DELHI SKILL AND ENTREPRENEURSHIP UNIVERSITY

B.Tech Semester-V (CSE)

End-Semester Examination (Set 2)

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 Θ-notation (Theta). What does it signify regarding algorithm complexity?

Answer: $\Theta(g(n))$ represents the **asymptotic tight bound**. A function $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. It implies the algorithm runs in time proportional to $g(n)$ in both best and worst cases.

Q2. What are 'Disjoint Sets' data structures? List two primary operations performed on them.

Answer: A Disjoint Set data structure maintains a collection of non-overlapping (disjoint) dynamic sets.

- **Union(x, y):** Merges the set containing element x and the set containing element y.
- **Find(x):** Returns the representative (or identifier) of the set containing element x.

Q3. Differentiate between **Backtracking** and **Branch and Bound**.

Answer:

- **Traversal:** Backtracking performs a Depth-First Search (DFS) on the state space tree. Branch and Bound typically uses Breadth-First (BFS) or Best-First Search.
- **Optimization:** Backtracking is often used for decision/constraint satisfaction problems (finding *any* solution). Branch and Bound is used for optimization problems (finding the *best* solution) using lower/upper bounds to prune branches.

Q4. What is the 'Greedy Choice Property'?

Answer: The Greedy Choice Property states that a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, we can make the choice that looks best at the moment without considering results from subproblems.

Q5. Write the recurrence relation for **Binary Search** and state its time complexity.

Answer: Recurrence: $T(n) = T(n/2) + c$ (or $T(n/2) + 1$).
Complexity: Solving this via Master Method gives $T(n) = O(\log n)$.

Q6. In the context of Branch and Bound, distinguish between a 'Live Node' and a 'Dead Node'.

Answer:

- **Live Node:** A node that has been generated but whose children have not yet been fully generated. It is a candidate for further expansion.
- **Dead Node:** A generated node that is either not to be expanded further (pruned due to bounds) or one whose children have all been generated.

Q7. Why is the worst-case time complexity of Naïve String Matching $O((n - m + 1)m)$?

Answer: In the worst case (e.g., Text="AAAA...A", Pattern="AAAAB"), for every possible shift s (from 0 to $n - m$), the algorithm compares all m characters of the pattern before finding a mismatch at the last character. Thus, roughly m comparisons occur n times.

Q8. What is the difference between **Internal** and **External** sorting?

Answer:

- **Internal Sorting:** The entire dataset fits into the main memory (RAM) during sorting (e.g., Quick Sort, Heap Sort).
- **External Sorting:** The dataset is too large for RAM and resides on external storage (disk), requiring data to be loaded in chunks (e.g., Merge Sort variant for large files).

Q9. Define 'Polynomial Time Verification'.

Answer: A problem has the property of polynomial-time verification if, given a proposed solution (certificate), there exists a deterministic algorithm that can verify the correctness of this solution in polynomial time. This is the defining characteristic of the class **NP**.

Q10. Identify the algorithmic paradigm used for **Prim's Algorithm** and **Floyd-Warshall Algorithm**.

Answer:

- **Prim's Algorithm:** Greedy Approach.
- **Floyd-Warshall Algorithm:** Dynamic Programming.

SECTION B – Descriptive Questions (75 Marks)

Detailed answers required. Draw diagrams wherever necessary.

Q11. Explain the working of **Bucket Sort**. Under what conditions does it perform in linear time $O(n)$?

Answer: **Working Principle:** Bucket Sort assumes the input is drawn from a uniform distribution over a range $[0, 1)$.

1. Create n empty buckets.
2. For each array element $A[i]$, insert $A[i]$ into bucket number $\lfloor n \times A[i] \rfloor$.
3. Sort the individual buckets (typically using Insertion Sort).
4. Concatenate all sorted buckets to produce the final output.

Complexity:

- Average Case: $O(n + k)$ (where k is number of buckets). If input is uniformly distributed, each bucket has few elements.
- Worst Case: $O(n^2)$ (if all elements fall into a single bucket and insertion sort is used).
- It performs in **linear time** when the elements are uniformly distributed over the interval.

