OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 7 Divide y Vencerás

Profesores EDA

cc BY NC SA

Objetivos

- Conocer el enfoque de divide y vencerás.
- Conocer importantes algoritmos basados en este enfoque: búsqueda binaria, encontrar el máximo elemento de una lista, o los algoritmos de ordenación mergesort y quicksort.
- Desarrollar algoritmos de divide y vencerás para resolver problemas.

Índice

- Introducción
- Divide y Vencerás
  - Ejemplos

Introducción

- Un **algoritmo** es un conjunto finito de pasos para resolver un problema.
- Una **Estrategia algorítmica**
  - Es un enfoque para resolver un problema.
  - Podría combinar distintos enfoques.
- **Estructura de un algoritmo:**
  - **Iterativa**: usa un bucle para encontrar la solución.
  - **Recursiva**: es una función que se llama a sí misma.

Introducción

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

Índice

- Introducción
- **Divide y Vencerás**
  - Ejemplos

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

Encontrar el mayor elemento en una lista

Problema [8,2,5,3,6,9,4,7]

divide

[8,2,5,3] [6,9,4,7]

divide divide

[8,2] [5,3] [6,9] [4,7]

divide divide divide divide

[8] [2] [5] [3] [6] [9] [4] [7]

vencer

8 2 5 3 6 9 4 7

8 5 9 7

combina

8 9

9

Divide y vencerás: ejemplos

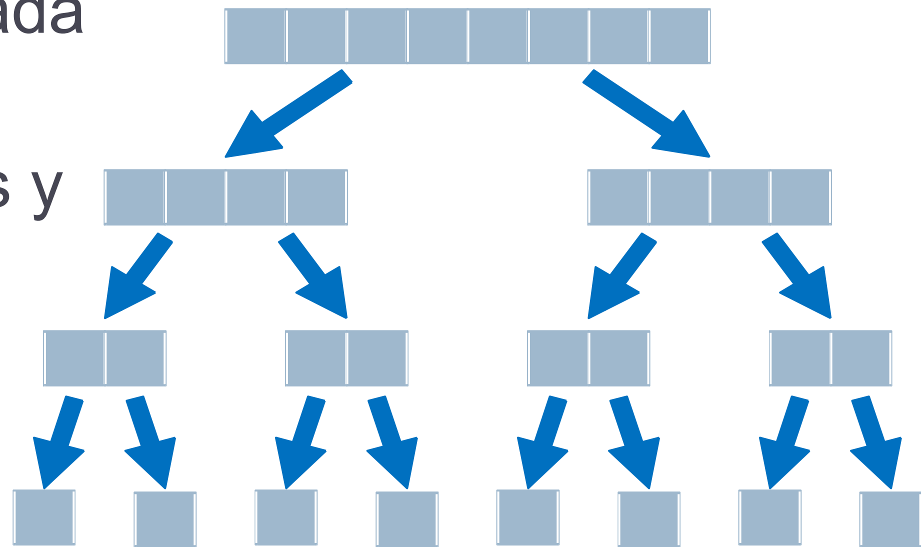
1. Encontrar el máximo elemento en una lista
2. Merge-sort
3. Quick-sort

Encontrar el elemento mayor

A

2	5	8	10	13	20	23	50	90

1. Divide la lista en dos mitades.
2. Encontrar el máximo en cada mitad.
3. Comparar ambos números y devolver el mayor.



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)
```

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 ya están ordenadas!!!
3. ***Combinar***:
   - Tenemos que mezclar las listas ya ordenadas creando una lista ordenada.
     - Repetimos el proceso hasta conseguir una única lista, que será la solución del problema inicial.

Mergesort

```
Algorithm mergesort(A: list):
  if len(A)>1:
    m=len(A)//2
    left=A[0:m]
    right=A[m:]
    sortedLeft=mergesort(left)
    sortedRight=mergesort(right)
    A=merge(sortedLeft,sortedRight)
  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

Complejidad temporal:

- ✓ En cada llamada recursiva, 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 merge** tiene complejidad temporal (O(n)).

Por tanto, 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: 7,3,2,1,9,6,4,8,0

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

- ¿Cómo **dividir** la lista?
  - **Elegir** un elemento **pivote**, para crear dos particiones de la lista:

_quicksort(A,0,left)

_quicksort(A,right,len(A)-1)

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(A,0,left)

_quicksort(A,right,len(A)-1)

Quicksort

Elegir un elemento **pivote**, distintas estrategias:

- **elemento central**
- primer elemento
- último elemento
- pivote aleatorio

Quicksort

- Una vez realizadas las particiones, se **recursión** a cada una de las particiones para ordenarlas

_quicksort(A,0,left)

_quicksort(A,right,len(A)-1)

