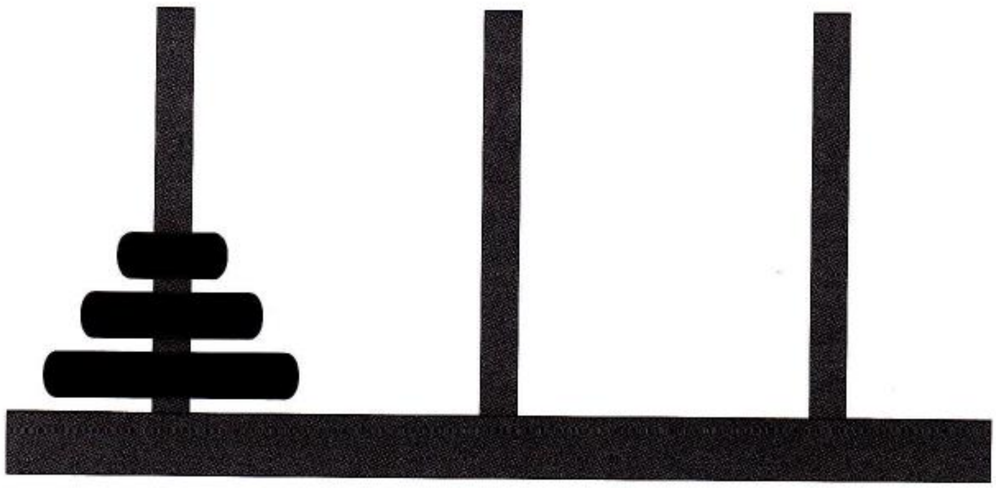
Torres de Hanoi

Se tienen 3 varillas y n discos agujereados por el centro. Los discos son todos de diferente tamaño y en la posición inicial se tienen los discos insertados en el primer palo y ordenados en tamaños decrecientes desde la base hasta la punta de la varilla. El problema consiste en pasar los discos de la 1ª a la 3ª varilla observando las siguientes reglas: se mueven los discos de uno en uno y nunca un disco puede colocarse encima de otro menor que éste.



Solución:

Tamaño umbral: n=1, es decir pasar un único disco.

Descomposición del problema:

- Pasar n-1 disco de la varilla 1 a la varilla 2
- Pasar el disco que queda en la varilla 1, que es el de mayor radio a la varilla 3.
- Pasar los n-1 discos que habíamos colocado en la varilla 2 a la varilla 3.

Combinación: se trata de un problema de reducción.

```
fun Hanoi ( Origen, Destino, Auxiliar: varilla, n:entero)
    si n==1 entonces
        mover (Origen, Destino)
    sino
        Hanoi (Origen, Auxiliar, Destino, n-1)
        mover (Origen, Destino)
        Hanoi (Auxiliar, Destino,Origen, n-1)
    finsi
ffun
```

Hanoi(1, 3, 2, 3)

Hanoi(1,2, 3, 2) → Hanoi(1, 3, 2, 1) → mover(1, 3)

mover(1, 2)

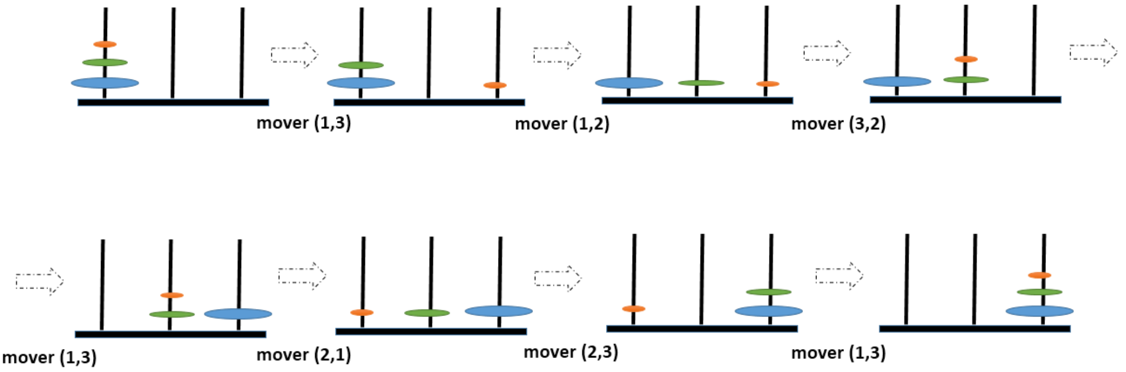
Hanoi(3, 2 , 1 , 1) → mover(3, 2)

Mover((1, 3)

Hanoi(2, 3, 1, 2) → Hanoi(2, 1, 3, 1) → mover(2, 1)

