Information, Computation, and Communication

Algorithm 1

What is an Algorithm?

- ***An algorithm is an effective method, expressed as a finite list of well-defined instructions for calculating a function.***

[Wikipedia]

- Algorithms exist **well before computers**: already in ancient history (e.g., Egyptian division, Euclid's algorithm for greatest-common-divisor)

Shaping the Future

- “**Algorithm’ is arguably the single most important concept in our world**. If we want to understand our life and our future, we should make every effort to understand what an algorithm is, and how algorithms are connected with emotions.”

Yuval Noah Harari, Homo Deus: A Brief History of Tomorrow

Examples

- Binary addition
- Binary to decimal conversion and vice versa
- Procedure to compute the 2s complement
- Solving a quadratic equation (e.g., $3x^2 + 5x + 2$)
- Sorting a list of elements

- Searching for an element in a list
- Identifying gene in a DNA-sequence
- Pagerank/Edgerank

How to describe an Algorithm? Pseudo-Code

- Name and description of inputs and output

Valeur minimale
entrée : *liste L (non-vide) de nombres entiers* sortie : *la valeur minimale de la liste*
(instructions)

Minimal Value
input : *(non-empty) liste L of integers* output : *the smallest value in the list*
(instructions)

Cf. document **pseudo-code** on Moodle

- Variables (Place holder) to refer to a single element or a list of elements
- Access an element in a list (parenthèse/crochet): L(1),L[1],A[4]
- Access a sub-list: L(1:4), A[5:6]

We start counting at 1, i.e., L(1) is the first element in the list.

- Assignments (Affectation) : $x \longleftarrow 3 \quad x_{\min} \longleftarrow L(i)$
- Mathematical operators: $+, -, \cdot, /, \mod, <, \leq, =, \neq, \geq, >, \lfloor\rfloor$

  $x \geq 2 \qquad x \longleftarrow L(1) + 2$

- Reference to another algorithm (sous-algorithme):

$n \longleftarrow \textbf{taille}(L) \qquad n \longleftarrow \textbf{algorithm1}(L, n)$

$L' \longleftarrow \textbf{tri par insertion}(L) \qquad sortedL \longleftarrow \textbf{sort}(L, n)$

Pseudo-Code (Cnt.)

- Control structures (if, for, while)

Tests	**Si** condition
	  instructions
	Sinon
	  instructions
		if condition
		  instructions
		else
		  instructions

Loops	**Pour** i allant de 1 à n
	  instructions
	Pour i allant de 1 à n de 2 en 2
	  instructions
	Pour i allant de n à 1 en descendant
	  instructions
		for i from 1 to n
		  instructions
		for i from 1 to n in increments of 2
		  instructions
		for i from n to 1 going down
		  instructions

Conditional Loops	**Tant que** condition
	  instructions
		while condition
		  instructions

- Termination statement:

Sortir : x
return : x

Pseudo-Code (Lists)

- Create an empty list: A=empty or A={}
- Create a list with some element: A = {2,3,4}
- Add an element to the list: A.append(2) or "append 2 to A"
- Overwrite an element in the list: A(1) ← 3
- **An element of the list can be anything! Even another list**, e.g., A = { {2,3,4}, {1,2,3}, {3,4}}, or a list of list, e.g., B = { { {1,2}, {2,3}, {3} }, { {1}, {2,3}} } }
  - A(1) gives the first element = {2,3,4}
  - A(1)(3) takes the 1st element and takes the 3rd element = 4

Your Tasks

1. Understand an algorithm
2. Write an algorithm
3. Analyze the correctness and complexity of an algorithm

Task 1: Understanding an Algorithm

Algorithm 1

input : *(non-empty) liste L with n integers*
output : *an integer*

$x_{\max} \longleftarrow L[1]$
for i *from* 2 *to* n
  if $L[i] > x_{\max}$
    $x_{\max} \longleftarrow L[i]$
return : $x_{\max}$

What computation does this algorithm perform for L = {3, 6, 2, 1, 10, 9}? (n=6)

Line 1, 4 x_{max}	Line 2 i	Line 3 $L[i] > x_{max}$
3		
6	2	6 > 3 (true)
6	3	2 > 6 (false)
6	4	1 > 6 (false)
10	5	10 > 6 (true)
10	6	9 > 10 (false)
10	return	

It computes the maximum of the list, i.e., it returns 10

Another Example

What is the output of algo1 if L = {3, 6, 2, 1, 10, 9}?

Algorithm 2

input : *liste L with n numbers*
output : *a number x*

$s \longleftarrow 0$
for i *from* 1 *to* n
  if $L[i] \mod 2 = 0$
    $s \longleftarrow s + 1$
return : s

Line 1,4 s	Line 2 i	Line 3 L[i] mod 2 = 0
0		
0	1	Is 3 even?
1	2	Is 6 even?
2	3	Is 2 even?
2	4	Is 1 even?
3	5	Is 10 even?
3	6	Is 9 even?
3	return	

Output=3
The algorithm counts the even numbers.

