Tutorial 5

Sorting algorithms: trade-offs, edge cases, and micro-benchmarks

18YZALG – Basics of Algorithmization, Summer Semester 2026

How today works

- We will compare five classic sorting algorithms.
- For each: **idea** → **properties** → **edge cases** → **measured examples**.
- Goal: learn to answer *"which sort should I use here?"* with reasons.

Ground rule

We care about **scaling and trade-offs**. One timing number is never the full story.

How today works

- We will compare five classic sorting algorithms.
- For each: **idea** → **properties** → **edge cases** → **measured examples**.
- Goal: learn to answer *“which sort should I use here?”* with reasons.

Ground rule

We care about **scaling and trade-offs**. One timing number is never the full story.

How today works

- We will compare five classic sorting algorithms.
- For each: **idea** → **properties** → **edge cases** → **measured examples**.
- Goal: learn to answer *“which sort should I use here?”* with reasons.

Ground rule

We care about **scaling and trade-offs**. One timing number is never the full story.

How today works

- We will compare five classic sorting algorithms.
- For each: **idea** → **properties** → **edge cases** → **measured examples**.
- Goal: learn to answer *"which sort should I use here?"* with reasons.

Ground rule

We care about **scaling and trade-offs**. One timing number is never the full story.

How today works

- We will compare five classic sorting algorithms.
- For each: **idea** $\rightarrow$ **properties** $\rightarrow$ **edge cases** $\rightarrow$ **measured examples**.
- Goal: learn to answer *"which sort should I use here?"* with reasons.

Ground rule

We care about **scaling and trade-offs**. One timing number is never the full story.

Today

1. Sorting problem + vocabulary (stable, in-place, adaptive).
2. Micro-benchmarking rules (how to measure fairly).
3. QuickSort and MergeSort (divide & conquer).
4. Insertion / Selection / Bubble (quadratic family, but with useful niches).
5. Summary: a practical decision table.

Today

1. Sorting problem + vocabulary (stable, in-place, adaptive).
2. Micro-benchmarking rules (how to measure fairly).
3. QuickSort and MergeSort (divide & conquer).
4. Insertion / Selection / Bubble (quadratic family, but with useful niches).
5. Summary: a practical decision table.

Today

1. Sorting problem + vocabulary (stable, in-place, adaptive).
2. Micro-benchmarking rules (how to measure fairly).
3. QuickSort and MergeSort (divide & conquer).
4. Insertion / Selection / Bubble (quadratic family, but with useful niches).
5. Summary: a practical decision table.

Today

1. Sorting problem + vocabulary (stable, in-place, adaptive).
2. Micro-benchmarking rules (how to measure fairly).
3. QuickSort and MergeSort (divide & conquer).
4. Insertion / Selection / Bubble (quadratic family, but with useful niches).
5. Summary: a practical decision table.

Today

1. Sorting problem + vocabulary (stable, in-place, adaptive).
2. Micro-benchmarking rules (how to measure fairly).
3. QuickSort and MergeSort (divide & conquer).
4. Insertion / Selection / Bubble (quadratic family, but with useful niches).
5. Summary: a practical decision table.

Why sorting shows up everywhere

- Preprocessing step: **sort once, query many times**. After sorting, operations like binary search, range queries, and grouping become straightforward and fast.
- Data work: ranking top-k items, deduplicating records, computing medians/percentiles, and performing merge-style joins all naturally rely on sorted order.
- Engineering benefit: a sorted sequence reveals patterns (runs, gaps, outliers), so debugging and validating intermediate results is often easier.
- Design benefit: many algorithms become simpler and more reliable when they can assume monotonic order in the input.

One-liner

Sorting is not just "put numbers in order" — it is often the cheapest way to impose **structure**, reduce downstream complexity, and unlock faster primitives.

Why sorting shows up everywhere

- Preprocessing step: **sort once, query many times**. After sorting, operations like binary search, range queries, and grouping become straightforward and fast.
- Data work: ranking top-k items, deduplicating records, computing medians/percentiles, and performing merge-style joins all naturally rely on sorted order.
- Engineering benefit: a sorted sequence reveals patterns (runs, gaps, outliers), so debugging and validating intermediate results is often easier.
- Design benefit: many algorithms become simpler and more reliable when they can assume monotonic order in the input.

