uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 7 Divide y Vencerás

Profesores EDA

Objetivos

- Conocer el enfoque de divide y vencerás.
- Desarrollar algoritmos de divide y vencerás para resolver problemas.
- Conocer importantes algoritmos basados en este enfoque: mergesort y quicksort.

Índice

- Introducción
- Divide y Vencerás
  - Ejemplos
    - Encontrar mayor elemento en un array.
    - Mergesort
    - Quicksort

Introducción

- Un **algoritmo** es un conjunto finito de pasos para resolver un problema.
- Una **Estrategia algorítmica**
  - Enfoque para resolver un problema.
  - Puede combinar distintos enfoques.

Introducción

- Principales estrategias algorítmicas:
  - Algoritmos de Fuerza Bruta
  - Algoritmos recursivos
  - **Algoritmos Divide y Vencerás**
  - Algoritmos Backtracking
  - Programación Dinámica
  - Algoritmos Voraces
  - Heurísticas

Índice

- Introducción
- **Divide y Vencerás**
  - Ejemplos
    - Encontrar mayor elemento en un array.
    - Mergesort
    - Quicksort

Divide y Vencerás

Enfoque en tres pasos:

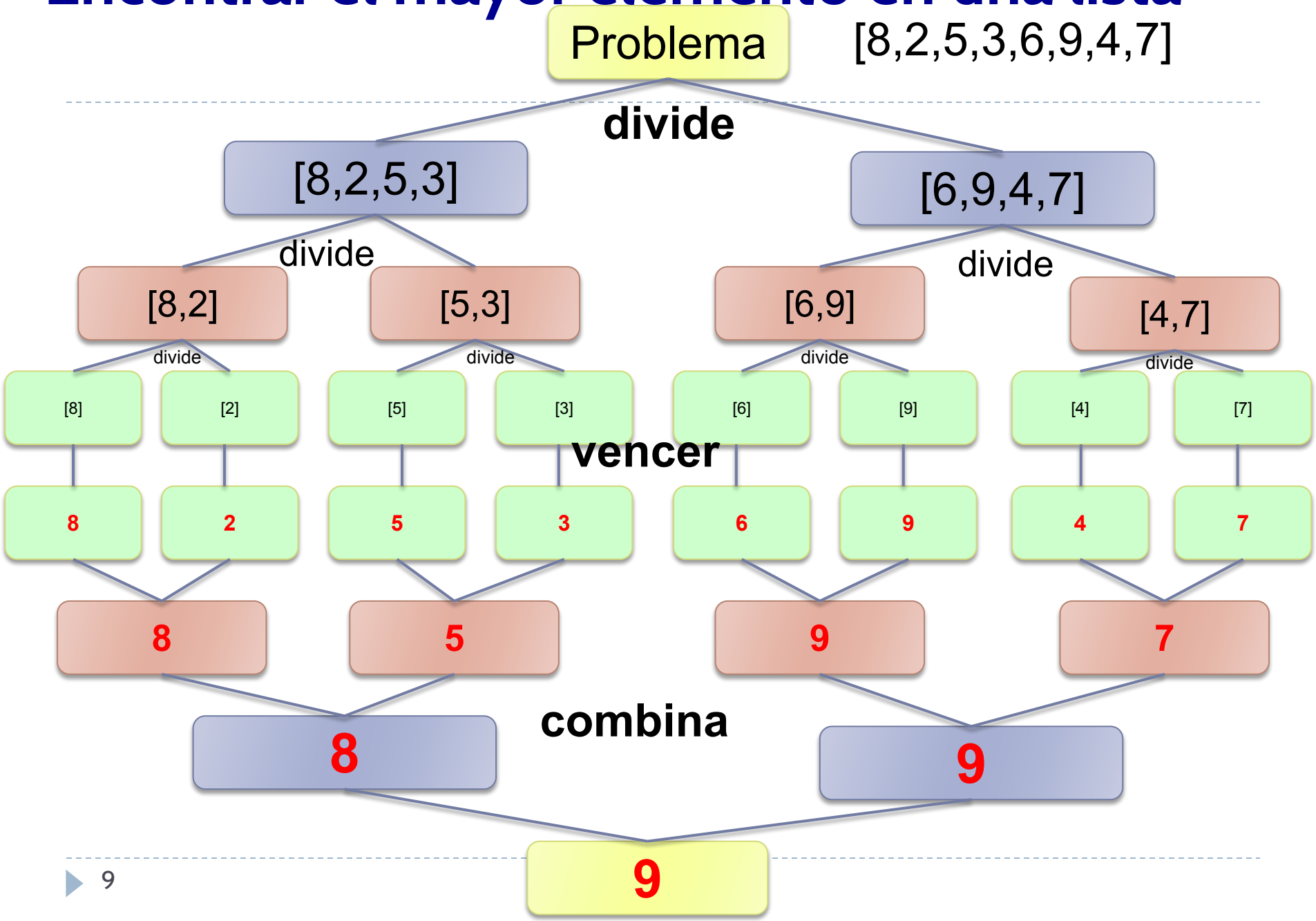
1) **Dividir**: divide el problema en subproblemas más pequeños del mismo tipo (que pueda resolver recursivamente)*
2) **Vencer**: resuelve cada subproblema.
3) **Combinar**: combina las soluciones de los subproblemas para resolver el problema original.

(*) normalmente contiene dos o más llamadas recursivas

Divide y vencerás: ejemplos

1. **Encontrar el máximo elemento en una lista**
2. Merge-sort
3. Quick-sort

Encontrar el mayor elemento en una lista



Encontrar el elemento mayor

```
Algorithm find_max(data: list) -> int:

    if data==None or len(data)==0:
       return None
    if len(data)==1:
       return data[0]

    mid = len(data)//2
    max1=find_max(data[0:mid])
    max2=find_max(data[mid:])
    return max(max1,max2)
```

Encontrar el elemento mayor en una lista

- La solución anterior está basada en divide y vencerás, y resuelve el problema.
- Un inconveniente de esta solución es que en cada llamada recursiva se crea una nueva sublista (A[0:m] o A[m:]), duplicando información que ya está en la lista original.
- **Ejercicio**: Piensa una solución también basada en divide y vencerás, pero que sea más eficiente desde el punto de vista de complejidad espacial.

Divide y vencerás: ejemplos

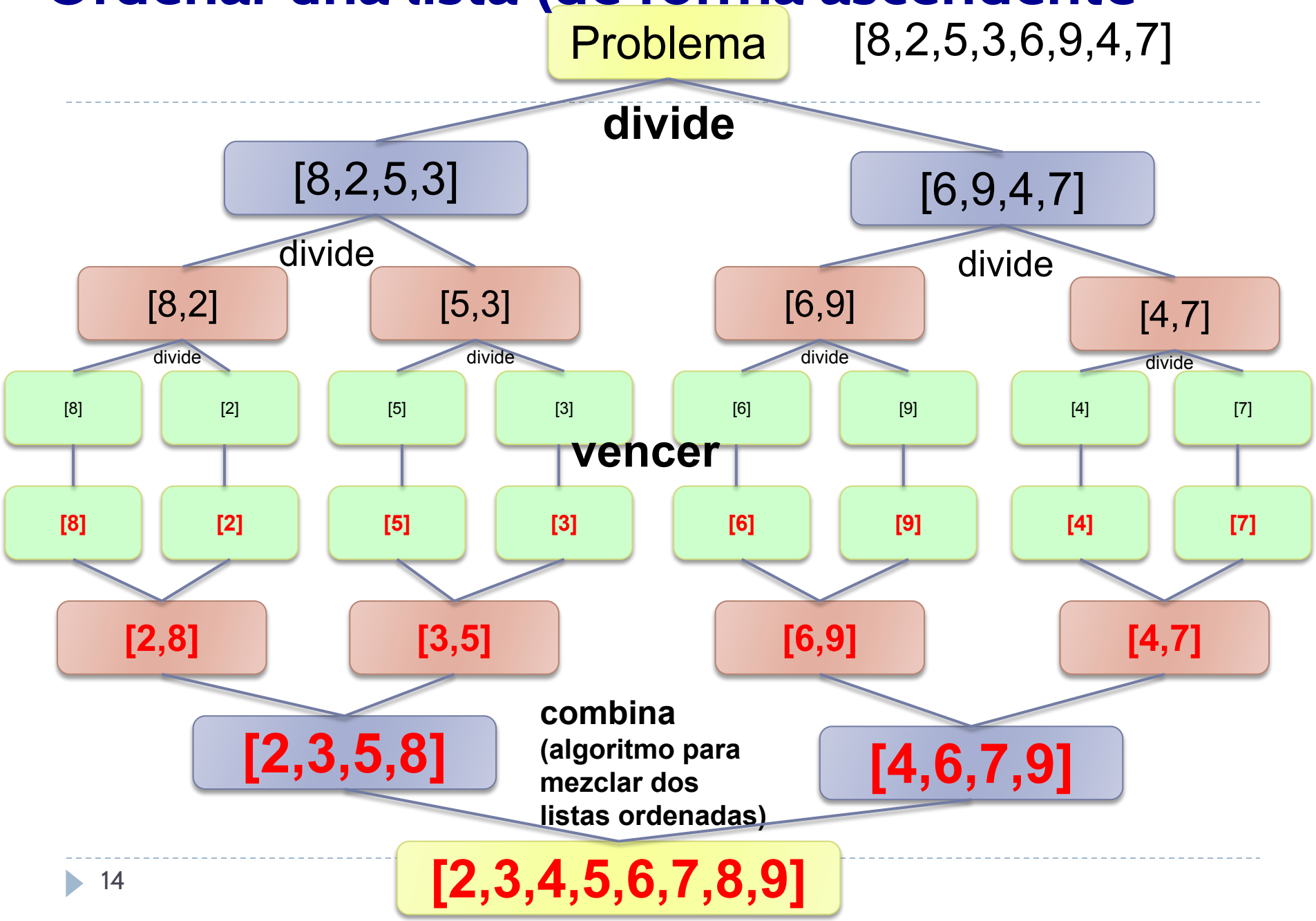
1. Encontrar el máximo elemento en una lista
2. **Merge-sort**
3. Quick-sort

Mergesort

Objetivo: ordenar una lista

1. ***Dividir***:
   - Dividir la lista en dos mitades.
   - Seguir dividiendo las listas mientras sea posibles (mientras la longitud de las listas es > 1).
2. ***Vencer***:
   - Las listas de longitud 1 (o vacías) ya están ordenadas!!!
3. ***Combinar***:
   - Necesitamos mezclar las sublistas ordenadas creando una lista ordenada.
     - Repetimos el proceso hasta conseguir una única lista, que será la solución del problema inicial.

