Manual_implementation_of_the_Mersenne_twister_PseudoRandom_N

May 4, 2017

1 Table of Contents

2 Manual implementation of the Mersenne twister PseudoRandom Number Generator (PRNG)

This small notebook is a short experiment, to see if I can implement the Mersenne twister Pseudo-Random Number Generator (PRNG).

And then I want to use it to define a `rand()` function, and use it to samples from the most famous discrete and continuous probability distributions. Random permutations will also be studied.

- *Reference*: Wikipedia, and this book: "Simulation and the Monte-Carlo method", by R.Y.Rubinstein & D.P.Kroese ([Rubinstein & Kroese, 2017]), chapter 2 pages 52-53.
- *Date*: 11 March 2017.
- *Author*: Lilian Besson.
- *License*: MIT Licensed.

2.1 Common API for the PRNG defined here

First, I want to define a simple object-oriented API, in order to write all the examples of PNRG with the same interface.

```
In [131]: import numpy as np
```

```
In [132]: class PRNG(object):
              """Base class for any Pseudo-Random Number Generator."""
              def __init__(self, X0=0):
                  """Create a new PRNG with seed X0."""
                  self.X0 = X0
                  self.X = X0
                  self.t = 0
                  self.max = 0

              def __iter__(self):
                  """self is already an iterator!"""
                  return self

              def seed(self, X0=None):
                  """Reinitialize the current value with X0, or self.X0.

                  - Tip: Manually set the seed if you need reproducibility in your results.
                  """
                  self.t = 0
                  self.X = self.X0 if X0 is None else X0

              def __next__(self):
```

```
        """Produce a next value and return it."""
        # This default PRNG does not produce random numbers!
        self.t += 1
        return self.X

    def randint(self, *args, **kwargs):
        """Return an integer number in [| 0, self.max - 1 |] from the PRNG."""
        return self.__next__()

    def int_samples(self, shape=(1,)):
        """Get a numpy array, filled with integer samples from the PRNG, of shape = 
        # return [ self.randint() for _ in range(size) ]
        return np.fromfunction(np.vectorize(self.randint), shape=shape, dtype=int)

    def rand(self, *args, **kwargs):
        """Return a float number in [0, 1) from the PRNG."""
        return self.randint() / float(1 + self.max)

    def float_samples(self, shape=(1,)):
        """Get a numpy array, filled with float samples from the PRNG, of shape = sh
        # return [ self.rand() for _ in range(size) ]
        return np.fromfunction(np.vectorize(self.rand), shape=shape, dtype=int)
```

2.2 First example: a simple linear congruential generator

Let me start by implementing a simple linear congruential generator, with three parameters m, a, c, defined like this :

- Start from X_0,
- And then follow the recurrence equation:

$$X_{t+1} = (aX_t + c) \mod m.$$

This algorithm produces a sequence $(X_t)_{t \in \mathbb{N}} \in \mathbb{N}^{\mathbb{N}}$.

In [133]:
```
class LinearCongruentialGenerator(PRNG):
    """A simple linear congruential Pseudo-Random Number Generator."""
    def __init__(self, m, a, c, X0=0):
        """Create a new PRNG with seed X0."""
        super(LinearCongruentialGenerator, self).__init__(X0=X0)
        self.m = self.max = m
        self.a = a
        self.c = c

    def __next__(self):
        """Produce a next value and return it, following the recurrence equation: X_
        self.t += 1
```

```python
            x = self.X
            self.X = (self.a * self.X + self.c) % self.m
            return x
```

The values recommended by the authors, Lewis, Goodman and Miller, are the following:

```python
In [134]: m = 1 << 31 - 1  # 1 << 31 = 2**31
          a = 7 ** 4
          c = 0
```

The seed is important. If $X_0 = 0$, this first example PRNG will only produce $X_t = 0, \forall t$.

```python
In [135]: FirstExample = LinearCongruentialGenerator(m=m, a=a, c=c)
```

```python
In [136]: def test(example, nb=3):
              for t, x in enumerate(example):
                  print("{:>3}th value for {.__class__.__name__} is X_t = {:>10}".format(t, exa
                  if t >= nb - 1:
                      break
```

```python
In [137]: test(FirstExample)
```

```
  0th value for LinearCongruentialGenerator is X_t =          0
  1th value for LinearCongruentialGenerator is X_t =          0
  2th value for LinearCongruentialGenerator is X_t =          0
```

But with any positive seed, the sequence will appear random.

```python
In [138]: SecondExample = LinearCongruentialGenerator(m=m, a=a, c=c, X0=12011993)
```

```python
In [139]: test(SecondExample)
```

```
  0th value for LinearCongruentialGenerator is X_t =   12011993
  1th value for LinearCongruentialGenerator is X_t =  923507769
  2th value for LinearCongruentialGenerator is X_t =   65286809
```

The sequence is completely determined by the seed X_0:

```python
In [140]: SecondExample.seed(12011993)
          test(SecondExample)
```

```
  0th value for LinearCongruentialGenerator is X_t =   12011993
  1th value for LinearCongruentialGenerator is X_t =  923507769
  2th value for LinearCongruentialGenerator is X_t =   65286809
```

Note: I prefer to use this custom class to define iterators, instead of a simple generator (with `yield` keyword) as I want them to have a `.seed(X0)` method.

2.3 Trying to write a cell in cython, for speeding things up

For more details, see this blog post, and this other one.

```
In [141]: # Thanks to https://nbviewer.jupyter.org/gist/minrk/7715212
          from __future__ import print_function
          from IPython.core import page
          def myprint(s):
              try:
                  print(s['text/plain'])
              except (KeyError, TypeError):
                  print(s)
          page.page = myprint
```

```
In [142]: %load_ext cython
```

```
The cython extension is already loaded. To reload it, use:
  %reload_ext cython
```

Then we define a function `LinearCongruentialGenerator_next`, in a Cython cell.

```
In [143]: %%cython
          def nextLCG(int x, int a, int c, int m):
              """Compute x, nextx = (a * x + c) % m, x in Cython."""
              cdef int nextx = (a * x + c) % m
              return (x, nextx)
```

```
In [144]: from __main__ import nextLCG
          nextLCG
          nextLCG?
```

```
Out[144]: <function _cython_magic_dde6682b939b6e9ea0a22da681a4bea1.nextLCG>
```

```
Docstring: Compute x, nextx = (a * x + c) % m, x in Cython.
Type:      builtin_function_or_method
```

Then it's easy to use it to define another Linear Congruential Generator.

```
In [145]: class CythonLinearCongruentialGenerator(LinearCongruentialGenerator):
              """A simple linear congruential Pseudo-Random Number Generator, with Cython acce

              def __next__(self):
                  """Produce a next value and return it, following the recurrence equation: X_
                  self.t += 1
                  x, self.X = nextLCG(self.X, self.a, self.c, self.m)
                  return x
```

Let compare it with the first implementation (using pure Python).

```
In [146]: NotCythonSecondExample = LinearCongruentialGenerator(m=m, a=a, c=c, X0=13032017)
          CythonSecondExample = CythonLinearCongruentialGenerator(m=m, a=a, c=c, X0=13032017)
```

They both give the same values, that's a relief.

```
In [147]: test(NotCythonSecondExample)
          test(CythonSecondExample)
```

```
  0th value for LinearCongruentialGenerator is X_t =   13032017
  1th value for LinearCongruentialGenerator is X_t =  151359921
  2th value for LinearCongruentialGenerator is X_t =  490433809
  0th value for CythonLinearCongruentialGenerator is X_t =   13032017
  1th value for CythonLinearCongruentialGenerator is X_t =  151359921
  2th value for CythonLinearCongruentialGenerator is X_t =  490433809
```

The speedup is not great, but still visible.

```
In [148]: %timeit [ NotCythonSecondExample.randint() for _ in range(1000000) ]
          %timeit [ CythonSecondExample.randint() for _ in range(1000000) ]
```

```
1 loop, best of 3: 766 ms per loop
1 loop, best of 3: 729 ms per loop
```

```
In [149]: %prun min(CythonSecondExample.randint() for _ in range(1000000))
```

```
         4000005 function calls in 1.291 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000000    0.481    0.000    0.584    0.000 <ipython-input-145-d36c39b118f6>:4(__next__)
  1000001    0.372    0.000    1.184    0.000 <string>:1(<genexpr>)
  1000000    0.228    0.000    0.812    0.000 <ipython-input-132-93560edf79a4>:28(randint)
        1    0.107    0.107    1.291    1.291 {built-in method builtins.min}
  1000000    0.103    0.000    0.103    0.000 {_cython_magic_dde6682b939b6e9ea0a22da681a4bea1.r
        1    0.000    0.000    1.291    1.291 {built-in method builtins.exec}
        1    0.000    0.000    1.291    1.291 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

2.4 Checking and plotting the result?

First, we can generate a matrix of samples, as random floats in $[0, 1)$, and check that the mean is about $1/2$:

```
In [150]: shape = (400, 400)
          image = SecondExample.float_samples(shape)
```

```
In [151]: np.mean(image), np.var(image)
```

```
Out[151]: (0.4996821506033815, 0.0833934768218O3994)
```

What about the speed? Of course, a hand-written Python code will always be really slower than a C-extension code, and the PRNG from the modules `random` or `numpy.random` are written in C (or Cython), and so will always be faster. But how much faster?

```
In [152]: import random
          import numpy.random

          print(np.mean(SecondExample.float_samples(shape)))
          print(np.mean([ [ random.random() for _ in range(shape[0]) ] for _ in range(shape[1]
          print(np.mean(numpy.random.random(shape)))
```

```
0.500181275654
0.500273447015
0.499487700229
```

```
In [153]: %timeit SecondExample.float_samples(shape)
          %timeit [ [ random.random() for _ in range(shape[0]) ] for _ in range(shape[1]) ]
          %timeit numpy.random.random(shape)
```

```
1 loop, best of 3: 271 ms per loop
100 loops, best of 3: 22.2 ms per loop
1000 loops, best of 3: 1.56 ms per loop
```

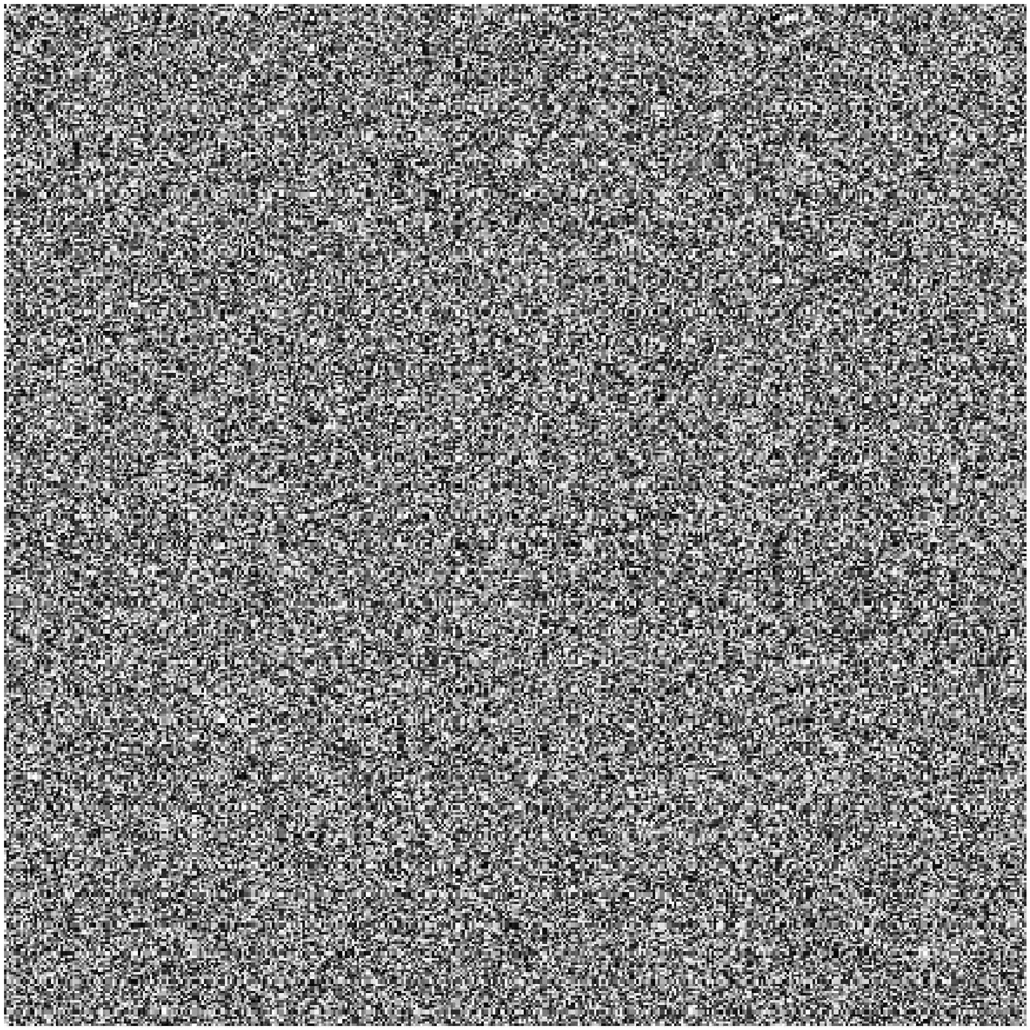
This was expected: of course `numpy.random.` functions are written and optimized to generate thousands of samples quickly, and of course my hand-written Python implementation for `LinearCongruentialGenerator` is slower than the C-code generating the module `random`.

We can also plot this image as a grayscaled image, in order to visualize this "randomness" we just created.

```
In [268]: %matplotlib inline
          import matplotlib.pyplot as plt

          def showimage(image):
              plt.figure(figsize=(8, 8))
              plt.imshow(image, cmap='gray', interpolation='none')
              plt.axis('off')
              plt.show()
```

In [269]: showimage(image)

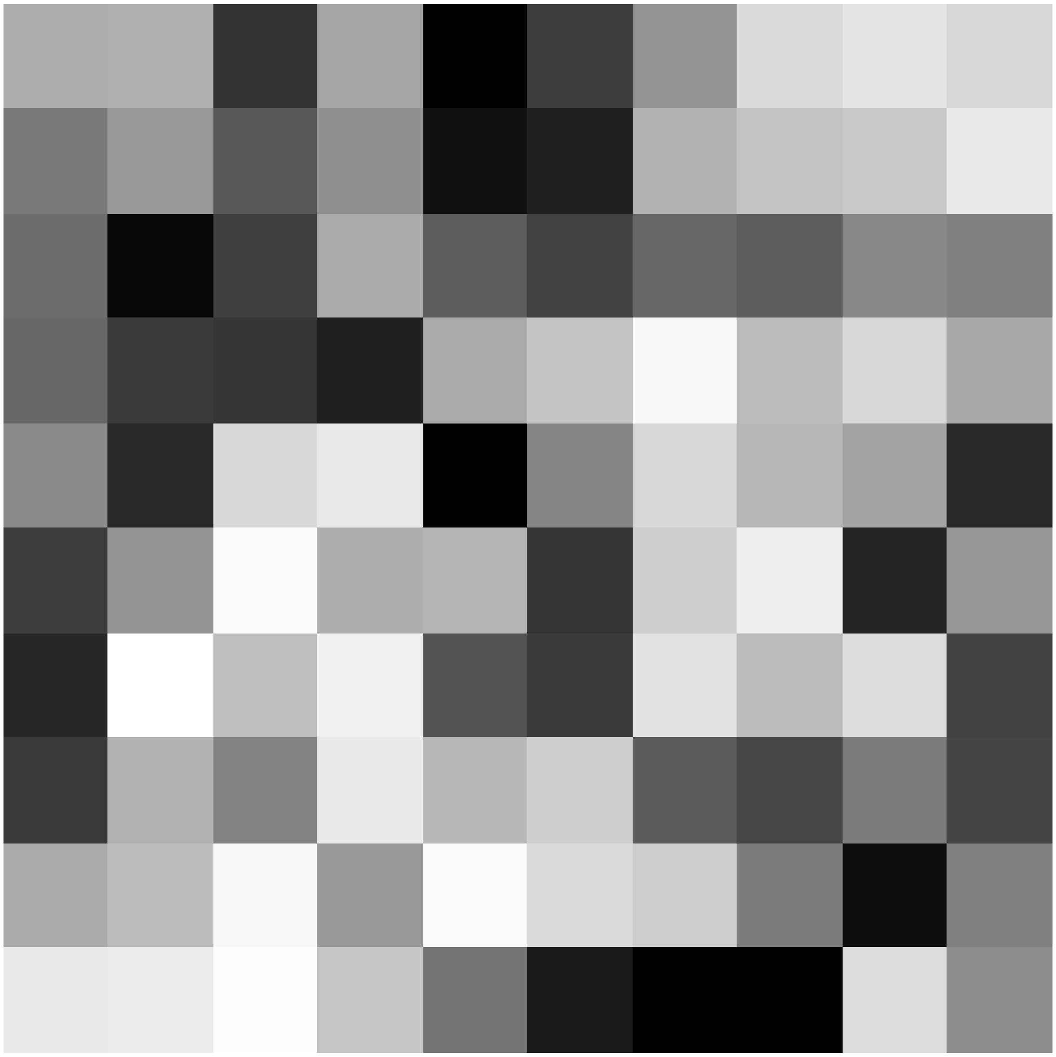
```
showimage(image)
```



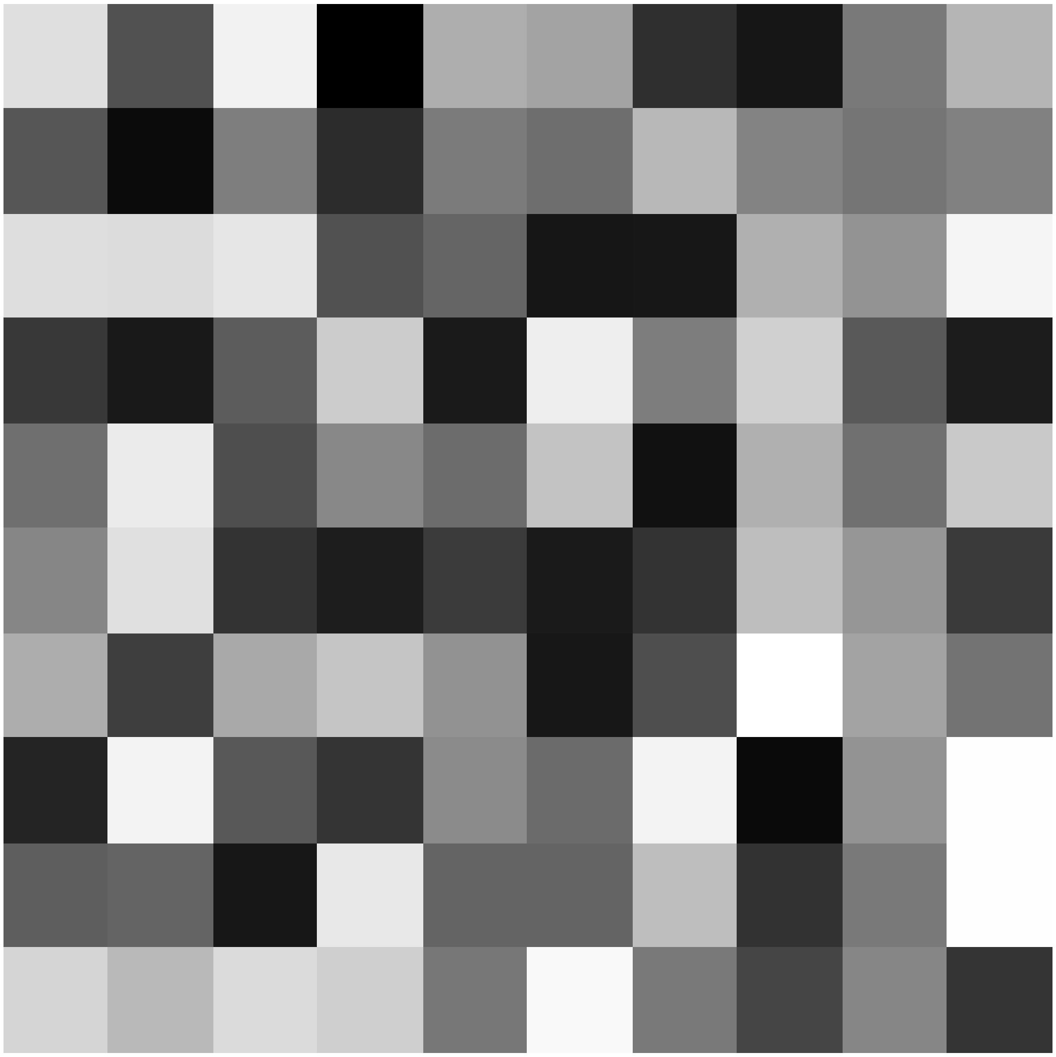
It looks good already! We can't see any recurrence, but we see a regularity, with small squares. And it does not seem to depend too much on the seed:

In [270]:

```
SecondExample.seed(11032017)
image = SecondExample.float_samples((10, 10))
showimage(image)
```

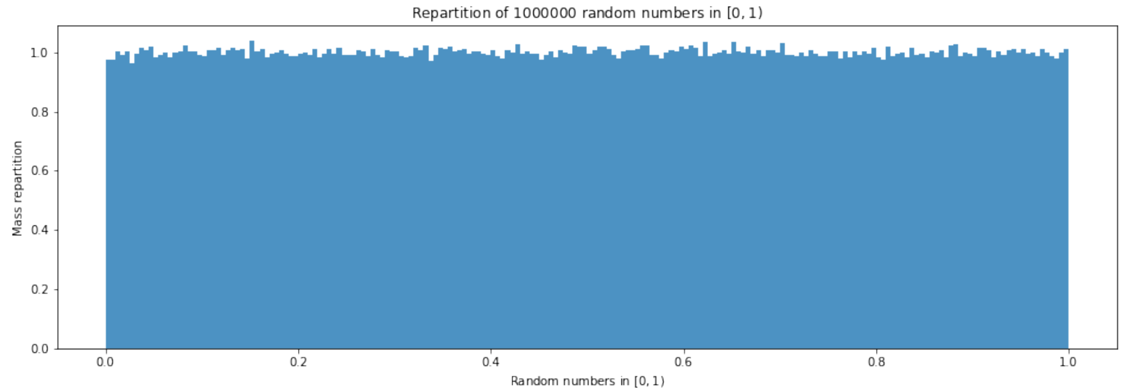


```
In [271]: SecondExample.seed(1103201799)
          image = SecondExample.float_samples((10, 10))
          showimage(image)
```



We can also visualize the generated numbers with a histogram, to visually check that the random numbers in $[0,1)$ are indeed "uniformly" located.

