Algoritmos y Estructuras de Datos

Tipos de datos secuenciales

Dividir y conquistar

Tipos de datos secuenciales

Dividir y conquistar

Interfaz del tipo vector

Vector	tiempo (peor caso)
▶ Crear un vector de n elementos.	$O(n)$
▶ Determinar el número de elementos.	$O(1)$
▶ Acceder al i-ésimo elemento.	$O(1)$
▶ Modificar el i-ésimo elemento.	$O(1)$

Vectores en Python

```
class Vector:

    def __init__(self, n):
        self._datos = [None for i in range(n)]

    def size(self):
        return len(self._datos)

    def get(self, i):
        return self._datos[i]

    def upd(self, i, x):
        self._datos[i] = x
```

Vectores dinámicos

Los vectores son de **tamaño fijo**.
¿Podríamos tener vectores *dinámicos* (de tamaño variable)?

Problema: extender un vector de tamaño n a tamaño $n+1$.

Método ingenuo

El vector se representa con un arreglo A de tamaño n.
Para extender el vector de tamaño n a tamaño $n+1$:

- Crear un arreglo A' de $n+1$ elementos. $O(n)$
- Copiar los primeros n elementos de A en A'.
  Es decir, poner $A'[i] = A[i]$ para cada $0 \leq i < n$. $O(n)$
- El nuevo arreglo es A' y el nuevo tamaño es $n+1$. $O(1)$

Extremadamente ineficiente

Si extendemos n veces un vector vacío, el costo total es $O(n^2)$.

Vectores dinámicos

Método de crecimiento geométrico

El vector se representa con:

- Un arreglo A de tamaño $K > 0$ (la *capacidad*).
- Un número $0 \leq n \leq K$ (el *tamaño*).
- Los elementos $A[n], \ldots, A[K-1]$ son "espacios disponibles".

Para extender el vector de tamaño n a tamaño $n+1$:

- Si $n < K$, alcanza con incrementar n. $O(1)$
- Si $n = K$:
  - Crear un arreglo A' de tamaño $2K$. $O(K)$
  - Copiar los primeros K elementos de A en A'. $O(K)$
  - El nuevo arreglo es A' y la nueva capacidad es $2K$.
    El nuevo tamaño es $n+1$. $O(1)$