Ordenar una lista (de forma ascendente



Mergesort

```
Algorithm mergesort(A: list):
   if len(A)==1:
       return A
   else:
     m=len(A)//2
     left=A[0:m]
     right=A[m:]

     sorted_left=mergesort(left)
     sorted_right=mergesort(right)

     A=merge(sorted_left,sorted_right)
      return A
```

Mergesort

```
Algorithm merge(l1,l2):
  newList=[]
  i=0
  j=0
  while i<len(l1) and j<len(l2):
    if l1[i]<=l2[j]:
      newList.append(l1[i])
      i+=1
    else:
      newList.append(l2[j])
      j+=1

  while i<len(l1):
    newList.append(l1[i])
    i+=1


  while j<len(l2):
    newList.append(l2[j])
    j+=1

  return newList
```

Mergesort

```
Algorithm merge(l1,l2):
  newList=[]
  i=0
  j=0
  while i<len(l1) and j<len(l2):
    if l1[i]<=l2[j]:
      newList.append(l1[i])
      i+=1
    else:
      newList.append(l2[j])
      j+=1

  while i<len(l1):
    newList.append(l1[i])
    i+=1


  while j<len(l2):
    newList.append(l2[j])
    j+=1

  return newList
```

¿Cuál es la complejidad de este algoritmo?

Mergesort

Complejidad temporal:

- ✓ La lista se va a dividir en dos mitades, en cada llamada recursiva
  - ○ n, n/2, n/2^2, n/2^3,..., n/2^k
  - ○ Después de k pasos, se llega al caso base caso base, n/2^k= 1
  - ○ En total, hay k = log(n) llamadas recursivas.
- ✓ En cada llamada recursiva, se aplica el algoritmo merge tiene complejidad temporal (O(n)).

Por tanto, la complejidad del método es **O(n*log n)**

Mergesort

Una demo online que os permitirá visualizar cómo funciona el algoritmo merge-sort

https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

For example: 7,3,2,1,9,6,4

Mergesort

Algunos vídeos divertidos:

https://www.youtube.com/watch?v=XaqR3G_NVoo

https://www.youtube.com/watch?v=JSceec-wEyw

Quicksort

Objetivo: Ordenar una lista (de menor a mayor).

- ¿Cómo **dividir** la lista?
  - **Elegir** un elemento **pivote**, para crear dos particiones de la lista.
  - En el siguiente ejemplo, seleccionamos el elemento central de la lista como pivote.

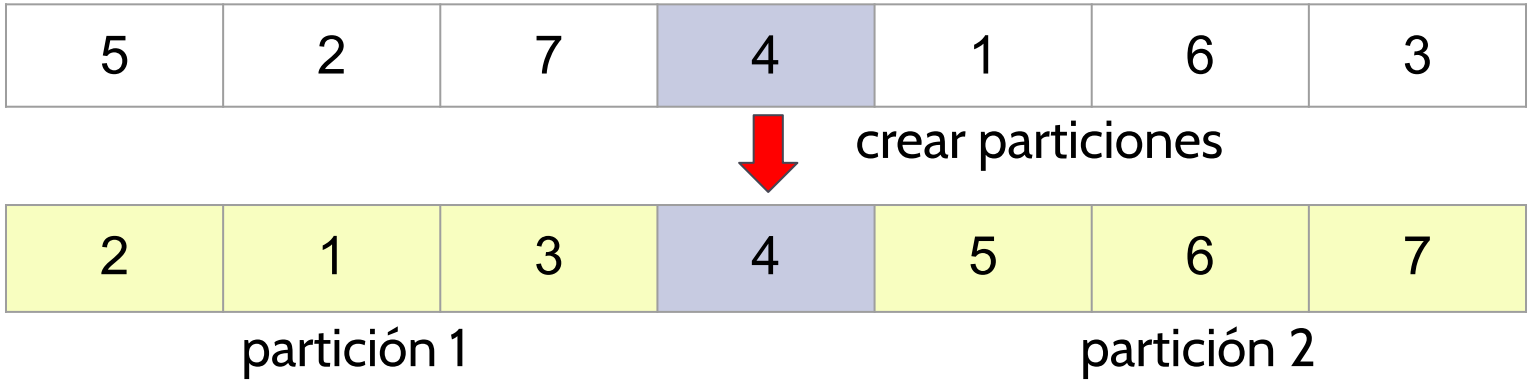
5	2	7	4	1	6	3

Quicksort

Las **particiones** se realizan de tal forma que todos los elementos

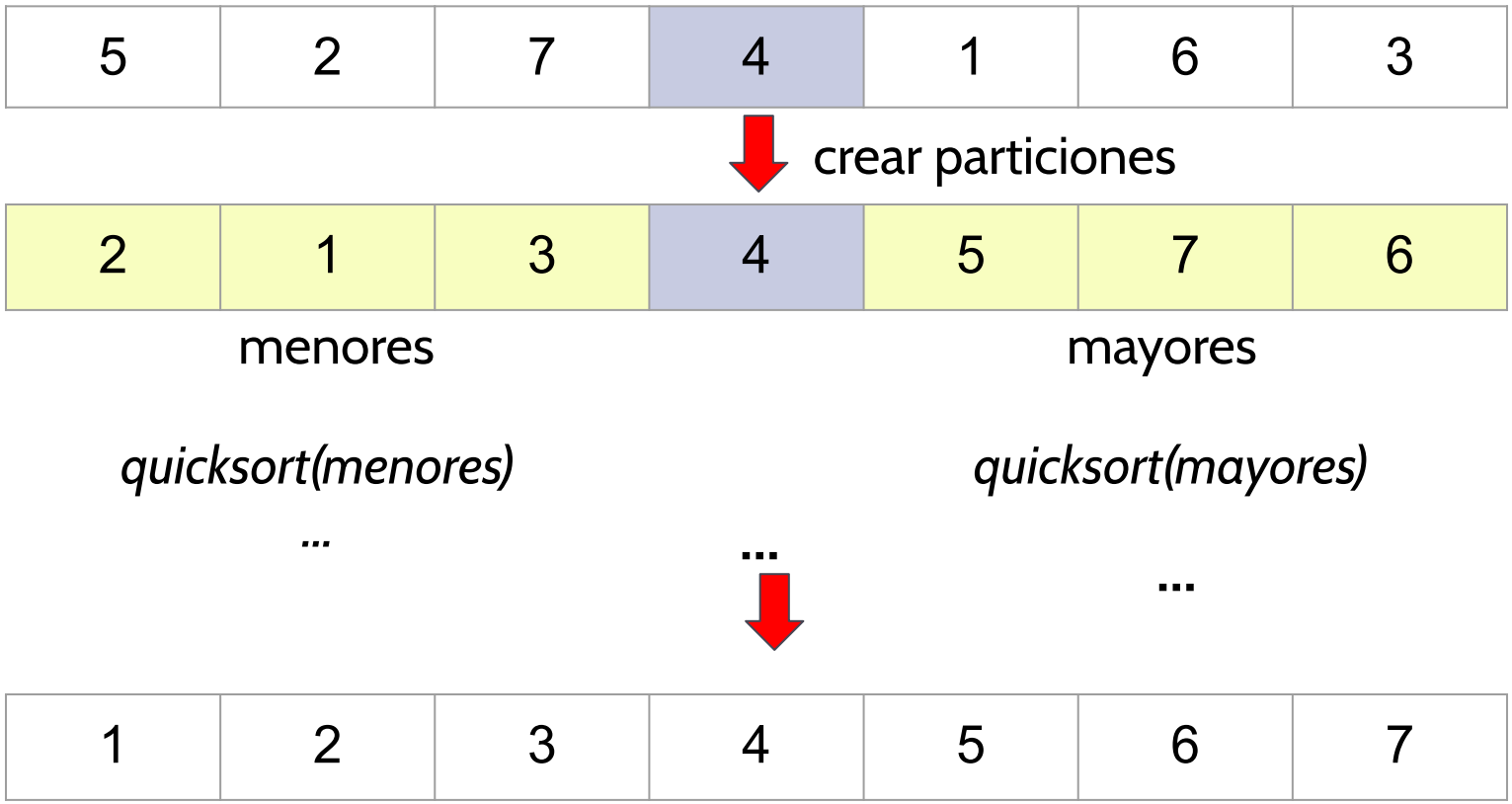
- **menores** que el **pivote** deben estar a su izquierda
- **mayores** que el **pivote** deben estar a su **derecha**

El **pivote ya estará colocado** en su posición correcta en la lista ordenada.



