

# Retrieving Data and Sorting

Advanced Programming and Algorithmic Design

Alberto Casagrande

*Email:* `acasagrande@units.it`

a.a. 2018/2019

The background of the slide features a large, faint watermark of the University of Trieste logo. The logo is circular and contains the text "UNIVERSITA' DEGLI STUDI DI TRIESTE" around the perimeter. Inside the circle, there is an illustration of a building with a dome and a tower, with the words "E SPLENDI" visible below the main text.

## Retrieving Data

# Retrieving Data

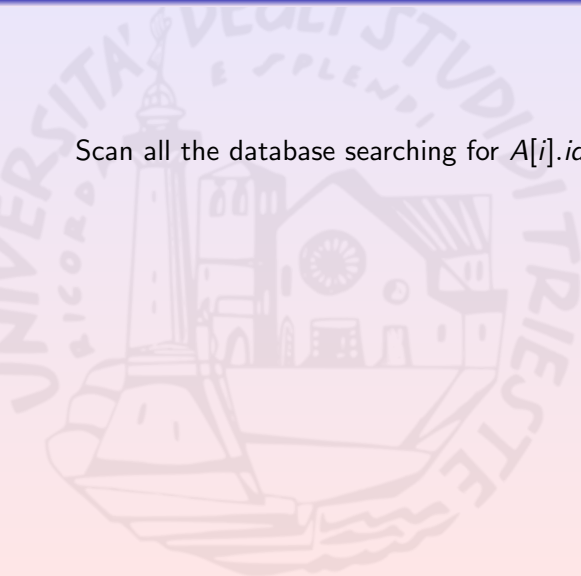
$A = \langle a_1, \dots, a_n \rangle$  contains some data, e.g., patient records

Each element is associated to an **identifier**,  $A[i].id$ , e.g., SSN

How to find the data associated to the identifier  $id_1$ ?

# A Naïve Solution and Outlook

Scan all the database searching for  $A[i].id = id_1$



# A Naïve Solution and Outlook

Scan all the database searching for  $A[i].id = id_1$

What is the asymptotic complexity in terms of big- $O$ ?

# A Naïve Solution and Outlook

Scan all the database searching for  $A[i].id = id_1$

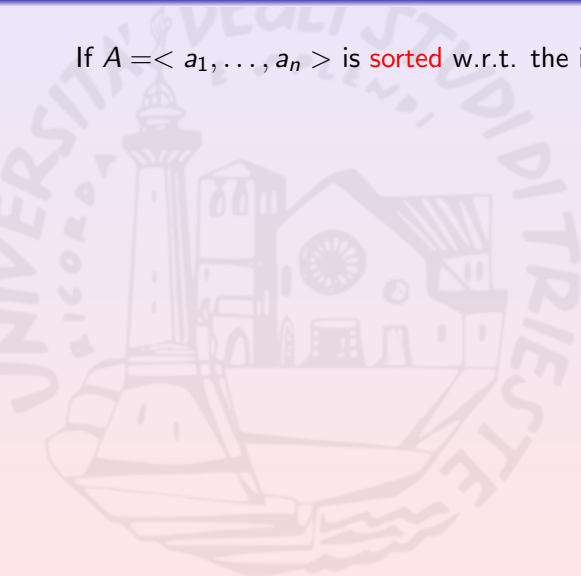
What is the asymptotic complexity in terms of big- $O$ ?  $O(n)$

Can we do better?

*Hint:* How do you search a page in a book? Why?

## A Better Technique: Dichotomic Search

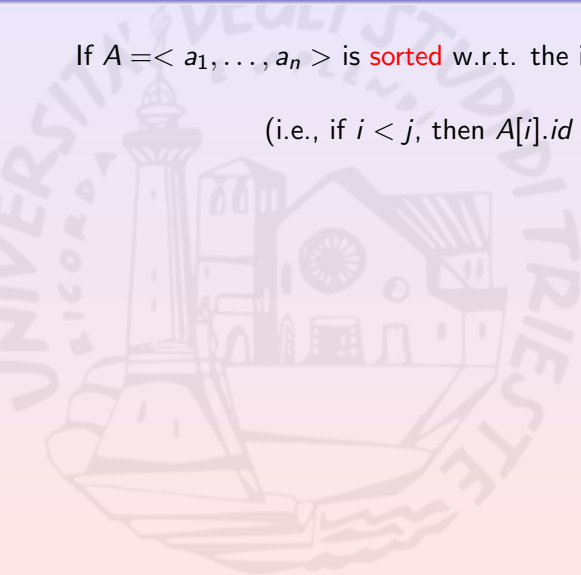
If  $A = \langle a_1, \dots, a_n \rangle$  is **sorted** w.r.t. the id's...



## A Better Technique: Dichotomic Search

If  $A = \langle a_1, \dots, a_n \rangle$  is **sorted** w.r.t. the id's...

(i.e., if  $i < j$ , then  $A[i].id \leq A[j].id$ )





## A Better Technique: Dichotomic Search

If  $A = \langle a_1, \dots, a_n \rangle$  is **sorted** w.r.t. the id's...

(i.e., if  $i < j$ , then  $A[i].id \leq A[j].id$ )

Look at element in the middle  $A[n/2]$

## A Better Technique: Dichotomic Search

If  $A = \langle a_1, \dots, a_n \rangle$  is **sorted** w.r.t. the id's...

(i.e., if  $i < j$ , then  $A[i].id \leq A[j].id$ )

Look at element in the middle  $A[n/2]$

if  $A[n/2].id = id_1$

Done!

## A Better Technique: Dichotomic Search

If  $A = \langle a_1, \dots, a_n \rangle$  is **sorted** w.r.t. the id's...

(i.e., if  $i < j$ , then  $A[i].id \leq A[j].id$ )

Look at element in the middle  $A[n/2]$

if  $A[n/2].id = id_1$

Done!

if  $A[n/2].id > id_1$

Focus on the 1st half  $A$ , i.e.,  $\langle a_1, \dots, a_{n/2-1} \rangle$

## A Better Technique: Dichotomic Search

If  $A = \langle a_1, \dots, a_n \rangle$  is **sorted** w.r.t. the id's...

(i.e., if  $i < j$ , then  $A[i].id \leq A[j].id$ )

Look at element in the middle  $A[n/2]$

if  $A[n/2].id = id_1$

Done!

if  $A[n/2].id > id_1$

Focus on the 1st half  $A$ , i.e.  $\langle a_1, \dots, a_{n/2-1} \rangle$

if  $A[n/2].id < id_1$

Focus on the 2nd half  $A$ , i.e.  $\langle a_{n/2+1}, \dots, a_n \rangle$

Repeat until  $A$  is not empty

# Dichotomic Search: An Example

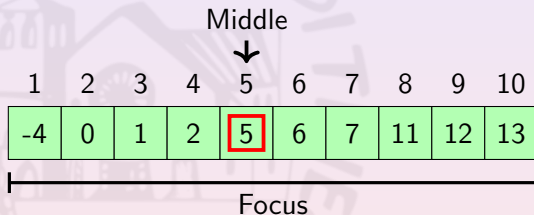
Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

## Focus

# Dichotomic Search: An Example

Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



# Dichotomic Search: An Example

Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

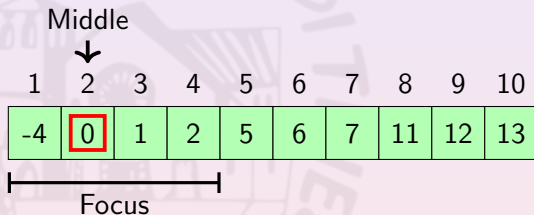
1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

┌───────────┐  
Focus



# Dichotomic Search: An Example

Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



# Dichotomic Search: An Example

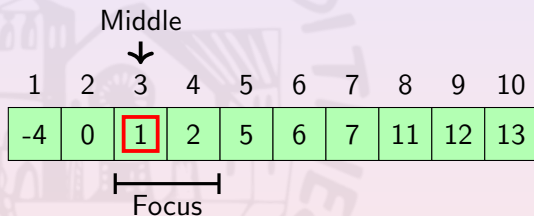
Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

┌──────────┐  
Focus

# Dichotomic Search: An Example

Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



# Dichotomic Search: An Example

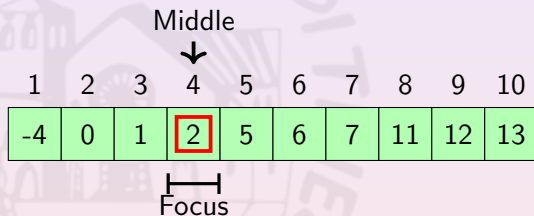
Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

┌───┐  
Focus

# Dichotomic Search: An Example

Search for 2 in  $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$ .



**Found:**  $A[4] = 2$

# Dichotomic Search: Pseudo-Code and Complexity

```
def di_find(A, a):  
    (l, r) ← (1, |A|)  
    while r > l:  
        m ← (l+r)/2  
        if A[m]==a:  
            return m  
        endif  
        if A[m]>a:  
            r ← m-1  
        else  
            l ← m+1  
        endif  
    endwhile  
  
    return 0  
enddef
```

At each iteration,  $l - r$  is halved.

So, if  $|A| \leq 2^m$ , di\_find ends after  $m$  iterations.

The while-block takes time  $O(1)$ .

The di\_find 's complexity is

$O(\log n)$

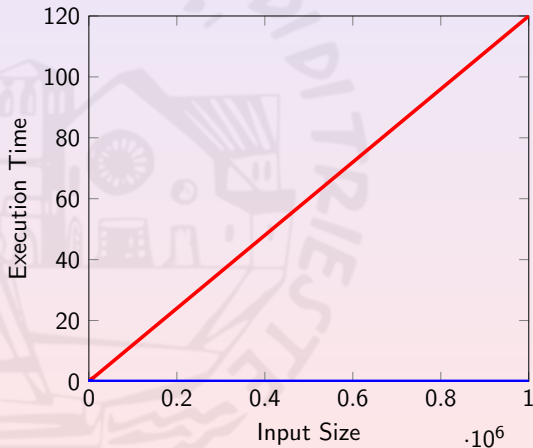
# Dichotomic Search vs Linear Search: Experiments

Execution time per  $1 \times 10^5$  random searches.

Input size	Linear Search	Dichotomic Search
$1 \times 10^1$	$3.3 \times 10^{-3}$ s	$3.2 \times 10^{-3}$ s
$1 \times 10^2$	$1.4 \times 10^{-2}$ s	$4.3 \times 10^{-3}$ s
$1 \times 10^3$	$1.2 \times 10^{-1}$ s	$5.9 \times 10^{-3}$ s
$1 \times 10^4$	1.2 s	$7.8 \times 10^{-3}$ s
$1 \times 10^5$	$1.2 \times 10^1$ s	$8.7 \times 10^{-3}$ s
$1 \times 10^6$	$1.2 \times 10^2$ s	$1.2 \times 10^{-2}$ s

# Dichotomic Search vs Linear Search: Experiments

Execution time per  $1 \times 10^5$  random searches.





The background of the slide features a large, faint watermark of the University of Trieste logo. The logo is circular and contains an illustration of a building with a dome and a tower, with the text "UNIVERSITA' DEGLI STUDI DI TRIESTE" and "E SPLENDI" around it.

## Sorting

# The Sorting Problem

**Input:** An array  $A$  of numbers

**Output:** The array  $A$  sorted i.e., if  $i < j$ , then  $A[i] \leq A[j]$

E.g.,

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0

⇓

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	4	5	6	7	11	13

# The Sorting Problem

**Input:** An array  $A$  of numbers

**Output:** The array  $A$  sorted i.e., if  $i < j$ , then  $A[i] \leq A[j]$

E.g.,

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0

⇓

1	2	3	4	5	6	7	8	9	10
-4	0	1	2	4	5	6	7	11	13

Any idea for a sorting algorithm? What is expected complexity?

