Recursividad

parte I

Departamento de Computación

Algoritmos y Programación I

Joel Rivas

Situaciones del mundo real

- Un espejo contra otro espejo

Situaciones del mundo real

- **Matrioska**

Situaciones del mundo real

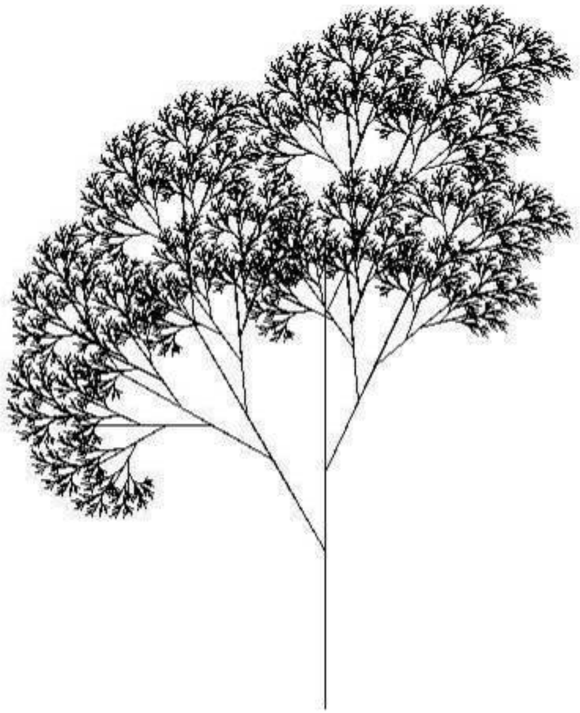
- **Soñar**

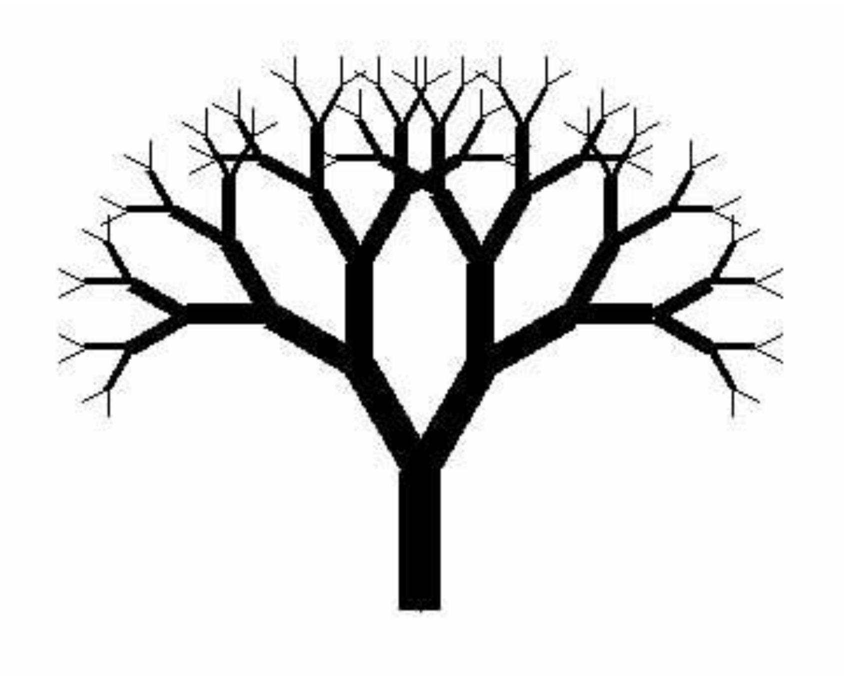
Situaciones del mundo real

- Flores

Situaciones del mundo real

- **Árboles**



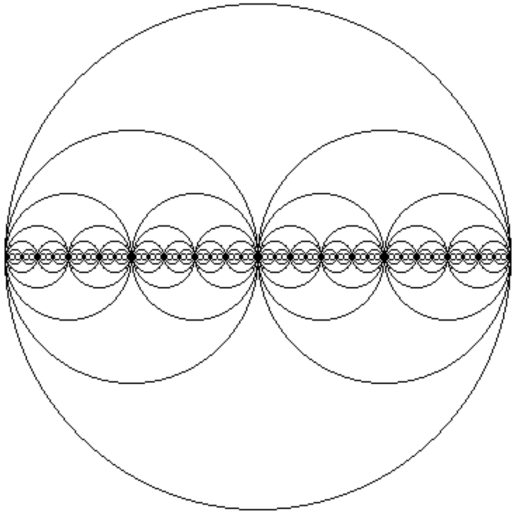


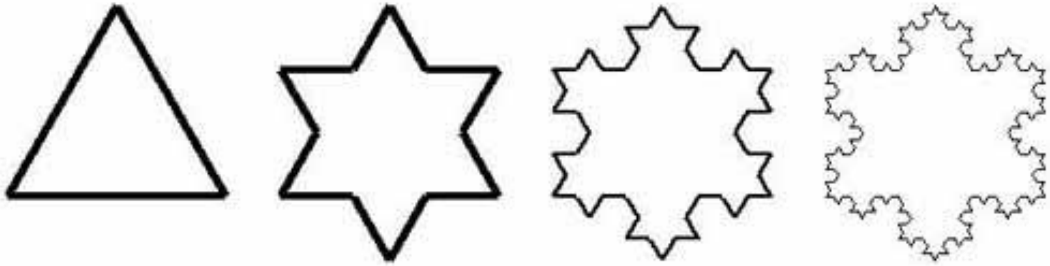
Situaciones del mundo real

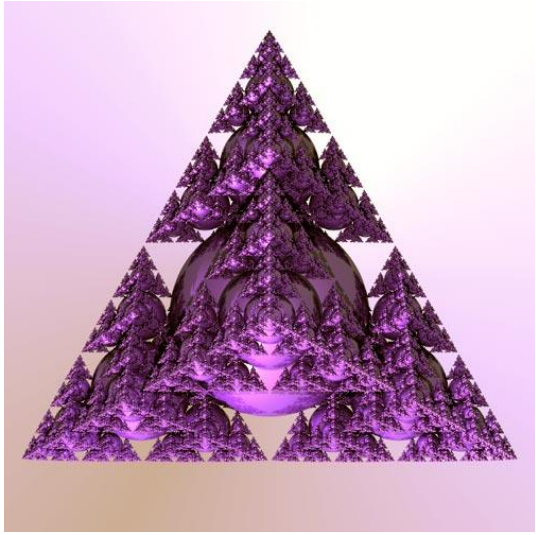
- **Conversación:**

Situaciones del mundo real

- **En general:**







Introducción

Recursividad › nueva forma de aplicar acciones repetitivas o iterativas › un subprograma se llama a sí mismo para resolver una versión más pequeña del problema original (generalmente parámetros)

Se necesita una

Relación de recurrencia

El objeto a definir recursivamente debe tener una relación entre sus componentes de manera que haga posible expresarlos o describirlos en base a estados o casos anteriores.

Uso de la Recursión

Cuando hay que repetir cierto tratamiento o cálculo (disponiendo de una relación de recurrencia), donde el número de repeticiones es variable.

Definición

Recursión

Técnica de programación muy potente, que consiste en definir una acción nominada (función o procedimiento) en términos de si misma.

Es utilizada comúnmente como una alternativa a la iteración.

Cláusulas para una definición recursiva

Cláusula base
para algunos argumentos (parámetros)

