Introducción a la Algorítmica y Programación (3300)

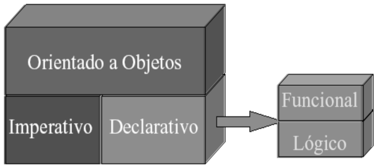
Prof. Ariel Ferreira Szpiniak - aferreira@exa.unrc.edu.ar
Departamento de Computación
Facultad de Cs. Exactas, Fco-Qcas y Naturales
Universidad Nacional de Río Cuarto

Teoría 16

Recursión o Recursividad

Paradigmas

- Un paradigma es un modelo de programación (un estilo) que engloba a ciertos lenguajes que comparten:
  - Elementos estructurales: ¿con qué se confeccionan los programas?
  - Elementos metodológicos: ¿cómo se confecciona un programa?
- Hay varias clasificaciones:
  - Orientado a Objetos, Imperativo, Declarativo (Funcional, Lógico).



  - Imperativo (por Procedimientos, POO), Declarativo (Funcional, Lógico), Concurrente.

Paradigmas

- **Imperativos**
  - Orientados hacia una máquina de estados:
    - variables.
  - Sentencias:
    - concepto básico.
    - cambio de estado: asignación, E/S
    - de control: secuencia, alternativa e iterativa.
  - Expresiones aritméticas y lógicas.
  - No orientados a nuestro modo de pensar.

Paradigmas

Paradigmas y lenguajes

- **Imperativo**: Fortran, Cobol, Algol, PL/I, Pascal, **C**, Modula-2, Ada
- **Funcional**: Lisp, ISWIN, Scheme, FP, Hope, ML, Miranda, Haskell, Gofer
- **Lógico**: Prolog
- **Orientado a objetos**: Smalltalk, C++, Eifell, Java

Programación Funcional

- Se caracteriza por el uso de **expresiones** y **funciones** para describir problemas.
- El concepto de **función** es el eje **central**.
- No hay modificación de estado. No hay asignación.
- Relaciones entre funciones mediante composición.
- La **recursión** enriquece el lenguaje.

Programación Funcional

Ventajas

- Formalismo matemático. Forma de pensar.
- Prototipación rápida.
- Razonamiento sobre las soluciones.
- Alto nivel de abstracción.
- Facilidad para dividir el problema en subproblemas.
- Aplicación en diversas áreas: IA, base de datos, lenguaje natural, computación simbólica, verificación de propiedades, web (Yahoo Store), lenguaje de extensión (AutoCAD, Emacs, Gimp).

Programación Funcional

Programas funcionales

- Están formados por un **conjunto de funciones**, donde el programa principal lo forma la función de más alto nivel.
- Expresan **qué** hay que calcular, y no **cómo** calcularlo.
- Son abstractos, pequeños y fáciles de mantener.
- Aumentan la legilibilidad y modificabilidad.
- No necesario manipular la memoria explícitamente.

Programación Funcional

Transparencia referencial

- **Propiedad** fundamental de las funciones matemáticas.
- El valor de una expresión depende solo del valor de sus subexpresiones.
- **Evita los efectos colaterales**.
- Una función posee esta característica si dados los mismos argumentos para su aplicación devuelve siempre el mismo resultado. Determinismo.
- Los efectos colaterales pueden producir que la función deje de cumplir con la propia definición.

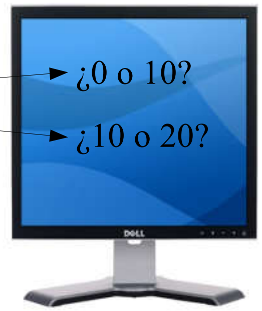
Programación Funcional

La Transparencia referencial evita los efectos colaterales que pueden tener los lenguajes como C.

```
#include <stdio.h>
int i;
int Incremento(int inc);

int main(){
  i=0;
  printf("%d \n",Incremento(10));
  printf("%d \n",Incremento(10));
  return(0);
}

int Incremento(int inc){
    i=i+inc;
    return(i);
}
```



Programación Funcional

- Técnicas
  - *Funciones*
    - definición (especificar)
    - aplicación (resolver)
  - *Composición*
  - Orden superior
  - **Recursión**

Programación Funcional

- Orden superior
  - Funciones que toman funciones como argumento o regresan funciones como resultado.
  - Las funciones pueden ser datos o resultados.
  - **Idea radical**: adiós a la metáfora de un proceso como una receta donde los ingredientes son los datos.
  - Las recetas manejan recetas como ingredientes. Meta-recetas.
  - Ejemplo
    - funciones de derivación y/o integración son de orden superior.
    - f(g, x, w) = g(x)+g(w)

Programación Funcional

- **Recursión**
  - posibilidad de definir una función en términos de sí misma.