Quicksort

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

Algorithm _quicksort(A: list,
        left: int,right: int):
    ...
```

Quicksort: quicksort(A,left=0,right=len(A)-1)

left = 0	1	2	3	4	5	6	7	8 = right
1	**8**	**2**	**10**	**5**	**4**	**3**	**20**	**6**

A

Quicksort: quicksort(A,left=0,right=len(A)-1)

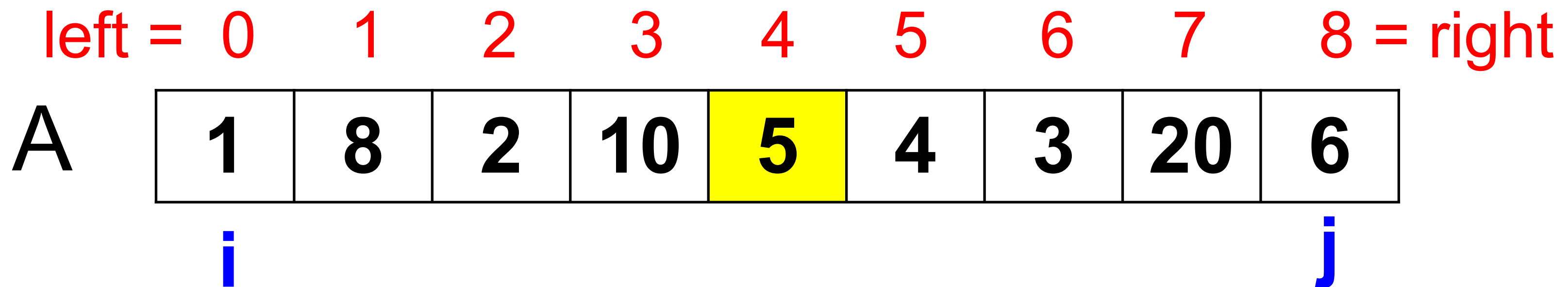
	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6

Seleccionamos el elemento central como pivote:

m=(left+right)//2=4

p=A[m]=5

Quicksort: quicksort(A,left=0,right=len(A)-1)

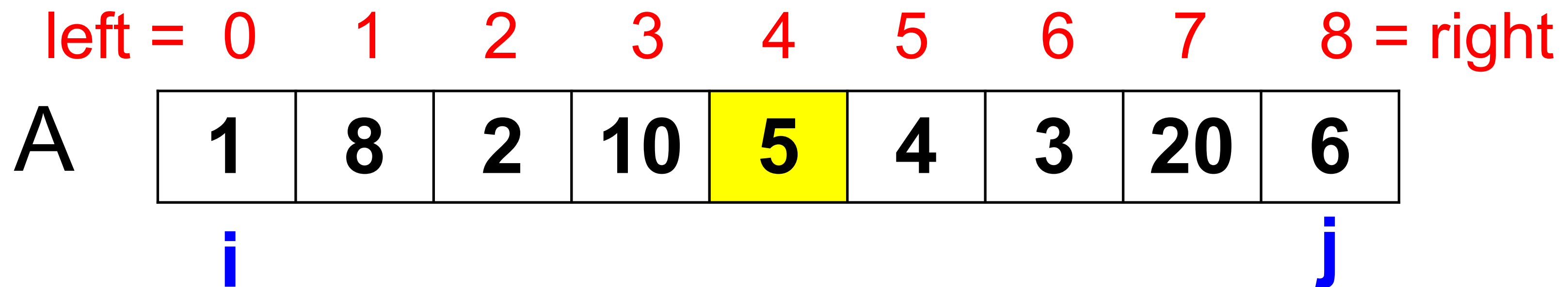


Definimos dos variables que nos van ayudar a recorrer el array desde los dos extremos:

i=left

j=right

Quicksort: quicksort(A,left=0,right=len(A)-1)



Avanzamos i mientras que A[i] sea menor que el pivote:

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

Quicksort:
quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6
		i							j

Avanzamos i mientras que A[i] sea menor que el pivote:

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

No debemos avanzar la variable i, porque A[1]=8 es mayor que el pivote = 5, y por tanto, debería estar a su derecha.

Quicksort: quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6
		i							j

Disminuimos j mientras que A[j] sea mayor que el pivote:

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

Quicksort: quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6
		i						j	

Disminuimos j mientras que A[j] sea mayor que el pivote:

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

Quicksort: quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6
		i					j		

Disminuimos j mientras que A[j] sea mayor que el pivote:

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

Debemos parar en j=6, porque A[6]=3 es menor que el pivote, y por tanto, debería estar a su izquierda.

Quicksort: quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6
		i					j		

Como tenemos dos elementos en particiones incorrectas, simplemente, podemos intercambiar los elementos en esas posiciones y avanzar los índices i y j.

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	3	2	10	5	4	8	20	6
			i			j			

Quicksort: quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	8	2	10	5	4	3	20	6
		i					j		

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

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	3	2	10	5	4	8	20	6
			i			j			

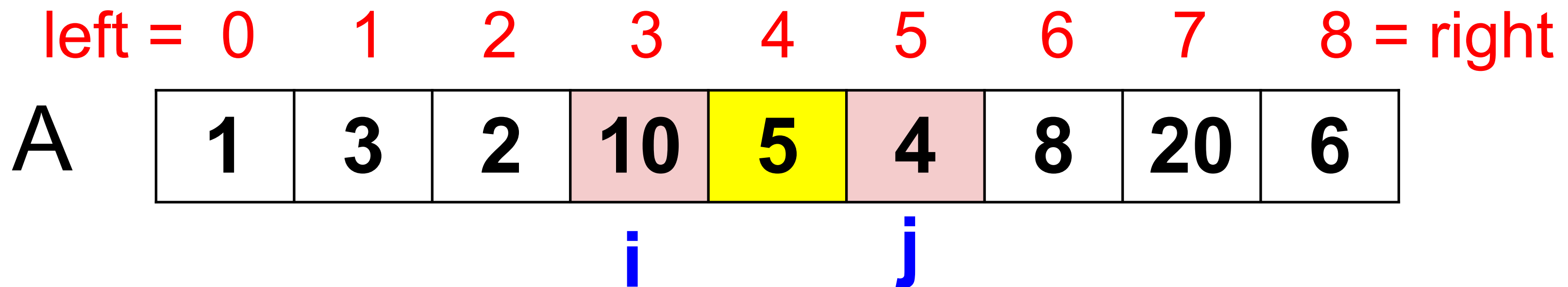
Quicksort: quicksort(A,left=0,right=len(A)-1)

	left = 0	1	2	3	4	5	6	7	8 = right
A	1	3	2	10	5	4	8	20	6
			i			j			

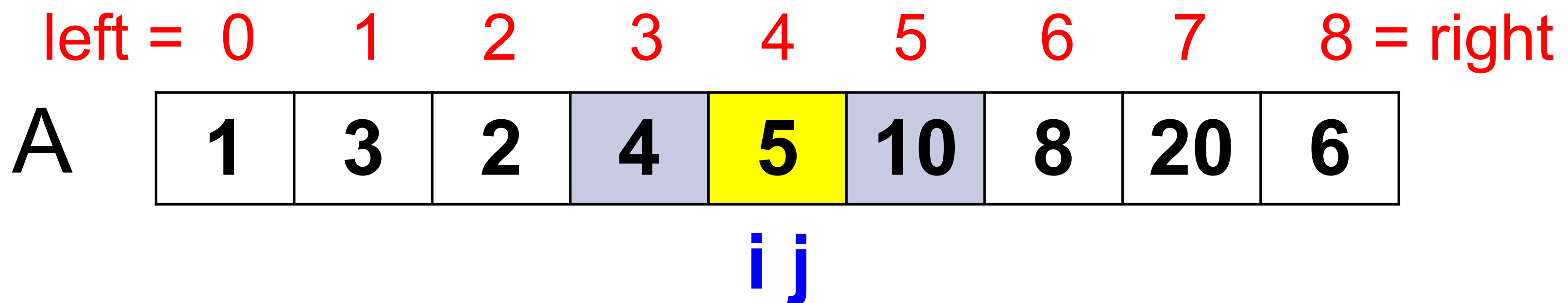
Repetimos el mismo proceso mientras i<=j

