AN ALTERNATIVE APPROACH FOR SELECTION OF PSEUDO RANDOM NUMBERS FOR ONLINE EXAMINATION SYSTEM

Shilpi Kumari Shaw[1], Aakash Sharma[2], Shoubhik Chowdhury[3], Aritra Biswas[4], Arnab Mitra[5]

[1,2,3,4,5] Department of Computer Science & Engineering
Adamas Institute of Technology, Barasat-700126, India
{shilpishaw.ait, aakashcal91, shoubhikchowdhury91, biswas.aritra8, mitra.arnab}@gmail.com

ABSTRACT

Fast and accurate selection of random pattern is needed for many scientific and commercial applications. One of the major applications is Online Examination system. In this paper, a sophisticated approach has been developed for the selection of uniform pseudo random pattern for Online Examination System. Three random integer generators have been compared for this purpose. Most commonly used procedural language based pseudo random number; PHP random generator and atmospheric noise based true random number generator have been considered for easy generation of random patterns. The test result shows a varying degree of improvement in the quality of randomness of the generated patterns. The randomness quality of the generated pseudo random pattern has been assured by diehard test suite. An experimental outcome for our recommended approach signifies that our approach selects a quality set of random pattern for Online Examination System.

KEYWORDS

Pseudo random number generator (PRNG), Pseudo random Pattern Generator (PRPG), Procedural Language Random Number Generator (PrRNG), PHP random Number Generator (PHPRNG), True Random Number Generator (TrRNG), Online Examination System (OES*)*

1. INTRODUCTION

Random number [1], [2] is a number engendered by a method, whose consequence is volatile, and which cannot be sub sequentially dependably replicated. It is purely impossible to authenticate whether the certain number was formed by a random number generator or not. In order to study the predictability of the output of such a producer, it is therefore unconditionally essential to reflect order of number. It is pretty direct to express whether an order of infinite extent is random or not. This arrangement is random if the amount of info it covers – in the logic of Shannon's information concept – is also infinite. In additional falling out, it must not be probable for a processer, whose extent is finite, to produces this arrangement. Excitingly, an infinite random arrangement comprises all probable predetermined sequences. Such an unbounded arrangement does for instance hold the Microsoft Windows source code or the copy of the Geneva conventions. Unfortunately, this description is not very worthwhile, as it is not promising in exercise to create and practice infinite orders.

In the case of a finite arrangement [3] of figures, it is validly difficult to confirm whether it is random or not. It is only conceivable to check that it shares the statistical stuffs of a random arrangement– similar to equi-probability of the whole numbers – but this is a challenging and complex task. To demonstrate this, let us for specimen consider a binary random number generator constructing sequences of ten bits. Although it is precisely as probable as any other ten bits sequences, 0 0 0 0 0 0 0 0 0 0 does look a lesser random than 1 0 0 1 0 1 0 1 1 1.

Pseudo-random number generators (PRNGs) [4], [5] are algorithms to instinctively produce elongated runs of numbers with superior random properties however eventually the series repeats. The sequence of values evaluated by such algorithms is frequently resolute by a rigid number known as a seed [4]. The most frequent PRNG is the linear congruential generator, which utilizes the recurrence in Equation 1.

$$A_{n+1} = (c * A_n + d) \bmod m \quad \ldots\ldots \ (1)$$

where A is the sequence of pseudorandom values, and
m, $0 < m$ — the "modulus";
c, $0 < c < m$ — the "multiplier";
d, $0 \leq d < m$ — the "increment";
A_0, $0 \leq A_0 < m$ — the "seed" or "start value".

There exist two major ways used to produce [6] random numbers. One processes certain physical phenomenon that is probable to be random and then recompenses for probable biases in the measurement procedure. The further practices computational algorithms that can yield long arrangements of apparently random outcomes, which are in fact absolutely determined by a smaller primary value, well-known as a seed. The latter forms are frequently known as pseudorandom number generators.

