Técnicas de diseño de algoritmos
Divide y Vencerás

Luis Javier Rodríguez Fuentes
Amparo Varona Fernández

Departamento de Electricidad y Electrónica
Facultad de Ciencia y Tecnología, UPV/EHU
luisjavier.rodriguez@ehu.es
amparo.varona@ehu.es

OpenCourseWare 2015
Campus Virtual UPV/EHU

Indice

Soluciones recursivas

- Un algoritmo recursivo expresa la solución a un problema en términos de una o varias llamadas a sí mismo (llamadas recursivas).
- En una solución recursiva, un problema de tamaño n se descompone en uno o varios problemas similares de tamaño $k < n$.
- Cuando el tamaño de los problemas es lo bastante pequeño, estos se resuelven directamente.
- La solución al problema se construye, por tanto, combinando las soluciones a los problemas más pequeños en que se ha descompuesto.
- La prueba de corrección de un algoritmo recursivo equivale a una demostración por inducción:

  1. **Caso base:** el método $\Phi(n)$ funciona cuando $n \leq n_0$
  2. **Hipótesis de inducción:** se supone que el método $\Phi(k)$ funciona $\forall k < n$
  3. **Prueba de inducción:** se demuestra $\Phi(n)$ a partir de $\Phi(k)$, con $k < n$

  Si el caso base resuelve el problema y la expresión general de la solución resuelve el problema a partir de las soluciones a problemas estrictamente más pequeños, entonces el algoritmo es correcto.

Prueba de inducción

Demostrar por inducción que:

$$\sum_{k=0}^{n} z^k = \frac{1 - z^{n+1}}{1 - z} \qquad \forall n \geq 0 \tag{1}$$

1. **Caso base ($n_0 \leq 1$).** Para $n = 0$, es obvio. Para $n = 1$, se tiene por un lado:

$$\sum_{k=0}^{1} z^k = z^0 + z^1 = 1 + z$$

y por otro lado:

$$\frac{1 - z^2}{1 - z} = \frac{(1 + z)(1 - z)}{1 - z} = 1 + z$$

2. **Hipótesis de inducción.** Se supone que la expresión (1) se cumple $\forall k < n$.
3. **Prueba de inducción:**

$$\begin{aligned} \sum_{k=0}^{n} z^k &= \sum_{k=0}^{n-1} z^k + z^n = \frac{1 - z^n}{1 - z} + z^n = \frac{1 - z^n + (1 - z)z^n}{1 - z} \\ &= \frac{1 - z^n + z^n - z^{n+1}}{1 - z} = \frac{1 - z^{n+1}}{1 - z} \end{aligned}$$

Por tanto, hemos demostrado el caso general.

Algoritmos recursivos

- En un algoritmo recursivo **deben aparecer**:
  - **Un caso base**: una solución directa del problema para tamaños pequeños ($n \leq n_0$).
  - **Un caso general**: una solución recursiva basada en la solución de problemas más pequeños ($\Phi(n)$ resuelto como combinación de $\Phi(k)$, con $k < n$).

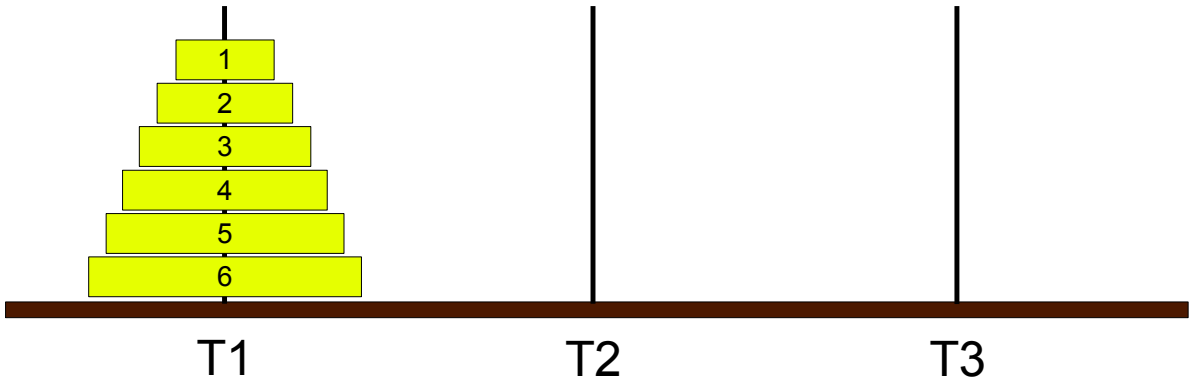
- Ejemplo 1: cálculo del factorial de n

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

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

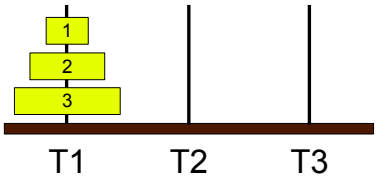
Algoritmos recursivos

Ejemplo 2: las torres de Hanoi

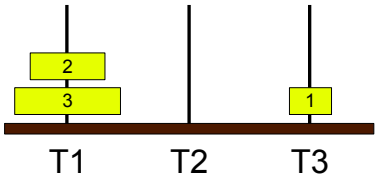


Algoritmos recursivos

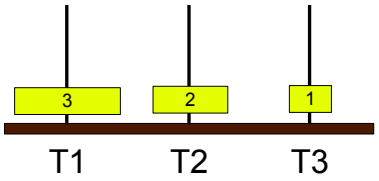
Torres de Hanoi: secuencia de movimientos para $n = 3$



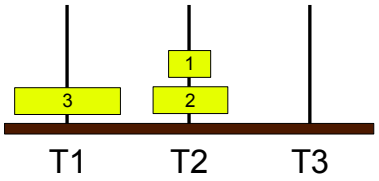
Estado inicial



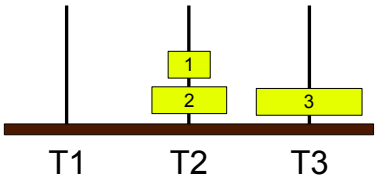
Movimiento 1



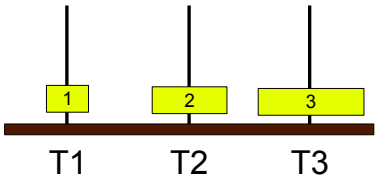
Movimiento 2



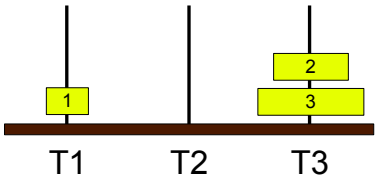
Movimiento 3



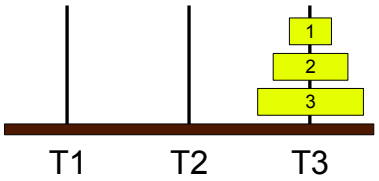
Movimiento 4



Movimiento 5



Movimiento 6



Movimiento 7

Estado final

Algoritmos recursivos

- Código en Python:

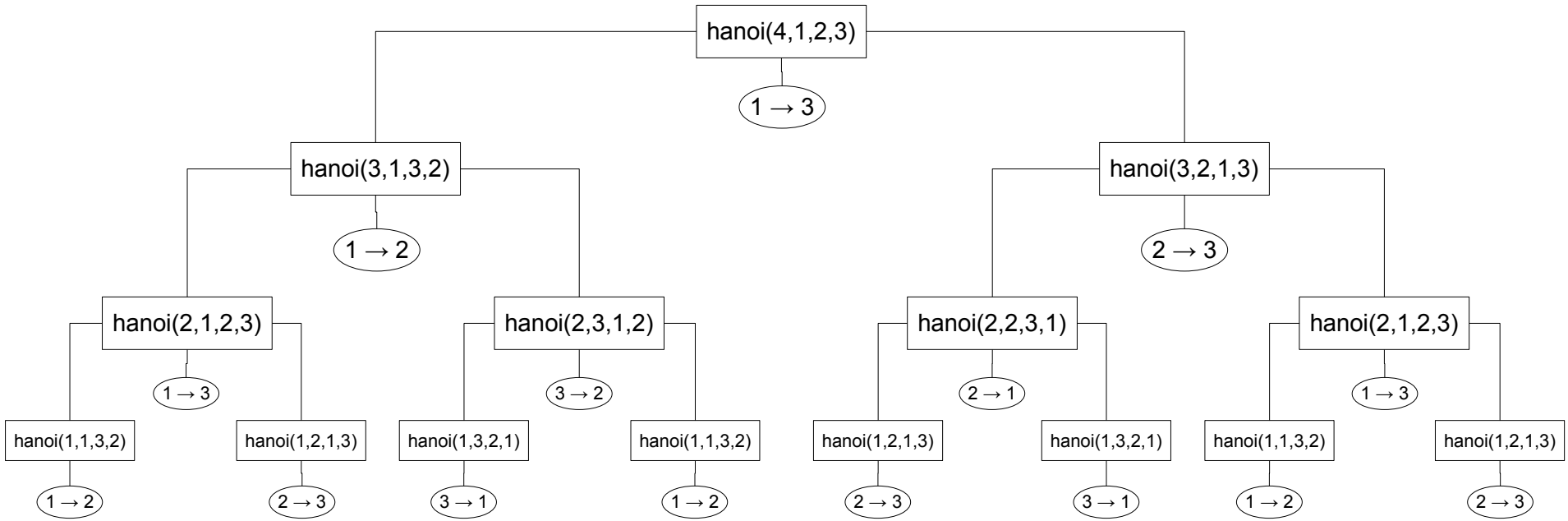
```
def hanoi(n,origen,auxiliar,destino):
    if n==1:
        print(origen," --> ",destino)
    else:
        hanoi(n-1,origen,destino,auxiliar)
        print(origen," --> ",destino)
        hanoi(n-1,auxiliar,origen,destino)
```

- Número de movimientos en función de n: número de piezas

n	#moves
3	7
4	15
5	31
6	63
7	127
8	255

Algoritmos recursivos

Torres de Hanoi: árbol de llamadas para $n = 4$



Complejidad temporal de algoritmos recursivos

- La complejidad temporal de un algoritmo recursivo puede obtenerse resolviendo una ecuación en diferencias. El primer paso consiste en plantear la ecuación, en la que se distinguirán un caso base y un caso general.
- **Ejemplos**
  - Factorial:

$$T(n) = \begin{cases} C_1 & \text{si } n = 0 \\ C_2 + T(n-1) & \text{si } n > 0 \end{cases}$$

  - Torres de Hanoi:

$$T(n) = \begin{cases} C_1 & \text{si } n = 1 \\ C_2 + 2T(n-1) & \text{si } n > 1 \end{cases}$$

- **Métodos de resolución**
  - Expansión de recurrencias
  - Ecuación característica

Complejidad temporal: expansión de recurrencias

- **Expansión de recurrencias:** consiste en ir expandiendo términos de manera sucesiva para *inducir* una expresión general de la recurrencia de grado k, lo que permite llegar al caso base.
- Ejemplo: **Factorial**

$$\begin{aligned} T(n) &= T(n-1) + C_2 = (T(n-2) + C_2) + C_2 \\ &= T(n-2) + 2C_2 = (T(n-3) + C_2) + 2C_2 \\ &= T(n-3) + 3C_2 = (T(n-4) + C_2) + 3C_2 \\ &= T(n-4) + 4C_2 = (T(n-5) + C_2) + 4C_2 \\ &= T(n-k) + kC_2 \end{aligned}$$

En este caso, haciendo $k = n$, llegamos al caso base:

$$T(n) = T(0) + nC_2 = C_1 + nC_2 \quad \in \Theta(n)$$

Complejidad temporal: expansión de recurrencias

- Ejemplo: **Torres de Hanoi**

$$\begin{aligned} T(n) &= 2T(n-1) + C_2 = 2(2T(n-2) + C_2) + C_2 \\ &= 4T(n-2) + 3C_2 = 4(2T(n-3) + C_2) + 3C_2 \\ &= 8T(n-3) + 7C_2 = 8(2T(n-4) + C_2) + 7C_2 \\ &= 16T(n-4) + 15C_2 \ldots \\ &= 2^k T(n-k) + (2^k - 1)C_2 \end{aligned}$$

En este caso, haciendo $k = n - 1$, llegamos al caso base:

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1)C_2 = \left(\frac{C_1 + C_2}{2}\right)2^n - C_2 \quad \in \Theta(2^n)$$

Complejidad temporal: ecuación característica

- **Ecuación característica:** se plantea y resuelve la ecuación característica correspondiente a una recurrencia lineal de coeficientes constantes.
- Se plantean soluciones particulares de la forma $t(n) = x^n$ y se construye una solución general como combinación lineal de soluciones particulares.

$$f(n), g(n) \quad \text{soluciones particulares}$$
$$\Downarrow$$
$$t(n) = c_1 f(n) + c_2 g(n) \quad \text{también es solución}$$

- Las condiciones iniciales se obtienen del caso base.