Q12. Using the **Substitution Method**, prove that the solution to the recurrence $T(n) = 2T(n/2)+n$ is $T(n) = O(n \log n)$.

Answer: **Goal:** Prove $T(n) \leq cn \log n$ for some constant $c > 0$. **Assumption:** Assume $T(k) \leq ck \log k$ holds for all positive $k < n$.
Substitution:

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

Substitute the assumption for $n/2$:

$$T(n) \leq 2\left(c\frac{n}{2}\log\frac{n}{2}\right) + n$$

$$T(n) \leq cn(\log n - \log 2) + n$$

$$T(n) \leq cn \log n - cn + n$$

$$T(n) \leq cn \log n - n(c - 1)$$

For $T(n) \leq cn \log n$ to hold, the term $-n(c - 1)$ must be ≤ 0.

$$-n(c - 1) \leq 0 \implies c - 1 \geq 0 \implies c \geq 1$$

Since we can choose $c \geq 1$, the hypothesis holds. Thus, $T(n) = O(n \log n)$.

Q13. Describe the **Rabin-Karp** string matching algorithm. Explain the Rolling Hash function used to optimize it.

Answer: **Algorithm Logic:** Rabin-Karp uses hashing to find any one of a set of pattern strings in a text. Instead of checking every character, it checks if the hash of the pattern equals the hash of the current substring of the text.

Rolling Hash: To calculate the hash of the next substring efficiently, we use a rolling hash formula. If $H(txt[i])$ is the hash of window starting at i, the hash of the next window $H(txt[i + 1])$ is computed by:

1. Removing the leading digit (high-order term).
2. Shifting the remaining digits.
3. Adding the new trailing digit.

Formula: $H_{i+1} = (d \times (H_i - txt[i] \times h) + txt[i+m]) \mod q$
Where d is radix (number of characters in alphabet), q is a prime number, and $h = d^{m-1} \mod q$. This makes the hash update $O(1)$.

Q14. Explain the **Sum of Subsets** problem. Draw a portion of the State Space Tree for Set $S = \{5, 10, 12, 13\}$ and Target Sum $W = 20$.

Answer: **Problem:** Given a set of non-negative integers and a target sum, find all subsets that add up exactly to the target sum.

Backtracking approach: We include an element or exclude it. If the current sum exceeds W, we prune.

State Space Tree (Partial): Target = 20.

```
                    Root (0)
                   /        \
           +5 (5)              no 5 (0)
          /      \                |
   +10 (15)    no 10 (5)      +10 (10)
Exceeds/    \       |          /     \
 +12 (27) no 12 (15) +12 (17) +12 (22) X  no 12 (10)
   Exceeds/     \Fail |
    +13 (28) no 13 (15) +13 (30) X
```

Note: The path (+5, +15) was a mistake in calculation in some manual traces, but here $5 + 15$ isn't possible as 15 isn't in set. Correct path: Included 5 -¿ Included 10 -¿ Sum=15. Next is 12 (Sum 27¿20, Backtrack). Next is 13 (Sum 28¿20, Backtrack). Actually, a solution isn't found in this branch. (Note: Solution doesn't strictly exist for 5,10,12,13 sum 20. $5 + 15$ no, $10 + 10$ no. $12 + 8$ no. Closest is empty). Question asks for Tree, not necessarily a valid solution.

Q15. Explain the **Approximation Algorithm** for the **Vertex Cover Problem**. State the approximation ratio.

Answer: **Algorithm:** 1. Initialize Cover Set $C = \emptyset$. 2. Consider the set of all edges E. 3. While E is not empty:

- Pick an arbitrary edge (u, v) from E.
- Add both u and v to C.
- Remove all edges from E that are incident to either u or v.

