

UNIT 9A

Randomness in Computation: Random Number Generators

Course Announcements

- We are in the process of setting up the tutoring help system.
- PS7 is due Wednesday 3/20 in class
- Midterm 2 (written) is Wed March 27

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 Ruby

- To generate random numbers in Ruby, we can use the **rand** function.
- The **rand** function take a positive integer argument (n) and returns an integer between 0 and n-1.

```
>> rand(15110)
```

```
=> 1239
```

```
>> rand(15110)
```

```
=> 7320
```

```
>> rand(15110)
```

```
=> 84
```

Is **rand** truly random?

- The function **rand** 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 **rand** 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 an array with that many “random” numbers.

```
>> prng1(9)
```

```
=> [0, 7, 2, 9, 4, 11, 6, 1, 8]
```

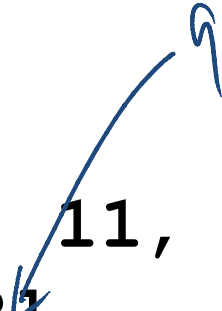
- Does this sequence look random to you?

Creating a PRNG

- Let's run `prng1` again:

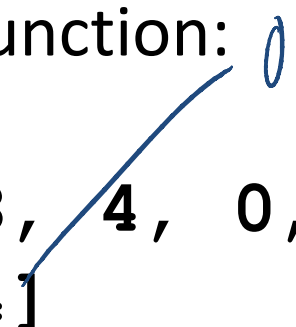
```
>> prng1(15)
```

```
=> [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?

Another PRNG

- Let's try another PRNG function:
=> `prng2(15)`
>> [0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4, 0, 8, 4]

- Does this sequence appear random to you?
- What do you think is the 16th number in this sequence?

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


Looking at prng1

```
def prng1(n)
  seq = [0]           ; seed (starting value)
  for i in 1..n-1 do
    seq << (seq.last + 7) % 12
  end
  return seq
end
```

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

Looking at `prng2`

```
def prng2(n)
  seq = [0]           ; seed (starting value)
  for i in 1..n-1 do
    seq << (seq.last + 8) % 12
  end
  return seq
end
```



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

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 $x_{i+1} = (a x_i + c) \text{ modulo } m$ where a , c , and m are pre-determined constants.

– **prng1:** $a = 1, c = 7, m = 12$

– **prng2:** $a = 1, c = 8, m = 12$

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

- **Theorem:** 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

$$x_{i+1} = (a x_i + c) \text{ modulo } m$$

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

- Compute x_1, x_2, \dots , for this LCG formula.
- What is the period of this formula?
 - 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:
 $a = 25214903917, c = 11, m = 2^{48}$
- The PRNG built into Ruby has a period of 2^{19937} .

Rest of the Week

- Uses of PRNG in games
- Cellular automata and pseudorandomness