

La estrategia algorítmica dividir para reinar

1. Dividir el problema original en dos (o más) subproblemas del mismo tipo
2. Resolver (recursivamente) cada subproblema
3. Encontrar la solución al problema original a partir de las soluciones a cada subproblema

mergesort es un algoritmo basado en la estrategia dividir para reinar

A. Dividir el problema original en dos subproblemas del mismo tipo

B. Resolver (recursivamente) cada subproblema

C. Encontrar la solución al problema original a partir de las soluciones a cada subproblema

mergeSort(secuencia *A*):

1. Si *A* tiene un solo elemento, terminar en este paso
2. Dividir la secuencia en mitades
3. Ordenar cada mitad recursivamente usando *mergeSort*
4. Combinar las mitades (ya ordenadas) usando *merge*

Búsqueda binaria es otro algoritmo basado en dividir para reinar

1. La división del problema en dos subproblemas se hace simplemente mirando el dato en la posición central

2. La naturaleza del problema hace que sólo sea necesario resolver uno de los subproblemas

3. La solución al problema original es la solución al subproblema resuelto en el paso 2

binSearch(A, x, i, f):

if $f < i$: *return false*

$m \leftarrow \left\lfloor \frac{i+f}{2} \right\rfloor$

if $A[m] = x$: *return m*

if $A[m] > x$:

return binSearch($A, x, i, m - 1$)

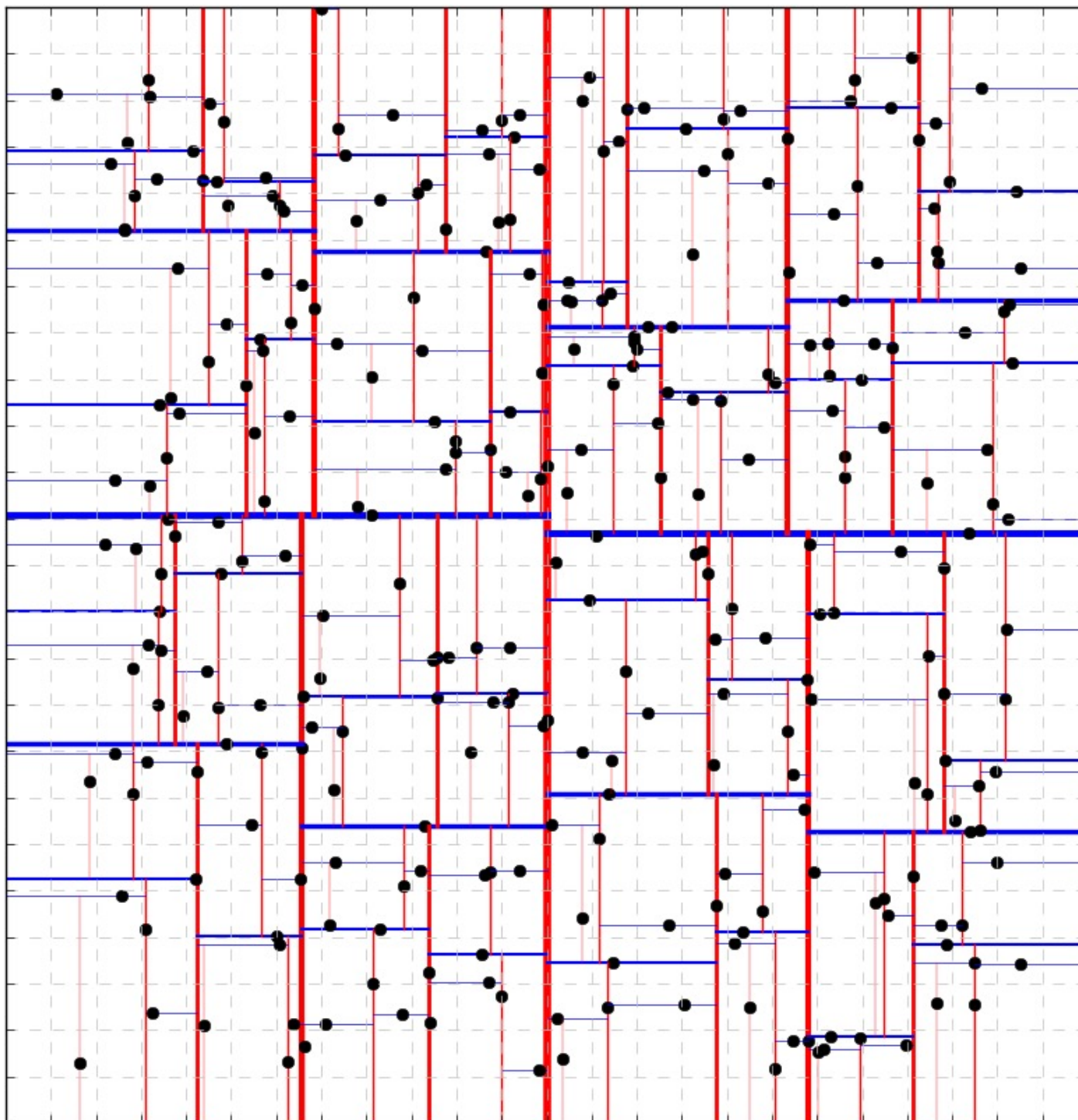
if $A[m] < x$:

return binSearch($A, x, m + 1, f$)

El conjunto de coordenadas

- Se quiere procesar un conjunto muy grande de coordenadas en 2D
- Para repartir la carga, se reparten los datos en zonas rectangulares
- La idea es que las zonas no se traslapen: deben particionar el espacio
- Queremos además que cada zona tenga la misma cantidad de datos

¿Cómo garantizamos que todas estas condiciones se cumplen?



El desafío de encontrar la mediana



¿Cómo encontrar la mediana de un conjunto de datos?

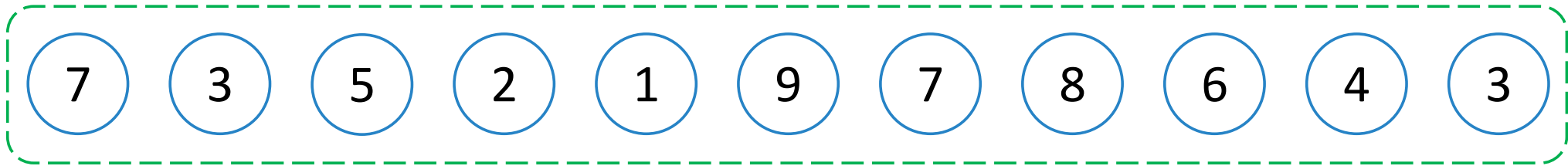
¿Podemos hacerlo sin recurrir a ordenar los datos?

Pensemos en la definición de mediana

Un posible procedimiento para encontrar la mediana ...



Set de datos:



Pivote: 6

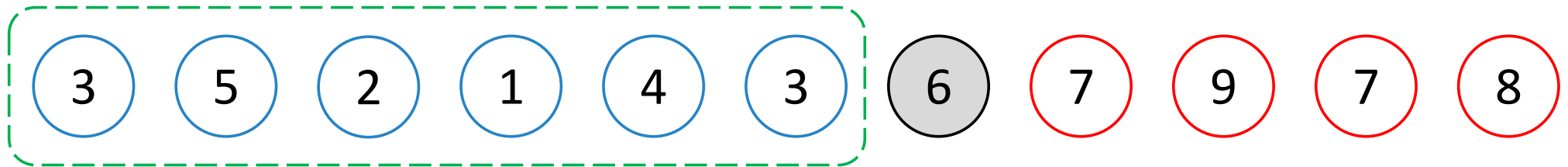
Menores: 3 5 2 1 4 3

Mayores: 7 9 7 8

¿Cuál de los dos grupos contiene la mediana? ¿Por qué?

... recursivamente

Repitamos el proceso



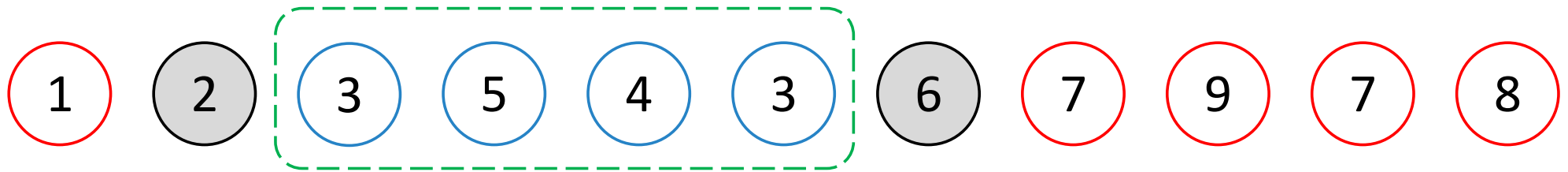
Pivote: 2

Menores: 1

Mayores: 3 5 4 3

...

