1. Merge-Sort

Explanation: Merge-sort is a divide-and-conquer algorithm that splits an array into halves, sorts each half, and then merges them together.

Key Term:

- **Divide-and-Conquer:** A strategy to solve a problem by dividing it into smaller parts, solving each part independently, and then combining them.

Time Complexity:

- O(n log n) for all cases.

Cookbook Method:

1. Split the array into two halves.
2. Recursively sort each half.
3. Merge the sorted halves back together.

2. Heap-Sort

Explanation: Heap-sort involves converting the input array into a heap structure, repeatedly removing the largest element from the heap, and rebuilding the heap until it's empty.

Key Term:

- **Heap:** A tree-based data structure where each parent node is either greater than or equal to (max heap) or less than or equal to (min heap) its children.

Time Complexity:

- O(n log n) for all cases.

Cookbook Method:

1. Build a max heap from the array.
2. Swap the first (largest) element with the last.
3. Reduce the heap size and heapify the root element.
4. Repeat step 2 and 3 until the heap is empty.

3. Quick-Sort

Explanation: Quick-sort selects a pivot element and partitions the array around this pivot, placing smaller elements to the left and larger to the right, then recursively sorts the partitions.

Key Term:

- **Pivot:** A selected element to compare others against during partitioning.

Time Complexity:

- Average case: O(n log n)
- Worst case: $O(n^2)$

Cookbook Method:

1. Select a pivot (often the last element of the array).
2. Partition the array around the pivot.
3. Recursively apply quick-sort to the left and right partitions.

4. Counting-Sort

Explanation: Counting-sort calculates the frequency of each value, then uses this count to place each element in its correct position in the output array.

Key Term:

- **Stable Sorting:** Maintaining the relative order of equal elements in the sorted output.

Time Complexity:

- O(n+k), where kk is the range of the input values.

Cookbook Method:

1. Count occurrences of each value.
2. Calculate starting index for each value.
3. Place each element at its correct position in the output array.

5. Bucket-Sort

Explanation: Bucket-sort distributes elements into several 'buckets', sorts each bucket individually, and then concatenates all buckets into the final sorted array.

Key Term:

- **Bucket:** A container used to collect elements that fall into the same range.

Time Complexity:

- Average case: O(n+k)
- Worst case: $O(n^2)$

Cookbook Method:

1. Divide elements into buckets based on a range.
2. Sort each bucket individually.

3. Concatenate all buckets into the final array.

6. Kruskal's Algorithm

Explanation: Kruskal's algorithm finds the minimum spanning tree in a graph by selecting the shortest edges while avoiding cycles.

Key Term:

- **Minimum Spanning Tree:** A tree that connects all vertices in a graph with the minimum total edge weight.

Time Complexity:

- O(E log E), where E is the number of edges.

Cookbook Method:

1. Sort all edges by weight.
2. Initialize a forest (a set of trees), where each vertex in the graph is a separate tree.
3. For each edge, if the two ends are in different trees, add it to the forest, merging the two trees.

7. Prim's Algorithm

Explanation: Prim's algorithm constructs a minimum spanning tree by expanding the tree one edge at a time, choosing the shortest edge connecting the tree to another vertex.

Key Term:

- **Greedy Algorithm:** An algorithm that makes the locally optimal choice at each step.

Time Complexity:

- O(V2) or O(E+log V) with a priority queue.

Cookbook Method:

1. Start with any vertex as the current tree.
2. Find the shortest edge connecting the tree to a new vertex and add it.
3. Repeat step 2 until all vertices are included.

8. Bellman-Ford Algorithm

Explanation: The Bellman-Ford algorithm calculates shortest paths from a single source vertex to all other vertices in a weighted graph, capable of handling negative weights.

Key Term:

- **Relaxation:** Updating the shortest path and predecessor of a vertex.

Time Complexity:

- O(VE), where V is the number of vertices and E is the number of edges.

Cookbook Method:

1. Initialize distances from the source to all vertices as infinite and to the source itself as 0.
2. Relax all edges V−1V−1 times.
3. Check for negative-weight cycles.

