(ht / Try HackMD (https://hackmd.io?utm_source=view-page&utm_medium=logo-nav)

Question 1 – Saiyam

Q1.1 Marking Scheme (Total: 5 Marks)

Part A: Greedy Strategy Analysis (1 Mark)

- **Disproof:** The verdict is that the strategy fails.
- **Counter-Example:** A scenario must be provided where the greedy algorithm gives a strictly worse solution than the optimal one
  - **Parameters:** $W = 10$
  - **Items Table:**
Item	Weight (W)	Value/Weight (v/w)	Value (v)
A	2	1.0	2
B	10	1.0	10
C	10	1.0	10

(Note: Value column inferred from $W \times v/w$ in source table)

- **Results:**
  - **Greedy Solution:** Picks A (free) + B. Total Value $= 12$.
  - **Optimal Solution:** Picks B (free) + C. Total Value $= 20$.
- **Marking:** Full 1 mark for any valid numeric counter-example satisfying Greedy Output < Optimal Output.

Part B: Dynamic Programming Solution (4 Marks)

1. Base Case (0.5 Marks)

- Define the state when there are no items or zero capacity:
  - $DP[0][j][0] = 0$.
  - $DP[0][j][1] = 0$.

2. DP Transition (1.5 Marks)

- Maintain two states: coupon not used (0) and coupon used (1).
- **State 0 (Coupon Not Used):**

- $DP[i][j][0] = max(DP[i-1][j][0], DP[i-1][j-w_i][0] + v_i)$

  .

- **State 1 (Coupon Used):**
  - There are three choices: skip, take normally, or take using the coupon.
  - $DP[i][j][1] = max(DP[i-1][j][1], DP[i-1][j-w_i][1] + v_i, DP[i-1][j][0] + v_i)$

    .

3. Complexity Analysis (1 Mark)

- **Time Complexity:** $O(NW)$.
- **Space Complexity:** $O(NW)$ or $O(W)$ (with optimization).

4. Proof of Correctness (1 Mark)

- **Optimal Substructure:** The optimal solution for the first i items and capacity j depends on optimal solutions to smaller subproblems (using the first $i-1$ items).
- **Exhaustiveness:** The recurrence considers all possible actions for each item: exclude the item, include it normally, or include it using the coupon.
- Because all possibilities are evaluated and the maximum is taken, the DP guarantees the globally optimal solution.

Q1.2 Marking Scheme (Total: 5 Marks)

Part A: Greedy Strategy Analysis (1 Mark)

- **Disproof:** The "Earliest Finish Time" strategy fails for the Weighted Interval Scheduling problem.
- **Counter-Example:**
  - **Segment A:** $[0, 2)$, $v_A = 1$.
  - **Segment B:** $[1, 10)$, $v_B = 100$.
- **Execution:**
  - **Greedy:** Sorting by finish time picks A first ($2 < 10$). A overlaps with B at $t = 1$, so B cannot be selected. Total $= 1$.
  - **Optimal:** Skip A, pick B. Total $= 100$.
- **Marking:** Full 1 mark for any valid counter-example where picking the earliest finisher blocks a higher-value interval.

Part B: Dynamic Programming Solution (4 Marks)

1. Base Case (0.5 Marks)

- **Sort:** Sort intervals by non-decreasing finish times: $f_1 \leq f_2 \leq \cdots \leq f_n$.
- **Initialization:** $DP[0] = 0$.

2. Pre-processing (0.75 Marks)

- Compute $p(i)$: For each interval i, let $p(i)$ be the largest $j < i$ such that $f_j \leq s_i$.
- If no compatible interval exists, set $p(i) = 0$.

3. DP Transition (0.75 Marks)

- Let $DP[i]$ denote the maximum total value obtainable from the first i intervals.
- $$DP[i] = max(DP[i-1], v_i + DP[p(i)])$$

  .
  - **Exclude i:** Value remains $DP[i-1]$.
  - **Include i:** Add v_i and combine with the best compatible set, $DP[p(i)]$.

4. Complexity Analysis (1 Mark)

- **Time Complexity:** $O(N \log N)$.
  - Justification: Sorting takes $O(N \log N)$, binary search for $p(i)$ takes $O(N \log N)$, and DP table fills in $O(N)$.
- **Space Complexity:** $O(N)$ to store DP and $p(i)$ arrays.

5. Proof of Correctness (1 Mark)

- **Optimal Substructure:** For intervals $\{1, \ldots, i\}$:
  - If interval i is excluded, the optimal value is $DP[i-1]$.
  - If interval i is included, we gain v_i and must combine it with the optimal solution to all intervals finishing before s_i, which is exactly $DP[p(i)]$.
- **Exhaustiveness:** The recurrence $DP[i] = max(DP[i-1], v_i + DP[p(i)])$ considers all mutually exclusive choices.
- Thus $DP[n]$ yields the global optimum.