Repitamos el proceso nuevamente



Pivote: 5

Menores: 3 4 3

Mayores:

... hasta que finalmente ...



¿Cómo sabemos cuándo encontramos la mediana?



El procedimiento de elegir un dato de $A[i] \dots A[f]$ como *pivote* y luego separar el resto dos grupos —los menores que el pivote y los mayores que el pivote— se llama *partition*:

partition(A, i, f):

$p \leftarrow$ un pivote aleatorio en $A[i, f]$

$m, M \leftarrow$ listas vacías

insertar p en M

for $x \in A[i, f]$:


if $x < p$, insertar x en m

else if $x > p$, insertar x en M

$A[i, f] \leftarrow$ concatenar m con M

return $i + |m|$

partition retorna la cantidad de datos en $A[i] \dots A[f]$ menores que el pivote **más** la cantidad de datos en A a la izquierda de $A[i]$



median encuentra y retorna la mediana de los datos en el arreglo A ; no ordena los datos de A , sino que aplica repetidamente *partition* al subarreglo $A[i] \dots A[f]$, que se va actualizando en cada repetición:

median(A):

$i \leftarrow 0, f \leftarrow n - 1$

$x \leftarrow \text{partition}(A, i, f)$

while $x \neq \frac{n}{2}$:

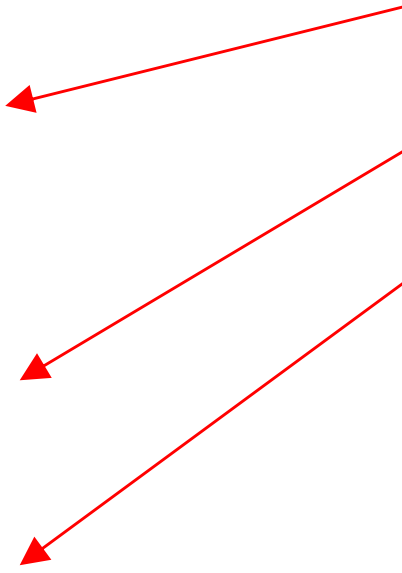
if $x < \frac{n}{2}$, $i \leftarrow x + 1$

else, $f \leftarrow x - 1$

$x \leftarrow \text{partition}(A, i, f)$

return $A[x]$

debido a esta llamada inicial, con $i = 0$ y $f = n-1$, y a la forma de ajustar i y f en el *while* antes de las siguientes llamadas, *partition* siempre retorna la cantidad de datos en A menores que el pivote



median también es un algoritmo basado en dividir para reinar

1. La división del problema en dos subproblemas se hace empleando *partition*
2. La naturaleza del problema hace que sólo sea necesario resolver uno de los dos subproblemas:
 - por lo mismo, podemos fácilmente cambiar la recursión por iteración
 - similarmente al caso de búsqueda binaria
3. La solución al problema original se obtiene a partir de la solución al subproblema resuelto en el paso 2

Analicemos el algoritmo *median*



¿Cuál es su complejidad?

- depende de cuán balanceadas son las particiones que va haciendo *partition*
... lo que a su vez depende del elemento que resulta elegido como pivote
... sobre lo cual no tenemos mucho control

