S3 Program

- **Reminders**
- **Chapter 1 : Algorithmic Complexity**
- Chapitre 2 : Sorting Algorithms
- Chapitre 3 : Trees
- Chapitre 4 : Graphs
- NB. TP with C/C++.

Chapter I :

Algorithmic Complexity

<u>What is an algorithm ?</u>

$\mathcal{P}$: a problem

$\mathcal{M}$: a method to solve problem P

Algorithme : description of the method M in an algorithmic language

Al Khuwarizmi (780 - 850) = Muslim mathematician of Persian origin.

Structures algorithmiques

Control Structures

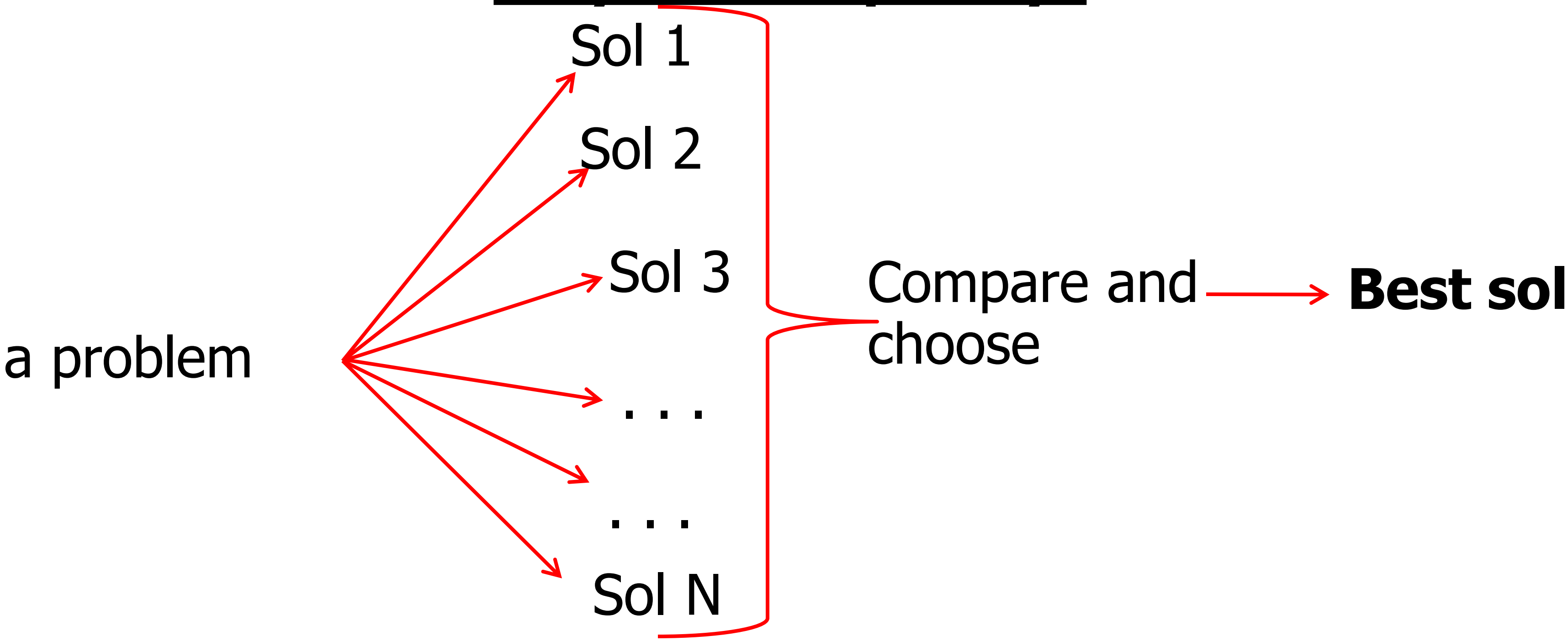
- sequence
- Test or condition (ou selection)
- loop (ou iteration)

Data Structures

- constants
- variables
- arrays

- recursive structures (lists, trees, graphs)

Why the complexity?



