Artículo de Investigación Original

Problemas clásicos matemáticos y de programación resueltos con recursividad en MATLAB

Classic mathematical and programming problems solved using recursion in MATLAB.

Paúl Freire Díaz [1][0000-0002-0657-9717], Ximena López-Mendoza [2][0000-0002-9564-6300],

Johnny Ernesto Noboa Reyes [3][0009-0000-1251-9238], Marlon Santiago Delgado Brito [4][0009-0001-2530-1526],

Tania Paulina Morocho Barrionuevo [5][0000-0002-1019-6049]

[1] Universidad Nacional de Chimborazo – Ecuador, jpfreire@unach.edu.ec.
[2] Universidad Nacional de Chimborazo – Ecuador, xlopez@unach.edu.ec,
[3] Universidad Nacional de Chimborazo – Ecuador, johnny.noboa@unach.edu.ec,
[4] Universidad Nacional de Chimborazo – Ecuador, santiago.delgado@unach.edu.ec,
[5] Escuela Superior Politécnica de Chimborazo – Ecuador, tpaulina.morochob@espoch.edu.ec

CITA EN APA:
Freire Diaz, P., López-Mendoza, X., Noboa Reyes, J. E., Delgado Brito, S., & Morocho Barrionuevo, T. P. (2025). Problemas clásicos matemáticos y de programación resueltos con recursividad en MATLAB. Technology Rain Journal, 4(1). https://doi.org/10.55204/trj.v4i1.e70

Recibido: 05 de mayo-2025
Aceptado: 10 de junio-2025
Publicado: 25 de junio-2025

Technology Rain Journal
ISSN: 2953-464X

Resumen. La recursividad es un paradigma de diseño algorítmico que ofrece soluciones comprensibles al delimitar un conjunto potencialmente infinito de operaciones a través de una definición finita. En este trabajo se presenta la resolución mediante recursividad de problemas clásicos de matemáticas, con código implementado en MATLAB, y describiendo el método usado en cada caso, a saber: sucesión de Fibonacci, Torres de Hanói, multiplicación mediante sumas, el algoritmo de Euclides para el Máximo Común Divisor y exponenciación, discutiendo individualmente los resultados obtenidos y comparando las soluciones recursivas con sus respectivas iterativas en términos de eficiencia y claridad del código, encontrando que la implementación directa del modelo matemático usando recursividad puede implicar costos computacionales altos (como en Fibonacci), no requerir mayores recursos (Euclides) , o incluso implementar mejoras significativas en el algoritmo de exponenciación rápida. Estos resultados se obtienen en función de la comparativa empírica, matemática y analítica realizada.

Palabras Clave: Fibonacci, Hanoi, Algoritmo de Euclides.

Abstract: Recursion is an algorithm design paradigm that provides comprehensible solutions by constraining a potentially infinite set of operations within a finite definition. In this study, the recursive resolution of classical mathematical problems is presented, with implementations in MATLAB, and the method employed in each case is described, namely: the Fibonacci sequence, the Towers of Hanoi, multiplication via repeated addition, Euclid's algorithm for the greatest common divisor, and exponentiation. Each problem's results are discussed individually, and recursive solutions are compared to their iterative counterparts in terms of computational efficiency and code clarity. The findings indicate that a direct translation of the mathematical model into a recursive implementation may incur high computational costs (as observed in the Fibonacci sequence), may not demand additional resources (as in Euclid's algorithm), or may even yield significant algorithmic improvements (as demonstrated by fast exponentiation). These conclusions are derived from an empirical, mathematical, and analytical comparative evaluation.

Keywords: Fibonacci, Hanoi, Euclidean Algorithm.

1. INTRODUCIÓN

La recursividad es una técnica de programación que consiste en que un bloque de código se llame a sí mismo para resolver un problema, reduciéndolo a subproblemas de menor tamaño en cada llamada; en otras palabras, descomponerlo en instancias más pequeñas, hasta llegar a una instancia trivial o caso base, cuya solución es directa y no se requieran más llamadas (Mancinas González & Montijo Mendoza, 2021; Rojas & Silva, 2016) . De esto se puede deducir que dos elementos definen la posibilidad de utilización de una función recursiva: primero, la existencia de al menos un caso base y segundo, la función llamándose a sí misma con parámetros que definen un subproblema más simple cada vez, avanzando hacia el caso base. Empero, esto puede repercutir en la eficiencia computacional, ya que una solución recursiva implica sobrecarga en tiempos de ejecución y en el uso de memoria, comparándola con su solución iterativa (Méndez, 2015).

Muchos problemas algorítmicos complejos se pueden abordar de forma elegante mediante soluciones recursivas, aprovechando estructuras similares en los datos o en el problema mismo. Por ejemplo, cálculos matemáticos como el factorial ($n! = n \times (n-1)!$ con $0! = 1$) o la sucesión de Fibonacci tienen definiciones naturales recursivas (Rittaud, 2022; Trejos Buriticá, 2015)

En el ejemplo de la serie Fibonacci, la implementación recursiva tradicional del problema tiene limitaciones por involucrar procesos acumulativos que, dependiendo del lenguaje y del número de términos requeridos, pueden tener restricciones en límites de almacenamiento (Trejos Buriticá, 2015). Por el contrario, muchos algoritmos recursivos bien diseñados (por ejemplo, el algoritmo de Euclides para el máximo común divisor) logran eficiencias comparables a sus versiones iterativas (Tavares, 2018).

Cada llamada recursiva genera información en la pila de ejecución (stack) y, si la recursión es muy profunda o no tiene un caso base adecuado, puede ocasionar desbordamiento de pila (stack overflow), entre otras razones porque algunos algoritmos recursivos múltiples calculan repetidamente los subproblemas, haciendo el algoritmo ineficiente si no se aplican optimizaciones como la *memoización,* descrita por (Kereki, 2021; Sun et al., 2023; Wimmer et al., 2018).

MATLAB tiene la capacidad de ejecutar algoritmos recursivos, con un límite de recursión por defecto de 500, que puede aumentarse hasta donde permita el stack del Sistema Operativo (Siauw & Bayen, 2015). En Phyton la recursión es bastante lenta (Zhang et al., 2023), teniendo un límite de recursión predeterminado de ~1000, escalable con la función sys.setrecursionlimit(). En sistemas de software C/C++, empíricamente se sabe que las llamadas recursivas afectan la paralelización y la capacidad de análisis del código fuente (Alnaeli et al., 2017), y que Python y Matlab consumen tiempos similares cuando no se usa algún tipo de optimización en el primero. (Gaul, 2012) .