Task 2: Writing an Algorithm

- **Problem**: computing the GC-Content in a given DNA seq.
- **GC-content** (or guanine-cytosine **content**) is the percentage of bases in a DNA (or RNA) molecule that are either guanine (G) or cytosine (C). Recall that a strand of DNA is a sequence of A (adenine), T (thymine), C, and G's. (RNA has U (uracil) instead of T (thymine)).
- E.g.,
  GC-content of ACCGC = 4/5=0.8
  GC-content of ATACTAAA = 1/8=0.125
- **Basic instructions needed**: access element in a list, compare the element in a list to G/C, addition and division

GC-Content

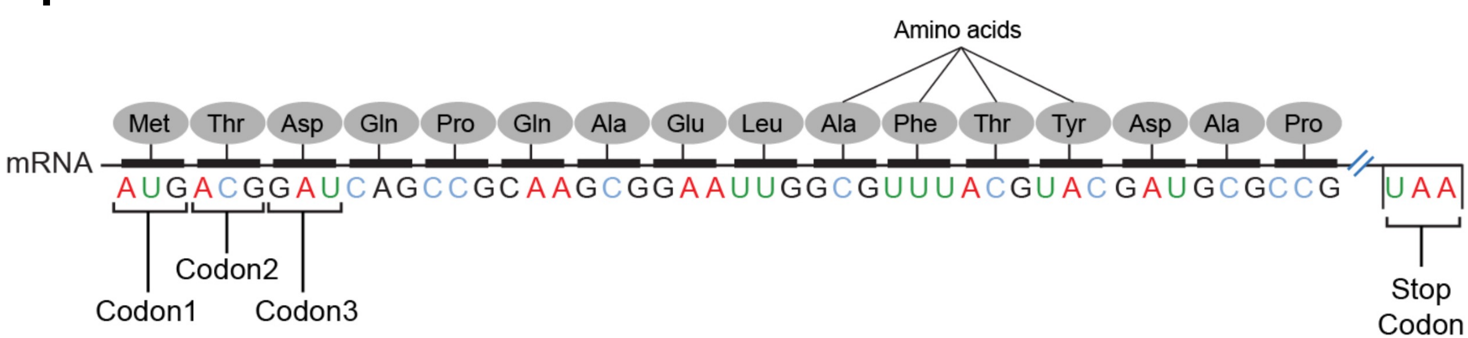
```
GC-Content
input : liste L with n DNA bases
output : percentage of G and C

countGC ⟵ 0
for i from 1 to n
    if L[i] = G or L[i] = C
        countGC ⟵ countGC + 1

return : countGC/n
```

Find Stop Codon

- A **codon** is a sequence of three DNA or RNA nucleotides that corresponds to a specific amino acid or a stop signal during protein synthesis.
- Of the 64 **codons**, 61 represent amino acids, and three are stop signals. For example, the RNA **codon** UAA is a stop **codon**.

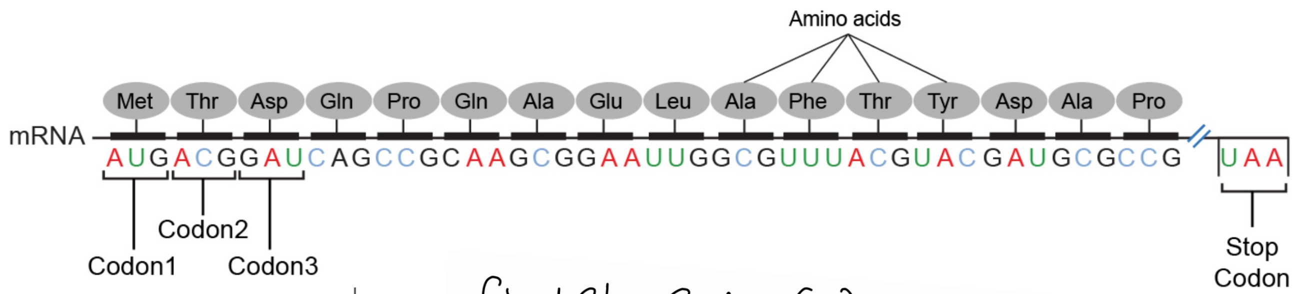


- **Problem**: Given a RNA sequence of length n, find the start position of a **UAA stop codon**.

FindUAACodon

input : *liste L with n RNA bases (containing at least one UAA Codon)*
output : *position in the list at which a UAA Codon starts*

```
pos ⟵ 1
found ⟵ “”
while pos <= n and found ≠ “UAA”
    if found = “”
        if L[pos] = U
            found ⟵ “U”
            pos ⟵ pos + 1
        else
            pos ⟵ pos + 3
    if found = “U”
        if L[pos] = A
            found ⟵ “UA”
            pos ⟵ pos + 1
        else
            found ⟵ “”
            pos ⟵ pos + 2
    if found = “UA”
        if L[pos] = A
            found ⟵ “UAA”
            pos ⟵ pos − 2
        else
            found ⟵ “”
            pos ⟵ pos + 1
return : pos
```



findStopCodon (L)

input : list L with n bases
output : position of first UAA Codon or -1

```
pos ← 1
found ← 0
while ( pos ≤ n-2 & found = 0 )
    if ( L[pos : pos+2] = {U, A, A} )
        found ← 1
    else
        pos ← pos + 3
if ( found = 0 )
    pos ← -1
return pos
```

Task 3: Analysis of an Algorithm

- Key Questions about an Algorithm
  - Is it correct?
  - Is it efficient?

Correctness

1. Does the algorithm terminate for all input?
2. Does it give us the result we aim for?
   - In the common case
   - In corner cases (e.g., empty list, value 0)
   - In all cases

Strategies to analyze correctness:

- Testing, i.e., “run” (with a computer or by hand) the algorithm with different inputs and compare the output with the expected output
- Mathematical reasoning (CS-550)

Be Aware of the Corner Cases!

- What happens if n=0?

