uc3m | Universidad Carlos III de Madrid

Grado en Ciencia e Ingeniería de Datos, 2018-2019

Unit 7. Divide and Conquer

Algorithms and Data Structures (ADS)

Author: Isabel Segura-Bedmar

CC BY NC SA

Index

- **Divide and conquer (definition)**
- Some algorithms:
  - Find maximum element in an array
  - Binary search
  - Merge-sort
  - Quick-sort

Divide and Conquer (definition)

- **Divide** problems into smaller subproblems (by using recursion) until to reach the simplest problems that can be solved directly (**conquer**).
- **Combine** the intermediate solutions to obtain the solution of the original problem.

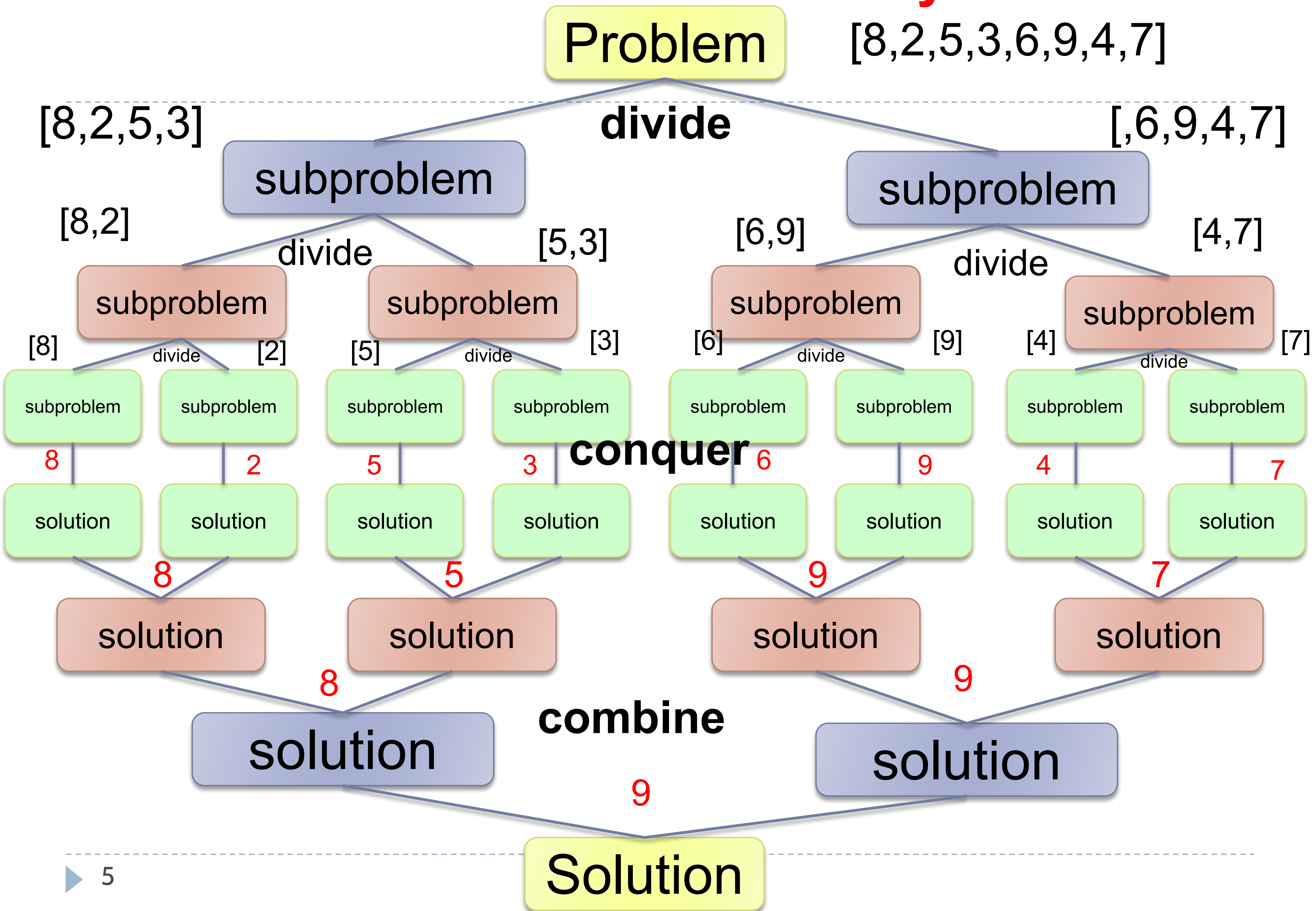
Divide and conquer

Approach in three steps:

1. ***Divide***: split the problem into smaller subproblems of the same type (recursively)*
2. ***Conquer***: solve each subproblem.
3. ***Combine***: combine solutions to solve the original problem

(*) usually contains two or more recursive calls

Find maximum element in an array



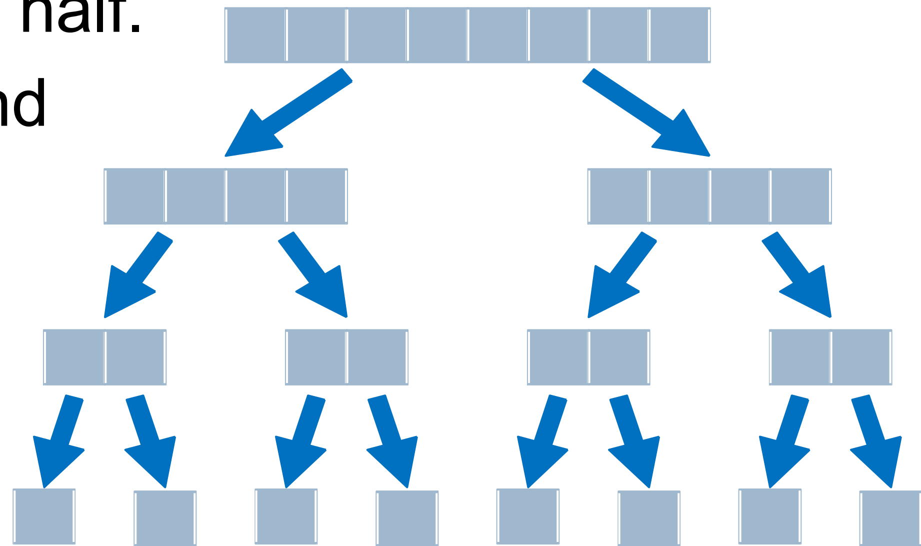
Index

- Divide and conquer (definition)
- Some algorithms:
  - **Find maximum element in an array**
  - Binary search
  - Merge-sort
  - Quick-sort

