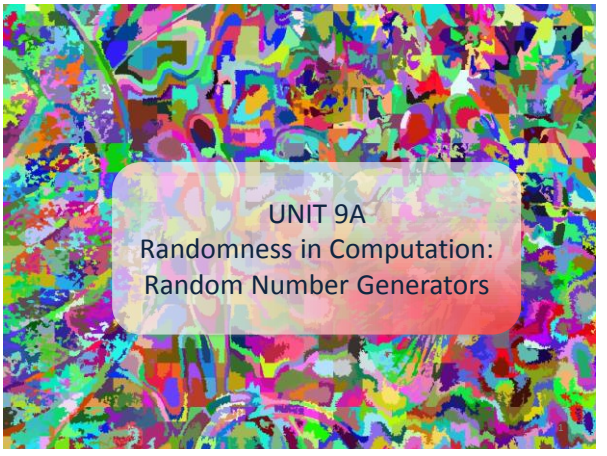


UNIT 9A
Randomness in Computation:
Random Number Generators

Last Unit

- Computer organization: what's under the hood

This Unit

- Random number generation
- Using pseudorandom numbers

Overview

- The concept of randomness
  - Let's wake up with a video about randomness https://scratch.mit.edu/projects/25843319/
- Pseudorandom number generators
  - Linear congruential generators
- Using random number generators in Python

Let's try in Excel and Python

- Generate random numbers in Excel

rand()	randbetween(200;500)
0,644126737	391
0,886918975	482
0,6176766	396
0,291156645	392
0,728014839	272
0,235209313	480
0,043148921	201
0,121162239	489
0,616295334	287
0,734229683	225
0,067019528	373

- Generate random numbers in Python
  - randint(x,y) gives us a random integer between x and y.

```
>>> from random import randint
>>> for i in range(10):
      print(randint(200,500), end=" ")

445 331 316 388 252 254 382 447 421 500
```


Randomness in Computing

- Determinism: input ⟹ **predictable output**
- Sometimes we want unpredictable outcomes
  - Games, cryptography, modeling and simulation, selecting samples from large data sets, randomized algorithms
- We use the word "randomness" for *unpredictability, having no pattern*

What is Randomness?

Tricky philosophical and mathematical question

Consider a sequence of integers. When is it "random"?

Defining Randomness

Philosophical question

- Are there any events that are really random?
- Does randomness represent lack of knowledge of the exact conditions that would lead to a certain outcome?

Some Randomness Properties

- A random sequence **should not be biased**: as length increases no element should be any more frequent than others
  an unfair coin is ***biased*** *(Long sequences will have more heads):*
  H T T **H** **H** T **H** **H** T T **H** T **H** **H** **H** T **H** …
- It's not good enough to be unbiased: consider
  0101010101010101010101010101…
  Unbiased but ***predictable***

Random sequence should be

- **Unbiased** (no "loaded dice")
- **Information-dense** (high entropy)
  - *Unpredictable sequences have high entropy.*
- **Incompressible** (no short description of what comes next)

But there are sequences with these properties that are **predictable anyway!**

Randomness is slippery!

- In summary, "**random**" to mean something like this:
  no gambling strategy is possible that allows a winner in the long run.

- **Non-randomness** can be detected and proved
- **Randomness** hard-to-impossible to prove
- Often we settle for
  "this sequence passes some tests for randomness"
  - high entropy
  - passes chi-square test
  - …
  - See example at http://www.fourmilab.ch/random/

Randomness in computing

• • •

Why do we want it? How do we get it?

Why Randomness in Computing?

