

Fundations

Advanced Programming and Algorithmic Design

Alberto Casagrande

Email: `acasagrande@units.it`

a.y. 2019/2020

The background of the slide features a large, faint watermark of the University of Trieste logo. The logo is circular, with the text "UNIVERSITA' DEGLI STUDI DI TRIESTE" around the top and "E SPLENDI" in the center. Below the text is a detailed illustration of a building with a dome and a tower, set against a background of stylized waves.

Algorithms and Computational Models

What is an Algorithm?

Definition (Algorithm)

Is a sequence of well-defined steps that transforms a set of inputs into a set of outputs in a finite amount of time

What is an Algorithm?

Definition (Algorithm)

Is a sequence of well-defined steps that transforms a set of inputs into a set of outputs in a finite amount of time

Definition (Computational Model)

Is a mathematical tool to perform computations.

What is an Algorithm?

Definition (Algorithm)

Is a sequence of well-defined steps that transforms a set of inputs into a set of outputs in a finite amount of time

Definition (Computational Model)

Is a mathematical tool to perform computations.

A function described by an algorithm is **calculable**.

A function *implementable* in a computational model is **computable**.

Functions, Computability and Calcolability

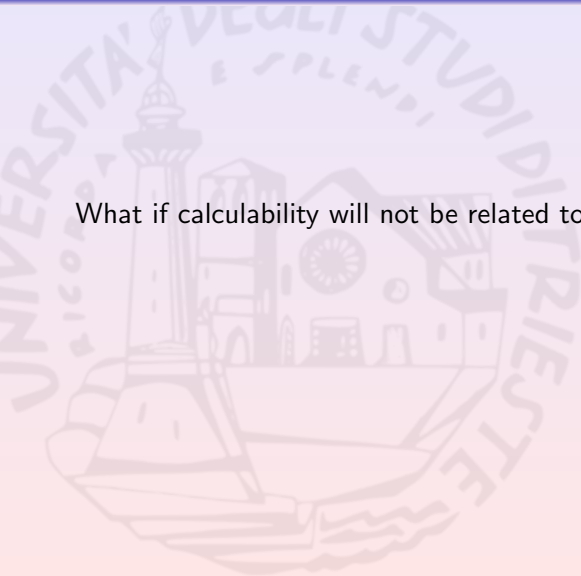
Are all the functions computable in any specific model?

If this is not the case

- are there calculable functions that are not computable?
- are there computable functions that are not calculable?

Why Is This Relevant For Us?

What if calculability will not be related to computability?



Why Is This Relevant For Us?

What if calculability will not be related to computability?

Algorithms would not guarantee implementability!

Halting Problem

Let h be the function that establish whether any program p eventually ends its execution (\downarrow) on an input i or runs forever(\uparrow)

$$h(p, i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } p(i) \text{ never ends} \\ 1 & \text{otherwise} \end{cases}$$

Definition (Halting problem)

Can we implement h ?

Computability of Halting Problem

For any computable function $f(a, b)$, define

$$g_f(i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f(i, i) = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

Since f is computable, so it is g_f . Let G_f implement it.

Can h be one of the f 's?

Computability of Halting Problem

For any computable function $f(a, b)$, define

$$g_f(i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f(i, i) = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

Since f is computable, so it is g_f . Let G_f implement it.

Can h be one of the f 's?

- If $f(G_f, G_f) = 0 \implies g_f(G_f) = 0$ and $h(G_f, G_f) = 1$
- If $f(G_f, G_f) \neq 0 \implies g_f(G_f) \uparrow$ and $h(G_f, G_f) = 0$

Computability of Halting Problem

For any computable function $f(a, b)$, define

$$g_f(i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f(i, i) = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

Since f is computable, so it is g_f . Let G_f implement it.

Can h be one of the f 's?

- If $f(G_f, G_f) = 0 \implies g_f(G_f) = 0$ and $h(G_f, G_f) = 1$
- If $f(G_f, G_f) \neq 0 \implies g_f(G_f) \uparrow$ and $h(G_f, G_f) = 0$

Computability of Halting Problem

For any computable function $f(a, b)$, define

$$g_f(i) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } f(i, i) = 0 \\ \uparrow & \text{otherwise} \end{cases}$$

Since f is computable, so it is g_f . Let G_f implement it.

Can h be one of the f 's?

- If $f(G_f, G_f) = 0 \implies g_f(G_f) = 0$ and $h(G_f, G_f) = 1$
- If $f(G_f, G_f) \neq 0 \implies g_f(G_f) \uparrow$ and $h(G_f, G_f) = 0$

Thus, $h \neq f$ for all computable f 's and h is not computable.

