Paradigma
Divide y Vencerás

Divide y vencerás

- El principio en que se basa el paradigma de diseño de algoritmos Divide y vencerás es que (a menudo) es más fácil resolver varios casos pequeños de un problema que uno grande.
- El enfoque de Divide y vencerás:
- *Divide el problema en* ejemplares más pequeños del mismo problema (en este caso, conjuntos más pequeños a ordenar),
- Luego resuelven (*vencen) los ejemplares más pequeños de forma recursiva (o sea, empleando el*
- mismo método).
- Por último *combinan las soluciones para obtener la solución correspondiente a* la entrada original.

Recursividad

- Una tarea difícil puede dividirse en varias tareas más simples, cada una de las cuales puede a su vez dividirse en tareas más simples todavía hasta llegar a un nivel de simplicidad en el que ya no se justifique la necesidad del volver a dividir.
- Sin embargo, la naturaleza de cierto tipo de problemas nos inducirá a pensar el soluciones basadas en funciones que se llamen a sí mismas para resolverlos.
- Esto no quiere decir que no puedan resolverse de “manera tradicional”; solo que, dada su naturaleza será mucho más fácil encontrar y programar una solución recursiva que una solución iterativa tradicional.
- Cuando una función se invoca a sí misma decimos que es una función recursiva.

Funciones recursivas

- Una definición es recursiva cuando se “define en función de sí misma”.
- Análogamente, se dice que una función es recursiva cuando, para resolver un problema, se invoca a sí misma una y otra vez hasta que el problema queda resuelto.

Finalización de la recursión

- Todo algoritmo recursivo debe finalizar en algún momento, de lo contrario el programa hará que se desborde la pila de llamadas y finalizará abruptamente.

Recurrencia

- Es la acción de volver a ocurrir o aparecer una cosa con cierta frecuencia o de manera iterativa.
- Las recurrencias van de la mano con el paradigma de “Divide y Vencerás”, porque ellas nos dan una manera natural para caracterizar los tiempos de ejecución de los algoritmos “Divide y vencerás”.
- Una recurrencia es una ecuación o desigualdad que describe una función en términos de su valor sobre pequeños valores.

Ordenamiento rápido

Quick Sort

Ordenamiento rápido

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del conjunto de elementos a ordenar, al que llamaremos pivote.

- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.

- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

- QuickSort es un algoritmo relativamente simple y extremadamente eficiente cuya lógica es recursiva y, según su implementación, puede llegar a requerir el uso de arreglos auxiliares.

