DAA UNIT – 3 (Greedy And Dynamic Programming algorithmic Strategies) – END-SEM PYQ Answers

- **MAY / JUN 2022**

Q1) a) Solve the matrix chain multiplication for the following 6 matrix problem using Dynamic programming. [10]

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimensions	10×20	20×5	5×15	15×50	50×10	10×15

Matrices:

- $A_1 : 10 \times 20$
- $A_2 : 20 \times 5$
- $A_3 : 5 \times 15$
- $A_4 : 15 \times 50$
- $A_5 : 50 \times 10$
- $A_6 : 10 \times 15$

Dimension array $p : [10, 20, 5, 15, 50, 10, 15]$(length 7 for 6 matrices).

We use the DP algorithm m[i,j] = minimum scalar multiplications for $A_i \ldots A_j$, and s[i,j] = index k that achieves the minimum.

Step-by-step results (summary of DP tables)

Key values (only upper triangle shown; m[i][i]=0):

- $m[1,2] = 10 \cdot 20 \cdot 5 = 1000$
- $m[2,3] = 20 \cdot 5 \cdot 15 = 1500$
- $m[3,4] = 5 \cdot 15 \cdot 50 = 3750$
- $m[4,5] = 15 \cdot 50 \cdot 10 = 7500$
- $m[5,6] = 50 \cdot 10 \cdot 15 = 7500$

Calculated DP table m[i,j] (only relevant entries):

m[1,2] = 1000

m[1,3] = 1750

m[1,4] = 7250

m[1,5] = 7750

m[1,6] = 8750 <-- global optimum cost

m[2,3] = 1500

m[2,4] = 8750

m[2,5] = 7250

m[2,6] = 8500

m[3,4] = 3750

m[3,5] = 6250

m[3,6] = 7000

m[4,5] = 7500

m[4,6] = 9750

m[5,6] = 7500

The minimum number of scalar multiplications for the whole chain $A_1 \dots A_6$is:

$$m[1,6] = 8750 \text{ scalar multiplications}$$

The s[i,j] split table yields the optimal parenthesization. The optimal parenthesization is:

$$((A_1 \times A_2) \times (((A_3 \times A_4) \times A_5) \times A_6))$$

(Equivalently written: $((A_1 A_2)((A_3 A_4) A_5) A_6)$with grouping as above.)

How we got it (algorithm outline)

Pseudocode (standard):

```
MatrixChainOrder(p)
 n = length(p) - 1
 for i = 1 to n
   m[i,i] = 0
 for l = 2 to n       // chain length
   for i = 1 to n-l+1
     j = i + l - 1
     m[i,j] = ∞
     for k = i to j-1
       q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
       if q < m[i,j]
```

m[i,j] = q

s[i,j] = k

return m and s

Applying this algorithm with the given p produced the tables and result above.

b) Explain Greedy strategy: Principle, control abstraction, time analysis of control abstraction with suitable example. [8]

Principle: The greedy strategy builds a solution incrementally by repeatedly choosing the best available *local* option (a "greedy choice") at each step, hoping these local optimum choices lead to a global optimum. Greedy algorithms are simple and fast but they only work when the problem has the greedy-choice property (a local optimal choice leads to a global optimum) and optimal substructure.

Control abstraction (conceptual steps / control structure)

A greedy algorithm typically follows this high-level control abstraction:

1. Initialize an empty (or trivial) feasible solution structure.
2. Repeat until a complete solution is formed:
    - Select the best candidate according to some greedy criterion (this is the greedy choice).
    - Check feasibility (optionally): if selecting the candidate keeps the partial solution feasible, accept it; otherwise reject it.
    - Update the partial solution and the set of candidate choices (remove or adjust candidates).
3. Return the constructed solution.

These three operations—select, feasibility-check, update—are the core control primitives for greedy algorithms.

Time analysis of the control abstraction

Time complexity depends on how the three primitives are implemented:

- Selection (choosing the next greedy candidate):
    - If candidates are scanned linearly each time, selection cost may be $O(n)$ per choice.
    - If candidates are kept in a priority queue (heap), selection is $O(\log n)$ per choice.
- Feasibility check:
    - Often $O(1)$ or $O(\log n)$ depending on data structures (e.g., checking overlap for intervals may be constant if we track current finish time).

- Update:
    - Removing/adjusting candidates can be $O(1)$amortized or $O(\log\ n)$if using heap/tree structures.

Total cost example: if there are $n$choices and you sort candidates first in $O(n\log\ n)$, then perform $n$selections each in $O(1)$, total cost is $O(n\log\ n)$. If you don't sort and pick by scanning $n$times with $O(n)$per pick, cost is $O(n^2)$.

Example — Activity Selection Problem (classic greedy example)

Problem: Given activities with start and finish times, select the maximum number of non-overlapping activities.

Greedy criterion: always pick the activity that finishes earliest among remaining activities.

Control abstraction applied:

1. Sort activities by finish time — cost $O(n\log\ n)$.
2. Initialize last_finish = -∞, selected = empty.
3. For each activity in sorted order:
    - If activity.start ≥ last_finish, select it and set last_finish = activity.finish (feasibility check and update are $O(1)$).
4. Return selected activities.

Time complexity: sorting dominates → $O(n\log\ n)$. The selection loop is $O(n)$so total $O(n\log\ n)$.