Church-Turing Thesis

Church-Turing Thesis

Every *effectively* calculable function is a computable function.

calculability \implies computability

If we have an algorithm for f , then f can be **formally** computed

Church-Turing Thesis

Church-Turing Thesis

Every *effectively* calculable function is a computable function.

calculability \implies computability

If we have an algorithm for f , then f can be **formally** computed

It also means that:

- all the “reasonable” computational models are equivalent
- we can avoid “hard-to-be-programmed” models (e.g., Turing machine)

Random-Access Machine (RAM)

- variables to store data (no types)
- arrays
- integer and floating point constants
- algebraic functions: $+$, $-$, $/$, $*$, $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$
- assignments
- pointers (no pointer arithmetic)
- conditional and loop statements
- procedure definitions and recursion
- simple “reasonable” functions, e.g., the length of an array

Algorithms are defined as programs on RAM.

A Simple Algorithm

Input: An array A of numbers $\langle a_1, \dots, a_n \rangle$.

Output: The maximum among a_1, \dots, a_n .

```
def find_max(A):  
    max_value  $\leftarrow$  A[1]  
    for i  $\leftarrow$  2.. $|A|$ :  
        if A[i] > max_value:  
            max_value  $\leftarrow$  A[i]  
        endif  
    endfor  
  
    return max_value  
enddef
```

RAM is not Real Hardware!!!

RAM models real hardware, but it lacks

- real HW limitations s.a. finiteness
- memory hierarchy
- instruction execution time

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. In the center, there is an illustration of a building with a dome and a tower, with the words "E SPLENDI" below it.

Time Complexity

How to Measure Algorithm Efficiency?

What about **execution time**?



How to Measure Algorithm Efficiency?

What about **execution time**? (for what input?)

Algorithms are not programs

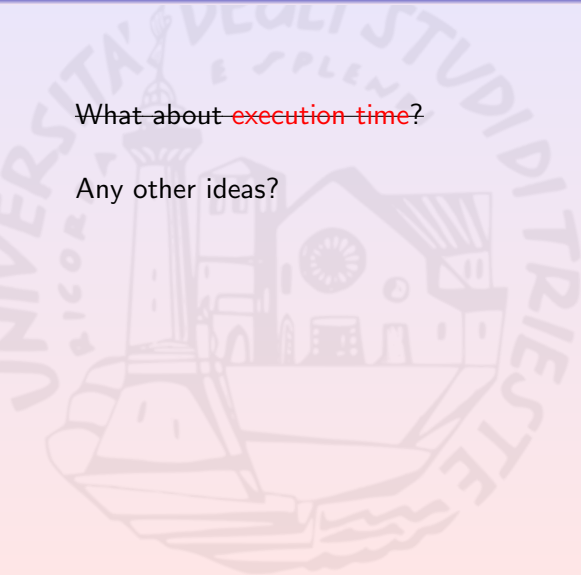
Assuming 1 time unit per instruction are not realistic because execution time depends on:

- CPU instruction sets
- CPU/Memory/Bus Clock
- language and compiler
- OS memory handling
- ...

How to Measure Algorithm Efficiency?

What about ~~execution time~~?

Any other ideas?



How to Measure Algorithm Efficiency?

What about ~~execution time~~?

Any other ideas?

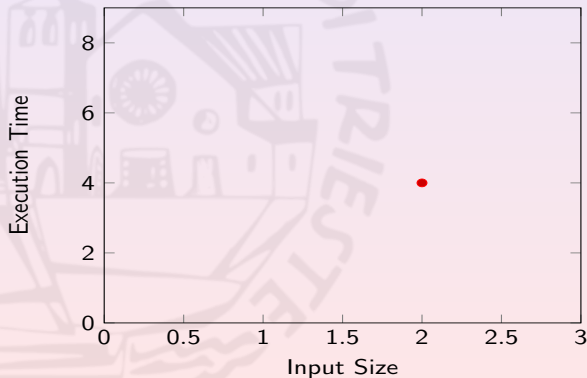
What about **scalability**?

Definition (Scalability)

Effectiveness of a system in handling input growth.

Growth Complexity

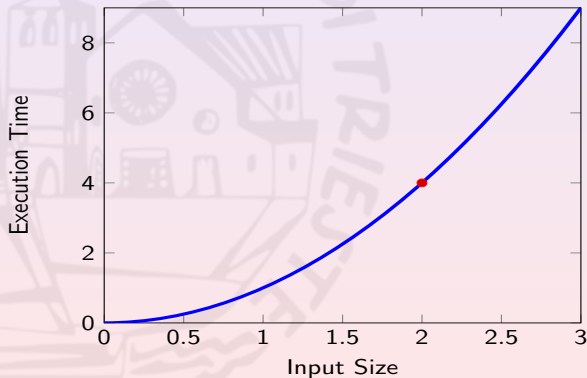
We do **not** measure the execution time **for a given input**



