Title

Random Number Generators

Research Question

HOW DOES THE PERFORMANCE OF DIFFERENT JAVA PSEUDO RANDOM NUMBER GENERATORS MAKES THEM SUITABLE FOR THEIR APPLICATIONS?

Word Count
3,930

Table of Contents

Table of Tables

Table of Figures

1. Introduction

Pseudo Random Number Generators are one of the few libraries which can be found in almost every programming language. While there exist flaws in Pseudo Random Number Generators (PRNG), they still find use in a number of applications since it is almost impossible to generate true random numbers at will. In True Random Number Generators (TRNG), the generation depends on external hardware and surroundings. This restricts the possibility of random number generation in the various areas where it is applied. Furthermore, a number of applications require that these random numbers are generated at a specific speed, a requirement which cannot be achieved by TRNGS. The PRNGS have a few obvious flaws in terms of the randomness in the generated numbers and the set of large numbers having limited uniformity. But, the speed at which these numbers are generated and the recent improvements in technology give the Pseudo Generators a crucial edge over the True Generators in many applications. Hence, a compromise is done between the time and quality of the generated random numbers.

It is evident that different Pseudo Generators address different categories of applications, which are dependent on the quality of data and time taken to achieve them. Hence, it is imperative that we determine the suitable generators for each scenario where these random numbers are applied. This leads to my research question:

HOW DOES THE PERFORMANCE OF DIFFERENT JAVA PSEUDO RANDOM NUMBER GENERATORS MAKES THEM SUITABLE FOR THEIR APPLICATIONS?

This investigation solely focuses on Pseudo Random Number Generators which are present as a part of Java's library or can be executed in Java. Due to paucity of time and huge resources required to investigate True Generators, this study only focuses on Pseudo Generators. Furthermore, as Java based random number generators are found in various well known fields such as gaming and allow the opportunity to conduct a number of elaborate quality and speed tests, they constituted the primary focus of this investigation.

Since different scenarios require random numbers of varying quality and speed, an analysis of different Pseudo Generators would ensure that the proper generator is used for a particular application. Given there exist a number of Pseudo Generators in the market, this research, by comparing popular generators, provides clarity on the rationale behind their applications.

To investigate various Pseudo Generators and how these fare with reference to the two components mentioned above (Quality and Speed), multiple tests determining randomness and production times of the numbers for the chosen generators were conducted. These tests were conducted with the help of the Java Randomness Test Tool and the obtained results were discussed on mathematical and logical grounds.

2. Background Information

2.1 Random Number Generators

Random numbers follows three rules: (1) They are in a sequence, (2) numbers are distributed uniformly across a specific period, and (3) it is almost impossible to forecast the values in the distribution looking at the values already present. These random numbers find applications in fields such as gambling, gaming and cryptography.

There are two ways to generate random numbers, namely, True Random Number Generators and Pseudo Random Number Generators. The Hardware Random Number Generator (HRNG) or True Random Number Generator is a gadget that produces random numbers from physical phenomenon, instead of algorithmic methods. Infinitesimal phenomenon such as *thermal noise* and *white noise*, the *photoelectric effect*, involving a *beam splitter*, and other *quantum* phenomena usually underlie such gadgets. These different infinitesimal happenings result in the production of minuscule, "buzz" signals which are *statistically random*.

A Pseudo Random Number Generator is a program written for, and used in, probability and statistics applications when large quantities of random digits are needed.[1] A Pseudo Generator is able to generate large sets of random numbers at a speed which is desirable for the various instances where it is applied.

[1] Rouse, Margaret. "What Is Pseudo-Random Number Generator (PRNG)?". *Https://Whatis.Techtarget.Com/*, March 2011, https://whatis.techtarget.com/definition/pseudo-random-number-generator-PRNG. Accessed 14 December 2019.

While there have been a number of developments in Pseudo Generators, which has vastly improved the randomness of the numbers produced, the very existence of the algorithm makes it inevitable that the prediction of the next number is possible at least in theory.

2.2 Applications of Random Numbers

There are a number of situations where random numbers are widely used.

Modern Gambling is one of these which has gained wide acceptance with the advent of Internet and E-commerce technologies. Old methods of rolling a die and shuffling the cards are no more used because of the large amount of time required for the generation of values. Gambling occurs at a larger scale requiring quicker and more efficient methods of randomness. While some casinos may still use True Generators despite its inherent lack of speed, the online gambling community has moved primarily towards Pseudo Generators.

Video games are another area where random numbers are being widely used. While the gaming community is divided over whether the randomness factor ought to come in play while deciding the outcome in some games, the fact is that the random numbers still continue to play a big role in video games. The Pseudo Generators find better usage in these applications.

Pseudo Random Number Generators also find application in cryptography, where random numbers are generated for the initialisation vector and other cryptographic keys used during the process of encryption. The speed and quality are characteristics which play an important role here and hence a combination of both, True and Pseudo Generators are used.

Here, one can see that the various applications of the Pseudo Generators have different needs for the random numbers generated. While certain applications such as Video Games prioritize speed over quality, other applications such as Gambling demand random numbers of higher quality regardless of production times. The above description cements the fact that a single Pseudo Random Number Generator may not be able to meet the needs or satisfy all the applications.

3. Experiment Methodology

This investigation is carried out using primary experimental data. Four different Pseudo Generators were chosen and were subject to a number of statistical tests, mentioned below, measuring randomness and speed. These tests were carried out by loading the appropriate Pseudo Random Number Generator in the Java Randomness Test Tool, running the appropriate test, and then obtaining the experimental data. The obtained experimental data provides information regarding the randomness (quality) of the generated random numbers and the speed of the generators. Lack of availability of secondary data to fulfil the investigation made an experiment methodology necessary. However, the employed methodology is subject to a few limitations. Paucity of time and the tedious data collection process, due to a large number of tests to be performed, allowed only a few generators to be analysed.

3.1 Dependent Variables

3.1.1 Randomness

The quality of the generated random numbers is determined by measuring randomness. The randomness of the different Pseudo Generators was recorded as part of the various tests conducted. For checking the randomness of a generator emphasis is laid on the P-value, which in statistics, helps us to determine the validity of a null hypothesis. This essentially checks to what extent the output of a generator is random. If the value of P is less than *.05* or greater than *0.95*, the null hypothesis is considered as invalid and the test failed. Any value between *.05* and *0.1* or between *0.9* and *0.95* are considered as a close pass. Each generator is taken through multiple trials for each test and they are ranked with reference to their fails and close passes.

