

Course Algorithms and Data Structures · Lecture 10

Simple Sorting: Selection, Bubble and Insertion Sort

Textbook: A. Levitin, Sections 3.1, 4.1

Sorting

Put items in order: numbers, names, products by price. One of the most frequent tasks in computing, and binary search needs it.

Today: three simple algorithms, all $O(n^2)$. We sort the same array with each (Levitin's example):

89, 45, 68, 90, 29, 34, 17

Selection sort

Find the smallest element and put it first; find the smallest of the rest and put it second; and so on.

```
ALGORITHM SelectionSort(A[0..n − 1])
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min]
                min ← j
        swap A[i] and A[min]
```

Selection sort, pass by pass

start	89	45	68	90	29	34	17
pass 1	**17**	45	68	90	29	34	89
pass 2	17	**29**	68	90	45	34	89
pass 3	17	29	**34**	90	45	68	89
pass 4	17	29	34	**45**	90	68	89
pass 5	17	29	34	45	**68**	90	89
pass 6	17	29	34	45	68	**89**	90

- Grey part: sorted and final
- Bold: the element just put in place
- Pass 1: 17 swaps with 89
- Pass 2: 29 goes to position 1

Selection sort: how many comparisons?

$$C(n) = (n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$$

- For 7 elements: 21 comparisons
- **Always** the same, even for a sorted array
- Only **n – 1 swaps**: good when moving elements is expensive

Bubble sort

Compare **neighbors** and swap them if they are in the wrong order. Each pass moves the largest element to the end.

```
ALGORITHM BubbleSort(A[0..n − 1])
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if A[j + 1] < A[j]
                swap A[j] and A[j + 1]
```

Bubble sort, pass by pass

start	89	45	68	90	29	34	17
pass 1	45	68	89	29	34	17	**90**
pass 2	45	68	29	34	17	**89**	90
pass 3	45	29	34	17	**68**	89	90
pass 4	29	34	17	**45**	68	89	90
pass 5	29	17	**34**	45	68	89	90
pass 6	17	**29**	34	45	68	89	90

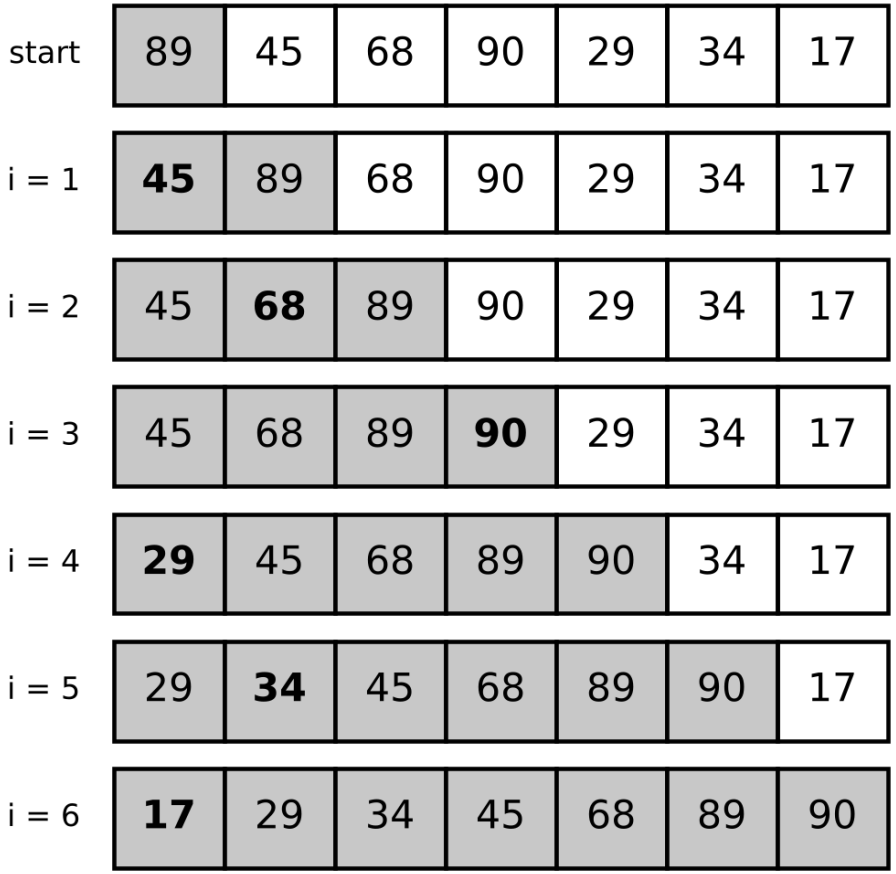
- Grey part at the end: sorted
- Pass 1: 90 bubbles to the end
- Comparisons: $n(n - 1)/2$
- Swaps: up to $n(n - 1)/2$
- Stop early if a pass makes no swaps

Insertion sort

Like sorting cards in your hand: take the next card and insert it into its place among the sorted ones.

```
ALGORITHM InsertionSort(A[0..n − 1])
    for i ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            A[j + 1] ← A[j]      //shift right
            j ← j − 1
        A[j + 1] ← v
```

Insertion sort, step by step



- Grey part: sorted (not yet final)
- Bold: the element just inserted
- i = 4: 29 moves to the front
- i = 3: 90 stays, one comparison

Insertion sort: it depends on the input

Case	Input	Comparisons
Worst	sorted in reverse	$n(n - 1)/2$: $O(n^2)$
Best	already sorted	$n - 1$: $O(n)$
Average	random	about $n^2/4$

On almost-sorted data insertion sort is almost linear. Real libraries use it for small or nearly sorted pieces.

Two important properties

In place

needs only O(1) extra memory: all three algorithms

Stable

equal elements keep their order: insertion and bubble sort, but not selection sort

Selection sort on 2a, 2b, 1: the first swap gives 1, 2b, 2a, so the two 2’s changed places.

Comparison

	Selection	Bubble	Insertion
Comparisons, worst	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
Comparisons, best	$n(n-1)/2$	$n-1$ (with stop)	$n-1$
Swaps / moves, worst	$n-1$	$n(n-1)/2$	$n(n-1)/2$
Stable	no	yes	yes
Best use	expensive moves	teaching	small, almost sorted

What $O(n^2)$ means in practice

8 min

to sort 10^6 numbers with $n^2/2$ comparisons at 10^9 per second

< 1 s

with merge sort next week: about $2 \cdot 10^7$ comparisons

Summary

- Selection sort selects the minimum; bubble sort swaps neighbors; insertion sort inserts into the sorted part.
- All three are in place and $O(n^2)$ in the worst case.
- Insertion sort is $O(n)$ on sorted input and the best simple sort in practice.
- Insertion and bubble sort are stable; selection sort is not.

Review questions

1. Explain in your own words how selection sort and insertion sort work. What is the main difference?
2. Sort 5, 2, 4, 1, 3 with insertion sort. Write the array after each step.
3. Which of the three algorithms is the fastest on an array that is already sorted? Why?

Reading: Levitin, Sections 3.1, 4.1; Bhargava, Chapter 2.