Using Conway's Cosmological Sequence for Pseudo-Random Number Generation

Anton Paolo del Mundo
William Valencia

Introduction

There is this riddle that often finds its way to brain teaser collections and puzzle books. The question is often posed this way:

1
11
21
1211
111221
What is the next number?

The reader often tries to solve such find-next-number problems by applying mathematical equations among the numbers (like a Fibonacci sequence). To find the next number, however, requires a novel way of looking at the problem—a non-mathematical mentality, in fact.

The answer to this sequence is 312211. This can be found by successively tallying the number of consecutive digits in a run and appending this and the digit to the new string. So looking at the last round of our sequence (111221), we can see that there are 3 ones in a row, 2 twos in a row, and 1 one.

This charming riddle is otherwise known as the Conway sequence, discovered and originally studied by John Conway

[C]. In this paper, we attempt to use Conway's sequence to generate pseudo-random numbers. We believe that generating pseudo-random numbers using Conway's sequence holds promise because of two properties:

i. **It is relatively quick.** Compared to traditional pseudo-random number generators, many of which use successive squaring and other such methods, Conway's sequence only needs to count the number of consecutive digits.

   This can be done in the order of O(n) time for every round (since we make n passes through the sequence in each round). Successive squaring, on the other hand, which takes the form

$$y \equiv g^{x} \bmod p,\ 0 \le y \le p - 1$$

   is polynomial in the number of bits of $|p| + |g| + |x|$ [LM]. Successive squaring is quick in itself, but Conway's sequence offers a slight improvement.

   LFSR is, of course, much quicker than Conway. But it cannot stand its own to simple linear algebra cryptanalysis.

ii. **It can easily be modified.**

   There are many modifications we can apply to Conway's sequence to generate more random

numbers, some of which we will present in this paper. Granted there are weaknesses to Conway's sequence, and we will show that a naïve implementation of this will lead to non-random, repeating sequences. Our modifications to the existing algorithm will, therefore, attempt to solve these weaknesses.

Naïve Implementation of Conway's Sequence

For our purposes, after performing Conway's operation n number of rounds, we convert all non-1 numbers to zero. In practice, this results to approximately 45% zeros and 55% ones (the fact that the numbers are not split evenly between ones and zeros is actually an advantage according to [NIST]). This is our naïve implementation of Conway's sequence for pseudo-random number generation.

One problem is, however, the length of the numbers become exponentially longer after every successive round in the sequence. To avoid this length explosion and shorten our running time, we limit the length of every round to an arbitrary number of digits, and digits after this threshold are dropped.

Problems with Conway's Naïve Implementation

By feeding Conway's sequence with various seeds and lengths, we can see that the sequence repeats after a number of rounds. This number of rounds is usually frequently low (around 100).

In Figure 1, we tabulate the number of rounds before Conway's sequence repeats given random seeds. Note that there is a positive and linear correlation with the number of rounds and the length of the sequence (if there are more digits to match, the number of rounds before the numbers match will logically be larger).

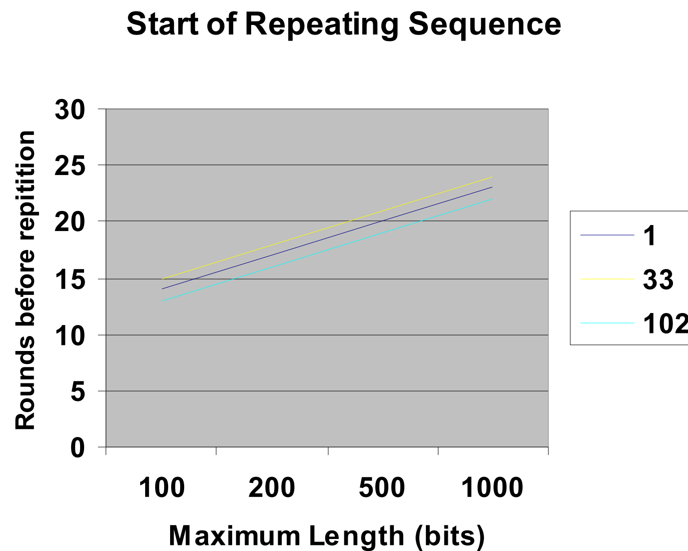

Figure 1. 1, 33 and 102 are the initial starting sequences. Conway's naive implementation repeats as fast as 14 rounds and repeats every 3 rounds afterward for the initial sequence 1. The graph shows that as the length of the sequence increases, the number of rounds before it repeats increases as well. However, this increase is negligible compared to the increase in length.

