Tareas de métodos no médicos (/github/kenichi-50/Tareas-de-Metodos-N-mericos/tree/main)
/
Algoritmos y Complejidad.ipynb (/github/kenichi-50/Tareas-de-Metodos-N-mericos/tree/main/Algoritmos y Complejidad.ipynb)

TAREA N° 3

Nombre:

Joel Stalin Tinitana Carrión

Fecha:

05/05/2025

Tema:

Algoritmos y complejidad

CONJUNTO DE EJERCICIO 1.3

1. Utilice aritmética de corte de tres dígitos para calcular las siguientes sumas.
Para cada parte, ¿qué método es más preciso y por qué?

a.

$$\sum_{yo=1}^{10}\left(\frac{1}{i^2}\right)$$

Primero por:

$$\frac{1}{1}+\frac{1}{4}+\cdots+\frac{1}{100}$$

y luego por:

$$\frac{1}{100}+\frac{1}{81}+\cdots+\frac{1}{1}$$

b.

$$\sum_{yo=1}^{10}\left(\frac{1}{i^3}\right)$$

Primero por:

$$\frac{1}{1}+\frac{1}{8}+\frac{1}{27}+\cdots+\frac{1}{1000}$$

y luego por:

$$\frac{1}{1000}+\frac{1}{729}+\cdots+\frac{1}{1}$$

Sumas con Aritmética de Corte de Tres Dígitos

El cálculo de dos sumatorias utilizando aritmética de corte a tres cifras significativas.

Se comparan los resultados obtenidos al realizar la suma de forma:

- Directa: desde (i = 1) hasta (i = 10)

- Inversa: desde (i = 10) hasta (i = 1)

Las sumas a evaluar son:

- $$\sum_{yo=1}^{10} \frac{1}{i^2}$$
- $$\sum_{yo=1}^{10} \frac{1}{i^3}$$

Método: Aritmética de Corte de Tres Dígitos

El método consiste en cortar (no redondear) todos los valores intermedios y resultados a **tres cifras significativas** .

Ejemplo:

$$\frac{1}{3^2} = \frac{1}{9} = 0.1111\ldots \Rightarrow \text{corte a 3 cifras} = 0.111 + +$$

El procedimiento se implementa con funciones que realiza esta operación en cada paso.

Pseudocódigo: Aritmética de Corte de Tres Dígitos

```
pseudo
Definir función corte_3_digitos(x)
    Si x == 0, retornar 0
    Calcular orden de magnitud: n = floor(log10(abs(x)))
    Factor = 10^(n - 2)
    Retornar trunc(x / Factor) * Factor

Definir función suma_directa(n, exponente)
    Inicializar suma = 0
    Para i desde 1 hasta n
        término = corte_3_digitos(1 / i^exponente)
        suma = corte_3_digitos(suma + término)
    Retornar suma

Definir función suma_inversa(n, exponente)
    Inicializar suma = 0
    Para i desde n hasta 1 (en reversa)
        término = corte_3_digitos(1 / i^exponente)
        suma = corte_3_digitos(suma + término)
    Retornar suma
```

Código

En [44]:
```
importar  matemáticas

# Función que corta a tres cifras significativas
def  corte_3_digitos ( x ):
    if  x  ==  0 :
        return  0.0
    n  =  math . piso ( math . log10 ( abs ( x )))
    factor  =  10  **  ( n  -  2 )
    return  math . trunc ( x  /  factor )  *  factor

# Suma directa: i = 1 hasta n
def  suma_directa ( n ,  exponente ):
    suma  =  0.0
    for  i  in  range ( 1 ,  n  +  1 ):
        termino  =  corte_3_digitos ( 1  /  ( i  **  exponente ))
        suma  =  corte_3_digitos ( suma  +  termino )
    return  suma

# Suma inversa: i = n hasta 1
def suma_inversa(n, exponente):
    suma = 0.0
    for i in range(n, 0, -1):
        termino = corte_3_digitos(1 / (i ** exponente))
        suma = corte_3_digitos(suma + termino)
    return suma
```

In [45]:
```
print("Parte a: Suma de 1 / i^2")
n = 10
print("Suma directa (i = 1 a 10):", suma_directa(n, 2))
print("Suma inversa (i = 10 a 1):", suma_inversa(n, 2))
```

```
Parte a: Suma de 1 / i^2
Suma directa (i = 1 a 10): 1.53
Suma inversa (i = 10 a 1): 1.54
```