```
In [272]: def plotHistogram(example, nb=100000, bins=200):
              numbers = example.float_samples((nb,))
              plt.figure(figsize=(16, 5))
              plt.hist(numbers, bins=bins, normed=True, alpha=0.8)
              plt.xlabel("Random numbers in $[0, 1)$")
              plt.ylabel("Mass repartition")
              plt.title("Repartition of ${}$ random numbers in $[0, 1)$".format(nb))
              plt.show()

In [273]: plotHistogram(SecondExample, 1000000, 200)
```



2.5 A second example: Multiple-Recursive Generator

Let start by writing a generic Multiple Recursive Generator, which is defined by the following linear recurrence equation, of order $k \geq 1$:

- Start from X_0, with a false initial history of $(X_{-k+1}, X_{-k}, \ldots, X_{-1})$,
- And then follow the recurrence equation:

$$X_t = (a_1 X_{t-1} + \cdots + a_k X_{t-k}) \mod m.$$

This algorithm produces a sequence $(X_t)_{t \in \mathbb{N}} \in \mathbb{N}^{\mathbb{N}}$.

```
In [160]: class MultipleRecursiveGenerator(PRNG):
              """A Multiple Recursive Pseudo-Random Number Generator (MRG), with one sequence
              def __init__(self, m, a, X0):
                  """Create a new PRNG with seed X0."""
                  assert np.shape(a) == np.shape(X0), "Error: the weight vector a must have the
                  super(MultipleRecursiveGenerator, self).__init__(X0=X0)
                  self.m = self.max = m
                  self.a = a

              def __next__(self):
                  """Produce a next value and return it, following the recurrence equation: X_
                  self.t += 1
                  x = self.X[0]
                  nextx = (np.dot(self.a, self.X)) % self.m
                  self.X[1:] = self.X[:-1]
                  self.X[0] = nextx
                  return x
```

For example, with an arbitrary choice of $k = 3$, of weights $a = [10, 9, 8]$ and $X_0 = [10, 20, 30]$:

```
In [161]: m = (1 << 31) - 1
          X0 = np.array([10, 20, 30])
          a = np.array([10, 9, 8])

          ThirdExample = MultipleRecursiveGenerator(m, a, X0)

          test(ThirdExample)
```

```
  0th value for MultipleRecursiveGenerator is X_t =         10
  1th value for MultipleRecursiveGenerator is X_t =        520
  2th value for MultipleRecursiveGenerator is X_t =       5450
```

We can again check for the mean and the variance of the generated sequence:

```
In [275]: shape = (400, 400)
          image = ThirdExample.float_samples(shape)
          np.mean(image), np.var(image)

Out[275]: (0.49952060992684566, 0.083438346428071117)
```

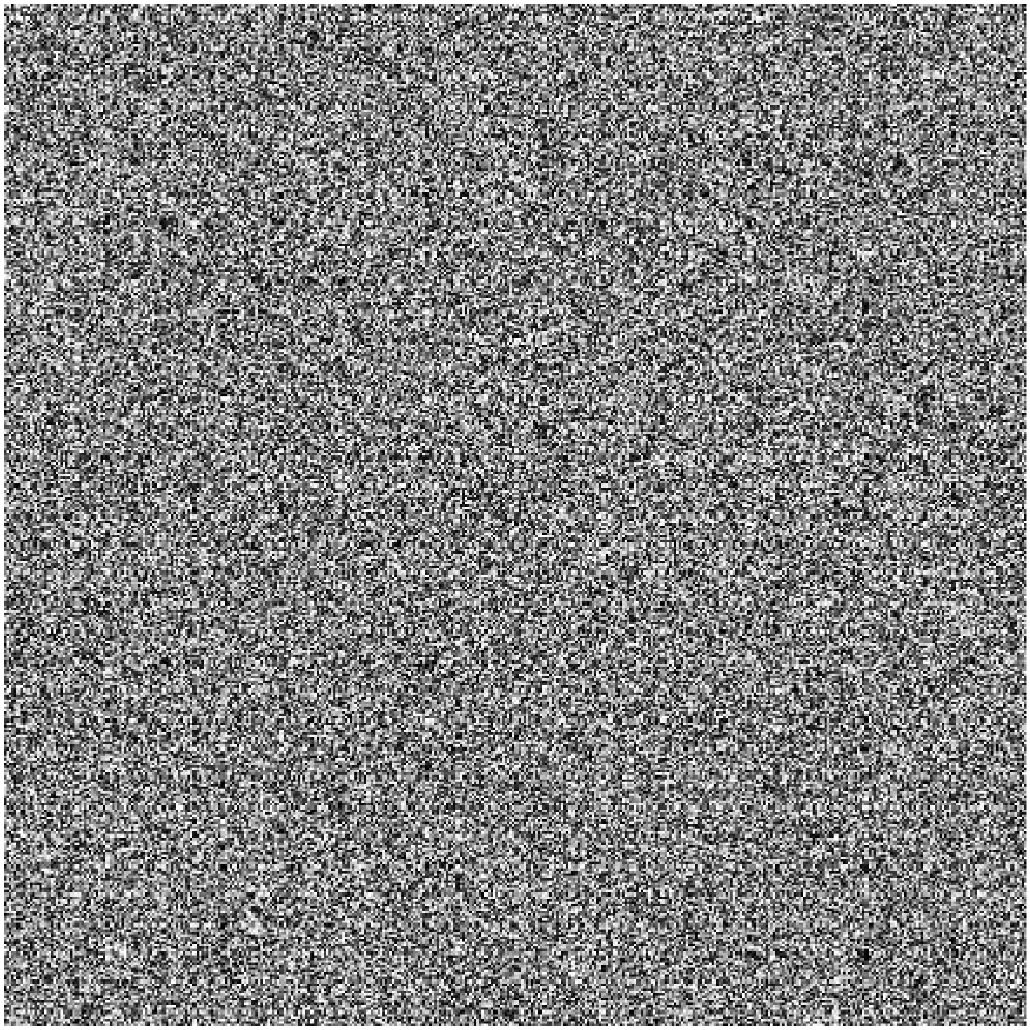
This Multiple Recursive Generator is of course slower than the simple Linear Recurrent Generator:

```
In [163]: %timeit SecondExample.float_samples(shape)
          %timeit ThirdExample.float_samples(shape)
```

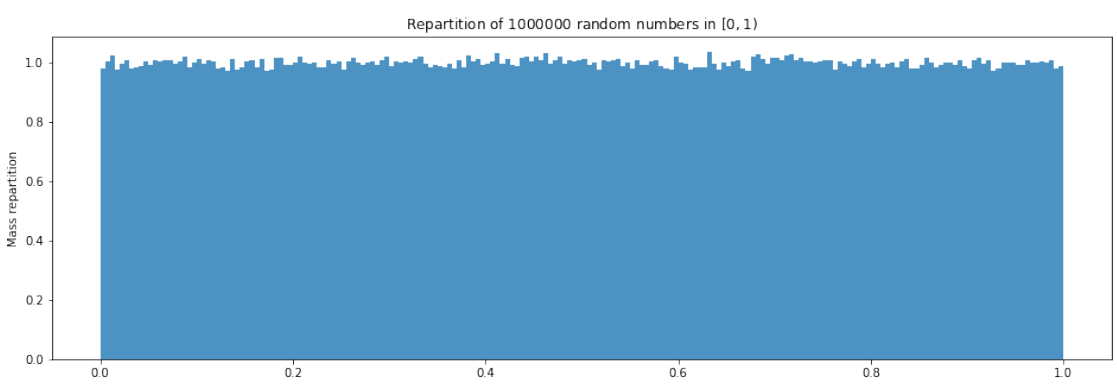
```
1 loop, best of 3: 221 ms per loop
1 loop, best of 3: 895 ms per loop
```

And it seems to work fine as well:

```
In [276]: showimage(image)
```



```
In [277]: plotHistogram(ThirdExample, 1000000, 200)
```



It looks also good!

2.6 A third example: combined Multiple-Recursive Generator, with `MRG32k3a`

Let start by writing a generic Multiple Recursive Generator, which is defined by the following coupled linear recurrence equation, of orders $k_1, k_2 \geq 1$:

- Start from X_0 and Y_0, with a false initial history of $(X_{-k_1+1}, X_{-k_1}, \dots, X_{-1})$ and $(Y_{-k_2+1}, Y_{-k_2}, \dots, Y_{-1})$,
- And then follow the recurrence equation:

$$X_t = (a_1 X_{t-1} + \cdots + a_{k_1} X_{t-k_1}) \mod m.$$

and

$$Y_t = (b_1 Y_{t-1} + \cdots + b_{k_2} Y_{t-k_2}) \mod m.$$

This algorithm produces two sequences $(X_t)_{t\in\mathbb{N}} \in \mathbb{N}^{\mathbb{N}}$ and $(X_t)_{t\in\mathbb{N}} \in \mathbb{N}^{\mathbb{N}}$, and usually the sequence used for the output is $U_t = X_t - Y_t + \max(m_1, m_2)$.

```
In [166]: class CombinedMultipleRecursiveGenerator(PRNG):
              """A Multiple Recursive Pseudo-Random Number Generator (MRG), with two sequences
              def __init__(self, m1, a, X0, m2, b, Y0):
                  """Create a new PRNG with seeds X0, Y0."""
                  assert np.shape(a) == np.shape(X0), "Error: the weight vector a must have the
                  assert np.shape(b) == np.shape(Y0), "Error: the weight vector b must have the
                  self.t = 0
                  # For X
                  self.m1 = m1
                  self.a = a
                  self.X0 = self.X = X0
                  # For Y
                  self.m2 = m2
                  self.b = b
                  self.Y0 = self.Y = Y0
                  # Maximum integer number produced is max(m1, m2)
                  self.m = self.max = max(m1, m2)

              def __next__(self):
                  """Produce a next value and return it, following the recurrence equation: X_
                  self.t += 1
                  # For X
                  x = self.X[0]
                  nextx = (np.dot(self.a, self.X)) % self.m1
                  self.X[1:] = self.X[:-1]
```

```
                self.X[0] = nextx
                # For Y
                y = self.Y[0]
                nexty = (np.dot(self.b, self.Y)) % self.m2
                self.Y[1:] = self.Y[:-1]
                self.Y[0] = nexty
                # Combine them
                u = x - y + (self.m1 if x <= y else 0)
                return u
```

To obtain the well-known MRG32k3a generator, designed by L'Ecuyer in 1999, we choose these parameters:

```
In [167]: m1 = (1 << 32) - 209                   # important choice!
          a = np.array([0, 1403580, -810728])    # important choice!
          X0 = np.array([1000, 10000, 100000])   # arbitrary choice!

          m2 = (1 << 32) - 22853                 # important choice!
          b = np.array([527612, 0, -1370589])    # important choice!
          Y0 = np.array([5000, 50000, 500000])   # arbitrary choice!

          MRG32k3a = CombinedMultipleRecursiveGenerator(m1, a, X0, m2, b, Y0)

          test(MRG32k3a)
```

```
  0th value for CombinedMultipleRecursiveGenerator is X_t = 4294963087
  1th value for CombinedMultipleRecursiveGenerator is X_t = 1442746955
  2th value for CombinedMultipleRecursiveGenerator is X_t =  970596549
```

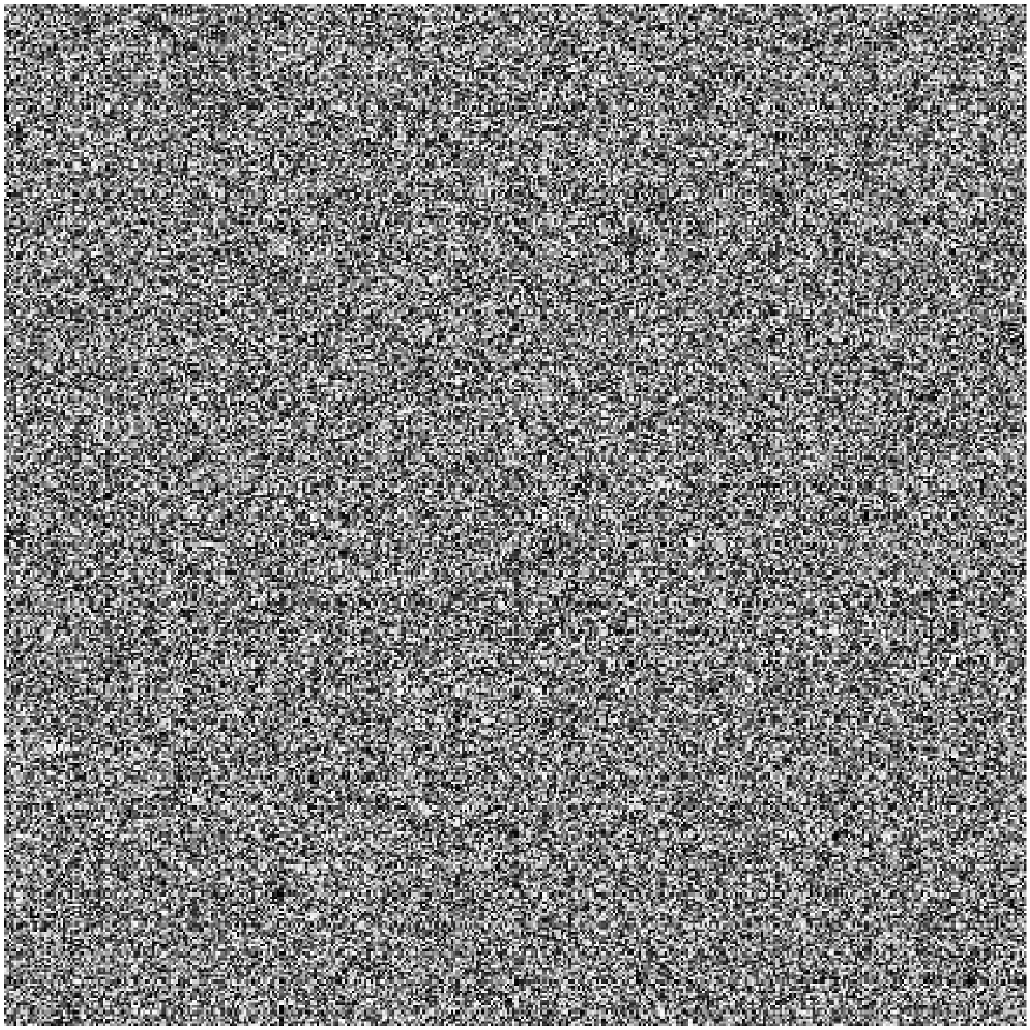
We can again check for the mean and the variance of the generated sequence:

```
In [278]: shape = (400, 400)
          image = MRG32k3a.float_samples(shape)
          np.mean(image), np.var(image)

Out[278]: (0.49952650455296843, 0.08318110283764904)
```

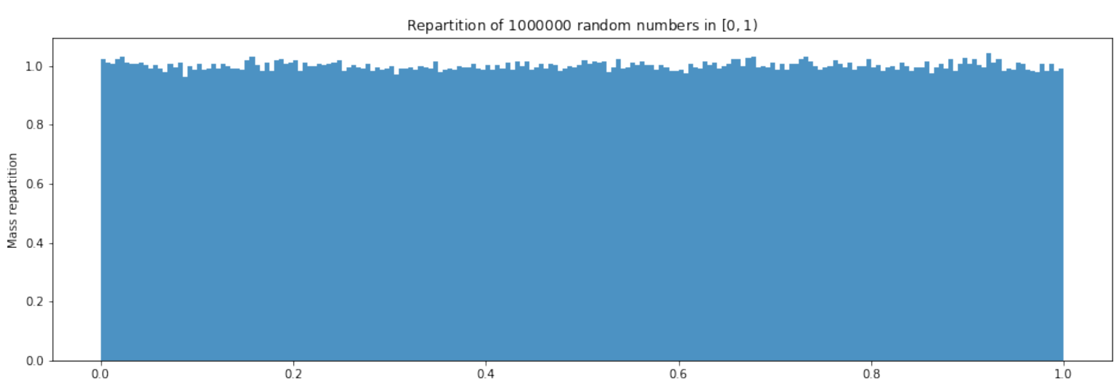
This combined Multiple Recursive Generator is of course slower than the simple Multiple Recursive Generator and the simple Linear Recurrent Generator:

```
In [169]: %timeit SecondExample.float_samples(shape)
          %timeit ThirdExample.float_samples(shape)
          %timeit MRG32k3a.float_samples(shape)

1 loop, best of 3: 251 ms per loop
1 loop, best of 3: 863 ms per loop
1 loop, best of 3: 1.24 s per loop
```

```
In [279]: showimage(image)
```



```
In [280]: plotHistogram(MRG32k3a, 1000000, 200)
```



This one looks fine too!

2.7 Finally, the Mersenne twister PRNG

I won't explain all the details, and will follow closely the notations from my reference book [Rubinstein & Kroese, 2017]. It will be harder to implement!

First, let us compute the period of the PRNG we will implement, with the default values for the parameters $w = 32$ (word length) and $n = 624$ ("big" integer).

2.7.1 Period

```
In [172]: w = 32
          n = 624
```

```
In [173]: def MersenneTwisterPeriod(n, w):
              return (1 << (w * (n - 1) + 1)) - 1

          MersenneTwisterPeriod(n, w) == (2 ** 19937) - 1
```

```
Out[173]: True
```

2.7.2 Random seeds

Then we need to use a previously defined PRNG to set the random seeds.

To try to have "really random" seeds, let me use that classical trick of using the system time as a source of initial randomness.

- Namely, I will use the number of microseconds in the current time stamp as the seed for a `LinearCongruentialGenerator`,
- Then use it to generate the seeds for a `MRG32k3a` generator,
- And finally use it to get the seed for the Mersenne twister.

```
In [174]: from datetime import datetime

          def get_seconds():
              d = datetime.today().timestamp()
              s = 1e6 * (d - int(d))
              return int(s)
```

```
In [175]: get_seconds()   # Example
```

```
Out[175]: 785506
```

In [176]:
```python
def seed_rows(example, n, w):
    return example.int_samples((n,))

def random_Mersenne_seed(n, w):
    linear = LinearCongruentialGenerator(m=(1 << 31) - 1, a=7 ** 4, c=0, X0=get_secon
    assert w == 32, "Error: only w = 32 was implemented"
    m1 = (1 << 32) - 209                        # important choice!
    a = np.array([0, 1403580, -810728])    # important choice!
    X0 = np.array(linear.int_samples((3,)))  # random choice!
    m2 = (1 << 32) - 22853                      # important choice!
    b = np.array([527612, 0, -1370589])    # important choice!
    Y0 = np.array(linear.int_samples((3,)))  # random choice!
    MRG32k3a = CombinedMultipleRecursiveGenerator(m1, a, X0, m2, b, Y0)
    seed = seed_rows(MRG32k3a, n, w)
    assert np.shape(seed) == (n,)
    return seed

example_seed = random_Mersenne_seed(n, w)
example_seed
```

Out[176]:
```
array([1871239779,  613260058,  244547519, 3267481671, 1554624298,
       1961991761, 3811287966, 4176129021,  848956982,  121246666,
       1754035200, 3424467876,  866268922,  230379068, 1178928465,
       2097034094, 1445939073,  398964532,  462460512, 2750298176,
       3359458013, 1075693109, 3633367586, 3584582396, 1524185041,
       3120497617, 1384358948,  760092626,  468632607, 3718611854,
        170312151,   93043999, 1302889854, 1822754143, 3094198579,
        358234833,  563461773, 3236562444, 1564295301, 3728669490,
        323737918, 4138492073, 2493003270,  437271590, 2630117485,
       2097611595, 3253545216, 2375586543, 2334725582, 2516044957,
        294245058, 2177777445, 1309507195, 3435158184, 1695128490,
       1098019007, 1029996593, 3838670278,  372127336,  262477370,
       3888725742,  138313317,  769363798, 1201876561, 2916189143,
       4117389322, 2323047164, 4273429928, 2568524443, 2070312259,
       1761871632, 4272177704, 3699864390, 1871995760,  127339982,
        761864309, 3849143015, 2018064955, 1426734324, 2180708922,
       1341176491, 3763535641, 2288304274, 3580498047, 2502082897,
       1837725291,  309840173,  638977987, 1629952507, 3562370204,
        459167486, 1159150577, 2422421702, 2049480456,  955606742,
       3318517220, 1592267301,  830866714, 2344862225, 3702789056,
       1922196881, 3742471018, 1541965974, 4142146636, 3623344569,
        279163925, 3836602443,  167259802, 1150841726, 2482754657,
       2655828943, 3320007058, 3983552941, 2617035024,  321365567,
       3536735818,  531553946,  621116189, 1468340664, 3372085586,
       2462640705, 4163215166, 4095620748, 3144005459, 2503935906,
       3723739935, 2765663758, 1150437271, 4285161495, 3181830612,
       4281399551, 1548429246, 3973263892, 1197689553, 2927586334,
       3652428993, 1726975315, 1410287868, 2788568833, 2628719687,
```

1518108225, 2051151561, 4093100836, 3218627959, 866978907,
7673933, 1741131969, 1023486388, 979415369, 84030971,
906919976, 1259848032, 2595271731, 1150309581, 2140692285,
3164682335, 3234552544, 3353877610, 461770900, 3327442941,
1826940947, 315178113, 590842540, 3074995021, 2944579553,
3344583820, 1062760645, 2109587268, 2795807192, 2319186164,
187075502, 1714589414, 2774104052, 3453621970, 1598122889,
899984745, 1570482228, 825029584, 2777306222, 1689646329,
82836012, 1759394550, 2042694026, 3652870993, 2503834779,
25989907, 3844683156, 5858153, 2960765079, 3299532383,
3364505341, 3314005986, 703364433, 2762877941, 392848013,
2950192981, 3546445627, 1393147189, 2762292792, 3048070016,
2420254782, 1931256725, 3224385554, 3111643288, 4265900604,
3619397695, 3349967630, 752273190, 1164687372, 4178198718,
372698645, 985091608, 2876386582, 216088733, 2940859240,
3447672867, 3261159225, 2176508969, 2526561005, 1561975407,
979546518, 943402798, 304330565, 127919861, 572374036,
361724688, 2042128148, 2835890243, 2875732453, 2310551627,
1118959925, 3602026288, 3472508057, 3431282832, 3134350121,
2010598072, 127088347, 2973107558, 2000852815, 320239350,
680940291, 396820841, 3679059793, 1881628617, 303615556,
227493472, 2862897834, 3380125342, 3212420406, 3241702604,
3497857423, 3044902665, 3997810242, 2819226446, 3836250340,
3071902144, 3044231609, 522790870, 3280296618, 1538648392,
1451279128, 846660538, 1932990326, 122293128, 4040211924,
1901162832, 1904580259, 1277687776, 944593750, 530550504,
721180654, 1124337506, 1245184760, 3109609950, 52617178,
4201242225, 3895642903, 1287140348, 4138067394, 94933038,
3552894923, 606686675, 3979810835, 3495120745, 2119495731,
3907577698, 412831626, 1325451416, 2167073398, 2589438056,
3363313475, 3035423769, 3602940889, 2023524015, 1344633747,
2974312586, 2656513143, 3564159070, 1263023414, 3626652266,
3733065320, 103921927, 2860598176, 1035417477, 2333161992,
1948127030, 1338038202, 208778934, 2448700181, 3997041044,
44825507, 2349939274, 3784895872, 2839541553, 3037912645,
2196314732, 442635973, 165298014, 541044006, 2059327483,
3790590621, 1763299261, 2497410676, 2844785294, 3021937416,
1918582056, 2067126237, 532963981, 1200662532, 3698447250,
3794861103, 2836275841, 4212201754, 2319959648, 520349230,
2966098732, 221723408, 2160821088, 3109477811, 1072896526,
1383202204, 2891329434, 3501245406, 3751157658, 69716829,
1505668878, 4098775850, 2545738577, 4204369634, 765672134,
208361897, 1214090833, 3582397093, 1608852417, 1208273721,
2526085509, 2619649789, 2784814564, 3055715720, 1441792327,
2272192152, 3724233894, 3369842723, 2989772752, 2289415600,
3533767613, 3490874775, 3827963379, 1584073120, 2853938450,
988337756, 3740176171, 1193018243, 2813578097, 3270561066,
3934637806, 1391677154, 805867164, 638376301, 832099524,

