UNIT–4 : Dynamic Programming, Backtracking & Branch and Bound (PYQ Solution)

AKTU · Design and Analysis of Algorithm (BCS503) · Solutions by **Chirag Jain**, Ycotes

SECTION A Answers (≈70 Words Each)

Q1. With a suitable example explain "Branch and Bound". Explain the Branch and Bound method in brief. (Jan 2025, 2022)

Branch and Bound is a technique for optimisation problems that explores the state-space tree of possible solutions. Branching splits a problem into sub-problems; bounding computes a lower bound (for minimisation) on the best solution inside each node. A node whose bound is not better than the best complete solution found so far is pruned. Nodes are expanded by the least bound first (best-first, using a priority queue).

Example: in the travelling salesman problem, the reduced cost matrix gives a lower bound for each partial tour, and partial tours with bounds above the best known tour are discarded. It is also used for 0/1 knapsack and job assignment.

Q2. In a Red-Black tree, how many black nodes are there on any path from the root to a leaf if the black height is 3? (Jan 2026)

The black height bh(x) of a node is the number of black nodes on any path from x down to a leaf (NIL), not counting x itself. Property 5 of a red-black tree says this number is the same on every such path.

Hence if the black height of the root is 3, every path from the root to a leaf contains exactly 3 black nodes (counting the black NIL leaf, and excluding the root). If the root is counted too (it is always black), the count is 4. The height of such a tree is at most 2·bh = 6 (excluding NIL).

Q3. Write down the Floyd-Warshall algorithm. (2022)

The Floyd-Warshall algorithm finds shortest paths between all pairs of vertices using dynamic programming, with $D^{(0)}$ equal to the weight matrix.

```
FLOYD-WARSHALL(W, n)
1  D = W
2  for k = 1 to n
3      for i = 1 to n
4          for j = 1 to n
5              D[i][j] = min(D[i][j], D[i][k] + D[k][j])
6  return D
```

At step k, paths may use vertices 1..k as intermediates. Running time $\Theta(n^3)$, space $\Theta(n^2)$. It allows negative edges but no negative cycles.

Q4. Define the term "Graph Colouring". (Jan 2024)

Graph colouring is the assignment of colours to the vertices of a graph so that no two adjacent vertices have the same colour. The m-colouring problem asks whether a graph can be coloured with at most m colours; the smallest such m is the chromatic number $\chi(G)$.

Example: a triangle needs 3 colours, a bipartite graph needs 2. Uses: map colouring, exam timetable scheduling, register allocation. It is NP-complete and is solved by backtracking.

SECTION B Answers (≈250 Words Each)

Q1. Write Floyd-Warshall's algorithm to find all-pairs shortest paths in a graph. Discuss its time complexity. (Jan 2025, 2022)

The **Floyd-Warshall algorithm** is a dynamic programming method for the all-pairs shortest path problem on a weighted directed graph (negative edges allowed, no negative cycles).

Idea: let d_ij^(k) be the shortest path length from i to j using only vertices {1, …, k} as intermediate vertices. Then

- d_ij^(0) = w_ij (0 on the diagonal, ∞ if there is no edge);
- d_ij^(k) = min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1)).

Either the best path avoids vertex k, or it goes i → k → j.

```
FLOYD-WARSHALL(W)
1  n = rows[W]
2  D(0) = W
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              d[i][j](k) = min(d[i][j](k-1), d[i][k](k-1) + d[k][j](k-1))
7  return D(n)
```

To recover paths, a predecessor matrix π is updated whenever a shorter route through k is found.

Time complexity: three nested loops, each running n times, with O(1) work inside: **Θ(n³)**. Space is Θ(n²) if one matrix is updated in place (Θ(n³) if every D^(k) is stored). Compared with running Dijkstra from every vertex (O(V·E log V)), it is simpler and works with negative edges.

Transitive closure is a related use: replace min and + by OR and AND.

Q2. Explain Merge Sort with an example and compute its time complexity. (Jan 2026)

Merge sort is a stable divide and conquer sorting algorithm: **divide** the array into two halves, **conquer** by sorting each half recursively, **combine** by merging the two sorted halves.

Example: sort A = [12, 4, 7, 9, 1, 3].

