https://algs4.cs.princeton.edu

RANDOMNESS

- *what it is and what it isn't*
- *Las Vegas and Monte Carlo*
- *Karger's algorithm*
- *more applications*

Last updated on 11/21/24 1:49PM

A brief recap: where we've already encountered randomness

Percolation. Monte Carlo simulation: open random blocked sites.

Randomized queues. Remove item chosen uniformly at random.

A brief recap: where we've already encountered randomness

```
Test 2: open random sites until the system percolates
Test 7: open random sites with large n
Test 12: call open(), isOpen(), and numberOfOpenSites()
         in random order until just before system percolates
Test 13: call open() and percolates() in random order until just before system
percolates
Test 14: call open() and isFull() in random order until just before system percolates
Test 15: call all methods in random order until just before system percolates
Test 16: call all methods in random order until almost all sites are open
         (with inputs not prone to backwash)
Test 20: call all methods in random order until all sites are open
         (these inputs are prone to backwash)
```

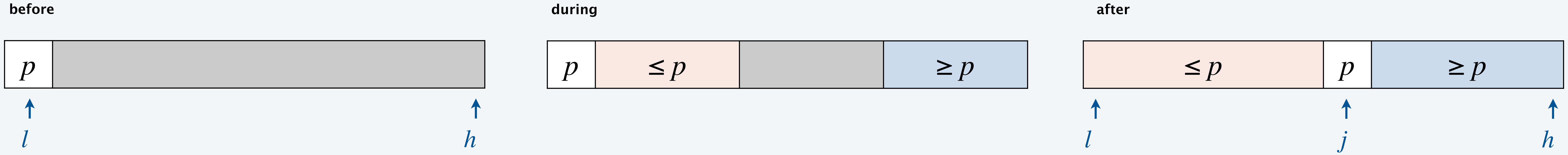
A brief recap: where we've already encountered randomness

```
Tests 1-8 make random intermixed calls to addFirst(), addLast(),
removeFirst(), removeLast(), isEmpty(), and size(), and iterator().
Test 12: check iterator() after random calls to addFirst(), addLast(),
         removeFirst(), and removeLast() with probabilities (p1, p2, p3, p4)
Tests 1-6 make random intermixed calls to enqueue(), dequeue(), sample(),
isEmpty(), size(), and iterator().
Test 16: check randomness of sample() by enqueueing n items, repeatedly calling
         sample(), and counting the frequency of each item
Test 17: check randomness of dequeue() by enqueueing n items, dequeueing n items,
         and seeing whether each of the n! permutations is equally likely
Test 18: check randomness of iterator() by enqueueing n items, iterating over those
         n items, and seeing whether each of the n! permutations is equally likely
```

A brief recap: where we've already encountered randomness

Quicksort is a randomized algorithm.

Shuffling is needed for performance guarantee.



Hash tables.


Algorithms

Robert Sedgewick | Kevin Wayne

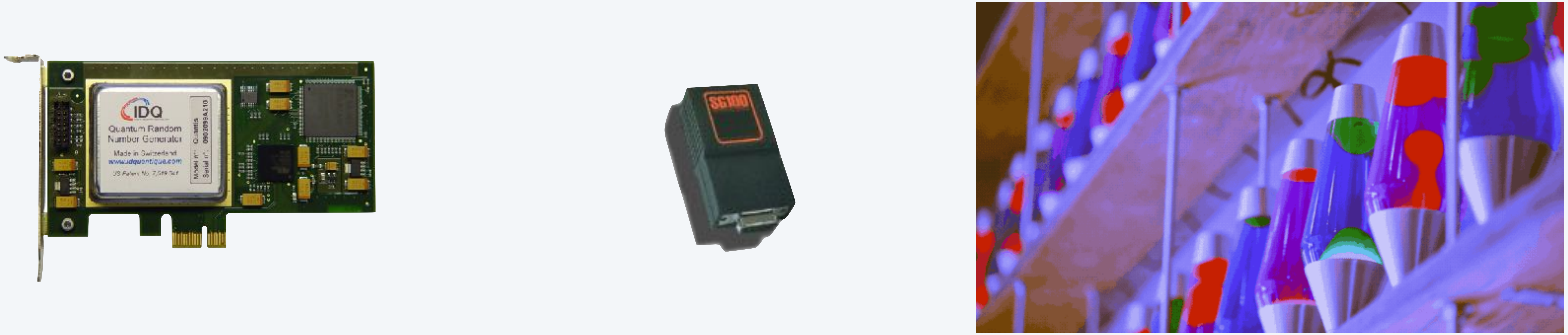
https://algs4.cs.princeton.edu

RANDOMNESS

- ▸ *what it is and what it isn't*
- ▸ *Las Vegas and Monte Carlo*
- ▸ *Karger's algorithm*
- ▸ *more applications*

Pseudorandomness

Computers can’t generate randomness (without specialized hardware).



Pseudorandom functions.

Class Random

java.lang.Object
    java.util.Random

All Implemented Interfaces:

`Serializable`

Direct Known Subclasses:

`SecureRandom`, `ThreadLocalRandom`

Randomness: quiz 1

Which of these outcomes is most likely to occur in a sequence of 6 coin flips?

A.

B.

C.

D. All of the above.

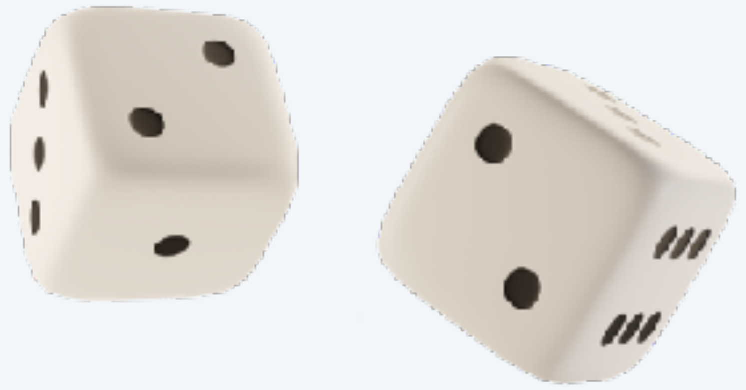
E. Both B and C.

The uniform distribution

Coin flip.

$\mathbb{P}[C \text{ lands heads}] = \mathbb{P}[C \text{ lands tails}] = \frac{1}{2}.$

Roll of a die.



$\mathbb{P}[D \text{ rolls } 1] = \mathbb{P}[D \text{ rolls } 2] = \cdots = \mathbb{P}[D \text{ rolls } 6] = \frac{1}{6}.$

Terminology and notation.

"C lands heads" and "D is even" are **events** with probabilities $\mathbb{P}[C \text{ lands heads}]$, $\mathbb{P}[D \text{ rolls even}]$.

Distribution: all outcome–probability pairs.

outcome	probability
heads	1/2
tails	1/2

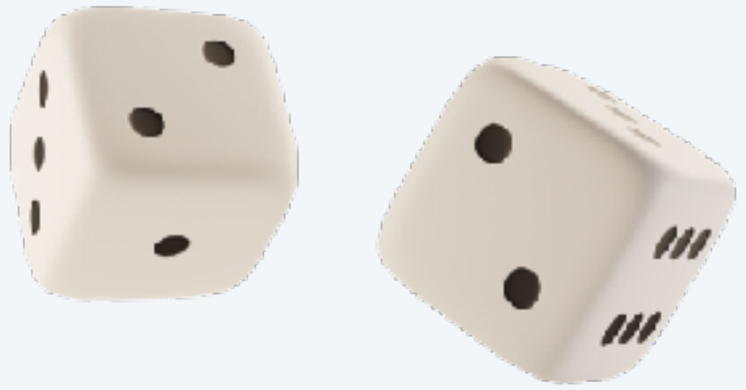
distribution of unbiased coin

The uniform distribution

Coin flip.

$\mathbb{P}[C \text{ lands heads}] = \mathbb{P}[C \text{ lands tails}] = \frac{1}{2}.$ ← *uniform over 2 outcomes*

Roll of a die.



$\mathbb{P}[D \text{ rolls } 1] = \mathbb{P}[D \text{ rolls } 2] = \cdots = \mathbb{P}[D \text{ rolls } 6] = \frac{1}{6}.$ ← *uniform over 6 outcomes*

Terminology and notation.

