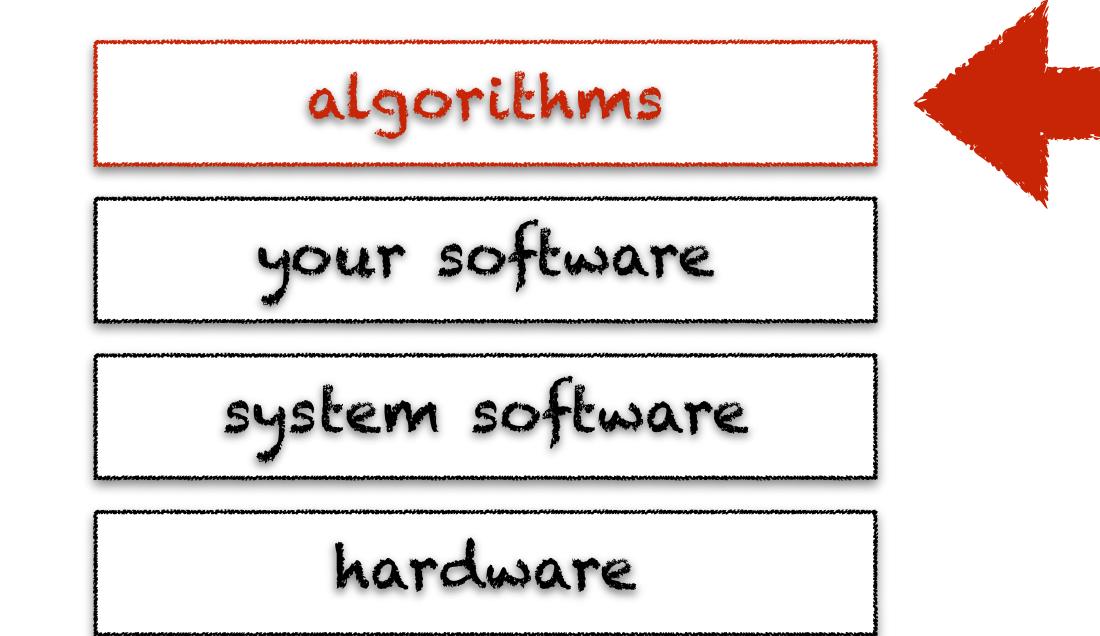


probabilistic
algorithms



learning objectives

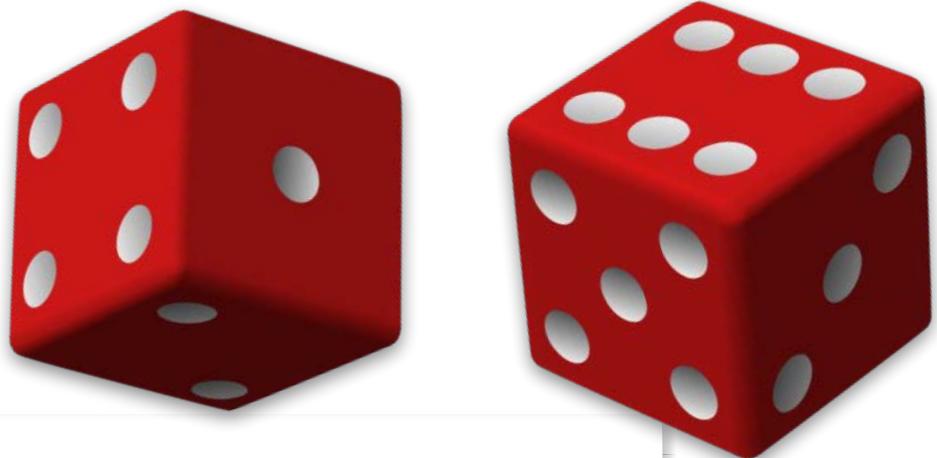


- learn what randomized algorithms are
- learn what they are useful for
- learn about pseudo-randomness



determinism vs randomness

R. M. Karp. *An introduction to randomized algorithms*. Discrete Applied Mathematics, 34(1-3):165–201, November 1991.



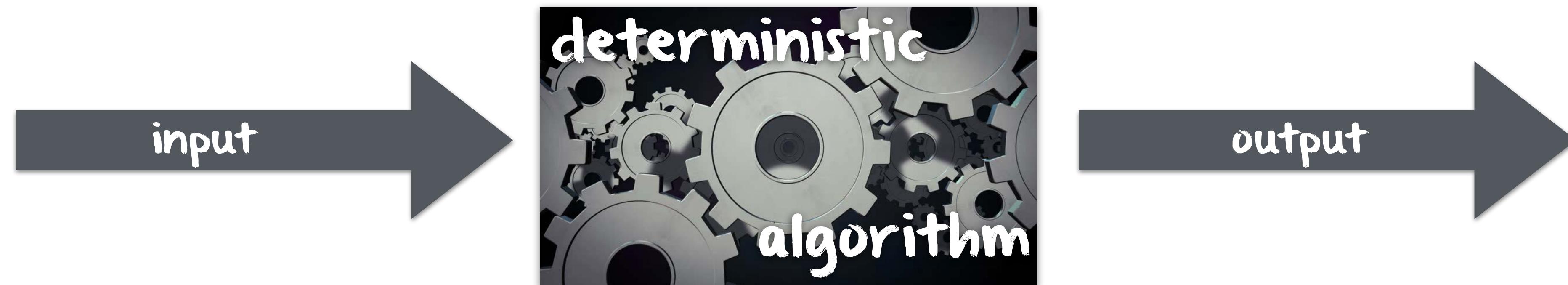
a computer is **deterministic** by design, so an algorithm executing on a computer is inherently deterministic

