Recursión

Capítulo 4

Introducción

- La recursión o recursividad es un concepto amplio, con muchas variantes, y difícil de precisar con pocas palabras. .

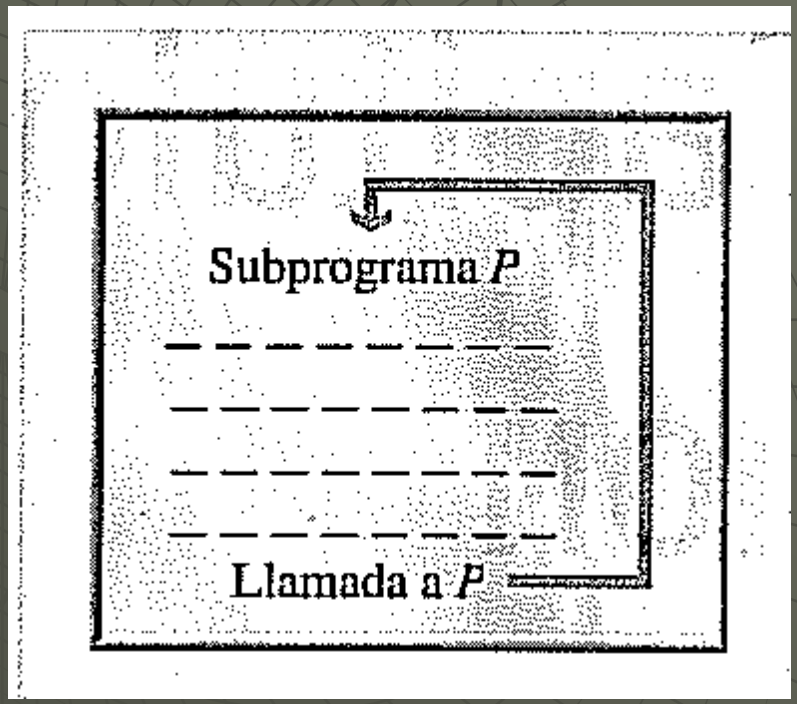
Actividades Cotidianas;

- fotografía donde se observa otra fotografía
- Reporteros (3 en distintos países)

- La recursión es un recurso muy poderoso que permite expresar soluciones simples y naturales a ciertos tipos de problemas. Es importante considerar que no todos los problemas son recursivos.
- Un objeto recursivo es el que aparece en la definición de sí mismo, así como el que se llama a sí mismo. Los árboles (cap. 6) son ED, no lineales y dinámicas, más eficientes que existen en computación. La característica de los árboles es que son estructuras inherentemente recursivas. En otras palabras cualquier actividad de programación que se realice con árboles se utiliza recursividad.

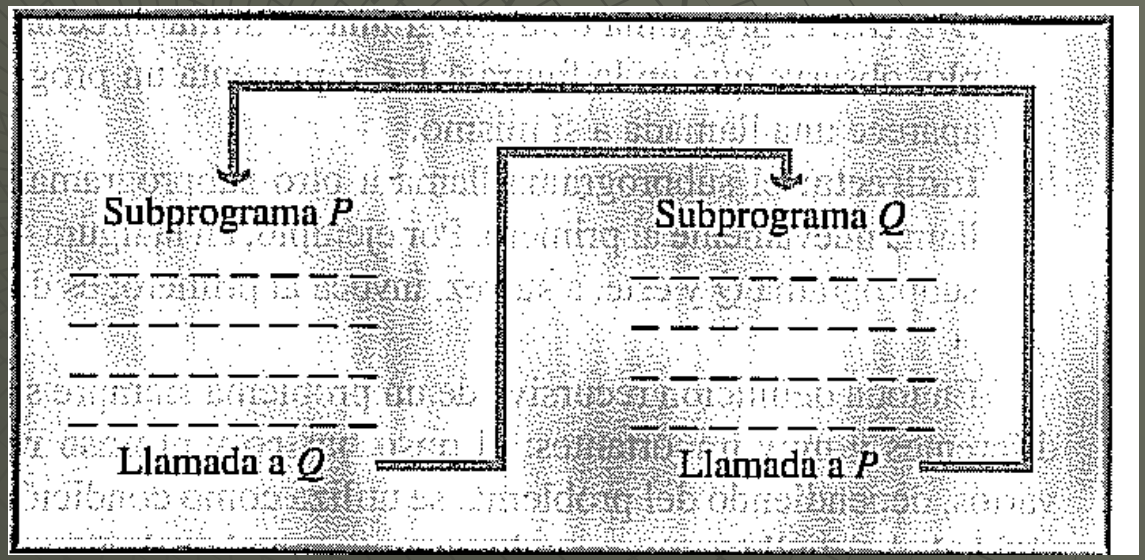
La recursión puede presentarse de 2 maneras;

- Directa: el programa o subprograma se llama directamente a sí mismo.



La recursión puede presentarse de 2 maneras;

- Indirecta: el subprograma llama a otro subprograma, y éste, en algún momento, llama nuevamente al primero.



- En toda definición recursiva de un problema siempre se deben establecer dos pasos diferentes y muy importantes; el paso básico y el paso recursivo. El primero, uno o varios, dependiendo del problema, se utiliza como condición de parada o fin de la recursividad. A éste llegamos cuando encontramos la solución del problema o cuando decidimos que ya no vamos a seguir, por que no están dadas las condiciones para hacerlo.

- El segundo paso, por otra parte propicia recursividad. Se pueden presentar uno o varios, nuevamente dependiendo del problema a resolver.
- Cuando se analiza la solución recursiva de un problema es importante determinar con precisión cuáles serán los pasos básico y recursivo. En cada vuelta del ciclo es importante que nos acerquemos cada vez más ala solución del problema en cuestión, o sea al paso básico. Si esto no ocurre podemos estar en un ciclo extraño. Puede que el problema este mal definido y entonces la maquina se quedaría ejecutando por tiempo indefinido el programa y sólo terminaría al agotarse la memoria.

Ejemplos de recursión.

