Game Project Interview Answers – Nikhil Domade

1. If interviewer asks: 'You need to create a game project. What will you do? Explain.'

If I’m asked to create a game project, I’ll follow a structured approach to ensure both good gameplay and technical implementation.

1. Understand the Requirements:
- Decide the game type (e.g., board game like Ludo, quiz game, arcade, etc.)
- Define gameplay rules, number of players, objectives, and winning conditions.

2. Plan the Game Logic:
- Write down the flow: how turns work, how scoring happens, and what ends the game.
- Example (Ludo): Players roll dice, move pieces, can knock out opponents, and the first to reach home wins.

3. Choose the Technology Stack:
- For web games: HTML, CSS, JavaScript or React (frontend)
- For online/multiplayer: Node.js + Express (backend), Socket.io (real-time communication)
- For storing data like scores: MongoDB or MySQL

4. Design the UI/UX:
- Use CSS or libraries like Tailwind and Framer Motion for animations.
- Ensure a clean and responsive interface.

5. Implement Core Features:
- Game board setup
- Dice roll or movement logic
- Player turns and rules
- Scoring system
- Win/lose detection
- Optional: Real-time multiplayer using Socket.io

6. Test the Game:
- Test with multiple users to fix logic and UI bugs.

7. Deployment:
- Host using Render, Vercel, or Netlify for frontend.
- Use Railway or Render for backend.

8. Optional Enhancements:
- Add sound effects, animations, leaderboard, or authentication.

Example Closing:
"I’ve planned a Ludo-like multiplayer game where 2–4 players move clockwise based on dice rolls. I’d build a local version first in JavaScript and later add online features using Socket.io and MongoDB."

2. If interviewer asks: 'Which data structures and algorithms will you use in your game project?'

I’ll use data structures and algorithms based on gameplay requirements:

1. Arrays / Lists:
- To store positions of players, pieces, or the game board state.

2. Hash Map / Object:
- For quick access to player details (ID → score, position, color). O(1) lookups.

3. Queue (Turn Management):
- To handle player turns efficiently in order.

4. Graph (Path-based Movement):
- Represent the board as a graph (nodes = positions, edges = valid moves).
- Use BFS or DFS to validate paths or reachability.

5. Stack (Undo Functionality):
- Store previous game states for undo operations.

6. Sorting / Priority Queue (Leaderboard):
- Maintain top scorers efficiently using heaps or sorting algorithms.

7. Random Number Generation:
- For dice rolls, card shuffling, or random events.

8. Time Complexity Awareness:
- Keep frequent operations in O(1) or O(log n) for smooth performance.

Short Verbal Version:
"I’ll use arrays for board states, hash maps for player info, queues for turn order, graphs for movement, stacks for undo, and heaps for leaderboard.
I’ll keep all frequent operations within O(1) or O(log n)."