In [46]:
```
print("\nParte b: Suma de 1 / i^3")
n = 10
print("Suma directa (i = 1 a 10):", suma_directa(n, 3))
print("Suma inversa (i = 10 a 1):", suma_inversa(n, 3))
```

```
Parte b: Suma de 1 / i^3
Suma directa (i = 1 a 10): 1.16
Suma inversa (i = 10 a 1): 1.19
```

Conclusión

Al comparar las sumas directas e inversas bajo aritmética de corte a tres cifras significativas, se observa que:

- **La suma inversa** (desde los términos más pequeños a los más grandes) es más precisa, porque se minimiza el error de cancelación.
- **La suma directa** tiende a perder precisión ya que los términos pequeños aportan menos cuando se suman a un número ya grande.

Este fenómeno se debe a cómo funciona la representación limitada de los números decimales en una máquina.

Ejercicio 2: Aproximación de π usando la serie de Maclaurin

La serie de Maclaurin para la función arcotangente converge para (-1 < x \leq 1) y está dada por:

$$\arctan(x) = \sum_{i=1}^{\infty} \frac{(-1)^{i+1} x^{2i-1}}{2i-1}$$

Sabemos que:

$$\arctan(1) = \frac{\pi}{4} \Rightarrow \pi \approx 4 \cdot \sum_{i=1}^{n} \frac{(-1)^{i+1}}{2i-1}$$

a)

Utilice el hecho de que (\tan\left(\frac{\pi}{4}\right) = 1) para determinar el número (n) de términos de la serie que se necesita sumar para garantizar que:

$$|4 \cdot P_n(1) - \pi| < 10^{-3}$$

b)

El lenguaje de programación C++ requiere que el valor de π se encuentre dentro de (10^{-10}).
¿Cuántos términos de la serie se necesitan sumar para obtener este grado de precisión?

Pseudocódigo para aproximar π usando la serie de arctan(1)

Definir función aproximar_pi_tolerancia(tolerancia, max_iter)

```
Inicializar suma = 0
Inicializar i = 1
Mientras |4 * suma - π| >= tolerancia y i ≤ max_iter:
    término = (-1)^(i+1) / (2i - 1)
    suma = suma + término
    i = i + 1
Si i > max_iter:
    Retornar -1, None
Si no:
    Retornar i - 1, 4 * suma
```

Código

In [47]:
```
import math

def aproximar_pi_tolerancia(tolerancia, max_iter=10**7):
    suma = 0.0
    i = 1
    while abs(4 * suma - math.pi) >= tolerancia and i <= max_iter:
        termino = ((-1) ** (i + 1)) / (2 * i - 1)
        suma += termino
        i += 1
    if i > max_iter:
        return -1, None  # Se excedió el límite
    return i - 1, 4 * suma

# Parte a: Error menor a 10^{-3}
n1, pi_aprox1 = aproximar_pi_tolerancia(1e-3)
if n1 == -1:
    print("Parte a: No se alcanzó la tolerancia de 10⁻³ con el número máximo de iteraciones.")
else:
    print(f"Parte a: Se requieren {n1} términos para obtener |4Pn(1) - π| < 10⁻³")
    print(f"Aproximación de π: {pi_aprox1:.15f}")

# Parte b: Error menor a 10^{-10}
n2, pi_aprox2 = aproximar_pi_tolerancia(1e-10)
if n2 == -1:
    print("\nParte b: No se alcanzó la tolerancia de 10⁻¹⁰ con el número máximo de iteraciones.")
else:
    print(f"\nParte b: Se requieren {n2} términos para obtener |4Pn(1) - π| < 10⁻¹⁰")
    print(f"Aproximación de π: {pi_aprox2:.15f}")
```

```
Parte a: Se requieren 1000 términos para obtener |4Pn(1) - π| < 10⁻³
Aproximación de π: 3.140592653839794

Parte b: No se alcanzó la tolerancia de 10⁻¹⁰ con el número máximo de iteraciones.
```

Ejercicio 3

Otra fórmula para calcular (\pi) se puede deducir a partir de la identidad:

$$\frac{\pi}{4} = 4 \cdot \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

Determine el número de términos que se deben sumar para garantizar una aproximación de (\pi) dentro de (10^{-3}).