Mover(2, 3)

Hanoi(1, 3 , 2 , 1) → mover(1, 3)



Estudio del coste:

Ecuación de recurrencia

$$T(n) = \begin{cases} cn^k & , \text{si } 1 \leq n < b \\ aT(n-b) + cn^k & , \text{si } n \geq b \end{cases}$$

Con a = 2, b=1 y k= 0 tenemos:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^k) & , \text{si } a < 1 \\ \Theta(n^{k+1}) & , \text{si } a = 1 \\ \Theta(a^{n/b}) & , \text{si } a > 1 \end{cases}$$

Obtenemos que el orden de complejidad es $O(2^n)$

Elemento de un vector igual a su índice

Se tiene un vector de enteros no repetidos y ordenados de menor a mayor. Diseñar un algoritmo que compruebe en tiempo logarítmico si existe algún elemento del vector que coincida con su índice.

Solución:

Se trata de un esquema Divide y Vencerás por dos motivos:

- Se exige un coste logarítmico y esto sólo es posible si el problema se reduce al menos a la mitad en cada paso.
- Se puede descomponer la estructura de datos en partes de su misma naturaleza.

Descartamos un algoritmo Voraz al no haber conjunto de Candidatos.

```
fun  solución-trivial(v:vector; i: natural)

        si v[i] = i entonces dev cierto
        sino dev falso
ffun

fun  índice (v:vector; i, j: natural)
        si i=j entonces solución-trivial(v,i)
        si no  hacer
                K ←(i+j) div 2
                si v[k] =k entonces solucion-trivial(v,k)
                si v[k] >k entonces índice(v,i,k)
                si v[k] <k   entonces índice(v,k+1,j)

ffun
```

Estudio del Coste:

$$T(n) = \begin{cases} cn^k & , \text{si } 1 \leq n < b \\ aT(n/b) + cn^k & , \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^k) & , \text{si } a < b^k \\ \Theta(n^k \log n) & , \text{si } a = b^k \\ \Theta(n^{\log_b a}) & , \text{si } a > b^k \end{cases}$$

Tenemos a=1, b=2 y k=0, la ecuación de recurrencia del algoritmo T(n)= T(n/2)+1 siendo T(n) O(log n).

Calculo de una función exponencial

El tiempo de ejecución de un algoritmo viene dado por T(n) = 2^{n^2}. Encontrar una forma eficiente de calcular T(n), suponiendo que le coste de multiplicar dos enteros es proporcional a su tamaño en representación binaria.

Solución:

Si a^n = $(a^{n/2})^2$ si n es par y $a^n = a \quad (a^{(n-1)/2})^2$ si n es impar, entonces tenemos que

a^n = $(a^{n\ div\ 2})^2\ a^{n\ mod\ 2}$

Elección de un esquema divide y vencerás.

Trivial y solución trivial: tamaño n=1 y la solución trivial devuelve el mismo dato de entrada.

Descomposición: los descomponemos el problema de tamaño n en otro de tamaño n/2.

Combinación: La función de combinación será cuando a=2 2^{n^2} = $(2^{n^2\ div\ 2})^2\ 2^{n^2\ mod\ 2}$

Algoritmo:

```
fun exp (a:entero; n:entero) dev entero
        si n ≤ 1 entonces solución_simple (a, n)
        sino hacer
                p ← n div 2
                r ← n mod 2
                t ← exp(a,p)
                dev combinación(t,r,a)
        fsi
ffun

fun solución_simple (a:entero; n:entero) dev entero
        si n = 1 entonces dev a
        sino dev 1
        fsi
ffun

fun combinación(t:entero; r:entero; a:entero) dev entero
        dev t ∗ t ∗ a^r
ffun

fun T(n:entero) dev entero
        m ← n ∗ n
        dev exp(2, m)
ffun
```

Estudio del coste:

$$T(n) = \begin{cases} cn^k & \text{, si } 1 \le n < b \\ aT(n/b) + cn^k & \text{, si } n \ge b \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{, si } a < b^k \\ \Theta(n^k \log n) & \text{, si } a = b^k \\ \Theta(n^{\log_b a}) & \text{, si } a > b^k \end{cases}$$

Tenemos que para a= 1, b= 2 y k= 1 estamos en el caso de a < b^k. Por lo tanto el orden de complejidad de nuestro algoritmo de exponenciación es Θ(n). Como la cantidad que necesitamos calcular es T(n) = 2^{n^2}, esta operación tendrá un coste $\Theta(n^2)$ en vez de un orden de complejidad $O(n^3)$ que hubiese resultado de realizar n^2 multiplicaciones.