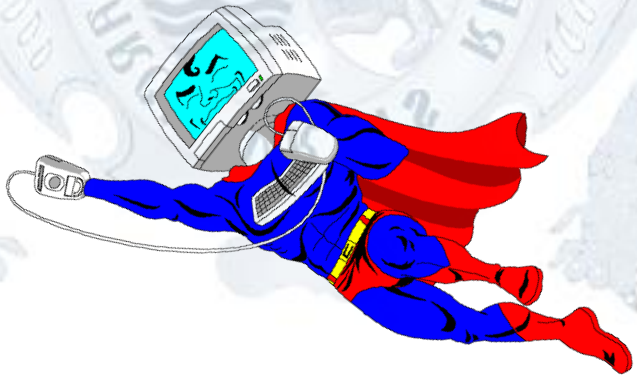
Robert Espinoza Domínguez

Análisis de las Estructuras de Control

Algoritmia vs Hardware

- Si las computadoras cada vez son más rápidas ¿vale la pena invertir tiempo diseñando algoritmos eficientes?

- O es mejor emplear equipos más potentes.



Algoritmia vs Hardware

- Supongamos que tenemos un algoritmo exponencial con $T_1(n) = 10^{-4} \times 2^n$ seg.
- Logramos una mejora de $T_2(n) = 10^{-6} \times 2^n$ seg. al invertir en un equipo más potente.

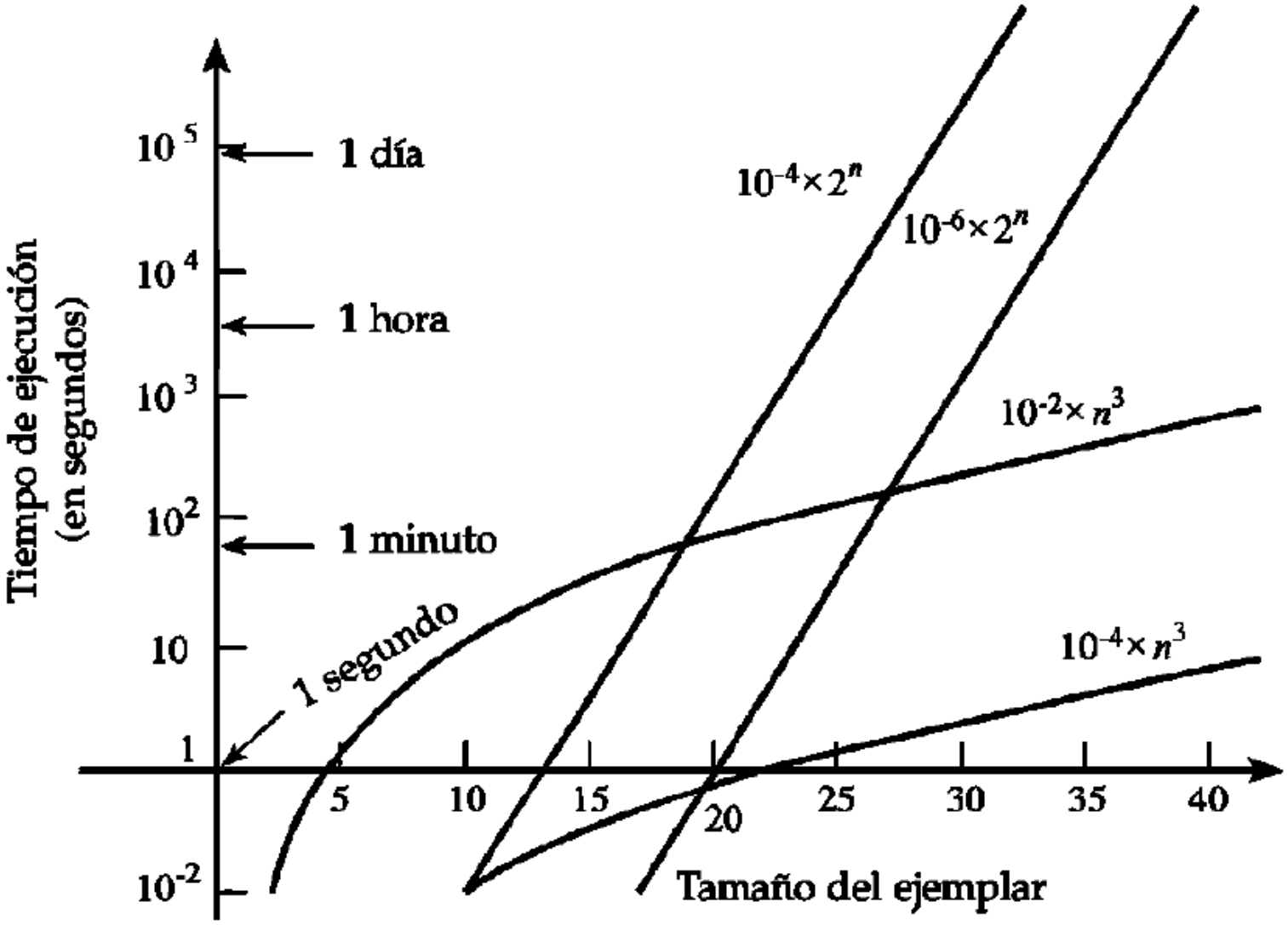
n	$10^{-4} \times 2^n$	$10^{-6} \times 2^n$
10	0.10 seg.	0.0010 seg.
20	1.74 min.	1.05 seg.
30	29.83 horas	17.90 min
40	1,272.58 días	12.73 días
45	40,722.65 días	407.23 días

Algoritmia vs Hardware

- En lugar de comprar un equipo más rápido, invertimos en encontrar un algoritmo más eficiente, un algoritmo cúbico con $T_3(n) = 10^{-2} \times n^3$ seg.

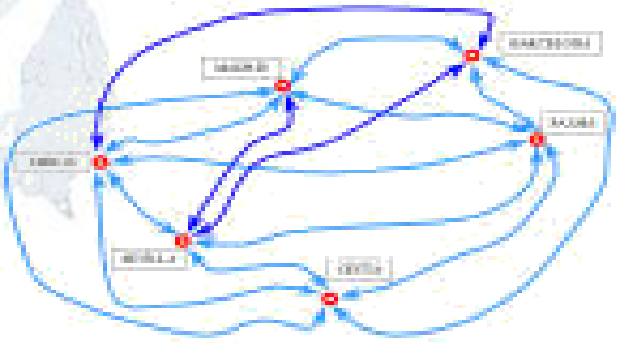
n	$10^{-4} \times 2^n$	$10^{-6} \times 2^n$	$10^{-2} \times n^3$
10	0.10 seg	0.0010 seg	10 seg
20	1.74 min	1.05 seg	1.33 min
30	29.83 horas	17.90 min	4.5 min
40	1,272.58 días	12.73 días	10.67 min
45	40,722.65 días	407.23 días	15.19 días
1500	…	…	393.63 días

Algoritmia vs Hardware



Análisis de la Eficiencia

- Un algoritmo es más eficiente si es mejor que un algoritmo evidente o se trata del mejor algoritmo posible para resolver nuestro problema.
- Un algoritmo de tiempo lineal puede no ser práctico si se considera la constante multiplicativa oculta es demasiado grande.
- Mientras que un algoritmo que requiera un tiempo exponencial en el peor caso puede resultar sumamente rápido en la mayoría de los casos.



Análisis de la Eficiencia

- Existen problemas prácticos que no se conoce ningún algoritmo eficiente. Ejemplos: el viajante de comercio, el coloreado óptimo de grafos, el problema de la mochila, los ciclos hamiltoneanos, la programación entera, la búsqueda del camino más simple más largo de un grafo.

- Algoritmo que resuelva cualquiera problema anterior proporciona un algoritmo eficiente para todos ellos.

- Problemas difíciles de resolver, pero se puede verificar eficientemente la validez de cualquier supuesta solución.

Problemas e Instancias

- Un problema es una cuestión o asunto que requiere una solución. Se produce cuando existe una diferencia entre una situación actual y una situación deseada.
- Una instancia de un problema está dado por un conjunto de datos particulares del problema. Por ejemplo:

  Problema: Calcular el área de un triángulo

  Instancia : base = 10, altura = 6
- La clase problema esta dada por el conjunto de todas las instancias posibles para el problema.

  Ejemplo: Considere el problema de ordenar en forma ascendente los n elementos de un vector de enteros positivos.

