DAA UNIT – 4 (Backtracking and Branch-n-Bound) – END-SEM PYQ Answers

► **MAY / JUN 2022**

Q3) a) 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. [9]

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

1. Branch and Bound:

Branch & Bound (B&B) is a systematic search method for combinatorial optimization that:

- Branches: explores the solution space by creating a search tree where each node represents a partial solution (some items decided in/out).
- Bounds: computes an upper bound (best possible) on the profit reachable from any node; if the bound ≤ current best (incumbent), the node is pruned.
- Use either best-first (priority by bound) or depth-first search with bounding.

For knapsack the usual upper bound uses the fractional knapsack (greedy) on the remaining capacity — that gives a fast, optimistic bound.

2. Node representation

Each node stores:

- level (index of last considered item),
- weight (current total weight),
- profit (current profit),
- bound (upper bound on profit achievable from this node).

Items are assumed sorted by descending value/weight ratio for tight bounds.

3. Bound computation (fractional knapsack)

To compute bound(node):

- Start with current weight and profit.
- Greedily add remaining items (in ratio order) fully while capacity allows.
- If an item does not fit fully, add the fractional part to compute an optimistic profit.
- bound = current profit + profit of items added (including fractional).

4. Best-first Branch & Bound pseudocode (outline)

```
Knapsack_BB(items, W):
  Sort items by decreasing ratio r_i = v_i/w_i
  bestProfit = 0
  root = Node(level=0, profit=0, weight=0)
  root.bound = bound(root)
  PQ = max-priority-queue ordered by bound
  PQ.insert(root)
  while PQ not empty:
    node = PQ.extract_max()
    if node.bound <= bestProfit: continue   // prune
    level = node.level + 1
    // Branch: include item[level]
    left = Node(level=level,
                weight=node.weight + w[level],
                profit=node.profit + v[level])
    if left.weight <= W:
      if left.profit > bestProfit: bestProfit = left.profit
      left.bound = bound(left)
      if left.bound > bestProfit: PQ.insert(left)
    // Branch: exclude item[level]
    right = Node(level=level, weight=node.weight, profit=node.profit)
    right.bound = bound(right)
    if right.bound > bestProfit: PQ.insert(right)
  return bestProfit
```

Complexity: worst-case exponential in n, but pruning often reduces nodes drastically.

5. Apply to given instance

Items (label, profit, weight):

Item	Profit (Rs.)	Weight (kg)	ratio (p/w)
A	18	9	18/9 = 2.0
B	10	4	10/4 = 2.5
C	12	6	12/6 = 2.0
D	10	2	10/2 = 5.0

Capacity $W = 15.$

Sort by ratio (descending): D (5.0), B (2.5), A (2.0), C (2.0).
For clarity we reindex after sorting: item1=D, item2=B, item3=A, item4=C.

Compute overall greedy (fractional) bound at root:

- capacity remaining =15
- take D (w2,p10) → rem 13, profit 10
- take B (w4,p10) → rem 9, profit 20
- take A (w9,p18) → rem 0, profit 38
- no fractional needed. So root bound = 38.

Thus the maximum possible profit ≤ 38. If we can find a feasible integral solution with profit 38, it's optimal.

Search by B&B (sketch):

- Root: level 0, profit=0, weight=0, bound=38. Insert in PQ.
- Extract root, branch level=1 (D):
    - Include D: weight=2, profit=10. Bound (recompute) = 38 (same process: remaining capacity 13 allows B and A whole). Since profit 10 < best(=0), update bestProfit=10.
    - Exclude D: weight=0, profit=0. Bound if exclude D: we can still take B,A,C fractional: B(4)->10 (rem11), A(9)->18 (rem2), C(6) can't fit but can add fraction 2/6 *12 = 4 → bound = 0+10+18+4 = 32. So right.bound = 32.
- PQ contains nodes with bounds 38 (include-D) and 32 (exclude-D); extract highest (include-D).
- From include-D, branch level=2 (B):
    - Include B: weight=2+4=6, profit=10+10=20. Bound recompute: remaining cap 9, take A full (w9,p18) -> profit 38; bound 38. Update bestProfit = 20.

    - Exclude B: weight=2, profit=10. Bound: can take A (9) and C (6) fractionally: rem13, A 9->+18 rem4, C 6-> + (4/6)*12 = 8 → bound = 10+18+8 = 36. Insert nodes accordingly.

- PQ highest bound node is include-B (bound 38). Extract it:
    - level=3 (A):
        - Include A: weight=6+9=15, profit=20+18=38. Weight ≤15 ⇒ feasible; update bestProfit = 38.
        - Since bestProfit reached bound 38, no other node can exceed it; any node with bound ≤38 must be pruned if ≤ bestProfit.
    - Exclude A: weight=6, profit=20; its bound ≤ 38 — but bound ≤ bestProfit? bound recompute might be ≤38; since bestProfit=38, only nodes with bound >38 would be useful. None exceed 38.
- At this point bestProfit = 38 and root bound = 38 → optimal solution found.

Conclusion (explicit optimal set): include items A, B and D (weights 9+4+2 = 15 kg, profit 18+10+10 = 38 Rs).
No other feasible combination yields more than 38 (we can verify by enumeration).

Final answer for (a):

- Greedy-bound B&B algorithm (pseudocode above).
- Optimal feasible solution for the instance: Items A, B and D, total weight 15 kg, total profit 38 Rs.

