DSA Comprehensive Exam 2024 | Nov.

Answersheet

Part-1

① Worst-case time complexity of finding the maximum element in a min-heap → $O(n)$ ↳ leaf nodes.

② Is $\log_e(n) = \theta(\log_2(n))$? Yes/No

Answer: Yes, $\underbrace{\log_e(n)}_{1.4427} = \frac{\log_2(n)}{\log_2(e)} = \frac{1}{\log_2(e)} \cdot \log_2(n)$

$\therefore \log_e(n) = c \cdot \log_2(n)$

$\left\{\text{where } c = \frac{1}{\log_2(e)}\right\} \Rightarrow \boxed{\log_e(n) = \theta(\log_2(n))}$

by definition of θ-notation,

$f(n) = \theta(g(n)) \Rightarrow \exists\ c_1, c_2 > 0$ & $n_0 > 0$

$\Rightarrow c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall\ n \geq n_0$.

③ If A is an-element array whose each element is b-bit integer, the worst-case complexity of sorting A using the best algorithm depends on the constraints.

* Comparision-based Sorting : Merge, Quick or Heap sort

→ Worst-case : $O(n \lg n)$

→ This is because comparison-based sorting is bounded by $\Omega(n \lg n)$ in the worst case, regardless of the bit-size b of integers.

* Non-comparison-based Sorting

If we exploit the facts that integer are b-bit values. thus range $[0, 2^b - 1]$...

→ Radix Sort : $O(nb)$

→ Counting Sort : $O(n+k)$, where $k = 2^b$.

Hence, best case is radix Sort for b-bits i.e. $O(nb)$

④ Dijkstra's Algorithm is a Greedy algorithm

⑤ Yes, the problem of determining whether there exist a path between 2 vertices s & t in an undirected graph $G = (V, E)$ is in NP, but it is in P.

→ NP membership → A sol^n can be verified in polynomial time given a "certificate".
→ The verification process runs in polynomial

For the given problem:

→ The certificate is the sequence of vertices (a path) that connects s to t.

→ The verification involves checking whether the given sequences is a valid path in G.
  - Verifying that each pair of consecutive vertices in the sequence has an edge in G.
  - Checking that the path starts at s & ends at t.

Ex:- BFS & DFS : $O(V+E)$.

Part-2

⑥ To solve the problem of finding an index i such that $A[i] = i$ in a sorted array $A[1, 2, \ldots, n]$, we can leverage the sorted property of A to design an efficient algo. i.e. Binary search.

Algorithm: ① Initialize 2 pointers low = 1 & high = n.

② while low ≤ high:
  - ① compute the middle index : mid = ⌊(low + High)/2⌋
  - ② check → if A[mid] = mid, return mid
    → if A[mid] < mid, set low = mid+1
    → if A[mid] > mid, set high = mid-1.

③ If the loop exits without finding A[i] = i, return "None".

* Time complexity : $O(\log n)$.

⑦ To prove that at least $n-1$ comparisons are needed to check whether all the elements in an array of n elements are the same, let's use a decision tree argument.

* Understanding the problem :

We have an array A[1], A[2] --- A[n] & want to determine if all elements are the same. The key operation is comparing 2 elements A[i] & A[j], which results in either:

→ Equal (A[i] == A[j]) or

→ Not equal (A[i] ≠ A[j])

If any 2 elements are not equal, we can immediately conclude that the array elements are not the same. However, to ensure all 'n' elements are equal, we must systematically check every pair.

Approach : Lower bound using decision tree.

A decision tree is binary tree representing the sequence of comparisons made & their outcomes (equal or not equal). Each leaf node corresponds to a final decision (all elements are the same or not).

* Proof :

① Base case : $n = 2$, we need exactly 1 comparison

A[1] = A[2]

$T(n) = n-1$

② Inductive step : Assume that the an array of size k, at least $k-1$ comparisons are required. Now, consider an array of size $k+1$:

→ Compare A[1] with A[2], A[3] --- A[k-1] sequentially

→ To confirm all elements are equal, we must verify that each $k+1$ elements matches the 1st element.

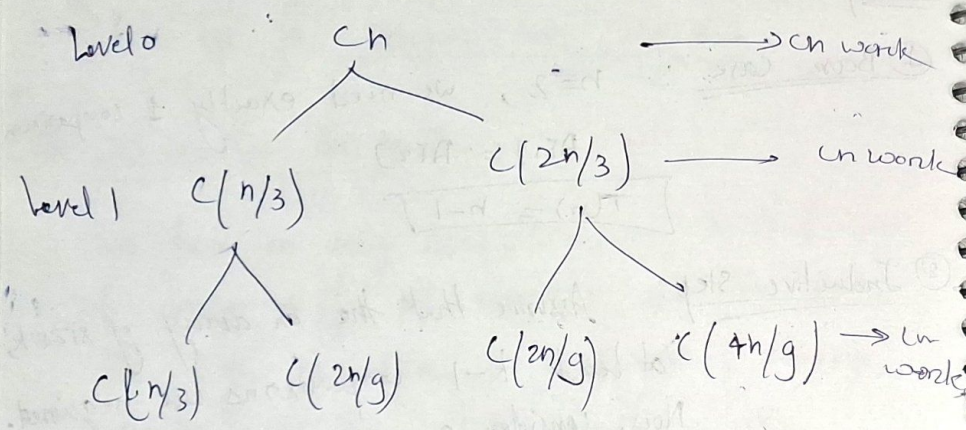
By the inductive hypothesis, verify the first k elements requires $k-1$ comparisons. Adding the comparison for $(k+1)$-th element requires one additional comparison

resulting $k+1-1 = k$ comparisons.