  Instancia = (4, (3, 5, 2 ,7))

  Donde n es el tamaño del vector y $v_1, v_2, .. ,v_n$ son sus elementos

Clase del problema = **{ (n, v_1, v_2, .. ,v_n) pertenece Z^{n+1} }**

Problemas e instancias

- Un algoritmo debe funcionar correctamente para todas las instancias o casos del problema que manifiesta resolver.
- Una forma de demostrar que un algoritmo es incorrecto sólo necesitamos encontrar un ejemplar del problema para el cual el algoritmo no encuentre una respuesta correcta.
- Es importante definir el ***dominio de definición*** del problema, es decir el conjunto de casos o instancias que deben considerarse.

Evaluación de la eficiencia

- **Enfoque empírico (a posteriori):**
  - Consiste en programar las técnicas competidoras e ir probándolas en distintos casos con la ayuda de una computadora.
- **Enfoque teórico (a priori):**
  - Consiste en determinar matemáticamente la cantidad de recursos (tiempo de ejecución y espacio) necesarios para cada uno de los algoritmos como *función del tamaño de los casos o instancias consideradas*.
  - No depende de la computadora, ni del lenguaje de programación, ni de las habilidades del programador.

Evaluación de la eficiencia

- El “tamaño” indica cualquier entero que mide de alguna forma el número de componentes de una instancia. Ejemplo:
  - Problema de Ordenación

    Tamaño: Número de ítems a ordenar
  - Problema de Grafos

    Tamaño: Número de nodos y aristas.
  - Operaciones que impliquen enteros: producto, factorial, etc.

    Tamaño: Valor del ejemplar ≠ Nro. bits que ocupa en memoria.
- **Enfoque híbrido:**
  - La forma de la función que describe la eficiencia del algoritmo se determina teóricamente.
  - Luego se determinan empíricamente aquellos parámetros numéricos que sean específicos para un cierto programa y para una cierta máquina.

Principio de Invariancia

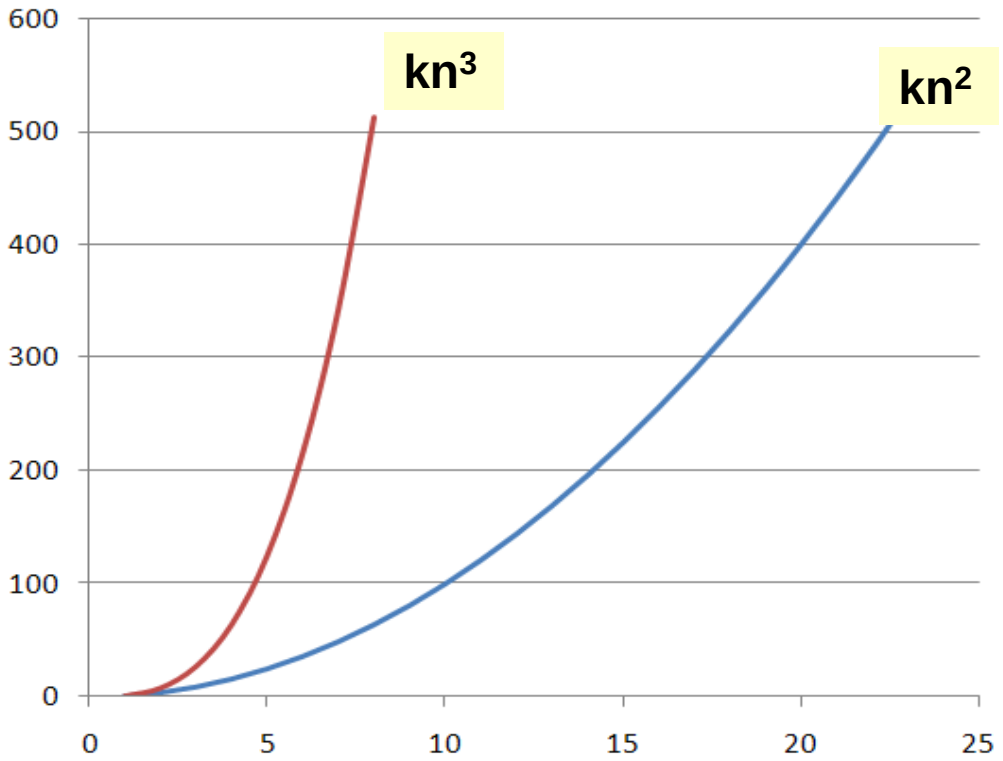
- Implementar un algoritmo depende de la máquina, del lenguaje y del factor humano.
- Dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en más de una constante multiplicativa.
  - Si dos implementaciones del mismo algoritmo necesitan $t_1(n)$ y $t_2(n)$ segundos para resolver un caso de tamaño n
  - Entonces, siempre existen constantes positivas c y d, tales que $t_1(n) <= ct_2(n)$ y $t_2(n) <= dt_1(n)$, siempre que n sea lo suficientemente grande
  - Concluimos que toda implementación t(n) esta acotada superiormente por k*t(n).

Principio de Invariancia

- El uso de segundos es arbitrario: sólo se necesita modificar la constante para acotar el tiempo por ***at(n)*** años o por ***bt(n)*** microsegundos.
- Ejemplos: Algoritmos y tiempos de ejecución.
  - Lineal: $T(n) <= kn$
  - Cuadrático: $T(n) <= kn^2$
  - Cúbico: $T(n) <= kn^3$
  - Polinómico: $T(n) <= n^k$
  - Exponencial: $T(n) <= k^n$
  - Logarítmico: $T(n) <= n \log(n)$

Constantes ocultas

- Normalmente se ignoran los valores exactos de las constantes, y se supone que todos ellos son aproximadamente del mismo orden en magnitud.
- Por ejemplo, podemos afirmar que un algoritmo cuadrático es más rápido que un algoritmo cúbico.



Constantes ocultas

- Sin embargo, en algunos casos hay que ser más cuidadosos.
- Por ejemplo dos algoritmos cuyas implementaciones requieren: n^2 días y n^3 segundos, solamente en casos que requieran más de 20 millones de años para resolverlos, el algoritmo cuadrático será más rápido que el cúbico.

n	n^2 años	n^3 años
10	0.27	0.000032
100	27.40	0.031710
500	684.93	3.963724
1,000	2,739.73	31.71
10,000	273,972.60	31,709.79
80,000	17,534,246.58	16,235,413.50
86,400	20,451,945.21	20,451,945.21
90,000	22,191,780.82	23,116,438.36
100,000	27,397,260.27	31,709,791.98

Operación elemental

- Es aquella operación cuyo tiempo de ejecución se puede acotar superiormente por una constante, y sólo dependerá de la implementación particular usada (máquina, lenguaje de programación, etc.)
- La constante no depende ni del tamaño ni de los parámetros del ejemplar que se esté considerando.
- Sólo importará el número de operaciones elementales en el análisis y no el tiempo exacto de cada una de ellas.

Operación Elemental

- Por ejemplo, un algoritmo hace *"a"* adiciones, *"m"* multiplicaciones y *"s"* asignaciones, con tiempos de ejecución para cada operación de t_a, t_m *y* t_s *que dependen* de la máquina utilizada.

  Entonces, el tiempo total ***t*** para el algoritmo estará acotado por:

$$t <= at_a + mt_m + st_s$$

$$t <= max(t_a, t_m, t_s) \times (a + m + s)$$

- Ya que el tiempo exacto requerido por cada operación elemental no es importante, simplificaremos diciendo que las operaciones elementales se pueden ejecutar ***a coste unitario***.

Operación Elemental

- Las sumas, restas, multiplicaciones, módulos, etc. en teoría no son operaciones elementales, pues el ***tiempo necesario para ejecutarlas aumenta con la longitud de los operandos***.
- Por ejemplo, suele producirse un desborde aritmético cuando calculamos el factorial de un número muy grande.
- Sin embargo, se puede considerar en la práctica como operaciones elementales, con tal de que los operandos implicados sean del tamaño razonable.

Criterios para definir la complejidad

- **Criterio de Costo Uniforme:** Cada instrucción de la máquina es ejecutada en una unidad de tiempo.
  - Sea x_1, x_2, ... x_n instrucciones; entonces $t(x_i) = 1$ para i = 1, 2, ..., n.
  - Entonces el tiempo de ejecución es el número de operaciones.
- **Criterio de Costo Logarítmico:** El tiempo de ejecución es proporcional al tamaño de los datos (número de bits necesario para codificar los datos).
  - El número de bits necesario para representar un número entero a esta dado por:

$$\omega(a) = \begin{cases} \lfloor \log_2 |a| \rfloor + 1 & a \in Z - \{0\} \\ 1 & a = 0 \end{cases}$$

Análisis de la Eficiencia

- El análisis de los algoritmos suele efectuarse desde adentro hacia fuera.
- Primero se determina el tiempo requerido por instrucciones individuales.
- Luego se combinan de acuerdo a las estructuras de control que enlazan las instrucciones.
- Tomaremos como coste unitario una operación que requiera realmente una cantidad de tiempo polinómica.
- Contaremos las sumas y multiplicaciones con un coste unitario, aun en aquellos operandos cuyo tamaño crece con el tamaño del caso, siempre que esté acotado por algún polinomio.