9. Dijkstra's Algorithm

Explanation: Dijkstra's algorithm finds the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights.

Key Term:

- **Priority Queue:** A queue where the element with the highest priority is served first.

Time Complexity:

- O((V+E)log V) with a priority queue.

Cookbook Method:

1. Set initial distances to infinity, except for the source vertex (set to 0).
2. Use a priority queue to find the vertex with the smallest distance.
3. Update distances of adjacent vertices.
4. Repeat steps 2 and 3 until all vertices are processed.

10. Breadth-First Search (BFS)

Explanation: BFS is a traversal method that explores the neighbour nodes at the current depth level before moving on to nodes at the next depth level.

Key Term:

- **Queue:** A data structure used to store nodes in the order they should be processed.

Time Complexity:

- O(V+E).

Cookbook Method:

1. Start from a selected node and visit its neighbours.
2. Add each neighbour to a queue.

3. Dequeue a node and visit its neighbours if they haven't been visited yet.
4. Repeat step 3 until the queue is empty.

11. Depth-First Search (DFS)

Explanation: DFS is a traversal method that explores as far as possible along each branch before backtracking.

Key Term:

- **Stack:** A data structure used to store nodes in their order of exploration.

Time Complexity:

- O(V+E).

Cookbook Method:

1. Start from a selected node, visit it, and mark it as visited.
2. Go to an adjacent unvisited node, visit it, and push it onto the stack.
3. Repeat step 2 until you reach a node with no unvisited neighbours.
4. Pop a node from the stack and backtrack.
5. Repeat steps 2-4 until the stack is empty.

12. Ford-Fulkerson Method

Explanation: The Ford-Fulkerson method computes the maximum flow in a flow network by finding augmenting paths and increasing the flow along these paths.

Key Term:

- **Augmenting Path:** A path from the source to the sink where the flow can still be increased.

Time Complexity:

- O(E·f), where f is the maximum flow.

Cookbook Method:

1. Start with zero flow in all edges.
2. Find an augmenting path using BFS or DFS.
3. Increase the flow along the path by the minimum residual capacity of the edges.
4. Repeat steps 2 and 3 until no augmenting path can be found.

13. Edmonds-Karp Method

Explanation: The Edmonds-Karp algorithm is a specific implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths, providing better performance in some cases.

Time Complexity:

- $O(VE^2)$.

Cookbook Method:

1. Initialize the flow in all edges to 0.
2. Use BFS to find the shortest augmenting path.
3. Increase flow along the path by the minimum residual capacity of the edges.
4. Repeat steps 2 and 3 until no augmenting path can be found.

14. Red-Black Trees

Explanation: Red-Black Trees are a self-balancing binary search tree where each node has an additional color attribute, used to ensure the tree remains approximately balanced.

Key Term:

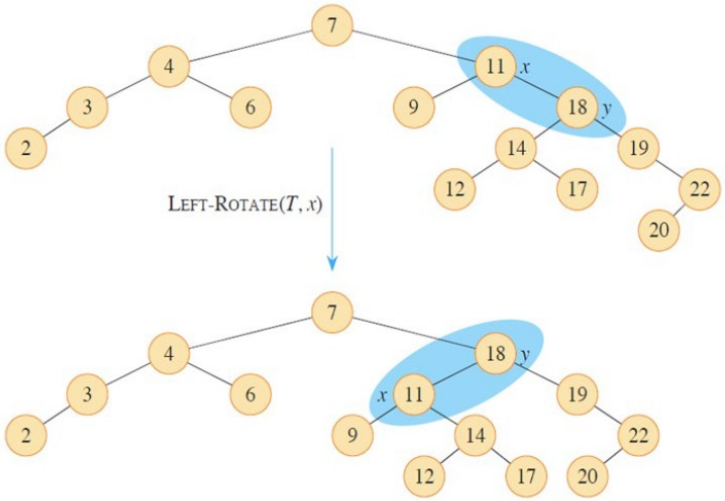
- **Color Property:** Ensures no two consecutive red nodes and that every path from a node to its descendant leaves has the same number of black nodes.

