Research and Plan

Here's a detailed plan and conceptual breakdown, tailored for the Hackathon challenge, based on using **Kociemba's Two-Phase algorithm** with a **binary cube representation**—but without code or pseudocode:

1. Problem Understanding

The challenge is to design a solver for a standard 3x3 Rubik's Cube that:

- Efficiently models and predicts states.
- Simulates moves accurately.
- Uses optimal data structures and algorithms (with bonus points for scalability and creativity).
- Optionally supports extra features, like visual simulation or handling other cube sizes.

2. Kociemba's Two-Phase Algorithm Overview

High-level approach:

- **Phase 1:** Brings the cube to a "reduced" state, grouping edges and orienting all edges/corners correctly using a restricted move set.
- **Phase 2:** Transitions from the reduced state to the solved state, resolving all remaining permutations using the full move set.

This method is highly efficient and outputs solutions that are close to optimal in move count.

3. Binary Representation of the Cube

Why You Chose Binary:

- Maximizes speed and memory efficiency.
- Bitwise operations are fast for both move application and state comparison.
- Well-suited for hashing, pattern lookups, and compact storage—important for optimal search and Kociemba's phase tables.

How a Cube Is Represented:

- Each "facelet" (colored sticker) or piece (corner, edge) is mapped to a certain bit or group of bits in one or more 64-bit integers.
- For 3x3:
  - **Corners:** 8 pieces, each has 3 possible orientations → typically 3 bits per corner for orientation and position.
  - **Edges:** 12 pieces, each has 2 possible orientations → 4 bits for position, 1 bit for orientation per edge.

Example:

- A corner at position 5 with orientation 1 could be represented as: position bits in the correct place in a 64-bit integer, orientation as a bit flag or field.
- Edges and corners use separate bitfields or packings.

Advantages:

- Single-integer comparison determines cube state equivalence.
- Easy to track, store, and manipulate states for lookup tables and during heuristic computation in both phases.

4. Modeling Cube State and Transitions

State Modeling:

- Cube state stored as a set of bitfields for each piece-type (corners, edges).
- Easy serialization for use in hash tables and quick comparison.

Move Engine:

- Precalculate binary masks/tables for each move (e.g., how an "R" turn repositions and reorients certain bits).
- Apply moves by shuffling/swapping/rotating relevant bits for positions and orientation fields.
- Each move updates the bitfields to reflect the result.

Tracking and Simulating Moves:

- Maintain a move sequence (list of moves, tracked as characters or indices).
- After each move, update the binary state accordingly.
- Store move sequences to reconstruct solutions or output examples.

5. State Prediction Logic & Search

- **Phase 1:** BFS or DFS with pattern/pruning tables, using the limited set of allowed moves, updating the binary representation at each step.
- **Phase 2:** Similar approach, but on the subset of states output by Phase 1, with the full move set.
- **Heuristics & Pattern Databases:** Lookup tables using the binary representation as keys for rapid state evaluation and pruning during search.

6. Algorithm Efficiency

- **Time Complexity:** Kociemba's algorithm is optimal for this kind of approach, typically solving position in under a second.
- **Space Complexity:** Controlled by the size of lookup/pattern tables (which are manageable with the binary compactness).

- **Scalability:** Binary representation allows scaling to 2x2 with simpler packing (fewer pieces), and can be adapted for 4x4, though state complexity increases sharply.

7. Presentation, Output, and Bonus Features

- **Walkthrough readiness:** You'll explain the phases, the cube's binary state, how moves alter the state, and how the algorithm zeroes in on a solution.
- **Output Examples:** After applying a scramble, show initial and final binary states, plus human-readable move sequence.
- **Optional Visuals/UI:** Visualize the cube and solution sequence by translating binary state back to facelet colors or piece positions.

