No Clockwork Universe

- Stuff doesn't always happen the same even when conditions seem pretty identical.
- The idea that we could abstract this is really pretty new.
  - Newton: Clockwork universe
  - Fermat, Pascal: Dice calculations
  - Kolmogorov: Axiomatization of probability
A Deterministic Machine

- A computer is designed to move from state to state deterministically.
  - Failure to do so is usually treated as a bug.
  - Still useful for analyzing randomness.

Questions:
- What do we mean by “random”?
- Can we make computers be random?
- Can we harvest random?
- Once we have random, what do we do with it?
Let's Play Poker

• Suppose I want to build a web-based poker server (like everyone else in the world).

• My server needs to shuffle and deal hands.
  – I need to ensure fair shuffling, or I may be in big trouble.
  – Players need to be sure I’ve shuffled fairly.

• Traditional shuffling would be acceptable, but simulating it online is hard
  – It also probably requires randomness
Randomness As Chance

• Randomness seems kind of about chance and probability. We say that a particular value is “uniform random” if it is chosen with equal likelihood from all possible values.
  
  – There’s no such thing as a “random integer”. But there is a “random real in a range”.

• This definition is unsatisfying. It is untestable.

• It can be improved: think random sequences.
Random Sequences

- As 2s 3s 4s 5s 6s : likely not random (?)
- As 2h 3s 4h 5s 6h : still likely not random
- 4s 7s 9s Ks 3s 2s : "more random"
- 4s 7h 9h Kc 3d 2h : looks fairly random

- Information Theory says that random sequences contain maximal information.
  - Kolmogorov: The description of the sequence is as big as the sequence itself.
Randomness as Uncorrelation

- Another way to think of randomness is as a lack of correlation with other sequences.
- This is a good definition for us: we want our poker hands to be uncorrelated with anything the players can imagine.
- Now what we have to do is figure out how to do this.
Shuffling as Permutation

• First, though, we need to nail down what we mean by a “fair shuffle”.
  
  − A fair shuffle will choose any possible permutation of the 52 cards with equal likelihood. \(52! \approx 2^{226} \approx 10^{68}\) choices

• Let’s assume we have a machine that can generate random integers in the range \(1..n\) for any given \(n\).

• How do we pick a random permutation?
Problem: Shuffling & Arrays

• The obvious way to represent cards in our program is as integers in the range 1..52.
• Then we can represent our deck as an array containing a permutation of cards.
• We start with a "new deck".
• We want to rearrange the cards such that all permutations are equally likely.
  - Simulate human shuffle: Maybe, but maybe not.
Captain Obvious’s Fail Shuffle

- for i in 1..52
  j ← random integer in 1..52
  swap a[i] with a[j]

- For subtle reason, doesn’t work
  - Elements near the front likely get swapped more times than elements near the back.
  - Hard to see by inspection of “shuffles”.

- Swapping random pairs doesn’t work either.
  - Don’t know how many swaps are needed.
Lame Shuffle-Sort

- What if we attach a random number in the range 1..1000 to each array element as a key and then sort the array?
  - Slow because sort: $O(n \lg n)$ time.
  - What about key ties? We would need to retry!
  - Turns out that the bigger the random number, the more expensive our "generator" will be.

- Works for some value of "work".
Selection Shuffle

- Repeatedly pick a random card out of the deck and stack it on top of the new deck.

- Requires `randint(52)`, then `randint(51)`, ...

- Requires counting down to the selected card in our source array, then marking it as used.
  - Can’t dodge “holes” otherwise.
  - So $O(n^2)$, not fast at all
  - “Obviously” generates all permutations equally.

- Requires extra destination array.
Selection Shuffle

[Diagram of a selection shuffle process with blue and red squares and an arrow indicating the shuffle operation.]
Selection Shuffle: Hole Plugging

- Can fix performance problem by "hole plugging": always move last card to fill the hole made by removing a card.
  - OK because we only select randomly from the source anyhow.

