Journal of Universal Computer Science, vol. 22, no. 8 (2016), 1023-1047
submitted: 9/3/16, accepted: 29/7/16, appeared: 1/8/16 © J.UCS

The Complete Set Simulation of Stochastic Sequences without Repeated and Skipped Elements

Aleksei F. Deon
(Department of Computer Science, Department of Mathematical Sciences
N.E. Bauman Moscow State Technical University, Moscow, Russia
deonalex@mail.ru)

Yulian A. Menyaev
(Winthrop P. Rockefeller Cancer Institute
University of Arkansas for Medical Sciences, Little Rock, AR, USA
yamenyaev@uams.edu)

Abstract: Random sequences are widely used in theoretical and practical areas of interests in human and technical activities. An important part of these fields is referred to as the procedures of producing stochastic values. One direction adapts the sequenced generating of pseudorandom numbers and the other direction uses all stochastic sequences in objects of completed sets. The first direction is well studied and is traditionally used in cryptography and technical systems in medical and biological trials. The second direction is generally used in systems for preliminary universal testing where all or characteristically important sequences belong to a given diapason of actions are required. In this current work we explore the second direction, where the underlying approaches in modern generators of random numbers are considered. The simulation of complete sets of random numbers shows that either skipping or repeating of generated values is possible. We've formed the requirements that if followed, the problems of skipping and repeating are overcome. Next, we've proposed novel algorithms to form completed ranked sets of random sequences. Also, we've proposed novel algorithms on the basis of factorial expansion of random numbers which provide fast generation of such sequences. A discussion of the advantages and disadvantages of the indicated statements completes this paper.

Key Words: Computer Simulation, Random Number Generator, Stochastic Sequence Algorithm, Probability and Statistics.
Categories: G.2.1, G.3, F.2

1 Introduction

The world around us is full of many cases of certain and unpredictable events. A lot of things are caused by occurrence. At such moments, we face stochastic phenomena and have to use our background knowledge or follow the recommendations of others who have experienced the same conditions. Because we are not sure if we are correctly choosing, the nature of uncertainty dictates us to search for adequate activities in current circumstances. In turn, this means that choosing itself reduces the uncertainty in a varying degree.

In 1948, Claude Shannon proposed that to choose the sequence of binary questions, only the answers of 'yes' and 'no' may be used [Shannon 1948]. The

structure of such approach may be presented as graphs of non-binary and binary trees, [Fig. 1].

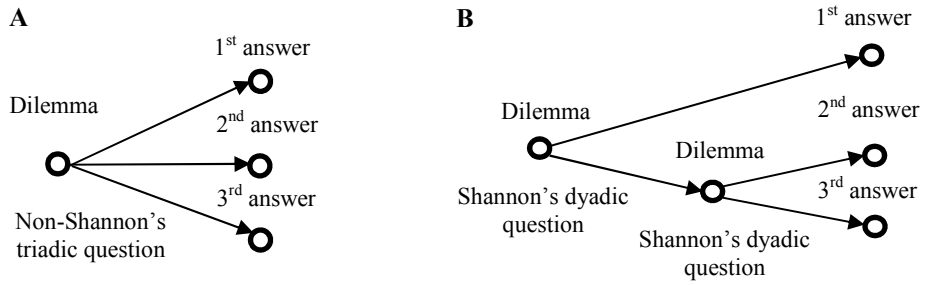

Figure 1: Non-binary (A) and binary (B) tree graphs of questions and answers

To discuss the terms of correct answers 'yes' and their probabilities, Shannon proposed the term 'entropy' $H(X)$ as a measure of uncertainty for choosing the final possible answer at the moment of uncertainty reduction for root of tree may occur [Shannon and Weaver 1963]:

$$H(X) = \sum_{i \in I} p_i log_2 p_i \quad (1)$$

Many opinions about the quality of such estimation were expressed, but now such fields as informatics, cybernetics, theory of transmitting and receiving, and others are using Shannon's entropy proposal [Banati and Bajaj 2013, Cover and Thomas 2006, Karaboga and Ozturk 2011]. Further, in practical realization of random number generators (RNG), the relevant algorithms have been developed, for which discussions about advantages and disadvantages may be found in [Feng et al. 2010, Maurer 1992, Wegenkittl 2001].

In the beginning of the computer era, Alan Turing did estimations of the time period required for uncertainty reduction [Turing 1950]. For this purpose he proposed the fundamentals to solve the task of how fast the computer may reduce or eliminate the uncertainty in the limited time period of its work. Turing's 'simple' machine used three actions: shift to the left; shift to the right; and reading the information bit having 0 or 1 values. Later, similar principles have been enlarged by Kolmogorov, subsequently named 'complexity theory' [Kolmogorov 1968, Li and Vitanyi 2008, Velmurugan and Santhanam 2010]. All those statements were applied at a basic level in cryptography; however the algorithms that were used and their practical results have been discussible up to now [Dagtas et al. 2004, Evans et al. 2001, Jain 2010, Tan et al. 2011]. Also, in the theory of random sequences, the different examples of ergodic processes are in use, which are known as Markov chains [Forsati et al. 2013, Hellekalek and Wegenkittl 2003, Miner et al. 2012].

In the above-mentioned techniques, the uniform distribution of random values is usually used. Based on principles of RNG, two basic approaches are known: 1) true random number generators (TRNG), which produce random numbers in real time from physical processes, and 2) pseudorandom number generators (PRNG), which use

algorithms to produce sequences of numbers whose properties are almost the same as natural random sequences. TRNG techniques use natural effects, such as noise of natural phenomena, properties of semi-conducting materials, etc. One direction in this field was hardware random number generators (HRNG), which found application in generating of cryptographic keys to encrypt data sent over computer networks. The mathematical statistics used to characterize the receiving uniform distribution of values in TRNG for what Kolmogorov-Smirnov test is typically applied. Special outer devices or circuits connect to the computers to provide this, as shown in [Nandy et al. 2012, O'Donnell et al. 2005, Suh et al. 2004, Yang 2010].

The evolution of algorithmic PRNG has occurred since the congruent generator was proposed [Mani and Derick 2010, Park and Miller 1988, Storm and Price 1997]. However, a short while after it was found that pseudorandom values are repeatable after some constant period of time. This fact energized Matsumoto and Nishimura to develop an algorithm that was capable of increasing the period of repeating up to the value of $2^{19937} - 1$ [Matsumoto and Nishimura 1998, Nishimura 2000]. The search for new methods did not stop after that, and today many promising techniques are known. For example, the quadratic generator, Blum-Blum-Shub generator, and others [Blum et al. 1986].

A linear congruential generator (LCG) is the next important issue which should be considered in some detail here [Fister et al. 2013, Hellekalek 1998, Yujian and Liye 2010]. It produces the next element of random sequence by using the following recurrence relation:

$$x_{n+1} = f(x_n)\ mod\ m. \tag{2}$$

If some kind of function $ax_n + c$ is used for an algorithm of realization $f(x_n)$, the generator is congruential:

$$x_{n+1} = (ax_n + c)\ mod\ m. \tag{3}$$

Moreover, modern consequent generators are not limited by ordinary linearity; they use additions of different types of shifting, inversion, bit disjunction with modulo 2 (*XOR)*, and other operations. The properties of LCGs found different applications in many areas such as information systems, cryptography and mathematics [Arora et al. 2015, Diffie and Hellman 1976, 1979, Wallace 1996, Leeb and Wegenkittl 1997, Dodis et al. 2013, Karloff and Raghavan 1993, Kasdin 1995, Dorrendorf et al. 2009], as well as in biological and medical research [Cai et al. 2016, Song et al. 2006, Juratly et al. 2015, 2016, Menyaev et al. 2006, 2013, 2016, Menyaev and Zharov 2006, Miklós et al. 2009, Sarimollaoglu et al. 2011, 2014, Zharov et al. 2001]. Such generators explore the statement about uniform distribution of random values, but without talking about completeness of distribution [Deon and Menyaev 2016]. The problem is that this type of generator can't produce all random values within a required period of time under the condition of one-shot generation of absolutely all random values that have given lengths. Unfortunately, this task isn't completely solved yet.