3.1.2 Speed

The quality (randomness) of the generated random numbers is one aspect of deciding on the suitability of the generators for a specific application. The second aspect is the speed at which these numbers are generated. The speed of the Pseudo Random Number Generators was determined by measuring the time taken by the generators to generate random numbers. This time was measured in seconds.

3.2 List of Pseudo Random Number Generators

The number of Pseudo Random Number Generators available are huge. Considering the limitation of resources, I had to choose a few. Java based Pseudo Random Number Generators were chosen due to their compatibility with the Java Randomness Testing Tool which would allow conducting the numerous tests detailed in the next section. Each chosen generator has applications which are distinct from another. This ensured that the obtained results would be unique for each generator and would allow for analysis which fulfils the investigation. Each of the generators described below has its own characteristics as explained under each of them.

3.2.1 Java util.random

The Java.util.random class implements what is generally called a linear congruential generator (LCG). A linear congruential generator is essentially a formula of the following form:

$$number_{i+1} = \left(a \times number_{i} + c\right) mod\ (m)$$

This is a formula which has been in use for a very long time. It starts with a random seed which gets multiplied with some fixed number *a*, this is then followed by the addition of another constant *c,* and then finally the *mod m* of the result is taken.

The selection of seed is critical and it is normally a system parameter, such as the number of nanoseconds since the system was powered on is unpredictable enough and acts as a good value for the seed. This method or its variants were widely used in old home computers and is part of the library functions of many compilers such as C.

3.2.2 Java Secure Random

If the need is to produce more secured random numbers like the ones used in cryptography, then we go for Java Secure Random generator. Here the seed selected is much more unpredictable and it is run through a series of Pseudo Random Number Generators like SHA1 Random to produce a much higher quality output of random numbers. It takes numerous years to crack a number generated using this process with the currently available computers and thus provides a good level of security.

3.2.3 Mersenne Twister

The Mersenne Twister finds application in the complex modelling of natural events. The algorithm starts with a seed of *19937* bit long arranged in an array consisting of *624* components , each being *32* bits long. The unused bits in the array (*31* in number) move through during the process of calculation of next seed and thus eliminate the possibility of a *1* bit storage cell replacing an array element. A large linear feedback shift register where the linear function of the preceding state is the input constitutes Mersenne Twister. This method is not considered good enough for cryptography but does find its use in various other applications.

3.2.4 SHA1 Random

SHA1 Random uses the SHA1 hash function for the generation of random numbers. The hash function is a one-way function, that is the input cannot be determined from the output. The seed is picked up from the operating system and it works at *256* bits which makes this number generator highly secure and the prediction of the generated numbers very difficult.

3.3 Tests used to measure Randomness and Speed

Randomness is a very important aspect in solving many philosophical and theoretical questions. The tests normally look for a recognisable pattern or finding regularities in the data set. Off late, there a lot of tests developed for checking the randomness and two of these find special mention here.

The Chi Square test is one of the first and most usually utilized test for randomness. Diehard tests are a collection of different tests used to check randomness in cryptographic situations where the application of randomness is much more stringent. Several Diehard tests are used to get a more comprehensive result.

Along with randomness, the speed of the different Pseudo Random Number Generators was also measured with the help of Speed testing. To ensure that the results obtained through Speed testing are precise enough to be analysed, five trials for the speed of each number generator were obtained which were then used to calculate an average.

3.3.1 Chi Square test

The Chi Square test is intended to test how likely it is that an observed distribution is due to chance.[2] It creates a "goodness of fit" statistic, because it finds out to what extent the observed distribution of data matches with the expected distribution if the variables are independent. It is seen that smaller the value of the Chi Square test statistic, higher the match between observed data and expected data. That is, there exists a relation. On the other hand, higher the value of the Chi Square test statistic, lower the match and existence of a well-defined relation.

[2] "Tutorial: Pearson's Chi-Square Test For Independence". 2008, *Ling.Upenn.Edu*, https://www.ling.upenn.edu/~clight/chisquared.htm. Accessed 22 December 2019.

The probability of a Chi Square statistic having two degrees of freedom quite larger than *19.58* is defined as the *P*-value. A Chi Square Distribution Calculator is employed to compute $P(X^2 > 19.58) = 0.0001$ and interpret results. The hypothesis is null and cannot be accepted as the *P*-value *(0.0001)* is quite small compared to the significance level *(0.05)*.

This test alongside the Kolmogorov–Smirnov test, is utilized broadly in the different tests referenced below.

3.3.2 Run test

Run test of randomness is a statistical test that is used to check the randomness in data and it is non parametric in nature.[3] This tests the auto correlation in data which means checking whether the data has any dependence with the tagged value.

This test is widely used in the stock market, to check whether the movement in share prices are random or is there a specific pattern. This test finds application to determine whether stock prices of a company are random or following a pattern.

3.3.3 Birthday Space test

The first level test selects at random $m = 210$ ”birthdays” from a ”year” of $n = 224$ days.[4] At the next level, the test calculates the gap between the birthdays for each pair of birthdays which are sequential. Then it finds out the number of pairs of birthdays where the gap is more than one day which are sequential and this is used as the *K* value. Now *K* is expected to have a close to Poisson distribution with the parameter *l* is equal to 16. This test determines 200 values of *Km* (*m= 1, 2, ..., 200*). To get the

[3] "Runs Test Of Randomness - Statistics Solutions". *Statistics Solutions*, https://www.statisticssolutions.com/runs-test-of-randomness/. Accessed 22 December 2019.
[4] "Birthday Spacing Test". *Software.Intel.Com*, 26 December 2019, https://software.intel.com/en-us/mkl-vsnotes-birthday-spacing-test. Accessed 28 December 2019

P-value *P*, the test uses the Chi-Square goodness-of-fit test to the calculated values.