Implementación utilizando arreglos auxiliares

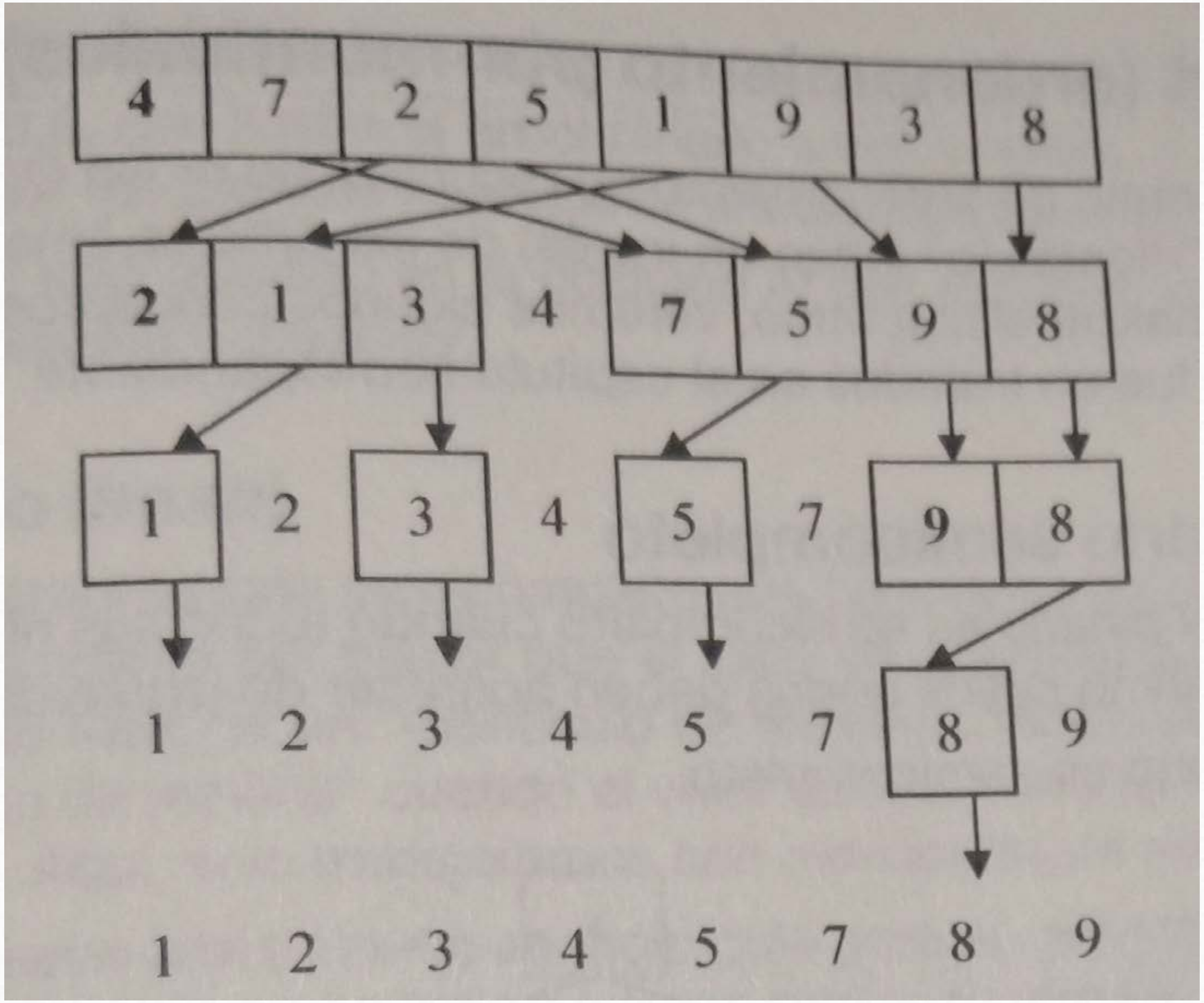
- Llamemos **arr** al arreglo que queremos ordenar.
- Tomamos cualquier elemento de **arr**.
- A este elemento lo llamaremos **pivote**.
- Luego recorremos **arr** para generar dos arreglos auxiliares:
  - El primero tendrá aquellos elementos de **arr** que resulten ser menores que **pivote**.
  - El segundo tendrá los elementos de **arr** que sean mayores que **pivote**.
- A estos arreglos auxiliares los llamaremos respectivamente **menores** y **mayores.**

- Ahora repetiremos el procedimiento, primero sobre **menores** y luego sobre **mayores**.
- Finalmente obtenemos el arreglo ordenado uniendo:

menores + **pivote** + **mayores**

- Por ejemplo: **arr** = {4, 7, 2, 5, 1, 9, 3, 8}

  Si consideramos **pivote** = 4 (es decir el primer elemento de **arr**), entonces:



Ordenamiento mediante Quick Sort

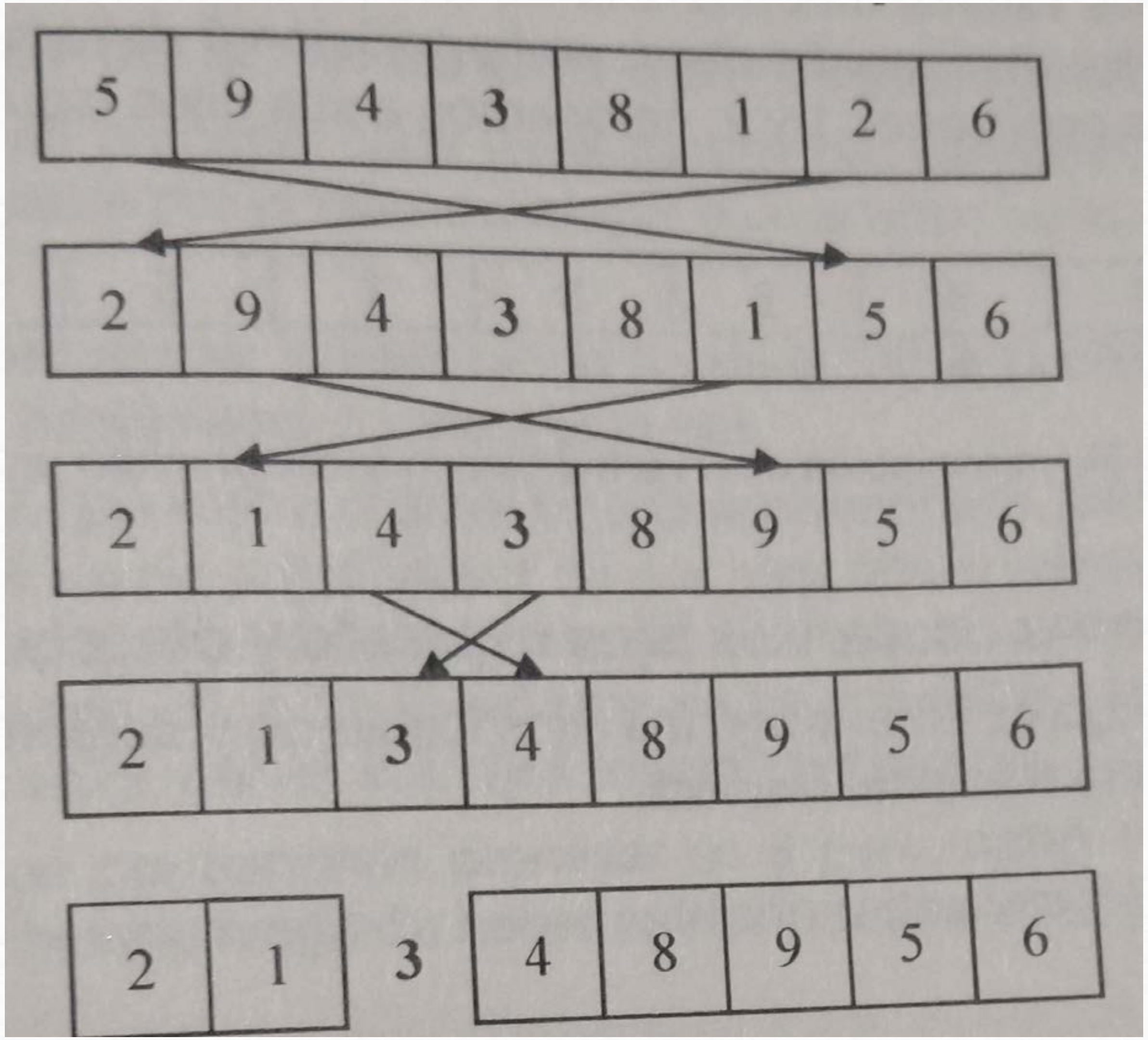
Implementación sin arreglos auxiliares

- Otra implementación de quicksort puede ser la siguiente:
- Luego de seleccionar el elemento **pivote** movemos todos los elementos menores a su izquierda y todos los elementos mayores a su derecha.
- Esto podemos lograrlo utilizando dos índices:
  - **i** – Inicialmente apuntando al primer elemento del arreglo.
  - **j** – Apuntando al último.
- La idea es recorrer el arreglo desde la izquierda hasta encontrar el primer elemento mayor que **pivote**.
- Luego recorrer desde la derecha hasta encontrar el primer elemento menor que el **pivote** y permutarlos.

