An Overview of The .NET System.Random ‘Pseudo-Pseudo-RNG’

By Martin Rupp
SCIENTIFIC AND COMPUTER DEVELOPMENT SCD LTD

Any developer has probably already needed at least once to call a random function during the development of a program or a library. Most programming languages possess their own random generators.

Here we will study the `System.Random.Rand` RNG and how it behaves in terms of randomness quality.

There are randomness tests such as DieHard or DieHarder. We do not wish to use them to check the randomness properties of the RNGs of the aforementioned programming languages. Instead we shall make some studies on our own.

Notions of entropy

Entropy in the context of randomness measures the frequency of occurrence of characters, e.g “Shannon’s Entropy”.

If we use an alphabet with N symbols - say $U_1, ..., U_N$ then the Shannon entropy $H(X)$ of a “word” X is:

$$H(X) = -\sum_{i=0}^{p-1} p_i Log_2(p_i)$$

Where p_i is the probability of appearance of the symbol U_i.

Here we will compute p_i by its frequency, e.g $p_i = \frac{S_i}{S}$ and S_i is the amount of occurences of the symbol U_i while S is the total amount of symbols in the word.

We also will consider that $0 * Log_2(0) = 0.$

The Shannon Entropy is a positive number which may be used to measure the randomness of a word. A maximal value for the entropy means that the word has "best" randomness.

$$H(X) =- \frac{1}{S}\sum_{i=0}^{p-1} S_i Log_2(S_i/S)$$

The function $f(x):\ x \to -\ xLog_2(x)$ is concave , therefore we have the following inequality:

$$\frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \le f(\frac{1}{N}\sum_{i=0}^{N-1} x_i)$$

Or:

$$-\frac{1}{p}\sum_{i=0}^{p-1}(S_i/S)Log_2(S_i/S) \le -\ (\frac{1}{p}\sum_{i=0}^{p-1} S_i/S) * Log_2(\frac{1}{p}\sum_{i=0}^{p-1} S_i/S)$$

Which leads to:

$$H(x) \le Log_2(p)$$

This means that $Log_2(p)$is the maximal value for the entropy of a word with p symbols.

As an example , if we consider an alphabet with three letters ‘A’, ‘B’ and ‘C”, we have the following values of Shannon entropy:

Word X	Shannon Entropy H(X)
ABAABBBC	$-(3Log_2(3/8) + 4Log_2(4/8) + Log_2(1/8))/8 \simeq + 1.4056$
AAAAAAAA	$-8Log_2(8/8)/8 = 0$
AAAAAAAC	$-7(Log_2(7/8)/8 + Log_2(1/8)) =+ \ 0.5435$

We will use the Shannon entropy to check the randomness property of the studied RNGs.Usually we shall consider the bytes as “words” created from an alphabet with 256 values ranging from 0x00 to 0XFF.

C# possess several random generation functions. The primary one is located in the `System.Random` class. Others are provided by the `System.Security.Cryptography.RNGCryptoServiceProvider` class or the `System.Security.Cryptography.RandomNumberGenerator.` class.

Here we will focus on the first one since the other ones are considered as a “secure” RNG and supposingly have (very) good randomness values.

Bruteforcing

The `System.Random.Rand` class uses an `Int32` value as a seed. If the seed is broken then of course the random generation is broken and generated numbers will be known in advance. The amount of possible value for the seed is 2^{32}, which means it is possible to bruteforce the RNG. All that is needed is to generate all possible seeds and search the corresponding value in the table.

The RNG generates Int32 integers through the `Next()` function. If we store three samples of the generator, we need to create a table of size 32*3*2^{32}bits. This is around 412 Gbytes.

The time needed to generate a random number with a seed is small, 1,000,000 generations are done in 6177 msecs on a slow Thinkpad machine equipped with a Celeron CPU 1007U 1.50 GHz. So the whole time needed to generate the 2^{32}possible seeds is $t = (2^{32}/10^{6}) * 6.177\ sec = 26530\ sec$. That is to say approximately 7 hours.

```
Stopwatch sw = new Stopwatch();

           sw.Start();

           //Test of the Rand function
           for (int i = 0; i < 1000000; i++)
           {
               Random r = new Random(i);
               r.Next();
           }

      Console.Out.WriteLine("Time elapsed:"+sw.ElapsedMilliseconds);
```

The default built-in C# random generator isn’t obviously secure at all and can be easily broken.

