✅ **Detailed Answers & Notes**

1️⃣ **DSA — Sliding Window (2 Problems)**

Problem A (Fixed Window): Maximum Average Subarray I

LeetCode 643: https://leetcode.com/problems/maximum-average-subarray-i/

Hints

- Maintain a sum of the first k elements.
- Slide by: sum += a[i] - a[i-k].
- Track max sum (or max average = maxSum / k).

C++ Solution

```
#include <bits/stdc++.h>
using namespace std;

double findMaxAverage(vector<int>& nums, int k) {
    long long window = 0;
    for (int i = 0; i < k; ++i) window += nums[i];
    long long best = window;

    for (int i = k; i < (int)nums.size(); ++i) {
        window += nums[i] - nums[i - k];
        best = max(best, window);
    }
    return (double)best / k;
}

int main(){
    vector<int> v{1,12,-5,-6,50,3};
    int k = 4;
    cout << fixed << setprecision(5) << findMaxAverage(v, k); // 12.75000
}
```

Complexity: O(n) time, O(1) space
Edge Cases: k == n, negatives in array, n == k == 1

Common Mistakes: Using double for rolling sum (overflow/precision); recomputing sums from scratch.

Problem B (Variable Window): Fruit Into Baskets

LeetCode 904: https://leetcode.com/problems/fruit-into-baskets/

Hints

- Maintain a window with **at most 2 distinct** fruit types.
- Use hashmap <type -> count>.
- Shrink from left when size of map > 2.

C++ Solution

```
#include <bits/stdc++.h>
using namespace std;

int totalFruit(vector<int>& fruits) {
   unordered_map<int,int> cnt;
   int left = 0, best = 0;

   for (int right = 0; right < (int)fruits.size(); ++right) {
      cnt[fruits[right]]++;
      while ((int)cnt.size() > 2) {
         if (--cnt[fruits[left]] == 0) cnt.erase(fruits[left]);
         ++left;
      }
      best = max(best, right - left + 1);
   }
   return best;
}

int main(){
   vector<int> f{1,2,1,2,3,2,2};
   cout << totalFruit(f); // 5  (window [2,1,2,3,2] or [1,2,3,2,2] depending on input)
```

}

Complexity: O(n) time, O(1) or O(k) space (k ≤ 2)
Edge Cases: All same type; alternating 2 types; a third type appears once.
Common Mistakes: Not erasing type when count hits 0; shrinking only one step instead of until map.size() ≤ 2.

2️⃣ SQL — GROUP BY, HAVING, Window Functions

Schema (example)

Employees(emp_id, emp_name, dept_id, salary)

Departments(dept_id, dept_name)

Sales(emp_id, product, region, amount, sale_date)

A) GROUP BY with Multiple Columns

Goal: total sales by region & product.

Stepwise

1. Base aggregation:

```
SELECT region, product, SUM(amount) AS total_amount
FROM Sales
GROUP BY region, product;
```

2. Add department dimension (via join):

```
SELECT s.region, s.product, d.dept_name, SUM(s.amount) AS total_amount
FROM Sales s
JOIN Employees e ON e.emp_id = s.emp_id
JOIN Departments d ON d.dept_id = e.dept_id
GROUP BY s.region, s.product, d.dept_name;
```

Optimization Tips

- Index join keys: Sales(emp_id), Employees(dept_id), Departments(dept_id).
- If filtering by date/region, add composite index like Sales(region, product, sale_date).

B) HAVING (filter groups)

Q: Regions where **sum(amount) > 1,000,000**:

```
SELECT region, SUM(amount) AS total_amount
FROM Sales
```

```
GROUP BY region
HAVING SUM(amount) > 1000000;
```

Use WHERE for row-level filters (e.g., sale_date >= '2025-01-01'), HAVING for aggregated filters.

C) Window Functions (Ranking within partitions)

1. **Top seller per region** (keep ties with RANK):

```
SELECT emp_id, region, SUM(amount) AS total_amount,
    RANK() OVER (PARTITION BY region ORDER BY SUM(amount) DESC) AS rnk
FROM Sales
GROUP BY emp_id, region
QUALIFY rnk = 1; -- In Snowflake/BigQuery; in others, wrap as subquery/CTE
```

Portable version (CTE):

```
WITH agg AS (
  SELECT emp_id, region, SUM(amount) AS total_amount
  FROM Sales
  GROUP BY emp_id, region
),
ranked AS (
  SELECT emp_id, region, total_amount,
      RANK() OVER (PARTITION BY region ORDER BY total_amount DESC) AS rnk
  FROM agg
)
SELECT * FROM ranked WHERE rnk = 1;
```

