s = [i for i in range(l)]
    i, j = 0, 0
    for i in range(l):
        j = (j + s[i] + k[i]) % l
        s[i], s[j] = s[j], s[i]
    while(True):
        num = 0
        for t in range(w):
            i = (i + 1) % l
```

```
            j = (j + s[i]) % l
            s[i], s[j] = s[j], s[i]
            num = set_bit(num, t, s[(s[i] + s[j]) % l] & 1)
        yield num
```

1.8 ГПСЧ на основе RSA

Алгоритм состоит из следующих шагов:

1. Инициализация чисел: $n = pq$, где p и q простые числа, числа e: $1 < e < f$, НОД(e, f) = 1, $f = (p-1)(q-1)$ и числа x_0 из интервала $[1, n-1]$.
2. `For i = 1 to w do`
   а) $x_i \leftarrow x_{i-1}^e \mod n$.
   б) $z_i \leftarrow$ последний значащий бит x_i
3. Вернуть $z_1, \ldots, z_w$.

Параметры запуска программы: модуль n, число e, w, начальное значение x.

Пример запуска:

```
python3 main.py -g rsa -i 10967,571,77,10 -n 10000 -f rsa.dat
```

Алгоритм:

```
def rsa(args=args_for_gen["rsa"]):
  n = int(args[0])
  e = int(args[1])
  x = int(args[2])
  w = int(args[3])
  while(True):
      z = 0
      for i in range(w):
          x = x ** e % n
          z = set_bit(z, w - i - 1, x & 1)
      yield z
```

1.9 Алгоритм Блюм-Блюма-Шуба

Алгоритм состоит из следующих шагов:

1. Инициализация числа: $n = 127 \cdot 131 = 16637$.
2. Вычислим $x_0 = x^2 \mod n$, которое будет начальным вектором.
3. `For i = 1 to w do`
   а) $x_i \leftarrow x_{i-1}^2 \mod n$.

б) $z_i \leftarrow$ последний значащий бит x_i

4. Вернуть $z_1, \ldots, z_w$.

Параметры запуска программы: начальное значение x (взаимно простое с n).

Пример запуска:

```
python3 main.py -g bbs -i 15621,10 -n 10000 -f bbs.dat
```

Алгоритм:

```
def bbs(args=args_for_gen["bbs"]):
  p, q = 127, 131
  n = p * q
  x, w = int(args[0]), int(args[1])
  while(True):
      z = 0
      for i in range(w):
          x = x * x % n
          z = set_bit(z, w - i - 1, x & 1)
      yield z
```

2 Преобразование ПСЧ к заданному распределению

2.1 Стандартное равномерное с заданным интервалом

Формула:

$$Y = bU + a$$

Пример запуска:

```
python3 main1.py -d st -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def st(x, a, b, m):
  return a + U(x, m) * b
```

2.2 Треугольное распределение

Формула:

$$Y = a + b(U_1 + U_2 - 1)$$

Пример запуска:

```
python3 main1.py -d tr -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def tr(x, a, b, m):
  y = []
  for i in range(0, len(x)-1, 2):
      y.append(a + b * (U(x[i], m) + U(x[i+1], m) - 1))
  return np.array(y)
```

2.3 Общее экспоненциальное распределение

Формула:

$$Y = -b\ln(U) + a$$

Пример запуска:

```
python3 main1.py -d ex -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def ex(x, a, b, m):
  return a - b * np.log(U(x, m))
```

2.4 Нормальное распределение

Формула:

$$Z_1 = \mu\sigma\sqrt{-2\ln(1-U_1)}\cos(2\pi U_2)$$

$$Z_2 = \mu\sigma\sqrt{-2\ln(1-U_1)}\sin(2\pi U_2)$$

Пример запуска:

```
python3 main1.py -d nr -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def nr(x, a, b, m):
  y = []
  for i in range(0, len(x)-1, 2):
      base = a + b * np.sqrt(-2 * np.log(1 - U(x[i], m)))
      trig_arg = 2 * np.pi * U(x[i+1], m)
      y.append(base * np.cos(trig_arg), base * np.sin(trig_arg))
  return np.array(y)
```

2.5 Гамма распределение

Алгоритм для $c = k$ (k – целое число)

$$Y = a - b\ln\{(1-U_1)\ldots(1-U_k)\}$$

Пример запуска:

```
python3 main1.py -d gm -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def gm(x, a, b, c, m):
y = []
u = U(x, m)
if type(c) == type(1):
    for i in range(0, len(u), c):
        y.append(a - b  * np.log(np.prod(1 - u[i:i+c])))
else:
    k = int(c - 0.5)
    for i in range(0, len(u), 2 * k + 2):
        z1, z2 = nr([x[i], x[i+1]], 0, 1, m)
```

```
            y.append(a + b * (z1 ** 2 / 2 - np.log(np.prod(1 - u[i+2:i+k+2]))))
            y.append(a + b * (z2 ** 2 / 2 - np.log(np.prod(1 - u[i+k+2:i+2*k+2]))))
    return np.array(y)
```

2.6 Логнормальное распределение

Формула:

$$Y = a + \exp(b - Z)$$

Пример запуска:

```
python3 main1.py -d ln -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def ln(x, a, b, m):
  return a + np.exp(b - nr(x, 0, 1, m))
```

2.7 Логистическое распределение

Формула:

$$Y = a + b \ln(\frac{U}{1 - U})$$

Пример запуска:

```
python3 main1.py -d ls -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def ls(x, a, b, m):
  u = U(x, m)
  return a + b * np.log(u / (1 - u))
```

2.8 Биномиальное распределение

```
  y = binominal(x, a, b, m):
    u = U(x)
    s = 0
    k = 0
    loopstart:
      s = s + C(k, b) + a^k * (1 - a)^(b - k)
      if s > u:
        y = k
        Завершить
      if k < b - 1:
        k = k + 1
        move to loopstart
      y = b
```

Пример запуска:

```
python3 main1.py -d bi -f 5p.dat -p1 1024 -p2 1664525 -p3 876
```

Алгоритм:

```
def bi(x, a, b, m):
  y = []
  u = U(x, m)
  for i in u:
      s, k = 0, 0
      while(True):
          s += (factor(b) / (factor(k) * factor(b - k))
                          * (a ** k) * ((1 - a) ** (b - k)))
          if s > i:
              y.append(k)
              break
          if k < b - 1:
              k += 1
              continue
          y.append(b)
  return np.array(y)
```

ПРИЛОЖЕНИЕ А

Файлы заданий

1.1 Аргументы для генерации

```
args_for_gen = {
    "lc": ["421", "17117","81000", "1"],
    "add": [1024, 24, 55, 23, 517, 897, 480, 376, 94, 23, 517, 897, 480, 376, 94, 23, 517, 897, 480, 376, 94, 23, 517, 897,
     480, 376, 94, 23, 517, 897, 480, 376, 94, 23, 517, 897, 480, 376, 94, 23, 517, 897, 480, 376, 94, 23, 517, 897,
     480, 376, 94, 23, 517, 897, 480, 376, 94, 23],
    "lfsr":["100000001001001", "0011000000"],
    "fp": [87, 20, 40, 69, 9, 771635173876655143518901931837654261831 1247],
    "nfsr": ["100000001001001", "0011000000", "101011001001001", "11010011000000", "101100010001001", "10110000111"],
    "rc4": [213,968,838,64,355,214,212,36,695,139,897,518,656,956,810,510,985,105,670,8,907,951,685,989,222,931,169,286,
            289,556,731,902,688,701,771,533,990,630,708,884,255,683,25,214,792,348,34,758,9,781,946,580,615,955,585,5,
            886,563,81,38,809,444,619,222,544,53,635,621,630,251,497,257,2,467,897,790,728,676,722,838,465,781,10,828,
            903,235,857,841,146,719,681,678,961,652,491,38,256,909,251,21,110,811,273,25,642,286,489,478,184,812,770,
            846,241,141,266,500,375,827,633,761,154,663,461,206,529,212,667,342,360,165,523,749,582,803,553,345,
            786,990,361,702,256,380,234,238,73,965,266,300,847,755,969,681,146,843,125,306,845,752,879,458,788,833,
            727,817,122,239,765,877,827,327,733,658,644,880,150,474,493,689,670,368,611,263,113,417,834,103,725,754,117,
            824,623,338,540,337,879,521,183,370,808,120,571,871,301,210,796,744,398,106,845,745,842,876,399,27,105,601,802,
            831,53,266,157,352,175,303,505,484,994,425,292,729,654,584,860,420,412,49,281,417,703,400,48,404,772,389,733,152
            271,585,404,333,381,696,928,609,659,180,9],
    "rsa": ["12709189", "53", "245", "10"],
    "bbs": ["15621", "10"],
    "mt" : ["1024","17461461673"],
}