One-liner

Sorting is not just "put numbers in order" — it is often the cheapest way to impose **structure**, reduce downstream complexity, and unlock faster primitives.

Why sorting shows up everywhere

- Preprocessing step: **sort once, query many times**. After sorting, operations like binary search, range queries, and grouping become straightforward and fast.
- Data work: ranking top-k items, deduplicating records, computing medians/percentiles, and performing merge-style joins all naturally rely on sorted order.
- Engineering benefit: a sorted sequence reveals patterns (runs, gaps, outliers), so debugging and validating intermediate results is often easier.
- Design benefit: many algorithms become simpler and more reliable when they can assume monotonic order in the input.

One-liner

Sorting is not just "put numbers in order" — it is often the cheapest way to impose **structure**, reduce downstream complexity, and unlock faster primitives.

Why sorting shows up everywhere

- Preprocessing step: **sort once, query many times**. After sorting, operations like binary search, range queries, and grouping become straightforward and fast.
- Data work: ranking top-k items, deduplicating records, computing medians/percentiles, and performing merge-style joins all naturally rely on sorted order.
- Engineering benefit: a sorted sequence reveals patterns (runs, gaps, outliers), so debugging and validating intermediate results is often easier.
- Design benefit: many algorithms become simpler and more reliable when they can assume monotonic order in the input.

One-liner

Sorting is not just "put numbers in order" — it is often the cheapest way to impose **structure**, reduce downstream complexity, and unlock faster primitives.

Why sorting shows up everywhere

- Preprocessing step: **sort once, query many times**. After sorting, operations like binary search, range queries, and grouping become straightforward and fast.
- Data work: ranking top-k items, deduplicating records, computing medians/percentiles, and performing merge-style joins all naturally rely on sorted order.
- Engineering benefit: a sorted sequence reveals patterns (runs, gaps, outliers), so debugging and validating intermediate results is often easier.
- Design benefit: many algorithms become simpler and more reliable when they can assume monotonic order in the input.

One-liner

Sorting is not just "put numbers in order" — it is often the cheapest way to impose **structure**, reduce downstream complexity, and unlock faster primitives.

Why sorting shows up everywhere

- Preprocessing step: **sort once, query many times**. After sorting, operations like binary search, range queries, and grouping become straightforward and fast.
- Data work: ranking top-k items, deduplicating records, computing medians/percentiles, and performing merge-style joins all naturally rely on sorted order.
- Engineering benefit: a sorted sequence reveals patterns (runs, gaps, outliers), so debugging and validating intermediate results is often easier.
- Design benefit: many algorithms become simpler and more reliable when they can assume monotonic order in the input.

One-liner

Sorting is not just "put numbers in order" — it is often the cheapest way to impose **structure**, reduce downstream complexity, and unlock faster primitives.

The sorting task (spec)

Specification

```
Input: list A of n comparable items
Output: permutation B of A such that
        B[0] <= B[1] <= ... <= B[n-1]

Optional constraints:
  - Stable? (keep relative order of equal keys)
  - In-place? (allowed extra memory?)
  - Adaptive? (gets faster on nearly-sorted inputs)
```

Vocabulary you will use to justify a choice

Stable

If $A[i]$ and $A[j]$ compare equal and $i < j$, then they appear in the same order in the output.

In-place

Uses $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ extra memory (beyond input).

Adaptive

Runs much faster if data is *already* (almost) sorted.

Vocabulary you will use to justify a choice

Stable

If $A[i]$ and $A[j]$ compare equal and $i < j$, then they appear in the same order in the output.

In-place

Uses $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ extra memory (beyond input).

Adaptive

Runs much faster if data is *already* (almost) sorted.

Benchmark focus for this tutorial

- We assume the standard measurement protocol from earlier tutorials.
- Today, the key variable is **input shape**: random, nearly sorted, already sorted, many equal keys.
- We compare **growth trends** and cross-over points, not one headline timing.
- For QuickSort, pivot policy is part of the algorithm and must be reported explicitly.

Interpretation lens

The goal is to explain **why** an algorithm wins on a dataset, not just to rank implementations.