Many extents of statistical investigation, research, and simulation depend on the superiority of random number generators. Maximum programs for statistical data analysis comprise a function for producing uniform random numbers. The Diehard suite [5] of tests has grown into a standard technique of evaluating the superiority of uniform random number generator procedures
Diehard tests [5], [7-8] are a battery of statistical test to measure the standard of random number generated. At first it was developed by George Marsaglia and first issued in 1995 on a CD-ROM of random numbers. Soon after advanced version of this statistical test been reformed and circulated thru University of Hong Kong.

The below mentioned tests are performed to extent the feature of the randomness of an individual random pattern creator:

i. **Birthday spacings:** The name birthday spacing is centred on birthday paradox. Here random points are selected on a large interval with the asymptotically exponentially distributed spacing among the points.
ii. **Overlapping permutation:** This test always analyse five successive random number arrangements, with statically identical probability, a total 120 possible ordering should occur.
iii. **Ranks of matrices:** choose few number of bits from certain amount of random number to formulate a matrix over [0, 1], then govern the rank of the matrix on the basis of determinant value of matrix.
iv. **Monkey tests:** The name monkey test is based on infinite monkey theorem. In this test sequence of few numbers of bits is considered as "words". Therefore overlapping word

present in a stream be counted. Any known distribution be followed by the number of "words" that don't appear.

v. **Count the 1's:** In this test, count the 1bit in every single of either successive or chosen bytes. Then convert the counted values into "letters" and lastly following the counting of the existences of five-letter 'words".

vi. **Parking lot test:** Unit circle in a 100 x 100 square placed randomly is tested to search whether any of the circles, overlaps a present one. After a repetition of 12000 tries, the number of effectively "parked" circles would follow a certain normal distribution.

vii. **Minimum distance test:** In this test, place randomly 8000 points in a 10.000 x 10.000 squares, after that minimum distance between the pair is to be found. The square of the distance with a certain mean is exponentially distributed.

viii. **Random spheres test:** In this test, in a cube of edge 1,000, choose randomly 4,000 points. On each point a sphere is to be centred, whose radius is the least distance to another point.

ix. The smallest sphere's volume ought to be exponentially distributed using a certain mean.

x. **The squeeze test:** Until one reach 1, multiply 2^{31} by random floats on [0,1). Repeat this thing 100,000 times. The number of floats required to arrive at 1 should pursue a certain distribution.

xi. **Overlapping sums test:** Create a extended series of random floats on [0,1]. Add series of 100 successive floats. The sums have to be normally distributed with characteristic mean with sigma.

xii. **Runs test:** create a extended sequence of random floats on [0, 1). Count up ascending and descending runs. The counts have to pursue a certain distribution.

xiii. **The craps test:** Count the success and the number of toss per game, play 200,000 games of craps. Each count ought to pursue a certain distribution.

The organization of this paper is as follows: Section 2 briefly discusses the related work. Section 3 expounds the proposed work. Section 4 records the experimental results. Section 5 draws the conclusion. Section 6 is devoted to acknowledgements, and Section 7 includes references.

2. RELATED WORK

A number of methods have been applied [9-13], [16] to generate better quality random numbers. Some novel labours have been established to produce better quality random numbers. Some significant efforts have also been made to produce pseudo-random numbers. Among all Intel true random number and true random number generation by Random.org is pretty remarkable where atmospheric noise is applied as the seed.

A true random number[14], [15] producer is a section of electronics that plugs inside a computer and yields genuine random numbers as disparate to the pseudo-random numbers that are created by a computer program. The standard technique is to intensify noise produced thru a resistor (Johnson noise) or a semi-conductor diode and feed this to a Schmitt trigger or comparator. If one samples the out-come (not too rapidly) get a sequence of bits which are statistically autonomous.
Utmost random numbers [4] applied in computer programs are pseudo-random, which means they are created in a expectable fashion via a mathematical formulation. This is adequate for several purposes; however it might not stay random in the way one imagines if it used to gamble like dice rolls and lottery drawings.

RANDOM.ORG [16], [17] proposed true random numbers to any person on the Internet. The randomness originates from atmospheric noise, which for numerous drives is superior to the pseudo-random number algorithms usually applied in computer programs. For gambling sites, for scientific applications and for art and music people use RANDOM.ORG.

