# Chapter 7: Randomized Algorithms

## Advanced Algorithms

Randomness and expected time | Las Vegas and Monte Carlo | Fisher–Yates shuffle | BogoSort | Randomized quicksort | Monte Carlo estimate of π | Miller–Rabin | Karger's min cut

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Chapter_07_Randomized_Algorithms.html

# Learning Outcomes

### Concepts

Define a randomized algorithm and its expected running time; distinguish Las Vegas from Monte Carlo algorithms.

### Shuffling and BogoSort

Write the Fisher–Yates shuffle, show why the naive shuffle is biased, and derive BogoSort's expected $\Theta(n \cdot n!)$ time.

### Randomized quicksort

Trace random-pivot partitioning step by step; justify the expected $O(n \log n)$ and worst-case $O(n^2)$ bounds.

### Monte Carlo

Compute the error probability of $\pi$ estimation, the Miller–Rabin primality test and Karger's min-cut algorithm, and reduce it by repetition.

# Algorithms That Gamble

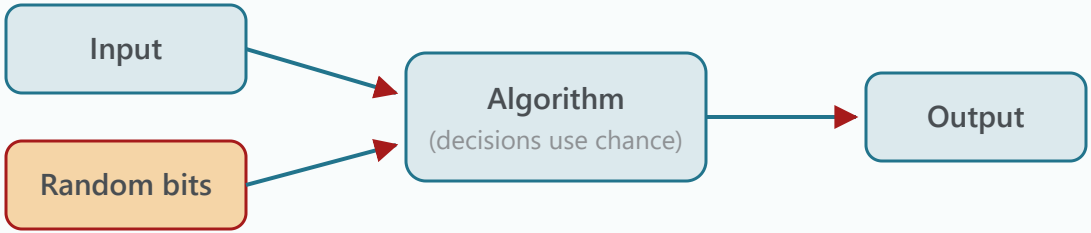
- Some problem-solving techniques resemble **games of chance**.
- The algorithm uses **randomness** when making decisions: it flips a coin, rolls a die, or picks a random element.
- Example game: roll two dice; the player wins if the sum is **7 or 11**.




win: 7 or 11

P(7) = 6/36, P(11) = 2/36 → P(win) = 8/36 = **2/9 ≈ 0.22**. Expected number of rolls until a win = 1 / (2/9) = **4.5**.

# Randomized Algorithms




- A **randomized algorithm** uses **random numbers** in addition to its input.
- Two runs on the same input may take **different steps**: the running time (and sometimes the output) can vary from run to run.
- The running time is now a **random variable**, so we study the **expected time** E[T(n)]. The expectation is over the algorithm's *own* coin flips, not over the input.

# Why Randomness?

### Protection against bad inputs

Deterministic quicksort takes $O(n^2)$ on a sorted array. With a random pivot, **no input** is always bad: the worst case arises only from unlucky coin flips.

### Simplicity

A randomized algorithm is often much shorter and clearer than the best deterministic one (Karger's min cut, randomized quickselect).

### Speed

Miller–Rabin decides whether a number with hundreds of digits is prime in milliseconds, with a tiny error probability.

### Sampling and symmetry breaking

Monte Carlo simulation, random sampling and leader election in distributed systems are hard without randomness.

The price: time or correctness is guaranteed only **probabilistically**, and we need a good random number generator.

# Probability Review

| Concept | Definition / rule | Example |
|---|---|---|
| **Expected value** | $E[X] = \Sigma\, x \cdot P(X = x)$ | expected value of a die: 3.5 |
| **Linearity** | $E[X + Y] = E[X] + E[Y]$ (no independence needed) | sum of two dice: 7 |
| **Indicator variable** | $X_A = 1$ (if A occurs), else 0 → $E[X_A] = P(A)$ | Quicksort: "were i and j compared?" |
| **Geometric distribution** | trials with success probability p: expected trials until the first success **1/p** | BogoSort: $p = 1/n!$ → $n!$ shuffles |
| **Independent repetition** | k independent trials, each failing with probability p: all fail with $\mathbf{p^k}$ | Miller–Rabin: $(1/4)^k$ |
| **Markov's inequality** | if $X \geq 0$, $P(X \geq c \cdot E[X]) \leq 1/c$ | Las Vegas → Monte Carlo |

# Las Vegas and Monte Carlo

| | Las Vegas | Monte Carlo |
|---|---|---|
| **Correctness** | **Always correct** result | **Wrong** result with small probability |
| **Time** | Random; we analyze the expected time | Bounded (deterministic upper bound) |
| **What is random** | how long it runs | whether the answer is correct |
| **Examples** | Randomized quicksort, quickselect, BogoSort, Fisher–Yates | Miller–Rabin, Karger's min cut, π estimation |

### One-sided error

When Miller–Rabin says "composite", it is **certainly** right; only the answer "prime" can be wrong.

### Two-sided error

Numerical simulations such as π estimation can err in either direction; the error shrinks with the number of samples.

# Reducing the Error Probability

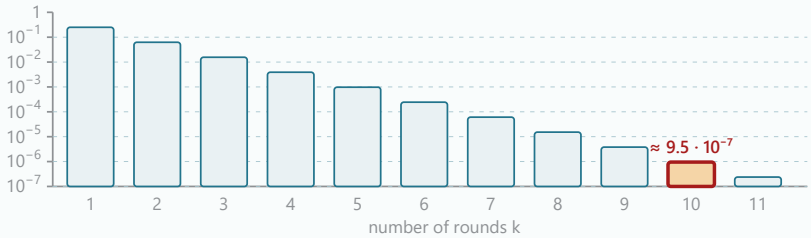
### Amplification by repetition

If one round errs with probability at most q, then after k **independent** rounds the error is $\le q^k$.

- Miller–Rabin: $q \le 1/4$ → with $k = 20$ rounds, $\le 4^{-20} \approx 9 \cdot 10^{-13}$.
- Karger: success $\ge 2/(n(n-1))$; with $n(n-1)/2 \cdot \ln n$ repetitions, failure $\le 1/n$.

### Converting between types

- **Las Vegas → Monte Carlo:** stop the algorithm after $2 \cdot E[T]$ steps. Markov: $P(T \ge 2\,E[T]) \le 1/2$ → "no answer" with probability at most 1/2.
- **Monte Carlo → Las Vegas:** if an answer can be **verified** quickly (e.g., multiply the factors found), repeat until it checks out. Expected repetitions = $1/p$.




Miller–Rabin error bound $4^{-k}$ (log scale): each round divides the bound by 4. $k = 10$ rounds suffice to get below $10^{-6}$.

# Example: Shuffling

- Randomness is used to **shuffle** the elements of a list: a deck of cards, random test data, a playlist.
- **Correctness:** every element lands in every position with **equal probability**; more strongly, each of the n! permutations must have probability **1/n!**.
- **Running time:** Fisher–Yates always performs exactly n − 1 swaps: **Θ(n)**. What varies is the *result*, not the time.
- Built into Java: `Collections.shuffle(list)` (Fisher–Yates).

| n | n! permutations |
|---|---|
| 3 | 6 |
| 4 | 24 |
| 10 | 3,628,800 |
| 52 | $\approx 8.07 \cdot 10^{67}$ |

# Fisher–Yates (Knuth) Shuffle

```
1  static void shuffle(int[] a) {
2    for (int i = a.length - 1; i > 0; i--) {
3      int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4      int t = a[i]; a[i] = a[j]; a[j] = t;
5    }
6  }
```




Start: `a = [1, 2, 3, 4, 5]`. Working from the end, each position receives a random element from those not yet placed.

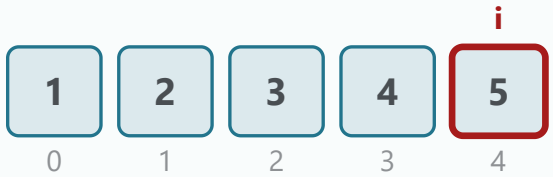
Step 1 / 14

VARIABLES

`a = [1, 2, 3, 4, 5]` `i = –` `j = –`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`i = 4` : one of the 5 elements in a[0..4] will move to a[4].

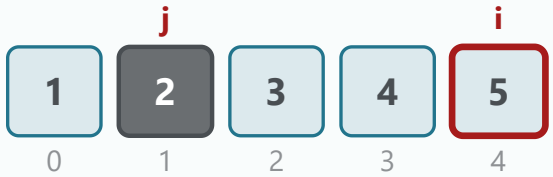
Step 2 / 14

VARIABLES

`a = [1, 2, 3, 4, 5]` `i = 4` `j = –`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`rng.nextInt(5)` → `j = 1` (probability 1/5).

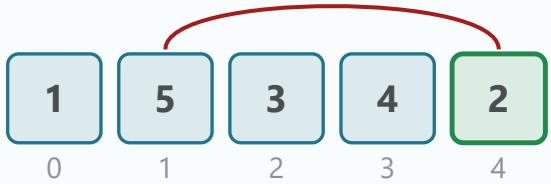
Step 3 / 14

VARIABLES

`a = [1, 2, 3, 4, 5]` `i = 4` `j = 1`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




Swap a[4] ↔ a[1] → `[1, 5, 3, 4, 2]`. a[4] = 2 is now **final** (green).

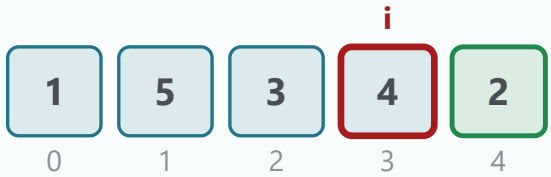
Step 4 / 14

VARIABLES

`a = [1, 5, 3, 4, 2]` `i = 4` `j = 1`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`i = 3` : one of the 4 elements in a[0..3] will move to a[3].

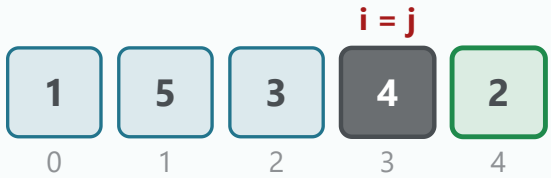
Step 5 / 14

VARIABLES

`a = [1, 5, 3, 4, 2]` `i = 3` `j = –`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`rng.nextInt(4)` → `j = 3` (j = i: the element stays put).

Step 6 / 14

VARIABLES

`a = [1, 5, 3, 4, 2]` `i = 3` `j = 3`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




Swap with itself: the array is unchanged; a[3] = 4 is final.

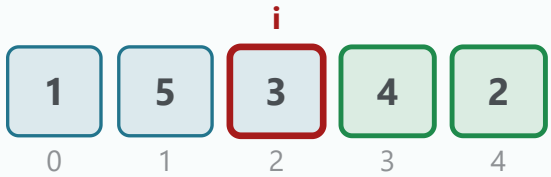
Step 7 / 14

VARIABLES

`a = [1, 5, 3, 4, 2]` `i = 3` `j = 3`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
▶   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`i = 2` : one of the 3 elements in a[0..2] will move to a[2].

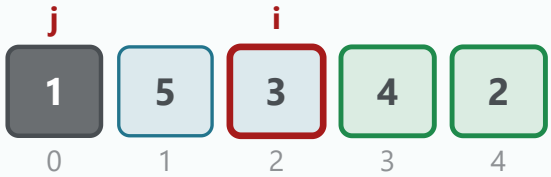
Step 8 / 14

VARIABLES

`a = [1, 5, 3, 4, 2]` `i = 2` `j = –`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`rng.nextInt(3)` → `j = 0` (probability 1/3).

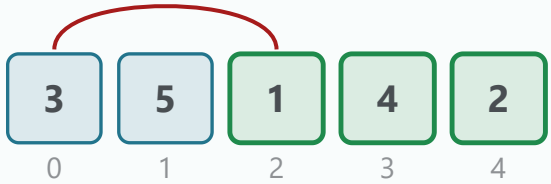
Step 9 / 14

VARIABLES

`a = [1, 5, 3, 4, 2]` `i = 2` `j = 0`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




Swap a[2] ↔ a[0] → `[3, 5, 1, 4, 2]`. a[2] = 1 is now **final** (green).

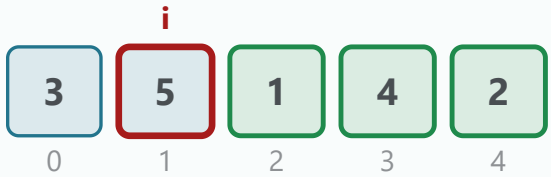
Step 10 / 14

VARIABLES

`a = [3, 5, 1, 4, 2]` `i = 2` `j = 0`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
▶   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`i = 1` : one of the 2 elements in a[0..1] will move to a[1].