"C lands heads" and "D is even" are **events** with probabilities $\mathbb{P}[C \text{ lands heads}]$, $\mathbb{P}[D \text{ rolls even}]$.

Distribution: all outcome-probability pairs.

[uniform distribution: all probabilities equal]

outcome	probability
1	1/6
2	1/6
3	1/6
4	1/6
5	1/6
6	1/6

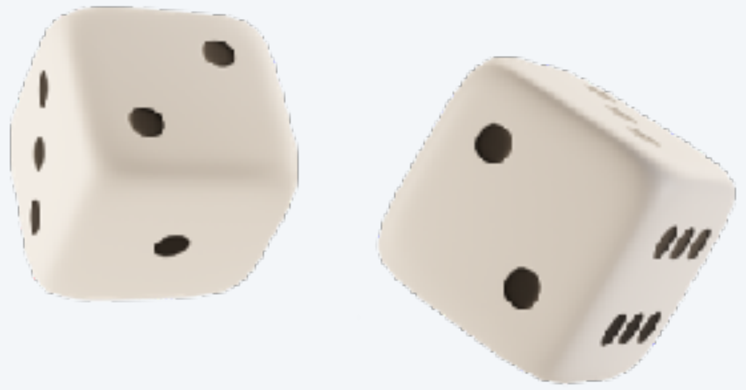
distribution of 6-sided die

The uniform distribution

Coin flip.

$\mathbb{P}[C \text{ lands heads}] = \mathbb{P}[C \text{ lands tails}] = \frac{1}{2}.$ ← *uniform over 2 outcomes*

Roll of a die.



$\mathbb{P}[D \text{ rolls } 1] = \mathbb{P}[D \text{ rolls } 2] = \cdots = \mathbb{P}[D \text{ rolls } 6] = \frac{1}{6}.$ ← *uniform over 6 outcomes*

Independent coin flips.

$\mathbb{P}[C_1 \text{ heads}, C_2 \text{ tails}, \ldots\ C_k \text{ heads}] = \frac{1}{2} \times \frac{1}{2} \cdots \times \frac{1}{2} = \frac{1}{2^k}.$ ← *uniform over 2^k outcomes*

Terminology and notation.

"C lands heads" and "D is even" are **events** with probabilities $\mathbb{P}[C \text{ lands heads}]$, $\mathbb{P}[D \text{ rolls even}]$.

Distribution: all outcome–probability pairs.

[uniform distribution: all probabilities equal]

Randomness: quiz 2

Flip a coin 6 times and count how often it lands heads. Which count is most likely?

A. 2

B. 3

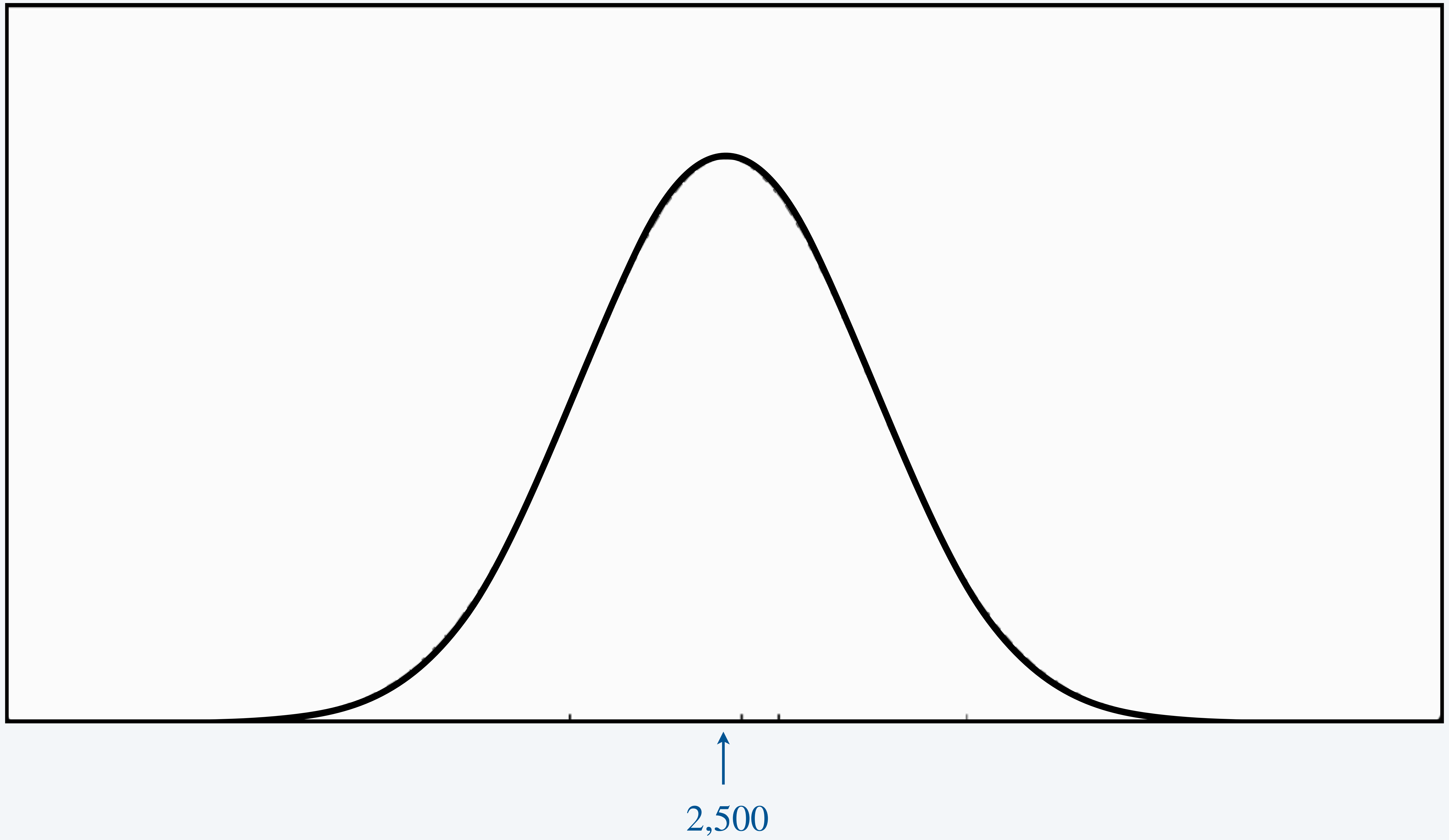
C. 4

D. All of the above.

E. None of the above.

Binomial distribution

Experiment. Flip 5000 coins, count # of heads.



Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

Randomness

- ‣ *what it is and what it isn't*
- ‣ ***Las Vegas and Monte Carlo***
- ‣ *Karger's algorithm*
- ‣ *more applications*

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.

Deterministic algorithms.

- scan the array left-to-right; return once treasure found. ← $\frac{n}{2} + 1$ *accesses in worst case*

A toy problem

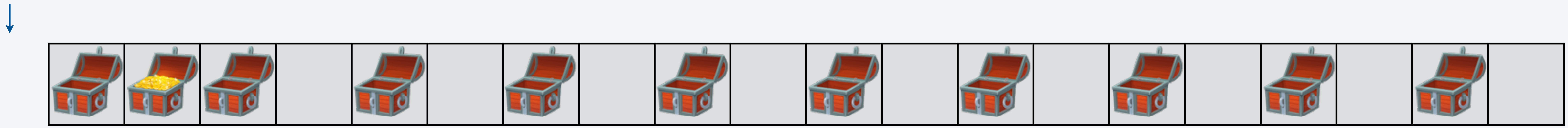
Treasure hunt. Length-n array with 50% treasures, 50% duds.

Deterministic algorithms.

- scan the array left-to-right; return once treasure found. ← $\frac{n}{2} + 1$ *accesses in worst case*
- scan the array right-to-left; return once treasure found. ← $\frac{n}{2} + 1$ *accesses in worst case*

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.



Deterministic algorithms.

