Retrieving Data and Sorting

Advanced Programming and Algorithmic Design

Alberto Casagrande
Email: acasagrande@units.it

a.a. 2018/2019

Retrieving Data

Retrieving Data

$A = < a_1, \ldots, a_n >$ 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 $A[i].id = \mathrm{id}_1$

A Naïve Solution and Outlook

Scan all the database searching for $A[i].id = \mathrm{id}_1$

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

A Naïve Solution and Outlook

Scan all the database searching for $A[i].id = \mathrm{id}_1$

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 $A =< a_1, \ldots, a_n >$ is sorted w.r.t. the id's...

A Better Technique: Dichotomic Search

If $A =< a_1, \ldots, a_n >$ 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 $A =< a_1, \ldots, a_n >$ is sorted w.r.t. the id's...

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

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

A Better Technique: Dichotomic Search

If $A =< a_1, \ldots, a_n >$ 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 $A[n/2].id = \mathrm{id}_1$

Done!

A Better Technique: Dichotomic Search

If $A =< a_1, \ldots, a_n >$ is sorted w.r.t. the id's...

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

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

if $A[n/2].id = \mathrm{id}_1$

Done!

if $A[n/2].id > \mathrm{id}_1$

Focus on the 1st half A, i.e, $< a_1, \ldots, a_{n/2-1} >$

A Better Technique: Dichotomic Search

If $A =< a_1, \ldots, a_n >$ is sorted w.r.t. the id's...

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

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

if $A[n/2].id = \mathrm{id}_1$

Done!

if $A[n/2].id > \mathrm{id}_1$

Focus on the 1st half A, i.e, $< a_1, \ldots, a_{n/2-1} >$

if $A[n/2].id < \mathrm{id}_1$

Focus on the 2nd half A, i.e, $< a_{n/2+1}, \ldots, a_n >$

Repeat until A is not empty

Dichotomic Search: An Example

Search for 2 in $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

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

Dichotomic Search: An Example

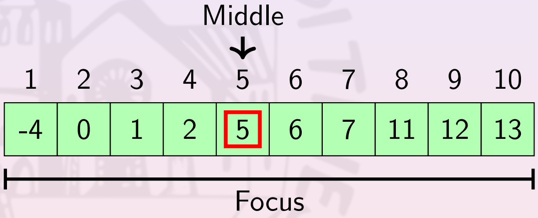
Search for 2 in $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

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 $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.



Dichotomic Search: An Example

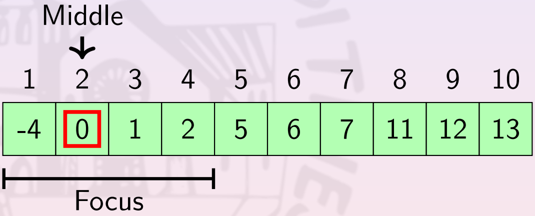
Search for 2 in $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

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 $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.



Dichotomic Search: An Example

Search for 2 in $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

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 $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

Middle

1	2	3	4	5	6	7	8	9	10
-4	0	$\boxed{1}$	2	5	6	7	11	12	13

Focus

Dichotomic Search: An Example

Search for 2 in $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

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 $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$.

Middle

1	2	3	4	5	6	7	8	9	10
-4	0	1	$\boxed{2}$	5	6	7	11	12	13

Focus

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

$$O(\log n)$$

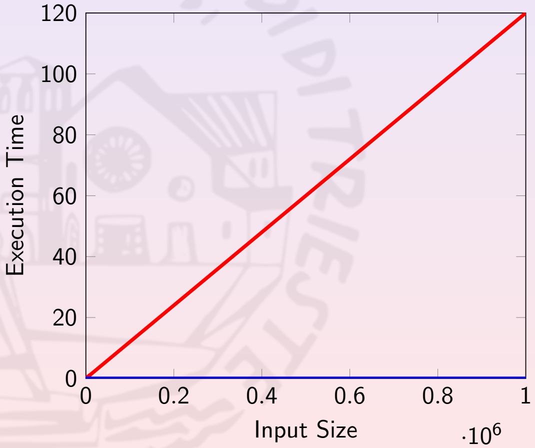
Dichotomic Search vs Linear Search: Experiments

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.