- Por último, invocamos recursivamente dos veces al algoritmo, primero pasándole el subarreglo comprendido entre el inicio y la posición que ocupa el **pivote** (no inclusive).

- Y luego pasándole el subarreglo formado por los elementos que se encuentran ubicados en posiciones posteriores a la del **pivote**.

- Por ejemplo, si **pivote es 3**, entonces:



- Luego de este proceso todos los elementos menores que **pivote** quedarán ubicados a su izquierda mientras que todos los elementos mayores quedarán ubicados a su derecha.

- Notemos que **pivote** quedó ubicado en su lugar definitivo.

- El próximo paso será repetir el proceso sobre cada uno de estos subarreglos.

Implementación "Divide y vencerás"

- Se aplican los tres pasos del paradigma "Divide y vencerás" para ordenar un subarreglo **A[*p..r*].**
- **Dividir:** Particione el arreglo A[p..r] en dos subarreglos (posiblemente vacíos) A[p..q-1] y A[q+1..r] tal que cada elemento de A[p..q-1] es menor o igual que A[q], que es, a su vez, menor o igual que cada elemento de A[q+1..r].

  Calcule el índice q como parte de este procedimiento de partición.
- **Conquistar:** Ordene los dos subarreglos A[p..q-1] y A[q+1..r] por llamadas recursivas a QuickSort.
- **Combinar:** Debido a que los subarreglos ya están ordenados, no es necesario trabajar para combinarlos. El arreglo completo A[p..r] ahora está ordenado.

- El siguiente procedimiento implementa QuickSort:

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

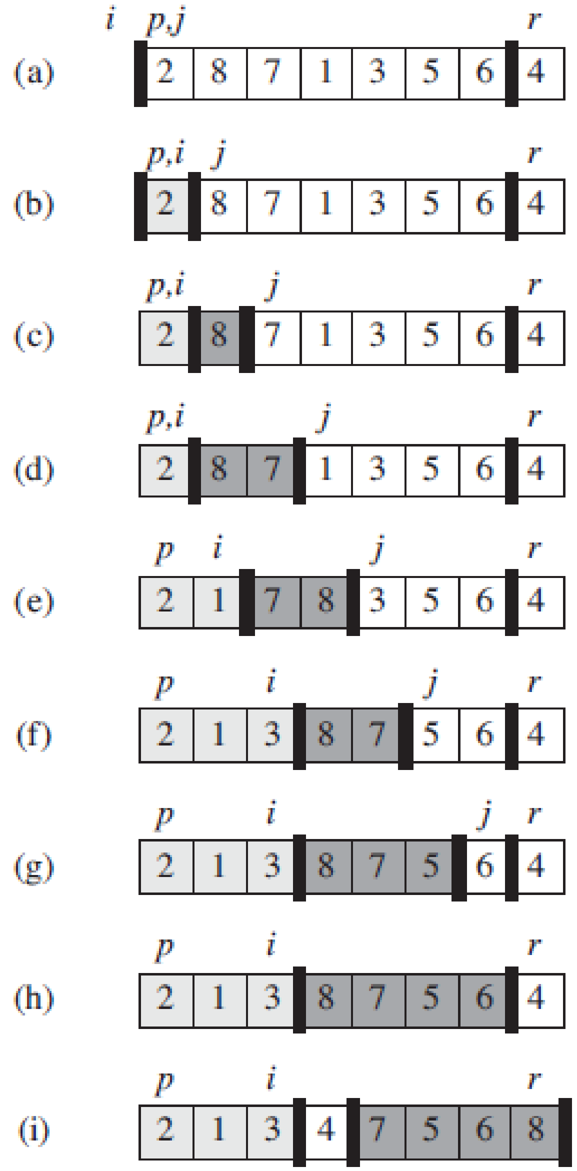
- Para ordenar un arreglo completo A, la llamada inicial es:

QUICKSORT(A, 1, A.length)

Particionando el arreglo

- La clave del algoritmo es el procedimiento de PARTICIÓN, el cual reorganiza el subarreglo A[p..r] en su lugar.