```
 909767166, 2558294659, 2138249556, 3942492888, 1210708829,
 104068362,  620694524, 3540731802, 2050859204, 1973043272,
2000877646, 2273213274, 1503292513, 1645780613, 1325892045,
2082049544, 1597725327,  696729750, 3357922476, 1791433290,
3006378686,  232795670, 4274222388, 3614272321, 2017489941,
3638051888,  315054746,  142045536, 4115602201,  417078816,
1702240374, 4219712046, 1089219343, 2871526145,  845376169,
1124383960, 2152432773, 2345721609, 2924788505, 4168640085,
4284335564, 3232561385,  155584805, 3570148724, 3326938343,
4105697636, 2632643363, 1666723936, 1440099044,  680961589,
 356011767, 2378600510,   13651577, 3813736373,  609952374,
 506997792,  886946253,  392862710, 2287913525, 1119639152,
 851165101, 4134565370, 1412858691,   23444368,     923437,
2689526255, 2270805862,  155417897, 3304267744, 3753270291,
 287608410, 2653855879, 1092528355, 1660551498, 1672214369,
 994760694, 1454465055,  180809258,  398171832, 3582972302,
1372157519, 1913948096,  699148529, 2363338116, 1915498460,
 110204858, 2313071172, 3296436850,  844301127, 4280175860,
1350657851, 2464529858,  397187298, 3211705042, 2292170038,
1111238565, 1280853189, 1779688143, 3854704930, 2048092710,
 370319143, 1186503081, 2253582330, 2894988877, 2693059081,
3174303371, 1665367233,   48241162, 3315195164, 2764409186,
 926421411, 4016649256,  322794763, 1511800056, 2264490811,
2861792678, 2339545991, 2135460562, 2776313230, 3167401938,
3268433721, 3702129285, 3641075716,  860122460,  125856162,
3305655026,  165496898, 1885756789,  500190595, 3935572826,
 894747617, 3133644978, 1526289952, 1168837256, 4017278431,
2171965999, 1104447011,  327894088, 3514912582, 3018660396,
1040193686, 3947341728, 2948643316, 3031481819, 3532554225,
3417828952, 2276259231,  530376740,  615559460, 3620057188,
3694586431,  743200431, 2776544714, 3244992256, 1416025503,
2512356848,  718405397, 1334387131,  697147504, 1142438326,
4252891009, 2184797146, 3266165526, 2760011335, 1978734280,
2676865440, 2614967515, 3885167354, 3197041332, 1493472995,
2077148832, 4155640942, 3364657610, 2315301646, 1351058597,
 978466409, 2915203450, 4265774007, 1639544022, 1041153282,
 786619918, 2471074780, 3925072875, 1414214583, 1827954335,
 933906235, 3006686438, 3631298850,  643958142,  911311576,
2529950011, 3405919962,  142496341,  949522371,  908076311,
1914623417, 1965182279, 2192742552, 3885699423, 1944076290,
 541678656, 2391536087,  316854319,   18424049, 2485161495,
 633174594, 3856720693, 1313045829,  247694843,  110381076,
1061663071, 2902807239, 2917247060,  675703803, 3766311495,
2946801748,  585112101, 2935817224, 2142413707,  398795855,
3755006652, 1567043098,  277852780, 1675396547, 3583522452,
4265050137, 1174975985, 3348363442, 3541378862, 2730613383,
4240573901, 1064098827, 3966639723, 2775717648, 3339443893,
2941191519, 1353655096, 3254930476,  904857047, 3907208435,
```

```
                    2948098651, 1078468758, 3981043271, 3907076547])
```

```
In [177]: for xi in example_seed:
              print("Integer xi = {:>12} and in binary, bin(xi) = {:>34}".format(xi, bin(xi)))
```

```
Integer xi =   1871239779 and in binary, bin(xi) =  0b1101111100010001101101001100011
Integer xi =    613260058 and in binary, bin(xi) =   0b100100100011011001101100011010
Integer xi =    244547519 and in binary, bin(xi) =     0b1110100100110111111110111111
Integer xi =   3267481671 and in binary, bin(xi) = 0b11000010110000001110100000100011 1
Integer xi =   1554624298 and in binary, bin(xi) =  0b1011100101010011010111100101010
Integer xi =   1961991761 and in binary, bin(xi) =  0b1110100111100011001111001010001
Integer xi =   3811287966 and in binary, bin(xi) = 0b11100011001010111010001110011110
Integer xi =   4176129021 and in binary, bin(xi) = 0b11111000111010101010101111111101
Integer xi =    848956982 and in binary, bin(xi) =   0b110010100110100000111000110110
Integer xi =    121246666 and in binary, bin(xi) =      0b111001110100001001111001010
Integer xi =   1754035200 and in binary, bin(xi) =  0b1101000100011000111010000000000
Integer xi =   3424467876 and in binary, bin(xi) = 0b11001100000111010011101110100100
Integer xi =    866268922 and in binary, bin(xi) =   0b110011101000100011011011111010
Integer xi =    230379068 and in binary, bin(xi) =     0b1101101110110100111000111100
Integer xi =   1178928465 and in binary, bin(xi) =  0b1000110010001010000010101010001
Integer xi =   2097034094 and in binary, bin(xi) =  0b1111100111111100011001101101110
Integer xi =   1445939073 and in binary, bin(xi) =  0b1010110001011110100011110000001
Integer xi =    398964532 and in binary, bin(xi) =    0b10111110001111011011100110100
Integer xi =    462460512 and in binary, bin(xi) =    0b11011100100001001011001100000
Integer xi =   2750298176 and in binary, bin(xi) = 0b10100011111011100011100001000000
Integer xi =   3359458013 and in binary, bin(xi) = 0b11001000001111010100001011011101
Integer xi =   1075693109 and in binary, bin(xi) =  0b1000000000111011100011000110101
Integer xi =   3633367586 and in binary, bin(xi) = 0b11011000100100001100101000100010
Integer xi =   3584582396 and in binary, bin(xi) = 0b11010101101010000110001011111100
Integer xi =   1524185041 and in binary, bin(xi) =  0b1011010110110010011011111010001
Integer xi =   3120497617 and in binary, bin(xi) = 0b10111001111111110000001111010001
Integer xi =   1384358948 and in binary, bin(xi) =  0b1010010100000111010010000100100
Integer xi =    760092626 and in binary, bin(xi) =   0b101101010011100001011111010010
Integer xi =    468632607 and in binary, bin(xi) =    0b11011111011101100010000011111
Integer xi =   3718611854 and in binary, bin(xi) = 0b11011101101001011000001110001110
Integer xi =    170312151 and in binary, bin(xi) =     0b1010001001101100000111010111
Integer xi =     93043999 and in binary, bin(xi) =      0b101100010111011110100011111
Integer xi =   1302889854 and in binary, bin(xi) =  0b1001101101010001000010101111110
Integer xi =   1822754143 and in binary, bin(xi) =  0b1101100101001010000010101011111
Integer xi =   3094198579 and in binary, bin(xi) = 0b10111000011011011011100100110011
Integer xi =    358234833 and in binary, bin(xi) =    0b10101010110100011101011010001
Integer xi =    563461773 and in binary, bin(xi) =   0b100001100101011011111010001101
Integer xi =   3236562444 and in binary, bin(xi) = 0b11000000111010100000011000001100
Integer xi =   1564295301 and in binary, bin(xi) =  0b1011101001111010100000010000101
Integer xi =   3728669490 and in binary, bin(xi) = 0b11011110001111101111101100110010
Integer xi =    323737918 and in binary, bin(xi) =    0b10011010010111101100100111110
Integer xi =   4138492073 and in binary, bin(xi) = 0b11110110101011000110000010101001
Integer xi =   2493003270 and in binary, bin(xi) = 0b10010100100110000011011000000110
```

```
Integer xi =    437271590 and in binary, bin(xi) =    0b11010000100000011110000100110
Integer xi =   2630117485 and in binary, bin(xi) = 0b10011100110001000110100001101101
Integer xi =   2097611595 and in binary, bin(xi) =  0b1111101000001110000001101001011
Integer xi =   3253545216 and in binary, bin(xi) = 0b11000001111011010010100100000000
Integer xi =   2375586543 and in binary, bin(xi) = 0b10001101100110001001001011101111
Integer xi =   2334725582 and in binary, bin(xi) = 0b10001011001010010001010111001110
Integer xi =   2516044957 and in binary, bin(xi) = 0b10010101111101111100110010011101
Integer xi =    294245058 and in binary, bin(xi) =    0b10001100010011101001011000010
Integer xi =   2177777445 and in binary, bin(xi) = 0b10000001110011100011111100100101
Integer xi =   1309507195 and in binary, bin(xi) =  0b1001110000011010111111001111011
Integer xi =   3435158184 and in binary, bin(xi) = 0b11001100110000000101101010101000
Integer xi =   1695128490 and in binary, bin(xi) =  0b1100101000010011001101110101010
Integer xi =   1098019007 and in binary, bin(xi) =  0b1000001011100100111000010111111
Integer xi =   1029996593 and in binary, bin(xi) =   0b111101011001001000000000110001
Integer xi =   3838670278 and in binary, bin(xi) = 0b11100100110011010111010111000110
Integer xi =    372127336 and in binary, bin(xi) =    0b10110001011100011011001101000
Integer xi =    262477370 and in binary, bin(xi) =     0b1111101001010001011000111010
Integer xi =   3888725742 and in binary, bin(xi) = 0b11100111110010010011111011101110
Integer xi =    138313317 and in binary, bin(xi) =     0b1000001111100111111001100101
Integer xi =    769363798 and in binary, bin(xi) =   0b101101110110111000111101010110
Integer xi =   1201876561 and in binary, bin(xi) =  0b1000111101000110010111001010001
Integer xi =   2916189143 and in binary, bin(xi) = 0b10101101110100011000001111010111
Integer xi =   4117389322 and in binary, bin(xi) = 0b11110101011010100110000000001010
Integer xi =   2323047164 and in binary, bin(xi) = 0b10001010011101101110001011111100
Integer xi =   4273429928 and in binary, bin(xi) = 0b11111110101101110101110110101000
Integer xi =   2568524443 and in binary, bin(xi) = 0b10011001000110001001001010011011
Integer xi =   2070312259 and in binary, bin(xi) =  0b1111011011001100111010101000011
Integer xi =   1761871632 and in binary, bin(xi) =  0b1101001000001000000011100010000
Integer xi =   4272177704 and in binary, bin(xi) = 0b11111110101001000100001000101000
Integer xi =   3699864390 and in binary, bin(xi) = 0b11011100100000111011100110100110
Integer xi =   1871995760 and in binary, bin(xi) =  0b1101111100101000110001101110000
Integer xi =    127339982 and in binary, bin(xi) =      0b111100101110000110111001110
Integer xi =    761864309 and in binary, bin(xi) =   0b101101011010010010000001110101
Integer xi =   3849143015 and in binary, bin(xi) = 0b11100101011011010100001011100111
Integer xi =   2018064955 and in binary, bin(xi) =  0b1111000010010010011101000111011
Integer xi =   1426734324 and in binary, bin(xi) =  0b1010101000010100011110011110100
Integer xi =   2180708922 and in binary, bin(xi) = 0b10000001111110101111101000111010
Integer xi =   1341176491 and in binary, bin(xi) =  0b1001111111100001011101010101011
Integer xi =   3763535641 and in binary, bin(xi) = 0b11100000010100101111111100011001
Integer xi =   2288304274 and in binary, bin(xi) = 0b10001000011001001100000010010010
Integer xi =   3580498047 and in binary, bin(xi) = 0b11010101011010100001000001111111
Integer xi =   2502082897 and in binary, bin(xi) = 0b10010101001000101100000101010001
Integer xi =   1837725291 and in binary, bin(xi) =  0b1101101100010010111011001101011
Integer xi =    309840173 and in binary, bin(xi) =    0b10010011101111100100100101101
Integer xi =    638977987 and in binary, bin(xi) =   0b100110000101100000011111000011
Integer xi =   1629952507 and in binary, bin(xi) =  0b1100001001001110001100111111011
Integer xi =   3562370204 and in binary, bin(xi) = 0b11010100010101010111010010011100
Integer xi =    459167486 and in binary, bin(xi) =    0b11011010111100101011011111110
```

```
Integer xi =   1159150577 and in binary, bin(xi) =  0b1000101000101110011101111110001
Integer xi =   2422421702 and in binary, bin(xi) = 0b10010000011000110011100011000110
Integer xi =   2049480456 and in binary, bin(xi) =  0b1111010001010001001011100001000
Integer xi =    955606742 and in binary, bin(xi) =   0b111000111101010110011011010110
Integer xi =   3318517220 and in binary, bin(xi) = 0b11000101110011001000110111100100
Integer xi =   1592267301 and in binary, bin(xi) =  0b1011110111010000001001000100101
Integer xi =    830866714 and in binary, bin(xi) =   0b110001100001100000010100011010
Integer xi =   2344862225 and in binary, bin(xi) = 0b10001011110000111100001000010001
Integer xi =   3702789056 and in binary, bin(xi) = 0b11011100101101000001001111000000
Integer xi =   1922196881 and in binary, bin(xi) =  0b1110010100100100110010110010001
Integer xi =   3742471018 and in binary, bin(xi) = 0b11011111000100011001001101101010
Integer xi =   1541965974 and in binary, bin(xi) =  0b1011011111010001000100010010110
Integer xi =   4142146636 and in binary, bin(xi) = 0b11110110111001000010010001001100
Integer xi =   3623344569 and in binary, bin(xi) = 0b11010111111101111101100110111001
Integer xi =    279163925 and in binary, bin(xi) =    0b10000101000111011010000010101
Integer xi =   3836602443 and in binary, bin(xi) = 0b11100100101011011110100001001011
Integer xi =    167259802 and in binary, bin(xi) =     0b1001111111000001011101001101 0
Integer xi =   1150841726 and in binary, bin(xi) =  0b1000100100110000111001101111110
Integer xi =   2482754657 and in binary, bin(xi) = 0b10010011111110111101010001100001
Integer xi =   2655828943 and in binary, bin(xi) = 0b10011110010011001011101111001111
Integer xi =   3320007058 and in binary, bin(xi) = 0b11000101111000110100100110010010
Integer xi =   3983552941 and in binary, bin(xi) = 0b11101101011100000011000110101101
Integer xi =   2617035024 and in binary, bin(xi) = 0b10011011111111001100100100010000
Integer xi =    321365567 and in binary, bin(xi) =    0b10011001001111010011000111111
Integer xi =   3536735818 and in binary, bin(xi) = 0b11010010110011100100111001001010
Integer xi =    531553946 and in binary, bin(xi) =    0b11111101011101101111010011010
Integer xi =    621116189 and in binary, bin(xi) =   0b100101000001010111101100011101
Integer xi =   1468340664 and in binary, bin(xi) =  0b1010111100001010001100110111000
Integer xi =   3372085586 and in binary, bin(xi) = 0b11001000111111011111000101010010
Integer xi =   2462640705 and in binary, bin(xi) = 0b10010010110010001110101001000001
Integer xi =   4163215166 and in binary, bin(xi) = 0b11111000001001011001111100111110
Integer xi =   4095620748 and in binary, bin(xi) = 0b11110100000111100011011010001100
Integer xi =   3144005459 and in binary, bin(xi) = 0b10111011011001011011011101010011
Integer xi =   2503935906 and in binary, bin(xi) = 0b10010101001111110000011110100010
Integer xi =   3723739935 and in binary, bin(xi) = 0b11011101111110011110000110001111 1
Integer xi =   2765663758 and in binary, bin(xi) = 0b10100100110110001010111000001110
Integer xi =   1150437271 and in binary, bin(xi) =  0b1000100100100100100011110010111
Integer xi =   4285161495 and in binary, bin(xi) = 0b11111111011010100110000000010111
Integer xi =   3181830612 and in binary, bin(xi) = 0b10111101101001101110000111010100
Integer xi =   4281399551 and in binary, bin(xi) = 0b11111111001100001111100011111111
Integer xi =   1548429246 and in binary, bin(xi) =  0b1011100010010110010011110111110
Integer xi =   3973263892 and in binary, bin(xi) = 0b11101100110100110011001000010100
Integer xi =   1197689553 and in binary, bin(xi) =  0b1000111011000110100101011010001
Integer xi =   2927586334 and in binary, bin(xi) = 0b10101110011111110110110000011110
Integer xi =   3652428993 and in binary, bin(xi) = 0b11011001101100111010010011000001
Integer xi =   1726975315 and in binary, bin(xi) =  0b1100110111011111000110101010011
Integer xi =   1410287868 and in binary, bin(xi) =  0b1010100000011110100100011111100
Integer xi =   2788568833 and in binary, bin(xi) = 0b10100110001101100010111100000001
```

```
Integer xi =   2628719687 and in binary, bin(xi) = 0b10011100101011110001010001000111
Integer xi =   1518108225 and in binary, bin(xi) =  0b1011010011111000111111001000001
Integer xi =   2051151561 and in binary, bin(xi) =  0b1111010010000100001011011001001
Integer xi =   4093100836 and in binary, bin(xi) = 0b11110011111101111100001100100100
Integer xi =   3218627959 and in binary, bin(xi) = 0b10111111101100001011101011101111
Integer xi =    866978907 and in binary, bin(xi) =   0b110011101011010000110001011011
Integer xi =      7673933 and in binary, bin(xi) =          0b11101010001100001001101
Integer xi =   1741131969 and in binary, bin(xi) =  0b1100111110001111001000011000001
Integer xi =   1023486388 and in binary, bin(xi) =   0b111101000000010010100110110100
Integer xi =    979415369 and in binary, bin(xi) =   0b111010011000001011000101001001
Integer xi =     84030971 and in binary, bin(xi) =      0b101000000010001101011111011
Integer xi =    906919976 and in binary, bin(xi) =   0b110110000011101000000000101000
Integer xi =   1259848032 and in binary, bin(xi) =  0b1001011000101111100000101100000
Integer xi =   2595271731 and in binary, bin(xi) = 0b10011010101100001011010000110011
Integer xi =   1150309581 and in binary, bin(xi) =  0b1000100100100000101010011001101
Integer xi =   2140692285 and in binary, bin(xi) =  0b1111111100110000101111100111101
Integer xi =   3164682335 and in binary, bin(xi) = 0b10111100101000010011100001011111
Integer xi =   3234552544 and in binary, bin(xi) = 0b11000000110010110101101011100000
Integer xi =   3353877610 and in binary, bin(xi) = 0b11000111111010000001110001101010
Integer xi =    461770900 and in binary, bin(xi) =    0b11011100001100001000010010100
Integer xi =   3327442941 and in binary, bin(xi) = 0b11000110010101001011111111111101
Integer xi =   1826940947 and in binary, bin(xi) =  0b1101100111001001110100000010011
Integer xi =    315178113 and in binary, bin(xi) =    0b10010110010010011110010000001
Integer xi =    590842540 and in binary, bin(xi) =   0b100011001101111000101010101100
Integer xi =   3074995021 and in binary, bin(xi) = 0b10110111010010001011001101001101
Integer xi =   2944579553 and in binary, bin(xi) = 0b10101111100000101011011111100001
Integer xi =   3344583820 and in binary, bin(xi) = 0b11000111010110100100110010001100
Integer xi =   1062760645 and in binary, bin(xi) =   0b111111010110000111000011000101
Integer xi =   2109587268 and in binary, bin(xi) =  0b1111101101111011011111101000100
Integer xi =   2795807192 and in binary, bin(xi) = 0b10100110101001001010000111011000
Integer xi =   2319186164 and in binary, bin(xi) = 0b10001010001110111111100011110100
Integer xi =    187075502 and in binary, bin(xi) =     0b1011001001101000101110101110
Integer xi =   1714589414 and in binary, bin(xi) =  0b1100110001100101000111011100110
Integer xi =   2774104052 and in binary, bin(xi) = 0b10100101010110010111011111110100
Integer xi =   3453621970 and in binary, bin(xi) = 0b11001101110110100001011011010010
Integer xi =   1598122889 and in binary, bin(xi) =  0b1011111010000010110101110001001
Integer xi =    899984745 and in binary, bin(xi) =   0b110101101001001010110101101001
Integer xi =   1570482228 and in binary, bin(xi) =  0b1011101100110111010100000110100
Integer xi =    825029584 and in binary, bin(xi) =   0b110001001011001111001111010000
Integer xi =   2777306222 and in binary, bin(xi) = 0b10100101100010100101010001101110
Integer xi =   1689646329 and in binary, bin(xi) =  0b1100100101101011111010011111001
Integer xi =     82836012 and in binary, bin(xi) =      0b100111011111111101000101100
Integer xi =   1759394550 and in binary, bin(xi) =  0b1101000110111100011101011110110
Integer xi =   2042694026 and in binary, bin(xi) =  0b1111001110000010000100110001010
Integer xi =   3652870993 and in binary, bin(xi) = 0b11011001101110100110001101010001
Integer xi =   2503834779 and in binary, bin(xi) = 0b10010101001111010111110010011011
Integer xi =     25989907 and in binary, bin(xi) =        0b1100011001001001100010011
Integer xi =   3844683156 and in binary, bin(xi) = 0b11100101001010010011010110010100
```