Why greedy works here: the activity-selection problem satisfies the greedy-choice property and optimal substructure, so the earliest-finish-first local choice leads to an optimal global solution.

Remarks / Cautions

- Greedy works only when problem structure guarantees global optimality from local choices (e.g., MST with Kruskal/Prim, activity selection).
- Some problems that seem similar (e.g., general coin change with arbitrary coin systems) do not always admit greedy solutions; correctness must be proved or a counterexample shown.

Q2) a) Explain the 'dynamic programming' approach for solving problems. Write a dynamic programming algorithm for creating an optimal binary search tree for a set of 'n' keys. Use the same algorithm to construct the optimal binary search tree for the following 4 keys. [10]

Key	A	B	C	D
Probability	0.1	0.2	0.4	0.3

1. What is dynamic programming (brief)?

Dynamic programming (DP) is a technique for solving optimization problems by:

- Decomposing the problem into overlapping subproblems,
- Solving each subproblem once and storing (memoizing) its solution,
- Reusing stored solutions to build up solutions to larger subproblems.
  DP is used when the problem has optimal substructure (optimal solution contains optimal solutions to subproblems) and overlapping subproblems.

2. OBST problem statement (simplified)

Given n ordered keys $K_1 < K_2 < \cdots < K_n$ with search probabilities p_i (probability that a search is for key K_i), build a binary search tree (BST) that minimizes the expected search cost:

$$\text{Expected Cost} = \sum_{i=1}^{n} p_i \cdot (\text{depth of } K_i)$$

(Depth defined as 1 for root, 2 for its children, etc.)

3. DP formulation (standard)

Let e[i][j] = minimal expected search cost for keys $K_i, K_{i+1}, \ldots, K_j$.

Let w[i][j] = $\sum_{k=i}^{j} p_k$ (total probability mass of keys in that interval).

For an interval $i \le j$, if we choose root r (where $i \le r \le j$), the cost for that choice is:

$$\text{cost}(r) = e[i][r-1] + e[r+1][j] + w[i][j]$$

because every key in the subtrees increases depth by 1 (so add w[i][j] once). The recurrence:

- Base: $e[i][i-1] = 0$ (empty tree)
- For $i \le j$:

$$e[i][j] = \min_{r=i..j} \{ e[i][r-1] + e[r+1][j] + w[i][j] \}$$

Store root[i][j] = r that attains the minimum so we can reconstruct the tree.

4. DP algorithm (pseudocode)

```
OBST(p[1..n])   // p[i] are probabilities for keys K1..Kn
 for i = 1 to n+1:
   e[i][i-1] = 0
   w[i][i-1] = 0
 for length = 1 to n:
   for i = 1 to n-length+1:
```

```
    j = i + length - 1
    w[i][j] = w[i][j-1] + p[j]
    e[i][j] = +∞
    for r = i to j:
      cost = e[i][r-1] + e[r+1][j] + w[i][j]
      if cost < e[i][j]:
        e[i][j] = cost
        root[i][j] = r
  return e, root
```

- After this, root[1][n] is the root of the optimal tree. Recur using root to print tree structure.

Time complexity: The triple nested loops (length × i × r) give $O(n^3)$time and $O(n^2)$space. (With special monotonicity/Knuth optimization, this can be improved to $O(n^2)$in some cases.)

5. Apply to the given 4 keys

Keys / probabilities:

- A: $p_1 = 0.1$
- B: $p_2 = 0.2$
- C: $p_3 = 0.4$
- D: $p_4 = 0.3$

Run the DP (using the recurrence above). The DP yields:

- $w[1,4] = 0.1 + 0.2 + 0.4 + 0.3 = 1.0$
- The minimal expected cost for the whole set is:

$$e[1,4] = 1.7$$

root[1,4] = 3 → choose key C as root.

Reconstructing the root table gives the optimal tree:

- Root: C (K3)
    - Left subtree (keys A,B): root[1,2] = 2 → B is left child of C
        - Left child of B: A
        - Right child of B: (empty)
    - Right subtree (key D): D is right child of C

So the optimal BST (with depths in parentheses) is:

C (depth 1)

/ \

B(2) D(2)

/

A(3)

Compute expected cost:

- C: $0.4 \times 1 = 0.4$
- B: $0.2 \times 2 = 0.4$
- A: $0.1 \times 3 = 0.3$
- D: $0.3 \times 2 = 0.6$
  Total = $0.4 + 0.4 + 0.3 + 0.6 = 1.7.$

Thus the OBST for the given keys is as above and has expected search cost 1.7.

b) Explain Dynamic programming: Principle, control abstraction, time analysis of control abstraction with suitable example. [8]

Principle: Dynamic programming converts a complex problem into simpler overlapping subproblems, solves each subproblem once, stores the result, and reuses it. Two required properties:

- **Optimal substructure**: optimal solution of the whole problem contains optimal solutions to subproblems.
- **Overlapping subproblems**: same subproblems are re-solved many times in naive recursion.

Control abstraction (typical DP recipe)

1. Define subproblem precisely (what parameters identify a subproblem).
2. Write recurrence that expresses solution of a subproblem in terms of smaller subproblems.
3. Boundary/base cases for smallest subproblems.
4. Order of computation (bottom-up iteration or top-down memoized recursion).
5. Store solutions in a table (array, matrix, etc.).
6. (Optional) Reconstruct solution using auxiliary root/choice table if needed.

