Sorting Algorithms

Table of contents

Selection sort 1
Insertion sort 3
Bubble sort 3
Merge sort 3
Quick sort 3
Which is best? 3

Five ways to solve one problem — sorting a list. Each is derived from a different restriction: *suppose the only thing you could do was X, could you still sort?* Taken together they are the course's worked example for comparing algorithms, which is what algorithmic-complexity[1] formalises.

Sorting is also the course's very first example of an algorithm at all (2025-07-29-a-brief-history-of-computing[2]). The selection-sort trace below needs no code beside it: *what* is being expressed matters more than the language it is expressed in.

Selection sort

> Suppose we could only *select* (find the index of) the largest element from a list.

Find the largest member of `xs` and make it the last element. Then find the next largest and make it the second last, and so on. After k passes the last k elements are in their final positions.

Two things the trace shows that the one-line description does not. The list of nine takes only eight passes — once the last eight are placed the first element has nowhere else to go, which is the `17 ok` row. And the *work* is invisible in the picture: each row costs a full scan of the unsorted prefix to find its grey cell, and it is those scans, not the eight moves, that make selection sort quadratic.

[1] courses/csse1001/algorithmic-complexity.pdf
[2] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf

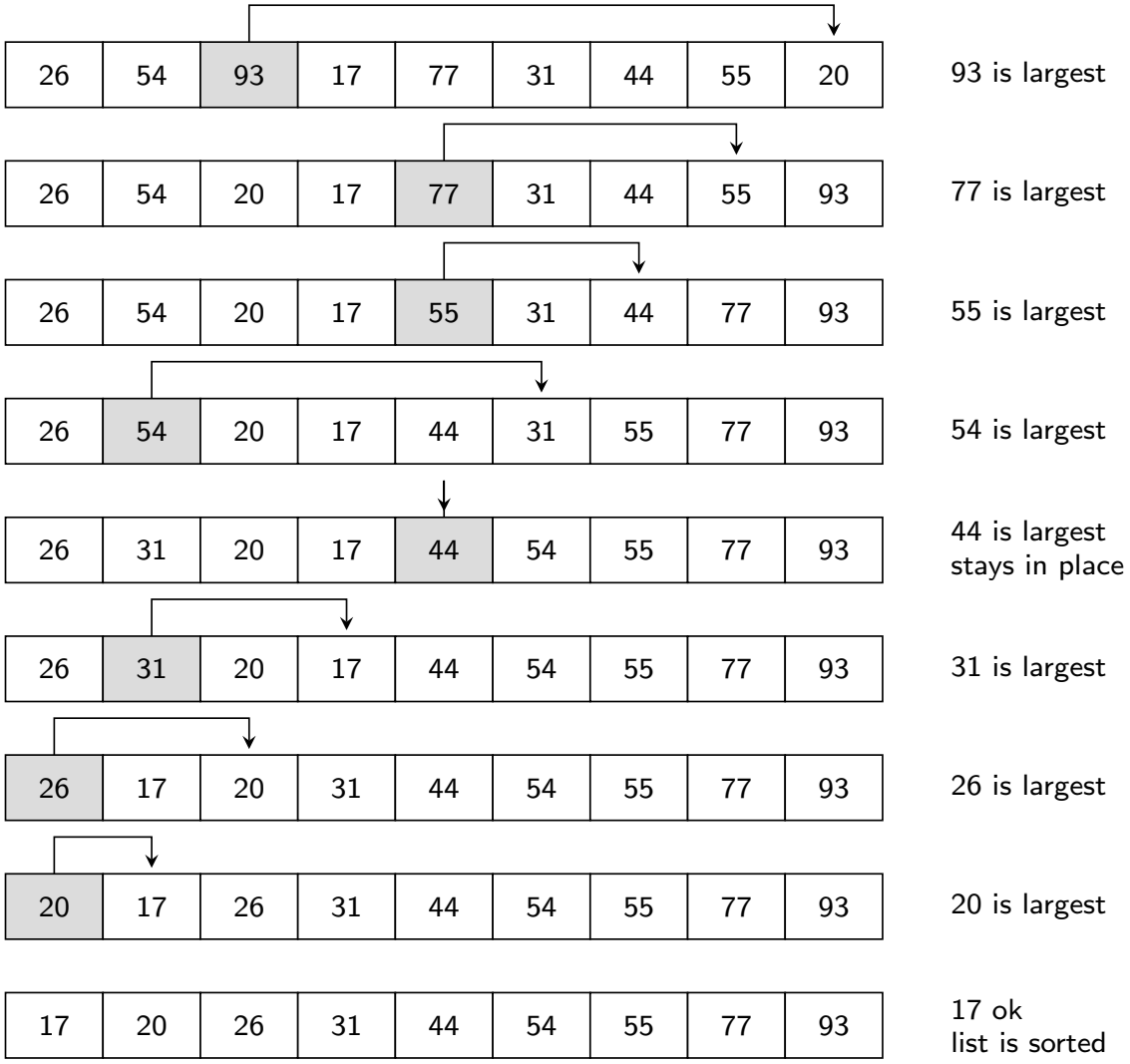


Figure 1: Selection sort on a list of nine. Each row is one pass; time runs downward. The grey cell is the largest element of the still-unsorted prefix and the arrow above the row takes it to the position it will keep, so after k rows the last k cells are final. In row five 44 is already where it belongs, so the arrow points straight down and nothing moves.

Insertion sort

> Suppose we could only *place an element correctly into an already-sorted list.*

Sort the list by inserting elements repeatedly into the front portion of the list. After k steps the first k elements of the list are sorted (though not necessarily in their final positions relative to the rest).

Bubble sort

> Suppose the only operation we could perform was *swapping adjacent cells.*

Pass through the list swapping neighbouring elements so that larger elements move forward — larger elements "bubble" to the top. Make repeated passes until the list is sorted.

Merge sort

> Suppose the only operation we could perform was *merging two sorted lists into one sorted list.*

Break the list in half, sort each half (using merge sort — this is recursion[3]), then merge the two sorted halves.

The evaluation has the same two phases as any recursion: a **winding** phase where the list is repeatedly halved down to trivial pieces, and an **unwinding** phase where the sorted pieces are merged back up.

Quick sort

> Could we sort a list if all we could do was *split* a list into two pieces based on some predicate?

Divide the list into two lists — those greater than or equal to the head, and those less than or equal to the head. Sort those lists (recursively) and then just concatenate them.

Which is best?

Rather than timing them, count how many times the characteristic operations are performed — for sorting, that means **comparisons** and **swaps**.

[3] courses/csse1001/python-recursion.pdf

Algorithm	Complexity
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Bubble sort	$O(n^2)$
Merge sort	$O(n \ln n)$

Merge sort's divide-and-conquer structure is what buys it the better class: the halving contributes a $\log n$ factor and each level of the recursion does $O(n)$ merging work.

Exercise: implement selection, insertion, bubble and merge sort. The `hanoi` function in 2025-10-20-recursion[4] is the closest worked example of the same divide-and-conquer shape that merge sort and quick sort use.

[4] courses/csse1001/2025-10-20-recursion.pdf