yet we can abstractly define the notion of **probabilistic** or **randomized algorithm** as follows:

a **randomized algorithm** is one that receives, in addition to its input data, a stream of random bits used to make random choices

so even for the same input, different executions of a randomized algorithm may give **different outputs**

deterministic algorithm vs randomized algorithm



...01010110101101101...



why introduce randomness?

because randomized algorithms tend to be much simpler than their deterministic counterpart

because randomized algorithms tend to be more efficient than their deterministic counterpart*

*in execution time and memory space

but some randomized algorithms do not always* provide a correct answer (only probabilistically)

*always \Leftrightarrow deterministically

principles to construct randomized algorithms

abundance of witnesses

fingerprinting

random partitioning

random sampling

foiling the adversary

random ordering

Markov chains

abundance of witnesses

are these two polynomials of degree $d = 5$ identical?

$$p(x) = (x - 7)(x - 3)(x - 1)(x + 2)(2x + 5)$$

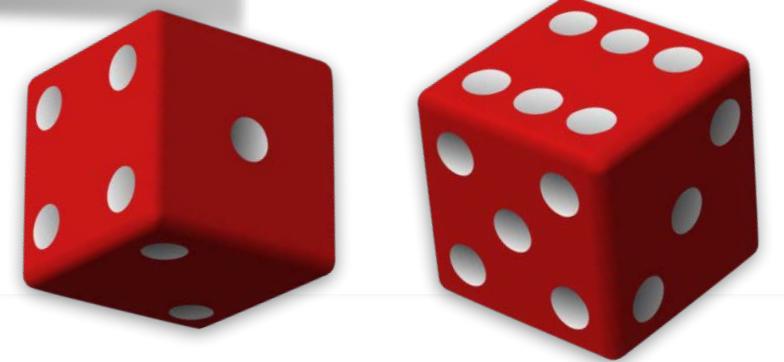
$$q(x) = 2x^5 - 13x^4 - 21x^3 + 127x^2 + 121x - 210$$



expanding $p(x)$ may take up to $O(d^2) = O(25)$ time*

*provided integer multiplication takes a unit of time

a randomized algorithm can take $O(d) = O(5)$ time



note
that:

- ◆ computing $p(\dot{x})$ and $q(\dot{x})$ for a given value $\dot{x} \in \mathbb{Z}$ takes $O(d)$
- ◆ $p(\dot{x}) = q(\dot{x})$ is true if at least one of the following conditions is true
 1. we have the following polynomial equality $p(x) = q(x)$
 2. the value \dot{x} is a root of polynomial $p(x) - q(x)$, i.e., if $p(\dot{x}) - q(\dot{x}) = 0$
- ◆ since $p(x) - q(x)$ is of degree $d = 5$, it has no more than 5 roots

abundance of witnesses

algorithm

- ◆ randomly choose \dot{x} from a very large range of integer $R \subset \mathbb{Z}$
- ◆ compute $r = p(\dot{x}) - q(\dot{x})$
- ◆ if $r = 0$, then $p(x) = q(x)$ is true with probability $1 - \frac{d}{|R|}$

\dot{x} is our potential witness that $p(x) \neq q(x)$

after n trials, the error

probability is $\left(\frac{d}{|R|}\right)^n$

after $d + 1$ trial, the error
probability drops to 0

this is a
Monte Carlo algorithm

Monte Carlo & Las Vegas algorithms

a Monte Carlo algorithm computes in a deterministic time
but only provides a correct answer probabilistically

a false-biased Monte Carlo algorithm is
always correct when returning false

a true-biased Monte Carlo algorithm is
always correct when returning true

a Las Vegas algorithm computes in some random
time but always* provides a correct answer

*always \Leftrightarrow deterministically

a Monte Carlo algorithm can be turned into a Las Vegas
algorithm if we have a way to verify that the output is correct

bob has x , a very long string of bits



fingerprinting

...1010011100010010110101001010010100101...

they want to check if $x = y$ but their channel has limited bandwidth



fingerprinting consist in computing much shorter strings of bits from x and y , so-called fingerprints, to then exchange them

alice has y , a very long string of bits

a typical fingerprinting function is $h_p(s) = h(s) \bmod p$, where $h(s)$ is the integer corresponding to the string of bits s and p is a prime number

algorithm

$h_p(s)$ is called a (high performance) hash function

- ◆ bob randomly chooses a prime number p less than M
- ◆ bob sends p and $h_p(x)$ to alice
- ◆ alice checks whether $h_p(x) = h_p(y)$ and sends the results to bob

bob has x , a very long string of bits



s

Pour qui sont ces serpents qui sifflent sur vos têtes ?

Pour qui sont **ses** serpents qui sifflent sur vos têtes ?

