RECURSIÓN

La recursividad tiene un concepto un tanto abstracto y complejo que implica la lógica y la matemática con otras ciencias como por ejemplo la programación.

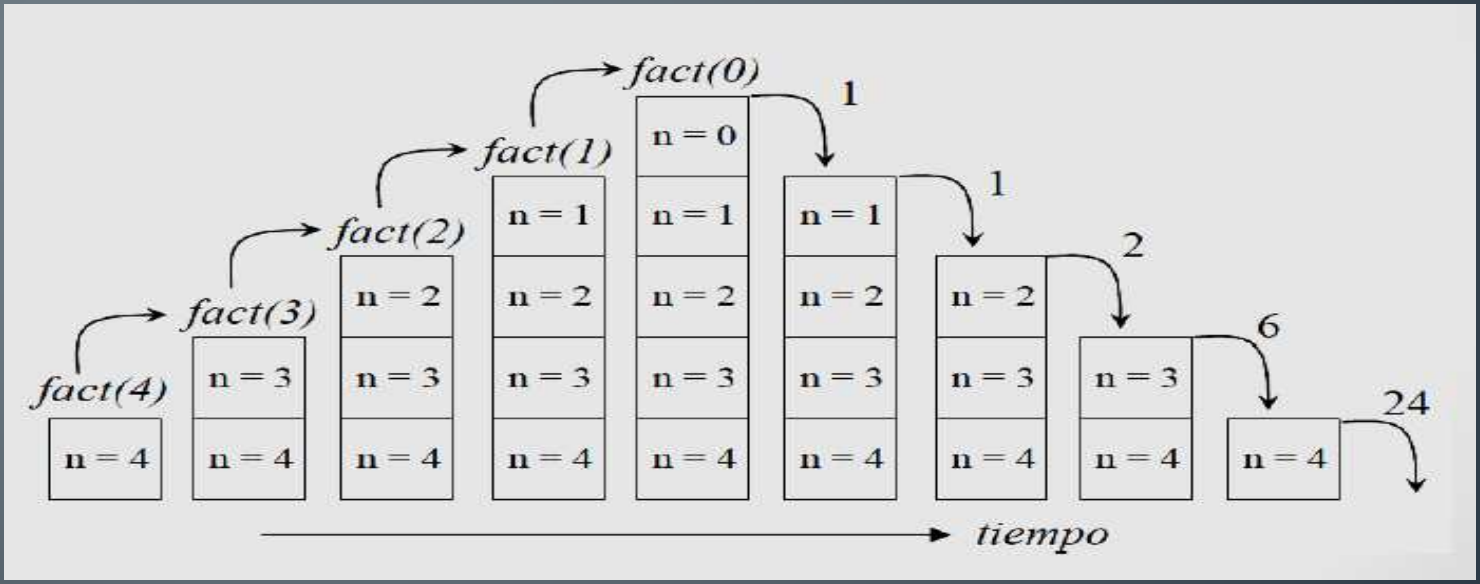
7. Definición e importancia en el desarrollo de problemas

La recursión es una técnica de programación que consiste básicamente en el uso de funciones que se llaman a sí mismas, repetidamente, con datos cada vez más simples, hasta que cumplan con una condición base que las haga terminar su proceso, donde cada acción se determina a través de un resultado anterior llegando así a la solución de los datos iniciales.

El ejemplo más común para explicar la recursión es el cálculo del factorial de un número entero:

```
 K Función recursiva para cálculo de factoriales      »
int factoriaL(int n) {
   if(n < 0) return 0;
   else if(n > 1) return n*factoriaL(n-1);          K Recursividad          »
   return 1; K Condición de terminación, n  ?1 ?n ?0          »
}
```

Esta función recursiva, verifica si el número que se le pasa (n) es negativo, en dado caso devuelve un 0; si el número es mayor que 1 multiplica ese número por el resultado de llamar a la función restando uno a ese número (n-1). Cuando el número que se le pasa es un 1 o un 0, la función devuelve un 1. Esta última condición conocida como "condición de salida" o “caso base” es fundamental para impedir que la función se esté llamando a sí misma eternamente (bucle infinito). Una ilustración de cómo funciona la recursión en el caso del factorial de 4 es la siguiente:



La recursión sirve como reemplazo para el uso de los bucles en el programa, sin embargo esta no siempre es la mejor opción, por ejemplo, el caso del factorial era un buen ejemplo para ver cómo funciona una función recursiva, no obstante, la recursión no es la forma más eficiente de resolver esta función, ya que con un bucle for la función sería más simple y rápida de resolver.