Retrieving Data  
○○○○○○○

Sorting  
○○

**Insertion Sort**  
●○○

Quick Sort  
○○○○○○○○○○○○

Heap Sort  
○○○○○

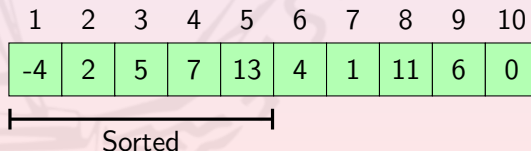
Sorting in Linear Time  
○○○○○○○○○○○○○○

Select  
○○○○○○○○○

## Insertion Sort

# Insertion Sort: Intuition

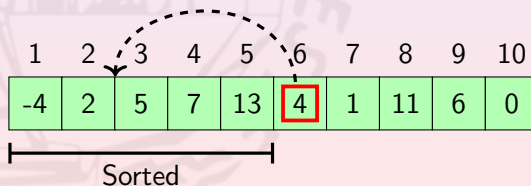
If the first fragment of the array is already sorted



# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “enlarge” it by inserting **next element**

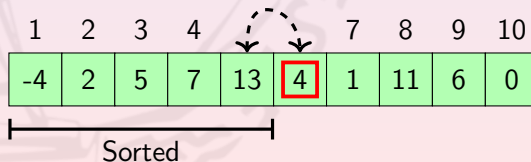


# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “enlarge” it by inserting **next element**

by swapping **it** and the previous one in the array

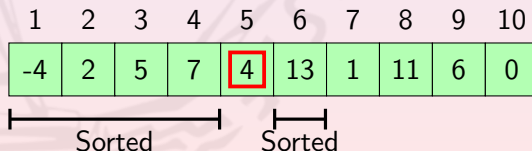


# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “enlarge” it by inserting **next element**

by swapping **it** and the previous one in the array



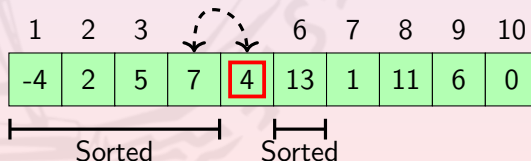


# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “enlarge” it by inserting **next element**

by swapping **it** and the previous one in the array

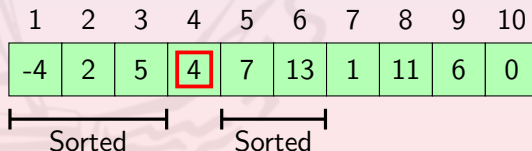


# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “enlarge” it by inserting **next element**

by swapping **it** and the previous one in the array

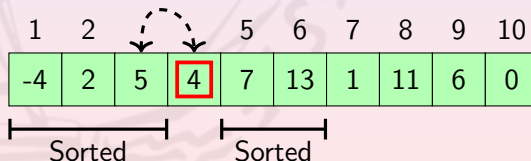


# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “*enlarge*” it by inserting **next element**

by swapping **it** and the previous one in the array



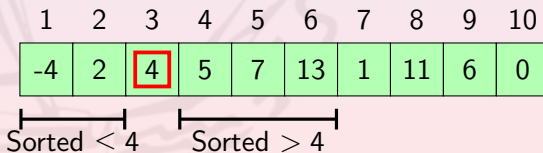
# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “enlarge” it by inserting **next element**

by swapping **it** and the previous one in the array

until the previous one (if exists) is greater than **it**



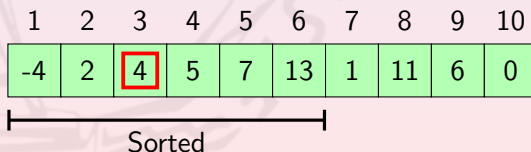
# Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can “*enlarge*” it by inserting **next element**

by swapping **it** and the previous one in the array

until the previous one (if exists) is greater than **it**



# Insertion Sort: Code and Complexity

```
def insertion_sort(A):  
    for i in 2..|A|:  
        j ← i  
        while (j>1 and  
               A[j]<A[j-1]):  
            swap(A, j-1, j)  
            j ← j-1  
        endwhile  
    endfor  
enddef
```

The while-loop block costs  $\Theta(1)$

It iterates  $O(i)$  and  $\Omega(1)$  times for  
all  $i \in [2, n]$

$$\sum_{i=2}^n O(i) * O(1) = O\left(\sum_{i=2}^n i\right) \\ = O(n^2)$$

$$\sum_{i=2}^n \Omega(1) * \Omega(1) = \Omega\left(\sum_{i=2}^n 1\right) \\ = \Omega(n)$$

Retrieving Data  
○○○○○○○

Sorting  
○○

Insertion Sort  
○○○

Quick Sort  
●○○○○○○○○○○○

Heap Sort  
○○○○○

Sorting in Linear Time  
○○○○○○○○○○○○○

Select  
○○○○○○○○○

## Quick Sort

## Quick Sort: Intuition

Select one element of the A: the **pivot**



1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0

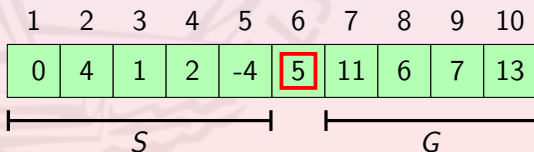


# Quick Sort: Intuition

Select one element of the  $A$ : the **pivot**

**partition**  $A$  in:

- subarray  $S$  of the elements smaller or equal to the pivot
- the pivot
- subarray  $G$  of the elements greater than the pivot



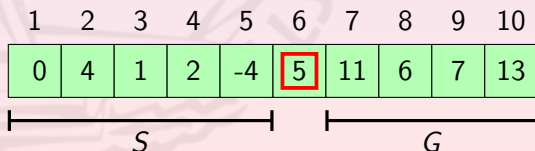
# Quick Sort: Intuition

Select one element of the  $A$ : the **pivot**

**partition**  $A$  in:

- subarray  $S$  of the elements smaller or equal to the pivot
- the pivot
- subarray  $G$  of the elements greater than the pivot

Repeat on the subarrays having more than 1 elements



## Quick Sort: Intuition (Cont'd)

At the end of every iteration of above steps:

- the elements in  $S$  stay in  $S$  if  $A$  is sorted
- the elements in  $G$  stay in  $G$  if  $A$  is sorted
- the pivot is in its “sorted” position
- $S$  and  $G$  are shorter than  $A$

## Quick Sort: Intuition (Cont'd)

At the end of every iteration of above steps:

- the elements in  $S$  stay in  $S$  if  $A$  is sorted
- the elements in  $G$  stay in  $G$  if  $A$  is sorted
- the pivot is in its “sorted” position
- $S$  and  $G$  are shorter than  $A$

An iteration places at least one element in the correct position

It prepares  $A$  for two recursive calls on  $S$  and  $G$ .

## Quick Sort: Intuition (Cont'd)

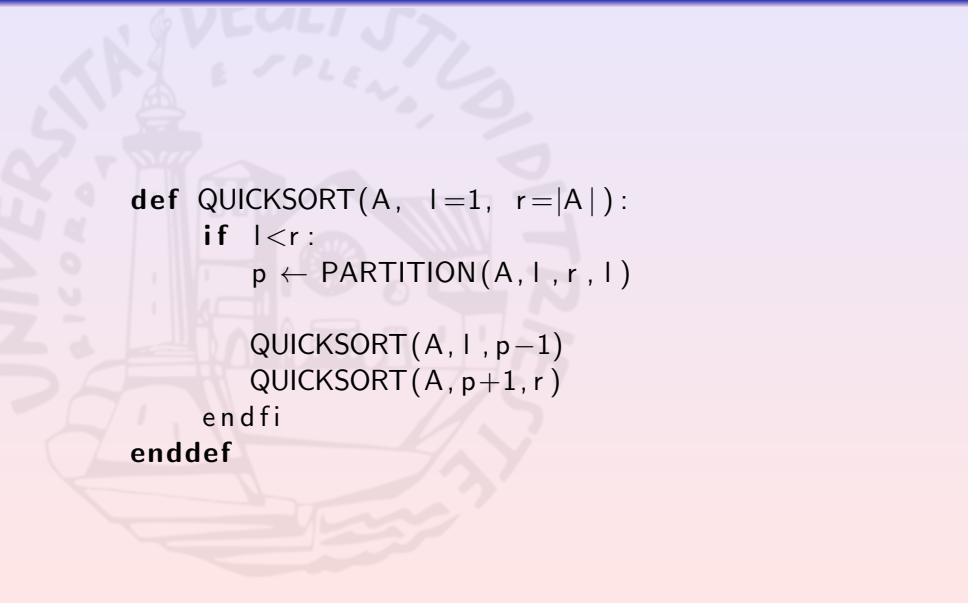
At the end of every iteration of above steps:

- the elements in  $S$  stay in  $S$  if  $A$  is sorted
- the elements in  $G$  stay in  $G$  if  $A$  is sorted
- the pivot is in its “sorted” position
- $S$  and  $G$  are shorter than  $A$

An iteration places at least one element in the correct position

It prepares  $A$  for two recursive calls on  $S$  and  $G$ .

## Quick Sort: Pseudo-Code



```
def QUICKSORT(A, l=1, r=|A|):  
    if l < r:  
        p ← PARTITION(A, l, r, l)  
  
        QUICKSORT(A, l, p-1)  
        QUICKSORT(A, p+1, r)  
    endfi  
enddef
```

## Quick Sort: Pseudo-Code

The last recursion call is a **tail recursion**

```
def QUICKSORT(A, l=1, r=|A|):  
    while l < r:  
        p ← PARTITION(A, l, r, l)  
  
        QUICKSORT(A, l, p-1)  
        l ← p+1  
    endwhile  
enddef
```

## Quick Sort: Complexity

The time complexity  $T_Q$  of quick sort will be

$$T_Q(|A|) = \begin{cases} \Theta(1) & \text{if } |A| = 1 \\ T_Q(|S|) + T_Q(|G|) + T_P(|A|) & \text{otherwise} \end{cases}$$

$T_P$  is the complexity of **partition**



# Quick Sort: Complexity

The time complexity  $T_Q$  of quick sort will be

$$T_Q(|A|) = \begin{cases} \Theta(1) & \text{if } |A| = 1 \\ T_Q(|S|) + T_Q(|G|) + T_P(|A|) & \text{otherwise} \end{cases}$$

$T_P$  is the complexity of **partition**

Is the pivot selection relevant?

# Quick Sort: Complexity

The time complexity  $T_Q$  of quick sort will be

$$T_Q(|A|) = \begin{cases} \Theta(1) & \text{if } |A| = 1 \\ T_Q(|S|) + T_Q(|G|) + T_P(|A|) & \text{otherwise} \end{cases}$$

$T_P$  is the complexity of **partition**

Is the pivot selection relevant? No, choose whatever you want

Which algorithm is the best for **partition**?

