

Join Community by clicking below links 

[Website Link](#)

👉 For question papers and notes visit website

[Click here](#)

[Telegram Channel](#)

[Click here](#)

[WhatsApp Channel](#)

[Click here](#)

This document is provided for **personal use only** by students who have purchased it. **Sharing, copying, distributing, or uploading** this material to any platform or with other students is **strictly prohibited**.

By using this document, you agree that it is **for your individual study and reference only**

Q1)

- A) Consider the following instance of the knapsack problem. Find the optimal solution by using dynamic programming approach.

Item	Weight	Profit
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

Capacity of the knapsack = 5.

Answer :

Step 1: Sort Items (by Weight Ascending)

Sorting by weight gives:

Item	Weight	Profit
2	1	10
1	2	12
4	2	15
3	3	20

We will use this **sorted order** for DP.

Step 2: DP Table

$dp[i][w] = \text{max profit using first } i \text{ items with capacity } w.$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
Item 2 (1,10)	0	10	10	10	10	10
Item 1 (2,12)	0	10	12	22	22	22
Item 4 (2,15)	0	10	15	25	27	37
Item 3 (3,20)	0	10	15	25	30	37

Final Maximum Profit = 37

Step 3 : Backtracking to Find Items

Start at $dp[4][5] = 37$

1 Compare $dp[4][5]$ and $dp[3][5]$

- $37 \neq 37 \rightarrow$ profit changed \rightarrow Item 4 (weight 2, profit 15) taken
Remaining capacity = $5 - 2 = 3$

2 Compare $dp[3][3]$ and $dp[2][3]$

- $25 \neq 25 \rightarrow$ Item 1 included
Remaining capacity = $3 - 2 = 1$

3 Compare $dp[2][1]$ and $dp[1][1]$

- $dp[2][1] = 10$
- $dp[1][1] = 10 \rightarrow$ same \rightarrow skip Item 1
Actually Item 2 is selected.

Selected Items

Item	Weight	Profit
Item 2	1	10
Item 1	2	12
Item 4	2	15

Total weight = $1 + 2 + 2 = 5$

Total profit = $10 + 12 + 15 = 37$

Watch this video to understand how this problem is solved

<https://youtu.be/PfkBS9qIMRE?si=gdrwkryPHS7F0fG0>

B) What is greedy approach? Explain Job scheduling algorithm using Greedy approach for following examples. Give the sequence of job scheduling.

Input: Four jobs with following deadlines and profits

Job	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Input: Five Jobs with following deadlines and profits

Job	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Answer :

1. What is Greedy Approach?

- The Greedy approach is an algorithmic strategy that makes the **locally optimal choice** at each step.
- It selects the best possible option **without reconsidering previous decisions**.
- The goal is to reach a **globally optimal solution** using a sequence of local optimal decisions.
- This strategy works only for problems that satisfy:
 - **Greedy-choice property**
 - **Optimal substructure**
- Examples of greedy-based problems include:
 - Job Scheduling
 - Fractional Knapsack
 - Activity Selection
 - Minimum Spanning Tree (Kruskal/Prim)

2. Job Scheduling Using Greedy Approach

Problem Statement

Each job has:

- A deadline
- A profit
- Each job takes **1 unit of time**
- We must schedule jobs so that **profit is maximized.**

Greedy Strategy

1. Sort all jobs in decreasing order of profit.
2. Create a **time slot array** of size = max deadline.
3. For each job (highest profit first):
 - o Place it in the **latest free slot** before its deadline.
 - o If the slot is free, schedule the job.
 - o If not free, skip the job.
4. At the end, the filled time slots represent the final job schedule

Example 1: Four Jobs

Job	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Step 1: Sort by profit (descending)

c (40), d (30), a (20), b (10)

Step 2: Maximum deadline = 4 \Rightarrow slots = [1,2,3,4]

Step 3: Schedule

- c \rightarrow deadline 1 \rightarrow slot 1 free \rightarrow place c
- d \rightarrow deadline 1 \rightarrow slot 1 filled \rightarrow skip
- a \rightarrow deadline 4 \rightarrow slot 4 free \rightarrow place a
- b \rightarrow deadline 1 \rightarrow slot 1 filled \rightarrow skip

Final Sequence

Job Sequence: [c, a]

Total Profit = 40 + 20 = 60

Example 2: Five Jobs

Job	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Step 1: Sort by profit

a (100), c (27), d (25), b (19), e (15)

Step 2: Maximum deadline = 3 \Rightarrow slots = [1,2,3]

Step 3: Schedule

- a \rightarrow deadline 2 \rightarrow slot 2 free \rightarrow place a
- c \rightarrow deadline 2 \rightarrow slot 2 filled \rightarrow try slot 1 \rightarrow slot 1 free \rightarrow place c
- d \rightarrow deadline 1 \rightarrow slot 1 filled \rightarrow skip
- b \rightarrow deadline 1 \rightarrow slot 1 filled \rightarrow skip
- e \rightarrow deadline 3 \rightarrow slot 3 free \rightarrow place e

Final Sequence

Job Sequence: [c, a, e]

Total Profit = 27 + 100 + 15 = 142

Watch this video to understand how this problem is solved

<https://youtu.be/zPtl8q9gvX8?si=BzYYeAJyx37tvFub>

- C) Find an optimal solution for the following knapsack instance using greedy method.
 Number of objects $n = 5$. Capacity of knapsack $m = 100$.

Object	Weight	Profit
01	20	10
02	30	20
03	66	30
04	40	40
05	60	50

Answer:

Fractional Knapsack — Greedy Method

Given:

- Number of objects: $n = 5$
- Knapsack capacity: $m = 100$

Step 1: Compute Profit/Weight Ratio

Object	Weight	Profit	Profit/Weight
01	20	10	0.50
02	30	20	0.67
03	66	30	0.45
04	40	40	1.00
05	60	50	0.83

Step 2: Sort objects in descending order of ratio

Order of selection:

- Object 04 → 1.00
- Object 05 → 0.83
- Object 02 → 0.67
- Object 01 → 0.50
- Object 03 → 0.45

Step 3: Fill Knapsack

Knapsack Capacity = 100

✓ Take Object 04

Weight = 40 → fully taken

Remaining capacity = $100 - 40 = 60$

Profit = 40

✓ Take Object 05

Weight = 60 → fully taken
Remaining capacity = $60 - 60 = 0$
Profit = 50
Knapsack is now full.
No more objects can be taken.

Final Answer

Objects Selected

Object 04 → full

Object 05 → full

Total Weight

$$40 + 60 = 100$$

Total Profit

$$40 + 50 = 90$$

Final Optimal Profit = 90

Watch this video to understand how this problem is solved

<https://youtu.be/xZfmHVi7FMg?si=FNZpo8di--VMoQr1>

Q2)

A) Consider 4 matrices A1, A2, A3 and A4. The orders of these matrices are given below:

Matrix Order

A1 3 × 5

A2 5 × 4

A3 4 × 2

A4 2 × 4

Find the optimal sequence of matrix chain multiplication using the dynamic programming approach. Clearly give the final parenthesization and the total number of scalar multiplications required

Answer:

Given

Matrix Order

A1 3 × 5

A2 5 × 4

A3 4 × 2

A4 2 × 4

So the dimension array is $P = [3, 5, 4, 2, 4]$ (where A_i is $P[i-1] \times P[i]$).

1. DP cost ($m[i][j]$) and split ($s[i][j]$) tables

- $m[i][j] =$ minimum number of scalar multiplications needed to compute product $A_i \dots A_j$.
- $s[i][j] =$ index k at which the optimal split $A_i \dots A_k$ and $A_{k+1} \dots A_j$ occurs.

We compute increasing chain lengths:

Length = 1 ($i = j$)

$$m[1][1] = m[2][2] = m[3][3] = m[4][4] = 0$$

Length = 2

$$m[1][2] = 3 * 5 * 4 = 60$$

$$m[2][3] = 5 * 4 * 2 = 40$$

$$m[3][4] = 4 * 2 * 4 = 32$$

Length = 3

- For $m[1][3]$ ($A_1 \dots A_3$):
 - $k = 1 \rightarrow \text{cost} = m[1][1] + m[2][3] + 352 = 0 + 40 + 30 = 70$

- $k = 2 \rightarrow \text{cost} = m[1][2] + m[3][3] + 342 = 60 + 0 + 24 = 84$
 $\Rightarrow m[1][3] = 70, s[1][3] = 1$
- For $m[2][4]$ (A2..A4):
 - $k = 2 \rightarrow \text{cost} = m[2][2] + m[3][4] + 544 = 0 + 32 + 80 = 112$
 - $k = 3 \rightarrow \text{cost} = m[2][3] + m[4][4] + 524 = 40 + 0 + 40 = 80$
 $\Rightarrow m[2][4] = 80, s[2][4] = 3$

Length = 4

- For $m[1][4]$ (A1..A4):
 - $k = 1 \rightarrow \text{cost} = m[1][1] + m[2][4] + 354 = 0 + 80 + 60 = 140$
 - $k = 2 \rightarrow \text{cost} = m[1][2] + m[3][4] + 344 = 60 + 32 + 48 = 140$
 - $k = 3 \rightarrow \text{cost} = m[1][3] + m[4][4] + 324 = 70 + 0 + 24 = 94$
 $\Rightarrow m[1][4] = 94, s[1][4] = 3$
-

2. Tables

Cost table $m[i][j]$ (rows i, columns j). Diagonal = 0.

$m =$

j=1	j=2	j=3	j=4	
i=1	0	60	70	94
i=2	-	0	40	80
i=3	-	-	0	32
i=4	-	-	-	0

Split table $s[i][j]$ (where split occurs):

$s =$

j=1	j=2	j=3	j=4	
i=1	-	1	1	3
i=2	-	-	2	3
i=3	-	-	-	3
i=4	-	-	-	-

(Entries - are not applicable or diagonal.)

3. Reconstruct optimal parenthesization

- $s[1][4] = 3 \Rightarrow \text{split } (A1..A3) | (A4) \rightarrow ((A1 A2 A3) A4)$
- Inside $A1..A3$: $s[1][3] = 1 \Rightarrow \text{split } (A1) | (A2 A3) \rightarrow (A1 (A2 A3))$

So final optimal parenthesization:

$((A1 (A2 A3)) A4)$

or equivalently: $((A1 * (A2 * A3)) * A4)$.

4. Verify by showing the multiplication sequence and costs

1. Multiply $A2$ (5×4) and $A3$ (4×2)
cost = $5 * 4 * 2 = 40$, result is a 5×2 matrix.
2. Multiply $A1$ (3×5) with result (5×2)
cost = $3 * 5 * 2 = 30$, result is a 3×2 matrix.
3. Multiply that (3×2) result with $A4$ (2×4)
cost = $3 * 2 * 4 = 24$, result is 3×4 .

Total scalar multiplications = **40 + 30 + 24 = 94**.

Final answer

Optimal parenthesization: $((A1 (A2 A3)) A4)$

Minimum number of scalar multiplications: 94

Watch this video to understand how this problem is solved

<https://youtu.be/prx1psByp7U?si=APq9D7pGnxNTHH7L>

B) Write a control abstraction for dynamic programming strategy and greedy method.
Comment on the time complexity of this abstraction

Answer :

1. CONTROL ABSTRACTION FOR DYNAMIC PROGRAMMING

Dynamic_Programming_Solver(X):

 1. Define a table OPT[0...n][0...m] // n = number of subproblems

 // m = size of solution state

 2. Initialize base cases in OPT table

 3. for i = 1 to n do

 4. for j = 1 to m do

 5. OPT[i][j] = Best_Choice(OPT of smaller subproblems)

 end for

 end for

 6. Return OPT[n][m] // optimal solution to full problem

Explanation

- Dynamic programming solves a complex problem by **dividing it into overlapping subproblems, solving each subproblem once, and storing the results in a table.**
- The main idea is to find $\text{OPT}[i][j]$ using previously computed subproblems.

Time Complexity

- Two nested loops $\rightarrow \mathbf{O(n \times m)}$
- Filling each table entry takes **O(1)** time
Therefore total complexity is:

Time Complexity = $O(n \times m)$

Space complexity = $O(n \times m)$

2. CONTROL ABSTRACTION FOR GREEDY METHOD

Greedy_Solver(X):

1. Initialize solution $S = \emptyset$
2. while X is not empty do
 3. Select the best candidate 'c' from X
according to some greedy criterion
 4. If c is feasible then
 5. Add c to solution S
 6. Remove c from X
- end while
7. Return S

Explanation

- Greedy method repeatedly picks the **locally best choice** (maximum profit, minimum weight, earliest deadline, etc.)
- Feasibility is checked before adding to solution.

⌚ Time Complexity

Greedy method usually requires:

- selecting best candidate $\rightarrow O(n)$
(or $O(\log n)$ if using priority queue)
- repeating for all items $\rightarrow n$ times

Thus:

Time Complexity = $O(n^2)$ (simple array selection)

C) With respect to dynamic programming, what is the principle of optimality? Give a mathematical representation for the same

Answer :

The **Principle of Optimality**, introduced by Richard Bellman, states:

“An optimal solution to a problem is composed of optimal solutions to its subproblems. If any subproblem solution is not optimal, then the overall solution cannot be optimal.”

This principle is **fundamental** for dynamic programming because it allows complex problems to be solved recursively by combining the solutions of smaller, overlapping subproblems.

Explanation in Detail

1. Optimal Substructure:

- A problem exhibits *optimal substructure* if the optimal solution can be constructed from optimal solutions of its subproblems.
- Without this property, dynamic programming cannot be applied.

2. Recursive Nature:

- DP problems are solved by breaking the problem into smaller subproblems.
- The principle ensures that solving each subproblem optimally will guarantee the optimality of the final solution.

3. Feasibility of Subproblems:

- Each subproblem is smaller and simpler than the original problem.
 - The solutions of subproblems are stored (memoized) and reused to avoid recomputation, exploiting overlapping subproblems.
-

Mathematical Representation

Let:

- **S** = set of states
- **f(s)** = optimal value of the problem starting from state **s**
- **A(s)** = set of feasible actions/decisions from state **s**
- **T(s, a)** = next state after taking action **a** in state **s**
- **C(s, a)** = immediate cost or profit of action **a**

Then, by the principle of optimality:

$$f(s) = \max_{a \in A(s)} \{C(s, a) + f(T(s, a))\}$$

- For **minimization problems**, replace max with min.
- This formula recursively defines the optimal solution using optimal solutions of subproblems.

Example (0/1 Knapsack Problem)

- Let $\text{OPT}[i][w]$ = maximum profit using first **i items** with capacity **w**.
- The recursive formula derived from the principle of optimality is:

$$\text{OPT}[i][w] = \max (\text{OPT}[i - 1][w], P_i + \text{OPT}[i - 1][w - W_i])$$

Explanation:

- Two choices for each item:
 1. **Exclude** item $i \rightarrow \text{profit} = \text{OPT}[i-1][w]$
 2. **Include** item $i \rightarrow \text{profit} = P_i + \text{OPT}[i-1][w-W_i]$
- The **maximum of these two** gives the optimal solution for the subproblem.
- This recursively guarantees that each subproblem's solution is optimal.

Q3)

A) Explain with suitable example Backtracking: Principle, control abstraction, time analysis of control abstraction

Answer :

1. Principle of Backtracking

Backtracking is a **depth-first search technique** used to solve problems step-by-step.

If at any step the chosen option does **not lead to a valid solution**, the algorithm:

- ✓ **Backtracks** (undoes the last step)
- ✓ **Tries a new alternative option**

Key Ideas

- Construct solution *incrementally*.
- At any point, check:
 - Is the **partial solution valid?**
 - Can it **lead to a final solution?**
- If **yes** → **continue**,
- If **no** → **backtrack** (return to previous decision point).

Applications

- N-Queens problem
- Sum of Subsets
- Graph Coloring
- Hamiltonian Cycle

2. Example of Backtracking: N-Queens Problem (4-Queens)

Goal: Place 4 queens on a 4×4 chessboard such that none attack each other.

Process (Short Summary)

1. Place Q in Row 1, Col 1 (valid)
2. Row 2: Try Col 1 → invalid.
Try Col 2 → invalid.
Try Col 3 → valid.
3. Row 3: Try available columns.
If all invalid → **backtrack** to Row 2.
4. Continue until a valid configuration is found.

This shows:

- **Incremental construction**
 - **Validity checks**
 - **Backtracking when no further progress is possible**
-

3. Control Abstraction for Backtracking

This is the **general structure** used for all backtracking algorithms.