Growth Complexity

We do **not** measure the execution time **for a given input**

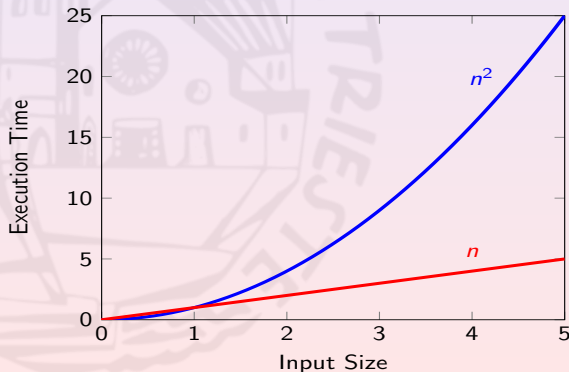
We **estimate** the relation between input size and execution time



Complexity Quiz!

Which growth is preferable between:

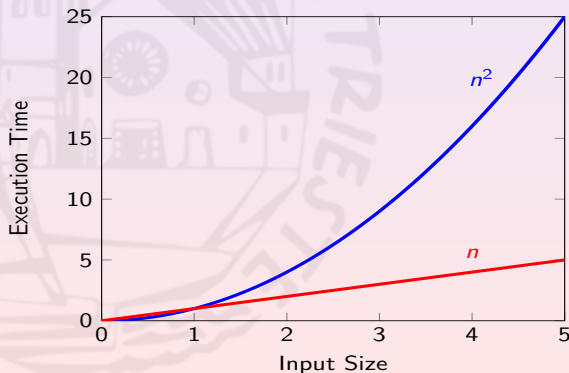
● n^2 and n ?



Complexity Quiz!

Which growth is preferable between:

● n^2 and n ?

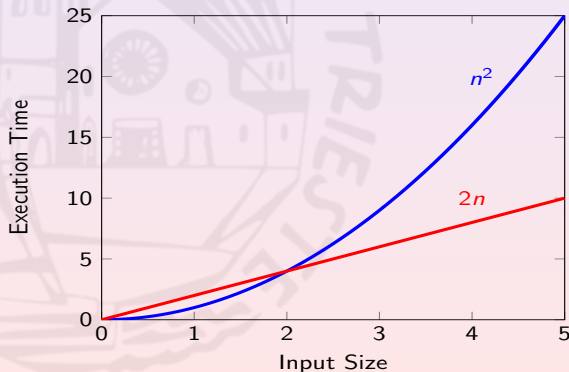


Complexity Quiz!

Which growth is preferable between:

● n^2 and n ?

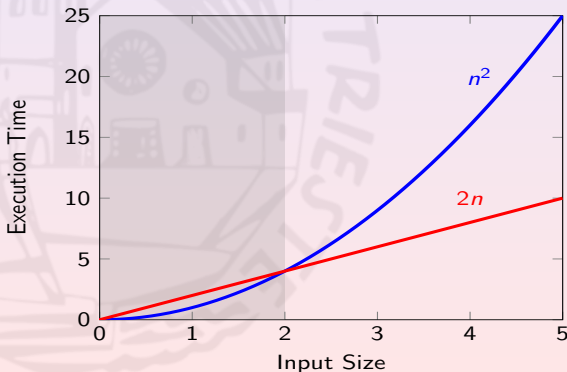
● n^2 and $2 * n$?



Complexity Quiz!

Which growth is preferable between:

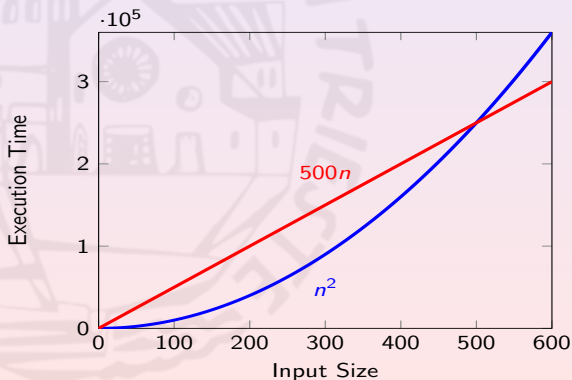
- n^2 and n ?
- n^2 and $2 * n$?



Complexity Quiz!