met = "\n1. Выберите метод:\n\tlc-линейный конгруэнтный;\n\tadd-адитивный;\n\t5p-пятипараметрический;\n\tlfsr-регистр сдвига
por = "\n2. Задайте порядок:\n\tlc-модуль, множитель, приращение, начальное значение\n\tadd-модуль, младший индекс, старший
dist = "\n1. Выберите распределение:\n\tst - стандартное равномерное с заданным интервалом;\n\ttr - треугольное распределени
```

1.2 Реализация генераторов

```
from utils import set_bit, get_bit, shift
from args import args_for_gen


def lc(args=args_for_gen["lc"]):
    a = int(args[0])
    c = int(args[1])
    mod = int(args[2])
    x_n = int(args[3])
    while(True):
        x_n = (x_n * a + c) % mod
        yield x_n


def add(args=args_for_gen["add"]):
    mod, k, j = int(args[0]), int(args[1]), int(args[2])
    x = list(map(int, args[3:]))
    i = 55
    while(True):
        val = (x[i-k] + x[i-j]) % mod
        x.append(val)
        i += 1
        yield val



def fp(args=args_for_gen["fp"]):
```

```
    p, w, x = int(args[0]), int(args[4]), int(args[5])
    mask_w, mask_p = 0, 0
    list_par = list(map(int, args[1:4]))
    list_par.append(0)

    for i in range(w):
        mask_w = set_bit(mask_w, i, 1)
    for i in range(p):
        mask_p = set_bit(mask_p, i, 1)

    while(True):
        cur_bit = 0
        for i in list_par:
            cur_bit ^= get_bit(x, i)
        x = shift(x, p)
        x = set_bit(x, 0, cur_bit)
        x = x & mask_p
        yield x & mask_w



def lfsr(args=args_for_gen["lfsr"]):
    s, x, y = len(args[1]), int(args[0], 2), int(args[1], 2)
    while(True):
        current = 0
        for i in range(s):
            current ^= get_bit(x, i) * get_bit(y, i)
        y = shift(y, s)
        y = set_bit(y, 0, current)
        yield y



def nfsr(args=args_for_gen["nfsr"]):
    r1 = lfsr([args[0], args[3]])
    r2 = lfsr([args[1], args[4]])
    r3 = lfsr([args[2], args[5]])
    w1 = 0
    for i in range(int(args[6])):
        w1 = set_bit(w1, i, 1)
    while True:
        a = next(r1)
        b = next(r2)
        c = next(r3)
        yield (a ^ b + b ^ c + c) & w1

def rc4(args=args_for_gen["rc4"]):
    k = list(map(int, args[:-2]))
    w = int(args[-1])
    l = len(k)
    s = [i for i in range(l)]
    j = 0
    for i in range(l):
        j = (j + s[i] + k[i]) % l
        s[i], s[j] = s[j], s[i]
    i = 0
    while(True):
        num = 0
        for t in range(w):
            i = (i + 1) % l
            j = (j + s[i]) % l
```