Es la cláusula donde se resuelve el problema para el caso más simple, por lo cual no se hace una llamada recursiva sino se retorna el resultado básico obtenido, permitiendo parar el proceso de llamadas recursivas y evitando que se genere un ciclo infinito de llamadas.

Cláusula recurrente

Es la cláusula donde se aplica la relación recurrente, por lo cual se realizan llamadas a la misma acción nominada. Una llamada siempre debe tender a una cláusula base.

representa cláusula base

cláusula recurrente: dentro de la muñeca cabe otra de igual forma un poco más pequeña

Estructura o patrón de un proceso recursivo

Entonces un método recursivo debe tener 2 partes:

Caso base
Solución trivial

Una solución trivial o solución básica para alguno de los argumentos.

Esta parte constituye la terminación o condición de parada en la que se dejan de hacer llamadas recursivas y al retornar se comienza a resolver el problema mas grande.

Siempre debe existir al menos un caso base.

Casos generales
Relación de recurrencia

Una relación de recurrencia (Casos Generales) mediante la cual se definen los valores sucesivos. Esta relación de recurrencia esta representada por un conjunto de llamadas recursivas.

Los casos generales siempre deben avanzar hacia un caso base. Es decir, la llamada recursiva se hace a un subproblema más pequeño y, en última instancia, los casos generales alcanzarán un caso base.

Ejemplo

Factorial de n $\in$ N.

$n! = n*(n-1)*(n-2)* \ldots *2*1$

$n! = n*(n-1)!$ **relación de recurrencia**

$4! = 4*3*2*1$

$3! = 3*2*1$

$2! = 2*1$

$1! = 1$

$4! = 4*3!$

$3! = 3*2!$

$2! = 2*1!$

$1! = 1$ **caso base**

$4! = 4*6 = 24$

$3! = 3*2 = 6$

$2! = 2*1 = 2$

Recuérdese también que aunque $0 \notin N$, se definió 0!

