

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 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.

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 $A[i].id = id_1$

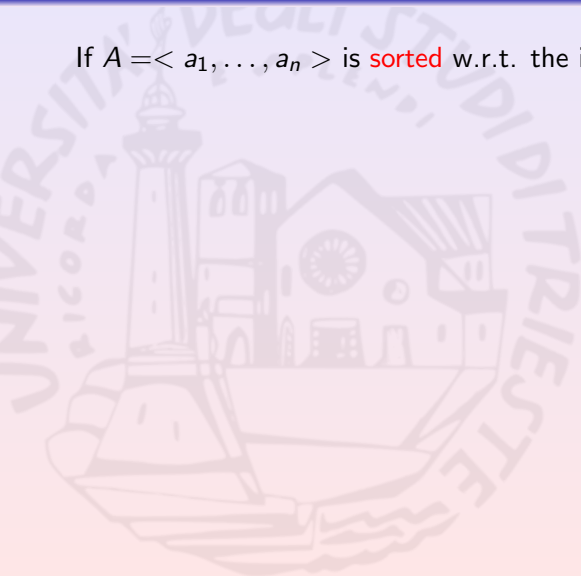
What is the asymptotic complexity in terms of big- O ? $O(n)$

Can we do better?

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

A Better Technique: Dichotomic Search

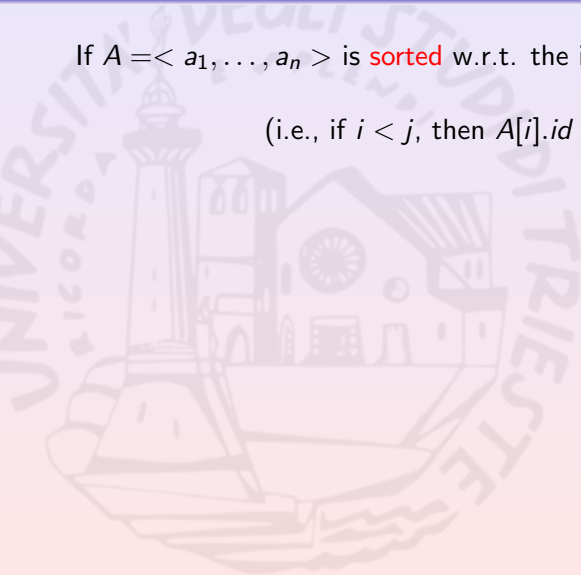
If $A = \langle a_1, \dots, a_n \rangle$ is **sorted** w.r.t. the id's...



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

Dichotomic Search: An Example

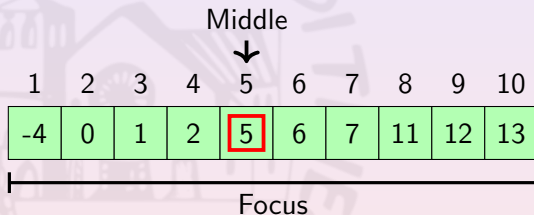
Search for 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.

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

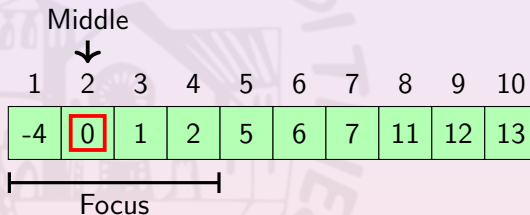
Search for 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.

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

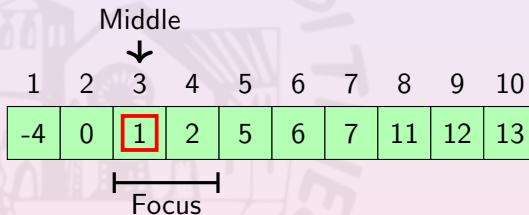
Search for 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.