Pour qui sont **ses** serpents qui sifflent sur **nos** têtes ?

fingerprinting

...10100111000100101101001010010100101...

$h_p(s)$

c460b365f80157be
06edf169828af6db
6d65cf1b8d997b65
bf8a331bde275a8d

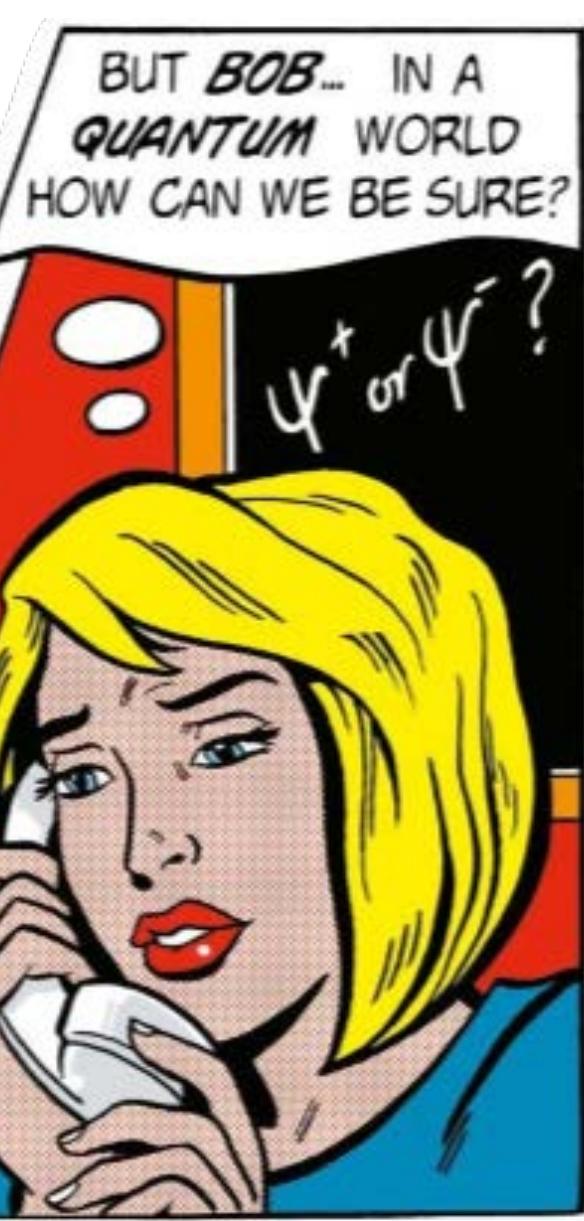
SHA-256 hash

SHA-256 hash

SHA-256 hash

fc9fd83cc8667559
e4ec841c6ab522fa
4de98c4c63d10637
0d36bc4f268f452a

c97db594b7966177
1f62ea2a5029b1ea
367b0211deae9459
a747198c3e33f6cd



alice has y , a very long string of bits

foiling the adversary via random ordering

we can see the execution of an algorithm
as a zero-sum two-person game



chooses
the input

the payoff is the execution time

long is good

short is good



chooses
the algorithm

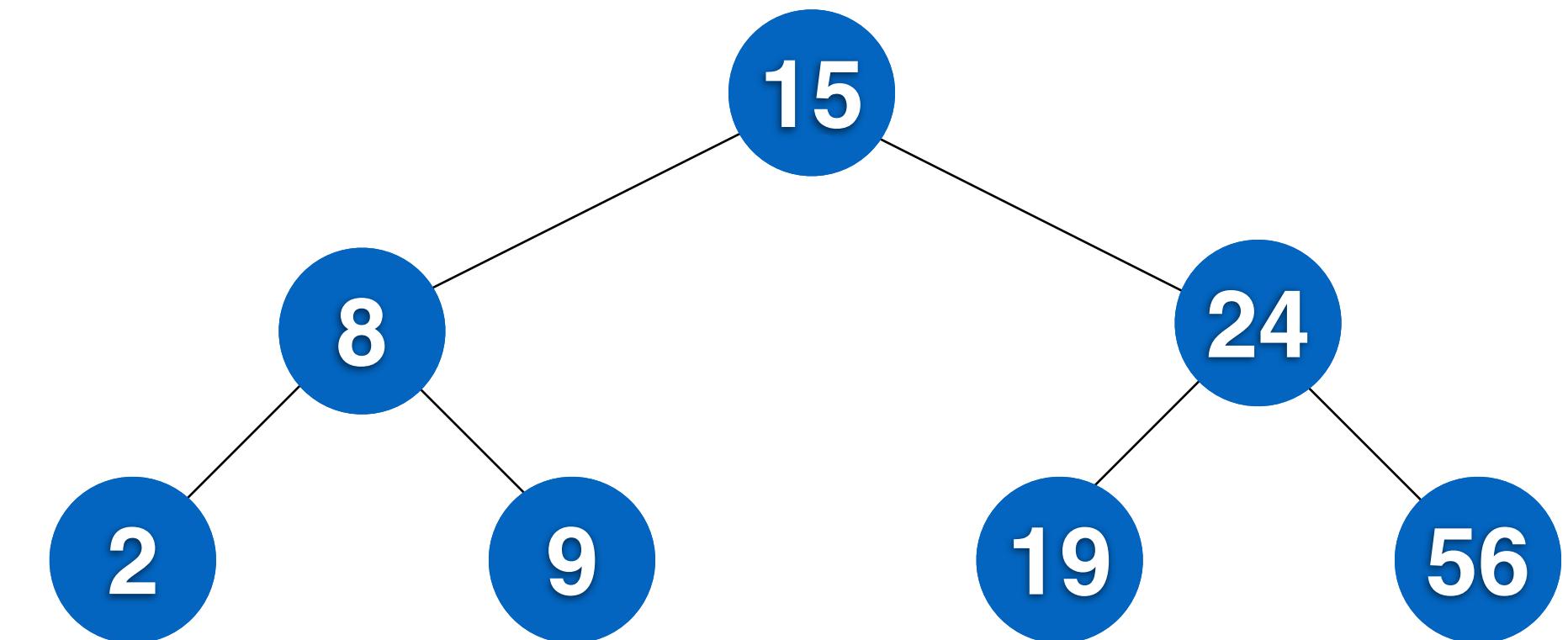
a randomized algorithm can be seen as a probabilistic distribution over deterministic algorithms, i.e., as mixed strategy for the algorithm player

