2. QUICK SORT

Introduction :

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the array around the pivot such that elements smaller than pivot are on the left and greater elements are on the right. Then recursively sorts the left and right subarrays.

Algorithm :

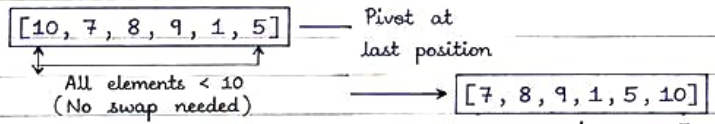
1. Choose a pivot element.
2. Partition the array such that all elements smaller than pivot are on the left and all greater elements on the right.
3. Recursively apply Quick Sort on the left subarray.
4. Recursively apply Quick Sort on the right subarray.

Partition Procedure :

1. Take first element as pivot.
2. Use two indices i (start) and j (end).
3. Move i forward till an element greater than pivot is found.
4. Move j backward till an element smaller than pivot is found.
5. Swap A[i] and A[j] if i < j.
6. Repeat steps 3 to 5 until i >= j.
7. Place pivot at index j (or i).
8. Return j as partition index.

Example : Sort the array : [10, 7, 8, 9, 1, 5]

Step 1 : Pivot = 10 (first element)



Step 2 : Left part → [7, 8, 9, 1, 5] Pivot = 7

Step 3 : Left → [1, 5] Pivot = 1 Right → [9, 8] Pivot = 9

Sorted array : [1, 5, 7, 8, 9, 10]

Properties :

→ In-place sorting algorithm.
→ Not stable (relative order may change).
→ Performs well on average.
→ Worst case occurs when array is already sorted or reverse sorted.

Applications :

→ Used in general purpose sorting.
→ Used in programming libraries (e.g., std::sort in C++).

Advantages :

→ Average time complexity is O(n log n).
→ In-place, requires less extra memory.
→ Faster in practice for smaller data sets.

Disadvantages :

→ Worst case time complexity is $O(n^2)$.
→ Not stable.
→ Requires good choice of pivot for better performance.

Time Complexity :

Best Case : O(n log n)
Worst Case : $O(n^2)$
Average Case : O(n log n)

Space Complexity :

O(log n) (Recursive stack)

Pseudocode :

```
QuickSort(A, low, high)
  If low < high
     pi = Partition(A, low, high)
     QuickSort(A, low, pi-1)
     QuickSort(A, pi+1, high)

Partition(A, low, high)
   pivot = A[low]
   i = low + 1, j = high
   While True
     While (i <= high and A[i] <= pivot)
        i = i + 1
     While (j > low and A[j] > pivot)
        j = j - 1
     If i < j
        Swap A[i] and A[j]
     Else
        Swap A[low] and A[j]
     Return j
```

4. AVL TREE ROTATIONS (LL, RR, LR, RL)

Introduction :

When AVL Tree becomes unbalanced after insertion or deletion, we perform rotations to restore balance. There are four types of rotations depending on the position of the heavy node.

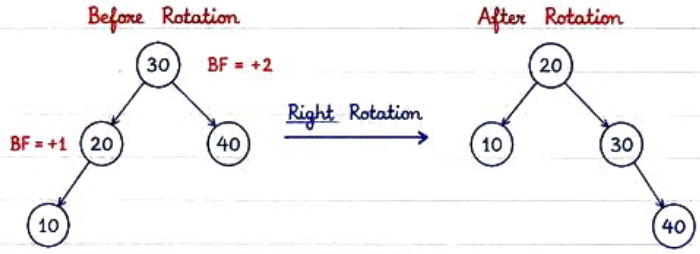
Unbalanced Condition :
If for any node N,
$|BF(N)| > 1$
then the tree is unbalanced and rotation is required.

Types of Rotations :

1. LL Rotation (Left Left Case)
2. RR Rotation (Right Right Case)
3. LR Rotation (Left Right Case)
4. RL Rotation (Right Left Case)

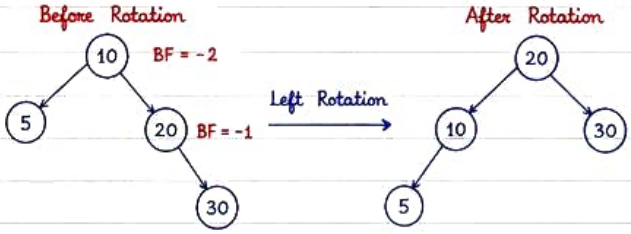
1. LL ROTATION (Left Left Case)

This case occurs when a node is inserted in the left subtree of the left child.
We perform single right rotation.