Más eficiente

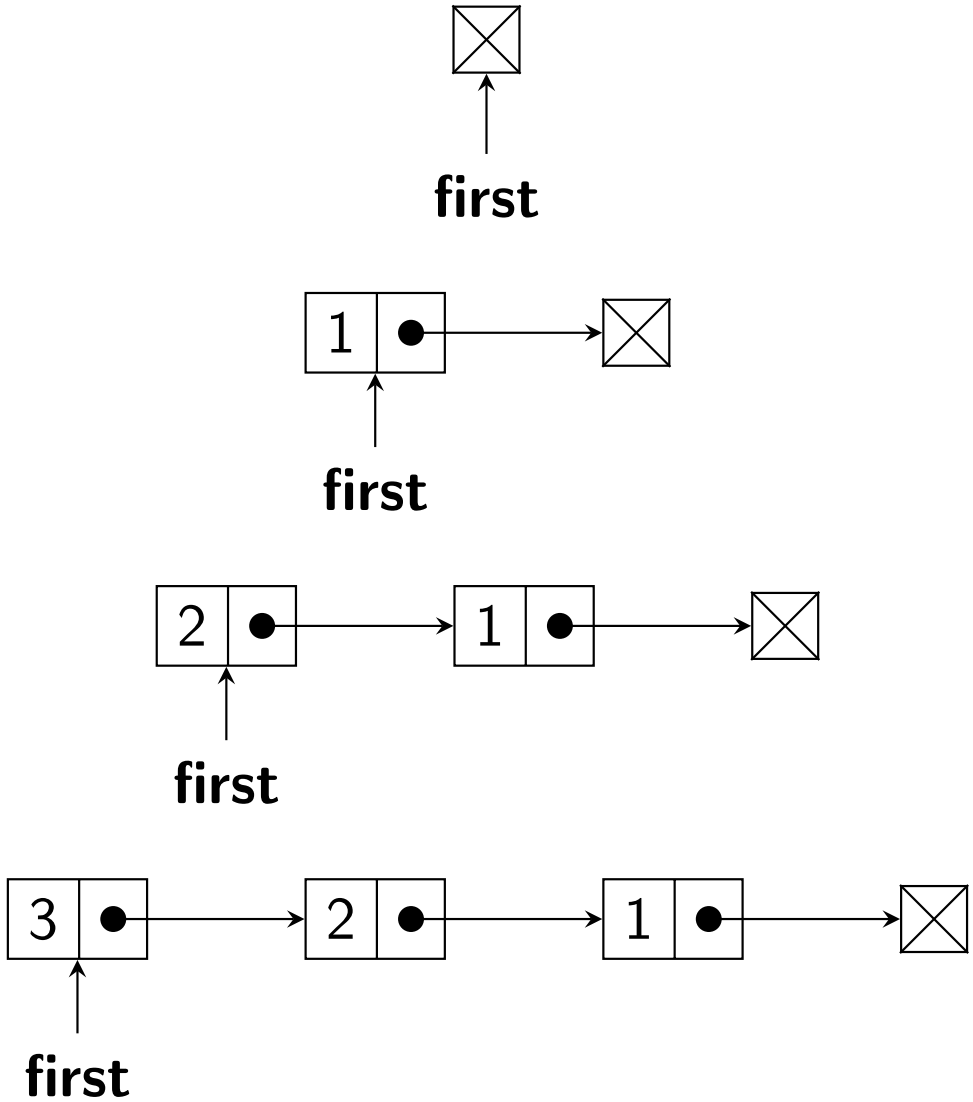
Si extendemos n veces un vector vacío, el costo total es $O(n)$.

(Costo $O(1)$ *amortizado*).

Interfaz del tipo pila

Pila (LIFO)	tiempo (peor caso)
▶ Crear una pila vacía.	$O(1)$
▶ Determinar si la pila está vacía.	$O(1)$
▶ Agregar un elemento en el tope (*push*).	$O(1)$
▶ Ver el elemento que está en el tope.	$O(1)$
▶ Sacar el elemento del tope de la pila (*pop*).	$O(1)$