GC-Content

input : *liste L with n DNA bases*
output : *percentage of G and C*

$countGC \longleftarrow 0$
for i *from* 1 *to* n
  if $L[i] = G$ *or* $L[i] = C$
    $countGC \longleftarrow countGC + 1$

return : $countGC/n$

```
#include <iostream>
using namespace std;

double gc_content(string dna) {
    int n(dna.length());

    double countGC(0);
    for (int i = 0; i < n; i++) {
        if (dna[i] == 'G' || dna[i] == 'C') {
            countGC = countGC + 1;
        }
    }
    return countGC/n;
}


int main()
{
    cout << gc_content("GCATATGCAATTTAGC") << endl;
    cout << gc_content("") << endl;
}
```

```
~/src > ./gc_content
0.375
-2147483648
```

Complexity of an Algorithm

- How much time and space (memory) does it need? We focus on **time complexity. (**Number of **elementary instructions)**
- In general the complexity is always given in terms of the total input size, e.g., if an algorithm has two inputs x, y, the complexity is a function of the size of x and y. We focus on **one input**.
- Actually running time can depend a lot on specific input: best, average worst-case behavior. We focus on **worst-case behavior**.
- Finally, for small input values usually all algorithms are fast. We are interested in the complexity on large inputs, the complexity when the size of the input goes towards infinity, called the **asymptotic complexity (Big Theta or Big O notation)**

Cf. document about complexity on Moodle

Elementary Instructions

- The runtime of an algorithm is computed in **terms of instructions** that can be performed in **constant time**. We assume that the following instructions can be performed in constant time:
  - Assignments (Affectation)
  - Mathematical operators (+,-,*,/,%, mod) using numbers with a **fixed** number of bits (e.g., 32 or 64)
  - Comparison operators (>, >=, <, <=, ==)
  - Bitwise operator (&, |, and, or,..)
  - Access an element (e.g., value or sub-list) in a list
- **Question to answer**: How many of these instructions does an algorithm use (or ”read”)?

Computing the Time Complexity

We aim to estimate how many basic instructions an algorithm executes in terms of its input size.

1. Compute the number of instructions per line
2. Compute the number of repetitions per line
3. Sum costs of all lines
4. Approximate functions

Complexity: Step 1

- Compute the number of instructions per line

GC-Content

input : *liste L with n DNA bases*
output : *percentage of G and C*

$countGC \longleftarrow 0$
for i *from* 1 *to* n
  if $L[i] = G$ *or* $L[i] = C$
    $countGC \longleftarrow countGC + 1$
return : $countGC/n$

Line	Instructions	Cost
1	1 assignment	1
2	1 assignments, 1 comparison, (1 addition)	3
3	2 list access, 3 comparison, 1 bitwise op.	6
4	1 addition, 1 assignment	2
5	1 division, 1 return	2

The precise costs per line (e.g., value 2, 3 or 5) are not important because we will approximate the overall complexity later.
Important is to know if the cost is constant or not, e.g., a line in which another algorithm is called.

Complexity: Step 2

- Compute how many times a line is execute or "read" (in the worst case)

GC-Content

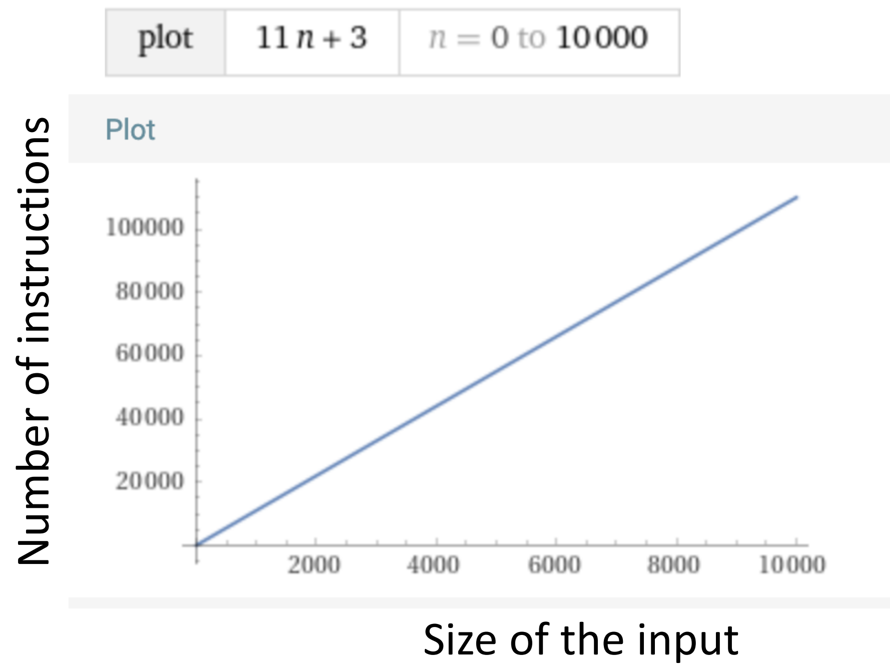
input : *liste L with n DNA bases*
output : *percentage of G and C*

$countGC \longleftarrow 0$
for i *from* 1 *to* n
    if $L[i] = G$ *or* $L[i] = C$
        $countGC \longleftarrow countGC + 1$
return : $countGC/n$

Line	Cost	Repetitions
1	1	1
2	3	n
3	6	n
4	2	n
5	2	1

Complexity: Step 3

Line	Cost	Repetitions
1	1	1
2	3	n
3	6	n
4	2	n
5	2	1



- Sum the costs · repetitions over all lines:

$$\sum_{i=1}^{5} c(i) \cdot r(i) = 1 + 3n + 6n + 2n + 2 = 11n + 3$$

Complexity: Step 4

- We don't care about the precise number of instructions but the general behavior.
- It is important to know how complexity **evolves according to the input size.**
- We aim to know if the number of instructions (aka runtime) is linear, quadratic, … or exponential in terms of the size of the input.
- We use the **Landau notations (also called Big O, Big Omega, or Big Theta notations)** to obtain this information.