Benchmark focus for this tutorial

- We assume the standard measurement protocol from earlier tutorials.
- Today, the key variable is **input shape**: random, nearly sorted, already sorted, many equal keys.
- We compare **growth trends** and cross-over points, not one headline timing.
- For QuickSort, pivot policy is part of the algorithm and must be reported explicitly.

Interpretation lens

The goal is to explain **why** an algorithm wins on a dataset, not just to rank implementations.

Benchmark focus for this tutorial

- We assume the standard measurement protocol from earlier tutorials.
- Today, the key variable is **input shape**: random, nearly sorted, already sorted, many equal keys.
- We compare **growth trends** and cross-over points, not one headline timing.
- For QuickSort, pivot policy is part of the algorithm and must be reported explicitly.

Interpretation lens

The goal is to explain **why** an algorithm wins on a dataset, not just to rank implementations.

Benchmark focus for this tutorial

- We assume the standard measurement protocol from earlier tutorials.
- Today, the key variable is **input shape**: random, nearly sorted, already sorted, many equal keys.
- We compare **growth trends** and cross-over points, not one headline timing.
- For QuickSort, pivot policy is part of the algorithm and must be reported explicitly.

Interpretation lens

The goal is to explain **why** an algorithm wins on a dataset, not just to rank implementations.

Benchmark focus for this tutorial

- We assume the standard measurement protocol from earlier tutorials.
- Today, the key variable is **input shape**: random, nearly sorted, already sorted, many equal keys.
- We compare **growth trends** and cross-over points, not one headline timing.
- For QuickSort, pivot policy is part of the algorithm and must be reported explicitly.

Interpretation lens

The goal is to explain **why** an algorithm wins on a dataset, not just to rank implementations.

Benchmark focus for this tutorial

- We assume the standard measurement protocol from earlier tutorials.
- Today, the key variable is **input shape**: random, nearly sorted, already sorted, many equal keys.
- We compare **growth trends** and cross-over points, not one headline timing.
- For QuickSort, pivot policy is part of the algorithm and must be reported explicitly.

Interpretation lens

The goal is to explain **why** an algorithm wins on a dataset, not just to rank implementations.

A minimal timing harness (Python)

Code

```
import time, gc

def time_one(sort_func, data, repeats=7):
    gc.disable()
    try:
        best = float('inf')
        for _ in range(repeats):
            A = data[:]              # same input
            t0 = time.perf_counter()
            sort_func(A)
            best = min(best, time.perf_counter() - t0)
        return best
    finally:
        gc.enable()
```

A useful mental model

- Comparison sorting has a hard lower bound: $\Omega(n \log n)$ comparisons.
- So $n \log n$ behavior (QuickSort/MergeSort family) is the right baseline for large, general inputs.
- Scale intuition: at $n = 10{,}000$, $n \log_2 n \approx 133{,}000$, while $n^2 = 100{,}000{,}000$.
- Constants and input shape still matter.

Decision lens

Use asymptotics first, then refine by data shape and constraints (stability, memory, worst-case risk).

A useful mental model

- Comparison sorting has a hard lower bound: $\Omega(n \log n)$ comparisons.
- So $n \log n$ behavior (QuickSort/MergeSort family) is the right baseline for large, general inputs.
- Scale intuition: at $n = 10{,}000$, $n \log_2 n \approx 133{,}000$, while $n^2 = 100{,}000{,}000$.
- Constants and input shape still matter.

Decision lens

Use asymptotics first, then refine by data shape and constraints (stability, memory, worst-case risk).

A useful mental model

- Comparison sorting has a hard lower bound: $\Omega(n \log n)$ comparisons.
- So $n \log n$ behavior (QuickSort/MergeSort family) is the right baseline for large, general inputs.
- Scale intuition: at $n = 10{,}000$, $n \log_2 n \approx 133{,}000$, while $n^2 = 100{,}000{,}000$.
- Constants and input shape still matter.

Decision lens

Use asymptotics first, then refine by data shape and constraints (stability, memory, worst-case risk).

A useful mental model

- Comparison sorting has a hard lower bound: $\Omega(n \log n)$ comparisons.
- So $n \log n$ behavior (QuickSort/MergeSort family) is the right baseline for large, general inputs.
- Scale intuition: at $n = 10{,}000$, $n \log_2 n \approx 133{,}000$, while $n^2 = 100{,}000{,}000$.
- Constants and input shape still matter.