3.3.4 Squeeze test

Random integers are floated to get uniforms on *[0,1)*.[5] The test begins with $k = 2^{31} = 2147483648$ to find the value of *j*. The value of *j* essentially defines the number of repetitions required to reduce *k* to *1*. This is carried out with the help of the reduction $k = ceiling(k \times U)$, where the value of *U* is found by integers which are floating from the tested file. The *j* values are calculated *100000* times. Finally, the tally for the amount of times the value of *j* was ≤ *6, 7, ..., 47,* ≥ *48* are employed to provide a Chi Square test for frequencies of the cell.

3.3.5 Minimum Distance test

This test chooses *8000* points in a square at random and measures the distance between all pairs of points.[6] You will get an exponential distribution of these points.

The variance from the expected value is then used to calculate the *P*-value.

3.3.6 32×32 Binary Rank test

Used to test those pseudo random number generators with an integer output.[7] Initially, the test selects groups of *32* bits from each output of the generator. Then, a 32×32 binary matrix is formed from these *32* groups. From the consecutive

---

[5] *Stojanov, Georgi, and Andrea Kulakov. ICT Innovations 2016. Springer, 2017, p. 86. Accessed 28 December 2019.*

[6] Bellamy, James. *Randomness Of D Sequences Via Diehard Testing*. p. 3, https://arxiv.org/ftp/arxiv/papers/1312/1312.3618.pdf. Accessed 27 December 2019.

[7] "Rank Of 32X32 Binary Matrices Test". *Software.Intel.Com*, 26 December 2019, https://software.intel.com/en-us/mkl-vsnotes-rank-of-32x32-binary-matrices-test. Accessed 28 December 2019.

components of the numbers generated by a Pseudo Random Number Generator, *40000* of such matrices are created.

Matrices with the rank of *32, 31, 30,* or less than *30* are then calculated. All possible matrix ranks are divided into *4* groups. This gives a Chi Square distribution to which goodness of fit test is applied. The test result is the *P*-value.

3.3.7 The Overlapping Pairs Sparse Occupancy (OPSO) test

OPSO (Overlapping-Pairs-Sparse-Occupancy) is a test for randomness. This test takes words of size *2* letters from an alphabet group having *1024* letters. The random number to be tested is a *32* bit integer and each letter is formed by a specified *10* bits from this sequence. It actually counts the number of *2*-letter words which do not appear in the sequence of 2^{21} (overlapping) *2*-letter words. This count will be close to a normal distribution with a mean value of *141909* and standard deviation value of *290*. The standard normal variable would be ($words\ missing\ -\ 141909) \div 290$. The test uses *32* bits at a time from the data set and uses a specified set of ten consecutive bits. It then restarts the file for the next designated *10* bits, and the process continues.

3.3.8 Speed testing

Speed testing is used to determine each Pseudo Random Number Generator's speed. This is carried out by measuring the time taken for a generator, in seconds, for the production of a particular set of random numbers

4. Experimental Results

The results of the various tests, mentioned above, on the Pseudo Random Number Generators are provided in the tables below. In the data tables, a large number of decimal places have been used so as to preserve accuracy in the obtained data and to also ensure that the tiny difference between some of the values is noticed. After each data table (except Speed testing), a bar graph representing the readings of the table is also mentioned. The data obtained from Speed testing was not represented on a bar chart as the results do not require segregation into close passes and fails. These graphs indicate the number of close passes

$(0.1 < P - value < 0.5 \; or \; 0.9 < P - value > 0.95)$ and fails

$(P - value < 0.5 \; or \; P - value > 0.95)$ on the *x*-axis and the appropriate Pseudo Random Number Generator on the *y*-axis. These bar charts aid in following the number of close passes and fails for each test which then become important for the analysis of the obtained readings. In the tables, the fails are marked in red and close passes are marked in yellow. Fails are doubled while plotting on the graph to show their significance as compared to close passes.

.


JRandTest
File Edit Help
Java Randomness Tests Copyright (c) 2005, Zur Aougav
Algorithm:
Java Random
Public key file...
Private key file...
File...
<< < Run... > >>
Runs
Runs test listing
The RUNS test for file Java Random @
(Up and down runs in a sequence of 10000 numbers)
Set 1
runs up; ks test for 10 p's: 0.8091616197026665
runs down; ks test for 10 p's: 0.2587210285500271
Set 2
runs up; ks test for 10 p's: 0.8109790174834024
runs down; ks test for 10 p's: 0.5753011187185226
The RUNS test for file Java Random @
(Up and down runs in a sequence of 10000 numbers)
Set 1
runs up; ks test for 10 p's: 0.956644398306207
runs down; ks test for 10 p's: 0.992973271990569
Set 2
runs up; ks test for 10 p's: 0.4514086539108239
runs down; ks test for 10 p's: 0.4141819638183566

4.1 Tabulated Results

4.1.1 Runs test

	Java.util.random	Java Secure Random	Mersenne Twister	SHA1 Random
Set 1 Up	0.956644492100	0.895174126310	0.181921051910	0.011593575917
Set 1 Down	0.983682719000	0.356234229230	0.048712637844	0.092468114525
Set 2 Up	0.456323953900	0.695872138180	0.857794751094	0.046431686380
Set 2 Down	0.419141863800	0.185351052400	0.685468185468	0.355200030724

