Tema III: Recursividad

Recursividad. Naturaleza. Definición. Implementación. Recursividad directa e indirecta. Recursividad Infinita. Ventajas y desventajas. Comparación con procesos iterativos.

Objetivos

1. Definir concepto de recursividad
2. Plantear su uso para formular soluciones a problemas que por sus características requieren diseñar algoritmos que deben llamarse a si mismos.
3. Presentar una estrategia de implementación de problemas recursivos
4. Análisis comparativo entre soluciones iterativas clásicas y las soluciones recursivas

3.1 Recursividad. Naturaleza. Definición.

3.1.1 Naturaleza

Se dice que **un objeto es recursivo cuando forma parte de sí mismo.**

- **Permite definir un** número infinito de objetos mediante un enunciado finito

En programación…

- La recursividad es la propiedad que tienen los procedimientos y funciones de llamarse a sí mismos para resolver un problema.
- Permite describir un número infinito de operaciones de cálculo mediante un programa recursivo finito **sin implementar de forma explícita estructuras repetitivas.**

Ejemplos de definiciones recursivas:

Números naturales

- 0 es un número natural.
- El sucesor del número natural x (sucesor(x)) es también un número natural.

Factorial de un número

- 0! = 1
- Si n es mayor que 0, n! = n * (n-1)!

Potencia de un número.

- $x^0 = 1$
- Si y > 0, $x^y = x * x^{y-1}$

Premisas

- Las definiciones recursivas suelen responder a funciones que se definen en base a un caso menor de sí mismas. Pero la recursividad en programación tiene otras implicaciones.
- Substancial diferencia entre una función matemática y una función programada.
- La recursividad en programación, aunque está permitida en prácticamente todos los lenguajes modernos, no es una herramienta demasiado útil en un entorno productivo.

¿Cuándo debo utilizar entonces recursividad?

3.1.2 Definición

La **recursividad** es una herramienta que permite expresar la resolución de un problema evolutivos, donde es posible que un módulo de software se invoque a sí mismo en la solución del problema (*Garland*, 1986) (*Helman*, 1991) (*Aho*, 1988).

Esta técnica puede entenderse como un caso particular de la programación con subprogramas en la que se planteaba la resolución de un problema en términos de otros subproblemas más sencillos.

El concepto de recursión aparece en varias situaciones de la vida cotidiana.

Otro tipo de recursión es la referente a los tipos de datos que se definen en función de si mismos.

La recursión como herramienta de programación permite definir un objeto en términos de sí mismo. (*listas circulares).*

Funciones recursivas en matemática.

3.2 Implementación

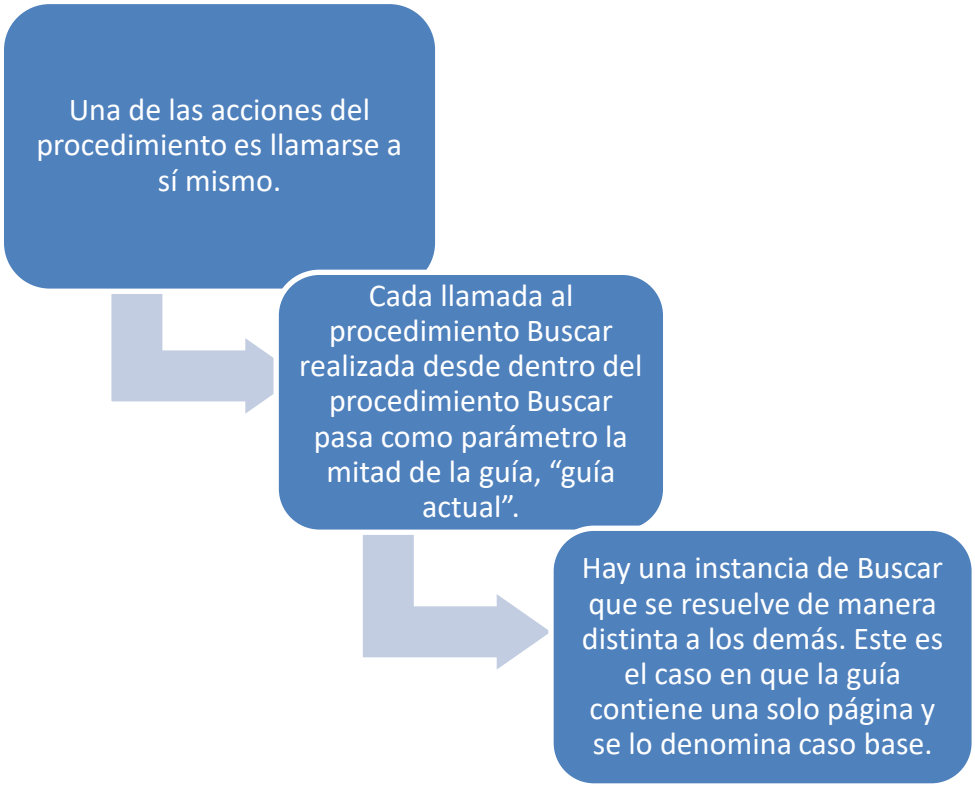
Si tenemos un problema de buscar una persona en una guía alfabética con los siguientes considerandos:

Si la guía contiene una sola página
Entonces Buscar secuencialmente la persona dentro de la página
Sino
Abrir la guía a la mitad
Determinar en qué mitad está la persona
Si la persona está en la primera mitad
Entonces Buscar en la primera mitad de la guía
Si no
Buscar en la segunda mitad de la guía
Fin

Tres observaciones sobre estos considerandos:

1. Una vez dividida la guía, y determinada la parte que contiene a la persona, el método de búsqueda a aplicar sobre esa mitad es el mismo que el empleado para la guía completa.
2. La mitad de la guía donde no está el dato se descarta, lo cual significa una reducción del espacio del problema (eficiencia algorítmica).
3. Hay un caso especial que se resuelve de una manera diferente que el resto, y sucede cuando la guía queda reducida a una sola página (luego de varias subdivisiones).

Al escribir como procedimiento podemos hacer las siguientes observaciones:



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.

Razonamiento Recursivo: Un razonamiento recursivo tiene dos partes

- Casobase (B)
- Regla recursiva de construcción (R)

Un conjunto de objetos está definido recursivamente siempre que:

(B) Algunos elementos del conjunto se definan explícitamente

(R) El resto de los elementos se definan en términos de los ya definidos.

Resumen de conceptos

Un programa o subprograma que se llama a si mismo se dice que es recursivo.

El concepto de recursividad está ligado, en los lenguajes de programación, al concepto de procedimiento o función.

La recursividad es una de las formas de control más importantes en la programación.

Los procedimientos recursivos son la forma más natural de representación de muchos algoritmos.

Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción. La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

Un caso de estudio

Un ejemplo fácil de ver y que se usa a menudo es **el cálculo del factorial de un número entero**. El factorial de un número se define como ese número multiplicado por el anterior, éste por el anterior, y así sucesivamente hasta llegar a 1. Así, por ejemplo, el factorial del número 5 sería: 5x4x3x2x1 = 120.

Definición

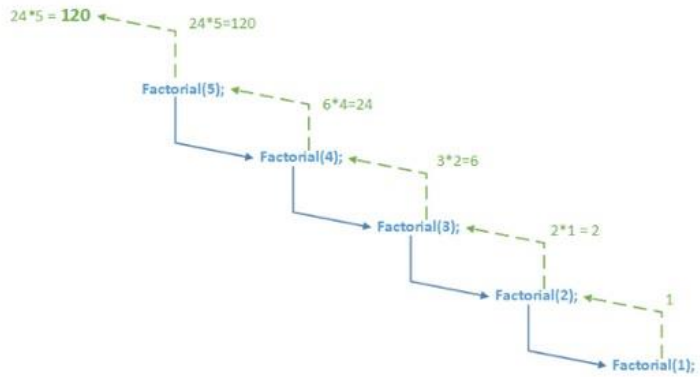
- Definición del Factorial
  - Factorial (n) = n * n-1 * n-2 * n-3 * …* 1 para n>0
  - Factorial (n) = 1 si n es igual a 0