faced with a mixed strategy, the input player does not know what the algorithm player will do with the input

this uncertainty makes it difficult for the input player to choose an input that will slow down the execution time

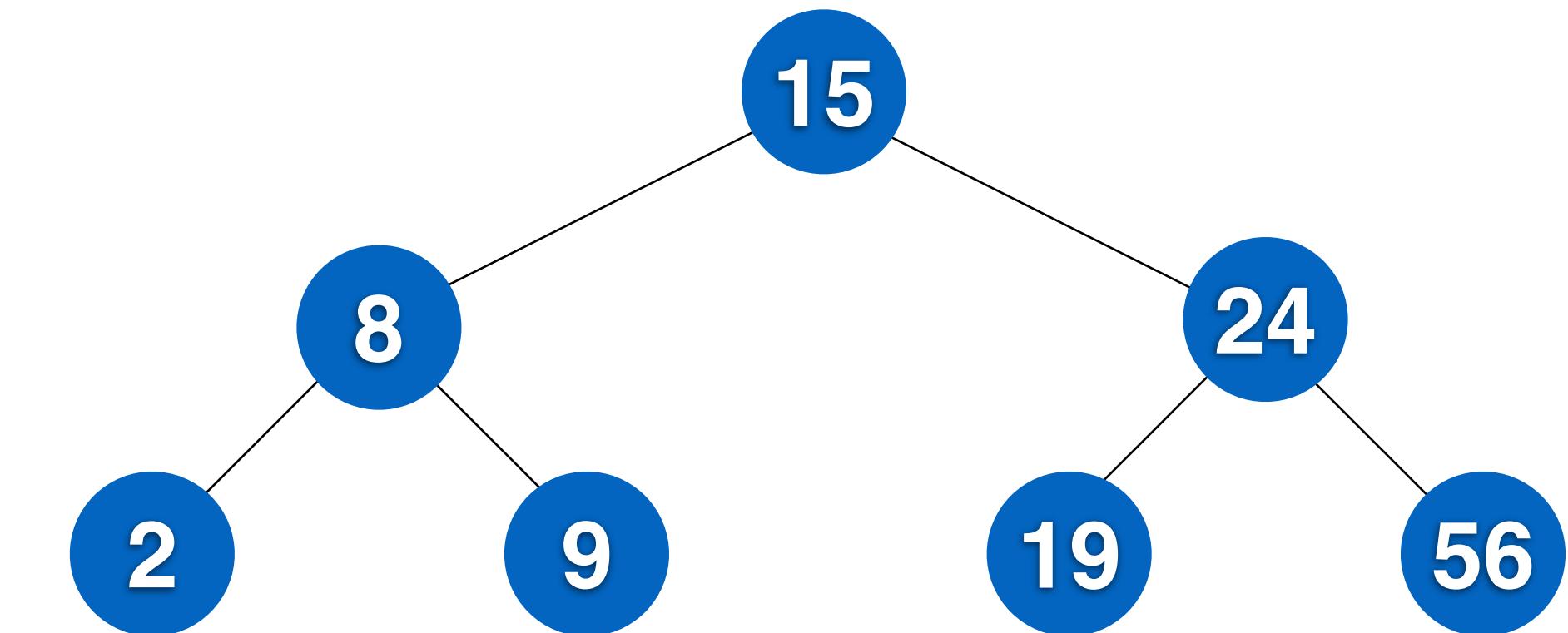
foiling the adversary via random ordering

the performance of a binary search tree depends on its structure, which in turn depends on the order in which its elements were inserted

