Universidad de Cundinamarca

Estructura de Datos

R3-A2-S14 Video participativo

Javier Mateo Barrero Vanegas
Jim Alejandro Quiñones Martínez
Luis Ángel Martínez Cuenca
Harick Yesid Villarraga Rincon
Cristian Esteban Ruiz Parra

Docente: Dilia Ines Molina Cubillos

1 de noviembre de 2025

# 1. Introducción

Las Torres de Hanói (o Torre de Hanói) es un rompecabezas matemático individual inventado por el matemático francés Édouard Lucas en 1883. Consiste en una torre de discos de diferentes tamaños apilados en orden decreciente en una de tres varillas. El objetivo es trasladar toda la torre a otra varilla, respetando un conjunto estricto de reglas.

Este juego es una herramienta didáctica esencial, pues su solución óptima es inherentemente recursiva, sirviendo como el ejemplo canónico para ilustrar este poderoso concepto de la programación. El presente informe aborda su contexto histórico, reglas fundamentales y, principalmente, el análisis de su solución algorítmica mediante funciones recursivas, un pilar en la ciencia de la computación.

# 2. Objetivos

## 2.1. Objetivo General

Realizar una investigación exhaustiva sobre el rompecabezas matemático de las Torres de Hanói, comprendiendo su solución algorítmica óptima y analizando su rol fundamental como caso de estudio para la enseñanza y aplicación de la recursividad en la ciencia de la computación.

## 2.2. Objetivos Específicos

1. Documentar el origen, la leyenda y las reglas estrictas que rigen el juego de las Torres de Hanói.
2. Determinar la complejidad del problema y establecer la fórmula matemática que calcula el número mínimo de movimientos necesarios para N discos.
3. Analizar el concepto de funciones recursivas en programación, identificando sus componentes esenciales (caso base y caso recursivo).
4. Demostrar la implementación del algoritmo de las Torres de Hanói, evidenciando el uso

eficiente de la recursividad para resolver el rompecabezas.

## 3. Funciones Recursivas

Las funciones recursivas en programación son aquellas que se llaman a sí mismas para resolver un problema. Este mecanismo se basa en la estrategia "Divide y Vencerás", donde un problema complejo se divide en subproblemas más pequeños del mismo tipo.

### 3.1.Estructura y Componentes

La recursividad se define por dos partes cruciales:

1. Caso Recursivo: La función se llama a sí misma, pero trabajando sobre un caso más reducido del problema original.

2. Caso Base: Es la condición que detiene la recursión y es esencial para prevenir una llamada infinita (desbordamiento de pila o *stack overflow*). Permite que la función comience a "desenrollarse" y a devolver resultados.

Un ejemplo clásico es el cálculo del factorial de un número, donde n! = n \times (n-1)!, y el caso base se establece cuando n=0, retornando 1. Las Torres de Hanói son otro ejemplo paradigmático donde la recursividad es la técnica clave para su resolución.

### 3.2.Ventajas y Desventajas de la Recursividad

| Ventajas | Desventajas |
|---|---|
| Código Elegante: Soluciones más legibles y concisas para problemas recursivos por naturaleza (árboles, grafos). | Consumo de Memoria: Mayor uso de la memoria debido a que cada llamada recursiva se apila en la *call stack*. |
| Simplicidad Algorítmica: Facilita la implementación de algoritmos de "Divide y Vencerás". | Riesgo de *Stack Overflow*: La ausencia o incorrección del caso base provoca recursión infinita. |

| | |
|---|---|
| Claridad Conceptual: Permite modelar problemas complejos de forma intuitiva. | Posiblemente Menos Eficiente: Puede ser más lento que las soluciones iterativas, especialmente en lenguajes que no optimizan la recursión de cola. |

## 4. Las Torres de Hanói

### 4.1. Origen y Leyenda

El rompecabezas fue inventado por Édouard Lucas en 1883. El juego se popularizó con una leyenda que involucra a monjes en un templo de la India (a veces asociado a Hanói). Se les encomendó mover una torre de 64 discos de oro de un poste a otro. La leyenda profetiza que cuando terminen su tarea, el mundo se acabará.