There have been various studies that conclusively prove that Conway's cosmological sequence break into blocks after a number of rounds. We refer you to [EZ] that claim that this number of rounds is 29.

These blocks will become mutually exclusive from the rest of the sequence (results of rest of the sequence do not affect the blocks and the blocks do not affect the rest of the sequence). These blocks themselves repeat after a certain number of rounds. For example, here is a sequence that creates an independent block quickly:

Round	Value
1	3
2	**13**
3	1113
4	3113
5	**13**2113
6	1113122113
7	311311222113
8	**13**211321322113
9	1113122113121113222113
10	31131122211311123113322113

Figure 2. '13' is an independent forming block. It first appears at round 2 and stays thereafter.

As you can see in Figure 2, the initial seed value 3 creates an independent block as early as the second round! The value **13** is uninhibited from the rest of the sequence, and just cycles every 3 rounds (1113, 3113, **13**2113 and so forth).

Consequently, the only way for this block to change is if the next block begins with a 3. For example, if by any chance we get a number that starts as **133xx**, we will get

1123xx and break out of the loop. The chance of this occurring, however, is very improbable([EZ]). The fact that this is improbable is also the reason why numbers tend to get grouped into independent blocks.

Given these observations, there is a great probability that each of these independent blocks will all have the same state as an earlier round. This probability would be greater than, say, all n digits being the same as an earlier round.

Modifications to Conway's Sequence

There are several modifications to our sequence that we have attempted to fix the problem of independent forming blocks. However, most of these failed. The ones listed below here with an asterisk(*****) are the ones that have been found out to replicate the problem. We provide the actual algorithm in the appendix.

i. **Reverse method.**

   Starting from the back*. We first reverse the string and then take the count as in the original sequence. For example take the string 21113, the next number will be 133112 (tallying on 31112). The purpose of this method was to eliminate the outlying digits at both ends of the number. This happens because the sequence grows every round

and each of the later rounds, some digits will be lost because we limit the length. We then replace all non-1 numbers with 0. The reasoning for this is that we noticed there are usually 50% 1's, 35% 2's, and 15% of 3's in the string. We want approximately 50% of 1's and 0's so we convert all non-1's to zeroes. This is also very helpful since an adversary will not be able to determine if a 0 was actually a 2 or a 3.

ii. **Reverse + replace method**

Instead of going through n number of rounds using the reverse method and substituting all non-1s to zeros at the end, we substitute after every round*****. We had thought this method would erase the independent forming blocks. However, by running this method with a number of various seeds, we find that we actually generate repeating patterns even more quickly. See results section for an example.

iii. **Middle Permutation**

This is a simple permutation to our given number. Instead of counting from the front or the back, we successively count the numbers from the mid-digit of the number. So if we were given the number 0123456789, we would permute it to

4536271809 and then tally the resulting string. This permutation will happen after every round. This will help escape the block formation problem encountered in our previous modifications as shown later below.

iv. Adding a substitution table.

The idea of adding a substitution table to our original algorithm is similar to the s-box in the Data Encryption Standard. We can map an arbitrary length block such as ‘121’ to any value. Also, like DES, we shift the s-box every round to ensure that two identical blocks will produce different results. The randomness of the sequence depends on the values in this so-called s-box. This modification can as well increase (or decrease) the length of the string very rapidly. Although we do not test the following method in this paper, further work may be done to check the randomness produced by this modification.

Results

Rounds	Value
1	**13211321322113311213212312311211131122211213211331121321 123123211231131122211211131221131112311332 21**

2	1113122113121113222123211211131211121311121321123113213 2 211211131221232112111312211213111213122112 13
3	311311222113111231133211121312211231131112311211133112 11 1312211213211312111322211231131122111213122 1
4	**132113213221133112132123123112111311222112132113311213 21 12312321123113112221121113122113111231133221**

Figure 3a. Naïve implementation.

Rounds	Value
1	**13211322112231131211322112111321132221221321121321132211 223113211312311311123112131112132113222122 13**
2	13112211321321121311123113111221131231132113121113211213 2113222122132112131112211213112211321321 1213
3	13111221121311121321222113111221223113111221121311221132 13211213111221121331121113211213211312111321
4	11121331121113211213111221121331122123111221223113111221 12131112132122211311122122311321132211223113
5	**13211322212213211213211322112231132132111213111231131112 21222311321132211223113121122212311122122311 3**

Figure 3b. Reversal Method.