- Internet gambling and state lotteries
- Simulation
  - (weather, evolution, finance [oops!], physical & biological sciences, …)
- Monte Carlo methods and randomized algorithms
  - (evaluating integrals, …)
- Cryptography
  - (secure Internet commerce, BitCoin, secret communications, …)
- Games, graphics, and many more

True Random Sequences

- Precomputed random sequences.
  - For example, *A Million Random Digits with 100,000 Normal Deviates (1955)*: A 400 page reference book by the RAND corporation
    - 2500 random digits on each page
    - Generated from random electronic pulses
- True Random Number Generators (TRNG)
  - Extract randomness from physical phenomena
    such as atmospheric noise, times for radioactive decay
- Drawbacks:
  - Physical process might be biased
    (produce some values more frequently)
  - Expensive
  - Slow

Pseudorandom Sequences

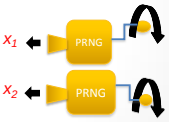
- *Pseudorandom number generator* (PRNG):
  algorithm that produces a sequence that looks random
  (i.e. passes some randomness tests)

- The sequence cannot be really random!
  - because an algorithm produces known output, by definition

(Pseudo) Random Number Generator

- A (software) machine to produce sequence $x_1, x_2, x_3, x_4, x_5, \ldots$ from x_0
- Initialize / **seed:**

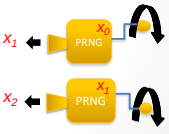
- Get pseudorandom numbers:



(Pseudo) Random Number Generator

- A (software) machine to produce sequence $x_1, x_2, x_3, x_4, x_5, \ldots$ from x_0
- Initialize / **seed:**

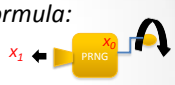
- Get pseudorandom numbers: (f is a function that computes a number):



- Idea: **internal state determines the next number**

Simple PRNGs

- *Linear congruential generator formula:*
  $x_{i+1} = (a\ x_i + c)\ \%\ m$



  - In the formula **a**, **c**, and **m** are **constants**
  - Good enough for many purposes
  - **…if a**, **c**, and **m** are properly chosen

Example

Linear Congruential Generator (LCG)

```
# global internal state/ seed
currentX = 0
```

```
# seed the generator
def prng_seed(s) :
    global currentX
    currentX = s
```


```
# LCG (a=1, c=7, m=12)
def prng1(n):
    return (n + 7) % 12
```

```
# state updater
def prng():
    global currentX
    currentX = prng1(currentX)
    return currentX
```

First 12 numbers: 1, 8, 3, 10, 5, 0, 7, 2, 9, 4, 11, 6

Does this look random to you?

Example LCG : Outputs

```
for i in range(12):
   print(prng(), end=" ")

7 2 9 4 11 6 1 8 3 10 5 0

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

prng_seed(6)
for i in range(12):
   print(prng(), end=" ")

1 8 3 10 5 0 7 2 9 4 11 6
```

```
prng_seed(6)
for i in range(20):
   print(prng(), end=" ")

1 8 3 10 5 0 7 2 9 4 11 6
1 8 3 10 5 0 7 2
```


Example LCG

- First 20 numbers:

5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10,
5, 0, 7, 2, 9, 4, 11, 6, ?

Random-looking?

- What do you think the next number in the sequence is?
- **Moral: just eyeballing the sequence not a good test of randomness!**
- This generator has a **period** that is too short: it repeats too soon.

(What else do you notice if you look at it for a while?)

Another PRNG

```
def prng2(n):
  return (n + 8) % 12     # a=1, c=8, m=12
```

```
>>> prng_seed(6)
>>> for i in range(20):
        print(prng(), end=" ")

[8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4, 0]
```

Random-looking?

Moral: choice of a, c, and m crucial!

PRNG Period

Let' s define the PRNG *period* as the number of values in the sequence before it repeats.

5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10,
5, 0, 7, 2, 9, 4, 11, 6, …

prng1, period = 12 next number = (last number + **7**) mod 12

8, 4, 0, 8, 4, 0, 8, 4, 0, 8, …

prng2, period = 3 next number = (last number + **8**) mod 12

We want the longest period we can get!

