Diseño de Algoritmos - Algoritmos II

Nazareno Aguirre, Sonia Permigiani, Gastón Scilingo,
Simón Gutiérrez
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 4: La Estrategia Divide & Conquer

Estrategias de Diseño de Algoritmos

Las estrategias de diseño de algoritmos nos brindan herramientas para atacar el problema de construir soluciones algorítmicas para problemas. Hemos visto una de ellas, Fuerza Bruta, que es muy general pero también tiene limitaciones.

Veremos ahora una segunda estrategia, Divide & Conquer, que corresponde a un mecanismo de resolución de problemas muy general (admite su aplicación en una amplia cantidad de casos) y suele ofrecer soluciones eficientes.

Divide & Conquer

Una forma general de describir Divide & Conquer, es la siguiente:

Una solución Divide & Conquer para un problema P dado consiste en:

- identificar un conjunto de instancias del problema P elementales, para las cuales la solución es inmediata.
- para aquellas instancias de P que no sean elementales, encontrar una forma de resolverlas en términos de soluciones para instancias del problema P "de menor tamaño", a través de un patrón dividir - conquistar - combinar.

En esta descripción de Divide & Conquer aparecen algunos conceptos importantes, como el concepto de "problema elemental", y de "tamaño del problema".

Divide & Conquer y Recursión

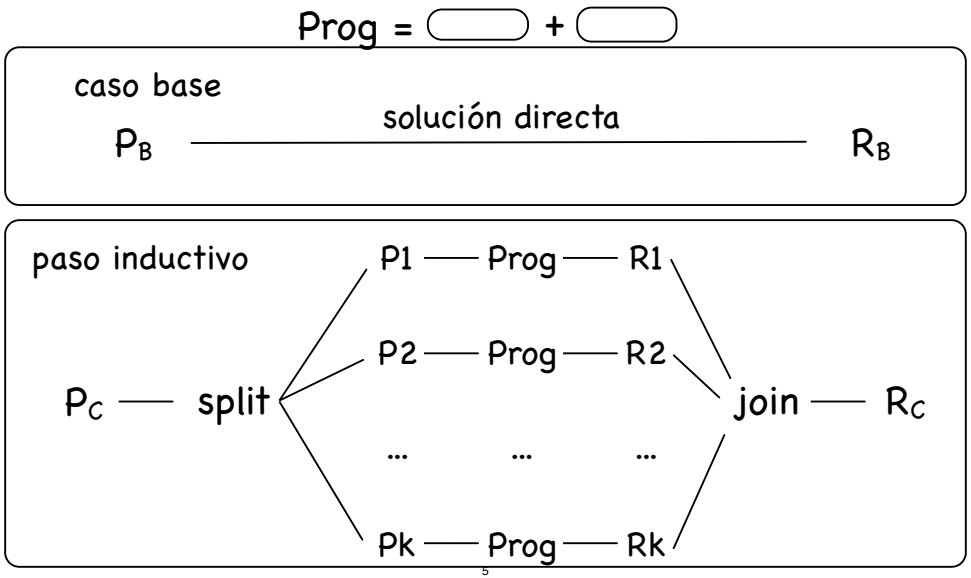
Existe una clara correspondencia entre la estrategia de diseño Divide & Conquer y recursión:

- los problemas elementales corresponden a los casos base
- la resolución de problemas de menor tamaño y su combinación corresponden al paso inductivo

La reducción a problemas de menor tamaño debe garantizar que sucesivas reducciones eventualmente alcanzan los casos base

La relación de orden inducida por la reducción a problemas de menor tamaño debe ser un orden bien fundado (i.e., sin cadenas descendentes de tamaño infinito).

Esquema General de la Estrategia Divide & Conquer



Ejemplo: Ordenamiento (cont.)

Veamos como ejemplo el problema de ordenar una secuencia de elementos, y un par de soluciones Divide & Conquer para el problema.

Mergesort:

caso base: secuencias de tamaño menor o igual a 1.