- scan the array left-to-right; return once treasure found. ← $\frac{n}{2} + 1$ *accesses in worst case*
- scan the array right-to-left; return once treasure found. ← $\frac{n}{2} + 1$ *accesses in worst case*
- look at even entries, then odd; return once treasure found. ← $\frac{n}{2} + 1$ *accesses in worst case*

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.

Proposition. For every deterministic algorithm, there is a 50%-treasure array where it makes $\frac{n}{2} + 1$ accesses.

Pf. Consider the sequence of $n/2$ accesses it makes when all are duds.
The array with duds there and treasures elsewhere requires $\frac{n}{2} + 1$ accesses.

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.

Randomized algorithms:

- look at `a[StdRandom.uniformInt(n)]`, return treasure (if found).

Fails with probability $\dfrac{n/2}{n} = \dfrac{1}{2}$.

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.

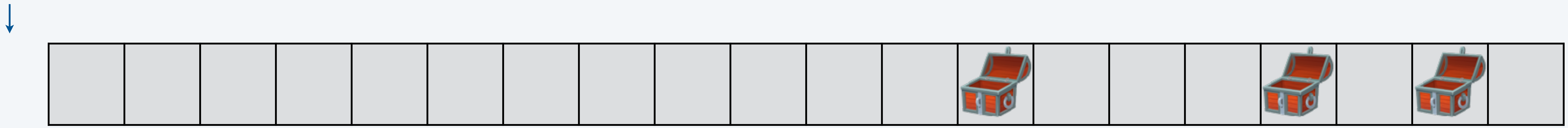
Randomized algorithms:

- look at `a[StdRandom.uniformInt(n)]`, return treasure (if found). ← *1 flip lands tails*
- look at two uniformly random entries, return 1st treasure found (if any).

Fails with probability $\frac{1}{2} \times \frac{1}{2}$.

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.



Randomized algorithms:

- look at `a[StdRandom.uniformInt(n)]`, return treasure (if found). ⟵ *1 flip lands tails*
- look at two uniformly random entries, return 1st treasure found (if any). ⟵ *2 flips land tails*
- look at three uniformly random entries.

Fails with probability $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}$.

A toy problem

Treasure hunt. Length-n array with 50% treasures, 50% duds.

Randomized algorithms:

- look at `a[StdRandom.uniformInt(n)]`, return treasure (if found). ← *1 flip lands tails*
- look at two uniformly random entries, return 1st treasure found (if any). ← *2 flips land tails*
- look at three uniformly random entries, return 1st treasure found (if any). ← *3 flips land tails*
- look at k uniformly random entries, return 1st treasure found (if any).

Fails with probability $\frac{1}{2^k}$.

Randomness: quiz 3

Suppose 1% of the array contains treasure and 99% contain duds. Then `a[StdRandom.uniformInt(n)]` **finds a treasure with probability**

A. 1%

B. 10%

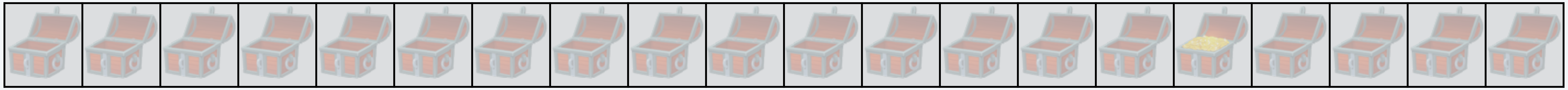
C. 50%

D. 99%

E. None of the above.

Rare treasures and biased coins

Treasure hunt. Length-n array with 1% treasures, 99% duds.



Randomized algorithm:

- look at k uniformly random entries, return treasure (if found).

outcome	probability
heads	1/100
tails	99/100

distribution of 99%-1% biased coin

Failure probability = $\mathbb{P}$[k biased coin flips land tails]

$= (0.99)^k$.

Example. If we want $0.99^k < 1\,\%$, setting $k = 459$ suffices!

Monte Carlo algorithms

Monte Carlo algorithm.

- Running time is deterministic.

  [doesn't depend on coin flips]
- Not guaranteed to be correct.

Error reduction.

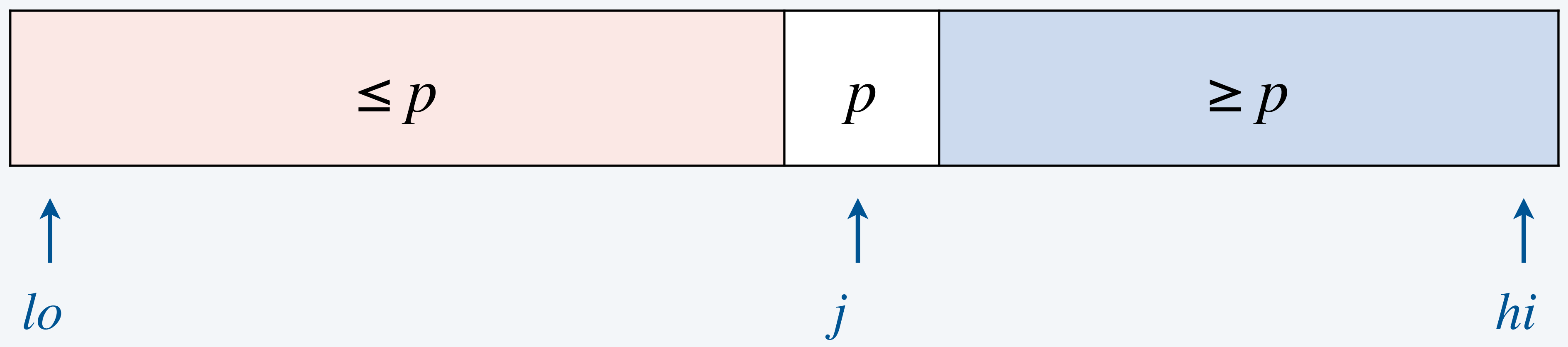
If $\mathbb{P}[A \text{ fails}] = p$ and want failure $\leq q$, repeat $k \geq \log_p q$ times.

Then, $\mathbb{P}[A \text{ fails } k \text{ times}] \underset{\substack{\uparrow \\ \textit{independence}}}{=} p^k \leq q$.

Las Vegas algorithms

- Guaranteed to be correct.
- Running time depends on outcomes of random coin flips.

Ex. Quicksort, quickselect.



Las Vegas vs. Monte Carlo

Treasure hunt. Length-n array with 50% treasures, 50% duds.

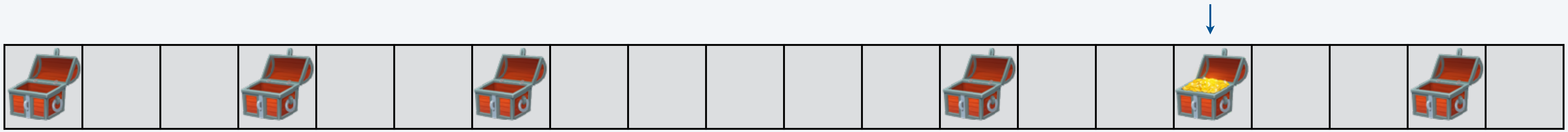
Randomized algorithm (Las Vegas):

- repeatedly look at uniformly random entry; return *only when* treasure found.

Returns in 1st try with probability $1/2$.

Las Vegas vs. Monte Carlo

Treasure hunt. Length-n array with 50% treasures, 50% duds.



Randomized algorithm (Las Vegas):

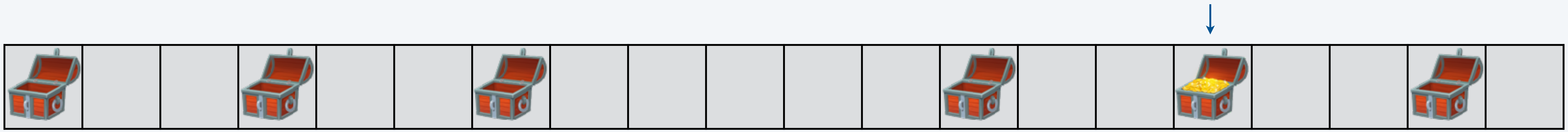
- repeatedly look at uniformly random entry; return *only when* treasure found.

Returns in 1st try with probability $1/2$.

Returns in 2nd try with probability $1/4$.