For OBST above:

- Subproblem: e[i][j] for interval of keys i..j.

- Recurrence: minimization over roots r.
- Base: e[i][i-1] = 0.
- Order: increase interval length from 1 to n (bottom-up).
- Store e and root tables.

Time analysis of the control abstraction

- Suppose there are S distinct subproblems. For each subproblem we may try up to k candidate choices, and each candidate's cost is computed using constant-time table lookups (if precomputed). Then time ≈ $O(S \times k)$.
- Space = memory to store the table(s) = O(S).

Examples:

- OBST (above): S = O(n^2) intervals, for each interval we try O(n) roots ⇒ time $O(n^3)$, space $O(n^2)$. (Knuth optimization can reduce to $O(n^2)$when applicable.)
- 0/1 Knapsack (DP by capacity): subproblems dp[i][w] (first i items, capacity w), S = O(nW); transition tries constant-time update ⇒ time $O(nW)$, space $O(nW)$(or $O(W)$with rolling array).
- Fibonacci: naive recursion exponential; DP (tabulation or memoization) gives O(n) time and O(n) space.

Worked small example (0/1 Knapsack control abstraction)

- Subproblem: best value using first i items with capacity w.
- Recurrence: dp[i][w] = max(dp[i-1][w], value_i + dp[i-1][w-weight_i]).
- Base: dp[0][*] = 0, dp[*][0] = 0.
- Complexity: O(nW).

➢ **MAY/JUNE 2023**

Q1) a) Consider the following instance of the knapsack problem. Find the optimial solution by using dynamic programming approach. Capacity of the knapsack = 5. [10]

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

Items:

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

Capacity $W = 5$

We build a DP table:

Let

$$dp[i][w] = \text{maximum profit using first } i \text{ items and capacity } w$$

DP Table Construction

Initialization

- $dp[0][w] = 0$
- $dp[i][0] = 0$

DP Table

i/item	w=0	1	2	3	4	5
0	0	0	0	0	0	0
1 (wt=2, p=12)	0	0	12	12	12	12
2 (wt=1, p=10)	0	10	12	22	22	22
3 (wt=3, p=20)	0	10	12	22	30	32
4 (wt=2, p=15)	0	10	15	25	30	37

Final Answer

Maximum Profit = 37

Selected Items

Traceback:

- $dp[4][5] = 37 \rightarrow$ item 4 included
  Remaining capacity: $5 - 2 = 3$
- $dp[3][3] = 22 \rightarrow$ item 3 not included
- $dp[2][3] = 22 \rightarrow$ item 2 included
  Remaining capacity: $3 - 1 = 2$
- $dp[1][2] = 12 \rightarrow$ item 1 included

Optimal Set = { Item 1, Item 2, Item 4 }

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

- Maximum profit = 37
- Items selected = {1, 2, 4}

Q1) b) – Job Scheduling Using Greedy Algorithm

b) What is job scheduling algorithm? How job scheduling algorithm can be solved using Greedy algorithmic approach? Explain your answer with respect to Principle, control abstraction, time analysis of control abstraction, of greedy approach for the following instance of knapsack problem. **[8]**

Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Jobs:

Job	J1	J2	J3	J4	J5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

1. What is Job Scheduling?

Job scheduling problem:
Each job has:

- Deadline
- Profit

Each job requires 1 unit time.
Objective: Maximize profit while completing jobs before deadlines.

2. Greedy Strategy to Solve Job Scheduling

Greedy Principle: Select the job with maximum profit first, and schedule it in the latest possible free slot before its deadline.

Why it works?

- Taking highest-profit job first ensures global optimality.
- Latest free slot ensures more space for earlier-deadline jobs.

3. Control Abstraction for Job Scheduling

1. Sort jobs in descending order of profit.
2. Create a slot array of size = max deadline.
3. For each job in sorted order:
    - Find latest empty slot ≤ its deadline.
    - If slot is free → schedule job.

4. Time Complexity Analysis

- Sorting jobs: $O(n \log n)$
- Scheduling each job: $O(n)$ in worst case

Total time:

$$O(n \log n + n^2)$$

With DSU optimization → $O(n \log n)$

5. Apply on Given Data

Step 1: Sort by Profit

Job	**Profit**	**Deadline**
J2	100	1
J1	60	2
J4	40	2
J3	20	3
J5	20	1

Max deadline = 3 → Slots = [_ _ _]

Step 2: Schedule

Job J2 (profit 100, deadline 1)

Slot 1 free → place:

- [J2 _ _]

Job J1 (profit 60, deadline 2)

Slot 2 free:

- [J2 J1 _]

Job J4 (profit 40, deadline 2)

Slot 2 full → try slot 1: full → reject

Job J3 (profit 20, deadline 3)

Slot 3 free:

- [J2 J1 J3]

Job J5 (profit 20, deadline 1)

Rejected

Final Job Sequence

$$J2,\ J1,\ J3$$

Maximum Profit

$$100+60+20=180$$

- Greedy solution picks jobs J2, J1, J3
- Maximum profit = 180

Q2) a) What is greedy approach? Explain Job scheduling algorithm using Greedy approach for following examples. Give the sequence of job scheduling. [8]

Input: Four jobs with following deadlines and profits

Job1D	deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Input: Five Jobs with following deadlines and profits

Job1D	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

What is the greedy approach?