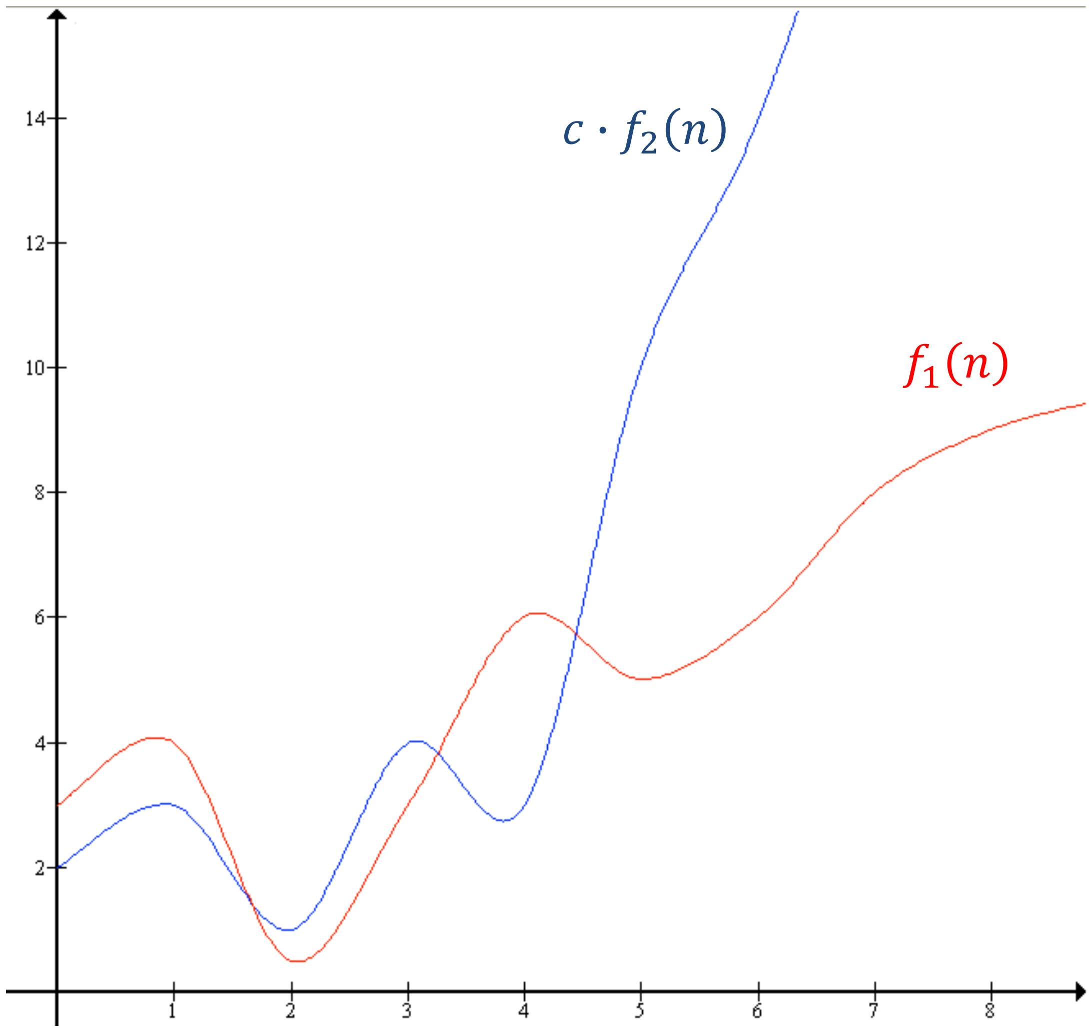
Landau Notation: O(...)

Big O notation is a mathematical notation that describes the **limiting behavior of a function** when the argument tends **towards infinity**.

For two functions f and g from $\mathbb{R}$ to $\mathbb{R}$,
$f \in O(g)$ (or $f = O(g)$) if and only if
$\exists c > 0 \; \exists n_0 \; \forall n > n_0 : |f(n)| \leq c \cdot g(n)$

In this case, we say that the function f is in "Big O" of g.
This means that f **grows asymptotically no faster than g**.
The function $c \cdot g$ is an **asymptotic upper bound** for f.

Example Big O



For two functions f and g from $\mathbb{R}$ to $\mathbb{R}$,
$f \in O(g)$ (or $f = O(g)$) if and only if
$\exists c > 0\ \exists n_0\ \forall\, n > n_0 : |\, f(n)| \leq c \cdot g(n)$

$f_1(n) \in O\big(f_2(n)\big)$
because there exists a c and n_0,
such that $f_1(n) \leq c \cdot f_2(n)$