Las Vegas vs. Monte Carlo

Treasure hunt. Length-n array with 50% treasures, 50% duds.



Randomized algorithm (Las Vegas):

- repeatedly look at uniformly random entry; return *only when* treasure found.

Returns in 1st try with probability $1/2$.

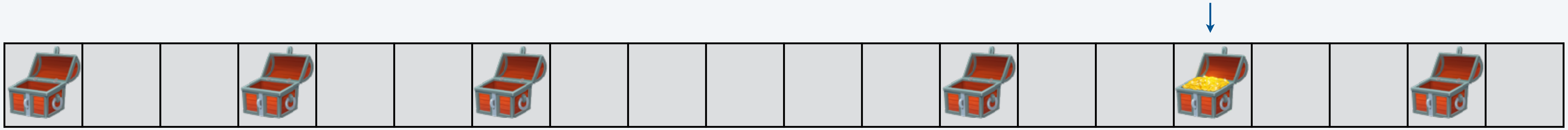
Returns in 2nd try with probability $1/4$.

⋮

Returns in kth try with probability $1/2^k$.

Las Vegas vs. Monte Carlo

Treasure hunt. Length-n array with 50% treasures, 50% duds.



Randomized algorithm (Las Vegas):

- repeatedly look at uniformly random entry; return *only when* treasure found.

Returns in 1st try with probability $1/2$.

Returns in 2nd try with probability $1/4$.

⋮

Returns in kth try with probability $1/2^k$.

same formula as quicksort (for compares)

↓

Expected # of array accesses: $1 \times \mathbb{P}[A \text{ makes } 1 \text{ access}] + 2 \times \mathbb{P}[A \text{ makes } 2 \text{ accesses}] + 3 \times \mathbb{P}[A \text{ makes } 3 \text{ accesses}] + \cdots$

Randomness: quiz 4

At most how many array accesses made by Las Vegas treasure hunt?

A. 1

B. 2

C. $n/2$

D. n

E. None of the above.

https://algs4.cs.princeton.edu

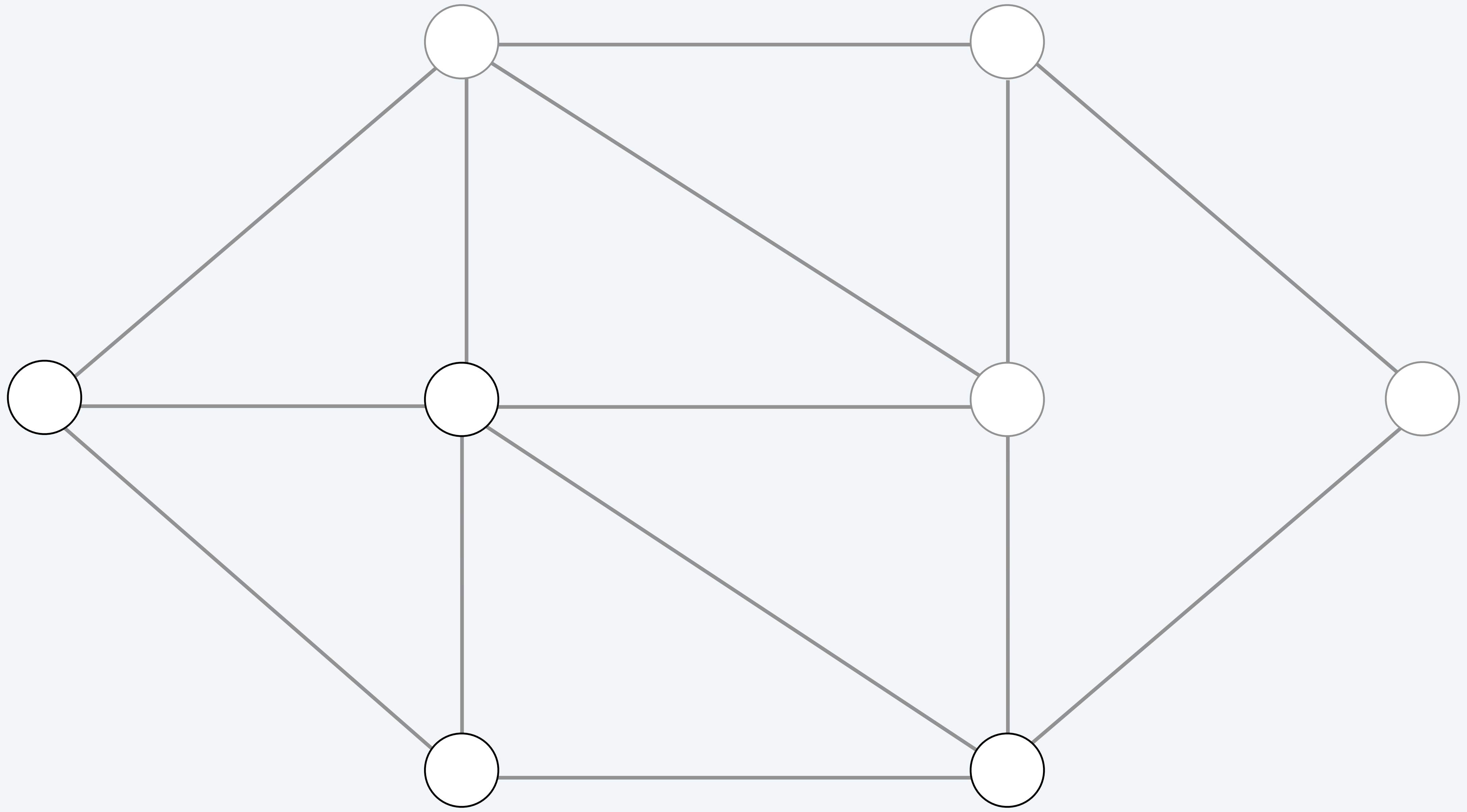
RANDOMNESS

- *what it is and what it isn't*
- *Las Vegas and Monte Carlo*
- *Karger's algorithm*
- *more applications*

Global mincut problem

Goal. Find cut in undirected graph with fewest edges (for any source and sink).

Equivalent. Smallest min st–cut among all pairs (s, t) with antiparallel edges of capacity 1.