Recursión o recursividad

- No solo se aplica en los lenguajes funcionales.
- Los lenguajes lógicos también usan fuertemente recursión.
- Los lenguajes imperativos y orientados objetos permiten usar recursión, aunque no son su fuerte.
- Nosotros analizaremos el concepto de recursión aplicado al **lenguaje C**.

Recursividad

Nociones Generales

Hemos visto 3 estructuras de programación:

- **Composición *Secuencial***
- **Composición *Condicional***
- **Composición *Iterativa***

Introduciremos un nuevo concepto que puede interpretarse como una nueva estructura de programación o como una metodología de resolución de problemas.

recursividad o recursión

Recursividad

Nociones Generales

La recursividad o recursión es otra forma de realizar repeticiones.

La recursión se puede aplicar tanto a ***funciones*** como a ***acciones***, a las que llamaremos *abstracciones*.

La recursión es natural en lenguajes funcionales.

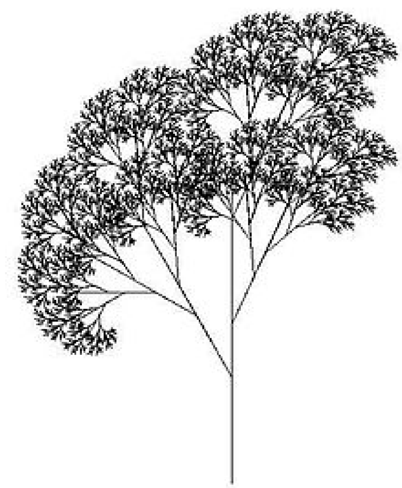
La iteración es natural en lenguajes imperativos.

Como estamos aprendiendo a construir algoritmos con estilo imperativo, y traduciendo a un lenguaje imperativo (C), veremos el concepto de recursión aplicado en éstos contextos.

Recursividad

Nociones Generales

- **Se dice que una acción o función es recursiva cuando se llama a sí misma.**
- **Existen numerosas situaciones en las que la recursividad aporta una solución simple y natural a un problema.**
- **La recursividad es una herramienta potente y útil en la resolución de problemas que tengan naturaleza recursiva.**



Árbol creado en el lenguaje Logo usando recursión.

Recursividad

Nociones Generales

Ejemplos sencillos de definiciones recursivas

• Un *ramo de rosas* es (1) una rosa, o (2) una rosa junto con un pequeño *ramo de rosas*.

• Una persona es *descendiente* de Platón si (1) esa persona es hija de Platón, o (2) esa persona es hija de un *descendiente* de Platón.

Las palabras *ramo de rosas* y *descendiente* se utilizan para definirse a sí mismas.

(1) Se denomina caso base y (2) caso inductivo

Recursividad

Nociones Generales

Una definición recursiva sigue el método de *inducción estructural y* puede dividirse en dos partes:

1. Caso base o Valor del dominio conocido

Cuando se conoce el valor que toma la abstracción para uno o más valores del dominio.

2. Caso Inductivo o Definición circular

Permite que la abstracción se defina en términos de sí misma suponiendo que se verifica la *hipótesis inductiva*.

Recursividad

Nociones Generales

Ejemplos:

<u>Factorial de un número natural</u>

0!=1

n!= n * (n-1)!