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

1. Principle: Backtracking is a technique for solving constraint satisfaction / combinatorial search problems by building candidate solutions incrementally and abandoning a candidate ("backtracking") as soon as it is determined that the candidate cannot possibly be completed to a valid solution.

Key features:

- Depth-first search in solution space tree.
- Prune partial solutions early when they violate constraints (feasibility check).
- Often used for N-Queens, subset sum, graph coloring, permutations, exact cover.

2. Control abstraction

```
Backtrack(partial_solution, depth):
  if partial_solution is a complete solution:
```

```
  process_solution(partial_solution)
  return
 for candidate in candidates_for_next_position(partial_solution):
  if is_feasible(partial_solution + candidate):
   extend partial_solution with candidate
   Backtrack(partial_solution, depth+1)
   remove candidate from partial_solution   // undo (backtrack)
```

- candidates_for_next_position enumerates choices.
- is_feasible enforces constraints early (pruning).
- process_solution records or prints solutions or best objective value.

3. Example — Subset sum / 0–1 selection (simple)

Problem: Given numbers $\{3,34,4,12,5,2\}$, find subset that sums to target 9.

Backtracking steps:

- Start with first element 3: include (current sum 3) or exclude (sum 0).
- Recurse; at each node compute current sum; if current sum > target, prune; if equal, output solution and backtrack.
- This avoids exploring branches that would exceed the target.

4. Time analysis of control abstraction

- Backtracking explores a search tree with branching factor b (typical max choices per level) and depth d.
- Worst-case number of nodes is $O(b^d)$(exponential).
- Practical performance depends heavily on pruning: good feasibility checks can cut many branches.
- Space complexity is typically $O(d)$for recursion stack (plus space for partial solution).

Examples:

- N-Queens: b = n choices at top but pruning reduces average nodes significantly; worst-case still exponential.
- Subset sum (target small): pruning by sum reduces work; worst case exponential.

5. When to prefer backtracking

- When problem has strong constraints that allow early pruning.

- When we need all solutions or exact feasible solutions (not just an optimum).
- When n small/moderate (e.g., up to 30–40 depending on pruning).

Q4) a) What is Branch and Bound method? Write control abstraction for Least Cost search? [9]

What is Branch and Bound?

Branch and Bound (B&B) is a general algorithmic strategy used to solve optimization problems (minimization or maximization) by systematically exploring the solution space as a state-space tree.

Key ideas:

1. Branching:
   The problem is divided into smaller subproblems (children nodes) by making choices (include/exclude, assign/not assign, etc.).
2. Bounding:
   For each node, a bound is computed which represents the *best possible solution* obtainable from that node.
   - In minimization: bound = *lower bound*
   - In maximization: bound = *upper bound*
3. Pruning:
   If the bound at a node is worse than the best solution found so far, then that entire branch is discarded.
4. Search Strategy:
   - *Least-Cost (LC) branch and bound* uses a priority queue that always expands the live node having the smallest cost / lowest bound.

This reduces the number of nodes explored and improves efficiency compared to brute force.

Control Abstraction for Least-Cost (LC) Branch and Bound Search

LC_Branch_and_Bound():

1. Create a root node representing the initial state.
2. Compute its cost (bound) and insert it into a priority queue (PQ).
3. bestSolution ← ∞ // for minimization problems
4. while PQ is not empty do
5. node ← PQ.deleteMin() // choose lowest-cost node
6. if node.cost ≥ bestSolution then
7. continue // prune

8. endif
9. if node represents a complete solution then
10. if node.cost < bestSolution then
11. bestSolution ← node.cost
12. endif
13. continue
14. endif
15. Generate all children of node // Branch
16. For each child:
17. compute child.cost (bound)
18. if child.cost < bestSolution then
19. PQ.insert(child) // Live node
20. endif
21. endwhile
22. return bestSolution

Explanation of Control Steps:

- Steps 1–3: Initialize tree, PQ, and upper bound.
- Step 5: Expand node with least cost.
- Steps 6–7: Bounding and pruning.
- Steps 9–14: Update the best solution found so far.
- Steps 15–20: Generate live nodes (children) only if their bound is promising.

Q4) b) Explain the backtracking with graph coloring problem. Find solution for following graph

	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Adjacency matrix for graph G

You are given the adjacency matrix:

	C1	C2	C3	C4	C5
C1	0	1	0	1	0
C2	1	0	1	0	0
C3	0	1	0	1	1
C4	1	0	1	0	1
C5	0	0	1	1	0

This describes a graph with vertices C1–C5.

Backtracking Strategy (Explanation)

Graph coloring assigns a color to each vertex such that no two adjacent vertices share the same color.

Backtracking process:

1. Assign a color to vertex C1.
2. Move to the next vertex.
3. Try all allowed colors that do not violate adjacency constraints.
4. If a conflict occurs → backtrack and try next color.
5. Continue until all vertices are colored.

Control Abstraction for Backtracking (Graph Coloring)

```
ColorGraph(k):    // k = number of colors
1. function AssignColor(v):
2.   if v > n then
3.       print valid coloring
4.       return
5.   for color = 1 to k do
6.       if color is safe for vertex v then
7.           x[v] = color
8.           AssignColor(v+1)
9.           x[v] = 0    // backtrack
```

10. endfor

- safe(v,color) checks if any adjacent vertex already uses color.

Now Solve Given Graph (Use Minimum Colors)

Let colors = {1, 2, 3} (typical for 5-vertex graphs unless forced higher).