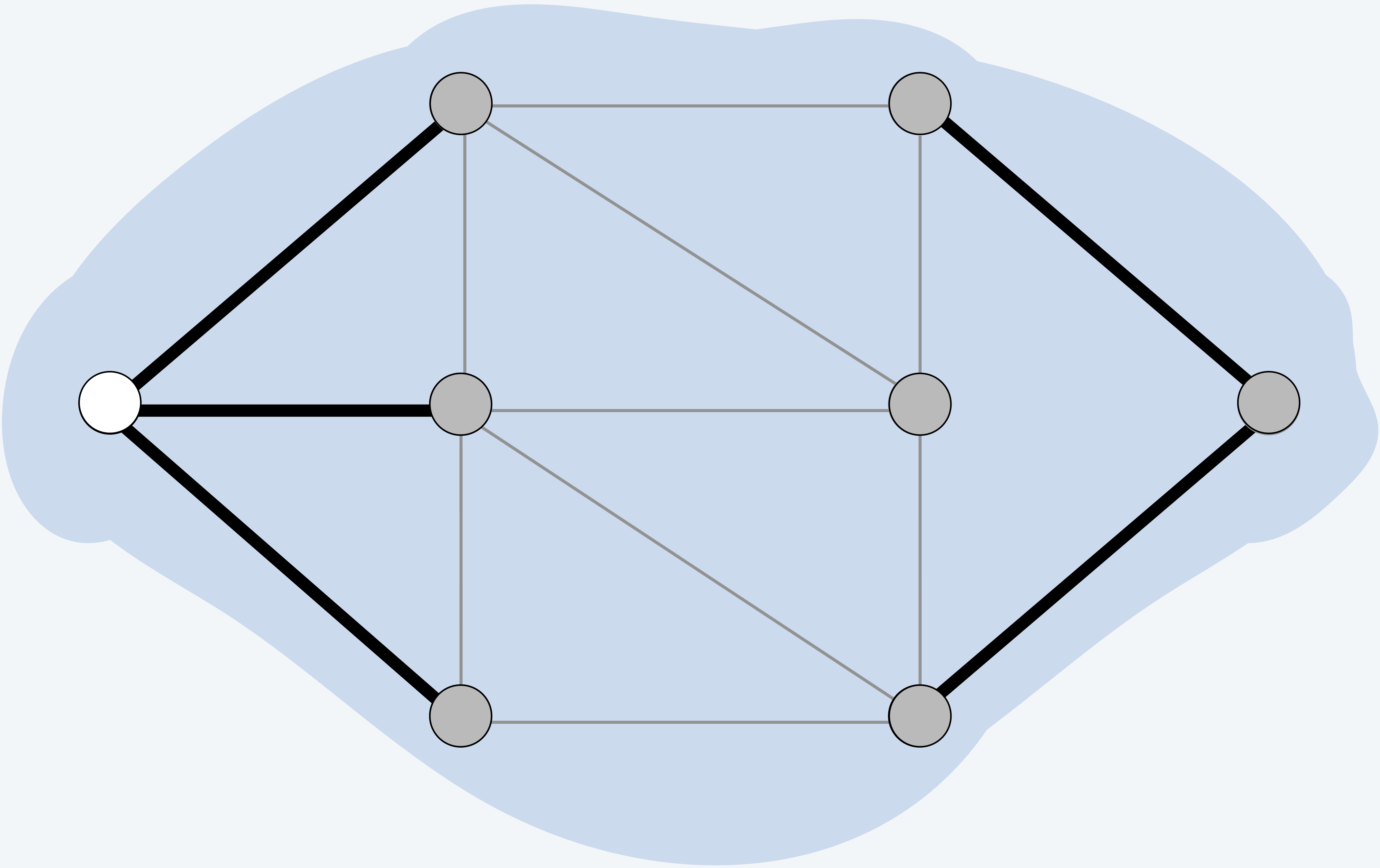


Global mincut problem

Goal. Find cut in undirected graph with fewest edges (for any source and sink).

Deterministic algorithms.

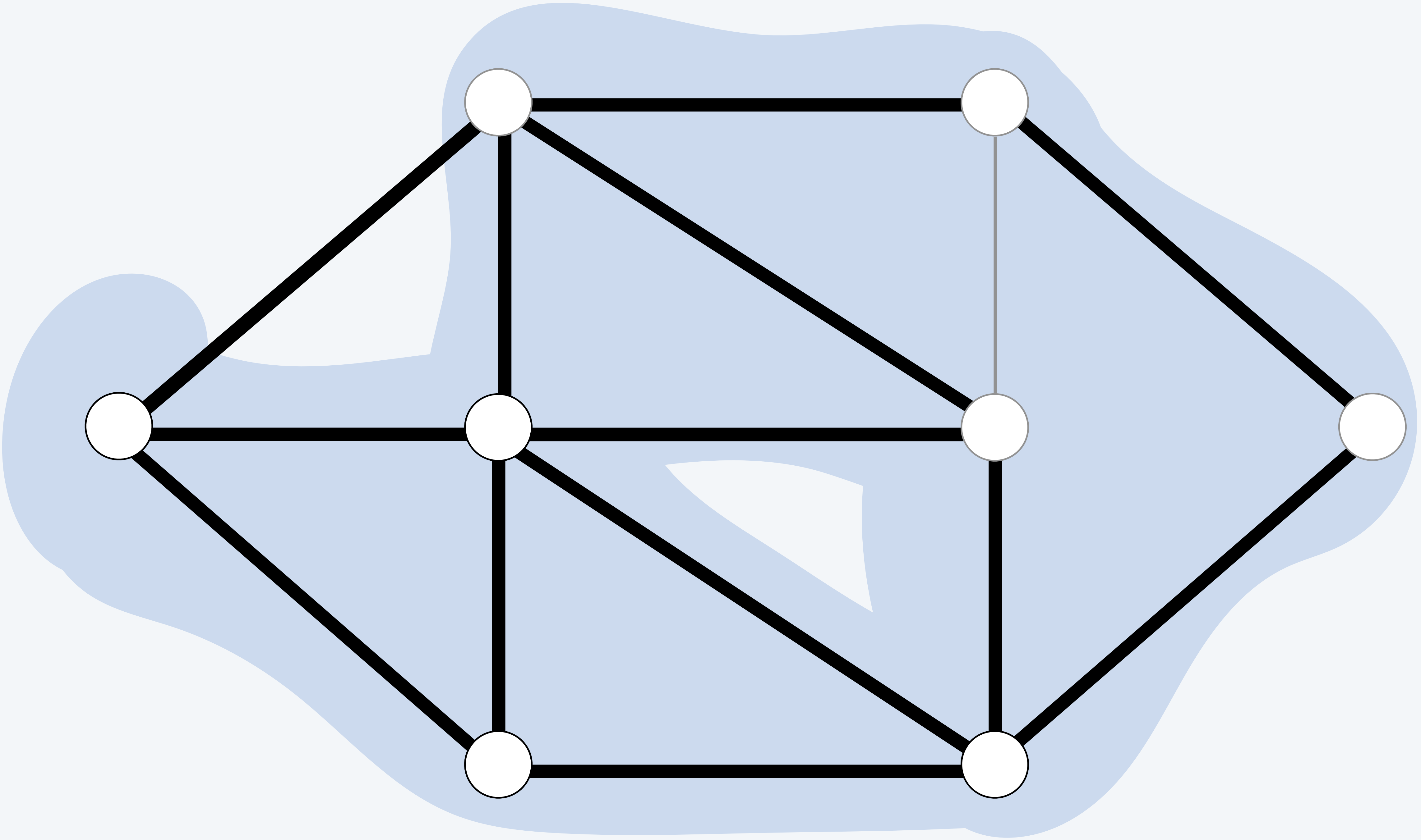
- Brute-force: iterate over all cuts, return smallest. [$2^{V-1} - 1$ cuts $\Longrightarrow$ exponential time!]
- Ford–Fulkerson-based: pick any s as source, try every t as target. [$V - 1$ runs of FF $\Longrightarrow$ $\Theta(VE^2)$ runtime.]



Global mincut problem

Goal. Find cut in undirected graph with fewest edges (for any source and sink).

Idea. Pick a random cut.



Global mincut problem

Uniformly? There may be 1 mincut but $2^{V-1} - 1$ total cuts — takes a *lot* of luck to find it.

Example.

V / 2 vertices on each side

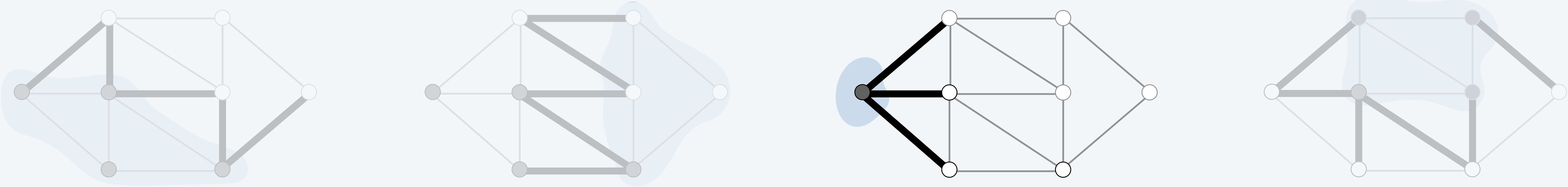
Karger's global mincut algorithm

Algorithm.

- Assign a random weight (uniform between 0 and 1) to each edge e.
- Run Kruskal's MST algorithm until 2 connected components left.
- Return cut defined by connected components.

Probability of finding a mincut: $\approx \dfrac{1}{V^2}$. [no mincut edges in each connected component]

Run algorithm many times and return best cut.



