LECTURE 31

Sorting: Efficient Algorithms

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

WHAT THIS LECTURE COVERS

Agenda

1. The divide-and-conquer strategy
2. Merge sort: split, sort each half, merge
3. Tracing merge sort’s divide and merge phases as a recursion tree
4. Quick sort: partition around a pivot, then recurse
5. Tracing quick sort’s partition step swap by swap
6. A real, counted comparison, and when each is the better choice

THE CORE IDEA BEHIND BOTH ALGORITHMS

Divide-and-Conquer Strategy

- Elementary sorts (Lecture 30) top out at $O(n^2)$. Merge sort and quick sort both break through to $O(n \log n)$ with the same three-step strategy.

1

1. Divide

Split the problem into smaller subproblems of the same kind.

2

2. Conquer

Solve each subproblem recursively.

3

3. Combine

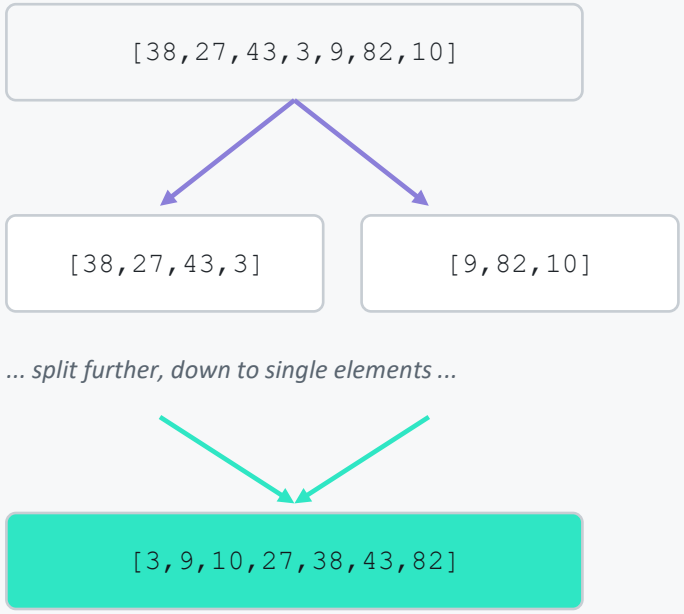
Merge the subproblems' solutions into the final answer.

Merge sort's hard work happens in COMBINE. Quick sort's hard work happens in DIVIDE.

SPLIT — SORT EACH HALF — MERGE

Merge Sort

- Splits the array in half, recursively sorts each half, then merges the two already-sorted halves back together in linear time.



- The merge step is where the real work happens: given two sorted halves, build the fully sorted result in one linear pass, always comparing just the current front of each half.
- This is the same idea as merging two sorted linked lists.
- It guarantees O(n log n) regardless of the input's initial order — unlike quick sort.

IMPLEMENTATION

Merge Sort — Code

merge_sort.cpp

```
void mergeSort(vector<int>& arr, int left, int right) {
    if (left >= right) return;   // base case

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);         // conquer left
    mergeSort(arr, mid + 1, right);    // conquer right
    merge(arr, left, mid, right);       // combine
}
```