$0! = 1$

Ejemplo

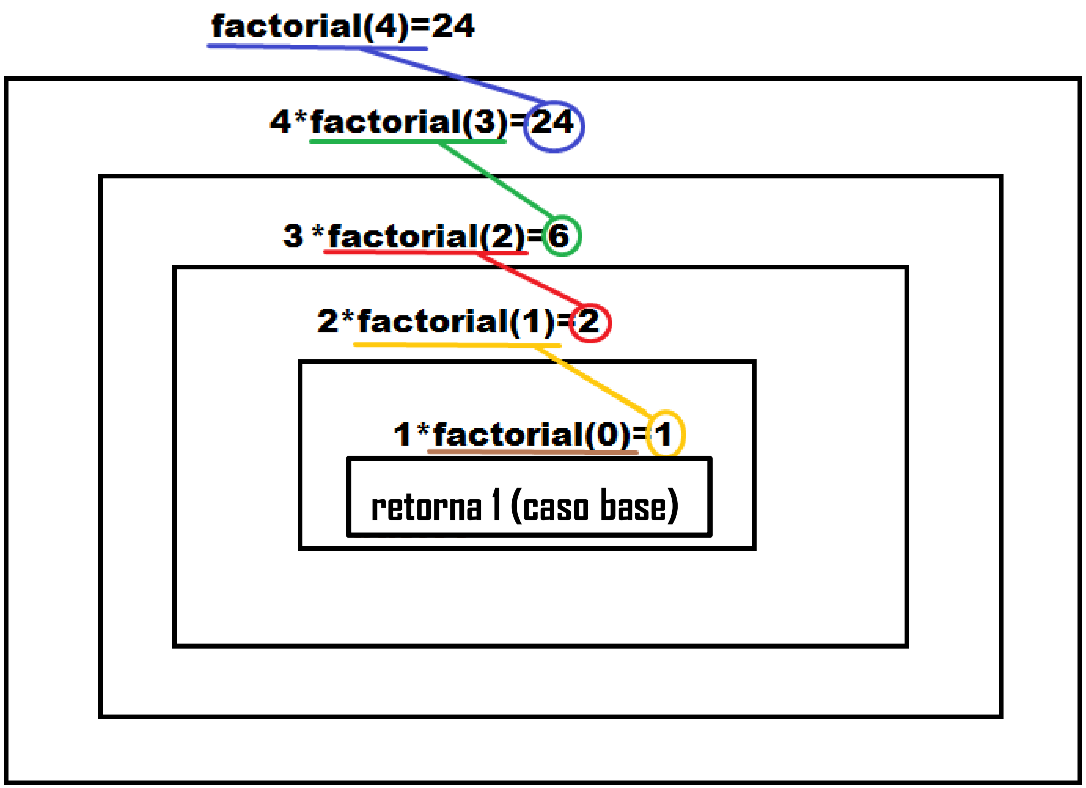
Factorial de $n \in N$.

$$\text{factorial}(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * \text{factorial}(n-1) & \text{si } n > 0 \end{cases}$$

Ejemplo

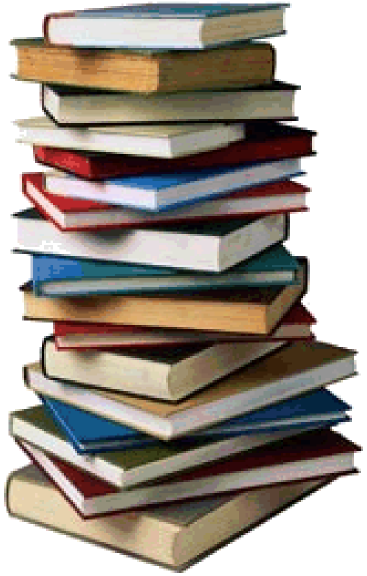
Factorial de n ∈ N.

```
func factorial(entero n): entero
inicio
   si (n = 0) entonces
      retornar(1)
   sino
      retornar(n * factorial(n-1) )
   fsi
ffunc   // fin factorial
```



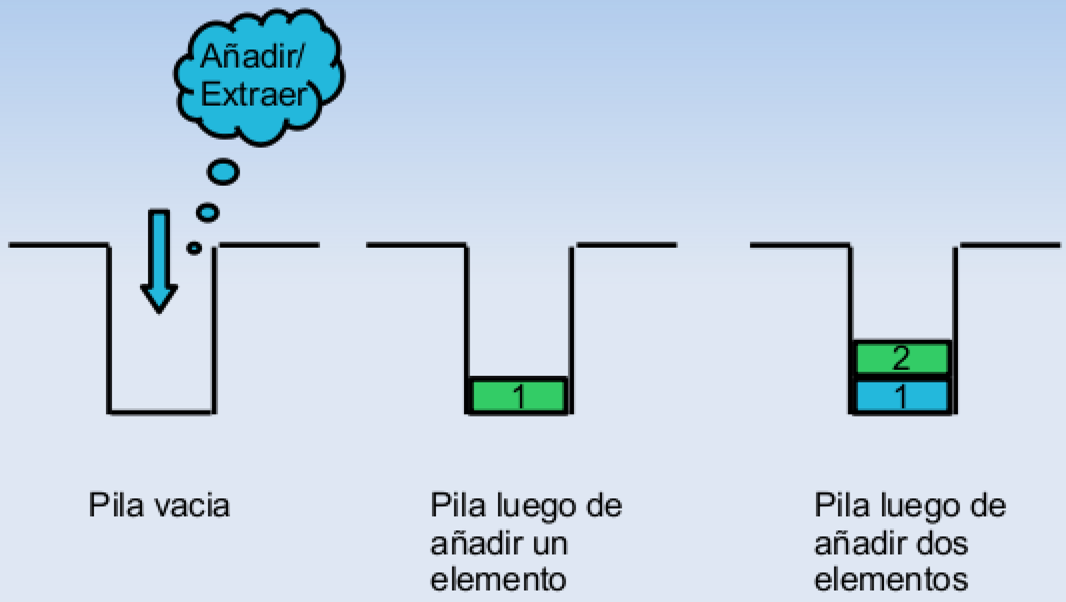
Implementación: Uso de una Pila

Pila.