A *greedy algorithm* builds a solution step-by-step by always taking the best (locally optimal) choice

available at each step, hoping these local choices lead to a global optimum. For job scheduling (maximize profit where each job takes 1 unit time and has a deadline), the standard greedy rule is:

1. Sort jobs in descending order of profit.
2. For each job in that order, place it in the latest available time slot ≤ its deadline (if any).
3. Continue until all jobs considered.

This yields an optimal schedule for the 1-unit, single-machine profit-deadline scheduling problem.

Instance 1

Input jobs:

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

Step 1 — sort by profit (desc): c(40), d(30), a(20), b(10)
Max deadline = 4 → slots 1..4 initially empty.

Place jobs:

- c (d=1, p=40) → place in slot 1: [c _ _ _]
- d (d=1, p=30) → slot 1 already used → cannot place → rejected
- a (d=4, p=20) → place in latest ≤4 → slot 4: [c _ _ a]
- b (d=1, p=10) → slot1 used → rejected

Final schedule (by slot order): slot1 = c, slot2 = —, slot3 = —, slot4 = a
Sequence of scheduled jobs: c, a
Total profit: $40 + 20 = 60$

(You cannot place more because b,d have deadline=1 which conflicts with c.)

Instance 2

Input jobs:

Job	**Deadline**	**Profit**
a	2	100

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

Sort by profit (desc): a(100), c(27), d(25), b(19), e(15)
Max deadline = 3 → slots 1..3.

Place jobs:

- a (d=2, p=100) → place in latest ≤2 → slot2: [_ a _]
- c (d=2, p=27) → slot2 occupied → place in slot1: [c a _]
- d (d=1, p=25) → slot1 occupied → rejected
- b (d=1, p=19) → slot1 occupied → rejected
- e (d=3, p=15) → slot3 free → place: [c a e]

Final schedule (by slot order): slot1 = c, slot2 = a, slot3 = e
Sequence of scheduled jobs: c, a, e (in execution order: c → a → e)
Total profit: $27+100+15=142$

Instance 1 schedule: c, a → profit 60

Instance 2 schedule: c, a, e (slots 1,2,3) → profit 142

b) What is optimal binary search tree? How dynamic programming approach is used to build OBST for following tale. [10]

	1	2	3	4
Keys→	10	20	30	40
Frequency→	4	2	6	3

What is an OBST?

Given sorted keys $K_1 < \cdots < K_n$ with search frequencies f_i (or probabilities), an *optimal binary search tree* is a BST that minimizes the expected search cost:

$\text{Cost} = \sum_{i=1}^{n} f_i \cdot \text{depth}(K_i)$ (depth of root = 1).

DP formulation (standard)

- Let e[i][j] = minimal total search cost (weighted by frequency) for keys $K_i .. K_j$.
- Let w[i][j] = \sum_{k=i}^j f_k (total frequency in that interval).
- Recurrence (for $i \leq j$):

$$e[i][j] = \min_{r=i..j} (e[i][r-1] + e[r+1][j] + w[i][j])$$

(choose root r for interval i..j; each key in subtrees increases depth by 1, hence add w[i][j]).

Base: e[i][i-1] = 0 (empty interval). Compute intervals by increasing length (bottom-up). Keep root[i][j] that attains the minimum to reconstruct tree.

Given data

Keys: 10, 20, 30, 40
Frequencies: 4, 2, 6, 3 (sum = 15)

DP results (selected table entries)

- w (total frequency in interval):
  - w[1,1]=4, w[1,2]=6, w[1,3]=12, w[1,4]=15, etc.
- e[i][j] (minimum total weighted cost for interval i..j):
  - e[1,1] = 4
  - e[2,2] = 2
  - e[3,3] = 6
  - e[4,4] = 3
  - e[1,2] = 8
  - e[2,3] = 10
  - e[3,4] = 12
  - e[1,4] = 26 ← minimal total weighted search cost for all keys
- root[i][j] (root choice that achieves e[i][j]):
  - root[1,2] = 1 (key 10)
  - root[1,3] = 3 (key 30)
  - root[1,4] = 3 (key 30)
  - root[2,3] = 3 (key 30)
  - root[3,4] = 3 (key 30)

- root[4,4] = 4 (key 40)

Constructed optimal tree

Using root entries, the optimal BST is:

30 (root)

/ \

10 40

\

20

Depths and contributions:

- key 30 (freq 6) at depth 1 → contribution $6 \times 1 = 6$
- key 10 (freq 4) at depth 2 → contribution $4 \times 2 = 8$
- key 20 (freq 2) at depth 3 → contribution $2 \times 3 = 6$
- key 40 (freq 3) at depth 2 → contribution $3 \times 2 = 6$

Total cost = $6 + 8 + 6 + 6 = 26$.

If you want average search depth (expected depth per search), divide by total frequency 15:

$$\text{Average depth} = \frac{26}{15} \approx 1.733$$OBST root = key 30; tree shown above; total weighted cost 26; average depth ≈ 1.733.