Picking the constants a, c, m

- Large value for *m*, and appropriate values for *a* and *c* that work with this *m*
  → a very long sequence before numbers begin to repeat.
- Maximum period is *m*

Picking the constants a, c, m

- The LCG will have a period of *m* (the maximum) if and only if:
  - *c* and *m* are ***relatively prime*** (i.e. the only positive integer that divides both *c* and *m* is 1)
  - *a*-1 is **divisible by all prime factors** of *m*
  - if *m* is a **multiple of 4** , then *a*-1 is also a multiple of 4
- (*Number theory* tells us so)

If c and m are not relatively prime, then c = nk and m = sk for some k.	→ *After s/n cycles you come back to the seed*

29

Picking the constants a, c, m

(1) *c* and *m* relatively prime

(2) a-1 divisible by all prime factors of m

(3) if m is a multiple of 4, so is a-1

- **Example for prng1** (*a* = 1, *c* = 7, *m* = 12)
  - **Factors of 7**: 1, 7 **Factors of 12**: 1, 2, 3, 4, 6, 12
  - **0** is divisible by all prime factors of 12 → true
  - if **12** is a multiple of 4 , then **0** is also a multiple of 4 → true
- prng1 will have a period of 12

30

Exercise for you

(1) *c* and *m* relatively prime

(2) a-1 divisible by all prime factors of m

(3) if m is a multiple of 4, so is a-1

$x_{i+1} = (5x_i + 3)$ modulo 8

$x_0 = 4$ $a = 5$ $c = 3$ $m = 8$

- What is the period of this generator? Why?
- Compute x_1, x_2, x_3 for this LCG formula.

31

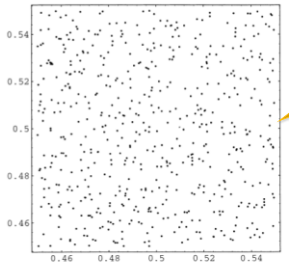
Some pitfalls of PRNGs

- **Predictable seed**.
  Example: Famous Netscape security flaw caused by using system time.
- **Repeated seed** when running many applications at the same time.
- **Hidden correlations**
- High quality but **too slow**

33

Finding hidden correlations

P. Hellekalek / Mathematics and Computers in Simulation 46 (1998) 485–505



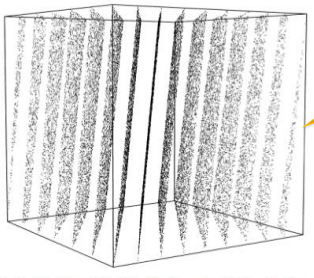
Looking good!

Fig. 1. LCG(2^{31}, 65539, 0, 1) Dimension 2: Zoom into the unit interval.

34

Finding hidden correlations

P. Hellekalek / Mathematics and Computers in Simulation 46 (1998) 485–505



OOPS!

Fig. 2. LCG(2^{31}, 65539, 0, 1) Dimension 3: The 15 planes.

35

Advice from an expert

Get expert advice ☺

A good generator is not so easy to find if one sets out to design it by oneself, without the necessary mathematical background. On the other hand, with the references and links we supply, a good random number generator designed by experts is relatively easy to find.

– P. Hellekalek

Using random number generators in Python

Random integers in Python

- To generate random integers in Python, we can use the `randint` function from the `random` module.
- `randint(a,b)` returns an integer n such that $a \leq n \leq b$ (note that it's **inclusive**)

```
>>> from random import randint
>>> randint(0,15110)
12838
>>> randint(0,15110)
5920
>>> randint(0,15110)
12723
```

List Comprehensions

One output from a random number generator not so interesting when we are trying to see how it behaves

```
>>> randint(0, 99)
    42
```

Easily get a list of outputs

```
>>> [ randint(0,99) for i in range(10) ]
[5, 94, 28, 95, 34, 49, 27, 28, 65, 65]

>>> [ randint(0,99) for i in range(5) ]
[69, 51, 8, 57, 12]

>>> [ randint(101, 200) for i in range(5) ]
[127, 167, 173, 106, 115]
```


