Appendix A
Presentation at SECAN-Lab Meeting Dagstuhl 10th and 11th December 2018
(with minor modifications and extensions, worked out for the reader)

A few probability considerations concerning the repetitions on 32-bit RN

A useful test for checking produced Random Numbers consists in searching the first repetition in the produced string. This test was first presented by Gil, Gonnet and Petersen (A Repetition Test for Pseudo-Random Number Generators, Monte Carlo Methods and Appl. Vol. 12, No 5-6, pp. 385-393 / 2006).
As reported in the original paper, the authors executed only three times 100 cases for every RNG under test, and calculated the average of these 300 test-cases. In our opinion this is absolutely inaccurate to get a revelatory result. That's the reason why we increased the number of cases to 100 billion, and included the calculation on the probability of the run-length per case.
After detecting the first repetition, we store the quantity of produced Random Numbers (the run-length), the position in the string of the identical value to the repetition, the value of the repetition, and start the next search. On average we will need near 82'138 different Random Numbers (32-bit integers) before we encounter the first repetition. For a better understanding we illustrate with the following example:
The RNG under test produces the following random numbers

```
12589,  8419, 42973,   415,  1824,  61845, 72492, 64821, 42973
 No 1   No 2   No 3   No 4   No 5    No 6   No 7   No 8   No 9
```

- **9** is the run-length, as 42973 is the first repetition (of the No 3)
- **3** is the position in the string of the identical value to the repetition
- **42973** is the value of the repetition

For all of the 100 billion cases we dispose of these three values, memorized in the order of their appearance. This huge amount of data allows very extensive statistics about the RNG on test.
But be aware: As the correct generator for the production of 32-bit random numbers has no memory concerning the generated numbers, the second number may be the same as the first, and the search may end at the run-length of two. In contrast, if we do some 100 billion runs, it is possible in practice to find a case with the run-length of 460'000 or above.[1]. If we have stored these 100 billion different cases, we may calculate how many cases with run-length 2, 3, 4, … up to the maximal run-length encountered. Theoretically it is not excluded that we encounter a case that excels even a run-length of 2'100'000, as the p-value (from 0 to 1 for always) for the run-length from 2 to 2'100'000 is not exactly 1. As we shall see later, the p-value therefore starts with 222 '9' after the decimal-point, so don't worry!
To do calculations about the probability and the numbers encountered, we need a table with the p-values per run-length. As a programmer I will not start to work with long formulas, but I do it in a logical way, calculating step by step with a very good precision[2]. For these purposes I am using for 30 Years the bc from Xenix, Unix, and now from Linux. bc stands for "Basic calculator", an arbitrary-precision calculator language [3], introduced a long time ago by the Unix programmers for their own use.
With a text-editor we write the following bc-program: [4]

```
scale = 230
expecval=0
ptot=0
pnot=1
pyes=0
b=2^32
 for (i=0; i<2100000; i=i+1)
  {pyes=i/b
   pyes=pyes*pnot
   ptot=ptot+pyes
   expecval=expecval+((i+1)*pyes)
   pnot=pnot*((b-i)/b)
   print "p[",i+1,"]= ",pyes,"\n"
   print "t[",i+1,"]= ",ptot,"\n"
  }
print "\n/* Number of trials: */\nn = ",i,"\n\n"
print "/* Expected Value: */\nptot = ",expecval,"\n\n"
print "/* Last pnot: */\npnot = ",pnot,"\n\n"
quit
```

After typing in this program in the file “repet32.bc”, we run the calculation and direct the result into the file “repet32-230dec_bis_2100000.bcx”

```
n81:/o # time bc repet32.bc > repet32-230dec_bis_2100000.bcx
```

```
real  1m36,453s
user  1m22,689s
sys   0m12,838s
```

For calculating the probability in 2.1 million cycles, every cycle with three multiplications and one division with the precision of 230 decimals, we needed 1 minute and 36 seconds!

