LECTURE 30

Sorting: Elementary Algorithms

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

WHAT THIS LECTURE COVERS

Agenda

1. The sorting problem, and what characterizes a sorting algorithm
2. Bubble sort, selection sort, and insertion sort
3. Tracing each algorithm pass by pass
4. In-place sorting, and proving bubble sort’s O(n) best case
5. A head-to-head comparison/swap count across all three
6. A direct comparison of all three algorithms

FOUNDATIONS

The Sorting Problem

- Sorting rearranges a collection into a defined order, using two core operations: comparing and swapping (or shifting).

1

Time complexity

How comparisons/swaps grow with n.

2

Space complexity

Extra memory beyond the input array.

3

Stability

Do equal elements keep their relative order?

4

In-place

Sorts within the original array, or needs a copy?

ALGORITHM 1

Bubble Sort

- Repeatedly steps through the array, swapping adjacent elements that are in the wrong order — each pass "bubbles" the largest remaining value to the end.

Original

64	25	12	22	11	90	5

Compare 64,25 → swap

25	**64**	12	22	11	90	5

Adjacent out-of-order pair gets swapped.

End of pass 1: 90 bubbled to the end

25	12	22	11	64	5	**90**

Largest remaining value reaches its final position.

A "swapped" flag lets the algorithm stop the instant a full pass makes zero swaps — giving bubble sort a best case of O(n).

IMPLEMENTATION

Bubble Sort — Code

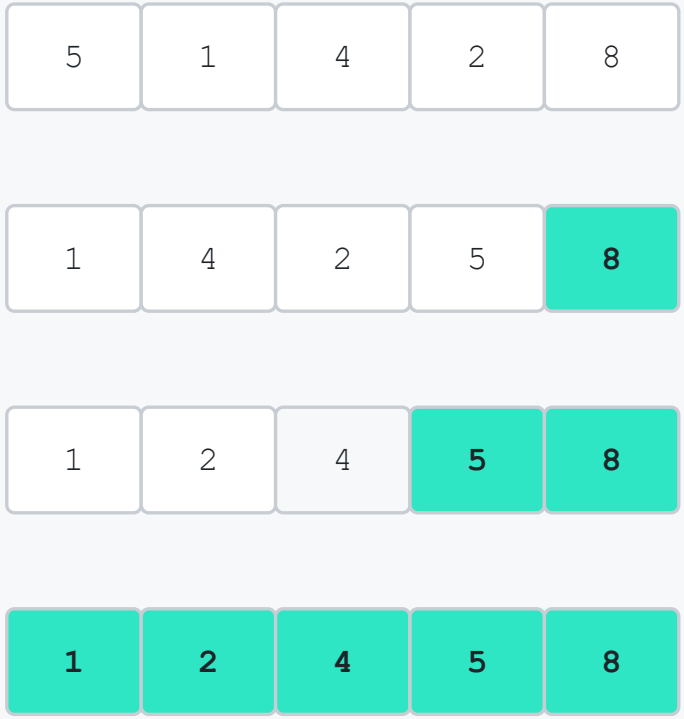
elementary_sorts.cpp

```
for (int pass = 0; pass < n - 1; pass++) {
    bool swapped = false;
    for (int i = 0; i < n - 1 - pass; i++) {
        if (arr[i] > arr[i + 1]) {
            swap(arr[i], arr[i + 1]);
            swapped = true;
        }
    }
    if (!swapped) break;  // already sorted
}
```

- Inner loop shrinks by one each pass (n-1-pass) — the sorted tail never needs re-checking.
- The swapped flag triggers early exit on nearly-sorted input.
- All swaps are adjacent — the algorithm never looks more than one position ahead.

5-ELEMENT ARRAY: 5 1 4 2 8

Tracing Bubble Sort Pass by Pass



Only 3 passes ran on a 5-element array, not the full n-1=4 the outer loop allows — confirming the early-exit optimization triggers in practice.