```
            s[i], s[j] = s[j], s[i]
            num = set_bit(num, t, s[(s[i] + s[j]) % l] & 1)
        yield num

def rsa(args=args_for_gen["rsa"]):
    n = int(args[0])
    e = int(args[1])
    x = int(args[2])
    w = int(args[3])
    while(True):
        z = 0
        for i in range(w):
            x = x ** e % n
            z = set_bit(z, w - i - 1, x & 1)
        yield z

def bbs(args=args_for_gen["bbs"]):
    p, q = 127, 131
    n = p * q
    x, w = int(args[0]), int(args[1])
    while(True):
        z = 0
        for i in range(w):
            x = x * x % n
            z = set_bit(z, w - i - 1, x & 1)
        yield z

def mt(args=args_for_gen["mt"]):
    p, w, r, q, a, u, s, t, l, b, c = 624, 32, 31, 397, 2567483615, 11, 7, 15, 18, 2636928640, 4022730752
    mod = int(args[0])
    MT = [int(args[1])]
    lower_mask = (1 << r) - 1
    upper_mask = (~lower_mask * -1) & w1
    w1 = 0
    ind = p

    for i in range(w):
        w1 = set_bit(w1, i, 1)

    for i in range(1, p):
        MT.append((MT[i - 1] ^ (MT[i - 1] >> 30)) + i)

    while(True):
        if (ind >= p):
            for i in range(p):
                x = (MT[i] & upper_mask) + (MT[(i + 1) % p] & lower_mask)
                xA = x >> 1
                if (x & 1):
                    xA ^= a
                MT[i] = MT[(i + q) % p] ^ xA
            ind = 0
        y = MT[ind]
        ind += 1
        y ^= (y >> u)
        y ^= (y << s) & b
        y ^= (y << t) & c
        y ^= (y >> l)
        yield y % mod
```

1.3 Реализация распределений

```
import numpy as np
from utils import factor
def U(x, m):
    return x / m

def st(x, a, b, m):
    return a + U(x, m) * b

def tr(x, a, b, m):
    y = []
    for i in range(0, len(x)-1, 2):
        y.append(a + b * (U(x[i], m) + U(x[i+1], m) - 1))
    return np.array(y)

def ex(x, a, b, m):
    return a - b * np.log(U(x, m))

def nr(x, a, b, m):
    y = []
    for i in range(0, len(x)-1, 2):
        base = a + b * np.sqrt(-2 * np.log(1 - U(x[i], m)))
        trig_arg = 2 * np.pi * U(x[i+1], m)
        y.append(base * np.cos(trig_arg), base * np.sin(trig_arg))
    return np.array(y)

def gm(x, a, b, c, m):
    y = []
    u = U(x, m)
    if type(c) == type(1):
        for i in range(0, len(u), c):
            y.append(a - b  * np.log(np.prod(1 - u[i:i+c])))
    else:
        k = int(c - 0.5)
        for i in range(0, len(u), 2 * k + 2):
            z1, z2 = nr([x[i], x[i+1]], 0, 1, m)
            y.append(a + b * (z1 ** 2 / 2 - np.log(np.prod(1 - u[i+2:i+k+2]))))
            y.append(a + b * (z2 ** 2 / 2 - np.log(np.prod(1 - u[i+k+2:i+2*k+2]))))
    return np.array(y)

def ln(x, a, b, m):
    return a + np.exp(b - nr(x, 0, 1, m))

def ls(x, a, b, m):
    u = U(x, m)
    return a + b * np.log(u / (1 - u))

def bi(x, a, b, m):
    y = []
    u = U(x, m)
    for i in u:
        s, k = 0, 0
        while(True):
            s += (factor(b) / (factor(k) * factor(b - k)) * (a ** k) * ((1 - a) ** (b - k)))
            if s > i:
                y.append(k)
                break
            if k < b - 1:
                k += 1
                continue
            y.append(b)
```