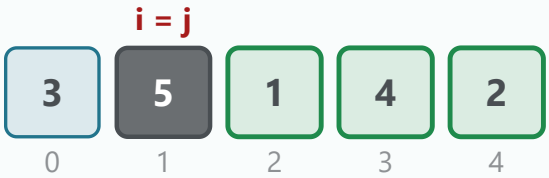
Step 11 / 14

VARIABLES

`a = [3, 5, 1, 4, 2]` `i = 1` `j = –`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
▶     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




`rng.nextInt(2)` → `j = 1` (j = i: the element stays put).

Step 12 / 14

VARIABLES

`a = [3, 5, 1, 4, 2]` `i = 1` `j = 1`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
▶     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
6 }
```




Swap with itself: the array is unchanged; a[1] = 5 is final.

Step 13 / 14

VARIABLES

`a = [3, 5, 1, 4, 2]` `i = 1` `j = 1`

# Fisher–Yates (Knuth) Shuffle

```
1 static void shuffle(int[] a) {
2   for (int i = a.length - 1; i > 0; i--) {
3     int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
4     int t = a[i]; a[i] = a[j]; a[j] = t;
5   }
▶ }
```




Loop done: `[3, 5, 1, 4, 2]`. The number of possible draw sequences is 5 · 4 · 3 · 2 = 120 = 5!: each permutation arises **exactly once** → uniform distribution.

Step 14 / 14

VARIABLES

`a = [3, 5, 1, 4, 2]` `i = 0` `j = –`

# The Naive Shuffle Is Biased

