

# Lecture 15: Randomness

CS105: Great Insights in Computer Science  
Michael L. Littman, Fall 2006

# Voting Machines

[Sign in](#)

[Web](#) [Images](#) [Video](#) [News](#) [Maps](#) [more »](#) [New!](#) [Upload you](#)

**Google**  
Video BETA

diebold   [Advanced Video Search](#)

[Top 100](#) [Comedy](#) [Music videos](#) [Movies](#) [Sports](#) [Animation](#) [TV shows](#) !

Buffering...

## Diebold AccuVote-TS Security Demonstration

☆☆☆☆☆ Avg: ☆☆☆☆☆ 79 ratings

9,272 views »

9 min 28 sec - Sep 13, 2006  
[itpolicy.princeton.edu](http://policy.princeton.edu)

[Add label](#)

for

Princeton researchers demonstrate security flaws in a **Diebold** electronic voting machine.

« [Prev](#) - [Next video](#) »

[Playlist](#) - [Details](#) - [From user](#) - [Related](#) - [Comments](#) New! - [Flag as inappropriate](#)

Continuous Playback: **ON** - **OFF**

# Average of List

- Let's say we've got a list of  $n$  small integers ( $n = 10,000,000,000$ , for example).
- We want to know the average value of the integers.
- How can we calculate this value?
- What running time would you expect?

# Straightforward Algorithm

```
def average(l):  
    total = 0  
    for i in l:  
        total = total + i  
    return (total + 0.0)/len(l)  
  
def average2(l):  
    total = 0  
    for i in range(l):  
        total = total + l[i]  
    return (total + 0.0)/len(l)
```

- Totals up all the elements in the list.
- Divides by the length of the list.
- Running time proportional to the list length ( $O(n)$ ), which could be quite long...

# Sampling

- What if we are content with 2% error?
- To estimate the mean of a population (of bounded variance), the mean of a random sample approaches the mean of the population proportionally to the square root of the sample size.
- Error depends on variance, confidence, and sample size: Not the list size!

# Why Random?

```
def averageSample(l):
```

```
    m = 100
```

```
    total = 0
```

```
    for i in range(m):
```

```
        total = total + l[randint(0,len(l))]
```

```
    return (total + 0.0)/m
```

```
def averageFirst(l):
```

```
    m = 100
```

```
    total = 0
```

```
    for i in range(m):
```

```
        total = total + l[i]
```

```
    return (total + 0.0)/m
```

- What can go wrong if sample not random?
- $l = [0, \dots, 0, 1, \dots, 1]$  (600 0s, then 400 1s)
- `averageSample(l): 0.44`; `averageFirst(l): 0.0`;  
`average(l): 0.40`.

# Using Random Bits

- Since numbers are made of bits, we can generate a random number using random bits.
- If there's a way to create random bits (coin flips), how make a random number from 0 to 3 (dreidel)?
- How about 0 to 15?
- Tricky: How about 0 to 2?

# Examples

```
def rand4():  
    return randbit()* 2 + randbit()
```

```
def rand16():  
    return randbit()* 8 + randbit()* 4 + randbit()* 2 + randbit()
```

```
def rand3():  
    x = 3  
    while x == 3:  
        x = randbit()* 2 + randbit()  
    return x
```

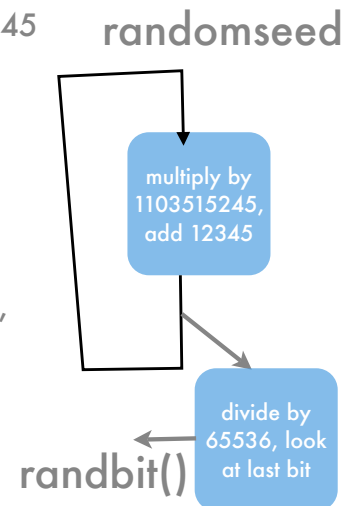
(1 1/3 calls per random number. 0, 1, 2 equally likely.)

# Simple randbit

```
randomseed = 1000
def randbit():
    global randomseed
    randomseed = randomseed * 1103515245 + 12345
    return int(randomseed / 65536) % 2 == 1
```

In 50 calls, 26 True, 24 False:

```
[True, True, True, True, False, True, False, True, False,
False, False, True, False, True, False, True, True, False,
True, False, True, True, True, True, False, True, True,
False, False, True, True, False, False, False, True, False,
True, False, False, True, True, True, False, True, False,
False, True, False, False, False]
```



# State of the Art

- The ... Mersenne twister algorithm, by Makoto Matsumoto and Takuji Nishimura in 1997 ... has a colossal period of  $2^{19937}-1$  iterations (probably more than the number of computations which can be performed in the future existence of the universe), is proven to be equidistributed in 623 dimensions (for 32-bit values), and runs faster than all but the least statistically desirable generators.
- Python has a package for this generator.