Which growth is preferable between:

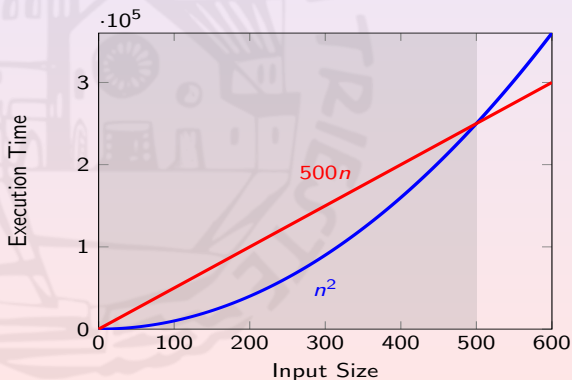
- n^2 and \underline{n} ?
- n^2 and $\underline{2 * n}$?
- n^2 and $500 * n$?



Complexity Quiz!

Which growth is preferable between:

- n^2 and \underline{n} ?
- n^2 and $\underline{2 * n}$?
- n^2 and $\underline{500 * n}$?



Asymptotic Time Complexity

Constants are not useful. We are looking at **asymptotic behaviour**.

We can abstract the single instruction execution time !!!

This intuition is also supported by **linear time speedup theorem**.

Asymptotic Time Complexity

Constants are not useful. We are looking at **asymptotic behaviour**.

We can abstract the single instruction execution time !!!

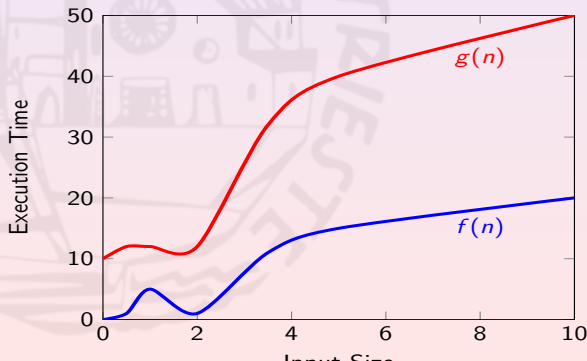
This intuition is also supported by **linear time speedup theorem**.

How to group all the functions that asymptotically are the same?

big O notation

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

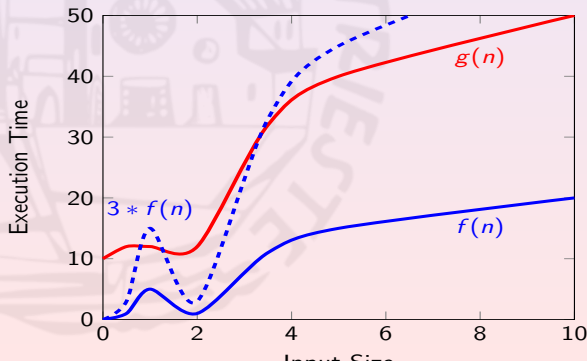
So, $g(n) \in O(f(n))$ iff



big O notation

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \, m \geq n_0 \implies g(m) \leq c * f(m)\}$$

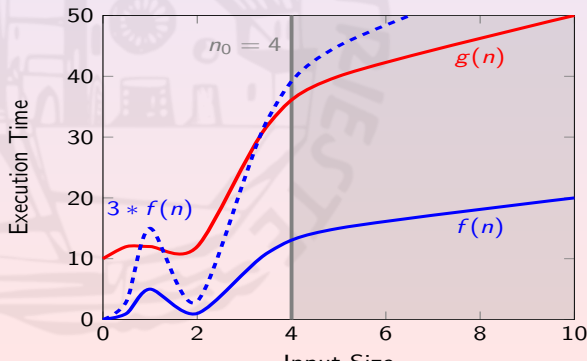
So, $g(n) \in O(f(n))$ iff



big O notation

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \, m \geq n_0 \implies g(m) \leq c * f(m)\}$$

So, $g(n) \in O(f(n))$ iff



Some Useful Properties

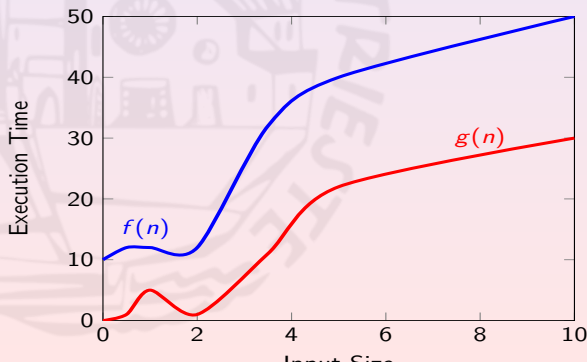
For any $c_1, c_2 \in \mathbb{N}$ and for any $k \in \mathbb{Z}$