Constant values for some generated numbers

There are strange patterns in the RNG, for instance the third random number generated will always be ‘84’ in certain conditions

```
for (int i = 0; i < 30; ++i)
            {
                int s1 = i ;

                var rnd_seed = new Random(s1);

                var s2 = rnd_seed.Next();


                var rnd = new Random(s2);

                var out1 = rnd.Next(200);
                var out2 = rnd.Next(200);
                var out3 = rnd.Next(200);
                var out4 = rnd.Next(200);
                Console.WriteLine(out1+"\t|"+out2 + "\t|" + out3 +
"\t|" + out4);
            }
```

The output of the above program is the following:

172	136	84	151
58	129	84	189
144	122	84	28
30	115	84	66
116	108	84	104
2	102	84	142
88	95	84	180
174	88	84	18
60	81	84	56
146	74	84	94
31	67	84	133
117	60	84	171
3	53	84	9
89	46	84	47
175	40	84	85
61	33	84	123
147	26	84	161
33	19	84	199
119	12	84	38
5	5	84	76
91	198	84	114
177	191	84	152
63	184	84	190

In fact this is even worse as the third number is “almost” always the same for the numbers generated by `Next(Lim)`.

The following program shows this behavior:

```
for (int j = 0; j < 300; j++)
            {

                bool f = true;
                var out_=0;

                for (int i = 0; i < 30; ++i)
                {
                    int s1 = i;

                    var rnd_seed = new Random(s1);
```

```
                var s2 = rnd_seed.Next();


                var rnd = new Random(s2);

                var out1 = rnd.Next(j);
                var out2 = rnd.Next(j);
                var out3 = rnd.Next(j);
                var out4 = rnd.Next(j);

                if (f == true)
                {
                    out_ = out3;
                    f = false;
                }

                if (out_ != out3)
                {
                    Console.WriteLine("seed="+j+ "  XXX");
                    break;
                }



            }
            Console.WriteLine("seed=" + j + " out3=" + out_);
        }



    }
```

The output of that code shows how deeply flawed the RNG is.There is an obvious relation between the third 'random' number generated and the seed…

Here it shows a relation between the third output of $rnd(rnd(i).next()).Next(j)$ and j where $rnd(rnd(i).next())$ is an instance of Rand generated by a seed equal to

$rnd(i).Next()$ where i runs from 0 to 29, given the fact that this third output is a common value to all the 29 values of the generator i.

seed	out3
0	0
1	0
2	0
3	1
4	1
5	2
6	2
7	2
8	3
9	3
10	4
11	4
12	5
13	5
14	5
15	6
16	6
17	7
18	7
19	8
20	8
21	8
22	9
23	9
24	10
25	10
26	11
27	11
28	11
29	12

seed	out3
30	12
31	13
32	13
33	XXX
33	14
34	14
35	14
36	15
37	15
38	16
39	16
40	16
41	17
42	17
43	18
44	18
45	19
46	19
47	19
48	20
49	20
50	21
51	21
52	22
53	22
54	22
55	23
56	23
57	24
58	24
59	XXX

seed	out3
60	25
61	25
62	26
63	26
64	27
65	27
66	XXX
66	28
67	28
68	28
69	29
70	29
71	30
72	30
73	30
74	31
75	31
76	32
77	32
78	33
79	33
80	33
81	34
82	34
83	35
84	35
85	36
86	36
87	36
88	37
89	37