Pseudocódigo

Definir función machin_arctan(x, n)

```
Inicializar suma = 0
Para i desde 1 hasta n:
    término = (-1)^{i+1} * x^{2i - 1} / (2i - 1)
    suma = suma + término
Retornar suma
```

Definir función aproximar_pi_machin(tolerancia)

```
Inicializar n = 1
Mientras error >= tolerancia:
    pi_aprox = 4 * (4 * machin_arctan(1/5, n) - machin_arctan(1/239, n))
    error = |pi_aprox - π|
    n = n + 1
Retornar n - 1, pi_aprox
```

Código

In [48]:
```
import math

def machin_arctan(x, n):
    suma = 0.0
    for i in range(1, n + 1):
        termino = ((-1) ** (i + 1)) * (x ** (2 * i - 1)) / (2 * i - 1)
        suma += termino
    return suma

def aproximar_pi_machin(tolerancia):
    n = 1
    while True:
        term1 = machin_arctan(1/5, n)
        term2 = machin_arctan(1/239, n)
        pi_aprox = 4 * (4 * term1 - term2)
        error = abs(pi_aprox - math.pi)
        if error < tolerance:
            return n, pi_aprox
        n += 1
```

Ejercicio 4: Comparación de algoritmos para producto

Compare los siguientes tres algoritmos. ¿Cuándo es correcto el algoritmo de la parte **1a**?

---

a.

ENTRADA: (n, x_1, x_2, \ldots, x_n)
SALIDA: PRODUCT

- Paso 1: Determine PRODUCT = 0
- Paso 2: Para (i = 1, 2, \ldots, n) haga

Determine PRODUCT = PRODUCT * (x_i)

- Paso 3: SALIDA PRODUCT;

PARE

---

b.

ENTRADA: (n, x_1, x_2, \ldots, x_n)
SALIDA: PRODUCT

- Paso 1: Determine PRODUCT = 1
- Paso 2: Para (i = 1, 2, \ldots, n) haga

Set PRODUCT = PRODUCT * (x_i)

- Paso 3: SALIDA PRODUCT;

PARE

---

c.

ENTRADA: (n, x_1, x_2, \ldots, x_n)
SALIDA: PRODUCT

- Paso 1: Determine PRODUCT = 1
- Paso 2: Para (i = 1, 2, \ldots, n) haga

Si (x_i = 0) entonces determine PRODUCT = 0;
SALIDA PRODUCT;
PARE
Sino: Determine PRODUCT = PRODUCT * (x_i)

- Paso 3: SALIDA PRODUCT;

PARE

a

In [49]:
```
def algoritmo_a(valores):
    print("Algoritmo A:")
    product = 0
    print(f"Paso 1 → PRODUCT = {product}")

    for i, x in enumerate(valores, start=1):
        product *= x
        print(f"Paso 2 (i={i}) → PRODUCT = PRODUCT * {x} = {product}")

    print(f"Paso 3 → SALIDA PRODUCT = {product}")
    return product
```

b

In [50]:
```
def algoritmo_b(valores):
    print("Algoritmo B:")
    product = 1
    print(f"Paso 1 → PRODUCT = {product}")

    for i, x in enumerate(valores, start=1):
        product *= x
        print(f"Paso 2 (i={i}) → PRODUCT = PRODUCT * {x} = {product}")

    print(f"Paso 3 → SALIDA PRODUCT = {product}")
    return product
```

c

In [51]:
```
def algoritmo_c(valores):
    print("Algoritmo C:")
    product = 1
    print(f"Paso 1 → PRODUCT = {product}")

    for i, x in enumerate(valores, start=1):
        if x == 0:
            product = 0
            print(f"Paso 2 (i={i}) → x = 0 ⇒ PRODUCT = 0 → SALIDA ANTICIPADA")
            return product
        else:
            product *= x
            print(f"Paso 2 (i={i}) → PRODUCT = PRODUCT * {x} = {product}")

    print(f"Paso 3 → SALIDA PRODUCT = {product}")
    return product
```