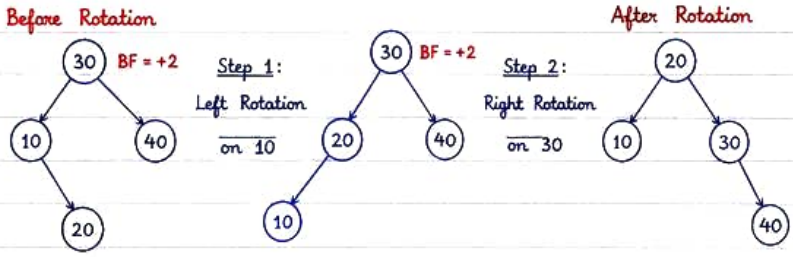
2. RR ROTATION (Right Right Case)

This case occurs when a node is inserted in the right subtree of the right child.
We perform single left rotation.



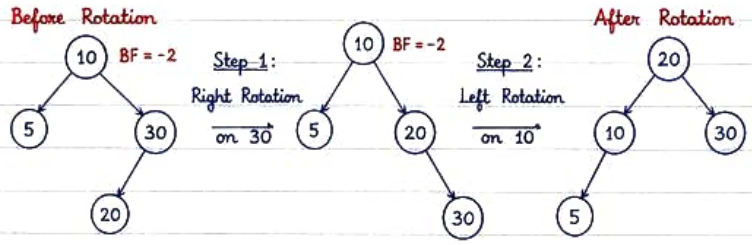
3. LR ROTATION (Left Right Case)

This case occurs when a node is inserted in the right subtree of the left child.
We perform two rotations :
(i) Left rotation on left child
(ii) Right rotation on root node



4. RL ROTATION (Right Left Case)

This case occurs when a node is inserted in the left subtree of the right child.
We perform two rotations :
(i) Right rotation on right child
(ii) Left rotation on root node



Summary Table :

Case	Condition	Type of Rotation	Rotation Performed
LL	BF > 1 at node and insertion in left-left	Single Right Rotation	Right Rotation on node
RR	BF < -1 at node and insertion in right-right	Single Left Rotation	Left Rotation on node
LR	BF > 1 at node and insertion in left-right	Double Rotation	Left Rotation on left child Right Rotation on node
RL	BF < -1 at node and insertion in right-left	Double Rotation	Right Rotation on right child Left Rotation on node

Important Points :
- → Rotations are used to maintain balance in AVL tree.
- → After rotation, height of tree remains balanced and BST property is also preserved.
- → Each rotation takes $O(1)$ time.
- → Insertion and Deletion in AVL Tree takes $O(\log n)$ time.

Note : Rotations do not change the in-order sequence of tree. So, BST property remains same.

3. Solve 0/1 Knapsack Problem using Dynamic Programming.

1. Problem Statement :

Given n items, each with weight w[i] and value v[i] and a knapsack of capacity W, the 0/1 Knapsack problem is to select a subset of items such that total weight is at most W and total value is maximum.

Note : Each item can be taken either once (1) or not taken (0).

Example :

n = 4, W = 7

Item (i)	1	2	3	4
Weight w[i]	1	3	4	5
Value v[i]	1	4	5	7

Find maximum value that can be obtained using capacity W = 7.

2. DP State and Recurrence Relation :

DP State :

Let K[i][w] = Maximum value that can be obtained using first i items and knapsack capacity w.

Recurrence Relation :

$$K[i][w] = \begin{cases} K[i-1][w], & \text{if } w[i-1] > w \\ \max(K[i-1][w],\ v[i-1] + K[i-1][w-w[i-1]]), & \text{if } w[i-1] \le w \end{cases}$$

where w[i-1] = weight of i-th item, v[i-1] = value of i-th item

Explanation :

If weight of current item is more than remaining capacity (w), we cannot include it.
Otherwise, we have two choices :
(1) Exclude the item.
(2) Include the item and add its value to remaining capacity (w - w[i-1]).