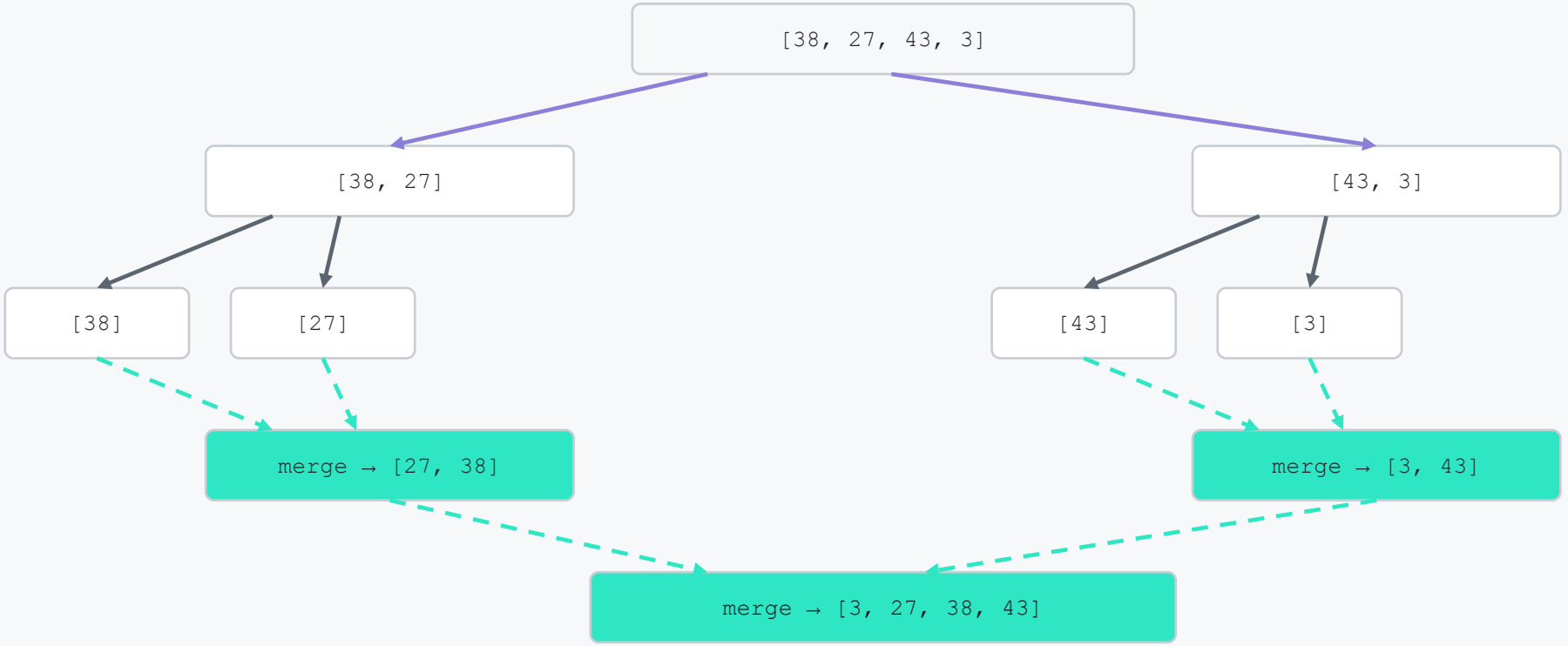
- Base case: 0 or 1 elements are already "sorted."
- Two recursive calls sort each half independently.
- merge() does the combine step: linear-time merge of two sorted halves.

O(n log n)

log n levels of recursion × O(n) work merged per level = O(n log n) total

4-ELEMENT ARRAY: [38, 27, 43, 3]

Tracing the Divide and Merge Phases



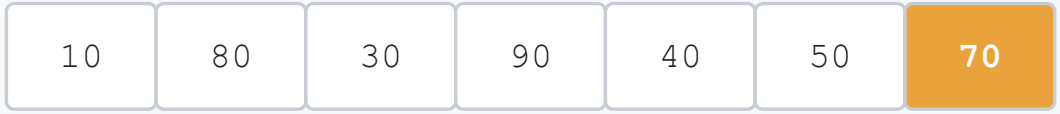
The recursion splits all the way to single-element base cases BEFORE any merging starts, then merges work back UP the tree, level by level.

PARTITION AROUND A PIVOT, THEN RECURSE

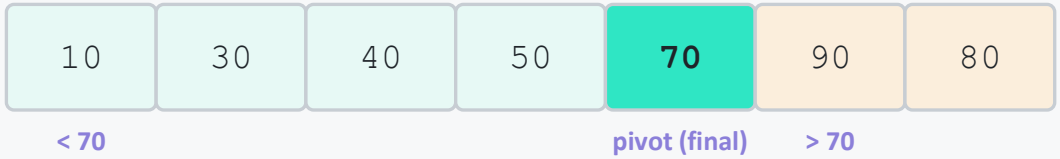
Quick Sort

- Picks a pivot, partitions the array so everything smaller ends up to its left and everything larger to its right, then recurses on each side.