➢ NOV/DEC 2023

Q1) a) — Job Sequencing (High-level Description + Feasible Solutions + Optimal Solution) High-level description of job sequencing (profit–deadline scheduling)

Q1)	a)	Write High-level description of job sequencing algorithm. Let number of jobs (n)=5; Profit vector P={20, 15, 10, 5, 1); Deadline vector D={2, 2, 1, 3, 3) Find the feasible solutions. What is the optimal solution and maximum profit? **[9]**

The goal is to schedule jobs such that:

- Each job takes 1 unit time
- Each job J_i has:
  - Profit P_i
  - Deadline D_i

- A job must be completed on or before its deadline
- Objective: maximize total profit

Greedy Strategy

1. Sort jobs in decreasing order of profit.
2. Make time slots from 1 to max deadline.
3. For each job in sorted order:
    - Attempt to place it in the latest free slot ≤ deadline.
    - If free → schedule it
    - Else → skip job
4. Return scheduled jobs + profit.

Given

Number of jobs = 5
Profit vector:

$$P = \{20,15,10,5,1\}$$

Deadline vector:

$$D = \{2,2,1,3,3\}$$

Let jobs be:

$$J_1(20,2), \quad J_2(15,2), \quad J_3(10,1), \quad J_4(5,3), \quad J_5(1,3)$$

Max deadline = 3 → slots = $[1,2,3]$

Step 1: Sort by profit

Order: J_1, J_2, J_3, J_4, J_5

Step 2: Schedule

Job	**Deadline**	**Place in slot**	**Result**
J1	2	slot 2	[_ , J1 , _]
J2	2	slot 1 (slot 2 full)	[J2 , J1 , _]
J3	1	slot 1 full → cannot schedule	skip
J4	3	slot 3	[J2 , J1 , J4]
J5	3	slot 3 full → skip	

Feasible scheduled jobs : {J2, J1, J4}

Maximum profit: 15 + 20 + 5 = 40

Q1 (b) — Knapsack Solution via Dynamic Programming

b) Consider the following instance of the knapsack problem. Find the optimal solution by using dynamic programming approach. **[9]**

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

Capacity of the knapsack = 5.

Items:

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

Capacity $W = 5$

DP Table Result Summary

After constructing the DP table:

Maximum profit = 37

Selected items (traceback):

- Item 4 (wt=2, p=15)
- Item 2 (wt=1, p=10)
- Item 1 (wt=2, p=12)

Optimal Solution

$$\text{Items } \{1,2,4\},\ \text{Maximum Profit } = 37$$

Q2) a) — Job Scheduling With Full Greedy Explanation

Q2) a) What is Job scheduling algorithm? How job scheduling algorithm can be solved using Greedy algorithmic approach? Explain your answer with respect to Principle, control abstraction, time analysis of control abstraction, of greedy approach for the following instance of Knapsack problem. **[12]**

Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Jobs:

Job	Deadline	Profit
J1	2	60
J2	1	100
J3	3	20
J4	2	40
J5	1	20

Greedy Principle

Choose jobs in descending order of profit, place them in the latest free slot before their deadline.

Control Abstraction

1. Sort jobs by profit.
2. For each job, scan backward from its deadline to 1.
3. If a slot is empty → assign job.
4. Continue.

Time Analysis

- Sorting: $O(n \log n)$
- Slot assignment: worst-case $O(n)$ per job
  Total: $O(n^2)$

Apply Greedy

Sorted by profit: J2(100), J1(60), J4(40), J3(20), J5(20)

Slots = [1,2,3]

Job	Deadline	Slot	Schedule
J2	1	1	[J2 _ _]
J1	2	2	[J2 J1 _]
J4	2	slot2 full → slot1 full → reject	—
J3	3	3	[J2 J1 J3]
J5	1	slot1 full → reject	—

Final Sequence: {J2, J1, J3}

Maximum Profit: 100 + 60 + 20 = 180

Q2) b) Write steps for Greedy approach for Job sequencing

Steps for Greedy Approach for Job Sequencing: The greedy approach for job sequencing with deadlines aims to maximize profit by scheduling jobs within their deadlines. The algorithm selects jobs based on profitability while respecting deadline constraints.

Algorithm Steps

1. **Sort jobs in descending order of profit**: Arrange all given jobs so that the job with the highest profit appears first, followed by jobs with decreasing profit values.
2. **Find the maximum deadline**: Determine the maximum deadline value (dmax) from all the jobs to establish the time slots available for scheduling.
3. **Initialize time slot array**: Create an array of size equal to the maximum deadline, where each index represents a time slot, and initialize all slots to empty or -1.
4. **Iterate through sorted jobs**: For each job in the sorted list, check if it can be scheduled within its deadline.
5. **Schedule in latest available slot**: For each job, try to place it in the latest free time slot available at or before its deadline. Scan backward from the deadline to slot 1 to find a free position.
6. **Add profit and mark slot**: If a free slot is found, schedule the job in that slot, mark it as occupied, and add the job's profit to the total profit. If no slot is available before the deadline, skip the job and move to the next one.
7. **Calculate maximum profit**: After all jobs are processed, compute the total profit by summing the profits of all scheduled jobs.

- **MAY/JUNE 2024**

Q1) a) Write a control abstraction for greedy method. Comment on the time complexity of this abstraction?