# Secret Codes

- In what context do computers try to keep secrets?
  - passwords
  - secure webpages
  - encrypted files
  - digital signatures / authentication
  - etc.

# Letter Substitution

- Caesar rotate (rot13).

abcdefghijklm  
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑  
nopqrstuvwxyz

```
def encode(c):  
    if c < 0 or c >= 26: return c  
    if c + 13 >= 26: return c-13  
    return c+13
```

```
def rot13(s):  
    return "".join([chr(encode(ord(i)-ord('a'))+ord('a')) for i in s])
```

```
rot13('michael littman') zvpunry yvggzna  
rot13('ravine')          enivar  
rot13('pbzchgref oht zr') ???
```

# Too Crackable

- rot13 is hard to read, but easy to decode.
- In fact, any letter-for-letter substitution code can be cracked given a long enough piece of text.
- Doesn't even need to be that long...

# Cryptogram

- “I ARRIVED AT THE AIRPORT ONE HOUR EARLY SO THAT, IN ACCORDANCE WITH AIRLINE PROCEDURES, I COULD STAND AROUND.” - \*DAVE \*BARRY
- B > I, PVS > THE, GP > AT, JBPV > WITH, BH > IN, DPGHA > STAND, KHS > ONE, GBOZKOP > AIRPORT, VKEO > HOUR, GOOBWSA > ARRIVED, SGOTU > EARLY, FKETA > COULD, QGOOU > BARRY

# Code in Bits

a	00000	i	01000	q	10000	y	11000
b	00001	j	01001	r	10001	z	11001
c	00010	k	01010	s	10010	_	11010
d	00011	l	01011	t	10011	.	11011
e	00100	m	01100	u	10100	?	11100
f	00101	n	01101	v	10101	!	11101
g	00110	o	01110	w	10110	,	11110
h	00111	p	01111	x	10111	:	11111

# A Message

a\_secret

00000**11010**1001000**100000**10**1000**100100**10011**

**a \_ s e c r e t**

Ok, well, that's actually not much of a secret, since I told you the code.

# XOR To Mix Things Up

- Here's an idea... we can flip some of the bits to make it harder to decode.
- $0 \text{ xor } 0 = 0$ ,  $0 \text{ xor } 1 = 1$ ,  $1 \text{ xor } 0 = 1$ ,  $1 \text{ xor } 1 = 0$
- (first bit = 1 means flip second bit)
- **pad** is the sequence of bits we will use to xor the message.

## Example

message = grade\_a

bits = 0011010001000000001100100111111101000000

pad = 0010001011100001111111000011111110010100

xor: 0001011010100001110011100100000011010100

encoded message = c\_q??qgu

Notice: repeated "q", repeated "?" don't correspond to repeated "a".

If pad is a secret, message is uncrackable.

# How Send Pad?

- Of course, now we're in the situation that we can send secret messages if we agree on a secret pad.
- But, how can we distribute the pad??
- Could be generated by a pseudo-random generator, if we agree on a seed.
- Hard to demo. Instead, let's use clicker numbers!

## Example

Each clicker number is a sequence of 40 bits:

#01751367      Jessica Bracey (jessbrcy)

<u>00000001</u>	0	0000	4	0100	8	1000	C	1100
<u>01110101</u>	1	0001	5	0101	9	1001	D	1101
<u>00010011</u>	2	0010	6	0110	A	1010	E	1110
<u>01100111</u>	3	0011	7	0111	B	1011	F	1111

# Public Key Idea

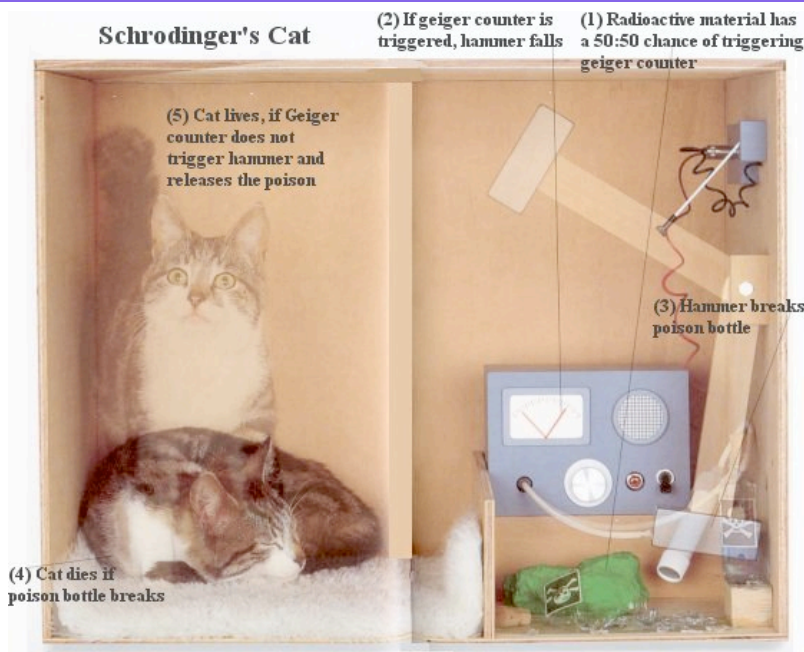
- I've got a number  $x$  that is the product of two big prime numbers.
- I tell everyone the product, which is what is needed to encrypt messages for me.
- Only *I* know the factors, which are what are needed to decrypt messages for me.
- Because factoring is hard, people can send me secret messages, even though I've publicized  $x$ .