Complejidad temporal: ecuación característica

Recurrencias lineales homogéneas de coeficientes constantes

$$a_0 t(n) + a_1 t(n-1) + \ldots + a_k t(n-k) = 0$$

$$t(i) = t_i \quad i = 0, \ldots, k-1 \leftarrow \text{condiciones iniciales}$$

$$t(n) = x^n \quad \text{solución} \quad \Rightarrow \quad a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0$$

$$a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0 \qquad \textbf{ECUACIÓN CARACTERÍSTICA}$$

$$P_k(x) = a_0 x^k + a_1 x^{k-1} + \ldots + a_k \qquad \textbf{POLINOMIO CARACTERÍSTICO}$$

Raíces de $P_k(x)$: $\{r_i$ con multiplicidad $m_i \mid i \in [1, s],\ \sum_{i=1}^{s} m_i = k\}$

$$P_k(x) = \prod_{i=1}^{s} (x - r_i)^{m_i}$$

$$t(n) = \sum_{i=1}^{s} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n \quad \leftarrow \quad \textbf{Solución general}$$

$$c_{ij} \leftarrow \text{condiciones iniciales}$$

Complejidad temporal: ecuación característica

- Ejemplo: Serie de Fibonacci

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

$$fib(n) - fib(n-1) - fib(n-2) = 0$$

$$x^2 - x - 1 = 0 \quad \rightarrow \quad r_1 = \frac{1+\sqrt{5}}{2},\ r_2 = \frac{1-\sqrt{5}}{2}$$

$$fib(n) = c_1 r_1^n + c_2 r_2^n$$

$$\left.\begin{array}{l} fib(0) = c_1 + c_2 = 0 \\ fib(1) = c_1 r_1 + c_2 r_2 = 1 \end{array}\right\} \quad \rightarrow \quad c_1 = \frac{1}{\sqrt{5}},\ c_2 = -\frac{1}{\sqrt{5}}$$

$$fib(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Complejidad temporal: ecuación característica

Recurrencias lineales no homogéneas de coeficientes constantes

$$a_0 t(n) + a_1 t(n-1) + \ldots + a_k t(n-k) = \sum_{j=1}^{w} p_j(n) b_j^n$$

donde $b_i \neq b_j \ \forall i \neq j$ y $p_j(n)$: polinomio de grado d_j en n

POLINOMIO CARACTERÍSTICO

$$P(x) = \underbrace{\left(a_0 x^k + a_1 x^{k-1} + \ldots + a_k\right)}_{\text{raíces de la parte homogenea}} \underbrace{\prod_{j=1}^{w} (x - b_j)^{d_j+1}}_{\substack{\text{raíces de la parte} \\ \text{no homogenea}}}$$

$$\Downarrow$$

Solución general

Complejidad temporal: ecuación característica

- Ejemplo de recurrencia no homogenea:

$$t(n) = \begin{cases} 0 & n = 0 \\ 2t(n-1) + n + 2^n & n > 0 \end{cases}$$

$$t(n) - 2t(n-1) = n1^n + 2^n$$

$$P(x) = (x-2) \cdot (x-1)^2 \cdot (x-2) = (x-2)^2 \cdot (x-1)^2$$

$$t(n) = c_1 2^n + c_2 n 2^n + c_3 1^n + c_4 n 1^n$$

Introduciendo la solución general en la recurrencia:

$$c_2 2^n - c_4 n + (2c_4 - c_3) = n + 2^n \quad \rightarrow \quad \begin{cases} c_2 = 1 \\ c_4 = -1 \\ c_3 = -2 \end{cases}$$

Finalmente, aplicando la condición inicial (caso base):

$$t(0) = c_1 + c_3 = 0 \quad \rightarrow \quad c_1 = 2$$

$$t(n) = n2^n + 2 \cdot 2^n - n - 2 \quad \in \Theta\left(n2^n\right)$$

Complejidad temporal: ecuación característica

- Otro ejemplo de recurrencia no homogenea: **Torres de Hanoi**

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

$$t(n) - 2t(n-1) = 1$$

$$P(x) = (x-2) \cdot (x-1)$$

$$t(n) = c_1 2^n + c_2 1^n$$

Introduciendo la solución general en la recurrencia, obtenemos $c_2 = -1$. Finalmente, aplicando la condición inicial (caso base):

$$t(1) = 2c_1 + c_2 = 1 \quad \rightarrow \quad c_1 = 1$$

$$t(n) = 2^n - 1 \quad \in \Theta\left(2^n\right)$$

Complejidad temporal: ecuación característica

- **Cambios de variable.** Muchas recurrencias no se expresan como ecuaciones en diferencias. Sin embargo, se pueden transformar mediante un cambio de variable, resolver la recurrencia resultante e invertir el cambio.
- **Hipótesis:** $T(n)$ continua y monótona no decreciente.
- **Ejemplo:**

$$T(n) = \begin{cases} 1 & n = 1 \\ 3T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Cambio de variable:

$$\left.\begin{array}{l} n = 2^i \quad \Leftrightarrow \quad i = \log_2 n \\ t(i) \equiv T(2^i) \end{array}\right\} \quad \rightarrow \quad t(i) - 3t(i-1) = 2^i$$

$$P(x) = (x-3)\cdot(x-2)$$

$$t(n) = c_1 3^i + c_2 2^i$$

$$T(n) = c_1 3^{\log_2 n} + c_2 n = c_1 n^{\log_2 3} + c_2 n$$

Introduciendo la solución general en la recurrencia obtenemos: $c_2 = -2$, y aplicando la condición inicial (caso base): $c_1 = 3$.

