DELHI SKILL AND ENTREPRENEURSHIP UNIVERSITY

B.Tech Semester-V (CSE)

End-Semester Examination (Set 3)

Time: 3 Hours

Instructions to Candidates:

1. This paper consists of two sections: Section A and Section B.
2. Section A is compulsory. Attempt all questions.
3. Section B contains descriptive questions.
4. Assume necessary data if not given.

SECTION A – Short Answer Questions (25 Marks)

Attempt all questions. Answers must be concise.

Q1. Define 'Little-oh' notation (o). How does it differ from Big-Oh (O)?

Answer: $f(n) = o(g(n))$ means $f(n)$ grows strictly slower than $g(n)$. Formally, for **any** positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < c \cdot g(n)$ for all $n \ge n_0$. Unlike O, the bound is not tight; the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Q2. What is the theoretical lower bound for any comparison-based sorting algorithm?

Answer: Any comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons in the worst case. This is derived from the decision tree model, where a tree with $n!$ leaves (permutations) must have a height of at least $\log(n!)$.

Q3. Define the 'Selection Problem' (finding the i^{th} order statistic).

Answer: The Selection Problem involves finding the i^{th} smallest element in a set of n unsorted elements.

- $i = 1$: Minimum.
- $i = n$: Maximum.
- $i = \lceil n/2 \rceil$: Median.

Q4. What is the 'Hamiltonian Cycle' problem? Is it in P or NP?

Answer: A Hamiltonian Cycle is a cycle in a graph that visits every vertex exactly once and returns to the starting vertex.

- It is an **NP-Complete** problem. There is no known polynomial-time algorithm to solve it for general graphs, but a solution can be verified in polynomial time.

Q5. Why might a Greedy Algorithm fail for the 'Coin Change Problem' with arbitrary denominations?

Answer: Greedy works for standard currencies (like 1, 5, 10), but fails for arbitrary sets because it lacks lookahead.

- Example: Coins $\{1, 3, 4\}$, Target 6.
- Greedy: Pick 4, then 1, then 1 (Total 3 coins).
- Optimal: Pick 3, then 3 (Total 2 coins).

Q6. Explain the transition function used in 'String Matching with Finite Automata'.

Answer: The transition function $\delta(q, a)$ determines the next state given current state q and input character a.

$$\delta(q, a) = \sigma(P_q a)$$

It maps to the length of the longest prefix of pattern P that is a suffix of $P_q a$ (the pattern prefix of length q concatenated with character a).

Q7. What is 'Memorization' (Memoization) in Dynamic Programming?

Answer: Memoization is a top-down optimization technique where we cache the results of expensive function calls (sub-problems) and return the cached result when the same inputs occur again, preventing redundant computations in recursive algorithms.

Q8. State the time complexity of 'Job Sequencing with Deadlines' if we use a Disjoint Set data structure.

Answer:

- Sorting jobs by profit: $O(n \log n)$.
- Using Disjoint Sets (Union-Find) to find the available slot nearest to the deadline: $O(n \cdot \alpha(n))$, where α is the inverse Ackermann function (nearly constant).
- Total: $O(n \log n)$ (dominated by sorting).

Q9. What is a 'Clique' in an undirected graph?

Answer: A **Clique** is a subset of vertices in an undirected graph such that every two distinct vertices in the clique are adjacent (complete subgraph). The 'Maximum Clique Problem' (finding the largest clique) is NP-Hard.

Q10. Identify the algorithmic approach used for the '15-Puzzle Problem'.

Answer: The **Branch and Bound** technique (specifically Least Cost Search) is typically used. It uses a heuristic cost function (like Manhattan distance of tiles from target positions) to explore the most promising moves first.

SECTION B – Descriptive Questions (75 Marks)

Detailed answers required. Draw diagrams wherever necessary.

Q11. Explain the **Radix Sort** algorithm. How does it sort integers without direct comparison? Analyze its time complexity.

Answer: **Working Principle:** Radix Sort avoids comparison by creating and distributing elements into buckets according to their radix (base). It typically uses a stable sort (like Counting Sort) as a subroutine to sort digits from the Least Significant Digit (LSD) to the Most Significant Digit (MSD).

Steps (LSD method): 1. Find the maximum number to know the number of digits, d. 2. Do the following for each digit i from 1 to d:

- Sort the array elements using a stable sorting algorithm (Counting Sort) according to the i^{th} digit.

Complexity: Let n be the number of elements, b be the base (usually 10), and d be the max number of digits.

- Counting sort takes $O(n + b)$.
- We repeat this d times.
- Total Time: $O(d(n + b))$. Since b and d are often small constants, it runs in linear time $O(n)$.