```
// BIASED: for each i, pick from 0..n-1
for (int i = 0; i < n; i++) {
  int j = rng.nextInt(n);
  swap(a, i, j);
}
```

- Each of the n draws takes n values → $\mathbf{n^n}$ equally likely paths.
- n = 3: $3^3$ = 27 paths, 3! = 6 permutations. 27 is not divisible by 6 → the permutations **cannot be equally likely**.
- The fix for this common pattern is Fisher–Yates: `nextInt(i + 1)`.

| result | paths | probability | should be |
|---|---|---|---|
| `123` | 4/27 | 0.148 | 1/6 ≈ 0.167 |
| `132` | 5/27 | 0.185 | 1/6 ≈ 0.167 |
| `213` | 5/27 | 0.185 | 1/6 ≈ 0.167 |
| `231` | 5/27 | 0.185 | 1/6 ≈ 0.167 |
| `312` | 4/27 | 0.148 | 1/6 ≈ 0.167 |
| `321` | 4/27 | 0.148 | 1/6 ≈ 0.167 |

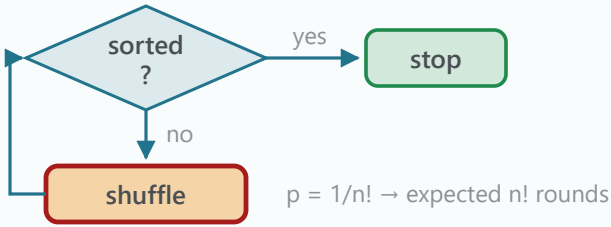
All 27 draw sequences for [1, 2, 3] were enumerated (Fisher–Yates has 3 · 2 = 6 paths, one per permutation).

# BogoSort

- One of the **worst-performing** sorting algorithms ("stupid sort", permutation sort).
- Basic idea: **shuffle randomly** until the list is sorted.
- A fun and instructive **Las Vegas** algorithm; never used in practice.

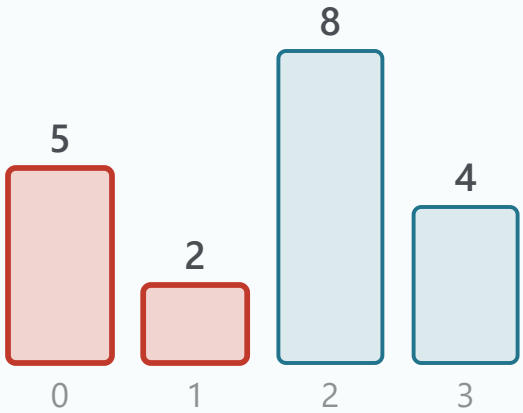
1. Is the list sorted? **Yes** → stop.
2. **No** → shuffle the list randomly and go back to step 1.

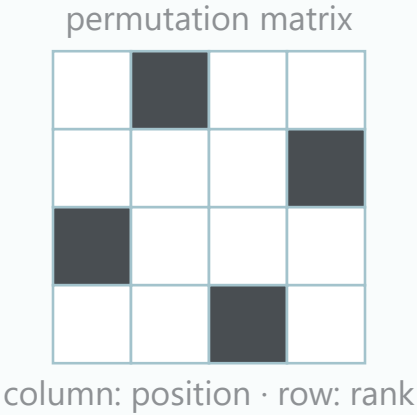
```
static void bogoSort(int[] a) {
  while (!isSorted(a))
    shuffle(a);         // Fisher–Yates
}
static boolean isSorted(int[] a) {
  for (int i = 0; i + 1 < a.length; i++)
    if (a[i] > a[i + 1]) return false;
  return true;
}
```



# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

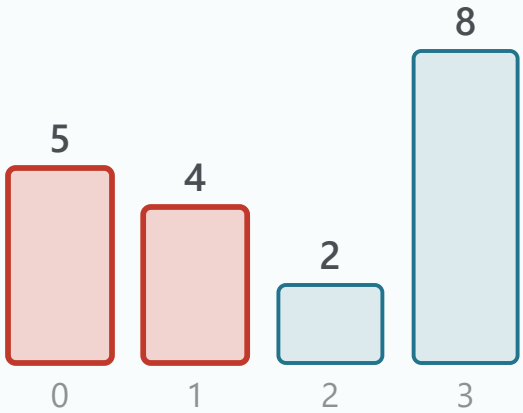


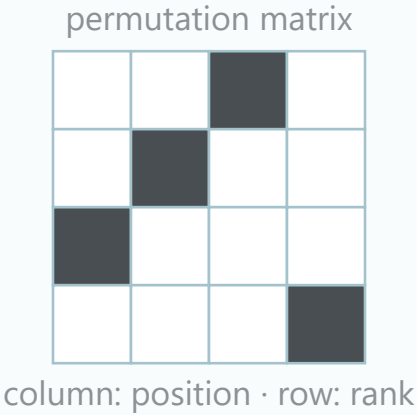




Start `[5, 2, 8, 4]` : a[0] = 5 > a[1] = 2 → **not sorted**, shuffling begins.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

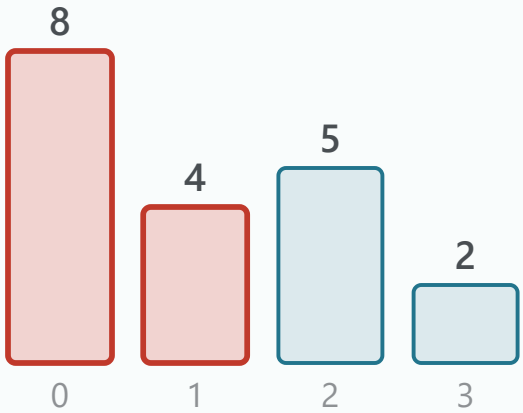


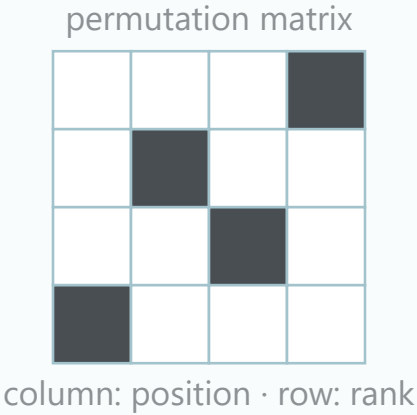




Shuffle 1: `[5, 4, 2, 8]` → a[0] = 5 > a[1] = 4, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

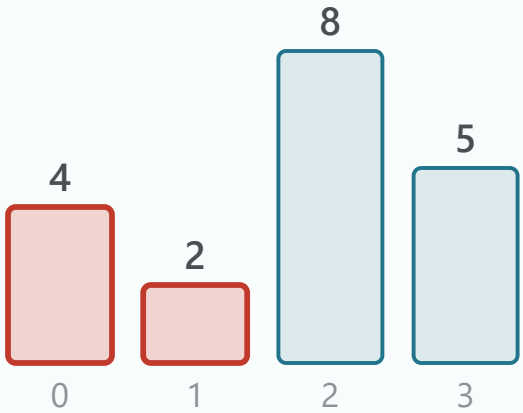


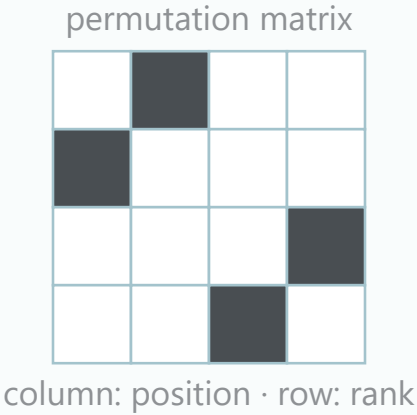




Shuffle 2: `[8, 4, 5, 2]` → a[0] = 8 > a[1] = 4, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

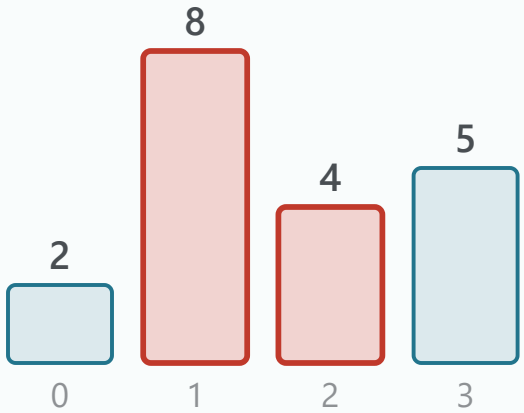


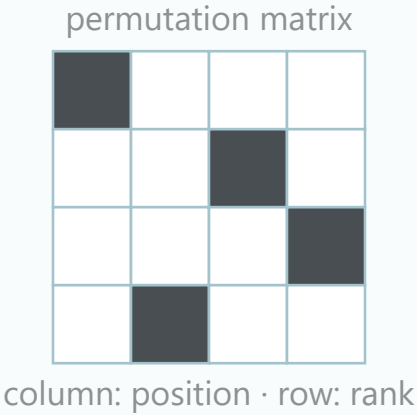




Shuffle 3: `[4, 2, 8, 5]` → a[0] = 4 > a[1] = 2, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

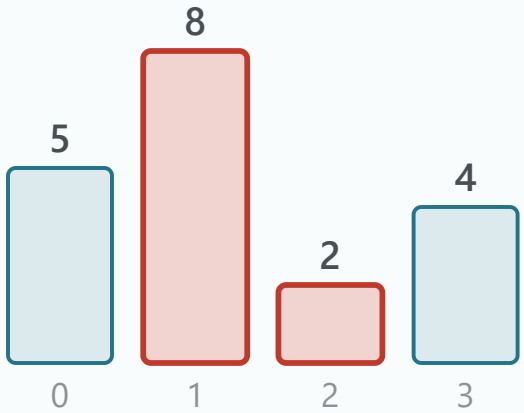


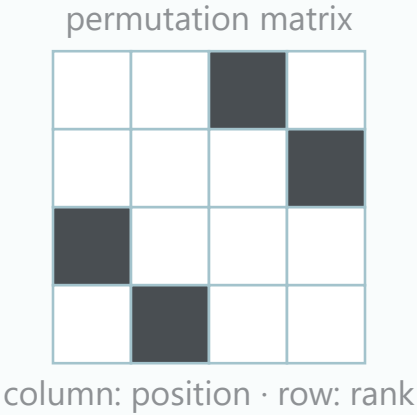




Shuffle 4: `[2, 8, 4, 5]` → a[1] = 8 > a[2] = 4, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

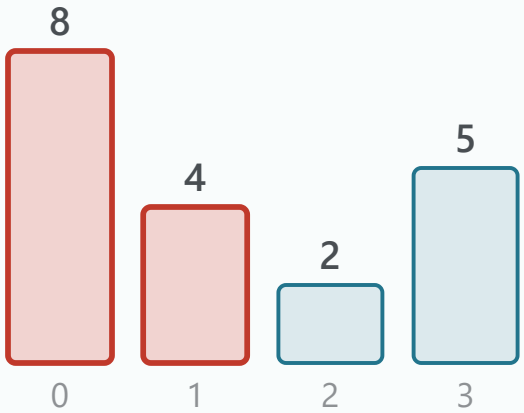


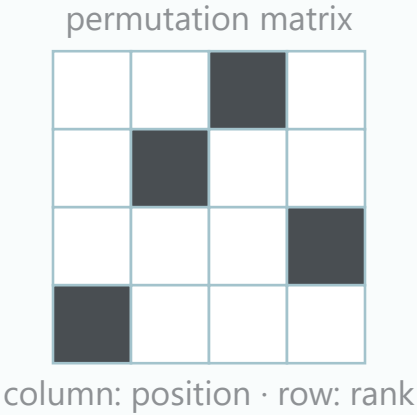




Shuffle 5: `[5, 8, 2, 4]` → a[1] = 8 > a[2] = 2, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.

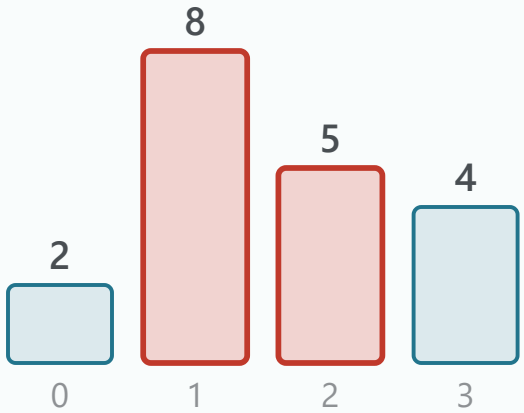


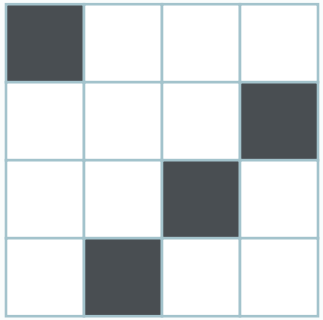

Shuffle 6: `[8, 4, 2, 5]` → a[0] = 8 > a[1] = 4, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.




permutation matrix

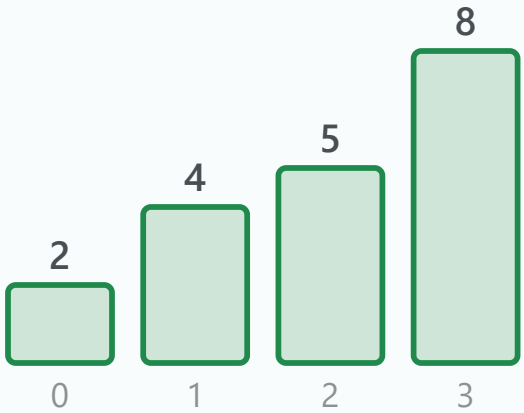


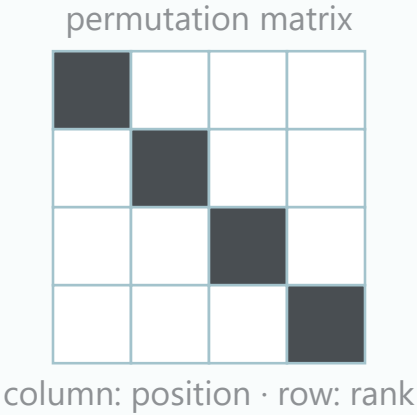
column: position · row: rank

Shuffle 7: `[2, 8, 5, 4]` → a[1] = 8 > a[2] = 5, not sorted.

# BogoSort Example: [5, 2, 8, 4]

The list [5, 2, 8, 4]; each shuffle uses Fisher–Yates (a seeded run). Red: the first adjacent pair out of order.




Shuffle 8: `[2, 4, 5, 8]` → **sorted!** The algorithm stops. This run needed 8 shuffles; the expected number is 4! = 24.

# Analysis of BogoSort

- With n distinct elements, a random shuffle is sorted with probability **p = 1/n!**.
- Trials are independent → geometric distribution: the expected number of shuffles is **1/p = n!**.
- Each round: shuffle O(n) + check O(n) → **expected time Θ(n · n!)**. This is the same order as O((n + 1)!), since (n + 1)! = (n + 1) · n!.
- **Best case**: the array is already sorted → one check, O(n).
- **Worst case: unbounded.** Unlucky shuffles can go on forever (with probability tending to 0). O((n + 1)!) bounds only the *expected* time, not the worst case.

| n | expected shuffles n! | time ($10^9$ shuffles/s) |
|---|---|---|
| 5 | 120 | 0.12 µs |
| 10 | 3,628,800 | 3.6 ms |
| 12 | 479,001,600 | 0.48 s |
| 13 | 6,227,020,800 | 6.2 s |
| 15 | 1,307,674,368,000 | ≈ 22 minutes |
| 20 | 2,432,902,008,176,640,000 | ≈ 77 years |

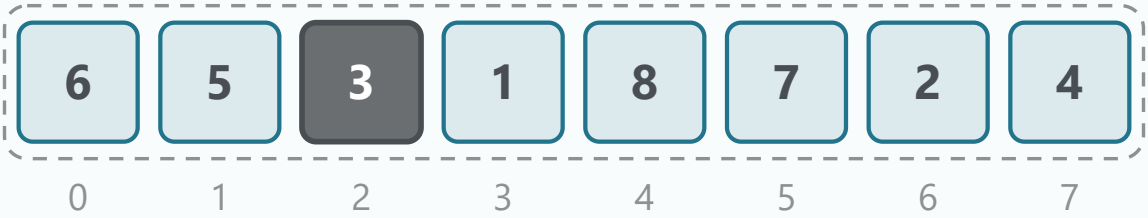
# Pivot Choice and Its Effect

| Pivot strategy | Bad input | Expected / typical | Worst |
|---|---|---|---|
| **First or last element** | sorted / reverse-sorted array (very common!) | O(n log n)* | $O(n^2)$ |
| **Median-of-3** | specially crafted "killer" arrays | O(n log n)* | $O(n^2)$ |
| **Random element** | **none**: bad runs come only from bad luck | O(n log n) for **every input** | $O(n^2)$ |
| **True median** (BFPRT) | none | O(n log n) | O(n log n) |

- * For deterministic strategies, O(n log n) holds only if the input is randomly ordered (**average case**). With a random pivot, the expectation is over the algorithm's own choices (**expected time**).
- A good pivot improves the **expected** time but does not remove the worst case: $O(n^2)$ is still possible with a random pivot, just very unlikely.
- BFPRT has large constants; in practice a random or median-of-3 pivot is preferred.

# Quicksort Run with Random Pivots

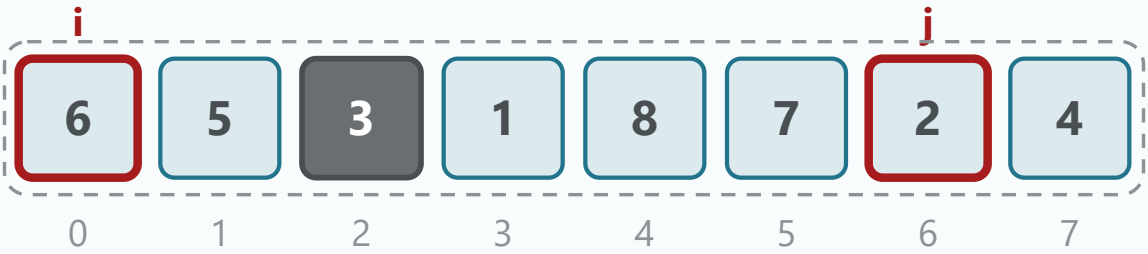
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Subarray a[0..7] = `[6, 5, 3, 1, 8, 7, 2, 4]` : the pivot is chosen **at random** → **3** (index 2, black box).

# Quicksort Run with Random Pivots

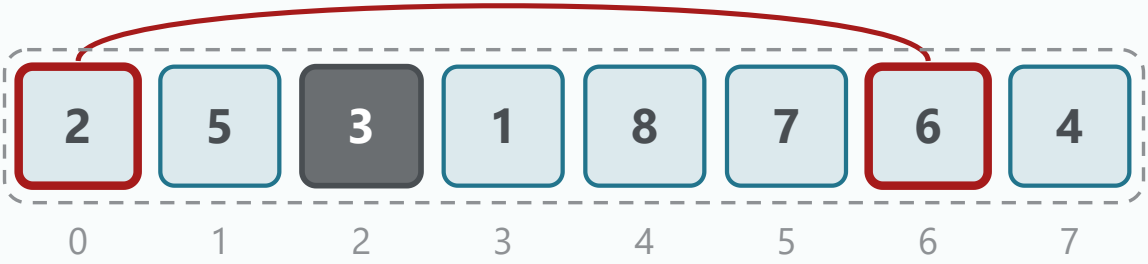
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