Rounds	Value
1	21131252222115114122161143112221111523211212221111211212 11 41121121211311122121111322111114321121241122
2	22113112211152121441213212213112411111226211111422211112 42111122511111222231511323111212311111222411
3	12524116211215213114131224211112211431121411221211154521 14122113111121155211113321221123331111222121
4	22211144211342112122111131141211421211213112131121155114 11152112122121146111122242112113541121121211
5	21111123221111122111411312212112121142151221311413412125 11122211112221114212311111222221141241136243 1
6	12125111111322114114112121131114211111222111111221251311 33121421431213312124115221213112211315512412
7	21211162213511512113421112333111112221111114211211214114 11234111112222121321311615112211111122114121 1
8	22114112126121321151313422111134221141123113112425111312 51111122511131131112212161131231211611112115
9	22214114321111422123421131131211311211145112112523113113 11112112111565112212111523111123421111232321
10	23111124311512212112121131121125311211232211411332112113 14112122421111422311311541161125421112122131

Figure 3c. Middle Permute

Figure 3. Examples of repeating sequences. Values in bold are the values that repeat. Note that the middle permute (3c) algorithm does not have any quickly repeating sequences.

Rounds	Value Before	Value After
1	5555	Initial Sequence
2	45	00
3	20	00
4	20	00

Figure 4a. Reversal + replace method never changes from value 00 after second round.

Rounds	Value

1	1011
2	011011
3	01100110
4	1001000110
5	100100110011
6	01000100110011
7	0100010011001110
8	100100010011001110
9	10010001001100110011
10	0100010001001100110011
11	010001000100110011001110
12	10010001000100110011001110

Figure 4b. Reversal + replace method slowly increasing in size.

Rounds	Value Before
1	**0001000100010001000100010001000100010001000** **10001000100010011001100110011001100** **110011001100110011001100110011**
2	01000100010001000100010001000100010 00100010001000100110011001100110011 001100110011001100110011001100
3	**00010001000100010001000100010001000** **10001000100010011001100110011001100** **110011001100110011001100110011**

Figure 4c. Reversal + replace method repeats at the third round.

Figure 4. Reversal + replace method. As seen in the examples above, the reversal + replace method is not a good alternative because it often can get "trapped" if using certain starting sequences (4a), does not produce large numbers quickly (4b), repeats even more quickly than the naïve implementation (4c).

Middle Permute	Value
Expansion	11122121311212221511212221211311114112121121124211311 2 2114154123124111211211232116141121233212411 31231
Final bit string	11100101011010010110100101101111011010110110001101 10 0110100100100111011011000110101101000010011 01001

Figure 5. Sample string using middle-permute with starting sequence 3, bit length 100 and 1000 rounds.

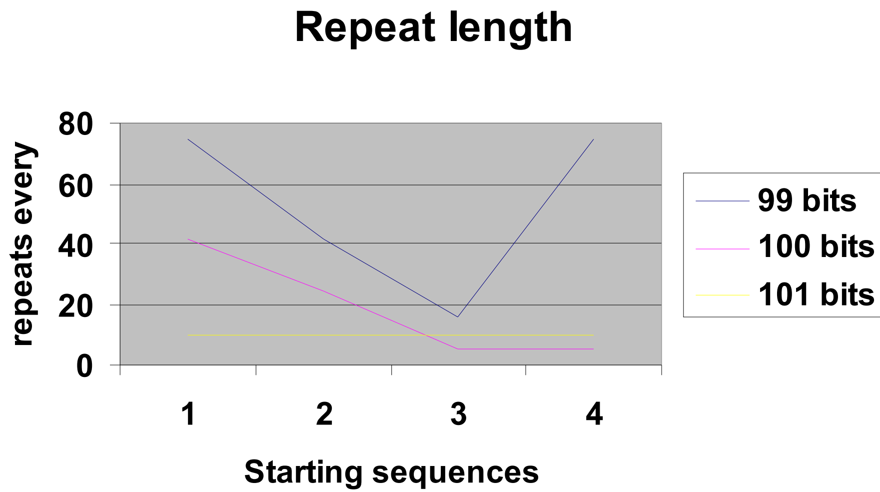

Figure 6. Reverse graph. This graph shows the various lengths of repeating patterns based on starting sequence and bit length. Notice for length 101, all starting sequences have the same repeat length.