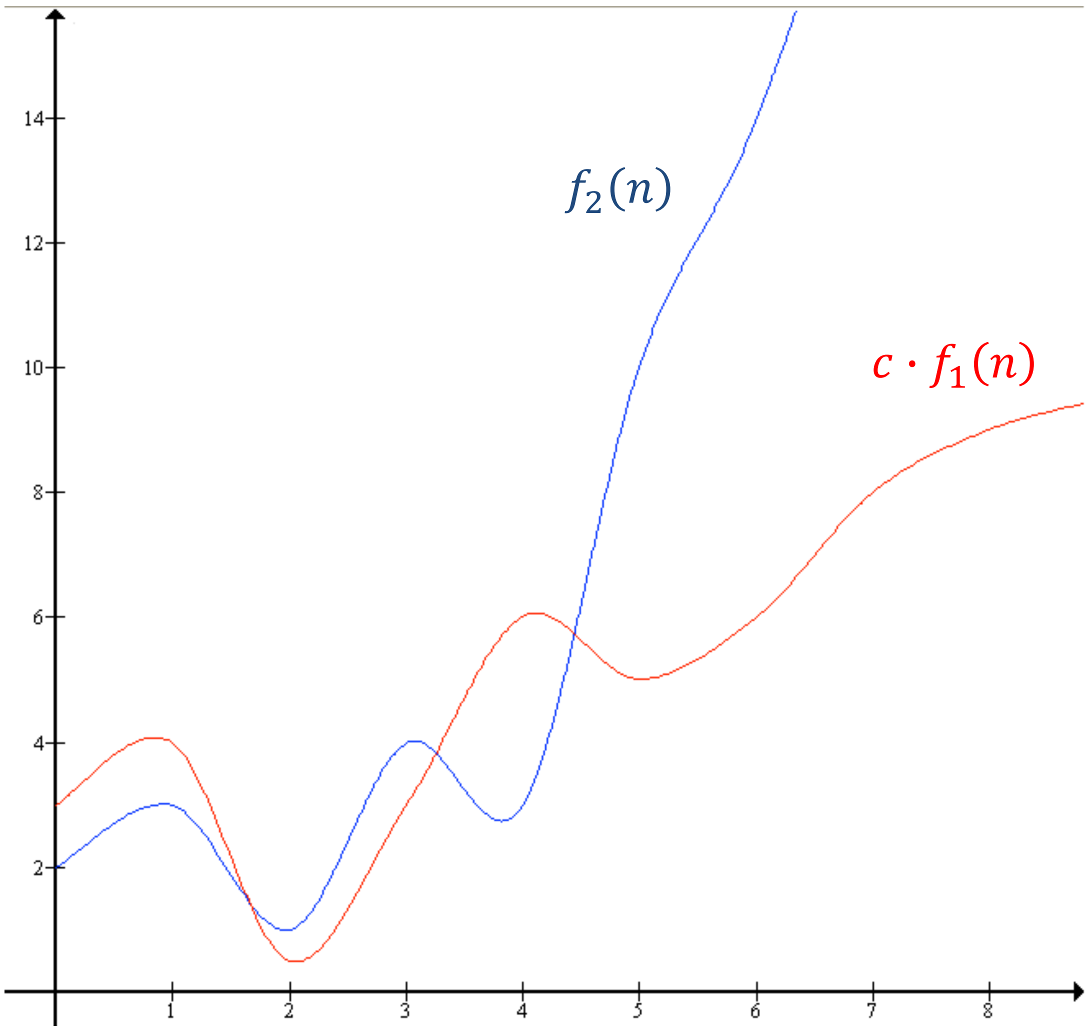
Landau Notation: Ω (...)

Big Ω notation is a mathematical notation that describes the **limiting behavior of a function** when the argument tends **towards infinity**.

For two functions f and g from $\mathbb{R}$ to $\mathbb{R}$,
$f \in \Omega(g)$ (or $f = \Omega(g)$) if and only if
$\exists c > 0 \; \exists n_0 \; \forall n > n_0 : c \cdot g(n) \leq | f(n) |$

In this case, we say that the function f is in "Big Omega" of g.
This means that f **grows asymptotically not slower than g**.
The function $c \cdot g$ is an **asymptotic lower bound** for f.

Example Big Omega



For two functions f and g from $\mathbb{R}$ to $\mathbb{R}$,
$f \in \Omega(g)$ (or $f = \Omega(g)$) if and only if
$\exists c > 0 \; \exists n_0 \; \forall n > n_0 : c \cdot g(n) \leq | f(n) |$

$f_2(n) \in \Omega\big(f_1(n)\big)$
because there exists a c and n_0,
such that $c \cdot f_1(n) \leq f_2(n)$

Landau Notation: Θ(...)

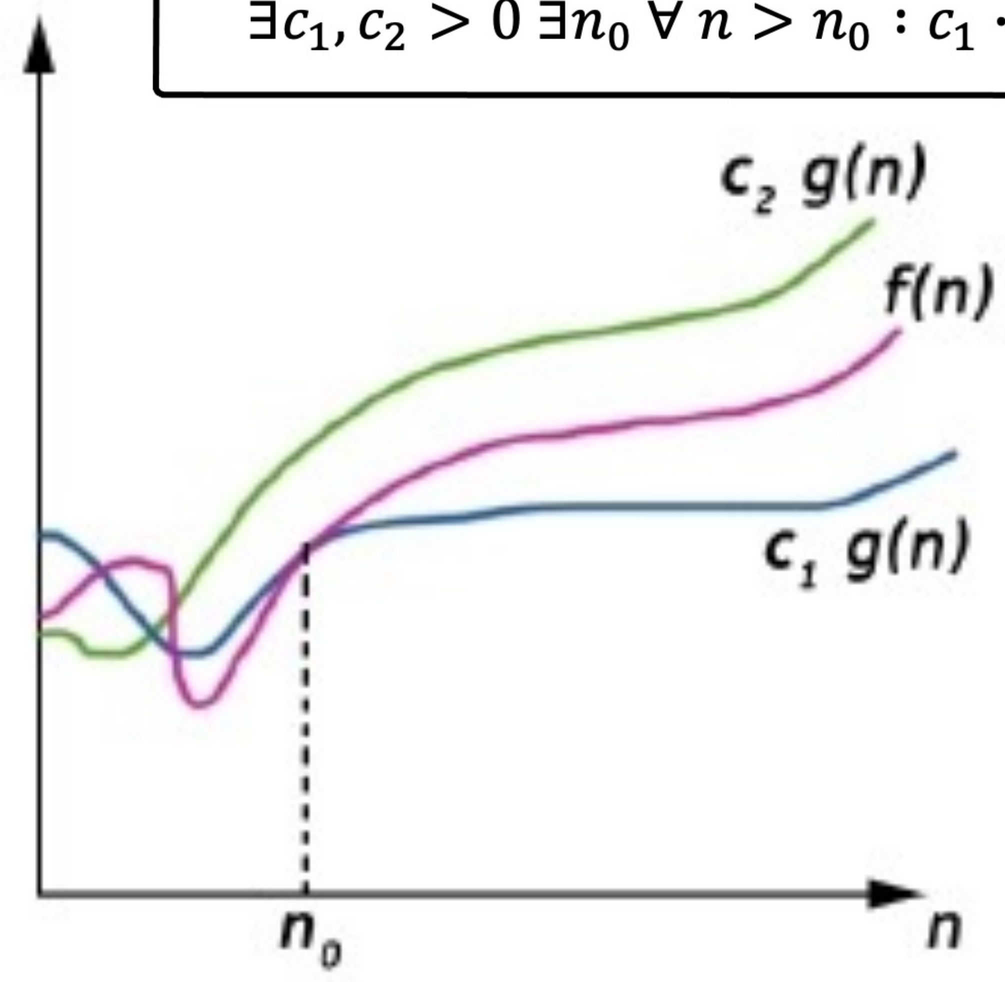
Big Theta notation is a mathematical notation that describes the **limiting behavior of a function** when the argument tends **towards infinity**.