Quicksort

Ahora se aplicará quicksort, sobre cada una de las sublistas



Quicksort

En el ejemplo anterior, la estrategia era seleccionar el elemento central como pivote, pero existen otras estrategias:

- último elemento
- primer elemento
- pivote aleatorio

Quicksort - pivote último elemento

crear particiones
(dividir)

menores que el pivote:

mayores que el pivote:

Quicksort - pivote último elemento

crear particiones (dividir)

menores que el pivote:

mayores que el pivote:


```
pivote = lista[-1]
menores, mayores= [],[]
Para cada x in lista (excluyendo al pivote):
   if x <= pivote:
        menores.append(x)
   else:
       mayores.append(x)
```

Quicksort - pivote último elemento

crear particiones
(dividir)

menores que el pivote:

mayores que el pivote:

combinar

quicksort(

) + 

+ quicksort(

)

Quicksort - pivote primer elemento

crear particiones (dividir)

menores que el pivote:

mayores que el pivote:


```
pivote = lista[0]
menores, mayores= [],[]
Para cada x in lista (excluyendo al pivote):
   if x <= pivote:
         menores.append(x)
   else:
       mayores.append(x)
```

Quicksort - pivote primer elemento

crear particiones
(dividir)

menores que el pivote:

mayores que el pivote:

combinar

quicksort(

) + 

+ quicksort(

)

Quicksort - pivote primer elemento

Implementa una funcion, **crear_particiones**, que reciba una lista, elija su primer elemento como pivote, y devuelva los siguientes datos:

- una lista con los elementos menores o igual que el pivote
- el pivote
- una lista con los elementos mayores que el pivote.

Quicksort - pivote primer elemento

```
def partitions(a: list) -> (list, int, list):
    pivote = a[0]
    menores = []
    mayores = []
    for i in range(1, len(a)):
        if a[i] <= pivote:
            menores.append(a[i])
        else:
            mayores.append(a[i])
    return menores, pivote, mayores
```

Quicksort - crear_particiones

Problema 1: Implementa la función, **crear_particiones**, seleccionando como pivote el último elemento de la lista.

Problema 2: Implementa la función, **crear_particiones**, seleccionando un pivote aleatorio (cualquier elemento de la lista).

Soluciones

Quicksort

Implementa el algoritmo quicksort (ordenar una lista de forma ascendente).

Pistas:

- ¿Cuál es el caso base para ordenar una lista?

Quicksort

Implementa el algoritmo quicksort (ordenar una lista de forma ascendente).

Pistas:

- ¿Cuál es el caso base para ordenar una lista?. **lista vacía o con un único elemento.**

Quicksort

```
def quicksort(a: list) -> list:
    if len(a) <= 1:     # Caso Base
        return a
    …
```

Quicksort

Implementa el algoritmo quicksort (ordenar una lista de forma ascendente).

El caso base es cuando la lista es vacía o tiene un único elemento.

Para una lista con tamaño > 1, utiliza el método crear_particiones

Quicksort

```
def quicksort(a: list) -> list:
    if len(a) <= 1:
        return a

    menores, pivote, mayores =
crear particiones(a)
    result = []
    result.extend(menores)
    result.append(pivote)
    result.extend(mayores)
    return result
```

Quicksort

- Las soluciones anteriores son poco eficientes desde el punto de vista de complejidad espacial, porque al crear las particiones menores y mayores, estamos duplicando información que ya existe en la lista a.
- A continuación veremos un algoritmos que nos permite crear las particiones dentro de la propia lista, sin necesidad de crear nuevas sublistas, simplemente utilizando dos parámetros que nos delimitan la partición sobre la que estamos trabajando dentro de la lista: start y left

Quicksort

```
def quicksort(a: list):
    if a!=None and len(a)>1:
        _quicksort(a,0,len(a)-1)


def _quicksort(a: list, start: int, end:int):
    ...
```

La función principal incluye una llamada a una función auxiliar, _quicksort, que va a ser la función recursiva basada en divide y vencerás que realmente resuelve el problema. La función recibe además de la lista a ordenar, los índices donde comienza y termina la partición de la lista que va a ser ordenada en cada llamada. En la primera llamada, start será 0, y end, será la posición del último elemento del array: len(a)-1

Quicksort

```
def quicksort(a: list):
    if a!=None and len(a)>1:
        _quicksort(a,0,len(a)-1)


def _quicksort(a: list, start: int, end:int):
    ...
```

Vamos a estudiar el comportamiento del algoritmo _quicksoft para una lista concreta, comenzando por la primera llamada
_quicksort(a, start=0, end=6):

	0	1	2	3	4	5	6
a:	5	2	7	4	1	6	3

quicksort(a,start=0,end=len(a)-1)

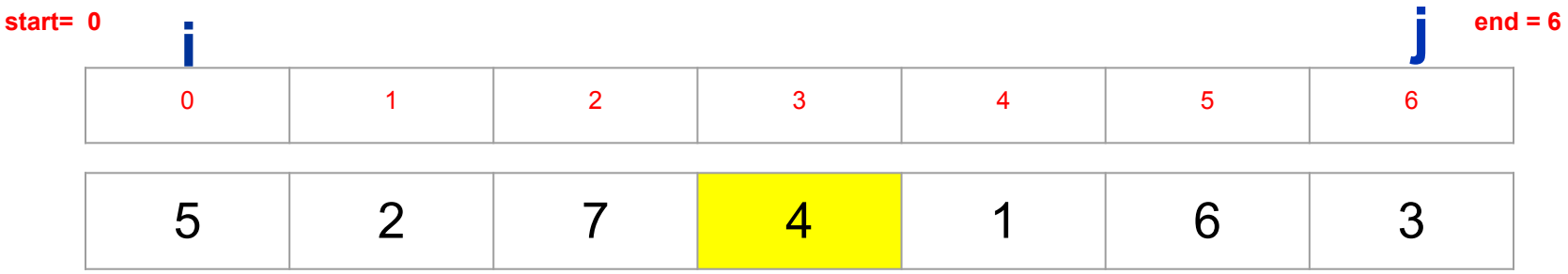
start= 0 **end = 6**

0	1	2	3	4	5	6
5	2	7	4	1	6	3

El primer paso del algoritmo será **seleccionar** el elemento **pivote** (primer elemento, último elemento, elemento central, elemento aleatorio, etc). En este caso, vamos a seleccionar el **elemento central** como pivote:

```
m=(start+end)//2= (0+6)//2 = 3
p=a[m]=a[3]=4
```

quicksort(a,start=0,end=len(a)-1)

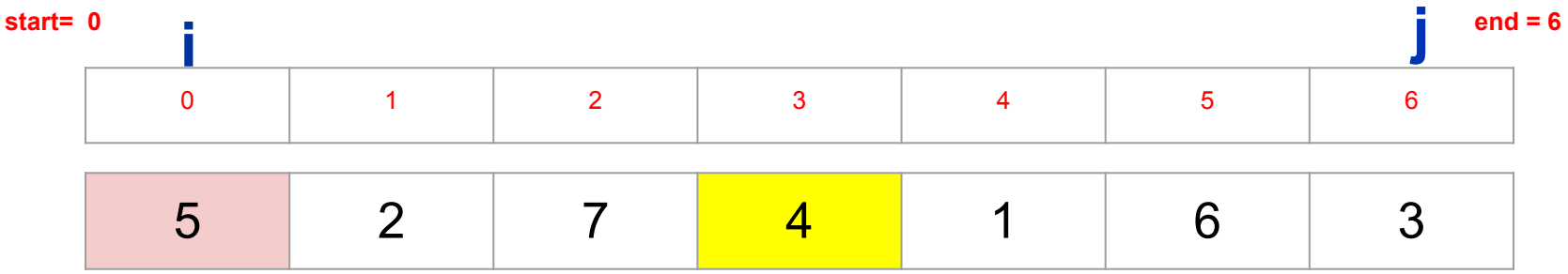


Como sabemos el algoritmo quicksort lo que busca es que los elementos menores que el pivote estén a su izquierda, y los elementos que sean mayores, estén a su derecha.
Para ello vamos a recorrer la partición (en este caso, es la lista completa) desde el principio (desde start) hasta el pivote, localizando aquellos elementos que son mayores que el pivote.
De la misma forma, también vamos a recorrer la partición (en este caso, es la lista completa) desde el final (desde end) hasta el pivote, localizando aquellos elementos que sean menores que el pivote.
Para hacer estos recorridos, vamos a declarar dos variables, i y j, que van a ser los valores de los índices de los elementos que vamos a visitar.

```
i = start
j = end
```

Mientras que a[i] sea menor que el pivote, deberemos avanzar i (i = i +1) para pasar al siguiente elemento, y pararemos cuando a[i] sea mayor o igual que el pivote.

quicksort(a,start=0,end=len(a)-1)



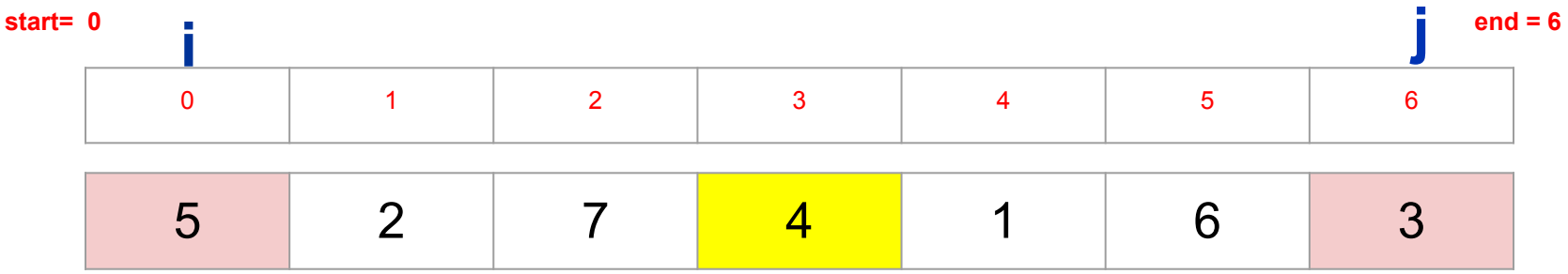
Mientras que a[i] sea menor que el pivote, deberemos avanzar i (i = i +1) para pasar al siguiente elemento, y pararemos cuando a[i] sea mayor o igual que el pivote.

```
while a[i] < p:
    i += 1
```

En nuestro ejemplo, la condición del bucle anterior, no se cumple, así que i no avanza.
Significa que el elemento que hay en i=0, a[0]=5, no debería estar en esa parte de la partición, sino que debería estar después del pivote.

