TEMA 8:
DIVIDE Y VENCERÁS

COMPUTABILIDAD Y ALGORITMIA

M. Colebrook Santamaría
J. Riera Ledesma
J. Hernández Aceituno

Objetivos

- Intro
- Esquema general
- Teorema maestro
- Búsqueda binaria
- Ordenación por fusión (*mergesort*)

Introducción

- **Divide y Vencerás** (*Divide & Conquer*) es una técnica para diseñar algoritmos que consiste en:
  - Descomponer el problema en un cierto número de **subproblemas** más pequeños,
  - **Resolver** de forma sucesiva e independiente cada uno de esos subproblemas, y
  - **Combinar** después todas las **soluciones obtenidas** para generar la **solución del problema original**.
- Ejemplos: Multiplicación de enteros muy grandes, búsqueda binaria, ordenación por fusión.

Esquema general de un algoritmo divide y vencerás (1)

- Consideremos un problema arbitrario, y sea *ad hoc* (del latín, significa "*para esto*", se refiere a una solución específicamente elaborada para un problema) un ***subalgoritmo básico*** sencillo capaz de resolver el problema de forma **eficiente** para casos pequeños.
- A continuación se muestra el pseudocódigo de los algoritmos tipo DyV.
- Sin embargo, hay que comentar que algunos algoritmos de tipo DyV **no** siguen exactamente este esquema. Por ejemplo, puede necesitar que el primer subproblema esté resuelto antes de formular el segundo subproblema.

Esquema general de un algoritmo divide y vencerás (2)

```
función DyV(x: problema de tamaño n) {
  si x es suficientemente pequeño o sencillo entonces
    Resolver x con el subalgoritmo ad_hoc(x)
  en otro caso {
    Descomponer x en p subproblemas más pequeños:
                    x_1, x_2, …, x_p
    para i <- 1 hasta p hacer
      y_i <- DyV(x_i)
    Recombinar los y_i para obtener una solución y de x
    devolver y
} }
```

Análisis de los algoritmos DyV (1)

- Para que el enfoque DyV sea apropiado, es necesario que se cumplan tres condiciones:
  a. La decisión de utilizar el **subalgoritmo básico** en lugar de hacer llamadas recursivas debe tomarse cuidadosamente.
  b. **Descomponer** el problema en **subproblemas** y **recomponer las soluciones parciales** debe ser suficientemente eficiente.
  c. Los subproblemas deben ser en lo posible **aproximadamente del mismo tamaño**.
- La mayoría de los algoritmos de DyV con un **tamaño de problema** original igual a ***n*** se dividen en ***p*** **subproblemas** de **tamaño *n/d***, para alguna **constante *d***.
- La ventaja de este enfoque es que el análisis de los algoritmos DyV es casi automático.

Análisis de los algoritmos DyV (2)

- Sea $f(n) = \Theta(n^k)$ el tiempo requerido por un algoritmo DyV para **descomponer** y **recomponer** problemas de tamaño n, sin contar el tiempo necesario para las llamadas recursivas.
- El tiempo total para cada iteración (salvo para el caso básico) *es* $T(n) = \boldsymbol{p} \cdot T(n / \boldsymbol{d}) + n^{\boldsymbol{k}}$, donde:

  n: tamaño del problema en cada iteración (salvo caso base)
  p: número de subproblemas en que se subdivide el problema
  d: factor de reducción de los subproblemas
  k: exponente que define la tasa de crecimiento $\Theta(n^k)$ de la descomposición y recomposición

- Obtenemos el **Teorema Maestro**:

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \boldsymbol{p} < \boldsymbol{d}^{\boldsymbol{k}} \\ \Theta(n^k \cdot \log(n)) & \text{si } \boldsymbol{p} = \boldsymbol{d}^{\boldsymbol{k}} \\ \Theta(n^{\log_{\boldsymbol{d}}(\boldsymbol{p})}) & \text{si } \boldsymbol{p} > \boldsymbol{d}^{\boldsymbol{k}} \end{cases}$$

Búsqueda binaria

- La búsqueda binaria es el algoritmo que usamos de forma natural para buscar un elemento en un conjunto ordenado.
- Es la **aplicación más sencilla de DyV**: todo problema suficientemente grande se reduce a un subproblema más pequeño, en este caso, a la **mitad del tamaño original**.
- En el Tema 6 calculamos la complejidad del algoritmo de la búsqueda binaria: $T(n) = \Theta(\log n)$
- En esta sección, veremos una versión recursiva.