> For two functions f and g from $\mathbb{R}$ to $\mathbb{R}$,
> $f \in \Theta(g)$ (or $f = \Theta(g)$) if and only if
> $\exists c_1, c_2 > 0\ \exists n_0\ \forall\, n > n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

In this case, we say that the function f is in "Big Theta" of g.
This means that f **grows asymptotically the same as g**.
The function $c_1 \cdot g$ is an asymptotic lower bound for f and
$c_2 \cdot g$ is an asymptotic upper bound for f.

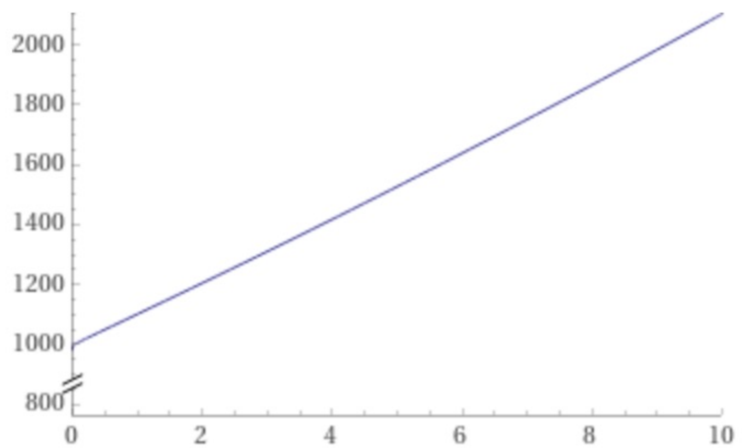
Big Theta

For two functions f and g from $\mathbb{R}$ to $\mathbb{R}$,
$f \in \Theta(g)$ (or $f = \Theta(g)$) if and only if
$\exists c_1, c_2 > 0 \; \exists n_0 \; \forall \, n > n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

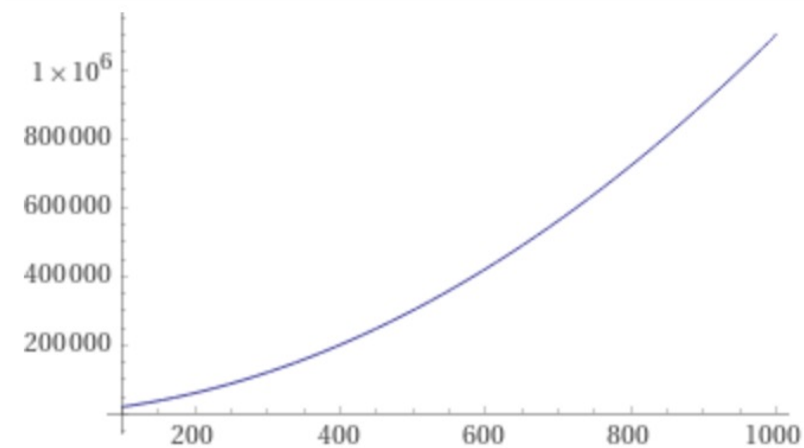


Example of Growth (continue)

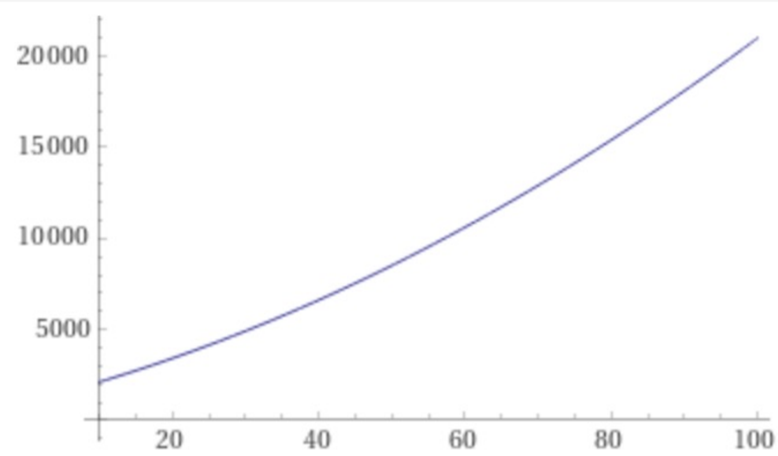
- $f(n) = n^2 + 100n + \log n + 1000$



Small values: behavior is linear



Large values: behavior is quadratic



Example of Growth

$$f(n) = n^2 + 100n + \log n + 1000$$

Table with the contributions of the different terms:

n	f (n)	n^2		100n		log n		1000	
		value	%	value	%	value	%	value	%
1	1’101	1	0.1	100	9.1	0	0.0	1000	90.82
10	2’101	100	4.8	1’000	47.6	1	0.0	1000	47.6
100	21’002	10’000	47.6	10’000	47.6	2	0.0	1000	4.8
1000	1’101’003	10^6	90.8	10^5	9.1	3	0.0	1000	0.1
10’000	101’001’004	**10^8**	**99.0**	10^6	1.0	4	0.0	1000	0.0
…									

$$f(n) \in O(n^2)$$

Functions

- Constant, e.g., $f(n) = 1$
- Logarithmic, e.g., $f(n) = \log(n)$
- Linear, e.g., $f(n) = n$
- “Linearithmic”, e.g., $f(n) = n \cdot \log(n)$
- Quadratic, e.g., $f(n) = n^2$
- Cubic, e.g., $f(n) = n^3$
- Polynomial, e.g., $f(n) = n^5 + n^2 + 10$
- Exponential: $f(n) = 2^n$