# Quantum Effects

- Pseudo-random-based encryption always has a chance of being cracked.
- Only source of true randomness: quantum mechanics (the rest of physics is deterministic, if chaotic).
- Einstein didn't like it. Tough.

# Quantum Randomness

## Schrodinger's Cat



# Quantum Computer

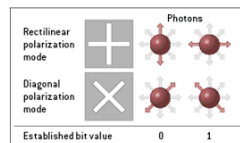
- A quantum bit (qubit) is simultaneously zero and one (a superposition).  $n$  qubits can represent  $2^n$  possibilities.
- When you look, one possibility presents itself, according to well understood probabilistic rules. A kind of parallel search.
- **Shor:** A computer with qubits can factor numbers in polynomial time!

# If Factoring is Easy...

- quantum computers invalidate standard cryptosystems. *No more secrets.*
- However, they also open up some wild possibilities.
- *quantum cryptography*: qubits can be completely random and correlated at a distance. The perfect pad!

## QUANTUM MECHANICS HIDES A SECRET CODE KEY

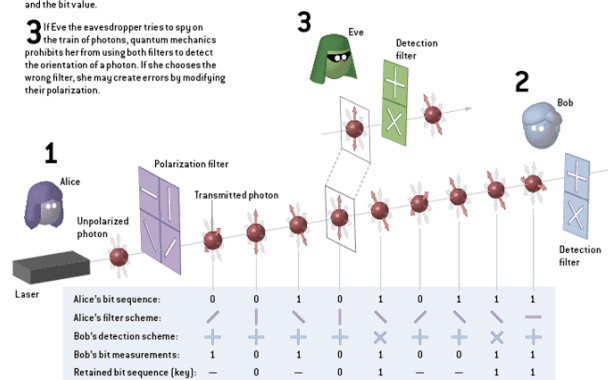
Alice and Bob try to keep a quantum-cryptographic key secret by transmitting it in the form of polarized photons, a scheme invented by Charles Bennett of IBM and Gilles Brassard of the University of Montreal during the 1980s and now implemented in a number of commercial products.



**1** To begin creating a key, Alice sends a photon through either the 0 or 1 slot of the rectilinear or diagonal polarizing filters, while making a record of the various orientations.

**2** For each incoming bit, Bob chooses randomly which filter slot he uses for detection and writes down both the polarization and the bit value.

**3** If the eavesdropper tries to spy on the train of photons, quantum mechanics prohibits her from using both filters to detect the orientation of a photon. If she chooses the wrong filter, she may create errors by modifying their polarization.



**4** After all the photons have reached Bob, he tells Alice over a public channel, perhaps by telephone or an e-mail, the sequence of filters he used for the incoming photons, but not the bit value of the photons.

**5** Alice tells Bob during the same conversation which filters he chose correctly. Those instances constitute the bits that Alice and Bob will use to form the key that they will use to encrypt messages.

# More on Randomness

- If event has probability  $p$ ,  $1/p$  tries before it happens (on average).
- If  $n$  distinct events are equally likely,  $\sim n \ln n$  tries before we see all  $n$  of them (order independent) or  $n^2$  (in order).
- If we start with a number  $n$ , we can cut it in half  $\log n$  times before 1 is reached.

# How About This?

- Start with a number  $n$ .
- Change it to a number between 1 and  $n$  at random.
- Stop when you reach 1.
- How many times do we do the change before 1 is reached (on average)?

# I Don't Know

- Here's a different game, which has to take no less time than the one I just described.
- With probability  $1/2$ , don't change  $n$ . With probability  $1/2$ , change  $n$  to  $n/2$ .
- On average, 2 tries before  $n$  is halved and  $\log n$  halves before 1 is reached.
- So, like  $2 \log n$ .

# Next Time

- Image processing.
- We've completed Hillis 1-5. We'll start on Chapter 6 after the break/midterm.