Question 2 – Vishal

Click Here! (https://drive.google.com/file/d/1FrUpg0hNRdP-CWPE2J3iwUZheqaPIYI1/view?usp=sharing)

Question 3 – Harpreet

1. Use φ(N) = N − (p+q) + 1 to compute p+q and solve quadratic for p,q. [1]
2. Yes—knowing φ(N) is equivalent to factoring N (a simple construction/proof sketch suffices). [2]
   Miller's Algorithm (refer: https://math.fel.cvut.cz/en/people/gollova/emkr/e-mcr10a.pdf (https://math.fel.cvut.cz/en/people/gollova/emkr/e-mcr10a.pdf))
   Implementation: https://pastebin.com/075YfTv6 (https://pastebin.com/075YfTv6)
   Implementation 2: https://pastebin.com/JErCjwp5 (https://pastebin.com/JErCjwp5)
3. Use CRT to reconstruct X = m^3, take integer cube root. [2]
4. Yes—if you have ≥ e ciphertexts, CRT→X=m^e→integer e-th root. [3]
5. Use Extended Euclidean Algorithm. For M=264, e=5, inverse is d=53. [2]

Question 4 – Shail

Question 4 Answer Key

December 5, 2025

Important Points

4(a) You must show indistinguishability for both a faulty initiator and a faulty participant; showing only one case is incomplete and earns only 2.5 marks.

4(b) If you only use the case $n = 3, f = 1$ to argue impossibility, you get at most 0.5 marks because you must show the argument for general n, not just one specific instance.

4(b) Showing that $n > 3t$ works has no marks unless you also prove that $n \leq 3t$ cannot work; proving only the positive direction gives 0 marks.

4(b) If you divide the system into f groups of 3 nodes, it leads only to 1 mark because the reduction must map to exactly 3 groups.

4(b) When dividing into 3 groups, you must explicitly state that all faulty nodes can be placed in a single group; failing to mention this loses 0.5 mark.

4(c) Showing the protocol on only a single test case earns just 0.5 marks, because you must present the full algorithm and argue correctness for all cases.

4(c) The marking scheme is: 1 mark for correctly writing the OM(1) protocol steps, and 1 mark for proving that it satisfies both Agreement and Validity.

4(d) Giving only a single example where BA fails with 4 edges earns only partial marks, because the question asks for a proof that BA is impossible for every 4-edge topology, not just one instance.

4(a) Impossibility of Perfect BA with 3 Processes and 1 Byzantine Fault (3 marks)

Goal: Show that perfect Byzantine Agreement (BA) is impossible in a synchronous, fully-connected system with $n = 3$ processes and $f = 1$ Byzantine fault.

Indistinguishability Argument

We construct two admissible executions that produce *identical* local views for P_b, yet validity requires different decisions. This contradiction proves impossibility.

Execution 1: P_c correct, P_a Byzantine.

- The initiator P_c is correct and proposes 0.
- Round 1: P_c sends 0 to P_a and P_b.
- P_a (Byzantine) sends value 1 to P_b in Round 2.

Thus P_b receives:

$$(P_c \to P_b = 0), \qquad (P_a \to P_b = 1).$$

By **validity**, since P_c is correct and proposes 0, P_b must decide 0.

Execution 2: P_c Byzantine, P_a, P_b correct.

- The initiator P_c is faulty.
- Round 1: P_c sends 1 to P_a but sends 0 to P_b.
- Round 2: P_a, being correct, relays 1 to P_b.

Thus P_b again receives:

$$(P_c \to P_b = 0), \qquad (P_a \to P_b = 1).$$

The *local view of P_b in both executions is identical.* Therefore P_b must produce the *same* decision in both executions.

Contradiction.

- In Execution 1, validity forces P_b to decide 0.
- In Execution 2, such a decision need not be 0 (since the initiator is faulty); both outputs 0 and 1 are admissible.

Since P_b cannot distinguish the two executions yet must produce different outputs to satisfy BA, perfect BA is impossible with 3 processes and 1 Byzantine fault.

$$\boxed{\text{Therefore, perfect BA is impossible for } n = 3,\ f = 1.}$$

4(b) Necessity of $n > 3t$: Extension to General n (2 marks)

Claim: If a synchronous system $S(n, f)$ has $f \geq n/3$ Byzantine faults, then perfect Byzantine Agreement (BA) is impossible. Hence a necessary condition for solvability is $n > 3t$.

Explanation and Proof (2 marks)

The impossibility for the system $S(3, 1)$ (3 nodes, 1 Byzantine fault) is already established: even in a synchronous, fully-connected network, no deterministic algorithm can guarantee Agreement and Validity. We now extend this impossibility to any larger system with $f \geq n/3$ faults.

Step 1: Partitioning the system. Take the system $S(n, f)$ and divide its n nodes into three equal subsets:

$$S_1,\ S_2,\ S_3, \qquad |S_1| = |S_2| = |S_3| = n/3,$$

assuming n is divisible by 3 (this simplifies notation without affecting the argument).