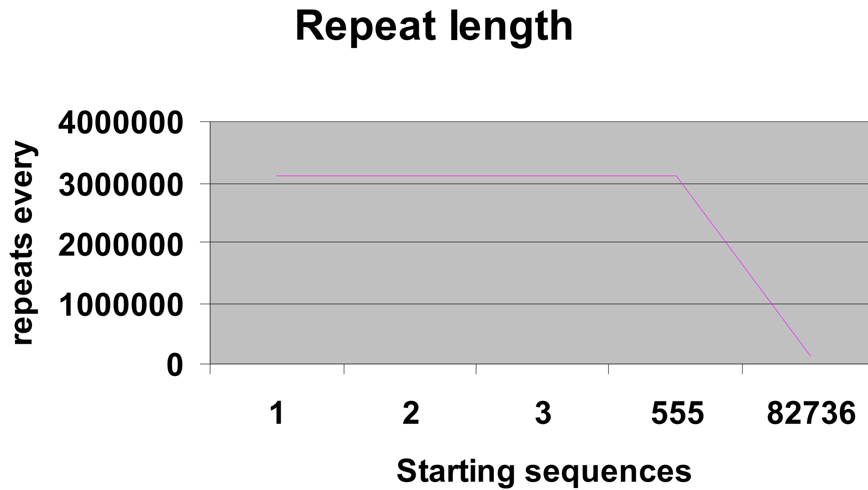

Figure 7. Middle permute graph. This graph shows the various lengths of repeating patterns based on starting sequence and a bit length of 100. Many starting sequences were tested all repeated with repeat length 3105406 except starting sequence 82736 which repeated with a length of 137360. Surprisingly, the starting sequences of 1, 2, 3 and 555 converge to the same sequence after an appropriate number of rounds.

Analysis

Reverse Method

The reverse method produces patterns that re-occur rather quickly. Depending on the maximum length of the number, sequences such as 1, 2 and 3 repeat as soon as 25, 30, and 70 rounds respectively (as seen in our results). It is surprising that re-occuring sequences happen with faithful accuracy after less than two-hundred rounds *regardless* of the maximum length of the number!

The reverse method is a step forward from the naïve implementation discussed earlier. Dropping the outlying digits removes the independent forming blocks at those ends; however, the independent blocks in the middle cause this method to produce repeating sequences.

Reverse and Replace

To our surprise, the reverse and replace method actually produced sequences that repeated more quickly than the simple replace method did. We had thought that by replacing the non-1 digits at every round, we would eliminate the independent blocks and, in effect, form new ones.

Although this theory sounds plausible, there are 2 reasons why this method does not work. After running this

method a number of times, we have noticed that the length of the number grows linearly. This is quite slower than our observed explosive growth of non-replace methods implemented for this paper.

But ultimately, this method fails because replacing non-1s with zeros at every round, the main operation in this method, actually facilitates in producing the independent forming blocks. As explained in our results section, the block '20' will become '00' and eternally remain the same independent block. This is just one example of blocks that never change.

Middle Permutation

Among all the permutations we have attempted in this paper, the middle permutation method was the only method that did not repeat after a few hundred rounds (it usually repeats after 3,105,406 rounds, and from our results, the lowest number of rounds is 137,360).

The middle permutation method solves the problem of independent forming blocks, so it does not repeat as quickly. By mixing the numbers randomly, we eliminate most blocks and results to a more repeat-averse sequence.

Note also that this method also allowed us to introduce numbers such as 4,5 and 6 back into the sequence.

This also split up and introduced new blocks into our sequence.

Conclusion

Although current day pseudo-random number generators do a good job of spawning random numbers, the question of finding a quicker generator always remains eternal. Our methodology of using John Conway's Cosomological sequence is completely atypical in the field of generating random numbers. However, we believe that such a method has promise and, depending on certain parameters, may even prove quicker than the norm.

In this paper, we've explored numerous ways to apply the cosmological theorem to generate random numbers. We've discovered that our initial attempt (the naïve method) produced repeating numbers early in the sequence. Our further attempts at obfuscating the sequence resulted in the reverse and reverse and replace methods. However, these too have proved unsatisfactory since they, too, repeat.

The middle permutation method, on the other hand, has been more successful than its counterparts. The method generates numbers much the same way the naïve implementation does, but mixes up the numbers and thus breaks out of the blocks it forms. This, as you have seen in our results, have resulted into more unpredictable

numbers in the sequence. The first observed repition was seen after 3 million rounds. Although a large number, 3 million is not enough for us to conclude that the middle permutation Conway sequence can be a full-fledged pseudo-random number generator. However, more operations can be built upon this to make the sequence more repeat-averse. Permutations are definitely a step in the right direction to make Conway's sequence a PRNG.

Further Work

As our conclusion suggests, permutations are the way to go. We can also use various permutations each round. Instead of just having 1 set permutation, lets say we have 5 different permutations, if round%5 = I, then we will use the Ith + 1 permutation to get the next string. This strategy will help randomize even further our resulting strings. Another strategy suggested is to use substitution tables (see modifications iv). These strategies seem to solve some problems but others may arise. Further work may be done to test these strategies. Once we have a strong enough sequence that takes a while to repeat (say 10 billion) we can perform more randomness tests using various tools such as the NIST battery of statistical tests. The NIST package contains tests such as runs, approximate

entropy, cumulative sum, template matching and others [NIST].