Q12. Using the **Recursion Tree Method**, solve the recurrence: $T(n) = 3T(n/4) + n^2$.

Answer: We expand the tree for $T(n) = 3T(n/4) + n^2$.

- **Level 0:** Cost n^2. (1 node).
- **Level 1:** 3 children, each size $n/4$. Cost $3 \times (n/4)^2 = 3/16n^2$.
- **Level 2:** 9 children, each size $n/16$. Cost $9 \times (n/16)^2 = 9/256n^2 = (3/16)^2n^2$.

Summing costs:

$$Total = n^2 + \frac{3}{16}n^2 + \left(\frac{3}{16}\right)^2 n^2 + \dots$$

This is a geometric series with ratio $r = 3/16 < 1$. The sum converges to $\frac{n^2}{1-r} = \frac{n^2}{1-3/16} = \frac{16}{13}n^2$.

Conclusion: $T(n) = \Theta(n^2)$.

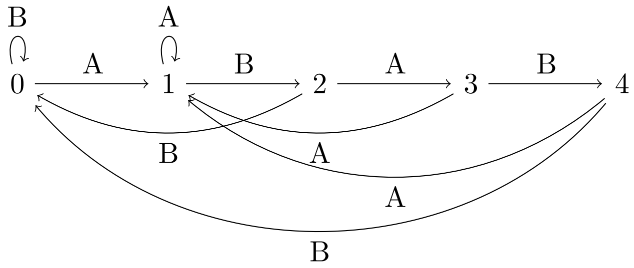
Q13. Discuss the **Finite Automata** approach for String Matching. Draw the transition diagram for the pattern $P =$ ABAB over alphabet $\Sigma = \{A, B\}$.

Answer: **Concept:** We build a Deterministic Finite Automaton (DFA) that accepts strings ending with Pattern P. We process the text T through the DFA. If we reach the final state, a match is found. Time complexity is $O(n)$ for matching.

Pattern $P =$ ABAB. States: 0, 1, 2, 3, 4 (Final).

- State 0: Start. Input A $\rightarrow$ 1. Input B $\rightarrow$ 0.
- State 1 (A): Input B $\rightarrow$ 2. Input A $\rightarrow$ 1.
- State 2 (AB): Input A $\rightarrow$ 3. Input B $\rightarrow$ 0.
- State 3 (ABA): Input B $\rightarrow$ 4. Input A $\rightarrow$ 1.

- State 4 (ABAB): Match found. Next A $\rightarrow$ 1, Next B $\rightarrow$ 0 (overlaps).



Q14. Explain the **N-Queens Problem**. How do we check if placing a queen at position (r, c) is valid in $O(1)$ time given previous placements?

Answer: **Problem:** Place N queens on an $N \times N$ chessboard such that no two queens attack each other (no shared row, column, or diagonal).

Validation Strategy: We place queens row by row. When placing at (r, c): 1. **Column:** Check if 'col[c]' is occupied. 2. **Main Diagonal:** Constant $(r - c)$. Check 'diag1[r-c + offset]' is occupied. 3. **Anti-Diagonal:** Constant $(r + c)$. Check 'diag2[r+c]' is occupied. By maintaining boolean arrays for columns and diagonals, validity is checked in $O(1)$ instead of iterating $O(N)$.

Q15. Explain the **Job Sequencing with Deadlines** problem. Solve for the given jobs (Profit, Deadline): $J_1(20, 2), J_2(15, 2), J_3(10, 1), J_4(5, 3), J_5(1, 3)$.

Answer: **Goal:** Maximize profit by scheduling jobs within their deadlines. Each job takes 1 unit of time.

Greedy Strategy: Sort jobs by Profit (Descending). 1. J_1 (20, D=2): Assign to slot [1-2]. (Status: 1-2 Filled). 2. J_2 (15, D=2): Slot [1-2] full. Check [0-1]. Empty. Assign [0-1]. (Status: 0-1, 1-2 Filled). 3. J_3 (10, D=1): Deadline 1. Slot [0-1] full. Reject. 4. J_4 (5, D=3): Deadline 3. Slot [2-3] empty. Assign [2-3]. 5. J_5 (1, D=3): Slot [2-3] full. Earlier slots full. Reject.

Selected Jobs: J_2, J_1, J_4. **Total Profit:** $15 + 20 + 5 = 40$.

Q16. (a) Design an algorithm to find both the **Minimum and Maximum** elements in an array using the **Divide and Conquer** strategy.
(b) Analyze the number of comparisons made. How does it compare to the naive linear scan?