Vamos a buscar un elemento que esté mal colocado en la parte derecha (es decir, un elemento más pequeño que el pivote pero después del pivote), para poder intercambiarlos.

quicksort(a,start=0,end=len(a)-1)



Para localizar los elementos menores al pivote pero que están colocados en la parte derecha de la partición (después del pivote), usaremos el siguiente bucle

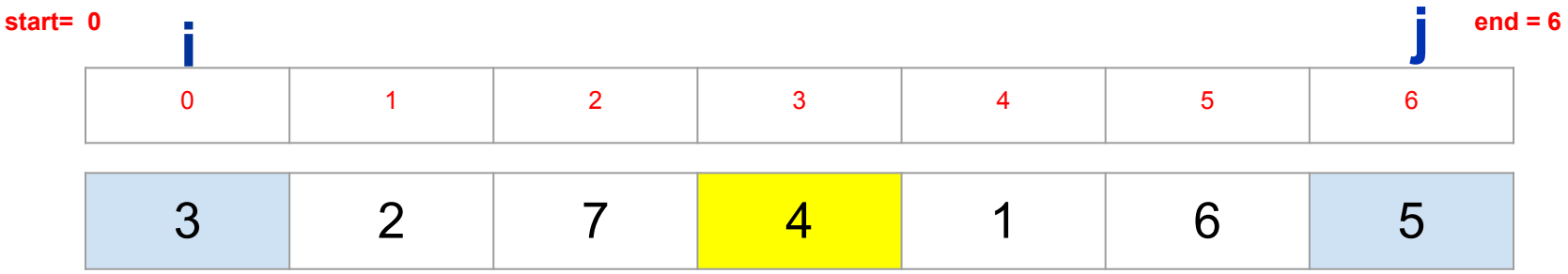
```
while a[j] > p:
    j -= 1
```

En nuestro ejemplo, la condición del bucle anterior, no se cumple, así que j no avanza.
Significa que el elemento que hay en j=6, a[6]=3, no debería estar en esa parte de la partición, sino que debería estar colocado antes del pivote.

Vamos a intercambiarlos:

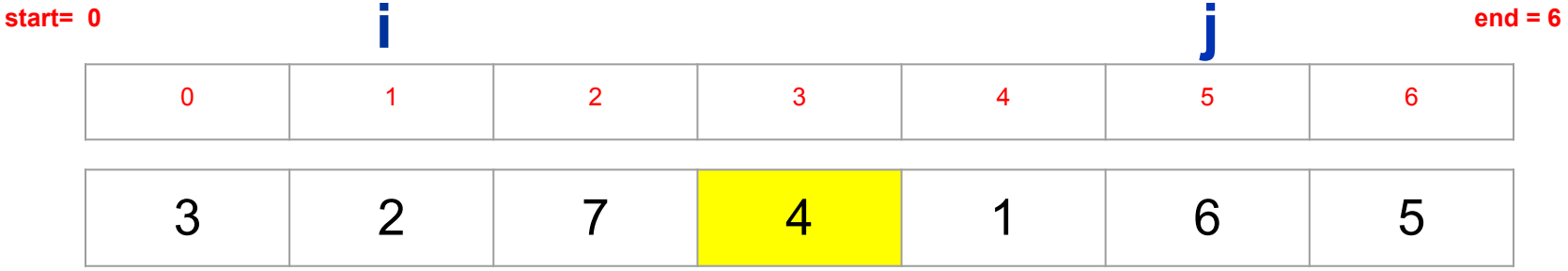
```
if i<=j:
    a[i], a[j] = a[j], a[i]
```

quicksort(a,start=0,end=len(a)-1)

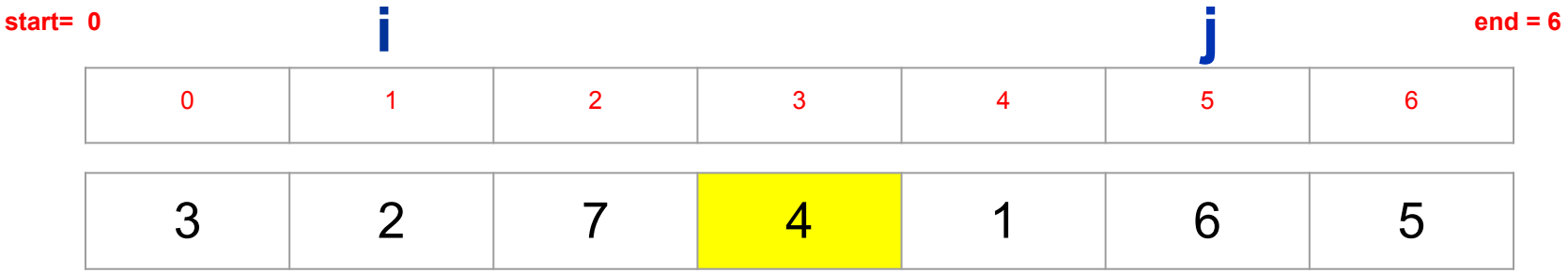


Después de haberlos intercambiado (ya están en la parte de la lista que les corresponde!!!), deberemos avanzar i (i=i+1) y j (j=j-1). En nuestro ejemplo, la condición del bucle anterior, no se cumple, así que j no avanza.

```
if i<=j:
    a[i], a[j] = a[j], a[i]
    i += 1
    j -= 1
```



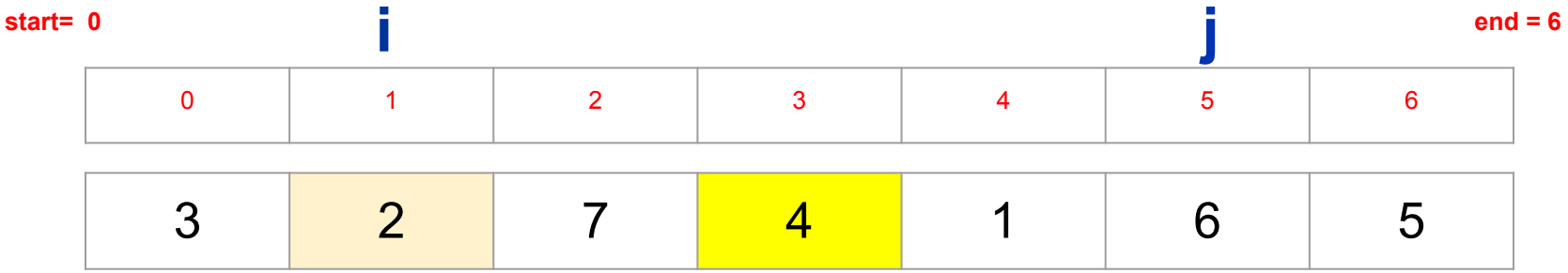
quicksort(a,start=0,end=len(a)-1)



Debemos seguir revisando los elementos que hay tanto antes del pivote, como después del pivote, para localizar el par de elementos que no están en su localización adecuada, y entonces intercambiarlos.
Este proceso se debe repetir mientras que i sea menor o igual que j.
Cuando j < i, significa que hemos revisado todos los elementos de ambas partes de la partición (antes y después del pivote)

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

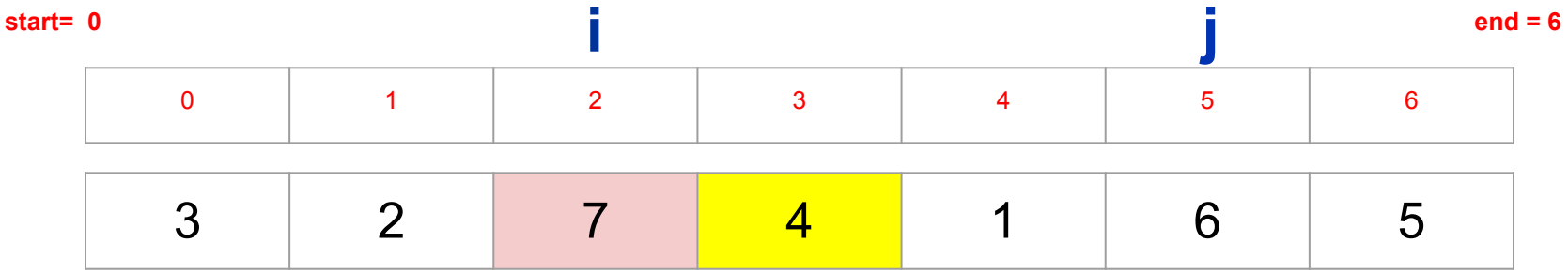
quicksort(a,start=0,end=len(a)-1)