Código de prubas

In [52]:
```
# Lista de pruebas
pruebas = {
    "sin ceros": [2, 3, 4],
    "con un cero": [2, 0, 4],
    "todos ceros": [0, 0, 0],
    "uno solo": [5],
}

# Ejecutar para cada ejemplo
for nombre, lista in pruebas.items():
    print(f"\n====== Ejemplo: {nombre.upper()} → {lista} ======")
    print("\n--- Algoritmo A ---")
    algoritmo_a(lista)
    print("\n--- Algoritmo B ---")
    algoritmo_b(lista)
    print("\n--- Algoritmo C ---")
    algoritmo_c(lista)
```

```
====== Ejemplo: SIN CEROS → [2, 3, 4] ======

--- Algoritmo A ---
Algoritmo A:
Paso 1 → PRODUCT = 0
Paso 2 (i=1) → PRODUCT = PRODUCT * 2 = 0
Paso 2 (i=2) → PRODUCT = PRODUCT * 3 = 0
Paso 2 (i=3) → PRODUCT = PRODUCT * 4 = 0
Paso 3 → SALIDA PRODUCT = 0

--- Algoritmo B ---
Algoritmo B:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 2 = 2
Paso 2 (i=2) → PRODUCT = PRODUCT * 3 = 6
Paso 2 (i=3) → PRODUCT = PRODUCT * 4 = 24
Paso 3 → SALIDA PRODUCT = 24

--- Algoritmo C ---
Algoritmo C:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 2 = 2
Paso 2 (i=2) → PRODUCT = PRODUCT * 3 = 6
Paso 2 (i=3) → PRODUCT = PRODUCT * 4 = 24
Paso 3 → SALIDA PRODUCT = 24

====== Ejemplo: CON UN CERO → [2, 0, 4] ======

--- Algoritmo A ---
Algoritmo A:
Paso 1 → PRODUCT = 0
Paso 2 (i=1) → PRODUCT = PRODUCT * 2 = 0
Paso 2 (i=2) → PRODUCT = PRODUCT * 0 = 0
Paso 2 (i=3) → PRODUCT = PRODUCT * 4 = 0
Paso 3 → SALIDA PRODUCT = 0

--- Algoritmo B ---
Algoritmo B:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 2 = 2
Paso 2 (i=2) → PRODUCT = PRODUCT * 0 = 0
Paso 2 (i=3) → PRODUCT = PRODUCT * 4 = 0
Paso 3 → SALIDA PRODUCT = 0

--- Algoritmo C ---
Algoritmo C:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 2 = 2
Paso 2 (i=2) → x = 0 ⇒ PRODUCT = 0 → SALIDA ANTICIPADA

====== Ejemplo: TODOS CEROS → [0, 0, 0] ======

--- Algoritmo A ---
Algoritmo A:
Paso 1 → PRODUCT = 0
Paso 2 (i=1) → PRODUCT = PRODUCT * 0 = 0
Paso 2 (i=2) → PRODUCT = PRODUCT * 0 = 0
Paso 2 (i=3) → PRODUCT = PRODUCT * 0 = 0
Paso 3 → SALIDA PRODUCT = 0

--- Algoritmo B ---
Algoritmo B:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 0 = 0
Paso 2 (i=2) → PRODUCT = PRODUCT * 0 = 0
Paso 2 (i=3) → PRODUCT = PRODUCT * 0 = 0
Paso 3 → SALIDA PRODUCT = 0

--- Algoritmo C ---
Algoritmo C:
```

```
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → x = 0 ⇒ PRODUCT = 0 → SALIDA ANTICIPADA

====== Ejemplo: UNO SOLO → [5] ======

--- Algoritmo A ---
Algoritmo A:
Paso 1 → PRODUCT = 0
Paso 2 (i=1) → PRODUCT = PRODUCT * 5 = 0
Paso 3 → SALIDA PRODUCT = 0

--- Algoritmo B ---
Algoritmo B:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 5 = 5
Paso 3 → SALIDA PRODUCT = 5

--- Algoritmo C ---
Algoritmo C:
Paso 1 → PRODUCT = 1
Paso 2 (i=1) → PRODUCT = PRODUCT * 5 = 5
Paso 3 → SALIDA PRODUCT = 5
```

Conclusión

- **Algoritmo a** es incorrecto en general. Inicia con `PRODUCT = 0`, por lo que siempre produce 0 sin importar los valores.
  Solo es correcto si algún (x_i = 0), ya que el producto real también sería 0.