Decision lens

Use asymptotics first, then refine by data shape and constraints (stability, memory, worst-case risk).

A useful mental model

- Comparison sorting has a hard lower bound: $\Omega(n \log n)$ comparisons.
- So $n \log n$ behavior (QuickSort/MergeSort family) is the right baseline for large, general inputs.
- Scale intuition: at $n = 10{,}000$, $n \log_2 n \approx 133{,}000$, while $n^2 = 100{,}000{,}000$.
- Constants and input shape still matter.

Decision lens

Use asymptotics first, then refine by data shape and constraints (stability, memory, worst-case risk).

A useful mental model

- Comparison sorting has a hard lower bound: $\Omega(n \log n)$ comparisons.
- So $n \log n$ behavior (QuickSort/MergeSort family) is the right baseline for large, general inputs.
- Scale intuition: at $n = 10{,}000$, $n \log_2 n \approx 133{,}000$, while $n^2 = 100{,}000{,}000$.
- Constants and input shape still matter.

Decision lens

Use asymptotics first, then refine by data shape and constraints (stability, memory, worst-case risk).

QuickSort (idea)

Divide & Conquer: split into smaller similar subproblems, solve recursively, then combine.

Divide & conquer via partition

Pick a **pivot**. Rearrange the array A so:

$$\{x \in A \mid x < p\} \;\mid\; p \;\mid\; \{x \in A \mid x \geq p\}$$

Then recursively sort the two sides.

- Average case is excellent, but the pivot choice matters.

QuickSort (idea)

Divide & Conquer: split into smaller similar subproblems, solve recursively, then combine.

Divide & conquer via partition

Pick a **pivot**. Rearrange the array A so:

$$\{x \in A \mid x < p\} \;\mid\; p \;\mid\; \{x \in A \mid x \geq p\}$$

Then recursively sort the two sides.

- Average case is excellent, but the pivot choice matters.

QuickSort (pseudocode)

Pseudocode (random pivot)

```
QUICKSORT(A, lo, hi):          # sort A[lo:hi)
  if hi - lo <= 1: return
  p = random index in [lo, hi)
  pivot = A[p]
  i = PARTITION(A, lo, hi, pivot)
  QUICKSORT(A, lo, i)
  QUICKSORT(A, i+1, hi)
```

Invariant (partition)

After partition: all indices $< i$ hold values $<$ pivot, and all indices $> i$ hold values $\geq$ pivot.

QuickSort (pseudocode)

Pseudocode (random pivot)

```
QUICKSORT(A, lo, hi):          # sort A[lo:hi)
  if hi - lo <= 1: return
  p = random index in [lo, hi)
  pivot = A[p]
  i = PARTITION(A, lo, hi, pivot)
  QUICKSORT(A, lo, i)
  QUICKSORT(A, i+1, hi)
```

Invariant (partition)

After partition: all indices $< i$ hold values $<$ pivot, and all indices $> i$ hold values $\geq$ pivot.

QuickSort (properties)

- Average time: $\mathcal{O}(n \log n)$
- Worst time: $\mathcal{O}(n^2)$ (bad pivots)
- Extra memory: recursion stack $\mathcal{O}(\log n)$ average
- Not stable (standard in-place partition)

When it shines

Fast in practice on random-ish data; in-place; good locality.

When it hurts

Adversarial order + unlucky pivot strategy; many equal keys without 3-way partition.

Measured example: random input (all 5 algorithms)

Random integers (best-of-7; pure-Python reference implementations)

n	Quick	Merge	Insertion	Selection	Bubble
400	**0.280**	0.336	1.34	1.66	3.46
800	**0.615**	0.701	6.04	6.99	15.8
1600	**1.34**	1.60	26.3	28.7	69.3
3200	**2.79**	3.52	108	115	295
4000	**3.72**	4.63	177	187	486

Times are **milliseconds**. Interpretation: on random data, QuickSort / MergeSort scale smoothly, while quadratic sorts explode.

Edge case: "bad pivot" QuickSort

Already sorted input: pivot = first element (a common mistake)

n	Quick (first pivot)	Quick (random)
200	0.787	0.116
400	3.02	0.266
800	9.21	0.588

Takeaway

Pivot strategy is part of the algorithm. The same "QuickSort" can be great or disastrous.