Growth rate increase

n=1000
1
3
1’000
3’000
1’000’000
1’000’000’000
$\cong 10^{15}$
$\cong 10^{300}$

Example: Big O Notation

- Bound the asymptotic complexity of the following functions from above and give a proof (i.e., give some constant c and some starting point n_0, s.t. for all $n > n_0$. $f(n) \leq c \cdot g(n)$

Function (f)	Complexity (g)	Constant c	Starting point n_0
2^{2+n}	$O(2^n)$	$2^{2+n} \leq c \cdot 2^n$?	
$3n^2 + 2^n$	$O(2^n)$	$3n^2 + 2^n \leq c \cdot 2^n$?	
$2n^2 + n^3 + 100$	$O(n^3)$	$2n^2 + n^3 + 100 \leq c \cdot n^3$?	
$10n^2 + n\log_2 n$	$O(n^2)$	$10n^2 + n\log_2 n \leq c \cdot n^2$?	
$\log_2(2n)$	$O(\log_2 n)$	$\log_2(2n) \leq c \cdot \log_2(n)$?	

Note that we do not need to find the smallest c or n_0.

Big O Notation

- Bound the asymptotic complexity of the following functions from above and give a proof (i.e., give some constant c and some starting point n_0)

Function	Compl.	Constant c	Starting point n_0
2^{2+n}	$O(2^n)$	$2^{2+n} = 2^2 \cdot 2^n \leq c \cdot 2^n$, $c = 2^2$	$n_0 \geq 0$
$3n^2 + 2^n$	$O(2^n)$	$c = 4$ $3 \cdot n^2 + 2^n \leq 3 \cdot 2^n + 2^n = 4 \cdot 2^n$	When is $3 \cdot n^2 \leq 3 \cdot 2^n$? $n_0 \geq 2$
$2n^2 + n^3 + 100$	$O(n^3)$	$c = 4$ $2 \cdot n^2 + n^3 + 100 \leq 2 \cdot n^3 + n^3 + n^3$	When is $2n^2 \leq 2n^3$ and $100 \leq n^3$? $n_0 \geq 5$
$10n^2 + n \log_2 n$	$O(n^2)$	$c = 11$ $10 \cdot n^2 + n \log_2 n \leq 10 \cdot n^2 + n^2$	When is $n \log_2 n \leq n^2$? $n_0 \geq 1$
$\log_2(2n)$	$O(\log_2 n)$	$c = 2$ $\log_2(2) + \log_2(n) \leq \log_2(n) + \log_2(n)$	$n_0 \geq 2$

Big-Theta Notation

- Given the asymptotic complexity of the following functions and a proof (i.e., give values c_1, c_2, and n_0 such that for all $n > n_0.\ c_1 \cdot g(n) \leq f(n)$ and for all $n > n_0.\ f(n) \leq c_2 \cdot g(n)$

Function (f)	Complexity (g)	c_1	n_0	c_2	$\hat{n}_0$
2^{2+n}					
$3n^2 + 2^n$					
$2n^2 + n^3 + 100$					
$10n^2 + n \log_2 n$					
$\log_2(2n)$					

Big-Theta Notation

- Given the asymptotic complexity of the following functions and a proof (i.e., give values c_1, c_2, and n_0 such that for all $n > n_0.\ c_1 \cdot g(n) \leq f(n)$ and for all $n > n_0.\ f(n) \quad \leq c_2 \cdot g(n)$

Function (f)	Complexity (g)	c_1	$\hat{n}_0$	c_2	n_0
2^{2+n}	$\Theta(2^n)$	$c_1 = 1$ $c_1 \cdot 2^n \leq 2^{2+n}$	$\hat{n}_0 \geq 0$	See Big-O	$\max(0{,}0)$
$3n^2 + 2^n$	$\Theta(2^n)$	$c_1 = 1$	$\hat{n}_0 \geq 0$	See Big-O	$\max(2{,}0)$
$2n^2 + n^3 + 100$	$\Theta(n^3)$	$c_1 = 1$	$\hat{n}_0 \geq 0$	See Big-O	$\max(5{,}0)$
$10n^2 + n\log_2 n$	$\Theta(n^2)$	$c_1 = 1$	$\hat{n}_0 \geq 1$	See Big-O	$\max(1{,}1)$
$\log_2(2n)$	$\Theta(\log_2 n)$	$c_1 = 1$	$\hat{n}_0 \geq 1$	See Big-O	$\max(2{,}1)$

Comparison of Algorithms

Examples:

- $f(n)=n$ is $O(n^2)$ but it is also $O(n)$
- $f(n)=12$ is $O(n^2)$, $O(n)$, but especially $O(1)$
- $f(n)=n^2 + n + 10$ is in $O(n^3)$ but not in $\Theta(n^3)$
- $f(n)= 5n \cdot \log n + n$ is in $\Theta(n \cdot \log n)$ and in $O(n^2)$ but not in $\Theta(n^2)$

Complexity: Step 4

- Approximate with Big O/Theta notation

GC-Content

input : *liste L with n DNA bases*
output : *percentage of G and C*

$countGC \longleftarrow 0$
for i *from* 1 *to* n
    if $L[i] = G$ *or* $L[i] = C$
        $countGC \longleftarrow countGC + 1$
return : $countGC/n$

Line	Cost	Repetitions
1	1	1
2	3	n
3	6	n
4	2	n
5	2	1

Recall: $\exists c > 0 \exists x_0 \forall x > x_0 |f(x)| \leq c \cdot |g(x)|$

$f(n) = 11 \cdot n + 3$ with $c = 12$ and $n_0 = 1$
$\forall n > 1 : 11 \cdot n + 3 \leq 12 \cdot n$
$\boldsymbol{f(n)}$ **is in** $\boldsymbol{O(n)}$

Which of the following statements is correct?