3. Base Case :

- If no items are available (i = 0), then value = 0
- If capacity is 0 (w = 0), then value = 0

So, K[0][w] = 0 for all w (0 ≤ w ≤ W)
and K[i][0] = 0 for all i (0 ≤ i ≤ n)

4. DP Table Construction :

Rows → Items (0 to n)
Columns → Capacity (0 to W)

Capacity (w)

Items (i)	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	1	4	4	5	5	5
3	0	1	1	4	5	5	6	6
4	0	1	1	4	5	7	8	9

Note : Row 0 and Column 0 are filled with 0 as per base case.

Result :
Maximum value = K[4][7] = 9
Items taken : {2, 4}
(i.e., item 2 and item 4)
Total weight = 3 + 5 = 8 ≤ 7 ? ✗
Wait! Check correctly.
From table, weight = 3 + 4 = 7
(Items 2 and 3)

5. Backtracking to find selected items :

Start from K[n][W] = K[4][7] = 9
K[4][7] ≠ K[3][7] ⇒ Item 4 is included.
W = 7 - w[4] = 7 - 5 = 2
K[3][2] ≠ K[2][2] ⇒ Item 2 is included.
W = 2 - w[2] = 2 - 3 < 0 (stop)
So items included are {2, 4}.

Backtracking Path

K[4][7] = 9
↓ (Include item 4)
K[3][2] = 4
↓ (Include item 2)
K[1][-1] Stop

Selected items : {2, 4}

Time & Space Complexity :

Time Complexity : O(n × W)
(n = number of items, W = capacity)

Space Complexity : O(n × W)

6. Application :

Resource allocation, Budget planning, Memory allocation, Project selection, etc.

Exam Tips : Always write recurrence relation, base case, DP table and backtracking steps for full marks. Neatly show table construction and final selected items. ★

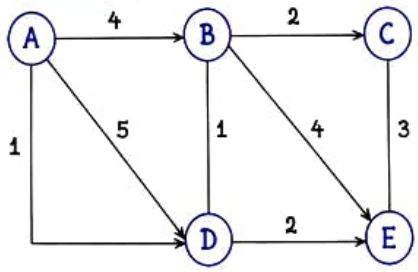
10. Explain Dijkstra's Single Source Shortest Path Algorithm.

1. Introduction :

Dijkstra's algorithm is used to find shortest distances from a single source vertex to all other vertices in a weighted graph with non-negative edge weights.

It is a greedy algorithm that always picks the vertex with the minimum tentative distance and updates the distances of its adjacent vertices.

Example Graph :



Source vertex = A

V = {A, B, C, D, E}

All weights are non-negative.

2. Problem Statement :

Given a weighted graph G = (V, E) and a source vertex $s \in V$, find shortest distance from s to every other vertex $v \in V$. Edge weights are non-negative.

Notation :

n	→	Number of vertices
dist[v]	→	Current shortest known distance from source to v
visited[v]	→	True if vertex v is included in shortest path tree
PQ	→	Priority Queue (Min-Heap) to select minimum dist[v]
parent[v]	→	Predecessor of v in shortest path tree

3. Idea :

- Initialize dist[s] = 0 and dist[others] = ∞.
- Insert all vertices into a min-priority queue.
- Repeatedly pick the vertex u with minimum dist[u].
- Mark u as visited and update distances of its adjacent vertices.
- Continue until all vertices are processed.

4. Algorithm (Dijkstra's Algorithm) :

Pseudocode :

```
Dijkstra(G, s)
 for each vertex v in V
     dist[v] = ∞, parent[v] = NIL, visited[v] = False
dist[s] = 0
Insert all vertices into Min-Priority-Queue PQ based on dist[]
while PQ is not empty
     u = Extract-Min(PQ)
     visited[u] = True
     for each neighbor v of u
         if not visited[v] and weight(u, v) + dist[u] < dist[v]
             dist[v] = weight(u, v) + dist[u]
             parent[v] = u
             Decrease-Key(PQ, v, dist[v])
```

5. Step-by-Step Execution from Source A :

Step	Vertex Picked	dist[A]	dist[B]	dist[C]	dist[D]	dist[E]	Visited Set
0	–	0	∞	∞	∞	∞	{}
1	A	0	4	∞	1	∞	{A}
	Update: B=4, D=1 (via A)						
2	D	0	2	∞	1	3	{A, D}
	Update: B=2 (via D), E=3 (via D)						
3	B	0	2	4	1	3	{A, D, B}
	Update: C=4 (via B) (E via B = 2+4=6 > 3, no change)						
4	E	0	2	4	1	3	{A, D, B, E}
	Check C via E = 3+3 = 6 > 4, no change						
5	C	0	2	4	1	3	{A, D, B, E, C}

No unvisited vertices left. Algorithm ends.