- $f(n) \in O(f(n))$
- $O(f(n)) = O(c_1 * f(n) + k)$
- if $c_1 \geq c_2$, then $O(f(n)^{c_1} + k * f(n)^{c_2}) = O(f(n)^{c_1})$
- $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$ es. $n \in O(n^2)$
- if $h(n) \in O(f(n))$ and $h'(n) \in O(g(n))$, then
 - $h(n) + h'(n) \in O(g(n) + f(n))$
 - $h(n) * h'(n) \in O(g(n) * f(n))$

big Ω notation

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

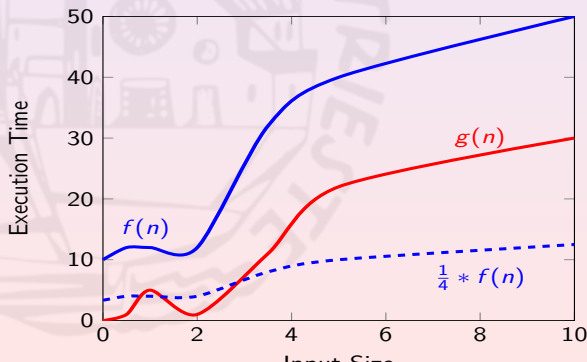
So, $g(n) \in \Omega(f(n))$ iff



big Ω notation

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

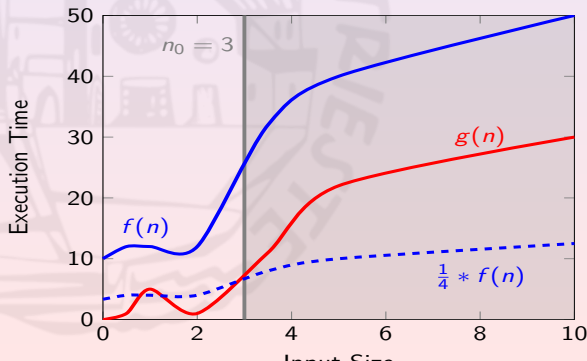
So, $g(n) \in \Omega(f(n))$ iff



big Ω notation

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

So, $g(n) \in \Omega(f(n))$ iff



big Θ notation

$$\Theta(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c_1, c_2 > 0 \exists n_0 > 0 \\ m \geq n_0 \implies c_1 * f(m) \leq g(m) \leq c_2 * f(m)\}$$

Theorem

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n))$$

Compute the Complexity of ...

Input: An array A of numbers $\langle a_1, \dots, a_n \rangle$.

Output: The maximum among a_1, \dots, a_n .

```

1  def find_max(A):
2      max_value ← A[1]
3      for i ← 2..|A|:
4          if A[i] > max_value:
5              max_value ← A[i]
6          endif
7      endfor
8
9      return max_value
10 enddef

```

- 2 costs $\Theta(1)$
- 4-6 cost $\Theta(1)$
- 4-6 repeated $\Theta(n)$ times
- 9 costs $\Theta(1)$

$\Theta(1) =$

Compute the Complexity of ...

Input: An array A of numbers $\langle a_1, \dots, a_n \rangle$.

Output: The maximum among a_1, \dots, a_n .

```

1  def find_max(A):
2      max_value ← A[1]
3      for i ← 2..|A|:
4          if A[i] > max_value:
5              max_value ← A[i]
6          endif
7      endfor
8
9      return max_value
10 enddef

```

- 2 costs $\Theta(1)$
- 4-6 cost $\Theta(1)$
- 4-6 repeated $\Theta(n)$ times
- 9 costs $\Theta(1)$

$$\Theta(1 + 1 \quad) =$$

Compute the Complexity of ...

Input: An array A of numbers $\langle a_1, \dots, a_n \rangle$.

Output: The maximum among a_1, \dots, a_n .

```

1  def find_max(A):
2      max_value ← A[1]
3      for i ← 2..|A|:
4          if A[i] > max_value:
5              max_value ← A[i]
6          endif
7      endfor
8
9      return max_value
10 enddef

```

- 2 costs $\Theta(1)$
- 4-6 cost $\Theta(1)$
- 4-6 repeated $\Theta(n)$ times
- 9 costs $\Theta(1)$

$$\Theta(1 + 1 * n) =$$

Compute the Complexity of ...

Input: An array A of numbers $\langle a_1, \dots, a_n \rangle$.

Output: The maximum among a_1, \dots, a_n .

```

1  def find_max(A):
2      max_value ← A[1]
3      for i ← 2..|A|:
4          if A[i] > max_value:
5              max_value ← A[i]
6          endif
7      endfor
8
9      return max_value
10 enddef

```

- 2 costs $\Theta(1)$
- 4-6 cost $\Theta(1)$
- 4-6 repeated $\Theta(n)$ times
- 9 costs $\Theta(1)$

$$\Theta(1 + 1 * n + 1) = \Theta(n)$$

The background of the slide features a large, faint watermark of the University of Pisa logo. The logo is circular and contains the text "UNIVERSITA' DEGLI STUDI DI PISA" around the top and "E SPLENDI" in the center. Below the text is a detailed illustration of the Leaning Tower of Pisa and other architectural elements.