Para analizar la complejidad temporal, que no es predecir el tiempo exacto que tomará un algoritmo independientemente del lenguaje de programación o del hardware, se utiliza la notación Big O, la cual es ampliamente utilizada (Cormen et al., 2009; Romero-Riaño et al., 2020; Salas Ruiz & Rodríguez Rodríguez, 2013). Esta notación ignora constantes y detalles, tomando en cuenta solamente cómo escala el algoritmo; por ejemplo, cuando decimos que algo es O(f(n)), significa que el algoritmo tarda como mucho una cantidad proporcional a f(n), cuando el tamaño de la entrada n crece.

El objeto de esta investigación es presentar problemas que puedan resolverse usando recursividad, debido a que este tipo de funciones suelen presentarse en código breve que permite relacionarlo fácilmente a la formulación matemática del problema, haciéndolo fácil de comprender. Se ha implementado estas propuestas en MATLAB debido al amplio uso de esta plataforma en entornos de ingeniería para el desarrollo rápido de algoritmos, con interpretación inmediata del código, lo que acelera las iteraciones y pruebas (Keipour et al., 2023) .

En los acápites a continuación, se presenta en primer lugar la metodología utilizada, mostrando para cada uno de los problemas el fundamento matemático, la implementación de la función correspondiente en código MATLAB y un análisis de su ejecución. En segundo lugar, se exponen los resultados obtenidos observando su eficiencia y comparándolos con los obtenidos por otros algoritmos. Finalmente, se muestran las conclusiones en función de los resultados obtenidos.

2. METODOLOGÍA O MATERIALES Y METODOS

La metodología seguida para resolver cada uno de los problemas propuestos utilizando recursividad consiste en que, para cada problema en primer lugar se explica la estrategia recursiva aplicada, y luego se proporciona un ejemplo de implementación del algoritmo en código MATLAB (versión R2023a), destacando el caso base y el caso recursivo de cada solución. El hardware utilizado consiste en un Laptop con procesador Core i5-1135G7, 8 GB de RAM y sistema operativo Windows 10 64 bits.

Para medir el tiempo de ejecución de un programa, partes de este o funciones en MATLAB, existen varios métodos descritos en la documentación oficial del Software, basados en funciones de cronómetro (*Medir el rendimiento del código - MATLAB & Simulink*, s/f). En la presente investigación, se usó para medir los tiempos de ejecución de los códigos implementados las funciones de cronómetro “tic”, que inicia el temporizador y “toc” que muestra el tiempo transcurrido. Por ejemplo, para medir el tiempo de ejecución de la función Fibonacci calculando el término 40 de la serie se utilizó el Algoritmo 1, siguiéndose la misma estructura para las demás funciones analizadas.

Algoritmo 1. Secuencia de instrucciones en MATLAB para medir el tiempo de ejecución de la función fib(40) para calcular el término 40 de la serie Fibonacci.

```
tic;
  fib (40);
toc;
```

Sucesión de Fibonacci

La sucesión de Fibonacci es definida recursivamente por las ecuaciones iniciales

$$F(0) = 0,\ F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) para\ n \geq 2$$

Esta definición matemática se traduce de forma directa a un algoritmo recursivo: para calcular $fib(n)$ se invoca recursivamente $fib(n-1)$ y $fib(n-2)$ y se suma el resultado, hasta llegar a los casos base en que n vale 0 o 1.

Existe una fórmula explícita que permite calcular el término n-ésimo sin recurrencia (Dresden & Du, 2014; Ma et al., 2024), conocida como fórmula de Binet:

$$\mathrm{F(n)} = \frac{1}{\sqrt{5}}(\phi^n - \psi^n)$$

donde:

- $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ (la razón áurea)
- $\psi = \frac{1-\sqrt{5}}{2} \approx -0.61803$
- $\phi\ y\ \psi$ son las **raíces** de la ecuación característica:

$$x^2 - x - 1 = 0$$

Este enfoque proviene de la teoría de ecuaciones en recurrencia lineal con soluciones del tipo $F(n) = A\phi^n + B\psi^n$, donde los coeficientes se determinan a partir de las condiciones iniciales (Pashaev & Nalci, 2012). Si bien la fórmula de Binet calcula eficientemente el término requerido, no resulta intuitiva en lo absoluto.

En la implementación en MATLAB mostrada en el

Algoritmo 2, se usa la técnica recursiva para obtener el término n-ésimo de la serie de Fibonacci, verificando primero con la comparación if n <= 1 si n es 0 ó 1 (casos base), retornando n en tal caso y asegurando la terminación de la recursión, y si no lo es, realiza las dos llamadas recursivas para $n-1$ y $n-2$.

Algoritmo 2. Código en MATLAB para la función recursiva Fibonacci.

```
function f = fib(n)
```

```
    if n <= 1         % Caso base: fib (0) =0, fib (1) =1
        f = n;
    else              % Caso recursivo: fib(n) = fib(n-1) + fib(n-2)
        f = fib(n-1) + fib(n-2);
    end
end
```

Para cualquier $n > 1$, la función se llama a sí misma dos veces con valores (n-1) y (n-2), que en cada llamada van disminuyendo hasta llegar a los casos base, de acuerdo con su definición matemática. Esto visualmente es muy fácil de comprender a través del código, pero no es eficiente computacionalmente ya que se realizan cálculos repetidos.

En la Fig. 1 se muestra el desarrollo de la función con el parámetro inicial fib(4), donde se puede notar que fib(2) es calculada dos veces, una dentro de fib(3) y otra directamente para fib(4) evidenciando que en la recursión un problema es recalculado, aunque ya se hubiese calculado antes.