Final Shortest Distances from A :

A = 0, B = 2, C = 4, D = 1, E = 3

Shortest Path Tree (Parent Array) :

parent[A] = NIL, parent[B] = D, parent[C] = B, parent[D] = A, parent[E] = D

6. Shortest Paths from A :

A → A : Cost = 0
A → D : A → D : 1
A → B : A → D → B : 2
A → E : A → D → E : 3
A → C : A → D → B → C : 4

7. Time Complexity :

Using Min-Priority Queue (Binary Heap)

- Insert / Extract-Min / Decrease-Key → O(log V)
- We perform these operations for each edge.

Time Complexity = O((V + E) log V)

For dense graph ($E \approx V^2$) → $O(V^2 \log V)$

For sparse graph ($E \approx V$) → O(V log V)

8. Space Complexity :

- Adjacency List → O(V + E)
- dist[], parent[], visited[] → O(V)
- Priority Queue → O(V)

Total Space = O(V + E)

9. Applications :

- ★ GPS Navigation Systems
- ★ Network Routing
- ★ Robot Path Planning
- ★ Shortest Path in Maps
- ★ Traffic Management

Exam Tip : Remember : Dijkstra works only for non-negative edge weights.
Always draw the graph, table and final shortest path tree for full marks. ★

9. Write short note on Parallel Algorithms.

1. Introduction :

Parallel Algorithms are those algorithms in which many computations are performed simultaneously using multiple processors or cores.
The main goal is to reduce the execution time by dividing the problem into independent sub-tasks.

2. Need for Parallel Algorithms :

- To solve large problems faster.
- To utilize multiple processors efficiently.
- To improve throughput and performance.
- To handle real-time applications.
- With the advancement of multi-core CPUs, GPUs and distributed systems.

3. Characteristics :

* Concurrency	Many tasks are executed at the same time.
* Decomposition	Problem is divided into smaller independent sub-problems.
* Communication	Processors exchange data when required.
* Synchronization	Coordination is needed to maintain correctness.
* Scalability	Performance should increase with more processors.

4. Types of Parallelism :

- Data Parallelism : Same operation is performed on different pieces of data.
  Example → Matrix addition.
- Task Parallelism : Different tasks are executed concurrently.
  Example → Sorting and searching at same time.
- Pipeline Parallelism : Output of one task becomes input of next task (like assembly line).
  Example → Instruction pipeline in CPU.
- Hybrid Parallelism : Combination of two or more parallelism types.

5. Speedup and Efficiency :

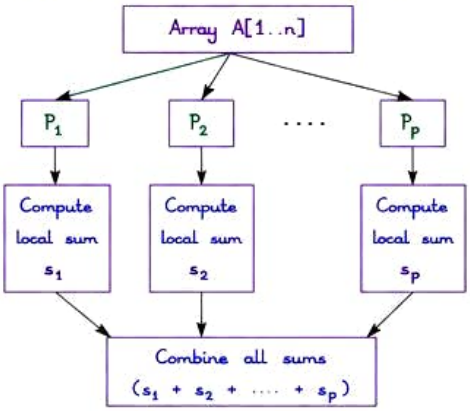
Speedup (S_p) : $S_p = \frac{T_1}{T_p}$
Where, T_1 = Time on 1 processor
T_p = Time on p processors

Efficiency (E_p) : $E_p = S_p / p$
Where, p = Number of processors

6. Example : Parallel Sum of Array

Problem : Find sum of n numbers.
Sequential Approach : Add one by one → Time = O(n)
Parallel Approach :

- Divide array into p parts.
- Each processor computes local sum.
- Finally combine all partial sums.



If each processor takes T_p time, then overall time reduces significantly.