4. Return C.

Approximation Ratio: This algorithm produces a Vertex Cover that is at most **2 times** the size of the optimal vertex cover. Hence, it is a 2-approximation algorithm. Reason: The optimal cover must pick at least one endpoint from every edge chosen by our algorithm. Since we pick both, we pick at most twice the optimal count.

Q16. (a) Trace the **Merge Sort** algorithm on the array: $A = \{38, 27, 43, 3, 9, 82, 10\}$. Show the splitting and merging phases.
(b) Why is Merge Sort preferred for **Linked Lists** over Quick Sort?

Answer: **(a) Trace: Splitting Phase:**

1. $[38, 27, 43, 3, 9, 82, 10]$
2. $[38, 27, 43, 3] \quad [9, 82, 10]$
3. $[38, 27] \quad [43, 3] \quad [9, 82] \quad [10]$
4. $[38][27] \quad [43][3] \quad [9][82] \quad [10]$

Merging Phase:

1. Merge pairs: $[27, 38] \quad [3, 43] \quad [9, 82] \quad [10]$
2. Merge 4-size: $[3, 27, 38, 43] \quad [9, 10, 82]$
3. Final Merge: Compare heads (3 vs 9 -¿ 3), (27 vs 9 -¿ 9), (27 vs 10 -¿ 10), etc.
4. **Sorted:** $[3, 9, 10, 27, 38, 43, 82]$

(b) Preference for Linked Lists:

- **Memory Access:** Merge Sort accesses data sequentially, which is ideal for Linked Lists (no random access required like Quick Sort's pivot swapping).
- **Pointer Manipulation:** Merging two linked lists can be done in $O(1)$ extra space by changing pointers, avoiding the extra auxiliary space ($O(n)$) required for arrays.

Q17. Explain the **Floyd-Warshall Algorithm** for finding All-Pairs Shortest Paths. Run the algorithm on the given adjacency matrix to find the final distance matrix $D^{(3)}$.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & \infty \\ 2 & 0 & \infty \\ \infty & 7 & 0 \end{pmatrix}$$

Answer: **Algorithm Logic:** DP formulation: $d_{ij}^{(k)}$ is the shortest path from i to j using only vertices $\{1, 2, ..., k\}$ as intermediates. Recurrence: $D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], \quad D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$.

Execution: Iteration k=1 (Intermediate Node 1): Check if path via 1 is shorter. $D^{(1)}[2][3] = \min(\infty, D[2][1] + D[1][3]) = \min(\infty, 2 + \infty) = \infty$. No changes as col 1 and row 1 don't change.

$$D^{(1)} = \begin{pmatrix} 0 & 3 & \infty \\ 2 & 0 & \infty \\ \infty & 7 & 0 \end{pmatrix}$$

Iteration k=2 (Intermediate Node 2): Updates using node 2 (Edges: $1 \to 2$ and $2 \to 1$ exist).

- $D[1][3] = \min(\infty, D[1][2] + D[2][3]) = \min(\infty, 3 + \infty) = \infty$.
- $D[3][1] = \min(\infty, D[3][2] + D[2][1]) = \min(\infty, 7 + 2) = \mathbf{9}$.

$$D^{(2)} = \begin{pmatrix} 0 & 3 & \infty \\ 2 & 0 & \infty \\ \mathbf{9} & 7 & 0 \end{pmatrix}$$

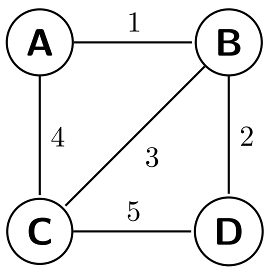
Iteration k=3 (Intermediate Node 3): Updates using node 3.

- $D[1][2] = \min(3, D[1][3] + D[3][2]) = \min(3, \infty + 7) = 3$.
- $D[2][1] = \min(2, D[2][3] + D[3][1]) = \min(2, \infty + 9) = 2$.
- No infinite paths became finite via 3 because column 3 is mostly ∞ except diagonal.