Implementación: Uso de una Pila

Pila.



- Sólo elemento coloreado en verde es accesible

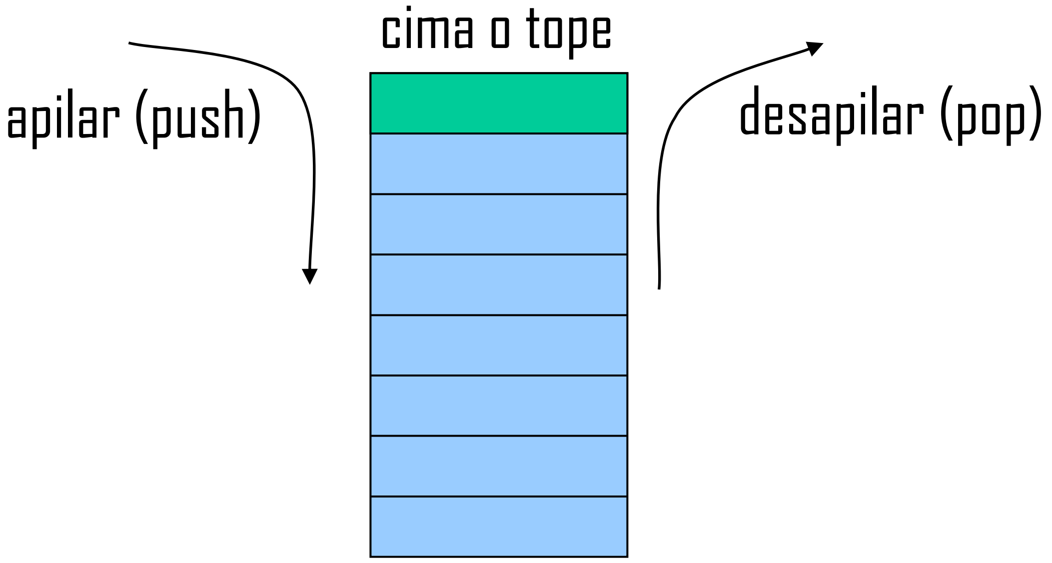
Implementación: Uso de una Pila

Pila.

Es una estructura de datos lineal que brinda acceso sólo al último elemento agregado; por eso, el modo de acceso a sus elementos es de tipo:

LIFO (del inglés last in first out, es decir, "último en entrar, primero en salir").

Siempre podemos agregar más elementos a la pila, pero cada vez que lo hacemos, el elemento agregado más recientemente se convierte en el elemento que puede ser eliminado primero

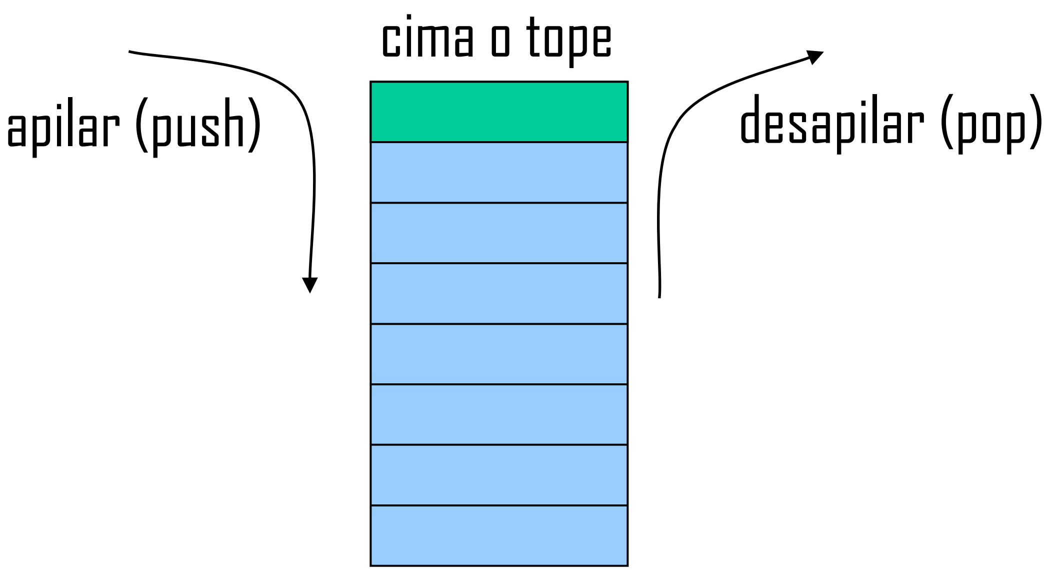


Implementación: Uso de una Pila

Pila.