Table 1: - Raw data for Runs test

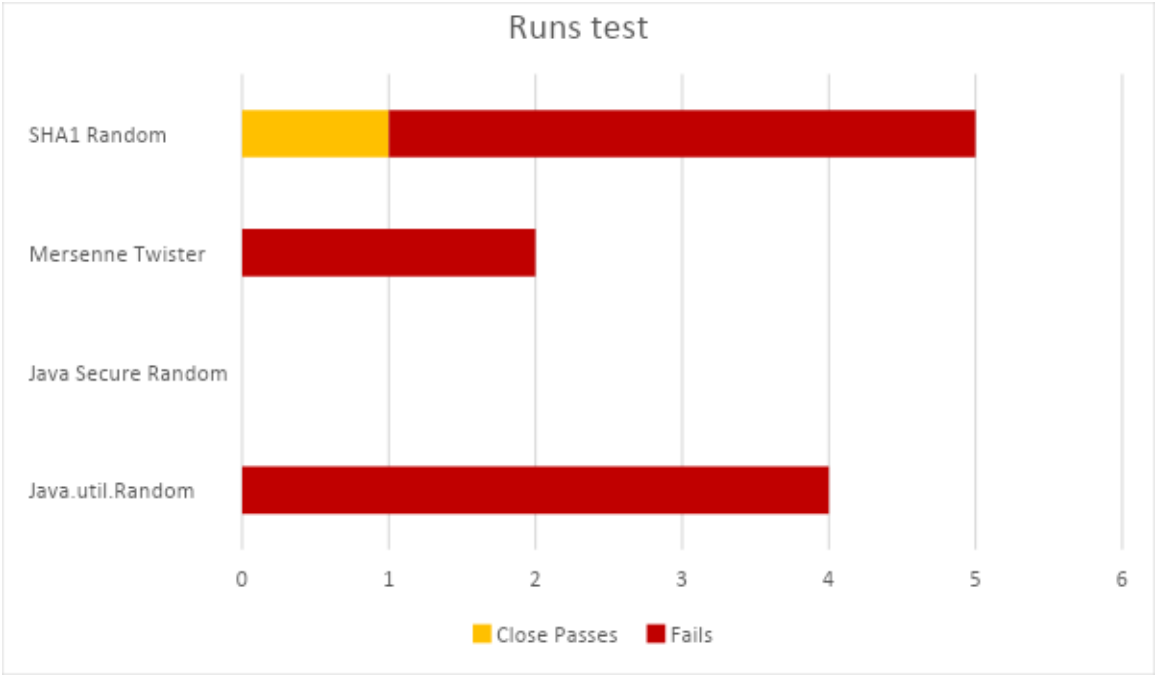

Figure 3: - Graph for Runs test

4.1.2 Birthday Spacing test

	Java.util.random	Java Secure Random	Mersenne Twister	SHA1 Random
1 to 24	0.2137	0.2936	0.0428	0.7291
2 to 25	0.2821	0.6137	0.2837	0.7396
3 to 26	0.6307	0.2388	0.3673	0.7567
4 to 27	0.1328	0.5612	0.1126	0.3298
5 to 28	0.1329	0.2391	0.8740	0.0536
6 to 29	0.0638	0.6566	0.4168	0.8005
7 to 30	0.0048	0.6853	0.8569	0.1785
8 to 31	0.6326	0.2591	0.2382	0.2470
9 to 32	0.1761	0.8329	0.6383	0.1286
Kolmogorov–Smirnov test P-value	0.0124	0.5493	0.5621	0.6365

Table 2: - Raw data for Birthday Spacing test

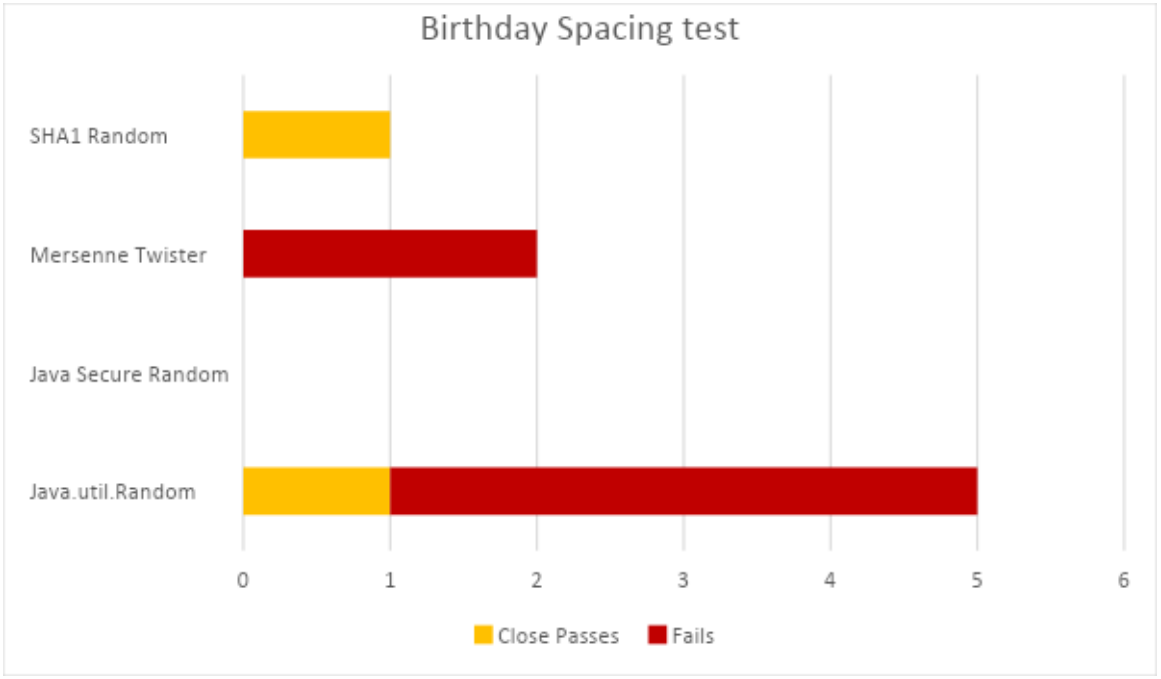

Figure 4: - Graph for Birthday Spacing test

4.1.3 Min Distance test

Serial Number	Java.util.random	Java Secure Random	Mersenne Twister	SHA1 Random
1	0.7806	0.6828	0.5621	0.4126
2	0.3126	0.7247	0.6893	0.6826
3	0.4814	0.5768	0.6326	0.8611
4	0.4387	0.7329	0.5796	0.7096
5	0.3179	0.4468	0.4891	0.7621
6	0.0621	0.4373	0.8698	0.3126