- Factorial de un número
- n!
- Por definición
- 0! = 1 Paso básico
- 1! = 1 Paso básico
- 2!
- 3!
- 4!
- n!= n*(n-1)!

Factorial

Si n = 0 o n = 1

$$n! \begin{cases} \\ n*(n-1)! & \text{Si } n > 1 \end{cases}$$

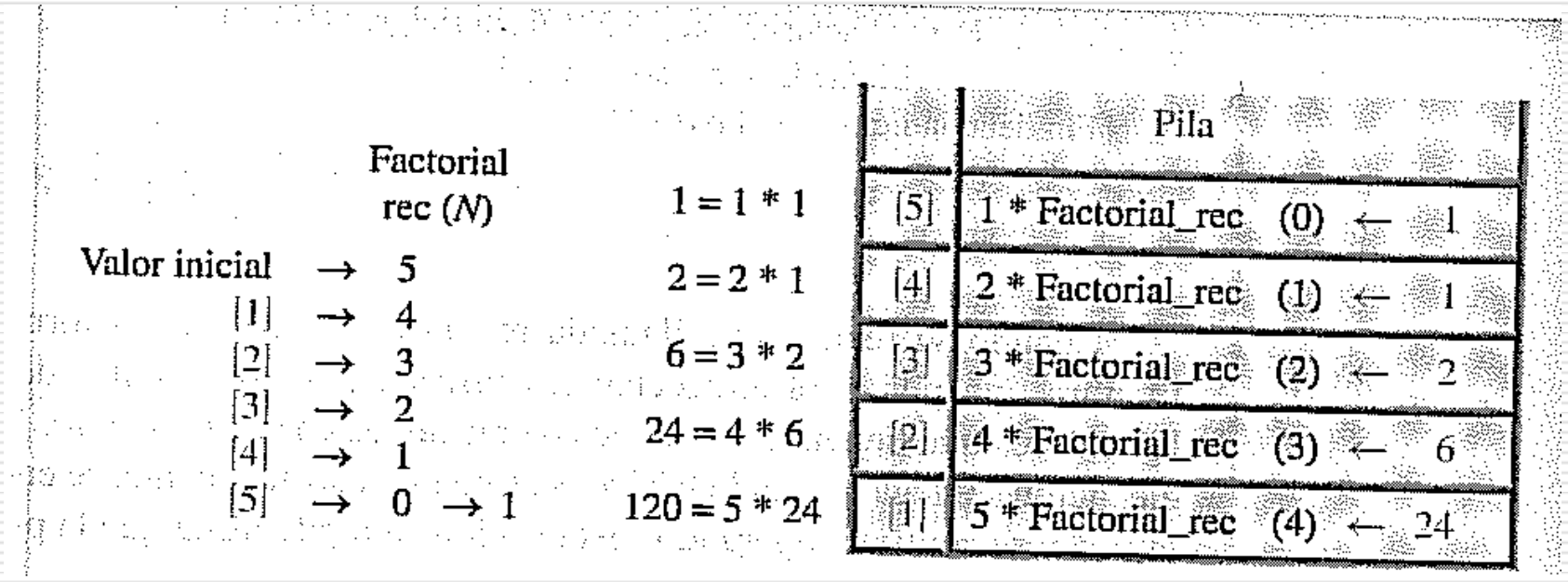
Algoritmo Factorial recursivo

Factorial_rec (N)

{Este algoritmo calcula el factorial de un número N en forma recursiva, donde N es un valor numérico entero, positivo o nulo}

1. *Si* ($N = 0$)
   entonces
      Hacer Factorial_rec ← 1 {Paso básico}
   si no
      Hacer Factorial_rec ← N * Factorial_rec ($N - 1$)
         {Llamada —paso— recursiva}
2. {Fin del condicional del paso 1}

Ejemplo


Factorial
rec (N)
Valor inicial → 5
[1] → 4
[2] → 3
[3] → 2
[4] → 1
[5] → 0 → 1
1 = 1 * 1
2 = 2 * 1
6 = 3 * 2
24 = 4 * 6
120 = 5 * 24
Pila
[5] 1 * Factorial_rec (0) ← 1
[4] 2 * Factorial_rec (1) ← 1
[3] 3 * Factorial_rec (2) ← 2
[2] 4 * Factorial_rec (3) ← 6
[1] 5 * Factorial_rec (4) ← 24

Factorial recursivo

- Se observa que en la pila el número que se encuentra entre corchetes en la primera columna de la izquierda hace referencia a la llamada recursiva. Este símbolo permite observar el orden en que se realizan las llamadas recursivas.
- A continuación se presenta una variante iterativa del cálculo factorial. Este es útil por que algunos lenguajes de programación tienen un espacio muy reducido dedicado a la pila. En tal caso, una forma para remediar este inconveniente es utilizando iteratividad en lugar de recursividad.

Algoritmo Factorial Iterativo

Factorial_ite (*N*)

{Este algoritmo calcula el factorial de un número *N* en forma iterativa, donde *N* es un valor numérico entero, positivo o nulo}

{FACT es una variable de tipo entero}

1. Hacer FACT ← 1
2. Mientras (*N* > 0) *Repetir* {Ciclo para calcular *N*!}
   Hacer FACT ← *N* * FACT y *N* ← *N* – 1
3. {Fin del ciclo del paso 2}
4. Escribir FACT

N	*FACT*
4	1
3	4
2	12
1	24
0	24

EJEMPLO

N	*FACT*
4	1
3	4
2	12
1	24
0	24

Sucesión de Fibonacci

- Otro problema clásico de recursividad es la sucesión de Leonardo de Pisa.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 etc..
- Propiedades (Wikipedia)
- Cualquier número natural se puede escribir mediante la suma de números de Fibonacci, c/u de ellos distinto a los demás.
- Ejemplo 17 = 13 + 3 + 1 y 65 = 55 + 8 + 2
- Tan solo un término de cada 3 es par, uno de cada cuatro es múltiplo de 3 y uno de cada 5 es múltiplo de 5. Esto se puede generalizar de forma que la sucesión de Fibonacci es periódica en la congruencias módulo m para cualquier m
- Si Fibonacci de p, F(p), es un número primo entonces p también lo es. Con una única excepción.
- F(4)= 3; donde 3 es primo y 4 no lo es.