Let's make a simple experiment, in which a generator *Random* is taken from the Microsoft Visual Studio 2013 compiler. The searching of a maximal random value in the programming language C# [Manning et al. 2008, McConnell 2002, Schildt 2010] may be fulfilled with the following program code by consequently using the length of random value *n*.

```
static void Main(string[] args)
{  int m = 1;
  for (int n = 0; n <= 10; n++, m *= 10)
  {  int max = 0;
    Random r = new Random(0);
    for (int j = 0; j < m; j++)
    {  int v = r.Next();
      if (max < v) max = v;
    }
    Console.WriteLine(
      "n = {0,2}  m = {1,12}  max = {2,12}",
      n, m, max);
  }
  Console.ReadKey();
}
```

The result of executing this code provides the following listing:

```
n =  0   m =            1   max =   1559595546
n =  1   m =           10   max =   2099272109
n =  2   m =          100   max =   2147425016
n =  3   m =         1000   max =   2147425016
n =  4   m =        10000   max =   2147425016
n =  5   m =       100000   max =   2147452437
n =  6   m =      1000000   max =   2147483082
n =  7   m =     10000000   max =   2147483591
n =  8   m =    100000000   max =   2147483618
n =  9   m =   1000000000   max =   2147483646
n = 10   m =  10000000000   max =   2147483646
```

The above example shows that if the length of a random value is 10 in decimal scale, the maximum generated value is limited by the number 2147483646. This value is equivalent to the constant of 31 bits, which is in hexadecimal form looks as 0x7FFFFFFF and in decimal one as 2147483647. Thus, according to the listing above the random values belong to the interval [0, 2147483647].

The next task is to explore the uniformity of the generation of values, but before we do this, we have to find the minimal number for *Random* generator, which determines the beginning of interval of random values. The next example of the program code allows us find it:

```
int n = 0x7FFFFFFE;                    // maximum value
Console.WriteLine(" n = {0}", n);
int min = n;              // beginning of maximum finding
Random r = new Random(0);
for (int j = 0; j <= n; j++)
{  int v = r.Next();
  if (v < min) min = v;
}
Console.WriteLine("min = {0}", min);
```

The result of executing this code shows the following:

```
n = 2147483646
min = 0
```

So, this is a true confirmation that the interval of random values generated is determined to be [0, 2147483646]. In each sequence of this set, uniformly distributed random values should be placed in the aforementioned interval. Let's check it by using some kind of values as: 10, 100, 1000, 10000, 100000, 1000000. The following program code solves this task.

```
int n = 0x7FFFFFFE;
Console.WriteLine(" n = {0}", n);
int[] q = new int[] { 1, 10, 100, 1000, 10000,
                      100000, 1000000};
int[] c = new int[q.Length];
Random r = new Random(0);
for (int j = 0; j <= n; j++)
{  int v = r.Next();
  for (int k = 0; k < q.Length; k++)
    if (v == q[k]) c[k] += 1;
}
for (int k = 0; k < q.Length; k++)
  Console.WriteLine(
    "q[{0}] = {1,10}  c[{0}] = {2}",
    k, q[k], c[k]);
```

After executing this code, the following listing appears.

```
n = 2147483646
q[0] =          1   c[0] = 0
q[1] =         10   c[1] = 1
q[2] =        100   c[2] = 0
q[3] =       1000   c[3] = 1
q[4] =      10000   c[4] = 1
```

q[5] = 100000 c[5] = 0
q[6] = 1000000 c[6] = 2

So, results such as *c[0]=0*, *c[2]=0*, *c[5]=0* refer to the fact that *Random* generator does not produce all the random values in a given interval [0, 2147483646].

As was mentioned above, TRNG are really complex and expensive physical tools. At the same time, based on observed references regarding PRNG, it could be seen that existing methods can't reach all the sequences which are distributed uniformly. The best case was found for the generator MT19937 [Matsumoto and Nishimura 1998], which has the biggest period of repetition. However, the problem of receiving real stochastic sequences is still not solved in this case.

Let's make a subtotal summary here. The task to create the universal RNG is yet unsolved and therefore it's still very important. However, the set of random sequences can be restricted, which may allow for organization of the random values generator. In the next section, we will demonstrate novel principles for how this kind of generator, which includes completed set of values, uniform distribution, and no skipping generating, could be made.

2 Fundamentals

When natural phenomena are under study, the first question that is necessary to clarify is what is their behavior under certain circumstances? In such phenomena, different objects may be involved and their properties may be similar, or different, such as leaves from the tree blown by the wind, for example. If a collection of such objects may be calculated mathematically, we may say that the calculating set, if impossible, is a continual set [Kolmogorov and Fomin 1999]. This criterion allows us to pay attention to the uncertainty of choosing any object in a certain set. So, in calculating sets, choosing objects may be accompanied by numbers. This means that the uncertainty of choosing consists in the fact that the object maybe accompanied by any number. At the same time, for continual sets, the problem of an uncertainty of choosing consists in the inability to mark the objects by certain numbers. For example, well-known mathematical constants such as π or e aren't defined as completely final values, but they definitely exist and their values are rather close to each other.

Previously the definition of an uncertainty was mentioned, and now it's time to clarify what kind of an uncertainty will be considered here next. For this purpose, the method of numeration or technique of identification of object choosing has to be clarified. Therefore, we consider the uncertainty of choosing of an object, which belongs to a completed set, where each object has its own unique number. Thus, in the current model, the phenomenon is underlying the set of similar primary objects, but they are different in unique counting numbers.

Now let's pay attention to the time-dependent circumstances of the aforementioned phenomena. In nature the objects maybe observed in sequence, which means they are observed one by one during a period of time. At once, the objects may appear simultaneously in the same period of time. So, the simplest mathematical case

refers to the appearance of one object in a certain moment of time. Let's take this statement as a basis for the simulation of the random sequences here. It is important that in this model, the observation of 'no object' is impossible because a 'void' object isn't presented in a set.

The circumstance of the time period means that it's crucially necessary to take into consideration the length of observation between two events. Thus, in the first and simplest case, at a moment of time the one object could be observed. In the second case, at two moments of time $[t_1, t_2]$, where $t_1 < t_2$, two objects appear one by one: either one object appears twice at moments t_1 and t_2, or the 1st object at t_1 and the 2nd one at t_2. Mathematically, this corresponds to sampling with or without repetition. For a more detailed explanation, let's specify the objects here as b_i and b_j with any chosen numbers i and j. In the second case, an uncertainty of sampling should be considered as functional, which may choose any kind of sequences inside the following set $\{< b_i, b_i >, < b_i, b_j >, < b_j, b_i >, < b_j, b_j >\}$. Here an uncertainty is determined by sampling of one sequence among four of them, but which sequence that is chosen is under identifying by natural phenomena.

In the following, let's talk about non-repeatable objects in the one sequence, which is a set of sequences $\{< b_i, b_j >, < b_j, b_i >\}$. Next, let's assume that in sequences we may observe n objects. This means that the minimal set of distinguished sequences, which consist of non-repeatable objects, is the set of probable sequences having all transpositions among n objects. The summarization of these statements appears as the following:

$$B = \{b_1, b_2, \dots, b_n\} \tag{4}$$

$$D = \left\{ \begin{matrix} < b_{i1}, b_{i2}, \dots, b_{in} > : i1, i2, in \in \overline{[1, n]}, \\ i1 \neq i2 \neq \dots \neq in \end{matrix} \right\} \tag{5}$$

where B is the set of observed objects and D is the set of observed sequences. Each sequence in set D includes all non-repeatable objects from set B.

Mathematically, D is the set of all transpositions of objects from set B. Element $d \in D$ is the one sequence that includes n non-repeatable objects $b \in B$. In this definition, we have a complete set of all sequences having n length that could be characterized as in each sequence all the objects are mentioned once. In other words, this is the simple completeness in observed objects, and simple completeness in sequences.