i moves right past elements smaller than 3 and stops at a[0] = 6; j moves left past elements larger than 3 and stops at a[6] = 2. i < j → swap.

# Quicksort Run with Random Pivots

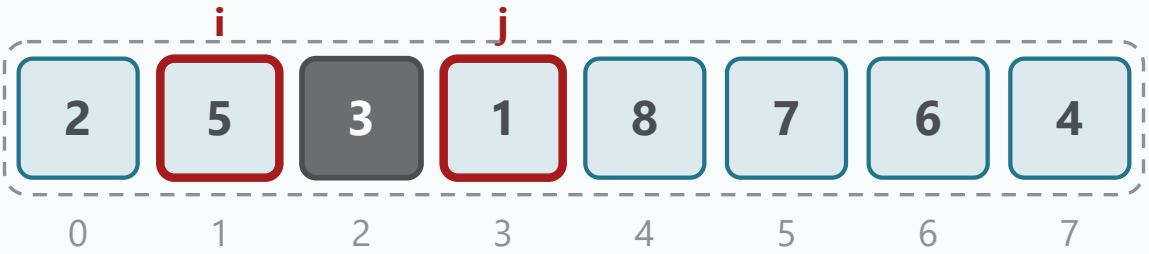
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Swap a[0] ↔ a[6] → `[2, 5, 3, 1, 8, 7, 6, 4]`.

# Quicksort Run with Random Pivots

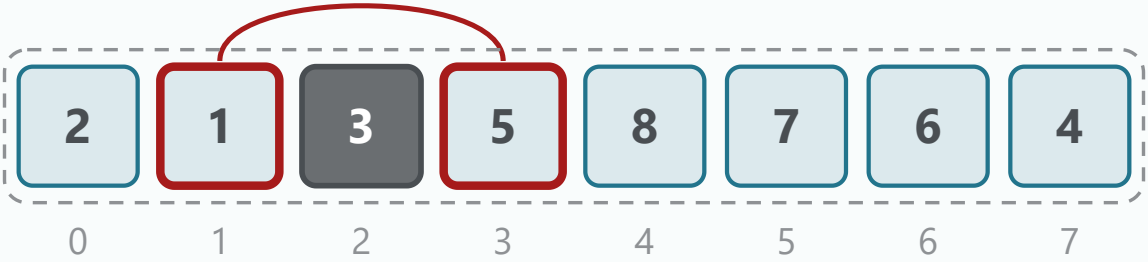
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




i moves right past elements smaller than 3 and stops at a[1] = 5; j moves left past elements larger than 3 and stops at a[3] = 1. i < j → swap.

# Quicksort Run with Random Pivots

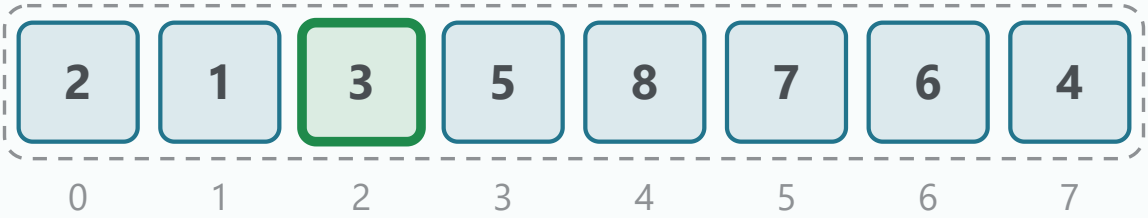
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Swap a[1] ↔ a[3] → `[2, 1, 3, 5, 8, 7, 6, 4]`.

# Quicksort Run with Random Pivots

[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The pointers have crossed (i ≥ j): pivot **3** is in its final place (index 2). Only smaller elements lie to its left, only larger ones to its right.

# Quicksort Run with Random Pivots

[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Subarray a[0..1] = `[2, 1]` : the pivot is chosen **at random** → **1** (index 1, black box).

# Quicksort Run with Random Pivots

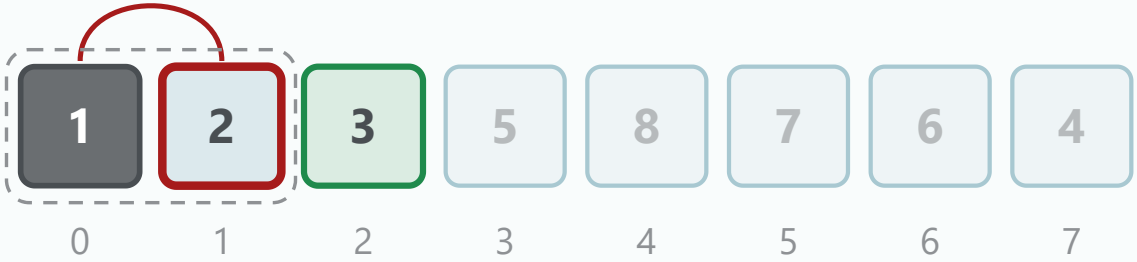
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




i moves right past elements smaller than 1 and stops at a[0] = 2; j moves left past elements larger than 1 and stops at a[1] = 1. i < j → swap.

# Quicksort Run with Random Pivots

[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Swap a[0] ↔ a[1] → `[1, 2, 3, 5, 8, 7, 6, 4]`. The pivot itself may move; we track its position.

# Quicksort Run with Random Pivots

[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The pointers have crossed (i ≥ j): pivot **1** is in its final place (index 0). Only smaller elements lie to its left, only larger ones to its right.

# Quicksort Run with Random Pivots

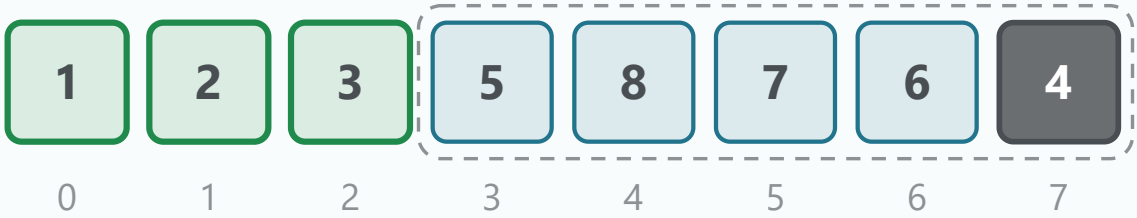
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The one-element subarray a[1] = 2 is already sorted: final.

# Quicksort Run with Random Pivots

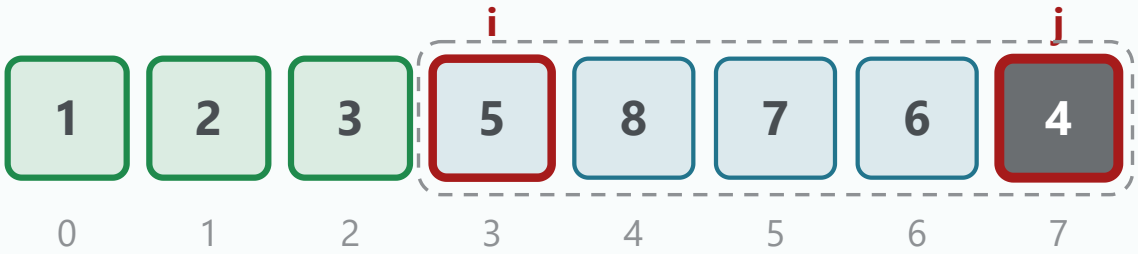
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Subarray a[3..7] = `[5, 8, 7, 6, 4]` : the pivot is chosen **at random** → **4** (index 7, black box).

# Quicksort Run with Random Pivots

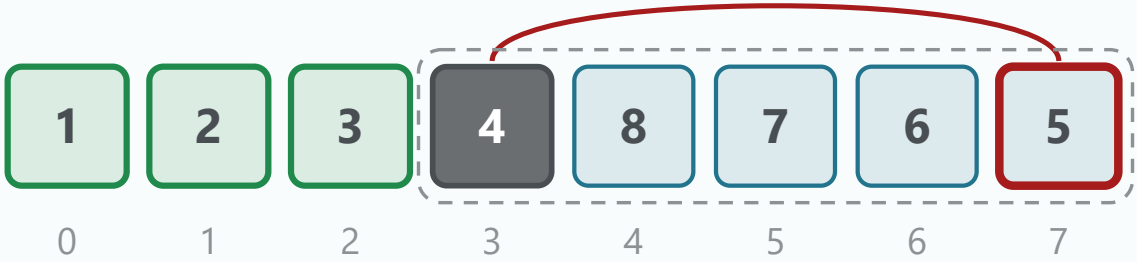
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




i moves right past elements smaller than 4 and stops at a[3] = 5; j moves left past elements larger than 4 and stops at a[7] = 4. i < j → swap.

# Quicksort Run with Random Pivots

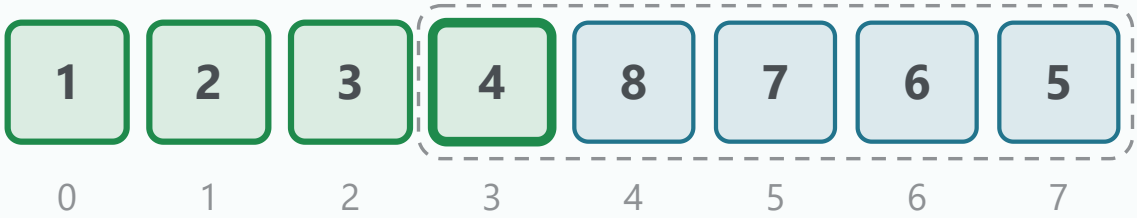
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Swap a[3] ↔ a[7] → `[1, 2, 3, 4, 8, 7, 6, 5]`. The pivot itself may move; we track its position.

# Quicksort Run with Random Pivots

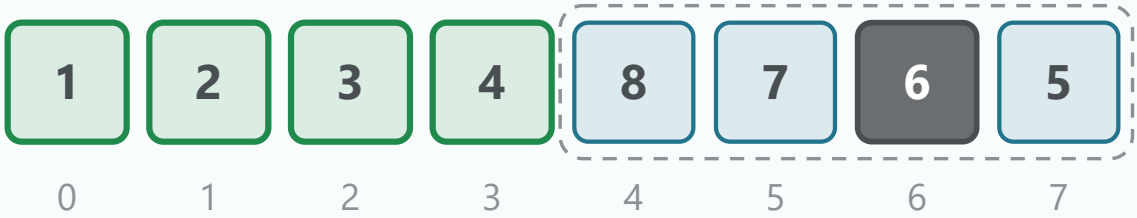
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The pointers have crossed (i ≥ j): pivot **4** is in its final place (index 3). Only smaller elements lie to its left, only larger ones to its right.

# Quicksort Run with Random Pivots

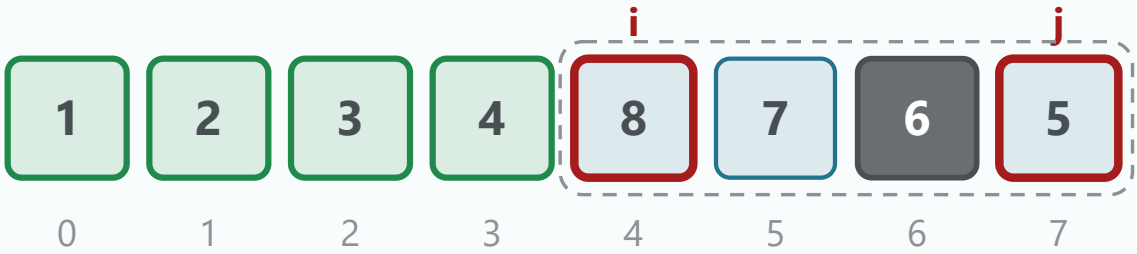
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Subarray a[4..7] = `[8, 7, 6, 5]` : the pivot is chosen **at random** → **6** (index 6, black box).

# Quicksort Run with Random Pivots

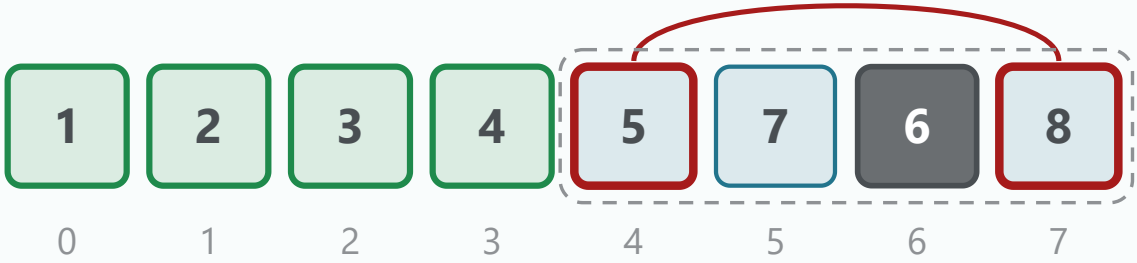
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




i moves right past elements smaller than 6 and stops at a[4] = 8; j moves left past elements larger than 6 and stops at a[7] = 5. i < j → swap.

# Quicksort Run with Random Pivots

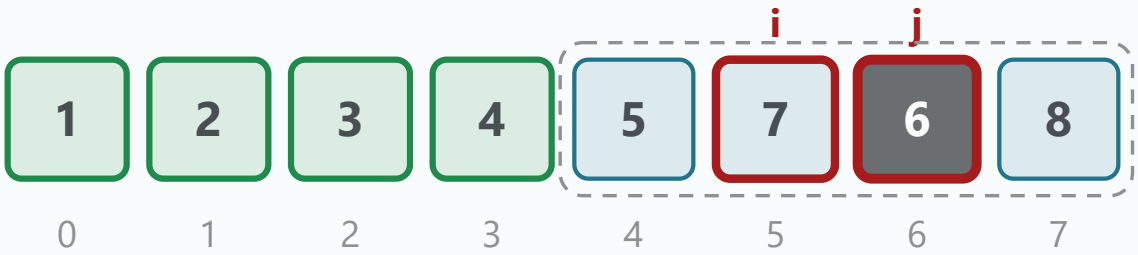
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Swap a[4] ↔ a[7] → `[1, 2, 3, 4, 5, 7, 6, 8]`.

# Quicksort Run with Random Pivots

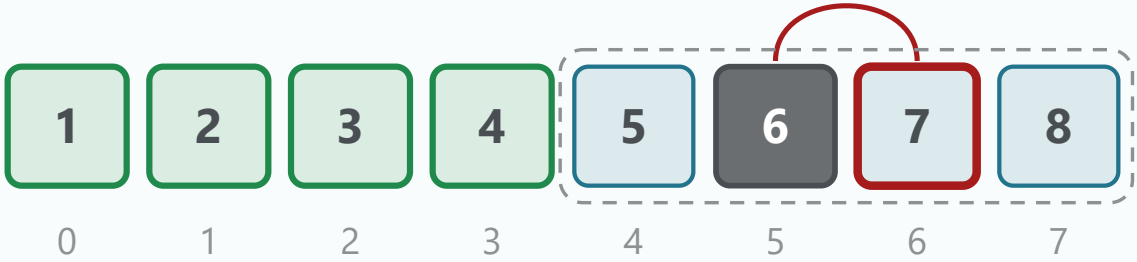
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




i moves right past elements smaller than 6 and stops at a[5] = 7; j moves left past elements larger than 6 and stops at a[6] = 6. i < j → swap.

# Quicksort Run with Random Pivots

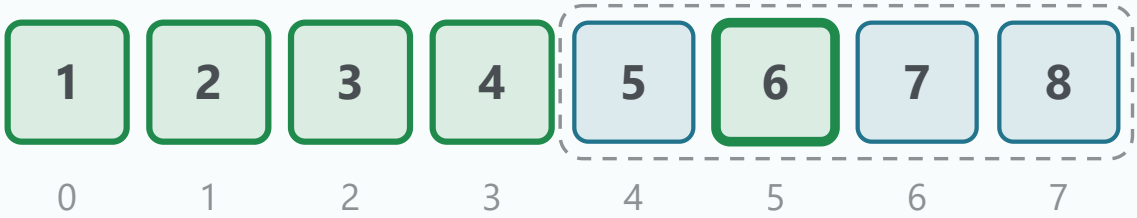
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Swap a[5] ↔ a[6] → `[1, 2, 3, 4, 5, 6, 7, 8]`. The pivot itself may move; we track its position.

# Quicksort Run with Random Pivots

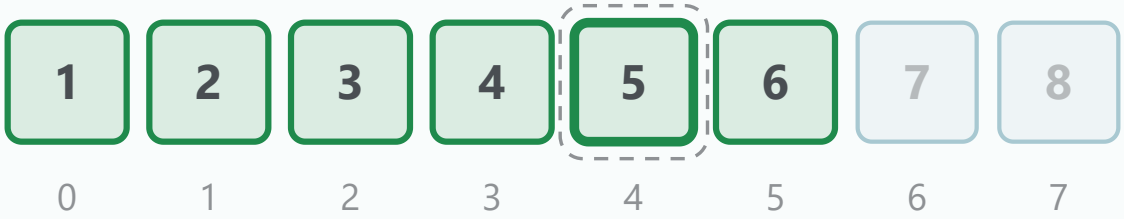
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The pointers have crossed (i ≥ j): pivot **6** is in its final place (index 5). Only smaller elements lie to its left, only larger ones to its right.

# Quicksort Run with Random Pivots

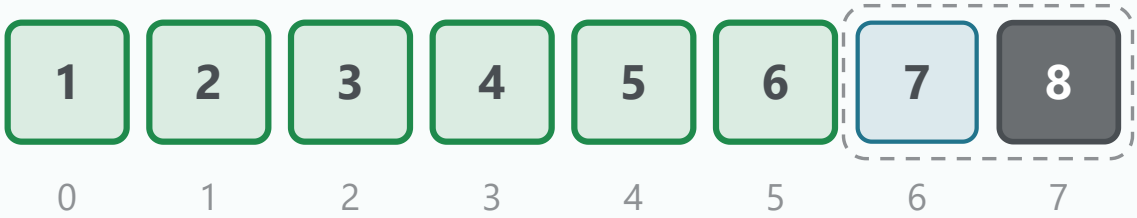
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The one-element subarray a[4] = 5 is already sorted: final.

# Quicksort Run with Random Pivots

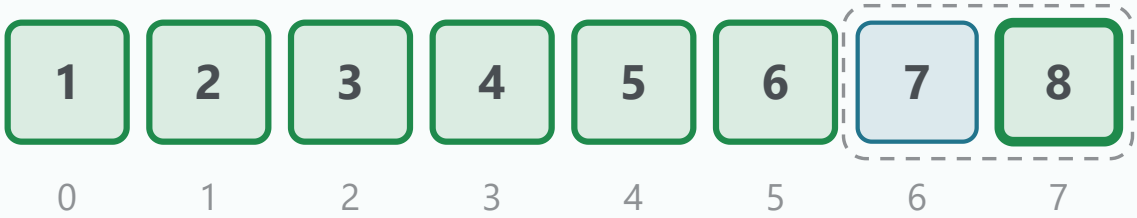
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Subarray a[6..7] = `[7, 8]`: the pivot is chosen **at random** → **8** (index 7, black box).

# Quicksort Run with Random Pivots

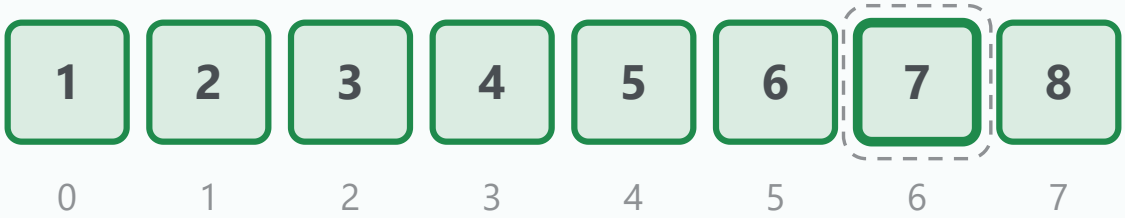
[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




Pivot 8: i skips 7 and stops at 8; j is already at 8. i ≥ j → no swap, **8** is in its final place (index 7).

# Quicksort Run with Random Pivots

[6, 5, 3, 1, 8, 7, 2, 4]; pivots chosen at random: 3, 1, 4, 6, 8. Black: pivot · red outline: pointers i and j · green: final · faded: outside the active subarray.




The one-element subarray a[6] = 7 is already sorted: final. **The array is sorted:** `[1, 2, 3, 4, 5, 6, 7, 8]`.

# Randomized Quicksort: Code

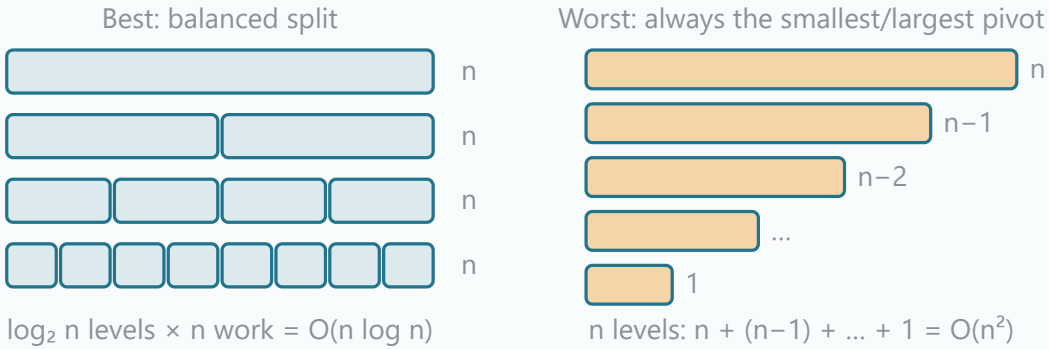
```
static Random rng = new Random();