2. **Unique row per dept (latest salary record)** with ROW_NUMBER:

```
WITH enriched AS (
  SELECT e.emp_id, e.emp_name, e.dept_id, e.salary, e.updated_at,
      ROW_NUMBER() OVER (PARTITION BY e.emp_id ORDER BY e.updated_at DESC) AS rn
  FROM Employees e
)
SELECT * FROM enriched WHERE rn = 1;
```

3. **Dense ranking by salary inside department**:

```
SELECT emp_id, dept_id, salary,
    DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS dense_rank_in_dept
FROM Employees;
```

Optimization Tips

- Partition keys should be selective; ensure useful indexes on join + filter columns.
- Avoid SELECT * in large windows; project only needed columns.

3️⃣ CS Core — Notes + 5 Q&A Each (Fully Detailed)

A) Operating Systems — Memory (Paging/Segmentation/VM/TLB/Thrashing)

Short Notes

- **Paging**: fixed-size frames/pages → eliminates external fragmentation.
- **Segmentation**: variable-size logical units (code/data/stack) → external fragmentation possible.
- **Virtual Memory**: logical > physical; demand paging loads on use.
- **TLB**: small associative cache for page table entries.
- **Thrashing**: excessive page faults → CPU idle, disk busy.

Q&A

1. **Paging vs Segmentation?**
    - Paging splits **physical memory** into equal frames; process split into equal pages → easy allocation, no external fragmentation; internal fragmentation possible in last page.
    - Segmentation splits program into **logical segments** (varying sizes) → supports protection/sharing; can cause external fragmentation; needs compaction.
2. **What is a Page Fault? Steps?**
    - Access to a page not in RAM. Steps: trap to OS → find a free frame (or evict via replacement) → read page from disk → update page table → restart instruction.
3. **Role of TLB?**
    - Caches recent virtual→physical translations; **hits** avoid page-table walk → huge speedup. TLB miss triggers page-table access.
4. **Thrashing — why & prevention?**
    - Cause: working set > available frames → constant evictions.
    - Fix: increase RAM/frames; working-set model; load control; better replacement (e.g., LRU approximation).

5. **Replacement policies (FIFO/LRU/Optimal)?**
    - **FIFO**: simple, Belady's anomaly possible.
    - **LRU**: approximates recency; needs hardware/software support.
    - **Optimal**: clairvoyant benchmark; not implementable.

B) DBMS — Normalization & Denormalization

Short Notes

- **1NF**: atomic columns, no repeating groups.
- **2NF**: 1NF + no partial dependency on a composite key.
- **3NF**: 2NF + no transitive dependency on key.
- **BCNF**: for all FDs X→Y, X must be a superkey (stronger than 3NF).
- **Denormalization**: deliberate redundancy for performance (e.g., analytics).

Q&A

1. **Why normalize?**
    - Reduce redundancy, update anomalies; improves integrity.
2. **3NF vs BCNF?**
    - 3NF allows X→Y when Y is prime attribute; BCNF requires X be a superkey always → BCNF stricter.
3. **When denormalize?**
    - Read-heavy analytics, frequent joins causing latency; use materialized views, summary tables.
4. **Functional dependency & candidate key?**
    - FD X→Y: X determines Y.
    - Candidate key: minimal attribute set that functionally determines all attributes.
5. **Example of transitive dependency**
    - Student(roll, dept, dept_hod) with dept → dept_hod. Remove dept_hod to a Department table.

C) Computer Networks — DNS, DHCP, ARP

Short Notes

- **DNS**: name→IP resolution (root → TLD → authoritative).
- **DHCP**: dynamic IP allocation (DORA).

- **ARP**: IP→MAC mapping within LAN.

Q&A

1. **DNS: iterative vs recursive?**
    - **Recursive**: resolver asks upstream servers to fetch final answer.
    - **Iterative**: resolver queries each server in turn using referrals.
2. **DHCP DORA?**
    - **Discover → Offer → Request → Acknowledge** (broadcast→unicast sequences).
3. **What is ARP cache?**
    - Local table mapping IP to MAC; entries time out; misses cause ARP broadcast.
4. **ARP spoofing & mitigation?**
    - Attacker poisons cache with fake MAC; mitigate using **Dynamic ARP Inspection**, static ARP, switch security.
5. **DNS caching benefits & risks**
    - Faster lookups; but stale records, cache poisoning risk → DNSSEC.

D) OOPs — Encapsulation, Ctors/Dtors (C++)

Short Notes

- **Encapsulation**: bundle data+methods; use access specifiers.
- **Constructors**: default, parameterized, copy, move; **Destructor**: cleanup.
- **RAII**: resource acquisition is initialization.