Some Useful Notions

Arrays and Lists (Abstract Data Types)

Arrays Are indexed collections of values fixed in length.

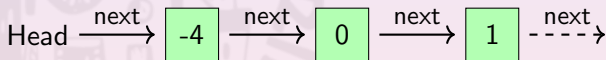
1	2	3	4	5	6	7	8	9	10
-4	0	1	2	5	6	7	11	12	13

Arrays and Lists (Abstract Data Types)

Arrays Are **indexed collections** of values **fixed in length**.

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

Single-Linked Lists Are **sequences** of values supporting **head** and **next** operations

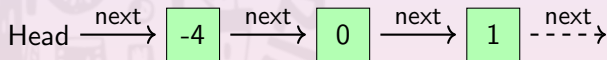


Arrays and Lists (Abstract Data Types)

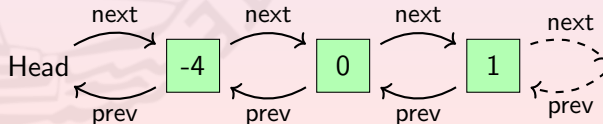
Arrays Are **indexed collections** of values **fixed in length**.

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

Single-Linked Lists Are **sequences** of values supporting **head** and **next** operations



Double-Linked Lists Are **sequences** of values supporting **head**, **next** and **previous** operations



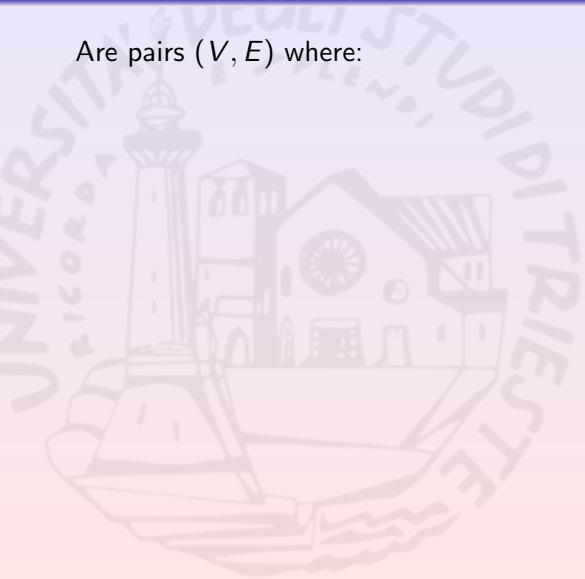
Queue and Stacks (Abstract Data Types)

Queues Are **collections** of values ruled according the **FIFO** policy. They support **head**, **is_empty**, **insert_back**, **extract_head** operations

Stacks Are **collections** of values ruled according the **LIFO** policy. They support **top**, **is_empty**, **insert_top**, **extract_top** operations

Graphs (Graph Theory)

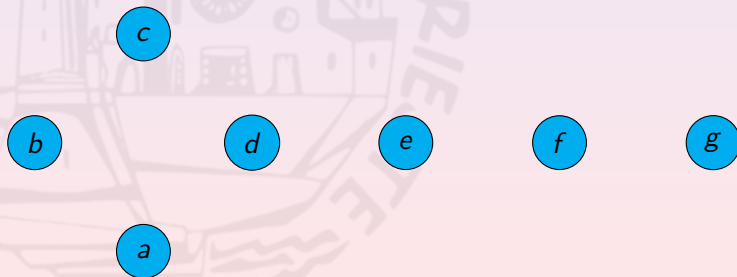
Are pairs (V, E) where:



Graphs (Graph Theory)

Are pairs (V, E) where:

V is a set of **nodes**

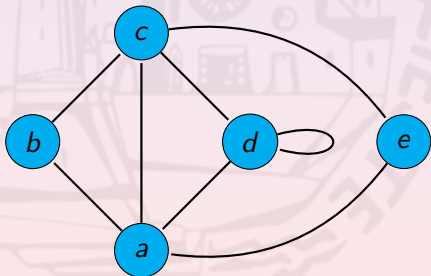


Graphs (Graph Theory)

Are pairs (V, E) where:

V is a set of **nodes**

E is a set of **edges**



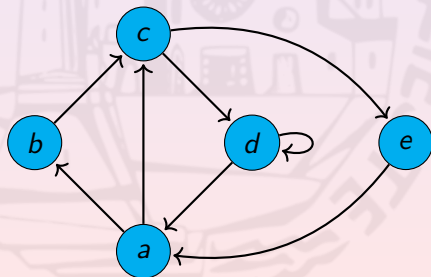
Graphs (Graph Theory)

Are pairs (V, E) where:

V is a set of **nodes**

E is a set of **edges**

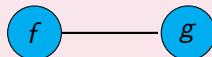
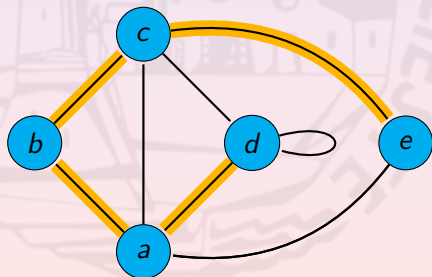
If the edges are **(un)directed**, the graph is (un)directed



Paths and Cycles

A **path** of length n between $a, b \in V$ is a sequence e_1, \dots, e_n s.t.