```
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
```

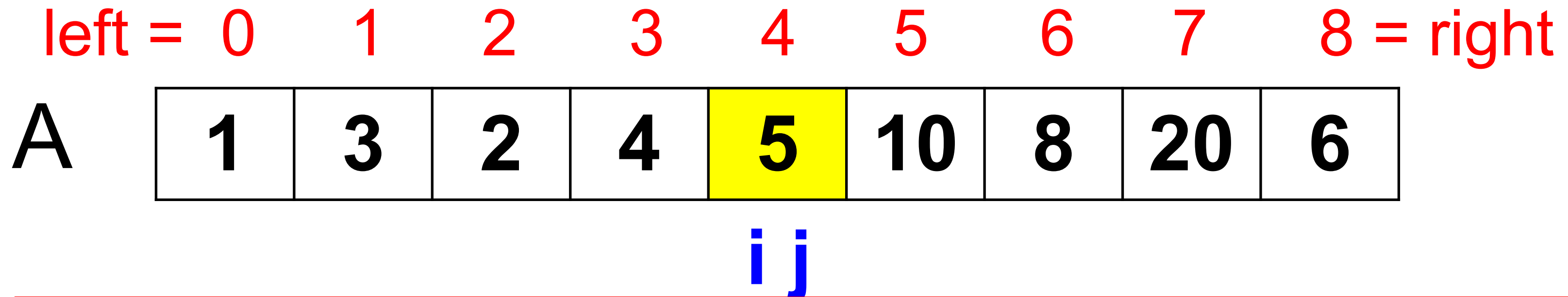
Quicksort: quicksort(A,left=0,right=len(A)-1)



Debemos intercambiar A[i] y A[j], y luego avanzar i y disminuir j



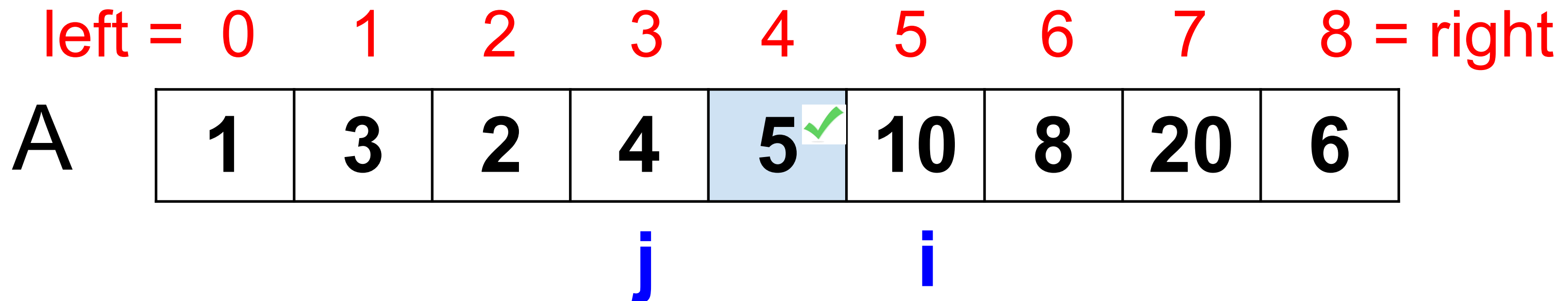
Quicksort: quicksort(A,left=0,right=len(A)-1)



El bucle se ejecuta una vez más, pero sin entrar en los bucles internos. Únicamente se ejecutan el último if

```
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
```

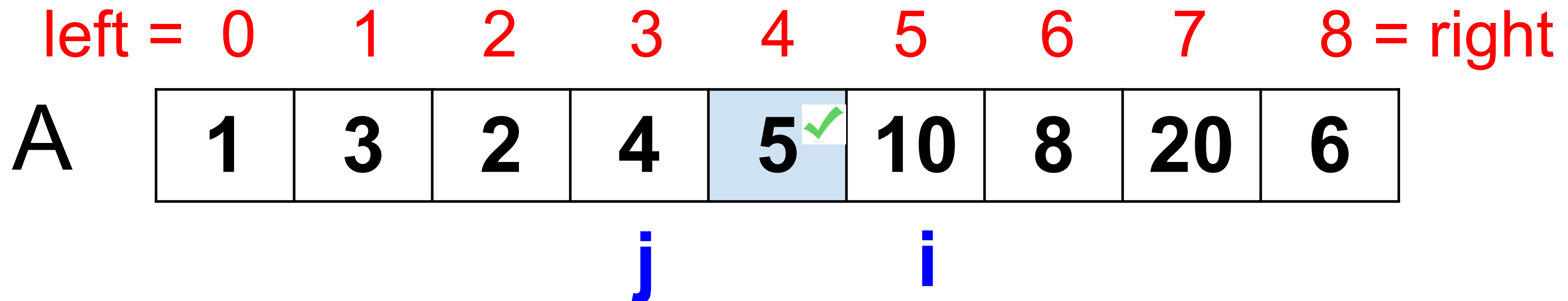
Quicksort: quicksort(A,left=0,right=len(A)-1)



Ya se han realizado las dos particiones.
Todo lo que hay a la izquierda del pivote es menor que el pivote.
Todo lo que hay a la derecha es mayor que el pivote.

El pivote ya ocuparía la posición correcta en la lista ordenada.

Quicksort: quicksort(A,left=0,right=len(A)-1)

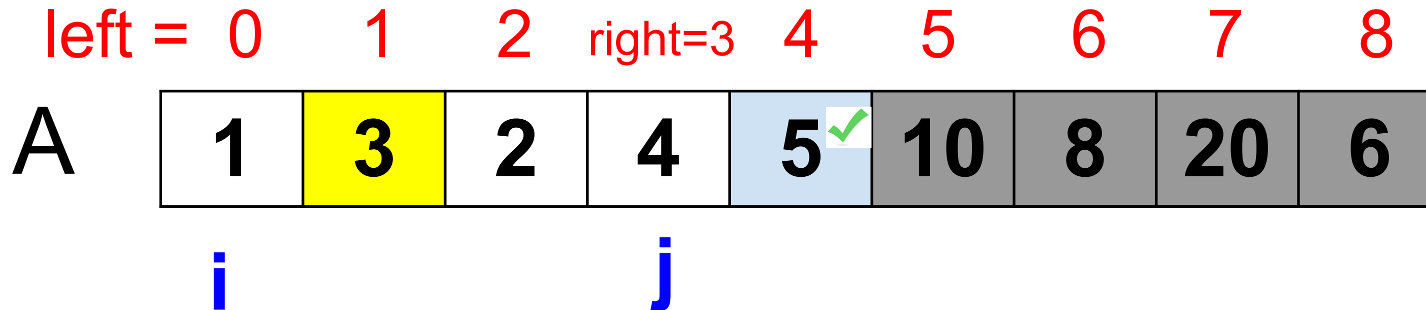


Ahora aplicamos recursión para ordenar cada una de las particiones:

```
if left < j:
    _quicksort(A,left=0,j=3)

if i< right:
    _quicksort(A,i=5,right=8)
```

Quicksort: quicksort(A,left=0,right=3)



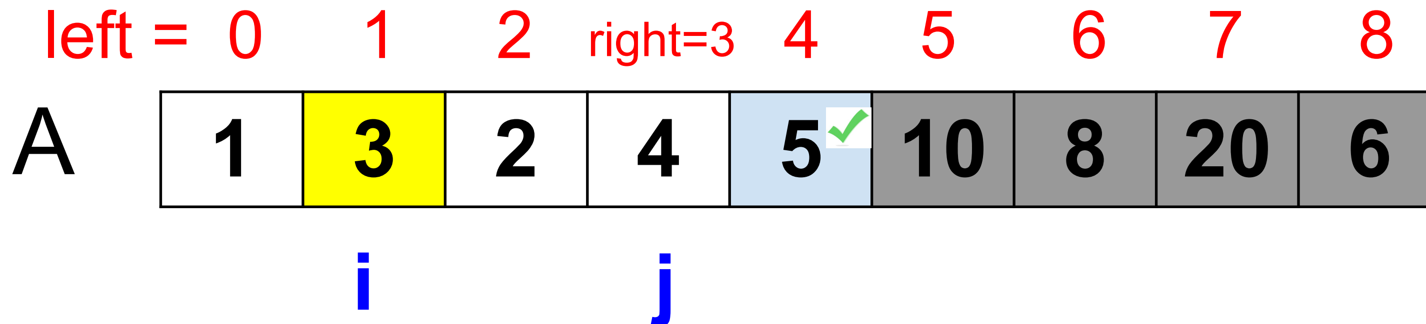
m=(left+right)//2=(0+3)//2=1

p=A[m]=3 (nuevo pivote)

i=left

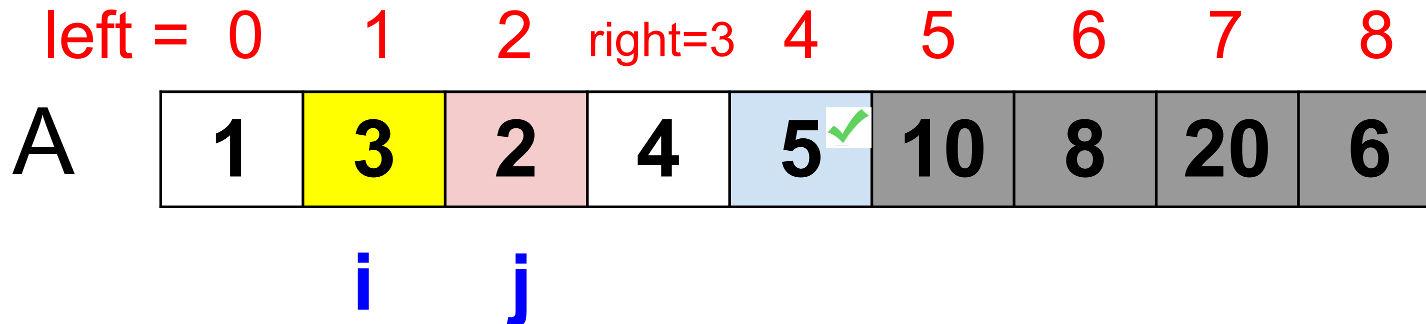
j=right

Quicksort: quicksort(A,left=0,right=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
```

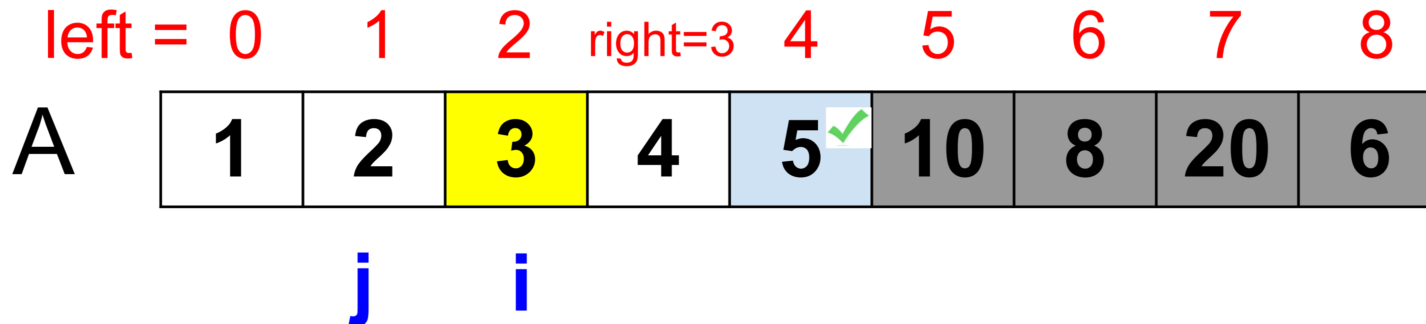
Quicksort: quicksort(A,left=0,right=3)



Debemos intercambiar sus valores

```
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
```

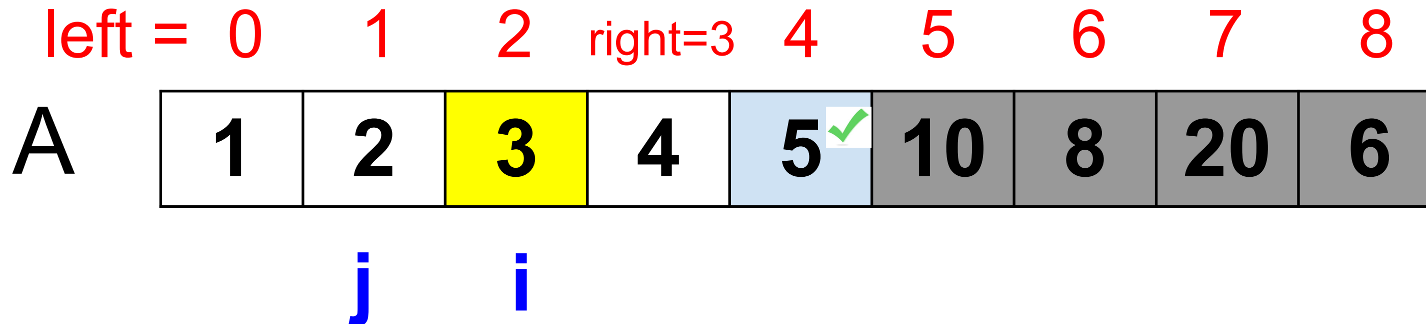
Quicksort: quicksort(A,left=0,right=3)