```
Integer xi =    5858153 and in binary, bin(xi) =          0b10110010110001101101001
Integer xi = 2960765079 and in binary, bin(xi) = 0b10110000011110011011000010010111
Integer xi = 3299532383 and in binary, bin(xi) = 0b11000100101010101101111001011111
Integer xi = 3364505341 and in binary, bin(xi) = 0b11001000100010100100011011111101
Integer xi = 3314005986 and in binary, bin(xi) = 0b11000101100000111101101111100010
Integer xi =  703364433 and in binary, bin(xi) =   0b101001111011000111110101010001
Integer xi = 2762877941 and in binary, bin(xi) = 0b10100100101011100010101111110101
Integer xi =  392848013 and in binary, bin(xi) =    0b10111011010100110001010001101
Integer xi = 2950192981 and in binary, bin(xi) = 0b10101111101100001011111101010101
Integer xi = 3546445627 and in binary, bin(xi) = 0b11010011011000100111011100111011
Integer xi = 1393147189 and in binary, bin(xi) =  0b1010011000010011011110100110101
Integer xi = 2762292792 and in binary, bin(xi) = 0b10100100101001010011111000111000
Integer xi = 3048070016 and in binary, bin(xi) = 0b10110101101011011101101110000000
Integer xi = 2420254782 and in binary, bin(xi) = 0b10010000010000100010100000111110
Integer xi = 1931256725 and in binary, bin(xi) =  0b1110011000111001010001110010101
Integer xi = 3224385554 and in binary, bin(xi) = 0b11000000001100000011100000010010
Integer xi = 3111643288 and in binary, bin(xi) = 0b10111001011101111110100010011000
Integer xi = 4265900604 and in binary, bin(xi) = 0b11111110010001000111101000111100
Integer xi = 3619397695 and in binary, bin(xi) = 0b11010111101110111010000000111111
Integer xi = 3349967630 and in binary, bin(xi) = 0b11000111101011000111001100001110
Integer xi =  752273190 and in binary, bin(xi) =   0b101100110101101100011100100110
Integer xi = 1164687372 and in binary, bin(xi) =  0b1000101011010111011100000001100
Integer xi = 4178198718 and in binary, bin(xi) = 0b11111001000010100100000010111110
Integer xi =  372698645 and in binary, bin(xi) =    0b10110001101101110111000010101
Integer xi =  985091608 and in binary, bin(xi) =   0b111010101101110100111000011000
Integer xi = 2876386582 and in binary, bin(xi) = 0b10101011011100100010110100010110
Integer xi =  216088733 and in binary, bin(xi) =     0b1100111000010100000010011101
Integer xi = 2940859240 and in binary, bin(xi) = 0b10101111010010011111001101101000
Integer xi = 3447672867 and in binary, bin(xi) = 0b11001101011111111010100000100011
Integer xi = 3261159225 and in binary, bin(xi) = 0b11000010011000010101011100111001
Integer xi = 2176508969 and in binary, bin(xi) = 0b10000001101110101110010000101001
Integer xi = 2526561005 and in binary, bin(xi) = 0b10010110100110000100001011101101
Integer xi = 1561975407 and in binary, bin(xi) =  0b1011101000110011101101001101111
Integer xi =  979546518 and in binary, bin(xi) =   0b111010011000101011000110010110
Integer xi =  943402798 and in binary, bin(xi) =   0b111000001110110010111100101110
Integer xi =  304330565 and in binary, bin(xi) =    0b10010001000111011011101000101
Integer xi =  127919861 and in binary, bin(xi) =      0b111100111111110011011110101
Integer xi =  572374036 and in binary, bin(xi) =   0b100010000111011011110000010100
Integer xi =  361724688 and in binary, bin(xi) =    0b10101100011110111101100010000
Integer xi = 2042128148 and in binary, bin(xi) =  0b1111001101110000110011100010100
Integer xi = 2835890243 and in binary, bin(xi) = 0b10101001000010000100000001000011
Integer xi = 2875732453 and in binary, bin(xi) = 0b10101011011010000011000111100101
Integer xi = 2310551627 and in binary, bin(xi) = 0b10001001101110000011100001001011
Integer xi = 1118959925 and in binary, bin(xi) =  0b1000010101100011111100100110101
Integer xi = 3602026288 and in binary, bin(xi) = 0b11010110101100101000111100110000
Integer xi = 3472508057 and in binary, bin(xi) = 0b11001110111110100100010010011001
Integer xi = 3431282832 and in binary, bin(xi) = 0b11001100100001010011100010010000
Integer xi = 3134350121 and in binary, bin(xi) = 0b10111010110100100110001100101001
```

```
Integer xi =   2010598072 and in binary, bin(xi) =  0b1110111110101110100101010111000
Integer xi =    127088347 and in binary, bin(xi) =      0b111100100110011011011011011
Integer xi =   2973107558 and in binary, bin(xi) = 0b10110001001101100000010101100110
Integer xi =   2000852815 and in binary, bin(xi) =  0b1110111010000101001011101001111
Integer xi =    320239350 and in binary, bin(xi) =    0b10011000101100111011011110110
Integer xi =    680940291 and in binary, bin(xi) =   0b101000100101100101001100000011
Integer xi =    396820841 and in binary, bin(xi) =    0b10111101001110000000101101001
Integer xi =   3679059793 and in binary, bin(xi) = 0b11011011010010011111111101010001
Integer xi =   1881628617 and in binary, bin(xi) =  0b1110000001001110101111111001001
Integer xi =    303615556 and in binary, bin(xi) =    0b10010000110001100111001000100
Integer xi =    227493472 and in binary, bin(xi) =     0b1101100011110100011001100000
Integer xi =   2862897834 and in binary, bin(xi) = 0b10101010101001000101101010101010
Integer xi =   3380125342 and in binary, bin(xi) = 0b11001001011110001001111010011110
Integer xi =   3212420406 and in binary, bin(xi) = 0b10111111011110011010010100110110
Integer xi =   3241702604 and in binary, bin(xi) = 0b11000001001110000111010011001100
Integer xi =   3497857423 and in binary, bin(xi) = 0b11010000011111010001000110001111
Integer xi =   3044902665 and in binary, bin(xi) = 0b10110101011111011000011100001001
Integer xi =   3997810242 and in binary, bin(xi) = 0b11101110010010011011111001000010
Integer xi =   2819226446 and in binary, bin(xi) = 0b10101000000010011111101101001110
Integer xi =   3836250340 and in binary, bin(xi) = 0b11100100101010001000100011100100
Integer xi =   3071902144 and in binary, bin(xi) = 0b10110111000110011000000111000000
Integer xi =   3044231609 and in binary, bin(xi) = 0b10110101011100110100100110111001
Integer xi =    522790870 and in binary, bin(xi) =    0b11111001010010010011111010110
Integer xi =   3280296618 and in binary, bin(xi) = 0b11000011100001010101101010101010
Integer xi =   1538648392 and in binary, bin(xi) =  0b1011011101101011110100101001000
Integer xi =   1451279128 and in binary, bin(xi) =  0b1010110100000001100001100011000
Integer xi =    846660538 and in binary, bin(xi) =   0b110010011101110000001110111010
Integer xi =   1932990326 and in binary, bin(xi) =  0b1110011001101110001011101110110
Integer xi =    122293128 and in binary, bin(xi) =      0b111010010100000101110001000
Integer xi =   4040211924 and in binary, bin(xi) = 0b11110000110100001011110111010100
Integer xi =   1901162832 and in binary, bin(xi) =  0b1110001010100010111000101010000
Integer xi =   1904580259 and in binary, bin(xi) =  0b1110001100001011001011010100011
Integer xi =   1277687776 and in binary, bin(xi) =  0b1001100001001111111011111100000
Integer xi =    944593750 and in binary, bin(xi) =   0b111000010011010101101101010110
Integer xi =    530550504 and in binary, bin(xi) =    0b11111100111111000111011101000
Integer xi =    721180654 and in binary, bin(xi) =   0b101010111111000101011111101110
Integer xi =   1124337506 and in binary, bin(xi) =  0b1000011000001000000011101100010
Integer xi =   1245184760 and in binary, bin(xi) =  0b1001010001110000000001011111000
Integer xi =   3109609950 and in binary, bin(xi) = 0b10111001010110001110000111011110
Integer xi =     52617178 and in binary, bin(xi) =       0b11001000101101111111011010
Integer xi =   4201242225 and in binary, bin(xi) = 0b11111010011010011101111001110001
Integer xi =   3895642903 and in binary, bin(xi) = 0b11101000001100101100101100010111
Integer xi =   1287140348 and in binary, bin(xi) =  0b1001100101110000011001111111100
Integer xi =   4138067394 and in binary, bin(xi) = 0b11110110101001011110010111000010
Integer xi =     94933038 and in binary, bin(xi) =      0b101101010001001000000101110
Integer xi =   3552894923 and in binary, bin(xi) = 0b11010011110001001101111111001011
Integer xi =    606686675 and in binary, bin(xi) =   0b100100001010010100110111010011
Integer xi =   3979810835 and in binary, bin(xi) = 0b11101101001101110001100000010011
```

```
Integer xi =   3495120745 and in binary, bin(xi) = 0b11010000010100110100111101101001
Integer xi =   2119495731 and in binary, bin(xi) =  0b1111110010101001111000000110011
Integer xi =   3907577698 and in binary, bin(xi) = 0b11101000111010001110011101100010
Integer xi =    412831626 and in binary, bin(xi) =    0b11000100110110100111110001010
Integer xi =   1325451416 and in binary, bin(xi) =  0b1001111000000001100100010011000
Integer xi =   2167073398 and in binary, bin(xi) = 0b10000001001010101110101001110110
Integer xi =   2589438056 and in binary, bin(xi) = 0b10011010010101111011000001101000
Integer xi =   3363313475 and in binary, bin(xi) = 0b11001000011110000001011101000011
Integer xi =   3035423769 and in binary, bin(xi) = 0b10110100111011001110010000011001
Integer xi =   3602940889 and in binary, bin(xi) = 0b11010110110000001000001111011001
Integer xi =   2023524015 and in binary, bin(xi) =  0b1111000100111001000011010101111
Integer xi =   1344633747 and in binary, bin(xi) =  0b1010000001001010111101110010011
Integer xi =   2974312586 and in binary, bin(xi) = 0b10110001010010000110100010001010
Integer xi =   2656513143 and in binary, bin(xi) = 0b10011110010101110010110001110111
Integer xi =   3564159070 and in binary, bin(xi) = 0b11010100011100001100000001011110
Integer xi =   1263023414 and in binary, bin(xi) =  0b1001011010010000011010100110110
Integer xi =   3626652266 and in binary, bin(xi) = 0b11011000001010100101001001101010
Integer xi =   3733065320 and in binary, bin(xi) = 0b11011110100000100000111001101000
Integer xi =    103921927 and in binary, bin(xi) =      0b110001100011011100100000111
Integer xi =   2860598176 and in binary, bin(xi) = 0b10101010100000001010000111010000
Integer xi =   1035417477 and in binary, bin(xi) =   0b111101101101110011011110000101
Integer xi =   2333161992 and in binary, bin(xi) = 0b10001011000100010011101000001000
Integer xi =   1948127030 and in binary, bin(xi) =  0b1110100000111100000111100110110
Integer xi =   1338038202 and in binary, bin(xi) =  0b1001111110000001101011110111010
Integer xi =    208778934 and in binary, bin(xi) =     0b1100011100011011011010110110
Integer xi =   2448700181 and in binary, bin(xi) = 0b10010001111101000011001100010101
Integer xi =   3997041044 and in binary, bin(xi) = 0b11101110001111110000000110010100
Integer xi =     44825507 and in binary, bin(xi) =       0b10101010111111101110100011
Integer xi =   2349939274 and in binary, bin(xi) = 0b10001100000100010011101001001010
Integer xi =   3784895872 and in binary, bin(xi) = 0b11100001100110001110110110000000
Integer xi =   2839541553 and in binary, bin(xi) = 0b10101001001111111111011100110001
Integer xi =   3037912645 and in binary, bin(xi) = 0b10110101000100101101111001000101
Integer xi =   2196314732 and in binary, bin(xi) = 0b10000010111010010001101001101100
Integer xi =    442635973 and in binary, bin(xi) =    0b11010011000100001011011000101
Integer xi =    165298014 and in binary, bin(xi) =     0b1001110110100011111101011110
Integer xi =    541044006 and in binary, bin(xi) =   0b100000001111111010110100100110
Integer xi =   2059327483 and in binary, bin(xi) =  0b1111010101111101101011111111011
Integer xi =   3790590621 and in binary, bin(xi) = 0b11100001111011111101001010011101
Integer xi =   1763299261 and in binary, bin(xi) =  0b1101001000110011100111110111101
Integer xi =   2497410676 and in binary, bin(xi) = 0b10010100110110110111011001110100
Integer xi =   2844785294 and in binary, bin(xi) = 0b10101001100011111111101010001110
Integer xi =   3021937416 and in binary, bin(xi) = 0b10110100000111110001101100001000
Integer xi =   1918582056 and in binary, bin(xi) =  0b1110010010110110011110100101000
Integer xi =   2067126237 and in binary, bin(xi) =  0b1111011001101011101011111011101
Integer xi =    532963981 and in binary, bin(xi) =    0b11111110001000110001010001101
Integer xi =   1200662532 and in binary, bin(xi) =  0b1000111100100001010100000000100
Integer xi =   3698447250 and in binary, bin(xi) = 0b11011100011100011101001110010010
Integer xi =   3794861103 and in binary, bin(xi) = 0b11100010001100001111110000101111
```

```
Integer xi =   2836275841 and in binary, bin(xi) = 0b10101001000011100010001010000001
Integer xi =   4212201754 and in binary, bin(xi) = 0b11111011000100010001100100011010
Integer xi =   2319959648 and in binary, bin(xi) = 0b10001010010001111100011001100000
Integer xi =    520349230 and in binary, bin(xi) =    0b11111000000111110011000101110
Integer xi =   2966098732 and in binary, bin(xi) = 0b10110000110010110001001100101100
Integer xi =    221723408 and in binary, bin(xi) =     0b1101001101110011101100010000
Integer xi =   2160821088 and in binary, bin(xi) = 0b10000000110010111000001101100000
Integer xi =   3109477811 and in binary, bin(xi) = 0b10111001010101101101110110110011
Integer xi =   1072896526 and in binary, bin(xi) =   0b111111111100110001101000001110
Integer xi =   1383202204 and in binary, bin(xi) =  0b1010010011100011111110110011100
Integer xi =   2891329434 and in binary, bin(xi) = 0b10101100010101100010111110011010
Integer xi =   3501245406 and in binary, bin(xi) = 0b11010000101100001100001111011110
Integer xi =   3751157658 and in binary, bin(xi) = 0b11011111100101100001111110011010
Integer xi =     69716829 and in binary, bin(xi) =      0b100001001111100101101011101
Integer xi =   1505668878 and in binary, bin(xi) =  0b1011001101111101010111100001110
Integer xi =   4098775850 and in binary, bin(xi) = 0b11110100010011100101101100101010
Integer xi =   2545738577 and in binary, bin(xi) = 0b10010111101111001110001101010001
Integer xi =   4204369634 and in binary, bin(xi) = 0b11111010100110011001011011100010
Integer xi =    765672134 and in binary, bin(xi) =   0b101101101000110011101011000110
Integer xi =    208361897 and in binary, bin(xi) =     0b1100011010110101100110101001
Integer xi =   1214090833 and in binary, bin(xi) =  0b1001000010111011000111001010001
Integer xi =   3582397093 and in binary, bin(xi) = 0b11010101100001110000101010100101
Integer xi =   1608852417 and in binary, bin(xi) =  0b1011111111001010010001111000001
Integer xi =   1208273721 and in binary, bin(xi) =  0b1001000000001001100101100111001
Integer xi =   2526085509 and in binary, bin(xi) = 0b10010110100100010000000110000101
Integer xi =   2619649789 and in binary, bin(xi) = 0b10011100001001001010111011111101
Integer xi =   2784814564 and in binary, bin(xi) = 0b10100101111111100111001011100100
Integer xi =   3055715720 and in binary, bin(xi) = 0b10110110001000101000010110001000
Integer xi =   1441792327 and in binary, bin(xi) =  0b1010101111110000000000101000111
Integer xi =   2272192152 and in binary, bin(xi) = 0b10000111011011101110011010011000
Integer xi =   3724233894 and in binary, bin(xi) = 0b11011101111110110100110010100110
Integer xi =   3369842723 and in binary, bin(xi) = 0b11001000110110111011100000100011
Integer xi =   2989772752 and in binary, bin(xi) = 0b10110010001101000100111111010000
Integer xi =   2289415600 and in binary, bin(xi) = 0b10001000011101011011010110110000
Integer xi =   3533767613 and in binary, bin(xi) = 0b11010010101000010000001110111101
Integer xi =   3490874775 and in binary, bin(xi) = 0b11010000000100101000010110010111
Integer xi =   3827963379 and in binary, bin(xi) = 0b11100100001010100001010111110011
Integer xi =   1584073120 and in binary, bin(xi) =  0b1011110011010110000100110100000
Integer xi =   2853938450 and in binary, bin(xi) = 0b10101010000110111010010100010010
Integer xi =    988337756 and in binary, bin(xi) =   0b111010111010001101011001011100
Integer xi =   3740176171 and in binary, bin(xi) = 0b11011110111011101000111100101011
Integer xi =   1193018243 and in binary, bin(xi) =  0b1000111000111000000001110000011
Integer xi =   2813578097 and in binary, bin(xi) = 0b10100111101100111100101101110001
Integer xi =   3270561066 and in binary, bin(xi) = 0b11000010111100001100110100101010
Integer xi =   3934637806 and in binary, bin(xi) = 0b11101010100001011100111011101110
Integer xi =   1391677154 and in binary, bin(xi) =  0b1010010111100110100111011100010
Integer xi =    805867164 and in binary, bin(xi) =   0b110000000010001000111010011100
Integer xi =    638376301 and in binary, bin(xi) =   0b100110000011001101100101101101
```

```
Integer xi =   832099524 and in binary, bin(xi) =   0b110001100110001101010011000100
Integer xi =   909767166 and in binary, bin(xi) =   0b110110001110011111000111111110
Integer xi =  2558294659 and in binary, bin(xi) = 0b10011000011111000111101010000011
Integer xi =  2138249556 and in binary, bin(xi) =  0b1111111011100110001100101010100
Integer xi =  3942492888 and in binary, bin(xi) = 0b11101010111111011010101011011000
Integer xi =  1210708829 and in binary, bin(xi) =  0b1001000001010011111001101011101
Integer xi =   104068362 and in binary, bin(xi) =      0b110001100111111010100001010
Integer xi =   620694524 and in binary, bin(xi) =   0b100100111111111000010111111100
Integer xi =  3540731802 and in binary, bin(xi) = 0b11010011000010110100011110011010
Integer xi =  2050859204 and in binary, bin(xi) =  0b1111010001111011010000011000100
Integer xi =  1973043272 and in binary, bin(xi) =  0b1110101100110100100000001001000
Integer xi =  2000877646 and in binary, bin(xi) =  0b1110111010000101111100001001110
Integer xi =  2273213274 and in binary, bin(xi) = 0b10000111011111110011110110101101 0
Integer xi =  1503292513 and in binary, bin(xi) =  0b1011001100110100110110001100001
Integer xi =  1645780613 and in binary, bin(xi) =  0b1100010000110001001111010000101
Integer xi =  1325892045 and in binary, bin(xi) =  0b1001111000001111000000111001101
Integer xi =  2082049544 and in binary, bin(xi) =  0b1111100000110011000111000001000
Integer xi =  1597725327 and in binary, bin(xi) =  0b1011111001110110101101010001111
Integer xi =   696729750 and in binary, bin(xi) =   0b101001100001110100000010010110
Integer xi =  3357922476 and in binary, bin(xi) = 0b11001000001001011101010010101100
Integer xi =  1791433290 and in binary, bin(xi) =  0b1101010110001110001101001001010
Integer xi =  3006378686 and in binary, bin(xi) = 0b10110011001100011011001010111110
Integer xi =   232795670 and in binary, bin(xi) =     0b1101111000000010111000010110
Integer xi =  4274222388 and in binary, bin(xi) = 0b11111110110000110111010100110100
Integer xi =  3614272321 and in binary, bin(xi) = 0b11010111011011010110101101000001
Integer xi =  2017489941 and in binary, bin(xi) =  0b1111000010000000111010000010101
Integer xi =  3638051888 and in binary, bin(xi) = 0b11011000110110000100010000110000
Integer xi =   315054746 and in binary, bin(xi) =    0b10010110001110101101010011010
Integer xi =   142045536 and in binary, bin(xi) =     0b1000011101110111000101100000
Integer xi =  4115602201 and in binary, bin(xi) = 0b11110101010011110001101100011001
Integer xi =   417078816 and in binary, bin(xi) =    0b11000110111000001111000100000
Integer xi =  1702240374 and in binary, bin(xi) =  0b1100101011101100010000001110110
Integer xi =  4219712046 and in binary, bin(xi) = 0b11111011100000111011001000101110
Integer xi =  1089219343 and in binary, bin(xi) =  0b1000000111011000010101100001111
Integer xi =  2871526145 and in binary, bin(xi) = 0b10101011001010000000001100000001
Integer xi =   845376169 and in binary, bin(xi) =   0b110010011000110110101010101001
Integer xi =  1124383960 and in binary, bin(xi) =  0b1000011000001001011110011011000
Integer xi =  2152432773 and in binary, bin(xi) = 0b10000000010010111000010010000101
Integer xi =  2345721609 and in binary, bin(xi) = 0b10001011110100001101111100001001
Integer xi =  2924788505 and in binary, bin(xi) = 0b10101110010101001011101100011001
Integer xi =  4168640085 and in binary, bin(xi) = 0b11111000011110000110011001010101
Integer xi =  4284335564 and in binary, bin(xi) = 0b11111111010111011100010111001100
Integer xi =  3232561385 and in binary, bin(xi) = 0b11000000101011001111100011101001
Integer xi =   155584805 and in binary, bin(xi) =     0b1001010001100000100100100101
Integer xi =  3570148724 and in binary, bin(xi) = 0b11010100110011000010010101110100
Integer xi =  3326938343 and in binary, bin(xi) = 0b11000110010011010000110011100111
Integer xi =  4105697636 and in binary, bin(xi) = 0b11110100101101111111100101100100
Integer xi =  2632643363 and in binary, bin(xi) = 0b10011100111010101111001100100011
```

