UNIT 9B

Randomness in Computation: Games with Random Numbers

Announcements

- Tutoring service by 15110 CAs starting tomorrow
  - Ask your CA about it in the recitations
- Written exam 2 next Wednesday (March 27)
- Exam review on Sunday (March 24)
  - Two sessions: 6-8 and 8-10
  - Location GHC 4303

Last Lecture: Pseudorandom Number Generation

- Linear Congruential Generators
  - 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
- 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 with a maximum period of m.
- To generate random numbers in Ruby we can use rand(n), which generates a random number between 0 and n-1

This Lecture

- Work at a higher level of abstraction to implement games
  - Use a function that is implemented by a pseudorandom number generator
- RubyLabs RandomLab module

Simulating a Die

- Suppose we want to have a random number between 1 and 6, without using the rand function.
- Algorithm: Range of Number:
  - Generate a pseudo random number using a PRNG with a very large m. [0, m-1]
  - Take the result from the previous step and modulo by 6. [0, 5]
  - Add 1 to the result from the previous step. [1, 6]

Using RubyLabs

```
>> include RandomLab
=> Object
>> p = PRNG.new(81, 337, 1000)
⇒ #<RandomLab::PRNG a: 81 c: 337 m: 1000>
>> seq = []
=> []
>> rolls = []
=> []
>> 10.times { num = p.advance; seq << num; rolls << (num % 6 + 1) }
⇒ 10
>> seq
=> [337, 634, 691, 308, 285, 422, 519, 376, 793, 570]
>> rolls
=> [2, 5, 2, 3, 4, 3, 4, 5, 2, 1]
```

generates the next number in sequence on demand

337 mod 6 = 1
634 mod 6 = 4
...

Seeding a PRNG

- If we run the same code again, we will get the same sequence since we're seeding with the same integer each time.
- To generate a new seed each time:

```
>> p = PRNG.new(81, 337, 1000)
=> #<RandomLab::PRNG a: 81 c: 337 m: 1000>
>> p.seed(Time.now.to_i % 1000)
=> 574
```

- When would you want to start with the same seed each time?

Think of debugging programs, replicating the results for simulations etc.

Testing for Randomness

- Visualize and check if we can observe any patterns
- Prove theorems
- ...

Visualizing with a Histogram

```
>> include RandomLab
=> Object
>> p = PRNG.new(81, 337, 1000)
⇒ #<RandomLab::PRNG a: 81 c: 337 m: 1000>
>> p.seed(Time.now.to_i)

>> view_histogram([1,2,3,4,5,6])
=> true
>> 1000.times { num = p.advance % 6 + 1; update_bin(num) }
⇒ 1000
```

Every outcome for a die roll looks equally likely according to the histogram.

Experiment with PRNG using a dot plot as in the text book.

Simulating a Deck of Cards

- A deck of cards is made up of 52 cards, where each card has a suit and a rank:
  - Suits: Spades (♠), Hearts (♥), Diamonds (♦), Clubs (♣)
  - Ranks: 2, 3, 4, 5, 6, 7, 8, 9, 10, J (Jack), Q (Queen), K (King), A (Ace)

- A standard deck of cards has 1 of each combination of suit and rank.

Cards in RubyLabs

- RubyLabs has an object called a Card that represents a standard playing card.

```
>> include RandomLab
=> Object
>> c = Card.new
=> KS
>> c = Card.new
=> 10C
>> c = Card.new
=> 9H
```

Use of cards also requires us to include RandomLab.

Cards in RubyLabs (cont'd)

- We can determine the rank or suit of a card:

```
>> c = Card.new
=> 2S
>> c.rank
=> :two
>> c.suit
=> :spades
```

The values for rank and suit are special constants that start with a colon (e.g. **`:king`**, **`:spades`**). These are not strings (no quotes).