Edge case: “bad pivot” QuickSort

Already sorted input: pivot = first element (a common mistake)

n	Quick (first pivot)	Quick (random)
200	0.787	0.116
400	3.02	0.266
800	9.21	0.588

Takeaway

Pivot strategy is part of the algorithm. The same “QuickSort” can be great or disastrous.

QuickSort: practical fixes

- **Randomized pivot:** pick a random index in each subarray to avoid predictable worst-case behavior on sorted or adversarial input.
- **Median-of-three pivot:** use the median of first/middle/last values to reduce highly unbalanced splits on partially ordered data.
- **3-way partitioning:** split into <, =, and > regions so equal keys are handled in one pass (fewer useless recursive calls).
- **Depth guard:** if recursion depth becomes too large, fall back to HeapSort to guarantee $\mathcal{O}(n \log n)$ worst-case time.

MergeSort (idea)

Divide & conquer via merging

Split the list in half, recursively sort both halves, then **merge** two sorted lists into one.

- The merge step is linear and very predictable.
- Standard MergeSort is stable.

MergeSort (idea)

Divide & conquer via merging

Split the list in half, recursively sort both halves, then **merge** two sorted lists into one.

- The merge step is linear and very predictable.
- Standard MergeSort is stable.

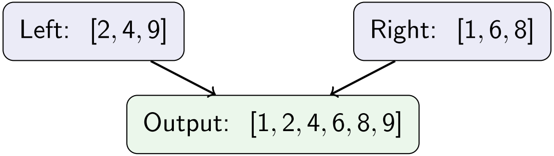
MergeSort (idea)

Divide & conquer via merging

Split the list in half, recursively sort both halves, then **merge** two sorted lists into one.

- The merge step is linear and very predictable.
- Standard MergeSort is stable.

Merge step (picture)



Always take the smaller front element.

MergeSort (pseudocode)

Pseudocode

```
MERGESORT(A):
  if len(A) <= 1: return A
  mid = len(A)//2
  L = MERGESORT(A[:mid])
  R = MERGESORT(A[mid:])
  return MERGE(L, R)

MERGE(L, R):
  i=j=0; out=[]
  while i<len(L) and j<len(R):
    if L[i] <= R[j]: out.append(L[i]); i+=1
    else:            out.append(R[j]); j+=1
  append remaining tail
  return out
```

MergeSort (properties)

- Time: $\mathcal{O}(n \log n)$ in **best/average/worst**
- Extra memory: typically $\mathcal{O}(n)$
- Stable (easy to keep)

Where it wins

Guaranteed worst-case; stable multi-key sorting; great for linked lists; foundation for external sorting.

Where it loses

Needs extra memory; constant factors can be higher than QuickSort.

Stability matters (tiny example)

Scenario: sort by `grade`, keep original order within equal grades

Input (arrival order):

```
[(Alice, 90), (Bob, 90), (Chad, 75), (Dana, 90)]
```

A **stable** sort by grade outputs:

```
[(Chad, 75), (Alice, 90), (Bob, 90), (Dana, 90)]
```

Takeaway

If you plan to do **multi-key sorting** (sort by secondary key, then primary), stability is essential.

Stability matters (tiny example)

Scenario: sort by `grade`, keep original order within equal grades

Input (arrival order):

```
[(Alice, 90), (Bob, 90), (Chad, 75), (Dana, 90)]
```

A **stable** sort by grade outputs:

```
[(Chad, 75), (Alice, 90), (Bob, 90), (Dana, 90)]
```

Takeaway

If you plan to do **multi-key sorting** (sort by secondary key, then primary), stability is essential.

Stability matters (tiny example)

Scenario: sort by `grade`, keep original order within equal grades

Input (arrival order):

```
[(Alice, 90), (Bob, 90), (Chad, 75), (Dana, 90)]
```

A **stable** sort by grade outputs:

```
[(Chad, 75), (Alice, 90), (Bob, 90), (Dana, 90)]
```

Takeaway

If you plan to do **multi-key sorting** (sort by secondary key, then primary), stability is essential.

Insertion Sort (idea)

Grow a sorted prefix

Maintain that $A[0 : i)$ is sorted. Insert $A[i]$ into the right position by shifting bigger items.