<u>Potencia de un número natural y exponente natural</u>

$a^0 = 1$ si $a \neq 0$

$a^1 = a$

$a^{n+1} = a \cdot a^n$

Recursividad

Ejemplos en Notación Algorítmica

<u>Función</u> factorial (<u>dato</u> n $\in$ N) $\rightarrow$ N

{Def: $(n_0=0 \wedge fact(n_0)=1) \vee (n_0>0 \wedge fact(n_0)=1*2*..*n_0)$}

<u>Inicio</u>

  <u>según</u>

    n=0: $\leftarrow$ 1

    n>0: $\leftarrow$ n * factorial(n-1)

  <u>fsegún</u>

<u>Ffunción</u>

Recursividad

Ejemplos en Notación Algorítmica

Invocación a la función factorial

```
Algoritmo FactorialDeUnNumero
Lexico
  m, fact ∈ N
  Función factorial (dato n ∈ N) → N
  Inicio
    según
      n=0: ← 1
      n>0: ← n * factorial(n-1)
    fsegún
  Ffunción
Inicio
 Entrada:m
 fact <- factorial(m)
 Salida:fact
Fin
```

o

```
Entrada:m
Salida:factorial(m)
```


Recursividad

Ejemplos en Notación Algorítmica

Supongamos la siguiente invocación: *factorial(2)*

Número de invocaciones	***n***	***factorial***
1	2	2 * *factorial* (1)
2	1	1 * *factorial* (0)
3	0	1

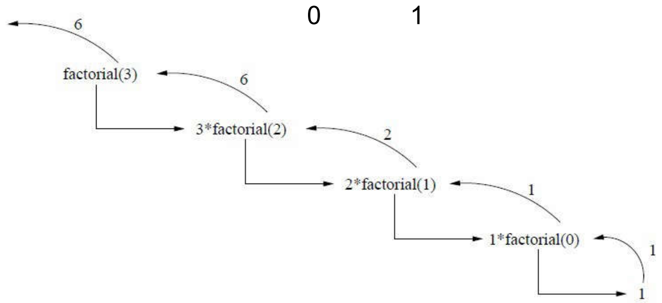
En la primera llamada debe llamarse a *factorial* con el valor *n* – 1 (1); en el segundo llamado, debe llamarse a *factorial* con el valor *n* – 1 (0); en el tercer llamado, que siendo *n* = 0, *factorial* = 1, podrá calcularse, "yendo hacia atrás", los distintos valores de la expresión. Esto, que el compilador se encarga de implementar, implica 'suspender' el cálculo de la función hasta tanto se obtenga un valor conocido para la función y volver a invocar la función mientras no se tenga un valor.

Recursividad

Ejemplos en Notación Algorítmica

Supongamos la siguiente invocación: *factorial(3)*

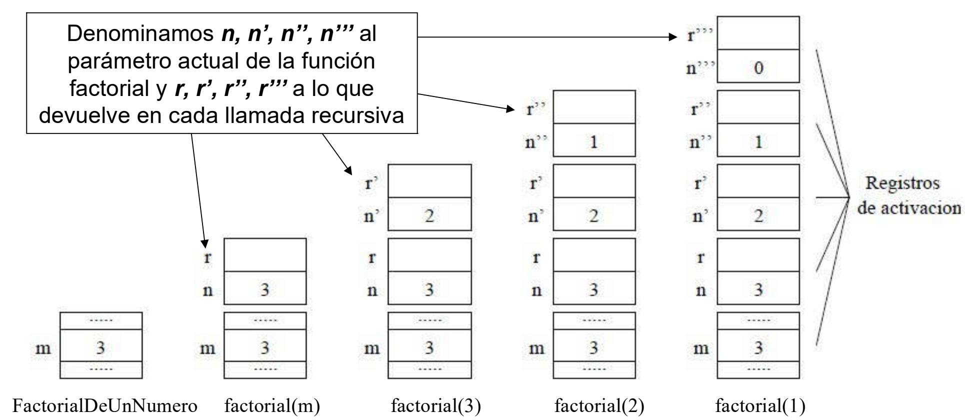
Número de invocaciones	***n***	***factorial***
1	3	3 * *factorial* (2)
2	2	2 * *factorial* (1)
3	1	1 * *factorial* (0)
4	0	1