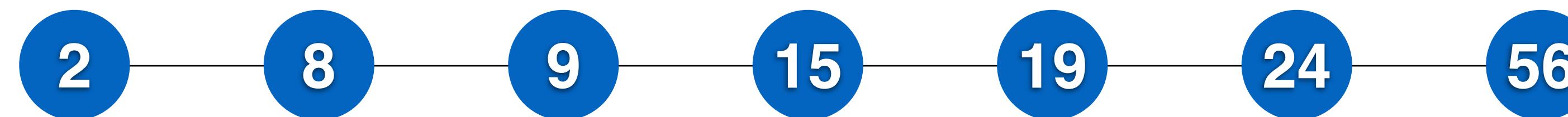


foiling the adversary via random ordering

the performance of a binary search tree depends on its structure, which in turn depends on the order in which its elements were inserted

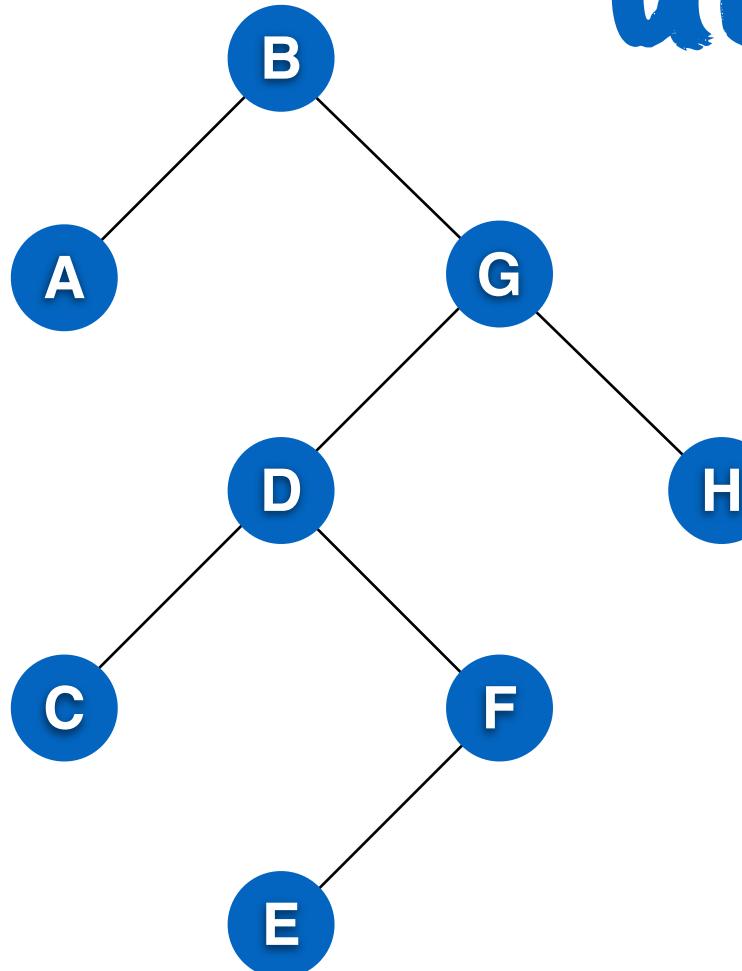


we cannot assume insertion are made in random order, so we can end up with a binary search tree with catastrophic performance



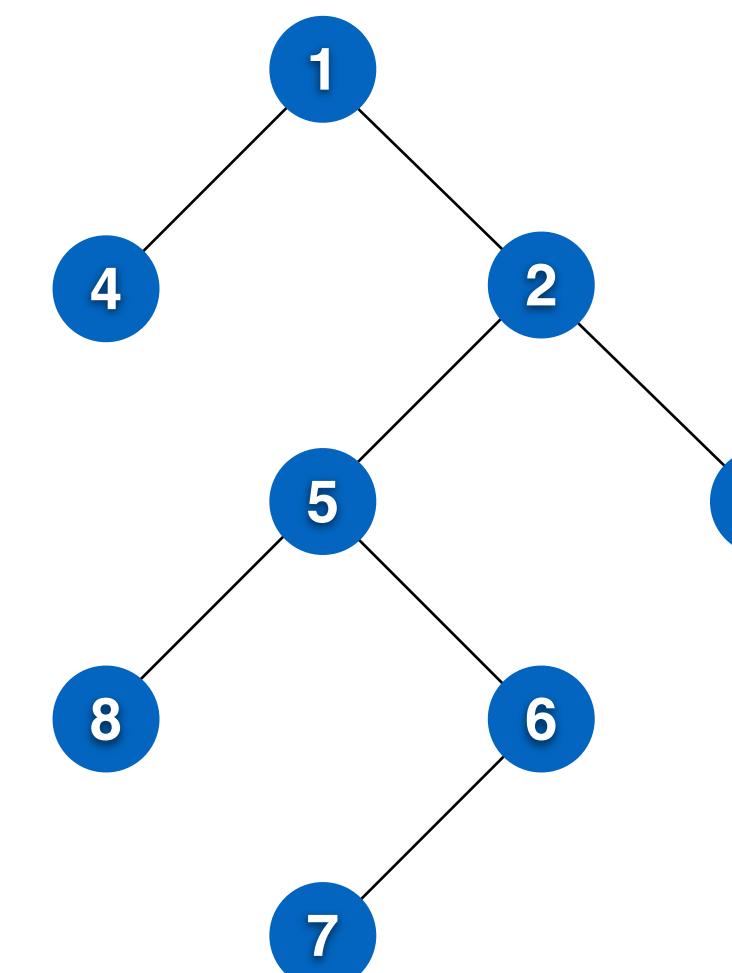
how can we get a binary search tree that looks like one resulting from insertions in random order whatever the execution?

