# Algoritmos y Estructuras de Datos II

## Recurrencias Divide y Vencerás

21 de marzo de 2016

# Contenidos

# Recurrencias

- Surgen al analizar algoritmos recursivos, como la ordenación por intercalación.
- El conteo de operaciones “copia” la recursión del algoritmo y se vuelve recursivo también.
- Ejemplo: conteo de comparaciones de la ordenación por intercalación.
- Es un ejemplo de algoritmo divide y vencerás.
- Es un ejemplo de recurrencia divide y vencerás:

$$t(n) = \begin{cases} 0 & \text{si } n \in \{0, 1\} \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n - 1 & \text{si } n > 1 \end{cases}$$

# Algoritmo divide y vencerás
## Características

- hay una solución para los casos sencillos,
- para los complejos, se **divide** o **descompone** el problema en subproblemas:
  - cada subproblema es de igual naturaleza que el original,
  - el tamaño del subproblema es una **fracción** del original,
  - se resuelven los subproblemas apelando al mismo algoritmo,
- se **combinan** esas soluciones para obtener una solución del original.

# Algoritmo divide y vencerás
## Forma general

**fun** DV($x$) **ret** $y$
    **if** $x$ suficientemente pequeño o simple **then** $y$:= ad_hoc($x$)
    **else** descomponer $x$ en $x_1, x_2, \ldots, x_a$
        **for** i:= 1 **to** a **do** $y_i$:= DV($x_i$) **od**
        combinar $y_1, y_2, \ldots, y_a$ para obtener la solución $y$ de $x$
    **fi**
**end**

Normalmente los $x_i$ son **fracciones** de $x$:

$$|x_i| = \frac{|x|}{b}$$

para algún $b$ fijo.

# Algoritmo divide y vencerás
## Ejemplos

- Ordenación por intercalación:
  - "$x$ simple" = fragmento de arreglo de longitud 0 ó 1
  - "descomponer" = partir al medio ($b = 2$)
  - $a = 2$
  - "combinar" = intercalar
- Ordenación rápida:
  - "$x$ simple" = fragmento de arreglo de longitud 0 ó 1
  - "descomponer" = separar los menores de los mayores ($b = 2$)
  - $a = 2$
  - "combinar" = yuxtaponer

# Algoritmo divide y vencerás
## Conteo

Si queremos contar el costo computacional (número de operaciones) $t(n)$ de la función *DV* obtenemos:

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

si $c$ es una constante que representa el costo computacional de la función ad_hoc y $g(n)$ es el costo computacional de los procesos de descomposición y de combinación.
Esta definición de $t(n)$ es recursiva (como el algoritmo *DV*), se llama **recurrencia**. Existen distintos tipos de recurrencia. Ésta se llama **recurrencia divide y vencerás**.

# Recurrencias divide y vencerás

Por la forma de la recurrencia

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

resulta más sencillo calcular $t(n)$ cuando $n$ es potencia de $b$.
Se organiza la tarea en dos partes

- calcular el orden de $t(n)$ cuando $n$ es potencia de $b$,
- extender el cálculo para los demás $n$.

# Calculando para $n$ potencia de $b$

- Supongamos que
  - $$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$
  - y $g(n)$ es del orden de $n^k$, es decir $g(n) \leq dn^k$ para casi todo $n \in \mathbb{N}$
- $n$ potencia de $b$, $n = b^m$
- $$\begin{aligned} t(n) &= t(b^m) \\ &= a * t(b^m/b) + g(b^m) \\ &\leq a * t(b^{m-1}) + d(b^m)^k \\ &\leq a * t(b^{m-1}) + d(b^k)^m \end{aligned}$$

# Iterando

$$
\begin{aligned}
t(b^m) &\leq at(b^{m-1}) + d(b^k)^m \\
&\leq a(t(b^{m-2}) + d(b^k)^{m-1}) + d(b^k)^m \\
&\leq a^2 t(b^{m-2}) + ad(b^k)^{m-1} + d(b^k)^m \\
&\leq a^3 t(b^{m-3}) + a^2 d(b^k)^{m-2} + ad(b^k)^{m-1} + d(b^k)^m \\
&\leq \ldots \\
&\leq a^m t(1) + a^{m-1} db^k + \ldots + ad(b^k)^{m-1} + d(b^k)^m \\
&= a^m c + d(b^k)^m((a/b^k)^{m-1} + \ldots + a/b^k + 1) \\
&= a^m c + d(b^m)^k(r^{m-1} + \ldots + r + 1)
\end{aligned}
$$

donde $r = a/b^k$

$$
\begin{aligned}
t(n) &\leq a^{\log_b n} c + dn^k(r^{m-1} + \ldots + r + 1) \\
&= n^{\log_b a} c + dn^k(r^{m-1} + \ldots + r + 1)
\end{aligned}
$$

# Propiedad del logaritmo

En el último paso hemos usado que $x^{\log_y z}$ es igual a $z^{\log_y x}$.

- En efecto, si aplicamos $\log_y$ a ambos, obtenemos
- $\log_y(x^{\log_y z})$ y $\log_y(z^{\log_y x})$, que luego de simplificar quedan
- $(\log_y x)(\log_y z)$ y $(\log_y z)(\log_y x)$ que son iguales.
- Como $\log_y$ es inyectiva, $x^{\log_y z} = z^{\log_y x}$ vale.