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



PARTITION(A, p, r)

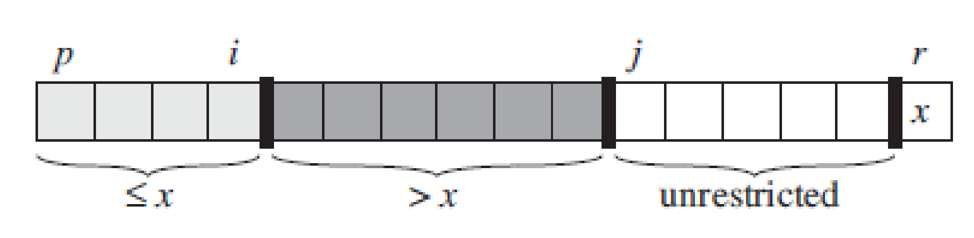
```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Esta imagen muestra cómo funciona la PARTICIÓN en un arreglo de 8 elementos.

PARTITION siempre selecciona un elemento **x = A[r]** como elemento pivote alrededor del cual se dividirá el subarreglo A[p..r].

The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

- A medida que se ejecuta el procedimiento, divide el arreglo en cuatro regiones (posiblemente vacías).
- Al comienzo de cada iteración del ciclo for en las líneas 3 a 6, las regiones satisfacen ciertas propiedades, que se muestran en la siguiente figura.



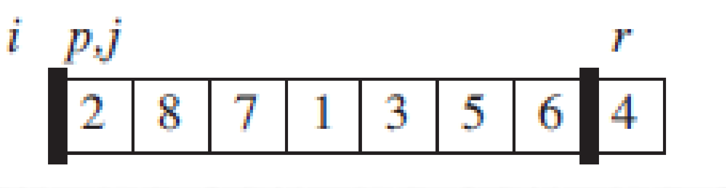
The four regions maintained by the procedure PARTITION on a subarray $A[p \,..\, r]$. The values in $A[p \,..\, i]$ are all less than or equal to x, the values in $A[i+1 \,..\, j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j \,..\, r-1]$ can take on any values.

- Declaramos estas propiedades como un bucle invariante:

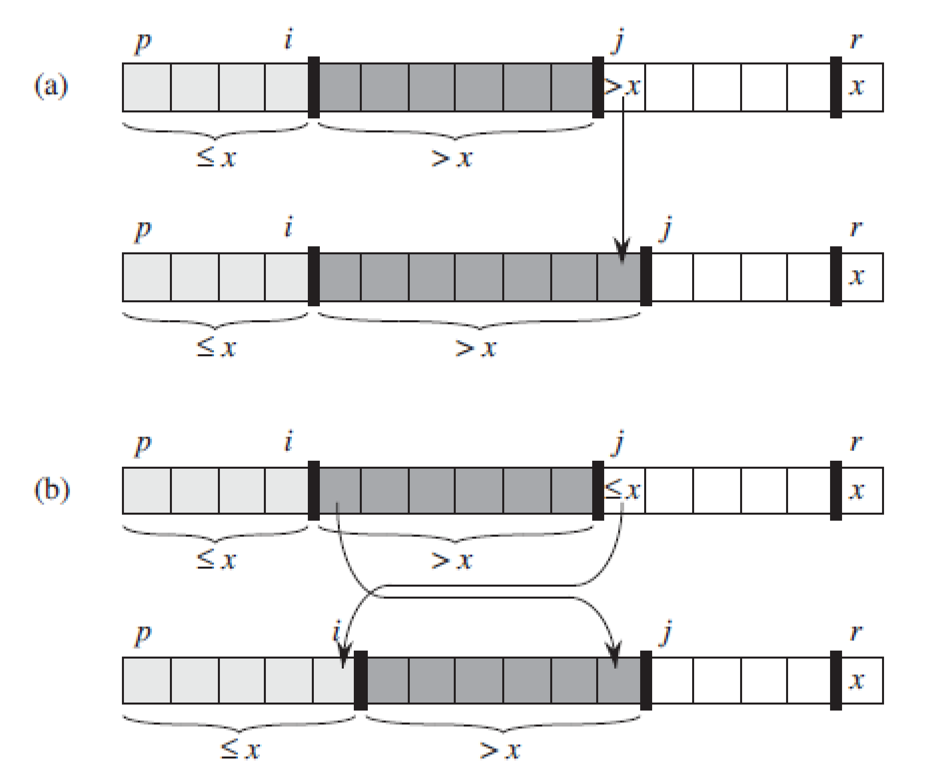
Al comienzo de cada iteración del ciclo de las líneas 3 a 6, para cualquier arreglo en el índice k:

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

- Los índices entre **j** y **r-1** no están cubiertos por ninguno de los tres casos, y los valores de estas entradas no tienen una relación particular con el pivote **x**.
- Necesitamos mostrar que este bucle invariante es verdadero antes de la primera iteración, y que cada iteración del ciclo se mantiene invariante, y que esta invariante proporciona una propiedad útil para mostrar la exactitud cuando el ciclo termina.
- **Inicialización:**
- Antes de la primera iteración del bucle, **i = p – 1** y **j = p**. Porque ningún valor se encuentra entre **p** e **i** y ningún valor se encuentra entre **i + 1** y **j – 1**.
- Las dos primeras condiciones del bucle invariante se satisfacen trivialmente.