- **Algoritmo b** es correcto en todos los casos. Inicia con `PRODUCT = 1` y recorre todos los elementos multiplicando.
  Sin embargo, no es eficiente cuando hay un 0, porque sigue multiplicando innecesariamente.

- **Algoritmo c** es el más eficiente y correcto. También comienza con `PRODUCT = 1`, pero termina anticipadamente si detecta un 0.

Ahorra tiempo y operaciones.

Por lo tanto:

- El algoritmo de la parte **1a solo es correcto cuando existe un (x_i = 0)**.
- Para una solución general, se deberia usar el algoritmo **b o c**, siendo **c el más óptimo**.

Ejercicio 5

a. ¿Cuántas multiplicaciones y sumas se requieren para determinar una suma de la forma:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} a_i b_j \ ?$$

b. Modifique la suma en la parte **a)** a un formato equivalente que **reduzca el número de cálculos**.

Resolución:

a) ¿Cuántas operaciones?

La expresión:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} a_i b_j$$

requiere:

- $$ \frac{n(n+1)}{2}

$$multiplicaciones-$$

\frac{n(n+1)}{2} - 1 $$ sumas

b) Reescritura equivalente y más eficiente:

Usando la propiedad de conmutatividad de la suma:

$$\sum_{i=1}^{n}\sum_{j=1}^{i} a_i b_j = \sum_{j=1}^{n} b_j \sum_{i=j}^{n} a_i$$

Esta forma reduce el número de cálculos porque:

- Se puede calcular previamente los valores

$$\sum_{i=j}^{n} a_i$$

  de forma acumulativa hacia atrás.
- Disminuye la cantidad de multiplicaciones requeridas, haciendo el cálculo más eficiente.

Pseudocódigo

- **a) Pseudocódigo forma original:**

Entradas: vectores a[1..n], b[1..n]

Inicializar SUMA = 0

Para i desde 1 hasta n hacer:

```
Para j desde 1 hasta i hacer:
    SUMA = SUMA + a[i] * b[j]
```

Retornar SUMA

- **b) Pseudocódigo forma optimizada:**

Entradas: vectores a[1..n], b[1..n] Inicializar suma_acumulada[n+1] = 0

Para i desde n hasta 1 hacer: suma_acumulada[i] = suma_acumulada[i+1] + a[i]

Inicializar SUMA = 0

Para j desde 1 hasta n hacer: SUMA = SUMA + b[j] * suma_acumulada[j]

Retornar SUMA

Código Python

In [53]:
```
def forma_original(a, b):
    n = len(a)
    suma = 0
    for i in range(n):
        for j in range(i + 1):
            suma += a[i] * b[j]
    return suma

def forma_optimizada(a, b):
    n = len(a)
    suma_acumulada = [0] * (n + 1)

    for i in range(n - 1, -1, -1):
        suma_acumulada[i] = suma_acumulada[i + 1] + a[i]

    suma = 0
    for j in range(n):
        suma += b[j] * suma_acumulada[j]
    return suma
```

In [54]:
```
# Prueba con vectores pequeños
a = [1, 2, 3, 4]
b = [5, 6, 7, 8]

res1 = forma_original(a, b)
res2 = forma_optimizada(a, b)

print("Resultado (forma original):", res1)
print("Resultado (forma optimizada):", res2)
```

```
Resultado (forma original): 185
Resultado (forma optimizada): 185
```

Conclusión

- La forma original requiere:

$$\frac{n(n+1)}{2}$$

multiplicaciones y sumas, lo que implica un crecimiento cuadrático.

- La versión optimizada reorganiza la suma, permitiendo acumular primero las sumas parciales de:

$$a_i$$

y luego multiplicar por cada:

$$b_j$$

- Esto **reduce el número de multiplicaciones y mejora el rendimiento computacional**, especialmente cuando (n) es grande.

DISCUSIONES

Ejercicio 1

Escriba un algoritmo para sumar la serie finita:

$$\sum_{i=1}^{n} x_i$$

en orden inverso.