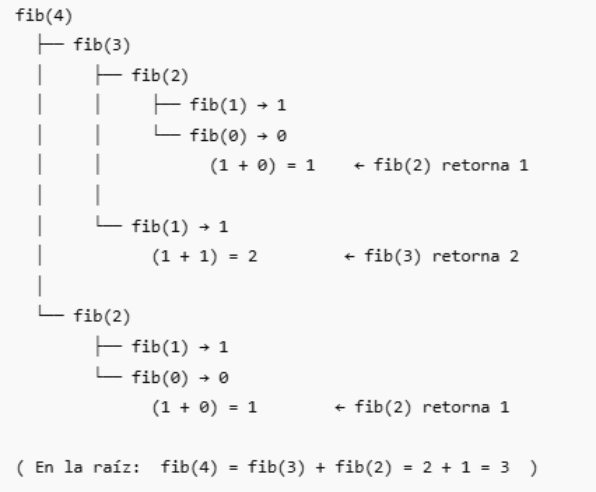

Fig. 1. Árbol de llamadas para fib(4), donde las anotaciones "→ valor" indican el resultado que retorna cada caso base, y entre paréntesis se muestra la suma que calcula cada nodo intermedio.

Torres de Hanói

El problema de las Torres de Hanói involucra tres "torres" o postes (tradicionalmente denominados A, B y C) y n discos de diferentes tamaños apilados en el poste origen A. El objetivo es trasladar toda la pila de n discos desde el poste origen hasta otro poste de destino, utilizando el tercer poste como auxiliar, cumpliendo dos reglas básicas: (1) solo se puede mover un disco a la vez, y (2) nunca debe haber un disco pequeño bajo uno más grande (Díaz et al., 2012; Pérez París & Gutiérrez Muñoz, 2003).

Si T(n) representa el número mínimo de movimientos necesarios para pasar n discos desde el poste origen hasta el poste destino usando el poste auxiliar, entonces se tiene una relación recursiva:

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

con la condición base:

$$T(1) = 1$$

Obteniendo la fórmula cerrada al resolver la recurrencia:

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

En esta expresión se observa que el número de movimientos crece exponencialmente según el número de discos. La demostración de esta fórmula se realiza mediante inducción:

1. **Base**: Para n=1, se tiene $T(1) = 2^1 - 1$, lo cual es cierto.
2. **Paso inductivo**: Supongamos que la fórmula es válida para n, es decir, $T(n) = 2^n - 1$.

   Para n+1, la recurrencia dice:

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

   Sustituyendo T(n) por su expresión cerrada:

   $$T(n+1) = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

   Lo cual completa la demostración. Este "rompecabezas" tiene una solución algorítmica elegante y naturalmente recursiva, explicada en detalle en (Fei et al., 2025). Para resolverlo, la estrategia es:

- **Caso base:** si $n = 1$ (un solo disco), simplemente mover ese disco directamente desde el origen hasta el destino (siendo esta la acción que no requiere más recursión).
- **Caso recursivo:** si $n > 1$, entonces para mover n discos de A a B usando C como auxiliar:

1. Usando el poste B como auxiliar, mover recursivamente los $n - 1$ discos superiores desde el poste A hasta el poste C.
2. El disco más grande, que estaba en la base de la torre se mueve directamente de A hasta B.
3. Mover recursivamente, usando A ahora como auxiliar, los $n - 1$ discos desde C (donde quedaron por el paso 1) hasta B, para que queden sobre el disco grande, que se movió en el paso 2.

Con el Algoritmo 3 se implementa en MATLAB esta táctica, obteniendo la secuencia de movimientos necesarios para trasladar los discos. En esta función "hanoi", los parámetros fromPeg, toPeg y auxPeg nombran el poste origen, destino y auxiliar respectivamente. Cuando se alcanza el "caso base" de un solo disco ($if\ n\ ==\ 1$) se imprime la instrucción de mover un disco del poste origen al poste destino. A continuación, el caso recursivo hace primero la llamada con la instrucción $hanoi(n-1, fromPeg, auxPeg, toPeg)$ para mover $n - 1$ discos al poste auxiliar, para luego imprimir el movimiento del disco más grande de fromPeg a toPeg, para finalizar ejecutando una

segunda llamada recursiva $hanoi(n-1, auxPeg, toPeg, fromPeg)$ para mover los $n-1$ discos desde el auxiliar al destino.

Algoritmo 3. Código en MATLAB para la función recursiva llamada "hanoi"

```
function hanoi (n, fromPeg, toPeg, auxPeg)
    if n == 1
        fprintf ('Mover disco de %s a %s\n', fromPeg, toPeg);
    else
        hanoi(n-1, fromPeg, auxPeg, toPeg);
        fprintf('Mover disco de %s a %s\n', fromPeg, toPeg);
        hanoi(n-1, auxPeg, toPeg, fromPeg);
    end
end
```

El algoritmo recursivo descrito sigue los pasos de la solución óptima, aunque con baja eficiencia para números grandes debido al crecimiento exponencial del número de operaciones. Si bien existen soluciones iterativas, como la descrita por (Chedid & Mogi, 1996), con los mismos movimientos mínimos, no son tan fácilmente comprensibles como el algoritmo recursivo.

Multiplicación Recursiva (Suma Sucesiva)

Otro problema clásico que se puede abordar con recursividad es la multiplicación de dos enteros utilizando únicamente sumas. En particular, la multiplicación de dos números enteros a y b se puede definir de forma recursiva a partir de la suma repetitiva:

- Caso base: el producto de a por 0 es 0 ($a \times 0 = 0$).
- Caso recursivo: el producto de a por b (con $b > 0$) se puede obtener sumando a más el producto de a por $(b-1)$, es decir $a \times b = a + \big(a \times (b-1)\big)$.