# Partition: An In-place Algorithm

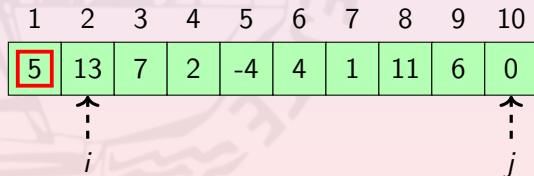
Switch the pivot **p** and the first element in *A*



# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in *A*

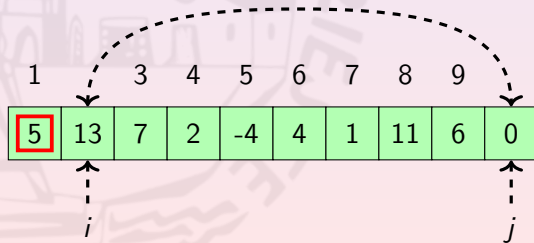
If  **$A[i] > p$** ,



# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in  $A$

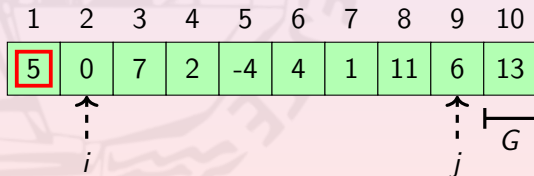
If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$



# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in  $A$

If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$

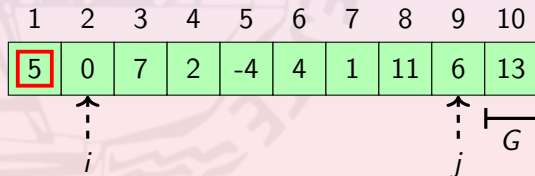


# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in  $A$

If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$

else ( $A[i] \leq p$ ),

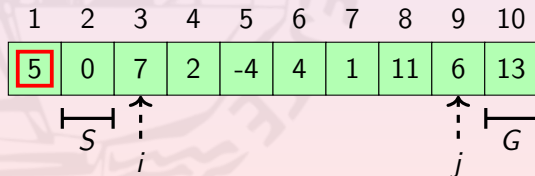


# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in  $A$

If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$

else ( $A[i] \leq p$ ), increase  $i$





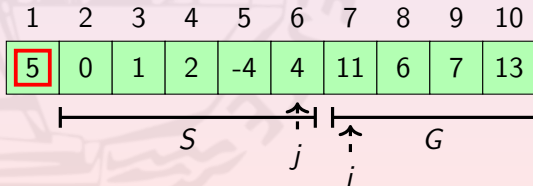
# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in  $A$

If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$

else ( $A[i] \leq p$ ), increase  $i$

Repeat until  $i \leq j$



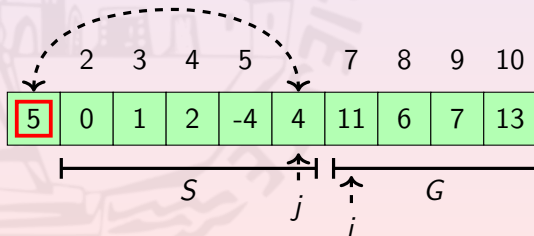
# Partition: An In-place Algorithm

Switch the pivot **p** and the first element in  $A$

If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$

else ( $A[i] \leq p$ ), increase  $i$

Repeat until  $i \leq j$  and swap **p** and  $A[j]$



# Partition: An In-place Algorithm

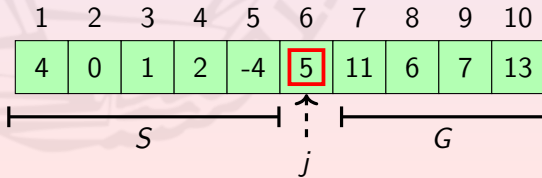
Switch the pivot **p** and the first element in  $A$

If  $A[i] > p$ , swap  $A[i]$  and  $A[j]$  and decrease  $j$

else ( $A[i] \leq p$ ), increase  $i$

Repeat until  $i \leq j$  and swap **p** and  $A[j]$

**The complexity is  $\Theta(|A|)$**



# Partition: Pseudo-Code

```

def PARTITION(A, i, j, p):
    swap(A, i, p)
    (p, i)  $\leftarrow$  (i, i+1)

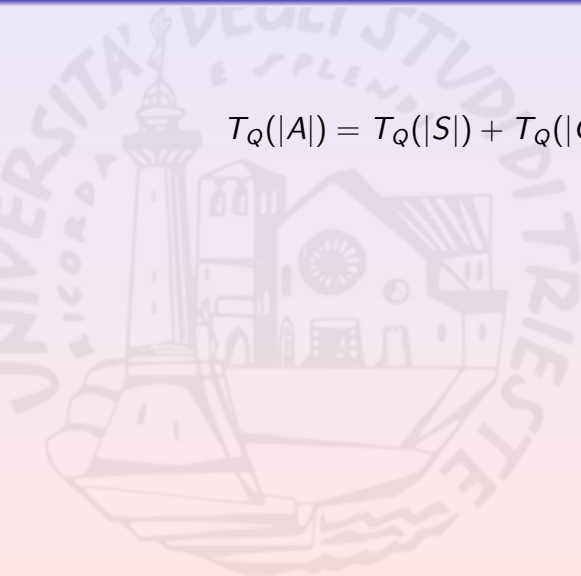
    while i  $\leq$  j:
        if A[i] > A[p]:      # if A[i] is greater than the pivot
            swap(A, i, j)  # place it in G
            j  $\leftarrow$  j+1  # increase G's size
        else              # otherwise
            i  $\leftarrow$  i+1  # A[i] is already in S
        endif
    endwhile

    swap(A, p, j)  # place the pivot between S and G
    return j
enddef

```

## Quick Sort Complexity: Worst Case

$$T_Q(|A|) = T_Q(|S|) + T_Q(|G|) + \Theta(|A|)$$



## Quick Sort Complexity: Worst Case

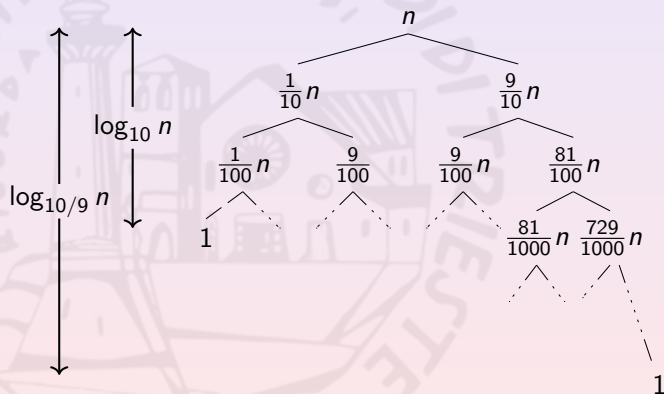
$$T_Q(|A|) = T_Q(|S|) + T_Q(|G|) + \Theta(|A|)$$

**Worst Case:**  $|G| = 0$  or  $|S| = 0$  for all recursive call.

$$\begin{aligned} T_Q(n) &= T_Q(n-1) + \Theta(n) \\ &= \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{i=0}^n i\right) \\ &= \Theta(n^2) \end{aligned}$$

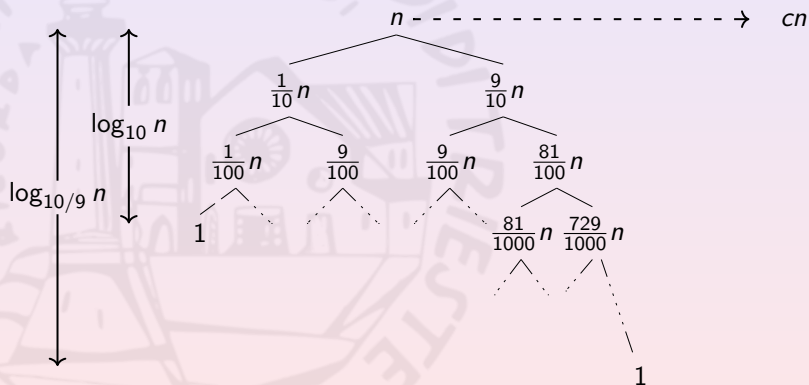
# Quick Sort Complexity: Best Case

**Best Case:** Balanced Partition



# Quick Sort Complexity: Best Case

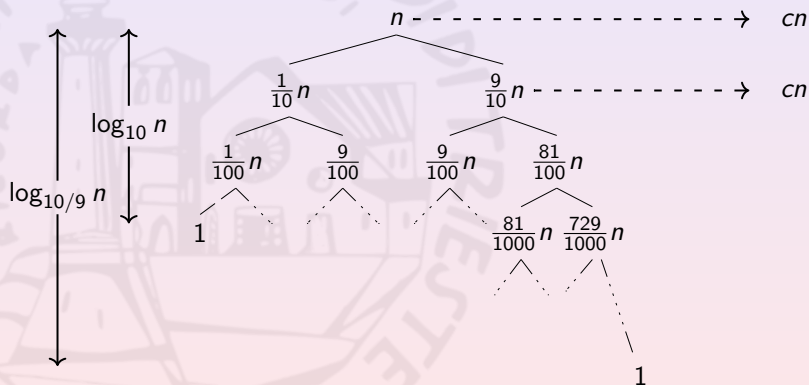
**Best Case:** Balanced Partition





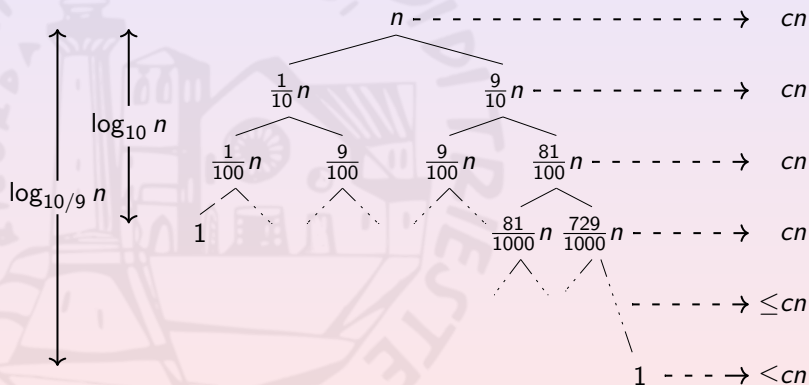
# Quick Sort Complexity: Best Case

**Best Case:** Balanced Partition



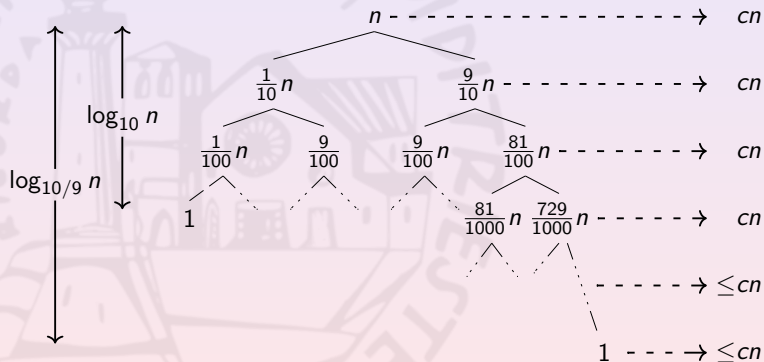
# Quick Sort Complexity: Best Case

**Best Case:** Balanced Partition



# Quick Sort Complexity: Best Case

**Best Case:** Balanced Partition



$$\Theta(n \log n)$$

## Quick Sort Complexity: Average Case

“Good” and “bad” cases depend on the ordering of  $A$

If all the permutations of  $A$  are equally likely,

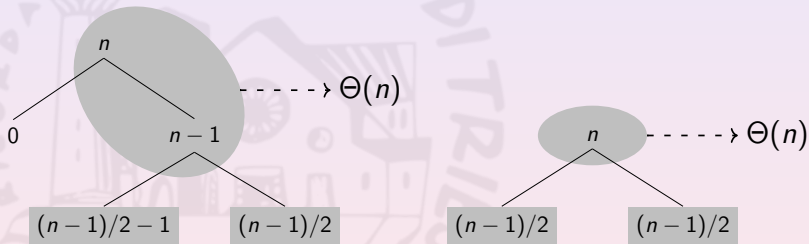
the partition has a ratio more balanced than  $1/d$  with probability

$$\frac{d-1}{d+1}$$

e.g., a partition “better” than  $1/9$  has probability 0.8

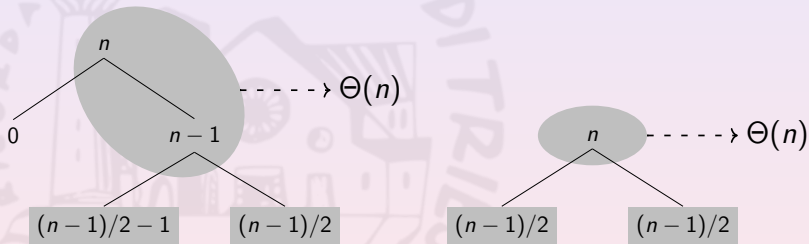
# Quick Sort Complexity: Average Case (Cont'd)

Even if “good” and “bad” cases alternate



## Quick Sort Complexity: Average Case (Cont'd)

Even if “good” and “bad” cases alternate



On the average  $\Theta(n \log n)$

The background of the slide features a large, faint watermark of the University of Trieste logo. The logo is circular and contains the text "UNIVERSITA' DEGLI STUDI DI TRIESTE" around the perimeter and "E SPLENDI" in the center. In the middle of the logo is a detailed illustration of a building, likely a university hall or library, with a dome and a clock tower.