Divide and Conquer – Find maximum

A	2	5	8	10	13	20	23	50	90

1. Divide the array in two halves.
2. Find the maximum in each half.
3. Compare both numbers and returns the greatest.



Divide and Conquer – Find maximum

```
Algorithm findMax(data):

  if len(data)==1:
    return data[0]

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

Index

- Divide and conquer (definition)
- Some algorithms:
  - Find maximum element in an array
  - **Binary search**
  - Merge-sort
  - Quick-sort
  - Closest Pair of points in the Euclidean plane (two dimensional space)
  - Strassen's Algorithm (multiply matrices)

Divide and Conquer – Binary Search

Input: a sorted list of integers and a number.

A

0	1	2	3	4	5	6	7	8
2	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

x = 23

Divide and Conquer – Binary Search

Input: a sorted list of integers and a number

0	1	2	3	4	5	6	7	8
2	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

x=23 **Output=True**

- ✓ Output: True if x is found, False otherwise

Divide and Conquer – Binary Search

Input: a sorted list of integers and a number

0	1	2	3	4	5	6	7	8
2	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

x=100 Output=False

Output: True if x is found, False otherwise

Divide and Conquer – Binary Search

A	0	1	2	3	4	5	6	7	8
	2	5	8	10	13	20	23	50	90

x = 23

```
for i = 0 to len(A)-1:
    if x==A[i]:
        return True

return False
```

Divide and Conquer – Binary Search

	0	1	2	3	4	5	6	7	8
A	2	5	8	10	13	20	23	50	90

x = 100

```
for i = 0 to len(A)-1:
    if x==A[i]:
        return True

return False
```

Divide and Conquer – Binary Search

- Time Complexity

	0	1	2	3	4	5	6	7	8
A	**2**	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

Best Case: 1 comparison
Worst Case: n comparisons

$O(n)$

Divide and Conquer – Binary Search

	0	1	2	3	4	5	6	7	8
A	**2**	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

How can we improve it?

Divide and Conquer – Binary Search

mid = len(A) // 2

	0	1	2	3	4	5	6	7	8
A	**2**	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**
					mid				

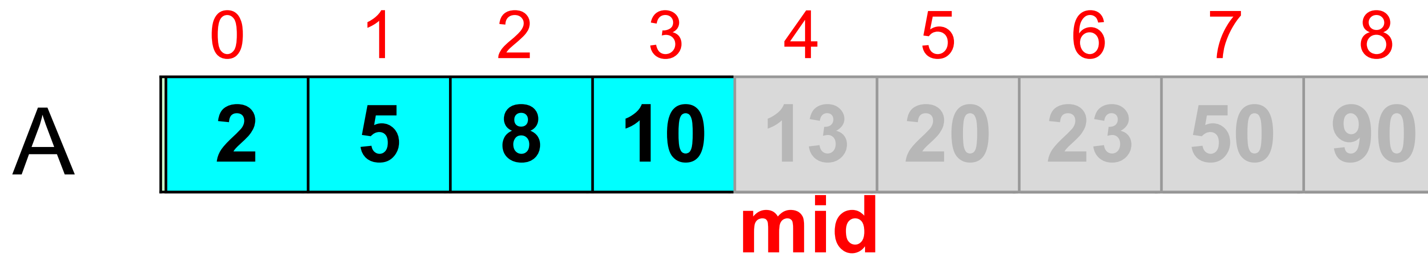
Divide and Conquer – Binary Search

mid = len(A) // 2

	0	1	2	3	4	5	6	7	8
A	**2**	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**
					mid				

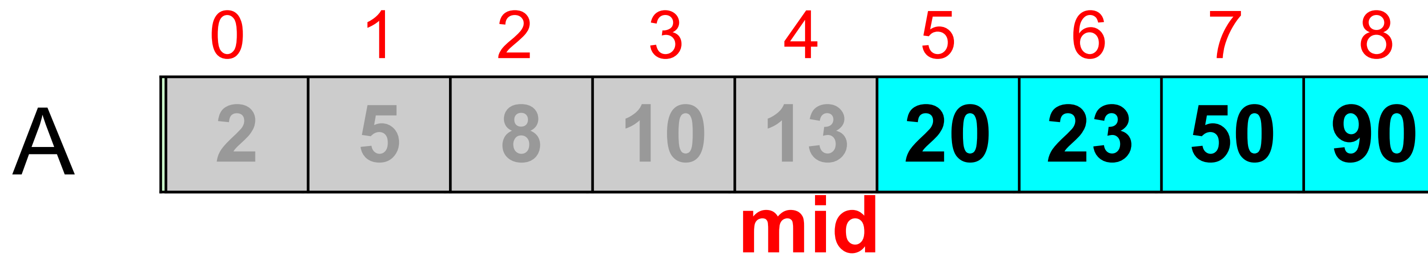
Case 1:
```
if x==A[mid]:
    return True
```

Divide and Conquer – Binary Search



Case 2: **if** x<A[mid]:
return binarySearch(A[0:mid],x)

Divide and Conquer – Binary Search



Case 3: **if** x>A[mid]:

return binarySearch(A[mid+1:],x)

Divide and Conquer – Binary Search

	0	1	2	3	4	5	6	7	8
A	**2**	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