- e_1 involves a
- e_n involves b
- e_i and e_{i+1} involve a common node n_i

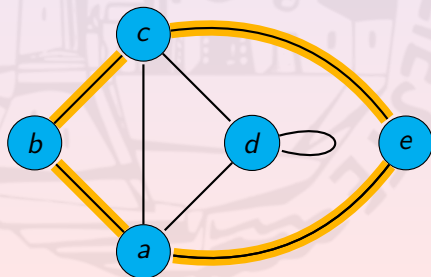


Paths and Cycles

A **path** of length n between $a, b \in V$ is a sequence e_1, \dots, e_n s.t.

- e_1 involves a
- e_n involves b
- e_i and e_{i+1} involve a common node n_i

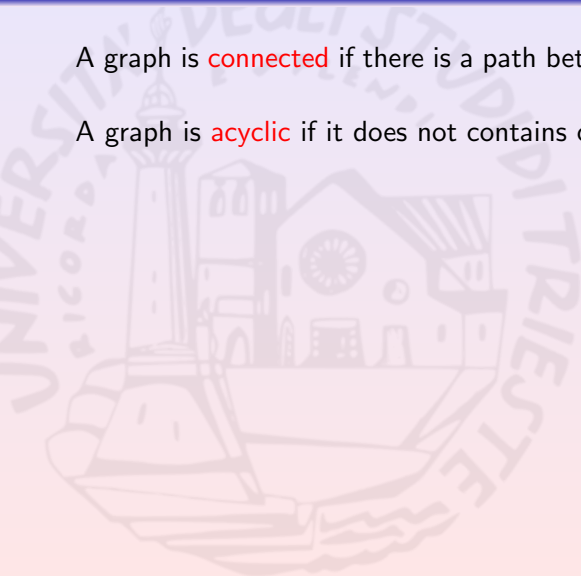
A **cycle** is a path whose initial and final node coincide.



Connected and Acyclic Graphs (Graph Theory)

A graph is **connected** if there is a path between every pairs of nodes

A graph is **acyclic** if it does not contains cycles

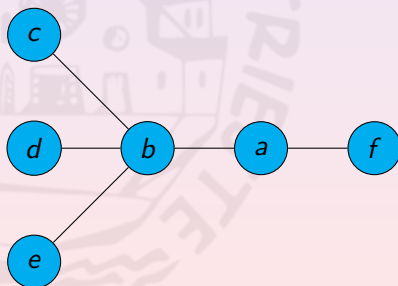


Connected and Acyclic Graphs (Graph Theory)

A graph is **connected** if there is a path between every pairs of nodes

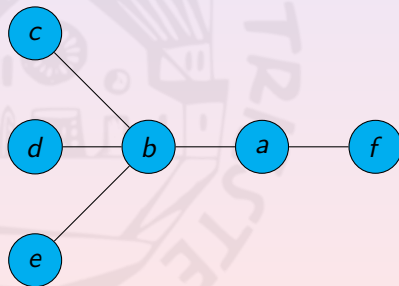
A graph is **acyclic** if it does not contains cycles

A **tree** is an connected and acyclic undirected graphs



Trees (Abstract Data Types)

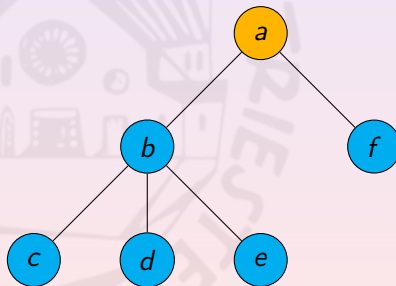
Organize data in a hierarchical finite tree (graph theory)



Trees (Abstract Data Types)

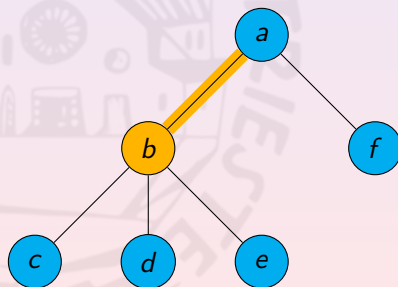
Organize data in a hierarchical finite tree (graph theory)