Cards in RubyLabs (cont'd)

- We can get a specific card using an index to the **new** function:

```
>> c = Card.new(4)
=> 10S
```

Cards are indexed as follows:

A♠ K♠ … 2♠ A♥ K♥… 2♥ A♦ K♦ … 2♦ A♣ K♣ … 2♣

0 1 12 13 14 25 26 27 38 39 40 51

Deck of Cards

- We can create a deck of cards also!

```
>> d = new_deck
=> [AS, KS, QS, JS, 10S, 9S, 8S, 7S,
6S, 5S, 4S, 3S, 2S, AH, KH, QH, JH,
10H, 9H, 8H, 7H, 6H, 5H, 4H, 3H, 2H,
AD, KD, QD, JD, 10D, 9D, 8D, 7D, 6D,
5D, 4D, 3D, 2D, AC, KC, QC, JC, 10C,
9C, 8C, 7C, 6C, 5C, 4C, 3C, 2C]
```

  - Note that the cards are in the same order as the indexes given in the previous slide.

Dealing Random Cards

- Suppose we have a card game like Poker where we want to be dealt a "hand" of 5 random cards from the deck.
- What is wrong with the following code?

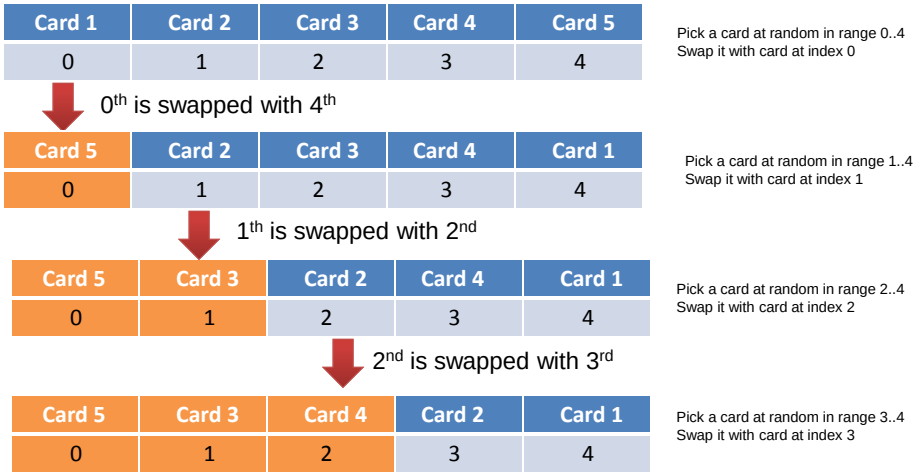
```
hand = []
5.times { hand << Card.new }
```

What if Card.new returns the same card?
We want the 5 cards to be different.

Shuffling the Deck

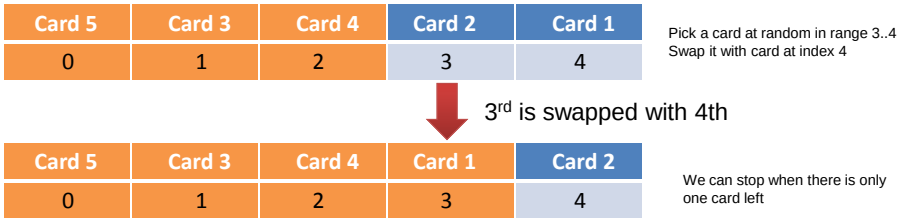
- We should shuffle a deck and then create a hand from the first 5 cards in the deck.
- There are many ways to shuffle a deck of cards.
- One algorithm:
  - Exchange (swap) the first card with a random card.
  - Exchange the second card with a random card except the first card.
  - Exchange the third card with a random card except the first two cards.
  - ... Repeat until all cards have been swapped.

Shuffling Example



17

Shuffling Example (cont'd)



Depending on the random selection a card may be swapped multiple times and not be swapped at all.