Recursividad

Ejemplos en Notación Algorítmica

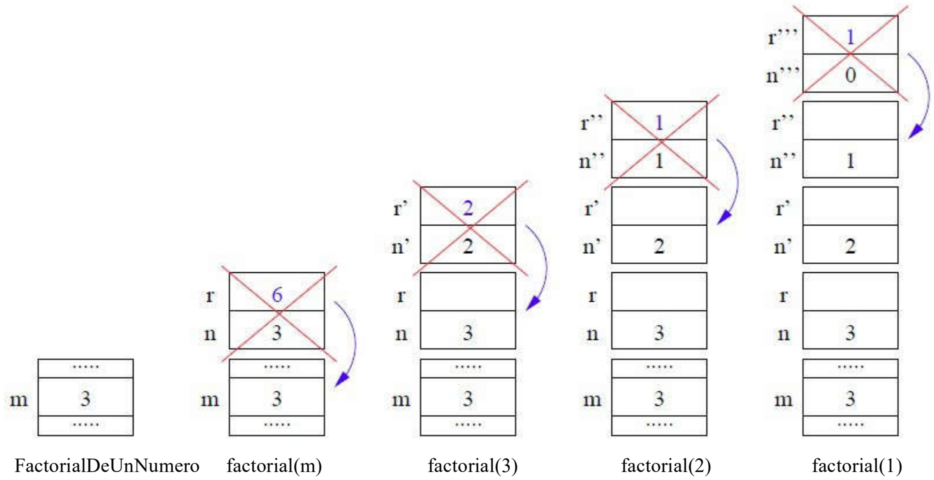
Supongamos la siguiente invocación: *factorial(3)*



Llamadas recursivas de *factorial(3)*

Recursividad

Ejemplos en Notación Algorítmica



Vuelta de las llamadas recursivas de *factorial(3)*

Recursividad

Ejemplos en Notación Algorítmica

<u>Función</u> potencia (<u>dato</u> a $\in$ N, e $\in$ N) $\rightarrow$ N

{Def: .... }

<u>Inicio</u>

....

<u>Potencia de un número natural y exponente natural</u>
$a^0 = 1$ si $a \neq 0$
$a^1 = a$
$a^{n+1} = a \cdot a^n$

<u>Ffunción</u>

Recursividad

Nociones Generales

Ejemplos:

<u>Serie de Fibonacci</u>: 1, 1, 2, 3, 5, 8, 13, 21, 34, …..

Fib(1) =
Fib(2) =
Fib(3) =
Fib(4) =
Fib(5) =

<u>Regla general</u>

Fib(1) =
Fib(2) =
Fib(n) =

https://youtu.be/0d4o57I3rn4

Recursividad

Ejemplos en Notación Algorítmica

<u>Función</u> fibonacci (<u>dato</u> n $\in$ N) $\rightarrow$ N

{Def: .... }

<u>Inicio</u>

....

<u>Ffunción</u>

Recursividad

Nociones Generales

Ejemplo: el algoritmo de Euclides para encontrar el máximo común divisor (mcd) de dos números consiste en mostrar que el mcd de m y n, (m > n > 0), es igual a m si n es cero, en otro caso es igual al mcd de n y el resto de m dividido por n, si n > 0.

Def: El mcd de dos enteros positivos es el entero mayor que divide a ambos.

	q1	q2	q3	…
M	N	r1	r2	…
r1	r2	r3	…	…

M	N
r1	q1
N	r1
r2	q2
r1	r2
r3	q3

<u>Regla general</u>

mcd(m,n) = m, si n=0

mcd(m,n) = mcd(n,resto de m divido por n)

<u>Ejemplos</u>

mcd(25,5) = mcd(5,0) = 5

mcd(18,6) = mcd(6,0) = 6

mcd(57,23) = mcd(23, 1) = mcd(1,0) = 1

mcd(35,16) = mcd(16,3) = mcd(3,1)=mcd(1, 0)=1

Recursividad

Ejemplos en Notación Algorítmica

<u>Función</u> mcd (<u>dato</u> m ∈ N, n ∈ N) → N

{Pre: .... }

{Def: .... }

<u>Inicio</u>
<u>según</u>

n=0: ← m

n≠0: ← mcd(n, m mod n)

<u>fsegún</u>
<u>Ffunción</u>

Recursividad

Ejemplos en Notación Algorítmica

<u>Función</u> cantPares (<u>dato</u> a ∈ arreglo [1..50] de N, inf ∈ N, sup ∈ N) → N

{Pre: *inf=1 y 1<=sup<=50 y **a** contiene datos entre 1 y sup*}

<u>Inicio</u>
<u>según</u>

sup<inf: ← 0

(a[sup] mod 2)=0: ← 1 + cantPares(a, inf, sup-1)

(a[sup] mod 2)<>0: ← 0 + cantPares(a, inf, sup-1)

