TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

FUNDAMENTOS DE PROGRAMACIÓN

EJEMPLOS CLÁSICOS DE RECURSIÓN

EJEMPLOS CLÁSICOS DE RECURSIÓN

La recursión es una técnica de programación fundamental que permite abordar problemas dividiéndolos en versiones más pequeñas de sí mismos. Su elegancia radica en la simplicidad del enfoque: una función que se llama a sí misma hasta alcanzar una condición base. Aunque puede parecer compleja al inicio, es clave en múltiples algoritmos clásicos como el cálculo del factorial, la serie de Fibonacci, la potencia de un número o la búsqueda binaria. Además, su aplicación se extiende al recorrido de estructuras jerárquicas como árboles y algoritmos eficientes como QuickSort. Este enfoque se contrapone a la iteración, y aunque ambos pueden resolver los mismos problemas, cada uno tiene sus ventajas dependiendo del contexto. A lo largo del texto, se exploran estos ejemplos y se contrastan ambos métodos para comprender mejor su utilidad en el desarrollo de soluciones computacionales.

Factorial de un Número

El cálculo del factorial de un número es uno de los ejemplos más comunes de recursión. Se define matemáticamente como:

$$n!=n\times(n-1)!n! = n \times (n-1)!n!=n×(n−1)!$$

con la condición base:

$$0!=10! = 10!=1$$

Figura 1. Función factorial en Python

```python
def factorial(n):
    if n == 0 or n == 1:  # Caso base
        return 1
    return n * factorial(n - 1)  # Llamada recursiva

print(factorial(5))  # Salida: 120
```

Explicación:

- Si n es 0 o 1, se devuelve 1.
- De lo contrario, la función se llama a sí misma con n-1 y multiplica el resultado por n.

Serie de Fibonacci

La serie de Fibonacci es otra aplicación clásica de la recursión. Se define como:

$$F(n)=F(n-1)+F(n-2)F(n) = F(n-1) + F(n-2)F(n)=F(n−1)+F(n−2)$$

con los valores iniciales:

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

Figura 2. Función Fibonacci en Python

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)  # Llamadas recursivas anidadas

print(fibonacci(6))  # Salida: 8
```

Potencia de un Número

El cálculo de una potencia puede implementarse recursivamente mediante la relación:

$$ab=a×a(b−1)a^b = a \times a^{(b-1)}ab=a×a(b−1)$$

con el caso base:

$$a0=1a^0 = 1a0=1$$

Figura 3. Potencia de numero en Python

```python
def potencia(a, b):
    if b == 0:  # Caso base
        return 1
    return a * potencia(a, b - 1)  # Llamada recursiva

print(potencia(2, 3))  # Salida: 8
```

Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada, con una complejidad de O(log n).

División y Conquista:

- Se divide el arreglo en dos mitades.
- Se compara el elemento buscado con el valor medio.
- Se realiza una búsqueda recursiva en la mitad correspondiente.

Figura 4. Búsqueda binaria

```python
def busqueda_binaria(lista, inicio, fin, objetivo):
    if inicio > fin:  # Caso base: el elemento no está en la lista
        return -1

    medio = (inicio + fin) // 2

    if lista[medio] == objetivo:
        return medio
    elif lista[medio] > objetivo:
        return busqueda_binaria(lista, inicio, medio - 1, objetivo)  # Buscar en la
```

```
    else:
        return busqueda_binaria(lista, medio + 1, fin, objetivo)  # Buscar en la der

lista_ordenada = [1, 3, 5, 7, 9, 11]
print(busqueda_binaria(lista_ordenada, 0, len(lista_ordenada) - 1, 5))  # Salida: 2
```

La recursión es una técnica esencial en programación que permite resolver problemas de manera elegante y estructurada al dividirlos en instancias más pequeñas de sí mismos (Juganaru Mathieu, 2015). Su aplicabilidad abarca diversos campos, desde la manipulación de estructuras de datos hasta la optimización de algoritmos en inteligencia artificial. A continuación, se presentan algunas de las aplicaciones prácticas más relevantes de la recursión en la informática.

Recursión en Estructuras de Datos

Recorrido de Árboles

Los árboles son estructuras jerárquicas en las que cada nodo puede tener múltiples nodos hijos. La recursión resulta ser la estrategia más natural para recorrerlos debido a su estructura inherentemente anidada (Ceballos Sierra, 2015).

Ejemplo: Recorrido en preorden (raíz -> izquierda -> derecha)

Figura 5. Recorrido de Árbol