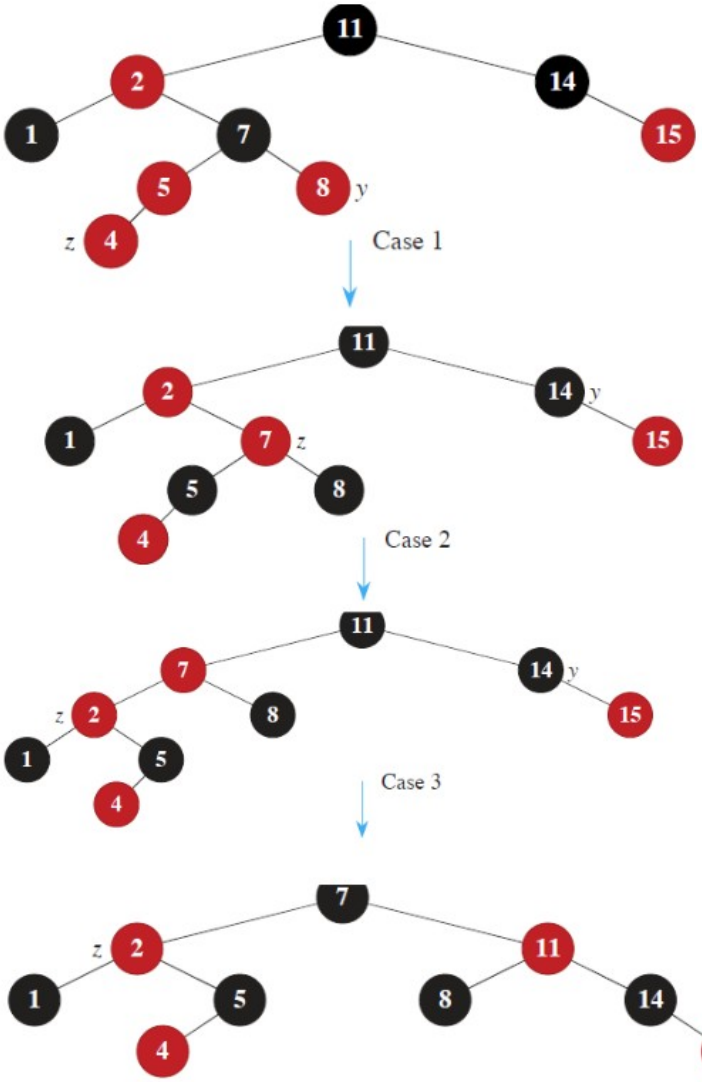
Time Complexity:

- O(log n) for insertion, deletion, and searching.

Cookbook Method:

1. Insert a node as in a standard binary search tree.
2. Color the new node red.
3. Fix the red-black properties if they are violated.
4. For deletion, find the node to delete, remove it, and fix any violations.


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20


11
2
14
1
7
15
5
8
y
z
4
Case 1
11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4
Case 3
7
z
2
11
1
5
8
14
4
15

Stable

A sorting algorithm is considered stable if it maintains the relative order of records with equal keys (or values). In other words, if two items are equal according to the sorting key, their order will be the same in the sorted output as it was in the input.

Example: Consider sorting the following list of pairs `(value, index)` by their first element: `[(3, 1), (2, 2), (3, 3), (1, 4)]`. A stable sort would produce `[(1, 4), (2, 2), (3, 1), (3, 3)]`. Notice how the pairs `(3, 1)` and `(3, 3)` maintain their original relative order.

In-Place

An algorithm is in-place if it requires a constant amount (O(1)) of extra space for its operation, besides the input data. In-place algorithms may modify the input data but do not use extra space proportional to the size of the input.

Example: An in-place sorting algorithm might swap elements within the array being sorted but does not require additional arrays or lists to hold elements. Quick sort is a common example of an in-place sorting algorithm, while merge sort is not in-place since it requires additional space for merging.

These properties are important in evaluating the suitability of algorithms in different contexts, like when memory usage is a concern (in-place) or when preserving the original order of equal elements is important (stable).