- Divide: [12, 4, 7] | [9, 1, 3] → [12, 4] [7] | [9, 1] [3] → [12] [4] [7] [9] [1] [3].
- Merge: [12] + [4] → [4, 12]; [4, 12] + [7] → [4, 7, 12]; [9] + [1] → [1, 9]; [1, 9] + [3] → [1, 3, 9].
- Final merge of [4, 7, 12] and [1, 3, 9]: 1 < 4 → 1; 3 < 4 → 3; 4 < 9 → 4; 7 < 9 → 7; 9 < 12 → 9; then 12.
- **Result: [1, 3, 4, 7, 9, 12].**



Merge sort steps

Time complexity: dividing takes O(1), merging n elements takes Θ(n). Hence T(n) = 2T(n/2) + Θ(n). The recursion tree has $\log_2 n + 1$ levels with total work cn at every level, so T(n) = cn $\log_2 n$ + cn = **Θ(n log n)** for best, average and worst cases (Master theorem, case 2). The auxiliary space is O(n).

Q3. Apply the Floyd-Warshall algorithm to find shortest paths for the directed graph with vertices {a, b, c, d, e} and edges a→b 3, a→e −4, d→a 2, b→e 7, b→d 1, e→d 6, d→c −5, c→b 8. (Jan 2024)

Order the vertices a, b, c, d, e. Initial matrix D^0 (∞ = no edge):

D^0	a	b	c	d	e
a	0	3	∞	∞	−4
b	∞	0	∞	1	7
c	∞	8	0	∞	∞
d	2	∞	−5	0	∞
e	∞	∞	∞	6	0

k = a: only d reaches a (d→a = 2): d→b = 2 + 3 = 5, d→e = 2 − 4 = −2.
D^1: row d = (2, 5, −5, 0, −2); other rows unchanged.

k = b: into b: a (3), c (8), d (5). From b: d (1), e (7). Updates: a→d = 3 + 1 = 4; c→d = 8 + 1 = 9; c→e = 8 + 7 = 15; (a→e stays −4; d→e stays −2).
D^2: a = (0, 3, ∞, 4, −4); c = (∞, 8, 0, 9, 15).

k = c: into c: only d (−5). From c: b (8), d (9), e (15). Updates: d→b = −5 + 8 = 3 (< 5).
D^3: d = (2, 3, −5, 0, −2).

k = d: into d: a (4), b (1), c (9), e (6). From d: a (2), b (3), c (−5), e (−2). Updates: a→c = 4 − 5 = −1; b→a = 1 + 2 = 3; b→c = 1 − 5 = −4; b→e = 1 − 2 = −1; c→a = 9 + 2 = 11; c→e = 9 − 2 = 7; e→a = 6 + 2 = 8; e→b = 6 + 3 = 9; e→c = 6 − 5 = 1.

D^4	a	b	c	d	e
a	0	3	−1	4	−4
b	3	0	−4	1	−1
c	11	8	0	9	7
d	2	3	−5	0	−2
e	8	9	1	6	0

k = e: into e: a (−4), b (−1), c (7), d (−2). From e: a (8), b (9), c (1), d (6). Updates: a→c = −4 + 1 = −3 (< −1); a→d = −4 + 6 = 2 (< 4). No other change (e.g. b→a: −1 + 8 = 7 > 3).

Final matrix D^5 (all-pairs shortest distances):

D^5	a	b	c	d	e
a	0	3	−3	2	−4
b	3	0	−4	1	−1
c	11	8	0	9	7
d	2	3	−5	0	−2
e	8	9	1	6	0

Diagonal entries are all 0, so there is no negative cycle. Example path: a→e→d→c has length −4 + 6 − 5 = −3 ✓.

SECTION C Answers (≈500 Words Each)

Q1. Illustrate the N-Queens problem. Draw the state space tree for the 4-queens problem using backtracking. (Jan 2025, 2022)

The **N-Queens problem** asks us to place N queens on an N × N chessboard so that **no two queens attack each other**: no two share a row, a column or a diagonal. For 4 queens, one queen goes in each row, so we only choose the column x_i of the queen in row i.

Constraint (safe test): queens at (i, x_i) and (j, x_j) attack if x_i = x_j (same column) or |i − j| = |x_i − x_j| (same diagonal).