El Fibonacci de una número se obtiene de la suma de los dos número de Fibonacci anteriores.

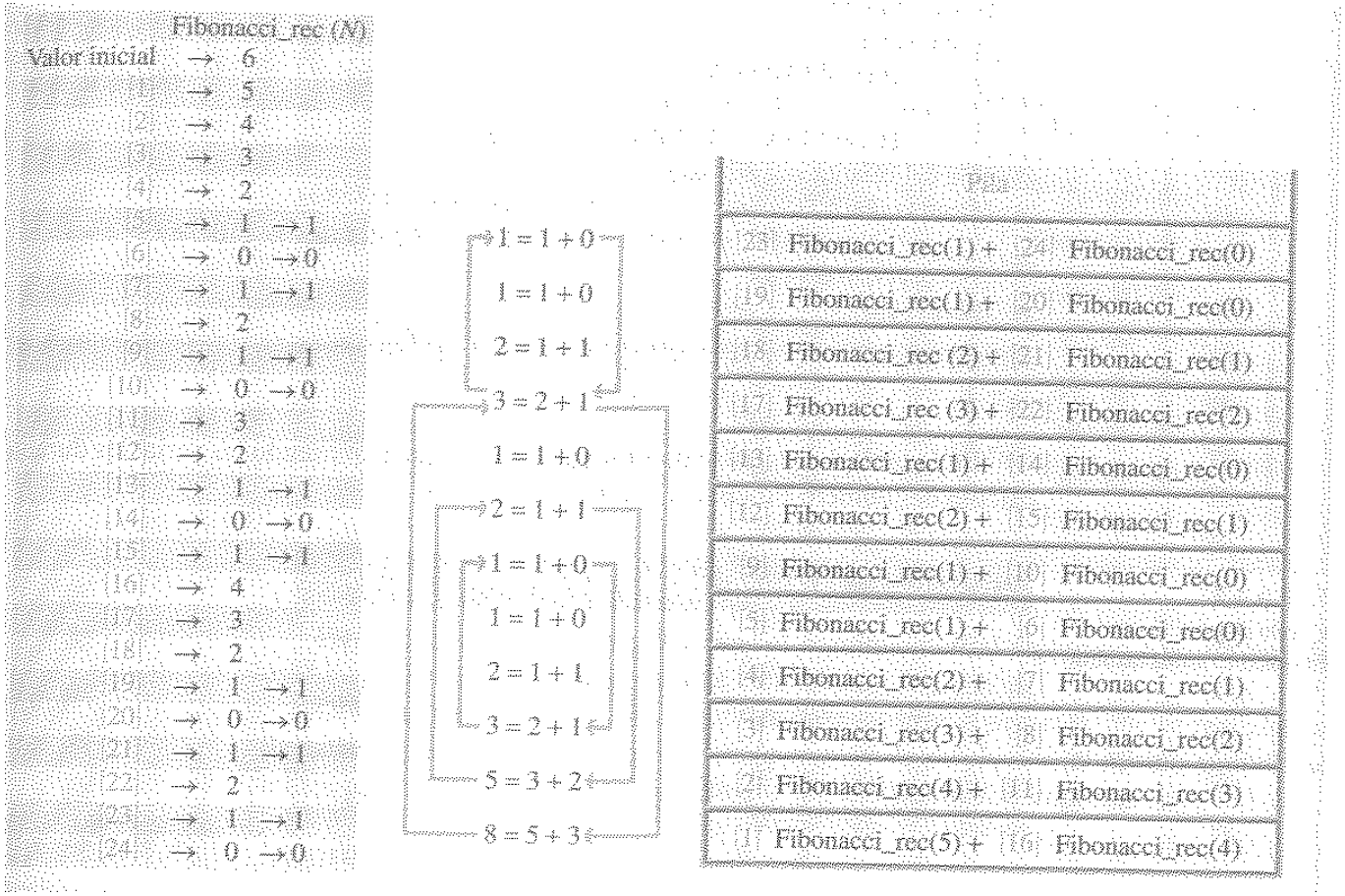
$$
\begin{aligned}
&\text{Fibonacci}(0) = 0 && \rightarrow \textbf{Paso básico}\\
&\text{Fibonacci}(1) = 1 && \rightarrow \textbf{Paso básico}\\
&\text{Fibonacci}(2) = \text{Fibonacci}(1) + \text{Fibonacci}(0)\\
&\qquad = 1 + 0 = 1\\
&\text{Fibonacci}(3) = \text{Fibonacci}(2) + \text{Fibonacci}(1)\\
&\qquad = 1 + 1 = 2\\
&\text{Fibonacci}(4) = \text{Fibonacci}(3) + \text{Fibonacci}(2)\\
&\qquad = 2 + 1 = 3\\
&\ldots\\
&\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}\,(n-2) && \rightarrow \textbf{Paso recursivo}
\end{aligned}
$$

Representación

- n si n=0 o n=1
- Fibonacci {
- Fibonacci(n-1)+fibonacci(n-2) si n>1

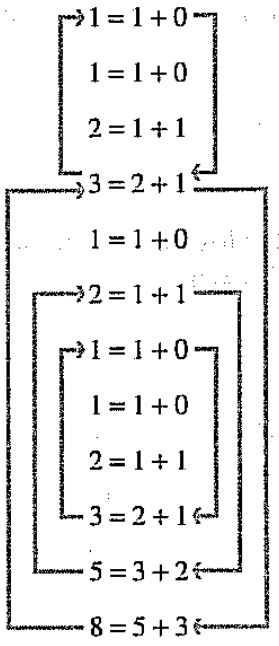
Algoritmo Fibonacci recursivo

- Fibonacci_rec
- N es un número entero positivo
- 1. Si ((N=0) o (N=1))
- Entonces
- Hacer Fibonacci_rec←N (paso básico)
- Si no
- Hacer Fibonacci_rec← Fibonacci_rec(N-1) + Fibonacci_rec(N-2)
- {llamadas recursivas}
- 2. {fin del paso 1}