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

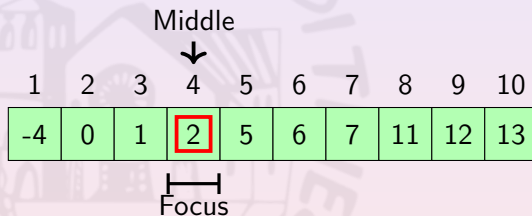
Search for 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.

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

$O(\log n)$

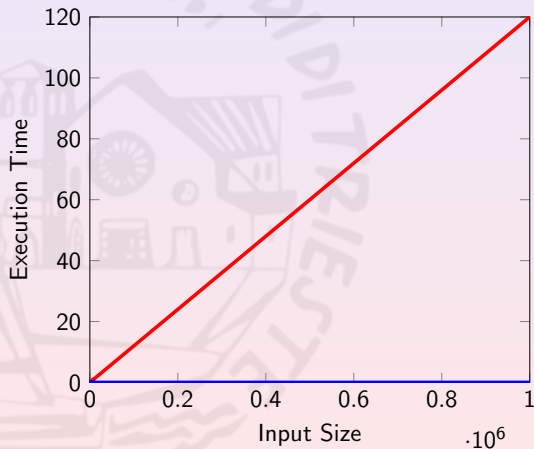
Dichotomic Search vs Linear Search: Experiments

Execution time per 1×10^5 random searches.

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

Dichotomic Search vs Linear Search: Experiments

Execution time per 1×10^5 random searches.



Retrieving Data
○○○○○○○

Sorting
●○

Insertion Sort
○○○

Quick Sort
○○○○○○○○○○○

Bubble Sort and Heap Sort
○○○

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

⇓

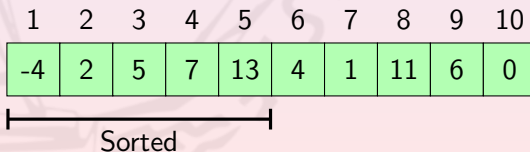
1	2	3	4	5	6	7	8	9	10
-4	0	1	2	4	5	6	7	11	13

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

Insertion Sort

Insertion Sort: Intuition

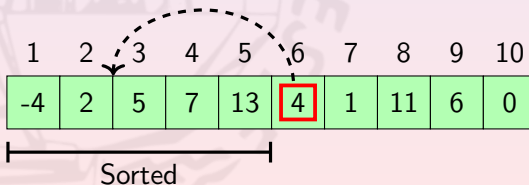
If the first fragment of the array is already sorted