Algoritmo recursivo de la búsqueda binaria

```
función búsqueda_binaria(x, A[1..n]): posición del valor {
  si n = 0 o x < A[1] o x > A[n] entonces
    devolver 0
  en otro caso
    devolver bb_recursiva(x, A[1..n])
}

función bb_recursiva(x, A[i..j]): posición del valor {
  si i = j entonces devolver i
  p <- (i + j) / 2
  si x ≤ A[p] entonces devolver bb_recursiva(x, A[i..p])
  en otro caso         devolver bb_recursiva(x, A[p+1..j])
}
```

Ejemplo

Búsqueda del valor **x=12** en el siguiente array **A[1..11]**:

1	2	3	4	5	6	7	8	9	10	11	
-5	-2	0	3	8	8	9	12	12	26	31	
[i]					**[p]**					**[j]**	**x > A[p]**
						[i]		**[p]**		**[j]**	**x ≤ A[p]**
						[i]	**[p]**	**[j]**			**x ≤ A[p]**
						p=i	**[j]**				**x > A[p]**
							i=j				STOP

Análisis del algoritmo de la búsqueda binaria recursiva

- Usando el Teorema Maestro, podemos observar que:
  - El problema se subdivide en un único subproblema por cada iteración. Por tanto, $\boldsymbol{p} = \mathbf{1}$.
  - El tamaño de este subproblema es la mitad del problema en cada iteración. Por tanto, $\boldsymbol{d} = \mathbf{2}$.
  - Tanto descomponer en subproblemas como recomponer la solución parcial tienen un orden de complejidad constante, $f(n) = \Theta(1) = \Theta(n^0)$. Por tanto, $\boldsymbol{k} = \mathbf{0}$.
- De ello obtenemos que $T(n) = T(n/2) + n^0$
- Como $\boldsymbol{p} = 1$ es igual a $\boldsymbol{d}^{\boldsymbol{k}} = 2^0 = 1$, su complejidad sería de orden $\Theta(n^0 \log n) = \mathbf{\Theta(log\ \boldsymbol{n})}$

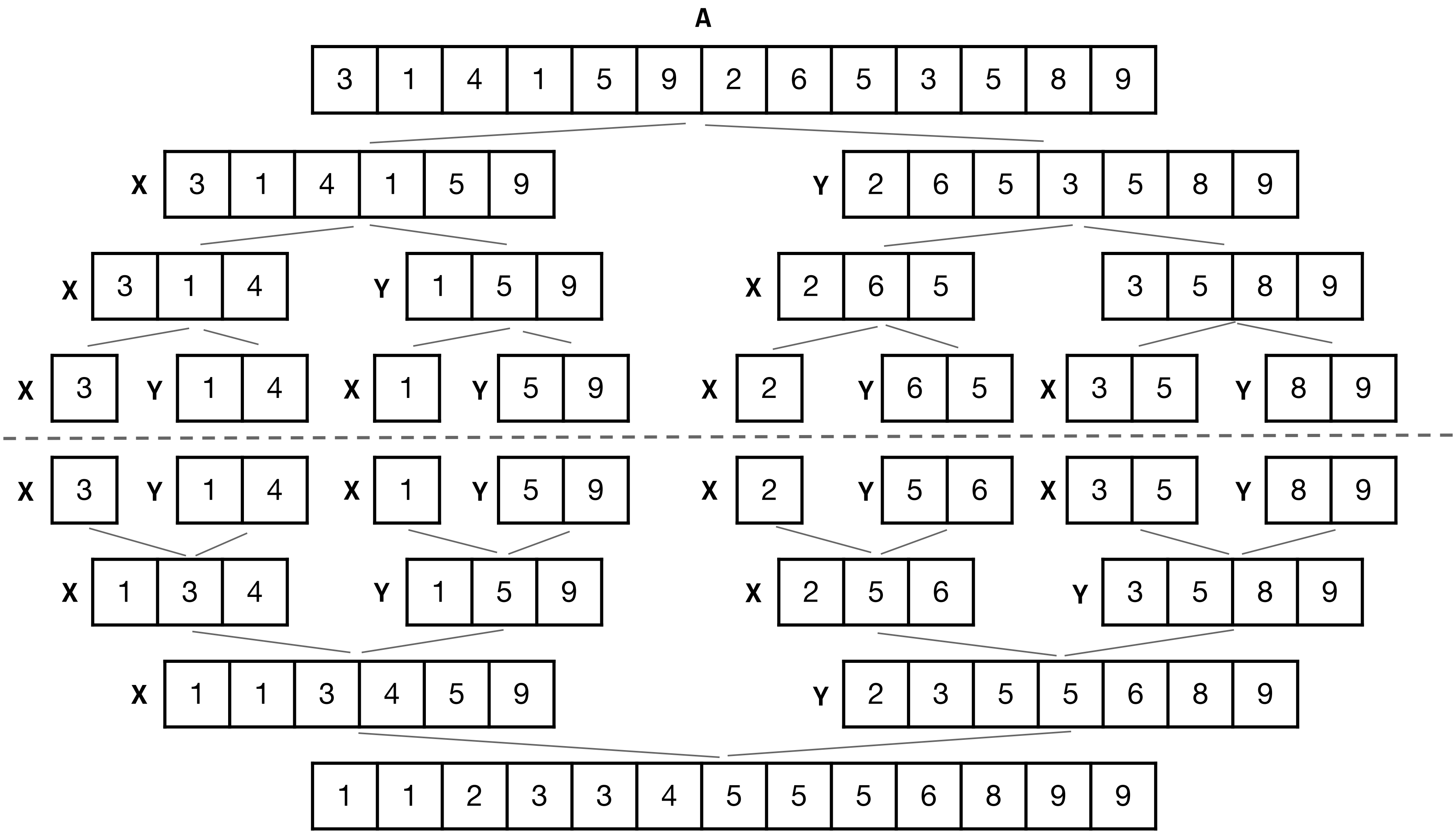
Ordenación por fusión (*mergesort*)