<u>fsegún</u>
<u>Ffunción</u>

Recursividad

Ejemplos en Notación Algorítmica

<u>Función</u> cantPares (<u>dato</u> a ∈ arreglo [1..50] de N, inf ∈ N, sup ∈ N) → N

{Pre: *inf=1 y 1<=sup<=50 y **a** contiene datos entre 1 y sup*}

<u>Inicio</u>
<u>según</u>

Idem pero variando inf

<u>fsegún</u>
<u>Ffunción</u>

Recursividad

Nociones Generales

Ejemplos: Informar si una palabra, contenida dentro de un arreglo de caracteres, es capicúa

Definición de tipos y variables

TCadena = arreglo de [1..255] de Caracter

TPalabra = <palabra ∈ TCadena, longitud ∈ (1..255)>

pal ∈ TPalabra

Suponemos que las palabras tienen un carácter como mínimo y 255 como máximo

Regla general

capicúa(pal, inf, sup) = Verdadero, si inf=sup
capicúa(pal, inf, sup) = (pal[inf]=pal[sup]), si inf+1=sup
capicúa(pal, inf, sup) = (pal[inf]=pal[sup]) y capicúa(pal, inf+1, sup-1)

Invocación

capicúa(pal.palabra, 1, pal.longitud)

Recursividad

Ejemplos en Notación Algorítmica

Función capicúa (dato pal ∈ TCadena, **dato-res** inf, sup ∈ (1..255)) → **Logico**

Inicio

Ffunción

Recursividad

Ejemplos en Notación Algorítmica

Invocación a la función capicúa

```
Algoritmo MostrarPalindromo
Lexico
  TCadena = arreglo de [1..255] de Caracter
  TPalabra = <palabra ∈ TCadena, longitud ∈ (1..255)>
  pal ∈ TPalabra
  Acción cargarPalabra(dato-res p ∈ TPalabra)
  Función capicúa(dato p ∈ TCadena, dato-res inf, sup ∈ [1..255]) → Logico
Inicio
 cargarPalabra(pal)
 según
   capicúa(pal.palabra, 1, pal.longitud): Salida:'es capi`
   ¬capicúa(pal.palabra, 1, pal.longitud): Salida:`no es capi`
 fsegún
Fin
```


Recursión de cola

Una función presenta **Recursión de Cola** cuando no hay ningún cómputo después de la llamada recursiva, es decir, son funciones que finalizan con una llamada recursiva que no crea ninguna operación diferida.

- mcd **es recursiva de cola**
- factorial **no es recursiva de cola** (recursión en aumento)

Función mcd (dato m ∈ N, n ∈ N) → N	**Función factorial (dato n ∈ N) → N**
Inicio	**Inicio**
según	**según**
n=0: ← m	n=0: ← 1
n≠0: ← mcd(n, m mod n)	n>0: ← n * factorial(n-1)
fsegún	**fsegún**
Ffunción	**Ffunción**

Recursión de cola

Hacer factorial recursiva de cola

```
Función factorial (dato n ∈ N) → N
Inicio
según
   n=0: ← 1
   n>0: ← n * factorial(n-1)
fsegún
Ffunción
```

```
Función factorial
Inicio

Ffunción
```


Recursión de cola

- Factorial crea operaciones diferidas que tienen que realizarse incluso después de que se complete la última llamada recursiva.

- Una función recursiva de cola se ejecuta usando un espacio constante. Así, el proceso que genera es esencialmente iterativo y equivalente a usar estructuras de control de lenguaje imperativo como los mientras o para.

Recursividad

Hacer cantPares recursiva de cola

Función cantPares

{Pre: *inf=1 y 1<=sup<=50 y* ***a*** *contiene datos entre 1 y sup*}

Inicio

Ffunción

Recursividad

Acciones recursivas

- Se sigue la misma idea que la funciones, solo que que la recursión es siempre **recursiva de cola**.
- La técnica es utilizar un parámetro ***dato-resultado*** donde ir almacenando los valores parciales y el final.

```
Acción factorial (dato n ∈ N, dato-res fact ∈ N)
{Pre: fact=1 }
Inicio
según
   n=0: fact ← 1                // o nada
   n=1: fact ← fact * n         // o nada
   n>1: factorial(n-1, fact*n)
fsegún
Facción
```

Recursividad

Funciones recursivas versus Acciones recursivas

• **Toda función** recursiva se puede traducir a una **acción** recursiva. Para ello plantear la solución como una recursión de cola.