- **Stable** if implemented with strict $>$: equal keys are not swapped past each other, so original relative order is preserved.
- **In-place** with $\mathcal{O}(1)$ extra memory: it only shifts values inside the same array and keeps implementation overhead low.
- Adaptive: running time is linked to the number of inversions, so nearly sorted inputs are close to linear in practice.

Insertion Sort (idea)

Grow a sorted prefix

Maintain that $A[0:i)$ is sorted. Insert $A[i]$ into the right position by shifting bigger items.

- **Stable** if implemented with strict $>$: equal keys are not swapped past each other, so original relative order is preserved.
- **In-place** with $\mathcal{O}(1)$ extra memory: it only shifts values inside the same array and keeps implementation overhead low.
- Adaptive: running time is linked to the number of inversions, so nearly sorted inputs are close to linear in practice.

Insertion Sort (pseudocode)

Pseudocode

```
INSERTION-SORT(A):
  for i in 1..n-1:
    x = A[i]
    j = i-1
    while j>=0 and A[j] > x:
      A[j+1] = A[j]      # shift right
      j -= 1
    A[j+1] = x
```

Insertion Sort (properties)

- Best-case (already sorted): $\mathcal{O}(n)$
- Worst-case (reverse sorted): $\mathcal{O}(n^2)$
- Extra memory: $\mathcal{O}(1)$
- Often used as a **base case** in hybrid sorts.

When it wins

Small n or nearly sorted data (few inversions).

When it loses

Large random data: quadratic behavior dominates.

Measured example: nearly sorted input (1% swaps)

Nearly sorted integers (best-of-7; ms)

n	Quick	Merge	Insertion	Selection	Bubble
400	0.277	0.260	**0.0490**	1.77	1.60
800	0.589	0.571	**0.241**	7.39	8.57
1600	1.35	1.27	**0.610**	29.1	32.8
3200	2.80	2.76	2.68	121	174
6400	5.94	6.04	13.4	483	733

Interpretation: InsertionSort is great when "almost sorted" really means "few inversions".

Selection Sort (idea)

Repeatedly pick the minimum

For position i, find the minimum of $A[i : n)$ and swap it into $A[i]$.

- Always does $\Theta(n^2)$ comparisons (input order does not help).
- In-place with only $\mathcal{O}(n)$ swaps.
- Usually **not stable** (swap breaks equal-key order).

Selection Sort (idea)

Repeatedly pick the minimum

For position i, find the minimum of $A[i:n)$ and swap it into $A[i]$.

- Always does $\Theta(n^2)$ comparisons (input order does not help).
- In-place with only $\mathcal{O}(n)$ swaps.
- Usually **not stable** (swap breaks equal-key order).

Selection Sort (idea)

Repeatedly pick the minimum

For position i, find the minimum of $A[i:n)$ and swap it into $A[i]$.

- Always does $\Theta(n^2)$ comparisons (input order does not help).
- In-place with only $\mathcal{O}(n)$ swaps.
- Usually **not stable** (swap breaks equal-key order).

Selection Sort (idea)

Repeatedly pick the minimum

For position i, find the minimum of $A[i:n)$ and swap it into $A[i]$.

- Always does $\Theta(n^2)$ comparisons (input order does not help).
- In-place with only $\mathcal{O}(n)$ swaps.
- Usually **not stable** (swap breaks equal-key order).

Selection Sort: when could you justify it?

- When **writes are expensive** (e.g., swapping large records on slow storage).
- When you need a very small, predictable piece of code.
- Educational value: makes the n^2 cost painfully clear.

Otherwise

If writes are cheap (typical RAM), there is almost always a better choice.

Selection Sort: when could you justify it?

- When **writes are expensive** (e.g., swapping large records on slow storage).
- When you need a very small, predictable piece of code.
- Educational value: makes the n^2 cost painfully clear.

Otherwise

If writes are cheap (typical RAM), there is almost always a better choice.

Measured example: selection minimizes swaps (writes)

Random integers: number of data movements (not time)

n	Selection swaps	Bubble swaps	Insertion shifts
100	94	2220	2220
200	198	10536	10536
400	392	41772	41772
800	791	161039	161039

Interpretation

If **writes are expensive** (e.g., swapping huge records on slow storage), SelectionSort's $\Theta(n)$ swaps can beat methods that do $\Theta(n^2)$ data movements.