Step 2: Simulating the smaller impossible system. We now construct a reduction: Treat each subset S_i as if it were a *single* node of the smaller 3-node system $S(3, 1)$. That is,

$$P_1 \leftrightarrow S_1, \qquad P_2 \leftrightarrow S_2, \qquad P_3 \leftrightarrow S_3.$$

Each subset S_i collectively simulates the behavior of the corresponding process P_i in $S(3, 1)$. Messages sent by "node" P_i are implemented by all nodes in S_i sending appropriate messages to the other two subsets.

Step 3: Mapping faults correctly. In $S(3, 1)$, exactly one node may be Byzantine. In our simulation, if P_i is the faulty node, then the entire subset S_i behaves Byzantine.

Since each subset has size $n/3$, the number of faulty nodes in the simulated system is $n/3$. Because we are given that $f \geq n/3$, this simulated fault pattern is valid in $S(n, f)$.

Thus any behaviour that is possible in $S(3, 1)$ (including faulty behaviour) is also possible in $S(n, f)$.

Step 4: Deriving the contradiction. Assume for contradiction that a perfect BA algorithm exists for $S(n, f)$ with $f \geq n/3$.

Then the three subsets S_1, S_2, S_3 could use this algorithm to simulate the three nodes of $S(3, 1)$. Since all messages and faults are preserved exactly, this would give us a correct BA algorithm for $S(3, 1)$.

But we know that perfect BA is *impossible* in $S(3, 1)$.

Therefore the assumption must be false. No algorithm can solve BA in $S(n, f)$ when $f \geq n/3$.

Hence perfect BA requires $n > 3t$.

4(c) Design for $n = 4,\ f = 1$ (2 marks)

There are four processes: a commander P_c and three lieutenants L_1, L_2, L_3. At most one process may be Byzantine ($f = 1$). Communication is synchronous and fully connected.

Protocol (OM(1))

1. **Round 1 (Commander → Lieutenants):** The commander sends its value v_c to all three lieutenants.
2. **Round 2 (Lieutenant forwarding):** Each lieutenant L_i forwards the value it received from the commander to the other two lieutenants.
3. **Decision:** Each lieutenant collects three values:

   x_0 (value directly from commander), x_1 (from one lieutenant), x_2 (from the other lieutenant).

   It decides by taking the **majority** of these three values.

Correctness

Validity. If the commander is correct, it sends the same value v to all lieutenants. Correct lieutenants all forward v in Round 2. Thus each correct lieutenant sees three identical values $\{v, v, v\}$, so the majority is v. Validity holds.

Agreement. With $n = 4$ and $f = 1$, at least *two* lieutenants are correct. These two correct lieutenants both:

receive the same value from the commander (if he is correct), and

forward the same value to everyone.

Therefore every correct lieutenant receives at least two identical values from correct senders. Since the majority of three values must include these two identical correct values, all correct lieutenants compute the same majority. Thus Agreement holds even if one process (commander or lieutenant) is Byzantine.

Hence OM(1) correctly solves Byzantine Agreement for $n = 4,\ f = 1$.

4(d) 4 nodes, 1 Byzantine — edge-count requirement

Claim 1 (Impossibility). If the synchronous undirected communication graph on four processes has at most 4 edges, then there is no deterministic Byzantine Agreement algorithm tolerating one Byzantine process.

Claim 2 (Sufficiency). There exists a graph with 5 edges on four processes (indeed K_4 minus any single edge) for which a simple two-round forwarding + majority protocol solves Byzantine Agreement for $f = 1$. Thus 5 edges suffice.

Notation and model. Four processes P_1, P_2, P_3, P_4. At most one process is Byzantine ($f = 1$). Communication is synchronous and takes place on an undirected graph G with vertex set $\{P_1, \ldots, P_4\}$. In each round each process can send different messages to different neighbors (oral messages). Agreement and validity are required as usual.

Proof of Claim 1 (Impossibility when $|E(G)| \leq 4$)

Overview. We show that any graph G with at most four edges has a vertex of degree at most 2. We then show that if some process Q has degree ≤ 2, an adversary can produce two admissible executions that are indistinguishable to Q but require different decisions — giving a contradiction with BA.

Degree bound. For $n = 4$, if $|E(G)| \leq 4$ then the sum of degrees is $2|E(G)| \leq 8$, so the average degree is ≤ 2. Hence some vertex has degree ≤ 2.

We treat the two nontrivial cases: degree 0 or 1 (trivial impossibility), and degree 2.

(a) Degree 0 or 1. If a process Q has degree 0 (isolated) it receives no information — impossible to satisfy validity when the initiator is elsewhere. If degree 1, Q is connected to exactly one neighbor N. Then all messages to Q must pass through N, so a Byzantine N can equivocate and make

Q see arbitrary values; indistinguishability arguments identical to the 2-node reduction make BA impossible. Hence degree 0 or 1 cannot tolerate $f = 1$.

(b) Degree 2. Let Q be a process of degree exactly 2, and let its two neighbors be A and B. Denote the fourth process by C (so the four nodes are A, B, C, Q). We will show that a single Byzantine can make Q's local view identical in two executions that demand different outputs.