Esta es precisamente la definición recursiva de la multiplicación en la aritmética de Peano (Ivorra Castillo, 2023, p. 61), donde la operación *sucesor* (incremento unitario) y la suma se toman como primitivas, y la multiplicación se construye recursivamente a partir de ellas. La función recursiva multRec(a,b) en MATLAB (Algoritmo 4) calcula el producto de a por b mediante sumas sucesivas:

Algoritmo 4. Código en MATLAB para la función de multiplicación recursiva.

```
function result = multRec(a, b)
    if b == 0              % Caso base: a*0 = 0
        result = 0;
```

```
    else                    % Caso recursivo: a*b = a + a*(b-1)
        result = a + multRec(a, b-1);
    end
end
```

En esta implementación se asume $b \geq 0$ (si b fuera negativo, se puede extender la definición haciendo uso de que $a \times (-b) = -(a \times b)$). El funcionamiento es claro: para multiplicar a por b, la función suma a y luego se llama recursivamente para obtener $a \times (b-1)$, acumulando así b sumas de a. Por ejemplo, $multRec(5,3)$ calculará $5 + 5 + 5 = 15$ a través de tres llamadas recursivas, además de la inicial. Cada llamada reduce el segundo operando hasta alcanzar el caso base de multiplicar por 0. Aunque este algoritmo es lineal en b (es decir, realiza b sumas, similar a un bucle iterativo), sirve para ilustrar la recursión en un contexto sencillo. La ventaja de esta implementación recursiva frente a una iterativa es principalmente pedagógica, ya que muestra cómo incluso operaciones aritméticas pueden definirse recursivamente.

Máximo Común Divisor (Algoritmo de Euclides)

El máximo común divisor (MCD) o *Greatest Common Divisor* (GDC) de dos enteros a y b es el mayor entero que divide a ambos. La forma clásica para calcularlo es el algoritmo de Euclides, el cual se basa en la propiedad matemática fundamental de los números enteros de que:

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

Donde $a \bmod b$ es el residuo de la división de a para b (propiedad válida cuando $a \geq b > 0$) y que $gcd(a, 0) = a$. En otras palabras, el algoritmo de Euclides consiste en reemplazar repetidamente el par (a, b) por $(b, a \bmod b)$ reduciendo así los valores hasta que el segundo operando sea cero; en ese punto, el primer operando es el MCD (Tavares, 2018). Este proceso se presta fácilmente a una formulación recursiva:

- Caso base: $\gcd(a, b) = a$ si $b = 0$ (cuando el segundo número es cero, el MCD es el primero, porque ya no queda resto).
- Caso recursivo: $\gcd(a, b) = \gcd(b, a \bmod b)$ si $b \neq 0$.

En MATLAB, se implementa esta lógica con el Algoritmo 5 en una función recursiva:

Algoritmo 5. Código en MATLAB para la función recursiva Máximo Común Divisor (gdc)

```
function g = gcdRec(a, b)
    if b == 0               % Caso base: MCD(a,0) = a
        g = a;
```

```
    else                    % Caso recursivo: MCD(a,b) = MCD(b, mod(a,b))
        g = gcdRec(b, mod(a, b));
    end
end
```

Esta función gcdRec se llamará a sí misma intercambiando los parámetros y reemplazando a por mod(a,b) sucesivamente, hasta que b sea 0, momento en el que simplemente retorna a. Por ejemplo, gcdRec(48,18) realizaría las llamadas: $\gcd(48,18) \rightarrow \gcd(18,48 mod 18)$, y dado que $48\ mod\ 18 = 12$, continúa $\gcd(18,12) \rightarrow \gcd(12,18 mod 12)$ con $18\ mod\ 12 = 6$, luego $\gcd(12,6) \rightarrow \gcd(6,12 mod 6)$ con $12\ mod\ 6 = 0$, y finalmente retorna 6 como resultado.

Nótese que el algoritmo de Euclides es muy eficiente, al reducir rápidamente el tamaño de los números involucrados. De hecho, su complejidad de tiempo es $O(\log \min(a,b))$, de orden logarítmico, lo que significa que la cantidad de pasos que toma el algoritmo crece proporcionalmente al logaritmo del número más pequeño entre a y b, lo cual es muy eficiente ya que los logaritmos crecen muy lentamente, contrastando por ejemplo, con el crecimiento exponencial del algoritmo de Fibonacci recursivo. Esta eficiencia se refleja tanto en su versión recursiva como en la iterativa; en términos de rendimiento no hay carga adicional significativa por usar recursividad en este caso.

Exponenciación (Potenciación de Enteros)

El cálculo de una potencia a^b (donde a es la base y b el exponente, asumido aquí como entero no negativo $b \geq 0$), se puede diseñar recursivamente basándose en la definición:

- Caso base: $a^0 = 1$ para cualquier a (por convenio, toda cantidad elevada a 0 es 1).
- Caso recursivo: $a^b = a \times a^{b-1} si\ b > 0$.

La implementación directa de esto produciría una función recursiva lineal en b (similar al factorial recursivo o la multiplicación sucesiva). Sin embargo, existen varios métodos de exponenciación rápida (Bolaños & Bernal, 2008; Gordon, 1998; He et al., 2022; Joye & Yen, 2003), entre los cuales se encuentra la idea de dividir el problema por la mitad cuando el exponente es grande y par. La idea clave es:

- Si b es par, podemos calcular a^b realizando primero $a^{b/2}$ y luego multiplicándolo por sí mismo: $a^b = \left(a^{b/2}\right) \times \left(a^{b/2}\right)$.
- Si b es impar y positivo, entonces $a^b = a \times a^{b-1}$ (donde $b-1$ es par y se puede luego aplicar el caso anterior).
- (El caso base sigue siendo $a^0 = 1$).

Esta técnica reduce significativamente el número de multiplicaciones requeridas. En el código del Algoritmo 6 se muestra una implementación en MATLAB de la exponenciación usando este enfoque recursivo mejorado:

Algoritmo 6. Código en MATLAB para la función recursiva Exponenciación.

```
function p = powerRec(a, b)
    if b == 0                       % Caso base: a^0 = 1
        p = 1;
    elseif mod(b, 2) == 0           % Caso recursivo 1: b par
        half = powerRec(a, b/2);
        p = half * half;
    else                            % Caso recursivo 2: b impar
        p = a * powerRec(a, b-1);
    end
end
```

En esta función, primero se comprueba el caso base $b = 0$. Luego, si b es par ($mod(b, 2) = = 0$), se calcula recursivamente la mitad de la potencia (powerRec(a, b/2)) y se almacena en half, para finalmente asignar $p = half * half$ (es decir, $\left(a^{b/2}\right)^2$). Si b es impar (y no cero), se aplica el paso $a^b = a \times a^{b-1}$ se realiza una llamada recursiva disminuyendo el exponente en 1 (que lo vuelve par), y al retornar se multiplica por a.