Backtracking: place queens row by row; for each row try columns 1 to N; if a column is safe, go to the next row; if no column is safe, **backtrack** to the previous row and move that queen. The state space tree has the partial placements as nodes; unsafe nodes are cut off (bounded).

State space tree for N = 4 (a node (r, c) means queen of row r in column c; × = no safe column, dead end):

```
Root
├─ Q1=1
│   ├─ Q2=3 → Q3: columns 1,2,3,4 all attacked  ×
│   └─ Q2=4 → Q3=2 → Q4: no safe column  ×
│             (Q3=3 attacked)
├─ Q1=2
│   └─ Q2=4 → Q3=1 → Q4=3  ✔ SOLUTION (2, 4, 1, 3)
├─ Q1=3
│   └─ Q2=1 → Q3=4 → Q4=2  ✔ SOLUTION (3, 1, 4, 2)
└─ Q1=4
    ├─ Q2=1 → Q3=3 → Q4: no safe column  ×
    └─ Q2=2 → Q3: no safe column  ×
```

Other children of Q1 = 2 and 3 (e.g. Q2 = 1, 2, 3 under Q1 = 2) are rejected immediately because they are in the same column or diagonal.

Step-by-step for Q1 = 1: row 2: column 1 (same column) ✗, column 2 (diagonal) ✗, column 3 ✓. Row 3: column 1 ✗ (col), 2 ✗ (diagonal with row 2), 3 ✗ (col), 4 ✗ (diagonal) → backtrack; Q2 moves to column 4. Row 3: column 2 ✓. Row 4: columns 1, 2, 3, 4 all attacked → backtrack; Q3 has no more columns, Q2 has none, so Q1 moves to column 2.

The two solutions (Q = queen):

```
. Q . .     . . Q .
. . . Q     Q . . .
Q . . .     . . . Q
. . Q .     . Q . .
```

These are mirror images of each other, so 4-queens has exactly **2 solutions**. For N = 8 there are 92.

Complexity: worst case O(N!) but pruning reduces the search greatly.

Q2. Find the optimal solution of the 0/1 Knapsack instance with n = 4 and capacity m = 8, where P = {1, 2, 5, 6} and W = {2, 3, 4, 5}. (Jan 2025)

Use dynamic programming: V[i, c] = max(V[i−1, c], p_i + V[i−1, c − w_i]) if w_i ≤ c, else V[i−1, c]; V[0, c] = 0.

i \ c	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (p1, w2)	0	0	1	1	1	1	1	1	1
2 (p2, w3)	0	0	1	2	2	3	3	3	3
3 (p5, w4)	0	0	1	2	5	5	6	7	7
4 (p6, w5)	0	0	1	2	5	6	6	7	8

Sample entries:

- V[2, 5] = max(V[1,5], 2 + V[1,2]) = max(1, 3) = 3.
- V[3, 7] = max(V[2,7], 5 + V[2,3]) = max(3, 7) = 7.
- V[3, 8] = max(V[2,8], 5 + V[2,4]) = max(3, 7) = 7.
- V[4, 8] = max(V[3,8], 6 + V[3,3]) = max(7, 8) = 8.

Maximum profit = V[4, 8] = 8.

Backtracking: V[4,8] = 8 ≠ V[3,8] = 7 → item 4 taken, c = 8 − 5 = 3. V[3,3] = 2 = V[2,3] → item 3 not taken. V[2,3] = 2 ≠ V[1,3] = 1 → item 2 taken, c = 0. Stop.

Selected items: 2 and 4, x = (0, 1, 0, 1), weight 3 + 5 = 8, profit 2 + 6 = **8**.

Check: items 3 + 1 give weight 6, profit 6; items 1 + 4 give weight 7, profit 7; item 3 alone gives 5; so 8 is the maximum. Time O(nm).

Q3. Explain the Branch and Bound approach for the Travelling Salesman Problem (TSP) using the "Reduced Cost Matrix" method. (Jan 2026)

In the **TSP**, a salesman must visit each of n cities exactly once and return to the start with minimum total cost. **Branch and Bound** searches a state-space tree of partial tours and uses the **reduced cost matrix** to get a lower bound at every node.

Reduction. A matrix is *reduced* if every row and every column contains at least one 0 (and no negative entries). To reduce:

1. Subtract the minimum of each row from that row (skip rows whose minimum is ∞ or 0).
2. Subtract the minimum of each column from that column.
3. The **reduction cost r** is the sum of all subtracted values. Any tour must pay at least r, so r is a lower bound.