• Pero **no toda acción** recursiva puede traducirse a una **función.** Por ejemplo no es posible cuando la acción modifica el entorno (el valor de alguna de las variables empleadas en la acción que no son específicamente del cálculo recursivo).

• Pensemos algunos ejemplos...

Recursividad

Hacer fibonacci, mcd y cantPares con usando acción

Recursividad: conceptos

Profundidad

Es la cantidad de veces que una abstracción (función o acción) es evaluada (invocada o llamada) para un conjunto de argumentos dados.

En el ejemplo de: *factorial(2)* , n=2, la función es evaluada 3 veces, es decir que la profundidad es 3.

La profundidad debe ser *razonable* y *finita.*

- **Razonable**: que los argumentos no determinen una profundidad tal que la recursión se hace muy lenta y tome mucho tiempo en terminar.
- **Finita**: que nos lleve hacia la terminación del algoritmo.

Recursividad: conceptos

Profundidad

Analizar la profundidad del siguiente ejemplo para m=2 y n=3

```
Función foo (dato n, m ∈ Z) → Z
Inicio
  según
    m=0:   ← n + 1
    m<>0:  si n=0
           entonces      ← foo (m-1,1)
           sino          ← foo (m-1, foo(m,n-1))
  fsegún
Ffunción
```

Recursividad: conceptos
Directa e Indirecta

Lo que hemos visto hasta ahora es la denominada *recursividad directa*, ya que una abstracción se llama así misma.

Existe otro tipo de recursión llamada *recursión indirecta* cuando una abstracción *A* llama a otra *B* y ésta llama a *A*.

Recursividad: conceptos
Directa e Indirecta - Ejemplo

Función a (x ∈ Lógico) → Lógico
Inicio
  si b(x)
  entonces ← false
  sino ← true
Ffunción

Función b (x ∈ Lógico) → Lógico
Inicio
  si x
  entonces ← false
  sino ← a(x)
Ffunción

Recursividad
Datos Recursivos

• Una aplicación de importancia de soluciones recursivas se da cuando los datos a tratar admiten o tienen una definición recursiva. En dicho caso el procesamiento se presta fácilmente a una solución recursiva.
• Este tipo de soluciones utilizan lo que se denomina **recursión estructural**.
• El caso de las **estructuras simple o doblemente encadenadas** es un ejemplo de datos que pueden definirse recursivamente y en consecuencia ser tratados de la misma manera.

Lista implementada con estructuras dinámicas
TLista = puntero a TElem
TElem = <info ∈ *m*, next ∈ puntero a TElem>

Recursividad
Datos Recursivos - Ejemplos

Mostrar una estructura simplemente encadenada

Acción print (**dato** lis ∈ TLista)
Inicio
  según
    lis=nil: nada
    lis≠nil: Salida:(^lis).info
        print((^lis).next)
  fsegún
Facción

Recursividad
Datos Recursivos - Ejemplos

Mostrar a la inversa una estructura encadenada

Acción printReversa (**dato** lis $\in$ TLista)
Inicio
  según
    lis=nil: nada
    lis$\neq$nil: printReversa((^lis).next)
      Salida:(^lis).info
  fsegún
Facción

Recursividad
Datos Recursivos - Ejemplos

¿Porqué muestra la lista al revés?

Porque en el primer caso (**print)** primero se ejecuta el *escribir* y luego la invocación recursiva, mientras que en el segundo caso (**printReversa**) primero se ejecuta la invocación recursiva y hasta que esta no termine (es decir que no se llegue al fin de la lista) no se ejecuta el *escribir*, que al ejecutarse por primera vez toma el último elemento de la lista y es allí donde termina la ejecución de la última invocación y al retornar toma el penúltimo elemento y así sucesivamente hasta el primero.
En este caso decimos que el *caso base* es el *nil*. Es así, entonces que las acciones *print* y *printReversa* tienen a *nil* como valor conocido para el puntero y es donde termina.