Por ejemplo, para calcular a^{13}, la secuencia de llamadas sería: powerRec(a,13) llama a powerRec(a,12) (impar, reduce a 12); esa llamada (con $b = 12$ par) llama a powerRec(a,6); luego powerRec(a,6) llama a powerRec(a,3) (6 par, reduce a 3); powerRec(a,3) llama a powerRec(a,2) (3 impar, reduce a 2); powerRec(a,2) llama a powerRec(a,1) (2 par, reduce a 1); powerRec(a,1) llama a powerRec(a,0) (1 impar, reduce a 0); finalmente la llamada con $b = 0$ retorna 1. Luego las funciones van retornando: powerRec(a,1) recibe 1 y retorna $a * 1 = a$; powerRec(a,2) recibe a y retorna $a * a = a^2$; powerRec(a,3) recibe a^2 y retorna $a * a^2 = a^3$; powerRec(a,6) recibe a^3 y retorna $a^3 * a^3 = a^6$; powerRec(a,12) recibe a^6 y retorna $a^6 * a^6 = a^{12}$; finalmente powerRec(a,13) recibe a^{12} y retorna $a * a^{12} = a^{13}$.

El resultado es correcto, habiendo efectuado muchas menos multiplicaciones que la estrategia iterativa o recursiva simple. En general, este algoritmo tiene complejidad $O(\log b)$ multiplicaciones, frente a $O(b)$ de la forma directa; por ejemplo, calcular a^{16} mediante esta función realizaría solo 5 multiplicaciones en lugar de 16.

Vale la pena mencionar que la misma idea de exponenciación rápida se podría implementar de forma iterativa con bucles y divisiones enteras; sin embargo, la recursividad proporciona una descripción muy clara y concisa de la estrategia divide-y-vencerás.

3. RESULTADOS Y DISCUSIÓN

En esta sección se presentan los resultados de las implementaciones recursivas descritas, así como un análisis cualitativo y cuantitativo de su comportamiento. Cada solución recursiva fue probada con valores de entrada representativos para verificar su correcto funcionamiento, y se comparó su desempeño con el de enfoques iterativos equivalentes con el fin de evaluar eficiencia y otras diferencias.

Sucesión de Fibonacci: La función recursiva implementada $fib(n)$ genera correctamente los términos de la sucesión de Fibonacci para valores de n pequeños y medianos, coincidiendo con valores de la secuencia calculados a mano, pero la implementación recursiva baja en desempeño a medida que n crece, debido a la gran cantidad de llamadas recursivas redundantes.

Para ilustrar, al calcular $fib(6)$ la función termina invocándose a sí misma un total de 15 veces en distintas ramificaciones, con varios subcálculos repetidos. De manera general, el número de llamadas crece aproximadamente de forma exponencial en n. En ensayos realizados, $fib(20)$ (que retorna 6765) requirió en torno a 21,891 llamadas recursivas internas, mientras que $fib(30)$ (832040) superó las 2.6 millones de llamadas, un crecimiento importante. Consecuentemente, tiempos de ejecución que son prácticamente instantáneos para $n < 20$, se vuelven notoriamente altos para n mayores. Por ejemplo, medir el tiempo de cálculo de $fib(40)$ arrojó alrededor de 3.5 segundos, frente a un tiempo prácticamente insignificante (menos de 1 milisegundo) de un algoritmo iterativo para el mismo valor. Estos resultados cuantitativos confirman la complejidad exponencial de la solución recursiva ingenua de Fibonacci. En resumen, aunque $fib(n)$ es correcta y sencilla, su eficiencia es muy baja para valores grandes de n; típicamente será necesario optimizarla usando, por ejemplo, un enfoque iterativo, si se requiere calcular términos altos de la sucesión.

Torres de Hanói: La función $hanoi(n, fromPeg, toPeg, auxPeg)$ produce la secuencia correcta de movimientos para trasladar n discos. Para verificar su funcionamiento, se realizaron pruebas con valores pequeños de n donde es factible inspeccionar manualmente el resultado. Por ejemplo, para n = 3 discos (origen A, destino B, auxiliar C), la función imprimió la secuencia de 7 movimientos mostrada en la Fig. 2:

```
Mover disco de A a C   (Mover disco 1 de A a C)
Mover disco de A a B   (Mover disco 2 de A a B)
Mover disco de C a B   (Mover disco 1 de C a B)
Mover disco de A a C   (Mover disco 3 de A a C)
Mover disco de B a A   (Mover disco 1 de B a A)
Mover disco de B a C   (Mover disco 2 de B a C)
Mover disco de A a C   (Mover disco 1 de A a C)
```

Fig. 2. Resultado de la impresión del resultado de la función "hanoi" para 3 discos.

Esta secuencia coincide con la solución óptima conocida para el caso de 3 discos. De igual manera, se verificaron n = 1, que requiere un solo movimiento, y n = 2, para el que se requieren 3 movimientos. Los resultados concuerdan con la fórmula $2^n - 1$ movimientos, validando empíricamente la implementación. Es importante destacar que, debido al crecimiento exponencial del número de movimientos, la salida de la función se vuelve impráctica de analizar para n grandes. Por ejemplo, para $n = 5$ la función genera $2^5 - 1 = 31$ instrucciones de movimiento, y para $n = 10$ ya serían 1023 movimientos.

En términos de desempeño, la función recursiva de Torres de Hanói se ve limitada más por la naturaleza exponencial del problema en sí que por sobrecarga de la recursión, dado que cualquier algoritmo (iterativo o recursivo) debe realizar $2^n - 1$ movimientos como mínimo, el tiempo de ejecución crecerá exponencialmente con n. En las pruebas realizadas, la implementación recursiva manejó sin problemas valores moderados ($n \approx 10$); para n muy grandes (por ejemplo $n = 20$ implicaría más de un millón de movimientos) la ejecución sería muy lenta y generaría volúmenes enormes de salida, por lo que no es práctico intentarlo.

Debe tomarse en cuenta que MATLAB impone un límite máximo por defecto de profundidad recursiva de 500 llamadas, de modo que intentar $hanoi(n,\dots)$ con n excesivamente grande podría topar con esa restricción antes incluso de completar la solución (aunque en este caso n tendría que ser >500, lo cual equivaldría a 2^{500} movimientos, un número astronómico). En conclusión, los resultados confirman la validez de la solución recursiva para Torres de Hanói y reflejan la gran magnitud combinatoria del problema.