Representación de pilas usando listas enlazadas



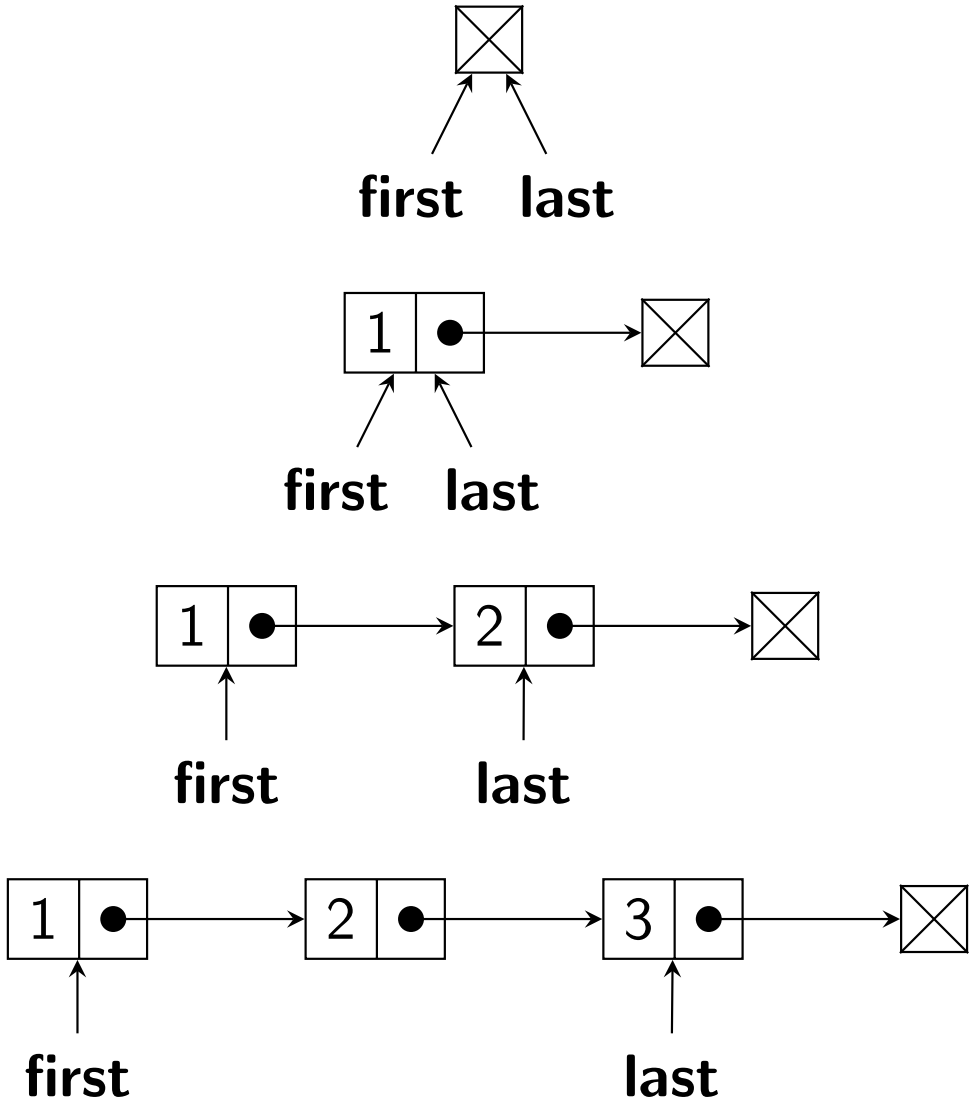
Pilas en Python

```
class Stack:

    def __init__(self):
        self._first = None

    def is_empty(self):
        return self._first is None

    def push(self, x):
        self._first = [x, self._first]

    def top(self, x):
        return self._first[0]

    def pop(self, x):
        self._first = self._first[1]
```

Interfaz del tipo cola

Cola (FIFO)	tiempo (peor caso)
▶ Crear una cola vacía.	$O(1)$
▶ Determinar si la cola está vacía.	$O(1)$
▶ Agregar un elemento al final.	$O(1)$
▶ Ver el próximo elemento.	$O(1)$
▶ Sacar el próximo elemento.	$O(1)$

Representación de colas usando listas enlazadas



Colas en Python

```
class Queue:
    def __init__(self):
        self._first = None
        self._last = None
    def is_empty(self):
        return self._first is None
    def add(self, x):
        if self._last is None:
            self._first = [x, None]
            self._last = self._first
        else:
            self._last[1] = [x, None]
            self._last = self._last[1]
    def first(self):
        return self._first[0]
    def remove(self):
        self._first = self._first[1]
        if self._first is None:
            self._last = None
```

Tipos de datos secuenciales

Dividir y conquistar

Dividir y conquistar

Divide & Conquer es una técnica para diseñar algoritmos.

La técnica es útil cuando las instancias de un problema se pueden dividir en partes más chicas que son instancias del mismo problema.

Dada una entrada $x \in X$, queremos calcular una respuesta $y \in Y$. (De acuerdo con algún contrato).

Idea de la técnica

- Si el dato de entrada $x \in X$ es suficientemente "simple", podemos calcular la respuesta $y \in Y$ de manera directa.
- Si el dato de entrada $x \in X$ **no** es simple:
  - **Dividir** $x \in X$ en partes $x_1, \ldots, x_n \in X$.
  - Resolver cada parte usando el mismo método.
    Para cada $x_i \in X$ el método arroja una salida $y_i \in Y$.
  - **Combinar** $y_1, \ldots, y_n \in Y$ en una única salida $y \in Y$.

