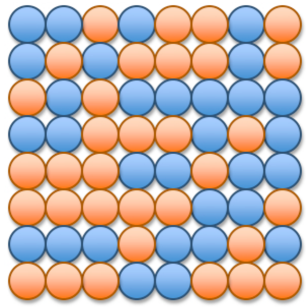

# UNIT 9A

## Randomness in Computation: Random Number Generators



# 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?



## 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

- Pre-computed 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 $a \leq n \leq b$.

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

**CAUTION: This function includes both endpoints! It is not like the `range` function!**

# 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.

```
>>> 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**

```
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]`

  period = 12

  next number = (last number + 7) mod 12

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

  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 $x_{i+1} = (a\ x_i + c)$ modulo $m$ where a, c, and m are pre-determined constants.
  - **`prng1`**: a = 1, c = 7, m = 12
  - **`prng2`**: a = 1, c = 8, m = 12

*There are rules on choosing values for a, c, and m to guarantee a maximum period for the random number generator.*



# 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 = $2^{48}$

## 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 additional Python functions from the **`random`** module

```
>>> import random
```

```
>>> random.random()
0.9607807406878415
```

random float
$0.0 \le x < 1.0$

```
>>> random.uniform(1,10)
5.4645226971373555
```

random float
$1.0 \le x < 10.0$

## Some Python functions from the **random** module (cont'd)

```
>>> import random

>>> random.randrange(10)
7
```

random int
$0 \le x < 10$

```
>>> random.randrange(0,101,2)
42
```

random even int
$0 \le x < 101$



## Some Python functions from the **random** module (cont'd)

```
>>> import random
>>> random.choice("abcdefghij")
'c'
```

random char from string

```
>>> items = [1,2,3,4,5,6]
>>> random.shuffle(items)
[3, 2, 5, 6, 4, 1]
```

randomly shuffled list

```
>>> random.sample(items, 3)
[4, 1, 5]
```

list of random samples without replacement