A greedy algorithm uses the following control structure:

Greedy(Input)

1. Initialize solution S = Ø

2. Initialize candidate set C = all elements of input

3. while C ≠ Ø do

4. choose x from C according to greedy criterion

5. if x is feasible with respect to S then

6. add x to S

7. remove x from C

8. return S

Explanation of Steps

1. Select the next best candidate using a *greedy choice*
   (e.g., highest profit, shortest job, earliest deadline).
2. Check feasibility
   (does adding this candidate violate constraints?).
3. Add the candidate to the partial solution.
4. Repeat until all candidates are considered.

Time Complexity

- Let n = number of candidates
- Selecting best candidate:
  - If sorting used → O(n log n)
  - If scanning for best each time → O(n²)
- Feasibility check: usually O(1) or O(log n)

Overall Time Complexity:

$$O(n \log n) \text{ (if sorted once)}, O(n^2) \text{ (if linear scan used)}$$

Q1 (b) — Greedy Knapsack (Fractional Knapsack)

b) Find an optimal solution for the following knapsack instance using greedy method. Number of objects n = 5. Capacity of knapsack m = 100. **[8]**

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

Objects:

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

Capacity = 100

Compute Profit/Weight Ratio

Obj	W	P	P/W
01	20	10	0.5
02	30	20	0.667
03	66	30	0.454
04	40	40	1.0
05	60	50	0.833

Sort by decreasing P/W

Order:

$$04,\quad 05,\quad 02,\quad 01,\quad 03$$

Greedy Filling

Start capacity = 100

1. Take 04: weight 40 → remaining 60 → profit = 40
2. Take 05: weight 60 → remaining 0 → profit = 40 + 50 = 90
   (Knapsack full)

Optimal Profit: 90
Items Taken: {04, 05}

Q1) c) Comment on Statement "Problems which do not satisfy the principle of optimality cannot be solved by dynamic programming."

Explanation

- Dynamic programming requires optimal substructure — solution of the main problem contains solutions of its subproblems.
- It also requires overlapping subproblems.
- If a problem's optimal solution does not include optimal solutions of its subproblems, DP fails.

Example

- 0/1 Knapsack violates optimal substructure for fractional decisions.
- TSP requires global checking – greedy and DP fail without additional constraints.

Final Statement

If optimal substructure does not hold, DP cannot be applied.

Q2) a) — Control Abstraction for Dynamic Programming + Time Complexity Control Abstraction for DP

DP(Input)

1. Define subproblems P(i,j)

2. Create DP table T[][]

3. Initialize base cases in T

4. for each subproblem in increasing order of size do

5. compute T[i][j] using recurrence

6. return T[1][n] or optimal value

Explanation

1. Break problem into subproblems
2. Solve each once
3. Store results
4. Use recurrence relation
5. Build solution bottom-up

Time Complexity

Let:

- S = number of subproblems
- k = number of choices per subproblem

Then:

$$T = O(S \cdot k)$$

Examples:

- Matrix chain multiplication → $O(n^3)$
- Bellman–Ford → O(VE)
- Knapsack → O(nW)

Q2) b) — Matrix Chain Multiplication (A1–A4)

b) Consider 4 matrices A1, A2, A3 and A4. The orders of these matrices are given below: **[10]**

Matrix	Order
A1	3×5
A2	5×4
A3	4×2
A4	2×4

Find the optimal sequence of chain matrix multiplication of these matrices using dynamic programming approach. Clearly give the final sequence and total number of multiplications involved.

Matrix dimensions:

Matrix	Order
A1	3×5
A2	5×4
A3	4×2
A4	2×4

Dimension array:

$$p = [3,5,4,2,4]$$

We compute m[i][j] = minimum multiplications.

Step-by-step DP Calculation

Chain Length = 2

- m[1,2] = 3×5×4 = 60
- m[2,3] = 5×4×2 = 40
- m[3,4] = 4×2×4 = 32

Chain Length = 3

m[1,3]

Try k = 1:
= m[1,1] + m[2,3] + (3×5×2)
= 0 + 40 + 30 = 70

Try k = 2:
= 60 + 0 + (3×4×2)
= 60 + 24 = 84

→ m[1,3] = 70 (k=1)

m[2,4]

k = 2:
= 0 + 32 + (5×4×4) = 32 + 80 = 112

k = 3:
= 40 + 0 + (5×2×4) = 40 + 40 = 80

→ m[2,4] = 80 (k=3)

Chain Length = 4 (whole chain m[1,4])

Try k = 1:
= 0 + 80 + (3×5×4) = 80 + 60 = 140

Try k = 2:
= 60 + 32 + (3×4×4) = 92 + 48 = 140

Try k = 3:
= 70 + 0 + (3×2×4) = 70 + 24 = 94

→ m[1,4] = 94 (k = 3)

FINAL ANSWER

Optimal Parenthesization

Split at k = 3 → (A1 A2 A3) A4
Inside split at k = 1 → (A1 (A2 A3)) A4

So:

$$((A_1(A_2A_3))A_4)$$

Minimum Number of Scalar Multiplications

94

- **NOV/DEC 2024**

Q1) a) You are given a set of tasks, each with a deadline and a penalty for missing the deadline. The objective is to schedule these tasks in a way that minimizes the total penalty incurred. However, you can only work on one task at a time, and once a task is started, it must be completed before moving on to the next task. Additionally, you can't start a task after its deadline has passed. Design a greedy algorithm to efficiently schedule these tasks to minimize the total penalty and prove its correctness. [8]

Problem restatement (unit-time jobs):
We have jobs $J_1, \ldots, J_n$. Each job J_i has a deadline d_i (positive integer) and a penalty p_i incurred if the job is not completed by its deadline. Each job requires 1 unit of time. At most one job can be processed in any time unit. Schedule some jobs in integer time slots $1,2, \ldots, max\ d_i$ (one job per slot). Jobs scheduled after their deadline are considered missed (or equivalently we only schedule jobs in slots ≤ their deadline). Objective: minimize total penalty of unscheduled (missed) jobs (equivalently maximize total penalty *saved* by scheduling jobs).