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

$$\Downarrow$$

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

$\Downarrow$

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

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

Sorted

Insertion Sort: Intuition

If the first fragment of the array is already sorted

we can "*enlarge*" it by inserting **next element**

1	2	3	4	5	6	7	8	9	10
-4	2	5	7	13	$\boxed{4}$	1	11	6	0

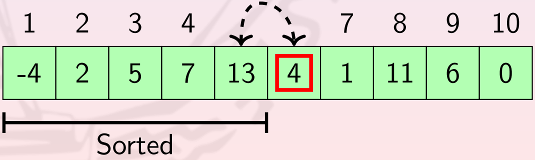
Sorted

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

1	2	3	4	5	6	7	8	9	10
-4	2	5	7	$\boxed{4}$	13	1	11	6	0

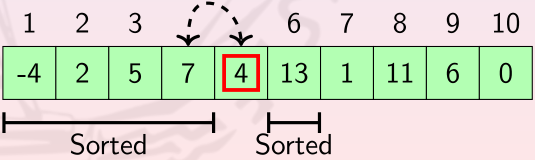
Sorted Sorted

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

1	2	3	4	5	6	7	8	9	10
-4	2	5	$\boxed{4}$	7	13	1	11	6	0

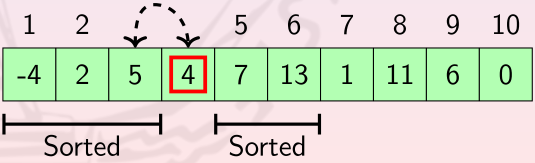
Sorted (positions 1–3) Sorted (positions 5–6)

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**

1	2	3	4	5	6	7	8	9	10
-4	2	$\boxed{4}$	5	7	13	1	11	6	0

Sorted ≤ 4 Sorted > 4

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**

1	2	3	4	5	6	7	8	9	10
-4	2	$\boxed{4}$	5	7	13	1	11	6	0

Sorted

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 $i \in [2, n]$

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

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

Quick Sort

Quick Sort: Intuition

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

1	2	3	4	5	6	7	8	9	10
13	$\boxed{5}$	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 then the pivot

1	2	3	4	5	6	7	8	9	10
0	4	1	2	-4	$\boxed{5}$	11	6	7	13
		S					G		

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 then the pivot

Repeat on the subarrays having more than 1 elements

1	2	3	4	5	6	7	8	9	10
0	4	1	2	-4	$\boxed{5}$	11	6	7	13

$$\underbrace{0\ \ 4\ \ 1\ \ 2\ \ {-4}}_{S}\quad 5\quad \underbrace{11\ \ 6\ \ 7\ \ 13}_{G}$$

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 then 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 then 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 then 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

		3	4	5	6	7	8	9	10
13	$\boxed{5}$	7	2	-4	4	1	11	6	0

Partition: An In-place Algorithm

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

If $\mathbf{A[i]} > \mathbf{p}$,

1	2	3	4	5	6	7	8	9	10
$\boxed{5}$	13	7	2	-4	4	1	11	6	0
	i								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