Fibonacci_rec (N)
Valor inicial → 6
→ 5
→ 4
→ 3
→ 2
→ 1 → 1
→ 0 → 0
→ 1 → 1
→ 2
→ 1 → 1
→ 0 → 0
→ 3
→ 2
→ 1 → 1
→ 0 → 0
→ 1 → 1
→ 4
→ 3
→ 2
→ 1 → 1
→ 0 → 0
→ 1 → 1
→ 2
→ 1 → 1
→ 0 → 0
1 = 1 + 0
1 = 1 + 0
2 = 1 + 1
3 = 2 + 1
1 = 1 + 0
2 = 1 + 1
1 = 1 + 0
1 = 1 + 0
2 = 1 + 1
3 = 2 + 1
5 = 3 + 2
8 = 5 + 3
Fibonacci_rec(1) + Fibonacci_rec(0)
Fibonacci_rec(1) + Fibonacci_rec(0)
Fibonacci_rec (2) + Fibonacci_rec(1)
Fibonacci_rec (3) + Fibonacci_rec(2)
Fibonacci_rec(1) + Fibonacci_rec(0)
Fibonacci_rec(2) + Fibonacci_rec(1)
Fibonacci_rec(1) + Fibonacci_rec(0)
Fibonacci_rec(1) + Fibonacci_rec(0)
Fibonacci_rec(2) + Fibonacci_rec(1)
Fibonacci_rec(3) + Fibonacci_rec(2)
Fibonacci_rec(4) + Fibonacci_rec(3)
Fibonacci_rec(5) + Fibonacci_rec(4)

Esta solución resulta totalmente impractica e ineficiente. A continuación se presenta una variante iterativa para resolver este problema.

	Fibonacci_rec (*N*)		
Valor inicial	→	6	
[1]	→	5	
[2]	→	4	
[3]	→	3	
[4]	→	2	
[5]	→	1	→1
[6]	→	0	→0
[7]	→	1	→1
[8]	→	2	
[9]	→	1	→1
[10]	→	0	→0
[11]	→	3	
[12]	→	2	
[13]	→	1	→1
[14]	→	0	→0
[15]	→	1	→1
[16]	→	4	
[17]	→	3	
[18]	→	2	
[19]	→	1	→1
[20]	→	0	→0
[21]	→	1	→1
[22]	→	2	
[23]	→	1	→1
[24]	→	0	→0



Pila
[23] Fibonacci_rec(1) + [24] Fibonacci_rec(0)
[19] Fibonacci_rec(1) + [20] Fibonacci_rec(0)
[18] Fibonacci_rec (2) + [21] Fibonacci_rec(1)
[17] Fibonacci_rec (3) + [22] Fibonacci_rec(2)
[13] Fibonacci_rec(1) + [14] Fibonacci_rec(0)
[12] Fibonacci_rec(2) + [15] Fibonacci_rec(1)
[9] Fibonacci_rec(1) + [10] Fibonacci_rec(0)
[5] Fibonacci_rec(1) + [6] Fibonacci_rec(0)
[4] Fibonacci_rec(2) + [7] Fibonacci_rec(1)
[3] Fibonacci_rec(3) + [8] Fibonacci_rec(2)
[2] Fibonacci_rec(4) + [11] Fibonacci_rec(3)
[1] Fibonacci_rec(5) + [16] Fibonacci_rec(4)

Algoritmo Fibonacci iterativo

- 1. N es un número natural
- FIBO, FIBA, FIBB e I son variables de tipo entero
- Si ((N=0) o (N=1))
- Entonces
- Hacer FIBO← 0, FIBB←1 e I ←2
- 2. {fin del paso 1}
- 3. Mientras (I ≤N) Repetir
  - Hacer FIBO← FIBB+FIBA, FIBA←FIBB, FIBB←FIBO e I← I+1
  - 4. {fin del paso 3}
- 5. Escribir FIBO

N	*FIBO*	*FIBA*	*FIBB*	*I*
6		0	1	2
	1	1	1	3
	2	1	2	4
	3	2	3	5
	5	3	5	6
	8	5	8	7

Aplicación del Algoritmo

N	*FIBO*	*FIBA*	*FIBB*	*I*
6		0	1	2
	1	1	1	3
	2	1	2	4
	3	2	3	5
	5	3	5	6
	8	5	8	7

Imprimir un Arreglo

- Dado como dato un arreglo unidimensional de tipo entero, escriba el contenido de las casillas del mismo de **izquierda a derecha**

Algoritmo Arreglo_impresión

Arreglo_imp(A, N)
{A tipo entero y N tamaño del arreglo}
1. Si (N≠0) entonces
   Arreglo_imp(A, N-1)
   Escribir A[N]
2. {fin 1}

Se A un arreglo unidimensional

8	12	21	27	32	56	78

Arreglo_imp(A,N)	*Impresión de resultados*
A7	8
A6	12
A5	21
A4	27
A3	32
A2	55
A1	78
A0	

PILA	
ESCRIBIR A[1]	E[1]
ESCRIBIR A[2]	E[2]
ESCRIBIR A[3]	E[3]
ESCRIBIR A[4]	E[4]
ESCRIBIR A[5]	E[5]
ESCRIBIR A[5]	E[6]
ESCRIBIR A[7]	E[7]

```
Arreglo_imp(A, N)
{A tipo entero y N tamaño del arreglo}
1. Si (N≠0) entonces
   Arreglo_imp(A, N-1)
   Escribir A[N]
2. {fin 1}
```

Impresión de un Arreglo Derecha a izquierda

```
Arreglo_imp(A, N)
{A tipo entero y N tamaño del arreglo}
1. Si (N≠0) entonces
    Arreglo_imp(A, N-1)
    Escribir A[N]
2. {fin 1}
```

Suma de un arreglo

8	12	21	27	31	44	48

- Dado como dato un A U de tipo entero, obtenga la suma del mismo
- Algoritmo Arreglo_sum
- N tamaño del arreglo
- 1. Si (N=0) entonces
- Hacer Arreglo_sum -- 0
- Sino
- Hacer Arreglo_sum – A[N] + Arreglo_sum(A, N-1)
- 2. fin1.

