

# Scientific Programming: Part B

---

## Lecture 2

Luca Bianco - Academic Year 2020-21  
luca.bianco@fmach.it  
[credits: thanks to Prof. Alberto Montresor]

# Introduction

**Goal:** *estimate the complexity in time of algorithms*

- Definitions
- Computing models
- Evaluation examples
- Notation

**Why?**

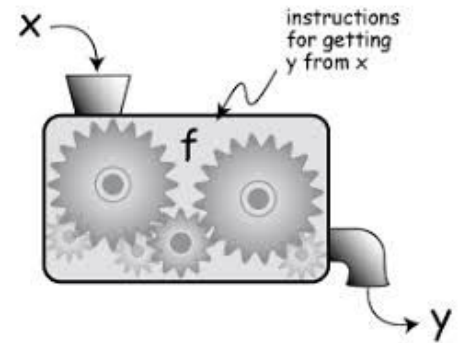
- To estimate the time needed to process a given input
- To estimate the largest input computable in a reasonable time
- To compare the efficiency of different algorithms
- To optimize the most important part

# Complexity

The **complexity** of an algorithm can be defined as a **function mapping the size of the input** to the **time** required to get the result. This is also called the **cost function**

We need to define two key aspects:

1. How to measure the **size of the input**
2. How to measure **time**



# How to measure the size of inputs

## Uniform cost model

- *The input size is equal to the number of elements composing it*
- Example: minimum search in a list of  $n$  elements

In some cases (e.g. factorial of a number) we need to consider how many bits we use to represent inputs

## Logarithmic cost model

- *The input size is equal to the number of bits representing it*
- Example: binary number multiplication of numbers of  $n$  bits

## In several cases...

- We can assume that the *elements* are represented by a constant number of bits
- The two measures are the same, apart from a constant multiplication factor

# Measuring time is trickier...

**Time  $\equiv$  wall-clock time**

The actual time used to complete an algorithm

It depends on too many parameters:

- how good is the programmer
- programming language
- code generated by the compiler/interpreter
- CPU, memory, hard-disk, etc.
- operating system, other processes currently running, etc.



We need a more abstract representation of time



# Random Access Machine (RAM): time

The computational model (abstract representation of the calculator) we will refer to:

## Memory

- Infinite size (we have all the memory we need)
- Access to memory is in constant time

## Processor (single CPU)

- The set of basic instructions are the ones we are used to:
  - +, -, \*, /, AND, OR, NOT, ...
  - control statements (for, while, if, return...)

## Cost of basic operations

- The cost is uniform and (as we will see) this is irrelevant for the calculation of the complexity of algorithms



# Random Access Machine (RAM): time

The computational model we will refer to

What are basic operations?

**Time**  $\equiv$  number of basic instructions

An instruction is considered basic if it can be executed in constant time by the processor

**Basic**

- $a = a*2$  ? Yes (unless numbers have arbitrary precision)
- `math.cos(d)` ? Yes
- $\min(A)$  ? No (modern GPUs are highly parallel and can be constant)



# Example: minimum

Let's count the **number of basic operations for min**.

- Each statement requires a constant time to be executed (even len()???) YES (if interested check <https://wiki.python.org/moin/TimeComplexity>)
- This constant may be different for each statement
- Each statement is executed a given number of times, function of n (size of input).

```
def my_faster_min(S):  
    min_so_far = S[0] #first element  
    i = 1  
    while i < len(S):  
        if S[i] < min_so_far:  
            min_so_far = S[i]  
        i = i + 1  
    return min_so_far
```



# Example: minimum

Let's count the **number of basic operations for min.**

- Each statement requires a constant time to be executed (even len???)  
(if interested check <https://wiki.python.org/moin/TimeComplexity>)
- This constant may be different for each statement
- Each statement is executed a given number of times, function of n (size of input).

	Cost	Number of times
<code>def my_faster_min(S):</code>		
<code>min_so_far = S[0] <i>#first element</i></code>	c1	1
<code>i = 1</code>	c2	1
<b>while</b> <code>i &lt; len(S):</code>	c3	n
<b>if</b> <code>S[i] &lt; min_so_far:</code>	c4	n-1
<code>min_so_far = S[i]</code>	c5	n-1 (worst case: S is sorted decreasingly )
<code>i = i + 1</code>	c6	n-1
<b>return</b> <code>min_so_far</code>	c7	1

$$\begin{aligned}T(n) &= c1 + c2 + c3*n + c4*(n-1) + c5*(n-1)+c6*(n-1)+c7 \\ &= (c3+c4+c5+c6)*n + (c1+c2-c4-c5-c6+c7) = \mathbf{a*n + b}\end{aligned}$$

# Example: lookup

Let's count the **number of basic operations for lookup**.

- The list is split in two parts: left size  $\lfloor (n-1)/2 \rfloor$  right size  $\lfloor n/2 \rfloor$

```
def lookup_rec(L, v, start, end):  
    if end < start:  
        return -1  
    else:  
        m = (start + end)//2  
        if L[m] == v: #found!  
            return m  
        elif v < L[m]: #look to the left  
            return lookup_rec(L, v, start, m-1)  
        else: #look to the right  
            return lookup_rec(L, v, m+1, end)
```

# Example: lookup

Let's count the **number of basic operations for lookup**.