Pseudocódigo

ENTRADA: vector x[1..n]

SALIDA: suma

Inicializar suma = 0

Para i desde n hasta 1 hacer:

```
suma = suma + x[i]
```

Retornar suma

Código

In [55]:
```python
def suma_inversa(x):
    suma = 0
    for i in reversed(range(len(x))):
        suma += x[i]
    return suma

# Ejemplo de uso
x = [1, 2, 3, 4, 5]
print("Suma en orden inverso:", suma_inversa(x))
```

```
Suma en orden inverso: 15
```

Ejercicio 2

Las ecuaciones (1.2) y (1.3) proporcionan formas alternativas para calcular las raíces (x_1) y (x_2) de:

$$ax^2 + bx + c = 0$$

Construya un algoritmo con entrada (a, b, c) y salida (x_1, x_2) que seleccione automáticamente la fórmula más precisa para cada raíz.

Pseudocódigo

ENTRADA: a, b, c

SALIDA: x1, x2

Calcular discriminante: d = b^2 - 4ac

Si d < 0:

```
Usar fórmula para raíces complejas
```

Sino:

```
Si b ≥ 0:
    x1 = (-b - sqrt(d)) / (2a)
Sino:
    x1 = (-b + sqrt(d)) / (2a)

x2 = c / (a * x1)  # para evitar cancelación
```

Retornar x1, x2

Código

In [56]:
```python
import math
import cmath

def raices_estables(a, b, c):
    d = b**2 - 4*a*c
    if d < 0:
        x1 = (-b + cmath.sqrt(d)) / (2*a)
        x2 = (-b - cmath.sqrt(d)) / (2*a)
    else:
        if b >= 0:
            x1 = (-b - math.sqrt(d)) / (2*a)
        else:
            x1 = (-b + math.sqrt(d)) / (2*a)
        x2 = c / (a * x1)
    return x1, x2

# Ejemplo:
a, b, c = 1, -1e8, 1
x1, x2 = raices_estables(a, b, c)
print("Raíces estables:", x1, x2)
```

```
Raíces estables: 100000000.0 1e-08
```

Ejercicio 3

Suponga que:

$$\frac{1-2x}{1-x+x^2}+\frac{2x-4x^3}{1-x^2+x^4}+\frac{4x^3-8x^7}{1-x^4+x^8}+\cdots=\frac{1+2x}{1+x+x^2}$$

para (x < 1) y (x = 0.25).

Escriba y ejecute un algoritmo que determine el número de términos necesarios en el lado izquierdo de la ecuación para que **la diferencia absoluta con el lado derecho sea menor que (10^{-6})**.

Pseudocódigo

ENTRADA: x = 0.25

SALIDA: n, suma, error

Inicializar suma = 0, n = 0

Calcular lado_derecho = (1 + 2x) / (1 + x + x^2)

Mientras |suma - lado_derecho| ≥ 10^{-6}:

```
    num = 2^n * x^{2^n - 1}
    den = 1 - x^{2^n} + x^{2^{n+1}}
    suma += num / den
    n += 1
```

Retornar n, suma

Código

In [57]:

```python
from decimal import Decimal, getcontext

def serie_aproximada(x=0.25, tolerancia=1e-6, max_iter=1000):
    # Establecer precisión decimal
    getcontext().prec = 50

    x = Decimal(x)
    tolerancia = Decimal(tolerancia)
    uno = Decimal(1)
    dos = Decimal(2)

    lado_derecho = (uno + dos * x) / (uno + x + x**2)
    suma = Decimal(0)
    n = 0

    while abs(suma - lado_derecho) >= tolerancia:
        if n >= max_iter:
            print(" Límite máximo de iteraciones alcanzado.")
            break

        exp1 = 2**n - 1
        exp2 = 2**n
        exp3 = 2**(n + 1)

        numerador = Decimal(2**n) * (x**exp1)
        denominador = uno - (x**exp2) + (x**exp3)

        suma += numerador / denominador
        n += 1

    return n, float(suma), float(abs(suma - lado_derecho))

# Ejemplo:
n_ter ,  suma_aprox ,  error  =  serie_aproximada ( x = 0.25 ,  tolerancia = 1e-6 )
print ( f "Términos requeridos: { n_ter } " )
print ( f "Suma aproximada: { suma_aprox } " )
print ( f "Error absoluto: { error } " )
```

```
Límite máximo de iteraciones alcanzadas.
Términos requeridos: 1000
Suma aproximada: 1.825122003206338
Error absoluto: 0.6822648603491952
```