Con i = 1, a[i]=2 es menor que el pivote, por tanto, debemos avanzar i (i = i +1 = 3)

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

quicksort(a,start=0,end=len(a)-1)

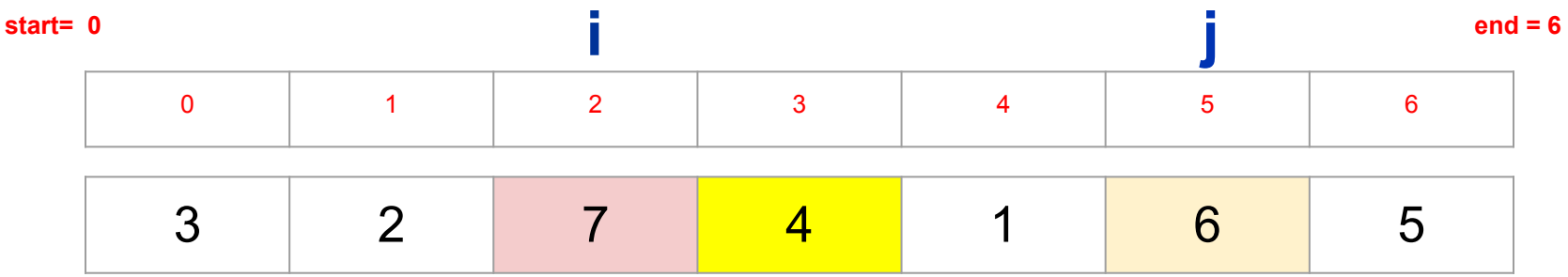


Con i = 2, a[i]=7 es mayor que el pivote. Ya hemos encontrado un elemento que deberemos intercambiar con otro elemento de la segunda parte. Como la condición del primer bucle anidado no se cumple, el algoritmo pasa al segundo bucle, que se ocupará de buscar un elemento en la segunda parte que son más pequeños que el pivote, y que no debería estar en esa segunda parte.

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

Para el bucle

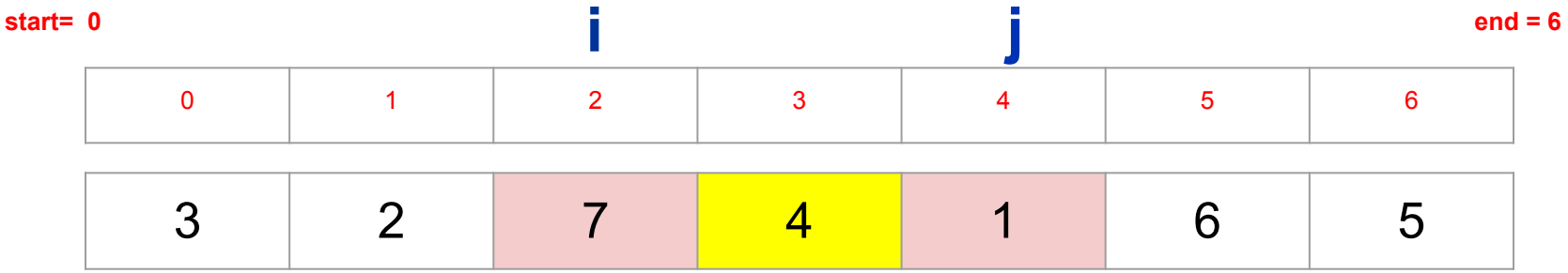
quicksort(a,start=0,end=len(a)-1)



Con j = 5, a[j]=6 es mayor que el pivote, es decir, es un elemento que está bien colocado a la derecha del pivote. Debemos continuar, avanzando j (j = j -1 = 4)

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

quicksort(a,start=0,end=len(a)-1)

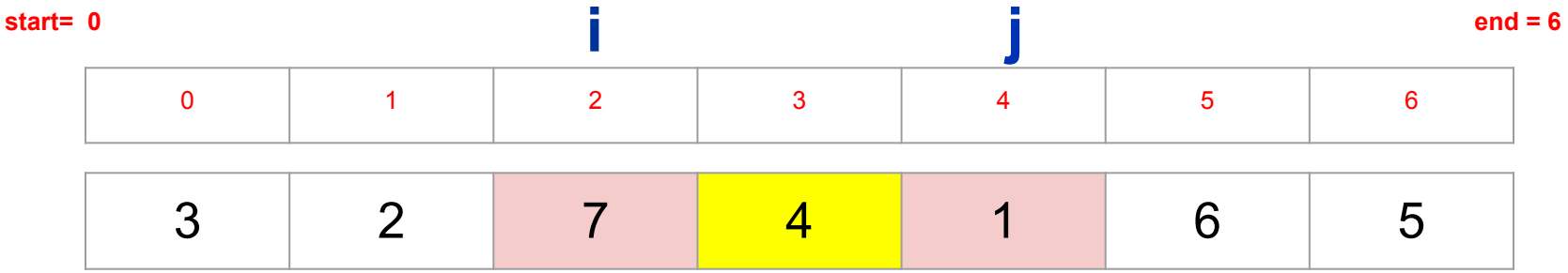


Con j = 4, a[j]=1 es menor que el pivote, es decir, ya hemos encontrado un elemento que no debería estar a la derecha del pivote.

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

Para el bucle

quicksort(a,start=0,end=len(a)-1)



El segundo bucle anidado no vuelve a ejecutarse, y debemos intercambiar los elementos que hay en i y j:

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

quicksort(a,start=0,end=len(a)-1)

start= 0 i j end = 6

0	1	2	3	4	5	6
3	2	1	4	7	6	5

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

quicksort(a,start=0,end=len(a)-1)

start= 0 i j end = 6

0	1	2	3	4	5	6
3	2	1	4	7	6	5

Después de intercambiar, avanzaremos i (i=i+1) y j (j=j-1)

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

quicksort(a,start=0,end=len(a)-1)

start= 0 i j end = 6

0	1	2	3	4	5	6
3	2	1	4	7	6	5

Después de avanzar i y j, la condición de bucle principal se sigue cupliendo porque i=j. Por tanto, se vuelve a ejecutar.
Sin embargo, ninguno de los dos bucles anidados se ejecutarán (porque sus condiciones son False).
La condición del if también se cumple, y se hará un intercambio del mismo elemento por el mismo elemento. Por último, i y j, son modificados

```
while i<=j:
      while A[i]<p:
            i+=1
      while A[j]>p:
            j-=1
      if i<=j:
            A[i],A[j]=A[j],A[i]
            i+=1
            j-=1
```

start= 0 j i end = 6

0	1	2	3	4	5	6
3	2	1	4	7	6	5

quicksort(a,start=0,end=len(a)-1)

start= 0 j i end = 6

0	1	2	3	4	5	6
3	2	1	4	7	6	5

El bucle principal no vuelve a ejecutarse porque j > i.

Podemos ver que ahora todos los elementos menores al pivote están a su izquierda (primera partición desde start a j) y todos los elementos mayores están a su derecha (segunda partición desde i hasta end).

Estas particiones no tiene porque estar ordenadas.

```
while i<=j:
    while A[i]<p:
        i+=1
    while A[j]>p:
        j-=1
    if i<=j:
        A[i],A[j]=A[j],A[i]
        i+=1
        j-=1
```

quicksort(a,start=0,end=len(a)-1)

start= 0 j i end = 6

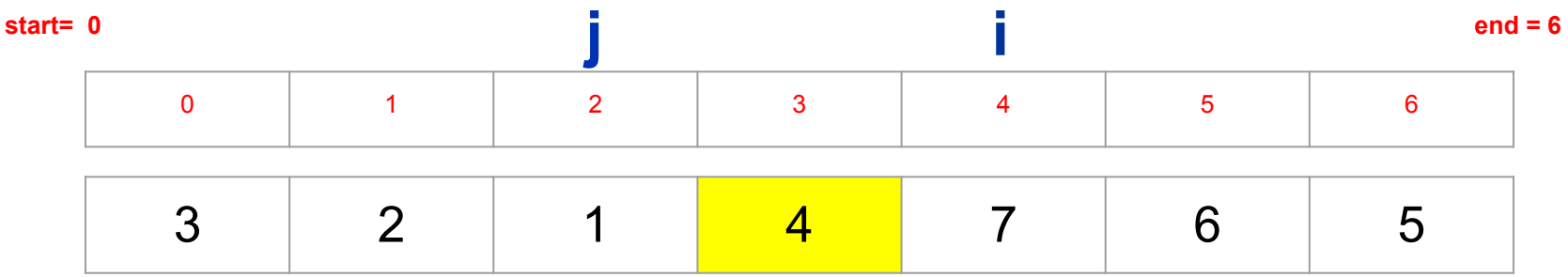
0	1	2	3	4	5	6
3	2	1	4	7	6	5

```
while i<=j:
     while A[i]<p:
          i+=1
     while A[j]>p:
          j-=1
     if i<=j:
          A[i],A[j]=A[j],A[i]
          i+=1
          j-=1
```

Para ordenar las particiones, debemos aplicar aplicando recursivamente el método _quicksort sobre cada una de ellas.
¿existe algún caso base?, ¿cuándo no es necesario volver a llamar a _quicksort sobre la partición?.

```
_quicksort(a, start, j)
_quicksoft(a, i, end)
```

quicksort(a,start=0,end=len(a)-1)



El caso base es que la partición no tenga elementos o tenga un único elemento.

Así por ejemplo, para la primera partición, si start > j o start == j, la partición estará vacía o tiene un único elemento, y ya está ordenada.

