📓 Zuma Game – Full Documentation

1. Problem Statement

You are playing a variation of the Zuma game with a single row of colored balls (board) and several colored balls in your hand (hand). Your goal is to clear the board by inserting the fewest number of balls from your hand. After each insertion:

- If three or more consecutive balls of the same color appear, they are removed.
- This may lead to a chain reaction of removals.
- The game ends when either the board is empty (you win) or there are no balls left in your hand.

Task:

Return the minimum number of balls you need to insert to clear the board. If it's not possible, return -1.

2. Intuition

To solve this problem optimally:

- Each insertion must lead to progress — either immediate removal or setup for future removal.
- Brute-force every valid insertion with a BFS (Breadth-First Search) strategy.
- Avoid revisiting same game states using a visited set.

3. Key Observations

- No initial sequence in the board is three or more — no pre-cleanup is needed.
- Group removal can **cascade** — multiple chains in one move.
- The hand is small (≤ 5 balls) and board ≤ 16, so BFS is feasible.
- Only meaningful insertions are allowed:
    - Between similar-colored balls.
    - Before or after a group that might become $3+$ after insertion.

4. Approach

Step-by-step BFS:

- Sort the hand – enables duplicate check.
- Initialize a queue with (board, hand, steps) and a visited set.
- While queue is not empty:
    - For every ball in hand, try all insertion positions.
    - After insertion, call remove_same() to collapse similar groups.
    - If board is empty, return the number of steps.
    - Add new state to queue if not visited.

Helper Function – remove_same(s, i) Recursively removes all sequences of ≥ 3 same-colored balls from position i.

5. **Edge Cases**

- Board is already empty: return 0.
- Hand has no useful balls: return -1.
- Board with single ball: at least two more needed for removal.
- Inserting the same ball multiple times without removal: avoid using that path.

6. **Complexity Analysis**

Time Complexity

- O(N * M!) in the worst case
  - N = board length (≤16)
  - M = hand length (≤5)
  - Each permutation of insertions is tried, but pruning and visited states reduce this.

Space Complexity

- O(N * M) due to:
  - Queue storing (board, hand) combinations.
  - Visited set for memoization.

7. **Alternative Approaches**

- DFS with memoization: May lead to TLE due to deeper recursion.
- Greedy: Not suitable because a locally optimal insertion might block future removals.
- *A search**: Possible with good heuristics but adds overhead.

8. **Test Cases**

✅ Example 1

board = "WRRBBW", hand = "RB"

Output: -1

✅ Example 2

board = "WWRRBBWW", hand = "WRBRW"

Output: 2

✅ Example 3

board = "G", hand = "GGGGG"

Output: 2

✅ Additional Cases

- board = "RRR", hand = "" → Output: 0
- board = "Y", hand = "Y" → Output: -1
- board = "RGB", hand = "RGB" → Output: -1

9. **Final Thoughts**

- The BFS strategy with pruning is efficient for small board and hand sizes.
- Sorting the hand and tracking visited states prevents redundant processing.
- Insertion logic must avoid wasteful placements, enabling a targeted search.