Table 3: - Raw data table for Min Distance test

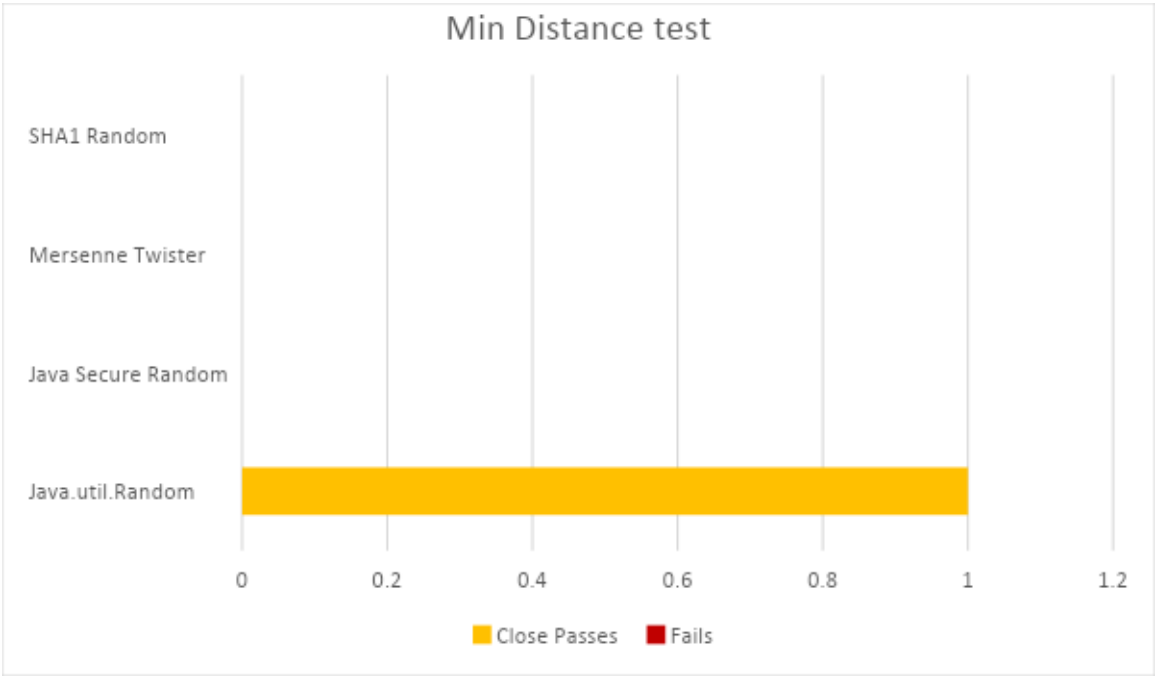

Figure 5: - Graph for Min Distance test

4.1.4 Overlapping Pair Sparse Occupancy test

Bits used	Java.util.random	Java Secure Random	Mersenne Twitter	SHA1 Random
23 to 32	1	0.2691	0.114	0.4986
22 to 31	0.98	0.0709	0.0711	0.6325
21 to 30	0.9621	0.6581	0.9402	0.4681
20 to 29	0.8963	0.5434	0.7637	0.4783
19 to 28	0.7891	0.1366	0.3077	0.7086
18 to 27	0.9909	0.6145	0.6596	0.5033
17 to 26	0.2391	0.7786	0.8845	0.5815
16 to 25	0.3612	0.4391	0.6065	0.8368
15 to 24	0.2826	0.147	0.621	0.5684
14 to 23	0.3299	0.4621	0.142	0.3326
13 to 22	0.4216	0.1804	0.0857	0.6176
12 to 21	0.9986	0.3826	0.5193	0.0239
11 to 20	0.1326	0.2621	0.5237	0.9865
10 to 19	0.1068	0.4256	0.3948	0.416
9 to 18	0.7138	0.2381	0.1656	0.8961
8 to 17	0.3162	0.8187	0.712	0.4937
7 to 16	0.3489	0.0028	0.2643	0.9236
6 to 15	0.5135	0.8891	0.1872	0.4729
5 to 14	0.2106	0.8526	0.9743	0.6189
4 to 13	0.7681	0.994	0.6353	0.9026
3 to 12	0.7981	0.4675	0.7872	0.6484
2 to 11	0.8265	0.2678	0.0058	0.6732
1 to 10	0.4162	0.1754	0.1328	0.8821

Table 4: - Raw data table of Overlapping Pair Sparse Occupancy test

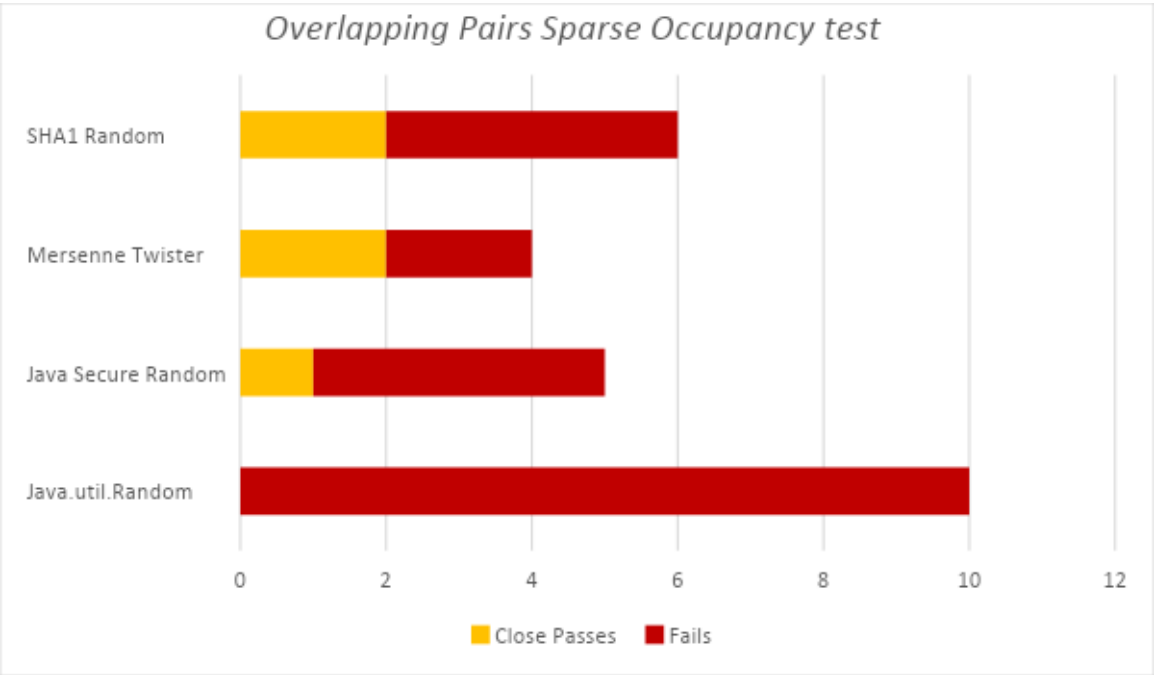

Figure 6: - Graph for Overlapping Pairs Sparse Occupancy test

4.1.5 32x32 Binary Rank test

Serial Number	Java.util.random	Java Secure Random	Mersenne Twister	SHA1 Random
1	0.4126	0.6568	0.5983	0.3275
2	0.7386	0.5022	0.3486	0.7168
3	0.6128	0.8326	0.5063	0.2932
4	0.3693	0.2184	0.4923	0.6894
5	0.2607	0.1943	0.2832	0.4659
6	0.9793	0.6869	0.4641	0.5367