De forma similar, para la segunda partición, si i > end o i == end, esta partición está vacía o tiene un único elemento, y ya está ordenada.

```
while i<=j:
     while A[i]<p:
          i+=1
     while A[j]>p:
          j-=1
     if i<=j:
          A[i],A[j]=A[j],A[i]
          i+=1
          j-=1

if start < j:
     _quicksort(a, start, j)
if i < end:
     _quicksort(a, i, end)
```

quicksort(a,**start=0,end=2**)

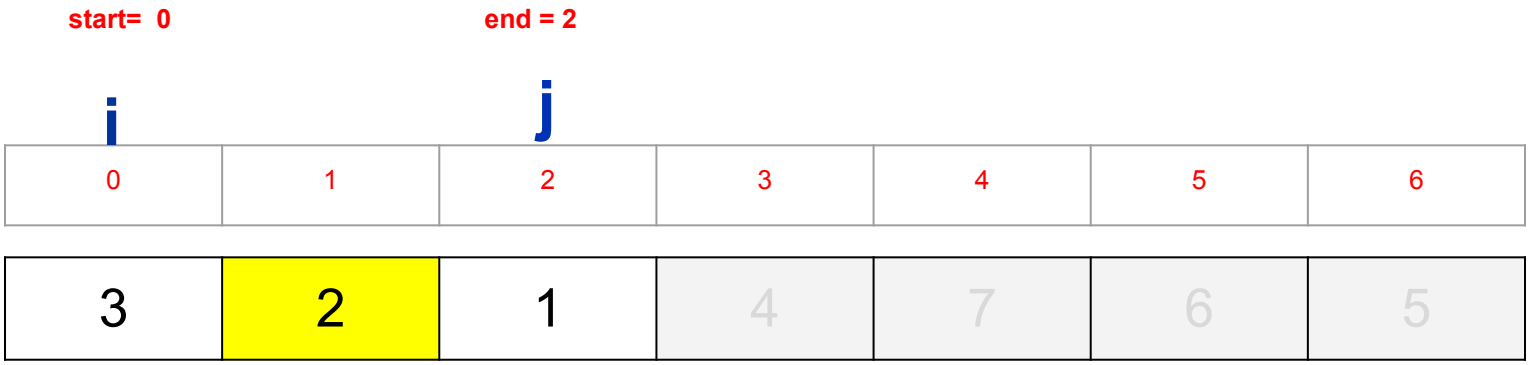
start= 0 end = 2

0	1	2	3	4	5	6
3	2	1	4	7	6	5

Vamos a centrarnos en la primera partición

_quicksort(a, 0, 2)

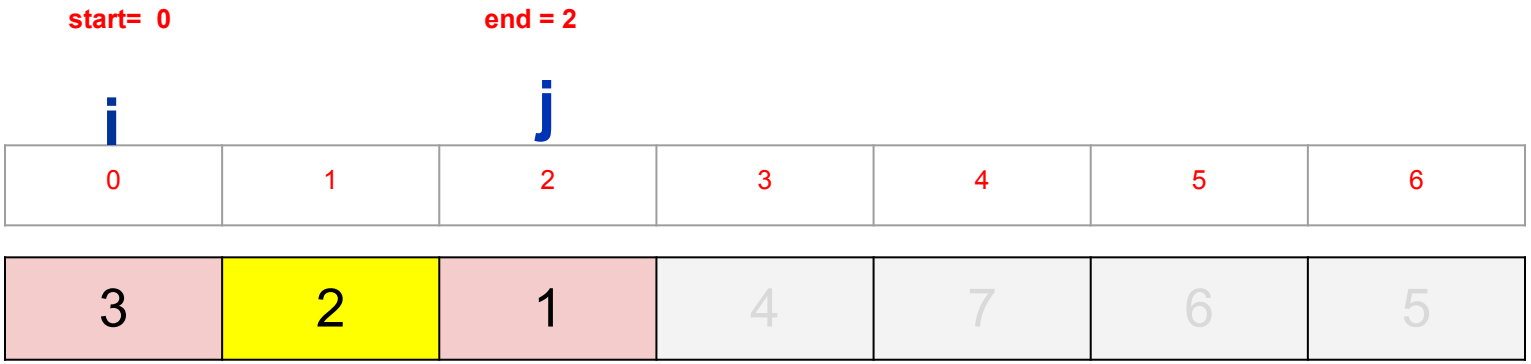
quicksort(a,start=0,end=2)



Seleccionamos el pivote
m = (start + end) // 2 = (0 + 2) // 2 = 1
p = a[m] = 2

Además, inicializamos i y j:
i, j = start, end

quicksort(a,start=0,end=2)

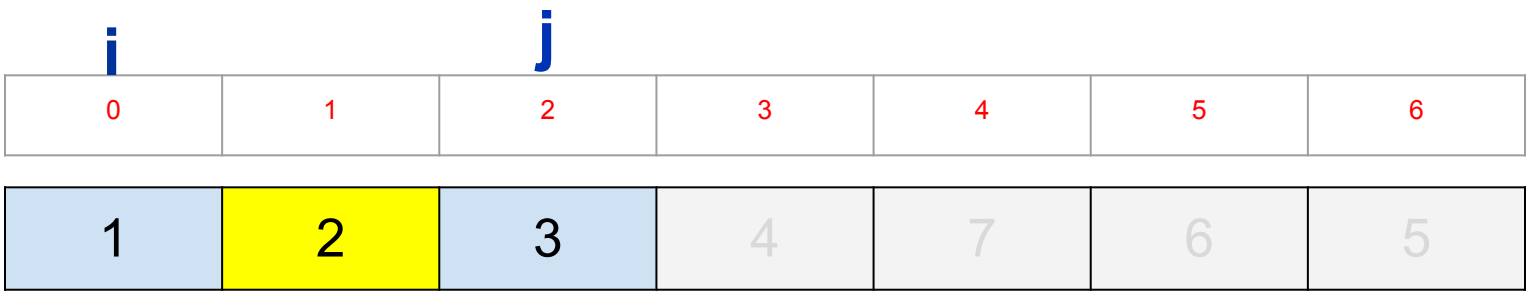



```
while i<=j:
      while A[i]<p:
            i+=1
      while A[j]>p:
            j-=1
      if i<=j:
            A[i],A[j]=A[j],A[i]
            i+=1
            j-=1
```

Ejecutamos ahora el bucle principal.
En el primer bucle anidado, i no avanza, porque a[0]=3 ya es mayor que el pivote.
En el segundo bucle anidado, j tampoco avanza, porque a[2]=1 es menor que el pivote.
Ambos elementos están en particiones que nos les corresponden.
El siguiente paso será intercambiarlos

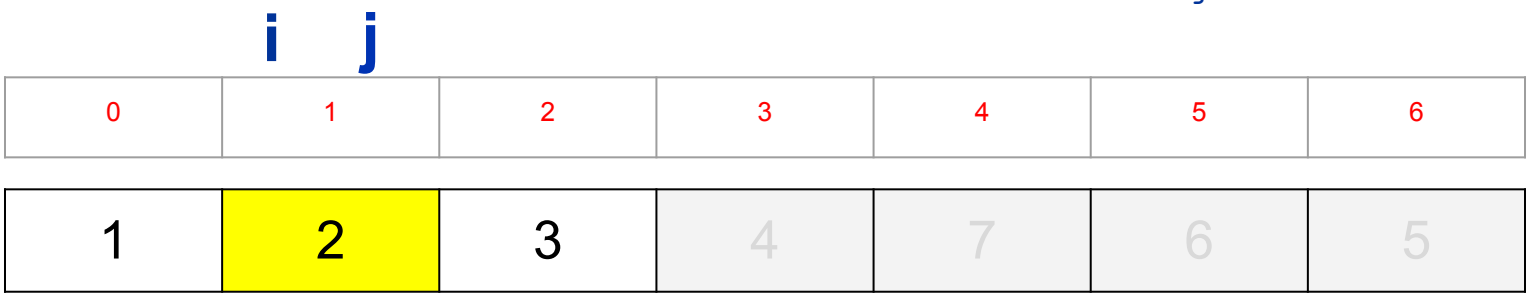
quicksort(a,start=0,end=2)

start= 0 end = 2

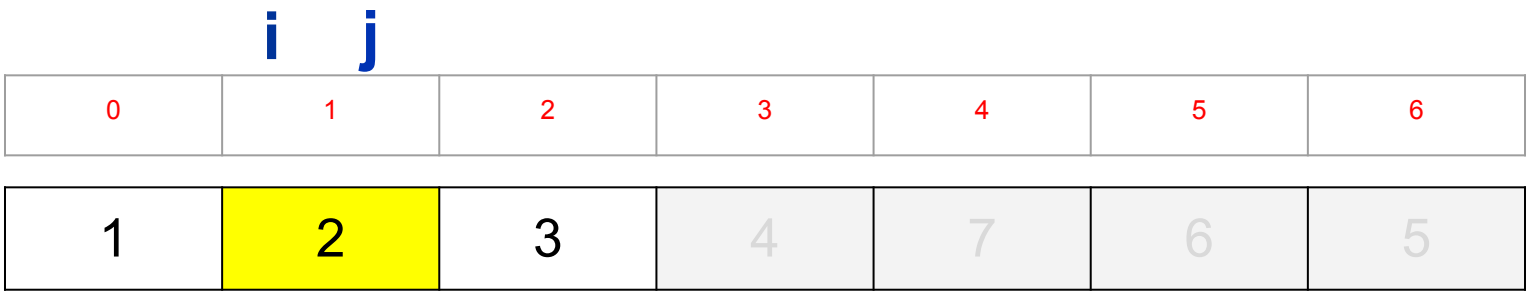


Una vez intercambiados, deberemos avanzar i y j

```
while i<=j:
      while A[i]<p:
            i+=1
      while A[j]>p:
            j-=1
      if i<=j:
            A[i],A[j]=A[j],A[i]
            i+=1
            j-=1
```