```
Algorithm binarySearch(A,x):
   mid=len(A)//2
   if A[mid]==x:
      return True
   if x<A[mid]:
      return binarySearch(A[0:mid],x)
   if x>A[mid]:
      return binarySearch(A[mid+1:],x)
```

Divide and Conquer – Binary Search

	0	1	2	3	4	5	6	7	8
A	2	5	8	10	13	20	23	50	90

```
Algorithm binarySearch(A,x):
    mid=len(A)//2
    if A[mid]==x:
        return True
    if x<A[mid]:
        return binarySearch(A[0:mid],x)
    if x>A[mid]:
        return binarySearch(A[mid+1:],x)
```

x=6???

Does it work when x does not exist in the list?

Divide and Conquer – Binary Search

	0	1	2	3	4	5	6	7	8
A	2	5	8	10	13	20	23	50	90

```
Algorithm binarySearch(A,x):
    if len(A)==0:
        return False
    mid=len(A)//2
    if A[mid]==x:
        return True
    if x<A[mid]:
        return binarySearch(A[0:mid],x)
    if x>A[mid]:
        return binarySearch(A[mid+1:],x)
```

Divide and Conquer – Binary Search

Time complexity

1. Best Case: $O(1)$ comparisons (x is in the middle of the array).
2. Worst Case: k comparisons. The search space is halved in every comparison. The number of steps is Logarithmic (1). The complexity of each comparison is constant ($O(1)$). Therefore, $O(1*\log n) = O(\log n) << O(n)$

(1) $n, n/2, n/2^2, n/2^3, \ldots n/2^k = 1 \Rightarrow$ **$k = \log n$**

Index

- Divide and conquer (definition)
- Some algorithms:
  - Find maximum element in an array
  - Binary search
  - **Merge-sort**
  - Quick-sort

Divide and Conquer – Merge-sort

Merge-sort algorithm: sort an array

- *Divide*:
  - Split the array into two halves.
  - Keep splitting the resulting arrays until you cannot split anymore (arrays of length one).
- *Conquer*:
  - The arrays of length one are already sorted!!!
- *Combine*:
  - Then merge the adjacent arrays to form a sorted array.

Repeat the process until we have a single sorted array (this is the solution to the original problem)

Divide and Conquer – Merge-sort

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

Divide and Conquer – Merge-sort

