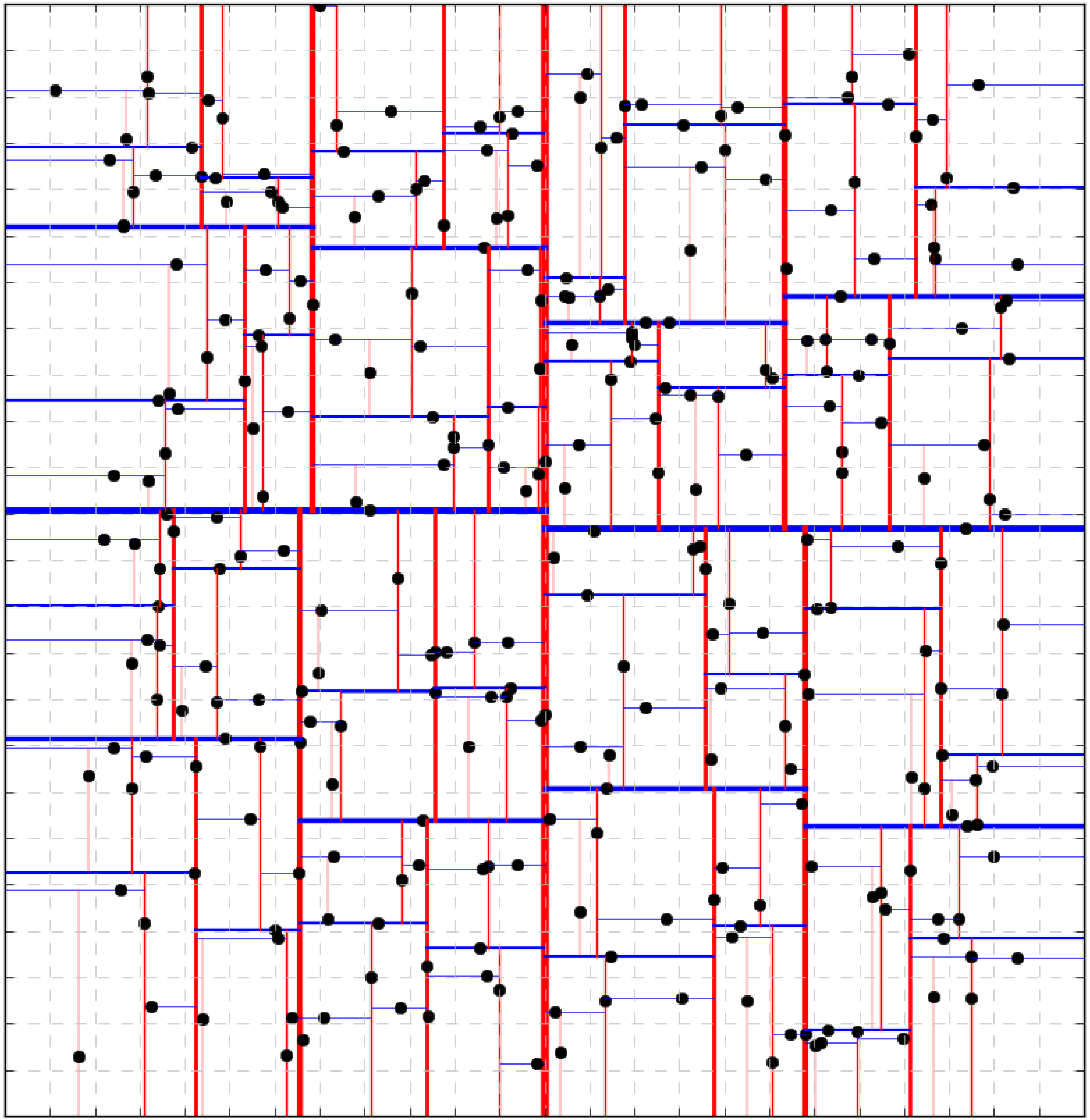
El conjunto de coordenadas

- 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?

¿Cómo hacerlo sin recurrir a ordenar los datos?

Pensemos en la definición de mediana

Un posible procedimiento para encontrar la mediana ...

Set de datos:

7 3 5 2 1 9 7 8 6 4 3

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

3 5 2 1 4 3 6 7 9 7 8

Pivote: 2

Menores: 1

Mayores: 3 5 4 3

…

Repitamos el proceso nuevamente

1 2 3 5 4 3 6 7 9 7 8

Pivote: 5

Menores: 3 4 3

Mayores:

… hasta que finalmente …

¿Como sabemos cuando encontramos la mediana?

1 2 3 4 3 5 6 7 9 7 8

$partition(A,\ i,\ f)$:

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

$m,\ M \leftarrow listas\ vacias$

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

$if\ x < p,\ insertar\ x\ en\ m$

$else,\ insertar\ x\ en\ M$

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

$return\ i + |m|$

$median(A)$:

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

$x \leftarrow partition(A, \; i, \; f)$

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

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

$else, \; f \leftarrow x - 1$

$x \leftarrow partition(A, \; i, \; f)$

$return \; A[x]$

Analicemos el algoritmo *median*

¿Cuál es su complejidad?

¿En el mejor caso? ¿En el peor caso?

¿Cómo extenderlo para buscar el dato que ocupe una posición arbitraria?

```
int quickSelect(a, p, r, j):
    if p == r:
        return a[p]
    q = partition(a, p, r)
    k = q-p+1
    if j == k:
        return a[q]
    else:
        if j < k:
            return quickSelect(a, p, q-1, j)
        else:
            return quickSelect(a, q+1, r, j-k)
```

Analicemos el algoritmo *quickselect*

¿Es la misma complejidad si hay elementos repetidos?

¿Cambia el caso promedio?

¿Cómo se puede mejorar?

¿Podemos aprovechar *quickselect* para ordenar?

En cada iteración, el pivote queda en su posición ordenada

¿Por qué?

¿Se puede usar esto para ordenar?

...

En cada iteración, el pivote queda en su posición ordenada

¿Por qué?

¿Se puede usar esto para ordenar?

$quicksort(A, i, f)$:

$if\ i \leq f$:

$p \leftarrow partition(A,\ i,\ f)$

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

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

Luego se llama $\boldsymbol{quicksort(A,\ 0,\ n-1)}$

Analicemos *quicksort*

¿Cual 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

$partition(A,\ i,\ f)$:

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

$A[x] \rightleftarrows A[f]$

$j \leftarrow i$

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

$if\ A[k] < p$:

$A[j] \rightleftarrows A[k]$

$j \leftarrow j+1$

$A[j] \rightleftarrows A[f]$

$return\ j$

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— 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