```
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None

def recorrer_preorden(nodo):
    if nodo is None:
        return
    print(nodo.valor)
    recorrer_preorden(nodo.izquierda)
    recorrer_preorden(nodo.derecha)

# Construcción del árbol
raiz = Nodo(1)
raiz.izquierda = Nodo(2)
raiz.derecha = Nodo(3)
raiz.izquierda.izquierda = Nodo(4)
raiz.izquierda.derecha = Nodo(5)

recorrer_preorden(raiz)
```

Recursión en Algoritmos de Ordenamiento

QuickSort (Ordenamiento Rápido)

El algoritmo QuickSort utiliza recursión para dividir una lista en sublistas más pequeñas y ordenarlas de manera eficiente.

Figura 6. Implementación de QuickSort en Python

```python
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[len(lista) // 2]
    izquierda = [x for x in lista if x < pivote]
    centro = [x for x in lista if x == pivote]
    derecha = [x for x in lista if x > pivote]
    return quicksort(izquierda) + centro + quicksort(derecha)

print(quicksort([3, 6, 8, 10, 1, 2, 1]))  # Salida: [1, 1, 2, 3, 6, 8, 10]
```

Recursión vs. Iteración

En el ámbito de la programación, la recursión y la iteración son dos enfoques fundamentales para resolver problemas que requieren la repetición de instrucciones (Ceballos Sierra, 2015). Ambos métodos tienen sus ventajas y desventajas, dependiendo del contexto en el que se apliquen. A continuación, se presenta un análisis detallado de sus diferencias, fortalezas y debilidades, junto con ejemplos que ilustran su uso en la práctica.

Definición y Concepto

Recursión

La recursión es una técnica en la que una función se llama a sí misma repetidamente hasta alcanzar una condición base (Juganaru Mathieu, 2015). Se utiliza comúnmente en problemas que pueden descomponerse en subproblemas más pequeños de la misma naturaleza.

Figura 7. Ejemplo de recursión: Cálculo del factorial de un número

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Salida: 120
```

Iteración

La iteración se basa en la repetición de un bloque de código mediante estructuras de control como bucles (for, while). En este enfoque, las instrucciones se ejecutan hasta que se cumple una condición de finalización (Juganaru Mathieu, 2015).

Figura 8. Ejemplo de iteración: Cálculo del factorial de un número

```python
def factorial_iterativo(n):
    resultado = 1
    for i in range(2, n + 1):
        resultado *= i
    return resultado

print(factorial_iterativo(5))  # Salida: 120
```

Tabla 1. Comparación entre recursión e iteración en programación

Aspecto	Recursión	Iteración
Estructura	Función que se llama a sí misma	Uso de bucles (for, while)
eficiencia	Puede ser menos eficiente debido a la sobrecarga de la pila de llamadas	Generalmente más eficiente al evitar llamadas adicionales
uso de memoria	Consume más memoria debido a las llamadas anidadas	Consume menos memoria al mantener solo variables en ejecución
Complejidad	Puede ser más complejo de entender y depurar	Más fácil de leer y seguir en la mayoría de los casos
Aplicaciones	Útil en problemas recursivos naturales como árboles, grafos y algoritmos de divide y vencerás	Preferible en problemas iterativos que requieren ejecución secuencial

Algunos problemas pueden resolverse tanto con recursividad como con bucles (iteración).

Figura 9. Ejemplo: Fibonacci con Recursión e Iteración

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))  # Salida: 8
```

Figura 10. Versión Iterativa en Python

```
def fibonacci_iterativo(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

print(fibonacci_iterativo(6))  # Salida: 8
```

La versión recursiva es más elegante, pero menos eficiente porque recalcula valores repetidos. La versión iterativa es más rápida y evita el consumo excesivo de memoria.

Bibliografía

- Ceballos Sierra, F. J. (2015). C/C++ curso de programación (3ª ed.). RA-MA Editorial. https://elibro.net/es/lc/tecnologicadeloriente/titulos/62460
- Juganaru Mathieu, M. (2015). Introducción a la programación. Grupo Editorial Patria. https://elibro.net/es/lc/tecnologicadeloriente/titulos/39449