- Now the algorithm touches each card at most twice, which is probably as good as we can do.
Hole Plugging
Selection Shuffle: In-place

- Now each time we select a card, we will leave a hole just below the remaining source.
- Plug the selected card into the hole.
- This achieves in-place sorting.
In-Place
Knuth-Morris-Pratt/etc. Shuffle

• for j in 1..52
  k ← random integer in j..52
  exchange a[j] with a[k]

• Correct, fast, uses little memory
  – Many easy small bugs, so be careful.
  – Many little performance improvements.

• Now all we need is a way to generate random integers in a range, and we’re set.
Pseudo-Random Numbers

- Remember that computers are deterministic.
- Maybe we can come up with some way of generating an unpredictable / uncorrelated sequence of random numbers in range $1..m$?
- Can then use the remainder operation to (almost) get the numbers to the range $1..n$ when $n$ is much smaller than $m$.
- These numbers are still not random: Knowing the algorithm in full makes them predictable.
Linear Congruential PRNG (MLCG)

- $s \leftarrow$ some "seed" value in $1..m$
- $r \leftarrow s \text{ rem } n$
  $s \leftarrow (s \times a) \text{ rem } m$
- Choice of $a$ and $m$ matters a lot.
- Note that $s$ may never be 0.
- Sample output ($a=55$, $m=251$, $s=12$, $n=15$):
  8, 6, 1, 5, 6, 9, 4, 12, 10, 0, 12
- Coefficients from paper:
MLCG As A Wheel

Diagram of a wheel with numbers 0 to 9.
Implement Me

- for j in 1..52
  k ← random integer in j..52
  exchange a[j] with a[k]

- r ← s rem n
  s ← (s × a) rem m
MLCG Fail

- Still have to pick a “random” seed.
  - Often use “time of day in ms”, but...
- Still have remainder error.
- Turns out that these generators can be easily reversed: Can discover \( s, a \) and \( p \) from very short sequences even given small \( n \).
  - But note that they didn’t do it right.
Cryptographic-Secure PRNG

- Has the property that hidden $s$ may be very difficult to discover from output sequence.
- Generally pretty expensive per random.
- Interesting relationship between $m$ and ability to generate all possible poker hands.
  - $m$ must be very large—hundreds of bits.
- Still have to choose initial $s$. 
Terrible Ways To Choose

- System clock: predictable.
- PID: small and predictable.
- Hash of memory: surprisingly predictable.
- User timing: Under user control.
Better Ways To Choose s

- True or likely “entropy” sources.
- Hardware RNG:
  - Pick a phenomenon (e.g. thermal noise) that is truly random.
  - Use measurements of that phenomenon.
  - Problems: slow / colored / expensive / fiddly.
  - We’re working on it. :-)


Other Uses Of Randomness

- Different distributions
- Selection and sampling:
  - Efficient selection tricks
- Modeling
- Cryptography
Adversary Games and Randomness

- Consider Rock-Scissors-Paper.
  - "Poor Bart. He always picks rock."
- John Nash: Sometimes the best strategy is a "mixed strategy"; a random choice of moves.
- In poker, for example, "never bluff" and "always bluff" are clearly both fail.
  - What is the best bluffing strategy?
Telephone Poker

• How can players ensure our server is shuffling and dealing randomly?
  - (In real life, they almost always aren't.)

• Idea: Arrange things in such a way that nobody has to be trusted.
  - Each side forces randomness on the other.
  - Each side can check at the end.
  - Typically relies on public-key cryptography. (c.f. RSA, GM, etc. in early 1980s)
Paying Attention To Random

- Bottom line? Random pops up all over the place, and is incredibly important.
  - I could easily do a 10-week course.

- It is easy to get wrong, with sometimes dire consequences.

- Get it right, and you’ll have a decent poker server that people will actually want to play on.