- The list is split in two parts: left size  $\lfloor (n-1)/2 \rfloor$  right size  $\lfloor n/2 \rfloor$

	Cost	Executed?	
		$end < start$	$end \geq start$
<code>def lookup_rec(L, v, start, end):</code>			
<code>if end &lt; start:</code>	c1	1	1
<code>return -1</code>	c2	1	0
<code>else:</code>			
<code>m = (start + end)//2</code>	c3	0	1
<code>if L[m] == v: #found!</code>	c4	0	1
<code>return m</code>	c5	0	0 (worst case)
<code>elif v &lt; L[m]: #look to the left</code>	c6	0	1
<code>return lookup_rec(L, v, start, m-1)</code>	$c7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
<code>else: #look to the right</code>			
<code>return lookup_rec(L, v, m+1, end)</code>	$c7 + T(\lfloor n/2 \rfloor)$	0	1/0

**Note: lookup\_rec is not a basic operation!!!**

# Lookup: recurrence relation

Assumptions:

- For simplicity, n is a power of 2:  $n = 2^k$
- The searched element is not present (worst case)
- At each call, we select the right part whose size is  $n/2$  ( instead of  $(n-1)/2$  )

if  $start > end$  ( $n=0$ ):

$$T(n) = c_1 + c_2 = c$$

if  $start \leq end$  ( $n>0$ ):

$$T(n) = T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 = T(n/2) + d$$

Recurrence relation:

$$T(n) = \begin{cases} c & n = 0 \\ T(n/2) + d & n \geq 1 \end{cases}$$

	Cost	Executed?	
		end < start	end ≥ start
<code>def lookup_rec(L, v, start, end):</code>			
<code>  if end &lt; start:</code>	c1	1	1
<code>    return -1</code>	c2	1	0
<code>  else:</code>			
<code>    m = (start + end)//2</code>	c3	0	1
<code>    if L[m] == v: #found!</code>	c4	0	1
<code>      return m</code>	c5	0	0 (worst case)
<code>    elif v &lt; L[m]: #look to the left</code>	c6	0	1
<code>      return lookup_rec(L, v, start, m-1)</code>	$c7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
<code>    else: #look to the right</code>			
<code>      return lookup_rec(L, v, m+1, end)</code>	$c7 + T(\lfloor n/2 \rfloor)$	0	1/0

# Lookup: recurrence relation

Solution from **recurrence relation** to **closed formula**

Remember that:  $n = 2^k \Rightarrow k = \log_2 n$

$$T(n) = \begin{cases} c & n = 0 \\ T(n/2) + d & n \geq 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n/2) + d \\ &= (T(n/4) + d) + d = T(n/4) + 2d \\ &= (T(n/8) + d) + 2d = T(n/8) + 3d \\ &\dots \\ &= T(1) + kd \\ &= T(0) + (k + 1)d \\ &= kd + (c + d) \\ &= d \log n + e. \end{aligned}$$

closed formula



as seen before, the complexity is logarithmic  
**Note:** in computer science log is log<sub>2</sub>.

# Asymptotic notation

A cost function is simply a function from natural numbers to real numbers:

$$f(n) : \mathbb{N} \mapsto \mathbb{R}$$

# Asymptotic notation

A cost function is simply a function from natural numbers to real numbers:

Cost functions → “big-Oh” notation (omicron)

$$f(n) : \mathbb{N} \mapsto \mathbb{R}$$

Cost functions seen so far...

- Lookup:  $T(n) = d \cdot \log n + e$
- Minimum:  $T(n) = a \cdot n + b$
- Naive Minimum:  $T(n) = f \cdot n^2 + g \cdot n + h$

logarithmic


$O(\log n)$

linear

$O(n)$

quadratic

$O(n^2)$



we ignore the “less impacting” parts (like constants or  $n$  in naive, ...) and focus on the predominant ones

# Asymptotic notation

## Complexity classes

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Type
$\log n$	3	6	9	13	logarithmic
$\sqrt{n}$	3	10	31	100	sublinear
$n$	10	100	1000	10000	linear
$n \log n$	30	664	9965	132877	log-linear
$n^2$	$10^2$	$10^4$	$10^6$	$10^8$	quadratic
$n^3$	$10^3$	$10^6$	$10^9$	$10^{12}$	cubic
$2^n$	1024	$10^{30}$	$10^{300}$	$10^{3000}$	exponential