Análisis de la Eficiencia

- Si se necesita operandos tan grandes, es preciso descomponerlos en n segmentos, guardarlos en un vector, e invertir el tiempo necesario para efectuar la aritmética de precisión múltiple; tales algoritmos no pueden ser de tiempo polinómico.
- Es necesario conocer principios generales para analizar las estructuras de control más frecuentemente conocidas.

Estructuras Secuenciales

- Sean P_1 y P_2 dos fragmentos de un algoritmo (instrucciones simples ó una simple y una compuesta).
- Sean t_1 y t_2 los tiempos requeridos por P_1 y P_2 respectivamente.
- **Regla de la composición secuencial:** El tiempo necesario para calcular "P_1;P_2" (primero P_1 y luego P_2) es simplemente t_1+t_2.

Estructuras Secuenciales

P1 Seguido de P2 = T1 + T2

Estructuras Condicionales Si

Regla 1:

- El tiempo de ejecución del condicional es la suma del tiempo de ejecución de la condicional y el mayor de los tiempos de ejecución de las alternativas (dos instrucciones simples ó uno compuesto y uno simple).

$$T_{SI} = T_{CONDICIONAL} + MAX(T_{ENTONCES}, T_{SINO}))$$

Estructuras Condicionales Si

Regla 2:

- El tiempo nunca es mayor que el tiempo de ejecución de la condicional más el mayor de los tiempos de ejecución de las alternativas (dos instrucciones compuestas).

Estructuras Condicionales Si

Ejemplo:

```
Si  (i = j) entonces                    O(1)
        suma ← suma + 1                 O(2)
Sino
        suma  ← suma + i * j            O(3)
fSi
```

$$t(n) = 1 + Max(2,3) = 4$$

Estructuras Condicionales Múltiple (case)

Se tomará el caso peor posible

Si (condición 1)
    Tratamiento 1
Sino Si condición 2
    Tratamiento 2

Sino Si condición n
    Tratamiento n
Fin Si

$$t(n) = Max(O(Trat1), O(Trat2), ..., O(Tratn))$$

Estructuras Para ó Desde

Para i desde 1 hasta m hacer
P(i)
fPara

- P(i) no depende de i entonces, suponiendo que t es el tiempo requerido para calcular P(i), el bucle se ejecuta m veces, cada una con un tiempo t.
- El tiempo total mt, es una cota inferior de la función tiempo.

Observaciones:

- No hemos tomado en cuenta el tiempo para el control del bucle.

$$T_{para}(n) = k >= mt$$

Estructuras Para ó Desde

```
i ←1
Mientras i <= m hacer
    P(i)
    i ← i + 1
fMientras
```

- Sea c tiempo para asignación y para la comparación.

```
k <=     c              para i ← 1
       + (m+1)c         para las comprobaciones si i<=m
       + mt             para las ejecuciones de P(i)
       + 2mc            para las ejecuciones de i ← i + 1
       + (m+1)c         para las ejecuciones de salto
       ➔ k <= (t + 4c) m + 3c
```

- Esto está acotado superiormente por (t + 4c) m + 3c.

Reglas de la Estructura Para

Regla1:

- El tiempo de ejecución de un ciclo ***Para*** es a lo más el tiempo de ejecución de las instrucciones que están en el interior del ciclo más el tiempo del control y todo por el número de iteraciones .

$$T = (T_{INTERIOR} + T_{PARA}) * N^{o}\ Ciclos$$

Reglas de la Estructura Para

- Ejemplo:

```
Para  i  desde  1  hasta  n  hacer
      A[i] ← 0                                  O(1)
fPara
```

$$\sum_{i=1}^{n} 4 + 3 = 4n + 3$$

Reglas de la Estructura Para

Regla2:

- El tiempo de un grupo de ciclos ***Para*** es a lo más el tiempo de ejecución de las instrucciones que están en el interior del ciclo multiplicado por el producto de los tamaños de todos los ciclos ***Para***.

Reglas de la Estructura Para

Ejemplo:

```
Para  i  desde  1  hasta  n  hacer
        Para  j  desde  1  hasta n  hacer
                k ← k +  1                O(2)
        fPara
fPara
```

$$\sum_{i=1}^{n}((\sum_{j=1}^{n}5+3)+3)+3=\sum_{i=1}^{n}(5n+6)+3=(5n+6)n+3=5n^2+6n+3$$

Estructuras Mientras y Repetir

- El tiempo es la multiplicación del tiempo de las instrucciones del interior por el número de ciclos de esta estructura.

$$T = (T_{INTERIOR} + T_{MIENTRAS}) * T_{\#CICLOS}$$

- No existe una forma evidente a priori para saber la cantidad de veces que hay que pasar por el bucle.
  - Hallar una función de las variables implicadas en el control del bucle.
  - Determinar el valor para el cual los ciclos de la estructura llegan a su final.
  - Determinar la forma en que disminuye las variables del control del bucle.

Estructuras Mientras y Repetir (Ejemplo)

J ← 1	O(1)
Mientras (j <= n) hacer	
Modulo A	O(m)
J ← B * J	O(2)
fMientras	

Solución:

a). T = n – j + 1

b). J > n

c). J ← 1, B, B^2, B^3, …,B^k, por tanto Modulo A se repite k veces.

➔ $B^k > n$ ➔ $Log_B B^k > Log_B n$ ➔ $k > Log_B n$ ➔ $k = Log_B n + c$

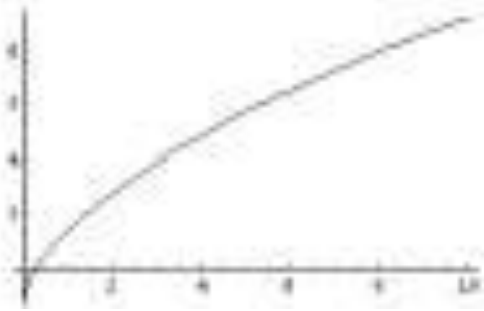
$$T_{MIENTRAS} = (m+4)*(Log_B n + c) + 3$$

Algoritmos de Tiempo Polinomial

- Son algoritmos eficientes y los problemas se resueltos con estos algoritmos se dice que son problemas tratables.

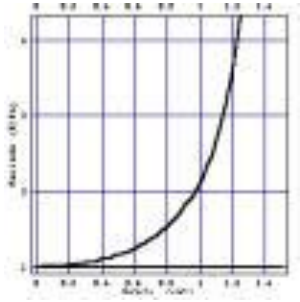
- Estos algoritmos presentan una función tiempo que se resuelven en un tiempo llamado tiempo polinomial.

- Se puede decir que un algoritmo tiene complejidad polinomial, o se ejecuta en **tiempo polinomial,** si tiene un orden $O(n^x)$.



Algoritmos de Tiempo Exponencial

- Los problemas que se resuelven usando algoritmos de tiempo exponencial se conoce como problemas intratables y a los algoritmos como algoritmos ineficientes.
- Un algoritmo tiene complejidad exponencial si la función de tiempo T(n) tiene un orden $O(x^n)$.
- Un algoritmo que tenga tiempo polinomial es eficiente ya que su tiempo crece mas despacio que un exponencial al aumentar el tamaño de la entrada, por ello aprovecha mejor los cambios tecnológicos.



Algoritmos recursivos

Recursión

- La recursión permite definir un objeto (problemas, estructuras de datos) en términos de sí mismo.
- Ejemplos de recursión son las estructuras: Árboles, Listas enlazadas.
- La representación de una recursión en forma gráfica se puede realizar por medio de un **árbol recursivo**.
- Un procedimiento recursivo debe tener:
  - Un criterio llamado ***criterio básico***, por el que el método no se llame a sí mismo, sino que llegue a una solución directa.
  - Cada vez que el método se llame a sí mismo (directa o indirectamente), debe estar más cerca del criterio base.

Ecuaciones de recurrencia

- Una ecuación de recurrencia es aquella que define una secuencia recursiva; cada término de la secuencia es definido como una función de términos anteriores.

$$a_n = f(a_{n-1}, a_{n-2}, \ldots)$$

- Ejemplos:
  - En Análisis de algoritmos, aparece la relación

    $T(n) = aT(n/b) + g(n)$ donde $a>=1$, $b>=1$
  - La “función logística”

    $x_{n+1} = r\, x_n\, (1 - x_n)$

    se usa para calcular modelos de crecimiento.

