What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ **and Nigel P. Smart** ⟨csnps@bristol.ac.uk⟩

git # 148def3 @ 2018-07-11

GENERATION AND TESTING OF RANDOM NUMBERS

It might seem unlikely, but there are some really great stories about **randomness** [15]. The way Michael Larson used knowledge of the *lack* of randomness on the US game show Press Your Luck in 1984, for example, is so great that it warrants being made into a film of some sort [13]. Part of the game involved the players moving around an eighteen square board. The squares were either empty, contained a prize or contained the so-called Whammy character: landing on a prize square won you that prize, landing on the Whammy square lost all prizes won so far.

To make things exciting, the contents of the squares was updated every second or so in a "random" manner. Except it was not random at all. Larson video taped Press Your Luck episodes and played them back frame-by-frame. Then, by writing down the sequence of board states, he discovered that the board in fact cycled through just five simple patterns. Better still, during a given turn there were some squares that would never contain the Whammy. So armed with this knowledge, Larson reasoned that he could carry on playing without really gambling at all: provided he could remember the patterns and which turn he was on, he could *always* avoid the Whammy. Larson went on the game show and stayed on so long it had to be split into multiple episodes; the look on the presenters face as he consistently avoided the Whammy with seemingly steel-eyed bravery must have been priceless. Well, not exactly priceless: Larson walked off with $110,000$ after lawyers from the TV station conceded that he had not cheated. You can still find video of the now legendary episodes on YouTube:

http://www.youtube.com/results?search_query=Press+Your+Luck

I hear a more mundane story much more often: my mother has a love/hate relationship with the UK National Lottery game Lotto [11]. The idea is that she picks six numbers between 1 and 49 and then every Saturday, a machine selects six numbers at random. If her numbers match the ones the machine picked, she wins something: typically the more numbers that match, the more money she gets. But invariably the numbers do not match and she is left to contemplate the injustice of it all. This takes the form of a ritual tirade against the machine: "it's a fix, having 21 *and* 22 *cannot* be random".

Examples like this beg some questions about what random numbers are and how we generate them. These are quite important questions because random numbers are used in lots of different areas of Computer Science. A good example is that the security of many cryptographic schemes relies on the fact that one can make random choices and choose random numbers. For example, it is common to assume that any key we choose is done so in a random way; if there was some way to predict how we selected it, the key would be more easily guessed and security more easily breached. Setting a password to "X4$ia0!l" is arguably better than "password" for example!

1   What *is* randomness?

The term **entropy** is often used by scientists to describe disorder: if a physical system has high entropy it behaves in an unpredictable way. For example as you heat up a gas, predicting how it will behave starts to become harder than if it is in a more stable, cooled state. When we say something is random we mean more or

less the same thing: the behaviour we observe follows no deterministic or predictable pattern. As a result we can not write an algorithm to describe it; instead we have to describe it in terms of **probability**, or chance.

Randomness is quite an abstract concept. To make things more concrete, we will talk exclusively about sequences of random numbers. Imagine we want to generate a sequence of such numbers, e.g.,

$$X = \langle 0, 1, 0, 1 \rangle.$$

The idea is to use coin flips (or tosses) [3]: in order to generate each X_i, we first throw the coin in the air then and inspect which side it lands on. A real coin has two sides, normally called the tails side and heads side, but we can make life easier by using the number 0 with tails and 1 with heads; we also rule out any freak occurances such as the coin landing on anything other than one of the sides (e.g., on the edge). As a result, we can say each coin flip makes a random selection from the set $S = \{0, 1\}$. Clearly we cannot write down an algorithm to describe how the coin behaves, so instead we use a **probability distribution**:

$$P(x) = \begin{cases} \frac{1}{2} & \text{if } x = 0 \text{ i.e., the coin flip was a tail} \\ \\ \frac{1}{2} & \text{if } x = 1 \text{ i.e., the coin flip was a head} \end{cases}$$

Looking at P, for a given output x we can say what the probability of selecting x from S is. In this case, we have a special name for the probability distribution: when the probability for each x is the same we say the distribution is **uniform**.

The subject of randomness is one where it is quite difficult to prove things categorically. More usually, we define randomness in terms of properties which we can measure. Typically we generate a sequence of random numbers and then use statistical tests of randomness [17] on the sequence. For example we might say that a sequence contains some feature that mean it is *not* random; we discuss two such features below.

1.1 Biased versus unbiased

Imagine we get a friend to flip a coin eight times, resulting in the sequence

$$Y = \langle 1, 1, 1, 1, 1, 1, 1, 0 \rangle.$$

We would have to ask ourselves whether or not this friend is cheating somehow: intuitively, we might say that the coin is **biased**. On the other hand, this sequence is just as probable as any other. The probability of getting Y is

$$\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2^8} = \frac{1}{256}$$

which is the same as any other sequence of the same length. So given Y is just as likely to occur as any other sequence, why would we conclude that the coin is unfair?

In more general terms, if we select uniformly at random from a set of m numbers the probability of selecting a number x is

$$P(x) = \frac{1}{m}.$$

If we repeat the selection n times, we would expect each of the numbers to appear roughly $\frac{n}{m}$ times *on average*. If some number x is selected significantly more or less than $\frac{n}{m}$ times, we could conclude that the selection process is not random at all: it is biased in favour or against x somehow. This means that there exists a number x with $P(x) \neq \frac{1}{m}$, but we expected that for all x we would have $P(x) = \frac{1}{m}$. This is another way of saying the selection process is biased, i.e., P is not uniform.