Ejemplo de D&C: *mergesort*

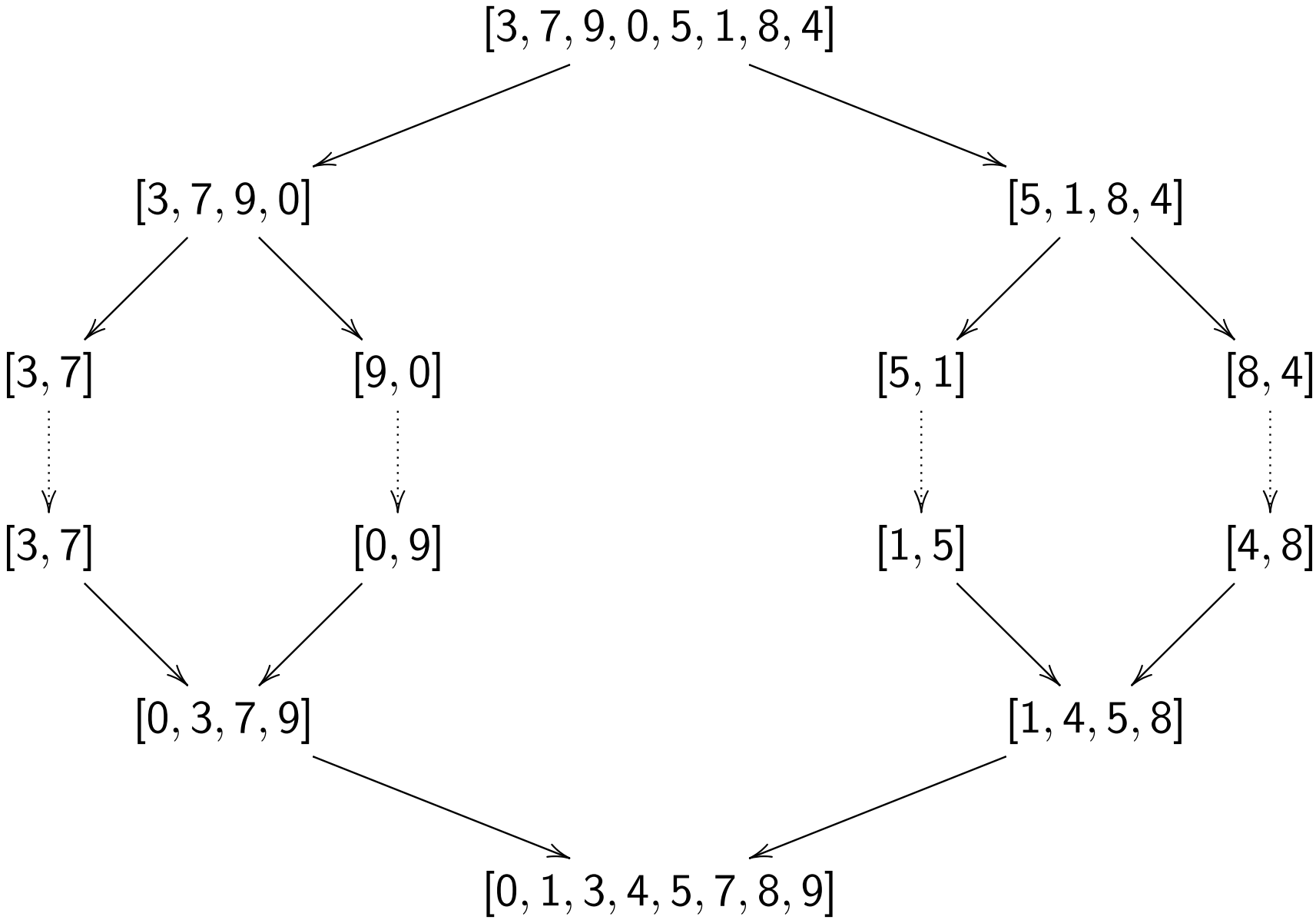
Algoritmo $\textsc{Mergesort}(A)$

Entrada: un arreglo A de elementos. (Con un orden total $<$).
Salida: una permutación ordenada de A.