Ejemplo: Factorial

$n! = n\ (n-1)\ (n-2).....1$

$(n-1)! = (n-1)\ (n-2).....1$

de donde:

Ley de recurrencia:

$0! = 1$ si $n=0$

$n! = n\ (n-1)!$ si $n>0$

Algoritmo Recursivo - Factorial

Clase **cFactorial**

```
Método calculaFactorial(num)
    Si num = 0 then
        retornar 1
    sino
        retornar num * calculaFactorial(num-1)
    fin si
Fin Método
```

Fin Clase

Factorial (num-1) crea una réplica del subprograma con el nuevo parámetro.

Algoritmo Recursivo - Factorial

Fac(4)=4*Fac(3)

Fac(4)=4*(3*Fac(2))

Fac(4)=4*(3*(2*Fac(1)))

Fac(4)=4*(3*(2*(1*Fac(0))))

Fac(4)=4*(3*(2*(1*1)))

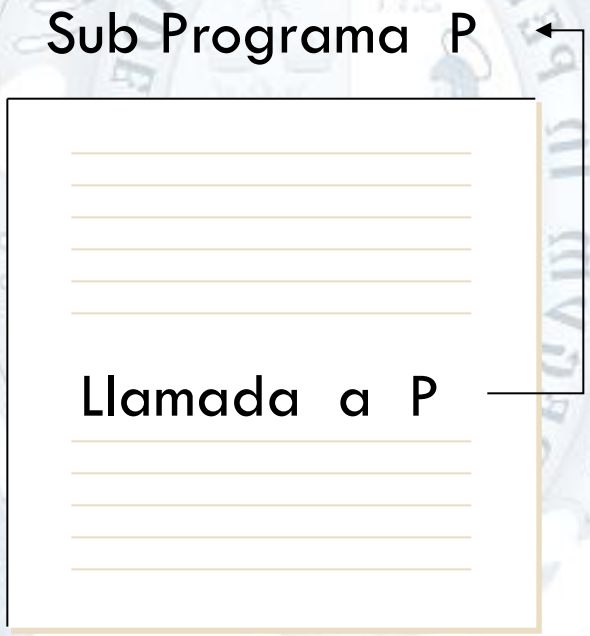
Fac(4)=4*(3*(2*1))

Fac(4)=4*(3*2)

Fac(4)=4*6=24

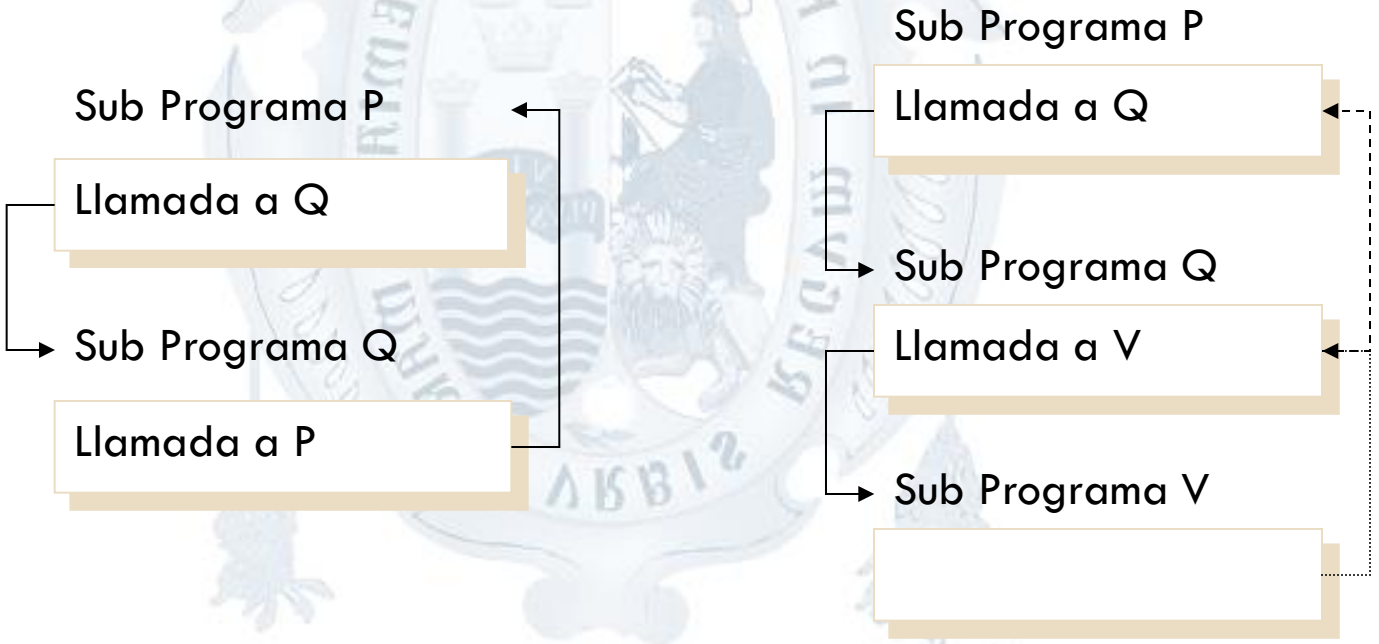
Tipos de Recursión

- **Recursión Directa**: El subprograma se llama directamente a sí mismo.



Tipos de Recursión

- **Recursión Indirecta**
  - El subprograma llama a otro subprograma, y éste a su vez llama al primero ó de lo contrario a un tercero.



Propiedades de la Recursión

- Un procedimiento recursivo con **criterio base** y **aproximación** constante a este criterio base se dice que está *bien definido*.
- Se dice profundidad recursiva al número de veces que se llama recursivamente un subprograma.
- La representación en forma gráfica se puede realizar por medio de un **árbol recursivo**.
- Un procedimiento recursivo con estas dos propiedades se dice que está *bien definido*.

Funcionamiento Interno de la Recursión

- **Valores a Almacenar:**

  Cuando un programa que llama a otro subprograma, debe guardarse varios elementos:

  - Debe guardarse los valores de los parámetros del subprograma que llama; para poder encontrarlos después que retorna el control a este subprograma.

  - Debe guardarse la dirección de la instrucción que se debe ejecutar a continuación del subprograma llamado, para retornar el control a esta instrucción.