The file produced:

```
n81: /o # l *230dec*
```

```
-rw-r--r-- 1 root root 1047777896  9 jun 18:13 repet32-230dec_bis_2100000.bcx
```

(I use the suffix .bcx for textfiles ready to serve as input to the bc program)

Let’s take a look to the end of the file:

```
p[2100000]= .0000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000004898
```

```
t[2100000]= .9999999999999999999999999999999999999999999999999999999\
99999999999999999999999999999999999999999999999999999999999999999999\
99999999999999999999999999999999999999999999999999999999999999999999\
999999999999999999999999999999987886448
/* Number of trials: */
n = 2100000
/* Expected Value: */
 ptot = 82137.861971368955568685051288766922397675686166540131869069\
00647517014174222327104971548616452342498172020465735021030669718070\
56392826059475265718821948877463868903010027078582018493953365515193\
1500573839602698058934660664591370685930
/* Last pnot: */
 pnot = .00000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000010013560
```

The p-value to encounter the first repetition after the position 2’100’000 is very small. The total of all p-values begins with 222 times 9 after the decimal-point. Considering this, we may conclude that it is possible to trust at least the first 210 decimals of the expected value (ptot), without any calculation about the error dimension $^{5)}$. In the original paper a value of 82’138 without decimals is indicated.
If we want to know the highest probability of all cases, we load the table with the program bc:

```
n81: /o # bc repet32-230dec_bis_2100000.bcx
bc 1.07.1
```

```
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
```

Then we have to type the following line:

```
for (i=2;i<2100000;i++) if (p[i]>p[i-1]) j=i
```

By typing `j` we get the result: **65537**

Let's check in the file:

```
p[65536]= .0000092549704677926530842180406598563353344254312379695552\
61086703504914989035009984507884125075454350550977062558847757759803\
92685975591151881868286477583190011229936657105059081938575816073571\
30190833106458727970590918328514522 55
t[65536]= .393466255281518890451359668299586837036235468877090996066\
30080766853459040816767902735222850184663820817788134547937232268246\
20918800011696733538374539867197237129403845709727213968623080895855\
26583123724860994722036472965598884 80
p[65537]= .0000092549704699475266959936574050966363977625203113236 84\
75240629871111244826647693360859142717740102175585666074607117656978\
32876267718505680897010217673647045940595559633701938976837636976915\
04127096815128710006794402519638586 43
t[65537]= .393475510251988837978055661956991933672633231397402319751\
05321396724570285643415596096081992902403922993373800622544349925224\
53795067730202414435384757540844283069999405343429152945460717872770\
30710220539989704728830875485237471 23
p[65538]= .0000092549704677926859644782184814858761715927710556211 98\
41819517908530044797153628308344318315952819575112961060653580425060\
10983126618118529609489521987071504278159803116920405925159205994785\
94312732653756735542491354656152735 45
t[65538]= .393484765222456630664020140175473419548804824168457940949\
47140914633100330440569224404426311218356742568486761683197930350284\
64778194348320944044874279527915787348159208460349558870619923867556\
25022953193746440271322230141390206 68
```

As I stated in the beginning, it is possible to encounter a case with a run-length of only two Random Numbers, that means that two identical 32-bit Random Numbers appear without any other between. Let's use the term "double" for this case. Considering the probability of 1 out of 2^{32} (4'294'967'296), we expect in 100'000'000'000 runs only a total of 23,28306436… cases.

Knowing the number of our test-cases (100 bill.) and the average run-length of a case (ptot), we may calculate the total number of expected doubles.

To limit ptot to the trusted value we truncate to 210 decimals:

```
scale=210
ptot=ptot/1    /* to cut the last 20 digit */
scale=230     /* to get a better precision */
totrn=ptot*100000000000
totrn
8213786197136895.5568685051288766922397675686166540131869069006475 17\
01417422232710497154861645234249817202046573502103066971807056392826\
05947526571882194887746386890301002707858201849395336551519315005738\
396026980589000000000000
totrn=totrn-1 /* we have to substract one, as the first can't be a double */
totrn
```

```
8213786197136894.556868505128876692239767568616654013186906900647517\
01417422232710497154861645234249817202046573502103066971807056392826\
05947526571882194887746386890301002707858201849395336551519315005738\
39602698058900000000000
```

Now we consider those Random Numbers as a unique string.

```
doubles_a = totrn * (1/2^32)
doubles_a
1912421.127114653251336073765735303107594970757293983638963406124971\
1118112835863383886284698854066660096916523145095050962916739666674\
42290933470214296804681976067438023795340722857368024194501180901596\
8805241818979848176240921020507812
```

That’s the expected number of doubles; but where are the missing doubles?