Table 5: - Raw data table of Binary Rank test

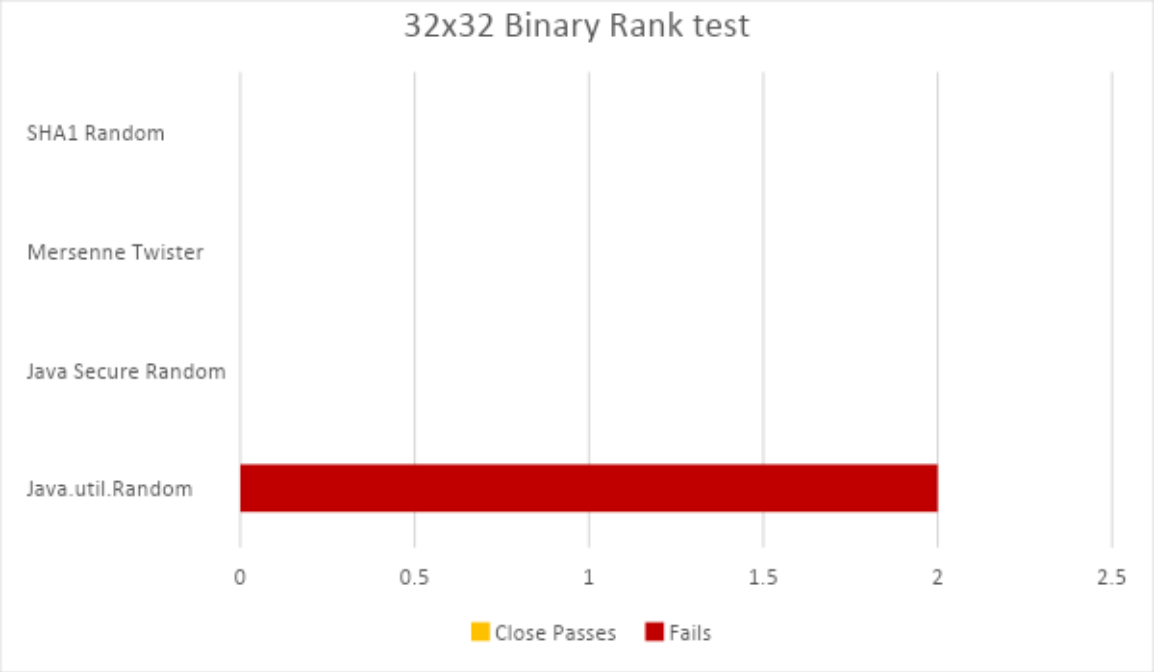

Figure 7: - 32x32 Binary rank test

4.1.6 Squeeze test

Serial Number	Java.util.random	Java Secure Random	Mersenne Twister	SHA1 Random
1	0.1863	0.6932	0.8691	0.7342
2	0.3698	0.8625	0.5326	0.3126
3	0.4216	0.6988	0.1561	0.8691
4	0.9263	0.2563	0.4693	0.9898
5	0.919	0.9862	0.8326	0.6126
6	0.8926	0.1876	0.5091	0.8613
7	0.7186	0.5465	0.8163	0.0573
8	0.3052	0.9829	0.8091	0.1926
9	0.6919	0.3187	0.9626	0.6893
10	0.4876	0.9025	0.7126	0.8805
11	0.8693	0.9391	0.2391	0.9426

Table 6: - Raw data table of Squeeze test

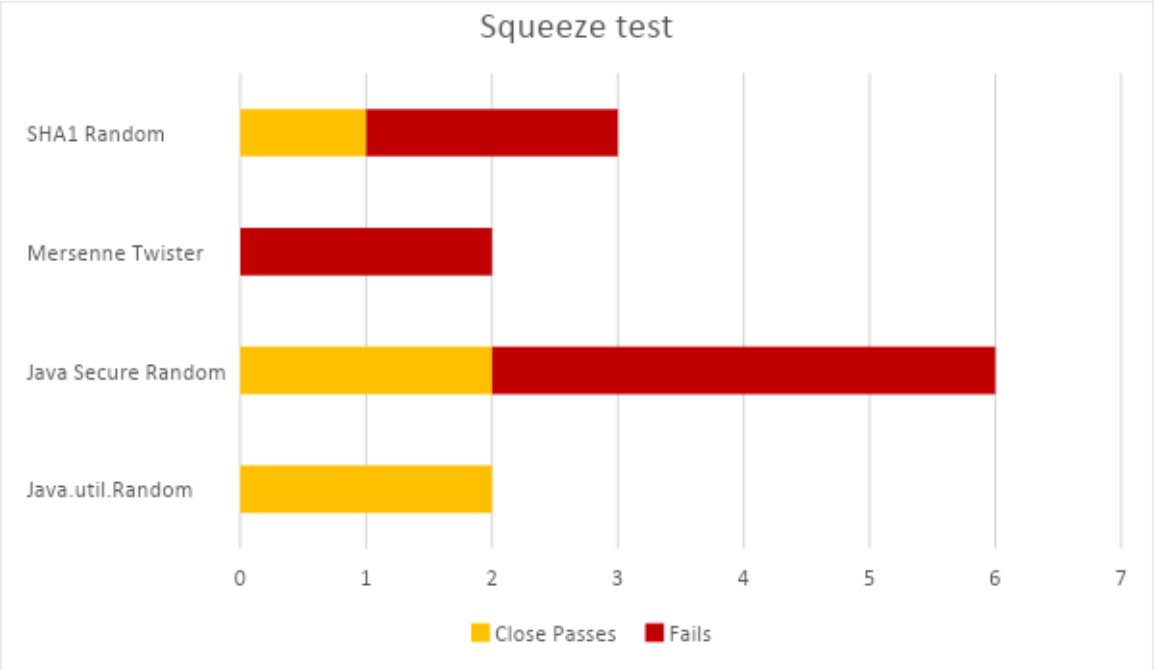

Figure 8: - Graph for Squeeze test

4.1.7 Speed testing

The results of the Speed testing are tabulated below for five trials and an average.

