🔄 Sorting

insertion sort over idea ,reason it works , implentation and why does htis effect time complecity best case

merge sort over idea ,implentation and why does htis effect time complecity best case =worstcase

heap sort over idea ,implentation and why does htis effect time complecity best case = worst case

why ios nlogn the most optimal ? proof

📘 Sorting Algorithms — Open Book Notes

Insertion Sort

Idea

- Build a **sorted prefix** from left to right.
- Insert each new element into its correct position in the prefix (by shifting larger elements right).

Why it works

- Any list of <2 elements is sorted.
- If a prefix is sorted, we can insert one new element and keep it sorted.
- By induction, this gives a fully sorted listlecture02_sorting (1).

Implementation sketch

- For each index `i = 1..n-1`:
    - Compare `xs[i]` backwards with elements in the sorted prefix.

- Swap/shift until it's in place.

Complexity reasoning

- Inserting into a prefix of size `k` can require `O(k)` shifts.
- Total = 1+2+...+(n−1)=O(n2)
- **Best case**: already sorted → each insertion = O(1) → total O(n).
- **Average/worst case**: $\Theta(n^2)$, since each element may be compared many times.

Merge Sort

Idea

- Divide list into halves, recursively sort each, then merge.
- Relies on merging two **already sorted halves** efficientlylecture02_sorting (1).

Why it works

- Merging two sorted lists takes linear time.
- Recursively sorting halves guarantees they're sorted before merging.
- Induction ensures correctness for full list.

Implementation sketch

- Base case: list length ≤ 1 is sorted.
- Recursive step: split in half, sort both halves, then merge.

Complexity reasoning

- Each merge step is O(n) for n elements.
- Recursion depth = log n.
- Total = O(n log n).
- **Best = worst = average = O(n log n)** (balanced recursion tree).

Heap Sort

Idea

- Use a **heap (priority queue)**:
  - Build a max-heap from input (largest at root).
  - Repeatedly extract root, restore heap property, and append to sorted listlecture02_sorting (1).

Why it works

- A heap guarantees fast access to the largest element.
- Heapify maintains order efficiently.
- By extracting repeatedly, we eventually output sorted order.

Implementation sketch

- Build heap in O(n).
- For i = n..1:
  - Swap root with last element.
  - Reduce heap size, heapify root.

Complexity reasoning

- Build heap: O(n).
- Each extract+heapify: O(log n).
- n extracts total → O(n log n).
- **Best = worst = average = O(n log n)** (heap always rebalanced).

Why nlognn is Optimal (Decision Tree Argument)

Setup

- A comparison sort determines order using yes/no comparisons ("is a[i] ≤ a[j]?").
- Model as a binary **decision tree**:
  - Internal node = comparison.

- Leaf = one possible sorted permutation.

Key facts

lecture02_sorting (1)

- There are n! possible input permutations.
- So decision tree must have ≥ n! leaves.
- A binary tree of height h has ≤ 2^h leaves.
- Thus:2h≥n!⇒h≥log2(n!)

Approximation (Stirling's formula)

log2(n!)≈nlog2n−nlog2e\

So height = Ω(n log n).

Conclusion

- Any comparison-based sorting needs at least Ω(n log n) comparisons.
- Merge sort and heapsort achieve O(n log n) → **optimal**.
- No comparison sort can be asymptotically faster.

✅ Summary Table

Algorithm	Best Case	Average Case	Worst Case	Reason
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Fixes inversions one by one
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Divide-and-conquer, bulk resolves inversions
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Heap property ensures log n per extract
Any Comparison Sort	≥ n log n	≥ n log n	≥ n log n	Decision tree lower bound

---

Would you like me to format these notes into a **1-page condensed cheat sheet (PDF)** so it’s easier to glance