$$T(n) = 3n^{\log_2 3} - 2n \quad \in \Theta\left(n^{\log_2 3}\right)$$

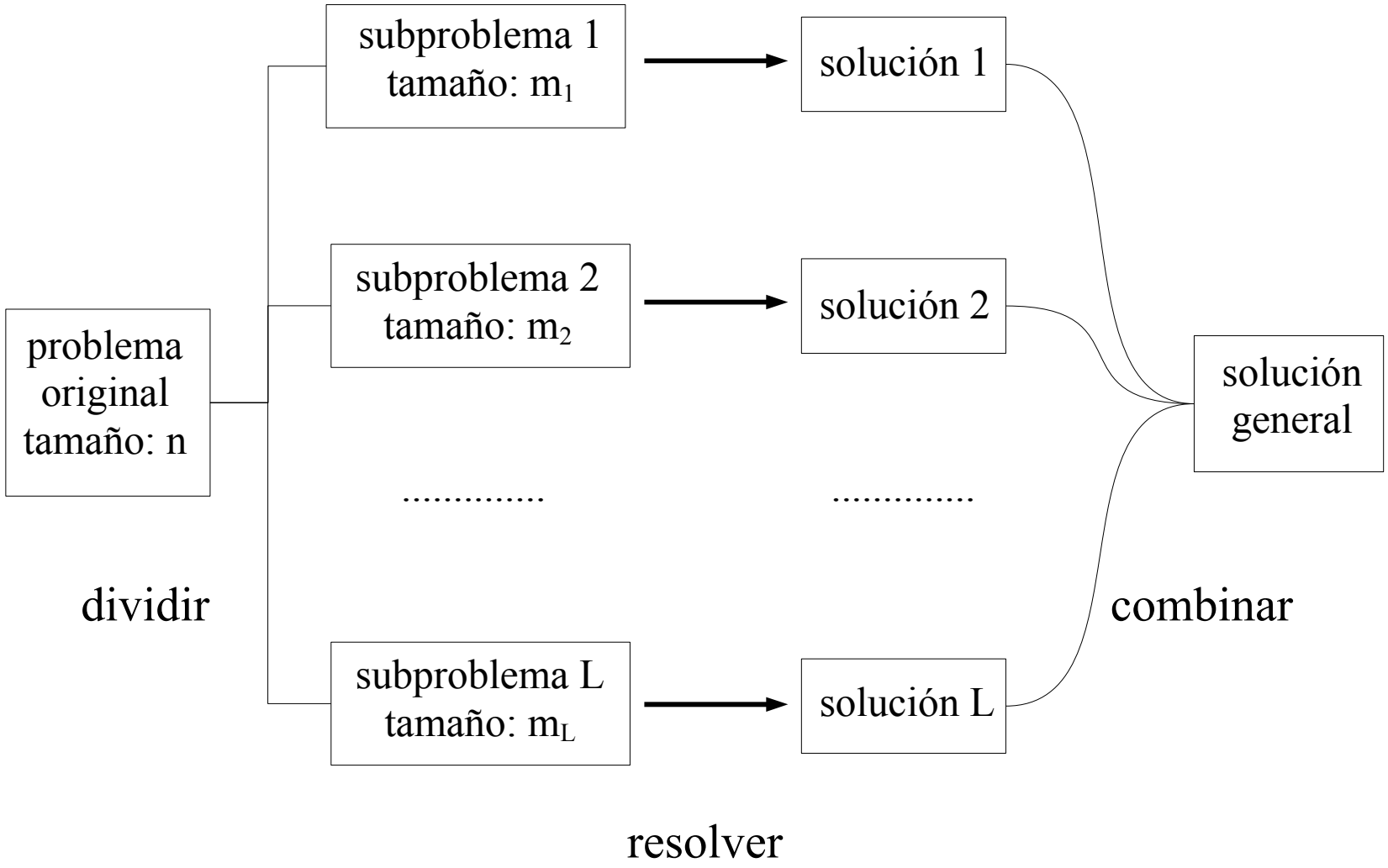
Divide y Vencerás: ideas generales

- El término **Divide y Vencerás**, en su acepción más amplia, es algo más que una técnica de diseño de algoritmos. Se suele considerar una filosofía general para resolver problemas y se utiliza en muchos ámbitos.
- La técnica de diseño de algoritmos *Divide y Vencerás* trata de resolver un problema de forma recursiva, a partir de la solución de subproblemas del mismo tipo pero de menor tamaño.
- A su vez, estos subproblemas se resolverán de la misma forma y así sucesivamente (recuérdese el caso de las torres de Hanoi), hasta alcanzar subproblemas lo bastante pequeños como para que se puedan resolver directamente.

Divide y Vencerás: Esquema

1. El problema original se descompone en L sub-problemas más pequeños:
   - Tamaño del problema original: n
   - Tamaño del sub-problema i: $m_i < n$
   - Típicamente (aunque no siempre) $\sum_{i=1}^{L} m_i \leq n$
2. **Hipótesis:**
   - Resolver un problema más pequeño tiene un coste menor
   - La solución de los casos muy pequeños ($n \leq n_0$) es trivial y tiene coste fijo
3. Los L sub-problemas se resuelven por separado, aplicando el mismo algoritmo
4. La solución al problema original se obtiene combinando las soluciones a los L subproblemas

Divide y Vencerás: Esquema



Divide y Vencerás: Esquema

```
def Divide_y_Venceras(p):
    if EsCasoBase(p):
        s = ResuelveCasoBase(p)
    else:
        lista_subp = Divide(p)
        lista_subs = list()
        for subp in lista_subp:
            subs = Divide_y_Venceras(subp)
            lista_subs.append(subs)
        s = Combina(lista_subs)
    return s
```

- El número de subproblemas (L) en que se divide el problema original debe ser pequeño, independiente de la entrada, y a ser posible balanceado (es preferible dividir n en $(n/2, n/2)$ que en $(n-1, 1)$).
- Cuando el problema original genera un único subproblema, se habla de **algoritmos de simplificación** (p.e., el algoritmo que calcula el factorial).