The "on average" part in the previous paragraph is important in the sense that we need n to be large before we start talking about average behaviour. For example if we flip a coin twice and get 1 twice, we cannot conclude that the coin is unfair because we do not have enough evidence. If we flip a coin eight hundred times and get 1 seven hundred times however, we can be more confident that something fishy is going on.

Looking again at the sequence Y, we can start to see why our friend might be cheating. We are selecting from $m = 2$ numbers and have repeated the selection $n = 8$ times so we would expect each number to occur $\frac{8}{2} = 4$ times. But they do not: we get 1 seven times and 0 just once, so we could conclude that for this limited sample the coin is biased toward 1 and is therefore unfair. Rather than being uniform, based on having seen Y we might say the behaviour of the coin is better described by

$$P(x) = \begin{cases} \frac{1}{8} & \text{if } x = 0 \\ \\ \frac{7}{8} & \text{if } x = 1 \end{cases}$$

Maybe we should get some more trustworthy friends!

> **Implement (task #1)**
>
> You can get some statistics about the UK National Lottery here
>
> http://www.lottery.co.uk/statistics/
>
> Based on the discussion above, i.e., explaining your answer in terms of Mathematics, probability in particular, is it biased or not?

1.2 Predictable versus unpredictable

Now imagine we get a different friend to flip another coin eight times, and that this time the resulting sequence is

$$Z = \langle 1, 0, 1, 0, 1, 0, 1, 0 \rangle.$$

The probability of getting this sequence is still

$$\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2^8} = \frac{1}{256}$$

so again we cannot really infer anything from it occurring rather than some other sequence. Also, we can see that the coin is not biased in the same way the other one was: we get 1 four times and 0 four times so the coin is not biased toward either case. On the other hand, we might intuitively say the new coin is still unfair because there is a clear pattern: if a given flip of the coin gives 1, there seems a strong chance the next flip will give 0. If we believe the pattern will continue to hold, this means we can predict the next result with some confidence.

You can think of this in terms of **conditional probability** [2]: the result of a given coin flip should not depend on anything other than the probability distribution. It especially should not depend on the results from previous coin flips: imagine you flip the coin ten times and every time you get 1. We might say a 0 was "due". But looking at our description of the behaviour, it clearly is not: the probability of getting a 0 on the eleventh flip is still $\frac{1}{2}$ and the probability of getting a 1 is still $\frac{1}{2}$. The tendency for people to ignore this is sometimes called the gambler's fallacy [6]. In our case, we can see that the result of the i-th coin flip probably is dependent on the $(i-1)$-th coin flip. For example if $Z_{i-1} = 1$ then it is more probable that $Z_i = 0$ than $Z_i = 1$. Therefore we could conclude that the coin behaviour is not as well described by the probability distribution as expected.

There is another, perhaps neater way to think about this. Imagine we want to take Z and compress it by coming up with a shorter way to describe the sequence, just as we did in Chapter 2. For example, imagine we say that the symbol $\star$ represents the sequence $\langle 1, 0 \rangle$. We might then describe Z as the sequence

$$\bar{Z} = \langle \star, \star, \star, \star \rangle.$$

Why does this work? Well, we know what $\star$ "means" so we can take $\bar{Z}$ and reconstruct Z by just replacing each occurrence of $\star$ with $(1, 0)$. But the way we describe $\bar{Z}$ is shorter than the way we describe Z (even if we include the definition of $\star$), so in some sense we have used the fact there is a "pattern" to compress the information. We have glossed over a lot of the detail, but there is a fancy name for this general concept: we call it Kolmogorov-Chaitin complexity [8]. Very roughly, this concept says that the more we can compress a sequence the less random it must be; equivalently, the more we can compress a sequence the more usable structure there must be in the sequence.

1.3 Random versus arbitrary

There used to be a joke about messages from computers that would prompt the user to "press any key" [1]; the joke was that users would find the space key and the escape key, but could not find the "any" key. It seems debatable whether this was ever actually funny, but looking at things more closely highlights an important point: the message did not read "press a random key", the choice of key does not matter so really what we mean is "press an arbitrary key".

The difference between arbitrary and random is sometimes important however. For example, many files used within a UNIX-based operating system start with a magic number [10] which allows us to easily identify their type; if the file starts with the number 8993 this tells us that we can execute it for example. In a sense the choice of 8993 is arbitrary; there is no real reason to choose 8993 rather than say 1234 or 9999, we just needed "any" number. In this sort of situation, it is tempting to just select the numbers at random.

> **Implement (task #2)**
>
> Test this claim using BASH. Remember, the `od` command can display the content of files in various ways: the option `-tux1` will print the content as a sequence of hexadecimal bytes for instance. If you can locate some different file types (e.g., JPEG images, PDF documents, BASH scripts, Java class files and so on) you should be able to see the magic number in the first few bytes (the `file` command uses this information to guess the file type).