The next step allows us to see that an uncertainty of sampling in natural phenomena might be considered in two aspects: 1) uncertainty of observing an object in an exact location inside sequences; 2) uncertainty of observing a sequence, but each sequence expresses itself if the total sampling inside D is done. So, we are talking about the uncertainty of the functional of sampling due to the fact that potency $|D|$, or in other words an amount of elements $card(D)$ in virtue of completeness of D, are equal:

$$|D| = card(D) = n!. \tag{6}$$

This equation comes directly from combinatorial analysis [Johnsonbaugh 2008]

and general algebra [Waerden 1991] where the estimation of the quantity of indexes transpositions for *n* elements is considered.

The same result may be taken from the probability theory [Gnedenko 1998] by considering the methods of sampling without repeating. In the 1st place in a sequence, any object $b_{i1} \in B$ from total n may be presented; in the 2nd place, any object $b_{i2} \in (B \backslash \{b_{i1}\})$ which belongs to the set having $n-1$ elements may be presented, and so on. Thus, by multiplying amount of all variations we will have the factorial value $n!$. Using this statement, it's possible to confirm that one of the characteristics of uncertainty, for the functional of sampling regarding to all sequences is a factorial completeness of the total amount of sequences.

It should be mentioned that when we started talking about functional sampling, we didn't use conception of function $f(r)$, which stands as a single number r to the sequence $d_r \in D$. Here r characterizes the sequence of actions, which, when applied, allows us to get one sequence from *D*. In turn, $|D|$ characterizes an uncertainty prior to the beginning of sampling, and also in accordance to *r,* will finish the sampling by pointing the concrete sequence d_r. The set of univocal functions $f(r)$ constitutes the functional *F* on the set of *D*, where *r* is the number of method to obtain the required sequence: $F = \{f_r \to d_r \in D\}$.

Since the amount of sequences $card(D)$ is matching with the potency $|D|$, this means the potency of functional *F* is matching $|D|$ as well:

$$|F| = |D| = n!. \tag{7}$$

So far it provides the statement that one variant of r identification could be $r \in [\overline{1,n!}]$. The question that logically follows is what number of $r \in [\overline{1,n!}]$ corresponds to the sequence $d_r \in D$? The answer will be given here later.

Now it's time to summarize all above-mentioned denotations together in a uniform model, which will allow us in the next section to reach the algorithms and their implementation in computer programs.

Let's name the model *M* of set *D* consisting in completed sequences having non-repeatable objects from *B* as triplet of sets, where the realization of reduction of uncertainty for the functional *F* is possible:

$$M = (B, D, F). \tag{8}$$

The set of objects *B* is being given a priori in accordance of chosen strategy of phenomenon study. The set of sequences *D* presents itself as a set of the minimal amount of the simplest completed sequences. So, each sequence in *D* includes all elements from *B* having different listings of objects. A set of functional *F* includes all functions, which may determine an uncertainty in sampling each time for the only one sequence.

Now let's return to the question of how to realize the functional *F* for a sampling of concrete sequence d_r. Based on a mathematical definition of the set, it could be determined by using two methods: 1) explicitly list all the elements of the set; 2) point to the actions which may present any element of the set. If procedures aren't

completed, this means it's impossible to present all elements of a set, which in this case the functional of actions is incomplete.

These features of set forming allow for pointing to the first technique in realization of functional in model *M*, i.e. direct listing of $d \in D$ until the sequence, which is satisfied when *r* criteria is found: $d = d_r$.

The uniqueness of this technique is obvious and that's why to form *D,* only the one sequence may be found which is satisfied to transposition *r* of objects from *B*. Transposition *r* is the sole finding, following directly from combinatorial analysis [Johnsonbaugh 2008].

The second technique in realization of functional *F* is based on analyses of the uncertainty for model $M = (B, D, F)$. The point is that the quantity of sequence-elements in set *D* is $card(D) = |D| = n!$. So, here *r* is some kind of a whole number in arithmetical range of whole numbers $[\overline{1,n!}] = [1,2,3,\dots,n!]$. All elements of $r \in [\overline{1,n!}]$ are strictly ordered. Thus, it's necessary to achieve the order for numbers of sequences from *D*. For this purpose, let's take in consideration the positional representation of whole number *X*, which consists of *n* digits of x_i in some numeration system with a basis of *s*:

$$X = x_{n-1}s^{n-1} + x_{n-2}s^{n-2} + \cdots + x_1 s^1 + x_0 s^0. \quad (9)$$

Now let's use combinatorial definition of sequence in set *D*, what is a transposition of non-repeatable numbers of indexes. Let's assume that objects $b \in B$ are numbered, or in other words, marked by whole numbers taken from arithmetical range $[\overline{1,n}]$. Then, the first sequence one which corresponds to the minimal positional number, where numbers themselves are taken from positional presentation of indexes.

Sequence	*X*	*r*
1, 2, 3	123	1
1, 3, 2	132	2
2, 1, 3	213	3
2, 3, 1	231	4
3, 1, 2	312	5
3, 2, 1	321	6

Table 1: Example of ordering

For demonstrative understanding of the theoretical basis presented above, let's consider a simple example where $n = 3$. Let's mark objects in *B* by numbers 1, 2, 3, which gives us $X_1 = 123$. Then the second sequence after applying the transposition gives $X_2 = 132$. Now it's possible to observe some kind of ordering: $123 < 132$, or $X_1 < X_2$. So, based on this logic, the next sequences are looking like those presented in [Tab. 1].

The total amount of sequences is $n! = 3! = 6$ for which the numbers of *r* of functions $f_1, \dots, f_6$ from functional *F* are corresponded directly. So, if an uncertainty of nature has chosen the function f_4, for example, then we may observe the 4th

sequence 2, 3, 1 from the table above.

The structure of such approach may be presented as graph in [Fig. 2].

The link between *r* and appropriate sequence is determined in the following. Let's write down now the factorial explicitly:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1. \quad (10)$$

This equation contains the same amount of factors as the amount of objects having *n* length long. Let's pay attention to the fact that in first place of the left part, any number taken from *n* may be placed. To determine the biggest number of the left part we need to exclude the right part of factorial by using its properties:

$$n! = n(n-1)!. \quad (11)$$

Due to $r_{max} = n!$ it's obvious that all other numbers will be found by using this technique of factorial decreasing, and also by the application of excluding those indexes which were determined in previous iterations.

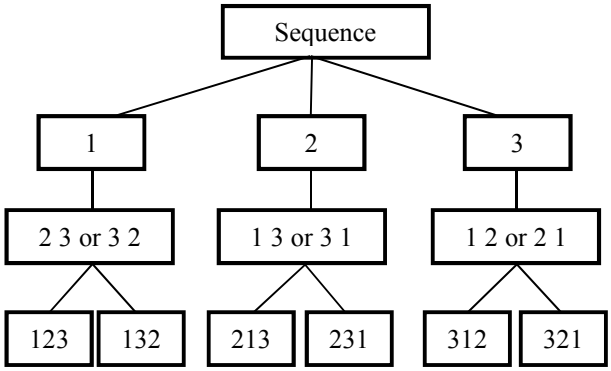

Figure 2: Tree graph for the example of ordering

In the next section of this article, the concrete realization of methods and techniques considered here are shown, where the programming language C# is used from Microsoft Visual Studio 2013. Additionally, in section *Theorem* the mathematical proof of the theorem which summarizes observed statements is given.

3 Constructions and Results

The forming of the sequences having maximal length in interval $[\overline{1,n}]$ incorporates the theoretical formula with respect to n_z numbers, which are included in sequence on *z* position:

$$n_z \in ([\overline{1,n}] \backslash \{n_1, n_2, \ldots, n_{z-1}\}). \quad (12)$$

Let's add a property of increasing ranking: the next chosen number is bigger than the previous one with an allowance for the formula mentioned above. This ensures that all elements will be presented in sequence. The initial minimal sequence is the only one and consists strictly in ordering numbers $1,2,3,\dots,(n-1),n$. To get number n_z, the function is required, and let's call it *NextFreeValue( )*. It can show for the exact position of z the next increasing value which isn't coincided in numbers with previous positions such as $1,2,\dots,z-1$ in generated sequence. For example, let the sequence consists of 4 elements 2, 3, 1, 4, and let's assume that position $z=3$ where number 1 is located ($n_{z=3}=1$), is interesting for us. After the applying of function *NextFreeValue( )* the next number 4 will be offered because of numbers 2 and 3 are located on previous positions, and using them is prohibited for the position of $z=3$. In another example let's consider the sequence 2, 3, 4, 1, and let's try to find the next increasing value for the same position $z=3$. It's impossible due to the next number is 5, which can't be presented in sequence having 4 elements. In this situation *NextFreeValue( )* returns 0, which is interpreted as abort to provide the action.