Q&A

1. **Encapsulation benefits?**
    - Invariants, controlled access, easier refactor; example: getters/setters validate input.
2. **Copy vs Move ctor?**
    - **Copy**: deep copy expensive; **Move**: steals resources (rvalues) → zero-cost transfers.
3. **When is destructor called?**
    - Object lifetime ends (scope exit, delete); free resources (files, sockets, heap).
4. **Rule of 3/5/0?**
    - If you define any of dtor/copy-ctor/copy-assign → define all three (Rule of 3). With move operations → Rule of 5. Prefer compositions with no manual mgmt → Rule of 0.
5. **Friend classes — when?**

- Tight coupling for performance or operator overloading; use sparingly.

E) SDLC — Waterfall vs Agile, Scrum

Short Notes

- **Waterfall**: sequential, rigid requirements.
- **Agile**: iterative, feedback-driven; Scrum ceremonies.

Q&A

1. **When prefer Waterfall?**
   - Stable requirements, regulated domains, heavy documentation.
2. **Scrum roles & ceremonies?**
   - Roles: Product Owner, Scrum Master, Dev Team.
   - Ceremonies: Planning, Daily Standup, Review, Retrospective.
3. **Definition of Done vs Acceptance Criteria**
   - DoD: quality checklist across stories; AC: story-specific conditions.
4. **Agile pros/cons**
   - Pros: adaptability, visibility. Cons: scope creep risk, discipline required.
5. **Velocity & capacity**
   - Velocity: past completed points; capacity: forecasted points this sprint.

F) Software Testing — Unit, Smoke, Regression, Test Pyramid

Short Notes

- **Unit**: test smallest units, fast & isolated.
- **Smoke**: quick sanity on core paths after build.
- **Regression**: ensure changes don't break old features.

Q&A

1. **Smoke vs Sanity?**
   - Smoke: broad, shallow post-build. Sanity: narrow, focused post-fix.
2. **Mocks vs Stubs?**
   - Stubs return canned responses; mocks verify interactions/behavior.
3. **Test Pyramid?**
   - More unit tests, fewer integration, even fewer UI tests → fast + stable.

4. **Regression suite selection**
    - Impact analysis, risk-based prioritization, critical user journeys.
5. **Code coverage caveat**
    - High coverage ≠ good tests; assert meaningful behavior.

4️⃣ Aptitude — Stepwise

Q1 (Time & Work): A in 12 days, B in 18 days. Together?

- Rates: A=1/12, B=1/18 → LCM 36 → A=3u, B=2u → together 5u = 5/36 work/day.
- Days = 36/5 = 7.2 days.

Q2 (Pipes): Inlet fills in 6 h, outlet empties in 9 h. Together?

- Net rate = +1/6 - 1/9 = (3-2)/18 = 1/18 tank/hour → **18 h**.

Q3 (Ratio): A:B = 2:3, B:C = 4:5. Find A:B:C

- Make B equal: LCM(3,4)=12 → A=8, B=12, C=15 → **8:12:15**.

Shortcuts

- Time & Work: Use LCM to get integer "work units".
- Pipes: Convert to net rate (in/out signs).
- Ratios: Equalize common term via LCM.

5️⃣ Behavioral — STAR

Q1: Worked under pressure

- **S**: Sprint end; prod bug degrading dashboard latency.
- **T**: Fix within 6 hours without downtime.
- **A**: Rolled back heavy query, added read-replica + cache on hot endpoints; wrote throttling rule; paired with QA for regression.
- **R**: Latency ↓ **48%**, incident resolved in **3.5h**, post-mortem actions adopted team-wide.

Q2: Motivation in tech

- **S/T**: I enjoy shipping tools that make decisions faster.
- **A**: I keep a rhythm of learning (DSA daily, CS core rotation), contribute to PrepToShine logs, and turn ideas into PoCs.
- **R**: Built SAR colorization demo & faculty dashboard; measurable improvements (accuracy +13%, review time −40%).
  Hook: I like the loop: learn → build → measure → iterate.

6️⃣ Projects/Resume — Actionables (do today)

- **SAR Colorization**: Add metric bullet —
  "Reduced manual review time by **40%** via auto-colorization & QC flags (SSIM + histogram checks)."
- **Faculty Dashboard**: Add metric bullet —
  "Cut reporting latency by **55%** using pre-aggregations + caching (Redis)."
- **Resume header**: Add GitHub link to **PrepToShine** and keep daily commit streak.