### 4.2. Reglas del Juego

El juego se rige por tres varillas (Origen, Destino y Auxiliar) y N discos de diferentes tamaños. Las reglas son:

1. Mover un solo disco a la vez: Solo se puede mover el disco superior de una pila por jugada.
2. Principio del Tamaño: Nunca se puede colocar un disco más grande encima de uno más pequeño.
3. Uso de la varilla auxiliar: Se usan las tres varillas para completar la transferencia.

## 5. Marco Teórico

### 5.1. Estructura de la Función

La función está diseñada para mover n discos desde la varilla origen a la varilla destino, utilizando la varilla auxiliar como apoyo. Su estructura se basa estrictamente en la lógica de la recursividad:

1. Caso Base (Condición de Parada)

El caso base es el punto donde la recursión se detiene. Esto ocurre cuando solo queda un disco por mover (n=1). Implementación:

- La función verifica la condición si n es igual a 1.

Si solo hay un disco (el más pequeño), no se requiere ninguna división del problema. El movimiento se realiza directamente de la varilla de origen a la de destino, se registra el movimiento en la lista movimientos y la función termina su ejecución para esta rama.

2. Caso Recursivo (Descomposición del Problema)

Cuando n > 1, la función descompone el problema de mover N discos en tres pasos:

- Paso A: Mover Subtorre (n-1 Discos)

La primera llamada recursiva se encarga de mover la pila superior de n-1 discos. Para lograrlo, la función se llama a sí misma, intercambiando los roles de las varillas de la siguiente manera: la varilla destino se utiliza temporalmente como auxiliar y la varilla auxiliar se convierte en el destino provisional de esta subtorre.

- Paso B: Mover Disco Mayor (Disco N)

Una vez que la subtorre de n-1 discos está en la varilla auxiliar, el disco más grande (disco N) queda libre en la base de la varilla de origen. Este movimiento directo de origen a destino es fundamental y se registra como un paso simple antes de la siguiente llamada recursiva.

- Paso C: Completar el Movimiento (n-1 Discos)

La segunda llamada recursiva mueve la subtorre de n-1 discos, que ahora reside en la varilla auxiliar, hacia la varilla destino final. Para este paso, la varilla origen original se utiliza

como varilla auxiliar temporal. Al completar esta llamada, la torre completa se ha movido al destino.

### 5.2.Número Mínimo de Movimientos y Complejidad

El algoritmo recursivo implementado garantiza la solución en la cantidad mínima de movimientos posibles. Esta cantidad está determinada por la fórmula de crecimiento exponencial:

M = 2^N - 1

Donde N es el número de discos. La complejidad temporal de este algoritmo es O(2^N), lo que lo clasifica como un problema de crecimiento exponencial, como se ilustra en la tabla:

| Número de Discos (N) | Movimientos Mínimos ($2^N - 1$) |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 10 | 1023 |
| 64 (Leyenda) | 18,446,744,073,709,551,615 |

La función hanoi_visual extiende esta lógica al manipular directamente una estructura de datos que representa las varillas, permitiendo visualizar el estado del juego en cada paso de la recursión.