1,000-ELEMENT INSTRUMENTED RUN

Proving the Best Case with Real Counts

999

comparisons, 0 swaps — ALREADY-SORTED input (n=1000)

499,500

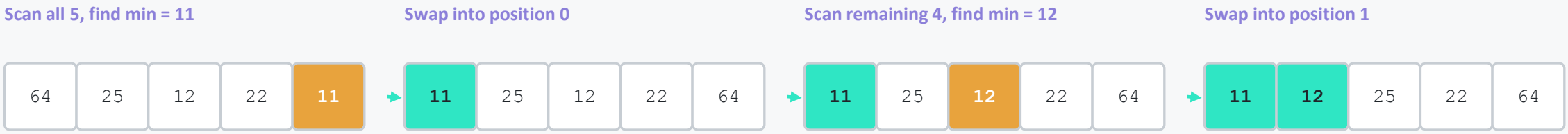
comparisons, 499,500 swaps — REVERSE-SORTED input

- 999 comparisons is exactly n-1 — one full pass, confirming the best case directly, not just in theory.
- 499,500 = 999 × 1000 / 2, exactly the n(n-1)/2 the $O(n^2)$ bound predicts for a pass that never exits early.
- Same size input (n=1000), differing only in starting order — almost a 500x gap in comparisons.

ALGORITHM 2

Selection Sort

- Finds the MINIMUM of the remaining unsorted portion and swaps it directly into place — exactly one swap per outer-loop pass.



Every round does the SAME amount of scanning regardless of how the data started — selection sort has no best-case advantage: best, average, and worst case are all $O(n^2)$.

IMPLEMENTATION

Selection Sort — Code

elementary_sorts.cpp

```
for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[minIndex])
            minIndex = j;
    }
    swap(arr[i], arr[minIndex]);
}
```

- Inner loop always scans the entire remaining unsorted region — no early exit possible.
- Exactly one swap() call per outer-loop iteration: n-1 swaps total, always.
- Swap count is far lower than bubble sort’s, at the cost of always doing the full scan.

ALGORITHM 3

Insertion Sort

- Builds the sorted portion one element at a time: each new element shifts backward through the sorted portion until it reaches its spot.

Each already-sorted element only shifts as far as it needs to — like bubble sort, best case O(n) on already-sorted input (the inner while loop never runs).

IMPLEMENTATION

Insertion Sort — Code

elementary_sorts.cpp

```
for (int i = 1; i < n; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];  // shift right
        j--;
    }
    arr[j + 1] = key;  // insert key
}
```

- The while loop shifts, not swaps — one comparison-driven pass moves the key directly to its spot.
- On already-sorted input the while condition is false immediately: O(n) best case.
- Real hybrid library sorts (Lecture 31) fall back to insertion sort for small sub-arrays.