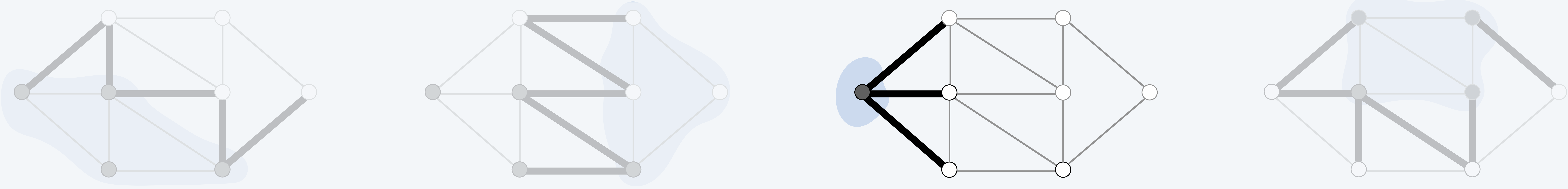
Karger's global mincut algorithm

Algorithm.

- Assign a random weight (uniform between 0 and 1) to each edge e.
- Run Kruskal's MST algorithm until 2 connected components left.
- Return cut defined by connected components.

Probability of finding a mincut: $\approx \frac{1}{V^2}$. [no mincut edges in each connected component]

Run algorithm many times and return best cut.



Remark 1. Finds global mincut in $\Theta(V^2 E \log E)$ time — better than Ford–Fulkerson-based!

Remark 2. With clever idea, improved to $\Theta(V^2 \log^3 V)$ time (still randomized).

Randomness: quiz 5

Smallest # of repetitions of Karger's algorithm to get correct answer with 99% probability?

A. $\Theta(1)$

B. $\Theta(V)$

C. $\Theta(V^2)$

D. $\Theta(V^3)$

E. None of the above.

https://algs4.cs.princeton.edu

RANDOMNESS

- what it is and what it isn't
- Las Vegas and Monte Carlo
- Karger's algorithm
- more applications

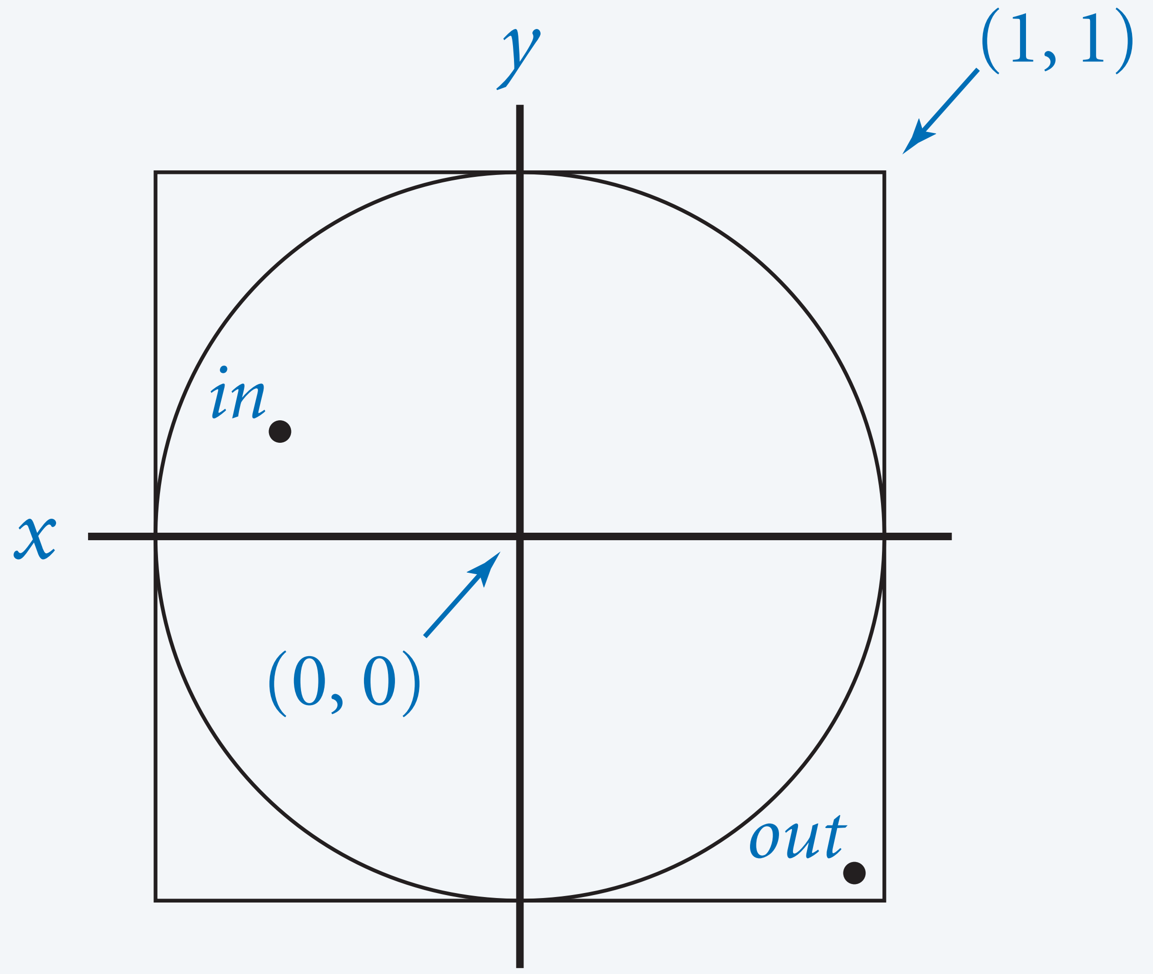
Uniform distribution in unit circle

Goal. Generate a random point in unit circle.

used in Fraud Detection!

Rejection sampling.

- Generate a random point in 2-by-2 square centered at origin.
- If point is inside circle, use that point; otherwise, repeat.



```
double x, y;
do {
   x = 2.0 * Math.random() - 1.0;
   y = 2.0 * Math.random() - 1.0;
} while (x*x + y*y > 1.0);
StdOut.println("(" + x + ", " + y + ")");
```

random (x, y) in square

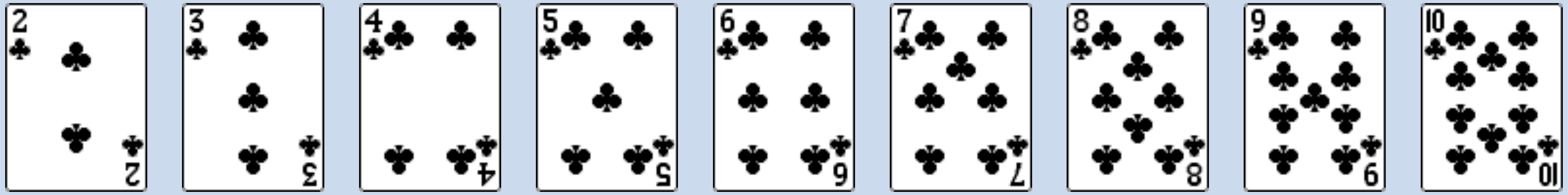
repeat until it's in the circle

Remark. If s out of t samples in unit circle, $\frac{s}{t} \approx \frac{\pi}{4}$.

Interview question: shuffle an array

Goal. Rearrange array so that result is a uniformly random permutation.

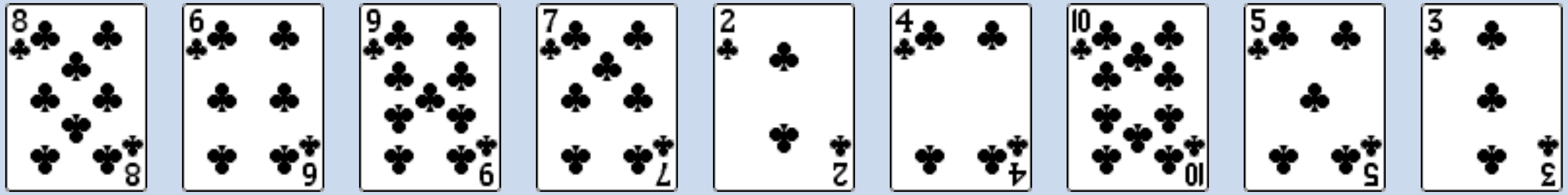
all $n!$ permutations equally likely



Interview question: shuffle an array

Goal. Rearrange array so that result is a uniformly random permutation.

