Data Structures

Week 5 – Sorting I: bubble sort, selection sort and insertion sort

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Binary search:** O(log n) – but only on a **sorted** array!
- Moving elements in an array is costly: insertion, deletion **O(n)**
- Comparing strings: **lexicographic** order

Questions for today

- How can we put an array in order?
- How many comparisons and moves does it take?
- When should we choose which simple sort?

The sorting problem

Given: a sequence of **n** elements that can be compared (≤ relation).

Task: rearrange the elements so that A[1] ≤ A[2] ≤ … ≤ A[n].

Why is it important?

- **Fast search:** binary search on a sorted array – O(log n)
- **Grouping:** equal elements end up next to each other (duplicates, frequencies)
- **Presentation:** alphabetical lists, rankings, chronological order
- By some estimates, a considerable part of all computing time is spent on sorting

Concepts

- **Comparison:** examining the relation of two elements (`A[i] < A[j]`) – the main cost of sorting
- **Move / swap:** relocating elements – one swap is 3 assignments
- **In-place sorting:** needs only O(1) extra memory
- **Stable sorting:** elements with equal keys **keep** their original order
- **Comparison sort:** decides the order of the elements by comparisons only

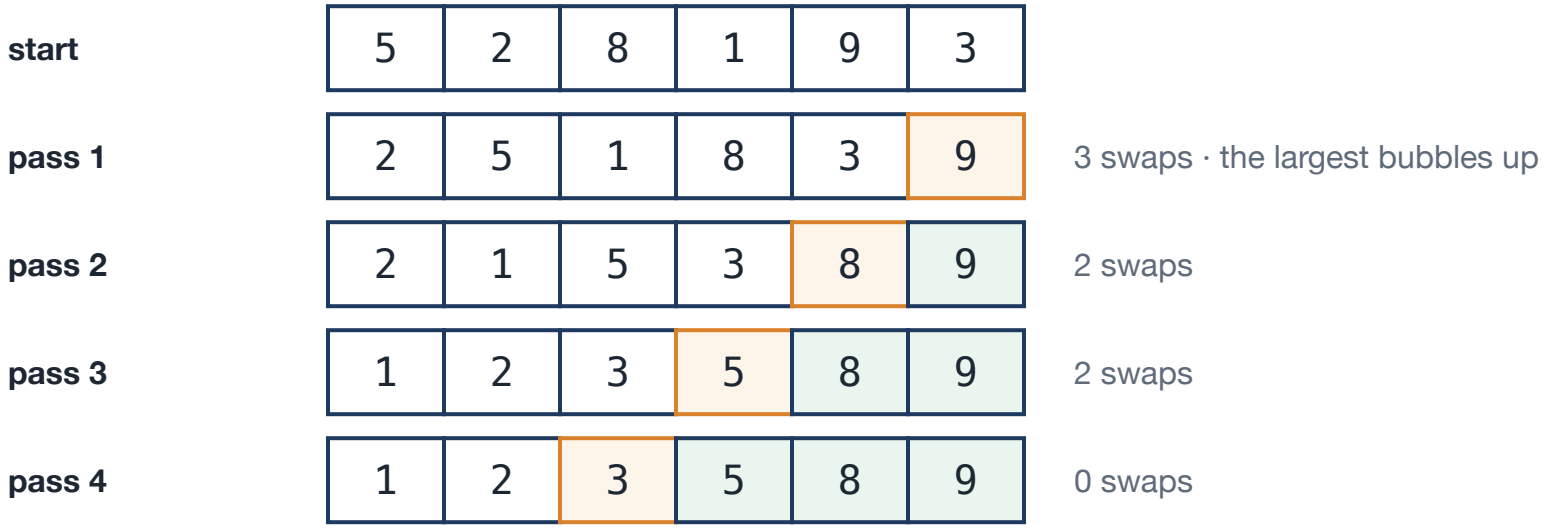
> Today we learn three **simple** sorting algorithms with $O(n^2)$ steps. The faster O(n log n) methods come in week 13.