Step-by-step Backtracking Coloring:

Vertex C1

Assign:

$$C1 = 1$$

Vertex C2 (adjacent to C1)

C1 = 1 → must choose ≠ 1

$$C2 = 2$$

Vertex C3 (adjacent to C2 & C4 & C5)

Adjacent: C2=2
So available = {1,3}

Try:

$$C3 = 1$$

Vertex C4 (adjacent to C1, C3, C5)

Adjacent so far:

- C1 = 1
- C3 = 1

Cannot use color 1. Choose:

$$C4 = 2 \text{ (allowed)}$$

Vertex C5 (adjacent to C3, C4)

- C3 = 1
- C4 = 2
  So available = {3}

$$C5 = 3$$

Final Valid Coloring

$$C1 = 1, \quad C2 = 2, \quad C3 = 1, \quad C4 = 2, \quad C5 = 3$$

This uses 3 colors and satisfies all graph constraints.

► **MAY/JUNE 2023**

Q3) a) Explain with suitable example Backtracking: Principle, control abstraction, time analysis of control abstraction. [8]

1. Principle of Backtracking

Backtracking is a systematic search technique used for solving combinatorial and constraint-satisfaction problems.
It builds solutions incrementally, and whenever a partial solution violates constraints, it backtracks (undoes the last step) and tries another possibility.

Key ideas:

- DFS search in solution space.
- Early pruning when constraints fail → avoids exploring invalid branches.
- Used for N-Queens, Graph Coloring, Subset Sum, Hamiltonian cycle, etc.

2. Control Abstraction for Backtracking

Generic template:

Backtrack(x):

1. If x is a complete solution:
2. Output x
3. return
4. For each candidate c in choices(x):
5. If feasible(x, c):
6. Add c to partial solution
7. Backtrack(x)
8. Remove c from partial solution // backtrack

Where:

- choices(x) = all valid next values.
- feasible(x, c) = constraint check.

3. Time Analysis of Backtracking Control Abstraction

Let:

- b = branching factor (choices per step)
- d = depth (length of solution)

Worst-case time:

$$O(b^d)$$

→ exponential, because every combination is explored.

Actual runtime is usually much lower due to pruning.

Space complexity:

$$O(d)$$

(for recursion stack + partial solution).

4. Short Example — 4-Queens

- Try placing queen row by row.
- If placing at column c causes conflict → backtrack immediately.
- Only valid nodes explored → faster than brute force.

b) Compare between greedy method and dynamic programming with respect to. [9]

i) Feasibility **ii) Optimality** **iii) Recursion**
iv) Memorization **v) Time complexity**

Feature	Greedy	Dynamic Programming
i) Feasibility	Makes *locally optimal* choice at each step. No guarantee of full feasibility for all problems.	Always considers **all subproblems** → ensures feasibility through tables.
ii) Optimality	Optimal only when **greedy-choice property** + **optimal substructure** hold.	Always optimal if **optimal substructure** holds. No greedy choice required.
iii) Recursion	Usually no recursion; mostly iterative (Pick best → update).	Strongly based on recursive subproblem definition (bottom-up or memoized top-down).
iv) Memorization	No memory used; each step is independent.	Uses memoization / tabulation to store subproblem results.
v) Time Complexity	Very fast: Typically **O(n log n)** or **O(n)**.	Slower: typically **O(n²)** or **O(nW)** etc., depending on table size.

Q4) a) What is sum of subset problem? Solve sum of subset problem for following instance using backtracking approach. [8]

Input : set [] = {2, 3, 5, 6, 8, 10}, sum = 10

Given:
Set = {2, 3, 5, 6, 8, 10}, Target sum = 10
Find subsets whose sum = 10.

Backtracking Approach

For elements sorted: {2,3,5,6,8,10}

We maintain:

- current_sum
- remaining_sum
- selected[] array

Step-wise Execution

Start at index 1 = 2

- Include 2 → sum = 2
- Move to next

Include 3

- sum = 2 + 3 = 5
- Next

Include 5

- sum = 10 → solution found

$$\{2,3,5\}$$

Backtrack from 5…

Try next elements:

Include 2, Exclude 3:

Try 5 again → sum = 7
Try 6 → sum = 13 (prune)
Try 8 → sum = 15 (prune)
Try 10 → sum = 12 (prune)

Start new branch: Include 10

- sum = 10 → solution found

$$\{10\}$$

All solutions from backtracking

$$\{2,3,5\},\{10\}$$

b) What is Branch and Bound method? Write control abstraction for Least cost search? [9]

1. What is Branch and Bound?

Branch and Bound (B&B) is a problem-solving strategy for optimization where the state space is explored as a tree:

- Branch: generate subproblems (children nodes).
- Bound: estimate best possible solution (upper/lower bound).
- Prune: discard nodes whose bound is worse than current best.
- Least-Cost search: always expands the node with minimum bound/cost first using a priority queue.

Used for 0/1 knapsack, TSP, scheduling, assignment problems.