Insertion Sort: Intuition

If the first fragment of the array is already sorted

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

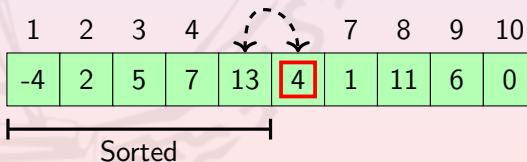


Insertion Sort: Intuition

If the first fragment of the array is already sorted

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

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

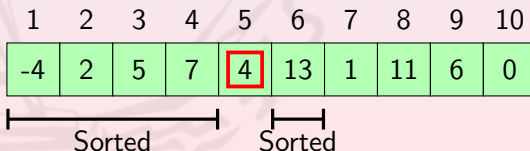


Insertion Sort: Intuition

If the first fragment of the array is already sorted

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

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

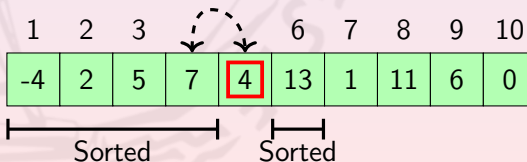


Insertion Sort: Intuition

If the first fragment of the array is already sorted

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

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

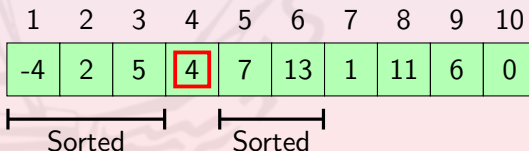


Insertion Sort: Intuition

If the first fragment of the array is already sorted

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

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

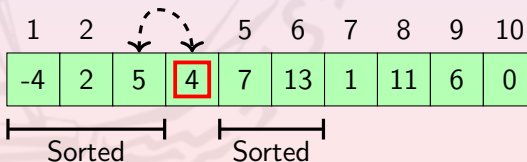


Insertion Sort: Intuition

If the first fragment of the array is already sorted

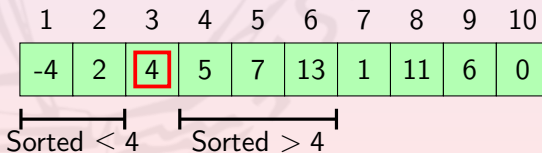
we can “*enlarge*” it by inserting **next element**

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



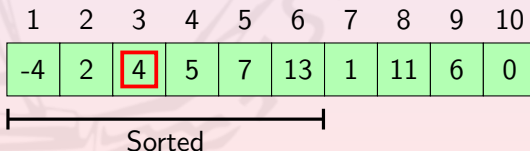
Insertion Sort: Intuition

If the first fragment of the array is already sorted
we can “enlarge” it by inserting **next element**
by switching **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 switching **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]):  
            switch(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
●○○○○○○○○○

Bubble Sort and Heap Sort
○○○

Quick Sort

Quick Sort: Intuition

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



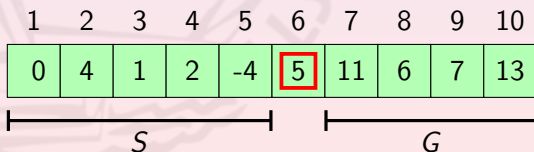
1	2	3	4	5	6	7	8	9	10
13	5	7	2	-4	4	1	11	6	0

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



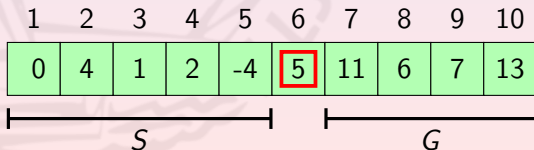
Quick Sort: Intuition

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

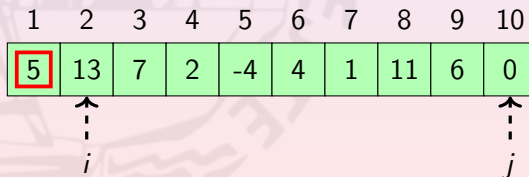
Switch the pivot **p** and the first element in *A*



Partition: An In-place Algorithm

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

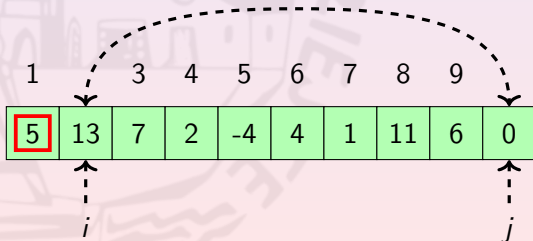
If **$A[i] > p$** ,



Partition: An In-place Algorithm

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

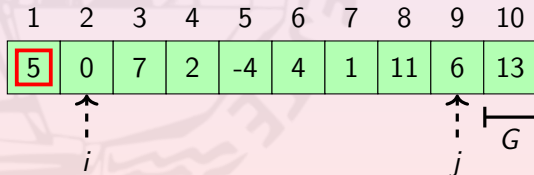
If **$A[i] > p$** , switch $A[i]$ and $A[j]$ and decrease j