- La asignación en la línea 1 satisface la tercera condición.



- **Mantenimiento:**
- Como se muestra en la siguiente figura, consideramos dos casos, dependiendo del resultado en la línea 4.
- La figura **(a)** muestra lo que sucede cuando **A[j] > x**; la única acción en el ciclo es incrementar el valor de **j**.
- Después de que **j** se incrementa, la condición 2 se mantiene para **A[j-1]** y todas las demás entradas permanecen sin cambios.
- La figura **(b)** muestra qué sucede cuando **A[j] ≤ x**; el bucle incrementa **i**, intercambia A[i] y A[j], y luego incrementa **j**.
- Debido al intercambio, ahora tenemos ese **A[i] ≤ x**, y se cumple la condición 1.
- De manera similar, también tenemos que **A[j-1] > x**, ya que el elemento que se intercambió en **A[j - 1]** es, por el ciclo invariante, mayor que **x**.



The two cases for one iteration of procedure PARTITION. (a) If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. (b) If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

- **Termino:**
- Al finalizar, **j = r**. Por lo tanto, cada entrada en el arreglo está en uno de los tres conjuntos descritos, y hemos dividido los valores en el arreglo en tres conjuntos: los menores o iguales a **x**, los mayores que **x**, y un conjunto que contiene **x**.

- Las dos últimas líneas de PARTICIÓN intercambian el elemento pivote con el elemento más a la izquierda mayor que x, moviendo así el pivote a su lugar correcto en el arreglo particionado, y luego devuelve el nuevo índice del pivote.
- La salida de PARTICIÓN ahora satisface las especificaciones dadas para el paso de dividir.
- De hecho, satisface una condición ligeramente más fuerte: después de la línea 2 de QUICKSORT, **A[q]** es estrictamente menor que todos los elementos en **A[q + 1..r]**.
- El tiempo de ejecución de PARTICIÓN en el subarreglo **A[p..r]** es **Θ(n)** donde:

$$n = r - p + 1.$$