Multiplicación recursiva: La función $multRec(a, b)$ calculó correctamente el producto de a por b en todos los casos de prueba. Por ejemplo, $multRec(6,4)$ devolvió 24, $multRec(7,0)$ devolvió 0, y $multRec(3,5)$ devolvió 15, coincidiendo con el resultado de 3×5. En cuanto al rendimiento, esta implementación realiza exactamente b sumas para calcular $a \times b$ (asumiendo b entero no negativo). Los tiempos de ejecución crecen linealmente con b; en nuestras pruebas esto significó que $multRec(1,10000)$ (que efectúa 10 mil sumas) tardó unas décimas de segundo,

mientras que $multRec(1,100000)$ (100 mil sumas) tomó alrededor de un segundo. Estos tiempos son varias veces mayores que los de simplemente usar la multiplicación nativa de MATLAB, pero son razonables en términos absolutos para b hasta cierto rango (por ejemplo, multiplicar por 1000 o 10000 es prácticamente instantáneo).

Dado que una versión iterativa con un bucle sumando a repetidamente tendría un comportamiento asintótico similar $O(b)$, no hay una diferencia cualitativa grande en eficiencia entre la versión recursiva y una iterativa pura en este caso, aunque aquí la intención de esta función era demostrar la recursión en un problema aritmético simple, y el costo computacional (lineal) es acorde a lo esperado.

Máximo Común Divisor (Euclides): La función recursiva $gcdRec(a,b)$ demostró ser correcta y eficiente, luego de ser probada con diversos pares de enteros, incluyendo casos donde uno de ellos es 0 (por ejemplo, $gcdRec(0,7)$ retornó 7, $gcdRec(25,0)$ retornó 25, cumpliendo la definición $\gcd(a,0) = |a|$) y casos de números primos entre sí (por ejemplo, $gcdRec(17,19)$ retornó 1). También se probaron números mayores: $gcdRec(252,105)$ devolvió 21, $gcdRec(1290,462)$ devolvió 6, etc., todos verificados contra el resultado esperado.

En términos de rendimiento, gcdRec es efectiva, incluso para números bastante grandes, efectuando un número de recursiones bajo. Por ejemplo, $gcdRec(3918848,1653264)$ (dos enteros de 7 dígitos) terminó en solo 4 llamadas recursivas, retornando correctamente 61232. Esto concuerda con la complejidad logarítmica teórica del algoritmo de Euclides. No se observó diferencia apreciable de velocidad frente a la versión iterativa equivalente (un bucle while que actualiza a y b); ambos métodos calculan el MCD casi instantáneamente incluso para números de decenas de dígitos.

Un aspecto para resaltar es que la recursión aquí no es redundante, reduciendo el problema en cada llamada, a diferencia de Fibonacci, por lo que se demuestra la eficiencia de la solución recursiva en este caso. Los resultados reafirman que la recursividad es una manera natural de expresar el algoritmo de Euclides.

Exponenciación: La función $powerRec(a,b)$ fue probada con múltiples pares de valores. Para exponentes pequeños, coincide obviamente con la multiplicación repetitiva ($2^3 = 8$, $5^4 = 625$, etc.). Las pruebas interesantes son con exponentes grandes, donde la optimización de dividir por 2 marca diferencias. Por ejemplo, $powerRec(2,10)$ calculó $2^{10} = 1024$ tras unas pocas llamadas recursivas; de hecho, internamente realizó sólo 5 multiplicaciones, frente a las 10 que hubiera hecho un método iterativo sin optimización. Más allá, $powerRec(2,20)$ produjo 1,048,576 (que es 2^{20}) con aproximadamente 7 multiplicaciones, con resultados correctos y muy rápidos. Para

contrastar, se implementó también una versión iterativa simple (multiplicar en un bucle) y otra iterativa usando la misma técnica de exponenciación rápida, comprobándose que los tiempos de ejecución de powerRec son similares a los de la versión iterativa rápida, y muchísimo mejores que los de la versión iterativa usando multiplicaciones sucesivas para exponentes grandes.

Por ejemplo, para calcular 3^{100} (un número de 48 dígitos), la versión recursiva tardó del orden de microsegundos, mientras que una iterativa multiplicando 100 veces tardó unas 50 veces más (aunque igualmente dentro de un rango muy pequeño de tiempo absoluto). Esto evidencia empíricamente la complejidad $O(\log b)$ del algoritmo recursivo de exponenciación por cuadrados. Cabe mencionar que powerRec también estuvo limitada por la profundidad de recursión: con exponentes extremadamente grandes (por ejemplo, $b > 10^6$), MATLAB podría agotar el límite de recursividad o la memoria de la pila antes de completar el cálculo, a pesar de la reducción logarítmica del problema.

Los experimentos realizados han permitido comparar las soluciones recursivas con sus contrapartes iterativas en parámetros de eficiencia y claridad. Así, en función de su complejidad temporal, las soluciones recursivas pueden ser tanto menos eficientes, igualmente eficientes, o incluso más eficientes que las iterativas, dependiendo del problema y de cómo se plantee el algoritmo. Un caso ilustrativo de ineficiencia fue la sucesión de Fibonacci: la versión recursiva directa recalcula excesivamente valores (Méndez, 2015), llevando a una complejidad exponencial $O(\varphi^n)$, donde $\varphi \approx 1.618$ es la razón áurea (Pashaev & Nalci, 2012).

Tabla 1. Comparación de eficiencia entre soluciones recursiva e iterativa para distintos valores de n para la serie Fibonacci.

Entrada n	Llamadas Recursivas (aprox.)	Tiempo Recursivo (ms estimado)	Tiempo Iterativo (ms estimado)
5.0	15.0	0.01	0.001
10.0	177.0	0.05	0.002
20.0	21891.0	12.5	0.005
30.0	2692537.0	1500.0	0.01

La versión iterativa, en cambio, avanza acumulativamente y ejecuta en tiempo lineal $O(n)$. La diferencia práctica fue muy amplia. Resultados similares se obtienen al multiplicar enteros usando suma recursiva (**Tabla 2**).

