1. Trees

Binary Trees

- **Definition**: A tree where each node has at most 2 children (left and right).
- **Properties**:
  - **Height**: Longest path from root to leaf.
  - **Full Binary Tree**: Every node has 0 or 2 children.
  - **Complete Binary Tree**: All levels are fully filled except possibly the last, filled left to right.
- **Traversals**:
  - **In-order (LNR)**: Left → Node → Right (sorted order in BST).
  - **Pre-order (NLR)**: Node → Left → Right (used for copying trees).
  - **Post-order (LRN)**: Left → Right → Node (used for deletions).
  - **Level-order (BFS)**: Visit nodes level by level (uses a queue).

Binary Search Trees (BSTs)

- **Definition**: A binary tree where for each node:
  - Left subtree contains values ≤ node.
  - Right subtree contains values > node.
- **Operations**:
  - **Search**: O(h) (h = height, O(log n) if balanced, O(n) worst case).
  - **Insert/Delete**: Must maintain BST property.
- **Balanced BSTs (AVL, Red-Black)**: Ensure O(log n) operations via rotations.

Heaps

- **Definition**: A complete binary tree where:
  - **Min-Heap**: Parent ≤ children.
  - **Max-Heap**: Parent ≥ children.
- **Operations**:
  - **Insert**: O(log n) – percolate up.
  - **Extract Min/Max**: O(log n) – replace root with last element, percolate down.
- **Heapify**: O(n) time to build a heap from an array.
- **Applications**: Priority queues, HeapSort.

2. Maps, Dictionaries, & Hashing

Maps/Dictionaries

- **Key-Value pairs**: Unordered (hash maps) or ordered (TreeMap).
- **Operations**:
  - `get(key)`, `put(key, val)`, `remove(key)` – ideally O(1) avg. (hash map), O(log n) (TreeMap).

Hashing

- **Hash Function**: Maps keys to array indices (should be uniform).
- **Collision Resolution**:
  - **Chaining**: Linked lists at each bucket.
  - **Open Addressing**: Linear/Quadratic probing, double hashing.
- **Load Factor (λ)**: Ratio of entries to buckets; resizing (rehashing) when λ > threshold.

3. Recursion

- **Base Case**: Stopping condition.
- **Recursive Case**: Calls itself with smaller inputs.
- **Examples**: Factorial, Fibonacci, tree traversals, backtracking.
- **Tail Recursion**: Optimized by compiler (recursive call is last operation).
- **Memoization**: Cache results to avoid redundant calls (e.g., Fibonacci).

4. Sorting & Selection

Comparison Sorts

Algorithm	Best	Avg	Worst	Space
Bubble	O(n)	O(n^2)	O(n^2)	O(1)
Selection	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion	O(n)	O(n^2)	O(n^2)	O(1)
Merge	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick	O(n log n)	O(n log n)	O(n^2)	O(log n)
Heap	O(n log n)	O(n log n)	O(n log n)	O(1)

Non-Comparison Sorts

- **Counting Sort**: O(n + k) (k = range of values).
- **Radix Sort**: O(d(n + k)) (d = digits).

Selection Algorithms

- **Quickselect**: Avg O(n), Worst O(n^2) – finds k-th smallest element.
- **Median of Medians**: Worst-case O(n).

5. Graph Search Algorithms

Breadth-First Search (BFS)

- **Uses a queue**.
- **Finds shortest path (unweighted graphs)**.
- **Time**: O(V + E), Space: O(V).

Depth-First Search (DFS)

- **Uses a stack (recursion or explicit stack)**.
- **Applications**: Topological sort, cycle detection, maze solving.
- **Time**: O(V + E), Space: O(V).

Dijkstra's Algorithm

- **Finds shortest path (weighted graphs, non-negative edges)**.
- **Uses a priority queue (min-heap)**.
- **Time**: O((V + E) log V).

*A Algorithm (for Maze Searching)**

- **Best-first search with heuristic (h(n))** to estimate cost to goal.
- **f(n) = g(n) + h(n)** (g = cost from start, h = admissible heuristic).
- **Optimal if h(n) is admissible (never overestimates)**.
- **More efficient than BFS/Dijkstra with a good heuristic**.