Por este modo de almacenar y recuperar datos, las pilas son una estructura importante en la Ciencia Computacional, y tienen muchas aplicaciones diferentes:

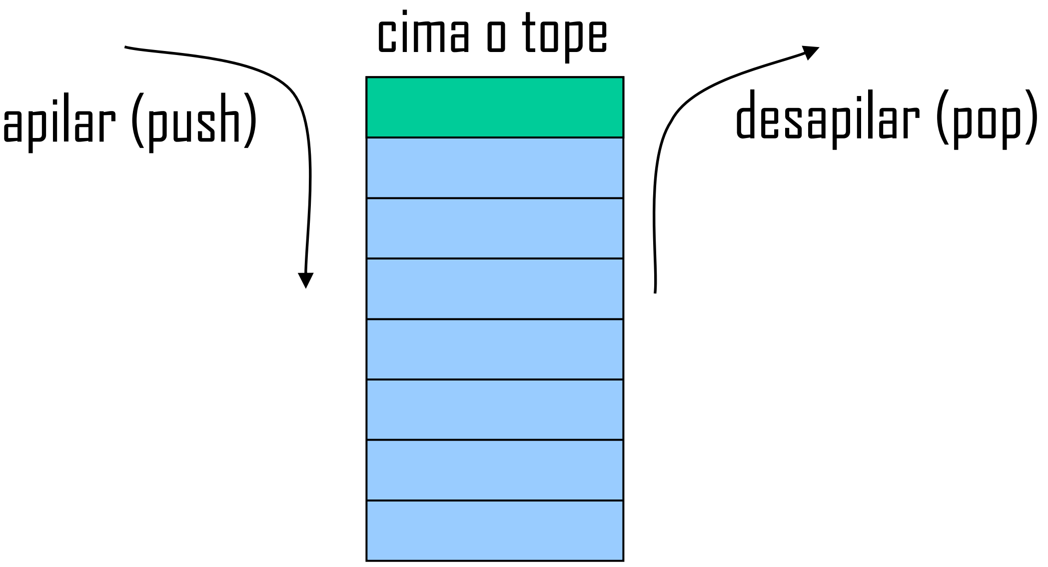
- Aplicaciones de tareas sencillas como invertir los caracteres en una cadena,
- Aplicaciones de tareas más complejas como la evaluación de expresiones aritméticas o ayuda a recorrer un árbol binario o a buscar vértices de grafos.



Implementación: Uso de una Pila

Pila.

- La mayoría de los procesadores utilizan una arquitectura basada en pilas,
- Los microprocesadores y lenguajes de programación usan pilas cuando se hace la llamada a una acción nominada, almacenando en el tope de la pila, la dirección de retorno, los argumentos (parámetros) y variables locales de la acción nominada.



Implementación: Un programa en memoria

Uso de la memoria durante la ejecución de un programa.

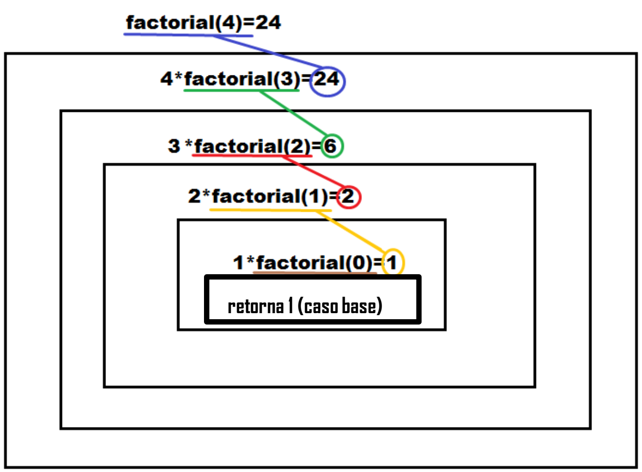
Durante la ejecución de un programa, se utilizan varias zonas de memoria diferenciadas para guardar el código, el contexto de la ejecución, los datos, etc... Estas zonas de memoria son:

- La pila de llamadas (call stack).
- El área de datos dinámicos (heap).
- El área de datos estáticos.
- El área del código.

Implementación: Uso de una Pila

Factorial de n ∈ N.

```
func factorial(entero n): entero
inicio
   si (n = 0) entonces
      retornar(1)
   sino
      retornar(n * factorial(n-1) )
   fsi
ffunc   // fin factorial
```



retornar(1), dir_llamada, n = 0
retornar(1 * factorial(0)), dir_llamada, n = 1,
retornar(2 * factorial(1)), dir_llamada, n = 2,
retornar(3 * factorial(2)), dir_llamada, n = 3,
retornar(4 * factorial(3)), dir_llamada, n = 4,

solo para explicar, esto no está en la pila

tipo de dato: apuntador

Pila de llamadas (call stack)