Si $|A| \leq 1$, el arreglo ya está ordenado. Terminar.
En caso contrario:

- $(B, C) := \textsc{Split}(A)$ — Dividir A en dos mitades B y C.
- $B := \textsc{Mergesort}(B)$ — Ordenar B (recursivamente).
- $C := \textsc{Mergesort}(C)$ — Ordenar C (recursivamente).
- $A := \textsc{Merge}(B, C)$ — Juntar B y C respetando el orden.

Ejemplo de D&C: *mergesort*



Ejemplo de D&C: *mergesort*

Algoritmo $\textsc{Merge}(A, B)$

Entrada: dos arreglos A y B ordenados.
Salida: un arreglo C con todos los elementos de A y B ordenados.

- Inicializar índices $i = 0$ (sobre A) y $j = 0$ (sobre B).
- Mientras $i < |A|$ && $j < |B|$:
  - Si $A[i] < B[j]$, agregar $A[i]$ a la salida y avanzar i.
  - Si no, agregar $B[j]$ a la salida y avanzar j.
- Si $i < |A|$ (sobraron elementos de A): agregarlos a la salida.
- Si $j < |B|$ (sobraron elementos de B): agregarlos a la salida.

Complejidad de MERGESORT

El tiempo de ejecución de MERGESORT(A) en peor caso está dado por una función $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Notamos $n = |A|$.
Asumimos que las comparaciones toman tiempo $O(1)$.

Sabemos que $T(0)$ y $T(1)$ toman tiempo constante. Si $n > 0$:

- $(B, C) :=$ SPLIT(A) ..... $O(n)$
- $B :=$ MERGESORT(B) ..... $T(\frac{n}{2})$
- $C :=$ MERGESORT(C) ..... $T(\frac{n}{2})$
- $A :=$ MERGE(B, C) ..... $O(n)$

Es decir, para ciertas constantes $\mathbf{a}, \mathbf{b} > 0$ tenemos:

$$T(n) \leq \begin{cases} \mathbf{a} & \text{si } n \leq 1 \\ 2\,T(\frac{n}{2}) + \mathbf{b}\,n & \text{si no} \end{cases}$$

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

Complejidad de Mergesort

$$
\begin{array}{rcll}
T(1) & \leq & \mathbf{a} & \\
T(2) & \leq & 2\,T(1) + 2\,\mathbf{b} & \leq \quad 2\,\mathbf{a} + 2\,\mathbf{b} \\
T(4) & \leq & 2\,T(2) + 4\,\mathbf{b} & \leq \quad 4\,\mathbf{a} + (4+4)\,\mathbf{b} \\
T(8) & \leq & 2\,T(4) + 8\,\mathbf{b} & \leq \quad 8\,\mathbf{a} + (8+8+8)\,\mathbf{b} \\
T(16) & \leq & 2\,T(4) + 16\,\mathbf{b} & \leq \quad 16\,\mathbf{a} + (16+16+16+16)\,\mathbf{b} \\
\vdots & & & \\
T(2^t) & \leq & 2^t\,\mathbf{a} + t\,2^t\,\mathbf{b} &
\end{array}
$$

Luego si $n = 2^t$, vale que $t = \log_2(n)$ y tenemos:

$$T(n) = T(2^t) \leq 2^t\mathbf{a} + t2^t\mathbf{b} = n \cdot \mathbf{a} + n\log_2(n) \cdot \mathbf{b} \in O(n \log n)$$

Ejemplo de D&C: búsqueda binaria

Algoritmo BinSearch(A, x)

Entrada: Un arreglo ordenado A de n elementos y un elemento x.
Salida: El mínimo índice $0 \leq k < n$ tal que $A[k] > x$. Si no hay ningún índice que verifique $A[k] > x$, devuelve n.