Solución: retornar la misma secuencia (ya está ordenada)

paso inductivo (para secuencias de tamaño mayor a 1):

Split: separar la secuencia en dos partes de aproximadamente el mismo tamaño

Split.xs = (take.#xs/2.xs, drop.#xs/2.xs)

Join: Realizar la "mezcla ordenada" de las dos secuencias (ya ordenadas)

Join.xs.ys = merge.xs.ys

Ejemplo: Ordenamiento (cont.)

QuickSort:

caso base: secuencias de tamaño menor o igual a 1.

Solución: retornar la misma secuencia (ya está ordenada)

paso inductivo (para secuencias de tamaño mayor a 1):

Split: elegir un pivote, y separar la secuencia en dos partes, de manera tal que en la primera todos los elementos son menores o iguales que el pivote y en la segunda todos los elementos son mayores que el pivote

Split.xs = partition.xs

Join: Concatenar las dos secuencias (ya ordenadas)

Join.xs.ys = xs++ys

Balance entre Split y Join

Los ejemplos anteriores de la estrategia Divide & Conquer nos muestran una relación que se da usualmente entre Split y Join

En soluciones Divide & Conquer, en general sucede que si la división de problemas en subproblemas es "simple", entonces el "pegado" de soluciones parciales para conseguir la solución global es "difícil". Igualmente, si la división de problemas en subproblemas es "difícil", el pegado es "simple".

En los ejemplos, QuickSort es un algoritmo en el cual la división es difícil (y el pegado es fácil), mientras que MergeSort es un algoritmo en el cual la división es fácil (y el pegado difícil).

Ejemplo de Análisis de Tiempo de Ejecución

El análisis de soluciones Divide & Conquer recursivas demanda la resolución de ecuaciones de recurrencia. Consideremos el caso de MergeSort sobre un lenguaje imperativo:

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

donde Merge está definido de la siguiente manera:

Ejemplo de Análisis de Tiempo de Ejecución (cont.)

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \leq C[j]$
$A[k] \leftarrow B[i];\ i \leftarrow i+1$
else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
$k \leftarrow k+1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Las ecuaciones de recurrencia para este caso son, suponiendo que la operación de interés es la escritura en arreglos, las siguientes:

$$\begin{array}{rcl} T(1) & = & 0 \\ T(n) & = & 2 \times T(\frac{n}{2}) + 2 \times n \end{array}$$

Ejemplo de Análisis de Tiempo de Ejecución (cont.)

Realicemos en este caso el análisis mediante expansión de recurrencias:

$$\begin{array}{rcl} T(n) & = & 2 \times T(\frac{n}{2}) + 2 \times n \\ & = & 2 \times (2 \times T(\frac{n}{4}) + 2 \times \frac{n}{2}) + 2 \times n \\ & = & 4 \times T(\frac{n}{4}) + 2 \times n + 2 \times n \\ & = & 4 \times (2 \times T(\frac{n}{8}) + 2 \times \frac{n}{4}) + 2 \times n + 2 \times n \\ & = & 8 \times T(\frac{n}{8}) + 2 \times n + 2 \times n + 2 \times n \end{array}$$

La forma cerrada para T(n) es entonces:

$$T(n) = 2^i \times T(\frac{n}{2^i}) + i \times (2 \times n)$$

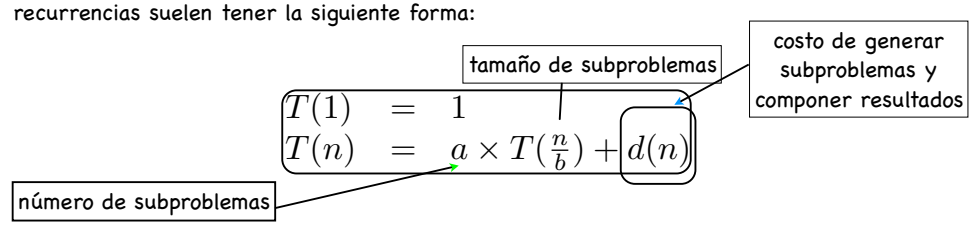
Reemplazando i por $\log_2 n$ tenemos:

$$\begin{array}{rcl} T(n) & = & 2^{\log_2 n} \times T(\frac{n}{2^{\log_2 n}}) + (\log_2 n) \times (2 \times n) \\ & = & n \times T(1) + (\log_2 n) \times (2 \times n) \\ & = & 0 + (\log_2 n) \times (2 \times n) \\ & = & (\log_2 n) \times (2 \times n) \end{array}$$