Bubble sort

Bubble sort – the idea

- Walk through the array and compare **neighbouring** elements
- If they are in the wrong order, **swap** them
- After one pass the **largest** element is at the end of the array (it “bubbles up”)
- Repeat on the unsorted part

Bubble sort: swapping adjacent pairs in each pass



Bubble sort – algorithm

Pseudocode

```
Procedure BubbleSort(A, n)
  For i ← n down to 2
    swapped ← false
    For j ← 1 to i − 1
      If A[j] > A[j + 1] then
        swap(A[j], A[j + 1])
        swapped ← true
      End if
    End for
    If not swapped then
      exit the loop
  End for
End procedure
```

Java

```java
for (int i = n - 1; i >= 1; i--) {
    boolean swapped = false;
    for (int j = 0; j < i; j++)
        if (a[j] > a[j + 1]) {
            int t = a[j];
            a[j] = a[j + 1];
            a[j + 1] = t;
            swapped = true;
        }
    if (!swapped) break;
}
```

Go

```go
if a[j] > a[j+1] {
    a[j], a[j+1] = a[j+1], a[j]
    swapped = true
}
```

Bubble sort – analysis

Case	Comparisons	Swaps
Best (already sorted)	$n - 1$	0
Worst (reversed)	$n(n - 1) / 2$	$n(n - 1) / 2$
Average	$\sim n^2 / 2$	$\sim n^2 / 4$

- Steps: **$O(n^2)$** · on a sorted array, with the `swapped` flag: **$O(n)$**
- Sorts **in place**, **stable** (never swaps equal elements)
- The number of swaps = the number of **inversions** (pairs in the wrong order)

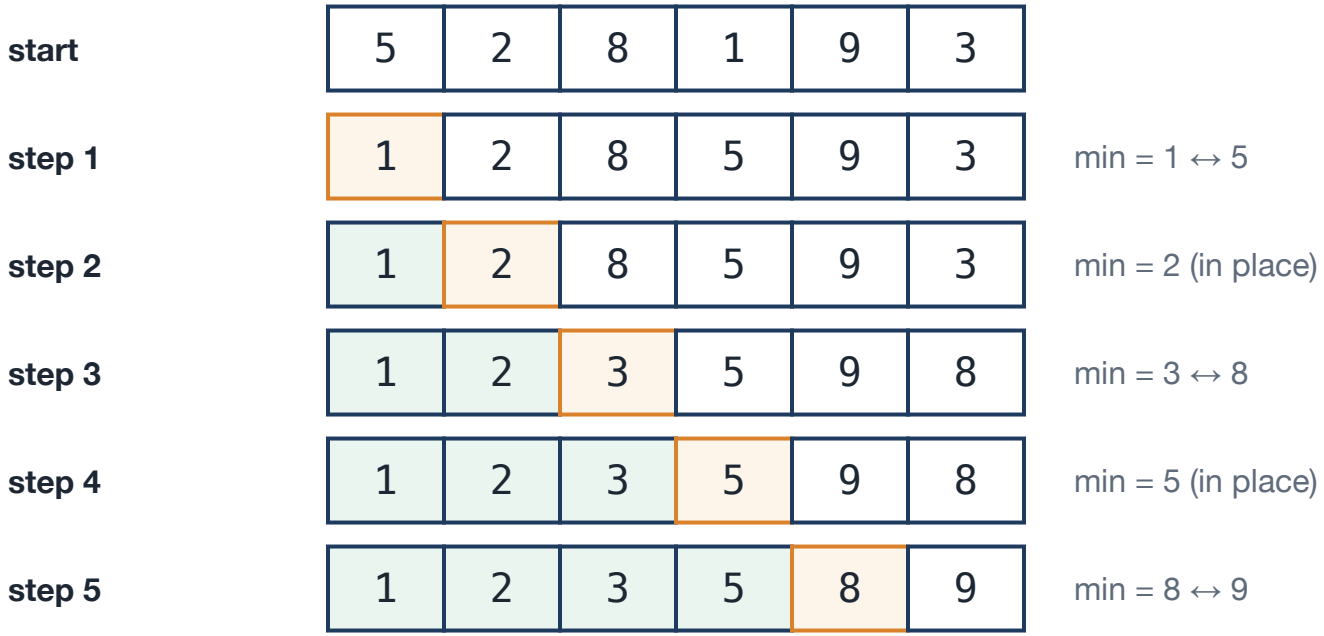
> In practice it is the slowest of the simple sorts – it makes a lot of swaps.