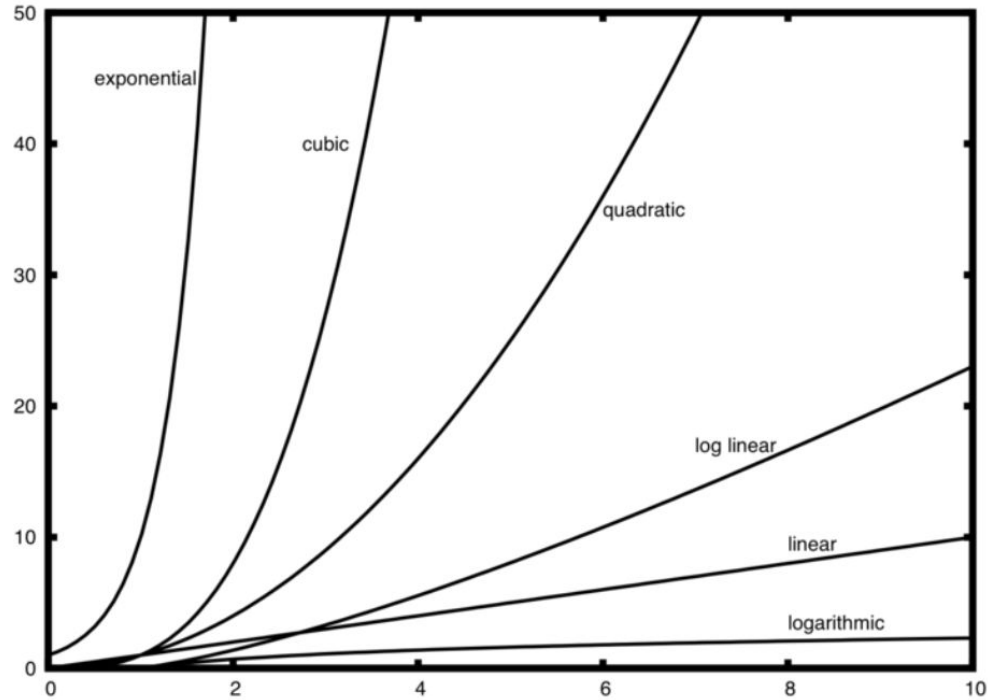


**Example:** If one operation takes 1/billionths of a second  $10^{(-9)}$  s, if  $n=10,000 \rightarrow$  cubic  $\sim 17$  minutes . The age of the universe is  $\sim 10^{18}$  seconds

**Note:** these are “trends” (we hide all constants that might have an impact for small inputs). For small inputs exponential algorithms might still be acceptable (especially if nothing better exists!)



# Asymptotic notation



# $O, \Omega, \Theta$ notations

## Definition – $O$ notation

Let  $g(n)$  be a cost function;  $O(g(n))$  is the set of all functions  $f(n)$  such that:

$$\exists c > 0, \exists m \geq 0 : f(n) \leq cg(n), \forall n \geq m$$

- How we read it:  $f(n)$  is “big-Oh” of  $g(n)$
- How we write it:  $f(n) = O(g(n))$
- $g(n)$  is asymptotic upper bound for  $f(n)$
- $f(n)$  grows at most as  $g(n)$



Note that this relation between  $f(n)$  and  $g(n)$  is valid asymptotically (i.e. for all  $n$  after a certain point  $m$ )




# O, $\Omega$ , $\Theta$ notations

## Definition – $\Omega$ notation

Let  $g(n)$  be a cost function;  $\Omega(g(n))$  is the set of all functions  $f(n)$  such that:

$$\exists c > 0, \exists m \geq 0 : f(n) \geq cg(n), \forall n \geq m$$

- How we read it:  $f(n)$  is “**big-omega**” of  $g(n)$
- How we write it:  $f(n) = \Omega(g(n))$
- $g(n)$  is an **asymptotic lower bound** for  $f(n)$  
- $f(n)$  grows at least as  $g(n)$

Again, note that this relation between  $f(n)$  and  $g(n)$  is valid asymptotically (i.e. for all  $n$  after a certain point  $m$ )



# O,Ω,Θ notations

## Definition – Notation Θ

Let  $g(n)$  be a cost function;  $\Theta(g(n))$  is the set of all functions  $f(n)$  such that:

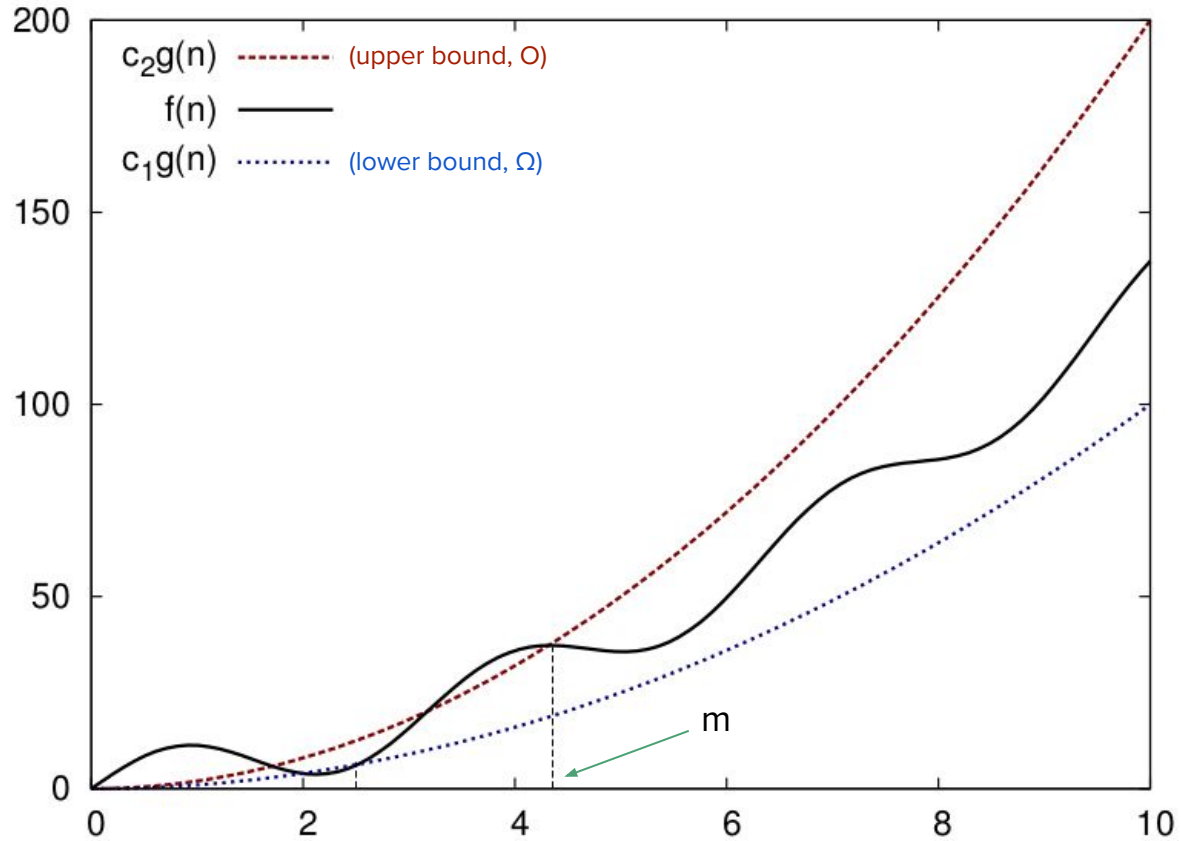
$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq m$$

