**Problem description**

Devise an algorithm for the following task: given a 2n × 2n (n > 1) board with one missing square, tile it with right trominoes of only three colors so that no pair of trominoes that share an edge have the same color. Recall that the right tromino is an L-shaped tile formed by three adjacent squares.

Use dynamic programing to solve this problem.

**Detailed assumptions**

1. The board is a square with dimensions 2n x 2n, where n > 1 (minimum size is 4x4).
2. There is exactly one missing square on the board.
3. The right tromino is an L-shaped tile formed by three squares.
4. There are three distinct colors available for the trominoes (Red, Green, Blue, for example).
5. The missing square can be located anywhere on the board.

**Detailed solution including the pseudo-code and the steps of solution**

**First Step**: we always put tromino of value 1 in the center of board

**Second Step**: we divide board into four quarters and if divided board is bigger than 2x2 board do first step then second step again else we do third step

**Third Step**: we do this step if board == 2x2 board if this is the first time we access this quarter then we see all other trominoes around it so that we place the right tromino the we record it because all quarters of same type in other divided boards will have the same tromino

**Pseudo code**

function tromino(grid\_size, start\_x, start\_y, grid)

// Base Case: single cell

if grid\_size == 1 then

return

// Find the location of the missing square

missing\_x = null

missing\_y = null

for i = start\_x to start\_x + grid\_size - 1 do

for j = start\_y to start\_y + grid\_size - 1 do

if grid[i][j] != 0 and grid[i][j] != -1 then

missing\_x = i

missing\_y = j

break 2; // Exit both loops after finding the first missing square

endif

endfor

endfor

// Handle different cases based on missing square location

if grid\_size == 2 then

// Place tromino based on missing square quadrant

if missing\_x < start\_x + 1 and missing\_y < start\_y + 1 then

placeTromino(start\_x + 1, start\_y + 1, start\_x + 1, start\_y, start\_x, start\_y + 1, 1, grid)

else if missing\_x >= start\_x + 1 and missing\_y < start\_y + 1 then

placeTromino(start\_x + 1, start\_y + 1, start\_x + 1, start\_y, start\_x, start\_y - 1, 2, grid)

else if missing\_x < start\_x + 1 and missing\_y >= start\_y + 1 then

placeTromino(start\_x + 1, start\_y + 1, start\_x, start\_y + 1, start\_x - 1, start\_y + 1, 3, grid)

else

placeTromino(start\_x + 1, start\_y + 1, start\_x, start\_y, start\_x - 1, start\_y - 1, 4, grid)

endif

else

// Handle missing square in center quadrant

if missing\_x < start\_x + grid\_size / 2 and missing\_y < start\_y + grid\_size / 2 then

placeTromino(start\_x + grid\_size / 2, start\_y + grid\_size / 2 - 1, start\_x + grid\_size / 2, start\_y + grid\_size / 2, start\_x + grid\_size / 2 - 1, start\_y + grid\_size / 2, 0, grid)

else if ... (similar checks for other quadrants with missing center square)

endif

// Recursively call tromino for sub-grids

tromino(grid\_size / 2, start\_x, start\_y + grid\_size / 2, grid)

tromino(grid\_size / 2, start\_x, start\_y, grid)

tromino(grid\_size / 2, start\_x + grid\_size / 2, start\_y, grid)

tromino(grid\_size / 2, start\_x + grid\_size / 2, start\_y + grid\_size / 2, grid)

endif

end function

function placeTromino(x1, y1, x2, y2, x3, y3, quadrant, grid)

// Update neighboring cells based on tromino placement and quadrant

// ... (similar logic to original placeTrom function)

// Set the quadrant value for all three tromino squares in the grid

grid[x1][y1] = quadrant

grid[x2][y2] = quadrant

grid[x3][y3] = quadrant

end function

**Complexity analysis for the algorithm**

**Base Case:** the base case tromino(n = 1) takes constant time (O(1)).

**Recursive Calls:** The tromino function makes four recursive calls for sub-grids with size n/2. This seems similar to a logarithmic depth, but there's a crucial difference.

**placeTrom** **function:** While the number of iterations in placeTrom itself is constant, the key point is that placeTrom is called for every cell in the grid during the recursion. This happens because in each sub-grid, all the cells are checked for potential tromino placement based on the missing square's location.

**Combining Time:** The number of recursive calls grows with the grid size (n), leading to placeTrom being called for an increasing number of cells. This translates to the total work done growing proportionally to the square of the grid size (n^2).

**T(n) = 4T(n/2) + n**

**Overall Time Complexity:**

Using master theorem then **time complexity: O(n2)**

**Sample output of the solution**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**A comparison between your algorithm and at least one other technique**

|  |  |  |
| --- | --- | --- |
|  | **Dynamic Programming** | **Divide & Conquer** |
| **Time Complexity** | O(n2) | O(n2) |
| **Space Complexity** | O(n) | O(n) |
| **Readability** | Complex and Difficult | Easy |

**Conclusion**

**Dynamic programming does have a big impact on this problem although it is not obvious on low input size**