2. Control Abstraction for Least-Cost Search

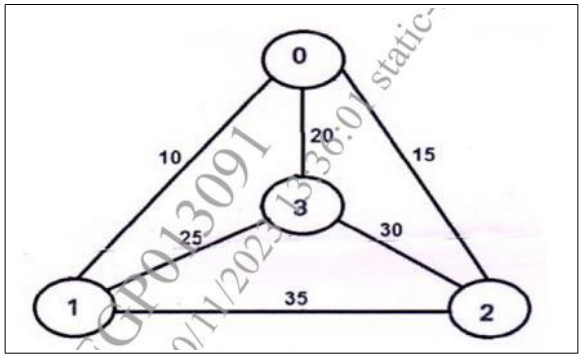
```
LC_Branch_and_Bound():
1. Create the root node (initial state)
2. Compute its bound (lower cost)
3. Insert into a priority queue PQ
4. best ← ∞     // minimization problem
5. while PQ is not empty do
6.      node ← PQ.deleteMin()   // expand least-cost node
7.      if node.bound ≥ best:
8.          continue        // prune
9.      if node is a complete solution:
10.         best ← min(best, node.cost)
11.         continue
12.     Generate children of node      // branching
13.     For each child:
14.         compute child.bound
15.         if child.bound < best:
16.             PQ.insert(child)
17. end while
18. return best
```

Explanation

- Bound ensures pruning of non-promising branches
- Priority queue ensures least-cost (best) node is expanded first
- Guarantees optimal solution when bound functions are correct

► **NOV/DEC 2023**

Q3) a) What is branch and bound algorithmic strategy? Apply branch n bound algorithmic strategy to solve traveling salesman problem for [9]



1. What is Branch and Bound (B&B)?

Branch and Bound is a systematic search strategy for solving hard optimization problems.
It explores the solution space using a state-space search tree, and prunes large parts of the tree using bounds.

Key Concepts

1. Branching
   – Split the problem into subproblems (partial paths in TSP).
2. Bounding
   – Compute a lower bound on the cost of any solution that can be generated from this node.
3. Pruning
   – If the bound ≥ best known solution → discard (prune).
4. Goal
   – Explore as few nodes as possible while guaranteeing optimality.

For TSP, the bound is computed using Reduced Cost Matrix.

2. Apply Branch and Bound to Given TSP Graph

Number the cities as per diagram:
0, 1, 2, 3

Edge Weights

Edge	Cost
0–1	10
0–2	15
0–3	20
1–2	35
1–3	25
2–3	30

We must find a minimum-cost Hamiltonian cycle starting at city 0.

STEP 1 — Construct Cost Matrix

	0	1	2	3
0	∞	10	15	20
1	10	∞	35	25
2	15	35	∞	30
3	20	25	30	∞

STEP 2 — Row Reduction

Row	Min	Reduce Row
0	10	[∞, 0, 5, 10]
1	10	[0, ∞, 25, 15]
2	15	[0, 20, ∞, 15]
3	20	[0, 5, 10, ∞]

Sum of row mins = 10 + 10 + 15 + 20 = 55

STEP 3 — Column Reduction

Column minimums = all 0
Total reduction = 55
This is the initial lower bound.

STEP 4 — Start branching from root node

We pick outgoing edges from city 0.

Branch: 0 → 1

To include 0→1:

- Set row 0 = ∞
- Set column 1 = ∞
- Set 1→0 = ∞ (prevent returning early)

Recompute reduced matrix and cost.

After reduction, lower bound becomes:

$$LB = 55 + \text{extra reductions} + 10 = 10 + 55 = 65$$

(10 is cost of edge 0→1)

Branch: 0 → 2

Similarly, cost:

$$LB = 55 + 15 = 70$$

Branch: 0 → 3

$$LB = 55 + 20 = 75$$

Choose smallest LB — branch: 0 → 1 (LB = 65)

STEP 5 — Continue branching from city 1

Next move choices:

1 → 2 (cost 35)
1 → 3 (cost 25)

Compute lower bounds:

(0 → 1 → 3)

Cost = 10 + 25 = 35, bound becomes approx:

$$LB = 65 + 25 = 90$$

(0 → 1 → 2)

Cost = 10 + 35 = 45, bound:

$$LB = 65 + 35 = 100$$

Minimum LB: (0 → 1 → 3)

STEP 6 — Branch from 3

Remaining city: 2

Path becomes:

$$0 \rightarrow 1 \rightarrow 3 \rightarrow 2$$

Cost so far:

- 0→1 = 10
- 1→3 = 25
- 3→2 = 30
  Total = 65

Return to 0: 2→0 = 15

Final tour cost:

$$65 + 15 = 80$$

FINAL ANSWER — Optimal TSP Tour

$$0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$$

Minimum Cost = 80

This is the optimal solution found by Branch and Bound.

Q3) b) Explain with suitable example Backtracking: Principle, control abstraction, time analysis of control abstraction.

1. Principle of Backtracking

Backtracking is a depth-first search technique for solving constraint-based problems.

You:

- build solution incrementally
- check feasibility at every step
- if partial solution violates constraints → backtrack

Used in:

- N-Queens
- Graph Coloring
- Subset Sum
- Hamiltonian cycle

2. Control Abstraction

Backtrack(x):

1. If x is a complete solution:

2. Output x

3. return

4. For each candidate c in choices(x):

5. If feasible(x,c):

6. Add c to x

7. Backtrack(x)

8. Remove c from x // backtrack

3. Time Analysis

Let:

- b = branching factor
- d = depth

Worst-case time:

$$O(b^d)\text{(exponential)}$$

Space:

$$O(d)$$

4. Example — 4-Queen

Place queens row by row.
If new queen conflicts → backtrack immediately.

Reduces search from 64 possibilities to only valid ones.