Measured example: selection minimizes swaps (writes)

Random integers: number of data movements (not time)

n	Selection swaps	Bubble swaps	Insertion shifts
100	94	2220	2220
200	198	10536	10536
400	392	41772	41772
800	791	161039	161039

Interpretation

If **writes are expensive** (e.g., swapping huge records on slow storage), SelectionSort's $\Theta(n)$ swaps can beat methods that do $\Theta(n^2)$ data movements.

Bubble Sort (idea)

Swap adjacent inversions

Repeatedly scan the array and swap any adjacent out-of-order pair.

- Stable (only swaps adjacent items).
- Quadratic worst-case: $\mathcal{O}(n^2)$.
- With an "early exit" flag, best-case is $\mathcal{O}(n)$.

Bubble Sort (idea)

Swap adjacent inversions

Repeatedly scan the array and swap any adjacent out-of-order pair.

- Stable (only swaps adjacent items).
- Quadratic worst-case: $\mathcal{O}(n^2)$.
- With an "early exit" flag, best-case is $\mathcal{O}(n)$.

Bubble Sort (idea)

Swap adjacent inversions

Repeatedly scan the array and swap any adjacent out-of-order pair.

- Stable (only swaps adjacent items).
- Quadratic worst-case: $\mathcal{O}(n^2)$.
- With an "early exit" flag, best-case is $\mathcal{O}(n)$.

Bubble Sort (idea)

Swap adjacent inversions

Repeatedly scan the array and swap any adjacent out-of-order pair.

- Stable (only swaps adjacent items).
- Quadratic worst-case: $\mathcal{O}(n^2)$.
- With an "early exit" flag, best-case is $\mathcal{O}(n)$.

Bubble Sort (early exit variant)

Pseudocode

```
BUBBLE-SORT(A):
  for pass in 1..n:
    swapped = False
    for j in 0..n-pass-1:
      if A[j] > A[j+1]:
        swap(A[j], A[j+1])
        swapped = True
    if not swapped:
      return   # already sorted
```

Edge case: already sorted input

Already sorted integers (best-of-7; ms)

n	Bubble (early)	Insertion	Merge	Quick
800	**0.0255**	0.0445	0.517	0.593
1600	**0.0545**	0.0904	1.12	1.35
3200	**0.110**	0.187	2.35	2.67
6400	**0.224**	0.395	4.84	5.79
12800	**0.446**	0.765	10.6	13.6

Interpretation

BubbleSort becomes a linear-time **sortedness check**. But on real unsorted data it is still a poor general sort.

Edge case: already sorted input

Already sorted integers (best-of-7; ms)

n	Bubble (early)	Insertion	Merge	Quick
800	**0.0255**	0.0445	0.517	0.593
1600	**0.0545**	0.0904	1.12	1.35
3200	**0.110**	0.187	2.35	2.67
6400	**0.224**	0.395	4.84	5.79
12800	**0.446**	0.765	10.6	13.6

Interpretation

BubbleSort becomes a linear-time **sortedness check**. But on real unsorted data it is still a poor general sort.

Bubble Sort: where it does not belong

- General-purpose sorting of medium/large arrays.
- Any situation with a meaningful time limit.
- When you can use InsertionSort for nearly-sorted inputs (usually strictly better).

Bubble Sort: where it does not belong

- General-purpose sorting of medium/large arrays.
- Any situation with a meaningful time limit.
- When you can use InsertionSort for nearly-sorted inputs (usually strictly better).

Bubble Sort: where it does not belong

- General-purpose sorting of medium/large arrays.
- Any situation with a meaningful time limit.
- When you can use InsertionSort for nearly-sorted inputs (usually strictly better).