Construct two executions (both admissible with at most one Byzantine):

1. **Execution I (A correct, Byzantine = B):** The designated commander is A and is correct, proposing value 0. In round 1 A sends 0 to its neighbors (including Q and whatever else it is connected to). Let B behave Byzantine and in round 2 send value 1 to Q (while messages to others are chosen so the rest of the network is consistent). In this execution Q receives a 0 purportedly from A and a 1 from B.
2. **Execution II (A Byzantine, B correct):** The designated commander A is Byzantine. A sends 0 to Q but sends 1 to B. In round 2 B (correct) forwards the value 1 to Q. Thus again Q receives a 0 from A and a 1 from B.

In both executions Q has *exactly the same local view*: the messages it received from A and B (and from whatever other neighbor paths exist) are identical. However:

- In Execution I the commander A is correct with value 0, so by *Validity* any correct process must decide 0.
- In Execution II the commander A is faulty and could have sent inconsistent values; there is therefore an admissible outcome in which the correct processes decide 1.

Because Q cannot distinguish these two executions it cannot decide in a way satisfying validity in both cases. Hence BA is impossible when some node has degree 2.

Combining the degree argument and the above cases, any graph with ≤ 4 edges contains a node of degree ≤ 2 and hence cannot support BA for $f = 1$. This proves Claim 1.

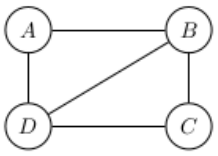
Proof of Claim 2 (Sufficiency of 5 edges with authentication)

The statement that "5 edges suffice" is *false* in the unauthenticated (oral messages) model. In fact, even on the 5-edge graph K_4 minus one edge, a simple two-round forwarding + majority protocol can be forced to fail.

Counterexample for the unauthenticated (oral) model. Even with 5 edges (i.e. K_4 minus one edge), Byzantine Agreement is impossible in the unauthenticated setting. Consider the graph G on $\{A, B, C, D\}$ with edges

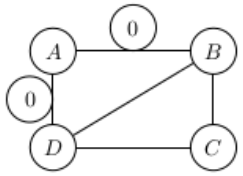
$$AB,\ BC,\ CD,\ DA,\ DB$$

so that **edge AC is missing**. This graph is shown below:



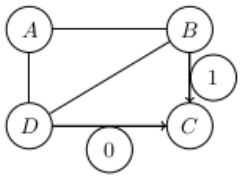
Let A be the commander with value 0, and let B be the unique Byzantine process.

Round 1. A sends 0 to both of its neighbors B and D.



Round 2 (forwarding).

- D is correct and forwards 0 to C.
- B is Byzantine and sends 1 to C.



Thus process C receives:

$$0 \text{ from } D \qquad \text{and} \qquad 1 \text{ from } B.$$

Since C is *not* directly connected to the commander A, it cannot verify which value is the real one. We can construct two admissible executions:

1. Commander A is correct with value 0, and B lies by sending 1.
2. Commander A is faulty and sent 1 to B and 0 to D.

In both executions C receives the *same* pair of messages $(0, 1)$, but the correct decision differs. Therefore C cannot satisfy Agreement and Validity simultaneously.

Conclusion. Even with 5 edges, the unauthenticated (oral) model allows equivocation by a Byzantine node, so Byzantine Agreement is *not* solvable. Hence "5 edges suffice" is only true in the **authenticated (signed)** message model, not in the unauthenticated one.

Authenticated model. However, 5 edges *do* suffice in the **authenticated (signed)** message model. Here, every message contains a digital signature of its originator, and signatures cannot be forged or altered. Therefore, a Byzantine process cannot make a correct process appear to have sent a different value.

On any 4-node graph with 5 edges (i.e., K_4 minus any edge), the following simple two-round protocol achieves Byzantine Agreement for $f = 1$:

1. **Round 1.** The commander signs its value v and sends $(v, \text{sig}_A(v))$ to all its neighbors.
2. **Round 2.** Every process forwards to all neighbors every signed commander-value it received in Round 1. Messages are forwarded without modification; signatures are not forgeable.

3. **Decision rule.** Each process collects all signed commander-values it has received (at most one valid signature from the commander is possible).
   - If it sees a valid signature of the commander on value v, it decides v.
   - If no signed commander-value is received, it decides a fixed default value.

Why this works. Because signatures prevent equivocation, a Byzantine process cannot cause two correct processes to accept two different commander-values. The only thing a faulty process can do is *drop* messages, but cannot forge or alter the commander's signature. Since the graph has 5 edges, every correct process has at least two vertex-disjoint paths from the commander, so at least one correct path must deliver the true signed value. Thus all correct processes that receive the commander's value receive the *same* signed value, guaranteeing Agreement. If the commander is correct, all correct processes receive its signed value, guaranteeing Validity.

Therefore, in the authenticated-message model, 5 **edges suffice** for Byzantine Agreement with 4 processes and $f = 1$.