3. PROPOSED WORK

Generation of our algorithm is to obtain random question number in on-line examination system using PHP. Compare to other similar on-line examination system this system is more innovative because of the generation of pseudo random pattern, as a result none of the question get repeated at boundless instant of time.

Procedural language provides random number generation function rand(), which is found in <stdlib.h> header. Function rand actually produces pseudorandom numbers. Calling rand recurrently produces a arrangement of numbers that seems to be random.

Based on the above mentioned, the following flowchart 1 and Algorithm 1, should generate the random integer set in procedural language.

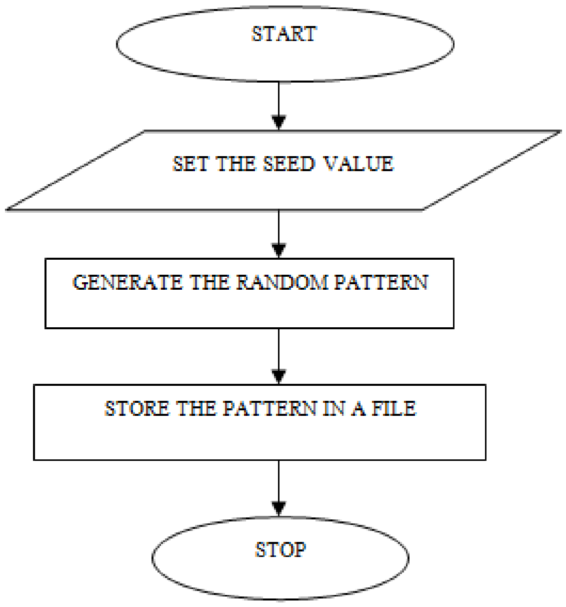


Figure 1: Flow chart for generating a random pattern in procedural language

Similarly just like Procedural Language random function, PHP also have its own random function mt_rand().The number generated in PHP is uniformly distributed over the specified range. PHP integers are 32 or 64 bits wide, and are represented by means of two's complement arithmetic.
Based on the above mentioned, the following flowchart 2 and Algorithm 2, should generate the random integer set in PHP.

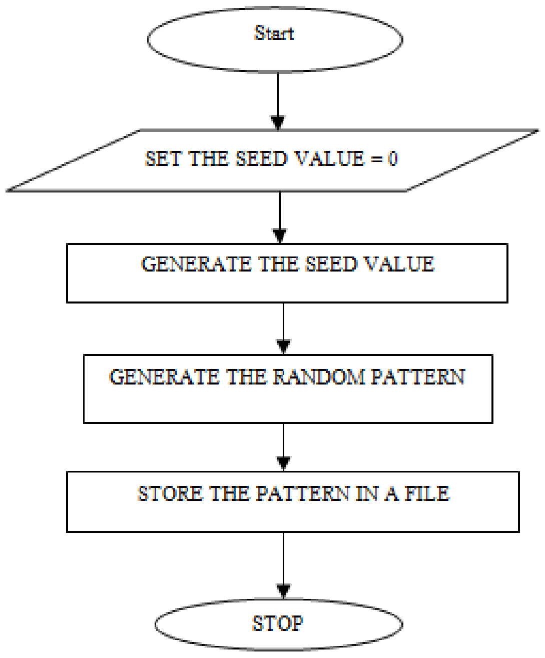


Figure 2: Flow chart for generating a random pattern in Scripting Language

Finally the proposed algorithm which has been proved to be better compare to previous algorithms, and gave better randomness quality. The following flowchart 3 and Algorithm 3 should generate the random integer set.