foiling the adversary



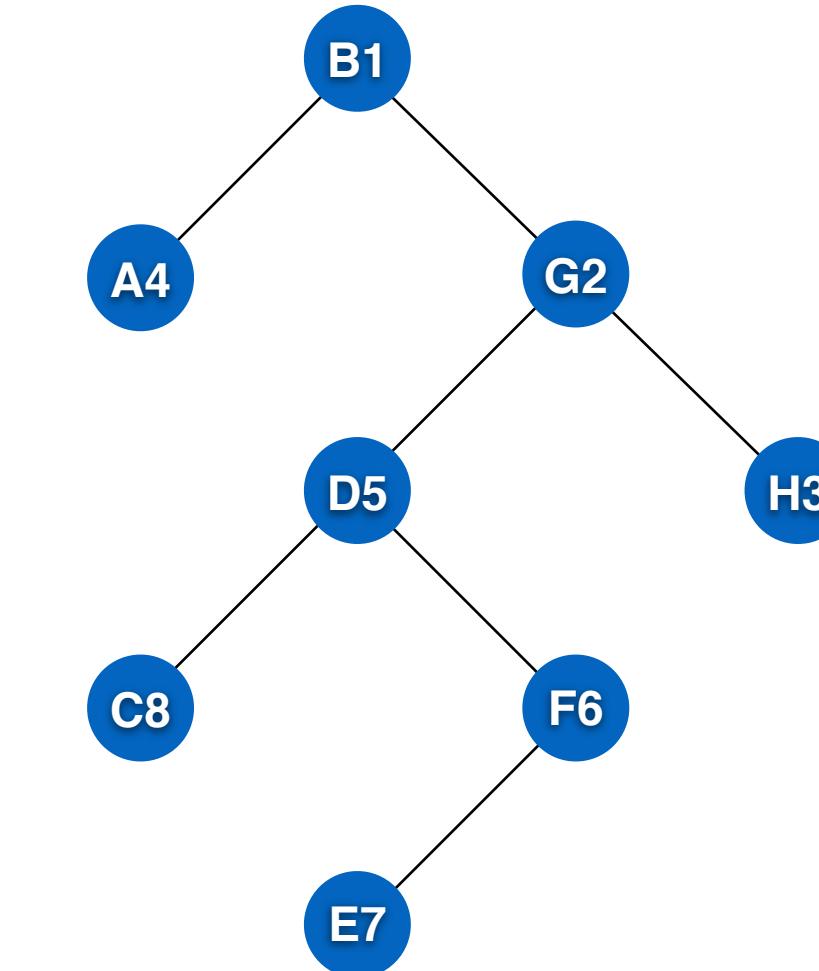
binary search tree

via



heap

random ordering



treap

a **heap** is a binary tree where the vertices on any path from the root to a leaf increase in value

a **treap** is a binary tree where each vertex v has two values, $v.key$ and $v.priority$ and which is a binary search tree with respect to key values and a heap with respect to $priority$ values

foiling the adversary via random ordering

given n items with associated keys and priorities, there exists a unique treap containing these n items

this unique treap has the same structure as a binary search tree where these n items would have been inserted in increasing order of priorities

algorithm for inserting key k

the random priority acts as a randomized timestamp

- ◆ draw a random priority p
- ◆ create new vertex v with $v.key = k$ and $v.priority = p$
- ◆ insert v in the treap

at any given time, we have a binary search tree obtained by random insertion

Markov chains

a Markov chain is a **stochastic*** process satisfying the **Markov property**, which states that the next state of the process only depends on its present state