- How we read it:  $f(n)$  is “**theta**” of  $g(n)$
- How we write it:  $f(n) = \Theta(g(n))$
- $f(n)$  grows as  $g(n)$
- $f(n) = \Theta(g(n))$  iff  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$  ←

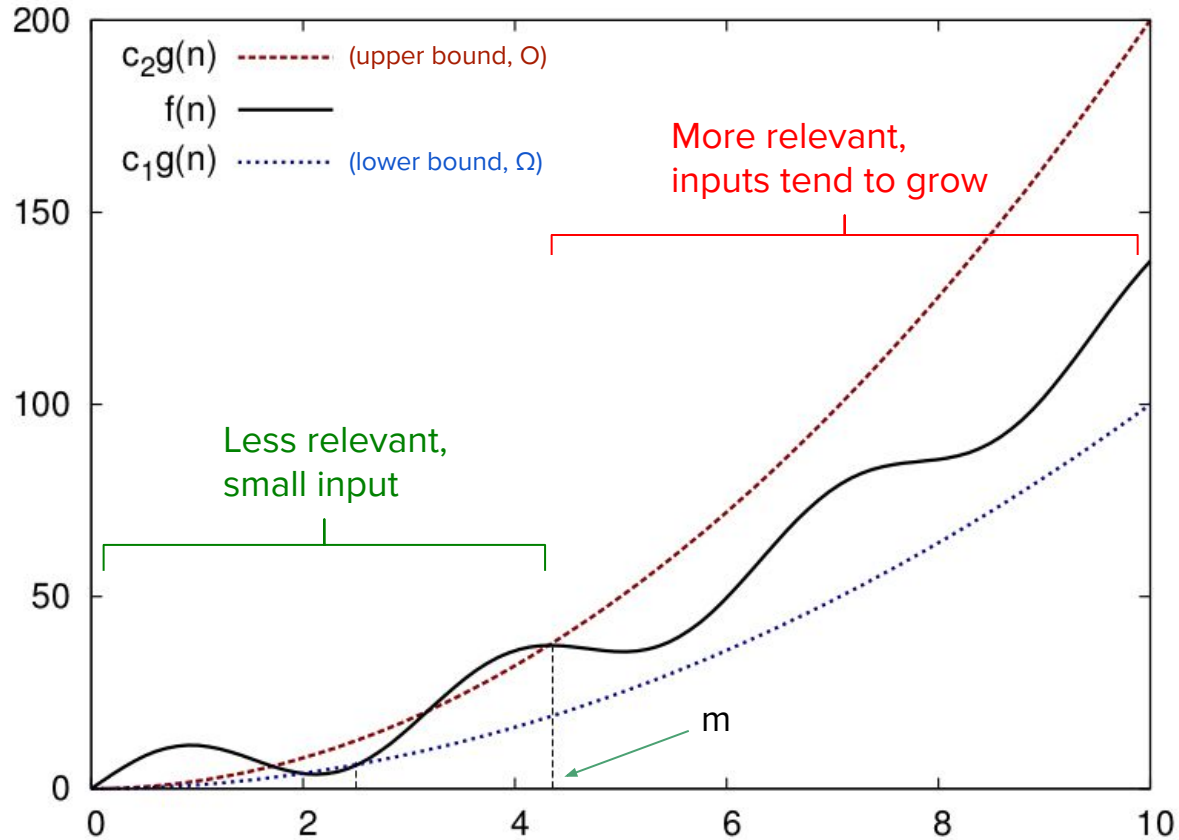
Again, note that this relation between  $f(n)$  and  $g(n)$  is valid asymptotically (i.e. for all  $n$  after a certain point  $m$ )



# $O, \Omega, \Theta$ notations



# $O, \Omega, \Theta$ notations



# Exercise: True or False?

$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

We need to prove that (i.e. find a  $c$  and  $m$  such that):