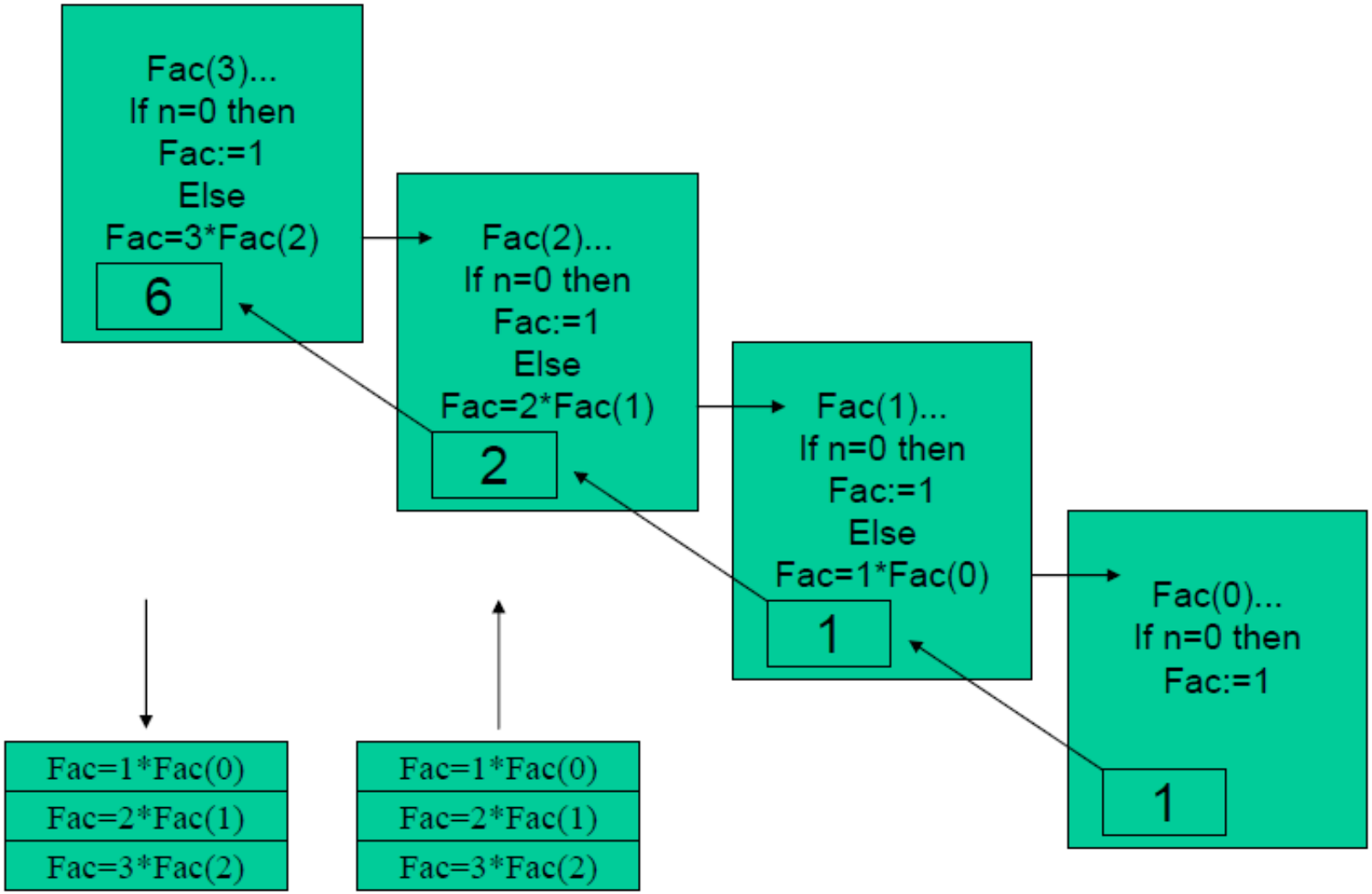
Funcionamiento Interno de la Recursión

- **Pila**
  - Es una lista de elementos a la cual se le puede insertar o eliminar elementos, sólo por uno de sus extremos. (LIFO)

- Internamente se utiliza una estructura tipo pila, que guarda una copia de los valores de variables y constantes locales del subprograma que efectúa la llamada.

- Además, se guarda una referencia a la siguiente instrucción a ejecutar.

Proceso de ejecución de un programa recursivo



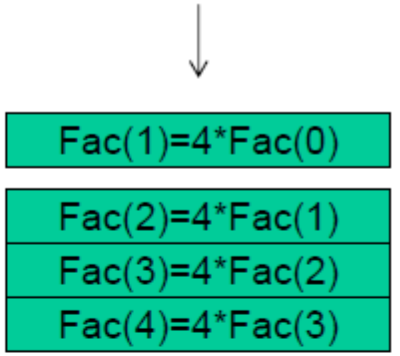
Proceso de ejecución de un programa recursivo

Fac(4)=4*Fac(3)

Fac(4)=4*(3*Fac(2))

Fac(4)=4*(3*(2*Fac(1)))

Fac(4)=4*(3*(2*(1*Fac(0))))



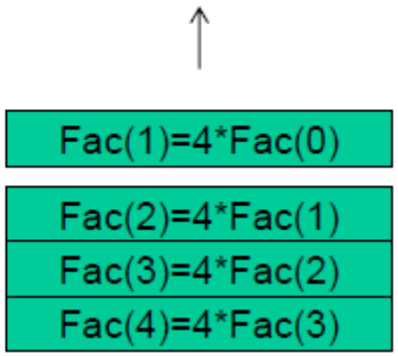
- Para obtener la solución final se deshacen las llamadas anteriores siguiendo el orden inverso (pila recursiva).

Fac(4)=4*(3*(2*(1*1)))

Fac(4)=4*(3*(2*1))

Fac(4)=4*(3*2)

Fac(4)=4*6=24



Más ejemplos de Recursividad

Sucesión de Fibonacci

- Sucesión de Fibonacci

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

$$\begin{cases} f_1 = 0;\ f_2 = 1 \\ f_n = f_{n-1} + f_{n-2} \qquad para\ n \geq 3 \end{cases}$$

Sucesión de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

Ley de recurrencia (n término fibonacci)

f(n)=0 si n = 1
f(n)=1 si n = 2
f(n)=f(n-1)+f(n-2) si n>=3

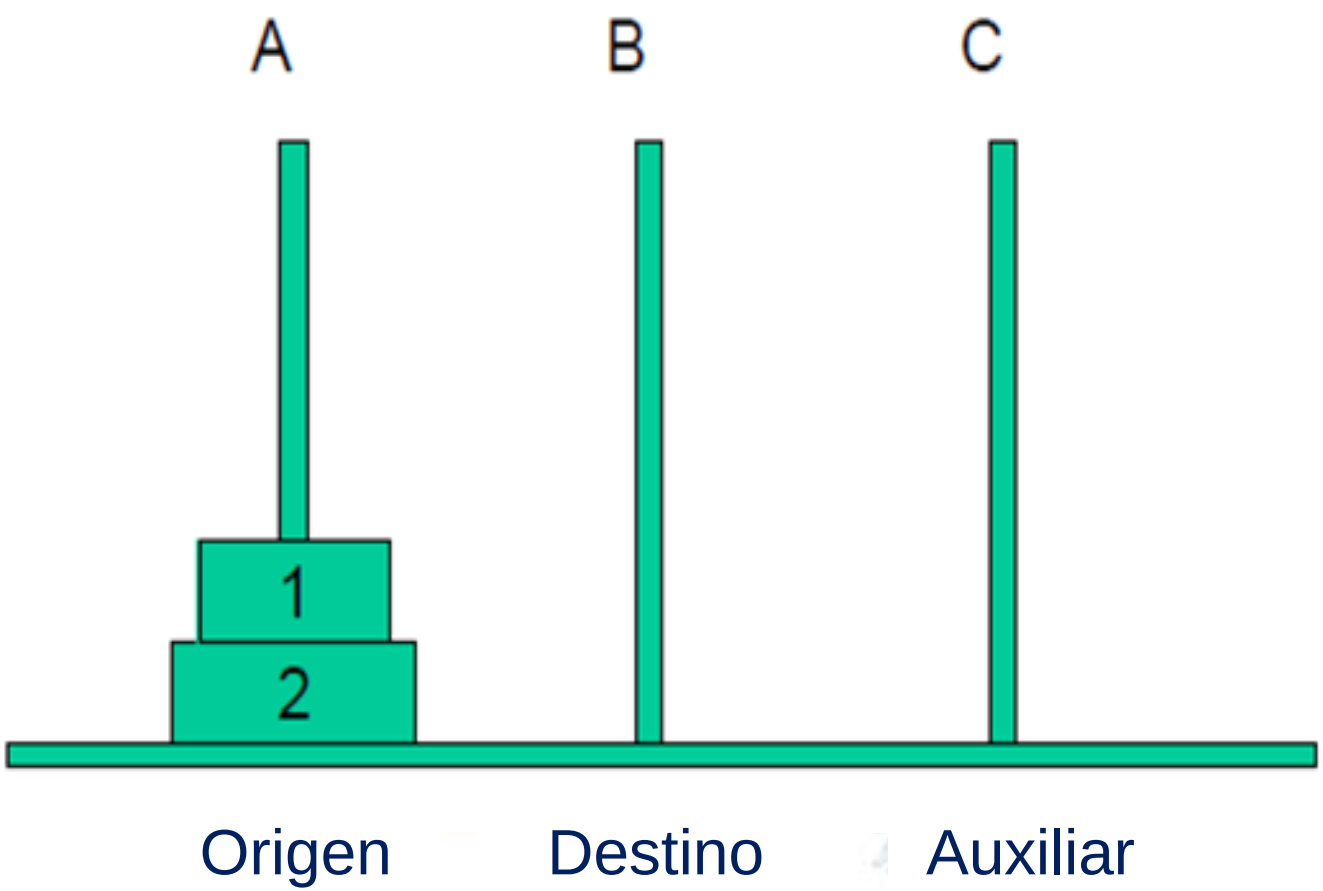
Sucesión de Fibonacci

```
Clase cFibonacci
    Método fibonacci(numero)
        Si (numero = 1) entonces
            retornar 0
        Si no
            Si (numero = 2 ) entonces
                retornar 1
            Si no
                retornar fibonacci(numero-1) + fibonacci(numero-2)
            Fin si
        Fin si
    Fin Método
Fin Clase
```