Q4) a) 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. [9]

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

1. What is Branch and Bound?

Branch and Bound (B&B) is an optimization strategy that explores the solution space using a state-space tree:

Steps involved:

1. Branching:
   Divide the problem into smaller subproblems (include/exclude an item).
2. Bounding:
   Compute an upper bound on the best possible solution that can be obtained from a node.
   - Usually use fractional knapsack bound.
3. Pruning:
   If a node's bound is ≤ current best solution, prune that node.
4. Search strategy:
   - Typically a best-first search using a priority queue ordered by highest bound.

2. Branch and Bound Algorithm for 0/1 Knapsack

Node Structure

Each node contains:

- level (index of item)
- profit
- weight
- bound

Upper Bound Calculation

bound(node):

```
  if node.weight > W: return 0


  bound_profit = node.profit
  total_weight = node.weight


  for next items in order of profit/weight:
     if total_weight + wi ≤ W:
        take whole item
     else:
```

take fractional part and break

return bound_profit

Algorithm (Best-first B&B)

Knapsack_BB():

1. Sort items by profit/weight ratio.

2. Create root node at level 0 with profit=0, weight=0.

3. Compute bound(root) and insert into PQ.

4. bestProfit = 0

5. while PQ not empty:

6. node = PQ.removeMax()

7. if node.bound ≤ bestProfit: continue // prune

8. Create left child (include next item)

9. If feasible, update bestProfit

10. Compute bound(left)

11. If bound(left) > bestProfit, insert into PQ

12. Create right child (exclude next item)

13. Compute bound(right)

14. If bound(right) > bestProfit, insert

15. end while

16. return bestProfit

3. Apply Algorithm to Given Knapsack Problem

Capacity = 15 kg

Item	A	B	C	D
Profit	18	10	12	10
Weight	9	4	6	2

Step 1 — Compute profit/weight ratio

Item	**p**	**w**	**p/w**
A	18	9	2.0

Item	p	w	p/w
B	10	4	2.5
C	12	6	2.0
D	10	2	5.0

Sort items:

$$D, \quad B, \quad A, \quad C$$

Explore using Branch & Bound

Start at Root Node

weight = 0, profit = 0

Using fractional knapsack:
D (2) → B (4) → A (9)

Bound ≈ (10 + 10 + 18) = 38

So: best possible = 38

Branch 1: Include D

profit = 10, weight = 2
Remaining capacity = 13
Take B (10), A (18) → bound still 38
bestProfit = 10

Branch 2: Include B

profit = 20, weight = 6
Remaining cap = 9
Take A → +18 = 38
bound = 38
bestProfit = 20

Branch 3: Include A

profit = 38, weight = 15
Valid (exact capacity)

Update:

$$bestProfit = 38$$

This is maximum possible (same as fractional bound), so this is optimal.

Q4) b) What is sum of subset problem? Solve sum of subset problem for following instance using backtracking approach [8]
Input: set[ ] = {2, 3, 5, 6, 8, 10}, sum = 10

1. What is Sum of Subset Problem?

Given:

- A set of positive integers
- A target sum S

Find all subsets whose elements sum to S.

It is solved using backtracking because:

- We try to include/exclude each element.
- Immediately prune paths where current sum > target.

2. Input

Set = {2, 3, 5, 6, 8, 10}
Sum = 10

3. Backtracking Solution Steps

Start with empty set

Remaining numbers: 2,3,5,6,8,10

1. Try 2

Current sum = 2

Try 3 → sum = 5

Try 5 → sum = 10 solution

$$\{2,3,5\}$$

Backtrack…

Try 6 → 11 (prune)

Try 8 → 13 (prune)

Try 10 → 15 (prune)

2. Try 3 without 2

sum = 3

Try 5 → sum = 8

Next = 6 → 14 (prune)

Try 8 → 11 (prune)

Try 10 → 13 (prune)

3. Try 5 alone

sum = 5

Try 6 → 11 (prune)

Try 8 → 13

Try 10 → 15

4. Try 6 alone

sum = 6

Try 8 → 14

Try 10 → 16

5. Try 8 alone

sum = 8
Next = 10 → 18 (prune)

6. Try 10 alone

sum = 10 solution

$$\{10\}$$

Final Subset Sum Solutions:

$$\{2,3,5\},\{10\}$$

- **MAY/JUNE 2024**

Q3) a) Assume that a graph with n vertices is represented by an adjacency matrix G. Let there be "m" number of colours available. Write a recursive backtracking algorithm to colour all the vertices of the graph. What is the time complexity of this algorithm? [8]

Problem: Graph with $n$vertices represented by adjacency matrix G[1..n][1..n] (G[i][j]=1 if edge i - j exists). We have m colors labelled 1..m. Colour every vertex so adjacent vertices have different colours.

Key helper: feasibility test

isSafe(v, color, colorArr) — returns true iff for every u with G[v][u] = 1 we have colorArr[u] ≠ color.

Complexity of isSafe is $O(n)$(scans row v).

Recursive backtracking (pseudocode)

```
procedure GRAPH_COLORING(G[1..n][1..n], m)
   color[1..n] ← 0      // 0 = unassigned
   if COLOR_VERTEX(1) then
      return color      // success: a valid colouring
   else
      return "No solution with m colors"


function COLOR_VERTEX(v):
   if v > n:
      return true      // all vertices coloured successfully
   for c in 1..m:
      if isSafe(v, c, color):       // check adjacency constraints
         color[v] ← c
         if COLOR_VERTEX(v+1) then
            return true
         color[v] ← 0             // backtrack
   return false                    // no color possible for vertex v
isSafe(v,c,color):
for u = 1 to n:
   if G[v][u] == 1 and color[u] == c:
      return false
return true
```

Time complexity

- At each vertex we try up to m colours.
- Depth of recursion is n.
- Worst-case number of leaves (assignments tried) is m^n.
- Each assignment attempt invokes isSafe costing $O(n)$.