Comparison table (properties)

Algorithm	Stable	In-place	Adaptive	Worst-case time
QuickSort (std)	no	yes	no	$\mathcal{O}(n^2)$
MergeSort	yes	no (extra n)	no	$\mathcal{O}(n \log n)$
Insertion	yes	yes	**yes**	$\mathcal{O}(n^2)$
Selection	no (typically)	yes	no	$\mathcal{O}(n^2)$
Bubble (early)	yes	yes	some (best-case)	$\mathcal{O}(n^2)$

Choosing in practice (rules of thumb)

- Need stability (multi-key, preserve order) $\Rightarrow$ MergeSort-family.
- Need in-place and great average speed $\Rightarrow$ QuickSort with good pivots.
- Data is almost sorted or n is tiny $\Rightarrow$ InsertionSort.
- Writes are expensive $\Rightarrow$ SelectionSort can be justified (rare).
- Already sorted detection $\Rightarrow$ Bubble early-exit (or just one linear pass check).

Meta-rule

Pick based on **constraints**: size, memory, stability, data "shape", and worst-case risk.

Choosing in practice (rules of thumb)

- Need stability (multi-key, preserve order) $\Rightarrow$ MergeSort-family.
- Need in-place and great average speed $\Rightarrow$ QuickSort with good pivots.
- Data is almost sorted or n is tiny $\Rightarrow$ InsertionSort.
- Writes are expensive $\Rightarrow$ SelectionSort can be justified (rare).
- Already sorted detection $\Rightarrow$ Bubble early-exit (or just one linear pass check).

Meta-rule

Pick based on **constraints**: size, memory, stability, data "shape", and worst-case risk.

Quick check (predict the winner)

For each scenario, name a likely good choice (and why)

1. $n = 200$, almost sorted, only a few swaps.
2. $n = 50000$, random order, memory is fine.
3. You sort records by (`last_name, first_name`) using two passes.
4. You suspect inputs may be adversarial (someone tries to slow you down).

Likely answers: 1) InsertionSort; 2) QuickSort (with good/random pivot) or MergeSort; 3) MergeSort; 4) MergeSort.

Quick check (predict the winner)

For each scenario, name a likely good choice (and why)

1. $n = 200$, almost sorted, only a few swaps.
2. $n = 50000$, random order, memory is fine.
3. You sort records by (`last_name, first_name`) using two passes.
4. You suspect inputs may be adversarial (someone tries to slow you down).

Likely answers: 1) InsertionSort; 2) QuickSort (with good/random pivot) or MergeSort; 3) MergeSort; 4) MergeSort.

Quick check (predict the winner)

For each scenario, name a likely good choice (and why)

1. $n = 200$, almost sorted, only a few swaps.
2. $n = 50000$, random order, memory is fine.
3. You sort records by `(last_name, first_name)` using two passes.
4. You suspect inputs may be adversarial (someone tries to slow you down).

Likely answers: 1) InsertionSort; 2) QuickSort (with good/random pivot) or MergeSort; 3) MergeSort; 4) MergeSort.

Quick check (predict the winner)

For each scenario, name a likely good choice (and why)

1. $n = 200$, almost sorted, only a few swaps.
2. $n = 50000$, random order, memory is fine.
3. You sort records by (`last_name, first_name`) using two passes.
4. You suspect inputs may be adversarial (someone tries to slow you down).

Likely answers: 1) InsertionSort; 2) QuickSort (with good/random pivot) or MergeSort; 3) MergeSort; 4) MergeSort.

Quick check (predict the winner)

For each scenario, name a likely good choice (and why)

1. $n = 200$, almost sorted, only a few swaps.
2. $n = 50000$, random order, memory is fine.
3. You sort records by `(last_name, first_name)` using two passes.
4. You suspect inputs may be adversarial (someone tries to slow you down).

Likely answers: 1) InsertionSort; 2) QuickSort (with good/random pivot) or MergeSort; 3) MergeSort; 4) MergeSort.

Quick check (predict the winner)

For each scenario, name a likely good choice (and why)

1. $n = 200$, almost sorted, only a few swaps.
2. $n = 50000$, random order, memory is fine.
3. You sort records by `(last_name, first_name)` using two passes.
4. You suspect inputs may be adversarial (someone tries to slow you down).

Likely answers: 1) InsertionSort; 2) QuickSort (with good/random pivot) or MergeSort; 3) MergeSort; 4) MergeSort.

Wrap-up

- "Same problem" does not mean "same best algorithm".
- Benchmarks should support a story about **scaling**.
- Next: **Heap** data structure and HeapSort (worst-case guarantee; priority-queue power).