Arreglo_sum	(A,N)
	A7
[1]	A6
[2]	A5
[3]	A4
[4]	A3
[5]	A2
[6]	A1
[7]	A0—0

8	12	21	27	31	44	48

Arreglo_sum	(A,N)
	A7
[1]	A6
[2]	A5
[3]	A4
[4]	A3
[5]	A2
[6]	A1
[7]	A0—0

8 + 0= 8

12 + 8 = 20

21 + 20 = 41

27 + 41 = 68

31 + 68 = 99

44 + 99= 143

48 + 143 = 191

Pila	
A[1] + Arreglo_sum (A,0)	[7]
A[2] + Arreglo_sum (A,1)	[6]
A[3] + Arreglo_sum (A,2)	[5]
A[4] + Arreglo_sum (A,3)	[4]
A[5] + Arreglo_sum (A,4)	[3]
A[6] + Arreglo_sum (A,5)	[2]
A[7] + Arreglo_sum (A,6)	[1]

Euclides

- Es un método efectivo para calcular el mcd entre dos números enteros positivos.
- En la primera división se toma el mayor como dividendo y el otro como divisor. Luego el divisor y el resto sirven como dividendo y divisor respectivamente.

- EUCLIDES (M,N) {

m Si N=0

Euclides (N, M MOD N) en cualquier otro caso

Algoritmo

Euclides (M,N)

- 1. Si N=0
- Entonces
- Hacer Euclides – M
- Sino
- Hacer Euclides – Euclides(N, M MOD N)
- 2. fin 1

Ejemplo

Euclides	(M, N)
2353	1651

Ejemplo

Euclides	(M, N)
2353	1651
1651	702
702	247
247	208
208	39
39	13
13	1. ---13

Ackerman

- Utilizada en la teoría de la computación, es una función recursiva que toma dos números naturales como argumentos y devuelve un número natural.

$$A(m,n)=\begin{cases} n+1, & si\ m=0; \\ A(m-1,1), & si\ m>0 \text{ y } n=0; \\ A\big(m-1,A(m,n-1)\big), & si\ m>0 \text{ y } n>0 \end{cases}$$

Ackerman

$$A(m,n)=\begin{cases} n+1, & si\ m=0; \\ A(m-1,1), & si\ m>0\ \text{y}\ n=0; \\ A\big(m-1,A(m,n-1)\big), & si\ m>0\ \text{y}\ n>0 \end{cases}$$

M	N	Ackermann	Profundidad
1	0	2	2
2	0	3	5
3	0	5	15
2	1	5	14

Ackerman Algoritmo

Ackermann (*M*, *N*)

{Este algoritmo calcula la función de Ackermann. *M* y *N* son valores numéricos ente positivos o nulos}

1. *Si* ($M = 0$)
   entonces
   Hacer Ackermann ← ($N + 1$) {Estado básico}
   si no
   1.1 *Si* ($N = 0$)
   entonces
   Hacer Ackermann ← Ackermann ($M - 1$, 1)
   si no
   Hacer Ackermann ← Ackermann ($M - 1$, Ackermann (M, $N - 1$))
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

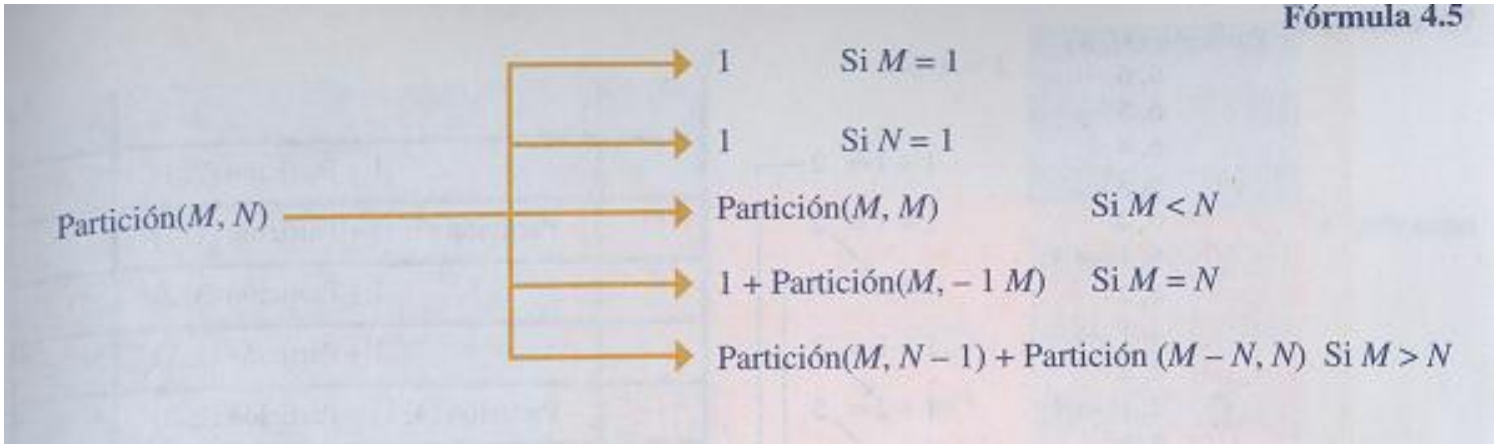
Ackermann (M, N)		
alor inicial	→ (2, 2)	
[1]	→ (2, 1)	
[2]	→ (2, 0)	
e de N = 0	→ (1, 1)	
[3]	→ (1, 0)	
e de N = 0	→ (0, 1)	→ 2
[4]	→ (0, 2)	→ 3
[5]	→ (1, 3)	
[6]	→ (1, 2)	
[7]	→ (1, 1)	
[8]	→ (1, 0)	
e de N = 0	→ (0, 1)	→ 2
[9]	→ (0, 2)	→ 3
[10]	→ (0, 3)	→ 4
[11]	→ (0, 4)	→ 5
[12]	→ (1, 5)	
[13]	→ (1, 4)	
[14]	→ (1, 3)	
[15]	→ (1, 2)	
[16]	→ (1, 1)	
[17]	→ (1, 0)	
de N = 0	→ (0, 1)	→ 2
[18]	→ (0, 2)	→ 3
[19]	→ (0, 3)	→ 4
[20]	→ (0, 4)	→ 5
[21]	→ (0, 5)	→ 6
[22]	→ (0, 6)	→ 7