7. Models of Parallel Computation :

1. PRAM (Parallel Random Access Machine)
   All processors share a common memory. Communication through read/write on shared memory.
2. BSP (Bulk Synchronous Parallel Model)
   Computation is divided into supersteps. Each superstep has computation + communication + synchronization.
3. Distributed Memory Model
   Each processor has its own memory. Communication occurs by sending messages.
4. GPU Model
   Thousands of lightweight threads execute same instructions on different data.

8. Advantages :

- Reduces execution time.
- Improves resource utilization.
- Solves bigger problems.
- Suitable for scientific, engineering and real-time systems.

9. Challenges :

- Need for synchronization.
- Communication overhead.
- Load balancing.
- Complexity in design and debugging.
- Not all problems can be parallelized.

10. Applications of Parallel Algorithms :

Area	Applications
Scientific Computing	Weather forecasting, simulation, nuclear research.
Multimedia	Image processing, video encoding, 3D animation.
Data Mining	Large dataset analysis, machine learning.
Artificial Intelligence	Neural networks training, pattern recognition.
Engineering	Finite element analysis, computational fluid dynamics.
Databases	Parallel query processing, large transaction systems.

Exam Tip :
1. Always define parallel algorithm first.
2. Explain need, types, models and metrics (speedup, efficiency).
3. Draw neat diagrams wherever possible.
4. Write applications and challenges in short points.

4. Solve Fractional Knapsack Problem using Greedy Method.

1. Introduction :

Fractional Knapsack problem is a variation of 0/1 Knapsack problem. In this, we are allowed to break items into fractions. The objective is to maximize total profit without exceeding the capacity of the knapsack.

2. Problem Statement :

Given n items, each with a weight (w_i) and profit (p_i), and a knapsack with capacity W. We need to choose fractions of items ($0 \le x_i \le 1$) such that total weight $\le$ W and total profit is maximum.

Here, x_i is the fraction of i^{th} item taken.

3. Greedy Strategy :

> Always select the item with highest profit per unit weight (p_i / w_i) first.
>
> If the item cannot be taken completely, take as much as possible (fraction of it).

4. Algorithm :

Step 1 : Calculate profit per unit weight for each item.

Step 2 : Sort items in decreasing order of (profit per unit weight).

Step 3 : Initialize total_profit = 0
remaining_capacity = W

Step 4 : For each item in sorted order
- a) If $weight_i \le$ remaining_capacity
  - Take the whole item
  - total_profit += $profit_i$
  - remaining_capacity -= $weight_i$
- b) Else
  - Take fraction = remaining_capacity / $weight_i$
  - total_profit += fraction x $profit_i$
  - remaining_capacity = 0
  - Break

Step 5 : Return total_profit.

> Time Complexity : O(n log n)
> (due to sorting of items)

5. Example :

Given 4 items and knapsack capacity W = 50

Items :

Item	Weight (w_i)	Profit (p_i)	Profit / Weight (p_i/w_i)
1	10	60	6.00
2	20	100	5.00
3	30	120	4.00
4	40	140	3.50

Step 1 : Sort items by decreasing (p_i / w_i)

Order : Item 1 (6.00), Item 2 (5.00), Item 3 (4.00), Item 4 (3.50)

Step 2 : Select items one by one

Step	Item Selected	Weight Taken	Profit Taken	Total Weight	Total Profit	Remaining Capacity
1	1	10	60	10	60	40
2	2	20	100	30	160	20
3	3	$20/30$ (fraction)	$(20/30) \times 120 = 80$	50	240	0

At Step 3, we can take only 20 weight from Item 3 (fraction = $\frac{20}{30} = \frac{2}{3}$) as remaining capacity is 20.

Profit from this fraction = $\frac{2}{3} \times 120 = 80$

> Maximum Total Profit = 240

6. Observations :

- We first select items with highest profit per unit weight.
- This ensures that we get maximum profit in limited capacity.
- Allowing fractions makes the solution always optimal for this problem.

7. Applications :

- Resource allocation where items can be divided.
- Investment planning.
- Cutting stock problem.
- Data compression and bandwidth allocation.
- Continuous knapsack type real world problems.

Summary : Fractional Knapsack problem is solved using Greedy approach by selecting items in decreasing order of profit per unit weight. It gives optimal solution.