Divide y Vencerás: ventajas e inconvenientes

El esquema Divide y Vencerás tiene las ventajas e inconvenientes que se derivan de un diseño recursivo:

- **Ventajas:**
  - simple, robusto y elegante
  - buena legibilidad
  - fácil depuración y mantenimiento del código
- **Inconvenientes:**
  - mayor coste espacial por el uso intensivo de la pila del sistema
  - en ocasiones no se reduce, sino que incluso aumenta la complejidad temporal con respecto a soluciones iterativas

Divide y Vencerás: complejidad temporal (1/6)

- **Caso general:**

$$T_{dyv}(n) = \begin{cases} T_{trivial}(n) & n \leq n_0 \\ T_{dividir}(n, L) + \sum_{i=1}^{L} T_{dyv}(m_i) + T_{combinar}(n, L) & n > n_0 \end{cases}$$

- **Caso particular (muy frecuente):**
  - El problema original (de tamaño n) se divide en L subproblemas de tamaño n/b, con $L \geq 1$ y $b \geq 2$.
  - $T_{trivial}(n) \in \Theta(1)$ $(n_0 \geq 1)$
  - $T_{dividir}(n, L) + T_{combinar}(n, L) \in \Theta(n^k)$, con $k \geq 0$

$$T_{dyv}(n) = \begin{cases} 1 & n \leq n_0 \\ LT_{dyv}\left(\frac{n}{b}\right) + n^k & n > n_0 \end{cases}$$

Divide y Vencerás: complejidad temporal (2/6)

- Vamos a resolver la recurrencia para $n \in \{bn_0, b^2 n_0, b^3 n_0, \ldots\}$
- Hipótesis: $T(n)$ continua y monótona no decreciente
- Cambio de variable:

$$n \equiv b^i n_0 \Leftrightarrow i \equiv \log_b(n/n_0)$$

- Nos queda la siguiente expresión:

$$\begin{aligned} t(i) &\equiv T(b^i n_0) \\ &= LT(b^{i-1} n_0) + \left(b^i n_0\right)^k \\ &\equiv Lt(i-1) + n_0^k b^{ik} \end{aligned}$$

- Y la recurrencia queda:

$$t(i) - Lt(i-1) = n_0^k \left(b^k\right)^i$$

- La solución general es:

$$t(i) = \begin{cases} c_1 L^i + c_2 \left(b^k\right)^i & L \neq b^k \\ c_1 \left(b^k\right)^i + c_2 i \left(b^k\right)^i & L = b^k \end{cases}$$

Divide y Vencerás: complejidad temporal (3/6)

$$\boxed{L \neq b^k}$$

- Invirtiendo el cambio nos queda:

$$T(n) = c_1 \left(\frac{n}{n_0}\right)^{\log_b L} + c_2 \left(\frac{n}{n_0}\right)^k = C_1 n^{\log_b L} + C_2 n^k$$

donde:

$$\begin{aligned} C_1 &= c_1 / n_0^{\log_b L} \\ C_2 &= c_2 / n_0^k \end{aligned}$$

- Sustituyendo en la recurrencia original se obtiene:

$$\begin{aligned} n^k &= T(n) - LT(n/b) \\ &= C_1 n^{\log_b L} + C_2 n^k - L\left(C_1 \left(\frac{n}{b}\right)^{\log_b L} + C_2 \left(\frac{n}{b}\right)^k\right) \\ &= \left(1 - \frac{L}{b^k}\right) C_2 n^k \end{aligned}$$

- Y de ahí: $C_2 = \left(1 - \frac{L}{b^k}\right)^{-1}$

Divide y Vencerás: complejidad temporal (4/6)

$$\boxed{L \neq b^k}$$

- Se pueden sacar conclusiones sobre el orden de magnitud de $T(n)$ independientemente del valor de C_1
- La solución general es:

$$T(n) = C_1 n^{\log_b L} + \left(1 - \frac{L}{b^k}\right)^{-1} n^k$$

- $L < b^k$
  - $\left(1 - \frac{L}{b^k}\right)^{-1} > 0$ y $k > \log_b L$
  - Por tanto: $T(n) \in \Theta(n^k) \quad \forall n \in \{bn_0, b^2 n_0, b^3 n_0, \ldots\}$
  - $T(n)$ continua y monótona no decreciente $\Rightarrow \quad \forall n \quad \boxed{T(n) \in \Theta(n^k)}$
- $L > b^k$
  - $\left(1 - \frac{L}{b^k}\right)^{-1} < 0$ y $k < \log_b L$
  - $C_1 > 0$, ya que ha de ser $T(n) > 0 \quad \forall n$
  - $\boxed{T(n) \in \Theta\left(n^{\log_b L}\right)}$

Divide y Vencerás: complejidad temporal (5/6)

$$\boxed{L = b^k}$$

- Recordemos que la solución general en este caso es:

$$t(i) = c_1 \left(b^k\right)^i + c_2 i \left(b^k\right)^i = c_1 \left(b^i\right)^k + c_2 i \left(b^i\right)^k$$

- Invirtiendo el cambio nos queda:

$$T(n) = c_1 \left(\frac{n}{n_0}\right)^k + c_2 \left(\frac{n}{n_0}\right)^k \log_b \left(\frac{n}{n_0}\right) = C_1 n^k + C_2 n^k \log_b \left(\frac{n}{n_0}\right)$$

donde $C_1 = c_1/n_0^k$ y $C_2 = c_2/n_0^k$

- Sustituyendo en la recurrencia original:

$$\begin{aligned} n^k &= T(n) - b^k T(n/b) \\ &= C_1 n^k + C_2 n^k \log_b \left(\frac{n}{n_0}\right) - b^k \left(C_1 \left(\frac{n}{b}\right)^k + C_2 \left(\frac{n}{b}\right)^k \log_b \left(\frac{n}{bn_0}\right)\right) \\ &= C_2 n^k \end{aligned}$$

Divide y Vencerás: complejidad temporal (6/6)

$$\boxed{L = b^k}$$

- De ahí obtenemos que $C_2 = 1$
- La solución nos queda:

$$T(n) = C_1 n^k + n^k \log_b \left(\frac{n}{n_0}\right) = (C_1 - \log_b n_0)\, n^k + n^k \log_b n$$

- Y por tanto, independientemente del valor de C_1, se tiene que:

$$\boxed{T(n) \in \Theta(n^k \log n)}$$

En resumen:

$$\boxed{T(n) \in \begin{cases} \Theta(n^k) & L < b^k \\ \Theta(n^k \log n) & L = b^k \\ \Theta(n^{\log_b L}) & L > b^k \end{cases}}$$

Búsqueda dicotómica

- Si el vector está ordenado, el algoritmo de búsqueda más eficiente es la **búsqueda dicotómica**:

```
def bdicr(v,i,j,x):
    if i>j:
        return None
    m=(i+j)//2
    if x<v[m]:
        return bdicr(v,i,m-1,x)
    if x>v[m]:
        return bdicr(v,m+1,j,x)
    return m
```

- La función se define recursivamente, reduciendo el intervalo de búsqueda a la mitad con cada llamada
- La primera llamada es: `bdicr(v,0,len(v)-1,x)`
- **Mejor caso:** 1 paso/llamada (`x` está justo en la mitad del vector)
- **Peor caso:** $\lfloor log_2(n) \rfloor + 1$ pasos/llamadas (`x` no se encuentra en el vector)