Partition: An In-place Algorithm

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

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

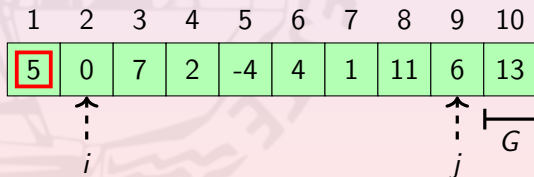


Partition: An In-place Algorithm

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

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

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

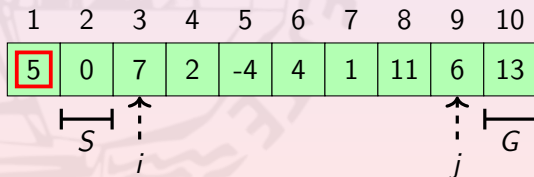


Partition: An In-place Algorithm

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

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

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



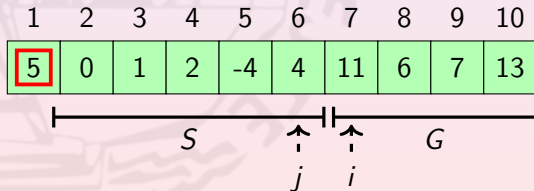
Partition: An In-place Algorithm

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

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

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

Repeat until $i \leq j$



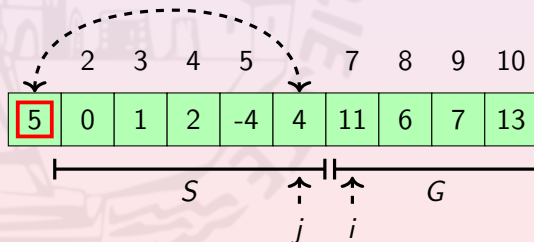
Partition: An In-place Algorithm

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

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

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

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



Partition: An In-place Algorithm

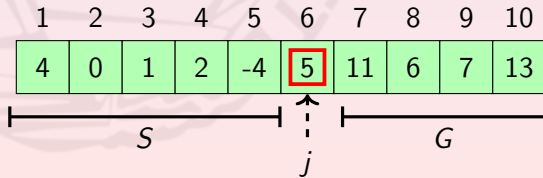
Switch the pivot **p** and the first element in A

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

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

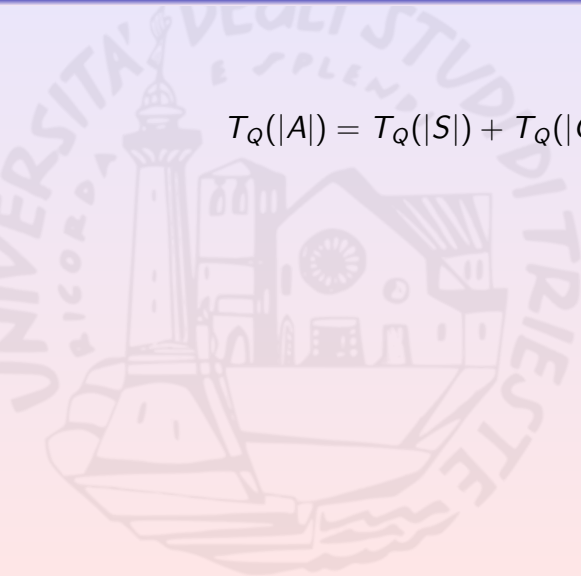
Repeat until $i \leq j$ and switch **p** and $A[j]$

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



Quick Sort Complexity: Worst Case

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



Quick Sort Complexity: Worst Case

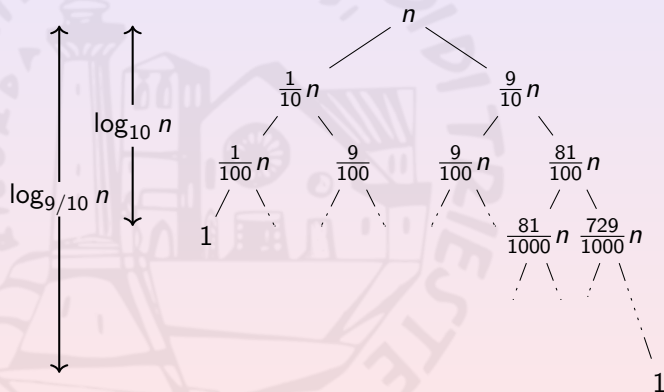
$$T_Q(|A|) = T_Q(|S|) + T_Q(|G|) + \Theta(|A|)$$

