88. Merge Sorted Array

Explain why merging from the back avoids extra buffers.

The problem statement requires an in-place solution, meaning we must sort the data using only the existing allocated memory (nums1) and a minimal amount of extra memory (like the few variables for the pointers).

By starting the merge at the end, we place the merged elements into the unused, empty space of nums1.

Crucially, the elements we are moving (nums1[p1]) are only moved once and elements that haven't been processed yet are not shifted.

This strategy completely eliminates the need for a temporary array (an extra buffer) or time-consuming shifting operations, making it highly efficient.

148. Sort List

Argue O(n log n) time and O(1) aux for bottom-up

Time Complexity: $O(n \log n)$

1. **Phases:** We start with sorted sub-lists of size 1 and double the size until it reaches n. The number of times we can double a value from 1 up to n is log_2 n.
    - **Total Phases = O(\log n)**
2. **Work per Phase:** In each phase (e.g., merging all lists of size 4 into size 8), we traverse the **entire list** of n nodes exactly once to perform all the necessary merges. The merging process involves a constant number of comparisons and pointer updates per node.
    - **Total Work per Phase = O(n)**
3. **Total Time:** Total Time = (Number of Phases) x (Work per Phase)
    - **Total Time = O(\log n) \times O(n) = O(n \log n)**

Auxiliary Space Complexity: O(1)

1. **Iterative Approach:** The **Bottom-Up** Merge Sort is **iterative** (using loops), not recursive. This is key. The **Top-Down** recursive approach would require $O(\log n)$ space for the function call stack.

2. **Pointer Manipulation:** The code only uses a **constant** number of extra variables to store pointers (dummy_head, tail, curr, list1, list2, etc.).
3. **No Extra Data Structures:** We never create new nodes or use extra structures (like an array or vector) whose size depends on the input size n. We sort the list entirely **in-place** by simply changing the next pointers of the existing nodes.

Therefore, the auxiliary space used is constant, which is **O(1)**

Q1. 414. Third Maximum Number

Solve via quicksort (or partial selection) and discuss tradeoffs vs. one-pass selection

Feature	**Quickselect (Partial Selection)**	**One-Pass Selection (Top K Tracker)**
Concept	Uses Quicksort's partitioning to iteratively narrow the search space until the k-th element's position is found.	Uses a fixed number of variables (three for k=3) to track the top k distinct elements while traversing the array once.
Time Complexity	**O(N) (Average Case)**. Highly efficient on average, as it avoids fully sorting the array.	**O(N) (Guaranteed)**. Always completes in a single pass, regardless of input order.
Worst-Case Complexity	**O(N^2)**- Occurs with consistently poor pivot choices (e.g., always picking the smallest/largest element).	**O(N)-** The worst case is the same as the average case.

Space Complexity	O(log N) (for the recursive call stack) or O(1) (if implemented iteratively).	O(1) Uses only a constant number of variables.
Suitability	Best for finding the k-th element when k is large or variable (e.g., finding the median, k=N/2).	Superior for finding a small, fixed k (like k=3), due to its simplicity and guaranteed performance.
Implementation	More complex, requiring separate logic for partitioning and recursive/iterative selection.	Simple, single loop logic with straightforward conditional updates.

.

Q2: 628. Maximum Product of Three Numbers

Use quicksort to get extremes, then compute candidates. Explain when sorting is simpler than linear scans

Method	Complexity	Primary Use Case	When is it Simpler?
Linear Scan	O(N)	Finding a small fixed number of extremes (e.g., top 3, bottom 2) in	**Runtime Simplicity:** It's faster. If this is the *only* thing you ever need from the array, O(N) is better than O(N \log N).

		a single pass.	
Sorting	O(N log N)	Ordering the entire array.	**Conceptual Simplicity & Reusability:** Once the array is sorted (either by QuickSort or `std::sort`), the five numbers you need are instantly available at the two ends (`nums[0]`, `nums[1]`, `nums[n-1]`, etc.). If you have a sequence of problems (find median, find top 10 sum, find max product of 3), sorting once solves them all. When using a robust built-in function like C++'s `std::sort`, the code is also simpler and less prone to edge-case errors than manually tracking five variables in a single loop.

Q3. 215. Kth Largest Element in an Array

Report average O(n), worst O(n2).

Scenario	Time Complexity	Rationale
Average Case	**O(n)**	On average, the randomized pivot selection ensures the array is partitioned into two subproblems where the smaller one is a constant fraction of the original size. This results in the recurrence relation T(n) = T(\alpha n) + O(n), where \alpha < 1, which simplifies to **linear time**.
Worst Case	**O(n^2)**	This occurs if the partitioning routine consistently selects the smallest or largest element as the pivot. In this scenario, the algorithm only reduces the problem size by one element in each step, leading to the recurrence relation T(n) = T(n-1) + O(n), which results in **quadratic time**.

Q5. 973. K Closest Points to Origin

Compare to heap (O(n log k))

Feature	Quickselect (Partition-based)	Max-Heap (Priority Queue)
Data Structure	None (In-place array manipulation)	Max-Heap of size K
Approach	Finds the distance of the K-th point, partitioning the array until this boundary is reached.	Maintains the K *smallest* distances seen so far.
Worst-Case Time	O(N^2) (If partitions are always highly unbalanced, but rare with randomization)	O(N \log K) (Predictable and guaranteed)
Average Time	**O(N)** (Much faster for large N and small K)	O(N \log K)
Space Complexity	O(1) (In-place sorting)	O(K) (To store the K points in the heap)

Use Case Strength	Best for **selection problems** (finding the k-th smallest/largest element) where *average* speed is critical.	Best for **streaming or online data** where you need the top K at all times, or when predictability is key.
Code Complexity	More complex to implement correctly (requires recursion and careful handling of the partition index).	Easier to implement using standard library priority queues (`std::priority_queue` in C++).