```
Integer xi =   1666723936 and in binary, bin(xi) =  0b1100011010110000011000001100000
Integer xi =   1440099044 and in binary, bin(xi) =  0b1010101110101100010101011100100
Integer xi =    680961589 and in binary, bin(xi) =   0b101000100101101010011000110101
Integer xi =    356011767 and in binary, bin(xi) =    0b10101001110000100111011110111
Integer xi =   2378600510 and in binary, bin(xi) = 0b10001101110001101001000000111110
Integer xi =     13651577 and in binary, bin(xi) =         0b110100000100111001111001
Integer xi =   3813736373 and in binary, bin(xi) = 0b11100011010100001111111110110101
Integer xi =    609952374 and in binary, bin(xi) =   0b100100010110110010001001110110
Integer xi =    506997792 and in binary, bin(xi) =    0b11110001110000010110000100000
Integer xi =    886946253 and in binary, bin(xi) =   0b110100110111101101100111001101
Integer xi =    392862710 and in binary, bin(xi) =    0b10111011010101001101111110110
Integer xi =   2287913525 and in binary, bin(xi) = 0b10001000010111101100101000110101
Integer xi =   1119639152 and in binary, bin(xi) =  0b1000010101111000101011001110000
Integer xi =    851165101 and in binary, bin(xi) =   0b110010101110111011111110101101
Integer xi =   4134565370 and in binary, bin(xi) = 0b11110110011100000111010111111010
Integer xi =   1412858691 and in binary, bin(xi) =  0b1010100001101101000001101000011
Integer xi =     23444368 and in binary, bin(xi) =        0b1011001011011101110010000
Integer xi =       923437 and in binary, bin(xi) =             0b11100001011100101101
Integer xi =   2689526255 and in binary, bin(xi) = 0b10100000010011101110100111101111
Integer xi =   2270805862 and in binary, bin(xi) = 0b10000111010110011011111101100110
Integer xi =    155417897 and in binary, bin(xi) =      0b1001010000110111110100101001
Integer xi =   3304267744 and in binary, bin(xi) = 0b11000100111100110001111111100000
Integer xi =   3753270291 and in binary, bin(xi) = 0b11011111101101100101110000010011
Integer xi =    287608410 and in binary, bin(xi) =    0b10001001001001000111001011010
Integer xi =   2653855879 and in binary, bin(xi) = 0b10011110001011101010000010000111
Integer xi =   1092528355 and in binary, bin(xi) =  0b1000001000111101010100011100011
Integer xi =   1660551498 and in binary, bin(xi) =  0b1100010111110100000000101001010
Integer xi =   1672214369 and in binary, bin(xi) =  0b1100011101010111111011101100001
Integer xi =    994760694 and in binary, bin(xi) =   0b111011010010101101011111110110
Integer xi =   1454465055 and in binary, bin(xi) =  0b1010110101100010110000000011111
Integer xi =    180809258 and in binary, bin(xi) =     0b1010110001101110111000101010
Integer xi =    398171832 and in binary, bin(xi) =    0b10111101110111001111010111000
Integer xi =   3582972302 and in binary, bin(xi) = 0b11010101100011111101000110001110
Integer xi =   1372157519 and in binary, bin(xi) =  0b1010001110010010111011001001111
Integer xi =   1913948096 and in binary, bin(xi) =  0b1110010000101001000011111000000
Integer xi =    699148529 and in binary, bin(xi) =   0b101001101011000010100011110001
Integer xi =   2363338116 and in binary, bin(xi) = 0b10001100110111011010110110000100
Integer xi =   1915498460 and in binary, bin(xi) =  0b1110010001011000010111111011100
Integer xi =    110204858 and in binary, bin(xi) =      0b110100100011001011110111010
Integer xi =   2313071172 and in binary, bin(xi) = 0b10001001110111101010101001000100
Integer xi =   3296436850 and in binary, bin(xi) = 0b11000100011110111010001001110010
Integer xi =    844301127 and in binary, bin(xi) =   0b110010010100110000001101000111
Integer xi =   4280175860 and in binary, bin(xi) = 0b11111111000111100100110011110100
Integer xi =   1350657851 and in binary, bin(xi) =  0b1010000100000010110011100111011
Integer xi =   2464529858 and in binary, bin(xi) = 0b10010010111001011011110111000010
Integer xi =    397187298 and in binary, bin(xi) =    0b10111101011001001100011100010
Integer xi =   3211705042 and in binary, bin(xi) = 0b10111111011011101011101011010010
Integer xi =   2292170038 and in binary, bin(xi) = 0b10001000100111111011110100110110
```

```
Integer xi =   1111238565 and in binary, bin(xi) =  0b1000010001111000010011110100101
Integer xi =   1280853189 and in binary, bin(xi) =  0b1001100010110000100010011000101
Integer xi =   1779688143 and in binary, bin(xi) =  0b1101010000100111110001011001111
Integer xi =   3854704930 and in binary, bin(xi) = 0b11100101110000100010000100100010
Integer xi =   2048092710 and in binary, bin(xi) =  0b1111010000100110110101000100110
Integer xi =    370319143 and in binary, bin(xi) =    0b10110000100101001111100100111
Integer xi =   1186503081 and in binary, bin(xi) =  0b1000110101110001001100110101001
Integer xi =   2253582330 and in binary, bin(xi) = 0b10000110010100101110111111111010
Integer xi =   2894988877 and in binary, bin(xi) = 0b10101100100011100000011001001101
Integer xi =   2693059081 and in binary, bin(xi) = 0b10100000100001001101001000001001
Integer xi =   3174303371 and in binary, bin(xi) = 0b10111101001101000000011010001011
Integer xi =   1665367233 and in binary, bin(xi) =  0b1100011010000110111110011000001
Integer xi =     48241162 and in binary, bin(xi) =       0b10111000000001101000001010
Integer xi =   3315195164 and in binary, bin(xi) = 0b11000101100110011101110100011100
Integer xi =   2764409186 and in binary, bin(xi) = 0b10100100110001011000100101100010
Integer xi =    926421411 and in binary, bin(xi) =   0b110111001110000001000110100011
Integer xi =   4016649256 and in binary, bin(xi) = 0b11101111011010010011010000101000
Integer xi =    322794763 and in binary, bin(xi) =    0b10011001111010111010100001011
Integer xi =   1511800056 and in binary, bin(xi) =  0b1011010000111000011110011111000
Integer xi =   2264490811 and in binary, bin(xi) = 0b10000110111111001011000110011011
Integer xi =   2861792678 and in binary, bin(xi) = 0b10101010100100110111110110100110
Integer xi =   2339545991 and in binary, bin(xi) = 0b10001011011100101010001110000111
Integer xi =   2135460562 and in binary, bin(xi) =  0b1111111010010001000101011010010
Integer xi =   2776313230 and in binary, bin(xi) = 0b10100101011110110010110110001110
Integer xi =   3167401938 and in binary, bin(xi) = 0b10111100110010101011011111010010
Integer xi =   3268433721 and in binary, bin(xi) = 0b11000010110100000101011100111001
Integer xi =   3702129285 and in binary, bin(xi) = 0b11011100101010100000001010000101
Integer xi =   3641075716 and in binary, bin(xi) = 0b11011001000001100110100000000100
Integer xi =    860122460 and in binary, bin(xi) =   0b110011010001000110110101011100
Integer xi =    125856162 and in binary, bin(xi) =      0b111100000000110100110100010
Integer xi =   3305655026 and in binary, bin(xi) = 0b11000101000010000100101011110010
Integer xi =    165496898 and in binary, bin(xi) =     0b1001110111010100100001000010
Integer xi =   1885756789 and in binary, bin(xi) =  0b1110000011001100101110101110101
Integer xi =    500190595 and in binary, bin(xi) =    0b11101110100000100110110000011
Integer xi =   3935572826 and in binary, bin(xi) = 0b11101010100101000001001101011010
Integer xi =    894747617 and in binary, bin(xi) =   0b110101010101001100001111100001
Integer xi =   3133644978 and in binary, bin(xi) = 0b10111010110001111010000010110010
Integer xi =   1526289952 and in binary, bin(xi) =  0b1011010111110010101011000100000
Integer xi =   1168837256 and in binary, bin(xi) =  0b1000101101010110000101010001000
Integer xi =   4017278431 and in binary, bin(xi) = 0b11101111011100101100110111011111
Integer xi =   2171965999 and in binary, bin(xi) = 0b10000001011101011001001000101111
Integer xi =   1104447011 and in binary, bin(xi) =  0b1000001110101001000011000100011
Integer xi =    327894088 and in binary, bin(xi) =    0b10011100010110100010001001000
Integer xi =   3514912582 and in binary, bin(xi) = 0b11010001100000010100111101000110
Integer xi =   3018660396 and in binary, bin(xi) = 0b10110011111011010001101000101100
Integer xi =   1040193686 and in binary, bin(xi) =   0b111110000000000001100010010110
Integer xi =   3947341728 and in binary, bin(xi) = 0b11101011010001111010011110100000
Integer xi =   2948643316 and in binary, bin(xi) = 0b10101111111000000101110011110100
```

```
Integer xi =   3031481819 and in binary, bin(xi) = 0b10110100101100001011110111011011
Integer xi =   3532554225 and in binary, bin(xi) = 0b11010010100011100111111111110001
Integer xi =   3417828952 and in binary, bin(xi) = 0b11001011101101111110111001011000
Integer xi =   2276259231 and in binary, bin(xi) = 0b10000111101011001111010110011111
Integer xi =    530376740 and in binary, bin(xi) =   0b11111100111001110100000100100
Integer xi =    615559460 and in binary, bin(xi) =  0b100100101100001011000100100100
Integer xi =   3620057188 and in binary, bin(xi) = 0b11010111110001011011000001100100
Integer xi =   3694586431 and in binary, bin(xi) = 0b11011100001101101110101000111111
Integer xi =    743200431 and in binary, bin(xi) =  0b101100010011000101011010101111
Integer xi =   2776544714 and in binary, bin(xi) = 0b10100101011111101011010111001010
Integer xi =   3244992256 and in binary, bin(xi) = 0b11000001011010101010011100000000
Integer xi =   1416025503 and in binary, bin(xi) = 0b1010100011001101101010110011111
Integer xi =   2512356848 and in binary, bin(xi) = 0b10010101101111111000010111110000
Integer xi =    718405397 and in binary, bin(xi) =  0b101010110100011111111100010101
Integer xi =   1334387131 and in binary, bin(xi) = 0b1001111100010010010000110111011
Integer xi =    697147504 and in binary, bin(xi) =  0b101001100011011010000001110000
Integer xi =   1142438326 and in binary, bin(xi) = 0b1000100000110000011100110110110
Integer xi =   4252891009 and in binary, bin(xi) = 0b11111101011111101111101110000001
Integer xi =   2184797146 and in binary, bin(xi) = 0b10000010001110010101101111011010
Integer xi =   3266165526 and in binary, bin(xi) = 0b11000010101011011011101100010110
Integer xi =   2760011335 and in binary, bin(xi) = 0b10100100100000100110111001000111
Integer xi =   1978734280 and in binary, bin(xi) = 0b1110101111110001000101101100100 0
Integer xi =   2676865440 and in binary, bin(xi) = 0b10011111100011011011100110100000
Integer xi =   2614967515 and in binary, bin(xi) = 0b10011011110111010011110011011011
Integer xi =   3885167354 and in binary, bin(xi) = 0b11100111100100101111001011111010
Integer xi =   3197041332 and in binary, bin(xi) = 0b10111110100011101111101010110100
Integer xi =   1493472995 and in binary, bin(xi) = 0b1011001000001001001011011100011
Integer xi =   2077148832 and in binary, bin(xi) = 0b1111011110011101100011010100000
Integer xi =   4155640942 and in binary, bin(xi) = 0b11110111101100100000110001101110
Integer xi =   3364657610 and in binary, bin(xi) = 0b11001000100011001001100111001010
Integer xi =   2315301646 and in binary, bin(xi) = 0b10001010000000001011001100001110
Integer xi =   1351058597 and in binary, bin(xi) = 0b1010000100001111000010010100101
Integer xi =    978466409 and in binary, bin(xi) =  0b111010010100100011011001101001
Integer xi =   2915203450 and in binary, bin(xi) = 0b10101101110000100111100101111010
Integer xi =   4265774007 and in binary, bin(xi) = 0b11111110010000101000101110110111
Integer xi =   1639544022 and in binary, bin(xi) = 0b1100001101110010111010011010110
Integer xi =   1041153282 and in binary, bin(xi) =  0b111110000011101011110100000010
Integer xi =    786619918 and in binary, bin(xi) =  0b101110111000101101111000001110
Integer xi =   2471074780 and in binary, bin(xi) = 0b10010011010010011001101111011100
Integer xi =   3925072875 and in binary, bin(xi) = 0b11101001111100111101101111101011
Integer xi =   1414214583 and in binary, bin(xi) = 0b1010100010010110011001110110111
Integer xi =   1827954335 and in binary, bin(xi) = 0b1101100111101000101111010011111
Integer xi =    933906235 and in binary, bin(xi) =  0b110111101010100100011100111011
Integer xi =   3006686438 and in binary, bin(xi) = 0b10110011001101100110010011100110
Integer xi =   3631298850 and in binary, bin(xi) = 0b11011000011100010011100100100010
Integer xi =    643958142 and in binary, bin(xi) =  0b100110011000100000010101111110
Integer xi =    911311576 and in binary, bin(xi) =  0b110110010100011000001011011000
Integer xi =   2529950011 and in binary, bin(xi) = 0b10010110110010111111100100111011
```

```
Integer xi =   3405919962 and in binary, bin(xi) = 0b11001011000000100011011011011010
Integer xi =    142496341 and in binary, bin(xi) =     0b1000011111100101001001010101
Integer xi =    949522371 and in binary, bin(xi) =   0b111000100110001000111111000011
Integer xi =    908076311 and in binary, bin(xi) =   0b110110001000000010010100010111
Integer xi =   1914623417 and in binary, bin(xi) =  0b1110010000111101101010110111001
Integer xi =   1965182279 and in binary, bin(xi) =  0b1110101001000100100110101000111
Integer xi =   2192742552 and in binary, bin(xi) = 0b10000010101100101001100010011000
Integer xi =   3885699423 and in binary, bin(xi) = 0b11100111100110110001000101011111
Integer xi =   1944076290 and in binary, bin(xi) =  0b1110011111000000100000000000010
Integer xi =    541678656 and in binary, bin(xi) =   0b100000010010010101110001000000
Integer xi =   2391536087 and in binary, bin(xi) = 0b10001110100010111111000111010111
Integer xi =    316854319 and in binary, bin(xi) =    0b10010111000101101000000101111
Integer xi =     18424049 and in binary, bin(xi) =        0b1000110010010000011110001
Integer xi =   2485161495 and in binary, bin(xi) = 0b10010100001000001000111000010111
Integer xi =    633174594 and in binary, bin(xi) =   0b100101101111010111101001000010
Integer xi =   3856720693 and in binary, bin(xi) = 0b11100101111000001110001100110101
Integer xi =   1313045829 and in binary, bin(xi) =  0b1001110010000110111110101000101
Integer xi =    247694843 and in binary, bin(xi) =     0b1110110000111000010111111011
Integer xi =    110381076 and in binary, bin(xi) =      0b110100101000100100000010100
Integer xi =   1061663071 and in binary, bin(xi) =   0b111111010001111011000101011111
Integer xi =   2902807239 and in binary, bin(xi) = 0b10101101000001010101001011000111
Integer xi =   2917247060 and in binary, bin(xi) = 0b10101101111000011010100001010100
Integer xi =    675703803 and in binary, bin(xi) =   0b101000010001100110101111111011
Integer xi =   3766311495 and in binary, bin(xi) = 0b11100000011111010101101001000111
Integer xi =   2946801748 and in binary, bin(xi) = 0b10101111101001001010000001010100
Integer xi =    585112101 and in binary, bin(xi) =   0b100010111000000001101000100101
Integer xi =   2935817224 and in binary, bin(xi) = 0b10101110111111010000010000001000
Integer xi =   2142413707 and in binary, bin(xi) =  0b1111111101100101010001110001011
Integer xi =    398795855 and in binary, bin(xi) =    0b10111110001010010010001001111
Integer xi =   3755006652 and in binary, bin(xi) = 0b11011111110100001101101010111100
Integer xi =   1567043098 and in binary, bin(xi) =  0b1011101011001110010111000011010
Integer xi =    277852780 and in binary, bin(xi) =    0b10000100011111011001001101100
Integer xi =   1675396547 and in binary, bin(xi) =  0b1100011110111001000010111000011
Integer xi =   3583522452 and in binary, bin(xi) = 0b11010101100110000011011010010100
Integer xi =   4265050137 and in binary, bin(xi) = 0b11111110001101111000000000011001
Integer xi =   1174975985 and in binary, bin(xi) =  0b1000110000010001011010111110001
Integer xi =   3348363442 and in binary, bin(xi) = 0b11000111100100111111100010110010
Integer xi =   3541378862 and in binary, bin(xi) = 0b11010011000101010010011100101110
Integer xi =   2730613383 and in binary, bin(xi) = 0b10100010110000001110110101000111
Integer xi =   4240573901 and in binary, bin(xi) = 0b11111100110000100000010111001101
Integer xi =   1064098827 and in binary, bin(xi) =   0b111111011011001101110000001011
Integer xi =   3966639723 and in binary, bin(xi) = 0b11101100011011100001111001101011
Integer xi =   2775717648 and in binary, bin(xi) = 0b10100101011100100001011100010000
Integer xi =   3339443893 and in binary, bin(xi) = 0b11000111000010111101111010110101
Integer xi =   2941191519 and in binary, bin(xi) = 0b10101111010011110000010101011111
Integer xi =   1353655096 and in binary, bin(xi) =  0b1010000101011110010001100111000
Integer xi =   3254930476 and in binary, bin(xi) = 0b11000010000000100100110000101100
Integer xi =    904857047 and in binary, bin(xi) =   0b110101111011110000010111010111
```

```
Integer xi =   3907208435 and in binary, bin(xi) = 0b11101000111000110100010011110011
Integer xi =   2948098651 and in binary, bin(xi) = 0b10101111101110000110101001011011
Integer xi =   1078468758 and in binary, bin(xi) =  0b1000000010010000010000010010110
Integer xi =   3981043271 and in binary, bin(xi) = 0b11101101010010011110011001000111
Integer xi =   3907076547 and in binary, bin(xi) = 0b11101000111000010100000111000011
```

2.7.3 Implementing the Mersenne twister algorithm

Finally, the Mersenne twister can be implemented like this:

```
In [179]: class MersenneTwister(PRNG):
              """The Mersenne twister Pseudo-Random Number Generator (MRG)."""
              def __init__(self, seed=None,
                           w=32, n=624, m=397, r=31,
                           a=0x9908B0DF, b=0x9D2C5680, c=0xEFC60000,
                           u=11, s=7, v=15, l=18):
                  """Create a new Mersenne twister PRNG with this seed."""
                  self.t = 0
                  # Parameters
                  self.w = w
                  self.n = n
                  self.m = m
                  self.r = r
                  self.a = a
                  self.b = b
                  self.c = c
                  self.u = u
                  self.s = s
                  self.v = v
                  self.l = l
                  # For X
                  if seed is None:
                      seed = random_Mersenne_seed(n, w)
                  self.X0 = seed
                  self.X = np.copy(seed)
                  # Maximum integer number produced is 2**w - 1
                  self.max = (1 << w) - 1

              def __next__(self):
                  """Produce a next value and return it, following the Mersenne twister algori
                  self.t += 1
                  # 1. --- Compute x_{t+n}
                  # 1.1.a. First r bits of x_t : left = (x_t >> (w - r)) << (w - r)
                  # 1.1.b. Last w - r bits of x_{t+1} : right = x & ((1 << (w - r)) - 1)
                  # 1.1.c. Concatenate them together in a binary vector x : x = left + right
                  left = self.X[0] >> (self.w - self.r)
                  right = (self.X[1] & ((1 << (self.w - self.r)) - 1))
```

```
        x = (left << (self.w - self.r)) + right
        xw = x % 2              # 1.2. get xw
        if xw == 0:
            xtilde = (x >> 1)            # if xw = 0, xtilde = (x >> 1)
        else:
            xtilde = (x >> 1) ^ self.a   # if xw = 1, xtilde = (x >> 1)   a
        nextx = self.X[self.m] ^ xtilde  # 1.3. x_{t+n} = x_{t+m}   \tilde{x}
        # 2. --- Shift the content of the n rows
        oldx0 = self.X[0]           # 2.a. First, forget x0
        self.X[:-1] = self.X[1:]    # 2.b. shift one index on the left, x1..xn-1 to x
        self.X[-1]  = nextx         # 2.c. write new xn-1
        # 3. --- Then use it to compute the answer, y
        y = nextx                       # 3.a. y = x_{t+n}
        y ^= (y >> self.u)              # 3.b. y = y  (y >> u)
        y ^= ((y << self.s) & self.b)   # 3.c. y = y  ((y << s) & b)
        y ^= ((y << self.v) & self.c)   # 3.d. y = y  ((y << v) & c)
        y ^= (y >> self.l)              # 3.e. y = y  (y >> l)
        return y
```

2.7.4 Small review of bitwise operations

The Python documentation explains how to use bitwise operations easily, and also this page and this StackOverflow answer.

The only difficult part of the algorithm is the first step, when we need to take the first r bits of $X_t =$ `X[0]`, and the last $w - r$ bits of $X_{t+1} =$ `X[1]`. On some small examples, let quickly check that I implemented this correctly:

```
In [180]: def testsplit(x, r=None, w=None):
              if w is None:
                  w = x.bit_length()
              if r is None:
                  r = w - 1
              assert x.bit_length() == w
              left = x >> (w - r)
              right = x % 2 if w == 1 else x & ((1 << (w-r) - 1))
              x2 = (left << (w - r)) + right
              assert x == x2
              print("x = {:10} -> left r={} = {:10} and right w-r={} = {:4} -> x2 = {:10}".forn

          x = 0b10011010
          testsplit(x)
          x = 0b10010011
          testsplit(x)
          x = 0b10011111
          testsplit(x)
          x = 0b11110001
          testsplit(x)
          x = 0b00110001
```

```
testsplit(x)
```

```
x = 0b10011010 -> left r=7 = 0b1001101  and right w-r=1 = 0b0  -> x2 = 0b10011010
x = 0b10010011 -> left r=7 = 0b1001001  and right w-r=1 = 0b1  -> x2 = 0b10010011
x = 0b10011111 -> left r=7 = 0b1001111  and right w-r=1 = 0b1  -> x2 = 0b10011111
x = 0b11110001 -> left r=7 = 0b1111000  and right w-r=1 = 0b1  -> x2 = 0b11110001
x = 0b110001   -> left r=5 = 0b11000    and right w-r=1 = 0b1  -> x2 = 0b110001
```

2.7.5 Mersenne twister algorithm in cython

As for the first example, let us write a Cython function to (try to) compute the next numbers more easily.

My reference was this page of the Cython documentation.