Acknowledgement

We would like to thank Dr. Washington for helpful remarks on an earlier version.

References

[C] J. Conway, The weird and wonderful chemistry of radioactive decay, in: "Open Problems in Communication and Computation", T.M. Cover and B. Gopinath, eds., Springer, 1987, pp. 173-188.

[EZ] S. Ekhad and D. Zeilberger, *Proof of Conway's Lost Cosmological Theorem*, Electron. Res. Announc. Amer. Math. Soc. 3 (1997), 78-82; available at

http://www.math.temple.edu/~zeilberg/mamarim/mamarimPDF/horton.pdf

[K] Knuth, D. E., *Seminumerical Algorithms* (Second ed.), Volume 2 of *The Art of Computer Programming*. Reading, Massachusetts: Addison-Wesley (1981).

[LM] B. La Macchia, *Discrete Exponentiation*. Available at http://www.-farcaster.com/papers/crypto-field/node1.html

[NIST] NIST, *Random Number Generation and Testing.* Available at http://csrc.nist.gov/rng/

Appendix

Conway 1: Conway's Naïve Implementation

```
string conway1(string seq, int rounds, int length) {
   for (i = 0; i < rounds; i++) {
      string newSeq= "";
      // continue until current string is passed over
      for (int j = 0; j < seq.length(); j++) {
         char charac= seq[j];    // determine the current character
         int count= 1;           // naturally, this is a count of 1

         // keep on looking forward (or backward in the reverse case)
         // while you see characters similar to charac.
         // Exception: Must be greater than zero
         while (j >= 0 && seq[j+1]==charac) {
            j++;
            count++;
         }

         // put count and character back in new sequence
         char tmp[3];
         sprintf(tmp, "%d%c", count, charac);
         newSeq= newSeq + string(tmp);
         // cut everything that is beyond 100
         newSeq= newSeq.substr(0,length);
      }
      seq= newSeq;
   }
   return replace(seq);
}

string replace(string seq) {
  int len= seq.length();    // sequence length
  for (int i= 0; i < len; i++) {
    if (seq[i] != '1')
       seq[i]= '0';
  }
  return seq;
}
```

Conway 2: Conway's Reverse

```
string conway2(string seq, int rounds, int length) {
   for (i = 0; i < rounds; i++) {
      string newSeq= "";
      for (int j = seq.length() - 1; j >= 0; j--) {
         char charac= seq[j];             int count= 1;
         while (j >= 0 && seq[j-1]==charac) {
            j--;
            count++;
         }
         char tmp[3];
         sprintf(tmp, "%d%c", count, charac);
         newSeq= newSeq + string(tmp);
         newSeq= newSeq.substr(0,length);
      }
      seq= newSeq;
   }
   return replace(seq);
}
```

Conway 3: Conway's reverse + substitute every round

```
string conway3(string seq, int rounds, int length) {
   for (i = 0; i < rounds; i++) {
      string newSeq= "";

      for (int j = seq.length() - 1; j >= 0; j--) {
         char charac= seq[j];
         int count= 1;
         while (j >= 0 && seq[j-1]==charac) {
            j--;
            count++;
         }
```

```
         char tmp[3];
         sprintf(tmp, "%d%c", count, charac);
         newSeq= newSeq + string(tmp);
         newSeq= newSeq.substr(0,length);
      }
      seq= newSeq;
      seq = replace(seq);

   }

   return seq;

}
```

Conway 4: Conway's Middle Permute

```
string conway4(string seq, int rounds, int length) {
   for (i = 0; i < rounds; i++) {
      string newSeq= "";
      for (int j = 0; j < seq.length(); j++) {
         char charac= seq[j];
         int count= 1;
         while (j >= 0 && seq[j+1]==charac) {
            j++;
            count++;
         }
         char tmp[3];
         sprintf(tmp, "%d%c", count, charac);
         newSeq= newSeq + string(tmp);
         newSeq= newSeq.substr(0,length);
      }
      seq= newSeq;
      seq = permute(seq);
   }
   return seq;
}

string permute(string seq)
{
  int len = seq.size();
  int middle = len/2-1;
  int inc = middle;
  int dec = middle+1;

  string temp = "";

  for(int i=0;i<len/2;i++)
    {
      temp += seq[inc--];
      temp += seq[dec++];
    }

  return temp;
}
```