Su importancia en el desarrollo de programas radica en que, permitirá simplificar el código, las soluciones a problemas complejos serán más sencillas, tendrán más claridad y elegancia, con ella no será necesaria la definición de una secuencia exacta de pasos para la resolución del problema, además de que proporciona una facilidad para determinar que la solución sea correcta.

Una solución recursiva, por lo general, es menos eficiente que una solución iterativa, debido a las operaciones auxiliares que implican las llamadas a las funciones (una simple llamada puede ocasionar un gran número de llamadas recursivas, la llamada a la función factorial(n) generarán llamadas recursivas). Sin embargo, la eficiencia de la recursión se basa en el hecho de que se puede emplear para resolver problemas de difícil solución iterativa, existen casos donde es conveniente implementar la recursión, pues algunos problemas son más simples de resolver a través de ella, por ejemplo cuando la estructura de datos es recursiva como en los árboles o en el algoritmo de ordenación “Quicksort”.

La recursión consume demasiados recursos de memoria y tiempo de ejecución, y se debe aplicar a funciones en donde realmente sea necesaria, es decir, en donde no exista una solución iterativa simple.

A pesar de todos estos inconvenientes, en varias situaciones implementar la recursión permite a los programadores definir soluciones lógicas y sencillas, que sin su uso serían

mucho más difíciles de resolver. Por lo que la recursión llega a ser una herramienta fundamental en la programación.

8. Procesos recursivos y declaraciones recursivas

- Procesos recursivos:

Un proceso recursivo es aquel que se invoca a sí mismo. Estos nos permiten repetir recursos sin procedimientos diferentes a los parámetros, es una alternativa a la iteración en la solución de problemas, especialmente si estos tienen naturaleza recursiva o repetitiva.

Estos se componen de:

- **Caso base**: Para emplear de forma correcta la recursividad debe existir un caso base, donde no se necesite volver a invocar el proceso, la resolución del problema debe ser inmediata, es decir, los procesos recursivos tienen que tener una condición de fin, debe haber una llamada a la función en donde esta se resuelve de manera directa (simplemente devuelve un resultado), evitando tener que llamar de nuevo a la función.

  Este caso base es indispensable para finalizar el proceso de llamadas repetitivas a la función, de tal manera que no se produzca un bucle infinito de llamadas, evitando a la vez un desbordamiento de pila (Stack Overflow).

  .

- **Caso recursivo**: Involucra una solución que se basa en volver a emplear la función original (se vuelve a llamar a la función), con parámetros más cercanos al caso base. Donde los pasos a seguir serían:
  a. El procedimiento se invoca a sí mismo
  b. Se encuentra una solución al problema, tratándolo desde el mismo enfoque pero de tamaño menor
  c. A medida que el tamaño del problema se reduce, se va llegando al caso base.

Un ejemplo de estos procesos, es la famosa serie de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, 21,…), la cual comienza con 0 y 1 y tiene la característica de que cada número subsiguiente es la suma de los dos números anteriores. Esta serie se puede definir de forma recursiva, donde:

```
fibonacci(0)=0
fibonacci(1)=1
```

```
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

De manera que su programa correspondiente es:

```
long fibonacci (int n){
     if (n ?0                ? n ?1) return n;      ?caso base
     else return fibonacci(n-1)+fibonacci(n-2);    ?caso recursivo
}
```

Cada vez que se llama a la función fibonacci se comprueba el caso base, es decir, se comprueba si n es 0 o 1. Si esto resulta verdadero se regresa n. De modo contrario, si n es mayor que 1, el paso de recursión genera dos llamadas recursivas donde cada una resuelve un problema un poco más "simple" que el original (valores de n menores).

Los procesos recursivos se basan en la técnica "divide y vencerás" para la resolución de algoritmos, donde:

a. Se divide a un problema en partes más pequeñas.
b. Estos sub-problemas son más sencillos de resolver que el problema original.
c. Las soluciones encontradas a los sub-problemas se combinan para encontrar la solución del problema original.

Se tiene que, las recursiones pueden ser directas o indirectas, es decir, la recursión es directa cuando la función se invoca a sí misma, y es indirecta cuando la función (f1) llama a otra función (f2) y esta función(f2) invoca a la función (f1). Estos procedimientos también pueden ser simples o múltiples, donde una recursión simple se da cuando en la función cada caso recursivo hace solo una llamada a la función, mientras que en una múltiple se hacen varias llamadas en cada caso.

Se deben tomar en cuenta algunas consideraciones para los procedimientos recursivos:

- **Condiciones de limitación**: Debe existir al menos una condición que pueda poner fin a la recursión; también se debe supervisar los casos en los que no se cumple ninguna condición dentro de un número razonable de llamadas. Si no existe al menos una condición que pueda cumplirse sin errores, el procedimiento corre el riesgo de ejecutarse en un bucle infinito.

- **Uso de la memoria**: Emplear esta técnica tiene cierta cantidad de espacio limitado para las variables locales. Cada vez que la función se llama a sí misma, utiliza más cantidad de ese espacio para las copias adicionales de sus variables locales.
- **Eficacia**: Casi siempre se puede sustituir un bucle por la recursión. Estos no lidian con la sobrecarga de transferir argumentos, inicializar el almacenamiento adicional y devolver valores. El rendimiento puede ser mucho mayor sin llamadas recursivas.
- **Recursividad mutua**: Si dos procedimientos se llaman mutuamente, el rendimiento puede ser muy deficiente o incluso puede producirse un bucle infinito. Aunque presenta los mismos inconvenientes que un procedimiento recursivo único, este puede ser mucho más difícil de detectar.
- **Llamadas con paréntesis**: En un procedimiento la función se debe llamar a sí misma de manera recursiva, para esto se debería agregar paréntesis detrás del nombre del procedimiento, aun cuando no exista una lista de argumentos. De lo contrario, se considerará que el nombre de la función representa al valor devuelto por ésta.
- **Pruebas**: Se debe probar minuciosamente para asegurarse de que los procedimientos siempre cumplen ciertas condiciones de limitación. Además de comprobar que la memoria no resulta insuficiente debido a la gran cantidad de llamadas recursivas.

- Declaraciones recursivas:

Las funciones recursivas se declaran de acuerdo con la siguiente sintaxis:

```
void recursion()
{
   recursion();                          K La función se llama así misma
               »
}