seed	out3	seed	out3	seed	out3
120	50	180	76	270	114
121	51	181	76	271	114
122	51	182	77	272	115
123	52	183	77	273	115
124	52	184	XXX	274	116
125	XXX	184	78	275	116
125	53	185	78	276	XXX
126	53	186	78	276	117
127	53	187	79	277	117
128	54	188	79	278	117
129	54	189	80	279	118
130	55	190	80	280	118
131	55	191	XXX	281	119
132	XXX	191	81	282	119
132	56	192	81	283	XXX
133	56	193	81	283	120
134	56	194	82	284	120
135	57	195	82	285	120
136	57	196	83	286	121
137	58	197	83	287	121
138	58	198	XXX	288	XXX
139	58	198	84	288	122
140	59	199	84	289	122
141	59	200	84	290	XXX
142	60	201	85	290	123
143	60	202	85	291	123
144	XXX	203	XXX	292	123
144	61	203	86	293	124
145	61	204	86	294	124
146	61	205	86	295	XXX
147	62	206	87	295	125

Distribution of the values

We simply compute the distribution of the values of the RNG, we expect, of course, to find a uniform distribution

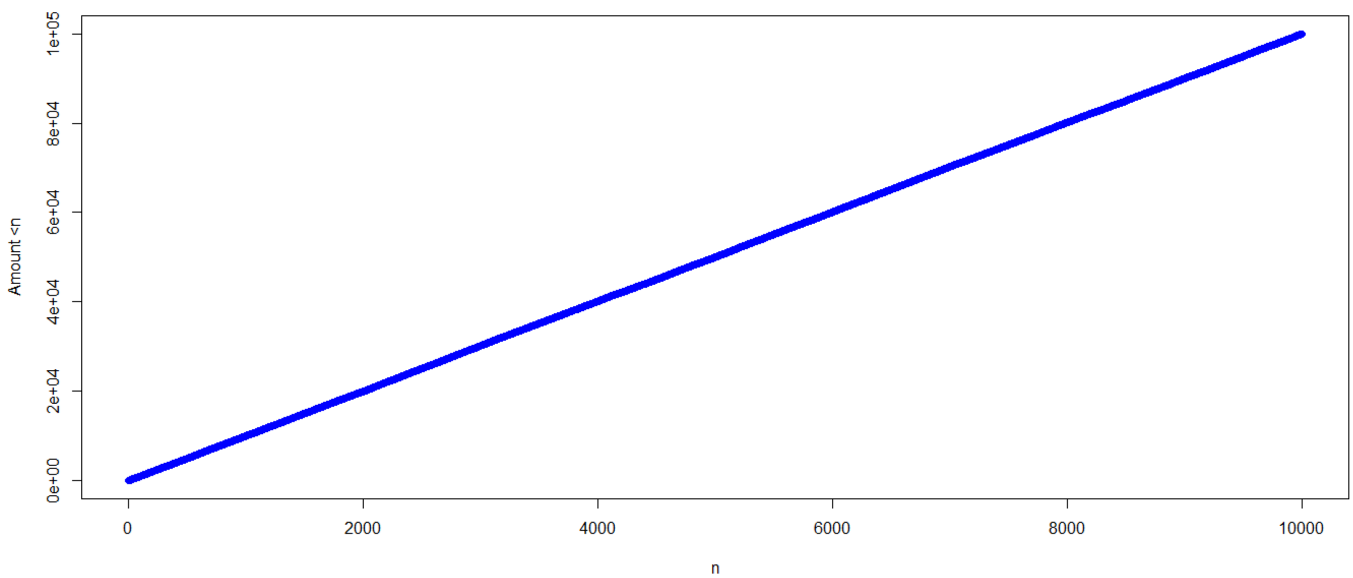
```
Random r = new Random();

            Int32[] values = new Int32[10000000];
            Int32[] dist = new Int32[100000];

            for (int i=0;i< 10000000; i++)
            {


                values[i] = r.Next(10000);


            }

            for (int j = 0; j < 10000; j++)
            {
                int s = 0;
                for (int i = 0; i < 100000; i++)
                {

                    if (values[i]<j)
                        s++;

                }

                dist[j] = s;
            }

            String csv = "";

            for (int j = 0; j < 10000; j++)
            {

                csv = csv + j + "," + dist[j] + "\n";

            }
```

```
            File.WriteAllText("dist.csv", csv);
```

We plot the csv file using CRAN-R.

```
myvalues <- read.csv("C:\\tmp\\dist.csv", header=FALSE, sep=",",
as.is=TRUE)
plot(myvalues,"n","Amount <n", col="blue")
```

Visually the distribution looks acceptable.