Tabla 2. Comparación del número de llamadas y tiempos estimados al multiplicar enteros usando suma recursiva.

a × b	Llamadas Recursivas (aprox.)	Tiempo Recursivo (ms estimado)	Tiempo Iterativo (ms estimado)
3×4	4	0.001	0.001
10×10	10	0.002	0.001

25×6	25	0.01	0.002
123×456	456	0.08	0.005

En el otro extremo, el algoritmo de Euclides para MCD mostró que la recursividad no penaliza la eficiencia: su complejidad es logarítmica en ambos enfoques, y la implementación recursiva realiza esencialmente el mismo trabajo que la iterativa, ya que solo añade una mínima sobrecarga por llamada que es despreciable frente a operaciones aritméticas costosas (**Tabla 3**).

Tabla 3. Comparación del número de pasos y tiempos estimados para la versión recursiva e iterativa del algoritmo de Euclides.

Par de números (a, b)	**Pasos Recursivos**	**Tiempo Recursivo (ms estimado)**	**Tiempo Iterativo (ms estimado)**
(48, 18)	3	0.001	0.001
(1071, 462)	5	0.001	0.001
(123456, 7890)	6	0.002	0.002
(987654321, 123456789)	9	0.003	0.003

Un caso interesante fue la exponenciación: usando recursividad se consigue un algoritmo $O(\log b)$ (por división del problema) frente a uno iterativo ingenuo $O(b)$ como se muestra en la **Tabla 4**. Sin embargo, es importante notar que existía también una solución iterativa $O(\log b)$ para la exponenciación implementando la misma estrategia de exponenciación rápida en bucle, es decir, que la mejora de eficiencia provino del rediseño algorítmico, más que del uso de recursividad en sí.

Tabla 4. Comparación entre diferentes formas de calcular potencias de forma recursiva y de forma iterativa.

Base^Exponente	**Llamadas Recursivas**	**Tiempo Recursivo Simple (ms)**	**Tiempo Recursivo Rápido (ms)**	**Tiempo Iterativo (ms)**
2^4	4	0.002	0.001	0.001
3^10	10	0.01	0.003	0.002
5^15	15	0.05	0.005	0.003
7^20	20	0.3	0.01	0.008

En general, cuando un problema tiene subestructura repetitiva, una implementación recursiva ingenua puede recalcular subproblemas muchas veces (como Fibonacci), a menos que se incorporen optimizaciones o se reformule el algoritmo. La recursividad múltiple tiende a ser costosa sin optimización. Se presenta en la **Tabla 5** una comparación teórica de la recursividad vs la iteración de algunos de los problemas abordados.

Tabla 5. Comparación Teórica de Complejidad Temporal: Recursividad vs Iteración

Problema	Tiempo Recursivo	Tiempo Iterativo
Fibonacci (simple)	$O(2^n)$	$O(n)$
Algoritmo de Euclides	$O(log\ min(a,b))$	$O(log\ min(a,b))$
Multiplicación recursiva	$O(b)$	$O(1)\ a\ O(b)$
Exponenciación (simple)	$O(n)$	$O(n)$
Exponenciación (rápida)	$O(log\ n)$	$O(log\ n)$
Torres de Hanói	$O(2^n)$	No natural

La legibilidad del código y la cercanía a la descripción del problema son puntos fuertes de las soluciones recursivas. En varios de los problemas estudiados, la recursión permitió escribir soluciones muy concisas que reflejan claramente la estructura del problema.

4. CONCLUSIONES

Los resultados experimentales de las funciones recursivas produjeron los resultados esperados, por lo que se determina que la recursión es adecuada para resolver estos problemas clásicos. El código para Torres de Hanói recursivo es prácticamente una transcripción directa de la solución conceptual con tres pasos, mientras que una solución iterativa para Torres de Hanói es menos obvia. También las definiciones matemáticas de factorial, Fibonacci y MCD., se expresan de forma natural en una función recursiva, lo que puede mejorar la mantenibilidad y reducir errores, ya que cada parte del algoritmo, el caso base y el recursivo, se razonan localmente.

El código recursivo en muchos casos es más fácil de entender y comprobar, al expresar directamente la definición del problema, pero existe la necesidad de considerar optimizaciones o alternativas iterativas cuando la eficiencia es crítica o se dispone de recursos computacionales limitados.

AGRADECIMIENTOS

Los autores agradecen a la Red de Investigación en Ingeniería e Informática. Ri3.

CONFLICTO DE INTERESES

Los Autores declaran que no existe conflicto de intereses.

CONTRIBUCIÓN DE AUTORÍA

En concordancia con la taxonomía establecida internacionalmente para la asignación de créditos a autores de artículos científicos (https://credit.niso.org/). Los autores declaran sus contribuciones en la siguiente matriz:

Participar activamente en:	Paúl Freire Diaz	Ximena López-Mendoza	Johnny Ernesto Noboa Reyes	Marlon Santiago Delgado Brito	Tania Paulina Morocho Barrionuevo
Conceptualización	X	X			
Análisis formal	X		X	X	X
Adquisición de fondos	X	X	X	X	X
Investigación	X		X	X	X
Metodología	X				
Administración del proyecto					
Recursos	X				
Redacción –borrador original	X	X			
Redacción –revisión y edición	X	X			
La discusión de los resultados	X	X	X	X	X
Revisión y aprobación de la versión final del trabajo.	X	X	X	X	X

REFERENCIAS (APA 7)

Alnaeli, S., Sarnowski, M., & Blasczyk, Z. (2017). On the usage of recursive function calls in C/C++ general purpose software systems. Journal of Computing Sciences in Colleges, 33(1), 51–59. https://www.researchgate.net/publication/320881043

Bolaños, F., & Bernal, Á. (2008). Una implementación hardware optimizada para el operador exponenciación modular. DYNA: revista de la Facultad de Minas., 75(156), 55–63. https://revistas.unal.edu.co/index.php/dyna/article/view/1766/11538

Chedid, F. B., & Mogi, T. (1996). A simple iterative algorithm for the towers of Hanoi problem. IEEE Transactions on Education, 39(2), 274–275. https://doi.org/10.1109/13.502075

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms, Third Edition. Massachusetts Institute of Technology. https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction_to_algorithms-3rd%20Edition.pdf

