Randomness in Computing

• Determinism -- in all algorithms and programs we have seen so far, given an input and a sequence of steps, we get a unique answer. The result is predictable.
• However, some computations need steps that have unpredictable outcomes
  – Games, cryptography, modeling and simulation, selecting samples from large data sets
• We use the word “randomness” for unpredictability, having no pattern
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?

Obtaining Random Sequences

• Definition we adopt: A sequence is random if, for any value in the sequence, the next value in the sequence is totally independent of the current value.
• If we need random values in a computation, how can we obtain them?
Obtaining Random Sequences

- Precomputed random sequences. For example, *A Million Random Digits with 100,00 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

- Pseudo-random Number Generators (PRNG)
  - Use a formula to generate numbers in a deterministic way but the numbers appear to be random

Random numbers in Python

- To generate random numbers in Python, we can use the `randint` function from the `random` module.
- The `randint(a, b)` returns an integer `n` such that `$a \leq n \leq b$.

```python
>>> from random import randint
>>> randint(0,15110)
12838
>>> randint(0,15110)
5920
>>> randint(0,15110)
12723
```
Is `randint` truly random?

- The function `randint` uses some algorithm to determine the next integer to return.
- If we knew what the algorithm was, then the numbers generated would not be truly random.
- We call `randint` a **pseudo-random number generator** (PRNG) since it generates numbers that appear random but are not truly random.

Creating a PRNG

- Consider a pseudo-random number generator `prng1` that takes an argument specifying the length of a random number sequence and returns a list with that many “random” numbers.
  ```python
  >>> prng1(9)
  [0, 7, 2, 9, 4, 11, 6, 1, 8]
  ```
- Does this sequence look random to you?
Creating a PRNG

• Let’s run `prng1` again:
  `>>> prng1(15)`
  `[0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5, 0, 7, 2]`

• Now does this sequence look random to you?
• What do you think the 16th number in the sequence is?

Looking at `prng1`

```python
def prng1(n):
    seq = [0]  # seed (starting value)
    for i in range(1, n):
        seq.append((seq[-1] + 7) % 12)
    return seq

>>> prng1(15)
[0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5, 0, 7, 2]
```
Another PRNG

def prng2(n):
    seq = [0]  # seed (starting value)
    for i in range(1, n):
        seq.append((seq[-1] + 8) % 12)
    return seq

>>> prng2(15)
[0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4]

• Does this sequence appear random to you?

PRNG Period

• Let’s define the PRNG period as the number of values in a pseudo-random number generator sequence before the sequence repeats.

  \[ [0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5, 0, 7, 2] \]
  \[ \text{period} = 12 \]
  next number = (last number + 7) mod 12

  \[ [0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4] \]
  \[ \text{period} = 3 \]
  next number = (last number + 8) mod 12
Linear Congruential Generator (LCG)

- A more general version of the PRNG used in these examples is called a linear congruential generator.
- Given the current value $x_i$ of PRNG using the linear congruential generator method, we can compute the next value in the sequence, $x_{i+1}$, using the formula $x_{i+1} = (a \cdot x_i + c) \mod m$ where $a$, $c$, and $m$ are predetermined constants.

- **prng1**: $a = 1, c = 7, m = 12$
- **prng2**: $a = 1, c = 8, m = 12$

Picking the constants $a$, $c$, $m$

- If we choose a large value for $m$, and appropriate values for $a$ and $c$ that work with this $m$, then we can generate a very long sequence before numbers begin to repeat.
  - Ideally, we could generate a sequence with a maximum period of $m$. 
Picking the constants $a, c, m$

- The LCG will have a period of $m$ for all seed values 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

- Example: prng1 ($a = 1, c = 7, m = 12$)
  - Factors of $c$: 1, 7
  - Factors of $m$: 1, 2, 3, 4, 6, 12
  - 0 is divisible by all prime factors of 12 \(\rightarrow\) true
  - if 12 is a multiple of 4, then 0 is also a multiple of 4 \(\rightarrow\) true

Example

\[
x_{i+1} = (a \times x_i + c) \mod m
\]

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

- Compute $x_1, x_2, ...$, for this LCG formula.

- What is the period of this generator?

  - If the period is maximum, does it satisfy the three properties for maximal LCM?
LCMs in the Real World

• glibc (used by the c compiler gcc):
  a = 1103515245, c = 12345, m = $2^{32}$

• Numerical Recipes (popular book on numerical methods and analysis):
  a = 1664525, c = 1013904223, m = $2^{32}$

• Random class in Java:
  a = 25214903917, c = 11, m = $2^{48}$

Using PythonLabs for Random Numbers

```python
>>> from PythonLabs.RandomLab import *
>>> p = PRNG(1, 7, 12)
>>> p
<PythonLabs.RandomLab.PRNG a: 1 c: 7 m: 12>
```

A seed is a number used to initialize a pseudorandom number generator. Its choice is critical in some applications.
Seeding a PRNG

```python
>>> from PythonLabs.RandomLab import *
>>> from time import time
>>> p = PRNG(1, 7, 12)
>> p.seed(int(time()))
1382377699
>>> p.advance()
2
>>> p.advance()
9
>> p.state()
9
```

You can use integer part of the current system time to seed a pseudorandom number generator

Python’s random module

• Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of $2^{19937}-1$.

• Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range $[0.0, 1.0)$. Source: http://docs.python.org
Some Python functions from the `random` module

```python
>>> random.random()    # random float 0.0 <= x < 1.0
0.9607807406878415
>>> random.uniform(1,10)  # random float 1.0 <= x < 10.0
5.4645226971373555
>>> random.randrange(10)  # random int 0 <= x < 9
7
>>> random.randrange(0,101,2)  # random even int 0 <= x < 101
42
>>> random.choice("abcdefghij")  # random char from string 'c'
>'c'<
>>> items = [1,2,3,4,5,6]
>>> random.shuffle(items)
[3, 2, 5, 6, 4, 1]
>>> random.sample([1,2,3,4,5,6], 3)  # 3 samples without replacement
[4, 1, 5]
```