Below is the program code for function *NextFreeValue( )* on C# dialect. The strings used for testing are commented out. In the list of value parameters the *int[] q* is a pointer to the array having the number of elements in random sequence. Parameter *int n* contains the amount of elements in array *q*. Parameter *int z* assigns the index of position for which is required in the selection of the next random value which in turn isn't repeating among the previous elements until index *z* is reached.

```
static public int NextFreeValue(int[] q, int n, int z)
{  int v = q[z] + 1;                        // possible value
   if (v > n) return 0;           // increasing impossible
   int w = 0;           // starting of checking in array q
   for (; v <= n; v++)         // area of possible values
   {
//    Console.WriteLine("v = {0}", v);
      int j = w;       // starting of checking continuing
      for (; j < z; j++)       // elements until z position
      {
//       Console.WriteLine("q[{0}] = {1}", i, q[i]);
         if (q[j] == v) break;   // value wasn't accepted
      }
      if (j >= z) break;  // value v wasn't faced before
      w = j + 1;        // continue from here henceforth
   }
   if (v > n) v = 0;
   return v;                              // next vacant value
}
```

So, the function *NextFreeValue( )* allows for either finding the next vacant value for position *z* which is one more than previous value, or generate the abort.

Next, acceptable values in the simulated sequence are placed either to the left from

position z or directly in position z in array q. Now we want to find the minimal number for position $z + 1$ which is located to the right of position z, but it can't be equal to those numbers which are to the left of position z. Below is the program code for function *FreeValue( )*, which solves this task. Cycled using of *FreeValue( )* allows us to form the rest of the sequence. The list of value parameters is the same as for *NextFreeValue( )*.

```
static public int FreeValue(int[] q, int n, int z)
{ for (int k = 1; k <= n; k++)
  {
    int j = 0;
    for (; j <= z; j++)
      if (q[j] == k) break;
    if (j > z) return k;
  }
  return 0;
}
```

Joining the functions *NextFreeValue( )* and *FreeValue( )* into one function as *ProcessUp( )* allows us to get a single random and increasing sequence having n length in the interval $[\overline{1,n}]$. It should be noted here that the next increasing sequence might require varying of preceding values. For example, let's take the sequence 1, 2, 3, 5, 4. The changing of the last value 4 to the following next value 5 is impossible due to the fact that the value 5 is already presented to the left. At the same time, the increasing of value 5 is impossible either due to it's a maximum allowable value. Thus, the next increasing and random sequence will be 1, 2, 4, 3, 5. So, the process of iteration in applying of function *ProcessUP( )* will form the sequence 1, 2, 4, 5, 3, and then 1, 3, 2, 4, 5, and so on. This will contribute the ranking in forming of sequences. Below is the program code for function *ProcessUp( )*. The list of value parameters includes array pointer *int[] q* having elements of random sequence. Parameter *int n* assigns the amount of elements in q.

```
static public int ProcessUp(int[] q, int n)
{  int z = n - 2;                  // next to last from position z
   while (true)
   {  int nv = NextFreeValue(q, n, z);
//    Console.WriteLine("z = {0}   nv = {1}", z, nv);
     if (nv != 0)   // moving of z to the left isn't required
     {  q[z] = nv;                          // next vacant value
        while (++z < n)                 // position z till the end
          q[z] = FreeValue(q, n, z - 1);
        return z;
     }
     if (z = 0) break;                              // no sequence
     z--;                                  // position z to the left
```

```
  };
  return 0;                              // no next sequence
}
```

If function *ProcessUp( )* is executed with given sequence 1, 2, 3, 5, 4, this results in the following information which demonstrates how this function has determined the next increasing sequence.

```
1  2  3  5  4
z = 3   nv = 0
z = 2   nv = 4
1  2  4  3  5
```

To get a completed set of random sequences having, for example, a length of 4, it's necessary to execute function *ProcessUp( )* which starts with the sequence 1, 2, 3, 4 and finishes with the sequence 4, 3, 2, 1. This completed set includes $4! = 24$ of all random sequences upon interval $[\overline{1,4}]$.

As an example, below the program code is showing how to realize the mentioned task for sequences where each of them has, for example, 7 elements. The total quantity of such sequences is $n! = 7! = 5040$.

```
int[] q = new int[] { 1, 2, 3, 4, 5, 6, 7 };
int n = q.Length;                        // length of a sequence
int r = 0;                               // number of the sequence
while (true)
{  Console.Write("r = {0,4}", ++r);
   Console.Write("     q =");
   foreach (int w in q)
     Console.Write("{0,4}", w);
   Console.WriteLine();                  // a new string
   if (ProcessUp(q, n) == 0) break;      // the process
}
```

The result of the execution of this code is the following completed ranking set (abridged).

```
r =    1    q =  1  2  3  4  5  6  7
r =    2    q =  1  2  3  4  5  7  6
r =    3    q =  1  2  3  4  6  5  7
–        - - - -
r =  100    q =  1  2  7  3  5  6  4
r =  101    q =  1  2  7  3  6  4  5
r =  102    q =  1  2  7  3  6  5  4
–        - - - -
r = 1000    q =  2  4  3  6  5  7  1
```

```
r = 1001    q =  2  4  3  6  7  1  5
r = 1002    q =  2  4  3  6  7  5  1
–         - - - -
r = 2200    q =  4  1  3  6  5  7  2
r = 2201    q =  4  1  3  6  7  2  5
r = 2202    q =  4  1  3  6  7  5  2
–         - - - -
r = 3300    q =  5  4  3  2  7  6  1
r = 3301    q =  5  4  3  6  1  2  7
r = 3302    q =  5  4  3  6  1  7  2
–         - - - -
r = 5038    q =  7  6  5  4  2  3  1
r = 5039    q =  7  6  5  4  3  1  2
r = 5040    q =  7  6  5  4  3  2  1
```

The task has been fulfilled, i.e. simulation of the completed and ranked set of sequences without repetitions having maximal length n in interval $[\overline{1,n}]$ has been done. Definitely, this is the exact completed set of all probable sequences. Further in this paper, this set is required to verify bijective correspondence between randomly given number r of function f_r in functional F and index of sequence d_r in set D for model $M = (B, D, F)$.

In the *Theorem* section, the proof of the theorem regarding bijection in model $M = (B, D, F)$ and the necessity clause supposes that if sequence $d \in D$ is given randomly, then it's possible to establish univocally such number as r for which it's true: $d_r = d$. Below, the function *DeonNumber( )* is presented, and it may perform this task. The list of value parameters for this function includes array pointer *int[] q* of given sequence. The function returns back the number r of this sequence in a completed and ranked set of sequences D, i.e. $d_r \in D$.