	Pila	
2	[18]	Ackermann(0, Ackermann(1, 0))	[17]
3	[19]	Ackermann(0, Ackermann(1, 1))	[16]
4	[20]	Ackermann(0, Ackermann(1, 2))	[15]
5	[21]	Ackermann(0, Ackermann(1, 3))	[14]
7 ← 6	[22]	Ackermann(0, Ackermann(1, 4))	[13]
2	[9]	Ackermann(0, Ackermann(1, 0))	[8]
3	[10]	Ackermann(0, Ackermann(1, 1))	[7]
4	[11]	Ackermann(0, Ackermann(1, 2))	[6]
2	[4]	Ackermann(0, Ackermann(1, 0))	[3]
3	[5]	Ackermann(1, Ackermann(2, 0))	[2]
5	[12]	Ackermann(1, Ackermann(2, 0))	[1]

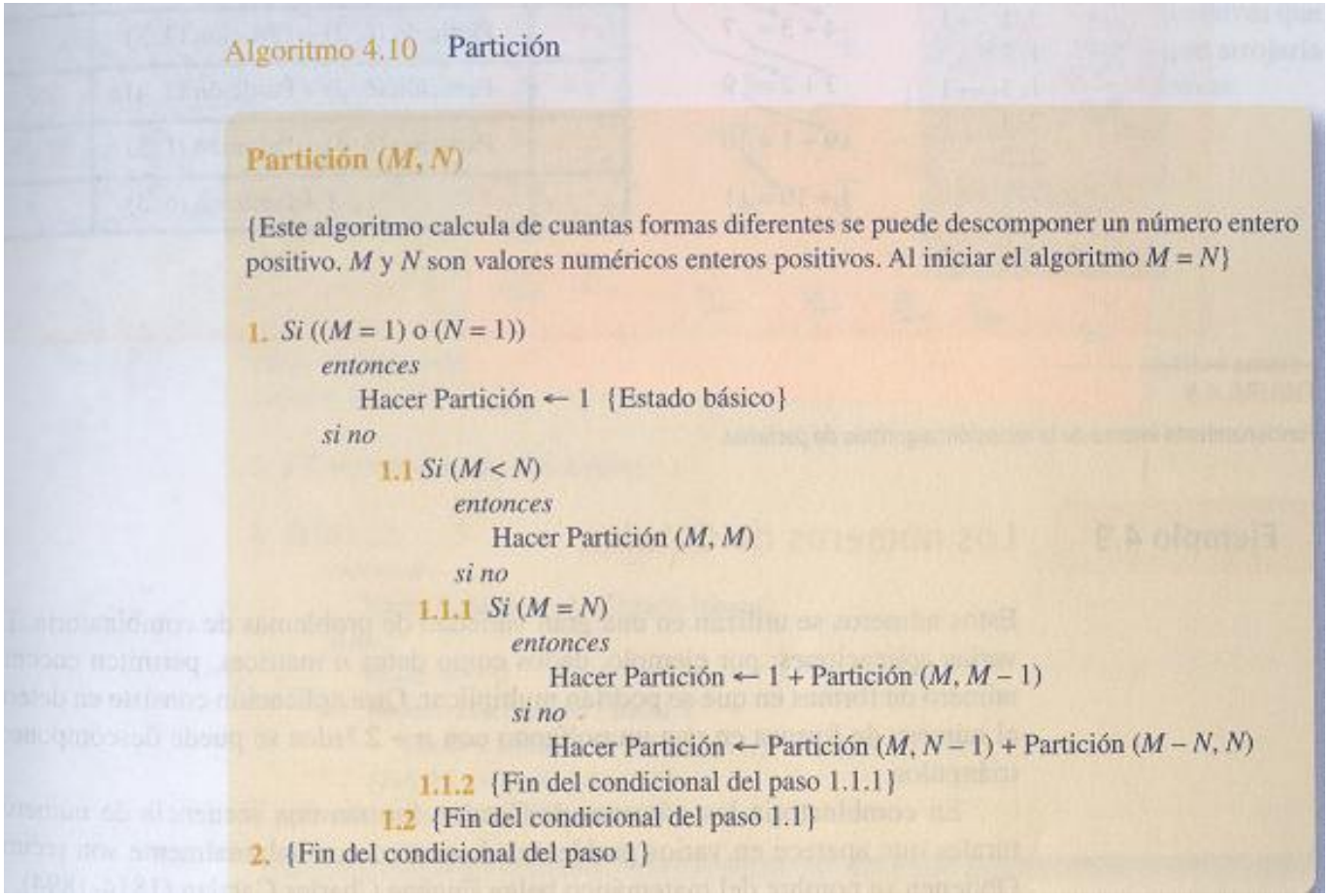
Ackerman Programa

M	*N*	Ackermann	Profundidad
2	2	7	27
2	3	9	44
2	4	11	65
2	7	17	152
3	5	253	42 438
3	6	509	172 233
3	10	8 189	44 698 325

Partición


Fórmula 4.5
Partición(M, N)
1 Si M = 1
1 Si N = 1
Partición(M, M) Si M < N
1 + Partición(M, – 1 M) Si M = N
Partición(M, N – 1) + Partición (M – N, N) Si M > N