Best Sol is the most **efficient algorithm.**

Efficiency = <u>Less</u> execution time, <u>less</u> memory space.

Complexity Objective

Estimate the "Execution Time" to compare multiple algorithms for the same problem.

Measuring time complexity

Definition : An operation OP is fundamental (elementary) for an algorithm A if the number of OP influences directly the execution time of algorithm A.

Examples of fundamental operations

Algorithm	Fondamental Operations
Searching for an element in a list in central memory	comparison between the searched element and the components of the list
Sorting a list of items	- comparison between two elements - moving elements
Multiply two matrices of numbers	- Multiplication - Addition

Complexity of instruction sequence

S = {

i_1 ; i_2 ; . . . ; i_n;

}

so $$Nb(S) = \sum_{p=1}^{n} Nb(i_p)$$

Nb (i_k) = the number of elementary operations in the instruction i_k.

Complexity of a conditional statement

Cond = if (Expr) E_1 ; else E_2 ;

so Nb(Cond) $\leq$ Nb(Expr) +Max (Nb(E_1) , Nb(E_2))

Complexity of a bounded finite loop

Iter = Iteration Expr(i)
S
IterEnd

so : Nb(Iter) = [Nb(S) + Nb (Expr(i))] x iterations_Nb

For example in the case of a For loop:

```
Iter =  for (int i = a; i<=b; i++)
         {
            i1 ; i2 ; . . . ;  in ;}
```

Complexity of increm ent i++ and condition i<=b

Donc :

$$Nb(Iter) = \left(\sum_{p=1}^{n} Nb(i_p) + 2 \right) x \left(|b - a| + 1 \right)$$

Complexity of subroutine calls:

Without recursion:

$$Nb(A)=Nb(I_1)+Nb(I_2)+Nb(B)+Nb(I_3)$$

Sous-pgme A

I_1;
I_2;
Call of B;
I_3;

In the case of recursion, calculating Nb(A) results in the resolution of a recurrence relation.

Example:

```
  int fact(int n ) {
        if (n<=1) return 1
           else return n*fact(n-1); }
```

The number T(n) of elementary operations (*) verifies:
$T(0)=0$ et $T(n)=T(n-1)+1$; for $n\geq 1$

By direct resolution we obtain: $T(n)=n$.

<u>Examples :</u>

<u>1) Algorithm for permuting an element X in a list S</u>

```
void permut (Element  S[n] ,  int i, int j)
{          Element tmp=S[i];        c1
           S[i]=S[j];               c2
           S[j]=tmp;                c3   }
```

T(n)= c1+c2+c3=O(1)

<u>2) Algo for the product of 2 vectors</u>

```
int prod(int A[n], int B[n]) {
int P=0;
for (int i=0; i<n; i++) P+=A[i]*B[i];
return P;}
```

T(n)= 1+1+4n+1=3+O(4n)=O(n)

Examples :

3) Algorithm for sequential search of an element X in a list L.