- Dado un array A[1..*n*] de *n* elementos, el problema de ordenación consiste en ordenar dichos elementos en **orden ascendente**.
- El estudio de los diferentes algoritmos de ordenación no es competencia de esta asignatura, pero se expondrá el algoritmo de **ordenación por fusión** (*mergesort*) como ejemplo de algoritmo tipo DyV.
- El enfoque consiste en **descomponer** el array A en **dos partes** cuyos tamaños sean tan parecidos como sea posible, ordenar estas partes mediante llamadas recursivas y después **fusionar las soluciones** de cada parte, manteniendo el orden.
- Para simplificar el proceso, puede usarse la técnica del **centinela**, que consiste en asignar a la última posición (**n**+1) del array un valor que es mayor que cualquier elemento de los arrays considerados. Dicho valor se representa con un **∞**.

Algoritmo de ordenación por fusión (*mergesort*)

```
procedimiento ordenar_por_fusión(A[1..n]) {
  si n es suficientemente pequeño entonces ordenar_ad_hoc(A[1..n])
  en otro caso {
    X[1..⌊n/2⌋] <- A[1..⌊n/2⌋];     ordenar_por_fusión(X[1..⌊n/2⌋])
    Y[1..⌈n/2⌉] <- A[⌊n/2⌋+1..n];  ordenar_por_fusión(Y[1..⌈n/2⌉])
    fusionar(X[1..⌊n/2⌋], Y[1..⌈n/2⌉], A[1..n])
} }
// fusiona X e Y (ya ordenados) en A
procedimiento fusionar(X[1..m], Y[1..s], A[1..m+s]) {
  i <- j <- 1
  X[m+1] <- ∞ ; Y[s+1] <- ∞ // centinelas
  para k <- 1 hasta m+s hacer {
    si X[i] <= Y[j] entonces { A[k] <- X[i] ; i <- i + 1 }
    en otro caso             { A[k] <- Y[j] ; j <- j + 1 }
} }
```

Ejemplo


A
3 1 4 1 5 9 2 6 5 3 5 8 9
X 3 1 4 1 5 9
Y 2 6 5 3 5 8 9
X 3 1 4
Y 1 5 9
X 2 6 5
3 5 8 9
X 3
Y 1 4
X 1
Y 5 9
X 2
Y 6 5
X 3 5
Y 8 9
X 3
Y 1 4
X 1
Y 5 9
X 2
Y 5 6
X 3 5
Y 8 9
X 1 3 4
Y 1 5 9
X 2 5 6
Y 3 5 8 9
X 1 1 3 4 5 9
Y 2 3 5 5 6 8 9
1 1 2 3 3 4 5 5 5 6 8 9 9

Análisis del algoritmo de ordenación por fusión (*mergesort*) (1)

- Cuando n es suficientemente pequeño se utiliza un **subalgoritmo básico** para la ordenación, que debería consumir un tiempo constante si $n \leq 2$.
- En otro caso, tanto la **descomposición** de A en X e Y como el procedimiento `fusionar(X,Y,A)` requieren un tiempo lineal para su ejecución: $T(n) = \Theta(n)$.
- Por tanto: $T(n) = 2 \cdot T(n/2) + n$
- Usando el **Teorema Maestro** ($T(n) = p \cdot T(n/d) + n^k$), tenemos: $\boldsymbol{p} = 2$ $\boldsymbol{d} = 2$ $\boldsymbol{k} = 1$
- Dado que $\boldsymbol{p} = 2$ es igual a $\boldsymbol{d}^k = 2^1 = 2$, tenemos que:

$$T(n) = \Theta(n^k \log n) = \boldsymbol{\Theta(n \log n)}$$

Análisis del algoritmo de ordenación por fusión (*mergesort*) (2)

- Esta complejidad se alcanza porque el tamaño de los subproblemas es aproximadamente igual. En caso de que no fuera así, la **complejidad del algoritmo sería mayor**.
- <u>Pregunta</u>: ¿qué complejidad tendría este algoritmo si el array X fuera de tamaño [1..n-1], e Y de tamaño [1..1]?

  $$T(n) = T(n - 1) + T(1) + n = \Theta(n^2)$$

  Si d = 1, no se puede aplicar el Teorema Maestro. Es necesaria una iteración (con descomposición y fusión, $f(n) = \Theta(n)$) por cada elemento del array ($g(n) = \Theta(n)$).

  Usando las reglas de simplificación, $f(n) \cdot g(n) = \Theta(n^2)$.

Referencias

- ★ Brassard, G. and Bratley, P. (1998) "Fundamentos de Algoritmia", *Prentice-Hall*. [**Capítulo 7**]
- ★ Shaffer, C. A. (2013) "Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), *Dover Publications*. **Freely** available for **educational** and **non-commercial use** at: people.cs.vt.edu/shaffer/Book/C++3elatest.pdf