## Heap Sort

# Sorting by Searching the Maximum

Find the maximum

1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0



# Sorting by Searching the Maximum

Find the maximum

**Move the maximum at the end of the array**

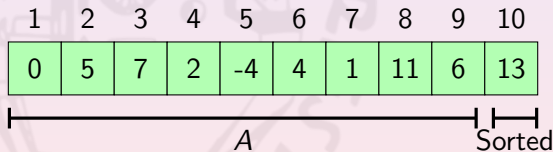
1	2	3	4	5	6	7	8	9	10
0	5	7	2	-4	4	1	11	6	13

# Sorting by Searching the Maximum

Find the maximum

Move the maximum at the end of the array

**If  $|A| > 1$ , repeat on the initial fragment of  $A$**

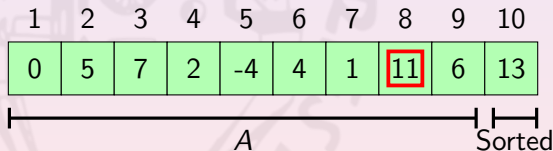


# Sorting by Searching the Maximum

## Find the maximum

Move the maximum at the end of the array

If  $|A| > 1$ , repeat on the initial fragment of  $A$

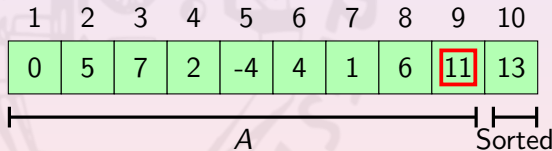


# Sorting by Searching the Maximum

Find the maximum

**Move the maximum at the end of the array**

If  $|A| > 1$ , repeat on the initial fragment of  $A$

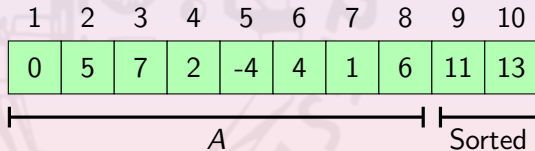


# Sorting by Searching the Maximum

Find the maximum

Move the maximum at the end of the array

**If  $|A| > 1$ , repeat on the initial fragment of  $A$**

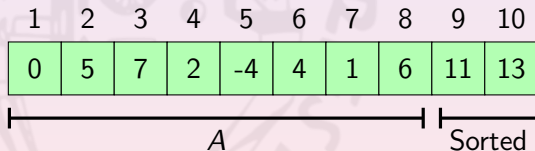


# Sorting by Searching the Maximum

Find the maximum

Move the maximum at the end of the array

**If  $|A| > 1$ , repeat on the initial fragment of  $A$**



The complexity is  $\sum_{i=1}^{|A|} (T_{\max}(i) + \Theta(1))$

# How to Find the Maximum?

By using ...

- pushing the max to the right

⇒ Bubble Sort

$$\begin{aligned}T(|A|) &= \sum_{i=1}^{|A|} (\Theta(i) + \Theta(1)) \\ &= \Theta(|A|^2)\end{aligned}$$

- binary heap (see [here](#))

⇒ Heap Sort

# Heap Sort: Pseudo-Code

The array-based implementation of binary heap plays a crucial role

```

def HEAPSORT(A):
    H  $\leftarrow$  BUILD_MAX_HEAP(A) # the root is the max

    for i  $\leftarrow$  |A| downto 2:
        swap(A, 1, i)

        H.size  $\leftarrow$  H.size - 1 # remove the last leaf
        HEAPIFY(H, 1) # fix the max-heap
    endfor
enddef
  
```



# Heap Sort: Complexity

Building the binary heap costs  $\Theta(n)$

HEAPIFY costs  $O(\log i)$  per iteration and in total

$$\sum_{i=2}^n \log i \leq \sum_{i=2}^n \log n \in O(n \log n)$$

The overall complexity of heap sort is  $O(n \log n)$

Retrieving Data  
○○○○○○○

Sorting  
○○

Insertion Sort  
○○○

Quick Sort  
○○○○○○○○○○○○

Heap Sort  
○○○○○

**Sorting in Linear Time**  
●○○○○○○○○○○○○

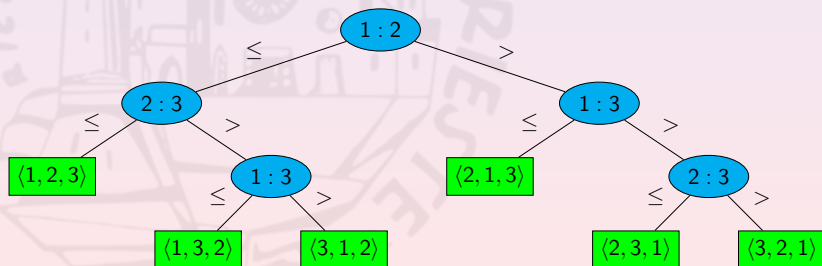
Select  
○○○○○○○○○

## Sorting in Linear Time

# Sorting By Comparison: Lower Bound

The execution of a sorting-by-comparison algorithm can be modeled as a **decision-tree model**

Any comparison between  $a_i$  and  $a_j$  corresponds to a node which branches the computation according to whether  $a_i \leq a_j$  or  $a_i > a_j$



## Sorting By Comparison: Lower Bound (Cont'd)

The decision tree's leaves are labeled by all the possible permutations of  $A$  which are  $n!$

The height  $h$  is the maximum # of comparisons required by the algorithm

Since a binary tree has no more than  $2^h$  leaves,

$$h \geq \log_2(n!) \in \Omega(n \log n)$$

## Sorting By Comparison: Lower Bound (Cont'd)

The decision tree's leaves are labeled by all the possible permutations of  $A$  which are  $n!$

The height  $h$  is the maximum # of comparisons required by the algorithm

Since a binary tree has no more than  $2^h$  leaves,

$$h \geq \log_2(n!) \in \Omega(n \log n)$$

The lower bound for comparison-based sorting is  $\Omega(n \log n)$

# Sorting in Linear Time?

There is no **general** algorithm to sort in linear time by using comparisons

# Sorting in Linear Time?

There is no **general** algorithm to sort in linear time by using comparisons

This bound does not hold if we introduce minor *ad-hoc* assumptions such as:

- bounded domain for the array values
- uniform distribution of the array values

# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$

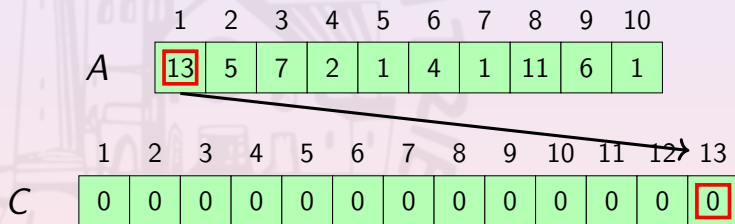
	1	2	3	4	5	6	7	8	9	10
$A$	13	5	7	2	1	4	1	11	6	1

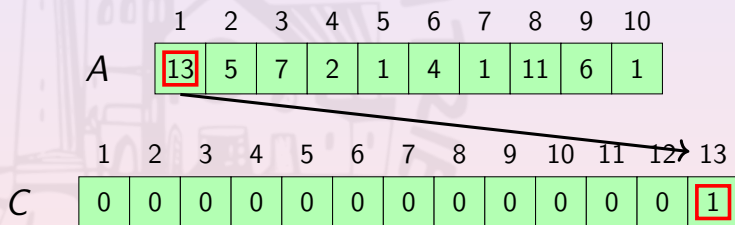




[illegible]

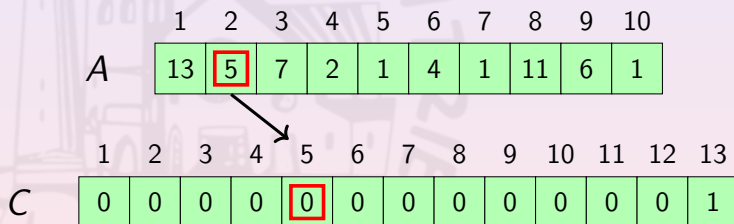
- count the occurrences of  $A$ 's values and place them in  $C$





- count the occurrences of  $A$ 's values and place them in  $C$

[illegible]



## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$

	1	2	3	4	5	6	7	8	9	10	
$A$	13	5	7	2	1	4	1	11	6	1	

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	0	0	0	0	1	0	0	0	0	0	0	0	1

# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$

	1	2	3	4	5	6	7	8	9	10
$A$	13	5	7	2	1	4	1	11	6	1

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	1	0	1	1	1	1	0	0	0	1	0	1



# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes

	1	2	3	4	5	6	7	8	9	10	
$A$	13	5	7	2	1	4	1	11	6	1	

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	1	0	1	1	1	1	0	0	0	1	0	1

Diagram illustrating the first step of Counting Sort: counting occurrences. An arrow points from the value 13 in array  $A$  to the index 13 in array  $C$ , where the value 1 is added to the existing value 3.

# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes

	1	2	3	4	5	6	7	8	9	10	
$A$	13	5	7	2	1	4	1	11	6	1	

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	4	0	1	1	1	1	0	0	0	1	0	1

Diagram illustrating the first step of Counting Sort. Array  $A$  contains values [13, 5, 7, 2, 1, 4, 1, 11, 6, 1]. Array  $C$  contains the counts for each index from 1 to 13: [3, 4, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1]. A curved arrow points from the first element of  $C$  (3) to a plus sign below it, indicating the start of the cumulative sum process.

# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes

	1	2	3	4	5	6	7	8	9	10	
$A$	13	5	7	2	1	4	1	11	6	1	

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	4	0	1	1	1	1	0	0	0	1	0	1

+  
↻

# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes

	1	2	3	4	5	6	7	8	9	10	
$A$	13	5	7	2	1	4	1	11	6	1	

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	4	4	1	1	1	1	0	0	0	1	0	1

+  
↻

# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes

	1	2	3	4	5	6	7	8	9	10
$A$	13	5	7	2	1	4	1	11	6	1

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	4	4	5	6	7	8	8	8	8	9	9	10

## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the  $\#$  elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$

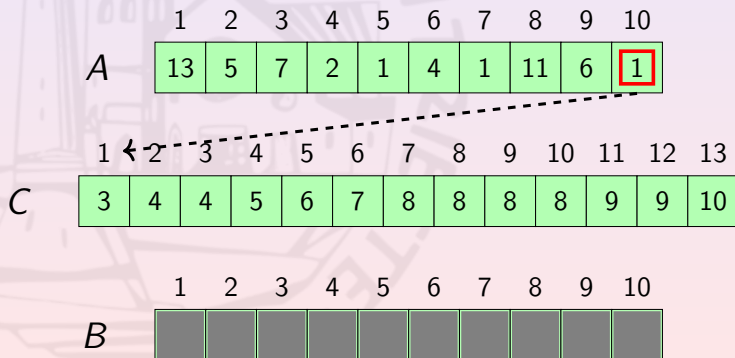
	1	2	3	4	5	6	7	8	9	10
A	13	5	7	2	1	4	1	11	6	1

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	3	4	4	5	6	7	8	8	8	8	9	9	10

$B$

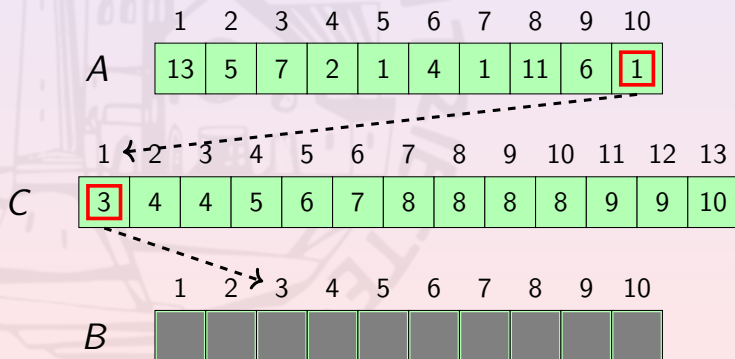
## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$



## Values in $[1, k]$ : Counting Sort

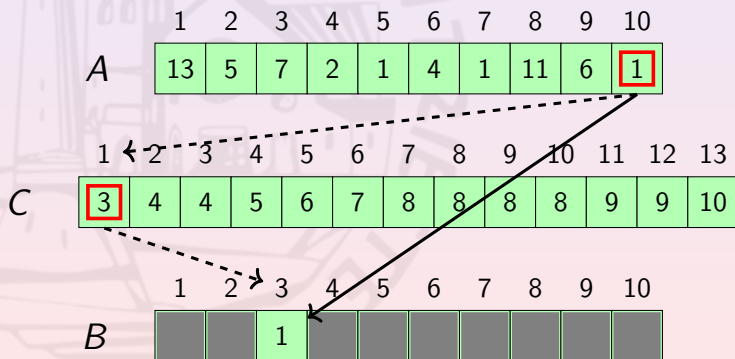
- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$





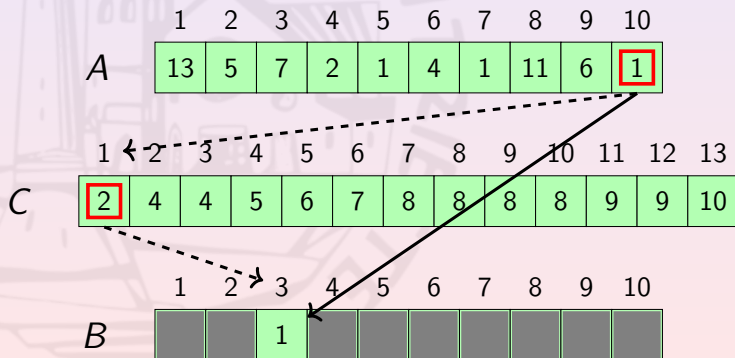
# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$



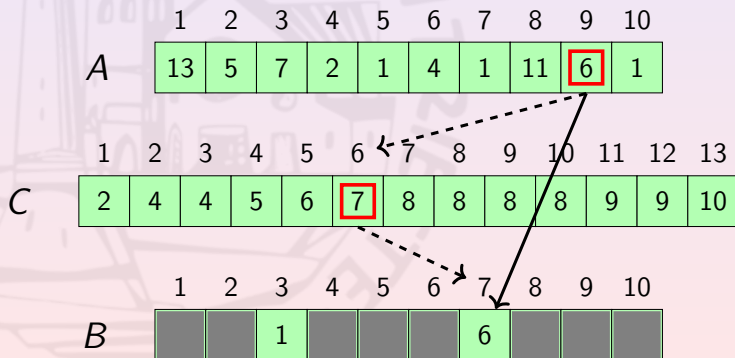
## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$



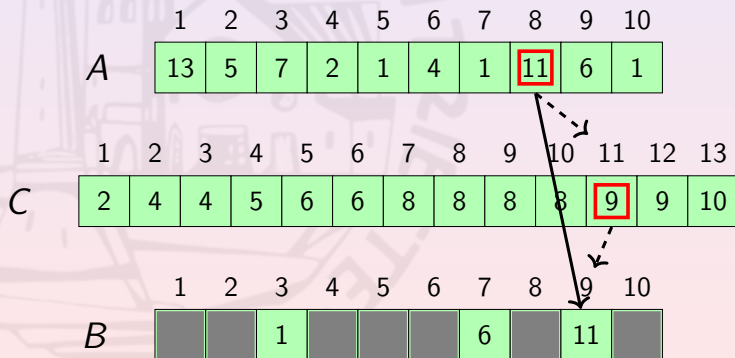
## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$



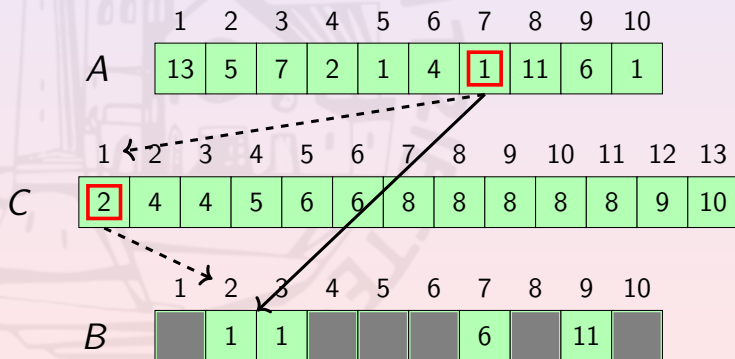
## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$



# Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$



## Values in $[1, k]$ : Counting Sort

- count the occurrences of  $A$ 's values and place them in  $C$
- sums the values in  $C$  and get the # elements  $\leq$  to  $C$ 's indexes
- use  $C$  to place the elements of  $A$  in the correct positions in  $B$

	1	2	3	4	5	6	7	8	9	10
$A$	13	5	7	2	1	4	1	11	6	1

	1	2	3	4	5	6	7	8	9	10	11	12	13
$C$	0	3	4	4	5	6	7	8	8	8	8	9	9

	1	2	3	4	5	6	7	8	9	10
$B$	1	1	1	2	4	5	6	7	11	13

# Some Observations about Counting Sort

Why backward placing?



## Some Observations about Counting Sort

Why backward placing? For **stability**, i.e., preserving relative order of equivalent elements



## Some Observations about Counting Sort

Why backward placing? For **stability**, i.e., preserving relative order of equivalent elements

Generalizing it to deal with any  $[k_1, k_2]$  domain is easy

## Some Observations about Counting Sort

Why backward placing? For **stability**, i.e., preserving relative order of equivalent elements

Generalizing it to deal with any  $[k_1, k_2]$  domain is easy

It is not **in-place** and it requires the array  $C$

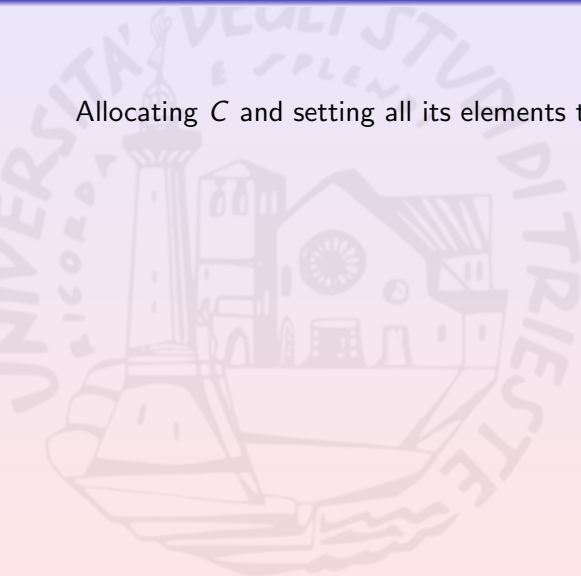
# Counting Sort: Pseudo-Code

```
def COUNTING_SORT(A,B,k):  
    C ← ALLOCATE_ARRAY(k, default_value=0)  
    for i ← 1 upto |A|:  
        C[A[i]] ← C[A[i]]+1  
    endfor # C[j] is now the # of j in A  
  
    for j ← 2 upto |C|:  
        C[j] ← C[j-1] + C[j]  
    endfor # C[j] is now the # of A's values ≤ j  
  
    for i ← |A| downto 1:  
        B[C[A[i]]] ← A[i]  
        C[A[i]] ← C[A[i]]-1  
    endfor  
enddef
```

# Counting Sort: Complexity

Allocating  $C$  and setting all its elements to 0

$$\Theta(k)$$



# Counting Sort: Complexity

Allocating  $C$  and setting all its elements to 0

$$\Theta(k)$$

Counting the instances of  $A$ 's values

$$\Theta(n)$$

# Counting Sort: Complexity

Allocating  $C$  and setting all its elements to 0

$\Theta(k)$

Counting the instances of  $A$ 's values

$\Theta(n)$

Setting in  $C[j]$  the # of  $A$ 's values  $\leq j$

$\Theta(k)$

# Counting Sort: Complexity

Allocating  $C$  and setting all its elements to 0

$\Theta(k)$

Counting the instances of  $A$ 's values

$\Theta(n)$

Setting in  $C[j]$  the # of  $A$ 's values  $\leq j$

$\Theta(k)$

Copying  $A$ 's values into  $B$  by using  $C$

$\Theta(n)$

# Counting Sort: Complexity

Allocating  $C$  and setting all its elements to 0

$$\Theta(k)$$

Counting the instances of  $A$ 's values

$$\Theta(n)$$

Setting in  $C[j]$  the # of  $A$ 's values  $\leq j$

$$\Theta(k)$$

Copying  $A$ 's values into  $B$  by using  $C$

$$\Theta(n)$$

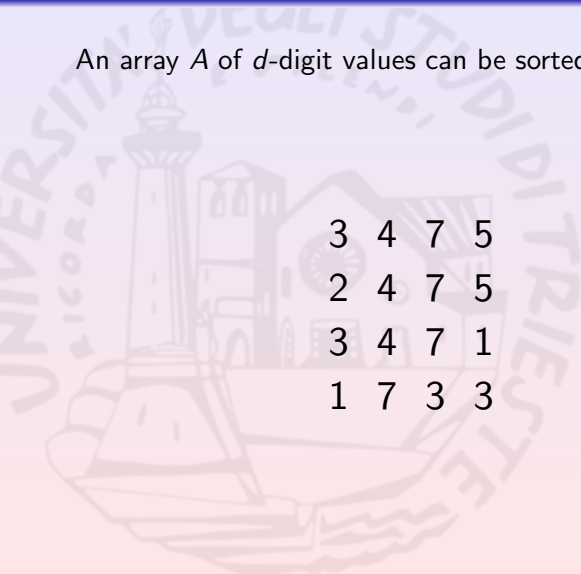
Total complexity

$$\Theta(n + k)$$



## Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit



3	4	7	5
2	4	7	5
3	4	7	1
1	7	3	3

## Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

3	4	7	5
2	4	7	5
3	4	7	1
1	7	3	3

↑  
⋮  
 $i$

## Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

3	4	7	1
1	7	3	3
3	4	7	5
2	4	7	5

↑  
⋮  
 $i$

# Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

3	4	7	1
1	7	3	3
3	4	7	5
2	4	7	5

↑  
⋮  
 $i$

# Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

1	7	3	3
3	4	7	1
3	4	7	5
2	4	7	5

↑  
⋮  
 $i$

## Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

1	7	3	3
3	4	7	1
3	4	7	5
2	4	7	5

↑  
⋮  
 $i$

# Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

3	4	7	1
3	4	7	5
2	4	7	5
1	7	3	3

↑  
⋮  
 $i$

# Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

3	4	7	1
3	4	7	5
2	4	7	5
1	7	3	3
↑			
$i$			



# Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

1	7	3	3
2	4	7	5
3	4	7	1
3	4	7	5

↑  
⋮  
 $i$

## Fixed Number of Digits: Radix Sort

An array  $A$  of  $d$ -digit values can be sorted digit-by-digit

- for each digit  $i$  from the rightmost down to the leftmost
- use a **stable algorithm** and sort  $A$  according the digit  $i$