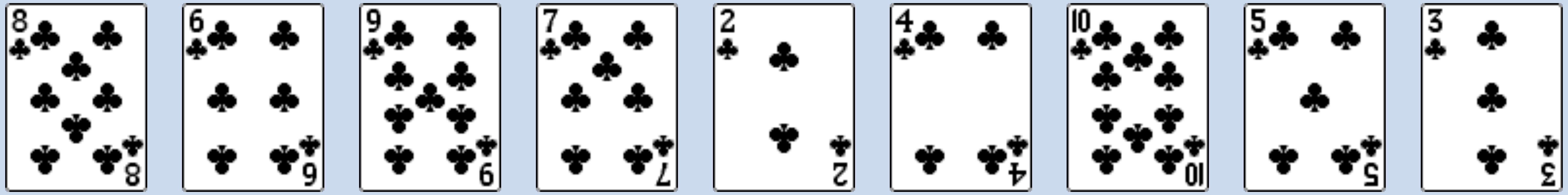
all n! permutations equally likely



Interview question: shuffle an array

Goal. Rearrange array so that result is a uniformly random permutation.

all n! permutations equally likely



Challenge. Design in-place linear-time algorithm using `StdRandom.uniformInt()`.

Randomness: quiz 5

Which of the following generate a uniformly random permutation of array `a[]`?

A. `StdRandom.shuffle(a);`

B.
```
for (int i = 0; i < a.length; i++)
   exch(a, i, StdRandom.uniformInt(a.length));
```

C.
```
for (int i = a.length - 1; i > 0; i--)
   exch(a, i, StdRandom.uniformInt(i + 1));
```

D. A and C.

E. All of the above.

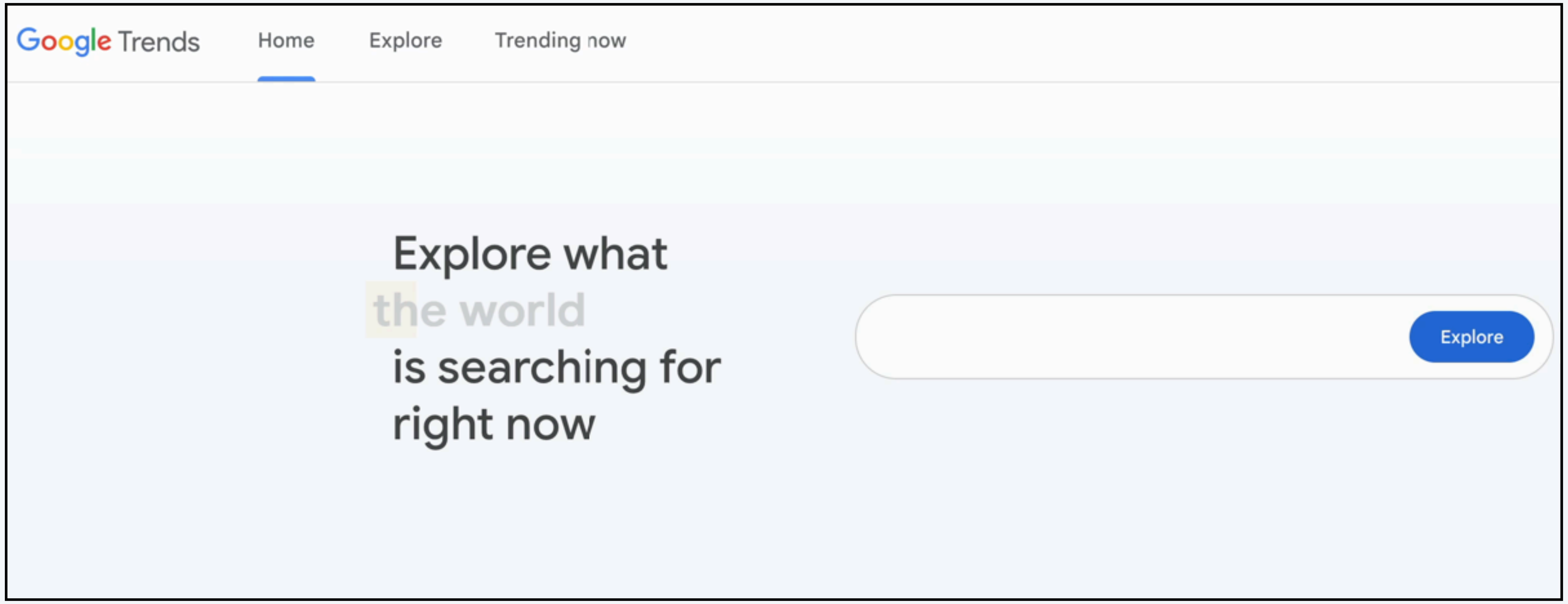
Approximate counting

Goal. Count to $\leq n$ with less memory: from $\log_2 n$ to $\Theta(\log \log n)$.

Why bother?

Database with 1 billion entries: $\log_2(10^9) \approx 30$ bits, but $\log_2 \log_2(10^9) \approx 5$ bits.

Factor-6 improvement matters *a lot*.



Explore what
the world
is searching for
right now

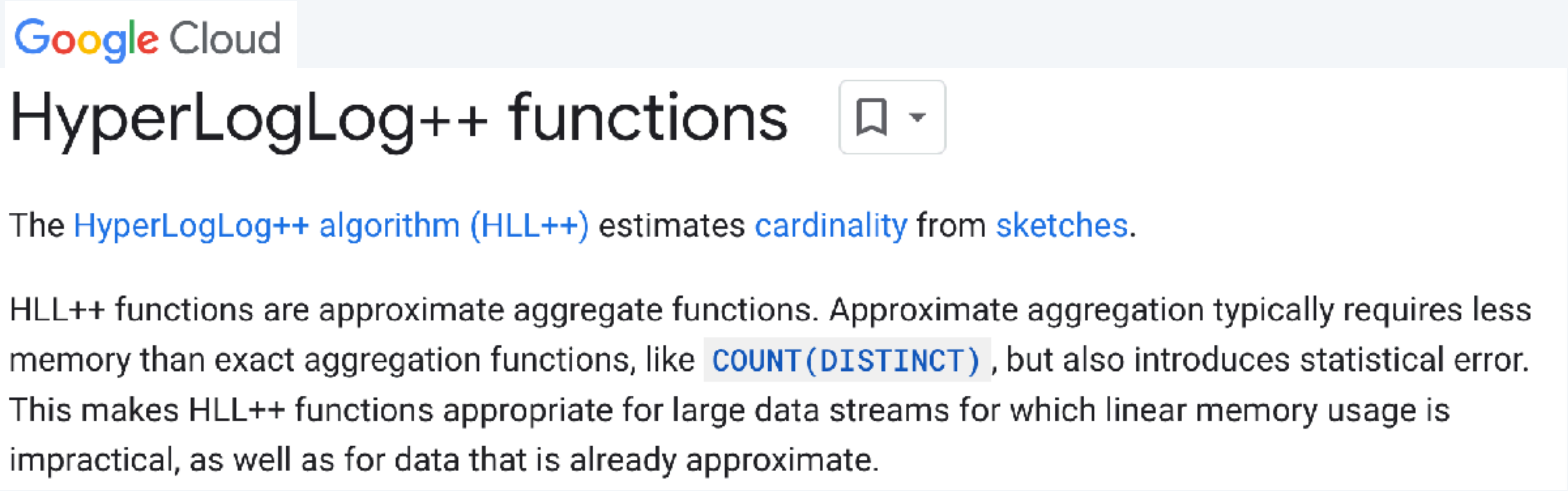
Approximate counting

Goal. Count to $\leq n$ with less memory: from $\log_2 n$ to $\Theta(\log \log n)$.

Why bother?

Database with 1 billion entries: $\log_2(10^9) \approx 30$ bits, but $\log_2 \log_2(10^9) \approx 5$ bits.

Factor-6 improvement matters *a lot*.