Incrementar i, y disminuir 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: quicksort(A,left=0,right=3)



Debemos ordenar la dos particiones:

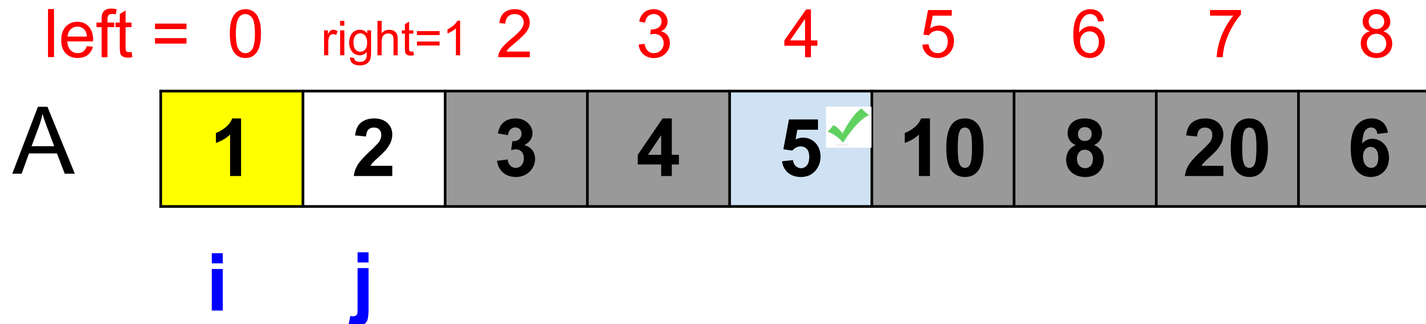
if left < j:

_quicksort(A,left=0,j=1)

if i< right:

_quicksort(A,i=2,right=3)

Quicksort: quicksort(A,left=0,right=3)



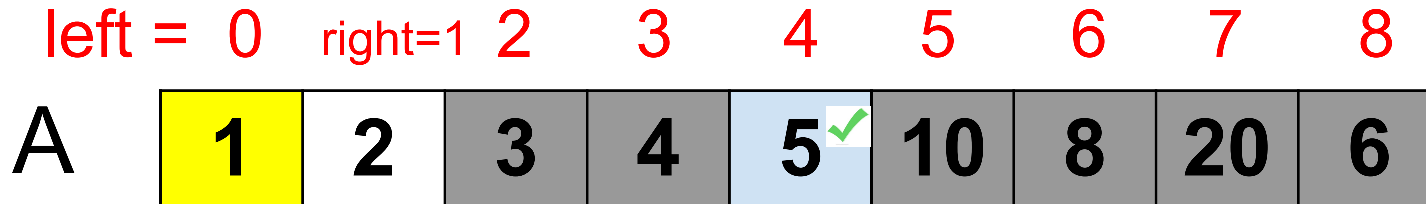
m=(left+right)//2=(0+1)//2=0

p=A[0]=1

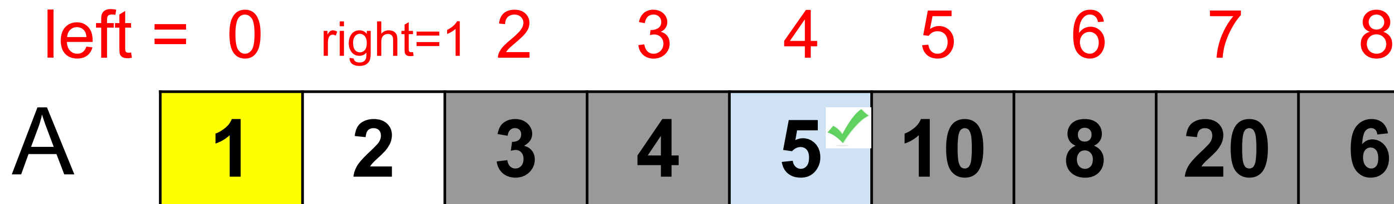
i=left=0

j=right=1

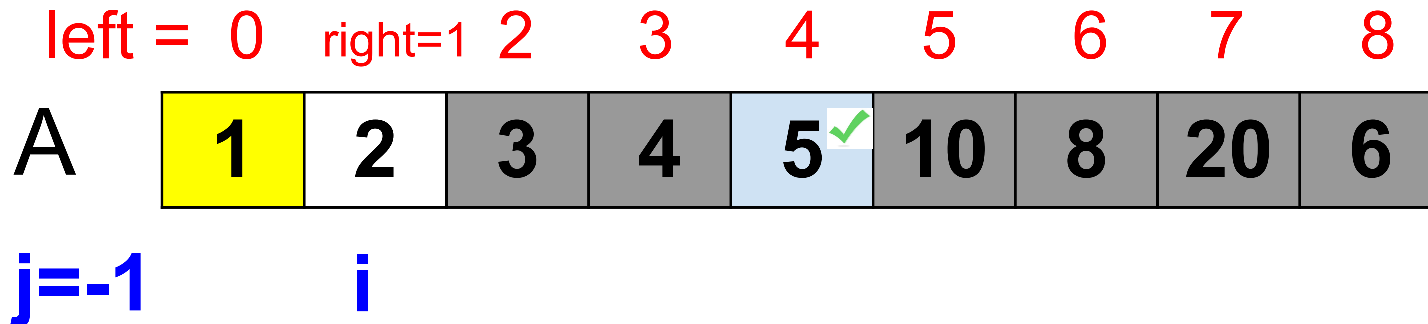
Quicksort: quicksort(A,left=0,right=1)




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



Quicksort: quicksort(A,left=0,right=1)



En este caso, **no hay llamadas recursivas** porque no se cumplen las condiciones (left<j) y i<right)

```
if left < j:
    _quicksort(A,left,j)

if i< right:
    _quicksort(A,i,right)
```

Quicksort: quicksort(A,left=0,right=1)