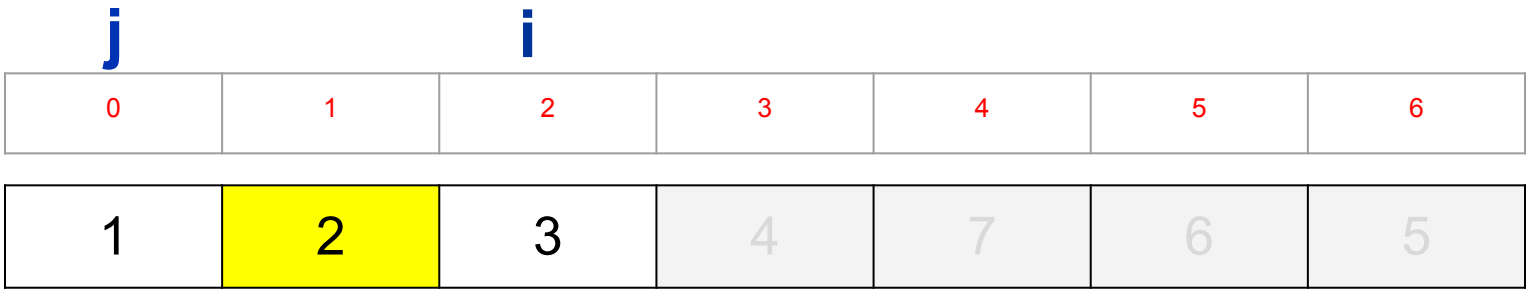


quicksort(a,**start=0,end=2**)

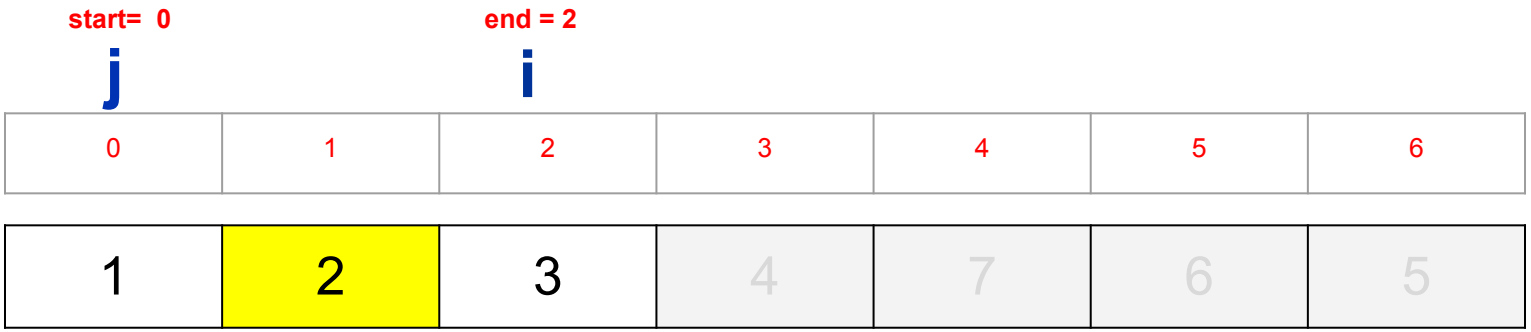


Como i=j, la condición de bucle principal sigue siendo cierta, y por tanto, su cuerpo se ejecutará.
Sin embargo, ninguno de los dos bucles anidados se ejecutarán (porque sus condiciones son False).
La condición del if también se cumple, y se hará un intercambio del mismo elemento por el mismo elemento. Por último, i y j, son modificados

```
while i<=j:
      while A[i]<p:
            i+=1
      while A[j]>p:
            j-=1
      if i<=j:
            A[i],A[j]=A[j],A[i]
            i+=1
            j-=1
```



quicksort(a,start=0,end=2)



Como i>j, la condición de bucle principal ya no se cumple.

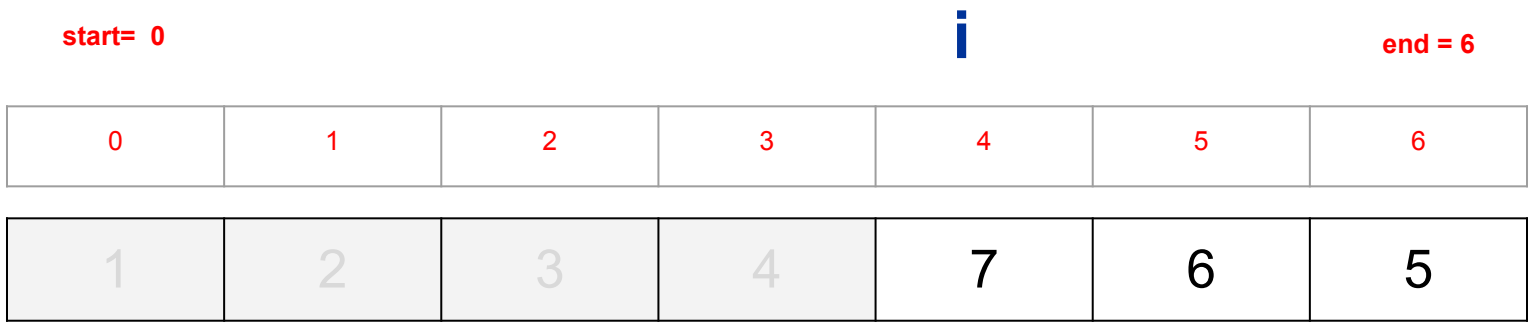
Pasaremos a las llamadas recursivas para ordenar las dos particiones creadas (desde start a j) y desde (i a end).
Sin embargo, podemos ver que ambas particiones son particiones con un único elemento, es decir, ya están ordenadas, y no es necesario realizar ninguna llamada recursiva.

Por tanto, ya habríamos terminado de ordenar la partición de la lista desde start=0 hasta end=2

```
while i<=j:
     while A[i]<p:
          i+=1
     while A[j]>p:
          j-=1
     if i<=j:
          A[i],A[j]=A[j],A[i]
          i+=1
     j-=1

if start < j:
     _quicksort(a, start, j)
if i < end:
     _quicksort(a, i, end)
```

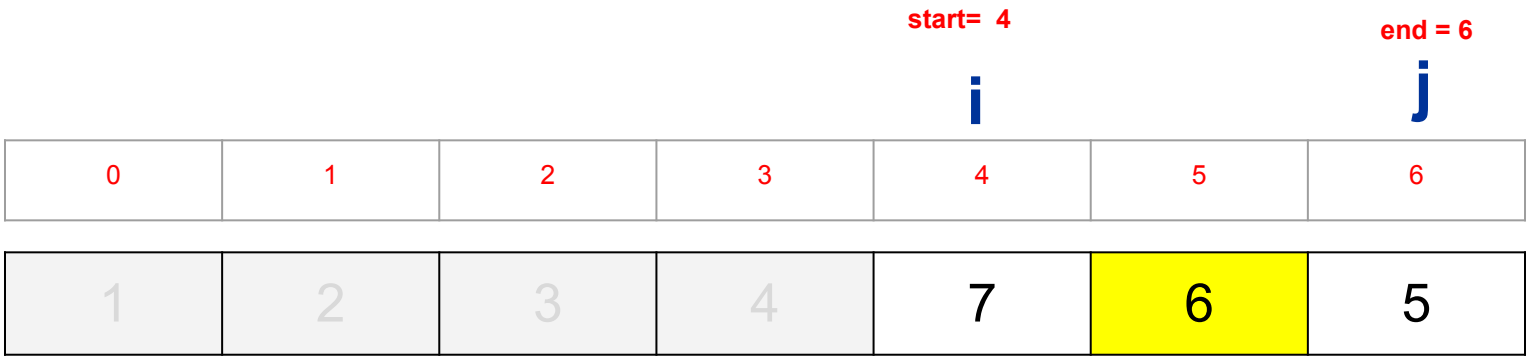
quicksort(a,**start=0,end=6**)



En la llamada del método _quicksort(a, start=0, end=6) (sobre toda la lista), ya habríamos terminado de ordenar la primera partición, pero aún nos queda ejecutar la segunda llamada recursiva sobre la segunda partición:

```
if start < j:
     _quicksort(a, start=0, j=2)
if i < end:
     _quicksort(a, i=4, end=6)
```