Control Abstraction (General Algorithm)

```

Backtrack(k)
{
    if (k > n)
        Output solution;
    else
    {
        for each candidate x in ChoiceSet(k)
        {
            if (isValid(x, k))
            {
                solution[k] = x;
                Backtrack(k + 1);
            }
        }
    }
}

```

Explanation of Components

- **k** → Level of the state space tree (current step)
- **ChoiceSet(k)** → All possible choices at level k
- **isValid(x, k)** → Feasibility check for the partial solution
- **Backtrack(k+1)** → Go to next level
- **If no candidate fits → backtracking happens automatically**

This abstraction works for:

- N-Queens

- Graph Coloring
 - Subset Sum
 - Knapsack (0/1) enumeration
 - Permutation generation
-

4. Time Analysis of Backtracking

Time complexity depends on:

- **Number of levels = n**
- **Choices at each level = m**
- **Validity checking time = O(1) or O(n)**

Worst-Case Time Complexity

If each level has m choices:

[
O(m^n)
]

(or exponential in general)

Why Exponential?

Because the algorithm explores a **state-space tree**.

Example

For N-Queens:

- At each row, approx n choices
- Depth = n

Complexity:

[
O($n!$)
]

Best Case

If the first few choices lead to valid results:

[
O(n)
]

(because pruning eliminates many branches)

B) Consider a graph, which is represented by the adjacency matrix given below:

	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

We wish to colour this graph using Red and Black colours using a backtracking algorithm.
Show the process of colouring it in stepwise manner using state space diagram.

Watch this video to understand how this problem is solved

<https://youtu.be/052VkJlaQ4?si=JAza6NgooaWPHkek>

C) Explain the ‘branch and bound’ approach for solving problems. Write a branch and bound algorithm for solving the 0/1 Knapsack problem. Use the same algorithm to solve the following 0/1 Knapsack problem. The capacity of the Knapsack is 15 kg.

Item	A	B	C	D
Profit (Rs.)	18	10	12	10
Weight (kg.)	9	4	6	2

Answer :

C) Branch and Bound — Explanation, Algorithm and Worked 0/1-Knapsack Example

1) What is Branch and Bound (BnB)?

Branch and Bound is an optimization technique that systematically explores a state-space tree of partial solutions (branching), and **prunes** any node whose best possible outcome (an upper bound) cannot beat the best solution found so far.

Key points:

- **Branching:** split a node into subproblems (e.g. include an item / exclude an item).
- **Bounding:** compute an optimistic upper bound on the maximum profit reachable from that node (often using the fractional knapsack bound).
- **Selection:** explore the most promising nodes first (best-first using a priority queue by bound) to find good solutions early and prune more effectively.
- Guarantees optimality if bounds are correct.

2) Upper bound used for 0/1-Knapsack

For a node with current profit p and current weight w and next item index i , an upper bound is computed by:

1. Add profits of full items (from i onward) while total weight \leq capacity.
2. If capacity remains, add a fractional part of the next item to fill the bag.

This fractional result is an **upper bound** for the 0/1 problem (you cannot do better in 0/1 than the fractional relaxation).

3) Branch and Bound algorithm (best-first) — pseudocode

BranchBoundKnapsack(items[], n, capacity)

{

sort items by profit/weight ratio (descending)

maxProfit = 0

create root node:

level = 0 // next item index is 0

profit = 0

weight = 0

bound = computeBound(root)

PQ = priority queue ordered by bound (highest first)

PQ.insert(root)

while PQ not empty:

node = PQ.remove() // node with highest bound

if node.bound <= maxProfit:

 continue // prune

// consider next item at index = node.level

if node.level == n: // no further item

 continue

// LEFT child: include item at node.level

left.level = node.level + 1

left.weight = node.weight + wt[node.level]

left.profit = node.profit + profit[node.level]

if left.weight <= capacity:

 if left.profit > maxProfit:

 maxProfit = left.profit

 left.bound = computeBound(left)

 if left.bound > maxProfit:

 PQ.insert(left)

```

// RIGHT child: exclude item at node.level
right.level = node.level + 1
right.weight = node.weight
right.profit = node.profit
right.bound = computeBound(right)
if right.bound > maxProfit:
    PQ.insert(right)

return maxProfit
}

```

computeBound(node) implements the fractional-knapsack upper bound described above.

4) Apply algorithm to the given instance

Item	Profit (Rs.)	Weight (kg)
A	18	9
B	10	4
C	12	6
D	10	2

Capacity = **15 kg**. Items:

Step 1 — sort by profit/weight (ratio):

Compute ratio = profit / weight

- A: $18/9 = 2.00$
- B: $10/4 = 2.50$
- C: $12/6 = 2.00$
- D: $10/2 = 5.00$

Sorted descending by ratio → order: **D, B, A, C**
 (we will refer to items in this order for branching)

Step 2 — root bound (fractional upper bound):

Start with weight = 0, profit = 0, capacity = 15.

Take items in sorted order fractionally to compute bound:

- Take D fully: w=2, p=10
- Take B fully: w=6, p=20
- Take A fully: w=15, p=38 ← now capacity used exactly
No fraction needed.
Root bound = 38. (This is the best possible fractional profit.)

Set maxProfit = 0 initially.

Step 3 — explore nodes (best-first by bound).

We will show the essential node expansions and pruning:

1. **Root node** (level 0, profit=0, weight=0, bound=38) → removed from PQ.
 - Bound (38) > maxProfit(0), so expand.
2. **Left child (include D)**: level=1, weight=2, profit=10.
 - Compute bound for this node by filling fractionally from next items (B, A, C):
 - B (w4,p10) → w=6,p=20
 - A (w9,p18) → w=15,p=38 → fits fully
 - Bound = 38
 - left.profit = 10 → update maxProfit = 10 (since ≤ capacity).
 - left.bound 38 > maxProfit → insert into PQ.

Right child (exclude D): level=1, weight=0, profit=0.

- Bound computed by fractional filling starting from B: B + A + C fractionally:
 - B (w4,p10) → w4,p10
 - A (w9,p18) → w13,p28
 - remaining cap = 2 → take fraction of C ($2/6 \times 12 = 4$)
 - bound = 28 + 4 = 32
- right.bound = 32 > maxProfit(10) → insert into PQ.

PQ now holds nodes with bounds {38 (include D), 32 (exclude D)}.

3. **Remove node with highest bound — include D (level=1, w=2, p=10):**
 - Expand this node.
 - **Left child (include B)**: level=2, weight=6, profit=20.
 - Bound: take A fully: w=15,p=38 → bound 38.

- profit=20 ≤ capacity ⇒ maxProfit updated to 20.
- bound 38 > maxProfit ⇒ insert.
- **Right child (exclude B):** level=2, weight=2, profit=10.
 - Bound: fill with A then C fractionally:
 - A (w9,p18) → w11,p28
 - C has remaining 4 kg → fraction $(4/6)*12 = 8$
 - bound = 28 + 8 = 36
 - bound 36 > maxProfit(20) ⇒ insert.

PQ now contains bounds {38 (include D,B), 36 (exclude D,exclude B), 32 (exclude D)}.

4. Remove next highest-bound node — include D,B (level=2, w=6, p=20):

- **Left child (include A):** level=3, weight=15, profit=38.
 - weight ≤ capacity → profit 38 → update maxProfit = 38.
 - compute bound = 38. Since profit equals bound and equals current best, still valid.
- **Right child (exclude A):** level=3, weight=6, profit=20.
 - Bound: can add C fully? w=12, p=32; remaining cap 3 → no more items → bound 32.
 - bound 32 ≤ maxProfit(38) ⇒ prune (do not insert).

At this point maxProfit = 38. Since 38 equals root bound, and we've found a feasible solution achieving the fractional upper bound, we can be confident (and the queue will prune nodes with bound ≤ 38) that this is optimal.

Remaining PQ nodes have bounds 36 and 32, both ≤ current maxProfit? 36 < 38 so will be pruned when removed; 32 < 38 pruned too. No further nodes can improve.

5) Final optimal solution

Selected items (in original labels): **A, B and D** (we included D, B, A).

Total weight = $2 + 4 + 9 = \mathbf{15 \text{ kg}}$ (fits capacity exactly).

Total profit = $10 + 10 + 18 = \mathbf{38 \text{ Rs.}}$

This equals the fractional bound (38), so it is optimal.

Watch this video to understand how this problem is solved

https://youtu.be/CwM-Mv0Bm4Y?si=_2WUmpkEszXLwO1

Q4)

A) What is sum of subset problem? Solve sum of subset problem for following instance using backtracking approach. Input : set [] = {2, 3, 5, 6, 8, 10}, sum = 10

Answer :

Below is a **perfect exam-oriented answer** with definition, steps, backtracking tree, and final solutions.

A) Sum of Subset Problem – Definition

The **Sum of Subsets problem** is a **backtracking** problem where:

- You are given a **set of positive integers**
- And a **target sum (M)**
- You must find **all subsets of the set whose total sum equals M**

It systematically explores all subsets but **prunes** (cuts off) any branch where:

- The partial sum exceeds the target sum
 - Even if all remaining elements cannot reach the target sum
-

Given Input

Set:

```
[  
S = {2,;3,;5,;6,;8,;10}  
]
```

Target Sum:

```
[  
M = 10  
]
```

Backtracking Approach (Step-by-Step)

We explore each element in order:

Partial set = {} , sum = 0

1) Include 2 → sum = 2

Include 3 → sum = 5

- Include 5 → sum = 10 → ✓ **Solution = {2, 3, 5}**

- Exclude 5
- Include 6 → 11 (exceeds) → prune
- Include 8 → 13 (exceeds) → prune
- Include 10 → 15 (exceeds) → prune

Exclude 3 → sum = 2

- Include 5 → sum = 7
 - Include 6 → 13 (exceeds)
 - Include 8 → 15 (exceeds)
 - Include 10 → 17 (exceeds)
 - Include 6 → sum = 8
 - Include 8 → 16 (exceeds)
 - Include 10 → 18 (exceeds)
 - Include 8 → sum = 10 → ✓ **Solution = {2, 8}**
 - Include 10 → 12 (exceeds)
-

2) Exclude 2 → sum = 0

Include 3 → sum = 3

- Include 5 → sum = 8
 - Include 6 → 14 (exceeds)
 - Include 8 → 16 (exceeds)
 - Include 10 → 18 (exceeds)
- Include 6 → sum = 9
 - Include 8 → 17 (exceeds)
 - Include 10 → 19 (exceeds)
- Include 8 → 11 (exceeds)
- Include 10 → 10 → ✓ **Solution = {10}**

Exclude 3

- Include 5 → sum = 5
 - Include 6 → 11 (exceeds)
 - Include 8 → 13 (exceeds)
 - Include 10 → 15 (exceeds)

- Include 6 → sum = 6
 - Include 8 → 14 (exceeds)
 - Include 10 → 16 (exceeds)
- Include 8 → sum = 8
 - Include 10 → 18 (exceeds)
- Include 10 → sum = 10 → ✓ **Solution = {10}** (duplicate, counted once)

Solutions for S = {2,3,5,6,8,10}, sum = 10:

- ✓ {2, 3, 5}
- ✓ {2, 8}
- ✓ {10}

Watch this video to understand how this problem is solved

<https://youtu.be/kyLxTdsT8ws?si=59Yyw9yZfigvtUuv>

B) Consider the three objects. The weights and associated values are given below.

	weight	value
O ₁	5	6
O ₂	4	5
O ₃	3	4

Assume the Knapsack capacity m = 7. Solve this 0/1 Knapsack problem using LC branch and bound method.

B) 0/1 Knapsack using LC (best-first) Branch & Bound

Given

Object	Weight	Value
O ₁	5	6
O ₂	4	5
O ₃	3	4

Capacity (m = 7).

1. What is LC (Least-Cost) / Best-first BnB (short)

- LC / Best-first BnB keeps a priority queue of live nodes and **expands the most promising node first**.
 - For a **maximization** problem (like knapsack) the “most promising” node is the one with the **highest upper bound** (i.e., the largest possible value achievable from that node using fractional relaxation).
 - Nodes whose bound \leq current best feasible value are **pruned**.
-

2. Preprocessing — sort by value/weight ratio (descending)

Compute ratio (v/w):

- O₁: (6/5 = 1.20)
- O₂: (5/4 = 1.25)
- O₃: (4/3 \approx 1.33)

Sorted order (highest ratio first): O₃, O₂, O₁.

We will consider items in this order.

3. Bound (fractional knapsack upper bound)

For a node with current weight (W) and value (V) and next item index (i):

- Fill remaining capacity greedily with full items in order; if last item doesn't fit, add fractional part.
 - Bound = (V +) (value of full items taken) (+) (possible fractional value).
-

4. Branch & Bound steps (best-first)

Root: level = 0 (next = O₃), (W=0, V=0).

Compute root bound by fractional fill in order O₃, O₂, O₁:

- Take O₃ fully: (W=3, V=4)
- Take O₂ fully: (W=7, V=9) → capacity full.

Root bound = 9.

maxValue = 0. PQ contains root (bound 9).

1) Expand root (bound 9):

- **Left child (include O₃):** level=1, (W=3, V=4).
Bound: fill with O₂ then O₁: O₂ fits → (W=7, V=9). Bound = 9.
Feasible value 4 → maxValue = 4. Insert node (bound 9).
- **Right child (exclude O₃):** level=1, (W=0, V=0).
Bound: take O₂ (W=4, V=5), remaining cap 3 → take fraction of O₁ (3/5 of 6 = 3.6).
Bound = (5 + 3.6 = 8.6). Insert node (bound 8.6).

PQ now: nodes with bounds {9 (include O₃), 8.6 (exclude O₃)}.

2) Expand node with highest bound = include O₃ (W=3, V=4, bound=9):

- **Left child (include O₂ as well):** level=2, (W=3+4=7, V=4+5=9).
Feasible and fits capacity → update maxValue = 9. Bound = 9. (Insert or can note as best solution.)
- **Right child (exclude O₂):** level=2, (W=3, V=4).
Bound: remaining cap 4 → take fraction of O₁: (4/5)*6 = 4.8 → bound = 4 + 4.8 = 8.8.
Since bound 8.8 < current maxValue = 9, prune (do not insert).

PQ now: node with bound 8.6 (exclude O₃). But 8.6 < maxValue 9 → will be pruned when considered.

No remaining node has bound > 9, so algorithm stops.

5. Final (optimal) solution

Selected items (in original labels): **O₂ and O₃** (we included O₃ and O₂).

- Total weight = (3 + 4 = 7)
- Total value = (4 + 5 = 9)

Since this equals the fractional upper bound (9) found at root, it is optimal.

Watch this video to understand how this problem is solved

https://youtu.be/yV1d-b_NeK8?si=xt06YrQe1VJ1cqRg

[Join Whatsapp Group](#)

C) Explain Branch and Bound method. List major drawbacks. Write control abstraction for Least Cost Search.

Branch and Bound (B&B) is a systematic search technique used to solve optimization problems such as:

- Knapsack problem
- Travelling Salesman Problem (TSP)
- Job scheduling
- Assignment problem

It works by exploring the **state space tree** and applies two key ideas:

1. Branching

- The algorithm splits a problem into smaller subproblems.
- Each subproblem corresponds to a **node** in the state-space tree.

2. Bounding

- For every node, a **bound** (best possible answer from that node) is calculated.
- If the bound is worse than the best known feasible solution, the node is pruned (ignored).
- This avoids exploring unnecessary paths → increases efficiency.

💡 Example (Simple Explanation)

Suppose we want to **maximize profit** (Knapsack problem).

At each node:

- **Branch:** include or exclude next item.
- **Bound:** compute upper-bound profit for this choice.
- If bound < current best → **prune**.

(You don't need to show the full tree unless asked; principle matters more.)

Major Drawbacks of Branch and Bound

Drawback	Explanation
1. Depends on bounding function	If the bound is not tight or efficient, a large number of nodes are explored.
2. Worst-case time is exponential	B&B does not guarantee polynomial time; worst case explores complete tree.
3. Large memory requirement	Needs to store multiple active nodes → huge memory for large problems.
4. Overhead of calculating bounds	Computing bounds repeatedly slows down execution.
5. Performance depends on problem structure	Works well only when pruning is effective.