The necessity clause in the *Theorem* section of this paper implies the effective tool of transforming a uniformly distributed sequence to a certain number, which provides unique identifiability in a complete factorial set of numbers of all sequences. So, if in the input of function *DeonNumber( )* an arbitrary uniform sequence is loaded, the result is that this function returns the unique number associated with this sequence. If all those numbers of all sequences are derived, it means that unique number of 'maximal' sequence is equal to factorial $n!$.

```
static int DeonNumber(int[] q)
{ int r = 0;          // number of function at this moment
  int n = q.Length;                    // length of sequence
  int nb = n;                  // initial amount of objects
  int[] b = new int[nb];   // position numbers of objects
  for (int i = 0; i < nb; i++) b[i] = i + 1;
  int zF = 1;                              // zF-factorial for b
  for (int i = 2; i < n; i++) zF *= i;
  for (int z = 0; z < n - 1; z++)  // cycle of positions in q
```

```
  {  int k = 0;
     for (; k < nb; k++)
        if (q[z] == b[k]) break;
     int rg = k * zF;              // number before the group
     r += rg;           // uncompleted number of sequence
       // displacement: removing the element b[k] from b
     for (int i = k; i < nb - 1; i++) b[i] = b[i+1];
     nb--;                                  // one object less
     zF /= n - z - 1;                      // zF-factorial for z
  }
  r++;           // accounting of last element of sequence
  return r;                                // number of function
}
```

To launch the function *DeonNumber( )* the following fragment of program code is sufficient, where sequence, for example, 5, 4, 3, 6, 1, 7, 2 is given randomly.

```
int[] q = new int[] {5,4,3,6,1,7,2};
int n = q.Length;                      // length of sequence
Console.WriteLine("n = {0}", n);              // monitor
Console.Write("q = ");
for (int i = 0; i < n; i++)
Console.Write("{0,4}", q[i]);
int r = DeonNumber(q);           // number of sequence
Console.WriteLine("\nr = {0}", r);            // monitor
```

After this fragment is executed, the following information is provided.

```
n = 7
q =   5   4   3   6   1   7   2
r = 3302
```

This result is equal to the same as what was received a little bit earlier.

So, according to the sufficiency clause in proof of the theorem in the *Theorem* section, now the ability to find the appropriate sequence d_r in relation to randomly given number *r* has appeared. The process of the theorem proving in the sufficiency clause implies the effective tool of transforming a unique number associated with a uniformly distributed sequence, to a real example of sequence in complete factorial set of numbers of all sequences. So, if in the input of function *DeonSequence( )* has an array-buffer q for the derived sequence and randomly given number *r*, the result is that this function provides factorial decomposition of number *r* to the elements of complete sequence q. If total amount of numbers $n!$ is loaded, it provides an absolutely completed set of all the uniformly distributed sequences.

Below the function *DeonSequence( )* is presented. The list of value parameters of this function includes array pointer *int[] q* of needed sequence, and its number *int r*.

The result of executing the function *DeonSequence( )* is defined as the filled array q having elements of sequence d_r from set D.

```
static void DeonSequence(int[] q, int r)
{ int n = q.Length;
  int nb = n;                              // amount of objects
  int[] b = new int[nb];                   // array of objects
  for (int i = 0; i < n; i++) b[i] = i + 1;
  int nF = 1;                          // nF-factorial of sequence
  for (int i = 2; i <= nb; i++) nF *= i;
  int iq = 0;    // index for element which is forming in q
  for (int z = n - 1; z > 0; z--)          // cycle of positions
  {  int ng = nF / nb;       // amount of elements in group
     int w = r / ng;              // amount of previous groups
     if ((w * ng) < r) w++;              // group number for r
     int zb = w - 1;             // position of choice in b for q
     q[iq++] = b[zb];                   // element in sequence
      // displacement: removing the element b[zb] from b
     for (int i = zb; i < z; i++) b[i] = b[i + 1];
     r -= (w - 1) * ng;                // r for the next position
     nb--;                                  // one object less
     nF /= z + 1;                          // next nF-factorial
  }
  q[n - 1] = b[0];                         // last element in q
}
```

To launch the function *DeonSequence( )* the following program code fragment is sufficient, where, for example, the random sequence having the number $r = 3302$ in set of all sequences D with amount of elements $n = 7$ is performed. In such set D the following amount of sequences are included $(D) = n! = 7! = 5040$.

```
int n = 7;                                 // length of sequence
Console.WriteLine("n = {0}", n);                 // monitor
int[] q = new int[n];                // elements of sequence
int r = 3302;              // function number in functional
Console.WriteLine("r = {0}", r);                 // monitor
DeonSequence(q, r);          // computation of sequence
for (int i = 0; i < n; i++)
  Console.Write("{0,4}", q[i]);
```

After executing this fragment, the following information is presented.

```
n = 7
r = 3302
```

5 4 3 6 1 7 2

This result is equal to the same as was received earlier above.

Finally, let's consider the short example which may confirm the above statements independently, i.e. all received numbers have uniform distribution and each of them can be found an equal amount of times. The following program code confirms this for 7 elements which were taken as an example. So, each element is appearing 7! = 5040 times, and, at the same time, each of them can be found once in unique sequence. Thus, such sequences are presented just once in a complete set.

```
static void Main(string[] args)
{  int[] q = new int[] { 1, 2, 3, 4, 5, 6, 7 };
   int n = q.Length;
   int[] cQ = new int[n];
   for (int i = 0; i < n; i++) cQ[i] = 0;
   while (true)
   {  if (ProcessUp(q, n) == 0) break;
      for (int i = 0; i < n; i++)
      cQ[q[i]-1]++;                    // quantity of repeating
    }
    for (int i = 0; i < n; i++)
    {  int z = cQ[i] + 1;                          // counter for i
       Console.Write("{0}) {1,5}   ", i + 1, z);
     }
     Console.ReadKey();
}
```

The result of the execution of this code is the following listing:

```
1) 5040   2) 5040   3) 5040   4) 5040
5) 5040   6) 5040   7) 5040
```

So, the practical implementation is over. As a bottom line, the main result that has been received is the realizing of truly random sequence that was derived univocally without skipping of any kind of initial objects in set B.

4 Theorem

In section *Fundamentals*, it has been mentioned that the theorem and its proof will be given here. It would be meaningful if mathematical bijection could give a quick opportunity to define an isomorphism. It makes us organize the exact proof method of how elements of two sets can determine each other mutually and simply. The next theorem and its proving here are showing how to do this task.

Theorem. In order that bijective relationship will exist between completed set D of final sequences and functional F in model $M = (B, D, F)$, it is necessary and

sufficient that set of objects B which are included in sequences, will be strictly ordered.

Proof. Let's assume that the strict relation between elements of objects exists, which are included in all sequences of completed set. Let's put in correspondence mentioned arithmetical set B^*, consisted in objects as numbers. The minimal set which satisfies this statement is any kind of arithmetical interval beginning from some number α. Without breaking communality, let's put $\alpha = 1$. Now it's obvious that minimal arithmetical interval is $[\overline{1,n!}]$. Because of strict ordering, for each element $b^* \in B^*$ there is one and only one corresponding element $b \in B$. So, b^* and b are simply isomorphic.

Necessity. The proof of necessity deals with the definite sequence of actions that has to be pointed to calculate number r for any random sequence $d_r \in D$. Let's use upper indexes for explicit pointing the random sequence $d=<{}^1d,{}^2d,\ldots,{}^zd,\ldots,{}^{n-1}d,{}^nd>$, where index $z \in [\overline{1,n}]$. All $n!$ sequences are presented in set D. Let's divide D into n non-overlapping subsets $D_1 + D_2 + \cdots + D_n$ in accordance with ranking. Based on ranking properties of D, all sequences which begin from 1 will be presented in D_1, all sequences which begin from 2 will be presented in D_2, and so on. Size of each group is equal due to factorial properties of $n! = n \cdot (n-1)!$:

$$card(D_1) = \cdots = card(D_n) = (n-1)! \tag{13}$$

Let's present the resulting number r of sequence d as a partial sum:

$$R={}^1r+{}^2r+\ldots+{}^nr. \tag{14}$$

Depending on value of ${}^1d \in D_z$, the calculation is:

$${}^1r = \sum_{i=1}^{z-1} card(D_i) = (z-1) \cdot (n-1)! \tag{15}$$

By using iteration sorting of numbers ${}^2d,\ldots,{}^{n-1}d$ in preset sequence d, the values ${}^2r,{}^3r,\ldots,{}^{n-1}r$ may be calculated accordingly. The last value is ${}^nr=1$, and the final number r is determined by the sum:

$$r = \sum_{i=1}^{n} {}^ir. \tag{16}$$

The proof of necessity is done.