```
In [262]: %%cython
          from __future__ import division
          import cython
          import numpy as np
          # "cimport" is used to import special compile-time information
          # about the numpy module (this is stored in a file numpy.pxd which is
          # currently part of the Cython distribution).
          cimport numpy as np

          # We now need to fix a datatype for our arrays. I've used the variable
          # DTYPE for this, which is assigned to the usual NumPy runtime
          # type info object.
          DTYPE = np.int64
          # "ctypedef" assigns a corresponding compile-time type to DTYPE_t. For
          # every type in the numpy module there's a corresponding compile-time
          # type with a _t-suffix.
          ctypedef np.int64_t DTYPE_t


          @cython.boundscheck(False) # turn off bounds-checking for entire function
          def nextMersenneTwister(np.ndarray[DTYPE_t, ndim=1] X, unsigned long w, unsigned long
              """Produce a next value and return it, following the Mersenne twister algorithm,
              assert X.dtype == DTYPE
              # 1. --- Compute x_{t+n}
              # 1.1.a. First r bits of x_t : left = (x_t >> (w - r)) << (w - r)
              # 1.1.b. Last w - r bits of x_{t+1} : right = x & ((1 << (w - r)) - 1)
              # 1.1.c. Concatenate them together in a binary vector x : x = left + right
              cdef unsigned long x = ((X[0] >> (w - r)) << (w - r)) + (X[1] & ((1 << (w - r)) -
              cdef unsigned long xtilde = 0
              if x % 2 == 0:  # 1.2. get xw
                  xtilde = (x >> 1)            # if xw = 0, xtilde = (x >> 1)
              else:
                  xtilde = (x >> 1) ^ a   # if xw = 1, xtilde = (x >> 1)  a
```

```
        cdef unsigned long nextx = X[m] ^ xtilde  # 1.3. x_{t+n} = x_{t+m}  \tilde{x}
        # 2. --- Shift the content of the n rows
        # oldx0 = X[0]            # 2.a. First, forget x0
        X[:-1] = X[1:]            # 2.b. shift one index on the left, x1..xn-1 to x0..xn
        X[-1]  = nextx            # 2.c. write new xn-1
        # 3. --- Then use it to compute the answer, y
        cdef unsigned long y = nextx        # 3.a. y = x_{t+n}
        y ^= (y >> u)             # 3.b. y = y  (y >> u)
        y ^= ((y << s) & b)       # 3.c. y = y  ((y << s) & b)
        y ^= ((y << v) & c)       # 3.d. y = y  ((y << v) & c)
        y ^= (y >> l)             # 3.e. y = y  (y >> l)
        return y
```

In [263]:
```
nextMersenneTwister
nextMersenneTwister?
```

Out[263]: `<function _cython_magic_254860e6a9a04121ac2ee81a80b68b1a.nextMersenneTwister>`

```
Docstring: Produce a next value and return it, following the Mersenne twister algorithm, implem
Type:      builtin_function_or_method
```

That should be enough to define a Cython version of our `MersenneTwister` class.

In [264]:
```
class CythonMersenneTwister(MersenneTwister):
    """The Mersenne twister Pseudo-Random Number Generator (MRG), accelerated with Cy

    def __next__(self):
        """Produce a next value and return it, following the Mersenne twister algori
        self.t += 1
        return nextMersenneTwister(self.X, self.w, self.m, self.r, self.a, self.u, se
```

2.7.6 Testing our implementations

In [265]:
```
ForthExample = MersenneTwister(seed=example_seed)
CythonForthExample = CythonMersenneTwister(seed=example_seed)
```

In [266]:
```
ForthExample.int_samples((10,))
CythonForthExample.int_samples((10,))
```

Out[266]:
```
array([ 306676890, 2556225439, 1979801398,   27614268, 1499672499,
        1289742183, 3706951411, 3441754698, 3090564280, 1104599484])
```

Out[266]:
```
array([ 306676890, 2556225439, 1979801398,   27614268, 1499672499,
        1289742183, 3706951411, 3441754698, 3090564280, 1104599484])
```

Which one is the quickest?

In [267]:
```
%timeit [ ForthExample.randint() for _ in range(100000) ]
%timeit [ CythonForthExample.randint() for _ in range(100000) ]
```

```
1 loop, best of 3: 402 ms per loop
10 loops, best of 3: 199 ms per loop
```

Using Cython gives only a speedup of $2\times$, that's disappointing!

```
In [187]: %prun [ ForthExample.randint() for _ in range(1000000) ]
```

```
         2000004 function calls in 6.187 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000000    5.304    0.000    5.304    0.000 <ipython-input-179-740d1415e4c6>:29(__next__)
        1    0.503    0.503    6.179    6.179 <string>:1(<listcomp>)
  1000000    0.371    0.000    5.675    0.000 <ipython-input-132-93560edf79a4>:28(randint)
        1    0.008    0.008    6.187    6.187 <string>:1(<module>)
        1    0.000    0.000    6.187    6.187 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
In [188]: %prun [ CythonForthExample.randint() for _ in range(1000000) ]
```

```
         3000004 function calls in 2.831 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000000    1.425    0.000    1.425    0.000 {_cython_magic_55c22c8ddb4d3f9b7a2c470dde3236f5.
  1000000    0.757    0.000    2.182    0.000 <ipython-input-183-7bbfd5fb3182>:4(__next__)
        1    0.397    0.397    2.823    2.823 <string>:1(<listcomp>)
  1000000    0.244    0.000    2.426    0.000 <ipython-input-132-93560edf79a4>:28(randint)
        1    0.008    0.008    2.831    2.831 <string>:1(<module>)
        1    0.000    0.000    2.831    2.831 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

$\Longrightarrow$ the Cython version is twice as fast as the pure-Python version. We can still improve this, I am sure.

We can again check for the mean and the variance of the generated sequence. Mean should be $\frac{1}{2} = 0.5$ and variance should be $\frac{(b-a)^2}{12} = \frac{1}{12} = 0.08333\ldots$:

```
In [189]: shape = (400, 400)
          image = ForthExample.float_samples(shape)
          np.mean(image), np.var(image)
```

```
Out[189]: (0.49923488385706588, 0.083302180051650965)
```

This Python hand-written Mersenne twister is of course slower than the previous PRNG defined above (combined Multiple Recursive Generator, simple Multiple Recursive Generator, and the simple Linear Recurrent Generator):

```
In [190]: %timeit SecondExample.float_samples(shape)
          %timeit ThirdExample.float_samples(shape)
          %timeit MRG32k3a.float_samples(shape)
          %timeit ForthExample.float_samples(shape)
```

```
1 loop, best of 3: 224 ms per loop
1 loop, best of 3: 895 ms per loop
1 loop, best of 3: 1.28 s per loop
1 loop, best of 3: 1.19 s per loop
```

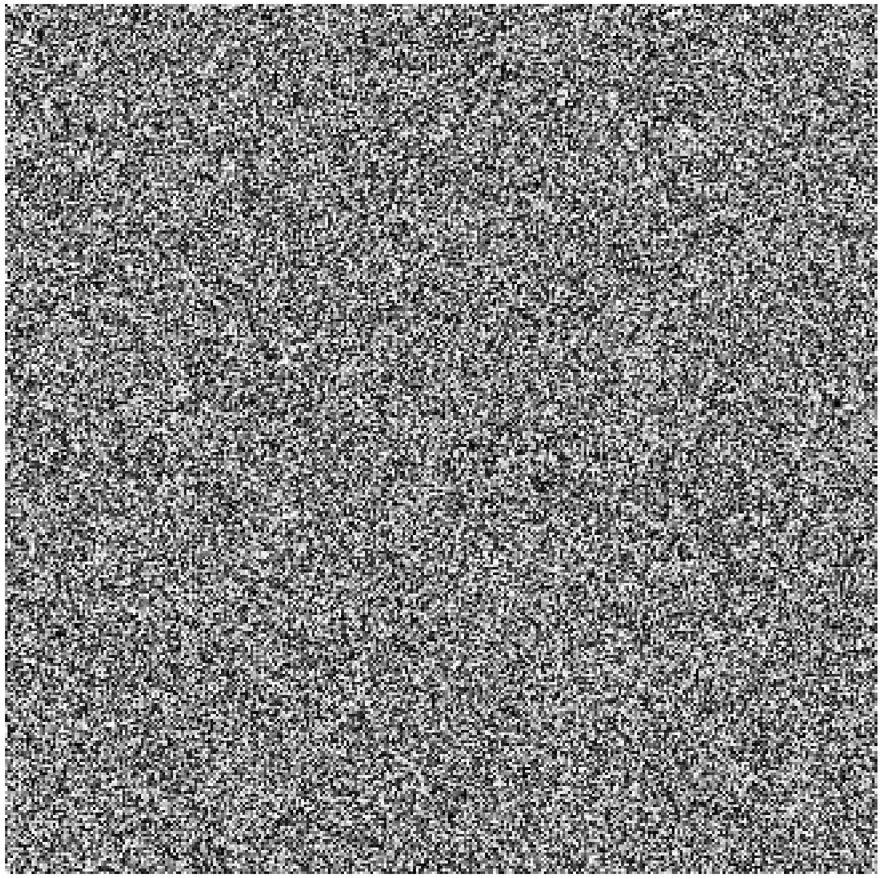
That's not too bad, for $400 \times 400 = 160000$ samples, but obviously it is incredibly slower than the optimized PRNG found in the `numpy.random` package.

```
In [191]: %timeit numpy.random.random_sample(shape)
```

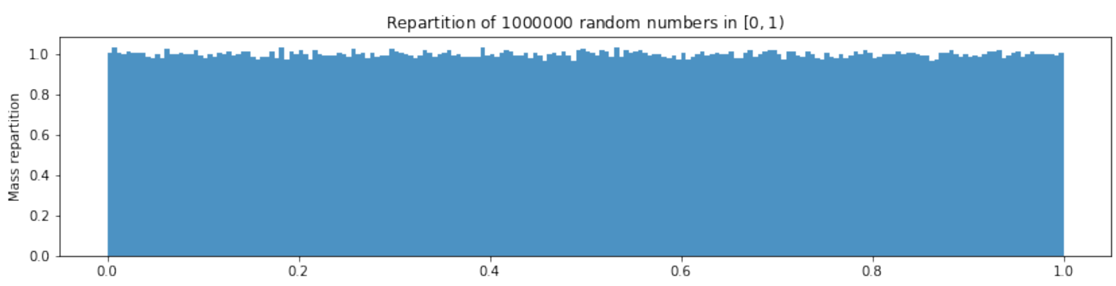
```
1000 loops, best of 3: 1.52 ms per loop
```

A good surprise is that this implementation Mersenne appears faster than the combined MRG of order $k = 3$ (i.e., `MRG32k3a`).

```
In [192]: showimage(image)
```



```
In [193]: plotHistogram(ForthExample, 1000000, 200)
```



2.8 Conclusion

Well, that's it, I just wanted to implement a few Pseudo-Random Number Generators, and compare them.

I should finish the job: - implement a test for "randomness", and check the various PRNG I implemented against it, - use these various `rand()` functions (uniform in $[0,1)$) to generate other distributions.

---

3 Generating samples from other distributions

So far, I implemented some PRNG, which essentially give a function `rand()` to produce float number uniformly sampled from $[0,1)$.

Let use it to generate samples from other distributions.

```
In [194]: def newrand():
              """Create a new random function rand()."""
              mersenne = MersenneTwister()
              rand = mersenne.rand
              return rand

          rand = newrand()
```

We will need an easy way to visualize the repartition of samples for the distributions defined below.

```
In [195]: def plotHistogramOfDistribution(distr, nb=10000, bins=200):
              numbers = [ distr() for _ in range(nb) ]
              plt.figure(figsize=(14, 3))
              plt.hist(numbers, bins=bins, normed=True, alpha=0.8)
              plt.xlabel("Random numbers from function %s" % distr.__name__)
              plt.ylabel("Mass repartition")
              plt.title("Repartition of ${}$ random numbers".format(nb))
              plt.show()
```

3.1 Bernoulli distribution

It is the simplest example, $X \in \{0,1\}$, $\mathbb{P}(X=0)=p$ and $\mathbb{P}(X=1)=1-p$ for some parameter $p \in [0,1]$.

```
In [196]: def bernoulli(p=0.5):
              """Get one random sample X ~ Bern(p)."""
              assert 0 <= p <= 1, "Error: the parameter p for a bernoulli distribution has to b
              return int(rand() < p)
```

In [197]:
```
print([ bernoulli(0.5) for _ in range(20) ])
print([ bernoulli(0.1) for _ in range(20) ])  # lots of 0
print([ bernoulli(0.9) for _ in range(20) ])  # lots of 1
```

```
[1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

We can quickly check that the frequency of 1 in a large sample of size n will converge to p as $n \to +\infty$:

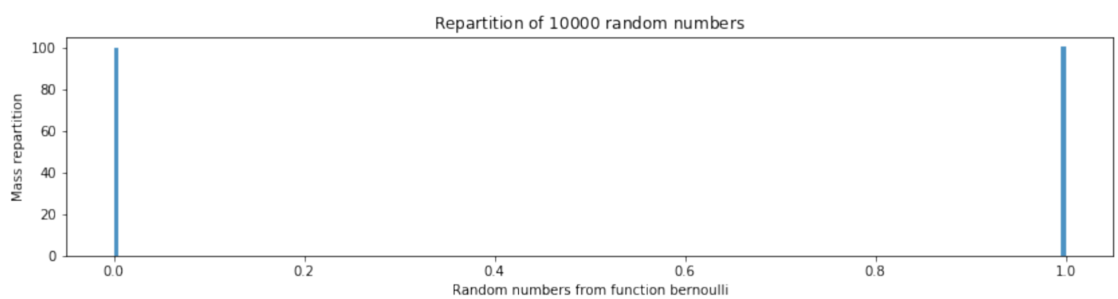
In [198]:
```
def delta_p_phat_bernoulli(p, nb=100000):
    samples = [ bernoulli(p) for _ in range(nb) ]
    return np.abs(np.mean(samples) - p)
```

In [199]:
```
print(delta_p_phat_bernoulli(0.5))
print(delta_p_phat_bernoulli(0.1))
print(delta_p_phat_bernoulli(0.9))
```

```
0.00089
0.00241
0.0017
```

That's less than 1% of absolute error, alright.

In [200]:
```
plotHistogramOfDistribution(bernoulli)
```

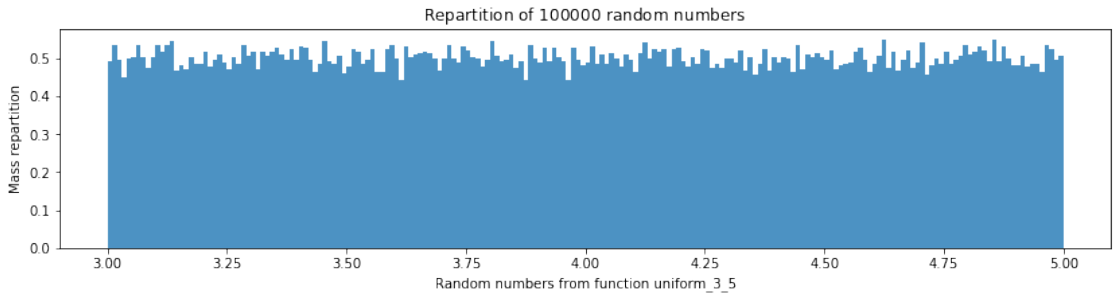


3.2 Uniform distribution on $[a, b)$, for floats and integers

This one is obvious too:

In [201]:
```
def uniform(a, b):
    """Uniform float number in [a, b)."""
    assert a < b, "Error: for uniform(a, b), a must be < b."
    return a + (b - a) * rand()
```

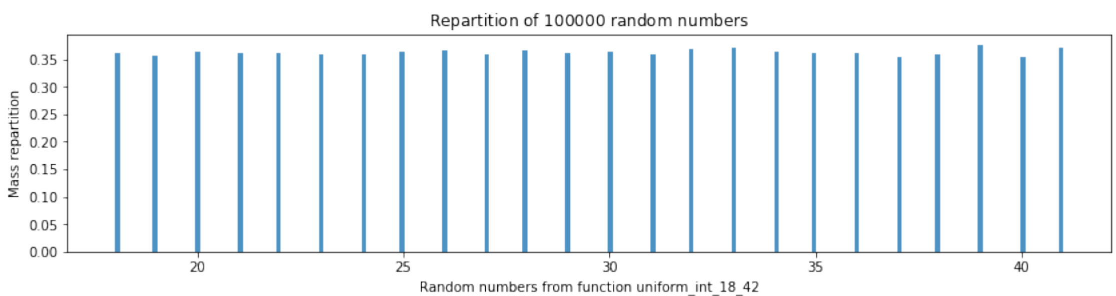
```
In [202]: def uniform_3_5():
              return uniform(3, 5)

          plotHistogramOfDistribution(uniform_3_5, 100000)
```



For integers, it is extremely similar:

```
In [203]: def randint(a, b):
              """Uniform float number in [a, b)."""
              assert a < b, "Error: for randint(a, b), a must be < b."
              assert isinstance(a, int), "Error: for randint(a, b), a must be an integer."
              assert isinstance(b, int), "Error: for randint(a, b), a must be an integer."
              return int(a + (b - a) * rand())
```

```
In [204]: def uniform_int_18_42():
              return randint(18, 42)

          plotHistogramOfDistribution(uniform_int_18_42, 100000)
```



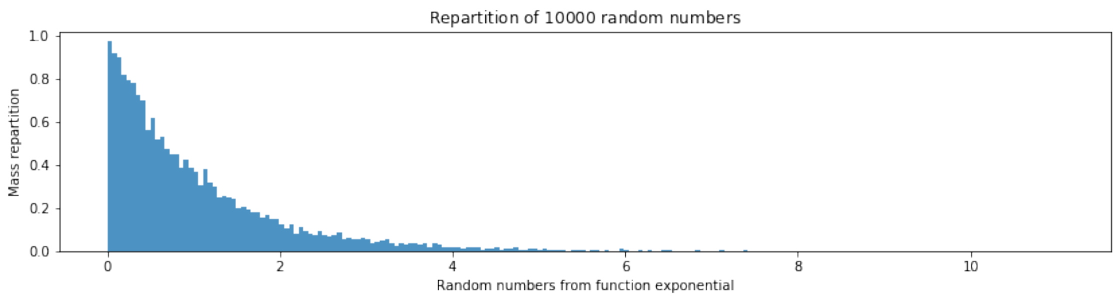
3.3 Exponential distribution

If $X \sim \mathrm{Exp}(\lambda)$, $F(x) = 1 - \mathrm{e}^{-\lambda x}$, and so $F^{-1}(u) = -\frac{1}{\lambda}\ln(1-u)$. The inversion method is easy to apply here:

```
In [205]: from math import log

          def exponential(lmbda=1):
              """Get one random sample X ~ Exp(lmbda)."""
              assert lmbda > 0, "Error: the parameter lmbda for exponential(lmbda) must be > 0
              u = rand()  # 1 - u ~ U([0, 1]), so u and 1 - u follow the same distribution
              return -(1.0 / lmbda) * log(u)
```

The resulting histogram has the shape we know as "exponential":

```
In [206]: plotHistogramOfDistribution(exponential)
```



We can compare its efficiency with `numpy.random.exponential()`, and of course it is slower.

```
In [207]: %timeit [ exponential(1.) for _ in range(10000) ]
          %timeit [ np.random.exponential(1.) for _ in range(10000) ]  # about 50 times slower

10 loops, best of 3: 86.3 ms per loop
100 loops, best of 3: 16.3 ms per loop
```

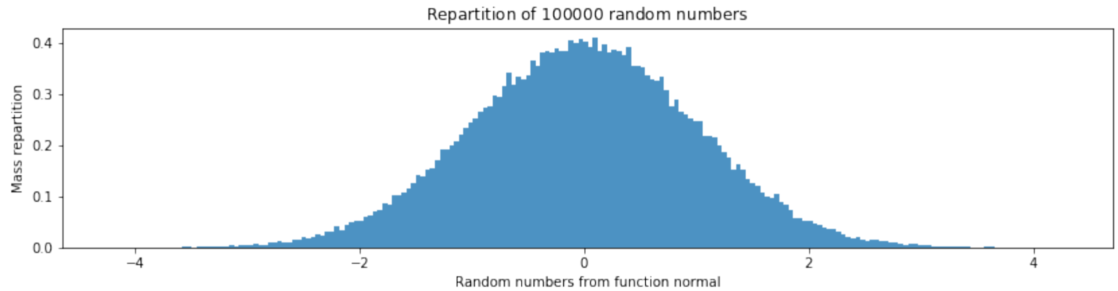
3.4 Gaussian distribution (normal)

By using the Box-Muller approach, if $U_1, U_2 \sim U(0,1)$ are independent, then setting $X = \sqrt{-2\ln U_1}\cos(2\pi U_2)$ and $Y = \sqrt{-2\ln U_1}\sin(2\pi U_2)$ leads to $X, Y \sim N(0,1)$.

Then $Z = \mu + \sigma * X$ will be distributed according to the Gaussian distribution of *mean* μ and *variance* $\sigma > 0$: $Z \sim N(\mu, \sigma)$.