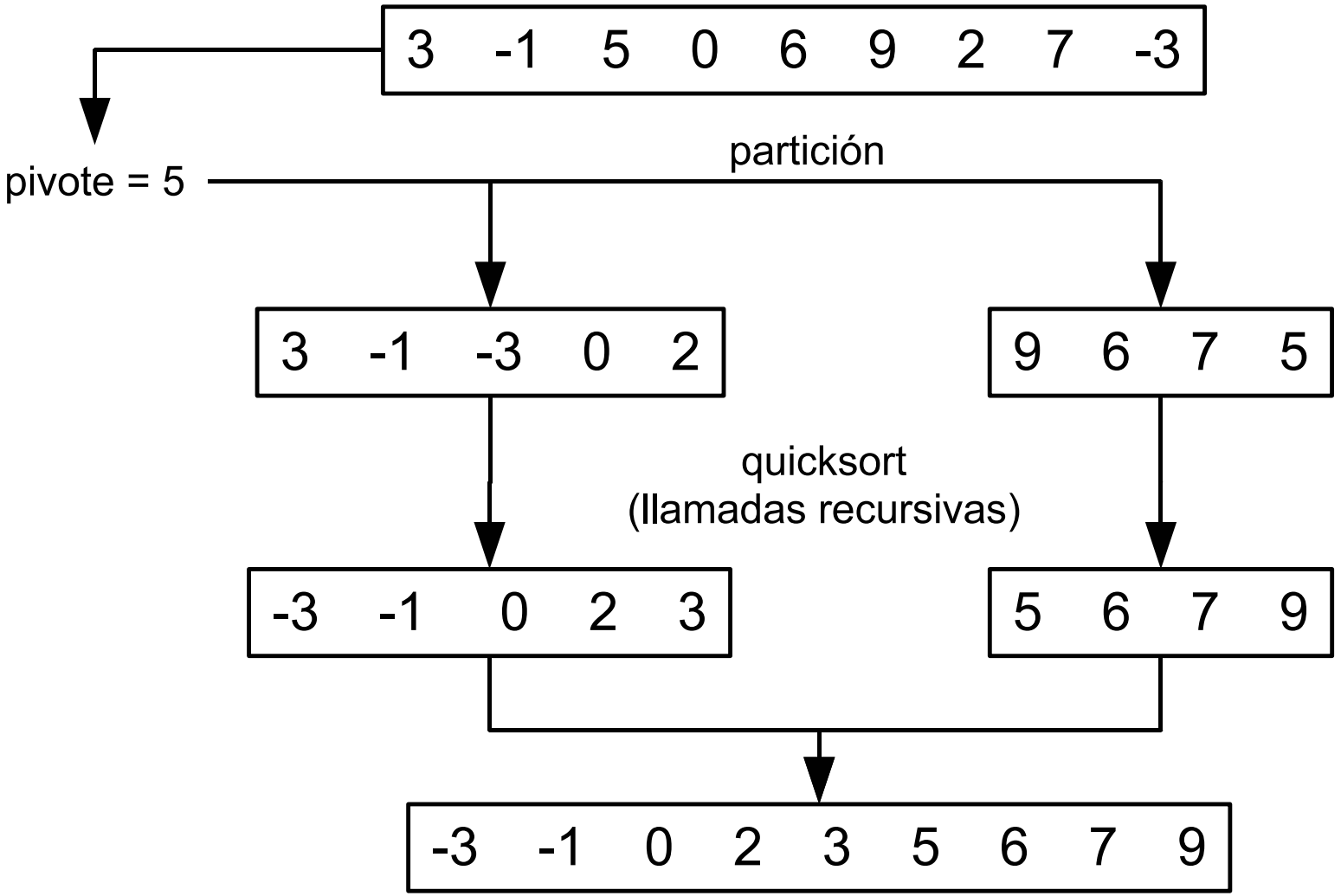
Búsqueda dicotómica

- La búsqueda dicotómica se puede definir también de forma iterativa (ligeramente más eficiente):

```
def bdici(v,x):
    i=0
    j=len(v)-1
    while i<=j:
        m=(i+j)//2
        if x<v[m]:
            j=m-1
        elif x>v[m]:
            i=m+1
        else:
            return m
    return None
```

- El coste temporal viene dado por el número de veces que se repiten las instrucciones del bloque `while` (que, en este caso, constituyen un paso)
- **Mejor caso:** 1 iteración (`x` está justo en la mitad del vector)
- **Peor caso:** $\lfloor log_2(n) \rfloor + 1$ iteraciones (`x` no se encuentra en el vector)

Ordenación por partición: quicksort



Ordenación por partición: quicksort

```
import random

def quicksort(v,izq,der):
    # inicializacion
    i=izq
    j=der
    pivote=v[random.randint(izq,der)]

    # ciclo principal (particion)
    while i<=j:
        while v[i]<pivote:
            i=i+1
        while v[j]>pivote:
            j=j-1
        if i<=j:
            tmp=v[i]
            v[i]=v[j]
            v[j]=tmp
            i=i+1
            j=j-1
    # fin ciclo principal

    # llamadas recursivas:
    # mismo indentado
    # que el while i<=j
    if izq<j:
        quicksort(v,izq,j)
    if i<der:
        quicksort(v,i,der)
```

Ordenación por partición: quicksort

- **Ejemplo:**

```
(0,5) pivote = 0     3  1  7 -1  0  5
                     0  1  7 -1  3  5
                     0 -1  7  1  3  5     j=1, i=2

(0,1) pivote = 0     0 -1  7  1  3  5
                    -1  0  7  1  3  5     j=0, i=1

(2,5) pivote = 5    -1  0  7  1  3  5
                    -1  0  5  1  3  7     j=4, i=5

(2,4) pivote = 3    -1  0  5  1  3  7
                    -1  0  3  1  5  7     j=3, i=4

(2,3) pivote = 1    -1  0  3  1  5  7
                    -1  0  1  3  5  7     j=2, i=3, FIN
```


Ordenación por partición: quicksort

Complejidad temporal

- n = der - izq + 1 (tamaño del intervalo a ordenar)
- Una parte de los pasos se realizan dentro de la propia llamada: inicialización y ciclo principal
- Al tratarse de un algoritmo recursivo, hay que contabilizar también los pasos que se realizan dentro de las llamadas recursivas (que pueden ser dos, una o ninguna)
- $t(n) = 1 + t_{cp}(n) + t(k) + t(n-k)$ (caso general)
- t(1) = 1 (caso base: los tramos de tamaño 1 ya están ordenados)
- Se puede demostrar que el número de incrementos/decrementos, comparaciones e intercambios en el ciclo principal es proporcional a n: $t_{cp}(n) = a \cdot n + b \quad \in \Theta(n)$