Final Matrix:

$$D^{(3)} = \begin{pmatrix} 0 & 3 & \infty \\ 2 & 0 & \infty \\ 9 & 7 & 0 \end{pmatrix}$$

Q18. (a) Differentiate between **Prim's** and **Kruskal's** algorithm.
(b) Apply **Kruskal's Algorithm** to find the Minimum Spanning Tree (MST) of the following graph. List the edges in the order they are selected.



Answer: **(a) Differences:**

- **Prim's:** Grows a single tree from a starting vertex. Adds the closest vertex to the current tree. Better for dense graphs.
- **Kruskal's:** Grows a forest of trees. Adds the global minimum weight edge that doesn't form a cycle. Uses Union-Find data structure. Better for sparse graphs.

(b) Kruskal's Trace: Edges sorted by weight: 1. (A, B) - 1 2. (B, D) - 2 3. (B, C) - 3 4. (A, C) - 4 5. (C, D) - 5

Selection Steps:

1. Select (A, B) [wt=1]. No cycle. Selected.
2. Select (B, D) [wt=2]. No cycle. Selected.
3. Select (B, C) [wt=3]. Connects B and C. No cycle (A-B-C). Selected.
4. Select (A, C) [wt=4]. Cycle A-B-C-A. Reject.
5. Select (C, D) [wt=5]. Cycle B-C-D-B. Reject.

Final Edges: $\{(A, B), (B, D), (B, C)\}$. Total Weight $= 1 + 2 + 3 = 6$.

Q19. Explain the **Branch and Bound (B&B)** technique using the **Least Cost Search (LC-Search)** method. How is the cost function $\hat{c}(x)$ typically defined? Briefly discuss its application in the **Traveling Salesman Problem**.

Answer: **Concept:** Branch and Bound is a systematic method for solving optimization problems. Unlike Backtracking (DFS), B&B explores the state space tree using a ranking function to select the most promising node (LC-Search uses a min-priority queue).

Cost Function $\hat{c}(x)$: For a node x, $\hat{c}(x) = f(x) + \hat{g}(x)$

- $f(x)$: Cost of reaching node x from the root (actual cost).

- $\hat{g}(x)$: Estimated lower bound cost from x to the goal solution.
- The node with the minimum $\hat{c}(x)$ is expanded next.

Application to TSP: In TSP, we model the problem as finding a tour with minimum cost.

- **Lower Bound Calculation:** For a node (partial tour), we can estimate the lower bound by taking half the sum of the two smallest edges incident to each vertex (Reduced Matrix method).
- If the Lower Bound of a node exceeds the cost of the best full tour found so far, the node is pruned (killed).

Q20. (a) Explain the **Fractional Knapsack Problem** and the greedy strategy used to solve it.
(b) Solve the following instance:
Capacity $M = 20$ kg.
Items (Value, Weight): A:(25, 18), B:(24, 15), C:(15, 10).

Answer: **(a) Strategy:** Unlike 0/1 Knapsack, items can be broken. The greedy choice is to select items with the highest **Value-to-Weight Ratio** (v_i/w_i).

1. Calculate ratio $r_i = v_i/w_i$ for all items.
2. Sort items in descending order of r_i.
3. Add items to knapsack. If an item fits completely, take it. If not, take the fraction that fits.

(b) Solution: Items: A(25, 18), B(24, 15), C(15, 10). Capacity = 20.
1. Calculate Ratios:

- A: $25/18 \approx 1.38$
- B: $24/15 = 1.6$
- C: $15/10 = 1.5$

Order: B (1.6) $\rightarrow$ C (1.5) $\rightarrow$ A (1.38).
2. Selection:

- Take Item B: Weight 15, Value 24. (Rem Cap: $20 - 15 = 5$).
- Take Item C: Weight 10. Only 5 fits.
  Fraction needed: $5/10 = 0.5$.
  Value added: $0.5 \times 15 = 7.5$.
- Knapsack full. Item A is ignored.

Total Value: $24 + 7.5 = \mathbf{31.5}$.