Before partitioning (pivot = last element, 70):



After partitioning: smaller-than-pivot on the left, larger on the right, pivot in its FINAL position:



Unlike merge sort, quick sort does its hard work BEFORE recursing (partitioning), and sorts entirely in-place — no second array needed.

IMPLEMENTATION

Quick Sort — Code

quick_sort.cpp

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);  // place pivot
    return i + 1;
}
```

- i tracks the boundary of "elements smaller than pivot so far."
- Each smaller element found gets swapped just past that boundary.
- The pivot is swapped into its final position last — already sorted, no further work needed.

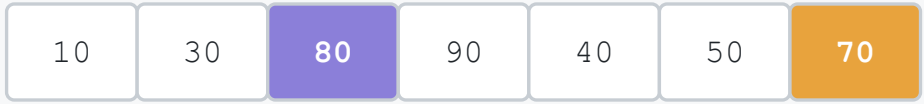
O(log n)

extra space — recursion stack only, sorts in-place

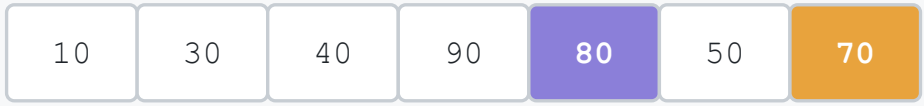
SWAP BY SWAP, PIVOT = 70

Tracing the Partition Step

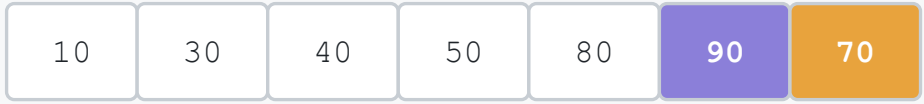
j=2: 30<70, swap(1,2)



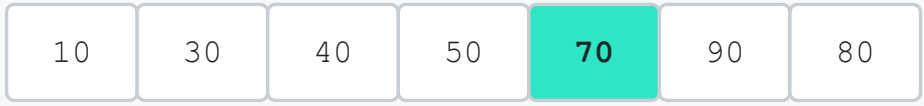
j=4: 40<70, swap(2,4)



j=5: 50<70, swap(3,5)



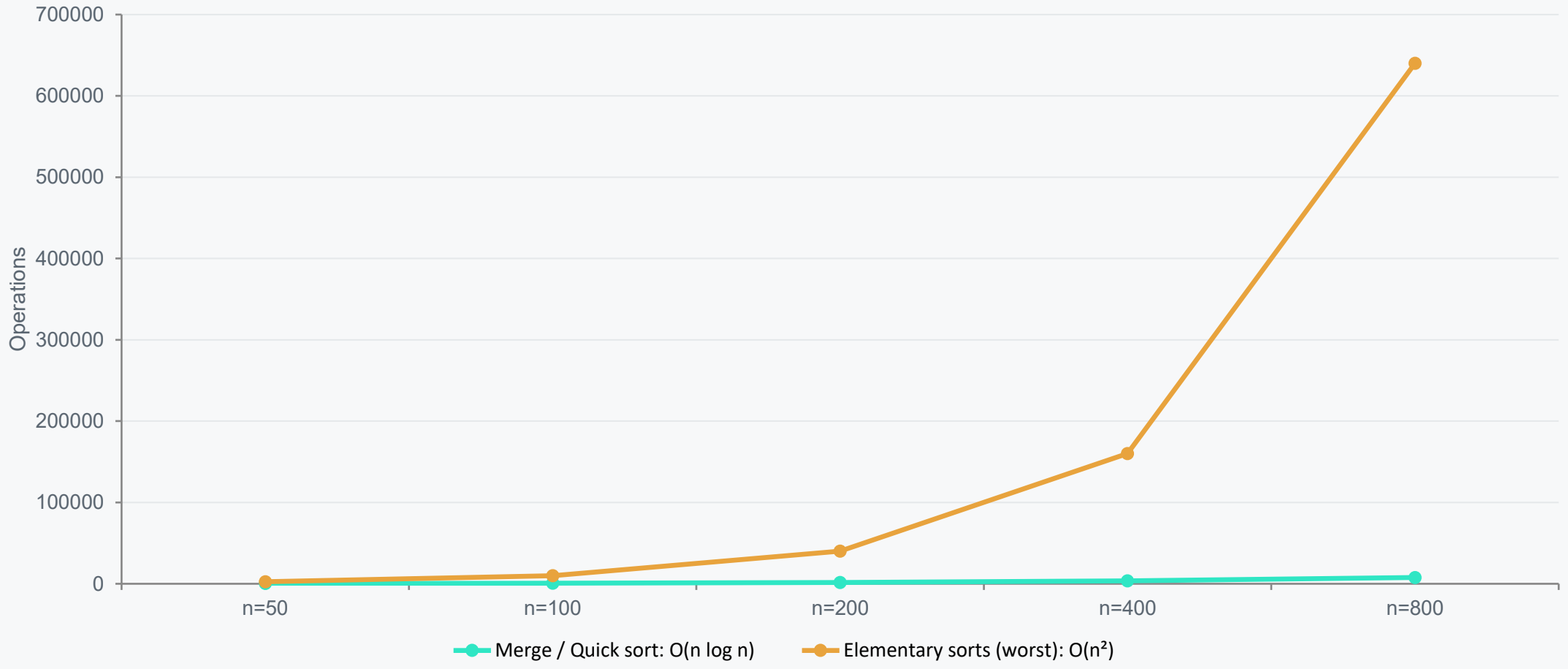
place pivot: swap(4,6)



