Tower of Hanoi: Recursion and Algorithm Design

Joshua Terranova

September 8, 2025

Contents

1 Concept Explanation: Tower of Hanoi Rules and Logic

The Tower of Hanoi is a classic mathematical puzzle consisting of three rods and a set of disks of different sizes. The puzzle begins with all the disks neatly stacked on one rod in ascending order, forming a conical shape. The objective is to move the entire stack to another rod under the following rules:

- Only one disk may be moved at a time.
- Each move must involve taking the top disk from one of the stacks and placing it on another rod.
- No disk may be placed on top of a smaller disk.

The problem lends itself naturally to a recursive solution. For n disks:

- Move $n-1$ disks to the auxiliary rod (recursive subproblem)
- Move the largest disk to the target rod
- Move the $n-1$ disks from the auxiliary rod to the target rod (recursive subproblem)

Let $T(n)$ be the minimum number of moves required. Then:

$$T(n) = 2T(n-1) + 1$$

Closed-form: $T(n) = 2^n - 1$

Proof of Closed-Form Using Induction

Base Case: $T(1) = 2^1 - 1 = 1$

Inductive Hypothesis: Assume $T(k) = 2^k - 1$

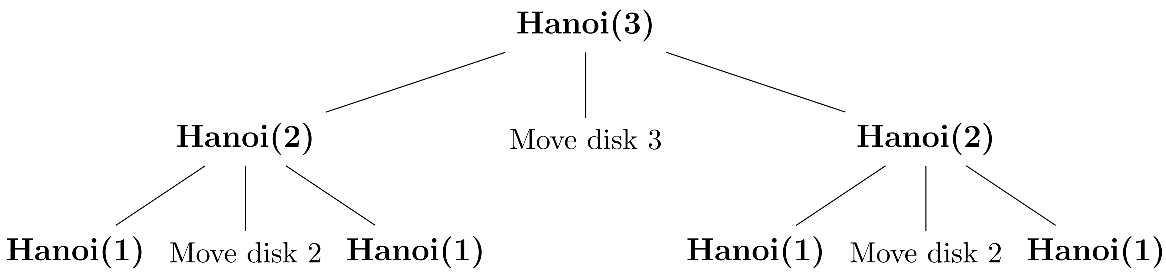
Inductive Step:

$$\begin{aligned} T(k+1) &= 2T(k) + 1 \\ &= 2(2^k - 1) + 1 \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

Conclusion: By the principle of mathematical induction, the closed-form $T(n) = 2^n - 1$ holds for all $n \geq 1$.

2 Visualizing the Recursion Tree

Below is a diagram representing the recursive breakdown of the Tower of Hanoi solution for $n = 3$. Each node represents a function call, and the arrows show the sequence of recursive steps taken to solve the subproblems.



3 Algorithm and Pseudocode

Pseudocode:

Listing 1: Recursive Hanoi Algorithm

```
Algorithm Hanoi(n, source, target, auxiliary):
    if n == 1:
        print("Move disk 1 from", source, "to", target)
    else:
        Hanoi(n-1, source, auxiliary, target)
        print("Move disk", n, "from", source, "to", target)
        Hanoi(n-1, auxiliary, target, source)
```

Explanation: The algorithm breaks down the problem recursively. Each recursive call handles one part of the movement:

- Move the top $n-1$ disks to the auxiliary rod.
- Move the bottom disk directly to the target.
- Move the $n-1$ disks onto the bottom disk.

Proof of Validity: By structural induction:

- Base case $n = 1$: move one disk directly
- Inductive step: assume it works for $n-1$, then prove it works for n