*stochastic \Leftrightarrow probabilistic \Leftrightarrow non-deterministic

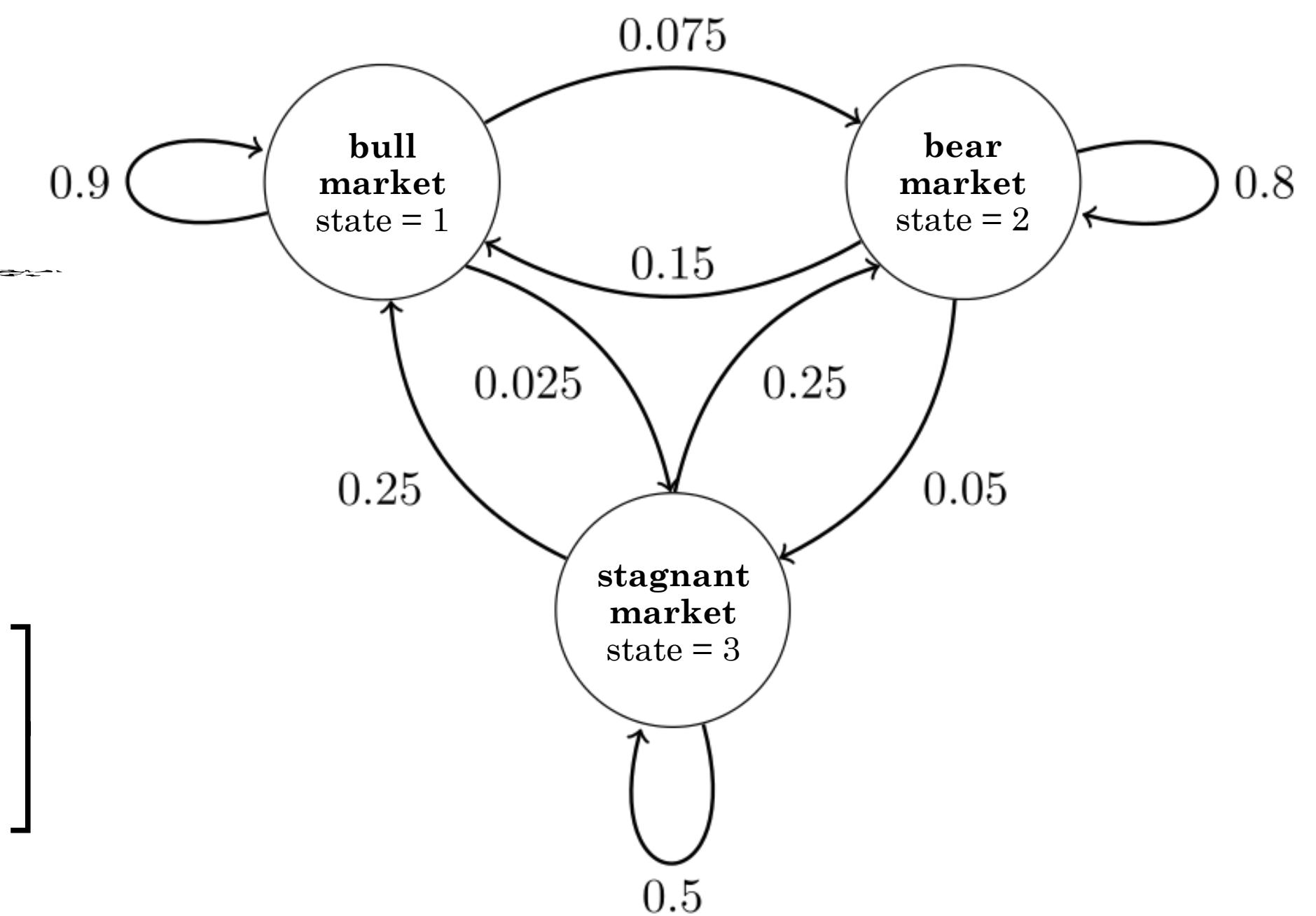
$$\Pr(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) = \Pr(X_{n+1} = x \mid X_n = x_n)$$

transition matrix

$$\begin{bmatrix} 0.9 & 0.075 & 0.025 \\ 0.15 & 0.8 & 0.05 \\ 0.25 & 0.25 & 0.5 \end{bmatrix}$$

assume that at time t , state = 2 then at time $t + 3$, we will have:

$$\begin{aligned} x^{(t+3)} &= [0 \ 1 \ 0] \begin{bmatrix} 0.9 & 0.075 & 0.025 \\ 0.15 & 0.8 & 0.05 \\ 0.25 & 0.25 & 0.5 \end{bmatrix}^3 \\ &= [0 \ 1 \ 0] \begin{bmatrix} 0.7745 & 0.17875 & 0.04675 \\ 0.3575 & 0.56825 & 0.07425 \\ 0.4675 & 0.37125 & 0.16125 \end{bmatrix} \\ &= [0.3575 \ 0.56825 \ 0.07425]. \end{aligned}$$



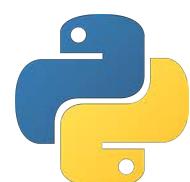
how to generate randomness in a deterministic machine?

pseudo random number generator

a parameterized set of function $g = \{g_n\}$ such that each function $g_n : \langle 0,1 \rangle^n \rightarrow \langle 0,1 \rangle^{t(n)}$
takes a seed string of n bits and stretches to a longer string of length $t(n)$

$O(n^k)$

not polynomial-time test can distinguish the output of
 g_n from a true random sequence of bits



```
import random

random.seed(666)
f = random.random()           → 0.0 ≤ f < 1.0
i = random.randint(2,9)        → 2 ≤ i ≤ 9
```



```
import java.util.Random;

Random rand = new Random();
rand.setSeed(666);
float f = rand.nextFloat();    → 0.0 ≤ f < 1.0
int i = rand.nextInt(9);       → 0 ≤ i < 9
```

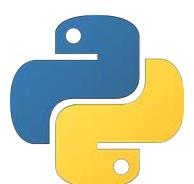
how to generate randomness in a deterministic machine?

pseudo random number generator

a parameterized set of function $g = \{g_n\}$ such that each function $g_n : \{0,1\}^n \rightarrow \{0,1\}^{t(n)}$
takes a seed string of n bits and stretches to a longer string of length $t(n)$

$O(n^k)$

not polynomial-time test can distinguish the output of
 g_n from a true random sequence of bits