$$\exists c > 0, \exists m \geq 0 : f(n) \leq c \cdot n^3, \forall n \geq m$$

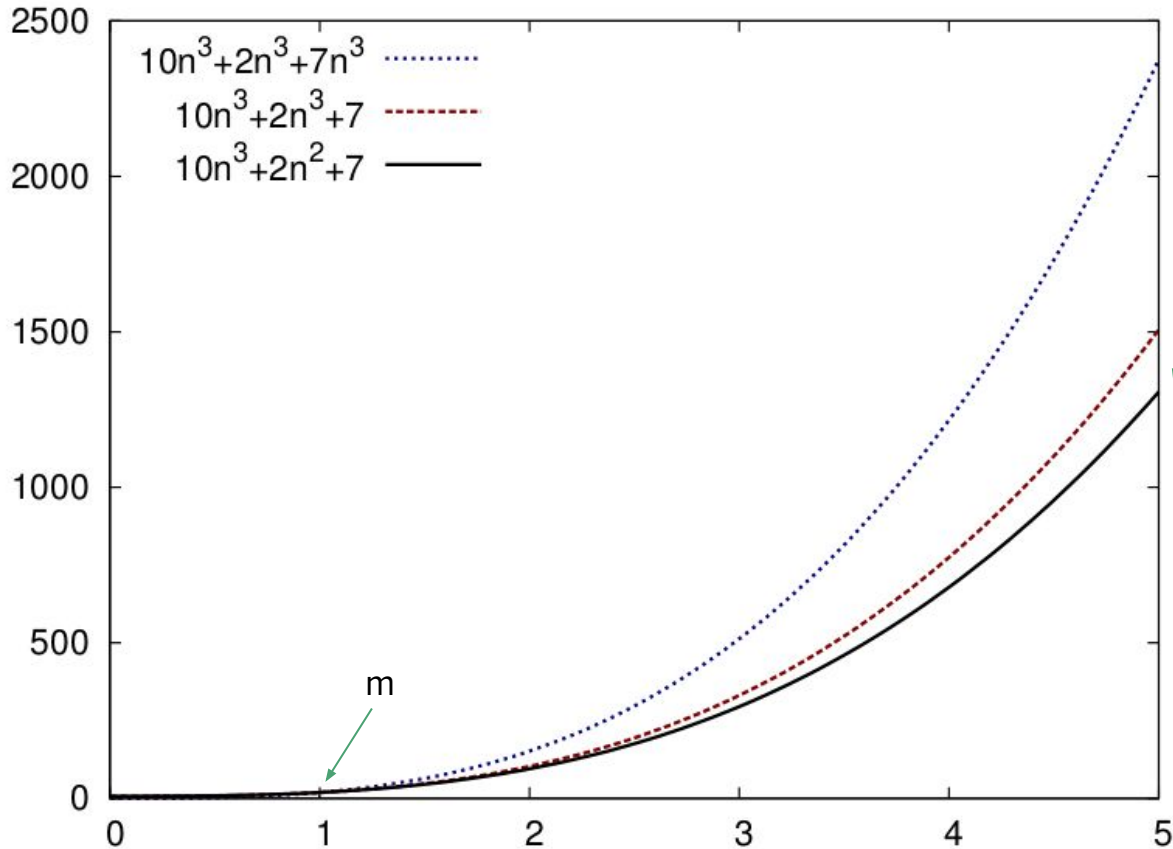
$$\begin{aligned} f(n) &= 10n^3 + 2n^2 + 7 \\ &\leq 10n^3 + 2n^3 + 7 && \forall n \geq 0 \\ &\leq 10n^3 + 2n^3 + 7n^3 && \forall n \geq 1 \\ &= 19n^3 \\ &\stackrel{?}{\leq} cn^3 \end{aligned}$$

which is true for each  $c \geq 19$  and for each  $n \geq 1$ , thus  $m = 1$ .

Hence I found  $c$ , and  $m$  that satisfy the disequation for all  $n \geq m \rightarrow \mathbf{f(n) = O(n^3)}$

# In graphical terms

$$f(n) = 10n^3 + 2n^2 + 7$$





# Exercise: True or False?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

We need to prove that (i.e. find a c and m such that):

$$\exists c_1 > 0, \exists m_1 \geq 0 : f(n) \geq c_1 \cdot n^2, \forall n \geq m_1 \quad \text{lower bound } (\Omega)$$

and that

$$\exists c_2 > 0, \exists m_2 \geq 0 : f(n) \leq c_2 \cdot n^2, \forall n \geq m_2 \quad \text{upper bound } (O)$$


# Exercise: True or False?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

We need to prove that (i.e. find a  $c$  and  $m$  such that):

$$\exists c_1 > 0, \exists m_1 \geq 0 : \underline{f(n) \geq c_1 \cdot n^2}, \forall n \geq m_1 \quad \text{lower bound } (\Omega)$$

$$\begin{aligned} f(n) &= 3n^2 + 7n \\ &\geq 3n^2 && n \geq 0 \\ &\stackrel{?}{\geq} c_1 n^2 \end{aligned}$$

which is true for each  $c_1 \leq 3$  and for each  $n \geq 0$ , thus  $m_1 = 0$    $f(n) = \Omega(n^2)$

# Exercise: True or False?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

We need to prove that (i.e. find a  $c$  and  $m$  such that):

$$\exists c_2 > 0, \exists m_2 \geq 0 : \underline{f(n) \leq c_2 \cdot n^2}, \forall n \geq m_2 \quad \text{upper bound (O)}$$

$$\begin{aligned} f(n) &= 3n^2 + 7n \\ &\leq 3n^2 + 7n^2 && n \geq 1 \\ &= 10n^2 \\ &\stackrel{?}{\leq} c_2 n^2 \end{aligned}$$

which is true for each  $c_2 \geq 10$  and for all  $n \geq 1$ , hence  $m_2 = 1$ .