In cryptography, this approach can have some disadvantages. The most famous example is illustrated by the **Data Encryption Standard (DES)** [4] which was designed and standardised in the US in the late 1970*s*. DES is a block cipher; it encrypts messages. One of the components in the algorithm is a large table called the S-box whose contents is chosen carefully but somewhat arbitrarily. The problem is, there was no explanation of *how* the content was generated. People began to become suspicious that the content had been chosen in a special way so that, for example, the US government could decrypt their messages. In turned out that the US government had not selected the S-box content either randomly *or* arbitrarily. They had actually been selected to prevent an attack which, at the time, only the US governments cryptographers knew about. Even so, the lack of openness over the design choices for the S-box sparked a trend toward use of so-called "nothing up my sleeve" numbers [12]. This idea is used when we want "any" number, but one which we can prove has not been selected with special properties. For example, the number π might not be a good random number, but if all you need is an arbitrary number then it is probably a good choice. For example, people would be hard pushed to prove your choice of π was underhand: it cannot have been generated in some special way.

2 Real randomness

2.1 Generating randomness

When we talk about *real* random numbers, we typically mean numbers that come from some physical process. The idea is that we cannot control, or to some extent understand, how these processes might work. In other words, we cannot write down an algorithm that describes them. Radioactive decay is a good candidate. The idea is that an unstable atom will decay by emitting energy in the form of radiation; we know the average rate at which this might happen, but when exactly a given atom will decay is unpredictable. So we could generate random numbers by simply taking a measuring device such as a Geiger counter [7] and have it tell us whenever radioactive decay is detected.

Another candidate is atmospheric noise. In the same way as radioactive decay, it is difficult to predict the level and characteristics of the noise around us. Sampling such noise using even a basic radio or microphone can give quite effective results: Mads Haahr, a lecturer at Trinity College, Dublin has rigged up such a system to the Internet at

<div align="center">

http://www.random.org/

</div>

The web-site offers a neat interface which we can easily use from BASH by employing the `wget` command. The idea is to issue a command that mimics what happens when we type a URL into the address bar of a web-browser:

```
<tegers/?num=100&min=0&max=255&col=5&base=10&format=plain&rnd=new'
243     161     55      11      19
205     21      115     159     68
1       236     213     147     155
31      10      84      94      14
186     168     43      170     238
180     84      249     155     64
183     49      77      215     157
227     86      249     137     239
211     163     103     28      188
134     34      43      152     123
202     8       218     185     222
183     236     79      53      30
195     103     227     236     2
62      254     148     24      179
44      204     7       177     138
164     60      129     219     44
230     174     99      210     84
147     60      222     148     12
27      148     169     168     205
121     70      72      177     210
bash$
```

Of course having a Geiger counter attached to *every* computer is not ideal, and neither is having to access a remote computer over the Internet every time we need to generate random numbers. Fortunately there are lots of devices already connected to your local computer which could do a similar job. Most operating systems have a mechanism for collecting together random events from such devices into what is called an **entropy pool** [5]. You can think of the entropy pool as a sequence of numbers; the idea is that each time a random event happens, we mix it into the entropy pool by adding some numerical representation of the event onto the end of the sequence. You could imagine that every time someone moves the mouse, the computer might add the mouse speed and direction to the sequence. Then, when someone or something needs to generate a random number, we take one from the start of the sequence.

UNIX systems commonly have two types of entropy pool which they let users access via the `/dev/random` and `/dev/urandom` files. Except the files are not *really* files at all: when we read from them, behind the scenes

we are taking random numbers from an entropy pool. The difference between the two is that /dev/random will wait for enough entropy to exist before allowing us to read from it, while /dev/urandom will let us read whenever we want. You can see this as a choice between the *quality* of the numbers we read and the length of *time* required to read them. On one hand, if there have not been enough random events to fill the entropy pool it does not make sense to start taking numbers from it; the numbers we read from /dev/random should be more random. On the other hand, waiting until enough random events happen might take a long time; although they might be less random, we can read numbers /dev/urandom with less of a delay.

Again, we can easily read some numbers from /dev/urandom using BASH:

```
bash$ cat /dev/urandom | od -Ad -tu1 -w5 -N50 | cut -c 9-
 21 150   7 129 204
231 101  44   9  33
244 130 163 100  94
 62  12 192 130 148
 82  36  17 243  23
 39  26  76  22 177
158  30 206 110 194
119  40  15 133  85
154 176 251 151 115
 40 120  18 131  48

bash$
```

That is quite a horrible looking command, so it makes sense to look at what is going on in more detail. Basically, we take /dev/urandom and pass it through a command called od which is controlled using a number of options: we tell od to give us fifty bytes of the input using -N50, to format the output in five columns using -w5, and to format the content as unsigned decimal integers in the range $0 \ldots 255$ inclusive using -t with the format u1. Finally we pass the output of od through cut to remove some information we do not need, i.e., to get just the random numbers.