Thus, the results holds for $k+1$.

Hence proved.

(8) $T(n) = T(n/3) + T(2n/3) + cn$



→ Total cost $= c\cdot\frac{n}{9} + c\cdot\frac{2n}{9} + c\cdot\frac{2n}{9} + c\cdot\frac{4n}{9}$

$= cn.$

& depth $= 3$ (Given)

$\Rightarrow d = -\log_3(n).$

$T(n) =$ (cost per level) * (No. of levels)

$T(n) = (cn \cdot \log_3 n)$ Ans.

(9) Mistake in the Inductive Proof of $T(n) = 2T(n/2) + O(n)$.

Given claim: The solution incorrectly concludes $T(n) = O(n)$ using induction.

*Mistake:

The key mistake lies in the inductive step:

→ The assumption $T(n) = O(n)$ $\forall$ $n<k$ & prove $n=k$.

→ When expanding $T(k) = 2T(k/2) + O(k)$, they directly replace $T(k/2) = O(k/2)$ resulting in $2\cdot O(k/2) + O(k) = O(k)$.

This reasoning is "flawed" because it ignores the multiplicative growth of the recurrence.

Specifically: $T(n) = 2T(n/2) + O(n)$ ≠ dominated by $O(n)$, as the recurrence grows logarithmically in depth.

Correct Sol^n: Using master theorem, $T(n) = aT(n/b) + O(n^d)$

$a = 2,\ b = 2$ & $d = 1$

$a = b^d \Rightarrow 2 = 2$

Case 2: $T(n) = \theta(n^{\log_b a} \log n)$

$\therefore T(n) = \theta(\log n)$ Ans.

(10) Algorithm to check whether a graph is Bipartite

→ Approach 1: DFS & BFS

→ Approach 2: 2-coloring method.

A bipartite graph can be defined as a graph whose vertices can be divided into 2 disjoint sets such that no 2 vertices in the same set are connected by an edge. This is equivalent to saying that the graph can be 2-colored, meaning that its vertices can be assigned one of 2 colors (i.e. 0 & 1) such that no two adjacent vertices share the same color.

Algorithm:

Step(1): Initialize coloring array:

→ Create an array color[] of size V (no. of vertices).

→ Initialize all vertices as uncolored i.e.

Color[v] = -1 ∀ v ∈ V.

Step(2): Iterate through all vertices:

→ Since the graph may be disconnected, iterate through all vertices.

→ If a vertices v is not colored, start a BFS/DFS from v & assign it the first color (e.g. color[v]=0)

Step(3): Perform BFS/DFS for 2-coloring.

→ During traversal, for every edge (u,v):

→ If v ≠ colored ⇒ assign opposite color of u. i.e. color[v] = 1 − color[u].

→ If already colored ⇒ color[v] = color[u], the graph is not bipartite (as adjacent vertices share the same color).

Step(4): Repeat for all components.

→ If BFS/DFS completes without conflicts for all components, the graph is bipartite.

Time complexity = O(V+E)

2nd Approach (BFS):

Algorithm:

Step ① Initialize an array color[] with size equal to the No. of vertices. Set all values to -1 (uncolored).

Step ② For each unvisited vertex v:

- ① Assign color[v] = 0 & start a BFS from v.
- ② During BFS, for each edge (u, v):
  - If color[v] = -1, ~~assign color[v] = 1~~ assign color[v] = 1 − color[u]
  - If color[v] = color[u], the graph is not bipartite.

Step ③ If the BFS completes without conflicts, the graph is bipartite.

* Time complexity = O(V+E).
- → No. of Edges
- → No. of vertices

⑪ Algorithm for minimum-Weight Spanning tree with negative weights.

Algorithm: Prim's OR Kruskal's Algorithm

Kruskal's Algorithm

① Sort all edges by weight (including -ve)

② Initialize a disjoint-set data structure (Union-Find) for the graph vertices.

③ Iterate through the sorted edges:
- If an edge does not form a cycle, add it to the MST.

④ Stop when V-1 edges are added to the MST.

* Time complexity:

→ Sorting the edges: O(E log E).

→ Union-Find operations: O(E α(V)), where α(V) is the inverse Ackermann function.

Hence, $T(n) = O(E \log E)$ Ans.

(12) Decreasing-key in a min-heap involves:

① Updating the key values of specific element.

② Restoring the heap property by moving the updated element upward (heapify-up).

* Time complexity:

→ In a binary min-heap, the height is $O(\log n)$

→ Thus, the time complexity of decreasing a key is $O(\log n)$.

(13) (a) Given a directed acyclic graph (DAG), analyze the properties of the adjacency matrix A.

① $A^n = 0$, where n is the no. of vertices. This is because no path in a DAG can contain more than n-1 edges

② The power of A (e.g. A^k) indicates the no. of paths of length k b/w any pair of vertices.

(b) NP-completeness of $Clique_{100}$

determine. if $Clique_{100}$, the problem of deciding whether has a clique of size 100, is NP-complete

Answer: → $Clique_{100}$ is in NP because a certificate (subset of 100 vertices) can be verified in polynomial time.

→ To prove NP-completeness:

→ The general clique problem is NP-complete.

→ $Clique_{100}$ is specific instance of clique with $k=100$, so it is also NP-complete.

(13) Reducing L to SAT or SAT to L

To prove the L is NP-hard, should we reduce SAT to L or L to SAT?

Answer: To prove L is NP-hard → Reduce SAT to L.

→ If SAT (a known NP-complete problem) can be reduced to L in polynomial time, the L is at least as hard as SAT.