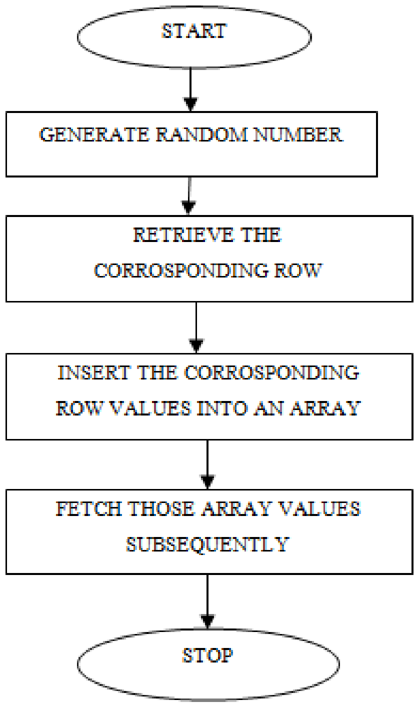

START
GENERATE RANDOM NUMBER
RETRIEVE THE CORROSPONDING ROW
INSERT THE CORROSPONDING ROW VALUES INTO AN ARRAY
FETCH THOSE ARRAY VALUES SUBSEQUENTLY
STOP

Figure 3: Flow chart for selecting a true random sequence for online examination

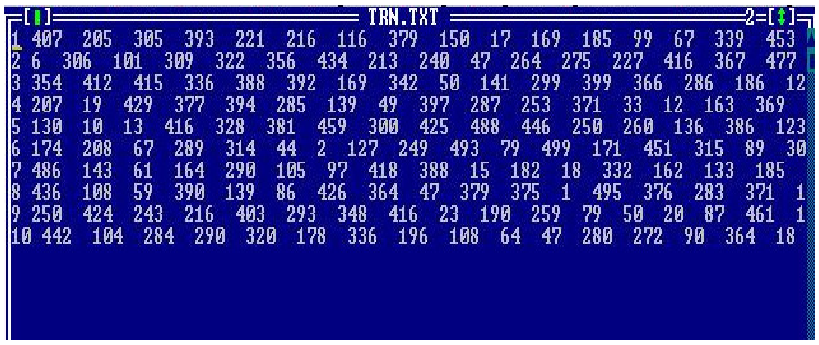
Algorithm 1: Selection of Pseudo Random Pattern Sequence
Input: *Set of true random sequence*
Output: *Random pattern for randomly generated question set*
(Write this following algorithm in step wise only. no pseudocode)

```
STEP 1: Start

STEP 2: Generating a random number using the seed value.

STEP 3: According to the random number, the position has been found from the file in which
        the pseudo random patterns are stored.
    STEP 3.1: Searching the random number from the file.
    STEP 3.2: In the case of more than one digit number, the digits are concatenated with
              each other.
    STEP 3.3: The total string has been converted into integer.

STEP 4: After matching the first element, insert the corresponding row elements into the array.
    STEP 4.1: Insert the element into the first position of the array until a space is occurred.
          Checks the next field is blank or a new line
          If matched
                The previous value has put into the array
    STEP 4.2: Repeat the above step (STEP 4.2) until a new line or the end of the file
          (EOF) is reached.
    STEP 4.3: The elements that are found in the row are converted to integer value.

STEP 5: Print the array elements of the selected row.

STEP 6: Get the selected random pattern sequence for randomly selected question set.

STEP 7: End
```

Our proposed above algorithm have generated random number up to 3 digits. That means it can generate up to 1000 random sequences which is much more than our requirement for online examination purpose, as we know in an exam session we hardly require 40/50 question. Further this algorithm is also efficient to produce random numbers larger than 3 digits on requirement, which has been shown in Figure 4(b).

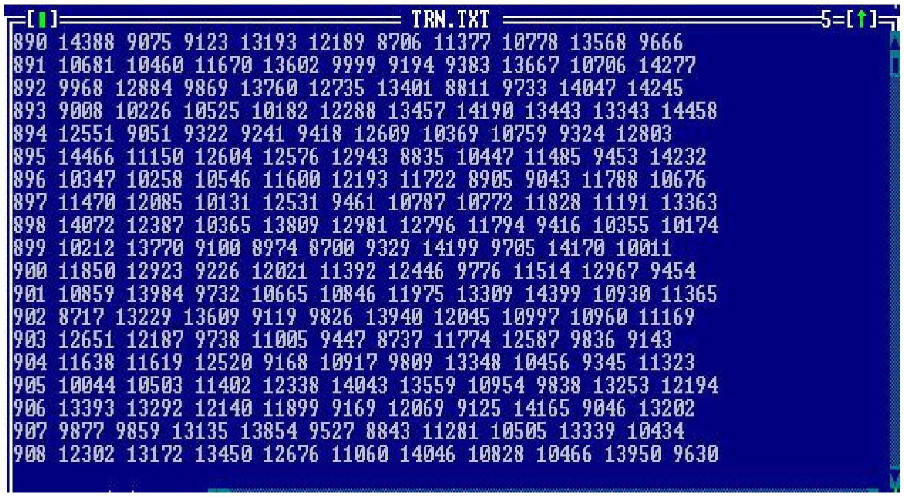
4. Experimental Observation and Result Analysis