```
In [208]: from math import sqrt, cos, pi

          def normal(mu=0, sigma=1):
              """Get one random sample X ~ N(mu, sigma)."""
              assert sigma > 0, "Error: the parameter sigma for normal(mu, sigma) must be > 0.
              u1, u2 = rand(), rand()
              x = sqrt(- 2 * log(u1)) * cos(2 * pi * u2)
              return mu + sigma * x
```

```
In [209]: plotHistogramOfDistribution(normal, 100000)
```



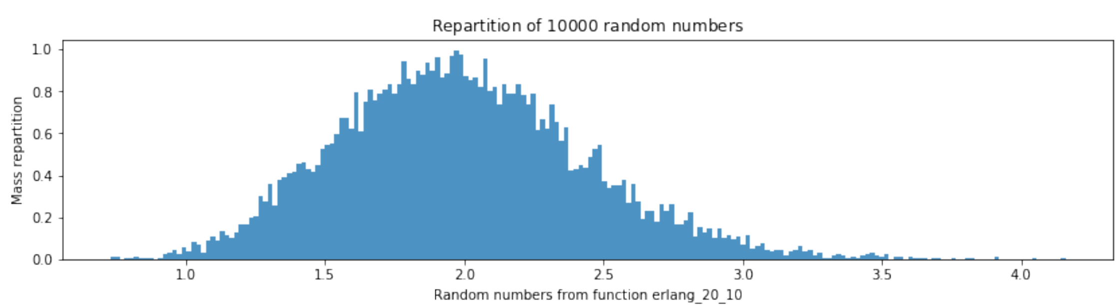
We can compare its efficiency with `numpy.random.normal()`, and of course it is slower.

```
In [210]: %timeit [ normal(0, 1) for _ in range(10000) ]
          %timeit np.random.normal(0, 1, 10000)  # 550 times quicker! oh boy!

1 loop, best of 3: 166 ms per loop
1000 loops, best of 3: 490 ţs per loop
```

3.5 Erlang distribution

If $X \sim \mathrm{Erl}(m, \lambda)$, then it simply is the sum of $m \in \mathbb{N}^*$ *iid* exponential random variables $Y_i \sim \mathrm{Exp}(\lambda)$.

```
In [211]: def erlang(m=1., lmbda=1.):
              """Get one random sample X ~ Erl(m, lmbda)."""
              assert m > 0, "Error: the parameter m for erlang(m, lmbda) must be > 0."
              assert lmbda > 0, "Error: the parameter lmbda for erlang(m, lmbda) must be > 0."
              return - 1. / lmbda * sum(log(rand()) for _ in range(int(m)) )

In [212]: def erlang_20_10():
              return erlang(20, 10)

          plotHistogramOfDistribution(erlang_20_10)
```

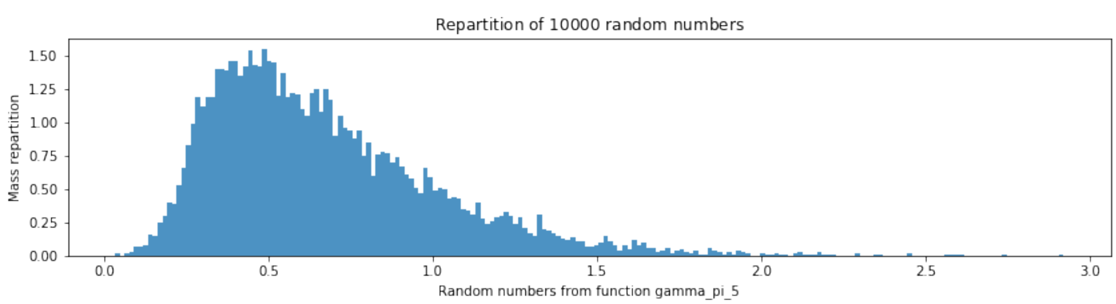


3.6 Gamma distribution

The algorithm is more complicated. The pdf of $X \sim \text{Gamma}(\alpha, \lambda)$ is $f(x) = x^{\alpha-1}\lambda^{\alpha}\mathrm{e}^{-\lambda x}/\Gamma(\alpha)$, for parameters $\alpha > 0, \lambda > 0$.

```
In [213]: def gamma(alpha=1., lmbda=1.):
              """Get one random sample X ~ Gamma(alpha, lmbda)."""
              assert alpha > 0, "Error: the parameter alpha for gamma(alpha, lmbda) must be > 0
              assert lmbda > 0, "Error: the parameter lmbda for gamma(alpha, lmbda) must be > 0
              if alpha <= 1:
                  x = gamma(alpha + 1., lmbda)
                  u = rand()
                  return x * (u ** (1. / alpha))
              else:
                  d = alpha - (1. / 3.)
                  oneByC = sqrt(9. * d)
                  c = 1. / oneByC
                  while True:
                      z = normal(0, 1)
                      if z > - oneByC:
                          v = (1. + c * z)**3
                          u = rand()
                          if log(u) < (.5 * (z**2)) + d*(v + log(v)):
                              break
                  return d * v / lmbda
```

```
In [214]: def gamma_pi_5():
              return gamma(pi, 5)

          plotHistogramOfDistribution(gamma_pi_5)
```



We can compare its efficiency with `numpy.random.gamma()`, and of course it is slower.

```
In [215]: %timeit [ gamma(pi, 5) for _ in range(10000) ]
          %timeit [ np.random.gamma(pi, 5) for _ in range(10000) ]  # 500 times quicker! oh boy
```

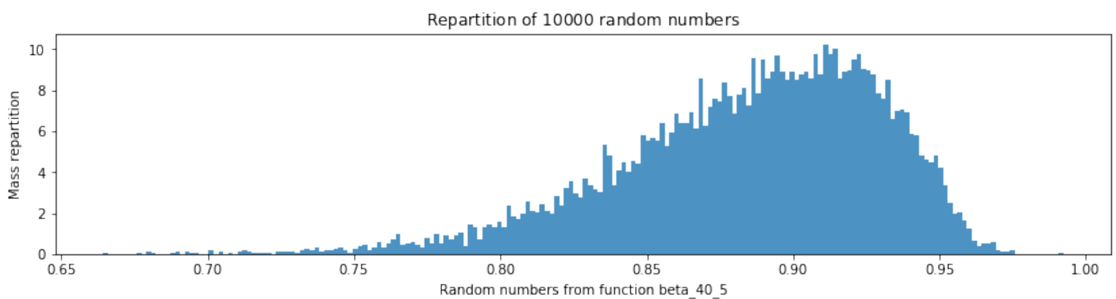
```
1 loop, best of 3: 285 ms per loop
10 loops, best of 3: 26.4 ms per loop
```

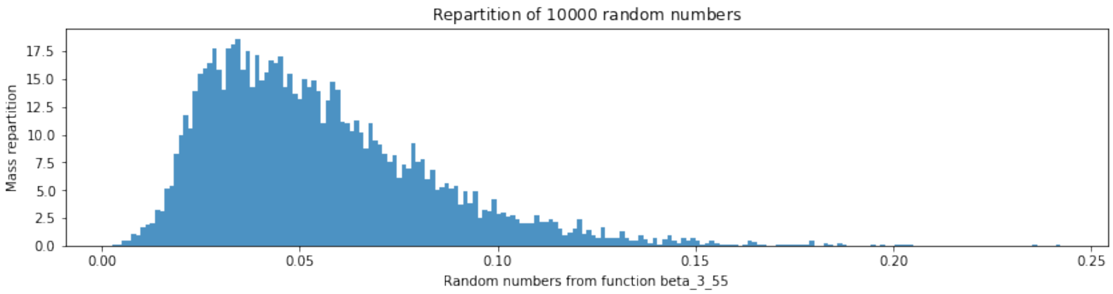
3.7 Beta distribution

By definition, a Beta distribution is straightforward to obtain as soon as we have a Gamma distribution: if $Y_1 \sim \mathrm{Gamma}(\alpha, 1)$ and $Y_2 \sim \mathrm{Gamma}(\beta, 1)$, then $X = \frac{Y_1}{Y_1+Y_2}$ follows $\mathrm{Beta}(\alpha, \beta)$.

```
In [216]: def beta(a=1., b=1.):
              """Get one random sample X ~ Beta(a, b)."""
              assert a > 0, "Error: the parameter a for beta(a, b) must be > 0."
              assert b > 0, "Error: the parameter b for beta(a, b) must be > 0."
              y1 = gamma(a, 1.)
              y2 = gamma(b, 1.)
              return y1 / float(y1 + y2)
```

```
In [217]: def beta_40_5():
              return beta(40, 5)

          plotHistogramOfDistribution(beta_40_5)

          def beta_3_55():
              return beta(3, 55)

          plotHistogramOfDistribution(beta_3_55)
```





We can compare its efficiency with `numpy.random.beta()`, and of course it is slower.

```
In [218]: %timeit [ beta(pi, 5*pi) for _ in range(1000) ]
          %timeit [ beta(5*pi, pi) for _ in range(1000) ]
          %timeit [ np.random.beta(pi, 5*pi) for _ in range(1000) ]  # 200 times quicker! oh bo
          %timeit [ np.random.beta(5*pi, pi) for _ in range(1000) ]  # 200 times quicker! oh bo
```

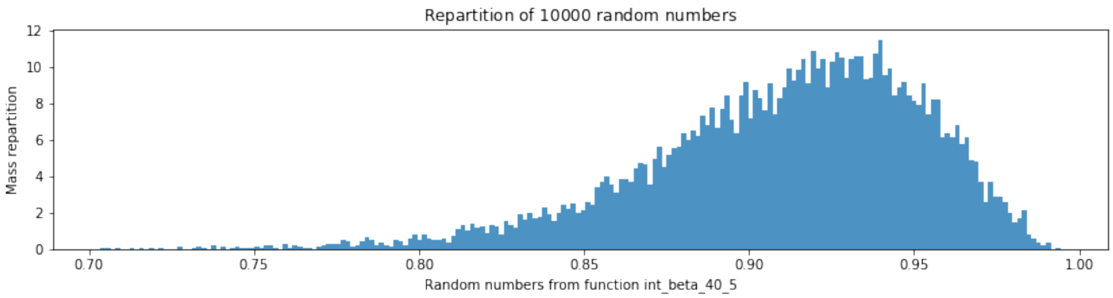
```
10 loops, best of 3: 68.2 ms per loop
10 loops, best of 3: 52.5 ms per loop
1000 loops, best of 3: 1.14 ms per loop
1000 loops, best of 3: 1.06 ms per loop
```

3.8 Integer Beta distribution

If $\alpha = m, \beta = n$ are integer, it is much simpler:

```
In [219]: def int_beta(m=1, n=1):
              """Get one random sample X ~ Beta(m, n) with integer parameters m, n."""
              assert m > 0, "Error: the parameter m for int_beta(m, n) must be > 0."
              assert n > 0, "Error: the parameter n for int_beta(m, n) must be > 0."
              us = [rand() for _ in range(m + n - 1)]
              return sorted(us)[m]  # inefficient to sort, but quick to write!
```

```
In [220]: def int_beta_40_5():
              return int_beta(40, 5)

          plotHistogramOfDistribution(int_beta_40_5)
```



We can again compare its efficiency with `numpy.random.beta()`, and of course it is slower, but this integer-specific implementation `int_beta()` is quicker than the generic `beta()` implementation.

```
In [221]: %timeit [ int_beta(40, 5) for _ in range(1000) ]
          %timeit [ int_beta(3, 55) for _ in range(1000) ]
          %timeit [ np.random.beta(40, 5) for _ in range(1000) ]  # 1500 times quicker! oh boy
          %timeit [ np.random.beta(3, 55) for _ in range(1000) ]  # 2000 times quicker! oh boy
```

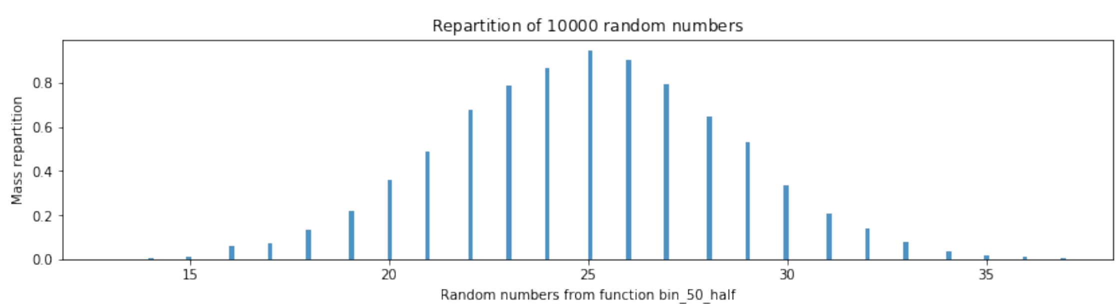
```
1 loop, best of 3: 340 ms per loop
1 loop, best of 3: 424 ms per loop
1000 loops, best of 3: 1.15 ms per loop
1000 loops, best of 3: 1.13 ms per loop
```

3.9 Binomial distribution

Very easy to obtain, by definition, from the sum of n Bernoulli distribution: if $Y_1, \dots, Y_n \sim \mathrm{Bern}(p)$, then $X = \sum_{i=1}^{n} Y_i \sim \mathrm{Bin}(n, p)$.

```
In [222]: def binomial(n=1, p=0.5):
              """Get one random sample X ~ Bin(n, p)."""
              assert 0 <= p <= 1, "Error: the parameter p for binomial(n, p) has to be in [0, 
              assert n > 0, "Error: the parameter n for binomial(n, p)  has to be in [0, 1]."
              return sum(bernoulli(p) for _ in range(n))
```

```
In [223]: def bin_50_half():
              return binomial(50, 0.5)

          plotHistogramOfDistribution(bin_50_half)
```



It is an integer distribution, meaning that $X \sim \mathrm{Bin}(n, p)$ always is $X \in \mathbb{N}$.

We can compare its efficiency with `numpy.random.binomial()`, and of course it is slower.

```
In [224]: %timeit [ binomial(10, 1. / pi) for _ in range(1000) ]
          %timeit [ np.random.binomial(10, 1. / pi) for _ in range(1000) ]  # 100 times quicker
```

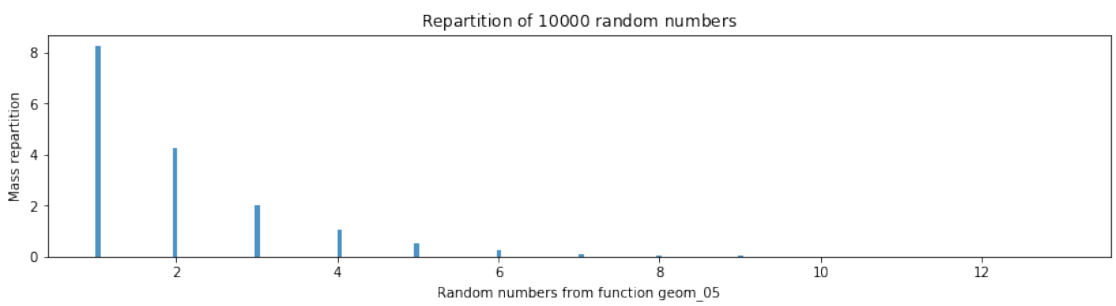
```
10 loops, best of 3: 99.5 ms per loop
1000 loops, best of 3: 1.86 ms per loop
```

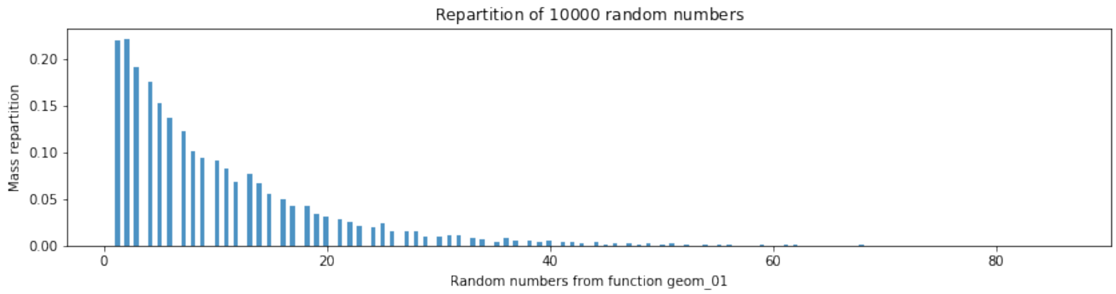
3.10 Geometric distribution

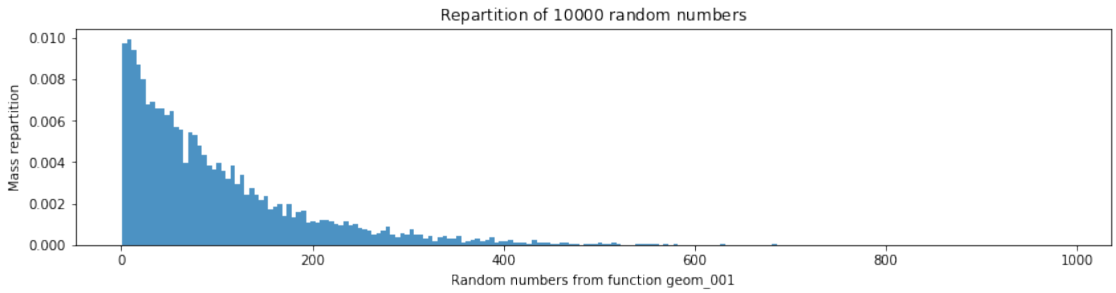
Again, it is very easy from the definition of a Geometric random variable.

```
In [225]: def geometric(p=0.5):
              """Get one random sample X ~ Geom(p)."""
              assert 0 <= p <= 1, "Error: the parameter p for binomial(n, p) has to be in [0, 1]."
              y = exponential(- log(1. - p))
              return 1 + int(y)
```

```
In [226]: def geom_05():
              return geometric(0.5)

          plotHistogramOfDistribution(geom_05)

          def geom_01():
              return geometric(0.1)

          plotHistogramOfDistribution(geom_01)

          def geom_001():
              return geometric(0.01)

          plotHistogramOfDistribution(geom_001)
```







We can compare its efficiency with `numpy.random.geometric()`, and of course it is slower.

```
In [227]: %timeit [ geometric(1. / pi) for _ in range(10000) ]
          %timeit [ np.random.geometric(1. / pi) for _ in range(10000) ]  # 50 times quicker, r
```

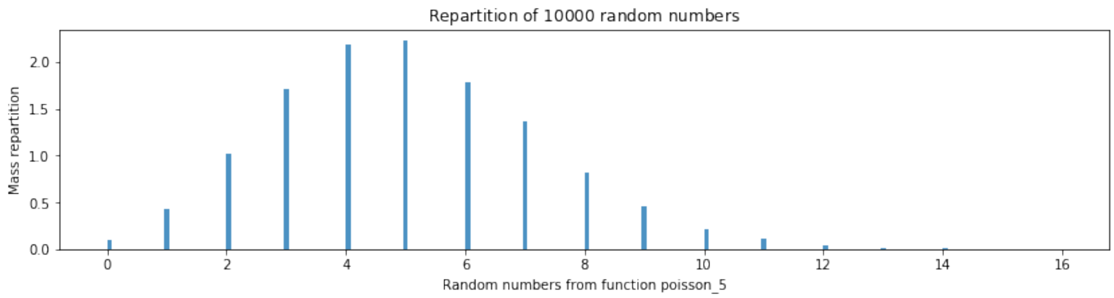
```
10 loops, best of 3: 87 ms per loop
100 loops, best of 3: 7.87 ms per loop
```

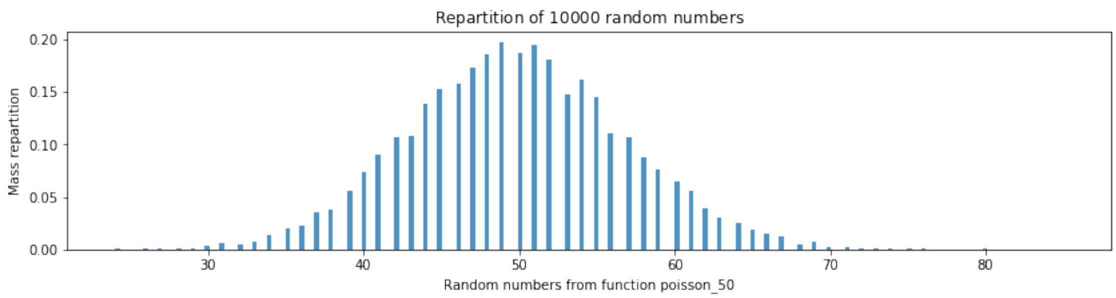
3.11 Poisson distribution

If $X \sim \mathrm{Pois}(\lambda)$, then its pdf is $f(n) = \frac{e^{-\lambda}\lambda^n}{n!}$. With the rejection method, and the close relationship between the Exponential and the Poisson distributions, it is not too hard to generate samples from a Poisson distribution if we know how to generate samples from a Exponential distribution.

```
In [228]: def poisson(lmbda=1.):
              """Get one random sample X ~ Poisson(lmbda)."""
              assert lmbda > 0, "Error: the parameter lmbda for poisson(lmbda) has to be > 0."
              n = 0
              a = 1
              while a >= exp(-lmbda):
                  u = rand()
                  a *= u
                  n += 1
              return n - 1
```

```
In [229]: def poisson_5():
              return poisson(5.)

          plotHistogramOfDistribution(poisson_5)

          def poisson_50():
              return poisson(50.)

          plotHistogramOfDistribution(poisson_50)
```





We can compare its efficiency with `numpy.random.poisson()`, and of course it is slower.