A SHARED ADVANTAGE

In-Place Sorting

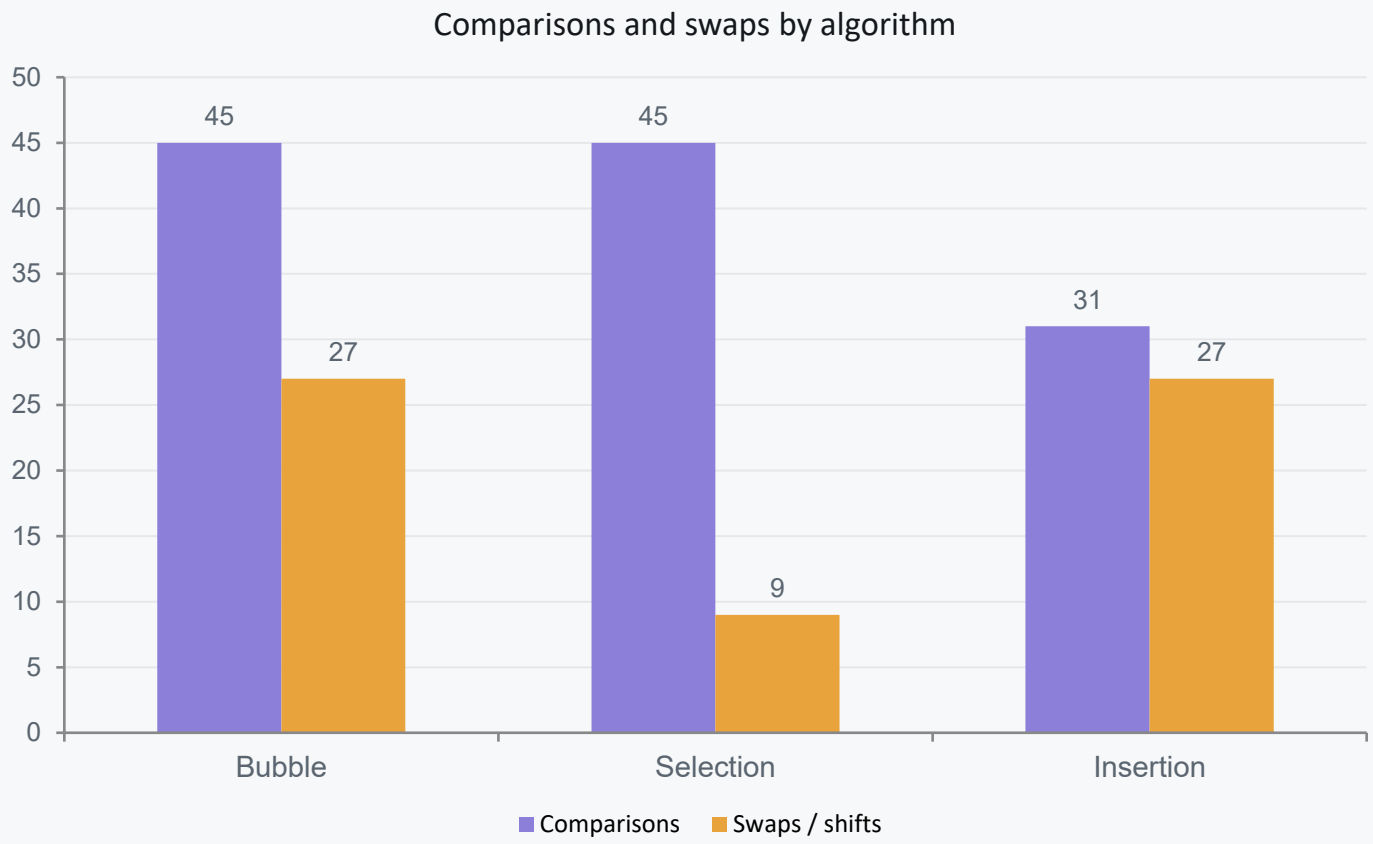
- All three algorithms rearrange the array's own elements using only a small, constant amount of extra memory (a few loop variables) — never a second array the size of the input.

Bubble Sort	Selection Sort	Insertion Sort
O(1)	O(1)	O(1)
extra space	extra space	extra space

A real advantage they share, despite their $O(n^2)$ time complexity.