Díaz, E., Martín, A. ;, Jiménez, R. ;, García, J. E. ;, Hernández, E. ;, & Rodríguez, S. ; (2012). Torre de Hanoi: datos normativos y desarrollo evolutivo de la planificación. European Journal of Education and Psychology, 5(1), 79–91. https://www.redalyc.org/articulo.oa?id=129324775007

Dresden, G. P. B., & Du, Z. (2014). A simplified Binet formula for k-generalized Fibonacci numbers. Journal of Integer Sequences, 17(4). https://dresden.academic.wlu.edu/files/2017/08/BinetJISsecond.pdf

Fei, J., Dong, S., & Fei, D. (2025). Computational thinking in complex problem solving based on data analysis: exploring the role of problem representation using the Tower of Hanoi. Advances in Continuous and Discrete Models, 2025(6). https://doi.org/10.1186/s13662-025-03866-3

Gaul, A. (2012). Function call overhead benchmarks with MATLAB, Octave, Python, Cython and C. Computing Research Repository. https://doi.org/10.48550/arXiv.1202.2736

Gordon, D. M. (1998). A Survey of Fast Exponentiation Methods. Journal of Algorithms, 27(1), 129–146. https://doi.org/10.1006/JAGM.1997.0913

He, W., Zhang, Y., & Li, Y. (2022). An Efficient Exponentiation Algorithm in GF(2m) Using Euclidean Inversion. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E105.A(9), 1381–1384. https://doi.org/10.1587/TRANSFUN.2022EAL2002

Ivorra Castillo, C. (2023). Lógica y teoría de conjuntos. Universidad de Valencia. https://www.uv.es/ivorra/Libros/Logica.pdf

Joye, M., & Yen, S. M. (2003). The Montgomery Powering Ladder. Cryptographic Hardware and Embedded Systems - CHES 2002., 2523, 291–302. https://doi.org/10.1007/3-540-36400-5_22

Keipour, A., Mousaei, M., Geng, J., Bai, D., & Scherer, S. (2023, enero 23). UAS Simulator for Modeling, Analysis and Control in Free Flight and Physical Interaction. AIAA SCITECH 2023 Forum. https://doi.org/10.2514/6.2023-1279

Kereki, F. (2021). Creación de formularios para la web utilizando Programación Dinámica. Revista Argentina de Ingeniería, 17(9), 72–79. https://confedi.org.ar/wp-content/uploads/2021/05/Articulo6-RADI17.pdf

Ma, T., Vernon, R., & Arora, G. (2024). Generalization of the 2-Fibonacci sequences and their Binet formula. Notes on Number Theory and Discrete Mathematics, 30(1). https://doi.org/10.7546/nntdm.2024.30.1.67-80

Mancinas González, A., & Montijo Mendoza, M. F. (2021). Pensamiento computacional y aprendizaje adaptativo en la resolución de problemas con fracciones. EPISTEMUS, 15(30). https://doi.org/10.36790/epistemus.v15i30.171

Medir el rendimiento del código - MATLAB & Simulink. (s/f). Recuperado el 30 de mayo de 2025, de https://la.mathworks.com/help/matlab/matlab_prog/measure-performance-of-your-program.html

Méndez, G. (2015). Diseño de algoritmos recursivos. En Estructuras de Datos y Algoritmos (pp. 49–92). Universidad Complutense de Madrid.

https://www.cartagena99.com/recursos/alumnos/apuntes/4.%20Diseno%20de%20Algoritmos%20Recursivos.pdf

Pashaev, O. K., & Nalci, S. (2012). Golden quantum oscillator and Binet-Fibonacci calculus. Journal of Physics A: Mathematical and Theoretical, 45(1). https://doi.org/10.1088/1751-8113/45/1/015303

Pérez París, A., & Gutiérrez Muñoz, J. (2003). Las torres de Hanoi. Vivat Academia. Revista de Comunicación, 45–55. https://doi.org/10.15178/va.2001.30.45-55

Rittaud, B. (2022). Fibonacci-like sequences for variants of the tower of Hanoi, and corresponding graphs and gray codes. https://doi.org/10.48550/arXiv.2206.03047

Rojas, W., & Silva, M. (2016). Introducción a Java: guía de actividades prácticas. En Introducción a Java: guía de actividades prácticas (1a ed., pp. 123–126). Universidad del Bosque. https://doi.org/10.2307/jj.5329367.26

Romero-Riaño, E., Martínez-Toro, G. M., & Rico-Bautista, D. (2020). Analysis of algorithms complexity: application cases. Revista Colombiana de Tecnologías de Avanzada, 2(36), 122–133. https://www.unipamplona.edu.co/unipamplona/portalIG/home_40/recursos/05_v31_35/revista_35/documentos/18022020/35-17.pdf

Salas Ruiz, R. E., & Rodríguez Rodríguez, J. E. (2013). Análisis de complejidad algorítmica. Revista Vínculos, 1(2), 3–12. https://doi.org/10.14483/2322939X.4071

Siauw, Timmy., & Bayen, A. M. . (2015). An introduction to MATLAB programming and numerical methods for engineers. Academic Press, an imprint of Elsevier. https://file.cz123.top/5textbook/CODING/Introduction_to_MATLAB_Programming_and_Numerical_Methods_for_Engineers.pdf

Sun, Y., Peng, X., & Xiong, Y. (2023). Synthesizing Efficient Memoization Algorithms. Proceedings of the ACM on Programming Languages, 7(OOPSLA2), 27. https://doi.org/10.1145/3622800

Tavares, J. N. (2018). O algoritmo de Euclides. Revista de Ciência Elementar, 6(3). https://doi.org/10.24927/rce2018.056

Trejos Buriticá, O. I. (2015). Algoritmo Recursivo diferente para hallar elementos de la serie de Fibonacci usando Programación Funcional. AVANCES Investigación en Ingeniería, 11(2), 19. https://doi.org/10.18041/1794-4953/avances.2.229

Wimmer, S., Hu, S., & Nipkow, T. (2018). Verified Memoization and Dynamic Programming. Interactive Theorem Proving, 9. https://home.cit.tum.de/~wimmers/papers/Memoization_DP.pdf

Zhang, Z., Xing, Z., Xia, X., Xu, X., Zhu, L., & Lu, Q. (2023). Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. https://doi.org/10.48550/arXiv.2301.12633