Control Abstraction for Least Cost Search (Branch and Bound)

Control Abstraction: Least Cost Search

LC_Search():

1. Initialize:

Add root node to Priority Queue (PQ)

PQ is ordered by increasing cost (min-heap)

2. Repeat until PQ is empty:

a) Remove node N from PQ having the least cost

b) If N is a solution node:

return N // Optimal solution found

c) Else:

Generate all children of N

For each child:

Compute cost (or bound)

If child is promising:

 Insert child into PQ

3. If PQ becomes empty:

 return FAILURE

Explanation of Control Abstraction Steps

- **Priority Queue** ensures the node with **minimum cost / best bound** is explored first.
 - At each iteration:
 - Extract best node.
 - If solution → stop (optimality guaranteed).
 - Otherwise, generate children and insert them into PQ.
 - Search ends when:
 - A solution is found → **optimal**.
 - PQ becomes empty → **no solution**.
-

Q5)

A) Explain amortized analysis? Explain aggregate and potential function methods used for amortized analysis with respect to stack operations

Answer:

Amortized Analysis (Simple and Human-Written)

- Amortized analysis is a way of finding the average cost of an operation in the long run, even if some individual operations are expensive.
- Amortized analysis is a method for analyzing algorithms that averages the cost of operations over a sequence, rather than considering the worst-case cost of individual operations.
- It provides a worst-case upper bound on the average cost of an operation in a sequence, even if some individual operations are very expensive.
- Instead of looking at the worst case of every single operation, we look at the overall performance of a sequence of operations.
- This is useful when most operations are cheap, but occasionally one operation becomes costly.
- A common example is stack operations with multipop, or dynamic array resizing. The goal is to show that the total cost spread over all operations remains small and predictable.
- There are three common methods to perform amortized analysis: aggregate, accounting, and potential. All three reach the same conclusion, but each uses a slightly different way to justify the cost.

Aggregate Method

In the aggregate method, we simply look at a group of n operations and find their total cost. Once we know the combined cost, we divide it by n to get the amortized cost per operation.

Consider a stack that supports push, pop, and multipop.

A push adds one element, pop removes one, and multipop removes several elements in one go.

If we perform a sequence of n stack operations, the costly operation in this set is multipop. But even multipop cannot remove more elements than were inserted earlier. For n operations, the maximum number of pushes is n , and the maximum number of pops or removals in multipop together can also be at most n .

So the total number of element removals is limited by how many were inserted.

Therefore, in n operations, the total work done is at most n pushes plus n pops. The total cost is at most $2n$, so the amortized cost is a constant. The conclusion is that even though multipop may seem expensive, its cost gets absorbed in the overall sequence, giving a constant amortized time per operation.

Potential Method

The potential method works like a “bank balance” idea. Every operation may increase or decrease the potential of the data structure. When the potential builds up, it pays for future costly operations.

For a stack, a natural potential can be the number of elements currently stored.

More elements on the stack means higher potential, because those elements might need to be popped later.

To find amortized cost, we take the actual cost of the operation and add the change in potential.

When we push an element:

Actual cost is one unit.

Potential increases by one because the stack grows.

So the amortized cost becomes one unit for the push plus the increase in potential.

This gives a simple constant amortized cost.

When we pop an element:

Actual cost is one unit.

Potential decreases by one because the stack becomes smaller.

So the amortized cost is the actual cost minus the drop in potential.

This again results in a constant amortized cost.

For multipop, even though it can remove many elements, each pop decreases the potential, which helps pay for part of the operation. The expensive part is balanced by the stored potential from earlier pushes. As a result, the amortized cost of multipop also becomes constant.

The overall conclusion is that even in the potential method, like the aggregate method, every stack operation ends up having constant amortized time. The difference is simply in how we justify it. The potential method gives a more detailed explanation by using the state of the data structure to carry forward or pay back cost.

B) Explain randomized algorithms? Enlist and explain in brief the primary reasons for using randomized algorithms."

Answer :

- A randomized algorithm is an algorithm that makes decisions using random choices during its execution. Instead of always following a fixed path, it uses random numbers to decide how to proceed in certain steps.
- Because of this, the algorithm may behave slightly differently on different runs even when the input is the same.
- The purpose of adding randomness is not to make the algorithm unpredictable, but to avoid the patterns and worst-case situations that affect some deterministic algorithms.
- Randomized algorithms are especially useful when the input data or structure might cause repeated worst-case behaviour in traditional methods.
- By introducing randomness, the expected running time often becomes much better and more stable. Examples include randomized quick sort, randomized search algorithms, and randomized hashing techniques.

Primary Reasons for Using Randomized Algorithms

1. Better average or expected performance

Some deterministic algorithms have a very bad worst case. Randomization helps avoid these worst-case patterns most of the time. For example, randomized quick sort avoids consistently picking the same poor pivot, giving a good expected running time.

2. Simplicity of design

Randomized algorithms are often easier to write and understand than complicated deterministic versions. Many difficult problems become easier when randomness is used to guide the solution, such as randomized primality testing or randomized selection.

3. Handling unpredictable or adversarial inputs

In some situations, the input may be arranged in a way that forces a deterministic algorithm to perform poorly. Randomized algorithms break this pattern because the choice depends on randomness, not the input. This makes them more robust.

4. Good performance on large data sets

When the amount of data is huge, perfect deterministic strategies may be slow or expensive. Randomized techniques such as random sampling work faster while still giving highly accurate results.

5. Useful in distributed and parallel systems

Random choices help avoid conflicts, such as two processors choosing the same resource. Randomization reduces waiting time and improves overall system performance.

C) Write short notes on the following.

- i) Aggregate Analysis
- ii) Potential Function method
- iii) Accounting Method
- iv) Tractable and Non-tractable Problems

(i) Aggregate Method

- The aggregate method studies the total work done by a sequence of operations instead of looking at each operation individually. The idea is to show that, although some operations may be costly, the overall work for n operations is limited, so the average cost per operation becomes small and predictable.
- You first count the total work performed by all n operations. Then you divide this total by n to get the amortized cost. This method works well when the number of costly operations is naturally limited and cannot exceed some simple bound.
- Example:
If a stack allows push and pop, and you perform n operations, the total number of pops cannot exceed the number of pushes. So the total work stays within a constant multiple of n .

(ii) Potential Function Method

- The potential method assigns a “potential” to the current state of the data structure. This potential represents stored work that will either be needed later or that was paid for earlier. The amortized cost of each operation is calculated using two things: the actual work done and the change in potential.
- If an operation increases potential, it means we are saving effort for the future. If it decreases potential, it means earlier stored work is now being used to pay for part of the current operation. When you analyze an entire sequence of operations, the total potential never becomes negative, so the amortized cost correctly reflects the actual total cost.
- example:
For a stack, potential can simply be the number of elements currently in the stack.
Push increases the potential; pop decreases it.

(iii) Accounting Method

- The accounting method assigns a fixed “charge” to each operation. Some operations are charged more than their real cost, and the extra amount is saved as credit. Later, expensive operations use this stored credit so that the overall cost remains balanced.
- You choose charges in such a way that the total credit is always enough to pay for operations that would otherwise be expensive. This ensures that each operation has a stable amortized cost, even if actual costs vary.
- Example:
You may charge slightly more for each push in a stack so that the extra credit can pay for future pops.

(iv) Tractable and Non-tractable Problems

- A problem is called **tractable** when it can be solved efficiently using an algorithm whose running time grows at a reasonable rate as the input size increases. In theory, these are problems that can be solved in **polynomial time**, meaning their time complexity is something like n , n square, or n cube. Such problems are considered practical because even if the input becomes large, the growth in running time stays manageable.
- A problem is called **non-tractable** when no known algorithm can solve it in polynomial time. These problems usually require **exponential** or **factorial** time, which grows extremely fast as input size increases. Because of this rapid growth, these problems become impossible to solve exactly for even moderate-sized inputs. Many of these problems belong to the NP-hard or NP-complete categories

Q6)

A) What is Potential function method of amortized analysis? To illustrate Potential method, find amortized cost of PUSH, POP and MULTIPOP stack operation

Potential-function method

- The **Potential Function Method** is a technique used in **amortized analysis** to determine the time complexity of a sequence of operations on a data structure
- It smooths out the cost of infrequent but expensive operations by associating a measure of "prepaid work" or "potential energy" with the data structure as a whole.

The potential method assigns a non-negative *potential* $\Phi(\text{state})$ to the data structure's state. For an operation that moves the structure from state (S) to (S') :

$$\text{amortized_cost} = \text{actual_cost} + \Phi(S') - \Phi(S).$$

- Intuition: the potential stores "prepaid work" (credits). A cheap operation can increase potential (save credit); an expensive operation can decrease potential (use credit).
- Over any sequence of operations the sum of amortized costs \geq sum of actual costs (with appropriate initial/final potential choices), so bounding amortized cost bounds total real cost.

Key Concepts

- **Potential Function (Φ)**: A function that maps the state (D) of the data structure to a non-negative real number $(\Phi(D) \geq 0)$.
 - It represents the total amount of prepaid work or credit stored in the data structure
 - The potential of the initial state (D_0) is usually defined as zero, $\Phi(D_0) = 0$.
- **Actual Cost (c_i)**: The actual time or cost of the i -th operation.
- **Amortized Cost (\hat{c}_i)**: The cost assigned to the i -th operation, defined by the formula:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Example: Stack with PUSH, POP, MULTIPOP

Model / costs:

- PUSH: actual cost = 1 (one push)
- POP: actual cost = 1 (one pop)
- MULTIPOP(k): pops up to k items; actual cost = $(t = \min(k, \text{current_size}))$ (one per popped item)

Choose potential function:

1) PUSH

- Actual cost = 1.
- New size = $s + 1 \Rightarrow \Delta\Phi = (s + 1) - s = +1$.
- Amortized cost = actual + $\Delta\Phi = 1 + 1 = 2$.

2) POP

- Preconditions: assume stack nonempty (if empty, POP may be disallowed or cost 0 — handle separately).
- Actual cost = 1.
- New size = $s - 1 \Rightarrow \Delta\Phi = (s - 1) - s = -1$.
- Amortized cost = $1 + (-1) = 0$.

(If POP is called on an empty stack and is defined to do nothing, actual = 0, $\Delta\Phi = 0 \Rightarrow$ amortized = 0.)

3) MULTIPOP(k)

- Let $t = \min(k, s)$ be number of items actually popped.
- Actual cost = t .
- New size = $s - t \Rightarrow \Delta\Phi = (s - t) - s = -t$.
- Amortized cost = actual + $\Delta\Phi = t + (-t) = 0$.

B) Explain approximation algorithm? How does performance ratios useful in approximation algorithms?

- An **approximation algorithm** is an algorithm designed to find a solution that is **close to optimal** for computational problems where finding the exact optimal solution is **very hard or impossible to compute efficiently**.
- These problems are typically **NP-Hard** (e.g., Travelling Salesman, Vertex Cover, Knapsack, Set Cover, etc.).
For NP-hard problems, no known polynomial-time exact algorithm exists. Therefore, we use **approximation algorithms** to compute a “good enough” solution **in polynomial time**.

Key characteristics of Approximation Algorithms

1. **Runs in polynomial time**
→ Much faster than exponential-time exact algorithms.
2. **Guarantees closeness to optimal**
→ Provides theoretical guarantee that the solution will not be too far from the optimal solution.
3. **Quality measured by approximation ratio/performance ratio**
→ This ratio tells how close the algorithm’s solution is to the optimal one.
4. **Used when exact solution is expensive or infeasible.**

Why do we use Approximation Algorithms?

- Exact algorithms for NP-hard problems take exponential time.
- In real-life applications (network routing, resource allocation, scheduling, cloud computing, machine learning), we need fast solutions.
- Approximation algorithms give solutions that are practically acceptable

2) Performance Ratio

The **performance ratio** measures **how close the output of an approximation algorithm is to the optimal solution**.

Let:

- C = cost of solution obtained by approximation algorithm
- C^* = cost of optimal solution (best possible)

For Minimization Problems:

$$\text{Performance Ratio } R = \frac{C}{C^*}$$

Since $C \geq C^*$, the ratio $R \geq 1$.

For Maximization Problems:

$$R = \frac{C^*}{C}$$

Since $C^* \geq C$, again $R \geq 1$.

Definition:

An approximation algorithm is called a **ρ -approximation algorithm** if for every input:

$$1 \leq R \leq \rho$$

Where ρ (rho) is the **approximation bound**.

Example:

If an algorithm is a **2-approximation** algorithm, it means:

- Its result is **at most 2 times worse** than the optimal (for minimization)
- Or **at least half as good** as optimal (for maximization)

How Performance Ratio is Useful in Approximation Algorithms?

1) Measures Quality of the Approximation

It tells **how good or bad** the algorithm is compared to the optimal.

Example:

If $R = 1.5$

→ Solution is at most 50% worse than the optimal.

2) Guarantees worst-case performance

Even in the worst case, the solution will be within factor ρ of optimal.

This gives confidence in using the algorithm in real-life systems.

3) Helps compare different approximation algorithms

If two algorithms A1 and A2 solve the same NP-hard problem:

- A1 has ratio 3
- A2 has ratio 1.5

→ A2 is better because its solution stays closer to optimal.

4) Useful when exact optimal is unknown

Even without computing the true optimal (which might be impossible),
the ratio bounds the quality of the approximation.

5) Helps in algorithm design

When designing new approximation algorithms, the goal is to **reduce the performance ratio** closer to 1.

6) Provides theoretical guarantees

It provides a **mathematical guarantee** of solution quality, making approximation algorithms reliable for critical applications.

C) What is embedded algorithm? Explain Embedded system scheduling using power optimized scheduling algorithm

An **embedded algorithm** is a specially designed algorithm that runs inside an **embedded system** (a system with a dedicated function such as a washing machine controller, pacemaker, vehicle ECU, smart sensor, etc.).

Definition:

An embedded algorithm is a **task-specific algorithm** implemented within an embedded system to control hardware, manage system behavior, process sensor data, and meet real-time constraints under limited resources (memory, CPU, energy).

Key Characteristics of Embedded Algorithms

1. **Real-time behavior** – must respond within strict time limits.
2. **Resource constraints** – optimized for limited CPU speed, RAM, battery.
3. **Reliability & safety** – must work continuously without failure.
4. **Energy efficient** – especially important for battery-powered devices.
5. **Hardware–software interaction** – algorithms often directly control hardware.

Examples of Embedded Algorithms

- Sensor data filtering (Kalman filter, moving average filter)
- Motor control algorithms (PID control)
- Task scheduling algorithms
- Power optimization algorithms
- Communication algorithms (I2C, SPI, UART handling)
- Error detection algorithms

Embedded System Scheduling Using Power-Optimized Scheduling Algorithm

Embedded systems often operate on **limited battery power**, so scheduling algorithms must aim to **reduce energy consumption** while meeting **real-time deadlines**.

Power-optimized scheduling is a scheduling strategy that arranges and executes tasks in such a way that the system **consumes minimum energy** while still meeting all real-time deadlines.

This is extremely important in:

- smartphones
- wearable devices

- IoT sensors
 - medical devices (pacemakers)
 - automotive controllers
-

Key Objectives of Power-Optimized Scheduling

1. Reduce **CPU active time**
 2. Maximize **sleep/idle time**
 3. Lower CPU **frequency/voltage** when possible (DVFS)
 4. Ensure **deadline constraints** of real-time tasks
 5. Reduce **overall energy consumption**
-

Power-Optimized Scheduling Algorithm (Detailed Explanation)

The most widely used strategy is:

Dynamic Voltage and Frequency Scaling (DVFS) based Scheduling Algorithm

This algorithm schedules tasks by reducing the CPU **frequency and voltage** whenever the full processing power is not needed.

Why DVFS Works?

CPU power consumption is approximately:

$$P \propto V^2 f$$

Where:

- V = supply voltage
- f = clock frequency

Reducing **voltage** saves **quadratic** power, and lowering **frequency** reduces linear power.

So even small reductions dramatically save power.

Working of DVFS-Based Power Optimized Scheduling

Step 1: Analyze Tasks

Each task T_i has:

- Execution time C_i
- Period P_i

- Deadline D_i

Compute total utilization:

$$U = \sum \frac{C_i}{P_i}$$

Step 2: Select Minimum CPU Frequency

Set CPU frequency such that:

$$f = U \times f_{max}$$

This guarantees all tasks still meet deadlines **but at lowest power.**

Step 3: Execute Tasks

Tasks run at the reduced frequency → lower energy usage.

Step 4: Use Idle Time Efficiently

If CPU has slack time:

- put CPU in **sleep/idle mode**
- wake only for next scheduled task

Step 5: Re-adjust Frequency Dynamically

If tasks change or new tasks arrive:

- update utilization
- adjust frequency again

Example (Simple Numeric Example)

Tasks:

- $T_1: C_1 = 2 \text{ ms}, P_1 = 10 \text{ ms}$
- $T_2: C_2 = 1 \text{ ms}, P_2 = 5 \text{ ms}$

Compute utilization:

$$U = \frac{2}{10} + \frac{1}{5} = 0.2 + 0.2 = 0.4$$

If CPU max frequency = 1 GHz

Power-optimized frequency = $0.4 \times 1 = 0.4 \text{ GHz}$

Result:

CPU runs at **400 MHz instead of 1 GHz**, saving >50% energy while still meeting deadlines.