Selection sort

Selection sort – the idea

- Find the **smallest** element and swap it into the **first** position
- In the remaining (unsorted) part find the smallest and put it into the **second** position
- … and so on

Selection sort: the minimum goes to the front of the unsorted part



Selection sort – algorithm

Pseudocode

```
Procedure SelectionSort(A, n)
  For i ← 1 to n − 1
    min ← i
    For j ← i + 1 to n
      If A[j] < A[min] then
        min ← j
      End if
    End for
    swap(A[i], A[min])
  End for
End procedure
```

The inner loop is a **minimum selection** (week 2: maximum selection).

Java

```java
for (int i = 0; i < n − 1; i++) {
    int min = i;
    for (int j = i + 1; j < n; j++)
        if (a[j] < a[min]) min = j;
    int t = a[i];
    a[i] = a[min];
    a[min] = t;
}
```

Go

```go
for i := 0; i < len(a)−1; i++ {
    min := i
    for j := i + 1; j < len(a); j++ {
        if a[j] < a[min] {
            min = j
        }
    }
    a[i], a[min] = a[min], a[i]
}
```

Selection sort – analysis

Case	Comparisons	Swaps
Every case	n(n – 1) / 2	n – 1

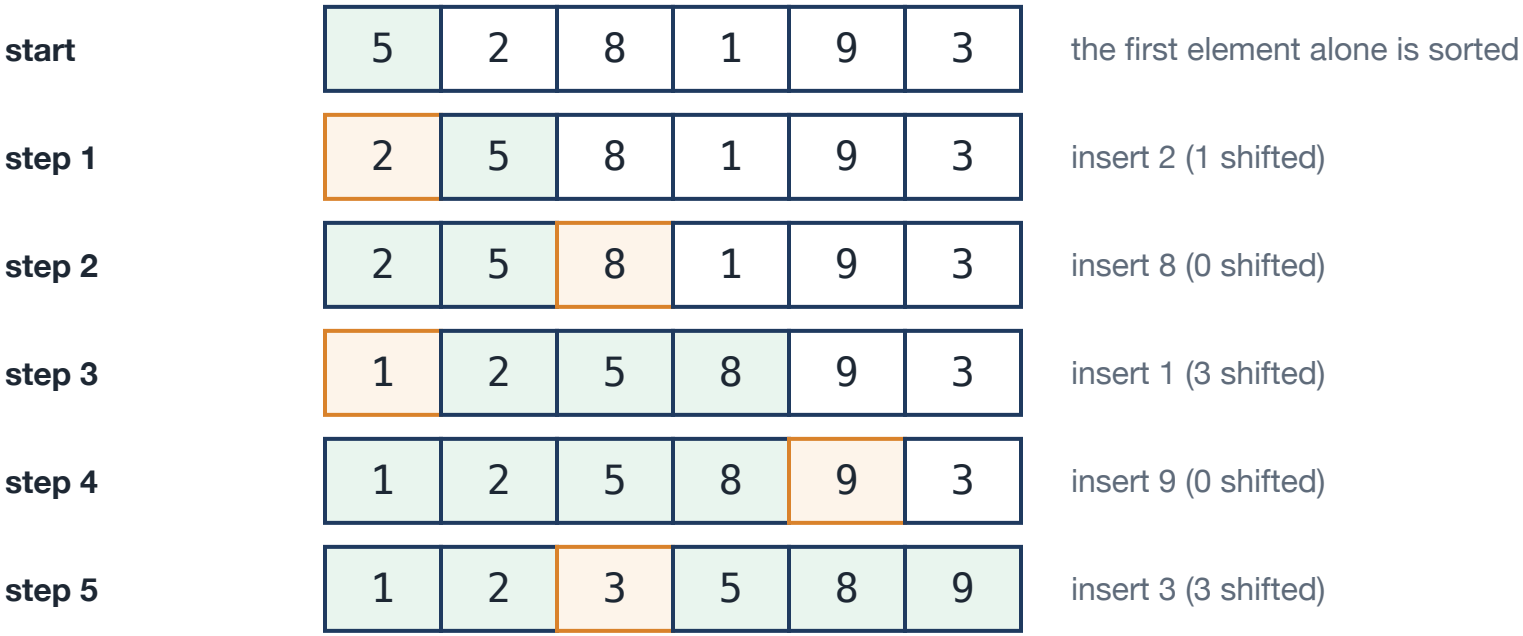
- Steps: **always $O(n^2)$** – it scans the rest even on a sorted array
- Only **n – 1 swaps** – useful when moving is very expensive (large records)
- Sorts **in place**
- **Not stable:** a long-distance swap may jump over an equal element
  `[5a, 5b, 2]` → `[2, 5b, 5a]`