Iteración frente a Recursión

Factorial de n $\in$ N.

```
func factorial(entero n): entero
var
   entero fact, i
inicio
   fact ← 1
   si (n ≠ 0) entonces
      para i ← n hasta 1 en -1 hacer
         fact ← fact * i
      fpara
  fsi
   retornar(fact)
ffunc   // fin factorial
```

solo para explicar, esto no está en la pila

retornar(fact),
i
fact
dir_llamada,
n = 4

Pila de llamadas (call stack)

Iteración frente a Recursión

Característica	Iteración	Recursión
Estructura de control	Utiliza una estructura repetitiva	Utiliza una estructura de selección
Repetición	Utiliza explícitamente una estructura repetitiva	Consigue la repetición mediante llamadas recursivas a funciones.
Condición de salida	Finaliza cuando la condición del bucle no se cumple	Finaliza cuando se reconoce un caso base (la condición de salida se alcanza)
Eficiencia	Es eficiente en cuanto a la ocupación de memoria de algunos problemas pero ineficientes en algunos métodos de ordenación	Es ineficiente en ocupación de espacio de memoria y de tiempo de ejecución pero eficiente en algunos métodos de ordenación

Para resolver un problema recursivo

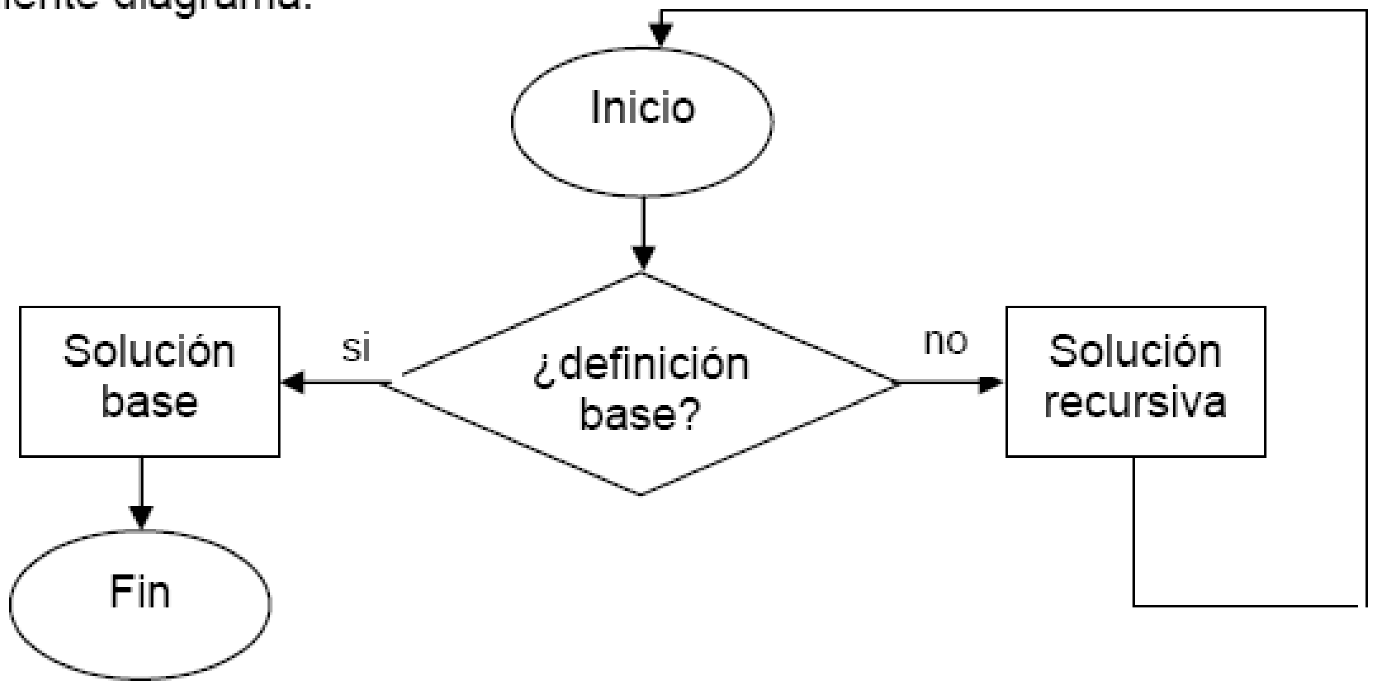
La solución recursiva a un problema de repetición se obtiene respondiendo dos preguntas:

1) **¿Cómo se resuelve el caso más pequeño del problema?**
   La respuesta a esta pregunta debe ser no-recursiva, permitiendo plantear una condición de salida, es decir, proporcionar el caso base.