(Bonus Question)

Suppose you are working on an embedded system for a medical device that monitors patient vital signs. The device continuously collects data from various sensors and needs to process and display this information in real-time. The data includes timestamps, temperature readings, heart rate, and blood pressure measurements. Suggest suitable sorting algorithm for this scenario. Clearly justify your answer with respect to key factors

Answer :

Suggested Sorting Algorithm: *Insertion Sort*

For a medical embedded system monitoring vital signs in **real-time**, the most suitable sorting algorithm is: **Insertion Sort**

Justification Based on Key Factors

1) Real-Time Requirement (Low Latency)

The device **continuously receives new sensor readings** (temperature, heart rate, BP). Each new reading needs to be inserted **quickly** into the existing sorted list.

Why insertion sort is best:

- It efficiently handles **online data** (data arriving one element at a time).
- Inserting a new measurement takes **O(n)** worst-case but usually much faster because sensor data changes slowly.
- Sorting can be maintained **incrementally**, not recomputing full sorts.

Most other algorithms (Merge Sort, Heap Sort, Quick Sort) require full re-sorting or complex data structures.

2) Small Input Size (Embedded Systems)

Medical devices typically store only the last few seconds or minutes of readings (small data windows).

Why insertion sort fits:

- Performs extremely well on **small** datasets.
 - Has very low constant overhead.
 - Beats more complex algorithms when data size is small ($n < 1000$, usually true for real-time windows).
-

3) Nearly-Sorted Data

Vital signs change **gradually**, not abruptly.

So each new timestamped reading is usually close to the correct sorted position.

Insertion sort advantage:

- **Best-case time = O(1)** when the new data is already at the end.
 - Performs extremely well on **nearly-sorted** datasets.
 - Minimizes unnecessary comparisons and shifting.
-

4) Memory Constraints (Very Low Space)

Embedded systems often have limited RAM.

Why insertion sort wins:

- Requires **O(1)** extra memory.
 - No recursive stack, no additional buffers.
 - Much lighter than quicksort or mergesort (which require more memory or recursion).
-

5) Deterministic Timing (Safety-Critical Medical System)

Medical devices must be **predictable**, not just fast.

Insertion sort is deterministic:

- No random pivot selection
- No worst-case stack usage
- No unpredictable branching

Algorithms like quicksort can have **O(n²) unpredictable worst-case**, which is unsafe for medical devices.

6) Simplicity & Reliability

Medical embedded systems favor **simple algorithms** because they:

- reduce chances of bugs
- are easier to validate
- improve reliability

Insertion sort is extremely simple to implement and verify.

7) Supports Continuous Time-Series Sorting

Your data includes:

- timestamps
- temperature readings
- heart rate
- blood pressure

Records can be kept sorted by **timestamp** or **priority** using insertion sort without rebuilding the entire dataset.

Q7)

A) i) Explain an algorithm for Distributed Minimum Spanning Tree.

ii) Write and explain Rabin-Karp algorithm for string matching.

a) i) Distributed Minimum Spanning Tree (MST) Algorithm

Problem Definition

In a **distributed system** (network of nodes), each node knows only:

- Its own ID
- The IDs of its neighbors
- The weights of edges to neighbors

Goal: Find a **Minimum Spanning Tree (MST)** in a **distributed manner**, i.e., each node participates in computing MST, without a central controller.

Distributed MST Algorithm (Gallager, Humblet, Spira - GHS Algorithm)

Overview:

- Each node starts as a **fragment** of the MST.
 - Fragments **merge step by step** via **minimum outgoing edge (MOE)**.
 - Continue until all nodes are in a single MST.
-

Step-by-Step Algorithm

1. Initialization

- Each node is a **fragment** of size 1.
 - Each fragment has a **level = 0**.
 - Each node knows only its neighbors and edge weights.
-

2. Find Minimum Outgoing Edge (MOE)

- For a fragment, find the **lightest edge connecting it to another fragment**.
 - Each node sends messages to its neighbors to detect edges going **outside its fragment**.
 - The minimum-weight edge among these is called **MOE**.
-

3. Merge Fragments

- Two fragments connected by MOE **merge**:
 - New fragment level = max(level of both fragments) + 1
 - Old fragment IDs are updated
 - Update nodes in merged fragment about the new fragment ID
-

4. Repeat

- Repeat **find MOE → merge fragments** until **all nodes belong to a single fragment**.
 - When all nodes belong to a single fragment, the **edges selected during merging form the distributed MST**.
-

Messages in the Algorithm

- INITIATE – to start merging
- TEST – to check if an edge is outgoing

- ACCEPT / REJECT – whether edge can be used for merging
 - REPORT – report MOE to fragment leader
 - CHANGE-ROOT – update fragment structure
-

Key Properties

1. **Distributed computation** – no central node needed.
 2. **Optimal MST** – edges chosen are always minimum-weight between fragments.
 3. **Message complexity:** ($O(E + N \log N)$), time complexity ($O(N \log N)$).
 4. **Safe for asynchronous networks.**
-

a) ii) Rabin-Karp Algorithm for String Matching

Problem

Given:

- Text string $T[0..n-1]$
- Pattern $P[0..m-1]$

Goal: Find all occurrences of P in T efficiently.

Idea

Rabin-Karp uses **hashing**:

- Compute **hash of pattern P** $\rightarrow \text{hash}(P)$
- Compute **hash of substrings of T** of length m
- If hashes match, check actual substring to avoid **hash collision**

This allows **average-case $O(n)$ matching**.

Step-by-Step Algorithm

1. Choose a Hash Function

- Example: use **rolling hash**:

$$H(s) = (s_0 \cdot d^{m-1} + s_1 \cdot d^{m-2} + \dots + s_{m-1}) \bmod q$$

Where:

- d = number of characters in alphabet
 - q = large prime number
 - s_i = ASCII value of character at position i
-

2. Compute Pattern Hash

- Compute $\text{hash}(P)$ using the above hash function.

3. Compute Initial Text Window Hash

- Compute hash for $T[0..m-1]$.

4. Slide Window Over Text

For each $i = 0$ to $n-m$:

1. Compare $\text{hash}(T[i..i+m-1])$ with $\text{hash}(P)$

- If hashes match → check actual substring to confirm match
 - Else → no match
2. Compute next window hash **efficiently using rolling hash**:
- $$\text{hash}(T[i+1..i+m]) = (d \cdot (\text{hash}(T[i..i+m-1]) - T[i] \cdot d^{m-1}) + T[i+m]) \bmod q$$
-

5. Output Matches

- Whenever the substring matches after hash comparison, report the index.

Key Features

1. **Average-case time complexity** = $O(n + m)$
 2. **Worst-case time complexity** = $O(nm)$ (if many hash collisions)
 3. **Uses hash to avoid unnecessary comparisons**
 4. **Good for multiple pattern matching** (can compute multiple hashes)
 5. **Simple to implement using modular arithmetic**
-

Example

Text: "ABABDABACDABABCABAB"

Pattern: "ABABCABAB"

Steps:

1. Compute $\text{hash}(\text{"ABABCABAB"})$
2. Slide window over text and compute rolling hashes:
 - "ABABDABAC" → hash mismatch → skip
 - "BABDABACD" → mismatch → skip
 - ...
 - "ABABCABAB" → hash match → verify → report index

B) With respect to Multithreaded Algorithms explain Analyzing multithreaded algorithms, Parallel loops, Race conditions.

Answer :

Multithreaded Algorithms

A **multithreaded algorithm** is designed to execute multiple tasks simultaneously using **threads** to improve **performance** and **parallelism** on multicore or multiprocessor systems.

Key points:

- Threads share memory space of a process.
- Tasks can run **concurrently**, but shared resources must be handled carefully.
- Common in high-performance computing, web servers, real-time systems, and embedded systems.

1) Analyzing Multithreaded Algorithms

When analyzing a multithreaded algorithm, we focus on **time complexity, speedup, work, and span**.

Key Metrics

1. Work (T_1)

- Total time taken by the algorithm if executed on a **single thread**.
- Essentially the sequential execution time.

2. Span (T_∞) (Critical Path Length)

- Time taken by the algorithm if we have **infinite threads** (perfect parallelism).
- Longest chain of dependent tasks in the computation.

3. Parallelism

$$\text{Parallelism} = \frac{\text{Work}}{\text{Span}} = \frac{T_1}{T_\infty}$$

- Measures the **maximum theoretical speedup** achievable.
- Higher parallelism \rightarrow more threads can help reduce execution time.

Example

Suppose a computation involves 16 tasks:

- T_1 (work) = 16 units
- T_∞ (span) = 4 units (longest dependent chain)

Then **parallelism** = $16 / 4 = 4 \rightarrow$ maximum speedup $\sim 4x$.

Tools for Analysis

- **DAG (Directed Acyclic Graph)**: represents tasks and dependencies.
 - Nodes = tasks
 - Edges = dependencies
- **Greedy scheduling bounds**:

$$T_P \leq \frac{T_1}{P} + T_\infty$$

where T_P is execution time on P processors.

2) Parallel Loops

A **parallel loop** executes iterations concurrently instead of sequentially.

Definition

A loop of the form:

```
for (i = 0; i < n; i++)
    do_something(i);
```

can be parallelized if **iterations are independent** (no data dependency between iterations).

Requirements for Parallel Loops

1. **Loop iterations independent** – no read/write conflicts.
2. **Minimal synchronization overhead** – frequent locking reduces speedup.
3. **Uniform workload distribution** – prevents some threads from being idle.

Example

Sequential loop:

```
for (i = 0; i < n; i++)
    A[i] = B[i] + C[i];
```

Parallelized (OpenMP style):

```
#pragma omp parallel for
for (i = 0; i < n; i++)
    A[i] = B[i] + C[i];
    • Each iteration can run on a separate thread safely because no iteration depends on another.
```

3) Race Conditions

Definition

A **race condition** occurs when multiple threads **access shared data concurrently**, and the **final result depends on the order of execution**, leading to **unpredictable or incorrect behavior**.

Example

```
int counter = 0;
```

Thread 1: counter = counter + 1;

Thread 2: counter = counter + 1;

- Both threads read counter = 0 simultaneously.
- Each adds 1 → writes 1 back → final counter = 1 (instead of 2).

This is a **race condition**.

Causes

1. **Concurrent read/write to shared variables**
2. **Lack of proper synchronization (locks, mutexes, atomic operations)**
3. **Non-deterministic thread scheduling**

Prevention

1. **Locks / Mutexes** – only one thread accesses critical section at a time.
2. **Atomic operations** – hardware-supported indivisible operations.
3. **Thread-safe data structures** – built-in synchronization.
4. **Barrier synchronization** – ensure all threads reach a point before proceeding.

C) Write short notes on the following.

i) Multithreaded matrix multiplication

ii) Multithreaded merge sort.

iii) Distributed breadth first search

iv) The Rabin-Karp algorithm.

Answer :

1) Multithreaded Matrix Multiplication

Problem

Given two matrices $A(n \times m)$ and $B(m \times p)$, compute the product $C = A \times B$ in parallel using multiple threads.

Sequential Approach

$$C[i][j] = \sum_{k=0}^{m-1} A[i][k] \cdot B[k][j]$$

- Time complexity: $O(n \cdot m \cdot p)$
 - Computed sequentially row by row, column by column.
-

Multithreaded Approach

Idea: Different threads compute **independent elements or blocks** of the result matrix simultaneously.

Methods:

1. Thread per Element

- Each thread computes **one element** $C[i][j]$.
- Number of threads = $n \times p$.
- Each thread executes:

$C[i][j] = 0;$

for ($k=0$; $k < m$; $k++$)

$C[i][j] += A[i][k] * B[k][j];$

- Pros: Maximum parallelism
 - Cons: Thread overhead is very high for large matrices
-

2. Thread per Row

- Each thread computes **one row** of C.
- Number of threads = n (rows).
- Each thread executes all elements of the row:

```
for (j=0; j<p; j++)  
    for (k=0; k<m; k++)  
        C[i][j] += A[i][k] * B[k][j];
```

- Pros: Less thread overhead, simpler synchronization
 - Cons: Slightly less parallelism than per-element
-

3. Blocked / Chunked Threads

- Divide matrix into **blocks** (submatrices).
 - Each thread computes one block.
 - Best for **large matrices** to optimize cache usage.
-

Synchronization

- Minimal or none because each thread writes to **disjoint elements** of C.
- Only need to **join threads** at the end to ensure all computations complete.

Advantages

1. Speedup proportional to number of threads (up to hardware limit)
 2. Exploits **data independence** in matrix multiplication
 3. Can use **multicore CPUs efficiently**
-

2) Multithreaded Merge Sort

Problem

Sort an array $A[0..n - 1]$ using **multithreading**.

Sequential Merge Sort

- Divide array into two halves
- Recursively sort left and right halves
- Merge two sorted halves

- Time complexity: $O(n \log n)$
-

Multithreaded Approach

Idea: Recursively sort **left and right halves in parallel**.

Steps

1. **Divide array** into left and right halves.
2. **Create two threads:**
 - Thread 1 → sort left half
 - Thread 2 → sort right half
3. **Merge sorted halves** (single thread or main thread).

Pseudocode:

```
function parallelMergeSort(A, low, high):  
    if low < high:  
        mid = (low + high)/2  
        // Create threads to sort halves  
        thread t1: parallelMergeSort(A, low, mid)  
        thread t2: parallelMergeSort(A, mid+1, high)  
        wait for t1 and t2 to finish  
        merge(A, low, mid, high)
```

Optimizations

1. **Threshold size:** For small subarrays, use sequential sort to reduce thread overhead.
 2. **Thread pool:** Limit number of threads to number of CPU cores.
 3. **Parallel merge:** Can also parallelize merging step for very large arrays.
-

Analysis

- Maximum parallelism = $O(\log n)$ levels of recursion
 - Total work = $O(n \log n)$ (same as sequential)
 - Span (critical path) $\approx O(\log^2 n)$
-

3) Distributed Breadth-First Search (BFS)

Problem

Given a **large graph** distributed across multiple machines, perform BFS starting from a source vertex s .

Sequential BFS

1. Initialize queue Q with s
 2. While Q not empty:
 - o Dequeue vertex u
 - o For each neighbor v of u :
 - If v not visited, mark visited and enqueue v
-

Distributed BFS

Idea: Partition graph across machines. Each machine stores **a subset of vertices and edges**. BFS proceeds **level by level**, communicating between machines.

Steps

1. **Initialization:**
 - o Mark source vertex s as visited on its machine
 - o Add s to **local frontier**
 2. **Level-synchronous BFS:**
 - o Each machine processes its **local frontier**
 - o For each neighbor of frontier vertices:
 - If neighbor belongs to same machine → mark visited locally
 - If neighbor belongs to another machine → send message to that machine
 3. **Communication:**
 - o Machines exchange newly discovered vertices at each BFS level
 - o Continue until frontier is empty across all machines
-

Key Points

- **Communication between machines** is main overhead
- BFS proceeds **in parallel on all partitions**
- Can use **asynchronous or synchronous variants** depending on network latency

4) Rabin-Karp Algorithm

Problem

Find all occurrences of a pattern P of length m in a text T of length n.

Idea

- Use **hashing** to avoid repeated comparisons
 - Compute **hash of P**
 - Compute **rolling hash of substrings of T** of length m
 - Compare hashes → if equal, verify by character comparison
-

Algorithm Steps

1. **Choose hash function:**

$$H(s) = (s_0 \cdot d^{m-1} + s_1 \cdot d^{m-2} + \dots + s_{m-1}) \bmod q$$

- d = alphabet size, q = large prime

2. **Compute hash(P)**

3. **Compute initial hash(T[0..m-1])**

4. **Slide window over text:**

For each i = 0 to n-m:

- If $\text{hash}(T[i..i+m-1]) == \text{hash}(P)$ → verify substring match

- Compute next window hash using **rolling hash**:

$$\text{hash}(T[i+1..i+m]) = (d \cdot (\text{hash}(T[i..i+m-1]) - T[i] \cdot d^{m-1}) + T[i+m]) \bmod q$$

5. **Output indices** where match occurs

Complexity

- Average-case: $O(n + m)$
- Worst-case: $O(nm)$ (rare, when many hash collisions)
- Efficient for **single or multiple patterns**

Q8)

Q8) a) Consider the graph represented by an adjacency matrix:

[10]

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	0	0	0	0
E	0	1	0	0	0	0	0
F	0	0	1	0	0	0	0
G	0	0	1	0	0	0	0

Show stepwise process how the distributed breadth first search algorithm works on the above graph.

Edges:

- $A \rightarrow B, C$
- $B \rightarrow A, D, E$
- $C \rightarrow A, F, G$
- $D \rightarrow B$
- $E \rightarrow B$
- $F \rightarrow C$
- $G \rightarrow C$

Step 1: Initialize BFS

- Start at **A**.
- BFS uses a **queue** to explore level by level.
- Maintain **visited set** and **distance from root** (for distributed BFS, think of each node sending messages to neighbors to inform them of the distance).