The given file contains the random patters stored in text format.



```
TRN.TXT
1 407  205  305  393  221  216  116  379  150  17  169  185  99  67  339  453
2 6  306  101  309  322  356  434  213  240  47  264  275  227  416  367  477
3 354  412  415  336  388  392  169  342  50  141  299  399  366  286  186  12
4 207  19  429  377  394  285  139  49  397  287  253  371  33  12  163  369
5 130  10  13  416  328  381  459  300  425  488  446  250  260  136  386  123
6 174  208  67  289  314  44  2  127  249  493  79  499  171  451  315  89  30
7 486  143  61  164  290  105  97  418  388  15  182  18  332  162  133  185
8 436  108  59  390  139  86  426  364  47  379  375  1  495  376  283  371  1
9 250  424  243  216  403  293  348  416  23  190  259  79  50  20  87  461  1
10 442  104  284  290  320  178  336  196  108  64  47  280  272  90  364  18
```

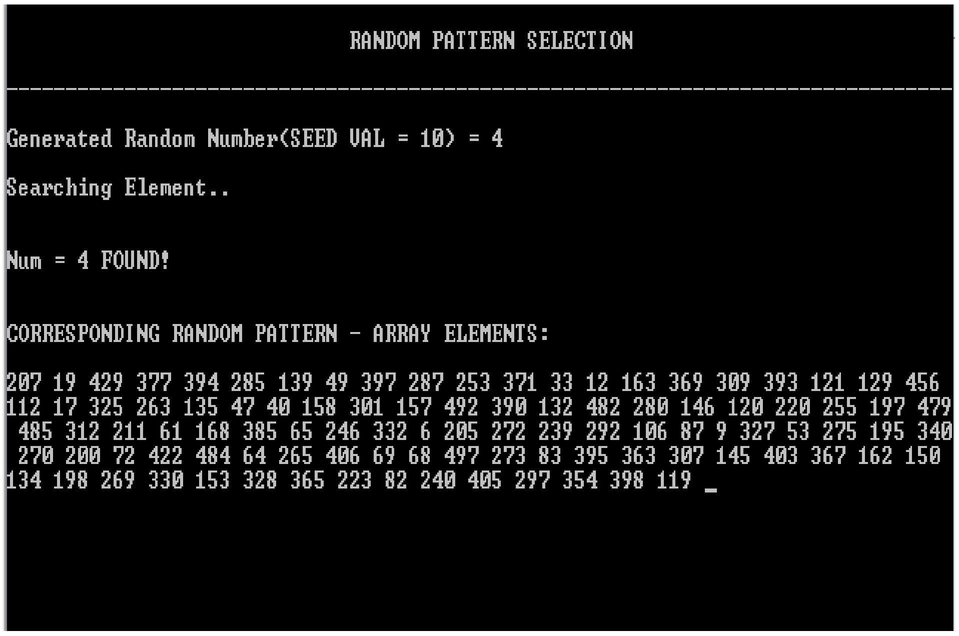
Figure 4(a): The text file in which the true random sequences are stored
]



```
TRN.TXT
890 14388 9075 9123 13193 12189 8706 11377 10778 13568 9666
891 10681 10460 11670 13602 9999 9194 9383 13667 10706 14277
892 9968 12884 9869 13760 12735 13401 8811 9733 14047 14245
893 9008 10226 10525 10182 12288 13457 14190 13443 13343 14458
894 12551 9051 9322 9241 9418 12609 10369 10759 9324 12803
895 14466 11150 12604 12576 12943 8835 10447 11485 9453 14232
896 10347 10258 10546 11600 12193 11722 8905 9043 11788 10676
897 11470 12085 10131 12531 9461 10787 10772 11828 11191 13363
898 14072 12387 10365 13809 12981 12796 11794 9416 10355 10174
899 10212 13770 9100 8974 8700 9329 14199 9705 14170 10011
900 11850 12923 9226 12021 11392 12446 9776 11514 12967 9454
901 10859 13984 9732 10665 10846 11975 13309 14399 10930 11365
902 8717 13229 13609 9119 9826 13940 12045 10997 10960 11169
903 12651 12187 9738 11005 9447 8737 11774 12587 9836 9143
904 11638 11619 12520 9168 10917 9809 13348 10456 9345 11323
905 10044 10503 11402 12338 14043 13559 10954 9838 13253 12194
906 13393 13292 12140 11899 9169 12069 9125 14165 9046 13202
907 9877 9859 13135 13854 9527 8843 11281 10505 13339 10434
908 12302 13172 13450 12676 11060 14046 10828 10466 13950 9630
```