static void quickSort(int[] a, int lo, int hi) {
  if (lo < hi) {
    int p = partition(a, lo, hi); // random
    quickSort(a, lo, p - 1);
    quickSort(a, p + 1, hi);
  }
}

static int partition(int[] a, int lo, int hi) {
  int r = lo + rng.nextInt(hi - lo + 1);
  swap(a, r, hi);           // pivot to end
  int pivot = a[hi], k = lo;
  for (int i = lo; i < hi; i++)
    if (a[i] <= pivot)
      swap(a, i, k++);      // smaller left
  swap(a, k, hi);           // pivot in place
  return k;
}
```

- The pivot is **uniformly random**. The output is always correct (**Las Vegas**); only the running time depends on the pivots.
- `partition` moves the random pivot to the end, then runs Lomuto: for [6, 5, 3, 1, 8, 7, 2, 4], r = 2 (pivot 3) → [1, 2, 3, 6, 8, 7, 5, 4], returns 2.
- Extra space: the recursion stack, expected depth O(log n); recursing on the smaller part first makes it O(log n) in the worst case too.
- With many duplicate keys, use three-way partitioning (<, =, >).

# Analysis of Quicksort



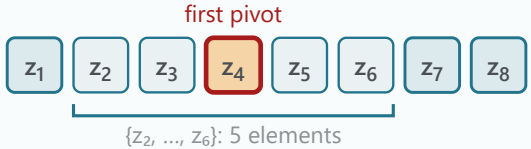

| Case | Recurrence | Time |
|---|---|---|
| **Best** | $T(n) = 2T(n/2) + \Theta(n)$ | $\Theta(n \log n)$ |
| **Expected** (random pivot) | $T(n) = (1/n)\,\Sigma_q\,[T(q) + T(n-1-q)] + \Theta(n)$ | $\Theta(n \log n)$ |
| **Worst** | $T(n) = T(n-1) + \Theta(n)$ | $\Theta(n^2)$ |

Probability of the worst case with a random pivot: the smallest or largest element must be chosen at every step → $2^{n-1}/n!$ ($\approx 2 \cdot 10^{-13}$ for $n = 20$).

# Expected Number of Comparisons

- Label the elements in sorted order: $z_1 < z_2 < \ldots < z_n$. Indicator $X_{ij} = 1 \Leftrightarrow z_i$ and $z_j$ are compared.
- Two elements are compared only if one is the **pivot**. If the **first** pivot from $\{z_i, \ldots, z_j\}$ lies strictly between them, they are split apart and never compared.
- That first pivot is uniformly random → $P(X_{ij} = 1) =$ **$2 / (j - i + 1)$**.
- Linearity: $E[X] = \Sigma_{i<j}\, 2/(j - i + 1) < 2n\, H_n \approx$ **$1.39\, n \log_2 n$**.
- Exact value: $E[X] = 2(n + 1)H_n - 4n$.

| n | random pivot (expected) | sorted input, first-element pivot |
|---|---|---|
| 10 | 24.4 | 45 |
| 100 | 648 | 4,950 |
| 1,000 | 10,986 | 499,500 |
| $10^6$ | $2.48 \cdot 10^7$ | $5.0 \cdot 10^{11}$ |




Pivot $z_4$ → $z_2$ goes left, $z_6$ goes right: they never meet.
Only if $z_2$ or $z_6$ is the first pivot: P = 2/5.

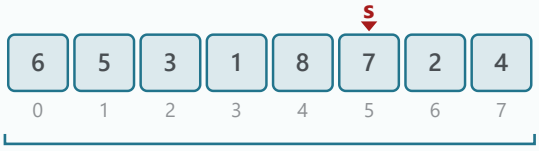
# Randomized Selection (Quickselect)

```
static int select(int[] a, int lo, int hi, int s) {
  if (lo == hi) return a[lo];
  int p = partition(a, lo, hi);
  if (s == p) return a[p];
  if (s < p) return select(a, lo, p - 1, s);
  return select(a, p + 1, hi, s);
}
```

- Goal: find the **s-th smallest** element (e.g., the median) without sorting.
- It uses the same random partitioning but recurses on **one side only**.
- Expected time **O(n)** (≈ 3.39 n comparisons for the median); worst case $O(n^2)$.
- Example: [6, 5, 3, 1, 8, 7, 2, 4] with s = 2 (0-based). Partitioning around pivot 3 gives p = 2 → the answer **3** on the first try.

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
▶   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




Array [6, 5, 3, 1, 8, 7, 2, 4]; target s = 5 (6th smallest). The first call covers the whole array: lo = 0, hi = 7.

Step 1 / 12

**CALL STACK**

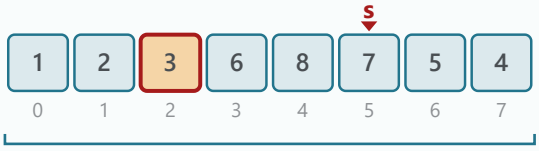
- `select(0, 7)`
- `main()`

**VARIABLES**

`lo = 0` `hi = 7` `s = 5` `p = –`

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
▶   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




`partition` chose random pivot 3; after partitioning its final position is p = **2**: left ≤ 3, right > 3.

Step 2 / 12

**CALL STACK**

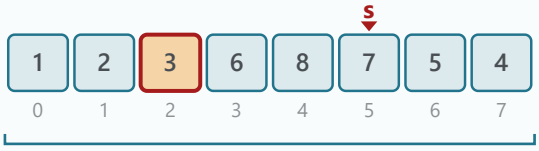
- **select(0, 7)**
- main()

**VARIABLES**

`lo = 0` `hi = 7` `s = 5` `p = 2`

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
▶   return select(a, p + 1, hi, s);
7 }
```




s = 5 > p = 2 → the target is on the right: recurse only on [3, 7]; the left side is never examined again.

Step 3 / 12

**CALL STACK**

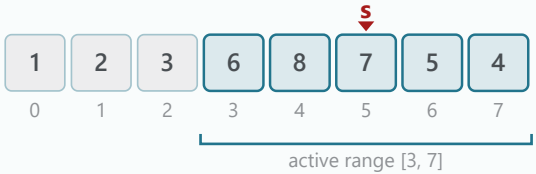
- **select(0, 7)**
- main()

**VARIABLES**

lo = **0** hi = **7** s = **5** p = **2**

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
▶   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




New call `select(a, 3, 7, 5)` : 5 elements in range, lo ≠ hi.

Step 4 / 12

**CALL STACK**

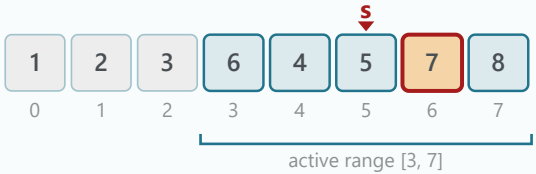
- **select(3, 7)**
- select(0, 7)
- main()

**VARIABLES**

lo = **3** | hi = **7** | s = **5** | p = **–**

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
▶   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




`partition` chose random pivot 7; after partitioning its final position is p = **6**: left ≤ 7, right > 7.

Step 5 / 12

**CALL STACK**

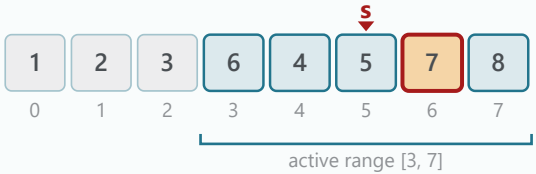
- **select(3, 7)**
- select(0, 7)
- main()

**VARIABLES**

`lo = 3` `hi = 7` `s = 5` `p = 6`

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
▶   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




s = 5 < p = 6 → the target is on the left: recurse only on [3, 5].

Step 6 / 12

**CALL STACK**

- **select(3, 7)**
- select(0, 7)
- main()

**VARIABLES**

lo = **3** hi = **7** s = **5** p = **6**

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
▶   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




New call `select(a, 3, 5, 5)` : 3 elements in range, lo ≠ hi.

Step 7 / 12

**CALL STACK**

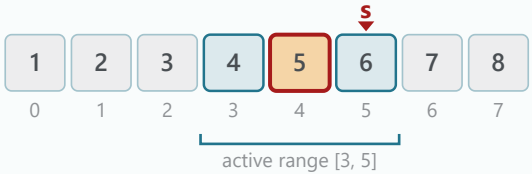
- **select(3, 5)**
- select(3, 7)
- select(0, 7)
- main()

**VARIABLES**

lo = **3** hi = **5** s = **5** p = **–**

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
▶   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




`partition` chose random pivot 5; after partitioning its final position is p = **4**: left ≤ 5, right > 5.