Queue: [A]

Visited: {A}

Level: 0

Step 2: Explore Neighbors of A (Level 1)

Neighbors of A: B, C

- B and C are **not visited**, so add them to queue.

- Mark B and C as visited.
- Set their **parent** as A (or distance = 1 from A).

Queue: [B, C]

Visited: {A, B, C}

Level: 1

Distributed BFS idea: Node A sends messages to B and C that "distance from root = 1".

Step 3: Explore Neighbors of B (Level 2)

Dequeue B:

Neighbors of B: A, D, E

- A already visited → ignore
- D and E not visited → enqueue D, E, mark visited
- Parent of D and E = B (distance = 2)

Queue: [C, D, E]

Visited: {A, B, C, D, E}

Level: 2

Distributed BFS: B sends messages to D and E: "distance from root = 2"

Step 4: Explore Neighbors of C (Level 2)

Dequeue C:

Neighbors of C: A, F, G

- A already visited → ignore
- F and G not visited → enqueue F, G, mark visited
- Parent of F and G = C (distance = 2)

Queue: [D, E, F, G]

Visited: {A, B, C, D, E, F, G}

Level: 2

Distributed BFS: C sends messages to F and G: "distance from root = 2"

Step 5: Explore D, E, F, G (Level 3)

Dequeue D: neighbors = B (already visited) → nothing to add

Dequeue E: neighbors = B (already visited) → nothing to add

Dequeue F: neighbors = C (already visited) → nothing to add

Dequeue G: neighbors = C (already visited) → nothing to add

Queue: []

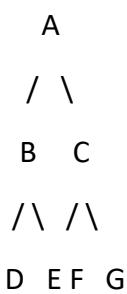
Visited: {A, B, C, D, E, F, G}

Level: 3

- Queue is empty → BFS ends
-

Step 6: BFS Tree / Result

BFS Tree (parent representation):



Distance from A (level-wise):

- Level 0: A
- Level 1: B, C
- Level 2: D, E, F, G

b) What do you understand by spawn and sync keywords used in multithreaded programming? Explain with the help of suitable example.

Answer :

In **multithreaded programming**, tasks can run **concurrently** to improve performance. Traditional threads require manual creation, joining, and management. To simplify **divide-and-conquer parallelism**, languages like **Cilk** introduced two important keywords:

- **spawn** → to create parallel tasks
- **sync** → to wait for all parallel tasks to complete

These keywords allow programmers to express **parallelism declaratively**, without managing low-level threads.

2. The spawn Keyword

- **Definition:** spawn marks a function call to run as a **separate task** concurrently with the current task.
- **Behavior:**
 1. The function call following spawn can execute in **parallel**.
 2. The **parent thread does not block**; it continues executing subsequent instructions.
- **Use Case:** Often used in **recursive divide-and-conquer algorithms** where independent tasks can run simultaneously.

Example Concept:

```
x = spawn functionA(); // functionA runs in parallel
```

```
y = functionB(); // functionB runs in current thread
```

Here, functionA and functionB execute in parallel.

3. The sync Keyword

- **Definition:** sync ensures that **all previously spawned tasks in the current function** are completed before moving forward.
- **Behavior:**
 1. Acts as a **barrier**.
 2. Guarantees correctness when the parent task needs results from spawned tasks.

Example Concept:

```
x = spawn functionA();
y = functionB();
sync; // Wait until functionA completes
result = x + y;
```

Without sync, the program might try to use x before functionA finishes → leads to incorrect computation.

4. How Spawn and Sync Work Together

1. Parent thread executes up to a spawn statement → creates a new **child task**.
2. Parent continues execution **without waiting** for the child.
3. Multiple spawn statements create multiple child tasks.
4. When sync is encountered, the parent **waits for all spawned child tasks to finish**.
5. After sync, parent thread can safely use results from all spawned tasks.

Key Idea:

- spawn → **parallel task creation**
 - sync → **ensure completion**
-

5. Example: Parallel Fibonacci

```
#include <stdio.h>

// Compute Fibonacci using parallel recursion
int fib(int n) {
    if (n <= 1)
        return n;

    int x, y;

    // Spawn a new task for fib(n-1)
    x = cilk_spawn fib(n - 1);

    // Compute fib(n-2) in current thread
    y = fib(n - 2);

    // Wait for all spawned tasks (fib(n-1)) to complete
    cilk_sync;

    return x + y;
}

int main() {
    int n = 5;
    printf("Fibonacci(%d) = %d\n", n, fib(n));
    return 0;
}
```

- c) Write a Rabin-Karp string matching algorithm. Input to the algorithm be: Original text "t" of length n and pattern text being matched is "p" of length m. What is the expected runtime and worst-case runtime of this algorithm?

Algorithm Steps:

Calculate Pattern Hash:

Compute a hash value for the pattern p. This typically involves a polynomial rolling hash function with a chosen base and a large prime modulus to reduce the chance of collisions.

Calculate Initial Text Substring Hash:

Compute the hash value for the first m-character substring of the text t.

Iterate and Compare:

- Compare the hash of the current text substring with the hash of the pattern.
- If the hashes match, perform a character-by-character comparison of the pattern and the current text substring to confirm a true match (as hash collisions are possible).
- If the hashes do not match, or if the character-by-character comparison reveals a mismatch, use a rolling hash technique to efficiently calculate the hash of the next m-character substring in t in constant time. This involves subtracting the contribution of the first character of the previous window and adding the contribution of the new character entering the window.

Repeat:

Continue this process until all possible m-character substrings in t have been examined.

Runtime Analysis:

Expected Runtime: $O(n + m)$

The expected runtime occurs when hash collisions are rare. Calculating the initial hashes of the pattern and first text window takes $O(m)$, and each subsequent rolling hash update takes $O(1)$. Since there are $n - m + 1$ windows, the total time for hash comparisons is $O(n)$. Character-by-character comparisons are infrequent, so the overall expected runtime is $O(n + m)$.

Worst-Case Runtime: $O(nm)$

The worst case happens when many hash collisions occur, requiring character-by-character comparison for almost every window. Each comparison takes $O(m)$, and there are about n windows, giving a worst-case runtime of $O(nm)$, similar to the naive algorithm.

(Bonus Questions and answers)

“Write multithreaded merge sort algorithm. Briefly discuss how does it differ from conventional merge sort.”

Input: Array A[0..n-1]

Output: Sorted array A

Algorithm Steps

1. **Start** with the full array A[0..n-1].
2. **Check base condition:**
 - o If the array has only 1 element → it is already sorted → return.
3. **Divide** the array into two halves:
 - o Left half: A[0..mid]
 - o Right half: A[mid+1..n-1]
4. **Spawn two parallel threads:**
 - o Thread 1: Sort the left half.
 - o Thread 2: Sort the right half.
5. **Wait (sync)** for both threads to complete sorting.
6. **Merge** the two sorted halves into a single sorted array.
7. **Return** the sorted array.

Key Points:

- Parallelism is achieved by sorting the left and right halves **simultaneously**.
- Step 5 ensures both halves are fully sorted before merging.

Difference from conventional merge sort:

- Conventional merge sort sorts halves **sequentially**.
- Multithreaded version sorts halves **in parallel**, reducing runtime on multicore systems.

“Write a simple multithreaded matrix multiplication algorithm based on parallelizing relevant loops of the conventional procedure.

Input: Matrices A[n×m] and B[m×p]

Output: Matrix C[n×p] = A × B

Algorithm Steps

1. **Start** with matrices A and B.
2. **Create result matrix** C[n][p] and initialize all entries to 0.
3. **Parallelize outer loop (rows of result matrix):**
 - o For each row i of matrix C, spawn a thread to compute row i.
4. **For each thread (computing a row):**
 - o For each column j of matrix C:
 - Initialize C[i][j] = 0.
 - Compute the dot product of row i of A and column j of B:
5. **Wait (sync)** for all threads to finish computing their respective rows.
6. **Return** the final result matrix C.

Key Points:

- Parallelism is achieved by **computing multiple rows simultaneously**.
- Each thread works independently on a row → no conflicts in shared memory.

Difference from conventional matrix multiplication:

- Conventional algorithm computes **all rows sequentially**.
- Multithreaded version computes rows **in parallel**, reducing runtime for large matrices.

Join Community by clicking below links 

[Website Link](#)

👉 For question papers and notes visit website

[Click here](#)

[Telegram Channel](#)

[Click here](#)

[WhatsApp Channel](#)

[Click here](#)

Join Community by clicking below links 

[Website Link](#)

👉 For question papers and notes visit website

[Click here](#)

[Telegram Channel](#)

[Click here](#)

[WhatsApp Channel](#)

[Click here](#)

[Join WhatsApp Group](#)

Q1

A) What do you mean by regression? Explain the concept of Lasso Regression and Ridge Regression.

Ans:

Regression –

- Regression is a method used to predict a continuous value (like price, marks, salary, etc.) based on input variables.
- It finds the relationship between independent variables (X) and a dependent variable (Y).
- Regression builds a best-fit line or curve that represents how changes in inputs affect the output.
- It helps in forecasting and understanding patterns in data, making it useful in many real-world applications.

Example

Predicting house price using features like area, number of rooms, and location.

Ridge Regression –

- Ridge Regression helps reduce overfitting in linear regression models. It does this by controlling how large the model's coefficients can become, which prevents the model from fitting noise in the training data.
- A penalty term is added to the original cost function of linear regression. This penalty discourages the coefficients from becoming too large, making the model simpler and more stable.
- The penalty used in Ridge Regression is based on the sum of squared coefficients (L2 regularization). This means the larger the coefficient, the more penalty is applied, pushing coefficients to be smaller.
- Ridge Regression cost function is:
[
$$J(w) = \frac{1}{2} \sum (y - h(y))^2 + \lambda \sum |w|^2$$

]
Here, the first part measures prediction error, and the second part is the L2 penalty term.

- As the penalty strength increases, the coefficients shrink toward zero. This reduces the model's variance and helps it generalize better to unseen data.
- Main drawback: Ridge Regression never completely removes any feature. The coefficients shrink but never become exactly zero, so the model keeps all features even if some are less useful.

Lasso Regression :

- Lasso regression (L1 regularization) is used to stop overfitting in linear regression models by adding a penalty term to the cost function.
- Unlike Ridge regression (which uses squared coefficients), Lasso uses the sum of the absolute values of the coefficients as the penalty.
- Lasso regression tries to minimize the following cost function:

$$J(w) = \frac{1}{2} \sum (y - h(y))^2 + \sum |w|$$

where:

 1. y = actual value
 2. $h(y)$ = predicted value
 3. w = feature coefficient
- Lasso regression can reduce some coefficients to zero, effectively removing unnecessary features.
- This behaviour makes Lasso useful for feature selection, especially when working with high-dimensional datasets.
- Lasso is helpful when many features may be unimportant or redundant, as it automatically eliminates them.
- The final model becomes less complex and easier to understand because only important features remain.
- By reducing overfitting, Lasso often improves predictive performance of the model.

B) Describe the Bias–Variance trade-off and its relationship to overfitting and underfitting.

Ans:

What is Bias?

- Bias refers to the error introduced when a model makes overly simple assumptions about the data.
- A model with high bias does not learn the underlying pattern well.
- It tends to produce predictions that are consistently wrong, no matter how much data it is trained on.

Example:

Using a simple linear model to fit a complex non-linear dataset.

What is Variance?

- Variance refers to the model's sensitivity to small variations in the training data.
- A model with high variance learns noise along with the actual pattern.
- It performs very well on training data but poorly on unseen data because it fails to generalize.

Example:

A highly complex model with too many parameters memorizing the training data.

Bias–Variance Trade-off (Core Concept)

- The Bias–Variance trade-off is about finding the right balance between a model being too simple (high bias) and too complex (high variance).
- The goal is to build a model that:
 - learns the true patterns in the data (low bias)
 - does not overreact to noise (low variance)
- Achieving this balance gives the lowest total prediction error, resulting in good generalization.

Mathematically:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

You cannot reduce both bias and variance at the same time—reducing one usually increases the other.

This is the trade-off.

Relationship to Underfitting and Overfitting

A) Underfitting (High Bias, Low Variance)

- Model is too simple.
- It cannot capture the pattern of the data.
- Performs poorly on training and testing data.
- High error because the model has failed to learn.

Causes of Underfitting:

- Not enough features
- Oversimplified model
- Too much regularization
- Linear model used for non-linear data

Example:

Predicting temperature with a straight-line model when the relationship is curved.

B) Overfitting (Low Bias, High Variance)

- Model is too complex.
- Learns the training data perfectly—including noise and random fluctuations.
- Very low training error, but high testing error.

Causes of Overfitting:

- Too many features
- Complex models (deep trees, higher-degree polynomials)
- Little or no regularization
- Small training dataset

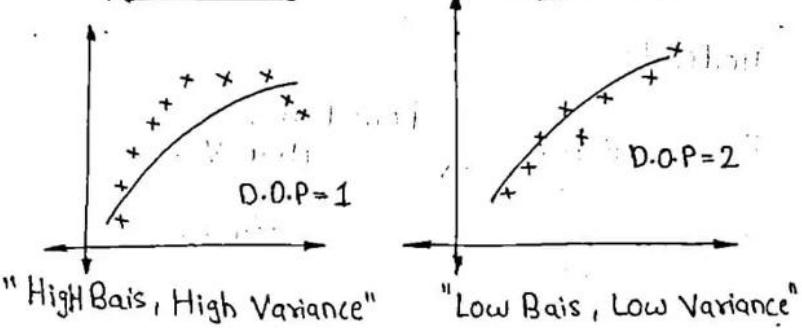
Example:

A decision tree that splits until every training sample is perfectly classified.

Bias and Variance

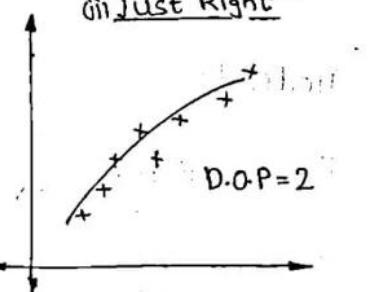
□ Regression:

(i) Underfitting



"High Bias, High Variance"

(ii) Just Right



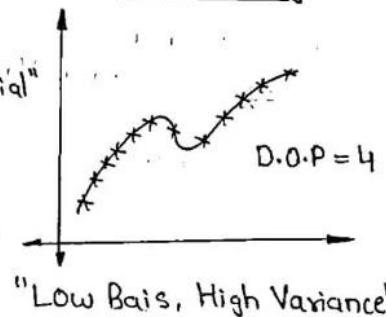
"Low Bias, Low Variance"

(iii) Over-fitting

Note:

D.O.P =

"Degree of polynomial"



"Low Bias, High Variance"

(i) Model 1 (Underfitting): Train Accuracy ↓
Test Accuracy ↓

(ii) Model 3 (Overfitting): Train Accuracy ↑
Test Accuracy ↓

C) Write a short note on evaluation metrics. Explain MAE, RMSE, and R².

Evaluation Metrics :

Evaluation metrics are used to measure how well a regression model predicts the target variable. They help compare models, identify errors, and understand model performance.

1) MAE – Mean Absolute Error

- MAE measures the average absolute difference between actual values and predicted values.
- It shows how much error the model makes on average, without considering direction.
- MAE treats all errors equally (no extra penalty on large errors).
- It is simple, easy to interpret, and uses same units as the output variable.

Formula : $MAE = \frac{1}{n} \sum |y - \hat{y}|$

Interpretation

- $MAE = 4$ means: on average, predictions are 4 units away from actual values.
- Lower MAE = better performance.

2) RMSE – Root Mean Squared Error

- RMSE is the square root of the average squared errors.
- Unlike MAE, RMSE gives higher penalty to larger errors because errors are squared.
- It is useful when large prediction errors are unacceptable.
- RMSE is very sensitive to outliers, making it good for detecting large mistakes.

Formula : $RMSE = \sqrt{\frac{1}{n} \sum (y - \hat{y})^2}$

Interpretation

- $RMSE = 6$ means: predictions deviate by 6 units on average, with big errors greatly affecting the metric.
- Lower RMSE = better model.
- RMSE is usually greater than or equal to MAE.

3) R² – Coefficient of Determination

- R² measures how much of the variance in the target variable is explained by the model.
- It checks the model's overall goodness of fit.
- R² ranges from 0 to 1:
 1. 0 → Model explains nothing
 2. 1 → Model explains everything
- R² can also be negative if the model is worse than a horizontal line (mean predictor).

Formula : $R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$

Interpretation

- $R^2 = 0.90$ means the model explains 90% of the variation in the data.
- Higher R^2 = better model.
- R^2 alone is not enough; should be checked along with MAE/RMSE.

Short Example :

Suppose actual values:

[10, 12, 13]

Predicted values:

[9, 11, 15]

1. Errors: 1, 1, 2
2. MAE: $(1 + 1 + 2)/3 = 1.33$
3. RMSE: $\sqrt{(1^2 + 1^2 + 2^2)/3} = \sqrt{6/3} = 1.41$
4. R²: Higher value = better explanation of variance.