1		3	4	5	6	7	8	9	
$\boxed{5}$	13	7	2	-4	4	1	11	6	0
	i								j

Partition: An In-place Algorithm

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

If $\mathbf{A[i]} > \mathbf{p}$, swap $A[i]$ and $A[j]$ and decrease j

1	2	3	4	5	6	7	8	9	10
$\boxed{5}$	0	7	2	-4	4	1	11	6	13
	i							j	G

Partition: An In-place Algorithm

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

If $\mathbf{A[i]} > \mathbf{p}$, swap $A[i]$ and $A[j]$ and decrease j

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

1	2	3	4	5	6	7	8	9	10
$\boxed{5}$	0	7	2	-4	4	1	11	6	13
	i							j	G

Partition: An In-place Algorithm

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

If $\mathbf{A[i]} > \mathbf{p}$, swap $A[i]$ and $A[j]$ and decrease j

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

1	2	3	4	5	6	7	8	9	10
$\boxed{5}$	0	7	2	-4	4	1	11	6	13
	S	i						j	G

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 $i \leq j$

1	2	3	4	5	6	7	8	9	10
$\boxed{5}$	0	1	2	-4	4	11	6	7	13

S G

j i

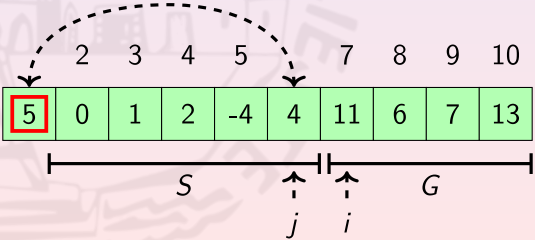
Partition: An In-place Algorithm

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



Partition: An In-place Algorithm

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

If $\mathbf{A[i]} > \mathbf{p}$, swap $A[i]$ and $A[j]$ and decrease j

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

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

The complexity is $\Theta(|\mathbf{A}|)$

1	2	3	4	5	6	7	8	9	10
4	0	1	2	-4	$\boxed{5}$	11	6	7	13
		S			j		G		

Quick Sort Complexity: Worst Case

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

Quick Sort Complexity: Worst Case

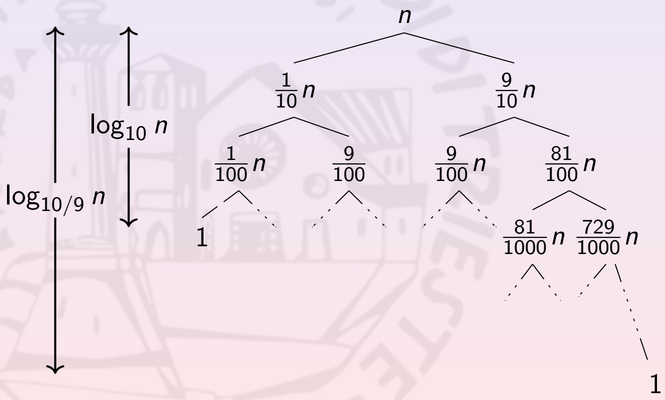
$$T_Q(|A|) = T_Q(|S|) + T_Q(|G|) + \Theta(|A|)$$

