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

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] = max profit using first i 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 ≠ 37 → profit changed → **Item 4 (weight 2, profit 15)** taken
  Remaining capacity = 5 − 2 = **3**

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

- 25 ≠ 25 → **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 → same → 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):
    - Place it in the **latest free slot** before its deadline.
    - If the slot is free, schedule the job.
    - 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 ⇒ slots = [1,2,3,4]

Step 3: Schedule

- c → deadline 1 → slot 1 free → place c
- d → deadline 1 → slot 1 filled → skip
- a → deadline 4 → slot 4 free → place a
- b → deadline 1 → slot 1 filled → 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 ⇒ slots = [1,2,3]

Step 3: Schedule

- a → deadline 2 → slot 2 free → place a
- c → deadline 2 → slot 2 filled → try slot 1 → slot 1 free → place c
- d → deadline 1 → slot 1 filled → skip
- b → deadline 1 → slot 1 filled → skip
- e → deadline 3 → slot 3 free → 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/zPtI8q9gvX8?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:

1. Object 04 → 1.00
2. Object 05 → 0.83
3. Object 02 → 0.67
4. Object 01 → 0.50
5. 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 Ai is P[i-1] × 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 Ai...Aj.
- s[i][j] = index k at which the optimal split Ai..Ak and Ak+1..Aj 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] (A1..A3):
    - k = 1 → cost = m[1][1] + m[2][3] + 352 = 0 + 40 + 30 = **70**

- k = 2 → cost = m[1][2] + m[3][3] + 342 = 60 + 0 + 24 = 84
  ⇒ **m[1][3] = 70**, s[1][3] = 1

- For m[2][4] (A2..A4):
  - k = 2 → cost = m[2][2] + m[3][4] + 544 = 0 + 32 + 80 = 112
  - k = 3 → cost = m[2][3] + m[4][4] + 524 = 40 + 0 + 40 = **80**
    ⇒ **m[2][4] = 80**, s[2][4] = 3

Length = 4

- For m[1][4] (A1..A4):
  - k = 1 → cost = m[1][1] + m[2][4] + 354 = 0 + 80 + 60 = 140
  - k = 2 → cost = m[1][2] + m[3][4] + 344 = 60 + 32 + 48 = 140
  - k = 3 → cost = m[1][3] + m[4][4] + 324 = 70 + 0 + 24 = **94**
    ⇒ **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 ⇒ split (A1..A3) | (A4) → ((A1 A2 A3) A4)
- Inside A1..A3: s[1][3] = 1 ⇒ split (A1) | (A2 A3) → (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=APq9D7pGnxNTHHZL

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 OPT[i][j] using previously computed subproblems.

Time Complexity

- Two nested loops → **O(n × m)**
- Filling each table entry takes **O(1)** time
  Therefore total complexity is:

Time Complexity = O(n × m)

Space complexity = O(n × m)

2. CONTROL ABSTRACTION FOR GREEDY METHOD

Greedy_Solver(X):

1. Initialize solution S = ∅
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 → **O(n)**
  (or **O(log n)** if using priority queue)
- repeating for all items → **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:

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

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

$$OPT[i][w] = \max\left(OPT[i-1][w],\ P_i + OPT[i-1][w - W_i]\right)$$

Explanation:

- Two choices for each item:
    1. **Exclude** item i → profit = OPT[i-1][w]
    2. **Include** item i → profit = P_i + 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/052VkKhIaQ4?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 ≤ 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)*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 = **15 kg** (fits capacity exactly).
Total profit = 10 + 10 + 18 = **38 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=_2WUmepkEszXLwO1

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=59Yyw9yZfjgvtUuv

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 ≤ current best feasible value are **pruned**.

---

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

Compute ratio (v/w):

- O_1: (6/5 = 1.20)
- O_2: (5/4 = 1.25)
- O_3: (4/3 \approx 1.33)

Sorted order (highest ratio first): **O_3, O_2, O_1**.
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 = O_3), (W=0, V=0).
Compute root bound by fractional fill in order O_3, O_2, O_1:

- Take O_3 fully: (W=3, V=4)
- Take O_2 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 O_3)**: level=1, (W=3, V=4).
  Bound: fill with O_2 then O_1: O_2 fits → (W=7,V=9). Bound = 9.
  Feasible value 4 → maxValue = 4. Insert node (bound 9).
- **Right child (exclude O_3)**: level=1, (W=0, V=0).
  Bound: take O_2 (W=4,V=5), remaining cap 3 → take fraction of O_1 (3/5 of 6 = 3.6).
  Bound = (5 + 3.6 = 8.6). Insert node (bound 8.6).

PQ now: nodes with bounds {9 (include O_3), 8.6 (exclude O_3)}.

---

2) Expand node with highest bound = include O_3 (W=3,V=4, bound=9):

- **Left child (include O_2 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 O_2)**: level=2, (W=3, V=4).
  Bound: remaining cap 4 → take fraction of O_1: (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 O_3). 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): **O_2 and O_3** (we included O_3 and O_2).

- 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; wors 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* Φ(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 ≥ 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 n 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 + ΔΦ = $1 + 1 = 2$.

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, ΔΦ = 0 ⇒ 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 + ΔΦ = $t + (-t) = 0$.

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:

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

Compute utilization:

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

If CPU max frequency = 1 GHz
Power-optimized frequency = $0.4 \times 1 = 0.4$ 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(n^2) 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 → 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**:

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

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 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 hash(T[i..i+m-1]) with hash(P)

- If hashes match → check actual substring to confirm match
- Else → no match

2. Compute next window hash **efficiently using rolling hash**:

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

---

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 hash("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 (T1)**
- 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 → more threads can help reduce execution time.

Example

Suppose a computation involves 16 tasks:

- T1 (work) = 16 units
- T∞ (span) = 4 units (longest dependent chain)

Then **parallelism** = 16 / 4 = 4 → maximum speedup ~ 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) ≈ $O(\log^2 n)$

---

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:
    - Dequeue vertex u
    - 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**:
    - Mark source vertex s as visited on its machine
    - Add s to **local frontier**
2. **Level-synchronous BFS**:
    - Each machine processes its **local frontier**
    - 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**:
    - Machines exchange newly discovered vertices at each BFS level
    - 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**:

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

- 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 hash(T[i..i+m-1]) == hash(P) → verify substring match
   - Compute next window hash using **rolling hash**:

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

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 → B, C
- B → A, D, E
- C → A, F, G
- D → B
- E → B
- F → C
- G → 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: []

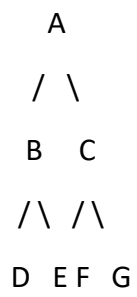
Visited: {A, B, C, D, E, F, G}

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**:
    - If the array has only 1 element → it is already sorted → return.
3. **Divide** the array into two halves:
    - Left half: A[0..mid]
    - Right half: A[mid+1..n-1]
4. **Spawn two parallel threads**:
    - Thread 1: Sort the left half.
    - 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)**:
    - For each row i of matrix C, spawn a thread to compute row i.
4. **For each thread (computing a row):**
    - 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:

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

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