- La definición Recursiva del Factorial es
  - Factorial (n) = 1 si n=0
  - Factorial (n) = n * Factorial (n-1) si n > 0

Tomando el factorial como base para un ejemplo, *¿cómo podemos crear una función recursiva que calcule el factorial de un número*?

```
# include <stdio.h>
long int factorial (long int i); /*
declaración de la función factorial */
int main ()
{
   int n;
   printf ("\n n = ");
   scanf ("%d", n);
   printf ("\n n! = %d\n", factorial
(n));
}
```

Aquí lo que se hace es que **la función se llama a sí misma** (eso es recursividad), y deja de llamarse cuando se cumple la **condición de parada**: en este caso que el argumento sea menor o igual que 1 (que es lo que hay en el condicional).



Es decir, cuando llamamos a la primera función, ésta se llama a sí misma pero pasándole un número menos y así sucesivamente hasta llegar a la última (la que recibe un 1 y por lo tanto deja de hacer más llamadas). En el momento en el que alguna de ellas empieza a devolver valores "hacia atrás", regresa la llamada a cada una de ellas, los valores devueltos se van multiplicando por el parámetro original en cada una de ellas, hasta llegar arriba del todo en el que la primera llamada devuelve el valor buscado.

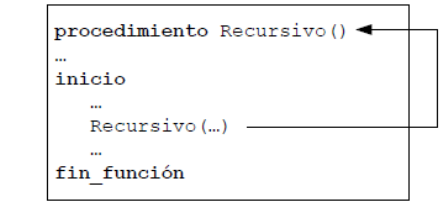
Es un concepto que cuesta un poco entender pero que es muy poderoso.

3.3 Recursividad directa e indirecta.

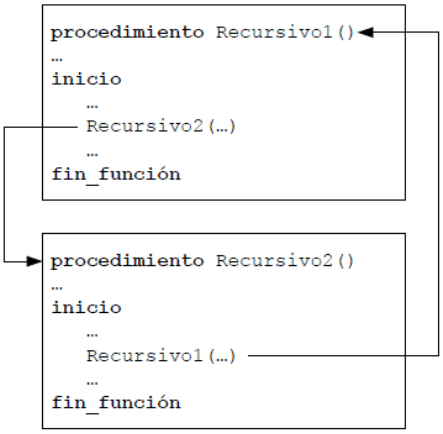
Se habla de recursión **directa** cuando la función se llama a sí misma. Es una construcción de programación que permite a una función llamarse a sí misma de forma directa.

Se habla de recursión **indirecta** cuando, por ejemplo, una función A llama a una función B, que a su vez llama a una función C, la cual llama a la función A.

La recursión indirecta o cruzada **es cuando una función involucra a otra o varias** hasta llegar a ella misma. Esta conlleva muchos más pasos, por ello no es tan popular a menos que el mismo código amerite su uso. Además, su implementación genera sobrecarga en el código. Esto se debe a que involucra muchas funciones.



Recursividad directa



Recursividad indirecta

3.4 Recursividad Infinita

Un bucle infinito ocurre si la prueba o test de continuación del bucle nunca se vuelve falsa. Una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de la salida.

La recursión infinita significa que cada llamada recursiva produce otra llamada recursiva y esta a su vez otra llamada recursiva, y así para siempre. En la práctica, dicha función se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

Es muy importante que toda función recursiva tenga un caso en el que no se llame a sí misma, o las llamadas serían infinitas y el programa no tendría fin. Por eso, siempre una función recursiva tiene una condición inicial en la que no debe llamarse a sí misma.