Root node: reduce the original matrix; cost(root) = r.

Branching: for a node with matrix A reached at city i, generate a child for each unvisited city j (edge i → j):

1. Set all entries of row i and column j to ∞.
2. Set A[j][1] = ∞ (so we cannot return to the start early).
3. Reduce the resulting matrix, with cost r′.
4. **cost(child) = cost(parent) + A[i][j] + r′.**

Bounding and selection: keep all live nodes in a priority queue and always expand the node of the **least cost**. When a complete tour is reached, its cost becomes the upper bound; nodes whose cost is ≥ the best tour are **pruned**.

Why it works: the cost of a node is the least cost that any tour through the chosen partial path can possibly have, so the first complete tour popped from the queue with the smallest bound is optimal.

Complexity: worst case O(n!) (same as brute force), but pruning makes it much faster in practice. Each node needs O(n²) work for reduction. The solved examples below (2022 and Jan 2024 questions) show the method in action.

Q4. Given a 3 × 3 cost matrix, calculate the initial "Lower Bound" using the row-reduction method. (Jan 2026)

No matrix is printed in the question, so the method is shown on a sample cost matrix (∞ on the diagonal):

	1	2	3
1	∞	20	30
2	10	∞	25
3	15	35	∞

Row reduction:

- Row 1 minimum = 20, subtract → (∞, 0, 10).
- Row 2 minimum = 10, subtract → (0, ∞, 15).
- Row 3 minimum = 15, subtract → (0, 20, ∞).
- Sum of row reductions = 20 + 10 + 15 = **45**.

Matrix after row reduction: (∞, 0, 10), (0, ∞, 15), (0, 20, ∞).

Column reduction (finishing the reduction): column 1 minimum 0, column 2 minimum 0, column 3 minimum 10 → subtract 10 from column 3. Extra cost = 10.

Initial lower bound = 45 + 10 = 55. If only rows are reduced, as the question says, the bound is 45.

Check against actual tours: 1→2→3→1 = 20 + 25 + 15 = 60 and 1→3→2→1 = 30 + 35 + 10 = 75. Both are ≥ 55, so the bound is valid, and the optimum 60 lies above it.

Q5. Write down the Bellman-Ford algorithm to solve the single-source shortest path problem and write its time complexity. (2022)

Bellman-Ford computes shortest paths from a source s in a weighted directed graph whose edges may be **negative**, and detects a **negative-weight cycle** reachable from s. It relaxes every edge V − 1 times.

```
BELLMAN-FORD(G, w, s)
1  for each vertex v: d[v] = ∞, parent[v] = NIL
2  d[s] = 0
3  for i = 1 to |V| − 1
4      for each edge (u, v) in E
5          if d[u] + w(u, v) < d[v]
6              d[v] = d[u] + w(u, v); parent[v] = u
7  for each edge (u, v) in E          // check for negative cycle
8      if d[u] + w(u, v) < d[v]: return FALSE
9  return TRUE
```

Why V − 1 passes: a shortest path has at most V − 1 edges, and after pass i every shortest path with at most i edges is correct. If an edge can still be relaxed after V − 1 passes, a negative cycle exists.

Example. Edges: s→t 6, s→y 7, t→x 5, t→y 8, t→z −4, x→t −2, y→x −3, y→z 9, z→s 2, z→x 7. Source s, edge order as listed with (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,s), (z,x), (s,t), (s,y).

After pass	t	x	y	z
1	6	∞	7	∞
2	6	4	7	2
3	2	4	7	2
4	2	4	7	−2

Final distances: t = 2, x = 4, y = 7, z = −2 (s→y→x→t→z). The check pass finds no further relaxation, so there is no negative cycle.

Time complexity: V − 1 passes over E edges give **O(V·E)**; space O(V). It is slower than Dijkstra's O((V + E) log V), but handles negative weights.

Q6. What is the travelling salesman problem (TSP)? Find the solution of the following TSP using the Branch and Bound method: cost matrix rows (0 20 30 10 11), (15 0 16 4 2), (3 5 0 2 4), (19 6 18 0 3), (16 4 7 16 0). (2022)

The **TSP** asks for the shortest closed tour visiting each of n cities exactly once and returning to the start. It is NP-hard; brute force needs (n − 1)! tours.