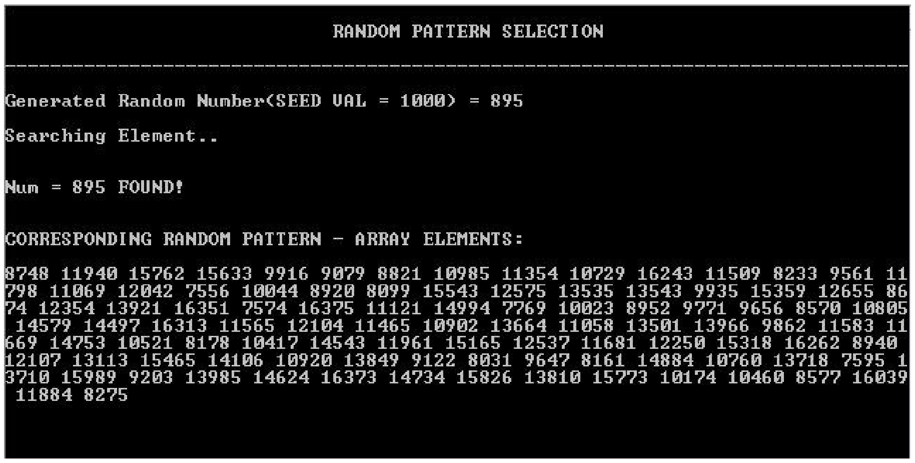
Figure 4(b): The text file in which the true random sequences are stored

The algorithm generates a random number between 0 to (seed value - 1), i.e. 0 - 9 in this example. It then starts searching from the beginning of the text file containing the random patterns. If the first number matches the random number, the entire row containing the random pattern is taken into an integer array else the row is scanned simply as text. The algorithm then scans the first element of the next row. If the number is found, it takes the random pattern into the array else continues searching. The algorithm continues until the random number is matched with the first element and the corresponding random pattern is selected.



```
                        RANDOM PATTERN SELECTION

--------------------------------------------------------------------------------

Generated Random Number(SEED VAL = 10) = 4

Searching Element..


Num = 4 FOUND!


CORRESPONDING RANDOM PATTERN - ARRAY ELEMENTS:

207 19 429 377 394 285 139 49 397 287 253 371 33 12 163 369 309 393 121 129 456
112 17 325 263 135 47 40 158 301 157 492 390 132 482 280 146 120 220 255 197 479
 485 312 211 61 168 385 65 246 332 6 205 272 239 292 106 87 9 327 53 275 195 340
 270 200 72 422 484 64 265 406 69 68 497 273 83 395 363 307 145 403 367 162 150
134 198 269 330 153 328 365 223 82 240 405 297 354 398 119 _
```