Si $|A| = 0$ o si $A[0] > x$, devolver 0.
Inicializar $i := 0$; $j := n$.
Mientras $i + 1 < j$:

- Sea $m = \lfloor \frac{i+j}{2} \rfloor$.
- Si $A[m] \leq x$, actualizar $i := m$.
- Si $A[m] > x$, actualizar $j := m$.

¿Cuál es el invariante?

En todas las iteraciones vale la condición $A[i] \leq x < A[j]$.

Complejidad de BinSearch

En cada paso la distancia $j - i$ se ahica a la mitad del tamaño.

Es decir, para ciertas constantes $\mathbf{a}, \mathbf{b} > 0$ tenemos:

$$T(n) \leq \begin{cases} \mathbf{a} & \text{si } n \leq 1 \\ T(\frac{n}{2}) + \mathbf{b} & \text{si no} \end{cases}$$

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

$T(1) \leq \mathbf{a} \quad T(2) \leq \mathbf{a}+\mathbf{b} \quad T(4) \leq \mathbf{a}+2\,\mathbf{b} \quad T(8) \leq \mathbf{a}+3\,\mathbf{b} \quad \ldots$

En general, $T(2^t) \leq \mathbf{a} + t\,\mathbf{b}$.
Luego si $n = 2^t$, vale que $t = \log_2(n)$ y tenemos:

$$T(n) = T(2^t) \leq \mathbf{a} + t\mathbf{b} = \mathbf{a} + \log_2(n) \cdot \mathbf{b} \in O(\log n)$$

Una aplicación de la búsqueda binaria

Ejemplo: cálculo de raíces cuadradas enteras

Entrada: un entero $n \geq 0$.

Salida: el entero más grande r tal que $r^2 \leq n$.

```
def raiz_cuadrada(n):
    i = 0
    j = n + 1
    while i + 1 < j:
        m = (i + j) // 2
        if m * m <= n:
            i = m
        else:
            j = m
    return i
```

Problema de D&C: subarreglo de suma máxima

Problema: dado un arreglo A de n enteros (positivos o negativos), encontrar el máximo número que se puede conseguir sumando elementos consecutivos de A.

Ejemplo

$$[2, -5, \underbrace{1, 1, 1, 1, -1, 3}_{6}, -1]$$

Método ingenuo

- Recorrer todos los subarreglos de A (anidando dos `for`).
  Un subarreglo de A está dado por dos índices $0 \leq i \leq j \leq n$.
- Calcular la suma de cada subarreglo y determinar el máximo.

Complejidad: $O(n^3)$.
Ejercicio: resolverlo en $O(n \log n)$.

Solución de ecuaciones de recurrencia

Notación

Notamos $f \in \Omega(g)$ si $g \in O(f)$.
Notamos $f \in \Theta(g)$ si $f \in O(g)$ y $g \in O(f)$.

El "teorema maestro" (*cf.* Teorema 4.1 de Cormen *et al.*)

Sea T dada por $T(n) = aT(\frac{n}{b}) + f(n)$ para $a \geq 1$, $b > 1$.
Sea $c = \log_b a$.

1. Si $f \in O(n^{c-\epsilon})$ para $\epsilon > 0$, entonces $T(n) \in \Theta(n^c)$.
2. Si $f = \Theta(n^c)$, entonces $T(n) \in \Theta(n^c \log n)$.
3. Si $f \in \Omega(n^{c+\epsilon})$ para $\epsilon > 0$ y vale $af(\frac{n}{b}) \leq cf(n)$ para cierta constante $c < 1$ y todo n suficientemente grande, entonces $T(n) \in \Theta(f(n))$.