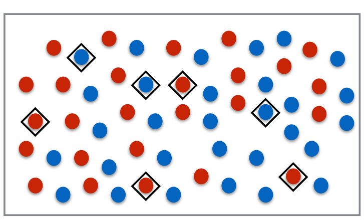
15-251
Great Ideas in Theoretical Computer Science

Lecture 21:
Randomized Algorithms I

April 3rd, 2018

Randomness and Computer Science

Statistics via Sampling



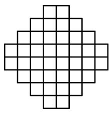
Population: 300m **Random sample size**: 2000

Theorem:

Randomized Algorithms

Dimer Problem:

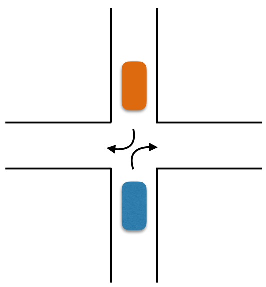
Given a region, in how many different ways can you tile it with 2x1 rectangles (dominoes)?

e.g.  → 1024 tilings

Captures thermodynamic properties of matter.

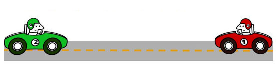
- Fast *randomized* algs can approximately count.
- No fast *deterministic* alg known.

Distributed Computing



Nash Equilibria in Games

The Chicken Game



	Swerve	Straight
Swerve	1 1	0 2
Straight	2 0	-3 -3

Theorem (Nash):

Cryptography

Adversary
Eavesdropper

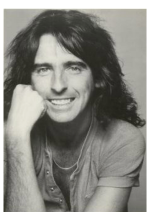
"loru23n8uladjkfb!#@"

"I will cut your throat"
encryption
"loru23n8uladjkfb!#@"

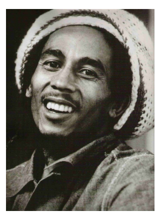
"loru23n8uladjkfb!#@"
decryption
"I will cut your throat"

Shannon:

Error-Correcting Codes



"bit.ly/vrxUBN"
noisy channel

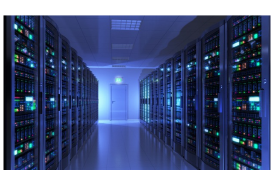


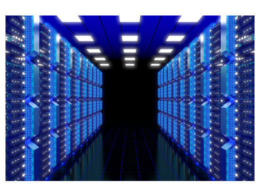
Alice

Bob

Each symbol can be corrupted with a certain probability.
How can Alice still get the message across?

Communication Complexity

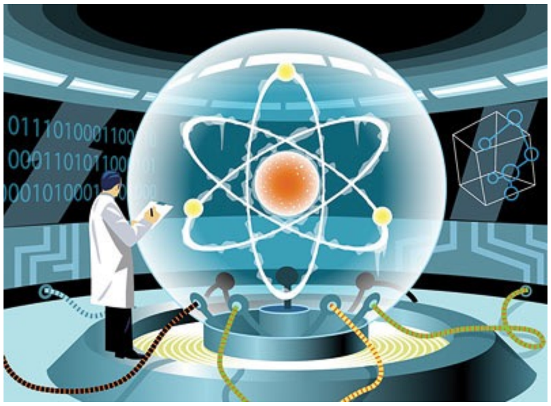




Want to check if the contents of two databases are exactly the same.

How many bits need to be communicated?

Quantum Computing



**Probability Theory:
The CS Approach**

The Big Picture

The Non-CS Approach

Real World → Mathematical Model

(random) experiment/process

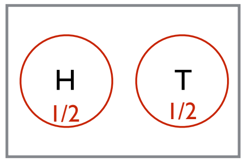
probability space

The Big Picture

Real World → Mathematical Model

Flip a coin.

Ω



Ω = "sample space"

= set of all possible outcomes

$\Pr : \Omega \to [0, 1]$ prob. distribution

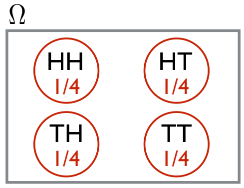
$$\sum_{\ell \in \Omega} \Pr[\ell] = 1$$

The Big Picture

Real World → Mathematical Model

Flip two coins.

Ω

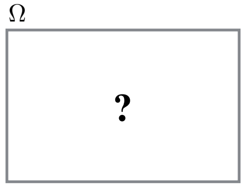


The Big Picture

Real World → Mathematical Model

Flip a coin.
If it is Heads, throw a 3-sided die.
If it is Tails, throw a 4-sided die.