```
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
```

Divide and Conquer – Merge-sort

Time complexity:

- ✓ In each recursive call, the search space is divided by half: $n, n/2, n/2^2, n/2^3, \ldots, n/2^k$ => $k = \log n$
- ✓ **Merge algorithm** has linear complexity (O(n)).

Therefore, O(n*log n)

Divide and Conquer – Merge-sort

- Try some examples with the demo:
- https://www.hackerearth.com/es/practice/algorithms/sorting/merge-sort/visualize/

- For example: 7,3,2,1,9,6,4,8,0

Index

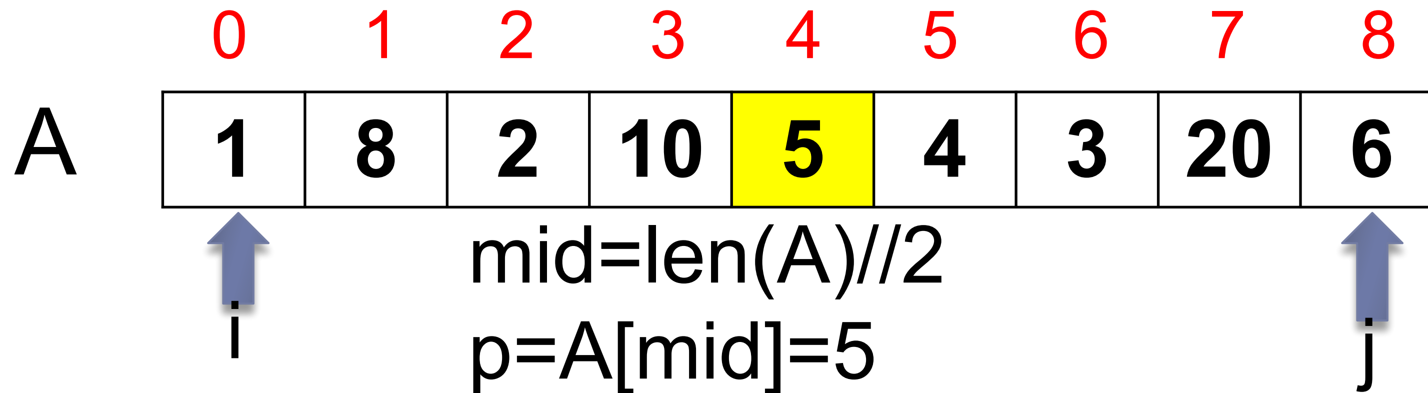
- Divide and conquer (definition)
- Some algorithms:
  - Find maximum element in an array
  - Binary search
  - Merge-sort
  - **Quick-sort**

Divide and Conquer – Quick-sort

- Sort an array.
  - Pick a pivot (for example, first element, last element, random element or element at middle).
    - Put all smaller than (or equal) to pivot to its left.
    - Put all greater elements than pivot after it (to its right).
    - Result: pivot is placed in its correct position in the array.
  - Recursively apply the above steps to the two sublists while their sizes are greater than 1. (base case: sublist contains only a single element)

Divide and Conquer – Quick-sort

