Documentation on Constructing a Quad-Tree from a Binary Grid

Table of Contents

1. Problem Statement
2. Intuition
3. Key Observations
4. Approach
5. Edge Cases
6. Complexity Analysis
    - Time Complexity
    - Space Complexity
7. Alternative Approaches
8. Test Cases
9. Final Thoughts

1. Problem Statement

Given an n x n binary grid consisting of only 0s and 1s, construct a **Quad-Tree** representation of the grid. Each node in the Quad-Tree has:

- val: Boolean value (True for 1, False for 0), only relevant for leaf nodes.
- isLeaf: Boolean indicating if the node is a leaf.
- topLeft, topRight, bottomLeft, bottomRight: Pointers to four child nodes.

Constraints:

- n == grid.length == grid[i].length (Square matrix)
- $n = 2^x$ where $0 <= x <= 6$

2. Intuition

A **Quad-Tree** is a recursive data structure where each node can be divided into four quadrants. If all values in a given sub-grid are the same (0s or 1s), we create a **leaf node**. Otherwise, we divide the grid into four equal parts and recursively process each.

3. Key Observations

i. If all values in a sub-grid are identical (all 0s or all 1s), it can be represented as a **single leaf node**.
ii. If values differ, the sub-grid must be divided into **four equal quadrants**.
iii. The process continues recursively until we reach uniform grids (leaf nodes) or base case grids of size 1x1.

4. Approach

i. **Check Uniformity:** If all values in a sub-grid are the same, return a leaf node.
ii. **Divide Grid:** If not uniform, divide into four quadrants:
   a. topLeft
   b. topRight
   c. bottomLeft
   d. bottomRight
iii. **Recursive Construction:** Recursively construct each quadrant.
iv. **Combine Nodes:** If all four quadrants are identical, merge them into a single node.

5. Edge Cases

- **Smallest Grid (1x1):** Should directly return a leaf node.
- **All Elements Same:** Should return a single leaf node without unnecessary recursion.
- **Alternating Values:** Requires full recursion down to 1x1 grid cells.

6. Complexity Analysis

Time Complexity

- **Worst Case (Completely Non-Uniform Grid):** Each level of recursion divides the grid into **four parts**. The recursion depth is log n, leading to an **O(n^2)** complexity.
- **Best Case (Uniform Grid): O(1)** (Single node returned).

Space Complexity

- **Recursive Call Stack:** Worst-case depth is log n, requiring **O(log n)** additional space.
- **Quad-Tree Storage:** In the worst case, each cell has its own node, leading to **$O(n^2)$**.

7. Alternative Approaches

i. **Iterative Approach:** Instead of recursion, we could use a queue-based level order traversal, but this would require more memory for bookkeeping.

ii. **Precompute Uniform Regions:** Instead of checking uniformity in $O(n^2)$, use prefix sums to speed up checking in O(1). However, this would increase space complexity.

8. Test Cases

Example 1

Input:grid = [[0,1], [1,0]]

Output:[[0,1],[1,0],[1,1],[1,1],[1,0]]

Example 2

Input:grid = [[1,1,1,1,0,0,0,0],

[1,1,1,1,0,0,0,0],
[1,1,1,1,1,1,1,1],
[1,1,1,1,1,1,1,1],
[1,1,1,1,0,0,0,0],
[1,1,1,1,0,0,0,0],
[1,1,1,1,0,0,0,0],
[1,1,1,1,0,0,0,0]]

Output: [[0,1],[1,1],[0,1],[1,1],[1,0],null,null,null,null,[1,0],[1,0],[1,1],[1,1]]

9. Final Thoughts

- **Pros:** Efficient and easy-to-understand recursive solution.
- **Cons:** Can be memory-intensive for non-uniform grids.
- **Potential Optimizations:** Precompute uniformity using prefix sums to speed up checking.