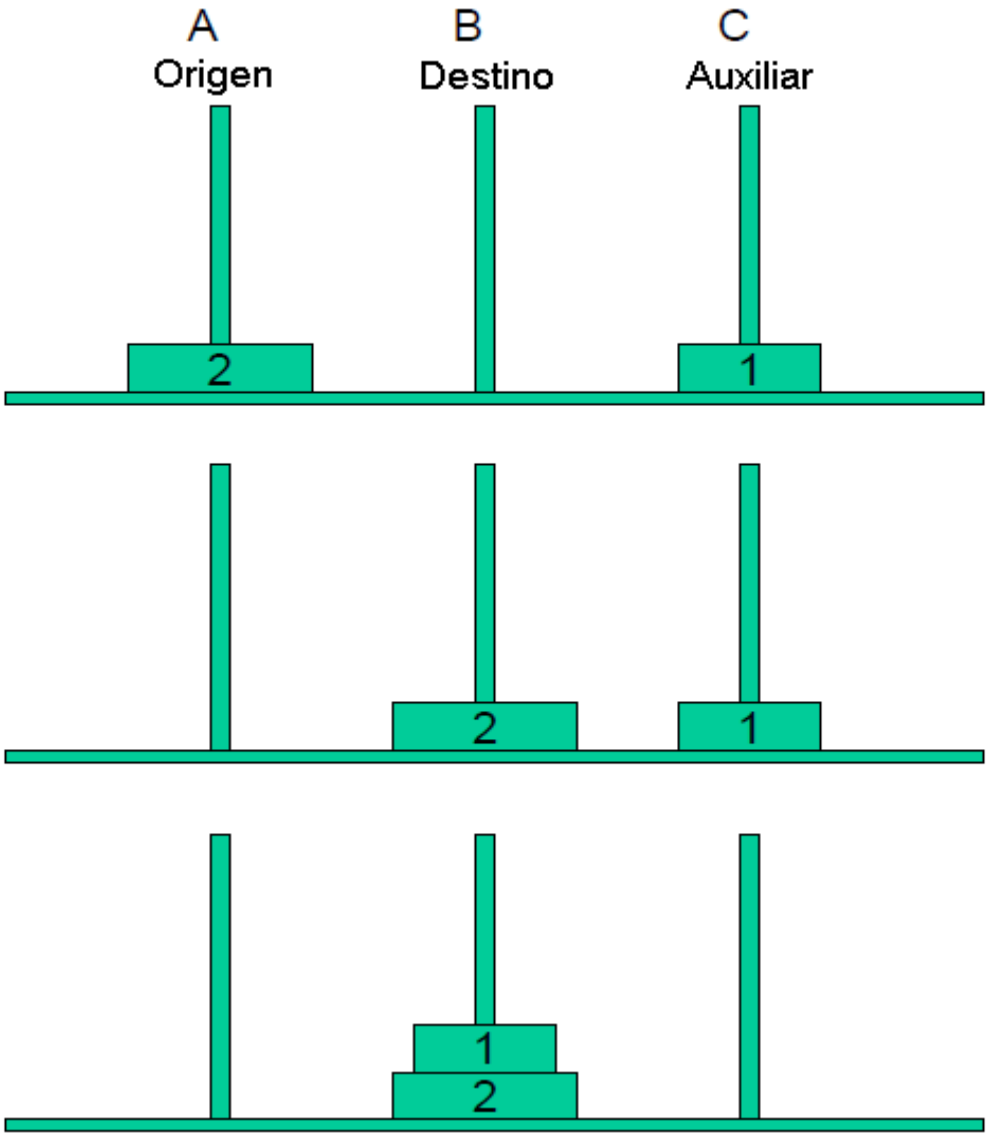
El problema de las Torres de Hanoi

- Tenemos 3 torres y N discos de diferentes tamaños; cada uno con una perforación en el centro que le permite deslizarse por las torres.
- Inicialmente los N discos están ordenados de mayor a menor en la primera de las torres.
- Se debe pasar los discos a otra torre en el mismo orden, utilizando la tercera torre como auxiliar.
- En cada movimiento sólo se puede mover un disco y no puede quedar uno de mayor tamaño sobre uno menor tamaño.