So worst-case time complexity is

$$O(n \cdot m^n)$$

(space complexity $O(n)$ for recursion/colour array).

(Pruning due to adjacency check often dramatically reduces actual work for sparse graphs or small m.)

b) Consider three items along with respective weights and value as [9]

	Weight	Value	Value/Weight
O1	5	6	6/5 = 1.2
O2	4	5	5/4 = 1.25
O3	3	4	4/3 = 1.3

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

Given

Three items (I'll label them 1..3 as given):

- O1: weight 5, value 6, ratio $= 1.2$
- O2: weight 4, value 5, ratio $= 1.25$
- O3: weight 3, value 4, ratio $= 1.333$
  Capacity $M = 7$.

Step 1 — sort by value/weight (descending)

Order: O3 (r≈1.333), O2 (1.25), O1 (1.2).
We will consider items in that order (index them accordingly).

Step 2 — bounding function (fractional knapsack upper bound)

For a partial node with current weight Wc and profit Pc, the bound is obtained by greedily filling remaining capacity with fractional parts of remaining items (in ratio order). Bound is an upper bound on achievable profit from that node.

Step 3 — best-first (priority queue by bound). Walkthrough:

Root node: level 0 (no items), Wc=0, Pc=0.
Compute fractional bound: take O3 (w3=3, v3=4) → rem cap 4, take O2 (4,5) → rem 0. Bound = 4+5 = 9.
Initial bestProfit = 0. PQ: {root(bound=9)}.

Expand root (highest bound = 9)
Consider branching on item O3 (first item):

- Left child (include O3):

    - Wc = 3, Pc = 4.
    - Bound: remaining cap = 4 → can take O2 fully (v=5) → bound = 4 + 5 = 9.
    - Insert node (bound 9). Update bestProfit? Pc=4 < bestProfit (0) → set bestProfit = 4 (improvement).
- Right child (exclude O3):
    - Wc = 0, Pc = 0.
    - Bound: can take O2 (4,5) → rem 3, take fraction of O1 (3/5 of weight 5) → +6*(3/5)=3.6 → bound = 5 + 3.6 = 8.6.
    - Insert node (bound 8.6).

PQ now: include-O3 (b=9), exclude-O3 (b=8.6).

Expand include-O3 (b=9) — branch on next item O2:

- Include O2 (from include-O3):
    - Wc = 3+4 = 7, Pc = 4+5 = 9. Feasible (exact capacity).
    - Bound for this node (fractional) = Pc = 9.
    - Update bestProfit ← max(4, 9) = 9.
- Exclude O2 (from include-O3):
    - Wc = 3, Pc = 4.
    - Remaining cap = 4 → can take fraction of O1 (4/5 of O1) → +6*(4/5)=4.8 → bound = 4 + 4.8 = 8.8.
    - Since bound 8.8 < current bestProfit 9, prune this node (do not insert).

At this point bestProfit = 9. Note root bound was 9, so no node can exceed 9; and we have a feasible solution that attains 9 → optimal.

Conclusion: Optimal selection is include O3 and O2 (weights 3+4 = 7) with total value = 9.

Final answer:

- **Optimal items: O3 and O2**
- **Total weight =** $3+4=7$**(fits capacity)**
- **Optimal profit = 9**

Q4) a) Compare backtracking with branch and bound method with respect to: search technique, exploration of state space tree and king of problems that can be solved. [8]

Aspect	Backtracking	Branch & Bound
Search technique	Depth-first systematic search that tries choices one-by-one and *immediately* abandons a partial solution when a constraint is violated (feasibility pruning).	Best-first or depth-first guided search that uses bounds (upper/lower) to prune nodes that cannot improve current incumbent. Often uses a priority queue (best-first).
Exploration of state-space tree	Explores nodes in DFS order; when a node is infeasible it backtracks to try next sibling. Pruning is based on *feasibility* only. May explore many nodes if pruning weak.	Explores nodes according to bound values (best-first) or depth-first with bounding. Uses optimistic bounds to prune whole subtrees even if partial solutions are feasible. Pruning is potentially stronger because it discards branches that cannot lead to better objective.
Kind of problems suited	Constraint satisfaction, enumeration, decision problems: e.g., N-Queens, graph coloring, subset-sum, permutations, SAT. Best when constraints allow early pruning and we need *all* or *first* feasible solutions.	Optimization problems where we want an optimal value: e.g., 0/1 knapsack, TSP, assignment problems, integer programming. Works when a reliable bound (relaxation) can be computed (e.g., fractional knapsack, reduced cost matrix).
Goal	Find feasible solutions (often all or first), or check existence.	Find optimal solution (minimization or maximization) while pruning suboptimal branches.
Typical pruning basis	Feasibility only (constraint violation).	Bounding (best possible objective from node) compared to incumbent; also feasibility.
Complexity behaviour	Worst-case exponential; effective when constraints prune heavily.	Worst-case exponential, but often prunes more than backtracking for optimization problems if good bounds exist.

b) Consider set A of five numbers {5,10,15,20.25}. We wish to find the subset of A such that sum of the numbers in this subject is equal to 30. Solve this problem to find the first solution using backtracking approach. Show space tree being created. [9]

1. Backtracking strategy (include-first)

We process elements in the given order. At each element x we try:

- Include x (if current_sum + x ≤ target), then recurse.
- If include branch fails to give a solution, exclude x and recurse.