7

Question 5 – Ishaan

https://drive.google.com/file/d/1sRxZ7sjBMsRoRKE9snYwWcIzdSobF1S1/view?usp=sharing (https://drive.google.com/file/d/1sRxZ7sjBMsRoRKE9snYwWcIzdSobF1S1/view?usp=sharing)

Question 6 – Karthik

Question 7 – Dev

Q7 Rubric

December 5, 2025

Q1

Maximum flow: 11.

- +1 mark for answering "11" without calculation.
- +1 marks for providing the correct calculation.

Min cut: Partition: $S = \{s, a, b\}$ and $T = \{c, d, t\}$

- +1 mark for correct min cut.

Q2

Edges in the MST:

- $a - b$ (weight 3)
- $b - c$ (weight 3)
- $c - d$ (weight 2)
- $d - t$ (weight 5)
- $s - b$ (weight 5)

Total MST weight:

$$3 + 3 + 2 + 5 + 5 = 18.$$

Marking scheme:

- 2 marks if completely correct.
- 0.5 marks if exactly one edge is wrong.
- 0 marks otherwise.

Q3

Task: Given a maximum flow f in G, suppose the capacity of a single edge (u, v) is increased by 1. Give an $O(V + E)$ algorithm to update the maximum flow.

Algorithm

1. Construct (or use the existing) residual graph G_f corresponding to the current maximum flow f.
2. Increase the capacity of edge (u, v) by 1. In the residual graph, this increases the residual capacity of the forward arc (u, v) by 1.
3. Run a BFS (or DFS) in the updated residual graph to check for *one* augmenting path from s to t.
    - If no augmenting path exists, the maximum flow remains unchanged.
    - If an augmenting path exists, augment 1 unit of flow along it.
4. Because the capacity increased by only 1, the flow can increase by at most 1 unit. BFS/DFS is $O(V + E)$, and augmenting along the path takes $O(V)$, so the total update time is $O(V + E)$.

Correctness Justification

Only one unit of capacity is added to the graph, so at most one unit of additional flow can be sent. The existence of a single augmenting path is both necessary and sufficient for increasing the max flow.

Marking Scheme

- Marks will be cut for not mentioning that BFS/DFS runs on the residual graph from source to vertex.
- 1.5 marks for algorithm idea and $O(V + E)$ complexity.
- 0.5 marks for correctness justification.

Q4: Second-Best MST

Below are three fully correct methods for finding the second-best MST. Any one detailed method earns full marks; additional marks may be given for multiple correct approaches.

Method 1 (Cut-Based Brute Force): Remove Each MST Edge and Reconnect (At most 2 marks)

Idea: If we delete any MST edge, the MST splits into two components. The best alternative spanning tree that differs at that edge reconnects these components using the *minimum-weight non-MST edge* that crosses the cut.

Algorithm

1. Compute MST T and its weight W_T.
2. For each edge $e \in T$:
   (a) Remove e, splitting T into two components C_1 and C_2.
   (b) Run DFS/BFS to label all vertices with component numbers.
   (c) Among all edges $(u, v) \in E \setminus T$ with $u \in C_1$ and $v \in C_2$, pick the minimum-weight one f.
   (d) Candidate tree weight:
   $$W' = W_T - w(e) + w(f).$$
3. The minimum W' (with $W' > W_T$) is the second-best MST.

Time Complexity

For each of the $V - 1$ MST edges: a DFS/BFS costs $O(V)$ and scanning edges costs $O(E)$.

$$O((V - 1)(V + E)) = O(VE).$$

Correctness

Removing an MST edge defines a unique cut; any other spanning tree that differs at that edge must reconnect the cut with some non-tree edge. Choosing the minimum such edge gives the best alternative tree differing at e. Testing all MST edges covers all possible single-edge substitutions.

Method 2 (Cycle Property): Add Each Non-MST Edge and Remove Max in Cycle (At most 2 marks)

Idea: For every non-tree edge e, adding it to the MST creates a unique cycle. Removing the *maximum-weight edge* in that cycle yields the best alternative spanning tree using e.

Algorithm

1. Compute MST T and its weight W_T.
2. For each edge $e = (u, v) \in E \setminus T$:

(a) Let P_{uv} be the unique path between u and v in T.

(b) Let h be the heaviest edge on P_{uv}.

(c) Candidate tree weight:

$$W' = W_T - w(h) + w(e).$$

3. The minimum W' (with $W' > W_T$) is the second-best MST.

Time Complexity

Naively, finding h costs $O(V)$ per non-MST edge, giving

$$O(EV).$$

Correctness

This follows from the cycle property: adding a non-MST edge creates a cycle, and removing its heaviest edge yields the minimum-weight alternative tree using that edge. Trying all non-MST edges covers all possible one-edge differences from the MST.

Method 3 (Optimized LCA Method): Max-Edge-on-Path Queries in $O(\log V)$

Idea: This is Method 2 with a faster way to find the heaviest edge on the path between two nodes in the MST using LCA (binary lifting).