1	7	3	3
2	4	7	5
3	4	7	1
3	4	7	5

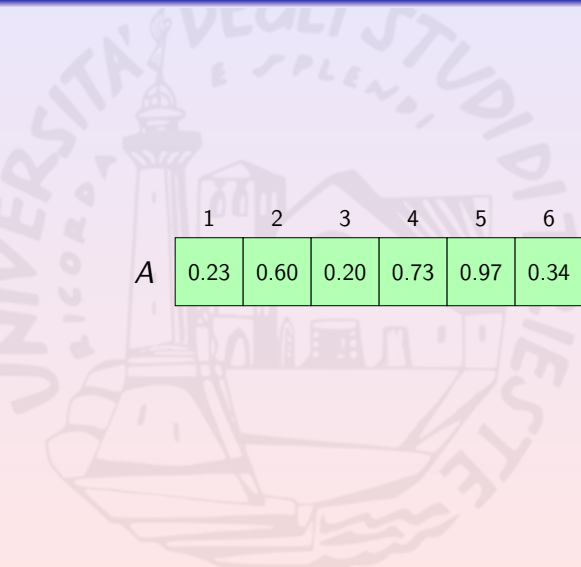
# Radix Sort: Complexity

If the digit sorting is in  $\Theta(|A| + k)$ , radix sort takes time

$$\Theta(d(|A| + k))$$

where  $d$  is the number of digits in each of  $A$ 's values

# Uniform Distribution in $[0, 1)$ : Bucket Sort

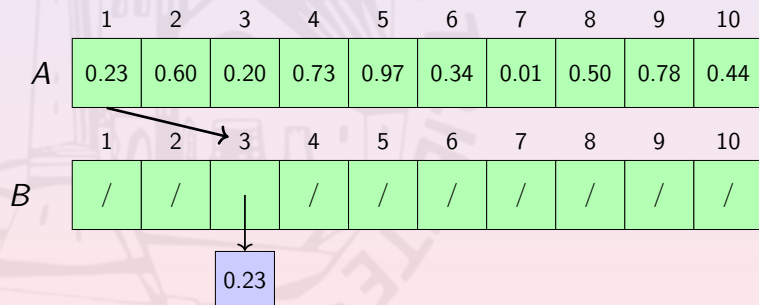


	1	2	3	4	5	6	7	8	9	10
A	0.23	0.60	0.20	0.73	0.97	0.34	0.01	0.50	0.78	0.44

[illegible]

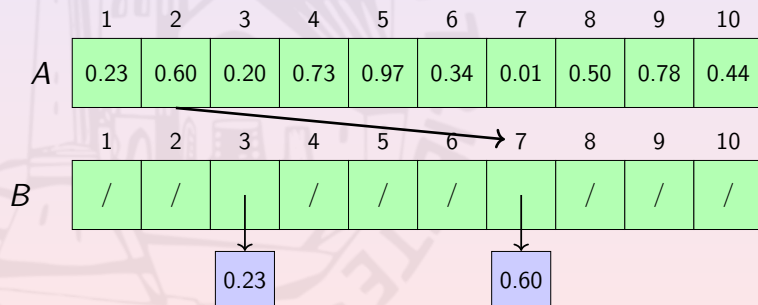
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket



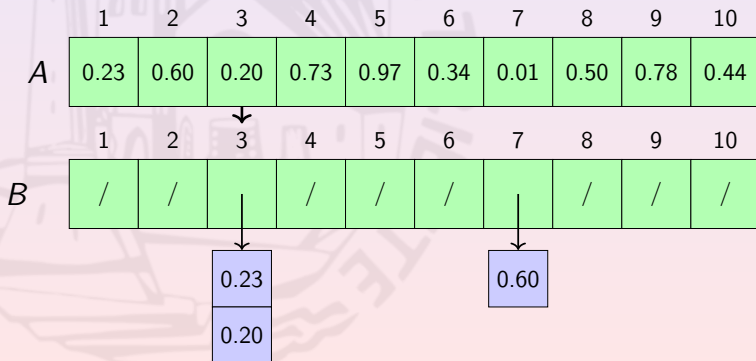
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket



# Uniform Distribution in $[0, 1)$ : Bucket Sort

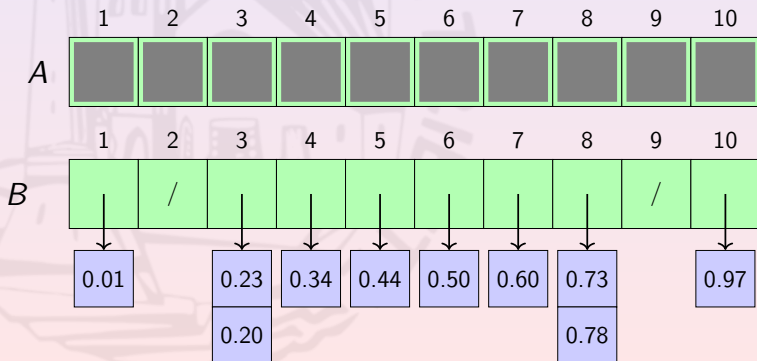
- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket





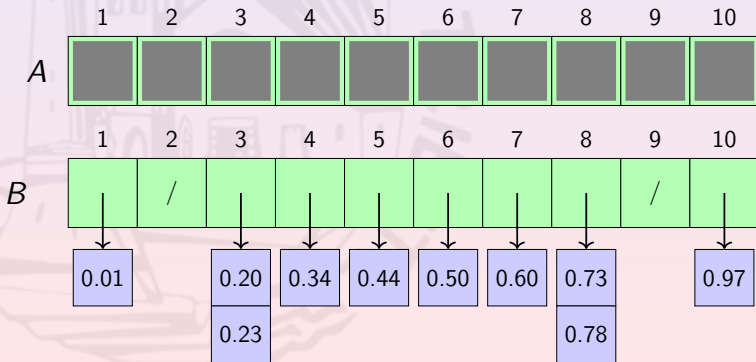
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket



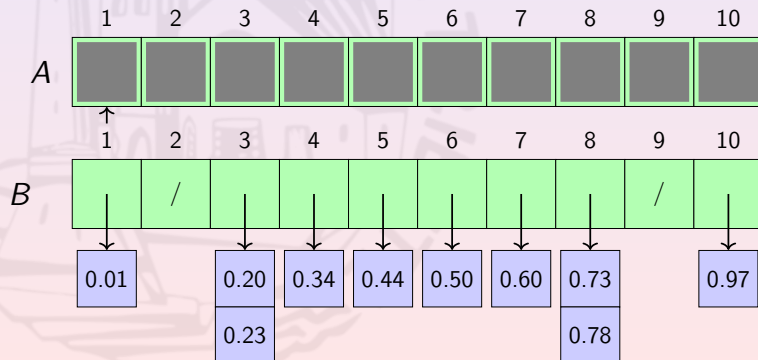
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket
- sort the buckets



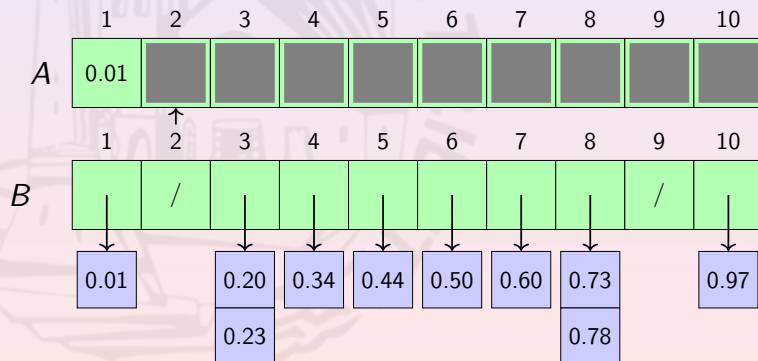
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket
- sort the buckets
- reverse the content of buckets in bucket order on  $A$



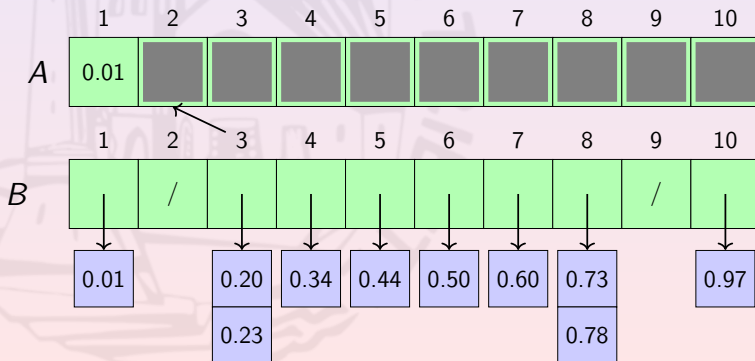
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket
- sort the buckets
- reverse the content of buckets in bucket order on  $A$



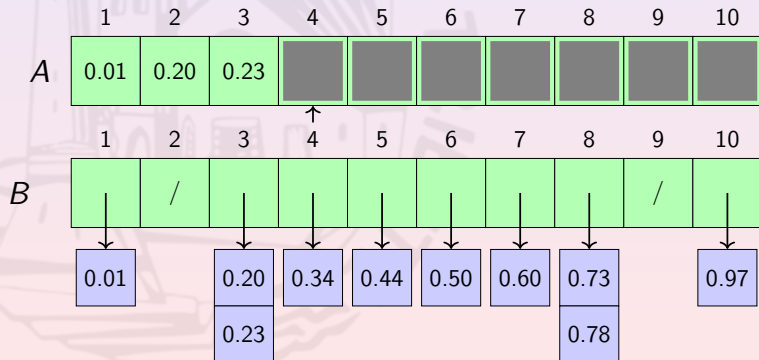
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket
- sort the buckets
- reverse the content of buckets in bucket order on  $A$