Replace the diagonal by ∞. Cities 1 to 5.

Root: reduce the matrix.

- Row minima: 10, 2, 2, 3, 4 → total 21.
- After row reduction, column minima: 1, 0, 3, 0, 0 → total 4.
- **Lower bound at root = 21 + 4 = 25.**

Reduced matrix:

	1	**2**	**3**	**4**	**5**
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

Level 1 (edge 1 → j): cost = 25 + A[1][j] + r′.

- 1→2: 25 + 10 + 0 = 35
- 1→3: 25 + 17 + 11 = 53
- 1→4: 25 + 0 + 0 = **25**
- 1→5: 25 + 1 + 5 = 31

Example for 1→4: set row 1 and column 4 to ∞ and A[4][1] = ∞; every row and column still has a 0, so r′ = 0.

Expand node 1→4 (cost 25):

- 1→4→2: 28
- 1→4→3: 50
- 1→4→5: 36

Expand 1→4→2 (28):

- 1→4→2→3: 52
- 1→4→2→5: **28**

Complete the tour: 1→4→2→5 then 3 (cost stays 28), and return 3→1.

Remaining live nodes have bounds 31, 35, 36, 50, 52, 53, all above 28, so they cannot beat it.

Optimal tour: 1 → 4 → 2 → 5 → 3 → 1, cost = 10 + 6 + 2 + 7 + 3 = **28**.

Q7. Explain the method of finding Hamiltonian cycles in a graph using the backtracking method with a suitable example. (2022)

A **Hamiltonian cycle** is a closed path that visits every vertex of a graph exactly once and returns to the start. Deciding whether one exists is NP-complete.

Backtracking method: build the path x[0..n−1] one vertex at a time. Fix x[0] = 0. For position k, try each vertex v and accept it if (1) v is adjacent to x[k−1], and (2) v is not already in the path; for the last position, v must also be adjacent to x[0]. If no vertex fits, backtrack and change the previous choice.

```
HAMILTONIAN(k)
1  repeat
2      NEXT-VALUE(k)                 // choose x[k] from 1..n-1 (valid)
3      if x[k] == 0 return
4      if k == n-1: print x[0..n-1]
5      else HAMILTONIAN(k+1)
6  until false
```

Example. Graph with vertices 0 to 4 and edges 0–1, 0–3, 1–2, 1–3, 1–4, 2–4, 3–4.

- x[0] = 0.
- k = 1: try 1 (adjacent to 0) ✓ → path 0, 1.
- k = 2: try 2 (adjacent to 1) ✓ → path 0, 1, 2.
- k = 3: try 3: not adjacent to 2 ✗; try 4: adjacent to 2 ✓ → path 0, 1, 2, 4.
- k = 4: try 3: adjacent to 4 ✓, unused ✓, and adjacent to x[0] = 0 ✓ → path 0, 1, 2, 4, 3.

Hamiltonian cycle: 0 – 1 – 2 – 4 – 3 – 0. If a vertex had failed (for instance 3 at position 3), the algorithm would have tried the next vertex, and if every vertex failed it would have backtracked to the previous position.

Complexity: O(n!) in the worst case, since all permutations of vertices may be tried; the adjacency test prunes many of them.

Q8. Determine an LCS of X = {A, B, C, B, D, A, B} and Y = {B, D, C, A, B, A}. (Jan 2024)

The **longest common subsequence** is found by dynamic programming: L[i, j] = L[i−1, j−1] + 1 if X_i = Y_j, else max(L[i−1, j], L[i, j−1]); L[0, ·] = L[·, 0] = 0.

Rows X (m = 7), columns Y (n = 6):

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

Length of LCS = L[7, 6] = 4.

Trace back from (7, 6):

- B ≠ A at (7,6); the upper value 4 ≥ left value 4, go up to (6, 6).
- (6, 6): X = A, Y = A match → **A**, go to (5, 5).
- (5, 5): D ≠ B; go up to (4, 5).
- (4, 5): B = B match → **B**, go to (3, 4).
- (3, 4): C ≠ A; left value 2 > up value 1, go left to (3, 3).
- (3, 3): C = C match → **C**, go to (2, 2).
- (2, 2): B ≠ D; left value 1 > up value 0, go left to (2, 1).
- (2, 1): B = B match → **B**.