Algorithm

1. Compute MST T and root it arbitrarily.
2. Preprocess using DFS to compute:
   - depth of each node,
   - parent table parent$[k][v]$,
   - maximum-edge table maxEdge$[k][v]$.
3. For each non-MST edge $e = (u, v)$:
   (a) Use LCA lifting to find the maximum-weight edge h on the path (u, v) in T in $O(\log V)$.
   (b) Compute:
   $$W' = W_T - w(h) + w(e).$$
4. The smallest $W' > W_T$ is the second-best MST.

Time Complexity

$$O((V+E)\log V).$$

MST: $O(E\log V)$, preprocessing: $O(V\log V)$, each query $O(\log V)$.

Correctness

Same justification as Method 2. The LCA/max-edge preprocessing simply accelerates the maximum-edge-on-path query while preserving correctness.

Question 8 – Hemang

1) EQ_TM = $\{(M_1,M_2) \mid M_1,M_2 \text{ are TMs and } L(M_1)=L(M_2)\}$ is neither RE nor co-RE.

Claim 1 — EQ_TM $\notin$ RE.
Assume for contradiction that EQ_TM is RE. We reduce the complement of the halting/acceptance problem (\overline{A_{TM}}) to EQ_TM.

Given input (\langle M,w\rangle) (where (A_{TM}={\langle M,w\rangle\mid M) accepts (w})), construct two machines (M_1,M_2):

- (M_1) ignores its input and **rejects every string** (so (L(M_1)=\varnothing)).
- (M_2) on any input (x) simulates (M) on (w). If (M) accepts (w) then (M_2) **accepts every** input (x); otherwise (M_2) **rejects** every input. So (L(M_2)=\Sigma^*) if (M) accepts (w), and (L(M_2)=\varnothing) if (M) does not accept (w).

Now (\langle M,w\rangle\in\overline{A_{TM}}) (i.e. (M) does **not** accept (w)) iff (L(M_1)=L(M_2)) iff ((M_1,M_2)\in EQ_{TM}). Thus (\overline{A_{TM}}\le_m EQ_{TM}). If EQ_TM were RE, then (\overline{A_{TM}}) would be RE — contradiction (since (\overline{A_{TM}}) is not RE). Hence **EQ_TM $\notin$ RE**.

Claim 2 — EQ_TM $\notin$ co-RE.
Assume for contradiction that EQ_TM is co-RE (so its complement (\overline{EQ_{TM}}) would be RE). We reduce (\overline{A_{TM}}) (again) to (\overline{EQ_{TM}}) or equivalently reduce (A_{TM}) to something that yields

contradiction; a simpler standard construction:

Given (\langle M,w\rangle) build machines (N_1,N_2):

- (N_1) accepts exactly one fixed string, say `"0"` (so (L(N_1)={0})).
- (N_2) on input (x) simulates (M) on (w). If (M) accepts (w) then (N_2) behaves like (N_1) (accepts exactly `"0"`); otherwise (N_2) accepts **no** strings.

Then (M) accepts (w) ⇔ (L(N_1)=L(N_2)) ⇔ ((N_1,N_2)\in EQ_{TM}). Thus (A_{TM}\le_m EQ_{TM}). If EQ_TM were co-RE (i.e. (\overline{EQ_{TM}}) RE), then (\overline{EQ_{TM}}) being RE would make (A_{TM}) decidable/lead to contradictions with known properties; equivalently, using the previous style mapping one shows (\overline{A_{TM}}) would be RE if EQ_TM were co-RE, contradicting known facts. Therefore **EQ_TM ∉ co-RE**.

(Conclusion: EQ_TM is neither RE nor co-RE.)

2) (a) From 3-SAT (NP-complete) prove computing maximum clique is NP-hard.

(b) Prove NP-hardness of minimum vertex-cover optimization.

(a) Clique is NP-hard (decision/optimization)

The standard polynomial-time reduction from **3-SAT → CLIQUE**:

Given a 3-CNF formula (\phi) with clauses (C_1,\dots,C_m). Build a graph (G) as follows:

- For every literal occurrence in each clause (C_i) create a vertex (v_{i,\ell}) (so each clause contributes up to 3 vertices).
- Add edges between every pair of vertices that come from **different clauses** provided the two literal occurrences are **not complementary** (i.e., not (x) and (\neg x)).
- Ask: does (G) contain a clique of size (m)?

Correctness: A clique of size (m) picks one literal from each clause and all chosen literals are pairwise non-contradictory, so they form a satisfying assignment (set those literals true). Conversely any satisfying assignment yields one true literal per clause which correspond to a clique of size (m). Hence 3-SAT ≤p CLIQUE-decision; CLIQUE is NP-complete; therefore **finding the maximum clique size (optimization)** is NP-hard (optimization version is at least as hard as the decision version).

(b) Minimum Vertex Cover is NP-hard

Vertex Cover decision is known NP-complete. For optimization: computing the minimum vertex cover size is NP-hard because if we could compute the optimum cover in polytime we would decide the decision version (compare optimum ≤ k). Also via graph complements: maximum independent set (MIS) is NP-hard; a vertex cover of size (k) exists iff the graph has an independent set of size (n-k). Because MIS is NP-hard, minimum vertex cover optimization is NP-hard as well. (Either argument is sufficient.)