```
In [230]: %timeit [ poisson(12 * pi) for _ in range(1000) ]
          %timeit [ np.random.poisson(12 * pi) for _ in range(1000) ]  # 1000 times quicker! ol

1 loop, best of 3: 280 ms per loop
1000 loops, best of 3: 994 ţs per loop
```

3.12 Conclusion

Except the Gamma distribution, the algorithms presented above are easy to understand and to implement, and it was quick to obtain a dozen of the most common distributions, both continous (Exponential, Gaussian, Gamma, Beta) and discrete (Bernoulli, Binomial, Geometric, Poisson).

---

4 Generating vectors

Now that we have a nice Pseudo-Random Number Generator, using Mersenne twister, and that we have demonstrated how to use its `rand()` function to produce samples from the most common distributions, we can continue and explain how to produce vectors of samples.

For instance, one would need a `choice()` function to get a random sample from a list of n values, following any discrete distribution, or a `shuffle()` function to randomly shuffle a list.

4.1 Discrete distribution

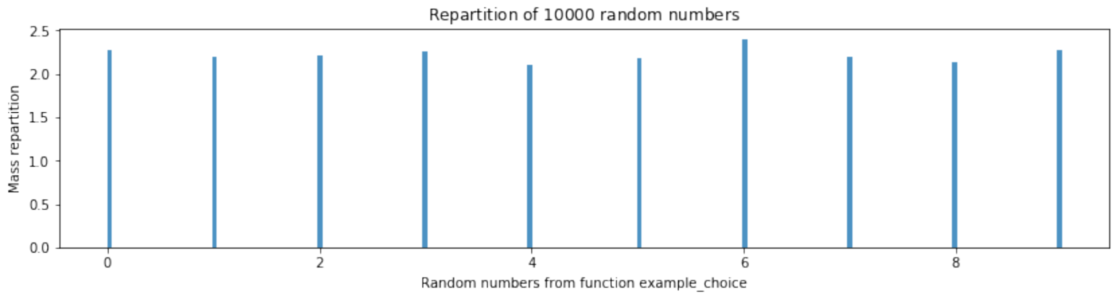
Let $p = [p_1, \dots, p_n] \in \mathbb{R}^n$ be a discrete distribution, meaning that $p_i > 0$ and $\sum_{i=1}^n p_i = 1$, then we can use the inverse-transform method to get a sample $i \in \{1, \dots, n\}$ with probability $\mathbb{P}(i = j) = p_j$.

```
In [231]: def discrete(p):
              """Return a random index i in [0..n-1] from the discrete distribution p = [p0,..
              n = len(p)
              assert n > 0, "Error: the distribution p for discrete(p) must not be empty!"
              assert all(0 <= pi <= 1 for pi in p), "Error: all coordinates of the distributio
              assert abs(sum(p) - 1) < 1e-9, "Error: the distribution p for discrete(p) does no
              u = rand()
              i = 0
              s = p[0]
              while i < n-1 and u > s:
                  i += 1
                  s += p[i]
              return i
```

Then it is easy to get *one* random sample from a list of values:

```
In [232]: def one_choice(values, p=None):
              """Get a random sample from the values, from the dsicrete distribution p = [p0,.
              if p is None:
                  return values[randint(0, len(values))]
              else:
                  return values[discrete(p)]
```

```
In [233]: def example_choice():
              return one_choice(range(10))

          plotHistogramOfDistribution(example_choice)
```



And it is also easy to generate many samples, with replacement.

```
In [234]: def choices_with_replacement(values, m=1, p=None):
              """Get m random sample from the values, with replacement, from the discrete dist
              if p is None:
                  return [ values[randint(0, len(values))] for _ in range(m) ]
              else:
                  return [ values[discrete(p)] for _ in range(m) ]
```

It is harder to handle the case without replacements. My approach is simple but slow: once a value is drawn, remove it from the input list, and update the discrete distribution accordingly. To be sure of not modifying the input list, I use `copy.copy()` to copy them.

```
In [235]: from copy import copy

          def choices_without_replacement(values, m=1, p=None):
              """Get m random sample from the values, without replacement, from the discrete d
              values = copy(values)
              if p is None:
                  samples = []
                  for _ in range(m):
                      i = randint(0, len(values))
                      samples.append(values[i])
                      del values[i]
              else:
                  p = copy(p)
                  samples = []
                  for _ in range(m):
                      i = discrete(p)
                      samples.append(values[i])
                      del values[i]
                      del p[i]
                      renormalize_cst = float(sum(p))
                      p = [ pi / renormalize_cst for pi in p ]
              return samples
```

```
In [236]: values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
          p = [0.5, 0.1, 0.1, 0.1, 0.1, 0.02, 0.02, 0.02, 0.02, 0.02]
```

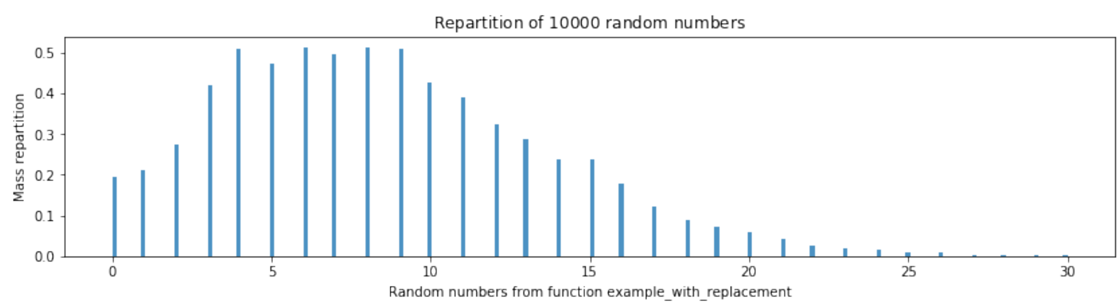
We can check that the input lists are not modified:

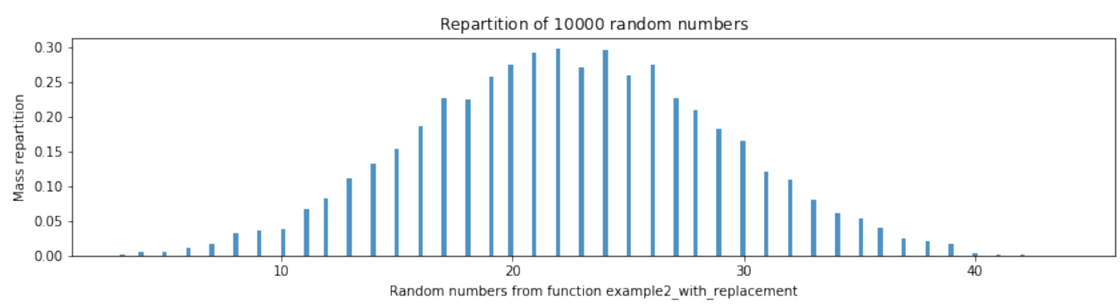
```
In [237]: print(choices_without_replacement(values, 5))
          print(choices_without_replacement(values, 5))
          print(choices_without_replacement(values, 5))

          print(choices_without_replacement(values, 5, p))
          print(choices_without_replacement(values, 5, p))
          print(choices_without_replacement(values, 5, p))
```

```
[7, 9, 1, 3, 2]
[0, 6, 5, 8, 1]
[1, 0, 9, 2, 6]
[0, 3, 4, 6, 1]
[0, 4, 3, 1, 2]
[0, 2, 1, 9, 3]
```

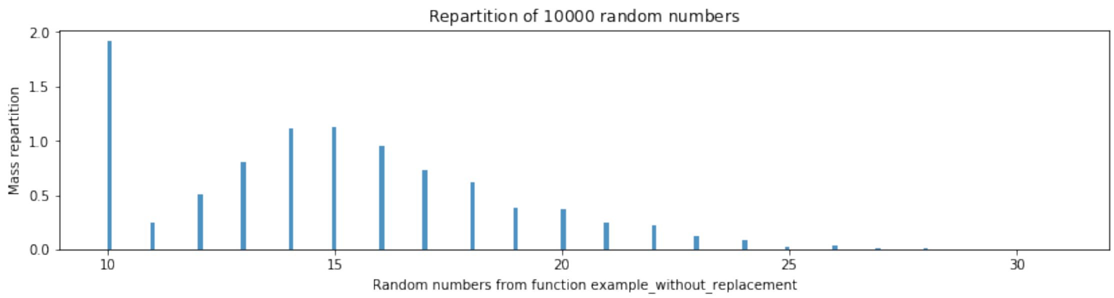
With an histogram, we can check that as a large weight is put on $0 =$ `values[0]`, the *sum* of $m = 5$ samples will be smaller if replacements are allowed (more chance to get twice a 0 value).

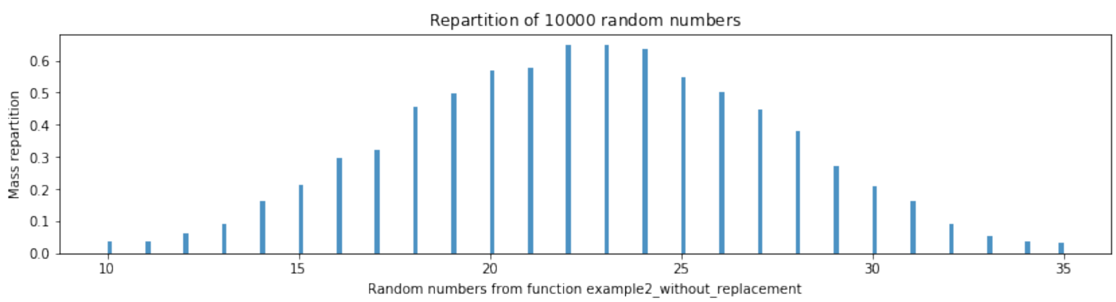
```
In [238]: def example_with_replacement():
              return np.sum(choices_with_replacement(values, 5, p))
          plotHistogramOfDistribution(example_with_replacement)

          def example2_with_replacement():
              return np.sum(choices_with_replacement(values, 5))
          plotHistogramOfDistribution(example2_with_replacement)
```





```
In [239]: def example_without_replacement():
              # this sum is at least >= 10 = 0 + 1 + 2 + 3 + 4 (5 smallest values)
              return np.sum(choices_without_replacement(values, 5, p))
          plotHistogramOfDistribution(example_without_replacement)

          def example2_without_replacement():
              # this sum is at least >= 10 = 0 + 1 + 2 + 3 + 4 (5 smallest values)
              return np.sum(choices_without_replacement(values, 5))
          plotHistogramOfDistribution(example2_without_replacement)
```





4.2 Generating a random vector uniformly on a n-dimensional ball

The acceptance-rejection method is easy to apply in this case. We use `uniform(-1, 1)` n times to get a random vector in $[0,1]^n$, and keep trying as long as it is not in the n-dim ball.

```
In [240]: def on_a_ball(n=1, R=1):
              """Generate a vector of dimension n, uniformly from the n-dim ball of radius R."
              rsquare = float('inf')
              Rsquare = R**2
```

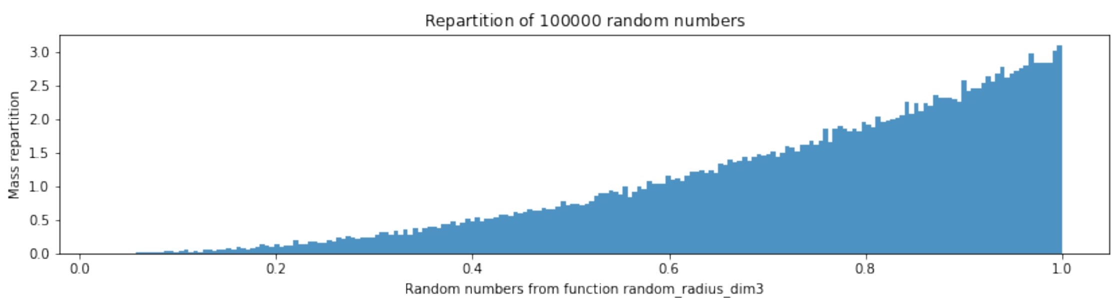
```
        while rsquare > Rsquare:
            values = [ uniform(-R, R) for _ in range(n) ]
            rsquare = sum(xi ** 2 for xi in values)
        return values
```

```
In [241]: print(on_a_ball(4, 1))
```

```
[0.047306195832788944, -0.77072893315926194, 0.37342280521988869, -0.42842319747433066]
```

The radius of such a vector can be plotted in a histogram.

```
In [242]: def random_radius_dim3():
              return sqrt(sum(xi**2 for xi in on_a_ball(3, 1)))

          plotHistogramOfDistribution(random_radius_dim3, 100000)
```

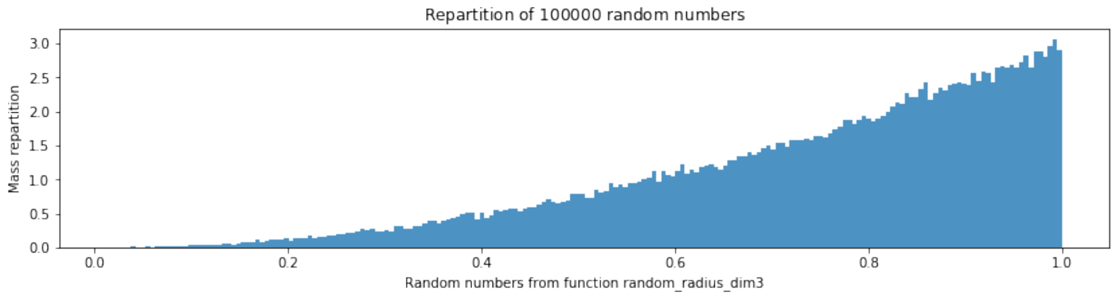


And similarly, if we normalize the values before returning them, to move them to the surface of the n-dimensional ball, then we get an easy way to sample a uniform *direction*:

```
In [243]: def on_a_sphere(n=1, R=1):
              """Generate a vector of dimension n, uniformly on the surface of the n-dim ball
              rsquare = float('inf')
              Rsquare = R**2
              while rsquare > Rsquare:
                  values = [ uniform(-1, 1) for _ in range(n) ]
                  rsquare = sum(xi ** 2 for xi in values)
              r = sqrt(rsquare)
              return [ xi / r for xi in values ]
```

All such samples have the same radius, but it can be interesting the see the smallest gap between two coordinates.

```
In [244]: def random_delta_dim3():
              return np.min(np.diff(sorted(on_a_sphere(3, 1))))

          plotHistogramOfDistribution(random_radius_dim3, 100000)
```



4.3 Generating a random permutation

The first approach is simple to write and understand, and it uses `choices_without_replacement([0..n-1], n)` with a uniform distribution p.

```
In [245]: def random_permutation(n=1):
              """Random permutation of [0..n-1], with the function choices_without_replacement
              return choices_without_replacement(list(range(n)), n)
```

```
In [246]: for _ in range(10):
              print(random_permutation(10))

[1, 4, 2, 8, 3, 7, 9, 6, 5, 0]
[9, 6, 0, 2, 1, 3, 8, 7, 5, 4]
[5, 1, 0, 6, 7, 9, 2, 3, 8, 4]
[1, 6, 4, 0, 5, 8, 2, 7, 3, 9]
[5, 4, 0, 2, 3, 6, 9, 8, 1, 7]
[0, 7, 6, 9, 1, 2, 3, 8, 5, 4]
[9, 7, 8, 4, 3, 0, 1, 2, 5, 6]
[4, 2, 0, 9, 1, 3, 7, 6, 5, 8]
[5, 9, 7, 8, 4, 2, 3, 1, 0, 6]
[0, 9, 7, 8, 2, 1, 5, 3, 4, 6]
```

It seems random enough!

To check this first implementation, we can implement the stupidest sorting algorithm, the "shuffle sort": shuffle the input list, as long as it is not correctly sorted.

```
In [247]: def is_sorted(values, debug=False):
              """Check if the values are sorted in increasing order, worst case is O(n)."""
              n = len(values)
              if n <= 1:
                  return True
              xn, xnext = values[0], values[1]
```

```
        for i in range(1, n + 1):
            if xn > xnext:
                if debug:
                    print("Values x[{}] = {} > x[{}+1] = {} are not in the good order!".
                return False
            if i >= n:
                return True
            xn, xnext = xnext, values[i]
        return True

    print(is_sorted([1, 2, 3, 4], debug=True))
    print(is_sorted([1, 2, 3, 4, 0], debug=True))
    print(is_sorted([1, 2, 5, 4], debug=True))
    print(is_sorted([1, 6, 3, 4], debug=True))
```

```
True
Values x[4] = 5 > x[5+1] = 0 are not in the good order!
False
Values x[5] = 4 > x[4+1] = 4 are not in the good order!
False
Values x[6] = 3 > x[3+1] = 3 are not in the good order!
False
```

We can easily apply a permutation, and return a shuffled version of a list of values.

```
In [248]: def apply_perm(values, perm):
              """Apply the permutation perm to the values."""
              return [values[pi] for pi in perm]

          def shuffled(values):
              """Return a random permutation of the values."""
              return apply_perm(values, random_permutation(len(values)))
```

Similarly, it is easy to shuffle in place a list of values.

```
In [249]: def shuffle(values):
              """Apply in place a random permutation of the values."""
              perm = random_permutation(len(values))
              v = copy(values)
              for (i, pi) in enumerate(perm):
                  values[i] = v[pi]
```

```
In [250]: def shuffle_sort(values):
              """Can you think of a more stupid sorting algorithm? or a shorter one?"""
              values = copy(values)
              while not is_sorted(values):
                  print(values)
                  shuffle(values)
              return values  # modified in place but also returned
```

In [251]: shuffle_sort([2, 1])

```
[2, 1]
```

Out[251]: [1, 2]

It is a **very** inefficient algorithm, but the fact that it works on small lists is enough to confirm that our algorithm to generate random permutations works fine.

In [252]:
```python
print(shuffle_sort(shuffled(list(range(3)))))
print(shuffle_sort(shuffled(list(range(4)))))
```

```
[0, 2, 1]
[0, 2, 1]
[2, 1, 0]
[1, 0, 2]
[0, 1, 2]
[1, 2, 0, 3]
[2, 3, 0, 1]
[0, 1, 2, 3]
```

We can think of another algorithm to generate a random permutation: - take n values $u_1, \ldots, u_n \sim U(0,1)$, - order them, - return the index of the sort.

In [253]:
```python
def random_permutation_2(n=1):
    """Random permutation of [0..n-1], by sorting n uniform values in [0,1]."""
    return list(np.argsort([rand() for _ in range(n)]))
```

In [254]:
```python
for _ in range(10):
    print(random_permutation_2(10))
```

```
[3, 8, 2, 1, 5, 6, 7, 0, 4, 9]
[7, 4, 1, 0, 5, 8, 2, 9, 3, 6]
[5, 7, 9, 6, 0, 2, 4, 1, 8, 3]
[3, 7, 1, 9, 6, 4, 5, 0, 2, 8]
[8, 7, 1, 5, 9, 4, 0, 3, 6, 2]
[2, 4, 3, 6, 8, 1, 0, 9, 5, 7]
[6, 5, 1, 8, 3, 4, 0, 2, 9, 7]
[3, 6, 2, 8, 0, 7, 4, 9, 5, 1]
[9, 4, 0, 1, 5, 6, 3, 8, 7, 2]
[0, 1, 6, 7, 9, 2, 8, 4, 5, 3]
```

It seems random enough too!
Let compare which of the two algorithms is the fastest:

In [255]:
```python
%timeit random_permutation(100)
%timeit random_permutation_2(100)
```

```
1000 loops, best of 3: 888 ţs per loop
1000 loops, best of 3: 754 ţs per loop
```

```
In [256]: %timeit random_permutation(10000)
          %timeit random_permutation_2(10000)
```

```
10 loops, best of 3: 94.5 ms per loop
10 loops, best of 3: 83.5 ms per loop
```

It seems that the first algorithm is slower, but this comes from the naively-written `choice_without_replacement()`, in fact we can implement it more efficiently.

```
In [257]: def random_permutation_3(n=1):
              """Random permutation of [0..n-1], with a smart implementation of choices_withou
              p = list(range(n))
              values = []
              for i in range(n):
                  j = randint(0, n - i)
                  values.append(p[j])
                  p[i], p[j] = p[j], p[i]
              return values
```

```
In [258]: for _ in range(10):
              print(random_permutation_3(10))
```

```
[0, 8, 0, 5, 3, 3, 2, 6, 7, 1]
[3, 7, 6, 4, 4, 3, 6, 2, 7, 5]
[5, 4, 3, 1, 4, 2, 5, 0, 6, 8]
[0, 1, 6, 4, 6, 3, 5, 0, 1, 7]
[1, 6, 3, 4, 3, 4, 5, 2, 6, 1]
[7, 5, 3, 6, 6, 7, 1, 2, 5, 0]
[3, 2, 7, 6, 7, 2, 5, 4, 3, 8]
[2, 0, 2, 0, 4, 3, 5, 6, 7, 1]
[6, 0, 2, 4, 2, 3, 4, 0, 7, 6]
[1, 1, 1, 1, 5, 1, 3, 0, 2, 7]
```

```
In [259]: %timeit random_permutation(1000)
          %timeit random_permutation_2(1000)
          %timeit random_permutation_3(1000)
          %timeit numpy.random.permutation(1000)

          %timeit random_permutation(10000)
          %timeit random_permutation_2(10000)
          %timeit random_permutation_3(10000)
          %timeit numpy.random.permutation(10000)  # About 1000 times slower! Oh boy!!
```

```
100 loops, best of 3: 9.28 ms per loop
100 loops, best of 3: 7.43 ms per loop
100 loops, best of 3: 9.32 ms per loop
The slowest run took 4.46 times longer than the fastest. This could mean that an intermediate r
10000 loops, best of 3: 23.3 ţs per loop
10 loops, best of 3: 95.7 ms per loop
10 loops, best of 3: 72.1 ms per loop
10 loops, best of 3: 87 ms per loop
10000 loops, best of 3: 173 ţs per loop
```

Hoho, not so sure on small lists... But for larger values of n, the second implementation of the first algorithm wins:

```
In [260]: %timeit random_permutation(100000)
          %timeit random_permutation_2(100000)
          %timeit random_permutation_3(100000)
          %timeit numpy.random.permutation(100000)  # About 1000 times slower! Oh boy!!
```

```
1 loop, best of 3: 1.62 s per loop
1 loop, best of 3: 749 ms per loop
1 loop, best of 3: 983 ms per loop
100 loops, best of 3: 2.03 ms per loop
```

And the second algorithm wins, as it uses the optimized `numpy.argsort()` function as its core operator.

Conclusion

This last part presented how to generate from any discrete distribution, and then two algorithms to generate a random permutation, uniformly sampled from Σ_n (set of $n!$ permutations of $\{0, \ldots, n-1\}$).

That's it for today, folks!