Volvamos a los cálculos.

# Finalizando

$$t(n) \leq n^{\log_b a} c + dn^k (r^{m-1} + \ldots + r + 1)$$

donde $r = a/b^k$

- si $r = 1$, entonces $a = b^k$ y $\log_b a = k$ y además

$$\begin{aligned} t(n) &\leq n^{\log_b a} c + dn^k m \\ &= n^k c + dn^k \log_b n \\ &\quad \text{es del orden de } n^k \log n \text{ para } n \text{ potencia de } b \end{aligned}$$

- si $r \neq 1$, entonces

$$t(n) \leq n^{\log_b a} c + dn^k \left(\frac{r^m - 1}{r - 1}\right)$$

# Finalizando
## caso $r \neq 1$

- si $r > 1$, como $r = a/b^k$ entonces $a > b^k$ y $\log_b a > k$ y además

$$\begin{aligned}
t(n) &\leq n^{\log_b a} c + d n^k \left(\frac{r^m - 1}{r-1}\right) \\
&\leq n^{\log_b a} c + \frac{d}{r-1} n^k r^m \\
&= n^{\log_b a} c + \frac{d}{r-1} n^k \frac{a^m}{(b^k)^m} \\
&= n^{\log_b a} c + \frac{d}{r-1} n^k \frac{a^m}{(b^m)^k} \\
&= n^{\log_b a} c + \frac{d}{r-1} n^k \frac{a^m}{n^k} \\
&= n^{\log_b a} c + \frac{d}{r-1} a^m \\
&= n^{\log_b a} c + \frac{d}{r-1} a^{\log_b n} \\
&= n^{\log_b a} c + \frac{d}{r-1} n^{\log_b a}
\end{aligned}$$

es del orden de $n^{\log_b a}$ para $n$ potencia de $b$

# Finalizando
## caso $r < 1$

- si $r < 1$, como $r = a/b^k$ entonces $a < b^k$ y $\log_b a < k$. Además, $r - 1$ y $r^m - 1$ son negativos, para evitar confusión escribimos $\frac{1-r^m}{1-r}$ en vez de $\frac{r^m-1}{r-1}$.

$$\begin{aligned} t(n) &\leq n^{\log_b a} c + dn^k \left(\tfrac{1-r^m}{1-r}\right) \\ &= n^{\log_b a} c + \tfrac{d}{1-r} n^k (1 - r^m) \\ &\leq n^{\log_b a} c + \tfrac{d}{1-r} n^k \\ &\quad \text{es del orden de } n^k \text{ para } n \text{ potencia de } b \end{aligned}$$

# Conclusión

- Para
  - $$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$
  - con $g(n)$ del orden de $n^k$,
  - demostramos

$$t(n) \text{ es del orden de } \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

para $n$ potencia de $b$.

# Extendiendo el resultado a todo $n$

- Hemos calculado el orden de cualquier algoritmo divide y vencerás, para $n$ potencia de $b$.
- Queremos calcular el orden para todo $n$.
- Se puede comprobar que si
  - $t(n)$ es no decreciente, y
  - $t(n)$ es del orden de $h(n)$ para potencias de $b$ para cualquiera de las tres funciones $h(n)$ que acabamos de considerar,

  entonces $t(n)$ es del orden de $h(n)$ (para $n$ arbitrario, no solamente las potencias de $b$).
- es decir, el resultado puede extenderse para $n$ arbitrario.

# Recurrencias divide y vencerás

- Para
  - $$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

    si $t(n)$ es no decreciente, y $g(n)$ es del orden de $n^k$, entonces
  - $$t(n) \text{ es del orden de } \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

# Ejemplo: búsqueda binaria

```
{Pre: 1 ≤ izq ≤ n+1 ∧ 0 ≤ der ≤ n ∧ a ordenado}
fun binary_search_rec (a: array[1..n] of T, x:T, izq, der : nat) ret i:na
    var med: int
    if izq > der → i = 0
      izq ≤ der → med:= (izq+der) ÷ 2
            if x < a[med] → i:= binary_search_rec(a, x, izq, med-1)
              x = a[med] → i:= med
              x > a[med] → i:= binary_search_rec(a, x, med+1,der)
            fi
    fi
end fun
{Post: (i = 0 ⇒ x no está en a[izq,der]) ∧ (i ≠ 0 ⇒ x = a[i])}
```

# Búsqueda binaria
## Función principal

{Pre: n $\geq$ 0 }
**fun** binary_search (a: **array**[1..n] **of T**, x:T) **ret** i:**nat**
    i:= binary_search_rec(a, x, 1, n)
**end fun**
{Post: (i $=$ 0 $\Rightarrow$ x no está en a) $\wedge$ (i $\neq$ 0 $\Rightarrow$ x = a[i])}

# Búsqueda binaria
## Análisis

- Sea $t(n)$ = número de comparaciones que hace en el peor caso cuando el arreglo tiene $n$ celdas.
- $$t(n) = \begin{cases} 0 & \text{si } n = 0 \\ t(n/2) + 1 & \text{si } n > 0 \end{cases}$$
- $a = 1$, $b = 2$ y $k = 0$.
- $a = b^k$.
- $t(n)$ es del orden de $n^k \log n$, es decir, del orden de $\log n$.