Some functions from the `random` module

```
>>> [ random() for i in range(5) ]
[0.05325137538696989, 0.9139978582604943,
0.6142995105641187, 0.3223156290220041,
0.8198417602039083]

>>> [ uniform(1,10) for i in range(5) ]
[4.777545709914872, 1.896613966653442,
8.334224863883207, 3.006025360903046,
8.968660414083441]

>>> [ randrange(10) for i in range(5) ]
[8, 7, 9, 4, 0]

>>> [randrange(0, 101, 2) for i in range(5)]
[76, 14, 44, 24, 54]
```

```
>>> colors = ['red', 'blue','green',
              'gray', 'black']

>>> [ choice(colors) for i in range(5) ]
['gray', 'green', 'blue', 'red', 'black']

>>> [ choice(colors) for i in range(5) ]
['red', 'blue', 'green', 'blue', 'green']

>>> sample(colors, 2)
['gray', 'red']

>>> [ sample(colors, 2) for i in range(3)]
[['gray', 'red'], ['blue', 'green'],
['blue', 'black']]

>>> shuffle(colors)
>>> colors
['red', 'gray', 'black', 'blue', 'green']
```


Adjusting Range

- Suppose we have a LCG with period *n* (*n* is very large)
- … but we want to play a game involving dice (each side of a die has a number of spots from 1 to 6)
- How do we take an integer between 0 and *n*, and obtain an integer between 1 and 6?
  - Forget about our LCG and use `randint(?,?)`
  - *Great, but how did they do that?*

what values should we use?

Adjusting Range

- Specifically: our LCG is the Linear Congruential Generator of glib (period = 2^{31} = 2147483648)
- We call `prng()` and get numbers like
  1533190675, 605224016, 450231881, 1443738446, …
- We define:

```
def roll_die():
    roll = prng() % 6 + 1
    assert 1 <= roll and roll <= 6
    return roll
```

- What's the smallest possible value for **`prng() % 6`** ?
- The largest possible?

46

Random range

- Instead of rolling dice, we want to pick a random (US) presidential election year between 1788 and 2012
  - election years always divisible by 4
- We still have the same LCG with period 2147483648. What do we do?
  - Forget about our LCG and use `randrange(1788, 2013, 4)`
  - *Great, but how did they do that?*

47

Random range

- Remember, `prng()` gives numbers like
  1533190675, 605224016, 450231881, 1443738446, …

```
def election_year() :
    year = ?
    assert 1788 <= year and year <= 2012 and year % 4 == 0
    return year
```

48

Random range

- First: think *how many numbers are there in the range we want? That is, how many elections from 1788 to 2012?*
  - 2012 – 1788? No!
  - (2012 – 1788) / 4? Not quite! (there's one extra)
  - (2012 – 1788) / 4 + 1 = 57 elections

→ So let's randomly generate a number from 0 to 56 inclusive:

```
def election_year() :
    election_number = prng() % ((2012 - 1788) // 4  + 1)
    assert 0 <= election_number and election_number <= 56
    year = ?
    assert 1788 <= year and year <= 2012 and year % 4 == 0
    return year
```


Random range

- Okay, but now we have random integers from 0 through 56
  - good, since there have been 57 elections
  - bad, since we want years, not election numbers 0 … 56

```
def election_year() :
    election_number = prng() % ((2012 - 1788) // 4  + 1)
    assert 0 <= election_number and election_number <= 56
    year = election_number * 4 + 1788
    assert 1788 <= year and year <= 2012 and year % 4 == 0
    return year
```


Random range

Sample output:

```
>>> [ election_year() for i in range(10) ]
[1976, 1912, 1796, 1800, 1984, 1852, 1976, 1804, 1992, 1972]
```

The same reasoning will work for a random sampling of any arithmetic series.
(Just think of the series and let the random number generator take care of the randomness!)

- How many different numbers in the series?
  If there are k, randomly generate a number from 0 to k.
- Are the numbers separated by a constant (like the 4 years between elections)?
  If so, multiply by that constant.
- What's the smallest number in the series?
  Add it to the number you just generated.

Next Time

Using pseudorandom numbers