Reading in reverse: **LCS = B C B A** (length 4). Other LCSs of the same length are BDAB and BCAB. Time and space O(mn).

Q9. Explain Backtracking. Let S = {1, 3, 4, 5} and X = 8; solve the subset sum problem using the backtracking approach. (Jan 2024)

Backtracking is a systematic way of searching a state-space tree depth first. A solution is built one component at a time; as soon as a partial solution violates a constraint (cannot lead to a solution) the algorithm **abandons (prunes)** that node and **backtracks** to the previous choice. It is used for N-Queens, graph colouring, Hamiltonian cycle and subset sum.

Subset sum: find a subset of S whose sum is X = 8. Arrange S in increasing order (1, 3, 4, 5). At each level decide whether to include or exclude the next element. Let s be the current sum and r the sum of remaining elements. Prune when **s + (next element) > X** or **s + r < X**.

Search (include → exclude):

- Include 1 (s = 1):
- Include 3 (s = 4):
  - Include 4 (s = 8) = X ✔ **Solution {1, 3, 4}**.
  - Exclude 4: s = 4, r = 5 → include 5 gives 9 > 8 ✗; exclude 5 gives 4 < 8 ✗.
- Exclude 3 (s = 1, r = 9):
  - Include 4 (s = 5): include 5 gives 10 ✗; exclude 5 gives 5 < 8 ✗.
  - Exclude 4: s + r = 1 + 5 = 6 < 8 ✗ pruned.
- Exclude 1 (s = 0, r = 12):
- Include 3 (s = 3):
  - Include 4 (s = 7): include 5 gives 12 ✗; exclude 5 gives 7 < 8 ✗.
  - Exclude 4 (s = 3, r = 5): include 5 gives s = 8 ✔ **Solution {3, 5}**.
- Exclude 3 (s = 0, r = 9): include 4 (s = 4): include 5 gives 9 ✗, exclude 5 gives 4 ✗; exclude 4 gives s + r = 5 < 8 ✗.

Solutions: {1, 3, 4} and {3, 5}, each with sum 8.

Q10. Apply the Branch and Bound technique to solve the travelling salesman problem for the cost matrix: (∞ 17 13 22 18), (13 ∞ 16 24 19), (15 18 ∞ 16 28), (19 13 15 ∞ 21), (28 24 19 18 ∞). (Jan 2024)

Step 1 – reduce the matrix.

Row minima: row 1 = 13, row 2 = 13, row 3 = 15, row 4 = 13, row 5 = 18; sum = **72**.

After row reduction:

	1	2	3	4	5
1	∞	4	0	9	5
2	0	∞	3	11	6
3	0	3	∞	1	13
4	6	0	2	∞	8
5	10	6	1	0	∞

Column minima: 0, 0, 0, 0, 5 → column 5 reduced by 5, sum = **5**.

Lower bound at root = 72 + 5 = 77.

Reduced matrix:

	1	2	3	4	5
1	∞	4	0	9	0
2	0	∞	3	11	1
3	0	3	∞	1	8
4	6	0	2	∞	3
5	10	6	1	0	∞

Step 2 – branch from city 1 (cost = parent + A[1][j] + r′):

- 1→2: 77 + 4 + 3 = 84
- 1→3: 77 + 0 + 2 = **79**
- 1→4: 77 + 9 + 2 = 88
- 1→5: 77 + 0 + 1 = **78**

Step 3 – expand the least cost node, 1→5 (78):

- 1→5→2: 88
- 1→5→3: **79**
- 1→5→4: **79**

Step 4 – expand a node of cost 79. From 1→3 (79): 1→3→2 = 87, 1→3→4 = 85, 1→3→5 = 85. From 1→5→3 (79): 1→5→3→2 = 98, 1→5→3→4 = **79**. Then 1→5→3→4→2 costs 79, and the remaining edge 2→1 is already forced with zero extra cost.

All other live nodes have bounds ≥ 79 (79, 80, 82, 84, 85 and above), so none can improve on it.

Optimal tour: 1 → 5 → 3 → 4 → 2 → 1.
Cost = 18 (1→5) + 19 (5→3) + 16 (3→4) + 13 (4→2) + 13 (2→1) = **79**.