We stop when we encounter the *first* subset whose sum = 30.

Pseudocode:

```
Backtrack(i, current_sum, solution):
 if current_sum == target:
  output solution and STOP (first solution found)
 if i > n or current_sum > target:
  return  // dead end
 // Try include
 solution.push(A[i])
 Backtrack(i+1, current_sum + A[i], solution)
 solution.pop()
 // Try exclude
 Backtrack(i+1, current_sum, solution)
```

We follow include-first order, which finds smallest-index combinations first.

2. Run on A = [5,10,15,20,25], target = 30

Walkthrough (include-first):

1. Start: i=1 (value 5), sum=0
    - Include 5 → sum=5, solution {5}
2. i=2 (value 10)
    - Include 10 → sum=15, solution {5,10}
3. i=3 (value 15)
    - Include 15 → sum=30, solution {5,10,15}
      → Found target. Stop and report this as the first solution.

So the first solution encountered is: {5, 10, 15}

3. Space-tree showing nodes up to the found solution

I'll show the explicit binary tree (I = include, E = exclude). Each node labelled as (index, sum) and the chosen set so far. We stop at the node where sum=30.

Root: (i=1, sum=0) {}

├─ I (include 5): (i=2, sum=5) {5}

│ ├─ I (include 10): (i=3, sum=15) {5,10}

│ │ ├─ I (include 15): (i=4, sum=30) {5,10,15} <-- FOUND (stop)

│ │ └─ E (exclude 15): (i=4, sum=15) {5,10}

│ └─ E (exclude 10): (i=3, sum=5) {5}

│ ├─ I (include 15): (i=4, sum=20) {5,15}

│ └─ E (exclude 15): (i=4, sum=5) {5}

└─ E (exclude 5): (i=2, sum=0) {}

├─ I (include 10): (i=3, sum=10) {10}

│ ├─ I (include 15): (i=4, sum=25) {10,15}

│ └─ E (exclude 15): (i=4, sum=10) {10}

└─ E (exclude 10): (i=3, sum=0) { }

├─ I (include 15): (i=4, sum=15) {15}

└─ E (exclude 15): (i=4, sum=0) {}

Only the leftmost path (Include 5 → Include 10 → Include 15) is expanded until we find the sum 30; we stop immediately upon finding it.

4. Observations

- First solution depends on traversal order. Include-first with the items in given order produced {5,10,15}. A different order (e.g., descending) may find a different first solution.
- Time to first solution in this run is very small: we only explored the leftmost branch to depth 3.
- Worst-case complexity of subset-sum backtracking: $O(2^n)$(exponential), but pruning by current_sum > target reduces work in practice.

► **NOV/DEC 2024**

Q3) a) Consider a graph represented by the adjacency matrix given below:

	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

Use a recursive backtracking algorithm to colour this graph with three colours R,G,B

(3 colours: R, G, B)
Graph Vertices: A, B, C, D, E, F, G
Colors available: {R, G, B}

Adjacency matrix shows the following edges clearly:

- A connected to B, C
- B connected to A, D, E
- C connected to A, F, G
- D connected to B
- E connected to B
- F connected to C
- G connected to C

We color in alphabetical order: A, B, C, D, E, F, G

Backtracking Algorithm (Recursive)

```
COLORS = {R, G, B}

color[A..G] = empty

function isSafe(v, c):
   for each u adjacent to v:
      if color[u] == c:
         return false
   return true

function colorGraph(v):
   if v > G:
```

```
    print color[]
    return true
  for each color c in COLORS:
    if isSafe(v, c):
      color[v] = c
      if colorGraph(next vertex):
        return true
      color[v] = empty   // backtrack
  return false
```

Step-by-Step Backtracking Coloring

1. Vertex A

A has no colored neighbors yet
→ Assign A = R

2. Vertex B

Neighbors of B: A, D, E
A = R → cannot use R
Remaining: G, B
→ Assign B = G

3. Vertex C

Neighbors: A, F, G
A = R → cannot use R
Choose C = G (but we must check conflict later)
Valid for now because B has G but B not adjacent to C.

So C = G

4. Vertex D

Neighbor: B
B = G → cannot use G
Possible: R or B
Pick first available → D = R

5. Vertex E

Neighbor: B
B = G → cannot use G
Choose first allowed → E = R

6. Vertex F

Neighbor: C
C = G → cannot use G
Pick F = R

7. Vertex G

Neighbor: C
C = G → cannot use G
Pick G = R

Final Valid Graph Coloring

$$A = R, \quad B = G, \quad C = G, \quad D = R, \quad E = R, \quad F = R, \quad G = R$$

This uses only 2 colours, but allowed up to 3.

All adjacency constraints satisfied
No two adjacent vertices share same color

Q3(b) Consider three items along with respective weights and value as

	Weight	Value
O_1	10	12
O_2	8	10
O_3	6	8

Assume the Knapsack capacity m = 14. Solve this 0/1 Knapsack problem using LC branch and bound method.