Ω



The Big Picture

The CS Approach

The Big Picture

Flip a coin.
If it is Heads, throw a 3-sided die.
If it is Tails, throw a 4-sided die.

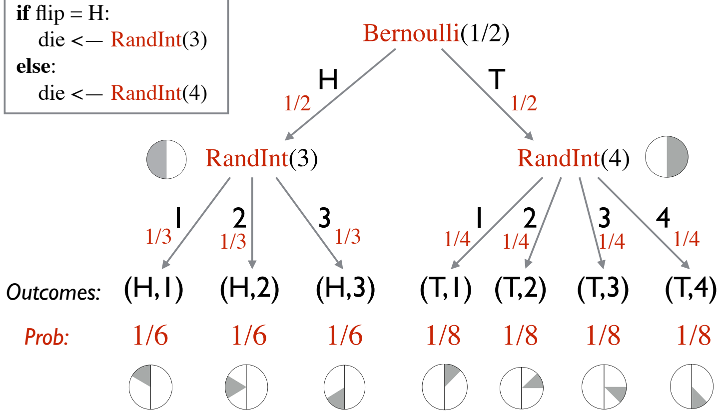
→

```
flip <— Bernoulli(1/2)
if flip = 1: # i.e. Heads
   die <— RandInt(3)
else:
   die <— RandInt(4)
```

→

Probability Tree

```
flip <— Bernoulli(1/2)
if flip = H:
   die <— RandInt(3)
else:
   die <— RandInt(4)
```



What is a Random Variable?

A **random variable** is a variable in some randomized code
(more accurately, the variable's value at the end of the execution)
of type 'real number'.

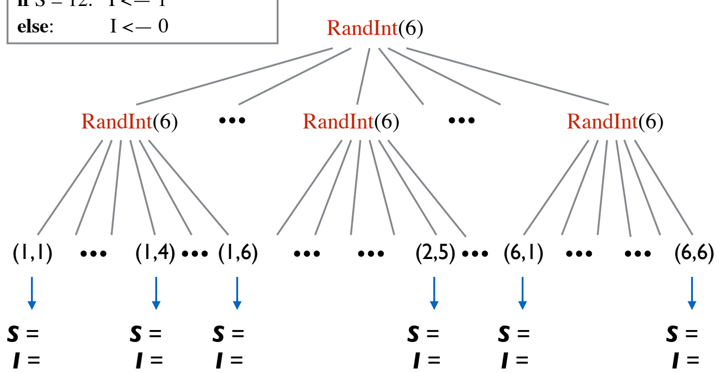
Example:

```
S <— RandInt(6) + RandInt(6)
if S = 12:   I <— 1
else:        I <— 0
```

Random variables:

What is a Random Variable?

```
S <— RandInt(6) + RandInt(6)
if S = 12:   I <— 1
else:        I <— 0
```



New Topic:

Randomized Algorithms

Randomness and algorithms

How can randomness be used in computation?

Given some algorithm that solves a problem:

(i) the input can be chosen randomly

(ii) the algorithm can make random choices

Which one will we focus on?

Randomness and algorithms

What is a <u>randomized algorithm</u>?

A *randomized algorithm* is an algorithm that is allowed to "***flip a coin***" (i.e., has access to random bits).

<u>In 15-251:</u>

A randomized algorithm is an algorithm that is allowed to call:

Deterministic vs Randomized

Deterministic	**Randomized**
def A(x): y = 1 **if**(y == 0): **while**(x > 0): x = x - 1 **return** x+y	**def** A(x): y = Bernoulli(0.5) **if**(y == 0): **while**(x > 0): x = x - 1 **return** x+y

For any <u>fixed</u> input (e.g. x = 3):

Deterministic	Randomized
- the output - the running time	- the output - the running time

Deterministic vs Randomized

A **deterministic algorithm** A computes $f : \Sigma^* \to \Sigma^*$ in time $T(n)$ means:

- **correctness**: $\forall x \in \Sigma^*, \quad A(x) = f(x)$.

- **running time**: $\forall x \in \Sigma^*$, $\#$ steps $A(x)$ takes is $\leq T(|x|)$.

Note: we require worst-case guarantees for correctness and run-time.

Deterministic vs Randomized

A Try

A **randomized algorithm** A computes $f : \Sigma^* \to \Sigma^*$ in time $T(n)$ means:

- **correctness**: $\forall x \in \Sigma^*$,

- **running time**: $\forall x \in \Sigma^*$,

Is this interesting?