2.2   Testing randomness

So we can generate random numbers, but can we apply similar statistical tests as those discussed earlier to *show* they are random? As an example, we will look at the tests for bias and predictability; we will compare the real random numbers against English text, which is not random at all. Again we will use Project Gutenberg as a source of text:

http://www.gutenberg.org/

For the sake of argument, we fetch the text for *War and Peace* by Tolstoy (which is quite large) and save it as the file A.txt. Using the du command we can see that it weighs in at roughly 3MB:

```
<A.txt 'http://www.gutenberg.org/files/2600/2600.txt'
bash$ du -b A.txt
0       A.txt
bash$
```

Next we take the same amount of random data from /dev/urandom (using du and cut to provide the number of bytes as an option for head), then save it as the binary file B.bin as follows:

```
bash$ cat /dev/urandom | head -c `du -b A.txt | cut -f 1` > B.bin
bash$
```

Remember that testing for bias in A.txt and B.bin basically means testing that each possible number we could select has the same probability of selection. The test will be performed in two steps:

1. Take the input file and chop it up into decimal numbers in the range $0 \ldots 255$. The output will need some cleaning up, but the end result will essentially be a long sequence of decimal numbers (one per-line) that represents the file content.

2. Take the sequence and count how many times each number occurs in it.

There are two main things to notice in the output for A.txt:

```
bash$ cat A.txt | od -Ad -tu1 -w1 -v | cut -c 9- | grep [0-9]* > D.txt
bash$ cat D.txt | tr -d [:blank:] | grep -v  | sort -n | uniq -c | paste -s
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.

bash$
```

First, some numbers do not appear at all; for example the output starts at 10 so clearly 0 . . . 9 appear zero times. Second, among the numbers that do appear, some appear much more often than others. For example the number 32 appears many times whereas the number 90 appears fewer times. The reason for this is simple. Since `A.txt` is an ASCII text file, the numbers will be biased toward those which relate to printable ASCII codes. Looking back to Chapter 5, we find that 32 is the ASCII code for *SPC*, the characters 'a' . . . 'z' have ASCII codes 97 . . . 122, and ASCII codes 10 and 13 produce a new line; unsurprisingly these all appear *very* often! What about `B.bin`?

```
bash$ cat B.bin | od -Ad -tu1 -w1 -v | cut -c 9- | grep [0-9]* > D.txt
bash$ cat D.txt | tr -d [:blank:] | grep -v  | sort -n | uniq -c | paste -s
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.

bash$
```

The difference is quite obvious: all numbers appear in the output and do so roughly 12600 times on average. There are a few cases which vary a little, but basically we can conclude that there is no bias in `B.bin` to the same extent there was in `A.txt`. We might use some further statistics to back this up, for example if we were to measure the standard deviation of `B.bin` it would not be very large.

Measuring predictability is a little more tricky. One thing we could do quite easily is to approximate the Kolmogorov-Chaitin complexity by seeing how easy it is to compress the file content. The command `bzip2` performs a full blown version of the compression ideas we discussed in Chapter 2. `bzip2` processes an input file to identify patters and replaces them with shorter symbols to produce a compressed output file. It needs to define what those symbols mean so it adds a dictionary to the start of the compressed file so that we can decompress it later if we want to. Using `bzip2` to compress `A.txt` and `B.bin` is simple; to be fair, we instruct `bzip2` to make the best effort it can at compressing the files rather than worrying about producing the result quickly:

```
bash$ bzip2 -c -9 A.txt > A.txt.bz2
bash$ bzip2 -c -9 B.bin > B.bin.bz2
bash$
```

We then inspect how big the resulting files are, again using the `du` command we used earlier:

```
bash$ du -b A.txt A.txt.bz2
0       A.txt
14      A.txt.bz2
bash$ du -b B.bin B.bin.bz2
0       B.bin
14      B.bin.bz2
bash$
```

Inspecting the results confirms more or less what we expected: the *War and Peace* text `A.txt` was compressed into `A.txt.bz2`, which is roughly a quarter of the size. This is what we would expect since `A.txt` represents English text; for example the word "and" probably appears very often so we might imagine replacing each occurrence with a one character symbol. However, the randomness produced by `/dev/urandom` in `B.bin` is processed by `bzip2` to produce the file `B.bin.bz2`. In this case the output of `/dev/urandom` is so hard to compress that the overhead of adding the dictionary to the compressed file has meant the end result is larger than what we started with!

Maybe `bzip2` is just not very good at compressing files (or at least this file? To test this, we can try the same thing with some other compression tools like `gzip` and get similar results:

```
bash$ gzip -c -9 A.txt > A.txt.gz
bash$ gzip -c -9 B.bin > B.bin.gz
bash$ du -b A.txt A.txt.gz
0       A.txt
26      A.txt.gz
bash$ du -b B.bin B.bin.gz
0       B.bin
26      B.bin.gz
bash$
```

Or perhaps using the `zip` compression method:

```
bash$ zip -q -9 A.zip A.txt
bash$ zip -q -9 B.zip B.bin
bash$ du -b A.txt A.zip
0       A.txt
160     A.zip
bash$ du -b B.bin B.zip
0       B.bin
160     B.zip
bash$
```

Of course, this still does not prove anything; perhaps `bzip2`, `gzip` and `zip` are *all* missing some obvious way to compress the files. But basically we conclude from our tests that `A.txt` is quite easy to compress whereas

`B.bin` is at least harder to compress. This means the Kolmogorov-Chaitin complexity of `A.txt` is quite low: we can make a shorter description of `A.txt`, so it is not very random. On the other hand, for `B.bin` the Kolmogorov-Chaitin complexity is higher: it is not as easy to describe `B.bin` in a shorter way, so in a sense it is more random.