Answer: **(a) Algorithm (MinMax):** Function 'FindMinMax(arr, low, high)':

1. If 'low == high': Return '(arr[low], arr[low])'.
2. If 'high == low + 1': Compare 'arr[low]' and 'arr[high]', return pair '(min, max)'.
3. Else:
   - 'mid = (low + high) / 2'
   - '(min1, max1) = FindMinMax(arr, low, mid)'
   - '(min2, max2) = FindMinMax(arr, mid+1, high)'
   - Return '(min(min1, min2), max(max1, max2))'

(b) Complexity Analysis: Recurrence for number of comparisons $C(n)$:

$$C(n) = 2C(n/2) + 2$$

Base case: $C(2) = 1$. Solving this:

$$C(n) = \frac{3n}{2} - 2$$

Comparison:

- **Naive Scan:** $2n - 2$ comparisons (worst case: verify every element for min and max independently).
- **Divide and Conquer:** $1.5n - 2$ comparisons.
- **Improvement:** D&C performs roughly 25% fewer comparisons.

Q17. (a) Define the **Longest Common Subsequence (LCS)** problem.
(b) Find the LCS of sequences X = BDCABA and Y = ABCBDAB using Dynamic Programming. Show the DP table.

Answer: **(a) Problem:** Given two sequences, find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguously.

(b) DP Solution: Formula: If $X[i] == Y[j]$, $c[i, j] = 1 + c[i - 1, j - 1]$. Else $c[i, j] = \max(c[i - 1, j], c[i, j - 1])$.

X: BDCABA ($m = 6$), Y: ABCBDAB ($n = 7$).

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

Traceback: $(6, 7) = 4$ (Match 'A'? No, from Left). $(6, 6) = 4$ (Match 'B'? No, from Top). ... Common chars traced: **BCBA**. (LCS Length = 4). *Note: Other valid LCS include BDAB, BCAB.*

Q18. Explain the **Bellman-Ford Algorithm** for single-source shortest paths. How does it handle negative edge weights? What is the condition to detect a negative weight cycle?

Answer: **Algorithm Concept:** Bellman-Ford computes shortest paths from a source vertex to all other vertices in a weighted digraph. Unlike Dijkstra, it can handle negative weights. It is based on the **Relaxation** principle applied repeatedly.

Steps:

1. Initialize distance to source = 0, others = ∞.
2. Relax all edges $|V| - 1$ times.
3. For each edge (u, v) with weight w:

$$\text{If } dist[u] + w < dist[v], \text{ then } dist[v] = dist[u] + w$$

Negative Cycle Detection: After $|V| - 1$ iterations, the shortest paths are guaranteed to be found if no negative cycle exists. We run the relaxation loop one more time (the $|V|^{th}$ time).

- If any distance value changes (i.e., $dist[u] + w < dist[v]$), then a **Negative Weight Cycle** exists in the graph. The algorithm returns False (no solution).

Q19. (a) Explain the **0/1 Knapsack Problem** using the **Branch and Bound** technique.
(b) How is the Upper Bound (UB) calculated for a node in the state space tree? (Hint: Relationship with Fractional Knapsack).

Answer: **(a) Branch and Bound Approach:** Instead of exploring all subsets (Backtracking) or building a table (DP), we organize the search in a State Space Tree.

- We explore nodes based on the "best possible" outcome (Upper Bound) they can lead to.
- We maintain a global variable 'maxProfit'.
- If a node's Upper Bound is less than 'maxProfit', we prune it.

(b) Calculating Upper Bound: To check how "promising" a node is, we relax the integer constraint. We calculate the profit obtainable by solving the **Fractional Knapsack** problem for the remaining capacity.

- $UB = \text{CurrentProfit} + \text{FractionalBound(RemainingItems, RemainingCap)}$
- Since Fractional Knapsack is greedy and optimal, this value represents the maximum possible profit achievable from this branch. If this value $\leq$ current best solution, the branch is dead.

Q20. Explain the concept of **Polynomial Time Reducibility**. Show that if Problem A is NP-Complete and Problem A reduces to Problem B in polynomial time ($A \leq_p B$), then Problem B is NP-Hard.

Answer: **Polynomial Time Reducibility ($A \leq_p B$):** This means there exists a function f computable in polynomial time that transforms any instance I_A of problem A into an instance I_B of problem B, such that:

$$I_A \text{ is YES instance} \iff f(I_A) \text{ is YES instance of B}$$

This implies B is at least as hard as A.

Proof Logic: 1. We know A is NP-Complete. This means every problem $X \in NP$ can be reduced to A ($X \leq_p A$). 2. We are given $A \leq_p B$. 3. By transitivity of reduction: If $X \leq_p A$ and $A \leq_p B$, then $X \leq_p B$. 4. Therefore, every problem in NP is reducible to B. 5. By definition, if every problem in NP is reducible to B, then **B is NP-Hard**. (Note: If B is also in NP, then B is NP-Complete).