Entropy study of the random byte generator

In terms of entropy, we compute the entropy of words generated by the byte generator through the `NextByte()` function.

We generate a long word of around 1 Megabyte (1 million of symbols) by a concatenation of the NextByte() values. and we compute its entropy.

We do this for a significant amount of seeds and we study the entropy distribution.

Obviously a good RNG should produce words with high entropy, “close” to the maximal value of $Log_2(256) = 8.$

We use the following function for computation of entropy:

```
private static double getEntropy(byte[] word)
      {
          int N = word.Length;
          double H = 0;

          for (int i = 0; i < 256; i++)
          {
              int s = 0;
              for (int j = 0; j < N; j++)
              {
                  if (word[j] == (byte)i)
                      s++;
               }

            //  Console.Out.WriteLine("s="+s);
              if (s > 0)
  H += s * (Math.Log((double)Decimal.Divide(s, N)) / Math.Log(2));

          //    Console.Out.WriteLine("H=" + H);
          }

          return -H/N;
      }
```

We compute the entropy of random words of 100,000 bytes generated by the RNG,

```
Random r = new Random();
     Byte[] words = new Byte[100000];
     double[] H_ = new double[1000];

     for (int i = 0; i < 1000;i++)
     {

         r.NextBytes(words);

         H_[i]=getEntropy(words);
         Console.Out.WriteLine(H_[i]);

     }
```

The computation for 100 randomly generated words produces the following Shannon entropy values:

7,99823032640388	7,9983113538283	7,99775169222549
7,99828624313363	7,99814346854661	7,99823239400712
7,998315348654	7,99830150653076	7,99829822106673
7,99826771552547	7,99790805375961	7,99825392865045
7,99807405553931	7,99799533804699	7,99822097314874
7,99805543820311	7,99818076110388	7,99818245947268
7,99821652975896	7,99787877349106	7,99801869789196
7,99810917106913	7,9977097866743	7,99820063936121
7,99811157511275	7,99791685230915	7,99826131643008
7,99811341434992	7,9982608922564	7,99811643629185
7,99810028597774	7,99804017967301	7,99805649129511
7,99830879187033	7,99832384722511	7,99792144786888
7,99838178840981	7,99837614082797	7,99814589201522
7,99848442219811	7,99812882163527	7,998298080822
7,99812782299489	7,9981454439674	7,99801831209942
7,99824164954725	7,99810280098585	7,99814768836675
7,99822417492894	7,99814425789982	7,99828198884281
7,99814410326633	7,9981309241144	7,99832537174521
7,99829296475336	7,99821857324085	7,99807012779352
7,99800195232145	7,99783047136812	7,99838415062449
7,99810369679359	7,99831770190317	7,9981519910259
7,99808926117266	7,99824457285042	7,99825746185536
7,99810137907029	7,99793249746241	7,99807083027865
7,99813371469752	7,99813493567317	7,99807881676417
7,99809154987319	7,99788682713365	7,9980311556611
7,9980257557848	7,99827764821462	7,99809562427614
7,9982985698453	7,99810480816228	7,99804339079616
7,99833829952274	7,99808903019212	7,99795757906399
7,99827823843699	7,99812002785753	7,99827023134384
7,99802448868977	7,9985477624685	7,99832012644341
7,99823705901078	7,99837260597397	7,99820780227374
7,99804521204064	7,99793490085075	7,99782392679545
7,99819151951271	7,99819087574906	
7,99868654776436	7,99819587350322	

As we see the entropy values are all > 7.997, which is acceptable. We also get similar results when generating the random words from a variating seed.

Conclusion: In this article, we have seen a few basic techniques to check the randomness of a RNG. The built-in .NET System.Random.Rand RNG has no

security and must never be used for cryptography or anything involving a secret number generation. It has an average and acceptable randomness even if numbers - at a fixed rank - will almost always have the same values.