Algortimo de Partición



Algoritmo 4.10 Partición

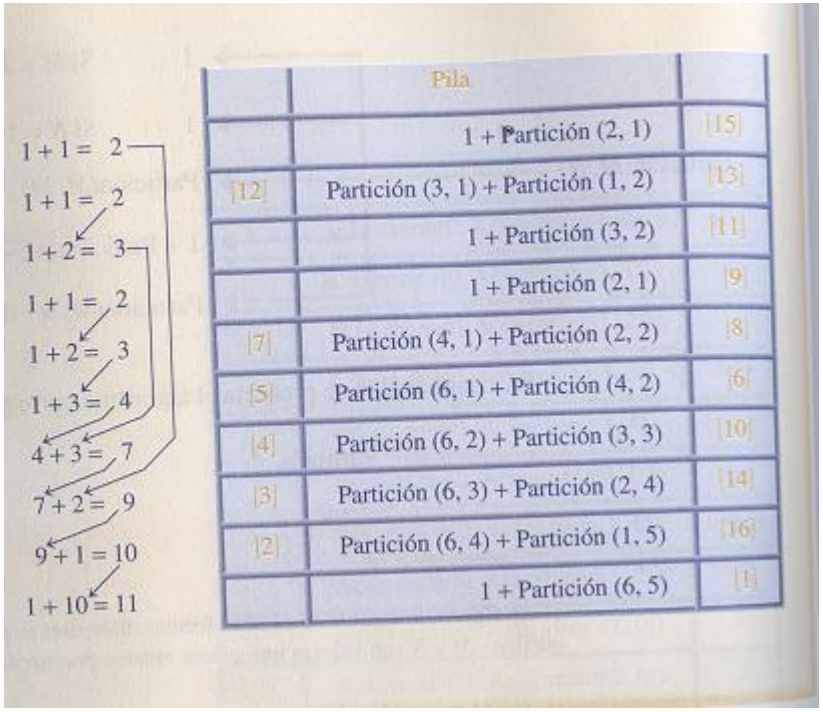
Partición (M, N)

{Este algoritmo calcula de cuantas formas diferentes se puede descomponer un número entero positivo. M y N son valores numéricos enteros positivos. Al iniciar el algoritmo $M = N$}

1. *Si* $((M = 1)$ o $(N = 1))$
   entonces
   Hacer Partición ← 1 {Estado básico}
   si no
   1.1 *Si* $(M < N)$
   entonces
   Hacer Partición (M, M)
   si no
   1.1.1 *Si* $(M = N)$
   entonces
   Hacer Partición ← 1 + Partición $(M, M - 1)$
   si no
   Hacer Partición ← Partición $(M, N - 1)$ + Partición $(M - N, N)$
   1.1.2 {Fin del condicional del paso 1.1.1}
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

Partición

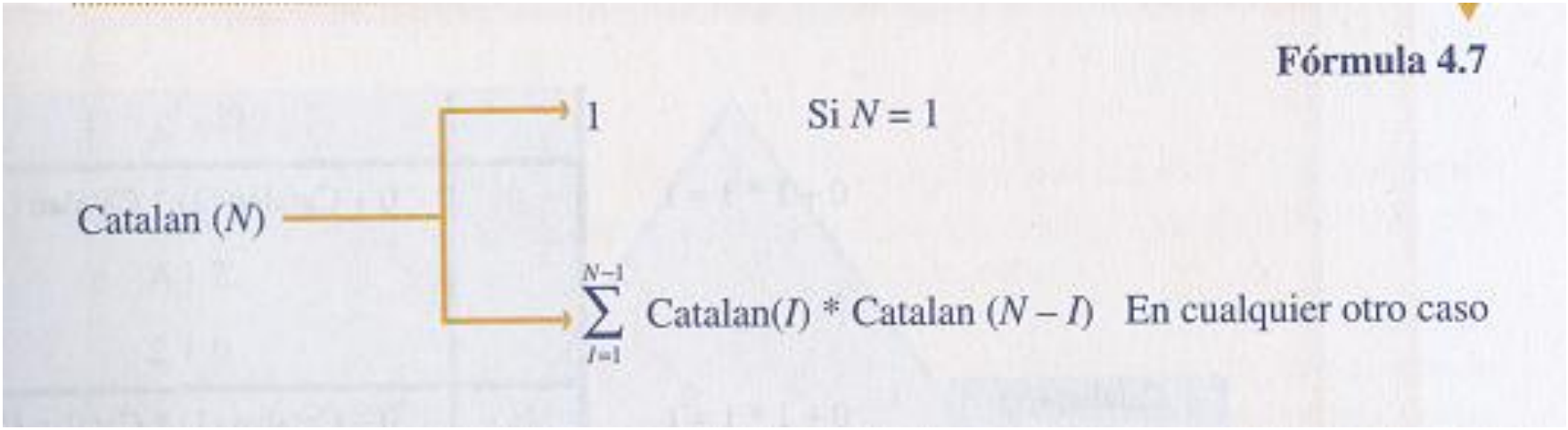
Partición (M, N)	
	6, 6
[1]	6, 5
[2]	6, 4
[3]	6, 3
[4]	6, 2
[5]	6, 1 → 1
[6]	4, 2
[7]	4, 1 → 1
[8]	2, 2
[9]	2, 1 → 1
[10]	3, 3
[11]	3, 2
[12]	3, 1 → 1
[13]	1, 2
$M < N$	1, 1 → 1
[14]	2, 4
$M < N$	2, 2
[15]	2, 1
[16]	1, 5
$M < N$	1, 1 → 1



	Pila	
	1 + Partición (2, 1)	[15]
[12]	Partición (3, 1) + Partición (1, 2)	[13]
	1 + Partición (3, 2)	[11]
	1 + Partición (2, 1)	[9]
[7]	Partición (4, 1) + Partición (2, 2)	[8]
[5]	Partición (6, 1) + Partición (4, 2)	[6]
[4]	Partición (6, 2) + Partición (3, 3)	[10]
[3]	Partición (6, 3) + Partición (2, 4)	[14]
[2]	Partición (6, 4) + Partición (1, 5)	[16]
	1 + Partición (6, 5)	[1]

Los Números de Catalán

Se utilizan en una gran variedad de problemas de combinatoria. Tienen varias aplicaciones. Ej. Dados como datos n matrices, permiten encontrar El número de formas en que se podrían multiplicar. Otra aplicación consiste en determinar el número de formas en que el polígono con n+2 lados se puede descomponer en n triángulos.