SAME 10-ELEMENT INPUT: 64 25 12 22 11 90 5 34 77 3

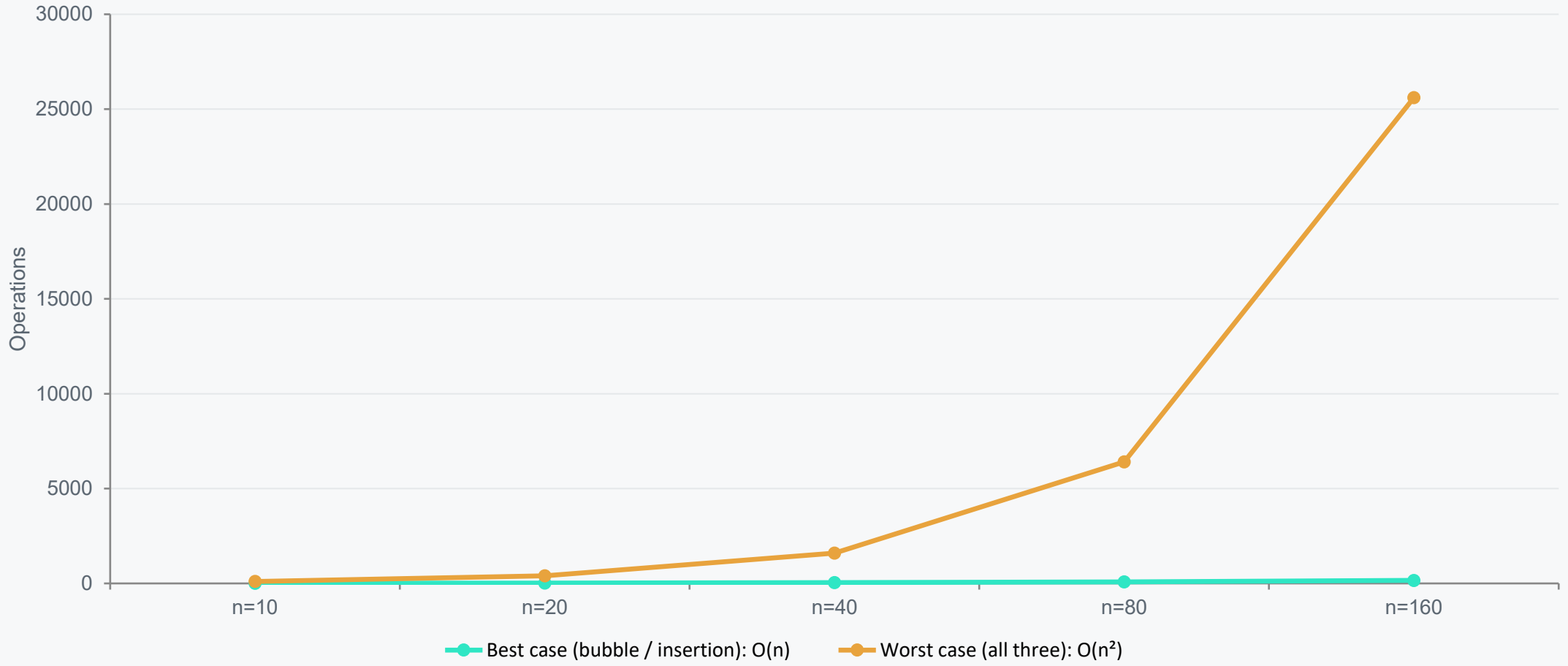
A Head-to-Head Count



- Bubble and selection both make 45 comparisons here — n(n-1)/2 for n=10.
- Selection's swaps (9 = n-1) are far lower than bubble's (27) — it always does minimal swaps.
- Insertion needed the fewest comparisons (31) — evidence it's generally preferred in practice.

BEST CASE VS. WORST CASE, ALL THREE ALGORITHMS

Big-O Growth: O(n) vs. O(n²)



BEST / AVERAGE / WORST / SPACE / STABILITY

Comparison of Elementary Sorts

	Best	Average	Worst	Space	Stable?
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes

Despite sharing the same worst-case complexity, insertion sort is generally preferred in practice — same best-case advantage as bubble sort, but does noticeably less work on average.

CLOSING

Key Takeaways

- Bubble sort’s early-exit "swapped" flag gives it O(n) best case — confirmed with 999 comparisons, 0 swaps on 1,000 sorted elements.
- Selection sort always scans the full remaining region — $O(n^2)$ in every case, with no benefit from partially-sorted input.
- Insertion sort builds a sorted portion incrementally — O(n) best case and generally the most efficient of the three in practice.
- Tracing an algorithm pass by pass turns an abstract "O(n) best case" claim into something you can actually see happen.
- All three are O(1) extra space (in-place), but all three are $O(n^2)$ worst case — Lecture 31 covers algorithms that do better.