Sufficiency. Let's assume the number $r \in [\overline{1,n!}]$ is defined, what allows using the function $f_r \in F$ to calculate appropriate sequence $d_r \in D$. Let's point d_r on the same principles as above, which consisted of elements $d_r=<{}^1d_r,{}^2d_r,\ldots,{}^{n-1}d_r,{}^nd_r>$. Let's divide D into n non-overlapping subsets $D_1 + D_2 + \cdots + D_n$ in accordance with ranking. So, the following will be the same as above: all sequences, which begin from 1, will be presented in D_1, etc.; size of each group ng due to factorial properties is equal:

$$ng = card(D_n) = (n-1)! \tag{17}$$

This statement allows for having the determination of some number w from previous group as regard to element ${}^1d_r$ due to it is a result of integer division:

$$w = \begin{cases} \frac{r}{ng}, \text{ if excess is} > 0 \\ \frac{r}{ng} + 1, \text{ if excess is} = 0 \end{cases} \quad (18)$$

So, the position of *w* allows choosing appropriate element *b* from the set of objects:

$$^{1}d_r = b_w \in B. \quad (19)$$

For the next iteration regarding $^{2}d_r$ the number *r* has to be reduced by $(w-1)\cdot ng$ value, while element b_w which was already chosen is necessarily excluded from set *B*. Following the next iterations helps in obtaining elements $^{2}d_r,\ldots,^{n-1}d_r$. After all iterations, there will only be one element in set *B*, and it will be the last element $^{n}d_r$ in sequence d_r.

The proof of sufficiency is done by constructing.

The proof of the theorem is over.

In section *Constructions and Results,* the function *DeonNumber( )* makes calculation of number *r* of sequence d_r in accordance with algorithm of proof of the theorem for necessity. Also, in that section, the function *DeonSequence( )* demonstrates generating of sequence d_r for given number *r* in accordance with algorithm of proof of the theorem for sufficiency.

The practical realization of the *Theorem* considered here is in the fact that it allows receiving absolutely all the uniformly distributed sequences based on their unique given numbers, where the total amount of them is equal to factorial $n!$. In this case the task of deriving the random sequences is associated with the task of arbitrary producing numeric numbers, which in turn may be reached by using different techniques, for example, based on timer or twister technologies.

5 Discussion

In the previous sections, *Fundamentals* and *Constructions and Results,* the prerequisites and practical realizations have been discussed regarding the simulation of a completed set of sequences consisted in *n* elements. How long might *n* be? For a sequenced RNG, it depends on the qualification of programmers and resources of exact computer to produce the recurrent number in a definite time. Any congruent generator may produce a very long sequence of numbers if the capabilities of the computer for memory size and processor speed are as unlimited as possible. Thus, for this purpose the finding of a required constant isn't difficult. The next random value isn't difficult to find either; however, it's impossible to give any guarantee that after a while it wouldn't be skipped because of nature of sequential algorithm.

Applying this to binary shifting, the simple numbers and other methods for sequenced generating in modern RNG leads to critical dispersions in the receiving of values. This is typical for sequenced RNG. However, some practical cases are known for which random additive components don't vary too much, although they remain random. In such cases, no other way exists than to use similar, but different sequences

that belong to the complete set. Because of such similar sequences in set *D* there may be close numbers univocally in functional of choosing *F*. This is the benefit: sequences may be controlled simply and externally, but they are still random. Just appropriate numbers *r* are required to be chosen, and no generating of complete set *D* is needed. The functions *DeonSequence()* and *DeonNumber()* may fulfill this task for any number with a result as a required random sequence $d_r \in D$.

In the simulation of completed sets of sequences, the added complication is that it's necessary to use factorial $n!$, which is growing very fast even for short sequences. The benefit is that the generation of absolutely all sequences having *n* length is guaranteed once only. Unfortunately, very large values of *n* do not fit in a computer's memory, and that is a limitation for the factorial method. Fortunately, factorial and completed sets of sequences have found large application in different areas of real human activities. Two of them are most commonly encountered in the literature: 1) testing of technical equipment, and 2) planning of evidence-based examinations in medicine.

The testing of technical equipment is an important requirement to verify that complicated technical systems are working correctly. It is quite valuable, for example, for turbine-generator sets, engines for sport cars, energetic aggregates for sea-, aero- and space-crafts, apparatuses working in dynamic and even in dangerous environments, etc. So, simulation of all required test conditions is crucially important. Skipping of some sort of regimes is unallowable due to the cost of production and exploitation of such systems could be very high and custom unique. Moreover, in the case of biomedical technical systems the cost of life is invaluable. It's evident, that we can't get by without completed sets of random sequences. The sequenced RNG don't fit completely because of pseudorandom sequences may occasionally skip the random numbers. But such skipping is absolutely inappropriate in questions of vital importance.

Medical planning of evidence-based examinations is frequently faced with sets of volunteer and patient groups that might be chosen randomly. Surely, the same person can't be involved in two different groups simultaneously due to, for example, the first group is used to check the efficacy of novel medication and the second group is used for placebo validation. Moreover, no volunteers, no patients, no medical staff can be aware of what is given to an exact person except the responsible researchers. This is typical medical planning for verification purposes. The application of RNG here of a general type, which is commonly used in cryptography, is very doubtful. Moreover, for some meaningful tests, the choosing of volunteers may require additional selections to exclude possible speculations in results. So, the skipping of any kind of numbers is inappropriate because of a matter of life and death in decision making processes.

Thus, completeness of all possible sequences is strongly required. No universal RNG exist; therefore in different areas, the complete sets of sequences under full variations of test examinations are demanded.

6 Conclusion

In this article we've considered the questions regarding the main goal of current work, which is how to strictly get the variety of random sequences having exact length, where the random values are to appear only once. For this purpose, a modern approach has been analyzed where the realization provides the production of different RNGs. These generators may be considered from both sides: the theory of random processes and the practical usage target. This leads us to those generators which are capable in realization of completed sets of random sequences. For this aim, we consider the uncertainty of occurrence as a dilemma in choosing of one sequence from all possible as it stands. The proved theorem about strict ranking of the initial set of objects in model $M = (B, D, F)$ resulted in theoretical confirmation of univocal choosing of functional F. The theorem-proving process led us to creating of program functions *DeonSequence( )* and *DeonNumber( )*, which allowed the retrieval of all random sequences $d_r \in D$ while number r was using for this, and no simulation of completed set of sequence D was made.

In section *Introduction* it was mentioned that technical systems could generate random numbers which are close to uniform distribution but unfortunately, not strictly uniform. The verification methods of mathematical statistics may confirm this statement. The novel algorithm demonstrated above to form completed ranked sets of random sequences is different from the approaches that are used in PRNG. In the current work here, proposed complete sets of uniform sequences are indeed consisting in strictly uniform random numbers.

When discussing the received results, it was noted that current RNGs, which realize the conception of consequent generating, allows for the retrieval of random values having some length n, but at the same time they can't guarantee getting them without skipping or with no repeating of random values. By using method has proposed in this paper it's possible to provide the generation of univocal random sequences; however the length of such sequences is limited by computer resources in calculations of factorial $n!$. We believe, in the near future, the simulation of random sequences having completed sets of objects will be in more and more demand.

Ethic, Contribution, Funding and Acknowledgments

This article is original and contains unpublished material. The authors equally contributed in this work, and they have no support or funding to report. The authors are thankful to Matthew Vandenberg, Jacqueline Nolan, Kai Carey and Walter Harrington (University of Arkansas for Medical Sciences, Little Rock, USA) for the proofreading.