Insertion sort

Insertion sort – the idea

- We sort the way we sort playing cards in our hand: the next card is **inserted** into its place among the already sorted cards
- The beginning of the array is sorted; the next element is shifted left until it reaches its place

Insertion sort: inserting the next element into the sorted part



Insertion sort – algorithm

Pseudocode

```
Procedure InsertionSort(A, n)
  For i ← 2 to n
    x ← A[i]
    j ← i − 1
    While j ≥ 1 and A[j] > x
      A[j + 1] ← A[j]
      j ← j − 1
    End while
    A[j + 1] ← x
  End for
End procedure
```

It does not swap but **shifts**: one step is one assignment.

Java

```java
for (int i = 1; i < n; i++) {
    int x = a[i];
    int j = i - 1;
    while (j >= 0 && a[j] > x) {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = x;
}
```

Go

```go
for i := 1; i < len(a); i++ {
    x, j := a[i], i-1
    for j >= 0 && a[j] > x {
        a[j+1] = a[j]
        j--
    }
    a[j+1] = x
}
```

Insertion sort – analysis

Case	Comparisons	Moves
Best (already sorted)	$n - 1$	$2(n - 1)$
Worst (reversed)	$n(n - 1) / 2$	$\sim n^2 / 2$
Average	$\sim n^2 / 4$	$\sim n^2 / 4$

- Steps: **$O(n^2)$**, on a sorted or **nearly sorted** array **$O(n)$**
- Sorts **in place**, **stable** (`A[j] > x` , not `≥` !)
- **Online:** it can sort the elements in the order they arrive
- The fastest on small arrays ($n < 20$) – professional sorts (Java, Go libraries) use it for small parts

Comparison

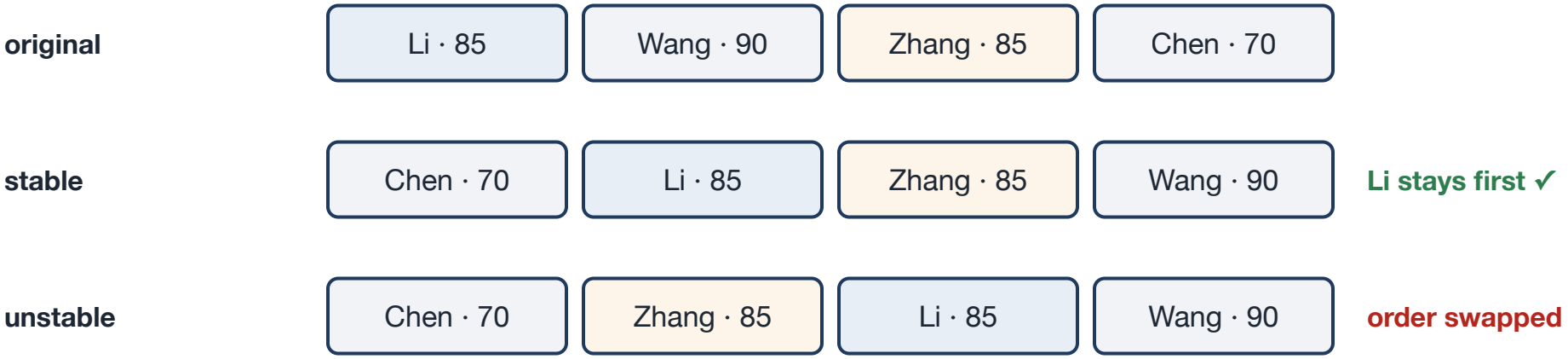
Measurements (n = 1000)

Number of comparisons / moves (`SortCompare` program):

	random	sorted	reversed
bubble	499 004 / 750 423	**999 / 0**	499 500 / 1 498 500
selection	499 500 / **2 997**	499 500 / 2 997	499 500 / 2 997
insertion	**251 585** / 252 589	**999 / 1 998**	499 500 / 501 498

> Move = assignment (one swap is 3 assignments). Note: selection sort always takes **the same** number of steps, insertion sort is **linear** on sorted data.