1. Pick a pivot (at middle of the array)

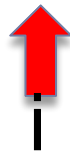


- We use two indexes: i (from left) and j (from right)
- Advance i (i+1) while A[i]<p
- Advance j (j-1) while A[j]>p
- We must stop if i>j

Divide and Conquer – Quick-sort

2. Put all smaller elements before than pivot, and all greater after than pivot.

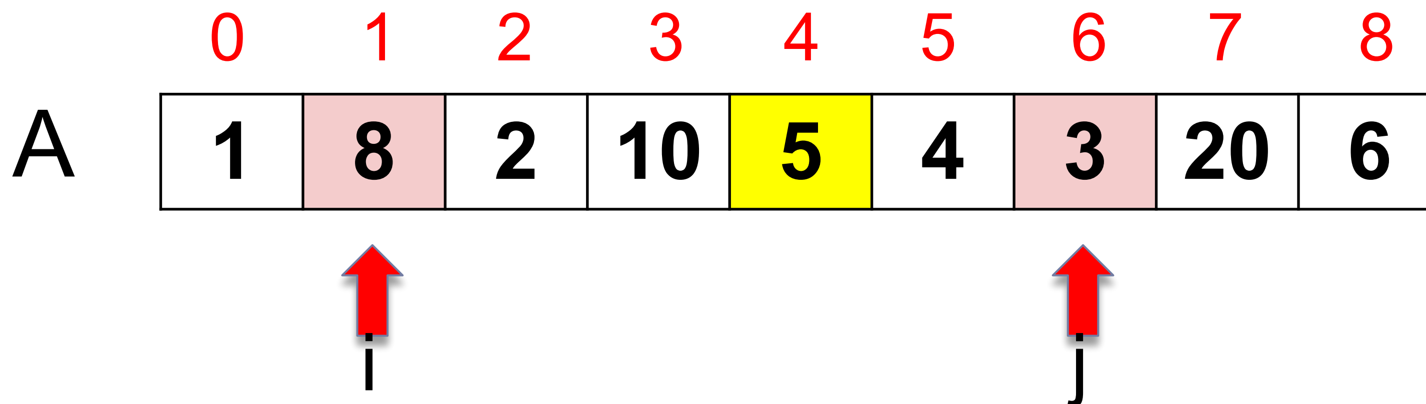
	0	1	2	3	4	5	6	7	8
A	**1**	**8**	**2**	**10**	**5**	**4**	**3**	**20**	**6**



We must swap them!!!
Advance i and j

Divide and Conquer – Quick-sort

2. Put all smaller elements before than pivot, and all greater after than pivot.

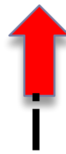


We swap them!!!
Advance i and j

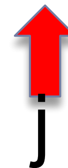
Divide and Conquer – Quick-sort

2. Put all smaller elements before than pivot, and all greater after than pivot.

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



i

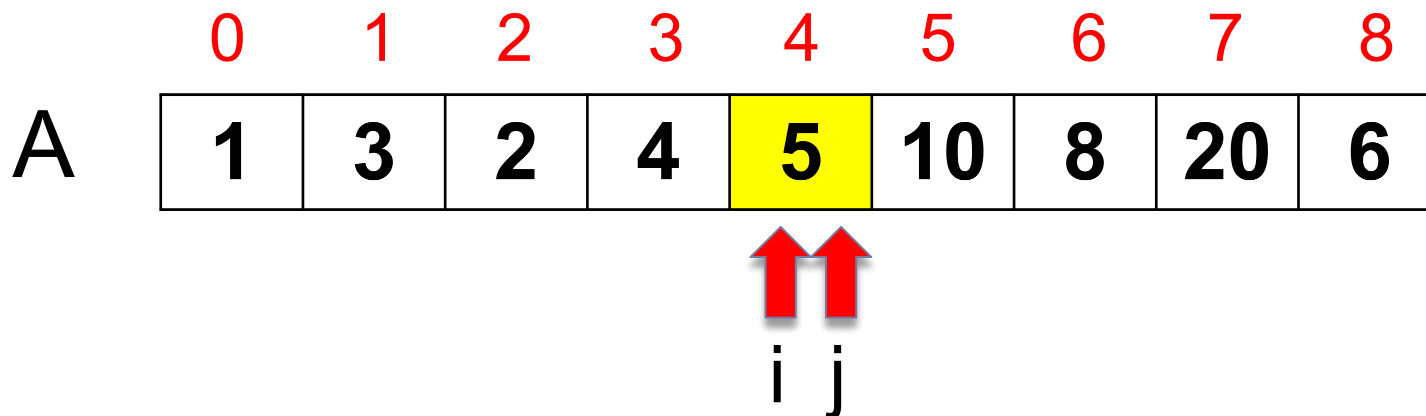


j

We must swap them!!!
Advance i and j

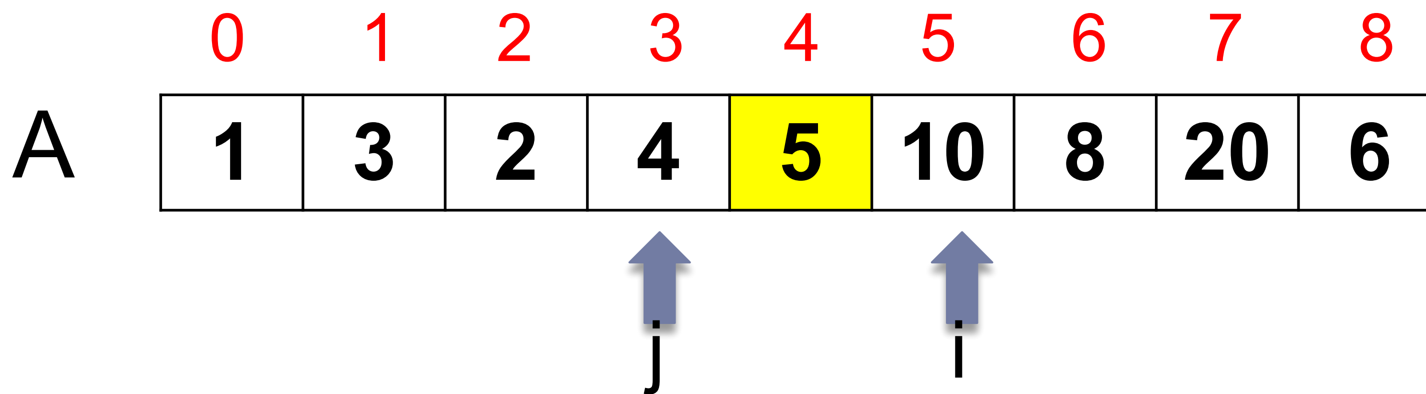
Divide and Conquer – Quick-sort

2. Put all smaller elements before than pivot, and all greater after than pivot.



Divide and Conquer – Quick-sort

2. Put all smaller elements before than pivot, and all greater after than pivot.