Java Random	Java Secure Random	Mersenne Twister	SHA1 Random
0.014 s	0.10 s	0.0015 s	0.10 s
0.017 s	0.14 s	0.0016 s	0.12 s
0.015 s	0.10 s	0.0016 s	0.07 s
0.018 s	0.11 s	0.0017 s	0.07 s
0.019 s	0.11 s	0.0019 s	0.08 s
0.0166 s	0.112 s	0.0017 s	0.088 s

Table 7: - Raw data table of Speed testing

5. Analysis

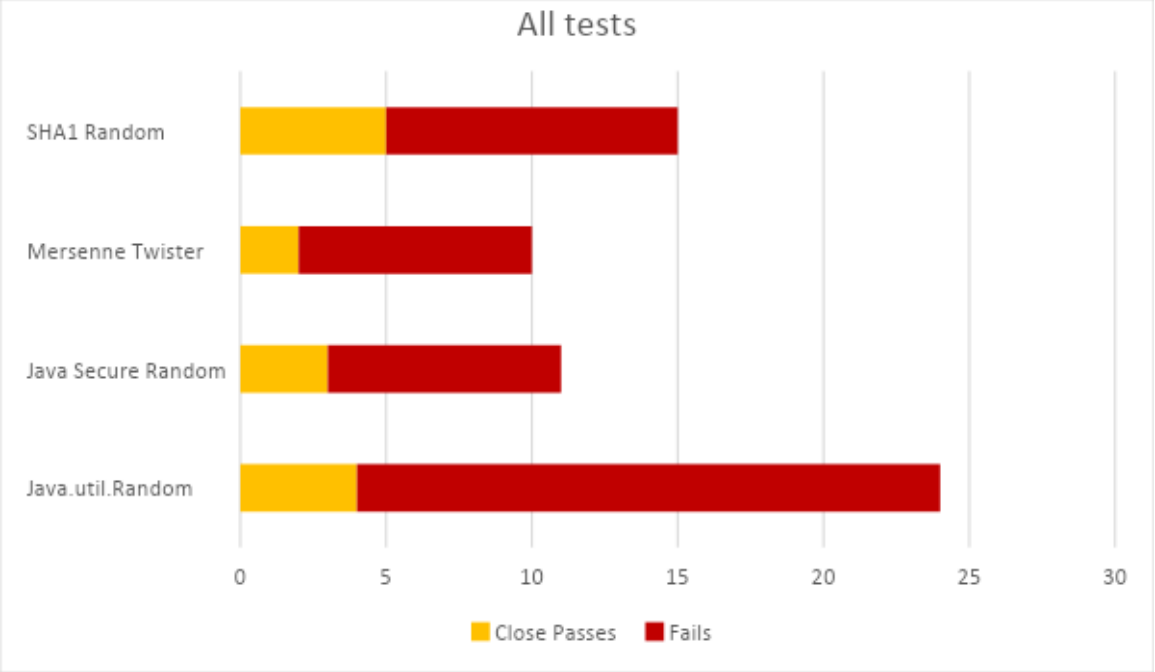

Figure 9: - Graphs of All tests

For the purpose of analysis, the fails and close passes for each generator for each test are taken into consideration. This is added up to calculate the consolidated score. The purpose is to calculate the total of fails and close passes with weights of *2* and *1* respectively. Since the non-performance is scored, the one with the minimum score is the best.

The consolidated score for all the tests is provided in the chart above where one can notice the close passes and fails.

For calculating the score, the fail was given 2 points and close pass was given 1.

The score for each generator is given below:

1. Java.util.random 22
2. Java Secure Random 11
3. Mersenne Twister 10
4. SHA1 Random 16

Since the score is about fails, the lowest is seen as the best Pseudo Random Number Generator in generating random numbers. Hence Mersenne Twister is the winner in creating good quality random numbers with Java Secure Random being a close second. The result is hardly surprising since Mersenne Twister is the most commonly used Pseudo Random Number Generator. The close second, Java secure random finds use in cryptographic applications.

The results indicate that the different generators perform differently on different tests. This may be due to the quality and characteristics of both the tests and the generators.

For instance, Mersenne Twister has output comprising long strings of numbers which makes it a favourite with Overlapping Pair's Sparse Occupancy test as compared to other generators. Java.util.random uses a simple algorithm which could be the reason why it is not doing well with most of the tests.

Java Secure Random and SHA1 Random have performed fairly well and that could be the reason for their usage in cryptographic applications. The internal algorithm of Java Secure Random is better than that of the SHA1 Random which explains its better performance.

Mersenne Twister scores well in this speed aspect also. Java Secure Random and SHA1 Random are much slower because their output has to be compatible with the cryptographic requirements. Java.util.random performs in the middle range as expected.

The Java.util.random is faster than Java Secure Random and SHA1 Random since securing the output cryptographically is not involved in the process. Thus, the Mersenne Twister algorithm is quite fast and produces a huge set of random numbers.

The Java Secure Random algorithm is part of SHA1 hash function which explains the similar behaviour between the two.

6. Further scope of the investigation

The research study is conducted to analyse different characteristics, which influence the applications of the random numbers, of Pseudo Random Number Generators on the basis of the specific tests conducted above. The study is limited in its scope due to the time constraints and level of complexity involved. The study can be further improved in the following ways.

1. More random number generators can be included in the study to give a better representation.
2. The tests were done using Java utilities. The results can be further analysed by the usage of utilities available in other compilers such as Python and C++ which might throw up a whole different set of results.
3. In the present study, the results were collected as abstracted outputs from the tests with the help of the Java Randomness Test Tool. There was no attempt to evaluate the internal code structure of the Pseudo Random Number

Generators and attribute reasons for the specific output obtained. The study can be expanded by understanding the reasons for a particular behaviour of the generators.