Torres de Hanoi con dos discos



Torres de Hanoi con dos discos

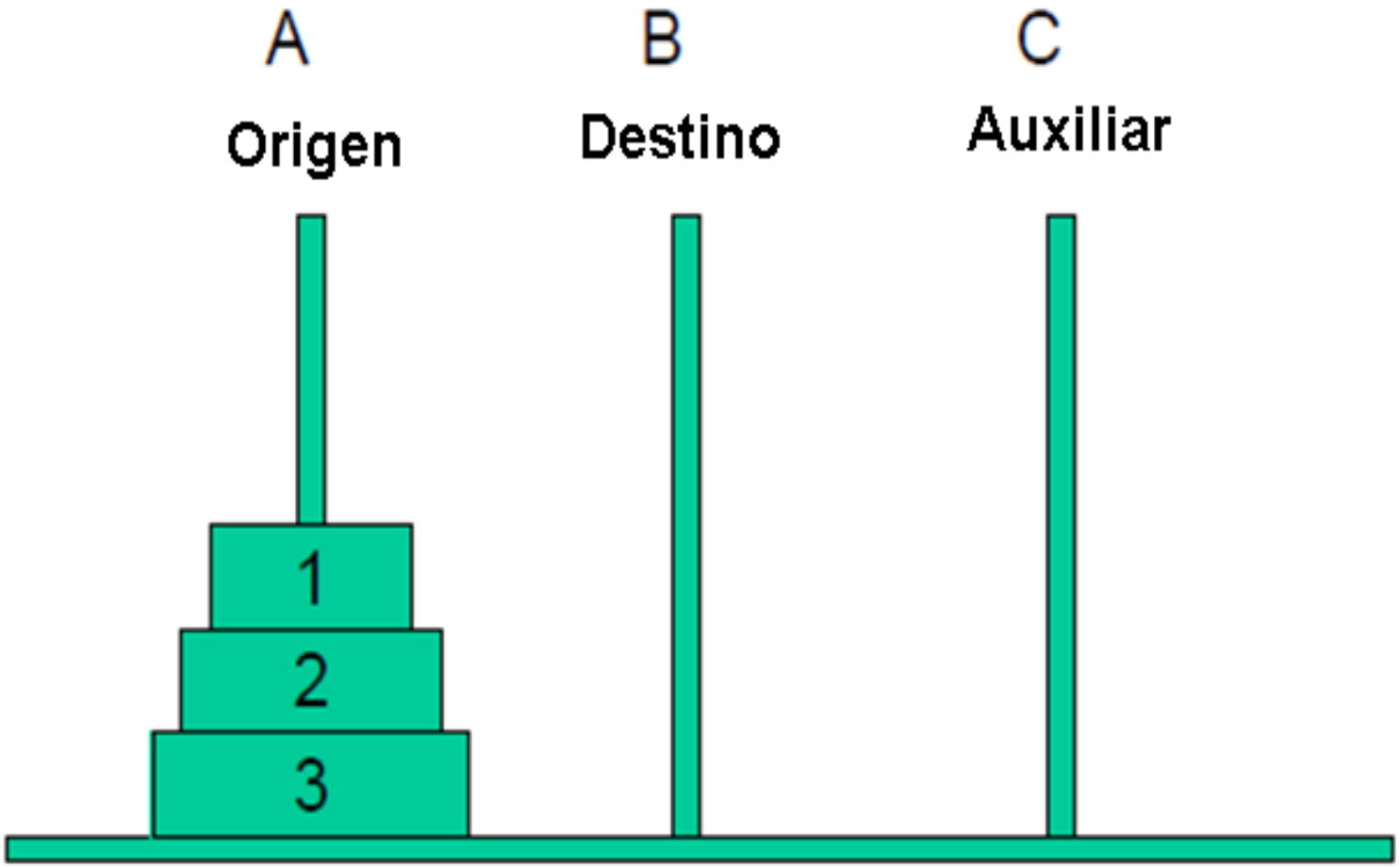


Pasar un disco de A a C

Pasar un disco de A a B

Pasar un disco de C a B

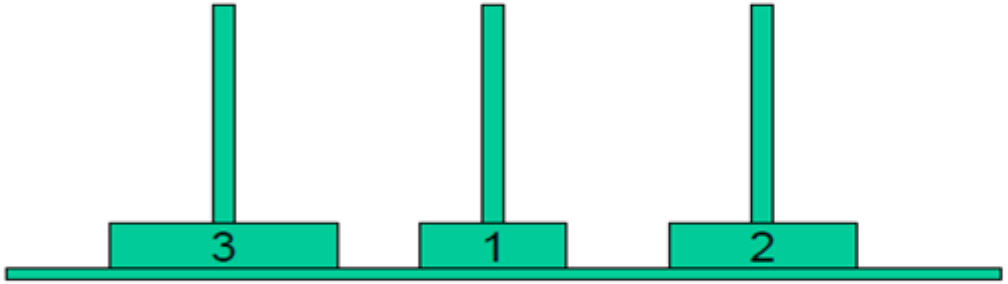
Torres de Hanoi con tres discos



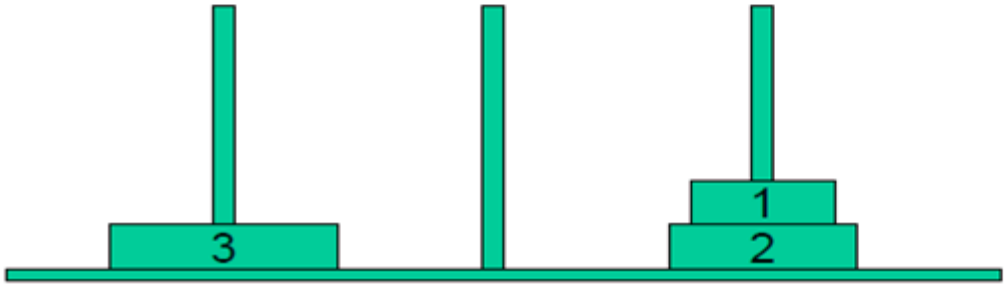
Torres de Hanoi con tres discos

Pasar los dos discos superiores de A a C

Origen - Destino



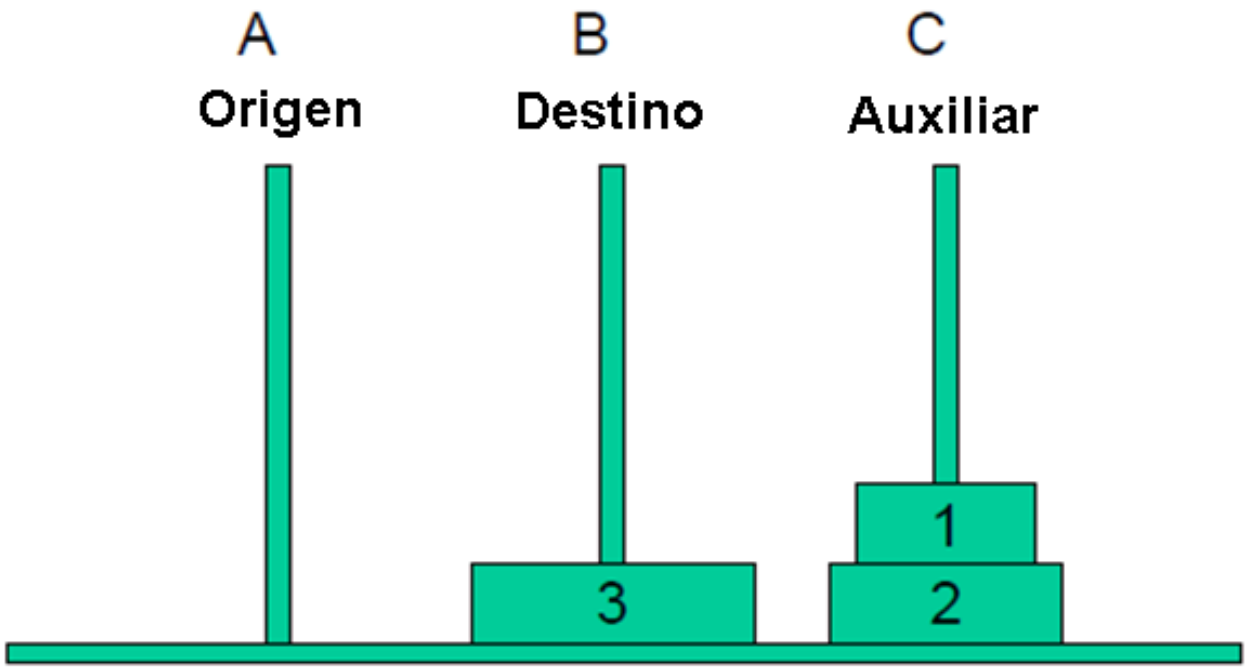
Origen - Auxiliar



Destino - Auxilar

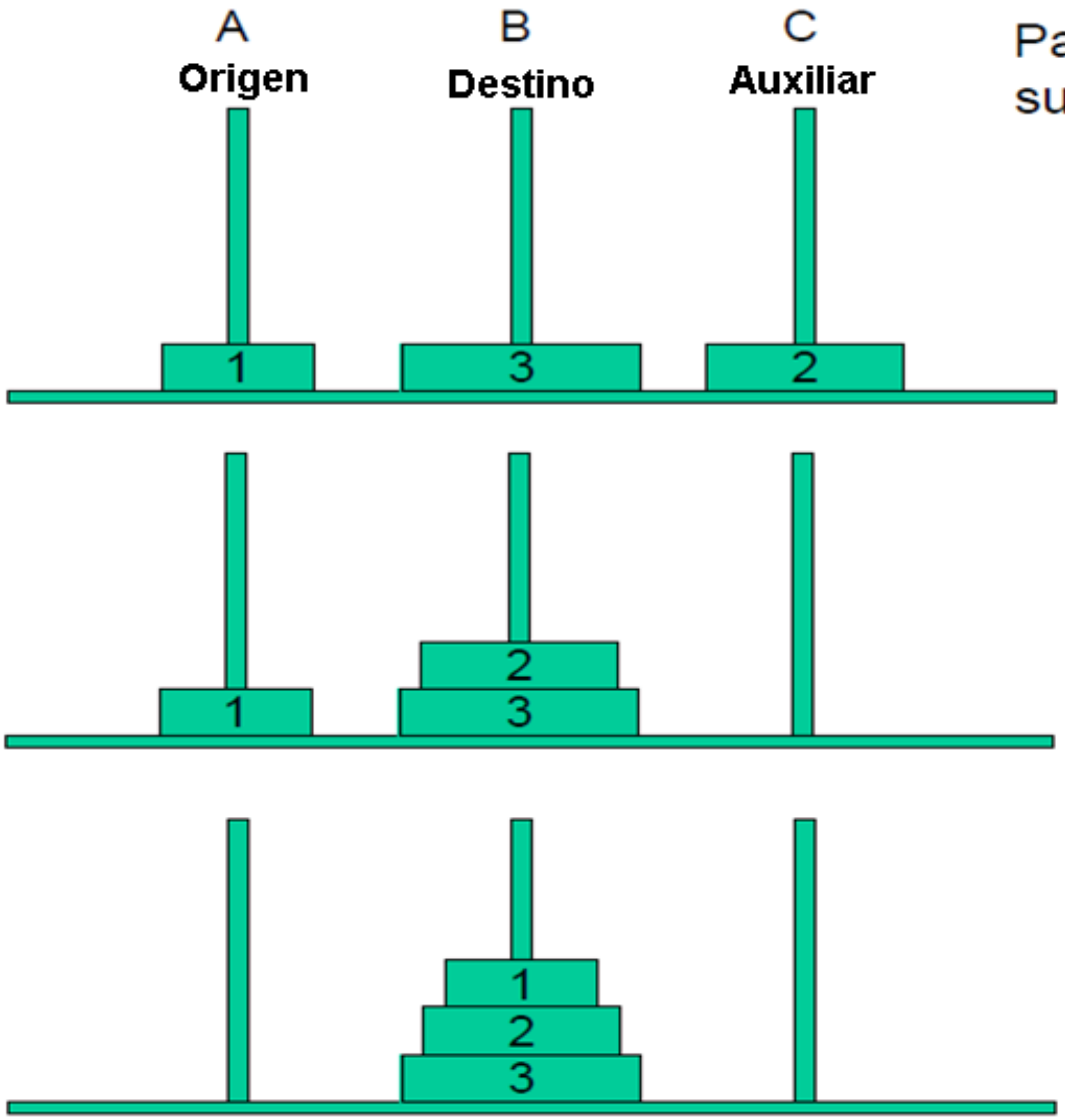
Torres de Hanoi con tres discos

Pasar el tercer disco
de A a B



Origen - Destino

Torres de Hanoi con tres discos



Pasar los dos discos superiores de C a B

Auxiliar - Origen

Auxiliar - Destino

Origen - Destino

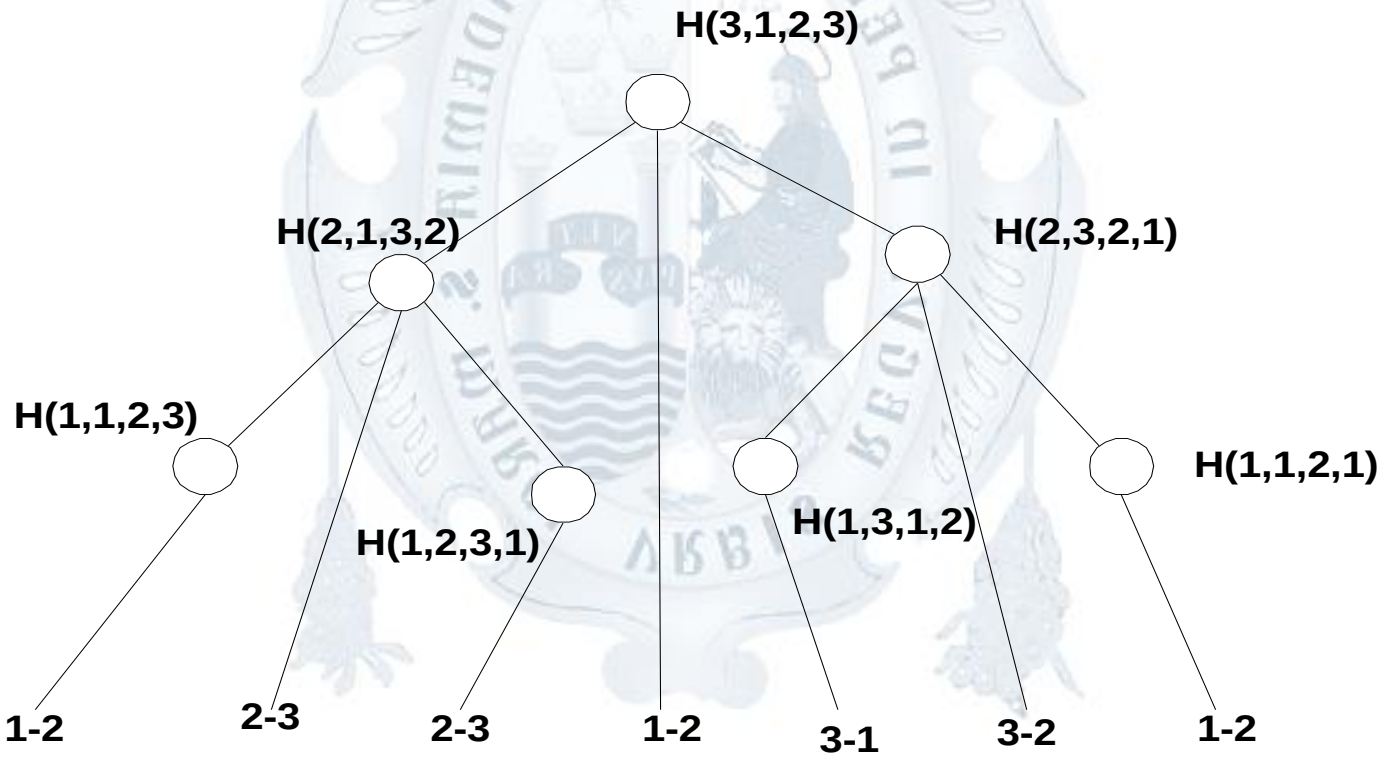
El problema de Torres de Hanoi

```
Método CTorres.Hanoi (nDiscos, Origen, Destino, Auxiliar)
Si (nDiscos = 1) entonces
        Escribir("Mover disco Origen a Destino")
sino
        Torres.Hanoi( nDiscos -1, Origen, Auxiliar, Destino)
        Escribir("Mover disco Origen a Destino")
        Torres.Hanoi( nDiscos-1, Auxiliar, Destino , Origen)
FMétodo
```

El problema de Torres de Hanoi

- Concluimos que árbol recursivo nos permite calcular en función del # de hojas, cuantas llamadas recursivas se dan: $2^n - 1$

- Árbol recursivo de Torres de Hanoi para n = 3



Inducción matemática

Deducción

- Inferencia de lo general a lo particular
- Si las dos premisas son ciertas, entonces la conclusión necesariamente será cierta.

  Todos los patos de este corral son blancos
  Estos patos son de este corral

  Entonces, estos patos son blancos
- Simétricamente, si la conclusión es falsa, entonces una o las dos premisas deben ser falsas.
- La deducción es el único método de inferencia que puede probar que una proposición es verdadera.

Inducción

- Inferir una ley general a partir de la observación de casos particulares.
- Aunque puede dar lugar a conclusiones falsas, es necesaria para formular conjeturas y es la clave para hacer progresar a las ciencias.
- Una vez que se ha descubierto por inducción una ley matemática general, debemos demostrarla rigurosamente aplicando el proceso deductivo.

Proceso inductivo - Ejemplo

- Si observamos que:

$$1^3 = 1 = 1^2 = 1^2$$
$$1^3 + 2^3 = 9 = 3^2 = (1 + 2)^2$$
$$1^3 + 2^3 + 3^3 = 36 = 6^2 = (1 + 2 + 3)^2$$
$$1^3 + 2^3 + 3^3 + 4^3 = 100 = 10^2 = (1 + 2 + 3 + 4)^2$$
$$1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225 = 15^2 = (1 + 2 + 3 + 4 + 5)^2$$

- Se puede inducir que:

  “La suma de los cubos de los primeros enteros positivos es siempre un cuadrado perfecto”

  o que:

  “La suma de los cubos de los primeros enteros positivos es el cuadrado de su suma”

Inducción matemática

Caso base:

Existe un entero a tal que