Q2)

A) What is linear regression? Explain least squares in the context of Linear Regression.

What is Linear Regression?

- Linear Regression is a statistical method used to **predict a continuous target variable (Y)** using one or more **input features (X)**.
 - It assumes a **linear relationship** between input variables and the output.
 - It fits a **straight line (for simple regression)** or a **hyperplane (for multiple regression)** through the data.
 - The goal is to find the **best-fit line** that minimizes errors between predicted and actual values.
-

Equation of Linear Regression

For one feature (Simple Linear Regression):

$$y = m x + c$$

Where:

- **m** = slope
- **c** = intercept
- **x** = input variable
- **y** = predicted output

For Multiple Linear Regression: $y = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$

Purpose of Linear Regression

1. **Prediction** (e.g., salary, price, marks).
 2. **Trend analysis** – understanding how variables are related.
 3. **Forecasting** – based on past data.
 4. **Understanding feature influence** on the output.
-

Least Squares in Linear Regression :

Definition

The **Least Squares Method** is a mathematical technique used to find the **best-fitting line** by minimizing the **sum of squared errors** between predicted and actual values.

What It Does

- Compares actual value (y) and predicted value (\hat{y}).
 - Computes the **error** = $(y - \hat{y})$.
 - Squares the error so that:
 1. Negative errors do not cancel positive ones.
 2. Larger errors receive a bigger penalty.
 - Adds all squared errors and finds the line that **minimizes this total**.
-

Least Squares Formula (Cost Function)

$$J(w) = \frac{1}{2} \sum (y - \hat{y})^2$$

Where:

- y = actual value
 - \hat{y} = predicted value
 - w = model parameters (slope, intercept, coefficients)
-

How Least Squares Finds the Best Fit Line

1. Start with a random line.
2. Calculate total squared error.
3. Adjust slope (m) and intercept (c).
4. Keep adjusting until total error is minimized.
5. Final line is the **best fit line**.

Example

Suppose you want to predict house price based on area.

For each house:

- Actual price = 500k
- Predicted price = 480k
- Error = $500 - 480 = 20$ k
- Squared error = $20^2 = 400$

Least squares will do this for all houses and choose the line that **makes the sum of squared errors minimum**.

B) What is Gradient Descent? Compare Batch Gradient Descent and Stochastic Gradient Descent.

1) What is Gradient Descent?

Gradient Descent is an **optimization algorithm** used to minimize a **cost function** in machine learning models, especially Linear Regression, Logistic Regression, and Neural Networks.

How it Works

- Gradient Descent starts with **initial random parameter values** (like weights).
- It calculates the **cost/error** of the model using the cost function.
- It computes the **gradient** (slope) of this cost function with respect to the parameters.
- Parameters are updated by **moving in the opposite direction of the gradient** because that direction **reduces the error**.
- This process repeats until the model reaches the **minimum error** or convergence.

Parameter Update Formula

$$w := w - \alpha \cdot \frac{\partial J(w)}{\partial w}$$

Where:

- (w) = model parameter
- (α) = learning rate
- ($J(w)$) = cost function

2) Types of Gradient Descent

There are mainly two important types:

1. **Batch Gradient Descent (BGD)**
2. **Stochastic Gradient Descent (SGD)**

Aspect	Batch Gradient Descent (BGD)	Stochastic Gradient Descent (SGD)
Meaning	Uses the entire dataset to compute gradient for each update.	Uses only one training example to update parameters.
Parameter Updates	Updates parameters once per epoch (after full dataset).	Updates parameters after every sample (many updates per epoch).
Speed	Slow because it processes all data before updating.	Very fast, updates immediately after each data point.
Gradient Quality	Stable, accurate gradient.	Noisy, fluctuating gradient.
Convergence Behavior	Smooth and stable convergence path.	Zig-zag convergence but often faster.
Risk of Local Minima	Higher chance of getting stuck.	Lower chance because noise helps escape.
Memory Requirement	High — requires full dataset in memory.	Very low — uses one sample at a time.
Best For	Small/medium datasets; high accuracy needed.	Large datasets; deep learning; online learning.
Error Surface Movement	Moves steadily in the true gradient direction.	Jumps around due to noise but heads toward minimum.
Training Time	Long training time.	Shorter training time.
Computational Cost per Update	High.	Low.
Risk of Overshooting	Low.	Higher (due to noisy updates).

Q3

A) What is Multi-class Classification? Explain its variants.

1) What is Multi-class Classification?

Multi-class classification is a type of machine learning problem where the goal is to **classify an input into one of three or more possible classes**.

Key Points

1. It deals with **more than two categories** (e.g., classifying fruits into apple, banana, mango).
2. The model predicts **only one class at a time** from multiple options.
3. Algorithms like Logistic Regression, SVM, Decision Trees, Random Forest, and Neural Networks can be adapted for multi-class tasks.
4. Commonly used when the output variable is **categorical with more than two labels**.

Examples

- Classifying handwritten digits (0–9) → 10 classes
- Classifying animals into cat, dog, horse, cow → 4 classes
- Identifying types of flowers → multiple classes

Variants of Multi-class Classification

A) One-vs-Rest (OvR) / One-vs-All (OvA)

How it works

1. For **K classes**, build **K separate binary classifiers**.
2. Each classifier predicts whether a sample belongs to **that class or not**.
3. The classifier with the **highest output score** determines the final class.

Example

If classes = {A, B, C}, create:

- Classifier 1: A vs not-A
- Classifier 2: B vs not-B
- Classifier 3: C vs not-C

Advantages

- Simple and easy to implement
- Works well with algorithms like Logistic Regression, SVM

Disadvantages

- Can struggle when classes are imbalanced
 - Predictions may overlap or conflict
-

B) One-vs-One (OvO)

How it works

1. For **K classes**, build classifiers for **every pair of classes**.
2. Total models = **$K(K - 1) / 2$** .
3. Each classifier predicts between **two classes only**.
4. Final class is chosen by **majority voting**.

Example

If classes = {A, B, C}, build:

- Model 1: A vs B
- Model 2: A vs C
- Model 3: B vs C

Advantages

- Small, simple classifiers
- Works well for SVMs

Disadvantages

- Large number of models for big K
- Slower prediction time

C) Softmax (Multinomial Logistic Regression)

How it works

1. Uses a **single model** that outputs probabilities for all classes.
2. Uses the **Softmax function** to convert raw scores into probabilities.
3. The class with the **highest probability** is selected.

Formula

$$P(y = k | x) = \frac{e^{z_k}}{\sum_{i=1}^K e^{z_i}}$$

Advantages

- Single model handles all classes
- Produces **probability distribution** across classes
- Standard method in **neural networks** and deep learning

Disadvantages

- Requires more computation
- Needs large data to perform well

B) Explain K-NN algorithm. What are its advantages and disadvantages?

K-NN Algorithm

1. Definition

- K-NN (K-Nearest Neighbors) is a **supervised learning algorithm** used for **classification** and **regression**.
- It is called “lazy learner” because it **does not build a model** during training; it simply **stores the data**.

How K-NN Works

1. Choose the value of K

- K = number of nearest neighbors you want to consider.

2. Calculate distance between the new data point and all existing data points.

- Common distances: Euclidean, Manhattan, Minkowski.

3. Sort distances in increasing order.

4. Select K nearest neighbors (the top K smallest distances).

5. Perform Voting (for Classification)

- The class that appears most among the K neighbors becomes the decision.

6. Take Average (for Regression)

- The prediction is the mean/average value of the K closest neighbors.

7. Return Output

- K-NN gives the final class/label/value.

Example :

- Suppose K = 3
- Three nearest neighbors → A, A, B
- Majority is class A → prediction = A

K-NN Advantages :

1. **Very simple and intuitive** to understand.
 2. **No training time** → fast training because it stores data only.
 3. **Works well with small datasets.**
 4. **Good for non-linear datasets**, because it makes no assumption about data distribution.
 5. **Naturally handles multi-class classification.**
 6. **Can adapt based on K value** (flexible).
-

K-NN Disadvantages :

1. **Slow at prediction time**
 - Because distance must be computed for all data points.
2. **Memory-heavy (High storage)**
 - Needs to store entire dataset.
3. **Sensitive to irrelevant or unscaled features**
 - Requires **feature normalization** (e.g., Min-Max, Standardization).
4. **Poor performance in high-dimensional data**
 - Suffers from **curse of dimensionality**.
5. **K value selection is difficult**
 - Small K → noisy
 - High K → too generalized
6. **Computationally expensive** for large datasets.

C) What are different distance metrics used in K-NN? Explain any three.

K-NN commonly uses the following distance measures:

1. Euclidean Distance
2. Manhattan Distance
3. Minkowski Distance
4. Hamming Distance
5. Chebyshev Distance
6. Cosine Similarity Distance
7. Mahalanobis Distance

1 Euclidean Distance :

Definition

- Measures the **straight-line** or **shortest** distance between two data points.
- It is the **default metric** for most K-NN implementations.

Formula

$$d = \sqrt{\sum (x_i - y_i)^2}$$

- It treats all features equally and assumes all features contribute proportionally.
- Very effective when feature scales are similar.
- **Sensitive to large differences:** big values impact the distance more.
- Easily influenced by **outliers**, because squaring amplifies large deviations.
- Works well when data is in **low-dimensional space**.
- Mirrors actual geometric (physical) distance.
- Often used in:
 - Image processing

- Face recognition
 - Handwriting recognition
 - Clustering algorithms like K-Means
-

2 Manhattan Distance (L1 Distance)

Definition

- Measures the distance by summing the **absolute** differences between coordinates.
- Called **City Block Distance** because movement is like city roads (up/down/left/right), not diagonal.

Formula

$$d = \sum |x_i - y_i|$$

Detailed Points

- Less sensitive to outliers because it **does not square** differences.
- Better for datasets with **high dimensionality** (100+ features).
- Works well when features are **sparse** (most values are zero), such as:
 - Text data
 - Document similarity
 - Bag-of-words models
- More robust when data has **extreme values** or noise.
- Produces simpler decision boundaries.
- Often used in:
 - NLP
 - Recommendation systems
 - Anomaly detection

3 Minkowski Distance — (Generalized Distance Metric)

Definition

- A **general form** of Euclidean and Manhattan distances.
- Uses a parameter **p** to adjust the type of distance.

Formula

$$d = \left(\sum |x_i - y_i|^p \right)^{1/p}$$

Special Cases

- If **p = 1** → **Manhattan Distance**
- If **p = 2** → **Euclidean Distance**

Detailed Points

- Very flexible: you can choose different **p values** to control the sensitivity.
- For **p > 2**, the distance becomes even more sensitive to large differences.
- As **p increases**, the metric gives more weight to **large deviations** in features.
- Helps tune K-NN for **best accuracy** with hyperparameter optimization.
- Works when the relationship between features is not clearly known.
- Used in:
 - Machine learning model tuning
 - K-NN optimization
 - Search and recommendation tasks
- Ideal for datasets where you want to experiment with multiple distance behaviors

Q 4)

A) Explain kernel methods suitable for SVM with examples.

- A kernel is a mathematical function used in SVM to transform data from low-dimensional space to high-dimensional space.
- It helps SVM classify data that is not linearly separable.
- Kernels allow SVM to create complex decision boundaries without explicitly converting data to high dimensions (this is called the kernel trick).

1 What is a Linear Kernel?

- A **Linear Kernel** is the simplest kernel used in Support Vector Machines (SVM).
- It does **not** transform data into higher dimensions.
- Instead, it uses the **dot product** of two feature vectors to measure similarity.

✓ Formula:

$$K(x, y) = x^T y$$

This means:

Similarity = multiplication of features of x and y.

When is a Linear Kernel Used?

The Linear Kernel is used when the data is:

- **Linearly separable** (can be separated with a straight line).
- **High-dimensional** (many features, e.g., text data).
- Not very complex or curved.

2 Polynomial Kernel

- The Polynomial Kernel allows SVM to create **curved decision boundaries**.
- It maps input data into a **polynomial feature space**.
- This helps classify data where the relationship between features is non-linear but follows a polynomial pattern.

✓ Formula:

$$K(x, y) = (x^T y + c)^d$$

Where:

- **d** = degree of polynomial
- **c** = constant (controls flexibility)

When is Polynomial Kernel Used?

- When the data shows **non-linear patterns**.
- When the model requires **curved boundaries**.
- When input features interact (e.g., $x_1 \times x_2$).

3 RBF Kernel

- The **most widely used kernel** in SVM.
- Maps data into an **infinite-dimensional space**.
- Perfect for datasets with complex, irregular boundaries.

✓ Formula:

$$K(x, y) = e^{-\gamma \|x - y\|^2}$$

Where:

- **γ (gamma)** controls how tightly the curve fits.

When is RBF Kernel Used?

- When data is **highly non-linear**.

- When patterns are not clear or structured.
- When we don't know which kernel to choose → **RBF is default.**

Real-Life Examples

- Face recognition
- Medical diagnosis
- Fraud detection
- Weather pattern classification

4 Sigmoid Kernel?

- The Sigmoid Kernel behaves like a **two-layer neural network activation function.**
- Not commonly used in SVMs but useful for some patterns.

Formula:

$$K(x, y) = \tanh(\alpha x^T y + c)$$

When is Sigmoid Kernel Used?

- When data behaves like neuron activations.
- When dataset has patterns similar to neural networks.

How It Works (Step-by-Step)

1. Computes dot product and applies a **tanh activation.**
2. Produces non-linear separation.
3. Useful for datasets with neural-network-like structures.

B) What are different techniques used for outlier handling? Explain any two.

Different Techniques Used for Outlier Handling

1. Z-Score Method (Standard Deviation Method)
2. IQR Method (Interquartile Range)
3. Winsorization
4. Log Transformation / Scaling
5. Capping (Percentile Capping)
6. Removing Outliers Manually
7. Clustering-based Detection (K-Means, DBSCAN)
8. Model-based Detection (Isolation Forest, LOF)

1 Z-Score Method (Standard Deviation Method)

Meaning

- Z-score measures how far a data point is from the mean in terms of standard deviations.
- Helps identify points that are unusually high or low.

✓ Formula:

$$Z = \frac{(x - \mu)}{\sigma}$$

Where:

- x = data point
- μ = mean
- σ = standard deviation

How It Works

1. Calculate mean and standard deviation of the data.
2. Compute the Z-score for each value.
3. If $Z\text{-score} > +3$ or $< -3 \rightarrow$ treat as an outlier.
4. Handle them by removal, capping, or replacing with mean/median.

Advantages

- Easy to calculate.
- Works well for normally distributed data.

Disadvantages

- Doesn't work well when data is not normally distributed.

- Sensitive to extreme values.

2 IQR Method (Interquartile Range)

Meaning

- IQR = difference between 75th percentile (Q3) and 25th percentile (Q1).
- Detects outliers based on spread of middle 50% of data.

Formula: $IQR = Q3 - Q1$

Outlier thresholds:

- Lower bound = $Q1 - 1.5 \times IQR$
- Upper bound = $Q3 + 1.5 \times IQR$

Any value outside this range = outlier.

How It Works

1. Sort the data.
2. Find Q1 and Q3.
3. Compute IQR.
4. Find upper & lower limits.
5. Outliers are those falling outside these limits.

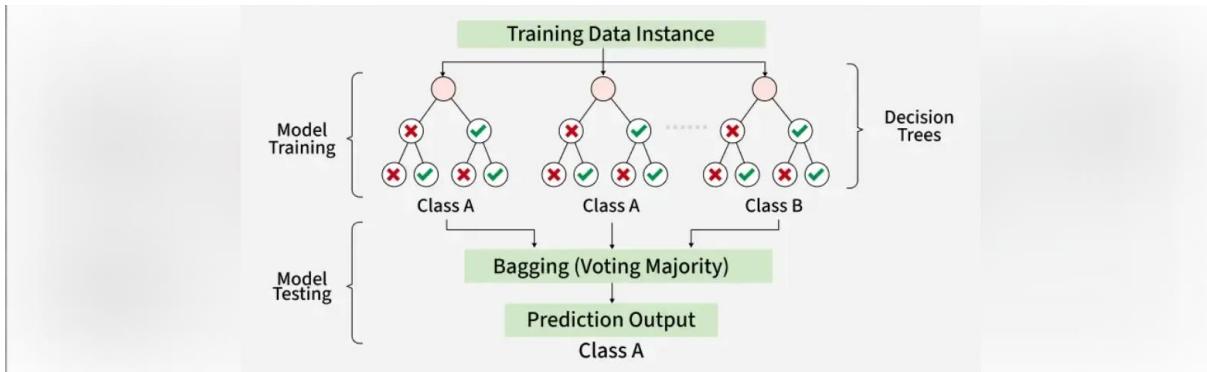
Advantages

- Works very well for skewed and non-normal data.
- Not affected by extreme values.