Summary Table

Component	Approach	Benefits
State Representation	Binary (bitwise encoding of edges/corners)	Fast, compact, scalable, easy hash/lookups
Algorithm	Kociemba's Two-Phase	Fast, near-optimal, well-supported
Move Engine	Bitmask/table-driven transitions	Precise, efficient rotations
Prediction/Tracking	Real-time binary updates + move sequence trail	Transparency, debugging, replayability
Output	Solution sequence + state mapping	Clarity for presentation and evaluation

Now plan for code

Here's a detailed, step-by-step plan for implementing a Rubik's Cube solver using **Kociemba's Two-Phase Algorithm** with a **binary state representation**—without writing any code:

Step 1: Specification and Requirements

- Clarify target: **Standard 3x3x3 cube** solver.
- Decide inputs/outputs: Accept a scramble (sequence or binary state); output a solution sequence and optionally, before/after cube visualization.
- Optional: Prepare for easy extension to 2x2 or 4x4 cubes.

Step 2: Binary Cube Representation

- **Design binary encoding** for all pieces:
  - **Corners**: Encode the position (0–7) and orientation (0–2) for each corner using bitfields.
  - **Edges**: Encode the position (0–11) and orientation (flip, 0–1) for each edge similarly.
- Plan bit-packing scheme for efficient storage and retrieval (e.g., use 64-bit integers for each major group).
- Define serialization/deserialization: Map facelet colors or a string scramble to this binary state, and vice versa.

Step 3: Move Engine and Tables

- Identify all basic moves (U, D, L, R, F, B, each with possible modifiers) and their effect on the cube's pieces.
- **Precompute** or describe how moves permute/flip the bitfields for positions and orientations of corners and edges.
- Organize these as lookup tables or bitwise operations for instant update during search.

Step 4: Phase 1 Search

- Phase 1 goal: Orient all edges/corners and group relevant edge pieces, using a restricted move set.
- Implement a state validator to check if a cube is in Phase 1's "reduced" group.
- **Search strategy**: Use BFS, IDA*, or similar (optimized by precomputed pruning/pattern tables) starting from input state.
  - Use binary keys for efficient state hashing/lookup and to avoid revisiting.
  - Store move sequences that reach valid Phase 1 group states.
- Define pruning heuristics and lookup tables for edge orientation, corner orientation, and edge grouping using binary keys.

Step 5: Phase 2 Search

- Phase 2 goal: Solve the cube completely from reduced group, now allowed to use the full set of moves.
- Start with all states output by Phase 1.
- Similar search procedure: BFS/IDA* with additional pruning tables geared for Phase 2 "distance to solved," again keyed by the binary representation.
- Store shortest move sequences found for each considered state.

Step 6: Solution Path Reconstruction

- After finding a Phase 1 and Phase 2 sequence, **concatenate** these into a single solution.
- Convert the sequence of binary state transformations into standard cube notation for display.

- Provide a function to replay the scramble/solution, mapping binary states back to human-readable format (for visualization/debugging).

Step 7: Utilities and Testing

- Implement bidirectional conversion:
  - Facelet (color) input ↔️ binary representation.
  - Binary state ↔️ human-readable or visualizable form for UI/debugging.
- Scramble generator tool: Produce random states for robust testing.
- Solution validator: After applying solution moves, verify that the cube is indeed solved (using binary state check).

Step 8: Optimization and Enhancement

- Optimize memory usage for all state tables and hashing.
- Tune pruning and pattern tables for the best trade-off of storage vs. search speed.
- Add logging for moves, state changes, and phase transitions to aid in debugging and live demonstration.

Step 9: User Interface / Output (Optional)

- Display initial, scrambled, and solved cube states (text or GUI).
- Show move-by-move animation or printout for visualization.
- Output metrics: Number of moves, computation time, memory used.

Summary Table

Step	Action
1. Specification	Define solver scope and requirements
2. Representation	Encode cube state in binary
3. Move Engine	Table-driven bitwise transformation of state
4. Phase 1	Phase 1 search and binary state management
5. Phase 2	Phase 2 search with full move set
6. Solution Output	Reconstruct moves, validate, and display
7. Utilities	Conversion, scrambling, and validation tools
8. Optimization	Pruning/lookup table improvement and performance tuning
9. User Interface	Present results, visualization, and metrics