We stop because i>j

Divide and Conquer – Quick-sort

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

p=5 is already in its correct position in the array!!!

Now we must apply the same algorithm for the first particion (0,mid-1) and for the second partition (mid+1,end)

Divide and Conquer – Quick-sort

1. Pick a pivot (at middle of the array)

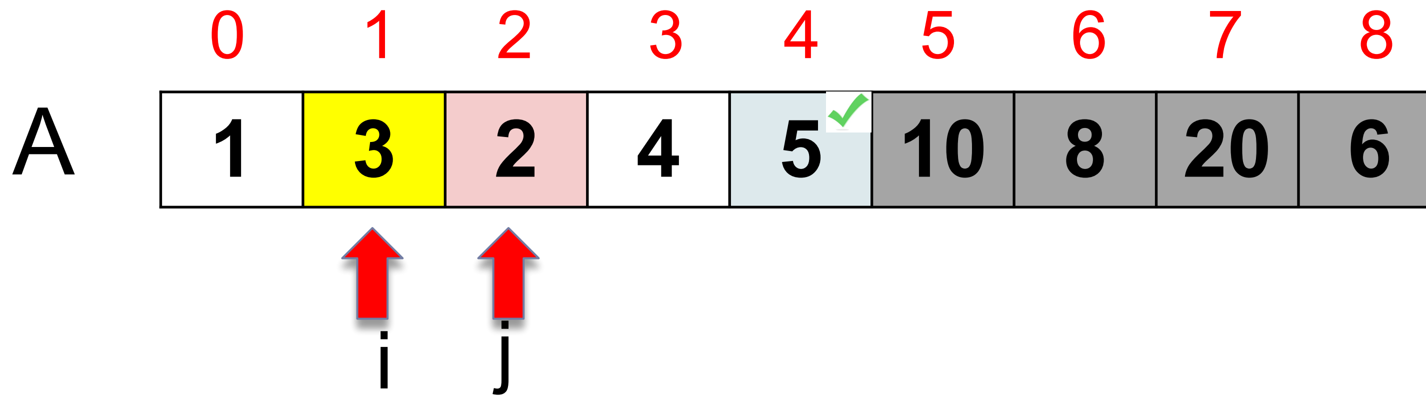
	0	1	2	3	4	5	6	7	8
A	1	3	2	4	5	10	8	20	6
	i			j					

mid=(0+3)/2=1

p=A[mid]=3 (the new pivot)

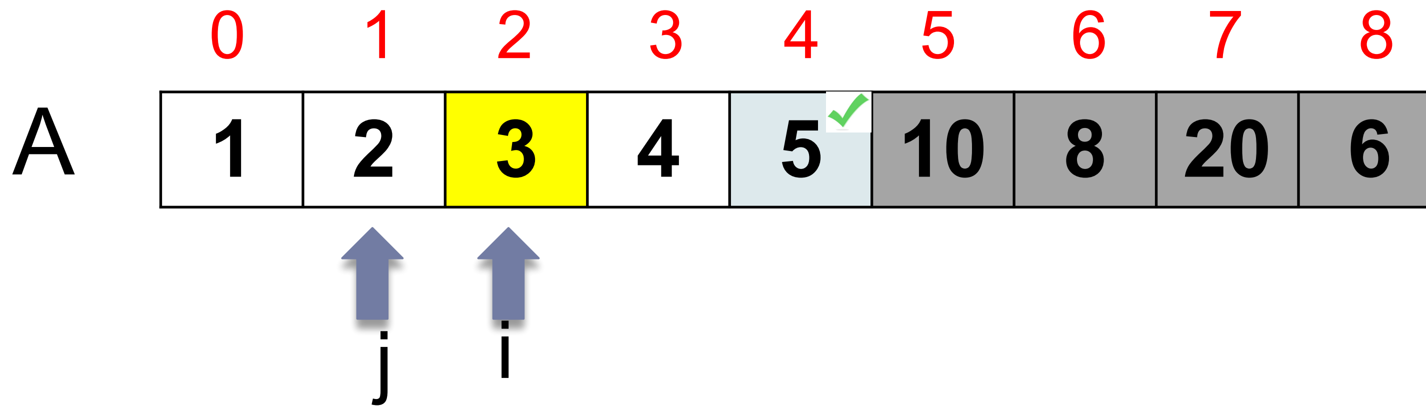
Divide and Conquer – Quick-sort

1. Pick a pivot (at middle of the array)



We swap them!!!. Then, advance i and j

Divide and Conquer – Quick-sort



We stop because i>j

Divide and Conquer – Quick-sort

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

p=3 is already in its correct position in the array!!!

[4] is already sorted!!! (only contains one single element).
We have to sort the first particion ([1,2]).

Divide and Conquer – Quick-sort

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

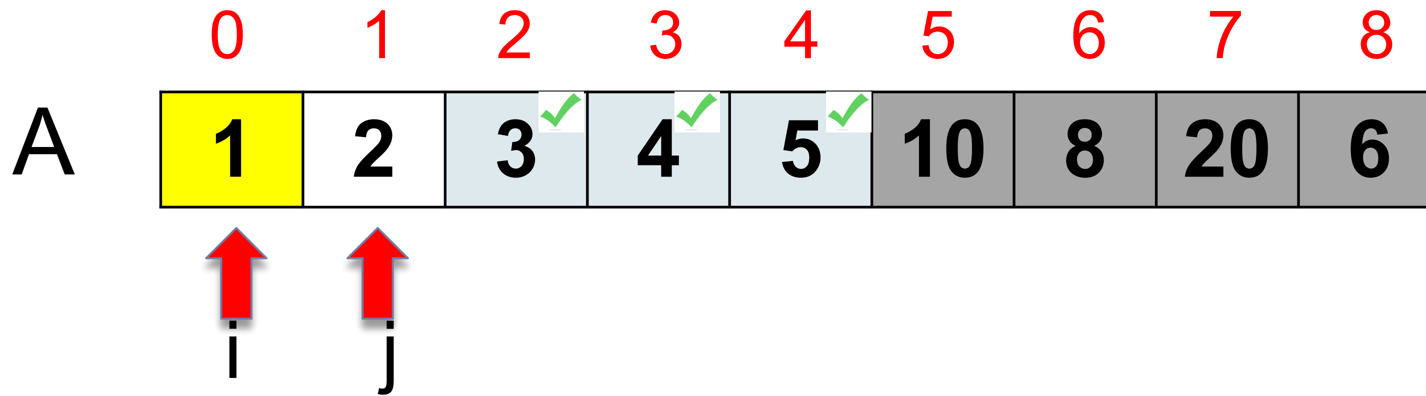
Divide and Conquer – Quick-sort

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

mid=(0+1)/2=0

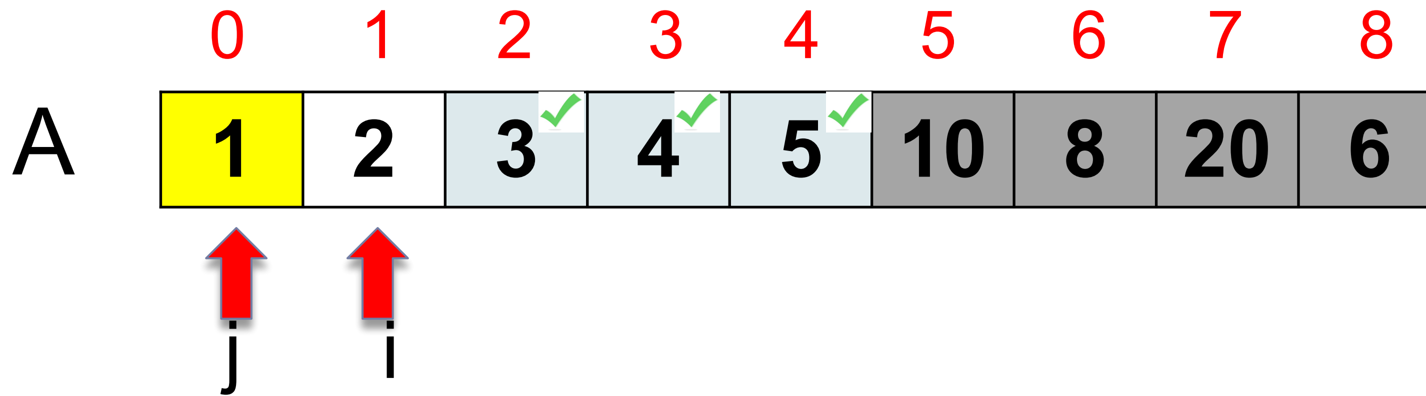
