Name : Najim Khan
2401420003
B tech CSE (ds)

Quick Sort Trace: Array
[34,7,23,32,5,62]

Step 1: Choose 34 as pivot (first element is a common strategy).

Partition: (less than 34), 34, (greater than 34).

Step 2: Recursively Quick Sort. Pivot: 7.

(less), 7, (greater).
Step 3: , Pivot: 23.
[] (less), 23, (greater).
Combine in order:, 34, 62.
Pivots chosen in each step: 34, 7, and 23.
Bottom-Up Heap Construction: Array
[20,15,30,5,10]
Heap is built level by level.

Representation as a binary tree:
text
20
/ \
15 30
/ \
5 10
Start from last parent (15 at index 1):
No need to swap-children 5, 10 < 15.

Move to root (20 at index 0): Children 15, 30. Largest is 30, so swap 20 and 30.

Intermediate:
text
30
/ \
15 20
/ \
5 10
15's children unchanged (5, 10).
Final heap:.
Merge Sort vs Quick Sort: Complexity

Algorithm	Time Complexity	Space Complexity	Notes
Merge Sort	$O(n \log n)$ all cases	$O(n)$	Requires auxiliary array
Quick Sort	$O(n \log n)$ avg, $O(n^2)$ worst	$O(\log n)$	In-place, but bad pivot causes worst

Quick sort is faster in practice but can degrade if pivots are poorly chosen.

Merge sort has predictable performance but higher space usage.

Hybrid Merge + Insertion Sort (for small subarrays)
Approach: During Merge Sort, use Insertion Sort on subarrays smaller than a threshold (e.g., 10 elements) to reduce recursive calls and exploit Insertion Sort's efficiency on small, partially sorted datasets.

Process:

During merge sort split, before recursively sorting a subarray, check its size.

If size ≤ threshold, use insertion sort directly.

Merge Sort for Linked List: Steps and Analysis

Divide: Use slow and fast pointers to find the midpoint and split the list.

Conquer: Recursively sort each half.)
due to halving and merging at each recursive level.

Auxiliary Space:
O(logn)
O(logn) stack for recursion,
O(1)
O(1) extra.

Insertion Sort Effectiveness: Nearly

Sorted Dataset

Best-case Time Complexity:
O(n)
O(n) when nearly sorted, each insert is minimal movement.

Justification: Fewer shifts are required.

General: Insertion sort is highly efficient for small/n nearly sorted data.

Sorting with Many Duplicates: Algorithm

Approach: Use 3-way partitioning Quick Sort.

Why: It groups elements == pivot together, reducing unnecessary comparisons.

Time Complexity:

O(nlogn) average,
O(n 2) worst, but typically faster for duplicates.

MergeSort Modification for Large Datasets

Modification: Use iterative/bottom-up Merge Sort (avoiding recursion) or parallel merge, and consider external sort for huge data.

Impact: Reduces stack space requirements, may improve cache efficiency or support out-of-core sorting.

Space Complexity: Still O(n) for merging; time remains O(n log n).

Hashing Definition

Hashing refers to computing a hash

code or key index from data, enabling fast insertion, deletion, and lookup in data structures like hash tables.

Hash Table Index Calculation

For table size 10 and

h(k)=kmod10:

53mod10=3
21mod10=1
87mod10=7
44mod10=4

Binary Search for 13: Array
[2,5,8,13,17,19]

Step 1: Start: left=0, right=5. mid=2 (
8
8).

Step 2: 13 > 8, search right half.
left=3, right=5. mid=4 (

17
17).

Step 3: 13 < 17, search left half. left=3, right=3, mid=3 (
13
13).

Found at index 3.

3rd Smallest by Quick Sort: Array

[11,5,8,9,3,10]

Pivot selection and partition finds kth element. After partitioning, 8 will be the third smallest:

Partition: less than 11.

Next, partition with 5 as pivot.

Remaining list after sorting: 3, 5, 8, 9, 10, 11. 3rd smallest is 8.

Straight Sequential Search for 18:
Array
[10,15,18,25,30]

Comparisons: 10 (no), 15 (no), 18 (yes, found at third element).

Total comparisons: 3

Selection Sort: Array
[14,7,9,3,12]; Steps and Swaps

Iter 1: min is 3. Swap 14 and 3: (1 swap)
Iter 2: min is 7-already in place.
Iter 3: min is 9-already in place.
Iter 4: min is 12. Swap 14 and 12: (2nd swap)

Total swaps: 2

Binary Search: Avg and Worst-Case Analysis

Case	Comparisons	Occurrence
Average	O(log n)	Most searches; divides search intervals.
Worst	O(log n)	When the element is not present or last

Both cases are O(log n); practical implications are negligible, but performance is degraded if used on unsorted or non-random-access data.

Optimized Sequential Search: Search 7 in [4,2,9,5,7,1]

Compare each element till 7:
comparisons = 5.

Time Complexity:
O (n)