### 5.3.Código

```
def hanoi(n, origen, auxiliar, destino, movimientos):
        # Caso base: si solo hay 1 disco, moverlo directamente
    if n == 1:
        movimientos.append(f"Mover disco 1 de {origen} a {destino}")
        print(f"Mover disco 1 de {origen} → {destino}")
        return

    # Paso 1: Mover n-1 discos de origen a auxiliar (usando destino como
apoyo)
    hanoi(n - 1, origen, destino, auxiliar, movimientos)

    # Paso 2: Mover el disco más grande de origen a destino
    movimientos.append(f"Mover disco {n} de {origen} a {destino}")
    print(f"Mover disco {n} de {origen} → {destino}")

    # Paso 3: Mover n-1 discos de auxiliar a destino (usando origen como
apoyo)
    hanoi(n - 1, auxiliar, origen, destino, movimientos)

# Función para visualizar las torres
def visualizar_torres(torres):
    """
    Muestra el estado actual de las tres torres.
    """
    print("\n" + "="*50)

    # Encontrar la altura máxima
    altura_maxima = max(len(torres['A']), len(torres['B']),
len(torres['C']))

    # Imprimir cada nivel desde arriba hacia abajo
    for nivel in range(altura_maxima - 1, -1, -1):
        linea = ""
        for torre_nombre in ['A', 'B', 'C']:
            if nivel < len(torres[torre_nombre]):
                disco = torres[torre_nombre][nivel]
                linea += f"  {disco}  "
            else:
                linea += "  |  "
        print(linea)

    # Imprimir las bases
    print("-----" * 3)
    print("  A    B    C  ")
```

```
    print("="*50 + "\n")

# Función recursiva con visualización
def hanoi_visual(n, origen, auxiliar, destino, torres):
    """
    Resuelve Torres de Hanoi mostrando cada paso visualmente.
    """
    # Caso base: mover un solo disco
    if n == 1:
        print(f"\n→ Mover disco 1 de {origen} a {destino}")
        disco = torres[origen].pop()
        torres[destino].append(disco)
        visualizar_torres(torres)
        return

    # Paso 1: Mover n-1 discos a auxiliar
    hanoi_visual(n - 1, origen, destino, auxiliar, torres)

    # Paso 2: Mover el disco más grande a destino
    print(f"\n→ Mover disco {n} de {origen} a {destino}")
    disco = torres[origen].pop()
    torres[destino].append(disco)
    visualizar_torres(torres)

    # Paso 3: Mover n-1 discos de auxiliar a destino
    hanoi_visual(n - 1, auxiliar, origen, destino, torres)

# Programa principal
if __name__ == "__main__":
    print("="*20)
    print("  TORRES DE HANOI  ")
    print("="*20)

    # Solicitar número de discos
    n = int(input("\n¿Cuántos discos? (2-8): "))

    # Calcular movimientos mínimos necesarios
    movimientos_minimos = (2 ** n) - 1
    print(f"\nMovimientos mínimos necesarios: {movimientos_minimos}")

    # Preguntar si quiere ver la visualización
    opcion = input("\n¿Ver solución paso a paso? (s/n): ").lower()

    if opcion == 's':
        # Inicializar torres
```

```
        torres = {
            'A': list(range(n, 0, -1)),  # Torre A con todos los discos
            'B': [],                      # Torre B vacía
            'C': []                       # Torre C vacía
        }

        print("\nEstado inicial:")
        visualizar_torres(torres)

              input("Presiona Enter para comenzar...")

              # Resolver con visualización
              hanoi_visual(n, 'A', 'B', 'C', torres)

              print(f"\n¡Completado! {movimientos_minimos} movimientos
   realizados")

          else:
              # Resolver sin visualización
              movimientos = []
              print()
              hanoi(n, 'A', 'B', 'C', movimientos)
              print(f"\nTotal de movimientos: {len(movimientos)}")
```

## 6. Aplicaciones en Informática

- Enseñanza de Algoritmos: Es el modelo universal para introducir y comprender la recursividad y la estructura de la pila de llamadas (*call stack*).
- Análisis de Complejidad: Permite a los programadores analizar y experimentar con algoritmos de complejidad exponencial.
- Modelado de Problemas: Se utiliza para modelar abstractamente problemas de planificación y asignación de recursos con reglas de precedencia, donde se requiere un almacenamiento temporal.

## 7. Conclusiones

La investigación sobre las Torres de Hanói demuestra que este rompecabezas no es solo un juego, sino un profundo modelo matemático y computacional.

1. Validación de la Recursividad: El problema de Hanói valida el poder de la programación recursiva, resolviendo un problema complejo de manera elegante a través de la descomposición en subproblemas idénticos y más pequeños, haciendo del caso base (N=1) el pilar de la solución.

2. Complejidad Exponencial: Se confirmó la fórmula M = 2^N - 1, destacando la naturaleza exponencial del problema, un concepto clave para entender el rendimiento de ciertos algoritmos en el contexto de la ciencia de la computación.

3. Valor Educativo: La implementación algorítmica de las Torres de Hanói sirve como una demostración clara y práctica del funcionamiento interno de una función que se llama a sí misma, cumpliendo así el objetivo de evidenciar la recursividad.