Google Cloud

HyperLogLog++ functions

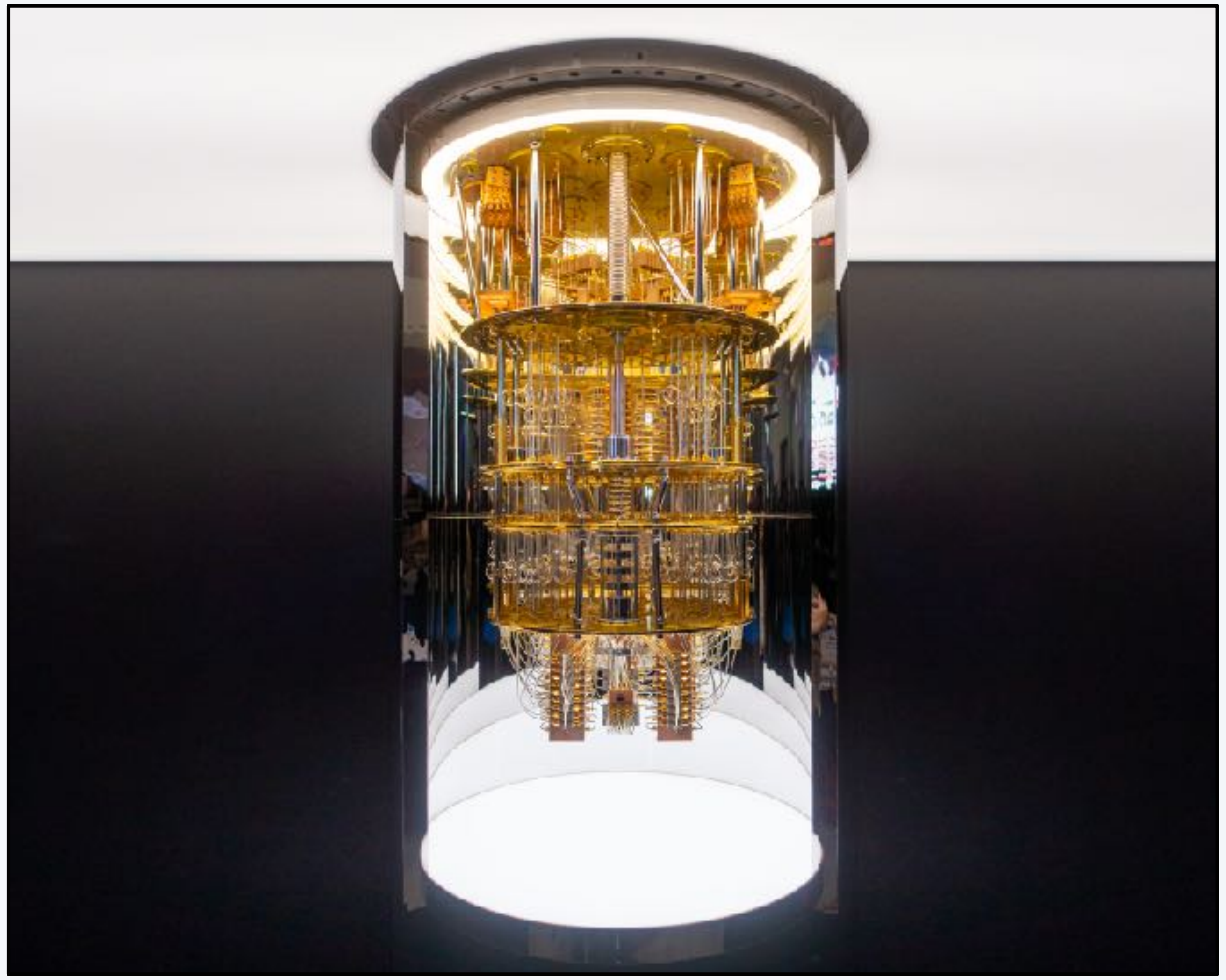
The HyperLogLog++ algorithm (HLL++) estimates cardinality from sketches.

HLL++ functions are approximate aggregate functions. Approximate aggregation typically requires less memory than exact aggregation functions, like `COUNT(DISTINCT)`, but also introduces statistical error. This makes HLL++ functions appropriate for large data streams for which linear memory usage is impractical, as well as for data that is already approximate.

https://cloud.google.com/bigquery/docs/reference/standard-sql/hll_functions

Beyond this course

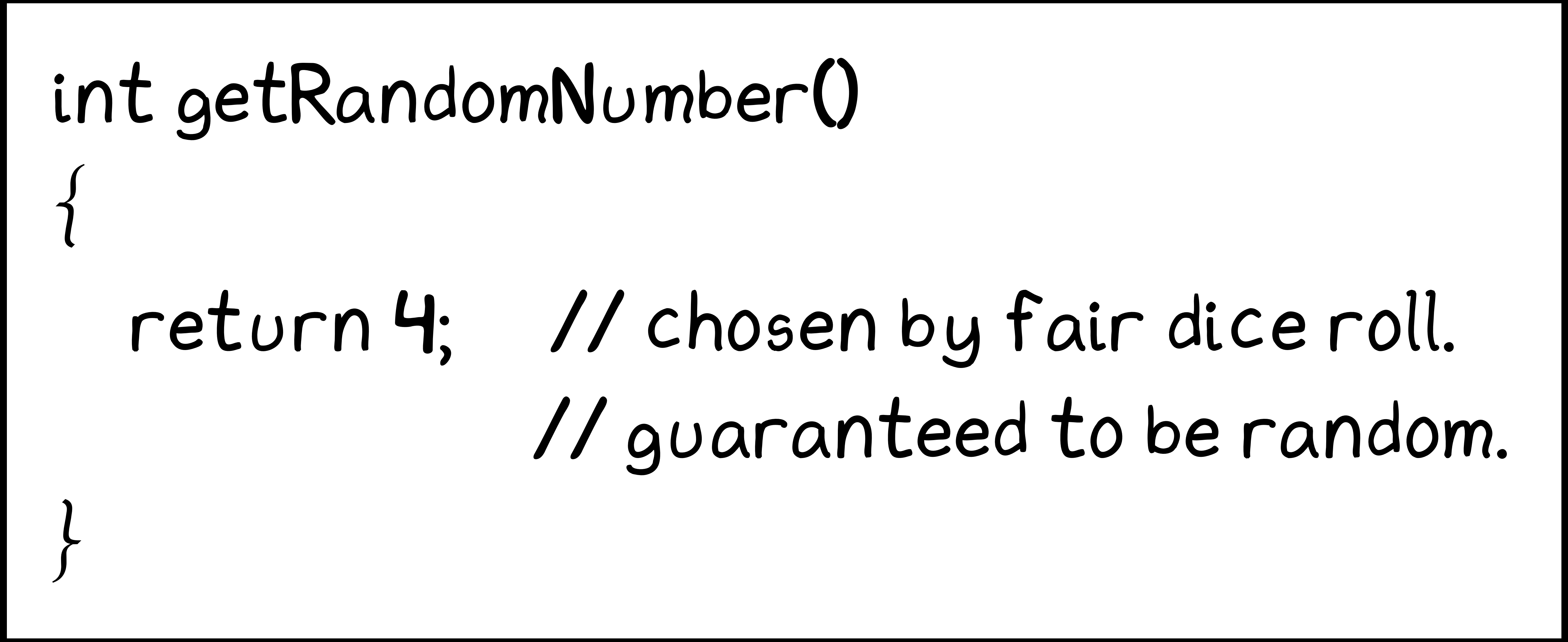
- Approximation algorithms [intractability: stay tuned!]
- Machine learning [randomized MW]
- Optimization [stochastic gradient descent]
- Cryptography [average-case hardness]
- Complexity theory [derandomization]
- Quantum computation [Shor's factoring algorithm]
- Networking [load balancing]
- Graphics [procedural generation]
- Mathematics [probabilistic method]
- Health sciences [randomized control trials]



IBM Quantum System One

ORF 309. Probability and Stochastic Systems

COS 433. Cryptography



```
int getRandomNumber()
{
   return 4;    // chosen by fair dice roll.
                // guaranteed to be random.
}
```

https://xkcd.com/221/

Credits

image	source	license
Quarter	Adobe Stock	Education License
6-sided dice	Adobe Stock	Education License
20-sided die	Adobe Stock	Education License
Lava lamps	Fast Company	
Coin Toss	clipground.com	CC BY 4.0
IDQ Quantum Key Factory	idquantique.com	
SG100	protego.bytehost16.com	
Las Vegas	Adobe Stock	Education License
Monte Carlo	Adobe Stock	Education License
Treasure chests	Adobe Stock	Education License
Random number generator	XKCD	CC BY-NC 2.5