Los Números de Catalán

Los primeros números de Catalan siguiendo la fórmula recursiva son:

1, 1, 2, 5, 14, 42, 132, 429, 1 430,...

Algoritmo 4.11 Catalan

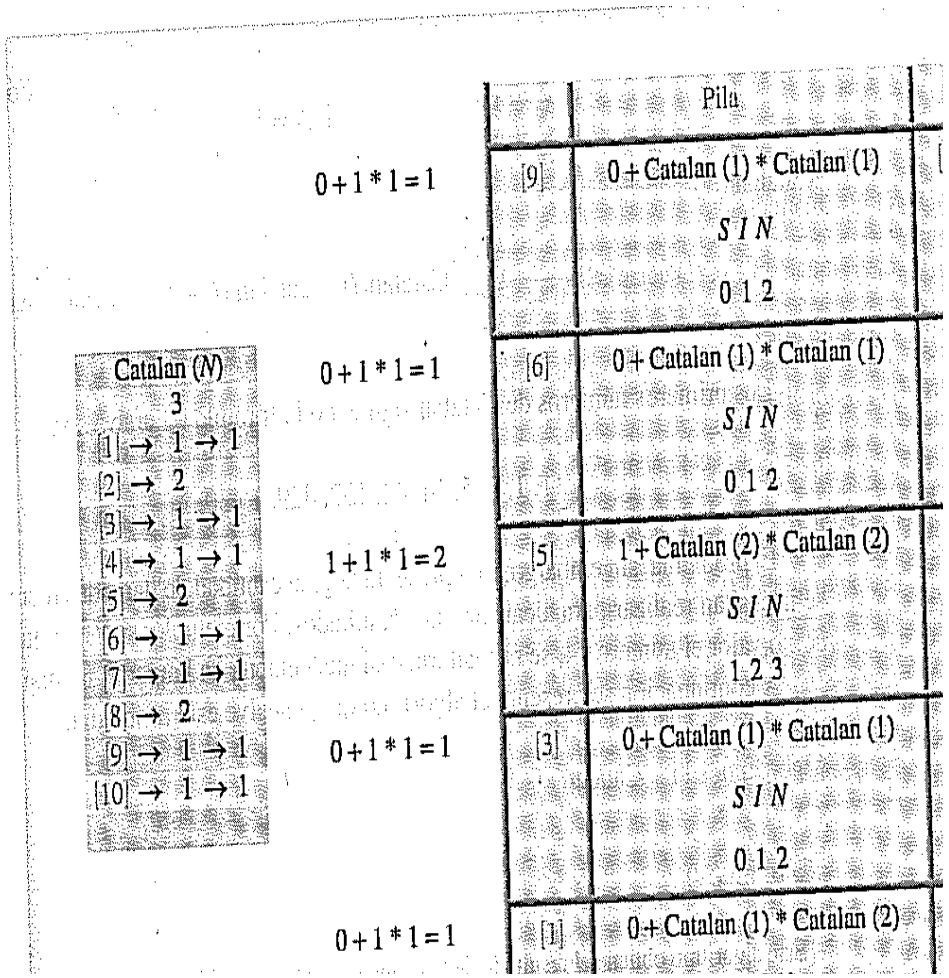
Catalan (*N*)

{Este algoritmo obtiene el resultado de los números de Catalan. *N* es un valor numérico entero positivo.}

{*I* y *S* son variables de tipo entero}

```
1. Si (N = 1)
      entonces
         Hacer Catalan ← 1  {Estado básico}
      si no
         Hacer S ← 0
      1.1 Repetir con I desde 1 hasta N
            Hacer Catalan ← S + Catalan (I) * Catalan (N – I)
      1.2 {Fin del ciclo del paso 1.1}
2. {Fin del condicional del paso 1}
```

Ejemplo



Catalan (*N*)	3
[1]	→ 1 → 1
[2]	→ 2
[3]	→ 1 → 1
[4]	→ 1 → 1
[5]	→ 2
[6]	→ 1 → 1
[7]	→ 1 → 1
[8]	→ 2
[9]	→ 1 → 1
[10]	→ 1 → 1

	Pila	
0+1*1=1	[9]	0+Catalan (1) * Catalan (1) *S I N* 0 1 2
0+1*1=1	[6]	0+Catalan (1) * Catalan (1) *S I N* 0 1 2
1+1*1=2	[5]	1+Catalan (2) * Catalan (2) *S I N* 1 2 3
0+1*1=1	[3]	0+Catalan (1) * Catalan (1) *S I N* 0 1 2
0+1*1=1	[1]	0+Catalan (1) * Catalan (2)

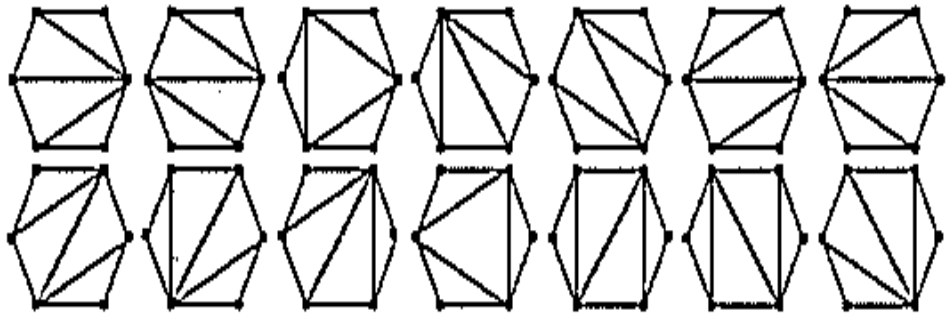


FIGURE 1. The fourteen triangulations of a hexagon.