3.5 Ventajas e inconvenientes

Ventajas:

1. Soluciones a problemas complejos de una manera más fácil, simple, clara y elegante.
2. No es necesario definir la secuencia de pasos exacta para resolver el problema.
3. Podemos considerar que “tenemos resuelto el problema” (de menor tamaño).
4. La eficiencia de la recursividad reside en el hecho de que se puede usar para resolver problemas de difícil solución iterativa.
5. Algunos problemas son más sencillos de implementar usando la recursividad.
6. Presenta una facilidad para comprobar y verificar que la solución es correcta.
7. Mayor simplicidad del código en problemas recursivos.
    - Si un problema se puede definir fácilmente de forma recursiva (por ejemplo, el factorial o la potencia) es código resultante puede ser más simple que el equivalente iterativo.
    - También es muy útil para trabajar con estructuras de datos que se pueden definir de forma recursiva, como los árboles.
8. Posibilidad de “marcha atrás”: *backtracking*.
    - Las características de la pila de llamadas hacen posible recuperar los datos en orden inverso a como salen, posibilitando cualquier tipo de algoritmo que precise volver hacia atrás.

Desventajas:

1. Ineficiencia.
2. Sobrecarga asociada con las llamadas a subalgortimos.
3. Una simple llamada puede generar un gran número de llamadas recursivas. (Fact(n) genera n llamadas recursivas).
4. Es necesario la creación de muchas variables lo que puede ocasionar problemas en memoria.
5. En general, una función recursiva toma más tiempo en generarse que una iterativa.
6. Mayor uso de la pila de memoria.
    - Cada llamada recursiva implica una nueva entrada en la pila de llamadas dónde se cargará tanto la dirección de retorno como todos los datos locales y
    - argumentos pasados por valor.
    - El tamaño que reserva el compilador a la pila de llamadas es limitado y puede agotarse, generándose un error en tiempo de compilación.
7. Mayor tiempo en las llamadas

Cada llamada a un subprograma implica:

- Cargar en memoria el código del procedimiento.
- Meter en la pila la dirección de retorno y una copia de los parámetros pasados por valor.
- Reservar espacio para los datos locales.
- Desviar el flujo del programa al subprograma, ejecutarlo y retornar al programa llamador.

Conclusión:

Es apropiada cuando el problema a resolver o los datos que maneja se pueden definir de forma recursiva o cuando se debe utilizar *backtracking*.

No es apropiada si la definición recursiva requiere llamadas múltiples.

Un programa recursivo puede ser menos eficiente que uno iterativo en cuanto a uso de memoria y velocidad de ejecución.

3.6 Comparación con procesos iterativos

Tanto la recursión como la iteración se usan para ejecutar algunas instrucciones repetidamente hasta que alguna condición sea verdadera. Pero existe una diferencia importante entre ambas, primero recordamos los conceptos: Recursividad se refiere a una situación en la que una función se llama a sí misma una y otra vez, iteración permiten repetir una sentencia o conjunto de ellas.

La recursión y la iteración son técnicas de programación que se suelen utilizar en programación para resolver rápidamente problemas complejos y repetitivos. Un programa recursivo simplifica un problema y lo resuelve desde abajo hacia arriba. Un proceso iterativo repite un proceso una y otra vez empezando cada nueva iteración con el resultado de la iteración anterior. El objetivo principal de estas técnicas es acelerar la ejecución de un programa.

La diferencia más importante entre las operaciones recursivas y las iterativas es que los pasos de una operación iterativa se realizan uno cada vez y dirigen la ejecución directamente al siguiente paso. En una operación recursiva, cada paso después del paso inicial es una réplica del paso anterior. Además, desde arriba hacia abajo, cada paso es un poco más sencillo que el que hay justo "encima". Al final de la operación, todas las soluciones se combinan para resolver el problema.

Recursividad versus iteración

- Si un subprograma se llama a si mismo se repite su ejecución un cierto número de veces.
- Los procesos recursivos suelen ocupar más memoria y tardar más que los iterativos.

Solución Iterativa	Si X >= 0	X! = 1*2*3*4….*X
Solución recursiva	Si X = 0 Si X > 0	X! = 1 X! = X * (X-1)!

Solución Iterativa

```
int fact = 1
for (int i= 1; i<= x;i++
     Fact = fact * i;
Return fact;
```

Solución recursiva

```
If ( ==0)
     Return 1;
else
     Return
     (x*factorial(x-1));
```