References

[Arora et al. 2015] Arora, M., Engles D., and Sharma S. (2015). MDS algorithm for encryption. J. Comp. Sci., 11(3):479-483. DOI: 10.3844/jcssp.2015.479.483

[Banati and Bajaj 2013] Banati, H. and Bajaj, M. (2013). Performance analysis of firefly

algorithm for data clustering. Int. J. Swarm Intell., 1: 19-35. DOI: 10.1504/IJSI.2013.055800

[Blum et al. 1986] Blum, L., Blum, M., and Shub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator. SIAM Journal on Computing, 15(2):364-383. DOI: 10.1137/0215025

[Cai et al. 2016] Cai, C., K.A. Carey, D.A. Nedosekin, Y.A. Menyaev, and M. Sarimollaoglu, *et al.*, 2016a. In Vivo Photoacoustic Flow Cytometry for Early Malaria Diagnosis. Cytometry A, 89A:531-542. DOI: 10.1002/cyto.a.22854

[Cai et al. 2016] Cai, C., D.A. Nedosekin, Y.A. Menyaev, M. Sarimollaoglu, and M.A. Proskurnin, *et al.*, 2016b. Photoacoustic Flow Cytometry for Single Sickle Cell Detection In Vitro and In Vivo. Anal. Cell. Pathol., 2642361:1-11. DOI: 10.1155/2016/2642361

[Cover and Thomas 2006] Cover, T.M. and Thomas, J.A. (2006). Elements of Information Theory. 2nd Ed. John Wiley & Sons, New York. ISBN: 978-0-471-24195-9, pp: 776.

[Dagtas et al. 2004] Dagtas, S., Sarimollaoglu, M., and Iqbal, K. (2004). A Multi-modal Virtual Environment with Text-Independent Real-Time Speaker Identification. Proceedings of the 6th IEEE ISMSE conference. Dec. 13-15, Miami, FL, pp. 557-560. DOI: 10.1109/MMSE.2004.14

[Deon and Menyaev 2016] Deon, A. and Menyaev, Y. (2016). Parametrical Tuning of Twisting Generators. J. Comp. Sci. DOI: 10.3844/jcssp.2016.___.___ [Epub ahead of print] http://thescipub.com/abstract/10.3844/ofsp.10806

[Diffie and Hellman 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. IEEE Trans. Inform. Theory, 22:644-654. DOI: 10.1109/TIT.1976.1055638

[Diffie and Hellman 1979] Diffie, W. and Hellman M. (1979). Privacy and authentication: An introduction to cryptography. Proc. IEEE, 67:397-427. DOI: 10.1109/PROC.1979.11256

[Dodis et al. 2013] Dodis, Y., Pointcheval, D., Ruhault, S., Vergniaud, D., and Wichs, D. (2013). Security analysis of pseudo-random number generators with input: /dev/random is not robust. Proceedings of ACM SIGSAC conference on Computer & communications security. Nov. 4-8, ACM, New York, pp: 647-658. DOI: 10.1145/2508859.2516653

[Dorrendorf et al. 2009] Dorrendorf L., Gutterman Z., and Pinkas B. (2009). Cryptanalysis of the random number generator of the Windows operating system. Journal ACM Transactions on Information and System Security (TISSEC). 13(1), Article No.10. DOI: 10.1145/1609956.1609966

[Evans et al. 2001] Evans, S., Bush, S.F., and Hershey, J. (2001). Information assurance through Kolmogorov complexity. Proceedings of DARPA Information Survivability Conference & amp, Exposition II, Jun. 12-14, IEEE Xplore Press, Anaheim, CA, pp: 322-331 vol.2. DOI: 10.1109/DISCEX.2001.932183

[Feng et al. 2010] Feng, L., Qiu, M.H., Wang, Y.X., Xiang Q.L., Yang Y.F., *et al*. (2010). Fast divisive clustering algorithm using an improved discrete particle swarm optimizer. Pattern Recognit. Lett., 31: 1216-1225. DOI: 10.1016/j.patrec.2010.04.001

[Fister et al. 2013] Fister, I., Jr., I.F., Yang, X.S., and Brest, J. (2013). A comprehensive review of firefly algorithms. Swarm Evolut. Computat., 13: 34-46. DOI: 10.1016/j.swevo.2013.06.001

[Forsati et al. 2013] Forsati, R., Mahdavi, M., Shamsfard, M., and Meybodi, M.R. (2013). Efficient stochastic algorithms for document clustering. Informat. Sci., 220: 269-291.

[Gnedenko 1998] Gnedenko, B. (1998). Theory of Probability. 6th Ed. CRC Press. ISBN-10: 9056995855, pp: 520.

[Hellekalek 1998] Hellekalek, P. (1998). Good random number generators are (not so) easy to find. Math. Comput. Simulat., 46(5-6):485-505. DOI: 10.1016/S0378-4754(98)00078-0

[Hellekalek and Wegenkittl 2003] Hellekalek, P. and Wegenkittl, S. (2003). Empirical evidence concerning AES. ACM T. Model. Comput. S., 13(4):322-333. DOI: 10.1145/945511.945515

[Jain 2010] Jain, A.K. (2010). Data clustering: 50 years beyond K-means. Patt. Recognit. Lett., 31: 651-666. DOI: 10.1016/j.patrec.2009.09.011

[Johnsonbaugh 2008] Johnsonbaugh, R. (2008). Discrete Mathematics. 7th Ed. Pearson Prentice Hall. ISBN-10: 0131354302, pp: 766.

[Juratly et al. 2015] Juratly, M.A., Siegel, E.R., Nedosekin, D.A., Sarimollaoglu, M., Jamshidi-Parsian, A., *et al*. (2015). In vivo long-term monitoring of circulating tumor cells fluctuation during medical interventions. PLoS One, 10(9):e0137613. DOI: 10.1371/journal.pone.0137613.

[Juratly et al. 2016] Juratly, M.A., Y.A. Menyaev, M. Sarimollaoglu, E.R. Siegel, D.A. Nedosekin, *et al.*, 2016. Real-Time Label-Free Embolus Detection Using In Vivo Photoacoustic Flow Cytometry. PLoS One, 11(5):e0156269. DOI: 10.1371/journal.pone.0156269

[Karaboga and Ozturk 2011] Karaboga, D. and Ozturk, C. (2011). A novel clustering approach: Artificial Bee Colony (ABC) algorithm. Applied Soft Comput., 11: 625-657. DOI: 10.1016/j.asoc.2009.12.025

[Karloff and Raghavan 1993] Karloff H. and Raghavan P. (1993). Randomized algorithms and pseudorandom numbers. Journal of the ACM (JACM). 40(3):454-476. DOI: 10.1145/174130.174132

[Kasdin 1995] Kasdin, N.J. (1995). Discrete simulation of colored noise and stochastic processes and $1/f^{\alpha}$ power law noise generation. Proc. IEEE. 83(5):802-827. DOI: 10.1109/5.381848

[Kolmogorov 1968] Kolmogorov, A.N. (1968). Three approaches to the quantitative definition of information. Int. J. Comput. Math., 2(1-4):157-168. DOI: 10.1080/00207166808803030

[Kolmogorov and Fomin 1999] Kolmogorov, A.N. and Fomin, S.V. (1999). Elements of the Theory of Functions and Functional Analysis. Dover Publication. Mineola, NY, ISBN-10: 0486406830, pp: 128.

[Leeb and Wegenkittl 1997] Leeb, H. and Wegenkittl, S. (1997). Inversive and Linear Congruential Pseudorandom Number Generators in Empirical Tests. ACM TOMACS, 7(2):272-286. DOI: 10.1145/249204.249208

[Li and Vitanyi 2008] Li, M. and Vitanyi, P. (2008). An Introduction to Kolmogorov Complexity and its Applications. 3rd Ed. Springer-Verlag, New York. ISBN: 978-1-4899-8445-6, pp: 790.

[Mani and Derick 2010] Mani, A. and Derick, A. (2010). An algorithm to reduce the size of cipher text. Global J. Comput. Sci. Technol., 10: 50-54.

[Manning et al. 2008] Manning, C.D., Raghavan, P. and Schütze, H. (2008). Introduction to Information Retrieval. 1 Edn., Cambridge University Press, New York, ISBN-10: 0521865719, pp: 482.

[Matsumoto and Nishimura 1998] Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: a 623-dimensionnally equidistributed uniform pseudorandom number generator. ACM TOMACS, 8(1):3-30. DOI: 10.1145/272991.272995

[Maurer 1992] Maurer, U.M. (1992). A universal statistical test for random bit generators. J.

Cryptology, 5(2):89-105.