int main()
{
   recursion();
}
```

Se refiere a un método usado para definir elementos de un conjunto de términos de otros elementos del conjunto, la misma define los valores de las funciones para algunas entradas en

términos de los valores de la misma función para otras entradas, como por ejemplo una función factorial n!, que está definida por las reglas:

- $0! = 1$
- $(N + 1)! = (N + 1)N!$

Según la recursividad esta declaración es válida para todos los casos y la recursividad cumple el caso inicial de 0; Se puede decir que la declaración es el procedimiento que describe la construcción de la función factorial n!, que inicia en N igual a 0, seguido de N igual a 1, N igual a 2 y así sucesivamente.

Estas funciones y conjuntos declarados recursivamente pueden ser probados a menudo a través de un principio de inducción que sigue la declaración recursiva, como por ejemplo la declaración de los números naturales implica que si una propiedad posee el numero natural 0, y la propiedad tiene N+1 cuando posee N, entonces la propiedad posee a todos los números naturales.

La declaración recursiva está fundamentada por un caso base y una clausula inductiva, casos que deben cumplir la declaración sin ser definidos en términos de la declaración misma, y los otros casos que componen la definición deben ser inferiores de alguna manera.

Por lo tanto, una declaración es recursiva si usa las mismas asociaciones (relación establecida entre un identificador y una entidad en un programa) que produce. Estas declaraciones son usadas para declarar:

- Tipos de datos para estructuras de datos recursivas
- Subprogramas recursivos

Una declaración no compuesta de un tipo de dato recursivo solo puede desarrollarse en los registros o clases. Ocurre cuando un campo del registro es de tipo referencia al propio registro.

Un ejemplo básico es un nodo de una lista enlazada, que contiene una referencia al nodo posterior, el cual es un registro del mismo tipo.

9. Condición de parada

Un algoritmo recursivo consta de una condición de parada, que cuando es cierta permite resolver el problema directamente, y de la llamada recursiva, que resuelve el problema cuando la solución no es directa.

Es importante nombrar "**El caso base"** que es el que evita que nos metamos en un bucle infinito, es el caso que hace parar la recursión ya que no llama a la función. Recordando que el caso recursivo, es aquél en el que una función se llama a sí misma, y es el que hace que la recursión continúe hasta que se encuentre con el caso base.

10. Corrida en frío y depuración

La corrida en frío es la ejecución manual de cada sentencia del programa, utilizando un conjunto de datos de entrada determinados y verificando que los resultados son correctos. Como una técnica de depuración, el programador debe utilizar datos que permitan recorrer todas las posibles rutas del programa.

La depuración de programas es el proceso de identificar y corregir errores de programación. Es conocido también por el término inglés "debugging", cuyo significado es eliminación de bugs (bichos en inglés), manera en que se conoce informalmente a los errores de programación.

Si bien existen técnicas para la revisión sistemática del código fuente y se cuenta con medios computacionales para la detección de errores (depuradores) y facilidades integradas en los sistemas lower CASE y en los ambientes de desarrollo integrado, sigue siendo en buena medida una actividad manual, la cual requiere de paciencia, imaginación e intuición por parte del o los programadores. Muchas veces se requiere incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando los valores de variables y direcciones de memoria y ralentizando la salida de datos.

El proceso de depuración suele requerir los siguientes pasos:

- Identificación de errores

Los desarrolladores, los encargados de las pruebas y los usuarios finales informan de los errores que descubren mientras prueban o usan el software. Los desarrolladores localizan la línea exacta de códigos o el módulo de código que causa el error. Este proceso suele ser tardado y requiere mucho tiempo.

- Análisis de errores

Los codificadores analizan el error mediante el registro de todos los cambios de estado del programa y los valores de los datos. También dan prioridad a la corrección de errores en función de su impacto en la funcionalidad del software. El equipo de software también

