2 Divide y vencerás

ALGORÍTMICA Y COMPLEJIDAD

Camilo Palazuelos Calderón

Curso 2023-2024

```
DV(X):
    (Y_1,...,Y_a) ← (NULL,...,NULL)      ▷ Soluciones de los subprobs.
    if EsSencillo(X)                     ▷ Caso directo
        return Directo(X)
    else                                 ▷ Caso recursivo
        (X_1,...,X_a) ← Descomponer(X)
        for i ← 1 to a
            Y_i ← DV(X_i)
        Y ← Combinar(Y_1,...,Y_a)
        return Y
```

Qué es divide y vencerás (DV)

- Suele utilizarse cuando un problema puede dividirse en subproblemas no solapados
- El diseño recursivo es la manera natural de pensar en DV
  - Se divide el problema en subproblemas
  - Se resuelve cada subproblema de forma independiente
  - Se combinan las soluciones de los subproblemas

Consideraciones sobre DV

- La complejidad de DIRECTO influye en el tiempo de ejecución del algoritmo
  - ¡Pero no en su *ritmo de crecimiento*!

- ESSENCILLO puede definirse de forma teórica o empírica

- Condiciones para usar DV
  - Debe asegurarse que no se resuelve el mismo subproblema más de una vez
  - DESCOMPONER y COMBINAR deben ser eficientes
  - Los subproblemas deben tener un tamaño similar

El teorema maestro

- Proporciona una solución en términos asintóticos para relaciones de recurrencia del tipo

$$T(n) = a\,T(n/b) + O(n^c)$$

  - a es el número de subproblemas
  - b es por cuánto se divide el tamaño del problema
  - $O(n^c)$ es el coste de DESCOMPONER y COMBINAR

- Solución
  - Si $a < b^c$, entonces $T(n)$ es $\Theta(n^c)$
  - Si $a = b^c$, entonces $T(n)$ es $\Theta(n^c \log n)$
  - Si $a > b^c$, entonces $T(n)$ es $\Theta(n^d)$, donde $d = \log_b a$

```
MULTIPLICAR(x, y, n):
    if n = 1
        return x · y
    else
        m ← ⌈n/2⌉
        a ← ⌊x/10^m⌋; b ← x mod 10^m        ▷ x = 10^m a + b
        c ← ⌊y/10^m⌋; d ← y mod 10^m        ▷ y = 10^m c + d
        e ← MULTIPLICAR(a, c, m)
        f ← MULTIPLICAR(b, c, m)
        g ← MULTIPLICAR(a, d, m)
        h ← MULTIPLICAR(b, d, m)
        return 10^2m e + 10^m (f + g) + h
```

Multiplicación rápida

- El algoritmo de la escuela multiplica dos números de n dígitos en $O(n^2)$
  - ¿Y si dividiéramos los *arrays de dígitos* a la mitad y explotáramos la siguiente identidad?

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

donde $m \leftarrow \lceil n/2 \rceil$, $a \leftarrow \lfloor x/10^m \rfloor$, $b \leftarrow x \bmod 10^m$,

$c \leftarrow \lfloor y/10^m \rfloor$, $d \leftarrow y \bmod 10^m$

- La recurrencia es $T(n) = 4\,T(n/2) + O(n)$
  - Por tanto, el coste temporal es $\Theta(n^2)$
  - *¡No hemos logrado mejorar la complejidad!*

```
KARATSUBA(x, y, n):
    if n = 1
        return x · y
    else
        m ← ⌈n/2⌉
        a ← ⌊x/10^m⌋; b ← x mod 10^m        ▷ x = 10^m a + b
        c ← ⌊y/10^m⌋; d ← y mod 10^m        ▷ y = 10^m c + d
        e ← MULTIPLICAR(a, c, m)
        fg ← MULTIPLICAR(a – b, c – d, m)
        h ← MULTIPLICAR(b, d, m)
        return 10^2m e + 10^m (e + h – fg) + h
```

Entra Karatsuba

- En 1960, Karatsuba observó que

$$bc + ad = ac + bd - (a - b)(c - d)$$

- Ahora, la recurrencia es $T(n) = 3\,T(n/2) + O(n)$
  - Por tanto, el coste temporal es $\Theta(n^{\lg 3}) = \Theta(n^{1.585})$
- Algoritmos más rápidos
  - *Schönhage–Strassen:* $O(n \log n \log \log n)$
  - *Harvey–Van der Hoeven:* $O(n \log n)$

Ordenamiento de arrays, I

- MERGESORT
  - Divide el array en dos subarrays del mismo tamaño
  - Ordena cada subarray independientemente
  - Mezcla ambos subarrays ordenados

- La recurrencia es $T(n) = 2\,T(n/2) + O(n^c)$
  - $O(n^c)$ es el coste del algoritmo FUSIONAR
  - Si $c = 1$, MERGESORT será $\Theta(n \log n)$
  - Si $c > 1$, MERGESORT será $\Theta(n^c)$

- Debe diseñarse un FUSIONAR lineal
  - Nos interesa que el coste temporal sea subcuadrático

```
MERGESORT(A[1..n]):
    if n > 1
        m ← ⌊n/2⌋
        MERGESORT(A[1..m])
        MERGESORT(A[m + 1..n])
        FUSIONAR(A[1..n], m)

FUSIONAR(A[1..n], m):
    B[1..n] ← (NULL,…,NULL)
    i ← 1
    j ← 1
    while i ≤ m or m + j ≤ n
        if i ≤ m and (m + j > n or A[i] ≤ A[m + j])
            B[i + j – 1] ← A[i]
            i ← i + 1
        if m + j ≤ n and (i > m or A[i] > A[m + j])
            B[i + j – 1] ← A[m + j]
            j ← j + 1
    A[1..n] ← B[1..n]
```

Ordenamiento de arrays, II

- QUICKSORT
  - Divide el array en dos subarrays con respecto a un *pivote* p
  - Reorganiza los elementos en función del pivote
  - El pivote siempre queda ordenado

- En el mejor caso, la recurrencia es $T(n) = 2\,T(n/2) + O(n)$
  - Por tanto, el coste temporal es $\Theta(n \log n)$

- En el peor caso, la recurrencia es $T(n) = T(n-1) + O(n)$
  - $T(n) = T(p-1) + T(n-p) + O(n)$, con peor caso cuando $p = 1$ o $p = n$
  - La recurrencia es de la forma $a\,T(n-b) + O(n^c)$

```
QUICKSORT(A[1..n]):
    if n > 1
        p ← REORGANIZAR(A[1..n])
        QUICKSORT(A[1..p – 1])
        QUICKSORT(A[p + 1..m])

REORGANIZAR(A[1..n]):
    A[1] ↔ A[n]
    k ← 0
    for i ← 1 to n – 1
        if A[i] < A[n]
            k ← k + 1
            A[k] ↔ A[i]
    A[k + 1] ↔ A[n]
    return k + 1
```

Ordenamiento de arrays, II

- Soluciones en términos asintóticos para ecuaciones de recurrencia del tipo

$$T(n) = a\ T(n - b) + O(n^c)$$

  - Si $a = 1$, entonces $T(n)$ es $\Theta(n^{c+1})$
  - Si $a > 1$, entonces $T(n)$ es $\Theta(n^c\ a^{n/b})$

- Por tanto, en el peor caso, QUICKSORT es $\Theta(n^2)$