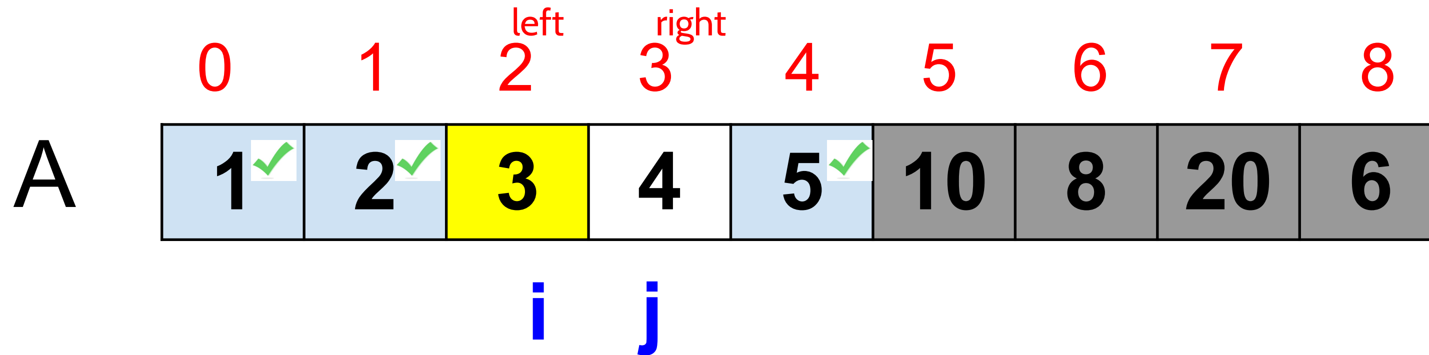
	0	1	2	3	4	5	6	7	8
A	1 ✓	2 ✓	3	4	5 ✓	10	8	20	6

Hemos terminado de ordenar esa sublista, y sus elementos ya están en las posiciones correctas. Ahora debemos resolver:

_quicksort(A,i=2,right=3)

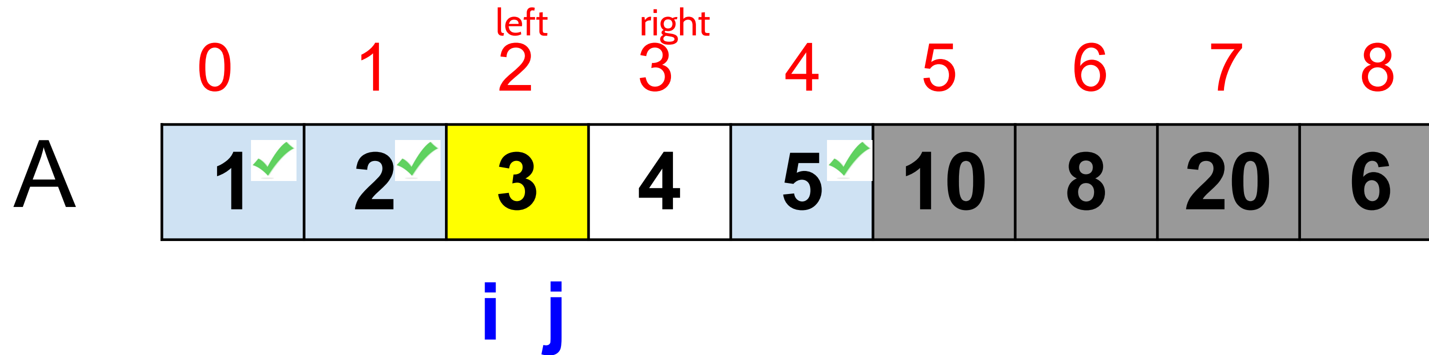
	0	1	2 (left)	3 (right)	4	5	6	7	8
A	1 ✓	2 ✓	3	4	5 ✓	10	8	20	6

Quicksort: quicksort(A,left=2,right=3)

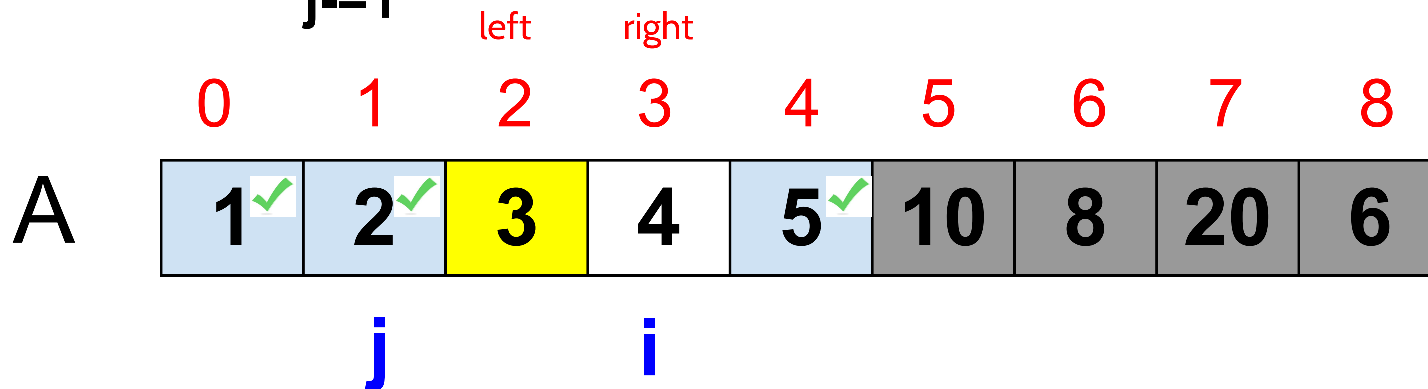


m=(left+right)//2=(2+3)//2=2
p=A[2]=3
i=left=2
j=right=3

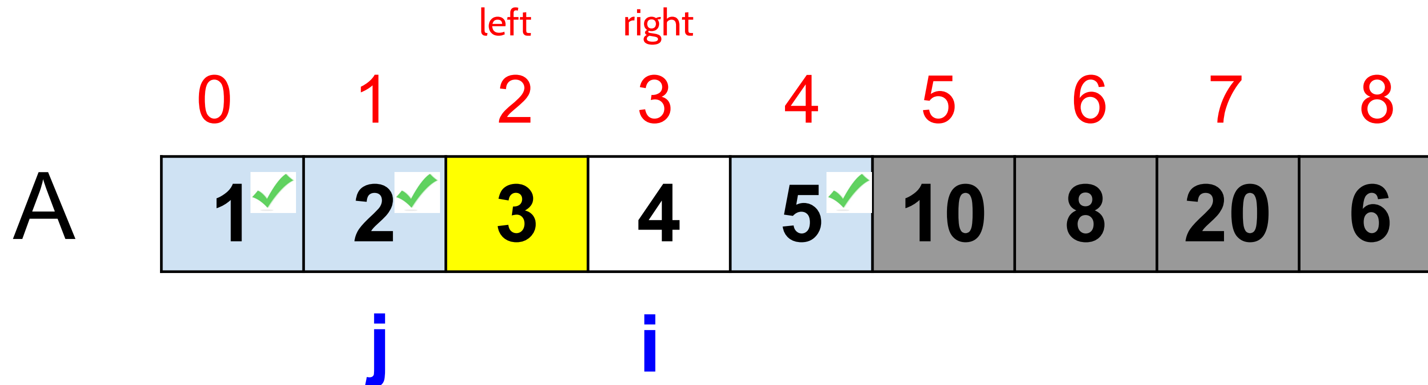
Quicksort: quicksort(A,left=2,right=3)