Greedy idea (standard)

Schedule jobs that have large penalty first — because skipping a high-penalty job costs more. Place each chosen job as late as possible before its deadline so earlier slots remain available for other jobs with earlier deadlines.

Algorithm (pseudo)

ScheduleByPenalty(Jobs)

1. Sort jobs in descending order of penalty p_i

2. Let T = max_i d_i

3. Create an array slot[1..T], initialize all to empty

4. for each job j in sorted order:

5. for t = min(d_j, T) down to 1:

6. if slot[t] is empty:

7. slot[t] = j

8. mark j as scheduled (no penalty)

9. break // next job

10. // jobs not placed incur their penalties

11. return scheduled set and total penalty = sum of p_i for unscheduled jobs

Complexity

- Sorting: $O(n \log n)$
- For each job, inner loop may check up to $T \leq n$slots → worst-case $O(n)$per job.
- Total worst-case: $O(n^2)$. With union-find (disjoint-set) optimization for finding latest free slot the scheduling stage can be reduced to nearly $O(n\alpha(n))$giving overall $O(n \log n)$.

Correctness (exchange argument sketch)

We show the greedy produces an optimal schedule (maximizes total saved penalty, equivalently minimizes total penalty):

- Let G be schedule produced by greedy and OPT be an optimal schedule.
- Consider jobs sorted by descending penalty. Suppose at the first position where G and OPT differ there is some job $x$that G scheduled (in some slot) but OPT either did not schedule or scheduled a different job $y$in that slot with penalty $p_y \leq p_x$(because greedy picks highest penalties first).
- If OPT did not schedule $x$but scheduled some lower-penalty job $y$in a slot that could accommodate x, we can exchange: replace $y$by $x$in OPT. This does not increase the total missed penalty (it decreases or keeps same), and keeps schedule feasible (deadlines respected because slot was ≤ deadlines for both originally-or-forced placement).
- Repeating such exchanges progressively transforms OPT into G (or another schedule with equal value) without increasing total penalty. Hence G is optimal.

b) Suppose we have a knapsack with a maximum weight capacity of 15 units, and we have the following items with their respective weights (Wi) and values (Vi) :

Use greedy approach to maximize the total value of items we can put into the knapsack without exceeding its weight capacity. [8]

Objects	Weight	Value
O1	8	10
O2	6	8
O3	4	3
O4	2	4

Given: Capacity $W = 15$. Items:

Item	**Weight** w_i	**Value** v_i	**Value/Weight** $r_i = v_i / w_i$
O1	8	10	$10/8 = 1.25$
O2	6	8	$8/6 \approx 1.3333$
O3	4	3	$3/4 = 0.75$
O4	2	4	$4/2 = 2.0$

Greedy strategy (fractional knapsack): sort by ratio r_i(value per unit weight) descending, take as much as possible of the best ratio item, then next, etc. This is optimal for the fractional knapsack problem.

Order by ratio: O4 (2.0), O2 (~1.3333), O1 (1.25), O3 (0.75).

Fill capacity $= 15$**step-by-step:**

1. Take O4 entirely: weight 2, value 4. Remaining capacity $= 15 - 2 = 13$. Total value $= 4$.
2. Take O2 entirely: weight 6, value 8. Remaining capacity $= 13 - 6 = 7$. Total value $= 4 + 8 = 12$.
3. Next O1 has weight 8 but only 7 capacity remains. Take fraction $7/8$of O1:
    - value contributed $= 10 \times (7/8) = 8.75$
    - remaining capacity becomes 0.
4. O3 not considered (no capacity left).

Total value (digit-by-digit): $4 + 8 + 8.75 = 20.75$.

Answer: Using the greedy fractional approach, we achieve total value 20.75. Items taken: O4 and O2 fully, and fraction $7/8$of O1.

Note: If the 0/1 knapsack (items must be whole) is required, greedy by ratio is not guaranteed to be optimal. For 0/1 here, you would need DP to find the true optimum; greedy may yield a suboptimal integer solution.

Complexity: sorting $O(n \log n)$, then one pass $O(n) \Rightarrow$ overall $O(n \log n)$.

c) With respect to dynamic programming, what do you understand by optimal substructure? [2]

Definition: A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to its subproblems. Formally, if the optimal solution to instance Pcan be composed from optimal solutions to subinstances $P_1, P_2, \ldots$, then the problem has optimal substructure. This property allows dynamic programming: compute and store optimal solutions to subproblems and combine them to form an optimal solution for the whole problem.

Example: In shortest-path (Dijkstra) or matrix chain multiplication, the optimal full solution uses optimal solutions for sub-ranges — hence DP works.

