Code Analysis Report: Tower of Hanoi Implementation

1. Introduction

The code implements a solution to the Tower of Hanoi puzzle. It uses a 2D deque of strings to create a visual, text-based representation of the towers and disks, which is updated in the console after every move.

The primary focus of this report is to analyze the program's structure, the purpose of each function, and the overall time and space complexity. A key finding is that the recursive algorithm used is a non-standard variant, resulting in a number of moves significantly greater than the optimal solution for the 3-peg Tower of Hanoi problem.

2. Data Structures and Global Variables

The program utilizes a primary data structure and several global variables to manage the state of the puzzle.

- deque<deque<string>> towers: This is a 2D deque (a grid) that visually stores the state of the three towers.
  - The outer deque represents the rows, corresponding to the levels of the towers (from top to bottom). The size is determined by n, the number of disks.
  - The inner deque represents the columns, corresponding to the three towers (or pegs).
  - Each string within the grid is a visual representation of either a disk (e.g., " ### ") or an empty space on a peg (e.g., " . ").
- string emptySpot: A global string variable that stores the representation of an empty level on a peg. This is used for consistent comparisons and assignments.
- int n: An integer that stores the total number of disks in the puzzle, as specified by the user.
- int numSteps: A counter that tracks the total number of moves made to solve the puzzle.

3. Functional Breakdown

The program is segmented into several functions, each with a distinct responsibility.

void setup(int n)

This function initializes the puzzle's starting state.

- **Purpose**: To construct the initial board with n disks stacked in ascending order of size on the first tower.
- **Steps**:
  1. It clears any pre-existing towers data.
  2. It calculates maxWidth, the width of the largest disk, which is 2n - 1 characters.
  3. It creates the empty_peg_str to represent an empty spot on a tower.
  4. It iterates from i = 0 to n-1, creating each disk as a string. The number of # characters corresponds to the disk size (2i + 1), and it's padded with spaces to match maxWidth.
  5. For each level i, it places the newly created disk string on the first tower and the empty_peg_str on the second and third towers, then pushes this level into the towers grid.

void print()

This function displays the current state of the three towers on the console.

- **Purpose**: To provide a visual representation of the board at any given moment.
- **Steps**:
  1. It iterates through each row (i) and each column (j) of the towers grid.
  2. It prints the string at towers[i][j] to the console.
  3. After printing all rows, it prints a separator line ("--------------------\n") for clarity.

void p(int start, int end)

This function executes and visualizes a single disk move.

- **Purpose**: To move the topmost disk from a start tower to an end tower.
- **Steps**:
  1. It identifies the topmost disk on the start tower by iterating downwards (i from 0 to n-1) until it finds a string that is not the emptySpot.
  2. It stores this disk string in a temporary variable t and replaces its original position with emptySpot.
  3. It increments the global numSteps counter.
  4. It finds the correct position to place the disk on the end tower by iterating upwards (j from n-1 to 0) until it finds the first emptySpot.
  5. It places the disk string t at this empty spot.
  6. Finally, it calls print() to display the board after the move.

void h(int n, int start, int end)

This is the core recursive function that solves the puzzle.

Purpose: To move a stack of n disks from the start tower to the end tower.

- **Algorithm**: This function implements a non-standard recursive algorithm, using tower 2 as a fixed intermediate peg.
    - **Base Case**: If n is 1, it moves the disk from start to tower 2, and then from tower 2 to end. This takes two moves.
    - **Recursive Step**: For n > 1, it performs the following sequence of operations:
        1. h(n - 1, start, end): Recursively move n-1 disks from start to end.
        2. p(start, 2): Move the largest disk (n) from start to the middle tower (2).
        3. h(n - 1, end, start): Recursively move the n-1 disks from end back to start.
        4. p(2, end): Move the largest disk (n) from the middle tower (2) to end.
        5. h(n - 1, start, end): Recursively move the n-1 disks from start to end one last time.

int main()

This is the entry point of the program.

- **Purpose**: To drive the program execution.
- **Steps**:
    1. Prompts the user to enter the number of disks and stores it in n.
    2. Calls setup(n) to initialize the puzzle.
    3. Prints the original state of the towers.
    4. Calls h(n, 1, 3) to start the recursive solving process for moving n disks from tower 1 to tower 3.
    5. Prints the total numSteps taken to complete the puzzle.

4. Complexity Analysis

Time Complexity

The time complexity is determined by the number of single moves (p function calls) generated by the recursive function h. Each call to p(start, end) involves loops that run at most n times. Therefore, the total time complexity is the number of moves multiplied by the work done per move.

Total Time Complexity: $O(n \,.\, 3^n)$

Space Complexity

The space complexity is determined by the storage required for the towers grid and the depth of the recursion stack.

- **Towers Grid**: The towers deque is an n x 3 grid. Each string in the grid has a length proportional to n (up to 2n - 1). This results in a space requirement of $O(n \,.\, n) = O(n^2)$.
- **Recursion Stack**: The maximum depth of the recursive calls to h is n. This requires O(n) space on the call stack.

Total Space Complexity: $O(n^2)$.