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

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

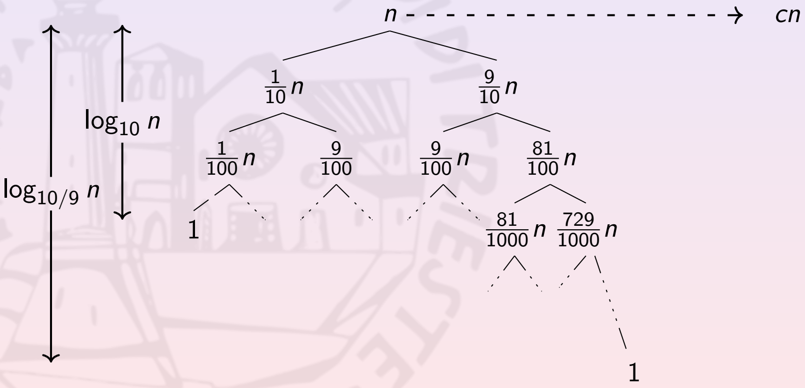
Quick Sort Complexity: Best Case

Best Case: Balanced Partition



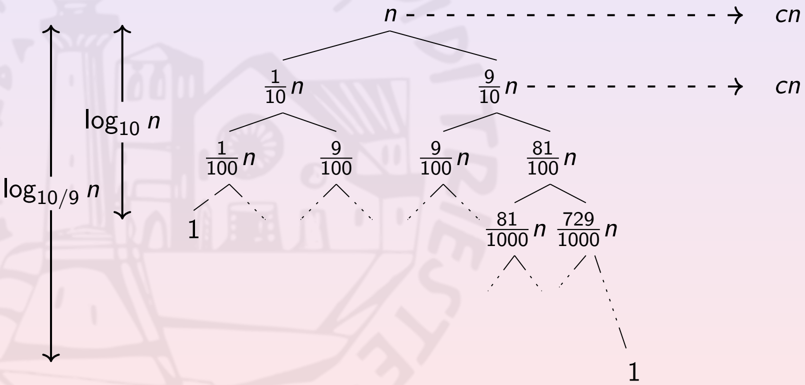
Quick Sort Complexity: Best Case

Best Case: Balanced Partition



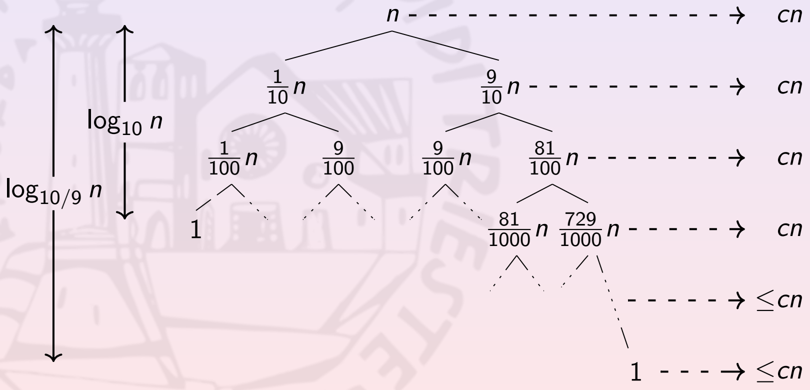
Quick Sort Complexity: Best Case

Best Case: Balanced Partition



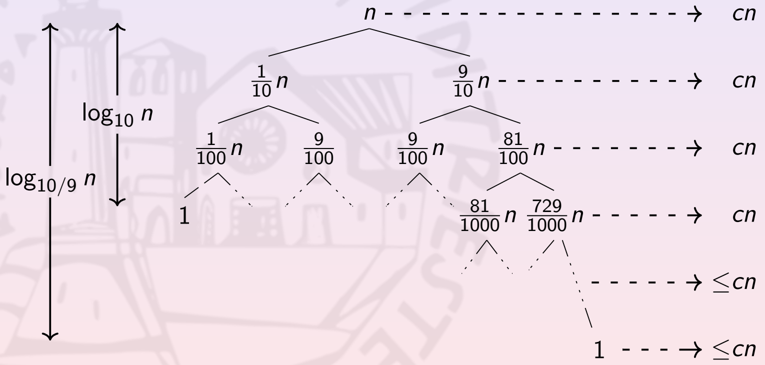
Quick Sort Complexity: Best Case

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,

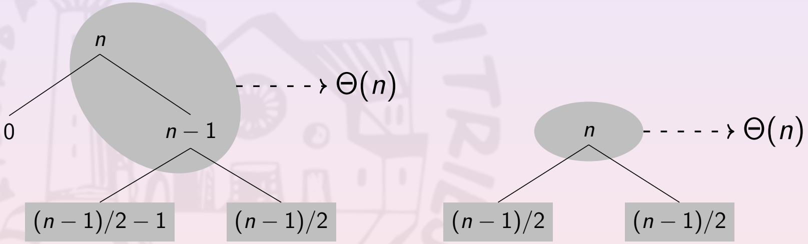
the partition has a ratio more balanced than $1/d$ with probability

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

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

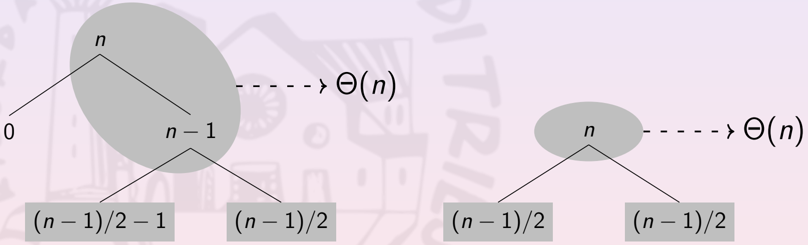
Quick Sort Complexity: Average Case (Cont'd)

Even if "good" and "bad" cases alternate



Quick Sort Complexity: Average Case (Cont'd)

Even if "good" and "bad" cases alternate



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

Heap Sort

Sorting by Searching the Maximum

Find the maximum

1	2	3	4	5	6	7	8	9	10
$\boxed{13}$	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

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

A (positions 1–9) Sorted (position 10)

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	$\boxed{11}$	6	13

A (positions 1–9) | Sorted (position 10)

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	$\boxed{11}$	13

$\underbrace{\text{positions } 1\text{–}9}_{A}$ $\underbrace{\text{position } 10}_{\text{Sorted}}$

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

$\underbrace{\text{positions } 1\text{–}8}_{A}$ $\underbrace{\text{positions } 9\text{–}10}_{\text{Sorted}}$

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

A Sorted

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

How to Find the Maximum?

By using ...

- pushing the max to the right $\Longrightarrow$ Bubble Sort

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

- binary heap (see here) $\Longrightarrow$ Heap Sort

Heap Sort: Pseudo-Code

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

```
def HEAPSORT(A):
    BUILD_HEAP(A)

    for i ← |A| downto 2
        swap(H,1,i)

        H.size ← H.size − 1
        HEAPIFY(H,1)
    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 $O(n \log n)$

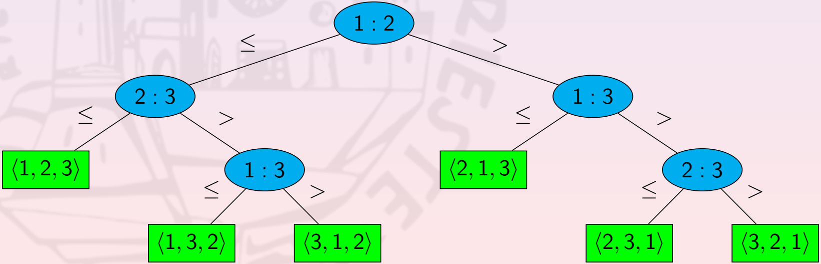
Sorting By Comparison: Lower Bound

you can demonstrate that any algorithm sorting by comparison needs at least (n log n)

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 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 $\Omega(n \log n)$

so there's no general algorithm using comparisono which can sort in linear time. How can we