2) **¿Cómo se resuelve un caso general del problema, sabiendo que ya se tiene el caso anterior más pequeño?**

Estructura de Control de un proceso recursivo

De una manera general un módulo que implementa un proceso recursivo tiene el siguiente diagrama:



Dependiendo del problema que implemente la acción nominada (el módulo) éste diagrama se puede ver modificado. Por ejemplo, si tiene más de una definición base, si tiene más de una definición recursiva o si en la solución recursiva tiene que realizar alguna operación antes de volverse a ejecutar o no

Ejemplo

Serie de Fibonacci.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

$\boxed{0 + 1 = 1}$

$\boxed{1 + 1 = 2}$

$\boxed{1 + 2 = 3}$

$\boxed{2 + 3 = 5}$

Ejemplo

Serie de Fibonacci.

$$
\text{fibonacci}(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2) & \text{si } n \geq 2 \end{cases}
$$

Ejemplo

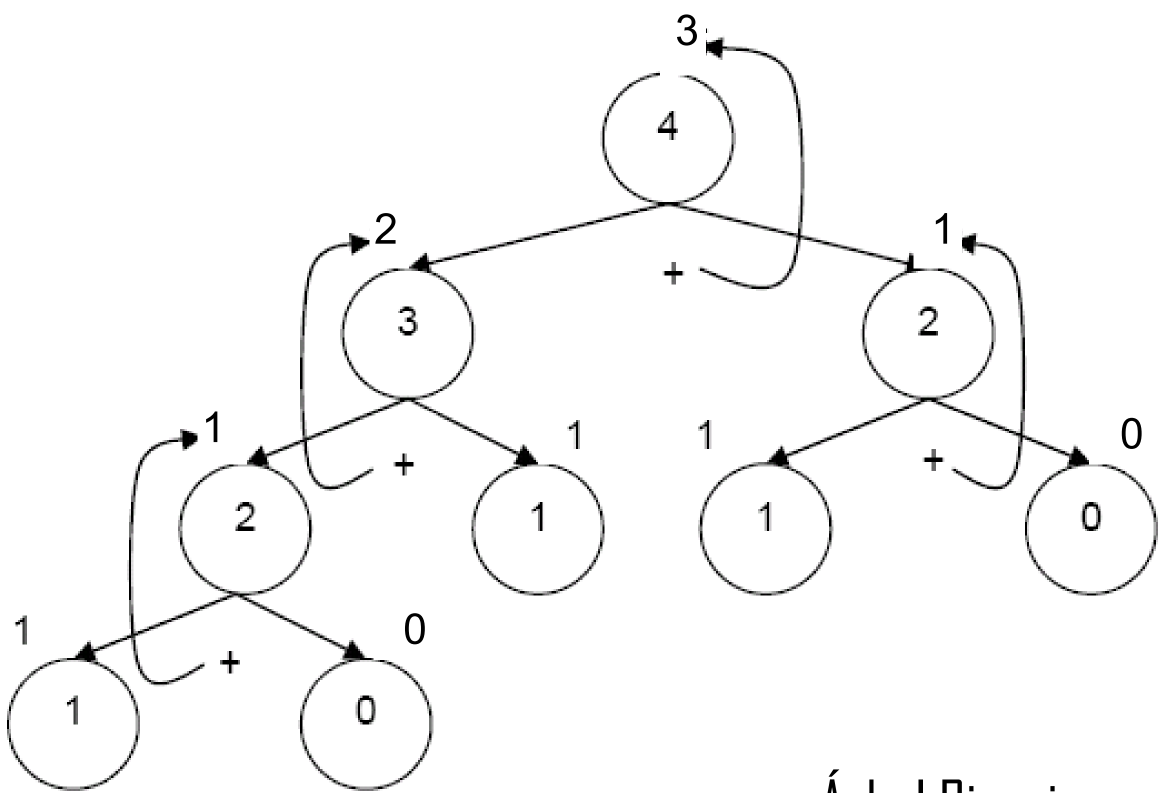
Serie de Fibonacci.

```
func fibonacci(entero n): entero
inicio
   si (n = 0 ∨ n = 1) entonces
      retornar(n)
   sino
      retornar( fibonacci(n–1) + fibonacci(n–2) )
   fsi
ffunc   // fin fibonacci
```

Ejemplo

Serie de Fibonacci.

La ejecución de Fibonacci(4) gráficamente, no formalmente, la podemos ver de la siguiente manera:



Árbol Binario

Ejemplo

Función de Ackermann, A(m, n) donde m, n $\in Z^+$.

$$
Ack(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ Ack(m-1, 1) & \text{si } m > 0 \wedge n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{si } m > 0 \wedge n > 0 \end{cases}
$$

Ejemplo

Función de Ackermann, A(m, n) donde m, n $\in Z^+$.

<u>func</u> ack(entero m, entero n): entero
<u>inicio</u>
  <u>si</u> (m = 0) **<u>entonces</u>**
    <u>retornar</u>(n+1)
  <u>sino</u> // m > 0
    <u>si</u> (n = 0) **<u>entonces</u>**
      <u>retornar</u>(ack(m–1, 1))
    <u>sino</u> // n > 0
      <u>retornar</u>(ack(m–1, ack(m, n–1)))
    <u>fsi</u>
  <u>fsi</u>
<u>ffunc</u> // fin fibonacci

Recursividad Anidada:
En este tipo de recursividad la función recibe como uno de sus parámetros una llamada recursiva.