4. The study can also include theoretical and mathematical analysis of the random generators and try to forecast the results of the generated random numbers.

7. Conclusion

The results show varying degrees of performance for different random number generators with reference to the quality and the speed of the output. This fits perfectly with the research question: ***HOW DOES THE PERFORMANCE OF DIFFERENT JAVA PSEUDO RANDOM NUMBER GENERATORS MAKES THEM SUITABLE FOR THEIR APPLICATIONS?***

It is only proper in this context to look at the applications of each of these generators. This is done by looking at the results and with the knowledge of the internal functioning of the algorithm used in these generators.

The Mersenne Twister stood at the pole position in terms of quality of output and the speed but the concern is the security. It is used in scenarios which require large sequences of random numbers such as scientific simulations and some type of video games. The Mersenne Twister algorithm produces large stream of random numbers quite quickly. Thus it is not surprising that the Mersenne Twister is the most commonly used generator for all applications requiring random numbers. The period (number of values generated before the repetition happens) of this generator is $(2^{19937}) - 1$, which is really good. It is used in many software systems such as MS

Excel, GAUSS, GNU multiple precision arithmetic library, and Glib because of its performance in graphics and data analytics applications.

Java Secured Random generator which is second in line is widely used for cryptographic applications. This requires compliance to the standard FIPS *140-2*, specific to the cryptographic modules. This is *128* bits long and takes random data from OS for seed generation which makes the generated numbers stronger. This generator shows the capability of producing fairly good quality of random numbers which are cryptographically secure but at a relatively slow speed. This generator finds applications in gambling and lottery where these characteristics are of prime importance.

SHA1 Random with its average performance finds use in producing cryptographically secure random numbers which makes it favourite in any application that require certain level of security.

Java.util.random, which had the worst performance in randomness, is used for generating random numbers of different data types such as integer, long, float etc. This cannot be used for cryptographic applications. This is used for functions that do not require very high level of randomness, safety, and large sequence of numbers thus finding applications in video games.

This essay with its investigation and results proves that every Pseudo Random Number Generator possesses its own characteristics which can be analysed and used before choosing the one for their application. Given Pseudo Random Number Generators are being applied at an unprecedented rate, hopefully, this paper proves

to be useful for those seeking to apply random number generator but are confused by the large number of Pseudo Random Number Generators available.

8. Bibliography

"Birthday Spacing Test". *Software.Intel.Com*, 26 December 2019, https://software.intel.com/en-us/mkl-vsnotes-birthday-spacing-test. Accessed 28 December 2019

"How Does Java.Util.Random Work And How Good Is It?". *Javamex.Com*, https://www.javamex.com/tutorials/random_numbers/java_util_random_algorithm.shtml. Accessed 16 December 2019.

"Rank Of 32X32 Binary Matrices Test". *Software.Intel.Com*, 26 December 2019, https://software.intel.com/en-us/mkl-vsnotes-rank-of-32x32-binary-matrices-test. Accessed 28 December 2019.

"Runs Test Of Randomness - Statistics Solutions". *Statistics Solutions*, https://www.statisticssolutions.com/runs-test-of-randomness/. Accessed 22 December 2019.

"Tutorial: Pearson's Chi-Square Test For Independence". 2008, *Ling.Upenn.Edu*, https://www.ling.upenn.edu/~clight/chisquared.htm. Accessed 22 December 2019.

“Entropy and Random Number Generators.” Calomelorg RSS, 1 May 2017 calomel.org/entropy_random_number_generators.html. Accessed 14 December 2019.

Andy Chalk. “Randomness in Gaming: Good or Bad?” The Escapist, 23 Sept. 2009, www.escapistmagazine.com/news/view/94927-Randomness-in-Gaming-Good-or-Bad. Accessed 13 December 2019.

Bellamy, James. *Randomness Of D Sequences Via Diehard Testing*. p. 3, https://arxiv.org/ftp/arxiv/papers/1312/1312.3618.pdf. Accessed 27 December 2019.

Ben Lynn. “Pseudo-Random Number Generators.” Applied Cryptography Group, crypto.stanford.edu/pbc/notes/crypto/prng.html. Accessed 13 December 2019.

Gaeini, Ahmad et al. “A General Evaluation Pattern for Pseudo Random Number Generators.” Enzymes Application in Diagnostic Prospects, Academic Journals Inc., USA, July 15 2015, scialert.net/fulltextmobile/?doi=tasr.2015.231.244. Accessed 16 December 2019.

Gary C.Kessler. File Signatures, 11 December 2019, www.garykessler.net/library/crypto.html. Accessed 18 December 2019.

GRC | Port Authority, for Internet Port 143, 6 November 2011, www.grc.com/otg/uheprng.htm Accessed 16 December 2019.

Keith Burgun. “Randomness and Game Design.”, 5 Nov. 2014, keithburgun.net/randomness-and-game-design/. Accessed 11 December 2019.

Lemire, Daniel. “Default Random-Number Generators Are Slow.” Daniel Lemire's Blog,

lemire.me/blog/2016/02/01/default-random-number-generators-are-slow/. Accessed 12 December 2019.

Mansi Sheth. “Cryptographically Secure Pseudo-Random Number Generator (CSPRNG).” CA Veracode, 29 March 2017, www.veracode.com/blog/research/cryptographically-secure-pseudo-random-number-generator-csprng. Accessed 10 December 2019.

Rouse, Margaret. "What Is Pseudo-Random Number Generator (PRNG)?". *Https://Whatis.Techtarget.Com/*, March 2011, https://whatis.techtarget.com/definition/pseudo-random-number-generator-PRNG. Accessed 14 December 2019.

Stojanov, Georgi, and Andrea Kulakov. ICT Innovations 2016. Springer, 2017, p. 86. Accessed 28 December 2019.

Turner, Nigel. “Randomness, Does It Matter?” Journal of Gambling Issues, August 2000, jgi.camh.net/index.php/jgi/article/view/3562/3522. Accessed 9 December 2019.

Yash Singla. “Pseudo Random Number Generator (PRNG).” GeeksforGeeks, 27 June 2017, www.geeksforgeeks.org/pseudo-random-number-generator-prng/. Accessed 9 December 2019.