18

Building the Function

- For the first card (at index 0) in deck **d**, how do we generate a random index for a card to swap?

```
r = rand(d.length)
```

- How do we swap the first card with the randomly-selected card?

```
temp = d[0]
d[0] = d[r]
d[r] = temp
```

or we can use ***parallel assignment*** in Ruby...

```
d[0], d[r] = d[r], d[0]
```


Building the Function (cont'd)

- For the second card (at index 1) in deck **d**, how do we generate a random index for any card except the first card?

```
r = rand(d.length-1) + 1
```

- How do we swap the first card with the randomly-selected card?

```
temp = d[1]
d[1] = d[r]
d[r] = temp
```

or we can use ***parallel assignment*** in Ruby...

```
d[1], d[r] = d[r], d[1]
```

Building the Function (cont'd)

- For the third card (at index 2) in deck **d**, how do we generate a random index for any card except the first two cards?

```
r = rand(d.length-2) + 2
```

- How do we swap the first card with the randomly-selected card?

```
temp = d[2]
d[2] = d[r]
d[r] = temp
```

or we can use ***parallel assignment*** in Ruby...

```
d[2], d[r] = d[r], d[2]
```


In general...

- For the card at index **i** in deck **d**, how do we generate a random index for a card to swap?

```
r = rand(d.length-i) + i
```

- How do we swap the first card with the randomly-selected card?

```
temp = d[i]
d[i] = d[r]
d[r] = temp
```

or we can use ***parallel assignment*** in Ruby...

```
d[i], d[r] = d[r], d[i]
```

Shuffling the entire deck and dealing five cards...

```
def permute!(d)
   for i in 0..d.length-2 do
       r = rand(d.length-i) + i
       d[i], d[r] = d[r], d[i]
   end
   return d
end
```

```
>> hand = permute!(new_deck).first(5)
=> [3H, AD, 3S, 3D, 7D]
```


Poker: Detecting a Flush

- In poker, a flush is a hand where all of the cards have the same suit.
- One possible algorithm:
  If all of the cards have a suit of spades, return true.
  If all of the cards have a suit of hearts, return true.
  If all of the cards have a suit of diamonds, return true.
  If all of the cards have a suit of clubs, return true.
  If none of the above tests returns true, return false.

Poker: Detecting a Flush (cont'd)

```
def all_spades?(hand)
  for i in 0..hand.length-1 do
     return false if hand[i].suit != :spades
  end
  return true
end
```

all_hearts?, **all_diamonds?** and **all_clubs?** are written similarly.

Poker: Detecting a Flush (cont'd)

```
def flush?(hand)
  return true if all_spades?(hand)
  return true if all_hearts?(hand)
  return true if all_diamonds?(hand)
  return true if all_clubs?(hand)
  return false
end
```

Simple dice game

- A player has two die. On each roll, if the player does not roll "doubles" (same value on each die), then the player wins the sum of the die values. Otherwise, the player earns a "strike". The game ends once the player has three strikes.
- Write a function that returns the amount the player wins in a simulated simple dice game.

Rolling a die

```
def roll()
  return rand(6) + 1
end
```

One round of the game

```
die1 = roll()
die2 = roll()
if die1 == die2 then
  strikes = strikes + 1
else
  sum = sum + die1 + die2
end
```


Putting it together

```
def simple_game()
    strikes = 0
    sum = 0
    while (strikes < 3) do
         die1 = roll()
         die2 = roll()
         if die1 == die2 then
              strikes = strikes + 1
         else
              sum = sum + die1 + die2
         end
    end
    return sum
end
```

What is the average winnings for 1000 players of this game?

```
>> games = []
=> []
>> 1000.times { games << simple_game() }
=> 1000
>> games
=> [61, 86, 127, 140, ... , 114, 292]
>> total = 0
=> 0
>> games.each { |score| total += score }
=> [61, 86, 127, 140, ... , 114, 292]
>> total/1000.0
=> 106.731
```