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



Quicksort: quicksort(A,left=2,right=3)



En este caso, **no hay más llamadas recursivas** porque no se cumplen las condiciones (left<j) y i<right)

```
if left < j:
    _quicksort(A,left,j)

if i< right:
    _quicksort(A,i,right)
```

Quicksort: quicksort(A,left=2,right=3)

	left	right						
0	1	2	3	4	5	6	7	8
1 ✓	2 ✓	3 ✓	4 ✓	5 ✓	10	8	20	6

A

Esa sublista A[left:right] ya ocuparía las posiciones que le corresponden en la lista ordenada

Quicksort: quicksort(A,left=5,right=8)

	left					right			
	0	1	2	3	4	5	6	7	8
A	1 ✓	2 ✓	3 ✓	4 ✓	5 ✓	10	8	20	6

Nos queda por resolver la llamada recursiva (slide 51):

```
if left < j:
        _quicksort(A,left=0,j=3)

if i< right:
        _quicksort(A,i=5,right=8)
```

Quicksort: quicksort(A,left=5,right=8)

	left								right
	0	1	2	3	4	5	6	7	8
A	1 ✓	2 ✓	3 ✓	4 ✓	5 ✓	10	8	20	6
						i			j

Elegimos pivote:

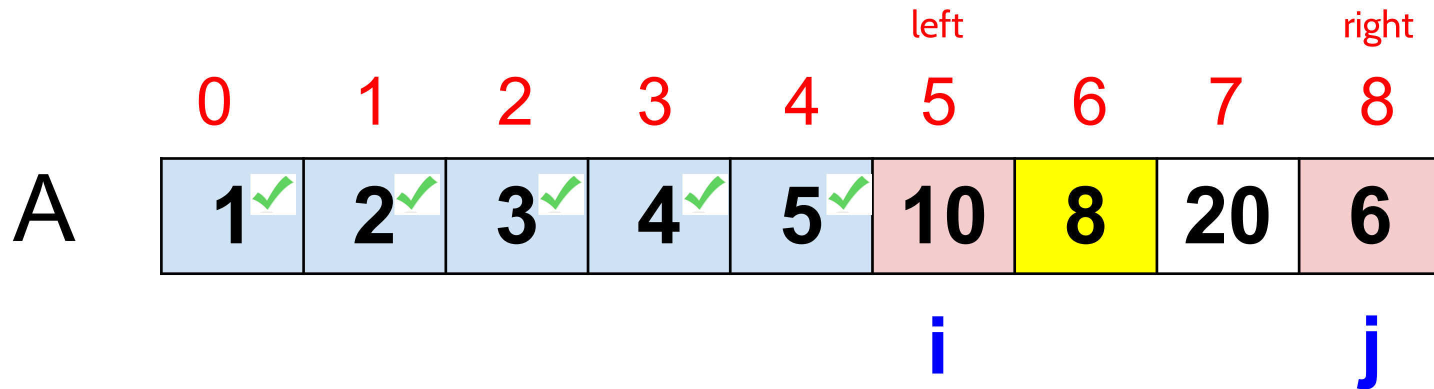
m=(left+right)//2=(5+8)//2=13//2=6

p=A[m=6]=8

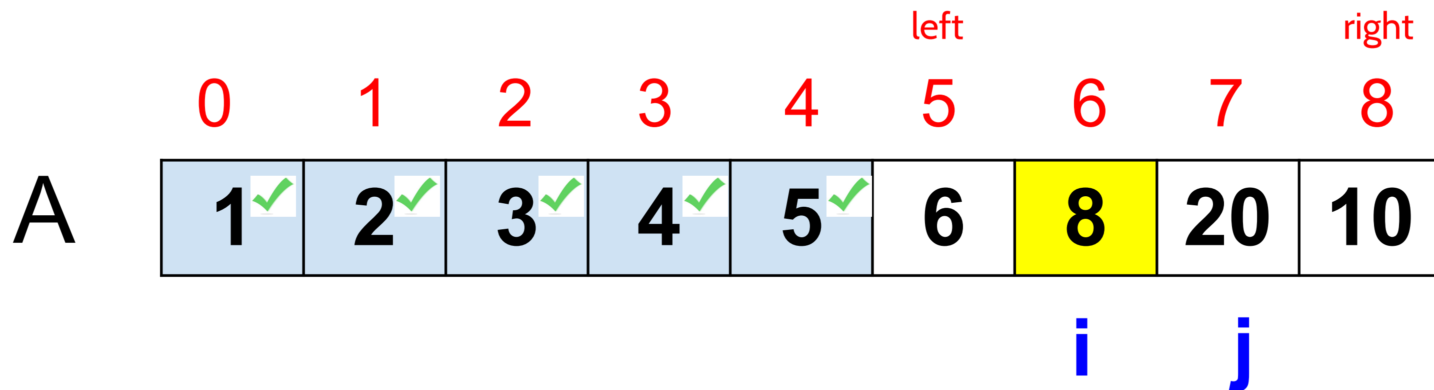
i=left=6

j=right=8

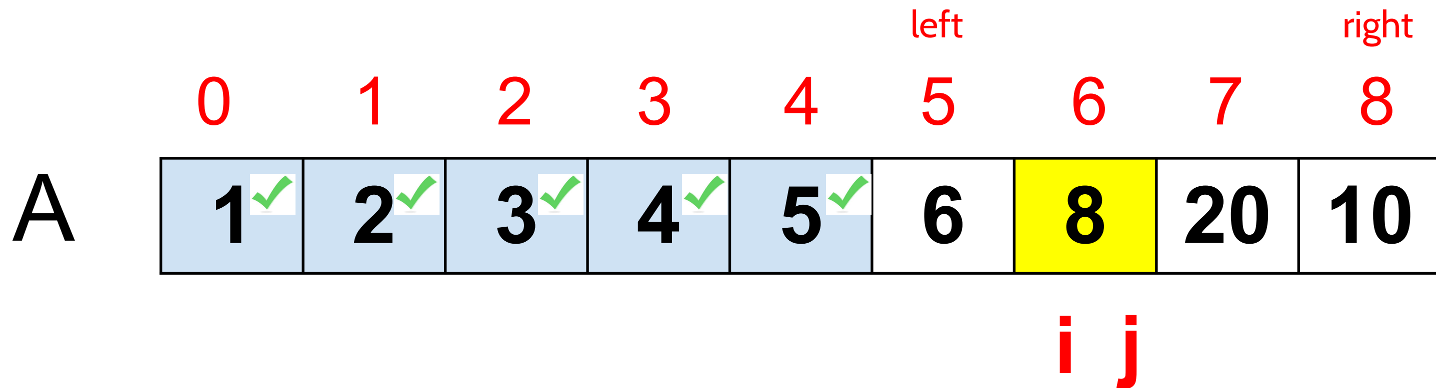
Quicksort: quicksort(A,left=5,right=8)



Debemos intercambiar A[i] y A[j], y mover i y j



Quicksort: quicksort(A,left=5,right=8)



No tenemos que avanzar i (porque A[i] no es menor que p), pero sí debemos disminuir j

Quicksort: quicksort(A,left=5,right=8)

					left			right
0	1	2	3	4	5	6	7	8
1✓	2✓	3✓	4✓	5✓	6	8	20	10
					j		i	

A