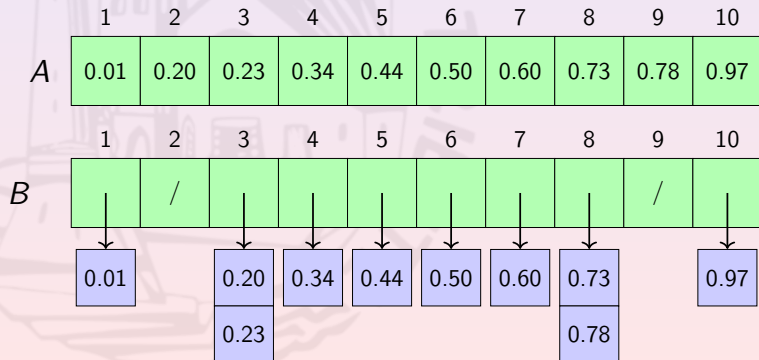
# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket
- sort the buckets
- reverse the content of buckets in bucket order on  $A$



# Uniform Distribution in $[0, 1)$ : Bucket Sort

- split  $[0, 1)$  in  $n$  buckets:  $[\frac{i-1}{n}, \frac{i}{n})$  for  $i \in [1, n]$
- add each value of  $A$  to the correct bucket
- sort the buckets
- reverse the content of buckets in bucket order on  $A$



# Bucket Sort: Pseudo-Code

```
def BUCKET_SORT(A):  
    B ← ALLOCATE_ARRAY_OF_EMPTY_LISTS(|A|)  
  
    for i ← 1 upto |A|:  
        B[FLOOR(A[i]*n)+1].append(A[i])  
    endfor # now B contains the buckets  
  
    i ← 0  
    for j ← 1 upto |B|  
        for v in B[j]: # reverse the bucket in A  
            A[i] ← v  
            i ← i+1  
        endfor  
  
    sort(A, i-|B[j]|, |B[j]|) # sort the bucket
```



# Bucket Sort: Expected Complexity

Allocating and initializing  $B$

$$\Theta(n)$$

# Bucket Sort: Expected Complexity

Allocating and initializing  $B$

$$\Theta(n)$$

Filling the buckets

$$\Theta(n)$$

# Bucket Sort: Expected Complexity

Allocating and initializing  $B$

$$\Theta(n)$$

Filling the buckets

$$\Theta(n)$$

Sorting each bucket (**expected**)

$$O(n)$$

# Bucket Sort: Expected Complexity

Allocating and initializing  $B$

$$\Theta(n)$$

Filling the buckets

$$\Theta(n)$$

Sorting each bucket (**expected**)

$$O(n)$$

Reversing buckets' content into  $A$

$$\Theta(n)$$

# Bucket Sort: Expected Complexity

Allocating and initializing  $B$

$$\Theta(n)$$

Filling the buckets

$$\Theta(n)$$

Sorting each bucket (**expected**)

$$O(n)$$

Reversing buckets' content into  $A$

$$\Theta(n)$$

Total **expected** complexity

$$O(n)$$

Retrieving Data  
○○○○○○○

Sorting  
○○

Insertion Sort  
○○○

Quick Sort  
○○○○○○○○○○○○

Heap Sort  
○○○○○

Sorting in Linear Time  
○○○○○○○○○○○○○

Select  
●○○○○○○○

Select

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1?

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1? Complexity?



## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1? Complexity?

$\Theta(n)$

- $n$ ?

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1? Complexity?

$\Theta(n)$

- $n$ ? Complexity?

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1? Complexity?  $\Theta(n)$
- $n$ ? Complexity?  $\Theta(n)$

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1? Complexity?  $\Theta(n)$
- $n$ ? Complexity?  $\Theta(n)$
- $i \in [1, n]$ ?

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

- 1? Complexity?  $\Theta(n)$
- $n$ ? Complexity?  $\Theta(n)$
- $i \in [1, n]$ ? Complexity?

## Some Interesting Questions

Let  $A$  be unsorted array

How to find the value that, if  $A$  was sorted, would be in position:

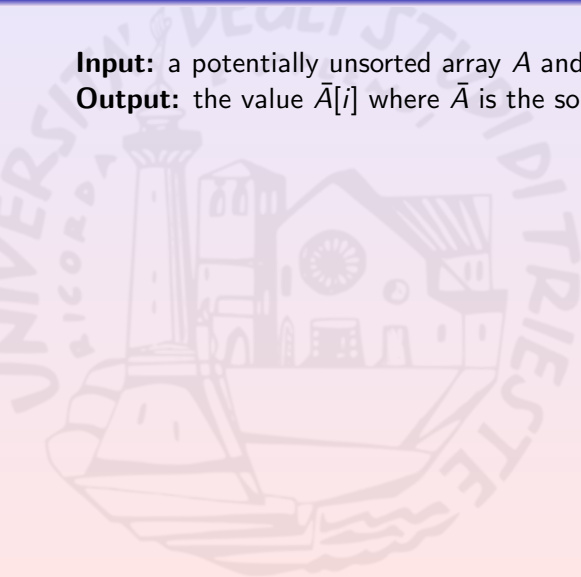
- 1? Complexity?  $\Theta(n)$
- $n$ ? Complexity?  $\Theta(n)$
- $i \in [1, n]$ ? Complexity?  $O(n \log n)$

Can we do better?

# The Select Problem

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** the value  $\bar{A}[i]$  where  $\bar{A}$  is the sorted version of  $A$



# The Select Problem

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** the value  $\bar{A}[i]$  where  $\bar{A}$  is the sorted version of  $A$

We reformulate it as:

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** an index  $j$  s.t.  $\tilde{A}[j] = \bar{A}[i]$  where  $\tilde{A}$  and  $\bar{A}$  are  $A$  after the computation and the sorted version of  $A$ , respectively.



# The Select Problem

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** the value  $\bar{A}[i]$  where  $\bar{A}$  is the sorted version of  $A$

We reformulate it as:

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** an index  $j$  s.t.  $\tilde{A}[j] = \bar{A}[i]$  where  $\tilde{A}$  and  $\bar{A}$  are  $A$  after the computation and the sorted version of  $A$ , respectively.

We do not want to build an **index**: it is a *una-tantum* query

# The Select Problem

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** the value  $\bar{A}[i]$  where  $\bar{A}$  is the sorted version of  $A$

We reformulate it as:

**Input:** a potentially unsorted array  $A$  and an index  $i \in [1, |A|]$

**Output:** an index  $j$  s.t.  $\tilde{A}[j] = \bar{A}[i]$  where  $\tilde{A}$  and  $\bar{A}$  are  $A$  after the computation and the sorted version of  $A$ , respectively.

We do not want to build an **index**: it is a *una-tantum* query

We will assume that  $A$  does not contains multiple instances of the same value (not necessary, but simplify things)

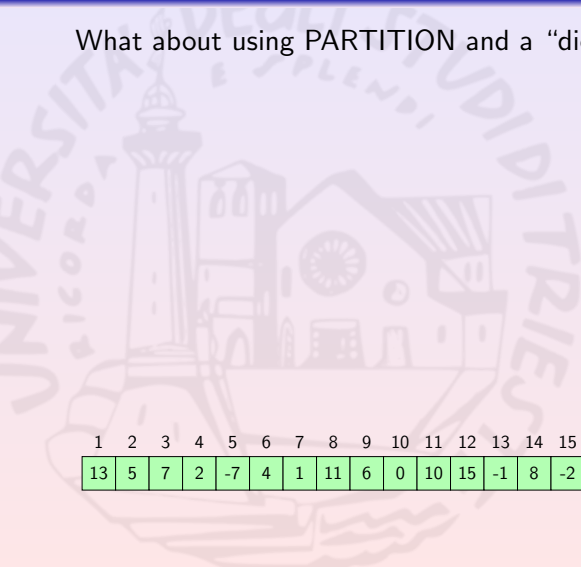
## A Possible Strategy

What about using PARTITION and a “dichotomic approach”?



## A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

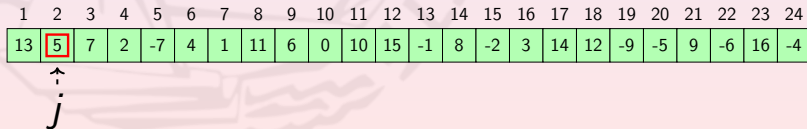


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	5	7	2	-7	4	1	11	6	0	10	15	-1	8	-2	3	14	12	-9	-5	9	-6	16	-4

## A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

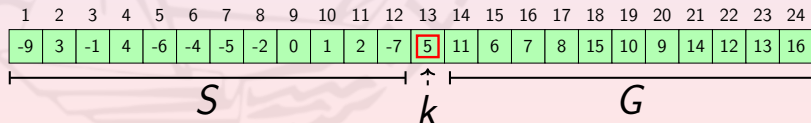
- select a pivot  $A[j]$



# A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

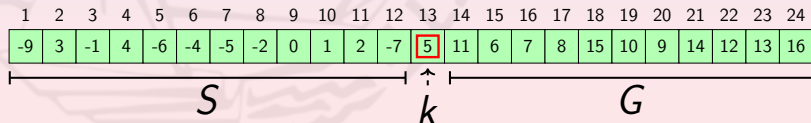
- select a pivot  $A[j]$
- compute  $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$  and get  $S$  and  $G$



## A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

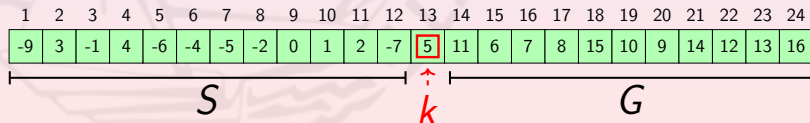
- select a pivot  $A[j]$
- compute  $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$  and get  $S$  and  $G$
- compare  $i$  and  $k$



# A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

- select a pivot  $A[j]$
- compute  $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$  and get  $S$  and  $G$
- compare  $i$  and  $k$ 
  - if  $i = k$ , then return  $A[k]$

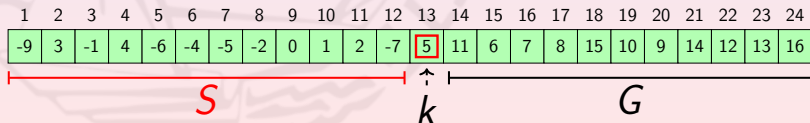




## A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

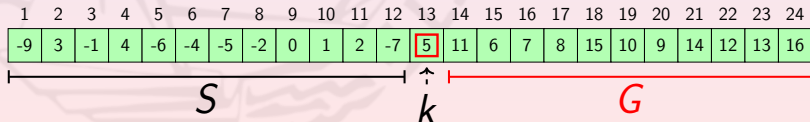
- select a pivot  $A[j]$
- compute  $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$  and get  $S$  and  $G$
- compare  $i$  and  $k$ 
  - if  $i = k$ , then return  $A[k]$
  - if  $i < k$ , then  $\bar{A}[i]$  is in  $S$



# A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

- select a pivot  $A[j]$
- compute  $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$  and get  $S$  and  $G$
- compare  $i$  and  $k$ 
  - if  $i = k$ , then return  $A[k]$
  - if  $i < k$ , then  $\bar{A}[i]$  is in  $S$
  - if  $i > k$ , then  $\bar{A}[i]$  is in  $G$

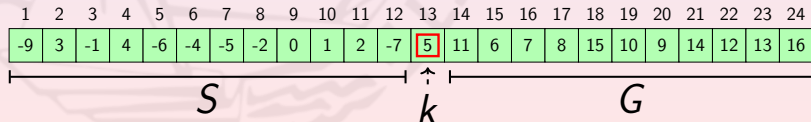


# A Possible Strategy

What about using PARTITION and a “dichotomic approach”?