Using the commands described above for compressing files, set yourself a challenge: what is the

Implement (task #3)

1. *largest* file you can create that can be compressed into the *smallest* size, and

2. *smallest* file you can create that can be compressed into the *largest* size.

Put more precisely, imagine the original and compressed files have sizes x and y bytes respectively: you are trying to maximises or minimise the compression ratio x/y. In each case, explain what your strategy for constructing the original file is, i.e., what features does it have that make it a good choice?

---

3 Fake randomness

3.1 Generating randomness

The problem with real random numbers is that sometimes they are *too* random for our purposes. Yes, really: imagine we want to simulate a scientific experiment using a computer; this is a good idea if the real experiment would be impractical. For example, maybe we would like to experiment with things that expand very fast and generate large amounts of heat; usually we call these things explosions and doing real experiments can be quite hazardous! So instead, we simulate the explosion on a computer. On one hand it is quite possible we would want a source of random numbers to model parts of the experiment, for example randomness in atomic-level behaviour. On the other hand it would be a good idea if we could repeat the simulation and get the *same* results. Using real random numbers is not ideal because of the second reason: we cannot reproduce the real randomness so we would need to generate all the numbers, store them somewhere and then look them up if we needed them again.

A solution is to use **pseudo-random** numbers [14]. Whereas real randomness cannot be described using an algorithm, pseudo-randomness can. The idea is to think of the pseudo-random numbers we generate as a sequence called R. We start by specifying the first element in the sequence R_0, this is called the **seed**. Then, to generate the next element R_1 we apply the algorithm to compute it for us; we can perform this over and over again so that more generally if we have R_i, the i-th element in the sequence, we can generate the next element R_{i+1}. The thing we need to be very careful about is that the sequence we generate still passes the tests for randomness we looked at previously: although pseudo-random numbers are in a sense fake, they should still satisfy our definition of what randomness looks like. If we are successful, reproducing pseudo-random numbers becomes a matter of using the algorithm over and over again rather than storing a large amount of real random numbers: we have traded more computation for less storage.

A **Linear Congruence Generator (LCG)** is one way to generate pseudo-random numbers [9]. The algorithm we use to compute the next element of the sequence given the current element is

$$R_{i+1} = a \cdot R_i + c \pmod{p}.$$

This means that we take R_i, multiply it by some number a, and finally add another number c: the result R_{i+1} is produced by using modular arithmetic where the modulus we are using is p. Imagine we select $a = 5$, $b = 1$, $p = 8$ and set the seed value $R_0 = 2$. We can apply the LCG equation to generate successive elements in the sequence

$$
\begin{aligned}
R_0 &&&&&&= 2 \\
R_1 &= 5 \cdot R_0 + 1 \bmod 8 &&= 11 \bmod 8 &&= 3 \\
R_2 &= 5 \cdot R_1 + 1 \bmod 8 &&= 16 \bmod 8 &&= 0 \\
R_3 &= 5 \cdot R_2 + 1 \bmod 8 &&= 1 \bmod 8 &&= 1 \\
R_4 &= 5 \cdot R_3 + 1 \bmod 8 &&= 6 \bmod 8 &&= 6 \\
R_5 &= 5 \cdot R_4 + 1 \bmod 8 &&= 31 \bmod 8 &&= 7 \\
R_6 &= 5 \cdot R_5 + 1 \bmod 8 &&= 36 \bmod 8 &&= 4 \\
R_7 &= 5 \cdot R_6 + 1 \bmod 8 &&= 21 \bmod 8 &&= 5 \\
R_8 &= 5 \cdot R_7 + 1 \bmod 8 &&= 26 \bmod 8 &&= 2 \\
R_9 &= 5 \cdot R_8 + 1 \bmod 8 &&= 11 \bmod 8 &&= 3 \\
&& \vdots && \vdots && \vdots
\end{aligned}
$$

Think back to our example application of pseudo-randomness where we wanted to use a computer to simulate something, and imagine we want to start the simulation half way through. Normally this would be tricky: we would first have to start our random number generator at the beginning using the seed element, and then repeatedly compute the next element until we got to the point we actually wanted to start. Depending on where this is, we might waste quite a bit of time just finding the R_i we want.

> **Research (task #4)** The LCG is just one example of a **Pseudo-Random Number Generator (PRNG)**. Find out about at least one other type, including how it works; compare it with the LCG in terms of any advantages and disadvantages each might offer given a particular context or application.

The LCG has a nice property which allows us to avoid this wasted time by skipping ahead in the sequence. We already have a way to skip ahead by one element, namely

$$R_{i+1} = a \cdot R_i + c \pmod{p}.$$

What if we take R_{i+1} and apply the equation again? We would skip ahead two elements, and the result would look a bit like

$$R_{i+2} = a \cdot S + c \pmod{p}.$$

where $S = a \cdot R_i + c$; writing things out fully gives a way to go straight from R_i to R_{i+2}

$$R_{i+2} = a \cdot (a \cdot R_i + c) + c \pmod{p}.$$

Pulling the same trick again we can compute what the result of skipping ahead three elements would be

$$R_{i+3} = a \cdot T + c \pmod{p}.$$

where $T = a \cdot (a \cdot R_i + c) + c$; things are starting to get a bit of a mess, but we can again write things out to get straight from R_i to R_{i+3}