```
    return np.array(y)
```

1.4 Дополнительные функции

```
import tqdm
import json
import numpy as np

def get_bit(num, num_bit):
    return (num & ( 1 << num_bit )) >> num_bit

def set_bit(num, num_bit, bit):
    mask = 1 << num_bit
    num &= ~mask
    if bit:
        return num | mask
    else:
        return num

def shift(num, s):
    new_num = 0
    bit = 0
    for i in range(s):
        new_num = set_bit(new_num, i, bit)
        bit = get_bit(num, i)
    new_num = set_bit(new_num, 0, bit)
    return new_num

def factor(x):
    y = 1
    for i in range(x):  y *= (i + 1)
    return y

def gen_write_file(gen, args):
    print("+-++-++-++-+_Генерирую_+-++-++-++-+")
    ans = [next(gen) % 1024 for _ in tqdm.tqdm(range(args.amount_of_numbers))]
    with open(args.filename, 'w') as fw:
        json.dump(ans, fw)
    print("+-++-++-++-+_Файл .dat готов_+-++-++-+")


def make_write_file(dist_dct, args):
    print("+-++-++-++-+_Генерирую_+-++-++-++-+")
    m = 1024
    with open(args.filename, 'r') as fr:
        nums = np.array(json.load(fr))
    if args.distribution == "gm":
        dist_nums = dist_dct[args.distribution](nums, args.p1, args.p2, args.p3, m)
    else:
        dist_nums = dist_dct[args.distribution](nums, args.p1, args.p2, m)
    np.savetxt(f"dist-{args.distribution}.dat", dist_nums, fmt='%1.4f')
    print("+-++-++-++-+_Файл .dat готов_+-++-++-+")
```

1.5 main задания 1

```
import argparse
from parser_arg import Pars_arg
from generators import *
from args import met, por
from utils import gen_write_file
```

```
gens_dct = {"lc": lc, "mt": mt, "add": add, "5p":fp, "lfsr": lfsr, "nfsr":nfsr, "rc4": rc4, "rsa":rsa, "bbs":bbs}


if __name__ == "__main__":
    parser = Pars_arg(description="Генераторы", argument_default=argparse.SUPPRESS, allow_abbrev=False, add_help=False)

    parser.add_argument("-g", "--generator", type=str, help="укажите генератор")
    parser.add_argument("-n", "--amount_of_numbers", type=int, default=10000, help="кол-во генерируемых чисел")
    parser.add_argument("-i", "--init", type=str, help="иниц. вектор генератора")
    parser.add_argument("-f", "--filename", type=str, default="random_sequence.dat", help="имя файла")
    parser.add_argument("-h", "--help", action="help", help="выводит это сообщение")

    print(met)
    print(por)

    args = parser.parse_args(namespace=None)
    gen = gens_dct[args.generator](args.init.split(sep=","))

    gen_write_file(gen, args)
```

1.6 main задания 2

```
import argparse
from parser_arg import Pars_arg
from utils import make_write_file
from distribution import *
from args import dist

dist_dct = {"st": st, "tr": tr, "ex": ex, "nr":nr, "gm": gm, "ln":ln, "ls": ls, "bi":bi}

def main_1():
    parser = Pars_arg(description="Приведение к определенному  распределению", argument_default=argparse.SUPPRESS, allow_abb
    parser.add_argument("-d", "--distribution", type=str, help="код распределения")
    parser.add_argument("-f", "--filename", type=str, help="имя файла")
    parser.add_argument("-p1", "-a", type=float, help="первый параметр")
    parser.add_argument("-p2", "-b", type=int, help="второй параметр")
    parser.add_argument("-p3", "-c", type=int, default=None, help="третий параметр")
    parser.add_argument("-h", "--help", action="help", help="выводит это сообщение")

    print (dist)

    args = parser.parse_args()

    make_write_file(dist_dct, args)

main_1()
```