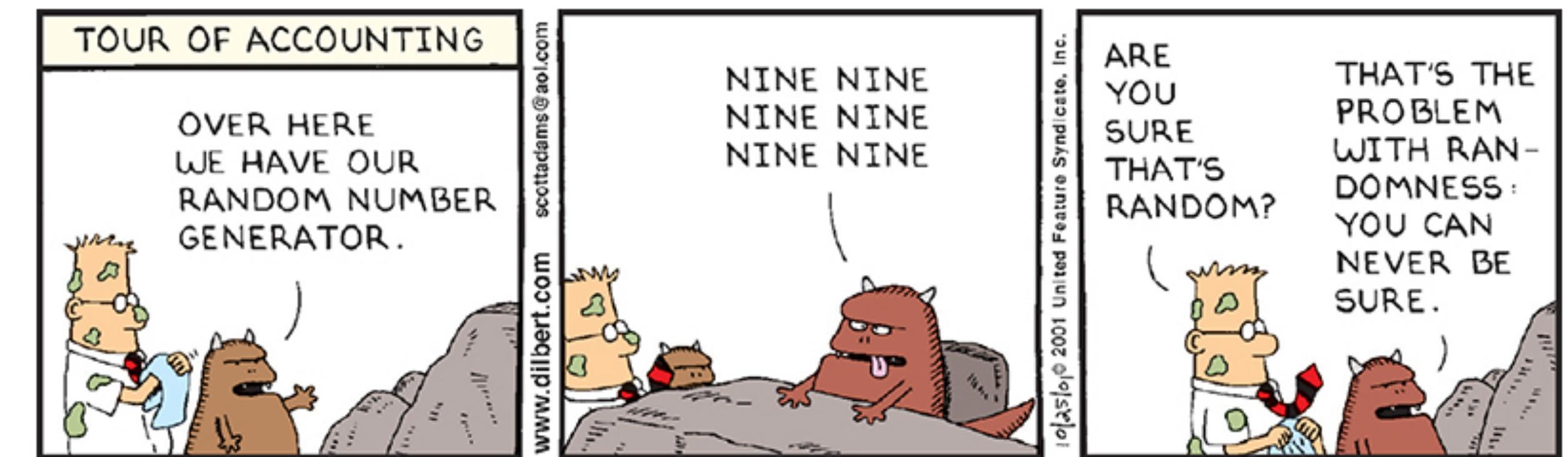
```
import random

random.seed(666)
f = random.random()           → 0.0 ≤ f < 1.0
i = random.randint(2,9)        → 2 ≤ i ≤ 9
```



```
import java.util.Random;

Random rand = new Random();
rand.setSeed(666);
float f = rand.nextFloat();    → 0.0 ≤ f < 1.0
int i = rand.nextInt(9);       → 0 ≤ i < 9
```

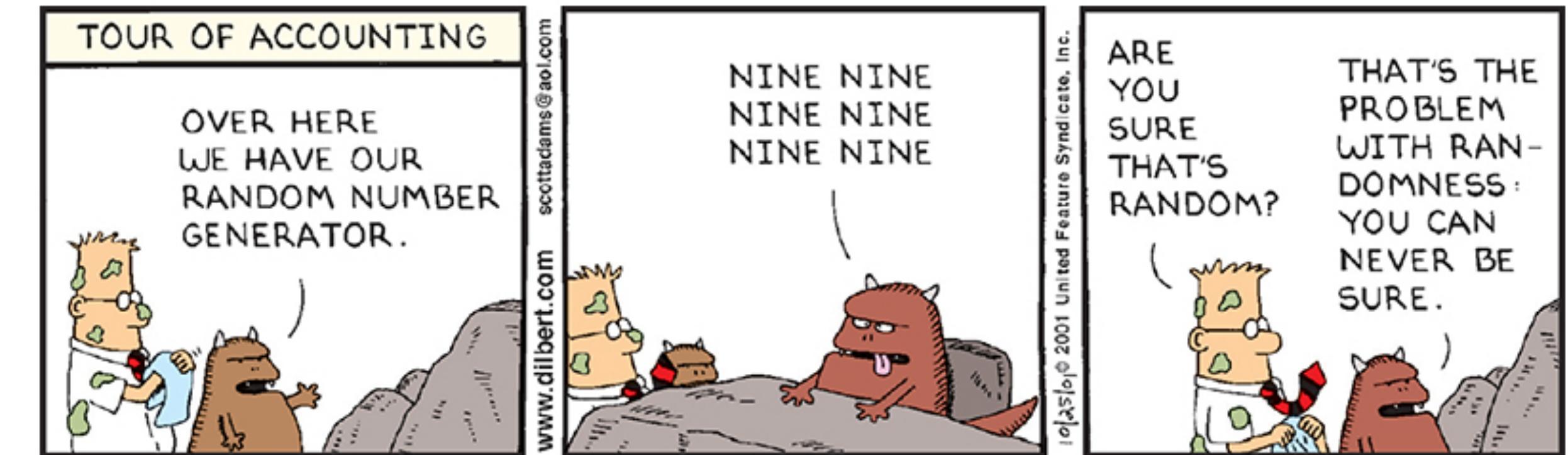


how to generate randomness in a deterministic machine?

true random number generator

do computers have a **real**
source of random bits?

augment computers with a intrinsically
non-deterministic **physical source**



nuclear decay radiation, thermal
noise from a resistor, etc...