$\forall x \in \Sigma^*$

		Correctness	Run-time
Deterministic			
Randomized	Type 0		
	Type 1		
	Type 2		
	Type 3		

Type 0:
Type 1:
Type 2:
Type 3:

Example

Input: An array B with **n/4** 1's and **3n/4** 0's.
Output: An index that contains a 1.

Deterministic	Randomized	
	Type 1 (Monte Carlo)	Type 2 (Las Vegas)

Example

Input: An array B with **n/4** 1's and **3n/4** 0's.
Output: An index that contains a 1.

	Correctness	Run-time
Deterministic		
Monte Carlo		
Las Vegas		

Formal Definitions

Formal Definition: Deterministic

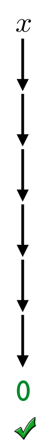
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem.

We say that deterministic algorithm A computes f in time $T(n)$ if:

$\forall x \in \Sigma^*,$ $\quad A(x) = f(x)$

$\forall x \in \Sigma^*,$ $\quad \#\text{ steps } A(x) \text{ takes is } \le T(|x|).$

Picture:



Deterministic:

Each input x induces a deterministic path.

Formal Definition: Monte Carlo

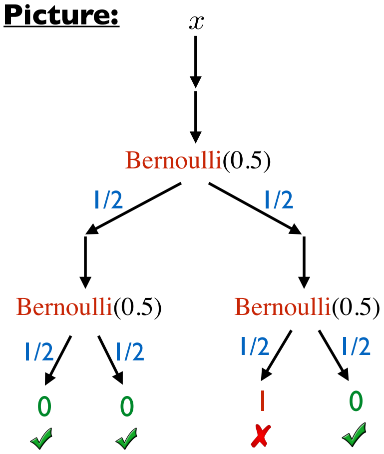
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem.

We say that randomized algorithm A is a $T(n)$-time **Monte Carlo algorithm** for f with ϵ error probability if:

$\forall x \in \Sigma^*,$

$\forall x \in \Sigma^*,$

Picture:



Monte Carlo:

Each input x induces a probability tree.

Formal Definition: Las Vegas

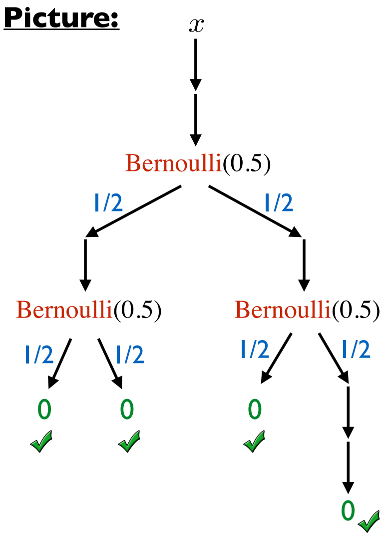
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem.

We say that randomized algorithm A is a $T(n)$-time **Las Vegas algorithm** for f if:

$\forall x \in \Sigma^*,$

$\forall x \in \Sigma^*,$

Picture:



Las Vegas:

Each input x induces a probability tree.

Examples

---

3 IMPORTANT PROBLEMS

Integer Factorization

Input: integer N

Ouput: a prime factor of N

isPrime

Input: integer N

Ouput: True if N is prime.

Generating a random n-bit prime

Input: integer n

Ouput: a random n-bit prime

Most crypto systems start like:

- pick two random n-bit primes P and Q.
- let N = PQ. (N is some kind of a "key")
- (*more steps…*)

We should be able to do **efficiently** the following:

- check if a given number is prime.
- generate a random prime.

We should **not** be able to do **efficiently** the following:

- given N, find P and Q. (the system is broken if we can do this!!!)

isPrime

```
def isPrime(N):
    if (N < 2):  return False
    maxFactor = round(N**0.5)
    for factor in range(2, maxFactor+1):
        if (N % factor == 0):  return False
    return True
```

Problems:

isPrime

Amazing result from 2002:

There is a poly-time algorithm for isPrime.

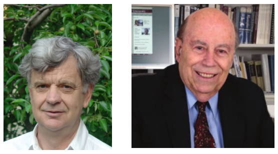
Agrawal, Kayal, Saxena

However, best known implementation is ~ $O(n^6)$ time.
Not feasible when $n = 2048$.

isPrime

So that's **not** what we use in practice.

Everyone uses the Miller-Rabin algorithm (1975).



The running time is:

Why is the previous result a breakthrough?

Generating a random prime

```
repeat:
    let N be a random n-bit number
    if isPrime(N): return N
```

Prime Number Theorem (informal):

$\Longrightarrow$ expected run-time of the above algorithm:

No poly-time deterministic algorithm is known to generate an n-bit prime!!!