Stability



- **Why does it matter?** If we sort first by name and then by score with a **stable** sort, students with equal scores stay in alphabetical order
- Stable: bubble, insertion · Not stable: selection

Which one should we choose?

	Bubble	Selection	Insertion
Worst case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Best case	$O(n)$	$O(n^2)$	$O(n)$
Swaps / moves	many	**few (n – 1)**	medium
Stable	yes	no	yes
In place	yes	yes	yes

- In general: **insertion sort** – simple, fast on nearly sorted data
- When moving is expensive: **selection sort**
- Bubble sort: mainly for teaching purposes

> None of them is good for large **n**: for $n = 10^6$, $\sim 10^{12}$ steps. Solution: **merge sort** and **quicksort** (week 13), $O(n \log n)$.

Built-in sorting

Java

```java
int[] a = {5, 2, 8, 1};
Arrays.sort(a);   // primitives: Dual-Pivot Quicksort

String[] s = {"Wuhan", "Eger"};
Arrays.sort(s);   // objects: TimSort (stable)

list.sort(Comparator
    .comparing(Student::score));
```

Go

```go
a := []int{5, 2, 8, 1}
slices.Sort(a)     // pdqsort

sort.Strings(s)

slices.SortStableFunc(st,
    func(x, y Student) int {
        return x.Score - y.Score
    })
```

In practice we use the library sort – but **we must know how it works** and how fast it is in which case.

Summary

- **Bubble:** swapping neighbours, the largest goes to the end in each pass · $O(n^2)$, stable
- **Selection:** the minimum goes to the front · always $O(n^2)$, only $n - 1$ swaps, not stable
- **Insertion:** the next element is inserted into the sorted part · $O(n^2)$, $O(n)$ on sorted data, stable
- All three sort **in place**
- **Stability:** the original order of elements with equal keys is preserved

Next week: the set data structure – boolean vector, unsorted and sorted representation, set operations.

Review questions

1. Sort the array `[4, 1, 3, 2]` on paper with all three methods! Write down the states step by step!
2. True or false: selection sort finishes in O(n) steps on a sorted array.
3. Which sort makes the fewest swaps? How many?
4. True or false: bubble sort is stable.
5. How many comparisons does insertion sort make on a reversed 10-element array?
6. What happens if insertion sort uses `A[j] ≥ x` instead of `A[j] > x`?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Simple sorts	`SimpleSorts.java`	`w05/simplesorts`	all three sorts, printed step by step
Comparison	`SortCompare.java`	`w05/sortcompare`	comparisons, moves, time on different inputs

Run the `SortCompare` program! Which sort depends on the input, and which does not? When **n** doubles, how does the running time change?

✎ Exercises

1. **Is it sorted?** Decide in O(n) steps whether an array is sorted in increasing order!
2. **Descending order:** rewrite insertion sort for descending order!
3. **Strings:** sort an array of city names with selection sort!
4. **Records:** sort students (name, score) by score **stably**!
5. **k smallest:** find the **k** smallest elements of an array with the first **k** steps of selection sort! How many steps does it take?
6. **Counting sort:** sort exam grades (1–5) without comparisons, using a 6-element counting array! How many steps does it take?
7. **Inversions:** count the pairs in the wrong order in an array! How is this related to bubble sort?

Solutions: `Exercises05.java`, `w05/exercises`

✎ Homework

1. **Cocktail sort:** a variant of bubble sort that goes alternately left to right and right to left. Implement it, and measure how many comparisons it saves on the array `[2, 3, 4, 5, 1]`!
2. **Binary insertion sort:** find the insertion position with binary search (week 2)! How do the numbers of comparisons and moves change?
3. Give an example proving that selection sort is **not stable**!
4. Sort the students **primarily by score** in descending order, **by name for equal scores**! (Hint: two stable sorts one after the other, or a composite comparison.)