Items:

Item	Weight	Value
O1	10	12
O2	8	10
O3	6	8

Knapsack capacity: 14

Step-1 — Compute value/weight ratio

Item	w	v	v/w
O1	10	12	1.2
O2	8	10	1.25

Item	w	v	v/w
O3	6	8	1.33

Sort items by ratio (descending):

1. O3
2. O2
3. O1

Step-2 — Bounding (fractional knapsack upper bound)

Root node:

weight = 0, value = 0
Fill capacity 14 greedily:

- Take O3 (6,8) → rem = 8
- Take O2 (8,10) → rem = 0
  Total bound = 18

So upper bound = 18

bestProfit = 0

Step-3 — Explore nodes (Best-first B&B)

Branch: include O3

value = 8
weight = 6
remaining cap = 8
Can add O2 fully → +10 → bound = 18

bestProfit = 8

PQ contains:
INCLUDE(O3) bound=18
EXCLUDE(O3) bound=16 (explained later)

Expand best node → include O3.

Branch: include O2

value = 8 + 10 = 18
weight = 6 + 8 = 14 (exact capacity)

This is feasible and optimal because:

- bound from root = 18

- we reached profit = 18
- cannot exceed 18

So optimal solution reached.

FINAL Optimal Set

Take O3 and O2

Weights:
6 + 8 = 14 ≤ 14 OK

Profit:
8 + 10 = 18

So optimal profit = 18

Q4) a) We have a salesman who needs to visit four cities (A, B, C, D) and return to the starting city. The distances between these cities are as follows :
Distance from A to B: 10 units
Distance from A to C: 15 units
Distance from A to D: 20 units
Distance from B to C: 35 units
Distance from B to D: 25 units
Distance from C to D: 30 units
Find the shortest possible route that visits each city exactly once and returns to the starting city. Use branch and bound method to find the optimum route for traveling salesman, assume A as a starting point of the tour. [8]

Given distances

- A–B = 10
- A–C = 15
- A–D = 20
- B–C = 35
- B–D = 25
- C–D = 30

We label cities: A = 0, B = 1, C = 2, D = 3. The full cost matrix:

	A	B	C	D
A	∞	10	15	20
B	10	∞	35	25

C 15 35 ∞ 30

D 20 25 30 ∞

Step 1 — Initial lower bound (matrix reduction)

- Row minima: A→10, B→10, C→15, D→20 → sum = 55.
- After row and column reductions columns have zeros → initial LB = 55.

This means any tour cost ≥ 55.

Step 2 — Branch on first move from A

Compute LB for each choice A→x by adding the edge cost to the reduced-matrix bound (we show a compact greedy-style bound):

- A → B: add edge cost 10 → LB ≈ 55 + 10 = 65.
- A → C: add edge cost 15 → LB ≈ 55 + 15 = 70.
- A → D: add edge cost 20 → LB ≈ 55 + 20 = 75.

So expand the smallest bound first → A → B (LB = 65).

Step 3 — From partial path A → B (cost so far = 10)

Remaining cities: C, D. Branch on B→C or B→D.

- Path A → B → C: partial cost = 10 + 35 = 45. Completing remaining edges minimally gives LB ≈ 45 + (smallest possible return edge) ≈ 100 (worse).
- Path A → B → D: partial cost = 10 + 25 = 35. Completing the tour by best remaining edges (D→C then C→A) yields actual feasible tour cost:
    - A→B = 10
    - B→D = 25
    - D→C = 30
    - C→A = 15
      Total = 10 + 25 + 30 + 15 = 80.

This gives a feasible solution cost = 80 and a node bound that led to it ≤ previously computed bounds. Update incumbent best = 80.

Step 4 — Check other branches quickly

Other branches (A→C or A→D) had LBs 70 and 75; they could still produce solutions but any node whose lower bound ≥ current best (80) can be pruned. Exploring shows other full tours produce costs 95 or 80 as well; none go below 80.

Conclusion (optimal)

One optimal tour (there can be symmetric variants) is:

$$A \to B \to D \to C \to A$$

with minimum total distance = 80 units.

(Another equivalent optimal order is A → C → D → B → A, cost 80 as well — they are mirror permutations.)

b) Write a short note on LC branch and bound method. [5]

- **Idea**: LC B&B always expands the live node with the smallest lower bound (for minimization problems). Use a priority queue keyed by node bound.
- **Node info**: each node stores partial solution, cost so far, weight/state, and a computed lower bound on best possible completion.
- **Bounding**: use a relaxation of the problem (e.g., fractional knapsack for 0/1 knapsack, reduced cost matrix for TSP) to compute an optimistic bound.
- **Process**: extract min-bound node → branch → compute bounds for children → insert children whose bound < incumbent into PQ → update incumbent when a full feasible solution is found.
- **Advantages**: often visits far fewer nodes than brute force; finds optimal solution and can be stopped early with best-known solution (useful for anytime behavior).
- **Complexity**: worst-case exponential; performance depends on quality of bound and branching order.

c) What are the drawbacks of branch and bound method? [4]

1. **Worst-case exponential time**: B&B does not avoid exponential explosion in the worst case; pruning helps but cannot guarantee polynomial time.
2. **High memory usage**: Best-first/B&B often stores many live nodes in a priority queue → large memory requirements.
3. **Bound quality dependent**: Efficiency depends heavily on how tight/cheap the bound is; weak bounds cause little pruning.
4. **Overhead**: computing bounds and managing PQ adds non-trivial overhead; for small instances simple heuristics can be faster.

NOTE: Please verify all answers before referring.