Disadvantages

- Not suitable for very small datasets.

C) With a suitable diagram, explain the Random Forest algorithm and the concept of bagging.



Random Forest Algorithm

The Random Forest algorithm extends bagging by introducing an additional layer of randomness, specifically in the feature selection process during the construction of each decision tree.

- **Bagging:**

Multiple bootstrap samples are drawn from the original training data, as described in the bagging concept.

- **Random Feature Subsets:**

For each individual decision tree, at each split point during its construction, only a random subset of features is considered for finding the best split, rather than all available features. This further decorrelates the individual trees, making the ensemble more robust.

- **Individual Tree Training:**

A full decision tree is grown on each bootstrapped sample, using the random feature subset selection at each split.

- **Aggregation:**

For a new data point, each tree in the forest makes a prediction. For classification, the final prediction is determined by a majority vote among the trees. For regression, the final prediction is the average of the predictions from all the trees.

Bagging (Bootstrap Aggregating)

Bagging is an ensemble meta-algorithm that improves the stability and accuracy of machine learning algorithms, reducing variance and helping to avoid overfitting. It involves two main steps:

- Bootstrapping:

From the original training dataset, multiple new datasets are created by sampling with replacement. Each new dataset has the same size as the original, but due to sampling with replacement, some original data points may appear multiple times in a bootstrapped sample, while others may not appear at all.

- Aggregating:

A base learner (e.g., a decision tree) is trained independently on each of these bootstrapped datasets. For a new data point, the predictions from all the individual models are combined (e.g., averaged for regression, or majority vote for classification) to produce a final, more robust prediction.

Q5)

A) Why is K-Medoid used? Explain the K-Medoid algorithm.

K-Medoid is used because it is a more robust clustering algorithm compared to K-Means, especially when the dataset contains noise, outliers, or non-Euclidean distances.

Reasons to use K-Medoid :

1. Robust to outliers
 - o K-Means uses the mean, which can be pulled by extreme values.
 - o K-Medoid uses medoids (actual data points), so outliers have less impact.
2. Works with any distance metric
 - o K-Means works mainly with Euclidean distance.
 - o K-Medoid can use Manhattan, cosine, Hamming, Jaccard, etc.
3. Better for categorical datasets
 - o Since medoids are real data points, mixed or categorical data works better.
4. More stable clusters
 - o Cluster centers do not move drastically between iterations.
5. Useful in applications where interpretability matters
 - o Because the medoid is an actual sample, it is easy to interpret.

K-Medoid Algorithm (PAM: Partitioning Around Medoids)

The most famous version is PAM – Partitioning Around Medoids.

Step-by-Step Explanation :

1. Choose K initial medoids

- Randomly select K actual data points as medoids.

2. Assign each data point to its nearest medoid

- Based on chosen distance measure
(e.g., Manhattan, Euclidean, Cosine, etc.)

3. Update medoids (swap step)

For each cluster:

- Try swapping the medoid with a non-medoid point.
- Compute the total cost (sum of distances).
- If the cost reduces → accept the swap (new medoid).

4. Repeat

- Continue assigning points + swapping until no improvement in total cost.

5. Final clusters formed

- When no more swaps reduce cost.

B) Why is density-based clustering used? Explain any one method (DBSCAN / OPTICS / DENCLUE)

Density-based clustering is used because it can identify arbitrary-shaped clusters and outliers, doesn't require the number of clusters to be specified in advance, and works well with noisy data

Why density-based clustering is used

- Arbitrary-shaped clusters: It can find clusters of any shape, not just the spherical shapes often found by other methods like k-means.
- Outlier detection: It can distinguish between dense clusters and sparse areas, automatically identifying and removing noise and outliers.
- No need to pre-specify cluster count: Unlike some algorithms, you don't need to know the number of clusters beforehand.
- Handles noise: It is robust to noise in the data, as points in low-density regions are simply marked as noise

DSBSCAN :

DBSCAN is a density-based clustering algorithm that groups data points that are closely packed together and marks outliers as noise based on their density in the feature space. It identifies clusters as dense regions in the data space separated by areas of lower density.

Key Parameters in DBSCAN

1. eps: This defines the radius of the neighborhood around a data point. If the distance between two points is less than or equal to **eps** they are considered neighbors. A common method to determine **eps** is by analyzing the k-distance graph. Choosing the right **eps** is important:

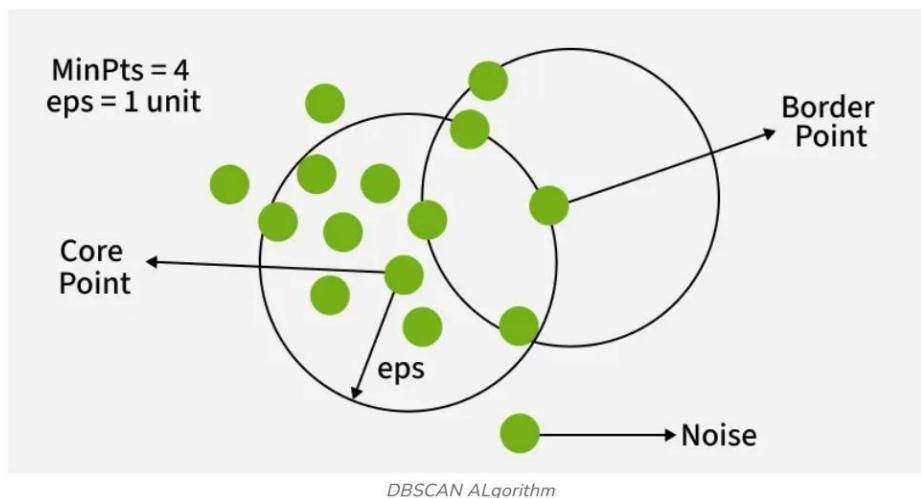
- If **eps** is too small most points will be classified as noise.
- If **eps** is too large clusters may merge and the algorithm may fail to distinguish between them.

2. MinPts: This is the minimum number of points required within the **eps** radius to form a dense region. A general rule of thumb is to set **MinPts** $\geq D+1$ where **D** is the number of dimensions in the dataset.

How Does DBSCAN Work?

DBSCAN works by categorizing data points into three types:

1. Core points which have a sufficient number of neighbors within a specified radius (epsilon)
2. Border points which are near core points but lack enough neighbors to be core points themselves
3. Noise points which do not belong to any cluster.



(Optional)

Steps in the DBSCAN Algorithm

1. **Identify Core Points:** For each point in the dataset count the number of points within its eps neighborhood. If the count meets or exceeds MinPts mark the point as a core point.
2. **Form Clusters:** For each core point that is not already assigned to a cluster create a new cluster. Recursively find all density-connected points i.e points within the eps radius of the core point and add them to the cluster.
3. **Density Connectivity:** Two points a and b are density-connected if there exists a chain of points where each point is within the eps radius of the next and at least one point in the chain is a core point. This chaining process ensures that all points in a cluster are connected through a series of dense regions.
4. **Label Noise Points:** After processing all points any point that does not belong to a cluster is labeled as noise.

C) Apply the K-Means algorithm to cluster the given points into three clusters.

Q6)

A) Explain Hierarchical Clustering with an example. Differentiate Agglomerative vs Divisive.

Answer :

Hierarchical Clustering is an unsupervised learning method used to group similar data points into clusters based on their distance or similarity. Instead of choosing the number of clusters in advance, it builds a tree-like structure called a dendrogram that shows how clusters merge or split at different levels. It helps identify natural groupings in data and is commonly used in pattern recognition, customer segmentation, gene analysis and image grouping

Imagine we have four fruits with different weights: an apple (100g), a banana (120g), a cherry (50g) and a grape (30g). Hierarchical clustering starts by treating each fruit as its own group.

- Start with each fruit as its own cluster.
- Merge the closest items: grape (30g) and cherry (50g) are grouped first.
- Next, apple (100g) and banana (120g) are grouped.
- Finally, these two clusters merge into one.

Finally all the fruits are merged into one large group, showing how hierarchical clustering progressively combines the most similar data points

Types of Hierarchical Clustering

1. Agglomerative Clustering
2. Divisive clustering

Parameters	Agglomerative Clustering	Divisive Clustering
Approach	Bottom-up: Starts with individual points and merges them.	Top-down: Starts with all data in one cluster and splits.
Complexity Level	More computationally expensive due to pairwise distance calculations.	Less computationally expensive but requires careful cluster splitting.
Handling Outliers	Better at handling outliers, as outliers can be absorbed into larger clusters.	Outliers may lead to inefficient splitting and suboptimal results.
Interpretability	More interpretable due to clear cluster merging shown in the dendrogram.	Harder to interpret due to recursive splitting decisions.
Implementation	Scikit-learn provides multiple linkage methods (Ward, Complete, Average, Single).	Not widely implemented in Scikit-learn or SciPy.
Example Applications	Image segmentation, customer segmentation, document clustering, gene expression analysis.	Less common; used in hierarchical data analysis or domain-specific studies.

C) What is Outlier Analysis? Explain methods used for outlier detection in clustering.

Outlier Analysis (also called **anomaly detection**) refers to the process of identifying data points that are **significantly different** from the majority of the data.

These points deviate from normal patterns and may indicate:

- Errors in data
- Rare events
- Fraudulent activities
- Unusual behavior
- Noise in the dataset

In clustering, outliers are the points that **do not belong strongly to any cluster** because they lie **far from dense regions**.

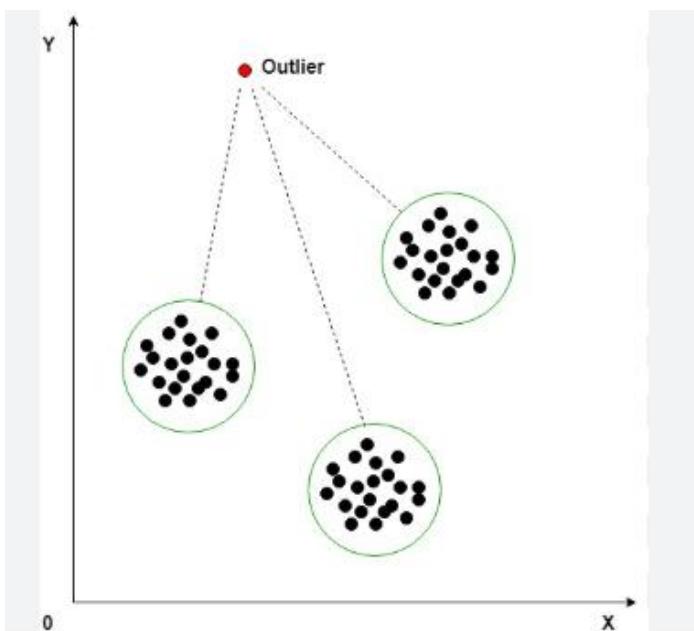


Fig : Outlier

Methods used for outlier detection in clustering.

1. Density-Based Methods (DBSCAN)
2. Distance-Based Outlier Detection (K-Means)
3. LOF (Local Outlier Factor)
4. Hierarchical Clustering-Based Outlier Detection

1. Density-Based Method (DBSCAN)

Detailed Points:

1. DBSCAN groups points based on density
 - o A region is considered a cluster if it has enough points close to each other.
 - o This helps separate dense clusters from sparse/noisy areas.
2. Uses two parameters: Eps and MinPts
 - o Eps: How far we look around a point (radius).
 - o MinPts: Minimum number of points needed to form a dense region.
3. Outlier Identification
 - o If a point does not have enough neighbors within Eps, it cannot form or join a cluster.
 - o Such isolated points are labeled noise, which are considered outliers.
4. Why it works well
 - o Outliers naturally lie in low-density areas.
 - o DBSCAN automatically separates them without extra logic.

2. Distance-Based Method (K-Means)

Detailed Points:

1. K-Means forms clusters around centroids
 - o After clustering, each point is assigned to the nearest centroid.
2. Outlier Detection Using Distance
 - o Calculate how far each point is from its cluster centroid.
 - o Very large distance = unusual point = outlier.
3. Threshold Criteria
 - o Points in the top 5% or 10% farthest distances are labeled as outliers.
 - o Or: If distance $>$ (mean distance + 2 \times standard deviation).
4. Why it helps find outliers
 - o Normal points stay close to centroid.
 - o Outliers remain far away because they don't fit well with any cluster.

3. LOF (Local Outlier Factor)

Detailed Points:

1. LOF compares a point's density with the density of its neighbors
 - o Normal points: Similar density as neighbors.
 - o Outliers: Much lower density (more isolated).
2. Local Reachability Density (LRD)
 - o Measures how close a point is to its neighbors.
 - o Low LRD = point is in a sparse region → likely outlier.
3. LOF Score Calculation
 - o $LOF = \text{Average density of neighbors} \div \text{density of the point.}$
 - o If $LOF > 1$ → lower density than neighbors.
 - o If $LOF > 1.5$ or 2 → strong outlier.
4. Why LOF is useful
 - o Detects local anomalies, even when the dataset has clusters with different densities.

- More accurate for real-world uneven clusters.

“DBSCAN,K-Means has already been explained in detail in the previous answer. Kindly refer to that section for the full explanation.”

D) Differentiate K-Means and Spectral Clustering. Explain when to use Spectral Clustering.

Parameter	K-Means Clustering	Spectral Clustering
Approach	Partitioning-based algorithm	Graph-based algorithm using eigenvalues/eigenvectors
Type of Clusters	Finds spherical/convex clusters	Finds arbitrarily shaped, complex clusters
Input Requirements	Works directly on raw data	Works on similarity (affinity) matrix
Cluster Shape Limitations	Struggles with non-linear boundaries	Handles non-linear, complex manifold structures
Algorithm Dependence	Depends on centroid initialization	Depends on graph construction and eigen decomposition
Computational Cost	Low ($O(n)$) → efficient for large datasets	High ($O(n^3)$) → costly for large datasets
Sensitivity to Noise	Sensitive to outliers (centroid shifts easily)	More robust due to similarity graph structure
Number of Clusters (k)	Must be known beforehand	Also requires k, but more accurate separation
Best For	Large, simple, well-separated clusters	Complex shapes, non-linear structures, image segmentation

When to Use Spectral Clustering :

Use Spectral Clustering when:

1. Clusters Have Irregular or Non-Convex Shapes

- Ideal for datasets where clusters are curved or connected in complex ways (e.g., spiral shapes, rings, intertwined patterns).

2. Data Cannot Be Separated with a Straight Line

- Spectral clustering uses eigenvectors to project data into a space where clusters become separable.

3. You Have a Similarity Matrix Instead of Raw Data

- Works well when you can define similarity (e.g., cosine similarity, RBF kernel).

4. K-Means Fails Due to Non-Linear Boundaries

- K-Means assumes spherical clusters, spectral clustering does not.

5. Applications Involving Graphs or Images

- Performs extremely well in:
 - Image segmentation
 - Social network community detection
 - Document/topic grouping
 - Speech separation
 - Bioinformatics

6. Datasets With Small to Medium Size

- Because eigen-decomposition is expensive, it is ideal for datasets up to a few thousand samples.

BONUS QUESTION FROM THIS UNIT :

Compare Hierarchical clustering and K-means clustering.

Parameter	Hierarchical Clustering	K-Means Clustering
Approach	Builds a hierarchy of clusters (top-down or bottom-up).	Partitions data into k clusters based on centroids.
Cluster Structure	Forms a tree-like structure (dendrogram) .	Forms flat, non-hierarchical clusters.
Number of Clusters (k)	Not required initially; chosen later by cutting the dendrogram.	Must specify k before running the algorithm.
Cluster Shape	Can detect arbitrary-shaped and nested clusters.	Works best for spherical/convex clusters.
Computation Cost	More expensive ($O(n^2)$ or higher). Not suitable for very large datasets.	Fast and efficient ($O(n)$), good for large datasets.
Memory Usage	High, since distance matrix must be stored.	Low memory requirement.
Sensitivity to Noise	More robust; small linkages can separate outliers.	Sensitive to outliers; centroid shifts easily.

Q 7)

A) Explain Radial Basis Function networks and its building blocks

Radial Basis Function Network (RBFN) is a type of **artificial neural network** used mainly for **function approximation, regression, classification, and time-series prediction**.

It is similar to a feed-forward network but has a special hidden layer that uses **radial basis functions** (mostly Gaussian functions).

RBF Networks work in **three layers**:

- 1. Input Layer**
- 2. Hidden Layer (with RBF neurons)**
- 3. Output Layer (linear neurons)**

Why is it called “Radial Basis”?

Because the activation of each hidden neuron depends on the **distance** between the input vector and the **center** of that hidden neuron.