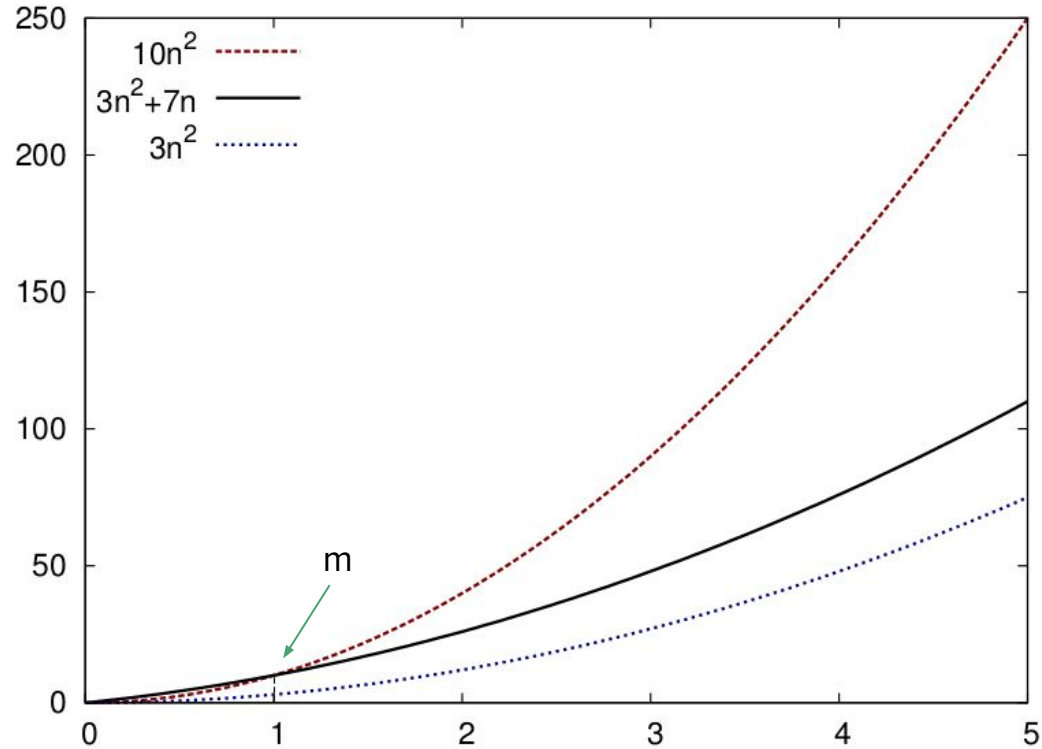
$$f(n) = O(n^2)$$



$$f(n) = 3n^2 + 7n = \Theta(n^2)$$

for all  $n > m = \max(m_1, m_2) = 1$

In graphical terms:  $3n^2+7n$  is  $\Theta(n^2)$



# True or False?

$$n^2 \stackrel{?}{=} O(n)$$

We want to prove that  $\exists c > 0, \exists m > 0 : n^2 \leq cn, \forall n \geq m$

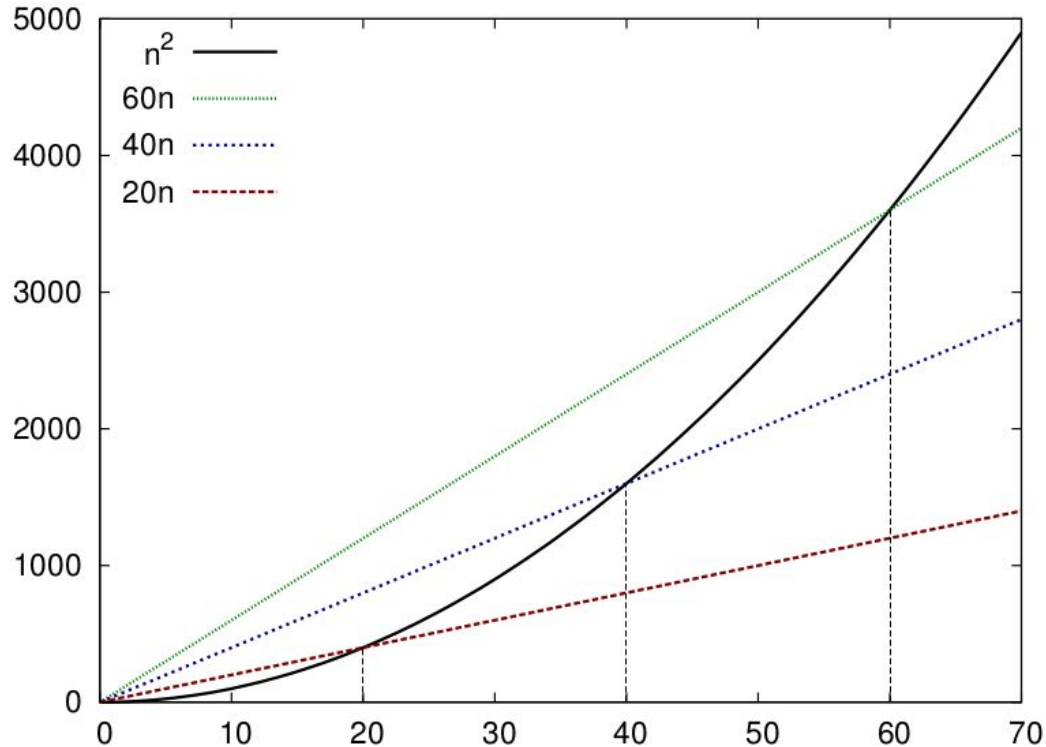
- We get this:  $n^2 \leq cn \Leftrightarrow c \geq n$
- This means that  $c$  should grow with  $n$ , i.e. we cannot choose a constant  $c$  valid for all  $n \geq m$



$$n^2 \neq O(n)$$

# True or False?

$$n^2 \neq O(n)$$



we cannot find a constant C making n grow faster than  $n^2$

Exercise:

$$n^2 = O(n^3)$$

Hint:  $c \geq 1/n$   
is monotonically  
decreasing

# True or False?

$$n^2 = O(n^3)$$

We need to prove that  $\exists c > 0, \exists m > 0 : n^2 \leq cn^3, \forall n \geq m$

- We get this:  $n^2 \leq cn^3 \Leftrightarrow c \geq \frac{1}{n}$
- Given that  $1/n$  is monotonically decreasing for  $n > 0$ , we can choose any value of  $m$  (e.g.,  $m = 1$ ), and select a constant  $c \geq 1/m$ , such as  $c = 1$ .