Figure 5(a): Outcomes of the Proposed Algorithm



```
                        RANDOM PATTERN SELECTION
--------------------------------------------------------------------------------
Generated Random Number(SEED VAL = 1000) = 895

Searching Element..

Num = 895 FOUND!

CORRESPONDING RANDOM PATTERN - ARRAY ELEMENTS:

8748 11940 15762 15633 9916 9079 8821 10985 11354 10729 16243 11509 8233 9561 11
798 11069 12042 7556 10044 8920 8099 15543 12575 13535 13543 9935 15359 12655 86
74 12354 13921 16351 7574 16375 11121 14994 7769 10023 8952 9771 9656 8570 10805
 14579 14497 16313 11565 12104 11465 10902 13664 11058 13501 13966 9862 11583 11
669 14753 10521 8178 10417 14543 11961 15165 12537 11681 12250 15318 16262 8940
12107 13113 15465 14106 10920 13849 9122 8031 9647 8161 14884 10760 13718 7595 1
3710 15989 9203 13985 14624 16373 14734 15826 13810 15773 10174 10460 8577 16039
 11884 8275
```

Figure 5(b): Outcomes of the Proposed Algorithm

Centred on this factor Table 2 replicates the end result of Diehard tests. The examined outcomes can be observed in Figure 4 with graph, which demonstrate the improved feature of randomness for resultant random integer originator

.Most of the tests in DIEHARD returns a p-value, which should be uniform on [0, 1) if the input file contains truly independent random bits. Those p-values are chosen up by p=F(Y), where F is the presumed distribution of the sample random variable Y---often normal. But that presumed F is just an asymptotic approximation, for which the fit will be worst in the tails. Thus outcome should not be surprised with occasional p-values near 0 or 1, such as .0012 or .9983. When a bit stream really FAILS BIG, it will get p's of 0 or 1 to six or more places. By every means, the value of p < .025 or p> .975 means that the RNG has "failed the test at the .05 level".

Table 1: Diehard Test

Test Number	Diehard Test Name
1.	Birthday Spacings
2.	GCD
3	Gorilla
4.	Overlapping Permutations
5.	Ranks of 31x31 and 32x32 matrices
6.	Ranks of 6x8 Matrices
7.	The Bitstream Test
8.	Monkey Tests OPSO,OQSO,DNA
9.	Count the 1`s in a Stream of Bytes
10.	Count the 1`s in Specific Bytes
11.	Parking Lot Test
12.	Minimum Distance Test
13.	Random Spheres Test
14.	The Sqeeze Test
15.	Overlapping sums Tests
16.	Runs Up and Down Test
17.	The Craps Test

Table 2: Test Result

Diehard Test Number	**Procedural Language**	**PHP**	**TRNG**
1.	Fail	Fail	Pass
2.	Fail	Fail	Fail
3.	Fail	Fail	Fail
4.	Fail	Fail	Pass
5.	Fail	Fail	Fail
6.	Fail	Fail	Pass
7.	Fail	Fail	Fail
8.	Fail	Fail	Pass
9.	Fail	Fail	Pass
10.	Fail	Fail	Pass
11.	Fail	Fail	Pass
12.	Fail	Fail	Pass
13.	Fail	Fail	Pass
14.	Fail	Fail	Pass
15.	Fail	Fail	Pass
16.	Fail	Fail	Pass
17.	Fail	Fail	Pass
Total Number of Diehard Test Passes	0	0	13

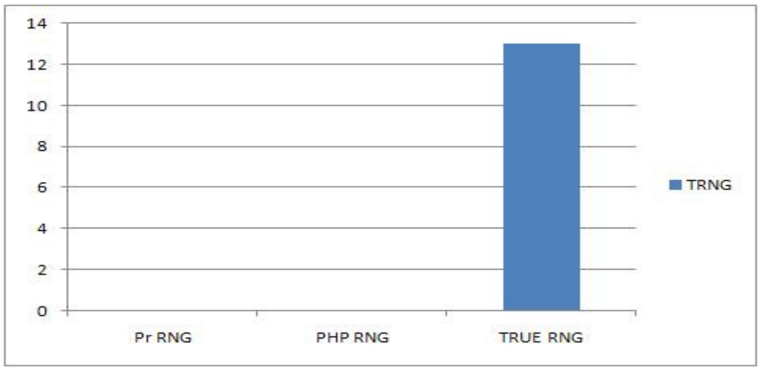


Figure: 6 Result of the above mentioned table is shown in the following graph**.**

The graph below show the randomness quality of a random pattern developed in C language, PHP, and proposed algorithm.