$$R_{i+3} = a \cdot (a \cdot (a \cdot R_i + c) + c) + c \pmod{p}.$$

The aim of all this is not to give you a headache, but rather to show that there is a pattern emerging: if we start at R_i then to skip ahead k elements means that first we multiply R_i by a a total of k times, but also add c a total of k times as we go. The problem is, the multiplications by a and additions of c are sort of mixed up. We can unravel the mess by rewriting things a bit

$$
\begin{aligned}
R_{i+3} &= a \cdot (a \cdot (a \cdot R_i + c) + c) + c &&\pmod{p} \\
&= a \cdot (a^2 \cdot R_i + a \cdot c + c) + c &&\pmod{p} \\
&= a^3 \cdot R_i + a^2 \cdot c + a \cdot c + c &&\pmod{p}
\end{aligned}
$$

The important thing to notice from the above is that given R_i we just need to compute two values which we can write down as

$$
\begin{aligned}
A(k) &= a^k \\
C(k) &= a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c
\end{aligned}
$$

and then we can use them to compute

$$R_{i+k} = A(k) \cdot R_i + C(k) \pmod{p}.$$

If we plug in the specific case of $k = 3$ the terms are as we expect, i.e.

$$
\begin{aligned}
A(3) &= a^3 \\
C(3) &= a^2 \cdot c + a \cdot c + c
\end{aligned}
$$

which gives

$$R_{i+3} = a^3 \cdot R_i + a^2 \cdot c + a \cdot c + c.$$

The great thing is, since a and c are constant values, so if we know k before we start then we can also compute $A(k)$ and $C(k)$ before we start. But what if we do not know k before we start? What is the best way to compute $A(k)$ and $C(k)$? Certainly we want to do this in an inexpensive way, otherwise we might as well perform k steps, skipping ahead 1 element each time.

The $A(k)$ part is quite easy, but the $C(k)$ part looks much less pleasant. We have got an expression of the form

$$a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots a^2 \cdot c + a \cdot c + c$$

to compute. This looks bad because for a large value of k, the number of additions and multiplications we need to do is also quite large. However, we are saved because this large expression is actually the same as the much nicer looking

$$\frac{(a^k - 1) \cdot c}{a - 1}.$$

To see why, multiply the first expression by $(a - 1)$, to obtain

$$
\begin{aligned}
(a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a \cdot c + c) \cdot (a - 1) &= (a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c) \cdot a \\
&\quad - (a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c) \\
&= (a^k \cdot c + a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c) \\
&\quad - (a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c) \\
&= a^k \cdot c - c \\
&= (a^k - 1) \cdot c.
\end{aligned}
$$

Now dividing both sides by $(a - 1)$ we obtain

$$a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c = \frac{(a^k - 1) \cdot c}{a - 1}.$$

This is great news because we already have to compute a^k in order to get $A(k)$, so given that $a - 1$ is also just a constant value we only need to do one subtraction, one multiplication and one division to get $C(k)$ rather than the mass of multiplications and additions that we started with. In other words we have

$$C(k) = \frac{(A(k) - 1) \cdot c}{a - 1}.$$

Computing the sequence an LCG produces by hand is somewhat tedious, so how might we automate the process? One way would be to write a dedicated program for the task; since we want to focus on the concepts rather than teach programming, we will instead try to automate the process using only existing BASH commands. Our approach is to write a small BASH **script** [16] which will generate n elements of the sequence given R_0, a, c and p. You can think of the script as creating a new command which we can use as follows:

```
bash$ ./P.sh n R a c p
```

In other words it will produce the output of the LCG for values of i in the range $1, 2, \ldots, n$, starting at position R_0. We create the script, which we call P.sh, as follows:

```
bash$ cat > P.sh
#!/bin/bash

R=

for (( i = 0; i < ; i += 1 )) ; do
  echo ""
  R=$[ ( (  *  ) +  ) %  ]
done
bash$ chmod 775 P.sh
bash$
```

Note that cat is used to capture input from the user and save it into a file called P.sh[1]. The permissions of P.sh are set using chmod so we can use it as a script (rather than just a normal file). For example, to replicate the sequence we looked at above, we might run P.sh as follows:

```
bash$ ./P.sh 100 2 5 1 8 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$
```

where we read ${1} as 100 meaning $n = 100$, ${2} as 2 meaning $R_0 = 2$, ${3} as 5 meaning $a = 5$, ${4} as 1 meaning $c = 1$, and finally ${5} as 8 meaning $p = 8$.

Although overall it might seem unfamiliar, each step of the script is easy to explain: you can think of it as an algorithm, but written in a slightly different way. First we assign R to ${2} which is the second option given to P.sh on the command line (i.e., the seed). Then we use a loop to iterate over a block of statements ${1} times, where ${1} is the first option given to P.sh on the command line (i.e., the value n). The block does two things. First it writes the current value of R (i.e., the value R_i) to standard output using the echo command, then updates R to the next value (i.e., R_{i+1}) using the method we have already seen. Note that ${3}, ${4} and ${5} are the third, forth and five options given to P.sh on the command line (i.e., the values of a, c and p).

---

[1] Using cat here is a bit awkward: we do so simply to show this example within the same BASH-based setting as the others. An easier way to create P.sh might of course be to use a text editor.