One of the nodes is the **root** of the graph



Tree Levels, Parents, Children and Siblings

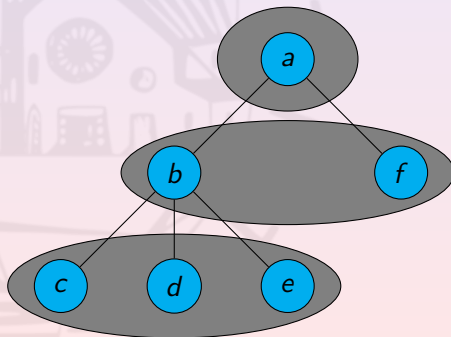
The **depth** of a node is its distance from the root



Tree Levels, Parents, Children and Siblings

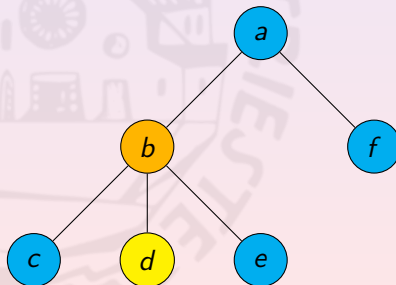
The **depth** of a node is its distance from the root

A **level** is a set of nodes having the same depth



Tree Levels, Parents, Children and Siblings

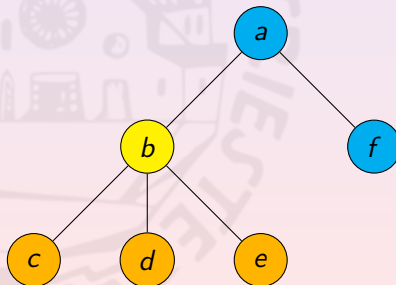
The **parent of a node** is a node one step closer to the root



Tree Levels, Parents, Children and Siblings

The **parent of a node** is a node one step closer to the root

The **children of a node** have it as parent

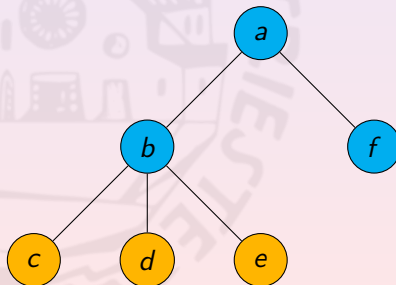


Tree Levels, Parents, Children and Siblings

The **parent of a node** is a node one step closer to the root

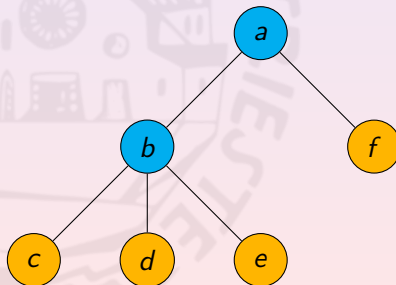
The **children of a node** have it as parent

Two nodes are **siblings** if they have the same parent



Tree Leaves and Height

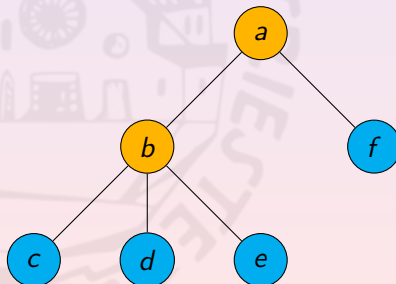
The **leaves** are nodes without children



Tree Leaves and Height

The **leaves** are nodes without children

The **internal nodes** have children

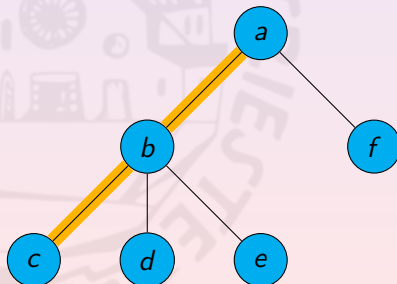


Tree Leaves and Height

The **leaves** are nodes without children

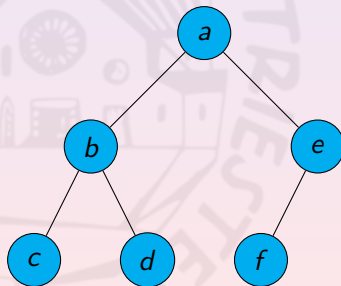
The **internal nodes** have children

The **height** of a tree is the max depth among those of its leaves



n -ary Tree and Completeness

Every node of a n -ary tree can have up to n children



n -ary Tree and Completeness

Every node of a n -ary tree can have up to n children

A n -ary tree is **complete** if the nodes in all the levels but the last one have n children

