Pseudo-code Problems - Day 1 (Week 2)

1. Java OOPs

🟢 Easy

1. Online Exam Portal (Basic)

Concepts: Inheritance, Method Overriding, Polymorphism

Story: You are building a **university's online examination portal** that will be used by thousands of students and administrators.

The portal needs two main types of users:

1. **Students** – log in to take exams, view results, and track progress.
2. **Admin** – log in to create exams, manage questions, and review student performance.

Scenario details:

- Create a **base class** `User` with common attributes like `username`, `password`, and a `login()` method.
- Derive two subclasses:
  - **Student**: Overrides `login()` to check if the student is **registered** and **enrolled** in a course before granting access to exams.
  - **Admin**: Overrides `login()` to verify **staff credentials** and **admin privileges** before allowing exam management.

Example run:

- A student enters the username `"john123"` and password. The system validates their credentials and enrollment before showing available exams.
- An admin logs in, and the system checks both credentials and role, then shows tools to create or manage exams.

Why it's real-world: Almost every **online learning platform** (Coursera, edX, university LMS systems) uses **role-based login systems**, making this a fundamental OOPS modeling example.

Pseudo-code Problems - Day 1 (Week 2)

2. Movie Ticket Booking

Concepts: Encapsulation, Object Interaction, Class Composition

Story: You've been hired by a startup cinema chain to develop their **movie ticket booking system**.

The system needs to handle:

1. **Ticket** – contains details like movie name, seat number, price, and booking status.
2. **Theatre** – holds movie schedules, available seats, and can issue or cancel tickets.
3. **Customers** – can browse movies, book tickets, and request cancellations.

Scenario details:

- A **Customer** selects a movie in a **Theatre**.
- The **Theatre** checks seat availability and creates a **Ticket** object.
- The customer can **cancel** the ticket before the showtime, and the theatre updates seat availability.

Example run:

- Customer "Ravi" books **Avengers: Endgame** at 6 PM, seat **A12**.
- The system issues a ticket, updates the seat map.
- Later, Ravi cancels the ticket, freeing up seat **A12** for others.

Why it's real-world: This models exactly how **PVR, BookMyShow, and AMC Theatres** implement **ticketing logic** in their backend systems.

Pseudo-code Problems - Day 1 (Week 2)

2. Data Structures & Algorithms

🟢 Easy

1. Bus Route Navigator

Concepts: LinkedList, Insertion, Deletion, Search

Story: You're working with the city's **transport department** to digitize bus route management.

Currently, bus stops are written on paper charts, but the system needs to handle **frequent route changes —** stops get added, removed, or rearranged based on traffic conditions.

Scenario details:

- Each **bus stop** will be a **node** in a `LinkedList`.
- The **LinkedList** represents the order in which the bus travels.
- **Insertion**: A new bus stop is added if the city wants to introduce a detour or service a new neighborhood.
- **Deletion**: A stop is removed if the route no longer passes there due to road construction or low demand.
- **Search**: Passengers can check whether their desired stop is part of the route.

Example run:

- Initial route: **Depot** → **Main Street** → **City Mall** → **Hospital** → **Airport**
- City adds **University Stop** after **City Mall**.
- Later, removes the **Hospital** due to road closure.
- Passenger searches for **Airport**, system confirms it's still in the route.

Why it's real-world: Many public transport systems (like **Delhi DTC, Bangalore BMTC**) dynamically change bus routes, and LinkedList makes insertion/removal operations efficient compared to arrays.

2. Attendance Marker

Concepts: 2D Array, Data Storage, Iteration

Story: You are designing an **attendance tracker** for a school that needs to maintain daily records for a full month.

Pseudo-code Problems - Day 1 (Week 2)

Scenario details:

- Rows represent **students**.
- Columns represent **days of the month** (1 to 30/31).
- Each cell contains **‘P’** for Present or **‘A’** for Absent.
- The system should:
  - Mark attendance daily.
  - Allow teachers to **update** attendance if corrections are needed.
  - Generate a **monthly attendance report** for each student.

Example run:

- Student 1’s attendance for the first week: `P, P, A, P, P, P, A`
- Teacher updates day 3 from **A** to **P** after verifying leave approval.
- At month’s end, the system calculates that Student 1 was present for **26 days**.

Why it’s real-world: Schools, colleges, and companies use exactly this data model for **tracking attendance over time** — it’s simple but powerful for reporting and analysis.

---

3. Collections, Generics, Streams

🟢 Easy

1. Daily Tasks Organizer

Concepts: `LinkedHashMap<Integer, String>`, Insertion Order Preservation

Story: You’re creating a **personal productivity tool** for a busy marketing manager named **Anita**. She plans her daily tasks every morning and wants them displayed **in the exact order she enters them**.

Scenario details:

- Use a `LinkedHashMap<Integer, String>` where:
  - **Key** = Task ID (auto-generated integer)
  - **Value** = Task description
- **Insertion order must be preserved** because:
  - Task 1 might be "Check emails"

    - Task 2 might be "Prepare sales report"
    - Task 3 might be "Call client"

- Even if a task is removed, the remaining tasks should stay in the order they were added.
- Anita can:
    - **Add** new tasks at any time.
    - **Remove** tasks once completed.
    - **View** all pending tasks in the order she planned them.

Example run:

- Morning entry:
  1 → Check emails
  2 → Prepare sales report
  3 → Call client
- Midday, task 2 is removed after completion.
- Remaining tasks shown: 1 → Check emails, 3 → Call client (still in original order).

Why it's real-world: This mirrors to-do list apps like **Google Keep** or **Microsoft To Do**, where maintaining the user's original task order improves usability.

2. Simple Word Counter

Concepts: `Map<String, Integer>`, Stream API, Word Frequency Analysis

Story: A local newspaper editor named **Ravi** wants to analyze the **most frequently used words** in articles to avoid repetitive language.

Scenario details:

- The input is a **paragraph of text** from the day's editorial.
- Use a `Map<String, Integer>` to store:
    - **Key** = word
    - **Value** = count of occurrences
- Use **Stream API** to:
    - Split text into words.
    - Normalize to lowercase (so "The" and "the" are counted the same).
    - Remove punctuation.
    - Count occurrences efficiently.

- At the end, display a list of words sorted by frequency.

Pseudo-code Problems - Day 1 (Week 2)

Example run:

- Paragraph: `"The news today is about the rise in technology trends."`
- Processed word counts:
  - the → 2
  - news → 1
  - today → 1
  - is → 1
  - about → 1
  - rise → 1
  - in → 1
  - technology → 1
  - trends → 1

- Sorted output: `the (2), about (1), in (1), is (1), news (1), rise (1), technology (1), today (1), trends (1)`

Why it's real-world: This is the foundation of text analytics tools used in **SEO optimization**, **plagiarism detection**, and **search engines**.