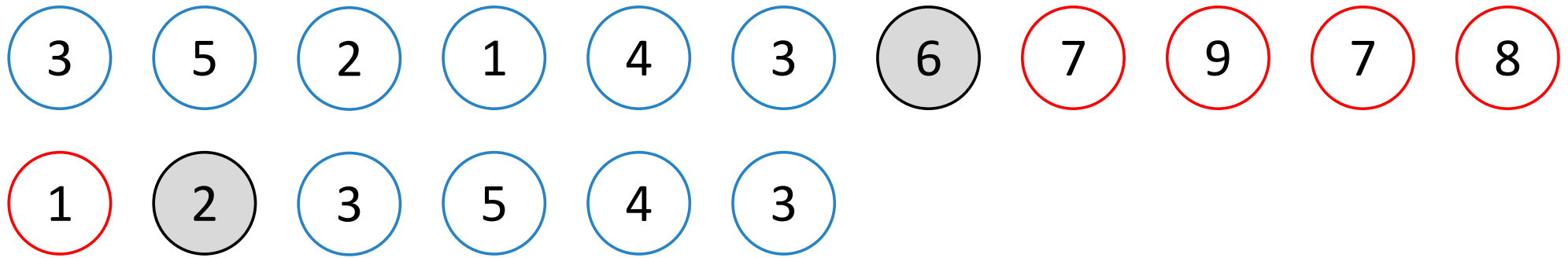
¿En el mejor caso? ¿En el peor caso?

- varía entre $O(n)$ y $O(n^2)$

¿Podemos aprovechar *partition* para ordenar?



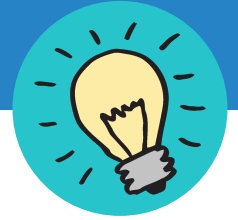
Observación: después de cada ejecución de *partition*, el pivote queda en su posición ordenada



¿Por qué?

¿Se puede usar esto para ordenar?

Podemos emplear una vez más la estrategia dividir para reinar



1. *partition* divide la secuencia en tres partes:
 - subsecuencia de elementos menores que el pivote, a la izquierda del pivote
 - el pivote
 - subsecuencia de elementos mayores que el pivote, a la derecha del pivote
2. Aplicamos recursivamente el algoritmo a cada una de las dos subsecuencias
3. No es necesario combinar nada

El algoritmo se llama *quickSort*

quicksort(A, i, f):

if $i \leq f$:

$p \leftarrow \textit{partition}(A, i, f)$

quicksort($A, i, p - 1$)

quicksort($A, p + 1, f$)

La llamada inicial es ***quicksort***($A, 0, n - 1$)

A S O R T I N G E X A M P L E
 A A E **E** T I N G O X S M P L R
 A A **E**
 A **A**
A

L I N G O P M **R** X T S
 L I G **M** O P N
G I L
 — I **L**
 — **I**

N P O
 — **O** P
 — — **P**

S T X
 — **T** X
 — **T**

A A E E G I L M N O P R S T X

En este ejemplo, el pivote es siempre el elemento que está en el extremo derecho del subarreglo, es decir, $A[f]$; al terminar *Partition*, el pivote queda en la posición que se muestra en rojo

Analicemos *quicksort*

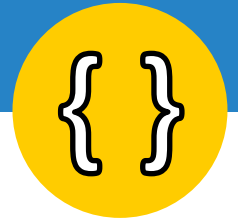


¿Cuál es su complejidad en el mejor caso?

¿Y en el peor caso?

¿Y en el caso promedio?

Quicksort en arreglos



En general, usar arreglos es más conveniente

Pero la operación de concatenar es muy cara en arreglos

Debemos reformular **partition** para que funcione en arreglos

partition(A, i, f):

$x \leftarrow$ un indice aleatorio en $[i, f]$, $p \leftarrow A[x]$

$A[x] \rightleftharpoons A[f]$

$j \leftarrow i$

for $k \in [i, f - 1]$:

if $A[k] < p$:

$A[j] \rightleftharpoons A[k]$

$j \leftarrow j + 1$

$A[j] \rightleftharpoons A[f]$

return j

Posibles mejoras

Cambiar a `insertionSort()` para subarreglos pequeños ($n \leq 20$)

Usar la mediana de tres elementos como pivote

Si hay cantidades grandes de claves duplicadas en el arreglo de entrada —p.ej., un archivo de fechas de transacciones diarias— es muy probable que `quickSort()` particione, innecesariamente, subarreglos en que todas las claves son iguales:

- podemos particionar el arreglo en tres partes: ítemes con claves menores que el pivote, iguales al pivote, y mayores que el pivote

Algoritmos de Ordenación

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n)$
QuickSort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(1)$
?	?	?	?	?

<https://www.youtube.com/watch?v=kPRA0W1kECg>