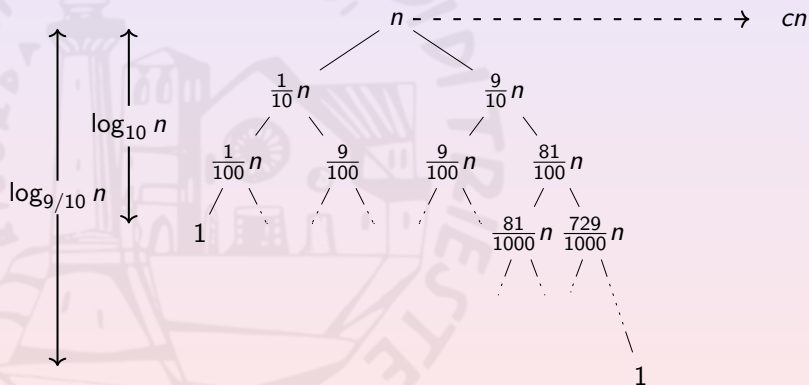
Worst Case: $|G| = 0$ or $|S| = 0$ for all recursive call.

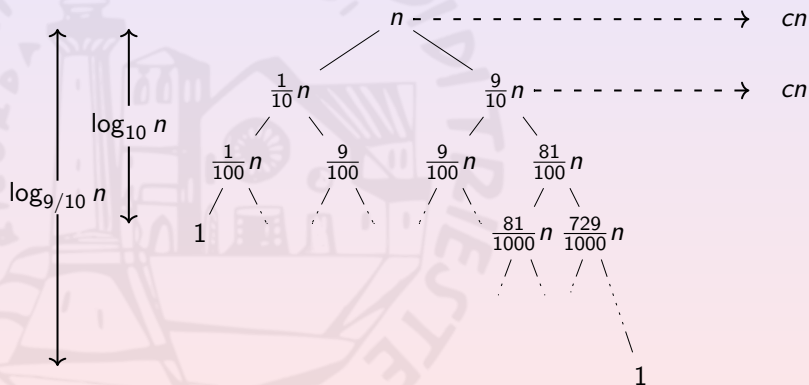
$$\begin{aligned} T_Q(n) &= T_Q(n-1) + \Theta(n) \\ &= \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{i=0}^n i\right) \\ &= \Theta(n^2) \end{aligned}$$