Every element < 70 ends up left of index 4; every element > 70 ends up right — the pivot never needs to be touched again.

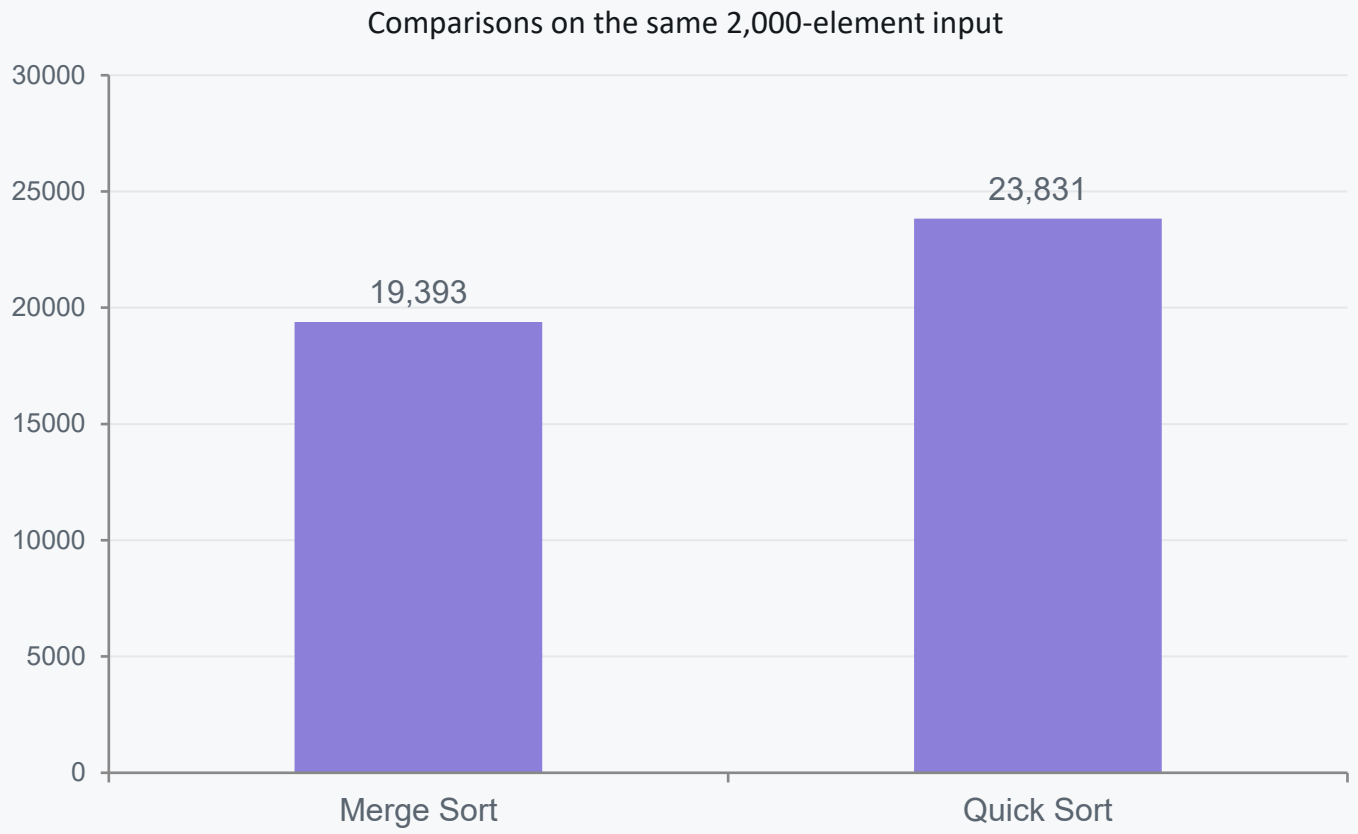
EFFICIENT SORTS VS. ELEMENTARY SORTS

Big-O Growth: O(n log n) vs. O(n²)



2,000 RANDOM INTEGERS, FIXED SEED

A Real, Counted Comparison



- Both land close to the n $\log_2$ n ≈ 22,000 estimate — neither is secretly doing quadratic work.
- Merge sort’s guaranteed, always-balanced split gave it slightly fewer comparisons here.
- Comparison counts (not wall-clock time) are reproducible across machines and runs.

GUARANTEED VS. USUALLY-FASTER

Merge Sort vs. Quick Sort

	Merge Sort	Quick Sort
Best / average case	O(n log n)	O(n log n)
Worst case	O(n log n) — always	$O(n^2)$ — poor pivot choice
Space	O(n) — second array	O(log n) — in-place
Stable?	Yes	No
Typical real-world speed	Consistently good	Usually faster in practice

THEORY VS. REAL-WORLD LIBRARIES

Avoiding Quick Sort’s Worst Case

! If the input is already sorted (or reverse-sorted) and the pivot is always the last element, every partition splits the array as unevenly as possible — degrading to $O(n^2)$.

✓ Real-world implementations avoid this by picking the pivot RANDOMLY, or as the median of a few sampled elements — making the worst case astronomically unlikely rather than eliminating it in theory.

Despite quick sort’s theoretical worst case, it is what most real standard libraries use by default (often hybridized with insertion sort for small sub-arrays) — its practical, average-case speed usually beats merge sort’s guaranteed-but-more-overhead performance.

CLOSING

Key Takeaways

- Divide and conquer — split, solve recursively, combine — achieves O(n log n) where elementary sorts only manage $O(n^2)$.
- Merge sort’s work is in the combine step: always O(n log n), stable, but needs O(n) extra space.
- Quick sort’s work is in the divide step: in-place with O(log n) space, but can degrade to $O(n^2)$ with a poor pivot.
- A real, counted comparison on 2,000 elements showed both algorithms landing close to the $n \log_2 n$ estimate.
- In practice, quick sort (randomized pivot) usually beats merge sort, which is why most standard libraries default to it.