Procedural language has its own random function for generating random pattern, this function give a random pattern between 0 to 99, which is plotted in the given graph in yellow colour.
Similarly PHP has also its own random function for generating random pattern, its randomness is also plated in the same graph with pink colour.

But after comparing the proposed algorithm for random pattern generation with C's and PHP's random pattern generation function by plotting the randomness in the same graph with blue colour by taking different values, the outcomes shows that this proposed algorithm gives improved random quality.

Outcomes based on figure 8, it can be concluded that our proposed algorithm can be appliance as an enhanced source of random patterns as it have maximum amount of randomness with allusion to all further random function.

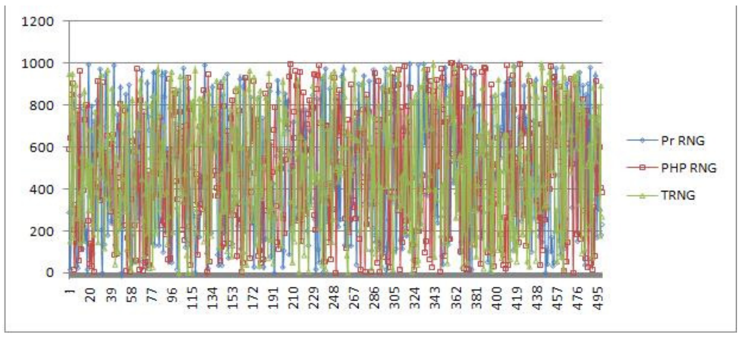


Figure 8: Randomness testing with higher number of data set

5. Conclusion

The algorithm selects a better random pattern and shows that the number selected from the pattern has no repetition. The graph represents as figure 2 express the superiority of randomness archived by this algorithm.

So as per our requirement, through this algorithm we can achieve random questions for our online examination system, and thus meets our requirements.

References

[1] http://en.wikipedia.org/wiki/Random_Number
[2] Arnab Mitra, Anirban Kundu; (2012) Cost optimized Approach to Random Numbers in Cellular Automata; The Second International Conference on Computer Science, Engineering & Applications (ICCSEA); India
[3] Wolfram; Wolfram Mathematica Tutorial Collection: Random Number .http://www.wolfram.com/learningcenter/tutorialcollection/RandomNumberGeneration/.pdf
[4] http://en.wikipedia.org/wiki/Pseudorandom_number_generator
[5] Arnab Mitra, Anirban Kundu; (2012) Cost Optimized Design Technique for Pseudo-Random Numbers in Cellular Automata; International Journal of Advanced Information Technology (IJAIT) Vol. 2, No.3, June 2012
[6] http://en.wikipedia.org/wiki/Random_Number_Generationen.wikipedia.org/wiki/Diehard_tests
[7] Robert G. Brown. dieharder: A Random Number Test Suite, 2006a.
[8] http://www.phy.duke.edu/~rgb/General/dieharder.php. C program archive dieharder, version 1.4.24.
[9] Colin Plumb; (1994) "Truly Random Numbers"; Dr.Dobbs Journal, November 1994, p.113.
[10] Tim Matthews; (1995) "Suggestions for random number generation in software"; RSA Data Security Engineering Report, 15 December 1995.
[11] Boaz Barak and Shai Halevi; (2005) An architecture for robust pseudo-random generation and applications to /dev/random; In Proc. ACM Conf. on Computing and Communication Security (ACM CCS); 2005.
[12] B. Jun and P. Kocher; (1999) The Intel Random Number Generator.; Cryptography Research Inc. white paper, Apr. 1999.
[13] Matsumoto, M. and Nishimura, T. (1998), "Mersenne-Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," ACM Transactions on Modeling and Computer Simulation, 8:1, pp. 3- 30..
[14] http://en.wikipedia.org/wiki/True_Random_Number_Generator
[15] Dirk Eddelbuettel; Random: An R package for true random numbers; http://dirk.eddelbuettel.com/bio/papers.html
[16] www.random.org
[17] http:// www.random.org/randomness/