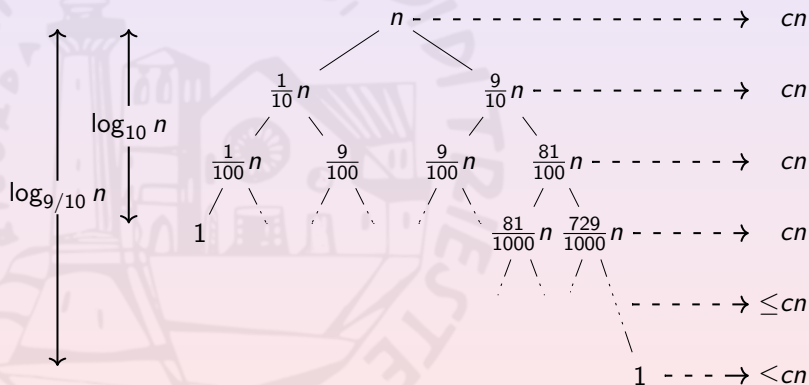
Quick Sort Complexity: Best Case

Best Case: Balanced Partition



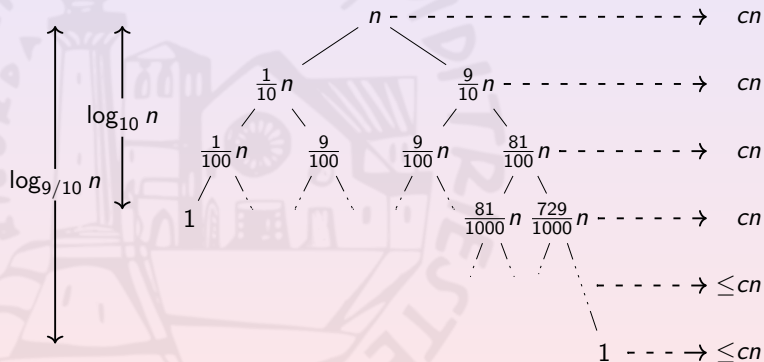






Quick Sort Complexity: Best Case

Best Case: Balanced Partition



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

Quick Sort Complexity: Average Case

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

If all the permutations of A are equally likely,

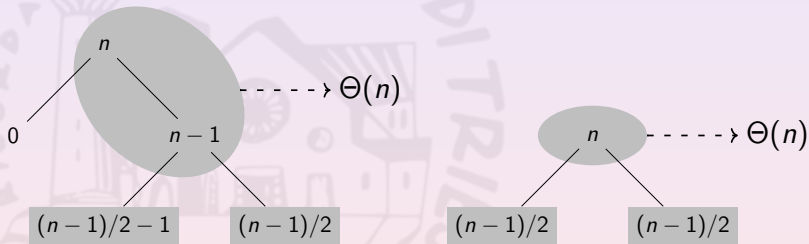
the partition has a ratio more balanced than $1/d$ with probability

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

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

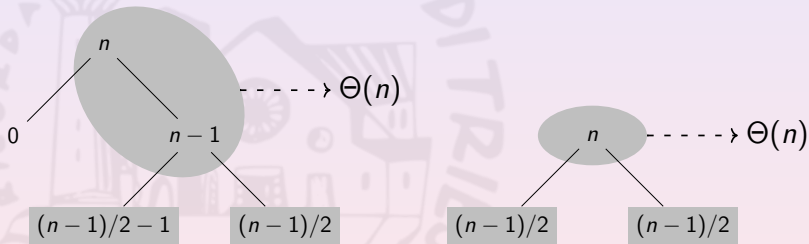
Quick Sort Complexity: Average Case (Cont'd)

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, light-colored watermark of the University of Trieste logo. The logo is circular and contains the text "UNIVERSITA' DEGLI STUDI DI TRIESTE" around the perimeter. In the center, there is an illustration of a building with a dome and a tower, with the words "E SPLENDI" visible below the main text.

Bubble Sort and 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

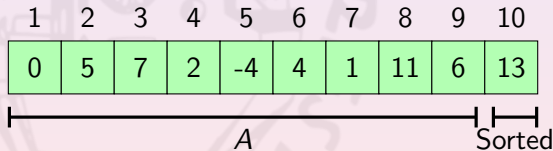
1	2	3	4	5	6	7	8	9	10
0	5	7	2	-4	4	1	11	6	13

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

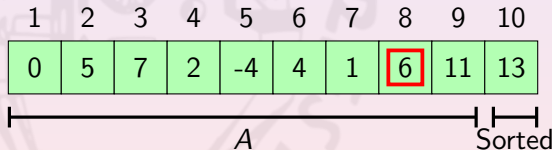


Sorting by Searching the Maximum

Find the maximum

Move the maximum at the end of the array

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

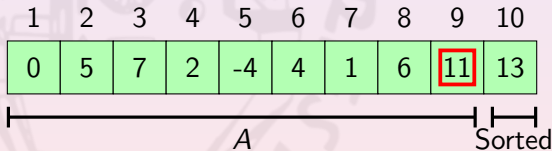


Sorting by Searching the Maximum

Find the maximum

Move the maximum at the end of the array

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

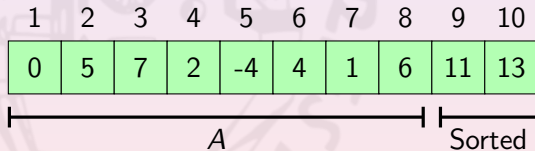


Sorting by Searching the Maximum

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

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

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

How to Find the Maximum?

By using ...

- a linear scanning

⇒ BubbleSort

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

- ???