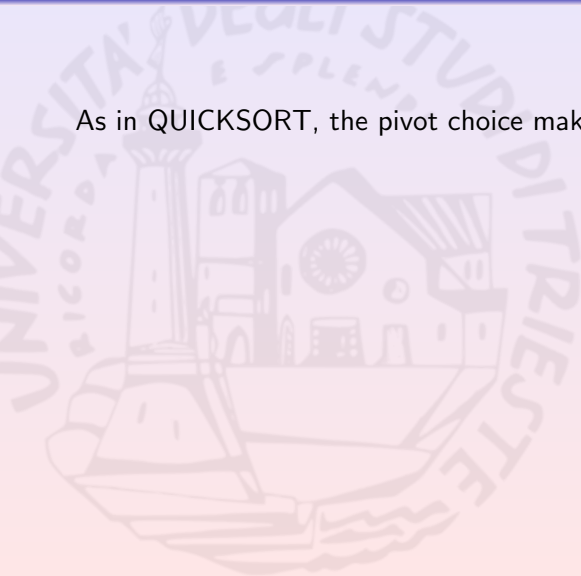
- select a pivot  $A[j]$
- compute  $k \leftarrow \text{PARTITION}(A, 1, |A|, j)$  and get  $S$  and  $G$
- compare  $i$  and  $k$ 
  - if  $i = k$ , then return  $A[k]$
  - if  $i < k$ , then  $\bar{A}[i]$  is in  $S$
  - if  $i > k$ , then  $\bar{A}[i]$  is in  $G$

A recursive algorithm can solve the problem!



# Recursive Partition Approach: Issues and Opportunities

As in QUICKSORT, the pivot choice makes the difference



# Recursive Partition Approach: Issues and Opportunities

As in QUICKSORT, the pivot choice makes the difference

When the partition is not balanced enough, we perform  $\Theta(n^2)$  ops

# Recursive Partition Approach: Issues and Opportunities

As in QUICKSORT, the pivot choice makes the difference

When the partition is not balanced enough, we perform  $\Theta(n^2)$  ops

The best pivot choice would be  $\bar{A}$ 's median, but  $A$  may be unsorted

# Recursive Partition Approach: Issues and Opportunities

As in QUICKSORT, the pivot choice makes the difference

When the partition is not balanced enough, we perform  $\Theta(n^2)$  ops

The best pivot choice would be  $\bar{A}$ 's median, but  $A$  may be unsorted

However, constant ratio partitions are QUICKSORT's best case scenarios as well

# Recursive Partition Approach: Issues and Opportunities

As in QUICKSORT, the pivot choice makes the difference

When the partition is not balanced enough, we perform  $\Theta(n^2)$  ops

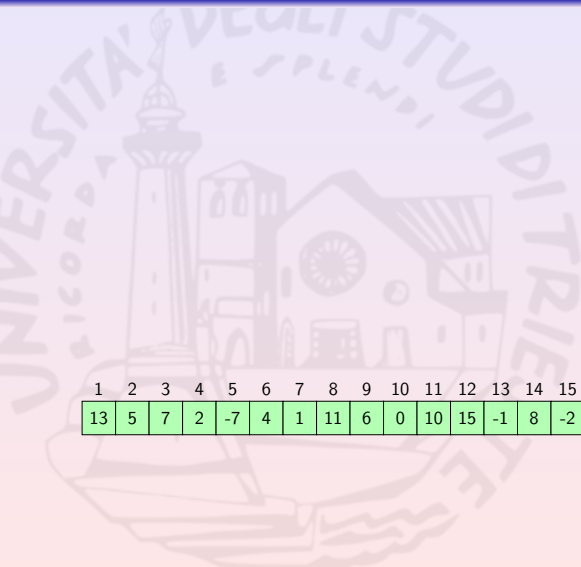
The best pivot choice would be  $\bar{A}$ 's median, but  $A$  may be unsorted

However, constant ratio partitions are QUICKSORT's best case scenarios as well

Is there a smart way to guess an **almost-median** value for  $\bar{A}$ ?



# Choosing the Pivot



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	5	7	2	-7	4	1	11	6	0	10	15	-1	8	-2	3	14	12	-9	-5	9	-6	16	-4

# Choosing the Pivot

- split  $A$  in  $\lceil n/5 \rceil$  chunks  $C_1, \dots, C_{n/5}$  each of size 5

C <sub>1</sub>					C <sub>2</sub>					C <sub>3</sub>					C <sub>4</sub>					C <sub>5</sub>			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	5	7	2	-7	4	1	11	6	0	10	15	-1	8	-2	3	14	12	-9	-5	9	-6	16	-4

# Choosing the Pivot

- split  $A$  in  $\lceil n/5 \rceil$  chunks  $C_1, \dots, C_{n/5}$  each of size 5
- find the median  $m_i$  of  $C_i$

C <sub>1</sub>					C <sub>2</sub>					C <sub>3</sub>					C <sub>4</sub>					C <sub>5</sub>			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	5	7	2	-7	4	1	11	6	0	10	15	-1	8	-2	3	14	12	-9	-5	9	-6	16	-4

# Choosing the Pivot

- split  $A$  in  $\lceil n/5 \rceil$  chunks  $C_1, \dots, C_{n/5}$  each of size 5
- find the median  $m_i$  of  $C_i$ , e.g., by sorting  $C_i$  itself

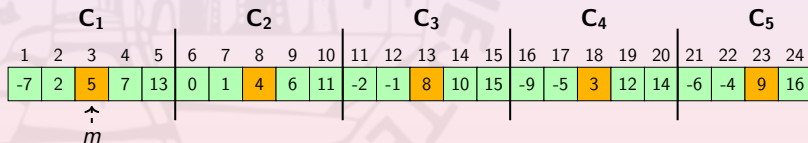
sorting  $n/5$  chunks of size 5, I have a complexity which isn't  $O(5 \log 5) = O(n)$ : li

C <sub>1</sub>					C <sub>2</sub>					C <sub>3</sub>					C <sub>4</sub>					C <sub>5</sub>			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
-7	2	5	7	13	0	1	4	6	11	-2	-1	8	10	15	-9	-5	3	12	14	-6	-4	9	16

# Choosing the Pivot

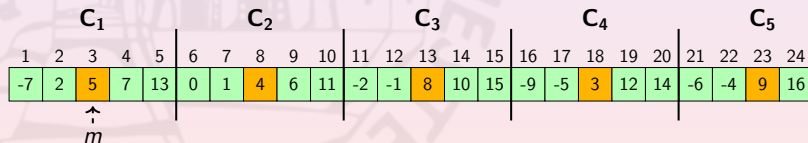
- split  $A$  in  $\lceil n/5 \rceil$  chunks  $C_1, \dots, C_{n/5}$  each of size 5
- find the median  $m_i$  of  $C_i$ , e.g., by sorting  $C_i$  itself

C <sub>1</sub>					C <sub>2</sub>					C <sub>3</sub>					C <sub>4</sub>					C <sub>5</sub>			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
-7	2	5	7	13	0	1	4	6	11	-2	-1	8	10	15	-9	-5	3	12	14	-6	-4	9	16



# Does the Selected Pivot Partition A Evenly Enough?

Think the chunks as they were the columns of a matrix



# Does the Selected Pivot Partition A Evenly Enough?

Think the chunks as they were the columns of a matrix

C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>
-7	0	-2	-9	-6
2	1	-1	-5	-4
5	4	8	3	9
7	6	10	12	16
13	11	15	14	



# Does the Selected Pivot Partition A Evenly Enough?

Sort the chunks according the medians

C <sub>4</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>3</sub>	C <sub>5</sub>
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Does the Selected Pivot Partition A Evenly Enough?

How many chunks are there?

$$\left\lceil \frac{n}{5} \right\rceil$$

C <sub>4</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>3</sub>	C <sub>5</sub>
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Does the Selected Pivot Partition A Evenly Enough?

How many  $m_i$  are greater or equal to  $m$ ?

$$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil$$

$C_4$	$C_2$	$C_1$	$C_3$	$C_5$
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Does the Selected Pivot Partition *A* Evenly Enough?

How many chunks at least have 3 elements greater than *m*?

$$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2$$

not the pivot chunk (median): it has only 2 elements greater than its pivotNor the last

C <sub>4</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>3</sub>	C <sub>5</sub>
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Does the Selected Pivot Partition A Evenly Enough?

How many elements at least are greater than  $m$ ?

$$3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right)$$

C <sub>4</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>3</sub>	C <sub>5</sub>
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Does the Selected Pivot Partition A Evenly Enough?

How many elements at least are greater than  $m$ ?

$$3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$

C <sub>4</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>3</sub>	C <sub>5</sub>
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Does the Selected Pivot Partition *A* Evenly Enough?

An upper bound for the # of elements smaller or equal to *m* is

$$n - \left( \frac{3n}{10} - 6 \right) = \frac{7n}{10} + 6$$

C <sub>4</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>3</sub>	C <sub>5</sub>
-9	0	-4	-2	-6
-5	1	2	-1	-4
3	4	5	8	9
12	6	7	10	16
14	11	13	15	

# Complexity of the Select Algorithm

cost to choose a proper pivot and then do partition to solve the problem

$$T_S(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_S(\lceil n/5 \rceil) + T_S(7n/10 + 6) + \Theta(n) & \text{otherwise} \end{cases}$$

to sort the ch

another recursive

cost of partition



# Complexity of the Select Algorithm

$$T_S(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_S(\lceil n/5 \rceil) + T_S(7n/10 + 6) + \Theta(n) & \text{otherwise} \end{cases}$$

by induction

Let us assume that  $T_S(m) \leq cm \in O(m)$  for  $m < n$

$$\begin{aligned} T_S(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + c'n \\ &\leq c(n/5 + 1) + c(7n/10 + 6) + c'n \\ &\leq \frac{9}{10}cn + c'n + 7c \end{aligned}$$

# Complexity of the Select Algorithm

$$T_S(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_S(\lceil n/5 \rceil) + T_S(7n/10 + 6) + \Theta(n) & \text{otherwise} \end{cases}$$

Let us assume that  $T_S(m) \leq cm \in O(m)$  for  $m < n$

$$\begin{aligned} T_S(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + c'n \\ &\leq c(n/5 + 1) + c(7n/10 + 6) + c'n \\ &\leq \frac{9}{10}cn + c'n + 7c \end{aligned}$$

Hence,  $T_S(n) \leq cn$  for  $c \geq 20c'$  and  $n \geq 140$  and

# Complexity of the Select Algorithm

$$T_S(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T_S(\lceil n/5 \rceil) + T_S(7n/10 + 6) + \Theta(n) & \text{otherwise} \end{cases}$$

Let us assume that  $T_S(m) \leq cm \in O(m)$  for  $m < n$

$$\begin{aligned} T_S(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + c'n \\ &\leq c(n/5 + 1) + c(7n/10 + 6) + c'n \\ &\leq \frac{9}{10}cn + c'n + 7c \end{aligned}$$

Hence,  $T_S(n) \leq cn$  for  $c \geq 20c'$  and  $n \geq 140$  and

$$T_S(n) \in \Theta(n)$$

## Select Algorithm: Pseudo-Code

```
def SELECT(A, i, l=1, r=|A|):  
    j ← SELECT_PIVOT(A, l, r)  
    k ← PARTITION(A, l, r, j)  
  
    if i=k: # dichotomic approach  
        return k  
    endif  
  
    if i<k: # search in S  
        return SELECT(A, i, l, k-1)  
    endif  
  
    # search in G  
    return SELECT(A, i, k+1, r) + k  
enddef
```