Como j>i, debemos parar. De las llamadas recursivas, sólo se ejecutará la segunda (la primera condición no se cumple porque left=j):

```
if left < j:
    _quicksort(A,left,j)

if i< right:
    _quicksort(A,i=7,right=8)
```

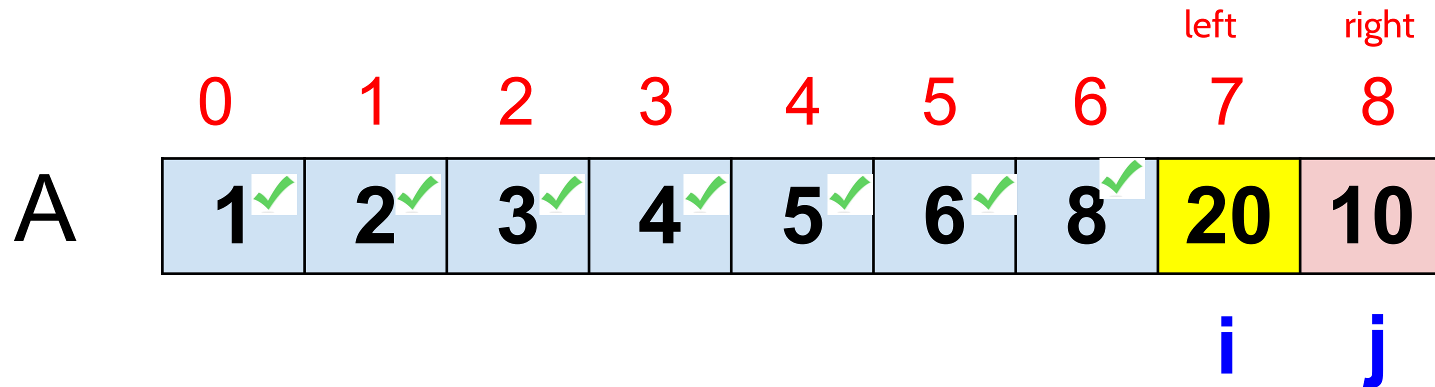
Quicksort: quicksort(A,left=7,right=8)

	0	1	2	3	4	5	6	7 (left)	8 (right)
A	1 ✓	2 ✓	3 ✓	4 ✓	5 ✓	6 ✓	8 ✓	20	10
								i	j

La sublista A[5:6] ya ocupa las posiciones correctas en la lista ordenada. Ahora debemos resolver _quicksort(A,7,8).

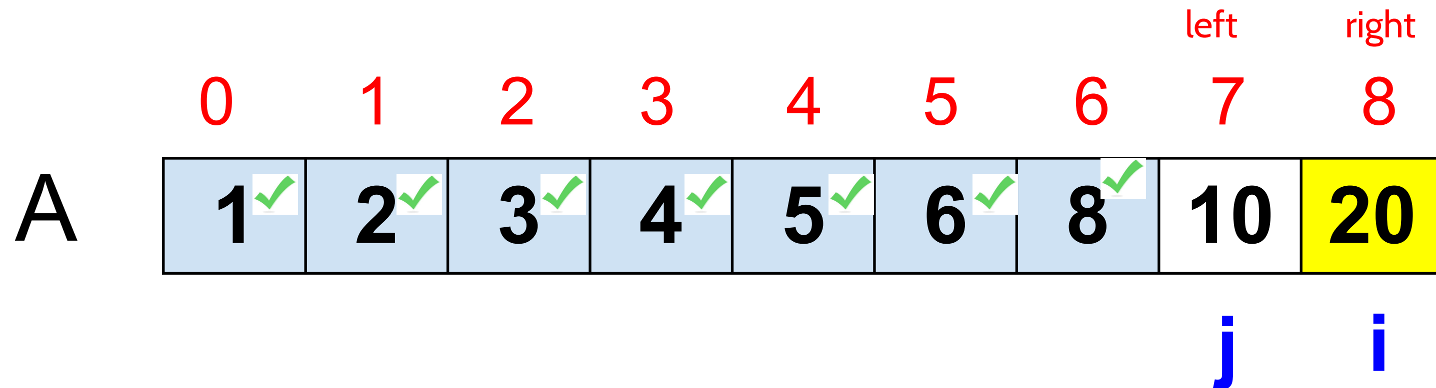
m=(left+right)//2=(7+8)//2=15//2=7
p=A[m]=20
i=left=7
j=right=8

Quicksort: quicksort(A,left=7,right=8)



No debemos mover i ni j. Debemos intercambiar A[i] y A[j], y después mover los índices.

Quicksort: quicksort(A,left=7,right=8)



El bucle no se ejecuta más porque j>i. Tampoco se realiza ninguna llamada recursiva porque left=j, e i=right.

Quicksort

	0	1	2	3	4	5	6	7	8
A	1	2	3	4	5	6	8	10	20

Por tanto, la lista ya está ordenada

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, left: int, right: int):
    m=(left + right) // 2
    p = A[m] # pivot element in the middle

    i = left
    j = right

    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 left < j: # sort left list
        _quicksort(data, left, j)

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

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 estrategía 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.