p=A[mid]=1

Divide and Conquer – Quick-sort



We must advance i and j

Divide and Conquer – Quick-sort



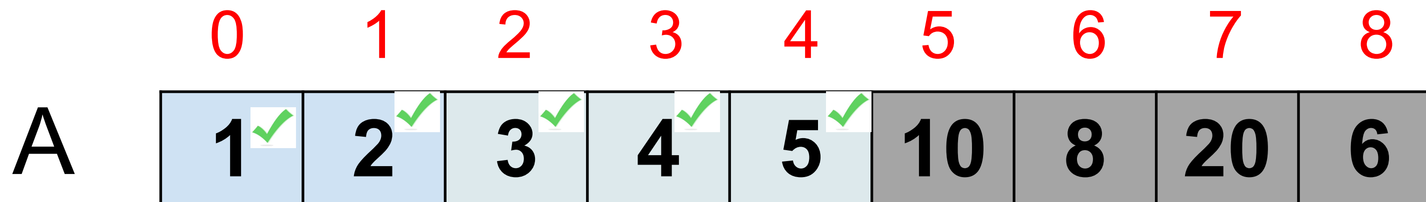
We stop because i>j

Divide and Conquer – Quick-sort

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

p=1 is already in its right position.

Divide and Conquer – Quick-sort



[2] has length 1, therefore, we do no need to sort it.

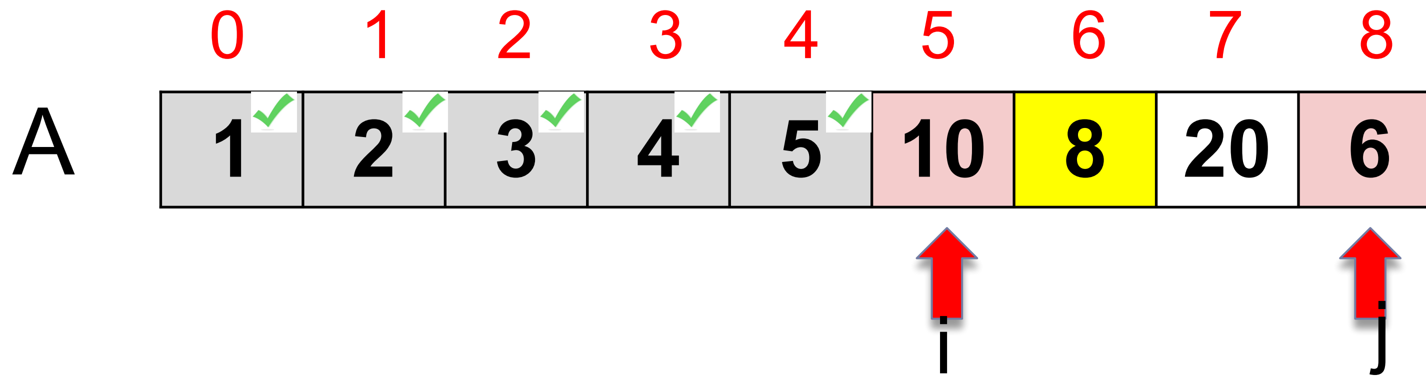
Divide and Conquer – Quick-sort

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

mid=(8+5)/2=6

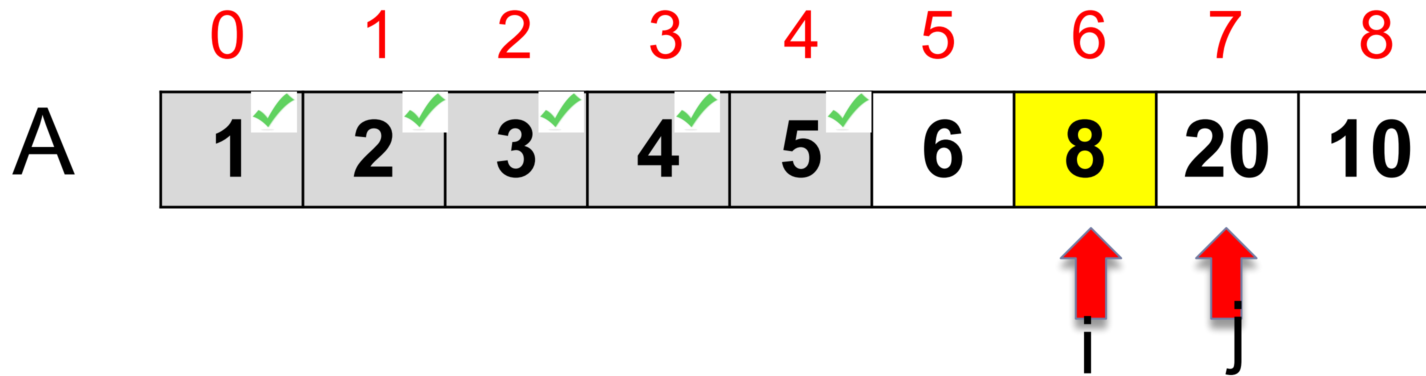
p=A[mid]=8 (the new pivot)

Divide and Conquer – Quick-sort

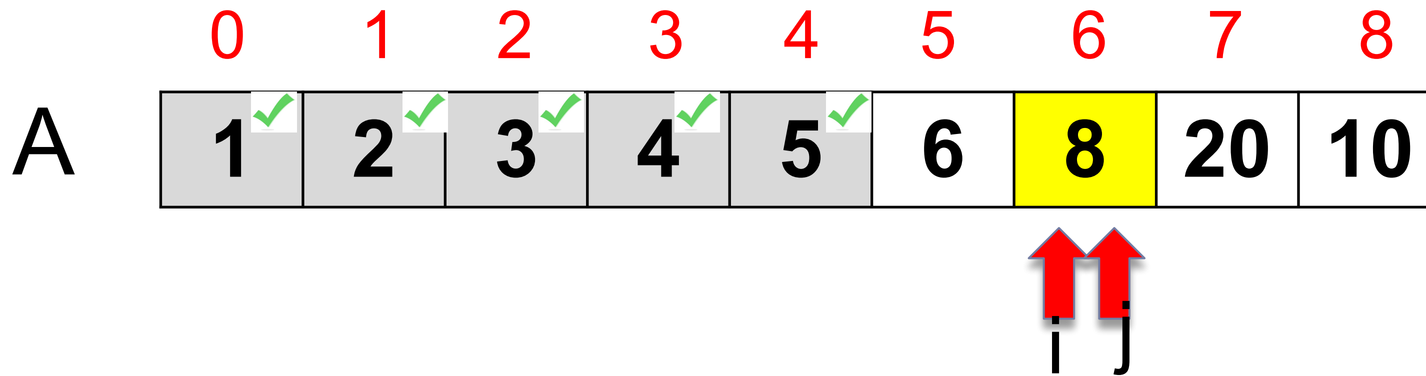


We must swap them
advance i and j

Divide and Conquer – Quick-sort



Divide and Conquer – Quick-sort



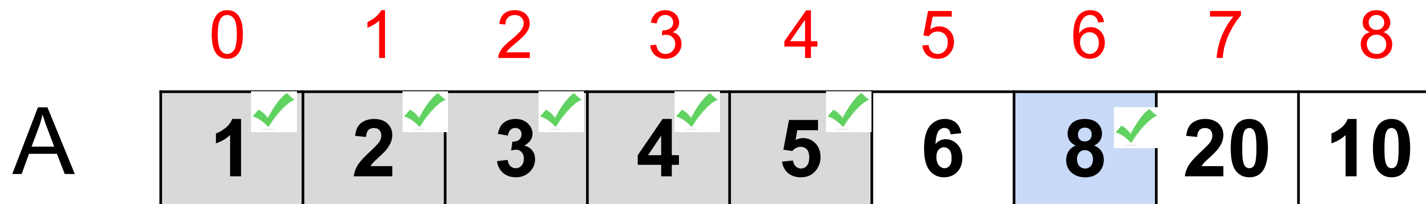
The pivot swaps with itself
Advance i and j

Divide and Conquer – Quick-sort

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

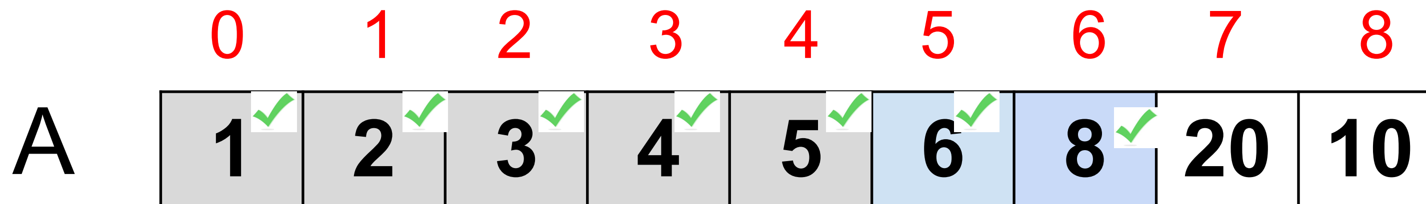
Stop because i>j

Divide and Conquer – Quick-sort



The pivot is already at its correct position.

Divide and Conquer – Quick-sort



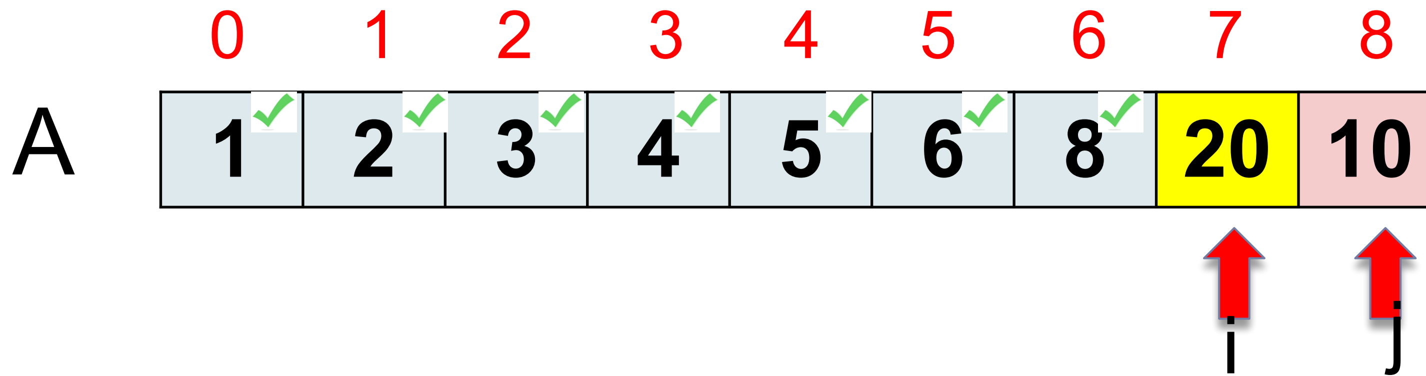
[6] is already sorted

Divide and Conquer – Quick-sort