Step 8 / 12

**CALL STACK**

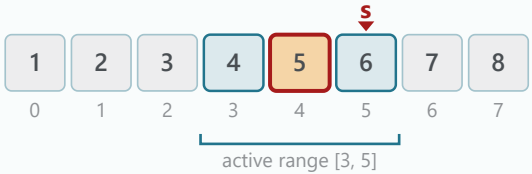
- **select(3, 5)**
- select(3, 7)
- select(0, 7)
- main()

**VARIABLES**

`lo = 3` `hi = 5` `s = 5` `p = 4`

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
▶   return select(a, p + 1, hi, s);
7 }
```




s = 5 > p = 4 → the target is on the right: recurse only on [5, 5]; the left side is never examined again.

Step 9 / 12

**CALL STACK**

- **select(3, 5)**
- select(3, 7)
- select(0, 7)
- main()

**VARIABLES**

lo = **3** hi = **5** s = **5** p = **4**

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
▶   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




New call `select(a, 5, 5, 5)` : 1 element in range, lo = hi.

Step 10 / 12

**CALL STACK**

- **select(5, 5)**
- select(3, 5)
- select(3, 7)
- select(0, 7)
- main()

**VARIABLES**

lo = **5** hi = **5** s = **5** p = **–**

# Quickselect: Code Trace

```
1 static int select(int[] a, int lo, int hi, int s) {
▶   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




lo = hi = 5: one element left → return a[5] = **6**.

Step 11 / 12

**CALL STACK**

- **select(5, 5)**
- select(3, 5)
- select(3, 7)
- select(0, 7)
- main()

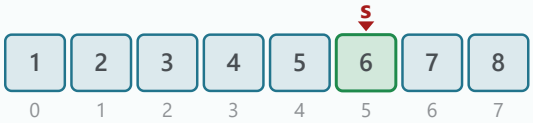
**VARIABLES**

lo = **5** hi = **5** s = **5**

returned = **6**

# Quickselect: Code Trace

```
▶ static int select(int[] a, int lo, int hi, int s) {
2   if (lo == hi) return a[lo];
3   int p = partition(a, lo, hi);
4   if (s == p) return a[p];
5   if (s < p) return select(a, lo, p - 1, s);
6   return select(a, p + 1, hi, s);
7 }
```




`select(a, 0, 7, 5)` = **6**: position 5 (0-based) of the sorted array. The array was never fully sorted; only the needed parts were partitioned.

Step 12 / 12

**CALL STACK**

`main()`

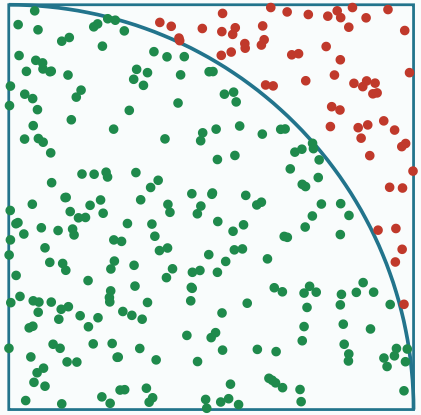
**VARIABLES**

`returned = 6`

# Monte Carlo Estimate of π

- Throw uniformly random points into the unit square. The quarter circle has area π/4 and the square area 1 → P(inside) = π/4, so **π ≈ 4 · inside / n**.
- The error falls like 1/√n: one more correct digit needs **100 times** as many points.

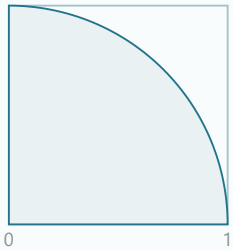
```
static double estimatePi(int n) {
  int inside = 0;
  for (int t = 0; t < n; t++) {
    double x = rng.nextDouble(), y = rng.nextDouble();
    if (x * x + y * y <= 1) inside++;
  }
  return 4.0 * inside / n;
}
```



300 points: 239 inside → π ≈ 4 · 239/300 = **3.187**.

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
▶   int inside = 0;
3   for (int t = 0; t < n; t++) {
4     double x = rng.nextDouble(), y = rng.nextDouble();
5     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 0

inside: 0

n = 8 points will be thrown; `inside` counts those landing inside the quarter circle. (A seeded `Random` makes the run reproducible.)

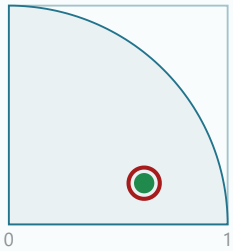
Step 1 / 10

**VARIABLES**

`n = 8` `inside = 0`

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 1

inside: 1

$x^2 + y^2 = 0.42$

$\leq 1 \rightarrow$ inside

t = 0: (0.62, 0.19), $x^2 + y^2 = 0.42 \leq 1$ → **inside**. inside = 1.

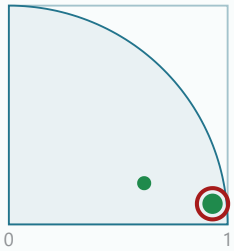
Step 2 / 10

VARIABLES

`t = 0` `x = 0.62` `y = 0.19` `x²+y² = 0.42` `inside = 1`

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 2

inside: 2

$x^2 + y^2 = 0.88$

$\leq 1 \rightarrow$ inside

t = 1: (0.93, 0.09), $x^2 + y^2 = 0.88 \leq 1$ → **inside**. inside = 2.

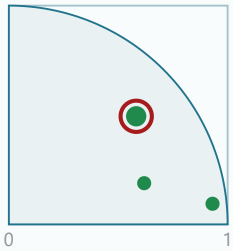
Step 3 / 10

VARIABLES

`t = 1` `x = 0.93` `y = 0.09` `x²+y² = 0.88` `inside = 2`

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 3

inside: 3

$x^2 + y^2 = 0.58$

$\leq 1 \rightarrow$ inside

t = 2: (0.58, 0.49), $x^2 + y^2 = 0.58 \leq 1$ → **inside**. inside = 3.

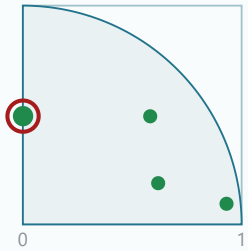
Step 4 / 10

VARIABLES

`t = 2` `x = 0.58` `y = 0.49` `x²+y² = 0.58` `inside = 3`

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 4

inside: 4

$x^2 + y^2 = 0.24$

$\leq 1 \rightarrow$ inside

t = 3: (0.00, 0.49), $x^2 + y^2 = 0.24 \leq 1$ → **inside**. inside = 4.

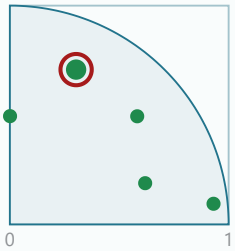
Step 5 / 10

VARIABLES

t = 3 | x = 0.00 | y = 0.49 | x²+y² = 0.24 | inside = 4

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 5

inside: 5

$x^2 + y^2 = 0.59$

$\leq 1 \rightarrow$ inside

t = 4: (0.30, 0.71), $x^2 + y^2 = 0.59 \leq 1$ → **inside**. inside = 5.

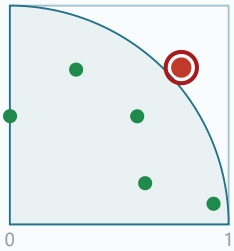
Step 6 / 10

VARIABLES

`t = 4` `x = 0.30` `y = 0.71` `x²+y² = 0.59` `inside = 5`

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 6

inside: 5

$x^2 + y^2 = 1.13$

$> 1 \rightarrow$ outside

t = 5: (0.78, 0.72), $x^2 + y^2 = 1.13 > 1$ → outside. inside = 5.

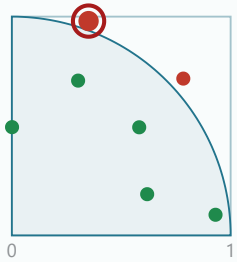
Step 7 / 10

VARIABLES

t = 5 | x = 0.78 | y = 0.72 | $x^2+y^2$ = 1.13 | inside = 5

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 7

inside: 5

$x^2 + y^2 = 1.08$

$> 1 \rightarrow$ outside

t = 6: (0.35, 0.98), $x^2 + y^2 = 1.08 > 1$ → outside. inside = 5.

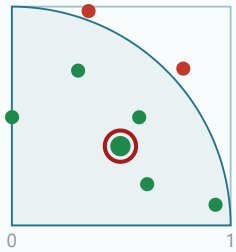
Step 8 / 10

VARIABLES

t = 6 | x = 0.35 | y = 0.98 | x²+y² = 1.08 | inside = 5

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
▶   for (int t = 0; t < n; t++) {
▶     double x = rng.nextDouble(), y = rng.nextDouble();
▶     if (x * x + y * y <= 1) inside++;
6   }
7   return 4.0 * inside / n;
8 }
```




points: 8

inside: 6

$x^2 + y^2 = 0.38$

$\leq 1 \rightarrow$ inside

t = 7: (0.50, 0.36), $x^2 + y^2 = 0.38 \leq 1$ → **inside**. inside = 6.

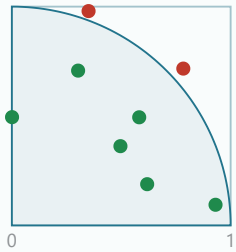
Step 9 / 10

VARIABLES

t = 7 | x = 0.50 | y = 0.36 | x²+y² = 0.38 | inside = 6

# Monte Carlo π: Code Trace