Q2) a) We are given the sequence {4, 10,3, 12,20, and 7}. We are given with five matrices of the size 4 × 10, 10 × 3,3 × 12, 12 × 20,20 × 7 respectively. Use dynamic programming to solve chain matrix multiplication. [10]

You have 5 matrices with dimensions (in order):

- $A_1 : 4 \times 10$
- $A_2 : 10 \times 3$
- $A_3 : 3 \times 12$
- $A_4 : 12 \times 20$
- $A_5 : 20 \times 7$

So the dimension array is $p = [4, 10, 3, 12, 20, 7]$. We want the minimum number of scalar multiplications to compute $A_1 A_2 A_3 A_4 A_5$.

DP recurrence

Let $m[i, j]$= minimum cost to multiply $A_i \cdots A_j$.
Base: $m[i, i] = 0$.
Recurrence:

$$m[i, j] = \min_{i \le k < j} \{ m[i, k] + m[k+1, j] + p_{i-1} \cdot p_k \cdot p_j \}.$$

Key computed subproblems (important steps)

Length 2 (direct multiplication):

- $m[1,2] = 4 \cdot 10 \cdot 3 = 120$
- $m[2,3] = 10 \cdot 3 \cdot 12 = 360$
- $m[3,4] = 3 \cdot 12 \cdot 20 = 720$
- $m[4,5] = 12 \cdot 20 \cdot 7 = 1680$

Length 3:

- $m[1,3] = \min(0 + 360 + 4 \cdot 10 \cdot 12,\ 120 + 0 + 4 \cdot 3 \cdot 12) = \min(840,\ 264) = 264.$
- $m[2,4] = \min(0 + 720 + 10 \cdot 3 \cdot 20,\ 360 + 0 + 10 \cdot 12 \cdot 20) = \min(1320,\ 2760) = 1320.$
- $m[3,5] = \min(0 + 1680 + 3 \cdot 12 \cdot 7,\ 720 + 0 + 3 \cdot 20 \cdot 7) = \min(1932,\ 1140) = 1140.$

Length 4:

$$m[1,4] = min\,(\,0+1320+4\cdot10\cdot20=2120\,,\ 120+720+4\cdot3\cdot20=1080\,,\ 264+0+4\cdot12\cdot20=1224\,) = 1080.$$
$$m[2,5] = min\,(\,0+1140+10\cdot3\cdot7=1350\,,\ 360+1680+10\cdot12\cdot7=2880\,,\ 1320+0+10\cdot20\cdot7=2720\,) = 1350.$$

Length 5 (whole chain):

- $$m[1,5] = min\begin{cases} k=1:\ 0+m[2,5]+4\cdot10\cdot7=0+1350+280=1630 \\ k=2:\ m[1,2]+m[3,5]+4\cdot3\cdot7=120+1140+84=1344 \\ k=3:\ m[1,3]+m[4,5]+4\cdot12\cdot7=264+1680+336=2280 \\ k=4:\ m[1,4]+0+4\cdot20\cdot7=1080+560=1640 \end{cases}$$

So the minimum is $m[1,5]=1344$ scalar multiplications, achieved at split $k=2$.

Optimal parenthesization (reconstructing splits)

- Overall split at $k=2$: $(A_1 A_2)$ and $(A_3 A_4 A_5)$.
- For the right part $A_3 A_4 A_5$, best split was $k=4$ (i.e. $(A_3 A_4) A_5$).

Therefore the optimal grouping is:

$$(A_1 A_2)\,((A_3 A_4)\ A_5)$$

and the minimum number of scalar multiplications is 1344.

b) Under what situation do you think the dynamic programming approach for solving a knapsack problem might struggle to find the optimal solution? Briefly explain. [4]

Dynamic programming for the classic 0/1 knapsack (DP by capacity) runs in pseudo-polynomial time $O(nW)$, where n is the number of items and W is the knapsack capacity (assumed integer). Situations where DP struggles:

1. Very large capacity W: If W is large (e.g. thousands/millions) the memory/time $O(nW)$ becomes impractical even for moderate n.
2. Non-integer weights or huge numeric ranges: Standard DP table indices require integer capacities; when weights are fractional or extremely large integers you either need to scale (causing blow-up) or use alternative methods.
3. High dimensional constraints or multiple constraints: For multi-constraint knapsacks (e.g., weight and volume), state space grows multiplicatively and DP becomes infeasible.
4. When values/weights cause state explosion: If a value-based DP is used (DP over total value) and total value is huge, that DP also becomes impractical.

In short: DP yields exact optimal solutions but can be computationally and memory expensive when the numeric range of capacities/values is large; in those cases one uses approximation schemes (FPTAS), greedy heuristics (for fractional knapsack), or branch-and-bound.

c) Enlist the uses of writing control abstraction for any algorithmic strategies.[4]

Writing a control abstraction (a concise description of the high-level control flow) is useful because it:

1. Clarifies algorithm structure — shows the main operations (select, test, update) and the order of steps.
2. Guides correctness reasoning and invariants — helps state and prove correctness (e.g., greedy choice property or DP recurrence).
3. Facilitates complexity analysis — makes it easy to count major operations and identify bottlenecks (time/space).
4. Aids implementation and reuse — provides a template that can be implemented in different languages or adapted to variants; helps decide appropriate data structures.

NOTE: Please verify all answers before referring.