A. $10 \cdot n^3 + 5 \cdot n$ is in $O(n^4)$ — Correct

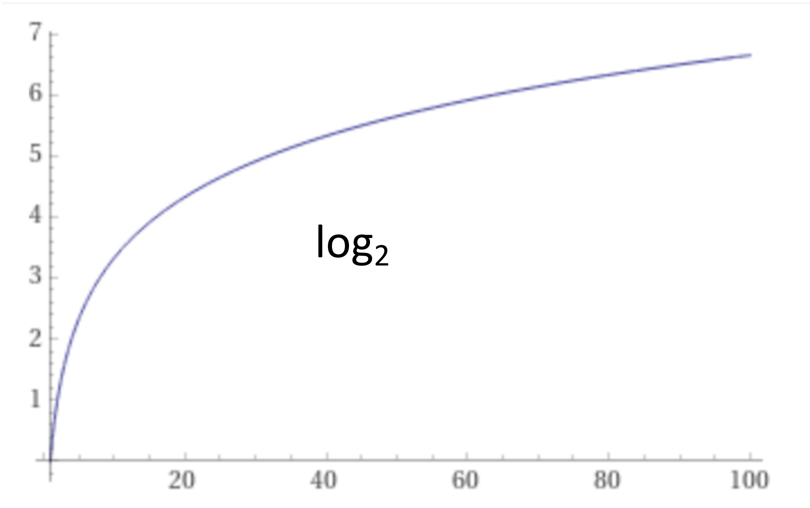
B. $2 \cdot n^2$ is in $\Theta(n^3)$ — Incorrect

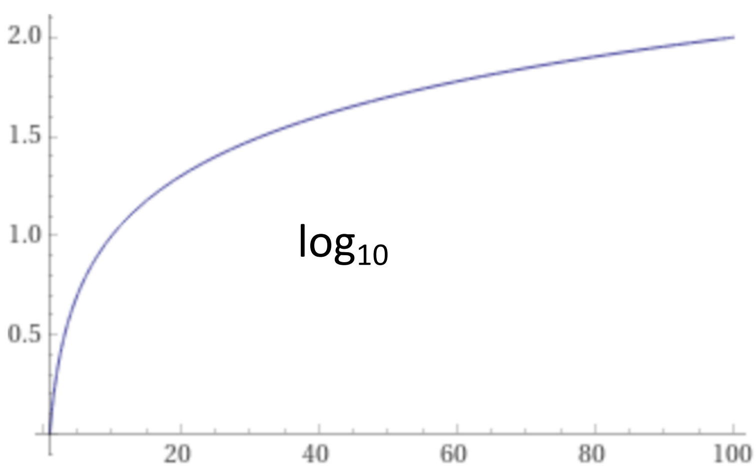
C. 10^{10} is in $\Theta(1)$ — Correct

D. $10 + 2n + n$ is in $O(n)$ — Correct

E. $2 \cdot log_2(n)$ is in $\Theta\big(log_{10}(n)\big)$ — Correct (see next slide)

Logarithms





$$log_a(n) = \frac{log_b(n)}{log_b(a)} = \frac{1}{log_b(a)} log_b(n)$$

$$log_{10}(n) = \frac{1}{log_2(10)} log_2(n)$$

What is the complexity of this algorithm?

Algorithm1

input : *a list L with n numbers*
output : *a number s*

$s \longleftarrow 0$
$i \longleftarrow 1$
while $i < n$
  $s \longleftarrow s + L[i]$
  $i \longleftarrow 2 \cdot i$
return : s

A. Θ(1)
B. Θ(log(n))
C. Θ(n)
D. Θ(n^2)

Correct, because in iteration k of the loop, i=2^k and at the end of the loop I > n, so when is 2^k > n?
If k > log(n) and we only have log(n) iterations of the loop.

What is the complexity of this algorithm?

Algorithm2

input : *a list L with n numbers*
output : *a number s*

$s \longleftarrow 0$
$i \longleftarrow 1$
while $i < n$
  for j *from* 1 *to* n
    $s \longleftarrow s + L[j]$
  $i \longleftarrow 2 \cdot i$
return : s

A. $\Theta(\log(n))$

B. $\Theta(n)$

C. $\Theta(n \cdot \log(n))$

D. $\Theta(n^2)$

log(n) iterations of the loop with n instructions per iteration.

Other Examples

- Example in videos:
  - Check if a set of objects has two identical objects
  - Greatest Common Divisor (Euclid)
  - Insertion sort

Sub-Algorithm: Insertion Sort

insertion_sort

input : *a list of numbers L with n elements*
output : *the list L in which all elements are sorted in increasing order*

for i *from* 2 *to* n
  if $L[i] < L[i-1]$
    $L \longleftarrow$ ***insert_element***(L,i)
return : L

insert_element

input : *a list of numbers L with n elements and an index i*
output : *the list L in which the element $L[i]$ is on the right place*

$j \longleftarrow i$
while $L[j] < L[j-1]$ *and* $j > 1$
  $L \longleftarrow$ ***permut***$(L,j,j\text{-}1)$
  $j \longleftarrow j-1$
return : L

permut

input : *a list of numbers L with n elements and two indices j and k*
output : *the list L in which the elements $L[j]$ and $L[k]$ are permuted*

$tmp \longleftarrow L[j]$
$L[j] \longleftarrow L[k]$
$L[k] \longleftarrow tmp$
return : L