# Properties

## Polynomial expressions

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0, a_k > 0 \Rightarrow f(n) = \Theta(n^k)$$

## Constant elimination

$$f(n) = O(g(n)) \Leftrightarrow a f(n) = O(g(n)), \forall a > 0$$

$$f(n) = \Omega(g(n)) \Leftrightarrow a f(n) = \Omega(g(n)), \forall a > 0$$

### Meaning:

- We only care about the highest degree of the polynomial
- Multiplicative constants do not change the asymptotic complexity (e.g. constants costs due to language, technical implementation,...)

### Example:

$$f(n) = n^4 + 4n^2 - 2n + 1 = \Theta(n^4)$$





# Properties

## Sums

$$\begin{aligned}f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) &\Rightarrow \\f_1(n) + f_2(n) &= O(\max(g_1(n), g_2(n))) \\f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) &\Rightarrow \\f_1(n) + f_2(n) &= \Omega(\min(g_1(n), g_2(n)))\end{aligned}$$

## Relation with algorithm analysis

- If an algorithm is composed by two parts, one which is  $\Theta(n^2)$  and one which  $\Theta(n)$ , the resulting complexity is  $\Theta(n^2 + n) = \Theta(n^2)$

We only care about the “computationally more expensive” part to solve of the algorithm.

$$O(n \cdot \log n + n) = O(n \cdot \log n)$$



# Properties

## Products

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

## Relation with algorithm analysis

- If algorithm  $A$  calls algorithm  $B$   $n$  times, and the complexity of algorithm  $B$  is  $\Theta(n \log n)$ , the resulting complexity is  $\Theta(n^2 \log n)$ .

```
for i in range(n):  
    call_to_function_that_is_n^2_log_n() }  $\Theta(n^2 \log n)$ 
```



# The total order

Is it possible to create a total order between the main function classes.

For each  $0 < r < s, 0 < h < k, 1 < a < b$ :

$$O(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n) \subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n)$$

Examples:

$$O(\log n) \subset O(\sqrt[3]{n}) \subset O(\sqrt{n})$$

$$O(2^{n+1}) = O(2 \cdot 2^n) = O(2^n)$$

No matter the exponent  $r$ ,  $(\log n)^r$  will always be better than  $n$ ...

Same thing for  $n \log n$  vs  $n$  etc...

# Complexity of maxsum: $\Theta(n^3)$

```
def max_sum_v1(A):
    max_so_far = 0
    N = len(A)
    for i in range(N):
        for j in range(i, N):
            tmp_sum = sum(A[i:j+1])
            max_so_far = max(tmp_sum, max_so_far)

    return max_so_far
```

Intuitively:

we perform two loops of length  $N$   
one into the other  $\rightarrow$  cost  $N^2$

sum is not a basic operation (cost  $N$ ):



overall cost  $N^3$

The complexity of this algorithm can be approximated as follows (we are counting the number of sums that are executed).

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$$

We want to prove that  $T(n) = \theta(n^3)$ , i.e.

$$\exists c_1, c_2 > 0, \exists m \geq 0 : c_1 n^3 \leq T(n) \leq c_2 n^3, \forall n \geq m$$

# Complexity of maxsum: $O(n^3)$

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) \\ &\leq \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} n \\ &\leq \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n \end{aligned}$$

This is actually =

$$\begin{aligned} &= \sum_{i=0}^{n-1} n^2 \\ &\leq n^3 \leq c_2 n^3 \end{aligned}$$

This inequality is true for  $n \geq m = 0$  and  $c_2 \geq 1$ .



$O(n^3)$

# Complexity of maxsum: $\Omega(n^3)$

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) \\ &\geq \sum_{i=0}^{n/2} \sum_{j=i}^{i+n/2-1} (j - i + 1) \longrightarrow 1 + 2 + \dots + \frac{n}{2} = \sum_{j=0}^{\frac{n}{2}} j = \frac{1}{2} \frac{n}{2} \left( \frac{n}{2} + 1 \right) = \frac{n^2}{8} + \frac{n}{4} \geq \frac{n^2}{8} \forall n \geq 0 \\ &\geq \sum_{i=0}^{\frac{n}{2}} \frac{n^2}{8} = \frac{n^3}{16} \geq cn^3 \end{aligned}$$


Gauss

This inequality is true  $\forall n \geq m = 0$  and  $c_1 \leq \frac{1}{16}$   $\Rightarrow \Omega(n^3)$

$\Rightarrow \text{maxsum} = \Theta(n^3)$

# Complexity of maxsum -version 2: $\Theta(n^2)$

```
def max_sum_v2(A):  
    N = len(A)  
    max_so_far = 0  
  
    for i in range(N):  
        tot = 0 #ACCUMULATOR!  
        for j in range(i,N):  
            tot = tot + A[j]  
            max_so_far = max(max_so_far, tot)  
    return max_so_far
```



at each iteration of  $i$ , we perform  $n - i$  sums and max

The complexity of this algorithm can be approximated as follows (we are counting the number of sums that are executed).

$$T(n) = \sum_{i=0}^{n-1} n - i$$

# Complexity of maxsum -version 2: $\Theta(n^2)$

We want to prove that  $T(n) = \theta(n^2)$ .

$$\begin{aligned}T(n) &= \sum_{i=0}^{n-1} n - i = n + (n-1) + \dots + 1 \\ &= \sum_{i=1}^n i \\ &= \frac{n(n+1)}{2} = \Theta(n^2)\end{aligned}$$

Gauss

This does not require further proofs.