By definition there are no doubles possible before the first repetition. But in fact, the first repetition may form with the previous Random Number a double. For a run-length of two, the p-value is 1, and for a run-length of 50’000, the p-value is only 1/49’999, as the first repetition may be the same as any of the 49’999 different numbers.
Now we may calculate the average probability for doubles by integrating the different probabilities for the specific run-lengths from 2 to 2’100’000 divided by the decreased probability to get a double with the first repetition and the number just before.
Let’s say immediately, after this calculation we still miss around 23 doubles! Where to find?
We have to consider also the case of a double between the first repetition and the first number of the next beginning. For 100 billion cases we have to count 99’999’999’999 cases, as there is no possibility after the last run. Again, the probability is 1 out of 4’294’967’296 cases.
If we have not yet left the bc program with the probability table, we calculate first the probability for a double per case :

```
scale=230
for (i=2;i<2100001;i=i+1) s+=(p[i]/(i-1))
s
.0000191239784405028809720382679460493201384497075729398363896340612\
49711118112835863383886284698854066660096916523145095050962916739666\
66744229093347021429680468197606743802379534072285736802419450118090\
159688052418189806550787251
```

The value s being the probability per case, we need to multiply s with 100 billion, and to add the expected number of doubles formed by the repetition and the beginning of the next run in 99’999’999’999 possible cases:

```
doubles_b = (100000000000 *s) + (99999999999*1/2^32)
doubles_b
1912421.127114653251336073765735303107594970757293983638963406124971\
1118112835863383886284698854066660096916523145095050962916739666674\
42290933470214296804681976067438023795340722857368024194501180901596\
88052418189806550787251000000000000
```

This is the expected number of doubles, calculated alternatively on the base of the number of cases plus the probability to have the repetition and the first of the next run as double.

We compare the result double_a with the result double_b, and we find them identical up to the 207. decimal.

Control:

```
scale=207
doubles_a = doubles_a / 1    /* to reduce the decimals */
doubles_b = doubles_b / 1
doubles_a - doubles_b

0
```

The result being the same with method a as with method b, we have demonstrated that first our average is correct, and second that our calculated probability table is correct as well. If we forget, in the first calculation, to deduct one from the 8.2137 million billion, a difference from up the tenth decimal should appear.

Thank You for listening / Alain Schumacher

---

1) If we do 100 billion times this test, the computed expectation to find cases with run-length 460’000 or above is 2.001962… cases, and only at 466’436 and above the expectation became smaller then 1 case, 0.99960056977… are expected.
2) As I am not a mathematician, I try first to approach a given problem with an ordinary logic. My goal being to compute the table with the expected probabilities, I have only to consider the following facts: After producing the first random value, of course no double is possible. After the generation of the second value, the probability to obtain the same value is one-to-2exp32. For the third and the followings, the probability increases with the number of different values produced before (n-to-2exp32), but is reduced by the probability that this cycle was already finished before. This way I feel more comfortable to program the job, as I use bc from LINUX, and no special software for mathematician.
   I guess that mathematicians will now start to laugh at such an approach, but may I ask them to take a look at the book “ars conjectandi 1713” from Jacob Bernoulli. They will see that there is no difference between his style of describing the probability problems and mine. As Jakob Bernoulli is considered as an eminent mathematician, I feel free to extend his research with the possibilities of modern HPC equipment.
3) For more details on bc see the article “bc (programming language)” in the English Wikipedia.
4) You need a version, like GNU bc, which supports more than one character for variable names. If not available, you may take different characters for every different variable.
5) To control the correctness of our estimation, we compared the result of ptot with 230 decimals up to 2’100’000 cases, against a file calculated with 600 decimals up to 3’400’000 cases. Both results for ptot are identical on the first 216 decimals.