3) If minimum vertex-cover has a 2-approx algorithm, can we adapt it to get a constant-factor approximation for maximum clique?

Short answer: No (in general). Explanation:

- There is a simple 2-approx greedy algorithm for **Minimum Vertex Cover** (pick any edge, add both endpoints to the cover, remove incident edges, repeat) and it guarantees a factor-2 approximation for vertex cover.
- A natural hope: use complement graph to turn vertex cover approx into clique approx. But this fails to give any constant factor guarantee for clique.

Reason: clique in (G) corresponds to independent set in complement (\overline{G}). A 2-approx for vertex cover in (\overline{G}) yields a 2-approx for minimum vertex cover there, which in turn gives only a weak lower bound for maximum independent set in (\overline{G}) (and independent set size ↔ clique size in (G)). The arithmetic inverting the approximation factor does **not** produce a nontrivial constant approximation for clique.

Concrete counter-example intuition: take graph (G) that is a large clique of size (k) plus many additional vertices connected in such a way that the optimal vertex cover is (n-k). A 2-approx algorithm may return a cover whose complement (an independent set) is extremely small compared to the optimal clique. In fact, **maximum clique** (and independent set) is known to be very hard to approximate: unless P=NP there is no polynomial-time (n^{1-\varepsilon}) approximation for clique for any (\varepsilon>0). So a constant factor approximation for clique is believed impossible and cannot be obtained simply by converting the 2-approx for VC.

Therefore the 2-approx for minimum vertex cover **cannot** be adapted to obtain a constant-factor approximation for maximum clique in general (unless unlikely complexity collapses hold).

4) Greedy algorithm for MINIMUM SET COVER and its approximation ratio

Problem: Universe (U) of (n) elements and collection (\mathcal{S}={S_1,\dots,S_m}). Want a subcollection of minimum size that covers (U).

Greedy algorithm:
Repeat until all elements covered:

1. Pick the set (S\in\mathcal S) that covers the largest number of currently uncovered elements.
2. Add (S) to solution and mark those elements covered.

Approximation guarantee: Greedy achieves an (H_n)-approximation where (H_n = 1 + 1/2 + \dots + 1/n \le 1+\ln n). So greedy is a ((1+\ln n))-approximation.

Proof (standard): Let OPT be the size of an optimal cover. Think of covering elements one by one and charge the cost of each chosen set to the newly covered elements.

Let (U_t) be the set of uncovered elements before the (t)-th greedy pick, and let (u_t=|U_t|). Initially (u_1=n). In any optimal cover, (OPT) sets cover all (u_t) remaining elements, so at least one of those OPT sets covers at least (u_t/OPT) elements. Since greedy picks the set covering the most uncovered elements, greedy picks a set covering at least (u_t/OPT) uncovered elements. Therefore

$$
u_{t+1} \le u_t - \frac{u_t}{OPT} = u_t\left(1 - \frac{1}{OPT}\right).
$$

Unrolling for (k) greedy picks:

$$
u_{k+1} \le n\left(1-\frac{1}{OPT}\right)^k \le n\exp\left(-\frac{k}{OPT}\right).
$$

Choose (k = OPT\cdot\ln n) to make (u_{k+1}<1) so all elements are covered. Thus the greedy uses at most (OPT\cdot\ln n) sets up to constant; more carefully, summation/charging analysis gives the harmonic bound:

$$
|\text{Greedy solution}|\le OPT\cdot H_n \le OPT\cdot(1+\ln n).
$$

So greedy is an (H_n) (≈ (1+\ln n)) approximation.

Complexity: Each greedy step can be implemented in (O(mn)) naive time; with appropriate bookkeeping (heap + frequencies) it runs in roughly (O\big(\sum |S_i|\log m\big)).

Tightness: The (H_n) bound is (asymptotically) tight: there are set-cover instances where greedy attains about (H_n) factor from optimal.

Question 9 – Aryan

Part 1

Assume 1 indexing.

Let $dp[a][b][c]$ denote the maximum length of LCS till first a characters of X, b characters of Y and c characters of Z.

Base Case [0.5 marks]: $dp[0][b][c] = dp[a][0][c] = dp[a][b][0] = 0$ for all valid a, b, c.

If the base case is handled in initialization, marks are given. **If they are not handled explicitly otherwise, 0 marks are awarded for base case**.

Transitions [0.5 marks]:

$$dp[a][b][c] = \begin{cases} \max \begin{Bmatrix} dp[a-1][b][c], \\ dp[a][b-1][c], \\ dp[a][b][c-1] \end{Bmatrix} & \text{if } X[a] \neq Y[b] \vee \\ & \quad Y[b] \neq Z[c] \vee \\ & \quad X[a] \neq Z[c] \\ dp[a-1][b-1][c-1] + 1 & \text{otherwise} \end{cases}$$

We can optimize the space complexity by seeing that just rows $a-1$ and a are used in the recurrence relation. Thus, effectively, if we just maintain the state of the current and previous row, the space complexity is optimized to $O(mk)$.