# Complexity of maxsum -version 4: $\Theta(n)$

```
def max_sum_v4(A):  
    max_so_far = 0 #Max found so far  
    max_here = 0 #Max slice ending at cur pos  
  
    for i in range(len(A)):  
        max_here = max(A[i] + max_here, 0)  
        max_so_far = max(max_so_far, max_here)  
    return max_so_far
```

} This is rather easy!  
Constant operations (sum and  
max of 2 numbers) performed  
n times

Complexity is  $\Theta(n)$

# Complexity of maxsum -version 3

```
from itertools import accumulate

def max_sum_v3_rec_bis(A,i,j):
    if i == j:
        return max(0,A[i])
    m = (i+j)//2
    maxL = max_sum_v3_rec_bis(A,i,m)
    maxR = max_sum_v3_rec_bis(A, m+1, j)
    maxML = max(accumulate(A[m:-len(A) + i - 1: -1]))
    maxMR = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxML+ maxMR)

def max_sum_v3(A):
    return max_sum_v3_rec_bis(A,0,len(A) - 1)
```

} Recursive algorithm,  
recurrence relation

Bear with me a minute.  
We will get back to this  
later...!

# Recurrences

## Recurrence equations

Whenever the complexity of a recursive algorithm is computed, this is expressed through **recurrence equation**, i.e. a mathematical formula defined in a... recursive way!

### Example

$$T(n) = \begin{cases} 2T(n/2) + n & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

# Recurrences

## Closed formulas

Our goal is to obtain, whenever possible, a **closed formula** that represents the complexity class of our function.

## Example

$$T(n) = \Theta(n \log n)$$

# Master Theorem

## Theorem

Let  $a$  and  $b$  two integer constants such that  $a \geq 1$  e  $b \geq 2$ , and let  $c, \beta$  be two real constants such that  $c > 0$  e  $\beta \geq 0$ . Let  $T(n)$  be defined by the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Given  $\alpha = \log a / \log b = \log_b a$ , then:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$



**Note:** the schema covers cases when input of size  $n$  is split in  $b$  sub-problems, to get the solution the algorithm is applied recursively  $a$  times.  $cn^\beta$  is the cost of the algorithm after the recursive steps.

# Examples

## Theorem

Let  $a$  and  $b$  two integer constants such that  $a \geq 1$  e  $b \geq 2$ , and let  $c, \beta$  be two real constants such that  $c > 0$  e  $\beta \geq 0$ . Let  $T(n)$  be defined by the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Given  $\alpha = \log a / \log b = \log_b a$ , then:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

Algo: splits the input in two, applies the procedure recursively 4 times and has a linear cost to assemble the solution at the end.

Recurrence	a	b	$\log_b a$	Case	Function
$T(n) = 4T(n/2) + n$	4	2	2	(1)	$T(n) = \Theta(n^2)$
$T(n) = 3T(n/2) + n$	3	2	$\log_2 3$	(1)	$T(n) = \Theta(n^{\log_2 3})$
$T(n) = 2T(n/2) + n$	2	2	1	(2)	$T(n) = \Theta(n \log n)$
$T(n) = T(n/2) + 1$	1	2	0	(2)	$T(n) = \Theta(\log n)$
$T(n) = 9T(n/3) + n^3$	9	3	2	(3)	$T(n) = \Theta(n^3)$

$\leftarrow n^{1.58}$

Note: the schema covers cases when input of size  $n$  is split in  $b$  sub-problems, to get the solution the algorithm is applied recursively  $a$  times.  $cn^\beta$  is the cost of the algorithm after the recursive steps.

# maxsum - version 3

```
from itertools import accumulate

def max_sum_v3_rec_bis(A,i,j):
    if i == j:
        return max(0,A[i])
    m = (i+j)//2
    maxL = max_sum_v3_rec_bis(A,i,m)
    maxR = max_sum_v3_rec_bis(A, m+1, j)
    maxML = max(accumulate(A[m:-len(A) + i -1: -1]))
    maxMR = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxML+ maxMR)

def max_sum_v3(A):
    return max_sum_v3_rec_bis(A,0,len(A) - 1)
```

The algorithm **splits the input in two** “equally-sized” sub-problems ( $m = i+j//2$ ) and **applies itself recursively 2 times**. The accumulate after the recursive part is **linear:  $cn$** .

For this, we need to define a recurrence relation:

$$T(n) = 2T(n/2) + cn$$

# maxsum - version 3

```
from itertools import accumulate

def max_sum_v3_rec_bis(A,i,j):
    if i == j:
        return max(0,A[i])
    m = (i+j)//2
    maxL = max_sum_v3_rec_bis(A,i,m)
    maxR = max_sum_v3_rec_bis(A, m+1, j)
    maxML = max(accumulate(A[m:-len(A) + i -1: -1]))
    maxMR = max(accumulate(A[m+1:j+1]))
    return max(maxL, maxR, maxML+ maxMR)

def max_sum_v3(A):
    return max_sum_v3_rec_bis(A,0,len(A) - 1)
```

For this, we need to define a recurrence relation:

$$T(n) = 2T(n/2) + cn$$

## Theorem

Let  $a$  and  $b$  two integer constants such that  $a \geq 1$  e  $b \geq 2$ , and let  $c, \beta$  be two real constants such that  $c > 0$  e  $\beta \geq 0$ . Let  $T(n)$  be defined by the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Given  $\alpha = \log a / \log b = \log_b a$ , then:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$



$$\alpha = \log_2 2 = 1 \text{ and } \beta = 1$$

$$T(n) = \Theta(n \log n)$$