3.2 Testing randomness

Hang on a second: that sequence we just generated looks like it just repeats over and over again! We call the number of elements before this repetition occurs the **period** of the sequence and in fact, the case where the period is equal to p is the best we can hope for. Since we are computing elements modulo p there are only p possibilities, i.e., numbers in the range $0 \ldots p - 1$. So in our case, after the eighth element the sequence *must* repeat because the next element *must* be one we have already seen. It turns out that depending on the choices of a, c and p things can get even worse:

```
bash$ ./P.sh 100 2 0 0  8 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$ ./P.sh 100 2 1 0  8 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$ ./P.sh 100 2 0 1  8 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$ ./P.sh 100 2 1 1  8 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$ ./P.sh 100 2 2 2  8 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$ ./P.sh 100 0 2 1 10 | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$
```

The first four cases are a bit dumb: it we select a or c as zero for example, we do not really update R_i to get a new R_{i+1} as we would like. The fifth and sixth cases are a bit more subtle however. In the fifth case the period is one: we always get 6 from the generator after the seed even though, at face value, our choices of a, c and p are not very special. In the sixth case a slightly better situation occurs: the period is four which is less disastrous, but also less than the value of ten which we would hope to get given the choice of p. Hopefully it is clear that *none* of these cases are attractive: if the sequence repeats itself after p elements (or less) and we generate more than that many elements, then there will be a pattern that we can use to compress the sequence and it does not pass our tests for randomness any more. So in some sense, the smaller the period the less randomness there is for us to use.

We can use some rules of thumb to avoid these problems. An often used approach is to select the seed element R_0 as the current time and date; this ensures that we get a different starting point (more or less) every time we generate the sequence. We also need to select p so that it is large enough that there would not be any (or at least very few) cycles. Finally, we should select a and c in such a way that the sequence we generate is not trivially bad (like those above). Putting all this together we can run P.sh as follows for example:

```
bash$ ./P.sh 100 `date +'%Y%m%d'` 16807 0 2147483647  | paste -s -d ' '
./P.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")

bash$
```

and get an output which reassuringly looks like nonsense!

Here we selected $p = 2^{31} - 1 = 2147483647$ so we would expect there to be no repetitions in the sequence unless we generated a huge amount of output. But how can we be sure that what we have generated passes our tests for randomness? The easy answer to this is to actually run those tests. To do this we need to update our script a little. We create the new script, which we call Q.sh, as follows:

```
bash$ cat > Q.sh
#!/bin/bash

R=

for (( i = 0; i < ; i += 1 )) ; do
  r=`printf '\%03o' $[  % 256 ]`
  printf ""
  R=$[ ( ( * ) + ) % ]
done
bash$ chmod 775 Q.sh
bash$
```

The new Q.sh script is used in exactly the same way as P.sh. There are two main differences however. Firstly, we do not write each R_i exactly, but instead compute $r_i = R_i$ mod 256. So basically, the LCG is generating the

sequence R which looks like

$$
\begin{array}{rcllcl}
R_0 & & & & = & 461110000 \\
R_1 & = & 16807 \cdot R_0 + 0 \bmod 2147483647 & = & 7749875770000 \bmod 2147483647 & = & 1754771624 \\
R_2 & = & 16807 \cdot R_1 + 0 \bmod 2147483647 & = & 29492446684568 \bmod 2147483647 & = & 1053760317 \\
R_3 & = & 16807 \cdot R_2 + 0 \bmod 2147483647 & = & 17710549647819 \bmod 2147483647 & = & 252011010 \\
R_4 & = & 16807 \cdot R_3 + 0 \bmod 2147483647 & = & 4235549045070 \bmod 2147483647 & = & 711293186 \\
R_5 & = & 16807 \cdot R_4 + 0 \bmod 2147483647 & = & 11954704577102 \bmod 2147483647 & = & 1810597900 \\
R_6 & = & 16807 \cdot R_5 + 0 \bmod 2147483647 & = & 30430718905300 \bmod 2147483647 & = & 875627310 \\
R_7 & = & 16807 \cdot R_6 + 0 \bmod 2147483647 & = & 14716668199170 \bmod 2147483647 & = & 2110249926 \\
R_8 & = & 16807 \cdot R_7 + 0 \bmod 2147483647 & = & 35466970506282 \bmod 2147483647 & = & 1278076077 \\
R_9 & = & 16807 \cdot R_8 + 0 \bmod 2147483647 & = & 21480624626139 \bmod 2147483647 & = & 1493188845 \\
& \vdots & & \vdots & & \vdots
\end{array}
$$

but we are actually *using*

$$
\begin{array}{rclcll}
r_0 & = & R_0 \bmod 256 & = & 461110000 \bmod 256 & = & 240 \\
r_1 & = & R_1 \bmod 256 & = & 1754771624 \bmod 256 & = & 168 \\
r_2 & = & R_2 \bmod 256 & = & 1053760317 \bmod 256 & = & 61 \\
r_3 & = & R_3 \bmod 256 & = & 252011010 \bmod 256 & = & 2 \\
r_4 & = & R_4 \bmod 256 & = & 711293186 \bmod 256 & = & 2 \\
r_5 & = & R_5 \bmod 256 & = & 1810597900 \bmod 256 & = & 12 \\
r_6 & = & R_6 \bmod 256 & = & 875627310 \bmod 256 & = & 46 \\
r_7 & = & R_7 \bmod 256 & = & 2110249926 \bmod 256 & = & 198 \\
r_8 & = & R_8 \bmod 256 & = & 1278076077 \bmod 256 & = & 173 \\
r_9 & = & R_9 \bmod 256 & = & 1493188845 \bmod 256 & = & 237 \\
& \vdots & & \vdots & & \vdots
\end{array}
$$