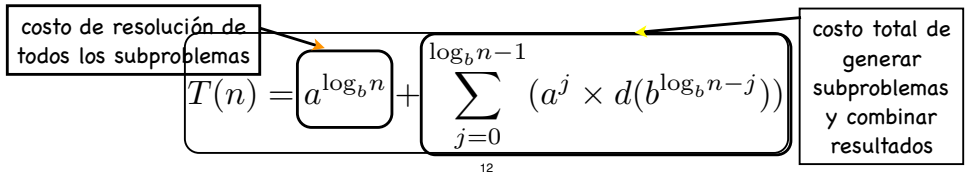
Luego, la tasa de crecimiento de T es $n \log_2 n$.

Tiempo de Ejecución de Algoritmos Divide & Conquer

Dijimos que el análisis de tiempo de ejecución de algoritmos recursivos requiere la solución de ecuaciones de recurrencias. En una cantidad importante de casos, estas ecuaciones de recurrencias suelen tener la siguiente forma:



Podemos resolverlas de forma genérica. La resolución de estas ecuaciones de recurrencia da como resultado lo siguiente:



Tiempo de Ejecución de Algoritmos Divide & Conquer (cont.)

La expresión anterior puede simplificarse si conocemos propiedades de $d(n)$ y $a^{\log_b n}$.

Si $d(n) = 0$ entonces $T(n) \in \Theta(a^{\log_b n})$

Si $d(n) \in \Theta(n^\alpha)$ entonces

si $a < b^\alpha$ entonces $T(n) \in \Theta(n^\alpha)$

si $a = b^\alpha$ entonces $T(n) \in \Theta(n^\alpha \log_b n)$

si $a > b^\alpha$ entonces $T(n) \in \Theta(n^{\log_b a})$

Este resultado se conoce como el Teorema Maestro

Ejemplo de Análisis de Tiempo de Ejecución

Consideremos ahora el caso de MergeSort implementado en un lenguaje funcional:

```
mergesort [] = []

mergesort [x] = [x]

mergesort x:xs = merge (mergesort (take (div n 2) x:xs)) (mergesort (drop (div n 2) x:xs))
```

Las ecuaciones de recurrencia resultantes son las siguientes:

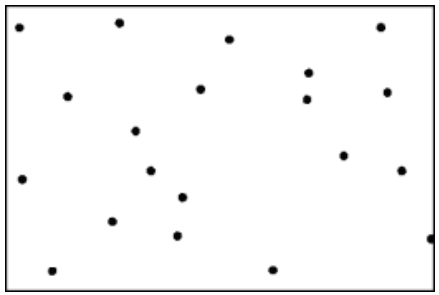
$$\begin{array}{lcl} T(1) & = & 1 \\ T(n) & = & 2 \times T(\frac{n}{2}) + 2 \times \frac{n}{2} + n \end{array}$$

Tenemos entonces: $a = 2, b = 2, d(n) = 2 \times n, d(n) \in \Theta(n)$

$$T(n) \in \Theta(n \log_2 n)$$

Otro Ejemplo: Puntos Más Cercanos entre Sí

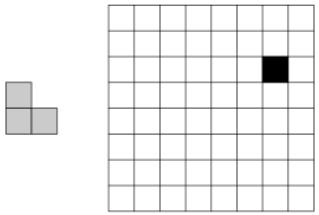
Dado un conjunto de puntos en el plano, se desea encontrar el par de puntos distintos cuya distancia entre ellos es la menor:



Pensemos en una solución Divide & Conquer para este problema.

Otro Ejemplo: El Problema de Trominós

Se desea cubrir una cuadrícula de 2^nx2^n cuadros, con uno de ellos ya cubierto, usando trominós:



Pensemos en una solución Divide & Conquer para este problema (Ayuda: consideremos varios subproblemas de igual tamaño del problema original).