2. Recursividad

La **recursividad** es una técnica de programación poderosa donde una función o procedimiento se llama a sí mismo para resolver un problema. Se basa en la idea de resolver un problema dividiéndolo en versiones más pequeñas de sí mismo hasta llegar a un caso base que se puede resolver directamente.

0.1. ¿Qué es la Recursividad?

En esencia, una función recursiva es aquella que se define en términos de sí misma. Para que la recursividad sea viable y no se ejecute indefinidamente, debe tener dos componentes clave:

1. **Caso Base:** Es la condición de terminación. Es el problema más simple de la recursión que se puede resolver directamente sin más llamadas recursivas. Sin un caso base, la función se llamaría a sí misma infinitamente, lo que provocaría un *desbordamiento de la pila* (stack overflow).

2. **Paso Recursivo (o Caso Recursivo):** Es la parte donde la función se llama a sí misma, pero con una entrada más pequeña o un problema simplificado que se acerca al caso base. Cada llamada recursiva debe moverse hacia el caso base.

—

0.2. ¿Cómo Funciona la Recursividad?

Cuando una función recursiva es llamada, se coloca en la **pila de llamadas (call stack)**. Cada vez que la función se llama a sí misma, una nueva instancia de la función se coloca en la parte superior de la pila. Este proceso continúa hasta que se alcanza el caso base. Una vez que el caso base se resuelve, las llamadas se "desapilan" de una en una, y los resultados de las llamadas anidadas se utilizan para resolver las llamadas anteriores, hasta que la llamada original se completa.

—

0.3. Ejemplos Clásicos de Recursividad en C++

Veamos algunos ejemplos comunes para entender mejor la recursividad.

Suma de los Primeros N Números (Suma Recursiva)

Este ejemplo calcula la suma de todos los números enteros desde 1 hasta n de forma recursiva.

Listing 1: Suma de los primeros N números de forma recursiva

```cpp
#include <iostream>

using namespace std; // Incluyendo namespace std como se solicit

// Funci n recursiva para calcular la suma de los primeros n n meros
int sumaRecursiva(int n) {
    // Caso Base: Si n es 0, la suma es 0.
    if (n == 0) {
        return 0;
    }
    // Paso Recursivo: La suma de n es n m s la suma de los (n-1) n meros anteriores.
    else {
        return n + sumaRecursiva(n - 1);
    }
}

int main() {
    int numero = 5;
    cout << "La suma de los primeros " << numero << " n meros es: " << sumaRecursiva(numero) << endl;
        // Salida: 15

    numero = 10;
    cout << "La suma de los primeros " << numero << " n meros es: " << sumaRecursiva(numero) << endl;
        // Salida: 55
    return 0;
}
```

Cálculo del Factorial

El factorial de un número entero no negativo n, denotado como $n!$, es el producto de todos los enteros positivos menores o iguales a n. El factorial de 0 es 1.

Listing 2: Cálculo del factorial de un número

```cpp
#include <iostream>

using namespace std; // Incluyendo namespace std como se solicit

// Funci n recursiva para calcular el factorial de un n mero
long long factorial(int n) {
    // Caso Base: El factorial de 0 es 1.
    if (n == 0) {
        return 1;
    }
    // Paso Recursivo: n! = n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int numero = 5;
    cout << "El factorial de " << numero << " es: " << factorial(numero) << endl; // Salida: 120

    numero = 0;
    cout << "El factorial de " << numero << " es: " << factorial(numero) << endl; // Salida: 1
    return 0;
}
```

Serie de Fibonacci

La serie de Fibonacci es una secuencia de números donde cada número es la suma de los dos anteriores, comenzando con 0 y 1. $(0, 1, 1, 2, 3, 5, 8, \dots)$

Listing 3: Cálculo del n-ésimo término de Fibonacci

```cpp
#include <iostream>

using namespace std; // Incluyendo namespace std como se solicit

// Funci n recursiva para calcular el n- simo  t rmino de Fibonacci
int fibonacci(int n) {
    // Casos Base: Los primeros dos t rminos de la serie.
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }
    // Paso Recursivo: El n- simo  t rmino es la suma de los dos anteriores.
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int n_fib = 6;
    cout << "El " << n_fib << "- simo  t rmino de Fibonacci es: " << fibonacci(n_fib) << endl; // Salida: 8

    n_fib = 0;
    cout << "El " << n_fib << "- simo  t rmino de Fibonacci es: " << fibonacci(n_fib) << endl; // Salida: 0
    return 0;
}
```

—

0.4. Análisis de Eficiencia de la Recursividad

El análisis de la complejidad de los algoritmos recursivos a menudo implica el uso de **ecuaciones de recurrencia**.

- **Suma de los Primeros N Números y Factorial:**
  - **Complejidad Temporal:** $\mathcal{O}(n)$. Cada llamada recursiva reduce el problema en 1, y hay n llamadas.
  - **Complejidad Espacial:** $\mathcal{O}(n)$. Debido a la pila de llamadas; cada llamada agrega una nueva entrada a la pila hasta que se alcanza el caso base.
- **Serie de Fibonacci (implementación recursiva básica):**
  - **Complejidad Temporal:** $\mathcal{O}(2^n)$. Esta implementación es ineficiente porque calcula repetidamente los mismos valores. Por ejemplo, para fibonacci(5), se calculan fibonacci(3) y fibonacci(2) varias veces. Esto se puede optimizar con técnicas como la **memorización** o **Programación Dinámica**, lo que reduciría la complejidad a $\mathcal{O}(n)$.
  - **Complejidad Espacial:** $\mathcal{O}(n)$. También debido a la profundidad máxima de la pila de llamadas.

—

0.5. Ventajas y Desventajas de la Recursividad

Ventajas:

- **Claridad y Simplicidad:** A menudo, las soluciones recursivas son más legibles y concisas para problemas que tienen una naturaleza recursiva inherente (como los recorridos de árboles o grafos, o problemas definidos por recurrencias matemáticas).
- **Diseño Elegante:** Puede llevar a diseños de código muy elegantes y naturales.

Desventajas:

- **Sobrecarga del Stack (Pila de Llamadas):** Cada llamada recursiva consume memoria en la pila. Si la recursión es muy profunda, puede ocurrir un `stack overflow`.
- **Eficiencia:** A veces, las soluciones recursivas pueden ser menos eficientes que sus equivalentes iterativas debido a la sobrecarga de las llamadas a funciones.

- **Duplicidad de Cálculos:** Como se vio en el ejemplo de Fibonacci, si no se maneja correctamente (ej., con memorización), una función recursiva puede realizar los mismos cálculos varias veces, lo que lleva a una complejidad temporal exponencial.

La recursividad es una herramienta fundamental para resolver ciertos tipos de problemas, especialmente aquellos que pueden descomponerse naturalmente en subproblemas idénticos. Es esencial comprender sus principios y cuándo es apropiado (y eficiente) usarla.