Example:

If the input is close to the center → high activation

If the input is far from the center → low activation

This creates **localized responses**, unlike sigmoid networks that respond globally.

Building Blocks of RBF Networks

1. Input Layer

- Passes input features directly to the hidden layer.
 - No computation is done here.
 - If input has n *features*, input layer has n *nodes*.
-

2. Hidden Layer (Core of RBF Network)

This layer uses **Radial Basis Functions**—normally **Gaussian functions**.

Each hidden neuron has:

- 1. Center (C_i)**

- A reference point representing where the neuron should respond the most.
- Centers are usually chosen using **K-means clustering**.

2. Spread / Width (σ_i)

- Controls how wide the Gaussian curve is.
- Larger $\sigma \rightarrow$ broader response
- Smaller $\sigma \rightarrow$ localized response

3. Activation Function

The most used RBF function is the **Gaussian Function**:

$$\phi_i(x) = e^{-\frac{\|x - C_i\|^2}{2\sigma_i^2}}$$

Where:

- x = input vector
- C_i = center of the i-th neuron
- σ_i = width of the Gaussian
- $\phi_i(x)$ = output of RBF neuron

Interpretation

- If x is close to $C_i \rightarrow$ activation ≈ 1
- If far \rightarrow activation ≈ 0

This makes RBF models **fast** and **highly interpretable**.

3. Output Layer

- Takes weighted sum of hidden layer outputs.
- Activation is **linear**.

$$y = \sum_{i=1}^m w_i \phi_i(x)$$

Output Layer Characteristics:

- **Simple linear combination**

- Training is fast (uses least squares)
- Good for both **classification** and **regression**

B) Explain Recurrent Neural Networks and its applications in brief

A **Recurrent Neural Network (RNN)** is a type of neural network designed to process **sequential data** such as text, speech, time-series, video frames, or sensor readings.

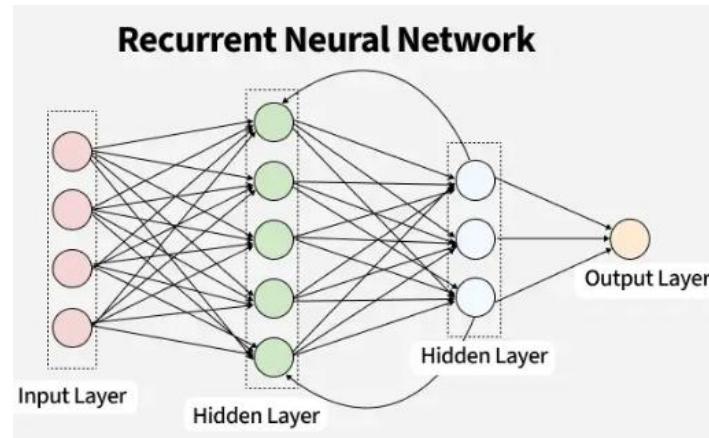
Unlike feedforward networks, an RNN has **feedback connections**, meaning:

The output of the previous step becomes an input for the next step.

This creates a **memory of past inputs**, allowing the network to learn patterns over time.

How RNNs work

- **Sequential processing:** RNNs process data one step at a time. At each step, they take an input and an activation from the previous step to produce an output and an updated "hidden state".
- **Memory:** The hidden state acts as the network's memory, carrying information from past inputs to help inform current predictions. This "memory" allows the network to understand context and dependencies in a sequence, such as the relationship between words in a sentence.
- **Looping mechanism:** The "recurrent" nature comes from a loop within the network, where the output of one step is fed back as an input to the next, creating a cycle that allows information to persist over time.



Applications of RNNs

- **Natural Language Processing (NLP):**
 - **Machine Translation:** Translating text from one language to another.
 - **Language Modeling:** Predicting the next word in a sentence for tasks like text generation and virtual assistants.
 - **Sentiment Analysis:** Determining the emotional tone of a piece of text.
- **Speech:**
 - **Speech Recognition:** Converting spoken language into written text.
 - **Text-to-Speech:** Generating human-like speech from text.
- **Time-Series Analysis:**
 - **Forecasting:** Predicting future values in a sequence, such as stock prices or weather patterns.
- **Other applications:**
 - **Music Generation:** Creating new sequences of musical notes.
 - **Video Analysis:** Understanding actions and events in video sequences.

C) What is multilayer perceptron? Describe with diagram.

Multi-Layer Perceptron (MLP) consists of fully connected dense layers that transform input data from one dimension to another.

It is called multi-layer because it contains an input layer, one or more hidden layers and an output layer.

The purpose of an MLP is to model complex relationships between inputs and outputs.

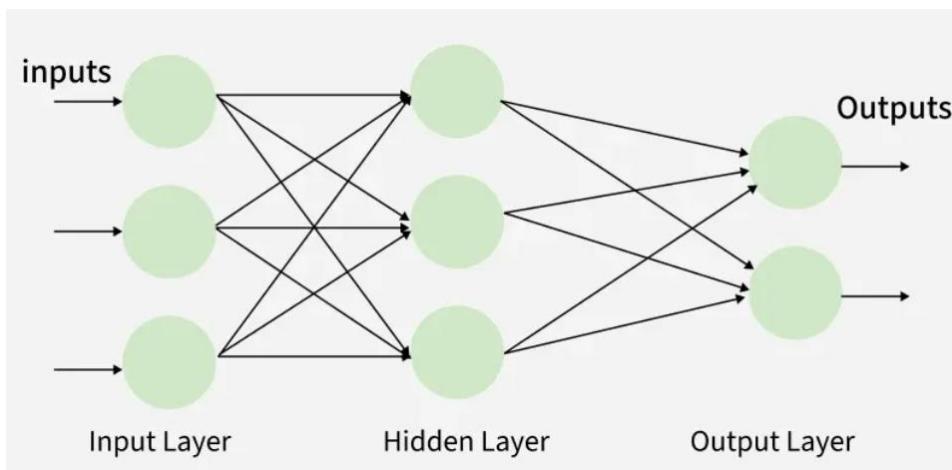


Fig: MLP

- **Input Layer:** Each neuron or node in this layer corresponds to an input feature. For instance, if you have three input features the input layer will have three neurons.
- **Hidden Layers:** MLP can have any number of hidden layers with each layer containing any number of nodes. These layers process the information received from the input layer.
- **Output Layer:** The output layer generates the final prediction or result. If there are multiple outputs, the output layer will have a corresponding number of neurons.

Every connection in the diagram is a representation of the fully connected nature of an MLP. This means that every node in one layer connects to every node in the next layer. As the data moves through the network each layer transforms it until the final output is generated in the output layer.

How It Works :

- **Forward Propagation:**

Input data flows from the input layer, through the hidden layers, to the output layer. At each neuron, a weighted sum of inputs is calculated, a bias is added, and an activation function is applied.

- **Backpropagation:**

The output of the network is compared to the true target values to calculate an error. This error is then propagated backward through the network, and the weights and biases of the connections are adjusted to minimize this error using optimization algorithms like gradient descent. This process allows the MLP to learn and improve its performance over time.

Q8)

A)draw and Explain Convolution Neural Network (CNN) with suitable example.

- **Convolutional Neural Network (CNN)** is an advanced version of **artificial neural networks (ANNs)**, primarily designed to extract features from grid-like matrix datasets.
- This is particularly useful for visual datasets such as images or videos, where data patterns play a crucial role.
- CNNs are widely used in **computer vision** applications due to their effectiveness in processing visual data.
- CNNs consist of multiple layers like the input layer, Convolutional layer, pooling layer, and fully connected layers.

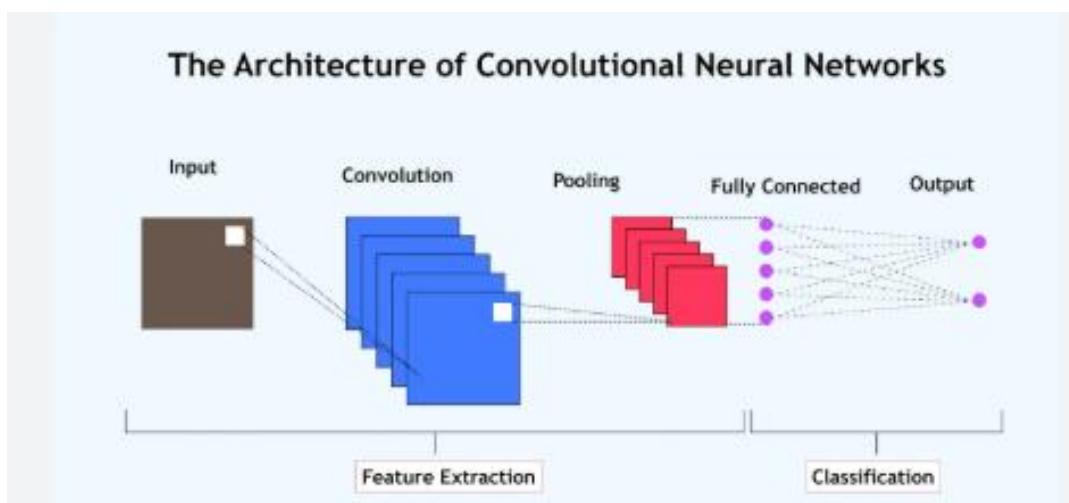


Fig :CNN Architecture

1. Input Layer

- Receives the **raw image data**.
- Input is usually a **3D tensor** → $(height \times width \times channels)$
Example: $224 \times 224 \times 3$ for an RGB image.
- No computation happens here — it only passes data to the next layer.

2. Convolutional Layer

- **Core layer of CNN** responsible for feature extraction.
- Applies **learnable filters/kernels** that slide across the image.
- Each filter performs a **dot product** with local regions of the input.
- Produces **feature maps** representing detected patterns.

- Different filters detect different features like:
 - Edges
 - Corners
 - Textures
 - Shapes
 - Example: A *vertical-edge* filter produces high values where vertical edges appear.
-

3. Activation Layer (ReLU)

- Adds **non-linearity** to the network.
- ReLU operation:

$$f(x) = \max(0, x)$$

- Negative values become **0**, while positive values remain unchanged.
 - Helps the network learn complex and non-linear patterns.
-

4. Pooling Layer (Max Pooling / Average Pooling)

- Reduces **spatial dimensions** of feature maps (height & width).
 - Lowers the number of parameters → faster & less memory.
 - Makes features **translation-invariant** (small shifts don't affect detection).
 - **Max Pooling (2×2)** picks the *maximum value* in each window.
 - Helps retain the most important/strongest features.
-

5. Fully Connected (Dense) Layers

- After convolution + pooling, feature maps are **flattened** into a 1D vector.
- Each neuron connects to all neurons from the previous layer.
- Learns **high-level relationships** between extracted features.
- Final dense layer gives **class scores**.
- Uses **softmax** activation for multi-class classification to output probabilities.

B) What are different activation function used in NN?

1) Sigmoid Activation Function

Meaning

Sigmoid converts any real number into a value between **0 and 1**.

It behaves like a **probability mapping** and is mainly used when the output must represent a probability.

Formula

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Where It Is Used

- Binary classification output layer
- Logistic regression
- Models where output must be between 0 and 1

Example

If input = 2

$$\sigma(2) \approx 0.88$$

Meaning the model predicts 88% probability.

2) Tanh (Hyperbolic Tangent)

Meaning

Tanh is similar to sigmoid but outputs values between **-1 and +1**.
It is **zero-centered**, making it suitable for hidden layers.

Formula

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Where It Is Used

- In hidden layers of simple neural networks
- In problems where negative outputs are useful

- In networks requiring symmetric activation

Example

If input = 2

$$\tanh(2) \approx 0.964$$

3) ReLU (Rectified Linear Unit)

Meaning

ReLU outputs **0 for negative values** and **x for positive values**.

This makes the network learn faster and handle deep layers efficiently.

Formula

$$ReLU(x) = \max(0, x)$$

Where It Is Used

- Hidden layers of CNNs
- Deep learning architectures
- MLPs and GANs
- Almost all modern neural networks

Example

$$x = -5 \rightarrow 0$$

$$x = 4 \rightarrow 4$$

4) Leaky ReLU

Meaning

Leaky ReLU is a modification of ReLU that allows a **small negative value** when the input is negative.

This helps keep neurons active instead of becoming stuck.

Formula

$$LeakyReLU(x) = \begin{cases} x & x > 0 \\ 0.01x & x < 0 \end{cases}$$

Where It Is Used

- CNN layers
- Deep models where small negative gradients are needed
- Situations where ReLU sometimes outputs many zeros

Example

$x = -8 \rightarrow -0.08$

$x = 8 \rightarrow 8$

5) Softmax Activation Function

Meaning

Softmax converts a vector of scores into a **probability distribution**. Each output represents the probability of belonging to a class, and all probabilities sum to **1**.

Formula

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Where It Is Used

- Multi-class classification output layer
(e.g., classify images into cat/dog/bird)
- NLP text classification
- Multi-class logistic regression

Example

If output = [2, 1, 0],
Softmax $\approx [0.67, 0.24, 0.09]$

C) Explain ANN with it's Architecture

An **Artificial Neural Network (ANN)** is a computational model inspired by the **human brain**.

It consists of interconnected units called **neurons** that work together to process information, learn patterns from data, and make predictions or decisions.

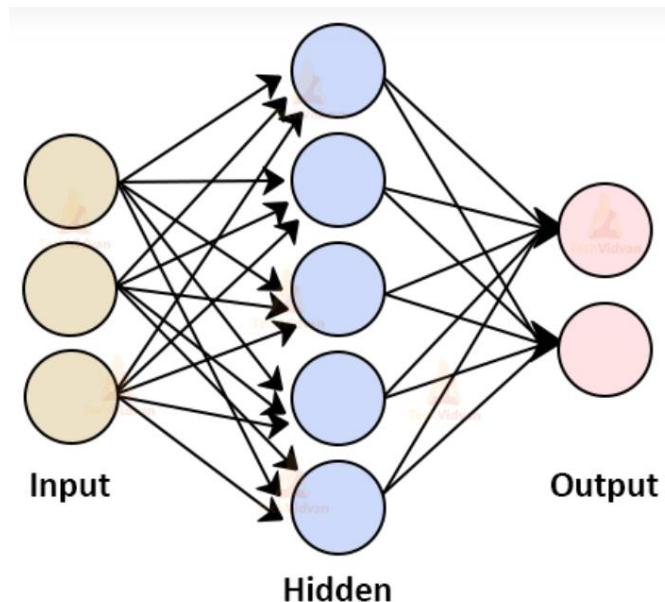


Fig: Artificial Neural Network (ANN)

1. Layers

a) Input Layer

- This is the first layer of ANN.
- It receives the raw input data (features).
- Each input neuron simply passes the information to the next layer.

b) Hidden Layer

- Performs all major computations on the input data.
- Contains neurons with **weights, biases, and activation functions**.
- Extracts patterns, relationships, and features from input.
- Can have one or multiple hidden layers depending on the complexity.

c) Output Layer

- Produces the final result of the neural network (classification, prediction, etc.).
 - Uses suitable activation functions like Softmax, Sigmoid, etc.
-

2. Components of ANN

a) Weights

b) Bias

c) Activation Function

d) Cost (Loss) Function

3. Working of ANN (Step-by-Step)

Step 1: Input Feeding

- Raw data is fed into the **input layer**.

Step 2: Weight Assignment

- Each connection from input to hidden layer is assigned **random weights** initially.

Step 3: Adding Bias

- A bias value is added to each weighted input.

Step 4: Weighted Sum Calculation

$$\text{Weighted Sum} = (w_1x_1 + w_2x_2 + \dots + b)$$

Step 5: Activation Function

- The weighted sum is passed through an activation function (ReLU, Sigmoid, etc.).
- Determines whether the neuron fires and what value it outputs.

Step 6: Forward Propagation

- The processed information moves from:
Input Layer → Hidden Layer → Output Layer
- Gives the predicted output.

Step 7: Error Calculation

- Predicted output is compared with actual output.
- Error is calculated using the loss function.

Step 8: Backpropagation

- Error is sent backward through the network.
- Weights are adjusted to reduce the error.
- Uses gradient descent or other optimization algorithms.

Step 9: Repeating for Epochs

- The process of forward propagation + backpropagation is repeated multiple times (epochs).
- Gradually improves accuracy and reduces error.

Step 10: Final Prediction

- After training, final optimized weights are used to make accurate predictions.

BONUS QUESTIONS

1. Explain in brief types of ANN based on layers.
2. Write short note on Back propagation network.
3. Write short note on importance of Activation function used in Neural Network
4. What is Personalized recommendation? What is content based recommendation

Join Community by clicking below links 

[Website Link](#)

👉 For question papers and notes visit website

[Click here](#)

[Telegram Channel](#)

[Click here](#)

[WhatsApp Channel](#)

[Click here](#)