Ordenación por partición: quicksort

Complejidad temporal

- Mejor caso:
  - $t_m(n) = n + 2 \cdot t_m(\frac{n}{2})$
  - $t_m(1) = 1$
  - Resolviendo esta recurrencia, se obtiene:

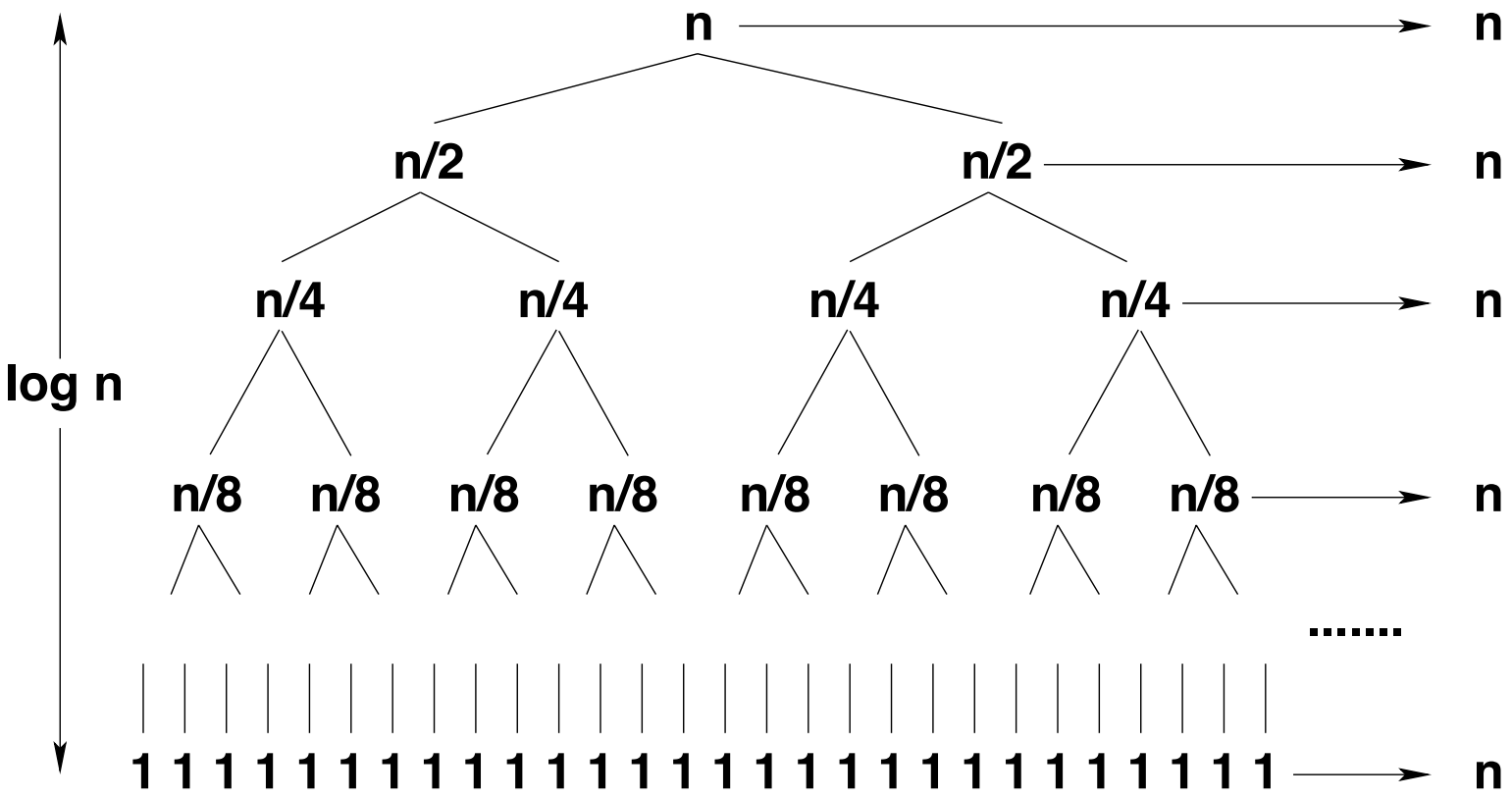
$$t_m(n) = n + n \cdot log_2(n)$$

- Peor caso:
  - $t_p(n) = n + t_p(n-1)$
  - $t_p(1) = 1$
  - Resolviendo esta recurrencia, se obtiene:

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

Ordenación por partición: quicksort

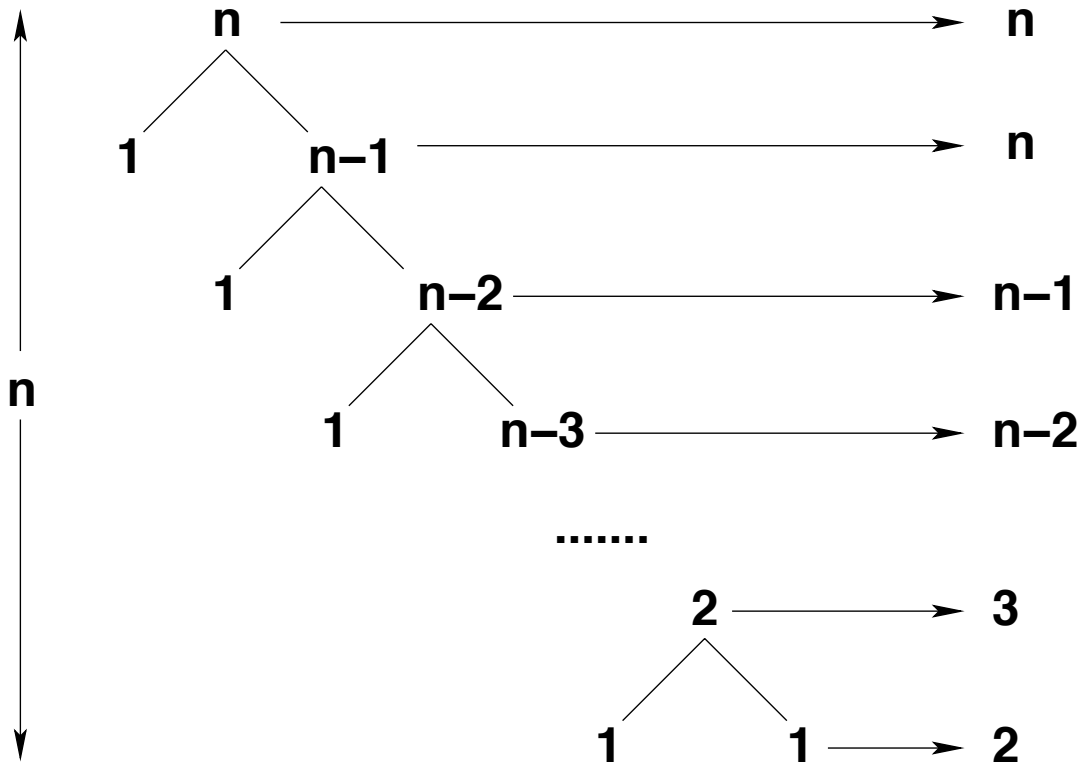
Esquema del mejor caso



$$\Theta(n \log n)$$

Ordenación por partición: quicksort

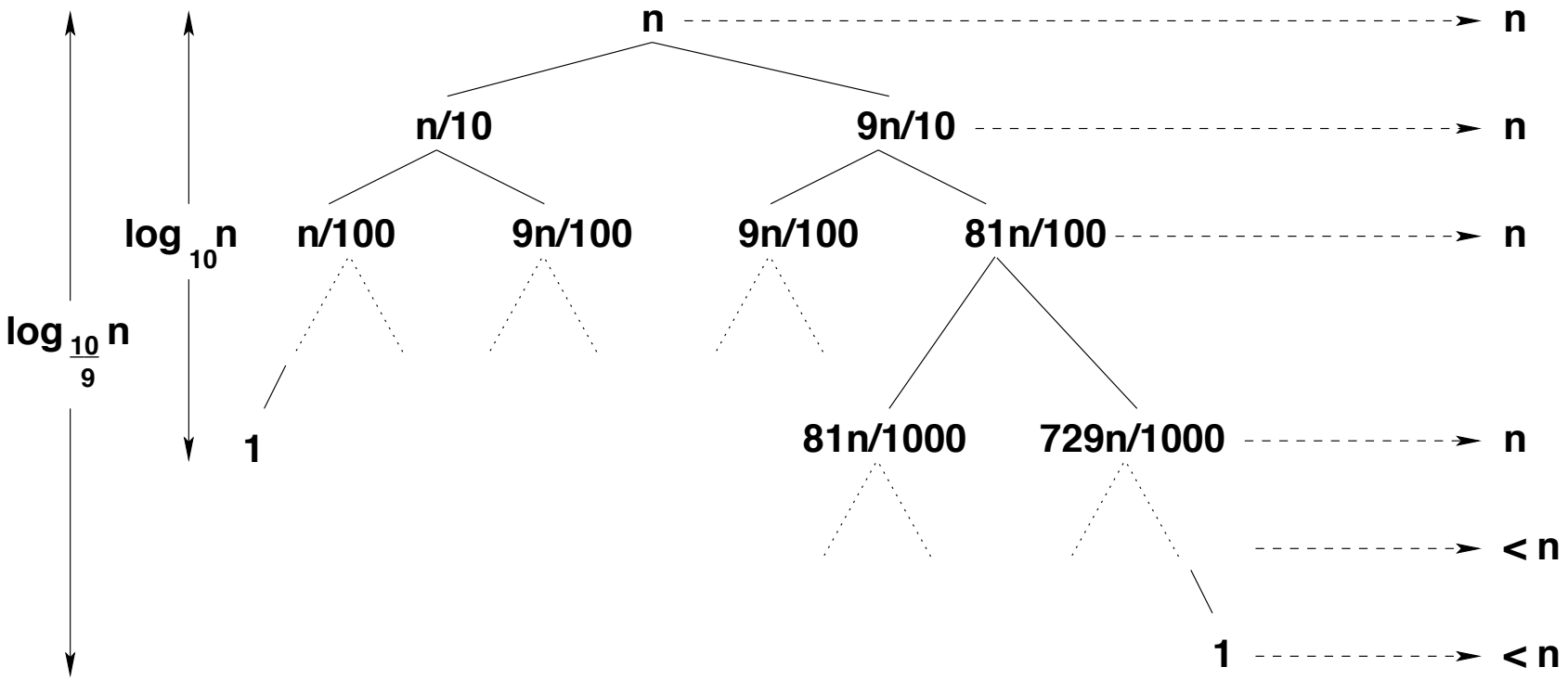
Esquema del peor caso



$$\Sigma_{i=1}^{n}\, i + n - 1 = \frac{1}{2}n^2 + \frac{3}{2}n - 1 \quad \in \quad \Theta(n^2)$$

Ordenación por partición: quicksort

Esquema de un caso muy desequilibrado ($n \to n/10,\ 9n/10$)



$$T(n) < n \log_{\frac{10}{9}} n = n \frac{\log_2 n}{\log_2 \frac{10}{9}} = \frac{1}{\log_2 \frac{10}{9}} n \log_2 n$$

$$T(n) > n \log_{10} n = n \frac{\log_2 n}{\log_2 10} = \frac{1}{\log_2 10} n \log_2 n$$

$$T(n) \in \Theta(n \log n)$$

Ordenación por partición: quicksort

- En resumen, tras analizar los casos mejor y peor, se verifica:

$$n \cdot log_2(n) \leq t_{qs}(n) \leq n^2$$

- No obstante, el coste cuadrático se da sólo en un caso muy extremo. Hemos visto que incluso con particiones muy desequilibradas $(n \to n/10,\ 9n/10)$ el número de pasos es del orden de $n \cdot \log_2(n)$.
- **Caso amortizado**: considera todas las posibles particiones y hace un promedio del número de pasos:

$$t_a(n) \quad = \quad n \ + \ \frac{1}{n} \ \sum_{k=1}^{n-1} (t_a(k) + t_a(n-k))$$

- Se puede demostrar (por inducción) que: $t_a(n) \leq A \cdot n \cdot log_2(n) + B$
- Así pues, también en un caso realista que promedia todas las posibles particiones el quicksort implica del orden de $n \cdot \log_2(n)$ pasos.