Pseudocode / Description [1 mark]:

0.5 marks cut for any minor mistake. 2 minor mistakes → 0

```
lcs(X, Y, Z) {
    // assume X, Y, Z start from 1 index
    // let x, y, z denote the respective sizes of the strings
    Initialize dp[2][y][z] to 0
    for i = 1 to x
        current_row <-- i % 2
        previous_row <-- (i - 1) % 2
        for j = 1 to y
            for k = 1 to z
                if X[i] == Y[j] and Y[j] == Z[k] and Z[k] == X[i]
                    dp[current_row][j][k] <-- dp[previous_row][j - 1][k - 1] +
                else
                    dp[current_row][j][k] <-- max(dp[current_row][j][k - 1], d
    return dp[x % 2][y][z]
}
```

Answer is at $dp[x \mod 2][y][z]$

Proof [1 mark]:

Time Complexity [0.5 marks]: $O(nmk)$

Space Complexity [0.5 marks]: $O(mk)$

- **[Note 1]:** Depending on your implementations in pseudocode, the space complexity can be $O(nm)$ or $O(nk)$ as well. Marks will be given for these, provided they are consistent with the pseudocode.
- **[Note 2]:** Marks won't be given for $O(nmk)$ space complexity.

Part 2

The claim is **wrong**. Counter case:

- $X = abcd$
- $Y = adbc$ ($W = abc$)
- $Z = ad$

The answer should be 2, but the claim gives 1.

Marks Distribution: Both the components are binary grading

- Writing that the claim is wrong: 0.5 marks
- Proving the claim is wrong: 1.5 marks
- -0.5 for each inconsistency in the proof **if the overall proof is correct**

Wrong Counterexample

Giving a counterexample like $X = abdc$, $Y = abcd$, $Z = abc$ doesn't work.

$W = \text{LCS}(X, Y)$ can be either abc or abd. **You don't know the output of the algorithm**. Hence, I may configure the algorithm to output abc, making your counterexample wrong.

Part 3

Wrong Solutions

Sorting them in decreasing order of weight and then choosing the independent set greedily doesn't work – take the following case:

Node 1 is connected to 2 and 3. $w_1 = 10, w_2 = 8, w_3 = 9$. It is better to take $8 + 9 = 17$.

Similarly, one cannot use odd and even depths for Independent Sets as if we have a graph with $1 - 2 - 3 - 4$ connected like this, and $w_1 = w_4 = 2$, while $w_2 = w_3 = 1$, the answer should be 4 not 3.

Model Solution

We assume the tree is rooted at node 1. A node u is in the subtree of another node v if we need to pass v to get to u when we start our search from 1.

Let the array $dp[x][2]$ denote the following:

- $dp[x][0]$ denotes the maximum total weight of the Independent Set considering **only vertices in subtree of x** and **not taking** the $w[x]$ into the Independent Set.
- $dp[x][1]$ denotes the maximum total weight of the Independent Set considering only vertices in subtree of x and **taking** the $w[x]$ into the Independent Set.

Base Case [0.5 marks]: For all leaves y, $dp[y][0] = 0$ and $dp[y][1] = w[y]$.

Transitions [0.5 marks]:

$$dp[i][0] = \sum_{u \in children(i)} \max(dp[u][0], dp[u][1])$$
$$dp[i][1] = w_i + \sum_{u \in children(i)} dp[u][0]$$

Pseudocode / Description [1 mark]:

0.5 marks cut for any minor mistake. 2 minor mistakes → 0

```
// N is the number of vertices
// dp[N][2] already exists and is filled with 0s
// w[current_node] exists along with G

dfs(current_node)
    ans_without_curent_node <-- 0
    ans_with_current_node <-- w[current_node]
    for child c in children of current_node
        dfs(c)
        ans_without_current_node += max(dp[c][0], dp[c][1])
        ans_with_current_node += dp[c][0]
    dp[current_node][0] <-- ans_without_current_node
    dp[current_node][1] <-- ans_with_current_node

findIndependentSet()
    dfs(1)
    return max(dp[1][0], dp[1][1])
```

Final Answer [0.5 marks]: The final answer is $\max(dp[1][0], dp[1][1])$. Marks are given if the final answer is clearly mentioned as a part of the pseudocode as well.

Proof [1.5 marks]:

When the node is a leaf, $dp[i][0]$ correctly calculates the value as 0 (not taking) and $dp[i][1]$ correctly calculates the value as v_i (taking).

By inductive hypothesis, lets assume for a particular node x, the children of x given as child_i has the correct value. Note that we can process the children independently as they do not affect each other while forming the independent set. Then if we do not choose the node i, we may or may not choose its children. Since the child has the correct value, the values are stored at $dp[\text{child}_i][1]$ and $dp[\text{child}_i][0]$ respectively. In order to maximize the output, we take the maximum of the two.

Similarly, if we choose node i, we cannot choose its children. This is denoted by $dp[\text{child}_i][0]$.

No marks for time and space complexity since it is given in the question.

Question 10