```
1 static double estimatePi(int n) {
2   int inside = 0;
3   for (int t = 0; t < n; t++) {
4     double x = rng.nextDouble(), y = rng.nextDouble();
5     if (x * x + y * y <= 1) inside++;
6   }
▶   return 4.0 * inside / n;
8 }
```




points: 8

inside: 6

π ≈ 4 · 6 / 8

= 3.0

`4.0 * 6 / 8` = **3.0**. With 8 points the error is large (≈ 1.64/√8 ≈ 0.58); the estimate approaches π as n grows.

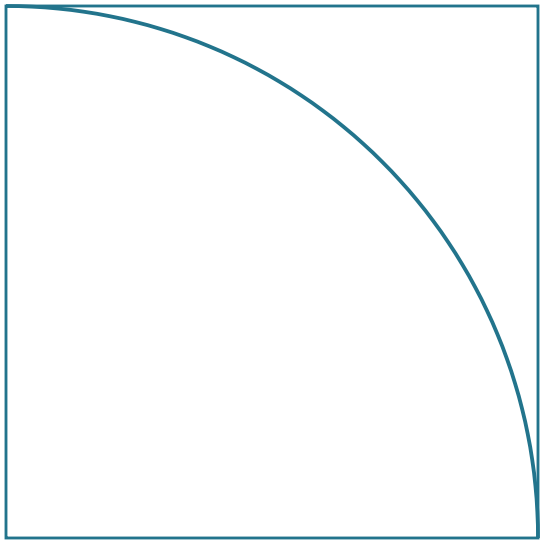
Step 10 / 10

VARIABLES

`n = 8` `inside = 6` `returned = 3.0`

# Try It: Monte Carlo π

+10 points | +100 | +1000 | +10000 | reset



| | |
|---|---|
| number of points n | 0 |
| inside | 0 |
| π estimate | — |
| error $\lvert\hat{\pi} - \pi\rvert$ | — |
| standard deviation ≈ 1.64/√n | — |

Press a button to add points.

Green: inside the quarter circle · red: outside. The plot shows the first 3000 points; counting continues over all points. 1.64 = 4 · √(p(1 − p)), p = π/4.

# Miller–Rabin Primality Test

- **Fermat's little theorem:** if n is prime and $1 < a < n$, then $a^{n-1} \equiv 1 \pmod{n}$. If this fails, n is certainly composite; but some composites (Carmichael numbers) satisfy it for every base.
- **Square-root property:** if n is prime, $x^2 \equiv 1 \pmod{n}$ holds only for $x \equiv \pm 1$.
- Write $n - 1 = 2^s \cdot d$ (d odd); for a random a, examine the sequence $a^d$, $a^{2d}$, …, $a^{2^{s-1}d} \pmod{n}$.
- If the sequence does not start with 1 and $n - 1$ never appears, a is a **witness** that n is **composite**. Otherwise n "passes" this round.
- For composite n, at most **1/4** of the bases are liars → with k independent rounds the error is $\leq \mathbf{4^{-k}}$. Each round takes O(log n) modular multiplications: $O(k \log^3 n)$ bit operations in total.




passes: the sequence starts with 1 or n − 1 appears somewhere

witness: first term ≠ 1 and n − 1 never appears → n certainly composite

One-sided error: the answer "composite" is **certain**; the answer "prime" is only **probably** right.

# Miller–Rabin: Code

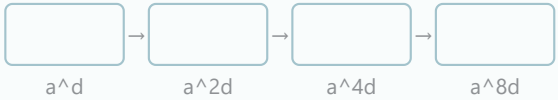
```
static boolean isPrime(long n, int k) {
  if (n < 4) return n == 2 || n == 3;
  if (n % 2 == 0) return false;
  long d = n - 1; int s = 0;      // n-1 = 2^s·d
  while (d % 2 == 0) { d /= 2; s++; }
  for (int round = 0; round < k; round++) {
    long a = 2 + rng.nextLong(n - 3); // [2, n-2]
    long x = modPow(a, d, n);     // a^d mod n
    if (x == 1 || x == n - 1) continue;
    boolean witness = true;
    for (int r = 1; r < s && witness; r++) {
      x = x * x % n;
      if (x == n - 1) witness = false;
    }
    if (witness) return false;    // certainly composite
  }
  return true;                    // probably prime
}
```

- `modPow`: fast modular exponentiation (square-and-multiply), O(log d) multiplications.
- The base a is drawn uniformly from [2, n − 2].
- With `long`, `x * x` does not overflow only for $n < 3 \cdot 10^9$; for larger n use `BigInteger.modPow` or `isProbablePrime`.

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
▶    if (n < 4) return n == 2 || n == 3;
▶    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^0 \cdot 560$




n = 561 = 3 · 11 · 17 is a Carmichael number: it passes the Fermat test for every base. n ≥ 4 and odd → the early checks do not return.

Step 1 / 13

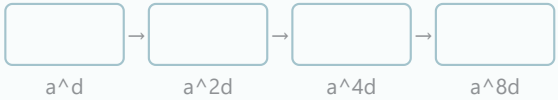
**VARIABLES**

n = **561** k = **1**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
▶    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^0 \cdot 560$




d = n − 1 = 560, s = 0. Now factor the 2s out of 560.

Step 2 / 13

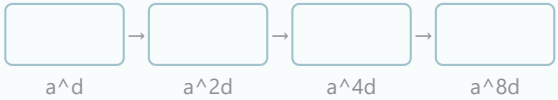
**VARIABLES**

n = **561** d = **560** s = **0**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^1 \cdot 280$




d is even → d /= 2, s++: d = 280, s = 1.

Step 3 / 13

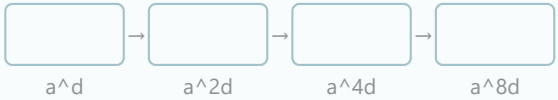
**VARIABLES**

n = **561** d = **280** s = **1**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^2 \cdot 140$




d is even → d /= 2, s++: d = 140, s = 2.

Step 4 / 13

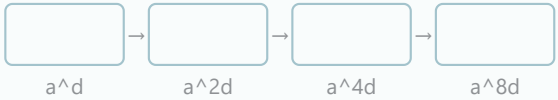
**VARIABLES**

n = **561** d = **140** s = **2**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^3 \cdot 70$




d is even → d /= 2, s++: d = 70, s = 3.

Step 5 / 13

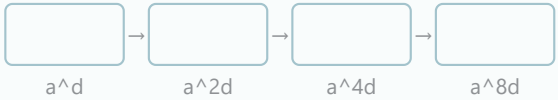
**VARIABLES**

n = **561** d = **70** s = **3**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
▶    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$




d is even → d /= 2, s++: d = 35, s = 4. d is odd: 560 = $2^4 \cdot 35$.

Step 6 / 13

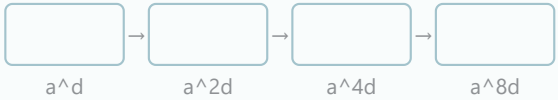
VARIABLES

n = 561 | d = 35 | s = 4

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
▶    for (int round = 0; round < k; round++) {
▶      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ $\quad a = 2$




Round 0: random base drawn, a = **2**.

Step 7 / 13

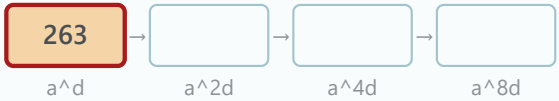
**VARIABLES**

`round = 0` `d = 35` `s = 4` `a = 2`

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
▶      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2




$x = 2^{35} \bmod 561 =$ **263** (by fast exponentiation).

Step 8 / 13

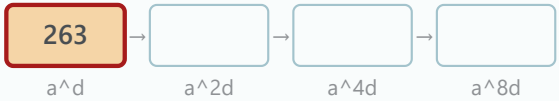
VARIABLES

round = 0 | d = 35 | s = 4 | a = 2 | x = 263

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
▶      if (x == 1 || x == n - 1) continue;
▶      boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2




x = 263 is neither 1 nor n − 1 = 560 → the round is not over; with `witness = true` we check the squares.

Step 9 / 13

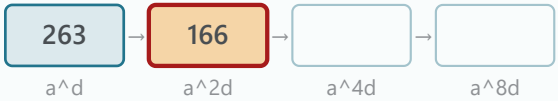
VARIABLES

a = 2 | x = 263 | witness = true

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
▶      for (int r = 1; r < s && witness; r++) {
▶        x = x * x % n;
▶        if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2




r = 1: $x = 263^2 \bmod 561 =$ **166**. Not 560.

Step 10 / 13

VARIABLES

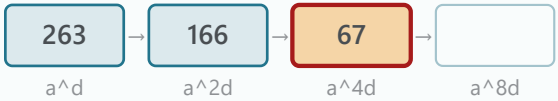
a = **2** r = **1** x = **166**

witness = **true**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
▶      for (int r = 1; r < s && witness; r++) {
▶        x = x * x % n;
▶        if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2




r = 2: $x = 166^2 \bmod 561 =$ **67**. Not 560.

Step 11 / 13

VARIABLES

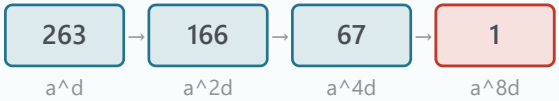
a = **2** r = **2** x = **67**

witness = **true**

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
▶      for (int r = 1; r < s && witness; r++) {
▶        x = x * x % n;
▶        if (x == n - 1) witness = false;
14     }
15     if (witness) return false;    // certainly
   composite
16   }
17   return true;                    // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2




r = 3: $x = 67^2 \bmod 561 = \mathbf{1}$. Not 560. We reached 1 without seeing n − 1.

Step 12 / 13

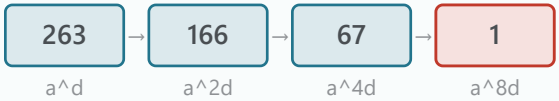
VARIABLES

a = 2 | r = 3 | x = 1 | witness = true

# Miller–Rabin: Code Trace (n = 561)

```
1  static boolean isPrime(long n, int k) {
2    if (n < 4) return n == 2 || n == 3;
3    if (n % 2 == 0) return false;
4    long d = n - 1; int s = 0;      // n-1 = 2^s·d
5    while (d % 2 == 0) { d /= 2; s++; }
6    for (int round = 0; round < k; round++) {
7      long a = 2 + rng.nextLong(n - 3); // [2, n-2]
8      long x = modPow(a, d, n);     // a^d mod n
9      if (x == 1 || x == n - 1) continue;
10     boolean witness = true;
11     for (int r = 1; r < s && witness; r++) {
12       x = x * x % n;
13       if (x == n - 1) witness = false;
14     }
▶      if (witness) return false;   // certainly
composite
16   }
17   return true;                   // probably prime
18 }
```

$n - 1 = 560 = 2^4 \cdot 35$ a = 2




$67^2 \equiv 1$ but $67 \neq \pm 1 \rightarrow$ a = 2 is a witness: 561 is composite

The sequence never reached 560: `witness` stayed true → `return false`. 561 is **certainly composite**; 67 is a square root of 1 other than ±1.

Step 13 / 13

VARIABLES

`a = 2` `x = 1` `witness = true` `returned = false`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
5      if ((d & 1) == 1)          // bit is 1
6        result = result * a % n;
7      a = a * a % n;             // square
8      d >>= 1;                   // next bit
9    }
10   return result;
11 }
```

$d = 35 = 100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 1**

a = 2

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

Compute $2^{35}$ mod 561. Since 35 = 32 + 2 + 1, $2^{35} = 2^{32} \cdot 2^2 \cdot 2^1$. Initially result = 1, a = 2.

Step 1 / 8

VARIABLES

`a = 2` `d = 35` `result = 1`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit is 1
▶        result = result * a % n;
▶      a = a * a % n;               // square
▶      d >>= 1;                     // next bit
9    }
10   return result;
11 }
```

d = 35 = $100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 2** × a: 1 · 2 mod 561

**a = 4** square: $2^2$ mod 561

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

Bit 0 = 1: result = 1 · 2 mod 561 = **2** ($2^1$ included). Then a = $2^2$ mod 561 = 4.

Step 2 / 8

VARIABLES

`a = 4` `d = 17` `result = 2`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit is 1
▶        result = result * a % n;
▶      a = a * a % n;               // square
▶      d >>= 1;                     // next bit
9    }
10   return result;
11 }
```

d = 35 = $100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 8** × a: 2 · 4 mod 561

**a = 16** square: $4^2$ mod 561

a becomes 2^1, 2^2, 2^4, 2^8 … (mod 561) in turn

Bit 1 = 1: result = 2 · 4 mod 561 = **8** ($2^2$ included). Then a = $4^2$ mod 561 = 16.

Step 3 / 8

VARIABLES

`a = 16` `d = 8` `result = 8`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit is 1
6        result = result * a % n;
▶      a = a * a % n;               // square
▶      d >>= 1;                     // next bit
9    }
10   return result;
11 }
```

d = 35 = $100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 8** bit 0: no multiply

**a = 256** square: $16^2$ mod 561

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

Bit 2 = 0: result unchanged (8). Only square: a = $16^2$ mod 561 = 256.

Step 4 / 8

VARIABLES

`a = 256` `d = 4` `result = 8`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit is 1
6        result = result * a % n;
▶      a = a * a % n;               // square
▶      d >>= 1;                     // next bit
9    }
10   return result;
11 }
```

d = 35 = $100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 8** bit 0: no multiply

**a = 460** square: $256^2$ mod 561

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

Bit 3 = 0: result unchanged (8). Only square: a = $256^2$ mod 561 = 460.

Step 5 / 8

VARIABLES

`a = 460` `d = 2` `result = 8`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)            // bit is 1
6        result = result * a % n;
▶      a = a * a % n;               // square
▶      d >>= 1;                     // next bit
9    }
10   return result;
11 }
```

$d = 35 = 100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 8** bit 0: no multiply

**a = 103** square: $460^2$ mod 561

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

Bit 4 = 0: result unchanged (8). Only square: a = $460^2$ mod 561 = 103.

Step 6 / 8

VARIABLES

`a = 103` `d = 1` `result = 8`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
▶      if ((d & 1) == 1)          // bit is 1
▶        result = result * a % n;
▶      a = a * a % n;             // square
▶      d >>= 1;                   // next bit
9    }
10   return result;
11 }
```

$d = 35 = 100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

**result = 263** × a: 8 · 103 mod 561

a = 511 square: $103^2$ mod 561

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

Bit 5 = 1: result = 8 · 103 mod 561 = **263** ($2^{32}$ included). Then a = $103^2$ mod 561 = 511.

Step 7 / 8

VARIABLES

`a = 511` `d = 0` `result = 263`

# modPow: $2^{35}$ mod 561 Step by Step

```
1  static long modPow(long a, long d, long n) {
2    long result = 1;
3    a %= n;
4    while (d > 0) {
5      if ((d & 1) == 1)          // bit is 1
6        result = result * a % n;
7      a = a * a % n;             // square
8      d >>= 1;                   // next bit
9    }
▶    return result;
11 }
```

d = 35 = $100011_2$

| 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 |

result = 263

a = 511

a becomes 2^1, 2^2, 2^4, 2^8 ... (mod 561) in turn

d = 0: loop done. `return` **263** = $2^{35}$ mod 561, using only 6 squarings + 3 multiplications instead of 34.