4 Code Implementation

Listing 2: Tower of Hanoi with ASCII Display

```python
# Tower of Hanoi - Recursive solution with ASCII visualization
# This script accepts user input for the number of disks and calculates the steps
    needed
# to move the disks from Peg A to Peg C following the Tower of Hanoi rules.
# It displays each step and the current state of the towers in ASCII notation.

def print_towers(towers):
    """
    Displays the current state of the towers in ASCII.

    Parameters:
    towers (dict): A dictionary with keys 'A', 'B', 'C' representing pegs,
                   and values as lists representing disks on each peg.
    """
    max_height = max(len(peg) for peg in towers.values())
    for level in reversed(range(max_height)):
        for peg in ['A', 'B', 'C']:
            if level < len(towers[peg]):
                # Print the disk number at the current level of the peg
                print(f" {towers[peg][level]} ", end="\t")
            else:
                # Print a vertical bar if no disk is present at this level
                print(" | ", end="\t")
        print()
    print(" A \t B \t C ")
    print("-" * 24)

def hanoi(n, source, target, auxiliary, towers):
    """
    Recursive function to solve Tower of Hanoi problem.

    Parameters:
    n (int): Number of disks to move.
    source (str): The peg to move disks from.
    target (str): The peg to move disks to.
    auxiliary (str): The peg used as auxiliary storage.
    towers (dict): The current state of the towers.
    """
    if n == 1:
        # Move the top disk from source to target
        disk = towers[source].pop()
        towers[target].append(disk)
        print(f"Move disk 1 from {source} to {target}")
        print_towers(towers)
    else:
        # Move n-1 disks from source to auxiliary, so they are out of the way
        hanoi(n - 1, source, auxiliary, target, towers)
        # Move the nth disk from source to target
```

```python
        disk = towers[source].pop()
        towers[target].append(disk)
        print(f"Move disk {n} from {source} to {target}")
        print_towers(towers)
        # Move the n-1 disks that we left on auxiliary to target
        hanoi(n - 1, auxiliary, target, source, towers)

def main():
    """
    Main function to run the Tower of Hanoi program.
    """
    print("Tower of Hanoi - Recursive with ASCII Visualization")
    n = int(input("Enter number of disks: "))
    # Initialize towers with disks on peg A, and pegs B and C empty
    towers = {
        'A': list(reversed(range(1, n + 1))),
        'B': [],
        'C': []
    }

    print("Initial State:")
    print_towers(towers)
    # Start the recursive solution
    hanoi(n, 'A', 'C', 'B', towers)

if __name__ == "__main__":
    main()
```

Note: Code is recursive, accepts input, prints the full move sequence, and uses ASCII visualization. If by any chance there are some errors with the code, I have provided a backup version on my GitHub page.

GitHub: github.com/thatrandomasiandev/LaTeX-Projects/tree/main

5 Worked-Out Example: n = 4

For $n = 4$, the sequence of 15 moves is:

1. Move disk 1 from A to C
2. Move disk 2 from A to B
3. Move disk 1 from C to B
4. Move disk 3 from A to C
5. Move disk 1 from B to A
6. Move disk 2 from B to C
7. Move disk 1 from A to C
8. Move disk 4 from A to B
9. Move disk 1 from C to B
10. Move disk 2 from C to A
11. Move disk 1 from B to A
12. Move disk 3 from C to B
13. Move disk 1 from A to C
14. Move disk 2 from A to B
15. Move disk 1 from C to B

6 Complexity Analysis and Efficiency

My algorithm for the Tower of Hanoi uses a recursive approach with the recurrence:

$$T(n) = 2T(n-1) + 1$$

Solving this gives the closed-form:

$$T(n) = 2^n - 1$$

So the time complexity is:

$$\mathcal{O}(2^n)$$

Reflection on Efficiency:

Although the algorithm is exponential, it is still efficient in recursion. In my implementation, each recursive call solves a distinct subproblem exactly once. For example, when calling `hanoi(n-1, ...)`, it completes that entire subproblem before moving on.

In contrast, the recursive Fibonacci algorithm repeatedly recalculates the same values. For example, `fib(5)` will recompute `fib(4)` and `fib(3)` multiple times, leading to a huge number of redundant calls.

Key Difference:

1. **My algorithm (Hanoi)**: No repeated work — each move is unique and necessary.
2. **Fibonacci**: Many repeated subproblems — highly inefficient without memorization.

Conclusion: Even though both algorithms are recursive and exponential, my Tower of Hanoi algorithm is more efficient than the recursive Fibonacci method because it avoids redundant computations.