Task 1 : Tromino Tilings

1.1 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.

1.2 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.

1.3 Detailed Solution

1.3.1 Description of Solution

1. we always put tromino of value 1 in the center of board
2. 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
3. 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

1.3.2 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
```

1.4 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)**

1.5 Sample Output of the Solution

```
Enter k which is 2^k: 2
Enter location of hole (x,y) 0 0
output:
-1      3       2       2
3       3       1       2
2       1       1       3
2       2       3       3
```

```
Enter k which is 2^k: 3
Enter location of hole (x,y) 5 4
output:
3       3       2       2       3       3       2       2
3       1       1       2       3       1       1       2
2       1       3       3       2       2       1       3
2       2       3       1       1       2       3       3
3       3       2       1       2       -1      2       2
3       1       2       2       2       2       1       2
2       1       1       3       2       1       1       3
2       2       3       3       2       2       3       3
```

1.6 Comparison with Another Algorithm

TABLE 1

	Dynamic Programming	Divide & Conquer
Time Complexity	$O(n^2)$	$O(n^2)$
Space Complexity	O(n)	O(n)
Readability	Complex and Difficult And not easy to understand	Conceptually simple and straightforward

1.7 Conclusion

By comparing the algorithms used to solve the problem we observe:
The dynamic programming approach is not any different compared to divide & conquer approach in time complexity and space complexity but when it comes to readability and implementation the divide & conquer is a lot easier compared to dynamic programming