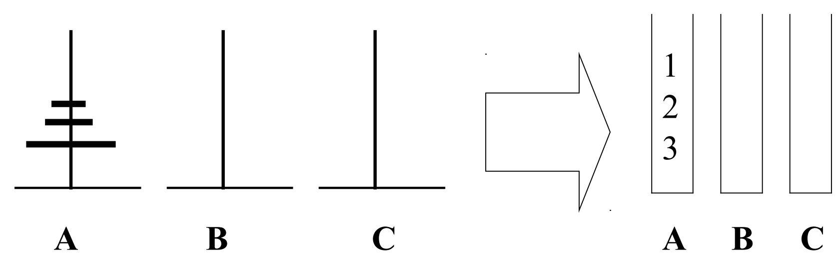
Recursividad
Ejemplos en Notación Algorítmica

Torres de Hanoi

El juego consiste en pasar una torre de objetos de un lugar a otro, respetando las siguientes reglas: (a) sólo puede emplearse un lugar auxiliar, (además del original de partida y el del destino); (b) sólo puede moverse un objeto por vez; (c) no puede nunca apilarse un objeto encima de otro más pequeño. Generalmente se representa tal como el juego se vende: los objetos son discos con un agujero en el medio y tres clavijas montadas cada una sobre una base, que representan las torres (la de partida u origen, la de llegada o destino y la auxiliar), donde hay tres discos en la clavija de origen. Nosotros podemos representarlo como tres torres de números naturales.

Recursividad
Ejemplos en Notación Algorítmica

Torres de Hanoi



Recursividad

Ejemplos en Notación Algorítmica - Torres de Hanoi

Ejemplo con 2 discos (n=2)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
inicio	**1** **2**			
1	**2**	**1**		**A → B**
2		**1**	**2**	**A → C**
3			**1** **2**	**B → C**

Recursividad

Torres de Hanoi - Ejemplo con 3 discos (n=3)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
Inicio	1 2 3			
1	2 3		1	A → C
2	3	2	1	A → B
3	3	1 2		C → B
4		1 2	3	A → C
5		2	1 3	B → C
6	2		1 3	B → A
7	1 2		3	C → A
Estamos en el caso de mover 2 discos (n-1)				

Recursividad

Torres de Hanoi - Ejemplo con 3 discos (n=3)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
Estamos en el caso de mover 2 discos (n-1)				
8	2	1	3	A → B
9		1	2 3	A → C
10			1 2 3	B → C

Recursividad

Torres de Hanoi - Ejemplo con 3 discos (n=3)

La solución consiste en mover primero todos los elementos, menos uno, de la torre original **A** (**orig**) a la auxiliar **B** (**aux**), cualquiera sea la cantidad de elementos. Luego, cuando queda un solo elemento, se mueve a la torre de destino **C** (**dst**). Por último se mueven todos los elementos de la torre auxiliar a la de destino.

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

Completar los pasos y movimientos

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
Inicio	1 2 3 4			
	2 3 4		1	
	3 4	2	1	
	3 4	1 2		
	4	1 2	3	
	4	2	1 3	

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
	2 4		1 3	
	1 2 4		3	
	1 2 4	3		
	2 4	3	1	
	4	2 3	1	
	4	1 2 3		

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
		1 2 3	4	
	1	2 3	4	
	1	3	2 4	
		3	1 2 4	
	3		1 2 4	
	3	1	2 4	

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
	2 3	1	4	
	1 2 3		4	
Estamos en el caso de mover 3 discos (n-1)				
	1 2 3		4	
	2 3		1 4	
	3	2	1 4	

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
	3	1 2	4	
		1 2	3 4	
		2	1 3 4	
	2		1 3 4	
	1 2		3 4	
Estamos en el caso de mover 2 discos (n-1)				

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

Paso	A (origen)	B (auxiliar)	C (destino)	Mov.
Estamos en el caso de mover 2 discos (n-1)				
	2	1	3 4	
		1	2 3 4	
			1 2 3 4	

Recursividad

Torres de Hanoi - Ejemplo con 4 discos (n=4)

La solución consiste en mover primero todos los elementos, menos uno, de la torre original **A** (**orig**) a la auxiliar **B** (**aux**), cualquiera sea la cantidad de elementos. Luego, cuando queda un solo elemento, se mueve a la torre de destino **C** (**dst**). Por último se mueven todos los elementos de la torre auxiliar a la de destino.