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

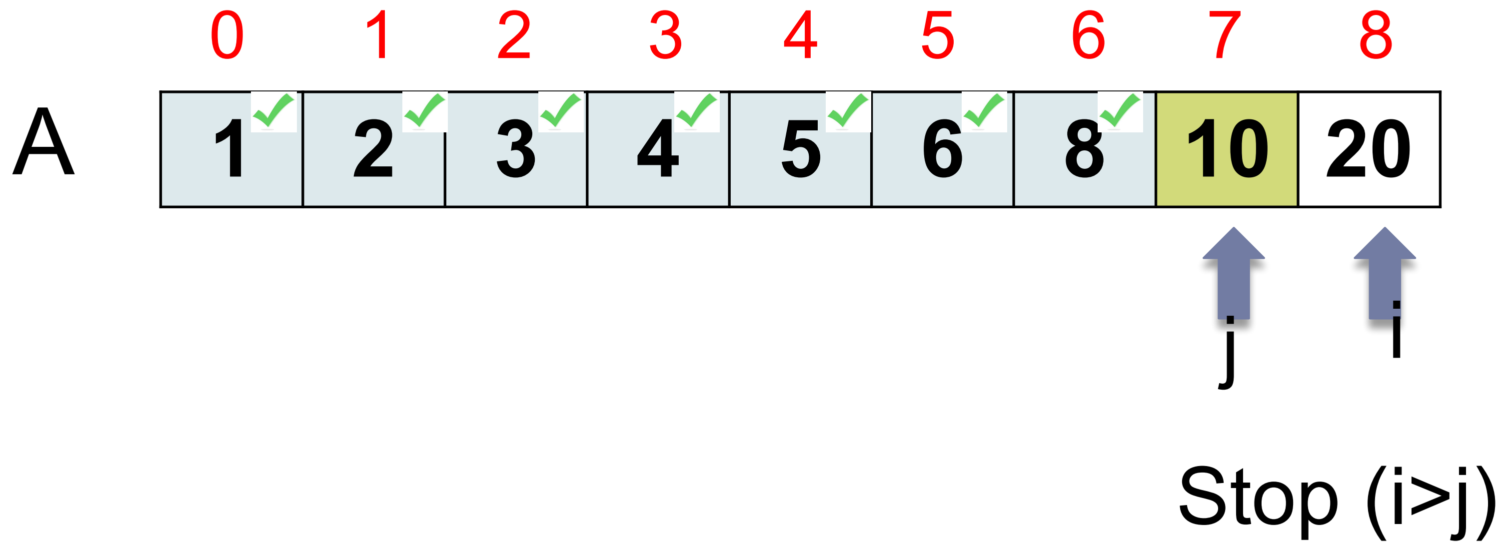
mid=(8+7)/2=7
p=20 (the new pivot)

Divide and Conquer – Quick-sort



We must swap them
advance i and j

Divide and Conquer – Quick-sort



Stop (i>j)

Divide and Conquer – Quick-sort

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

The list is sorted!!!

Divide and Conquer – Quick-sort

```
Algorithm quicksort(data):
    _quicksort(data,0,len(data)-1)
```

Divide and Conquer – Quick-sort

```
Algorithm _quicksort(data, left, right):
    i = left
    j = right

    m=(left + right) // 2

    p = data[m] # pivot element in the middle

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

Divide and Conquer – Quick-sort

Time complexity:

- ✓ In each particion, the search space is divided by half (best case): $n, n/2, n/2^2, n/2^3, \ldots, n/2^k$
  => $k = \log n$
- ✓ **Partition algorithm** (put all smaller elements before the pivot and all greater ones after) has linear complexity (O(n)).

Therefore, O(n*log n)

Why recursion is a good friend?

Divide and Conquer (recursion) provides better performance to sort arrays than iterative algorithms (such as buble, insertion or selection sort)

$$O(nlogn) << O(n^2)$$