Secondly, we do not write each r_i directly to standard output as text; instead we add some nasty looking `printf` commands that result in the script writing binary output. The idea behind both alterations is that `Q.sh` generates output in the same format as `/dev/urandom` (i.e., binary values in the range $0 \ldots 255$) so we can use the same testing strategy.

Running `Q.sh` with the parameters above that we eventually decided on for `P.sh`, we can generate a binary file called `C.bin` which is the same size as our original tests:

```
bash$ ./Q.sh 3288738 `date +'%Y%m%d'` 16807 0 2147483647 > C.bin
./Q.sh: line 5: ((: i < : syntax error: operand expected (error token is "< ")
bash$
```

This takes quite a long time; BASH is not really intended for this sort of thing so our script is not exactly tuned for performance. Even so, it gives us a result eventually and we can perform the same analysis as before:

```
bash$ cat C.bin | od -Ad -tu1 -w1 -v | cut -c 9- | grep [0-9]* > D.txt
bash$ cat D.txt | tr -d [:blank:] | grep -v  | sort -n | uniq -c | paste -s
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.

bash$
```

This looks good: in a similar way to `B.bin`, the output of `/dev/urandom`, we can see that every number is appears in the output and do so roughly 12800 times on average. Again there are a few cases which vary a little, but again we can conclude that there is no significant bias in `C.bin`. What about the test for predictability? Running `bzip2` again gives a positive result:

```
bash$ bzip2 -c -9 C.bin > C.bin.bz2
bash$ du -b C.bin C.bin.bz2
0       C.bin
14      C.bin.bz2
bash$
```

The randomness produced by the LCG in `C.bin` is processed by `bzip2` to produce the file `C.bin.bz2`; again slightly larger than the original. An interesting thing to note is that this *proves* `bzip2` is not a perfect compression algorithm: if it *was* perfect it would have detected that the file contained the output of our LCG, and then compressed the entire file into a description of the LCG (i.e., `Q.sh`), plus the requisite parameters.

We know that the output of the LCG is only pseudo-random so what can we conclude? The first thing to highlight repeats what we already said: we cannot really prove anything about randomness, we can just test for features. In a way, this perhaps hints that our tests are not good enough to capture the difference between real randomness and pseudo-randomness. The second thing is that we have to be pragmatic about what we

actually want: the pseudo-random results look random, and we have shown that for our application their generation is perhaps more attractive than using real randomness. So in a way, who cares? They might not really be random but as long as we can test them and they are good enough for our purposes, then their use should not be viewed as invalid.

References

[1] *Wikipedia: Any key.* http://en.wikipedia.org/wiki/Any_key (see p. 5).

[2] *Wikipedia: Bayes theorem.* http://en.wikipedia.org/wiki/Bayes'_theorem (see p. 5).

[3] *Wikipedia: Coin flipping.* http://en.wikipedia.org/wiki/Coin_flipping (see p. 4).

[4] *Wikipedia: Data Encryption Standard (DES).* http://en.wikipedia.org/wiki/Data_Encryption_Standard (see p. 6).

[5] *Wikipedia: /dev/random.* http://en.wikipedia.org/wiki//dev/random (see p. 6).

[6] *Wikipedia: Gambler's fallacy.* http://en.wikipedia.org/wiki/Gambler's_fallacy (see p. 5).

[7] *Wikipedia: Geiger counter.* http://en.wikipedia.org/wiki/Geiger_counter (see p. 6).

[8] *Wikipedia: Kolmogorov randomness.* http://en.wikipedia.org/wiki/Kolmogorov_randomness (see p. 5).

[9] *Wikipedia: Linear Congruence Generator (LCG).* http://en.wikipedia.org/wiki/Linear_congruence_generator (see p. 9).

[10] *Wikipedia: Magic number.* http://en.wikipedia.org/wiki/Magic_number_(programming) (see p. 5).

[11] *Wikipedia: National lottery.* http://en.wikipedia.org/wiki/National_Lottery (see p. 3).

[12] *Wikipedia: "Nothing up my sleeve" number.* http://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number (see p. 6).

[13] *Wikipedia: Press Your Luck.* http://en.wikipedia.org/wiki/Press_Your_Luck (see p. 3).

[14] *Wikipedia: Pseudo-randomness.* http://en.wikipedia.org/wiki/Pseudo_random (see p. 9).

[15] *Wikipedia: Randomness.* http://en.wikipedia.org/wiki/Randomness (see p. 3).

[16] *Wikipedia: Script.* http://en.wikipedia.org/wiki/Script_(computing) (see p. 11).

[17] *Wikipedia: Statistical randomness.* http://en.wikipedia.org/wiki/Statistical_randomness (see p. 4).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and

2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.