Clasificación

(**a**) **Recursividad lineal**: es aquella donde el cuerpo de la función contiene una llamada explicita a si misma.
Ejemplo: Factorial

(**b**) **Recursividad no lineal**: el cuerpo de la función contiene varias llamadas explícitas a si mismas.
Cuando se tiene dos llamadas se habla de recursividad binaria.
Ejemplo: Fibonacci

(**c**) **Recursividad anidada**: el cuerpo de la función contiene llamadas explícitas a si mismas, pero recibe como uno de sus parámetros una llamada recursiva.
Ejemplo: Ackermann

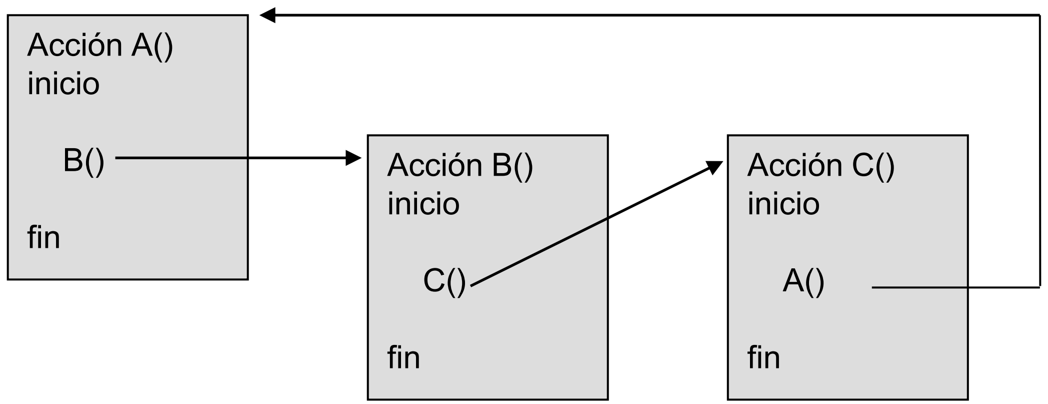
Las tres anteriores forman parte de lo que se conoce como **Recursión Directa**.

Clasificación

(**d**) **Recursividad Mutua**: en este caso la recursividad se presenta cuando varias acciones nominadas se llaman entre sí.

Este tipo de recursividad también se conoce como **recursividad indirecta**.

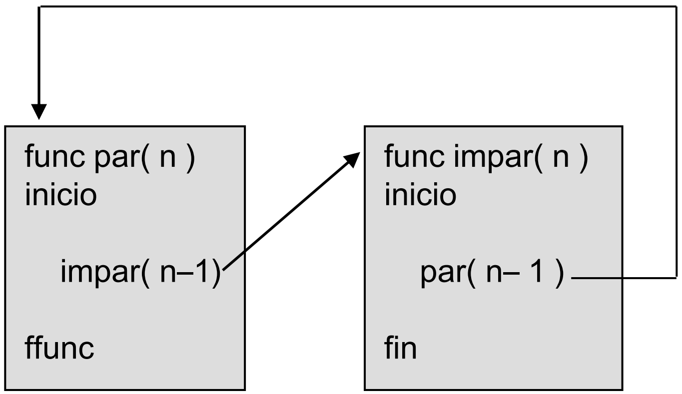
Cuando una acción A, puede generar una secuencia de llamadas a otras acciones, que termina con una invocación a la acción original, se dice que A es indirectamente recursiva.



Ejemplo

```
func par(entero n): logico
inicio
   si (n = 0) entonces
      retornar(verdad)
   sino
      retornar( impar(n-1) )
   fsi
ffunc   // fin par
```

```
func impar(entero n): logico
inicio
   si (n = 0) entonces
      retornar(falso)
   sino
      retornar( par(n-1) )
   fsi
ffunc   // fin impar
```



Cuando usar recursión

1) **¿Cuándo usar recursividad?**

Los algoritmos recursivos son más cercanos a la descripción matemática.

Para simplificar la solución de algunos problemas.

Los algoritmos recursivos son más compactos para algunos tipos de problemas, son más legibles y más fáciles de ser comprendidos e implementados; en efecto, el programa que resulta es más sencillo y más elegante.

Se adapta mejor cuando la estructura de datos es recursiva; ejemplo : árboles.

En muchos problemas cuyo desarrollo es inherentemente recursivo.

2) **¿Cuándo no usar recursividad?**

Cuando las acciones nominadas reciban parámetros de valores altos o tengan muchas variables locales que ocupen bastante memoria, por ejemplo: arreglos largos.

Conclusiones

- Es importante señalar que la recursividad tiene gran cantidad de desventajas.
- La razón fundamental del uso de la misma es que existen numerosos problemas complejos que poseen naturaleza recursiva y en consecuencia son más fáciles de implementar con algoritmos de este tipo.
- En condiciones críticas de tiempo de ejecución y de memoria es recomendable utilizar la solución iterativa.
- Con pocos datos quizás no haya problemas, pero una gran cantidad de datos o datos grandes puede causar un desbordamiento de la zona de memoria de la pila (stack overflow.)

Nota:

Cualquier problema que pueda ser expresado de manera recursiva, se puede resolver también iterativamente, más no lo contrario.