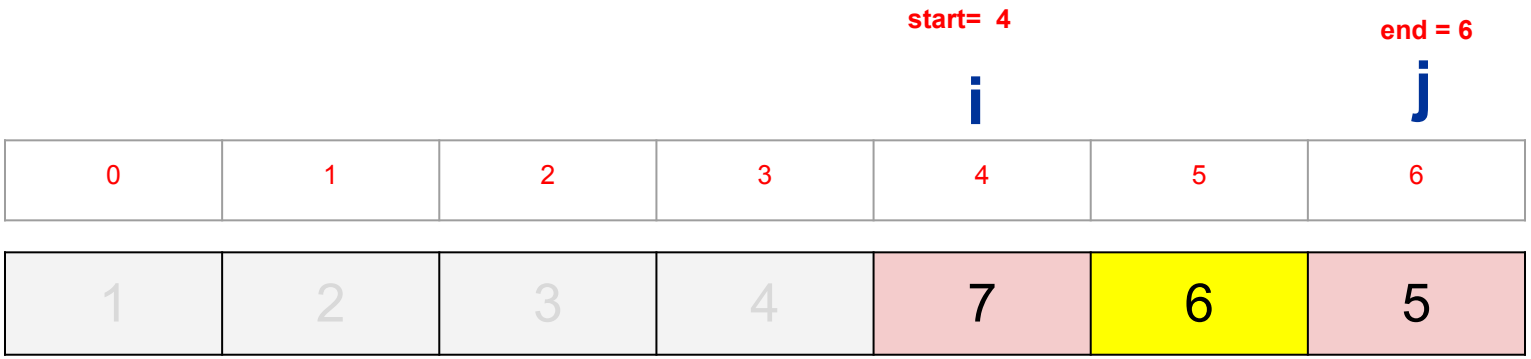
quicksort(a,**start=4,end=6**)



Seleccionamos el pivote
m = (start + end) // 2 = (4 + 6) // 2 = 5
p = a[m] = 6

Además, inicializamos i y j:
i, j = start, end

quicksort(a,**start=4,end=6**)

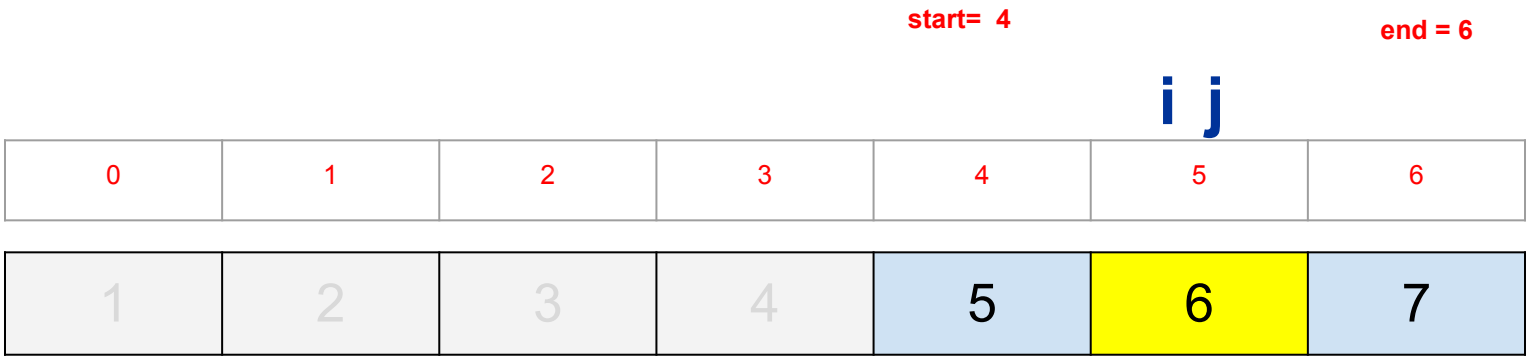


La condición del bucle principal se cumple.
i = 4, no avanza porque a[4]=7 es mayor que el pivote (6).
De la misma forma, j = 6, tampoco avanza porque a[6] es más pequeño que el pivote (6).
Debemos intercambiarlos

```
while i<=j:
      while A[i]<p:
            i+=1
      while A[j]>p:
            j-=1
      if i<=j:
            A[i],A[j]=A[j],A[i]
            i+=1
            j-=1

if start < j:
      _quicksort(a, start, j)
if i < end:
      _quicksort(a, i, end)
```

quicksort(a,**start=4,end=6**)

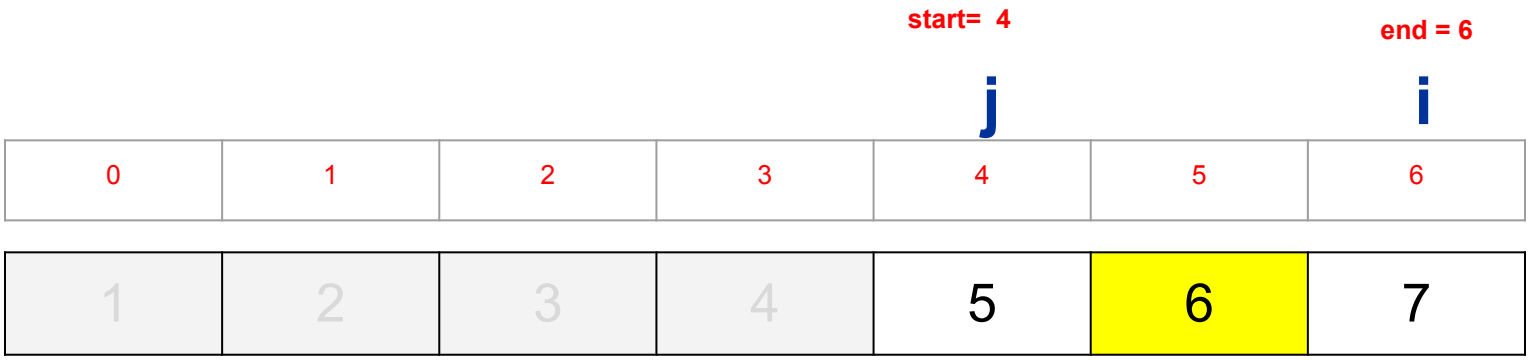


Una vez intercambiados, avanzaremos i y j.
El bucle principal volverá a ser ejecutado (porque i <= j), pero las condiciones de los bucles anidados se evaluaran a False, y no serán ejecutados.
La condición del if es cierta, así que se intercambiará el elemento por sí mismo, y se avanzará los índices.

```
while i<=j:
     while A[i]<p:
          i+=1
     while A[j]>p:
          j-=1
     if i<=j:
          A[i],A[j]=A[j],A[i]
          i+=1
          j-=1

if start < j:
     _quicksort(a, start, j)
if i < end:
     _quicksort(a, i, end)
```

quicksort(a,**start=4,end=6**)



La condición del bucle principal ya no se cumple (i > j).
El algoritmo salta a las llamadas recursivas, pero ninguna se ejecuta porque ambas particiones son particiones de un único elemento (caso base) y ya están ordenadas.

Por tanto, la ejecución de _quicksort(a, 4, 6), ya ha terminado.

```
while i<=j:
     while A[i]<p:
          i+=1
     while A[j]>p:
          j-=1
     if i<=j:
          A[i],A[j]=A[j],A[i]
          i+=1
          j-=1

if start < j:
     _quicksort(a, start, j)
if i < end:
     _quicksort(a, i, end)
```

quicksort(a,start=0,end=6)

start= 0

end = 6

0	1	2	3	4	5	6
1	2	3	4	5	6	7

El fin del ejecución de _quicksort(a, start=4, end=6), también termina la llamada al método principal _quicksort(a, 0, 6). Es decir, la lista ya está ordenada

```
if start < j:
     _quicksort(a, start=0, j=2)
if i < end:
     _quicksort(a, i=4, end=6)
```

0	1	2	3	4	5	6
1	2	3	4	7	6	5

Quicksort (pseudocódigo)

```
Algorithm quicksort(A: list):
    if A!=None and len(A)>1:
            _quicksort(A,0,len(A)-1)
```

Quicksort (pseudocódigo)

```
Algorithm _quicksort(a: list, start: int, end: int):
    m=(start + end) // 2
    p = a[m] # pivot element in the middle

    i = start
    j = end

    while i <= j:
        while A[i] < p:
          i += 1
        while A[j] > p:
          j -= 1
        if i <= j: # swap
            A[i], A[j] = A[j], A[i]
            i += 1
            j -= 1

    if start < j: # sort left list
        _quicksort(a, start, j)

    if i < end: # sort right list
        _quicksort(a, i, end)
```

Quicksort

Complejidad temporal:

- En cada partición, el espacio de búsqueda es dividido por la mitad: n, n/2, $n/2^2$, $n/2^3$,…, $n/2^k$ => k= log n
- Algoritmo de partición(mueve todos los elementos menores a la izquierda que el pivote y los mayores a su derecha) tiene complejidad lineal (O(n)).

Por tanto, O(n*log n)

Quicksort (demos y vídeos)

- https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/
- https://visualgo.net/bn/sorting?slide=1 (pivote primer elemento)
- http://www.cs.armstrong.edu/liang/animation/web/QuickSortPartition.html (pivote primer elemento)
- http://www.algostructure.com/sorting/quicksort.php (pivote primer elemento)

Quicksort (demos y vídeos)

- https://www.youtube.com/watch?v=ywWBy6J5gz8&t=121s **(pivote primer elemento**
- https://www.youtube.com/watch?v=UIBaYOUs8K8 **(pivote central)**
- https://www.youtube.com/watch?v=a4bPE8G5o9Q **(pivote central)**
- https://www.youtube.com/watch?v=biOjCLbdr7Y&t=37s (**pivote último elemento**)

¿Por qué la recursión es un buen amigo (a veces)?

- Divide y Vencerás consigue ordenar una lista con menor complejidad temporal (O(nlogn)) que otros algoritmos iterativos (por ejemplo, bubble sort), cuya complejidad es $O(n^2)$

Conclusiones

- Divide y vencerás es una estrategia recursiva que consiste en:
  - **Dividir** problemas en otros más pequeños (mediante recursión) hasta alcanzar un problema simple que podamos resolver directamente (**vencer**).
  - **Combinar** las soluciones intermedias para obtener la solución del problema inicial.
- El uso de divide y vencerás para ordenar una lista es capaz de obtener soluciones más eficientes (nlogn < n^2)