Step 8 / 8

VARIABLES

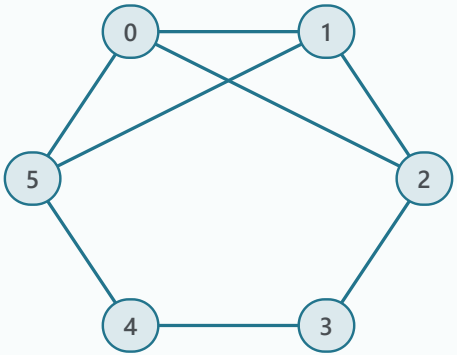
`result = 263` `returned = 263`

# Miller–Rabin Examples

| n | n − 1 | a | $a^d$, $a^{2d}$, … (mod n) | result |
|---|---|---|---|---|
| 61 | $2^2 \cdot 15$ | 2 | `11, 60` | passes (may be prime) |
| 221 | $2^2 \cdot 55$ | 174 | `47, 220` | passes (may be prime) |
| 221 | $2^2 \cdot 55$ | 137 | `188, 205` | **composite** (a is a witness) |
| 561 | $2^4 \cdot 35$ | 2 | `263, 166, 67, 1` | **composite** (a is a witness) |

- **61** is prime: the sequence reaches n − 1 = 60.
- **221 = 13 · 17**: a = 174 is a **strong liar** (the sequence reaches 220), while a = 137 is a witness. That is why we need several rounds.
- **561 = 3 · 11 · 17** is a Carmichael number: $2^{560} \equiv 1$, so the Fermat test is fooled. Miller–Rabin sees that $67^2 \equiv 1$: 67 is a square root of 1 other than ±1 → **composite**.

# Karger's Min-Cut Algorithm

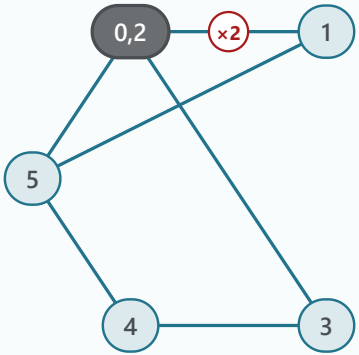



Start: 6 vertices, 8 edges. The minimum cut is 2 (e.g., {3, 4} | the rest). Dark vertex: the supervertex formed by the last contraction.

Step 1 / 5

- **Problem:** find a cut with the fewest edges that splits the graph into two parts (global min cut).
- **Algorithm:** contract a uniformly random edge until 2 vertices remain; the remaining edges form a cut.
- One run finds a minimum cut with probability at least $\mathbf{2/(n(n-1))}$ ($n = 6$: $\geq 1/15$). On this graph, 200,000 simulations gave $\approx 0.46$.
- The output can be wrong (a larger cut) → **Monte Carlo**. E.g., if the first edges chosen are (1,2), (4,5), (3,4), (0,1), we get the cut {0,1,2} | {3,4,5} with 3 edges.
- $n(n-1)/2 \cdot \ln n$ repetitions → failure $\leq 1/n$; $O(n^4 \log n)$ in total (simple implementation).

# Karger's Min-Cut Algorithm

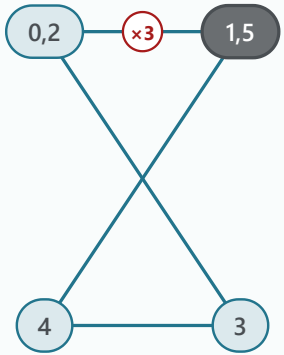



Random edge (0, 2) **contracted**: its endpoints merge into one supervertex, edges between them (self-loops) are deleted, parallel edges (×k) are kept. Remaining: 7 edges.



- **Problem:** find a cut with the fewest edges that splits the graph into two parts (global min cut).
- **Algorithm:** contract a uniformly random edge until 2 vertices remain; the remaining edges form a cut.
- One run finds a minimum cut with probability at least $\mathbf{2/(n(n-1))}$ ($n = 6$: $\geq 1/15$). On this graph, 200,000 simulations gave $\approx 0.46$.
- The output can be wrong (a larger cut) → **Monte Carlo**. E.g., if the first edges chosen are (1,2), (4,5), (3,4), (0,1), we get the cut {0,1,2} | {3,4,5} with 3 edges.
- $n(n-1)/2 \cdot \ln n$ repetitions → failure $\leq 1/n$; $O(n^4 \log n)$ in total (simple implementation).

# Karger's Min-Cut Algorithm

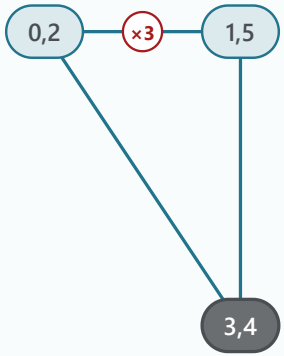



Random edge (1, 5) **contracted**: its endpoints merge into one supervertex, edges between them (self-loops) are deleted, parallel edges (×k) are kept. Remaining: 6 edges.



- **Problem:** find a cut with the fewest edges that splits the graph into two parts (global min cut).
- **Algorithm:** contract a uniformly random edge until 2 vertices remain; the remaining edges form a cut.
- One run finds a minimum cut with probability at least $\mathbf{2/(n(n-1))}$ ($n = 6$: $\geq 1/15$). On this graph, 200,000 simulations gave $\approx 0.46$.
- The output can be wrong (a larger cut) → **Monte Carlo**. E.g., if the first edges chosen are (1,2), (4,5), (3,4), (0,1), we get the cut {0,1,2} | {3,4,5} with 3 edges.
- $n(n-1)/2 \cdot \ln n$ repetitions → failure $\leq 1/n$; $O(n^4 \log n)$ in total (simple implementation).

# Karger's Min-Cut Algorithm

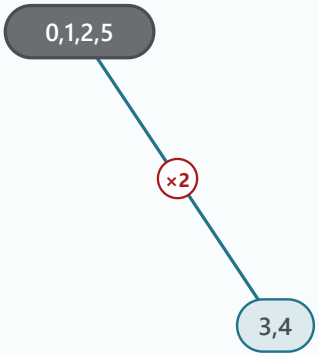



Random edge (3, 4) **contracted**: its endpoints merge into one supervertex, edges between them (self-loops) are deleted, parallel edges (×k) are kept. Remaining: 5 edges.



- **Problem:** find a cut with the fewest edges that splits the graph into two parts (global min cut).
- **Algorithm:** contract a uniformly random edge until 2 vertices remain; the remaining edges form a cut.
- One run finds a minimum cut with probability at least **$2/(n(n-1))$** ($n = 6$: $\geq 1/15$). On this graph, 200,000 simulations gave $\approx 0.46$.
- The output can be wrong (a larger cut) → **Monte Carlo**. E.g., if the first edges chosen are (1,2), (4,5), (3,4), (0,1), we get the cut {0,1,2} | {3,4,5} with 3 edges.
- $n(n-1)/2 \cdot \ln n$ repetitions → failure $\leq 1/n$; $O(n^4 \log n)$ in total (simple implementation).

# Karger's Min-Cut Algorithm




Random edge (0, 1) **contracted**: its endpoints merge into one supervertex, edges between them (self-loops) are deleted, parallel edges (×k) are kept. Remaining: 2 edges. Two supervertices left → cut found = **2** (minimum; a successful run).



- **Problem:** find a cut with the fewest edges that splits the graph into two parts (global min cut).
- **Algorithm:** contract a uniformly random edge until 2 vertices remain; the remaining edges form a cut.
- One run finds a minimum cut with probability at least $\mathbf{2/(n(n-1))}$ ($n = 6$: $\geq 1/15$). On this graph, 200,000 simulations gave $\approx 0.46$.
- The output can be wrong (a larger cut) → **Monte Carlo**. E.g., if the first edges chosen are (1,2), (4,5), (3,4), (0,1), we get the cut {0,1,2} | {3,4,5} with 3 edges.
- $n(n-1)/2 \cdot \ln n$ repetitions → failure $\leq 1/n$; $O(n^4 \log n)$ in total (simple implementation).

# Other Applications

### Graph coloring

Tries random colorings, searching for one in which no two neighbors share a color.

### Randomized DFS

Visits neighbors in random order to find connectivity and spanning trees (e.g., maze generation).

### Matching

Randomized incremental algorithms add edges in random order to approach a maximum matching.

### Approximate counting

Count-Min Sketch uses random hash functions to estimate frequencies in large data streams with controlled error.

### Matrix computations

Randomized matrix approximation yields low-rank approximations: dimensionality reduction, data compression.

### Computational geometry

Randomized incremental convex hull, closest pair, Delaunay triangulation: expected O(n log n).

Randomized scheduling (job assignment) and metaheuristics such as simulated annealing and genetic algorithms also use randomness to search; metaheuristics do not guarantee convergence to the optimum.

# Pseudorandom Numbers

- Computers produce **pseudorandom** numbers rather than true randomness: a deterministic sequence that looks random.
- **Linear congruential generator** (LCG): $X_{k+1} = (a \cdot X_k + c) \bmod m$. `java.util.Random`: $a = 25214903917$, $c = 11$, $m = 2^{48}$.
- The same **seed** gives the same sequence: runs with `new Random(42)` are reproducible (debugging, experiments).
- Use `SecureRandom` for cryptography and `ThreadLocalRandom` in multithreaded code.

### Random ranges

- `rng.nextInt(n)` → 0 … n − 1
- `lo + rng.nextInt(hi − lo + 1)` → lo … hi
- `rng.nextDouble()` → [0, 1)
- `rng.nextInt(6) + 1` → a die roll: 1 … 6

# Summary

| Topic | Key points |
|---|---|
| **Randomized algorithm** | input + random bits; time/output are random variables; expectation over the algorithm's choices |
| **Las Vegas** | always correct, random time: Quicksort, quickselect, BogoSort |
| **Monte Carlo** | bounded time, small error probability; repetition drives it down to $q^k$ |
| **Fisher–Yates** | `j = nextInt(i + 1)`, $\Theta(n)$, all n! outcomes equally likely; the naive shuffle is biased |
| **BogoSort** | $p = 1/n!$, expected $\Theta(n \cdot n!)$, worst case unbounded |
| **Randomized quicksort** | expected $\approx 1.39\, n \log_2 n$ comparisons, $O(n \log n)$ for every input; worst case $O(n^2)$ |
| **Miller–Rabin** | $n - 1 = 2^s d$; error $\leq 4^{-k}$; "composite" is certain |
| **Karger** | random edge contraction; success $\geq 2/(n(n-1))$; repeat many times |
| **π estimation** | $4 \cdot$ inside / n; error $\propto 1/\sqrt{n}$ |

SUMMARY

# Common Mistakes

### Naive shuffle

Swapping with `nextInt(n)` for every i is biased ($n^n$ paths, n! permutations). Use `nextInt(i + 1)`.

### "A random pivot eliminates the worst case"

No: the worst case is still $O(n^2)$. What changes is that no input is always bad, and the worst case becomes very unlikely.

### Confusing average and expected time

The average case assumes an input distribution; expected time is computed for each input over the algorithm's coin flips.

### Treating a Monte Carlo answer as certain

In Miller–Rabin, "prime" means only **probably** prime; the number of rounds k sets the error bound.

### Overflow

With `int`, `x * x % n` can overflow for n > 46,341. Use `long`, `Math.multiplyHigh` or `BigInteger.modPow`.

### new Random() on every call

Creates needless objects, and the seeds may be correlated. Share a single `Random`; in Karger, pick the edge **uniformly** (choosing a vertex first and then a neighbor is not uniform).

SUMMARY

# Sample Problems

1. Fisher–Yates is applied to `[1, 2, 3, 4]`; for i = 3, 2, 1 the drawn values of j are 1, 2, 0. What is the result?
2. For an array of 5 distinct elements, what is the probability that BogoSort finishes after the first shuffle, and what is the expected number of shuffles?
3. In randomized quicksort, with what probability are the smallest and largest elements compared? Two elements adjacent in sorted order?
4. Miller–Rabin: run the test on n = 25 with bases a = 7 and a = 2.
5. In a Monte Carlo run, 787 of 1000 points fell inside the quarter circle. What is the estimate of π?

▶ **Show answers**

SUMMARY

# Exercises

1. `partition` is called on `[3, 8, 2, 5, 1, 4, 7, 6]` and draws `r = 5` (pivot 4). What are the array after partitioning and the return value?
2. How many comparisons does quicksort make on a sorted 8-element array when the first element is the pivot?
3. How many Miller–Rabin rounds are needed to push the error probability below $10^{-6}$?
4. On a 6-vertex graph (success ≥ 1/15), how many times must Karger be repeated to bring the failure probability below 1/6?
5. Convert a Monte Carlo algorithm with success probability 1/3 and a verifiable result into a Las Vegas algorithm. What is the expected number of repetitions?

▶ **Show answers**

# References


- R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
- M. Mitzenmacher and E. Upfal, *Probability and Computing*, 2nd ed., Cambridge University Press, 2017.
- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., 2022 (Chapters 5, 7, 9, 31).
- C. A. R. Hoare, "Quicksort", *The Computer Journal* 5(1), 1962.
- M. O. Rabin, "Probabilistic Algorithm for Testing Primality", *Journal of Number Theory* 12(1), 1980.
- D. R. Karger, "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm", *SODA*, 1993.
- D. E. Knuth, *The Art of Computer Programming*, Vol. 2, 3rd ed., 1997 (Section 3.4.2: shuffling).

# The End

## Chapter 7: Randomized Algorithms

Next chapter: Chapter 8: Online Algorithms