identifica un calendario para la corrección de errores en función de los objetivos y requisitos de desarrollo.

- Corrección y validación

Los desarrolladores arreglan el error y hacen pruebas para asegurarse de que el software continúa en funcionamiento como se espera. Pueden escribir nuevas pruebas para comprobar si el error se repite en el futuro.

- Comparación entre la depuración y las pruebas

La depuración y las pruebas son procesos complementarios que garantizan que los programas de software funcionen como deben. Después de escribir una sección completa o parte de un código, los programadores realizan pruebas para identificar fallos y errores. Una vez encontrados los errores, los codificadores pueden comenzar el proceso de depuración y trabajar para librar al software de cualquier error.

11. El problema de sobredimensionamiento de la pila de ejecución (stack overflow)

El problema de sobredimensionamiento de la pila de ejecución, conocido como "Stack Overflow", ocurre cuando un programa llama a demasiadas funciones y el tamaño de la pila se agota. La pila es una estructura de datos utilizada por el sistema operativo para mantener el seguimiento de las funciones que están siendo ejecutadas y de las que están en espera.

El problema se produce cuando una función llama a otra función y esta última llama a otra función, y así sucesivamente, hasta que el tamaño máximo de la pila se agota. Esto puede ocurrir debido a un ciclo infinito de llamadas de funciones, una función que llama a sí misma de manera recursiva sin una condición de salida, o simplemente por un diseño incorrecto del programa.

El resultado es un error de Stack Overflow, que interrumpe la ejecución del programa y puede causar una falla en el sistema. Para solucionar este problema, es necesario identificar la causa subyacente y corregirla, ya sea a través de una mejor gestión de la memoria o una corrección en el diseño del programa.

Es importante tener en cuenta que los lenguajes de programación difieren en la cantidad de memoria que asignan a la pila, y algunos lenguajes también permiten aumentar el tamaño de la pila, lo que puede ayudar a prevenir el problema de Stack Overflow en algunos casos.

Además, otras causas comunes de Stack Overflow incluyen la falta de verificación de errores y la falta de validación de los datos de entrada. Por ejemplo, si una función espera un argumento de tamaño fijo, pero no verifica si el argumento es válido o no, es posible que la función produzca un error de Stack Overflow si el argumento es demasiado grande. De manera similar, si una función no valida los datos de entrada antes de procesarlos, es posible que el programa entre en un ciclo infinito o que se produzca un error de Stack Overflow.

Es importante destacar que el problema de Stack Overflow puede ser difícil de detectar y solucionar, ya que puede aparecer como un comportamiento inesperado y aleatorio en el programa. Además, puede ser difícil reproducir el problema, ya que puede depender de la configuración del sistema y de la cantidad de memoria disponible en el momento de la ejecución del programa.

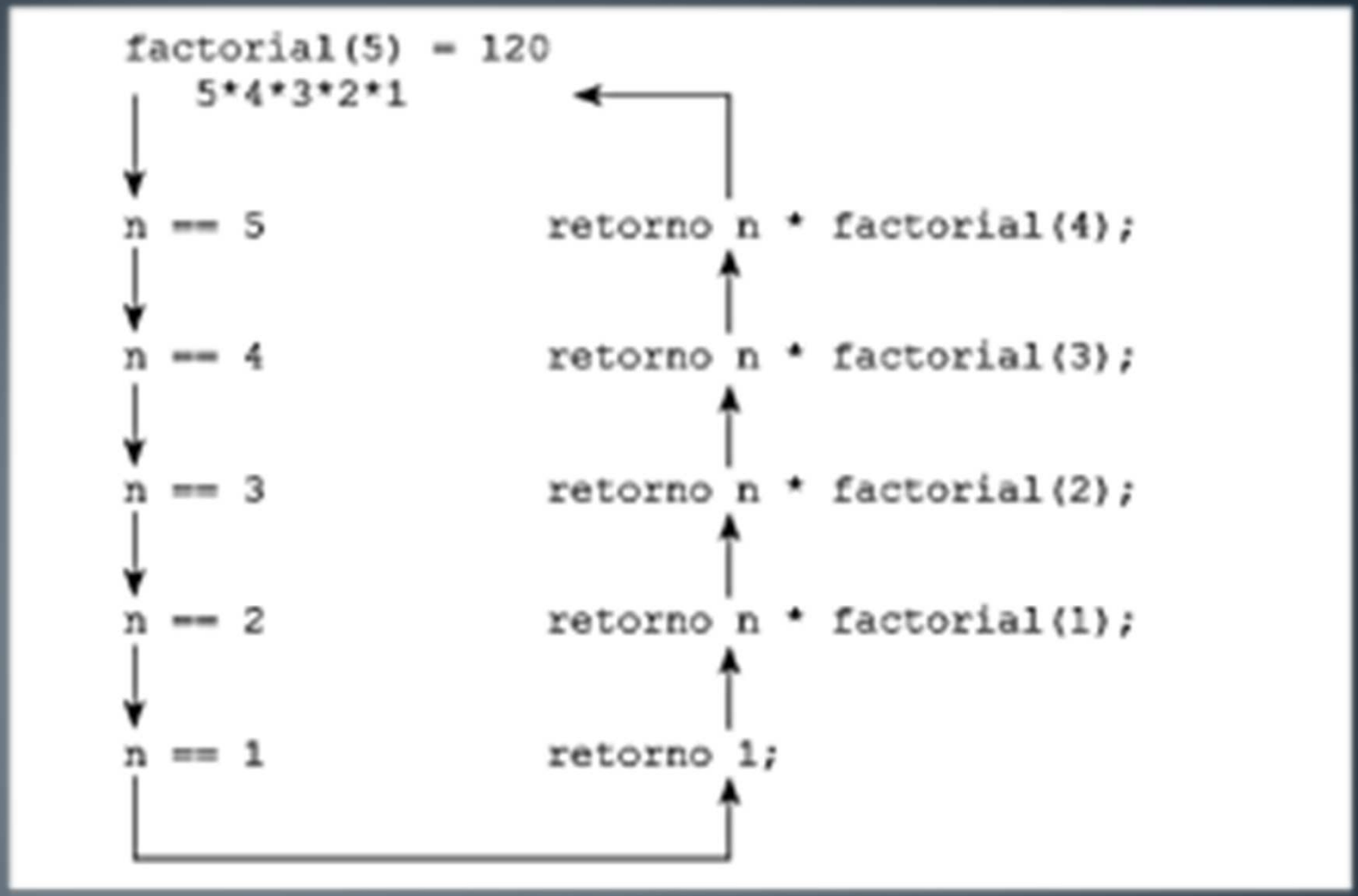
Por esta razón, es esencial que los programadores utilicen buenas prácticas de programación y técnicas de depuración para prevenir y solucionar problemas de Stack Overflow. Algunas buenas prácticas incluyen:

- Evitar ciclos infinitos y llamadas recursivas sin una condición de salida.
- Verificar los argumentos y los datos de entrada antes de procesarlos.

- Utilizar estructuras de datos alternativas, como colas y listas, en lugar de recursión para solucionar problemas complejos.
- Realizar pruebas exhaustivas y rigurosas para asegurarse de que el programa se ejecute de manera eficiente y estable.
- Monitorizar la memoria y el uso de la pila durante la ejecución del programa para detectar problemas de Stack Overflow a tiempo.

Al seguir estas buenas prácticas y técnicas, los programadores pueden prevenir y solucionar problemas de Stack Overflow y garantizar que sus programas se ejecuten de manera estable y eficiente.

Recordar, que el Stack Overflow es simplemente cuando se queda sin memoria necesaria para seguir llamando a dicha función, por lo cual se presenta el error y se desborda la pila.



Función recursiva para factorial