$P(a)$ es cierto.

Hipótesis de inducción:

Para cualquier entero $n \geq a$

$P(n\text{-}1)$ es válido

Conclusión

Entonces, para todos los enteros $n \geq a$

$P(n)$ es cierto

Inducción matemática

- Para demostrar por inducción se debe:
  - Probar la validez del enunciado para el caso base P(a)
  - Suponiendo que la hipótesis de inducción P(n-1) es verdadera se debe demostrar la afirmación para el caso P(n)

Inducción matemática - Ejemplo

Demostrar por inducción matemática que:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Para a = 1

P(1) = 1(1+1) / 2 = 1

Suponiendo que para un n $\geq$ 1, P(n-1) es válido, debemos demostrar para P(n)

P(n) = 1 + 2 + 3 + … + (n-1) + n

= (n-1)((n-1)+1)/2 + n = (n-1)n/2 + n

= n((n-1)/2 + 1) = n(n+1) / 2

Por lo tanto: P(n) = n(n+1) / 2, que era lo que queríamos demostrar.

Demostrar un algoritmo mediante Inducción matemática

Vamos a demostrar que el siguiente algoritmo, devuelve el cuadrado de un número para todo n >= 0

Función cuadrado (n)

Si (n = 0) entonces

retornar **0**

Sino

retornar **2n + cuadrado (n-1) – 1**

Fin si

Fin Función

Demostrar un algoritmo mediante Inducción matemática

- Si probamos con unas cuantas entradas se obtiene:

  cuadrado (0) = 0

  cuadrado (1) = 1

  cuadrado (2) = 4

  cuadrado (3) = 9

  cuadrado (4) = 16

- Por inducción parece evidente que:

  cuadrado (n) = n^2 para todos los n >= 0

- Usaremos la inducción matemática para demostrar la conclusión rigurosamente.

Demostrar un algoritmo mediante Inducción matemática

- Está demostrado que: cuadrado (1) = 1
- Sea cualquier entero n >=1, supóngase que

  cuadrado (n-1) = $(n-1)^2$
- Debemos probar, que para todos los n >= 0

  cuadrado (n) = n^2

  Reemplazando:

  cuadrado (n) = 2n + cuadrado(n-1) – 1

  = $2n + (n-1)^2 – 1$

  = $2n + (n^2 – 2n + 1) – 1 = n^2$
- Por lo tanto:

  cuadrado (n) = n^2, para n >= 0, pues para n = 0 ya se demostró.

Demostrar un algoritmo mediante inducción matemática

- Ahora, para demostrar la corrección (exactitud) del algoritmo por **contradicción**, supongamos que existe al menos un entero positivo en el cual falla el algoritmo.
- Sea n el menor de estos enteros.
- En primer lugar n >=5, pues ya está demostrado cuadrado(n) = n^2 para n < 5.
- En segundo lugar cuadrado(n-1) debe tener éxito, porque en caso contrario **n** no sería el primer entero positivo en el cual falla.
- Pero esto implica que el algoritmo también tiene éxito en **n**, por regla general.

Demostrar un algoritmo mediante inducción matemática

- Lo anterior contradice nuestra suposición acerca de la selección de n.
- Por lo tanto este n no puede existir, lo cual significa que el algoritmo tiene éxito para todos los enteros positivos, inclusive 0 que ya está probado.