[McConnell 2002] McConnell, M. (2002). Information Assurance in the twenty-first century. IEEE Comput., 35: 16-19. DOI: 10.1109/MC.2002.1012425

[Menyaev et al. 2013] Menyaev, Y.A., Nedosekin, D.A., Sarimollaoglu, M., Juratli, M.A., Galanzha, E.I., *et al*. (2013). Optical clearing in photoacoustic flowcytometry. Biomed. Opt. Express, 4(12):3030-41. DOI: 10.1364/BOE.4.003030.

[Menyaev et al. 2016] Menyaev, Y.A., K.A. Carey, D.A. Nedosekin, M. Sarimollaoglu, and E.I. Galanzha *et al.*, 2016. Preclinical photoacoustic models: application for ultrasensitive single cell malaria diagnosis in large vein and artery. Biomed. Opt. Express, 7(9):3643-58. DOI: 10.1364/BOE.7.003643

[Menyaev and Zharov 2006] Menyaev, Y.A. and Zharov, V.P. (2006). Experience in Development of Therapeutic Photomatrix Equipment. Biomedical Engineering, 40(2):57-63. DOI: 10.1007/s10527-006-0042-6

[Menyaev and Zharov 2006] Menyaev, Y.A. and Zharov, V.P. (2006). Experience in the Use of Therapeutic Photomatrix Equipment. Biomedical Engineering, 40(3):144-147. DOI: 10.1007/s10527-006-0064-0

[Menyaev et al. 2006] Menyaev Y.A., V.P. Zharov, E.A. Mishanin, A.P. Kuzmich, S.E. Bessonov, 2006. Combined photovacuum therapy of copulative dysfunction. Proc. SPIE, 6078, pp.241-248. DOI: 10.1117/12.656713

[Miklós et al. 2009] Miklós, I., Novák, A., Satija, R., Lyngs, R. and Hein, J. (2009). Stochastic models of sequence evolution including insertion-deletion events. Stat. Methods Med. Res. 18(5):453-485. DOI: 10.1177/0962280208099500

[Miner et al. 2012] Miner, G., Elder, J., Fast, A., Hill, T., Nisbet, R., *et al*. (2012). Practical Text Mining and Statistical Analysis for Non-Structured Text Data Applications. 1st Edn., Academic Press, Elsevier, ISBN-01: 012386979X, pp: 1000.

[Nandy et al. 2012] Nandy, S., Sarkar, P.P. and Das, A. (2012). Analysis of a nature inspired firefly algorithm based back propagation neural network training. Int. J. Comput. Applic., 43: 8-16. DOI: 10.5120/6401-8339

[Nishimura 2000] Nishimura, T. (2000). Tables of 64-bit Mersenne Twisters. ACM TOMACS, 10(4):348-357. DOI: 10.1145/369534.369540

[O'Donnell et al. 2005] O'Donnell, C.W., Suh, G.E. and Devadas, S. (2005). PUF-Based Random Number Generation. Technical report, MIT CSAIL CSG Technical Memo 481.

[Park and Miller 1988] Park, S.K. and Miller, K.W. (1988). Random number generators: good ones are hard to find. Commun. ACM., 31(10):1192-1201. DOI: 10.1145/63039.63042

[Sarimollaoglu et al. 2011] Sarimollaoglu M., Nedosekin D.A., Simanovsky Y., Galanzha E.I., Zharov V.P. (2011). In vivo photoacoustic time-of-flight velocity measurement of single cells and nanoparticles. Opt. Lett. 36(20):4086-4088. DOI: 10.1364/OL.36.004086

[Sarimollaoglu et al. 2014] Sarimollaoglu, M., Nedosekin, D.A., Menyaev, Y.A., Juratly, M.A. and Zharov, V.P. (2014). Nonlinear photoacoustic signal amplification from single targets in absorption background. Photoacoustics, 2(1):1-11. DOI: 10.1016/j.pacs.2013.11.002

[Schildt 2010] Schildt, H. (2010). C# 4.0: The Complete Reference. The McGraw-Hill Companies. New York. ISBN-10: 007174116X, pp: 949.

[Shannon 1948] Shannon, C.E. 1948. A mathematical theory of communication. Bell System

Technical Journal, 27(4):623-656. DOI: 10.1002/j.1538-7305.1948.tb00917.x

[Shannon and Weaver 1963] Shannon, C.E. and Weaver, W. (1963). The Mathematical Theory of Communication. University of Illinois Press, Urbana, IL. ISBN-10: 0252725484, pp: 127.

[Song et al 2006] Song, Y., Liu, C., Malmberg, R.L., He, C. and Cai, L. (2006). Memory efficient alignment between RNA sequences and stochastic grammar models of pseudoknots. Int. J. Bioinform. Res. Appl., 2(3):289-304.

[Storm and Price 1997] Storm, R. and Price, K. (1997). Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. J. Global Optimizat., 11: 341-359. DOI: 10.1023/A:1008202821328

[Suh et al. 2004] Suh, G.E., O'Donnell, C.W., Sachdev, I. and Devadas, S. (2004). Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. Technical report, MIT CSAIL CSG Technical Memo 483.

[Tan et al. 2011] Tan, S.C., Ting, K.M. and Teng, S.W. (2011). A general stochastic clustering method for automatic cluster discovery. Patt. Recognit., 44: 2786-2799. DOI: 10.1016/j.patcog.2011.04.001

[Turing 1950] Turing, A.M. (1950). Computing Machinery and Intelligence. Mind, 49(236):433-460. DOI: 10.1093/mind/LIX.236.433

[Velmurugan and Santhanam 2010] Velmurugan, T. and Santhanam, T. (2010). Computational complexity between k-means and k-medoids clustering algorithms for normal and uniform distributions of data points. J. Comput. Sci., 6:363-368. DOI: 10.3844/JCSSP.2010.363.368

[Waerden 1991] Waerden, B.L. van der. (1991). Algebra: Volume I. Springer-Verlag, New York. ISBN: 978-0-387-40624-4, pp: 265.

[Waerden 1991] Waerden, B.L. van der. (1991). Algebra: Volume II. Springer-Verlag, New York. ISBN: 978-0-387-40625-1, pp: 284.

[Wallace 1996] Wallace C.S. (1996) Fast pseudorandom generators for normal and exponential variates. Journal ACM Transactions on Mathematical Software (TOMS). 22(1):119-127. DOI: 10.1145/225545.225554

[Wegenkittl 2001] Wegenkittl, S. (2001). Entropy estimators and serial tests for ergodic chains. IEEE Inform. Theory, 47(6):2480-2489. DOI: 10.1109/18.945259

[Yang 2010] Yang, X.S. (2010). Nature-Inspired Metaheuristic Algorithms. 2nd Edn., Luniver Press, United Kingdom, ISBN-10: 978-1-905986-28-6, pp: 116.

[Yang 2010] Yang, X.S. (2010). Firefly algorithm, stochastic test functions and design optimization. Int. J. Bio-Inspired Comput., 2: 78-84. DOI: 10.1504/IJBIC.2010.032124

[Yujian and Liye 2010] Yujian, L. and Liye, X. (2010). Unweighted multiple group method with arithmetic mean. Proceedings of the IEEE 5th International Conference on Bio-Inspired Computing: Theories and Applications, Sept. 23-26, IEEE Xplore Press, Changsha, pp: 830-34. DOI: 10.1109/BICTA.2010.5645232

[Zharov et al. 2001] Zharov, V.P., Menyaev Y.A., Gorchak Y.Y., Utkina K.V., and Y.A. Menyaev. (2001). Methods for photoultrasonic treatment of festering wounds in oncological patients. Crit. Rev. Biomed. Eng., 29(1):111-24. DOI: 10.1615/CritRevBiomedEng.v29.i1.50

[Zharov et al. 2001] Zharov, V.P., Menyaev Y.A., Kabisov R.K., Al'kov S.V., and Nesterov A.V. *et al*. (2001). Design and application of low-frequency ultrasound and its combination with laser radiation in surgery and therapy. Crit. Rev. Biomed. Eng., 29(3):502-19. DOI: 10.1615/CritRevBiomedEng.v29.i3.130