Es decir, igual que para 3 discos, n=3.

Recursividad

Torres de Hanoi - Ejemplo con 3 discos (n=3)

La solución consiste en mover primero todos los elementos, menos uno, de la torre original **A** (**orig**) a la auxiliar **B** (**aux**), cualquiera sea la cantidad de elementos. Luego, cuando queda un solo elemento, se mueve a la torre de destino **C** (**dst**). Por último se mueven todos los elementos de la torre auxiliar a la de destino.

Torres de Hanoi - Ejemplo con 4 discos (n=4)

La solución consiste en mover primero todos los elementos, menos uno, de la torre original **A** (**orig**) a la auxiliar **B** (**aux**), cualquiera sea la cantidad de elementos. Luego, cuando queda un solo elemento, se mueve a la torre de destino **C** (**dst**). Por último se mueven todos los elementos de la torre auxiliar a la de destino.

Recursividad

Torres de Hanoi – Solución Algorítmica

Explicación

- Mover primero todos los elementos, menos uno, de la torre original (orig) a la auxiliar (aux), cualquiera sea la cantidad de elementos:

  hanoi (n – 1, orig, dst, aux)

- Luego, cuando queda un solo elemento, se mueve a la torre original (orig) a la de destino (dst):

  hanoi (1, orig, aux, dst)

- Por último se mueven todos los elementos de la torre auxiliar (aux) a la de destino (dst):

  hanoi (n – 1, aux, orig, dst)

Recursividad

Torres de Hanoi – Solución Algorítmica

```
Acción hanoi (dato n ∈ Z, dato-res orig, aux, dst ∈ THanoi)
Lexico
  c ∈ Z
Inicio
  si (n = 1)
  entonces
    mover el disco que está en
    tope de la torre orig a la
    torre dst
  sino {n <> 1}
    hanoi (n – 1, orig, dst, aux)
    hanoi (1, orig, aux, dst)
    hanoi (n – 1, aux, orig, dst)
  fsi
Facción
```

Donde THanoi es un tipo que representa las torres

La solución consiste en mover primero todos los elementos, menos uno, de la torre original **A** (**orig**) a la auxiliar **B** (**aux**), cualquiera sea la cantidad de elementos. Luego, cuando queda un solo elemento, se mueve a la torre de destino **C** (**dst**). Por último se mueven todos los elementos de la torre auxiliar a la de destino.

Recursividad

Ejercicios

- Dada una lista simplemente encadenada del siguiente tipo:

```
TElemento = <nro ∈ Z, sig ∈ puntero a TElemento>
```

escribir las siguientes funciones o acciones recursivas:
a) **longElem**, función que dado un puntero al primer elemento de la lista retorne la longitud de la misma.
b) **masUlt**, acción que dado un puntero al primer elemento de la lista modifica la lista original sumándole a cada elemento el último número de la lista.
c) **com**, función que dado un puntero al primer elemento de la lista retorne la lista original sin el último elemento.
d) **fin**, función que dado un puntero al primer elemento de la lista retorne la lista original sin el primer elemento.

- Dado un arreglo de 12 caracteres, que contiene una palabra, escribir una acción o función recursiva que determine si el la palabra contenida en el arreglo posee alguna vocal.

Bibliografía

- Bird, R. J.; Wadler, P.; Introduction to Functional Programming; Prentice Hall 1988.
- Peyton Jones, S.L.; The Implementation of Functional Programming Languages; Prentice Hall 1987.
- MacLennan, B.; Functional Programming: practice and theory; Addison-Wesley 1990.
- Allen, J.; Anatomy of LISP; McGraw-Hill 1978.
- Pratt, T.; Programming Languages: Design and implementation; Prentice Hall 1984.
- Watt, D.;Programming Language Concepts and Paradigms; Prentice-Hall International Series in Computer Science 1990.
- Tasistro, A.; Vidart, J.; Programación Lógica y Funcional; EBAI 1988.
- Hernández-Novich, E. Lenguajes de Programación I: Iteradores y recursión. http://ldc.usb.ve/~emhn/cursos/ci3641/200909/Clases/07/clase07.pdf 2006.
- Mendez, G. Diseño de algoritmos recursivos. http://www.fdi.ucm.es/profesor/rgonzale/TRecursivos.pdf 2012.