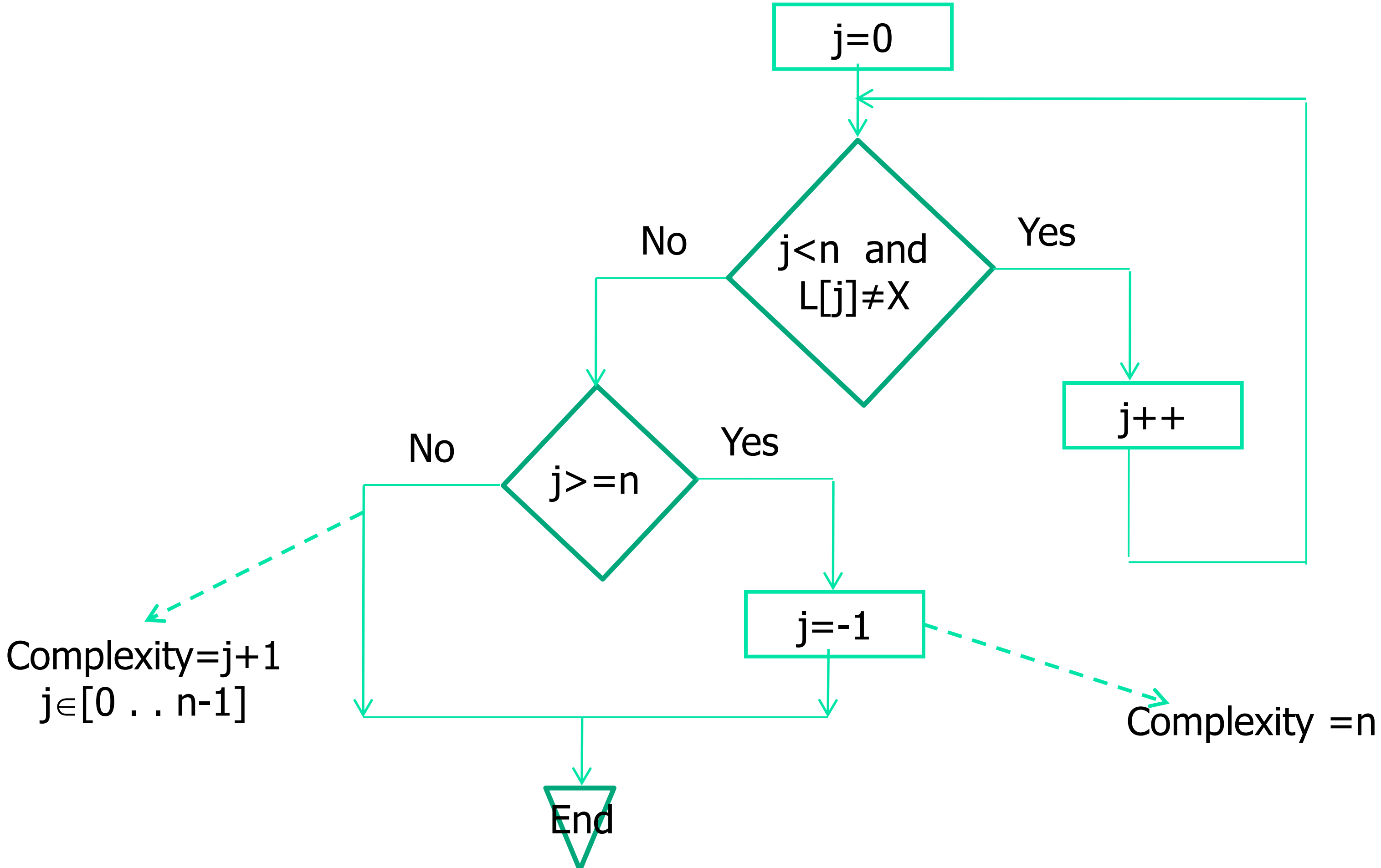
```
int Search(Element L[n] , Element X)
{
(1)     int j=0;
(2)     while (j<n and L[j]!=X)    j++;
(3)     if (j>=n) j=-1;
        return j;
}
```

The complexity of this algorithm = Function (number of iterations, number of operations per iteration).
max number of iterations = n

3 candidate fundamental operations in line (2).

Fundamental operation to retain = comparison (L[j]!=x) in line (2).

The other 2 operations depend on the programming.



<u>Conclusion :</u>

The complexity (number of comparisons L[j] ≠ x) depends on the data size.

Worst-case complexity = n

Best-case complexity = 1

Average complexity between 1 and n

Average and worst case complexity:

Let $cost_A(d)$ =complexity of algorithm A for data d.

The set of all data of size n is denoted D_n.

Complexity at the best case

$$Min_A(n) = \min \{cost_A(d) \; ; \; d \in D_n\}$$

Complexity at the worst case

$$Max_A(n) = \max \{cost_A(d) \; ; \; d \in D_n\}$$

Average complexity

$$Moy_A(n) = \sum_{d \in D_n} P(d) \times cost_A(d)$$

where P(d) is the probability of having data *d* as input to algorithm *A*

Note :

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n)$$

DT 01 Exercises

Exercise 01: Let the real matrices n x n be : A=(a_{ij}), B=(b_{ij}) and C=(c_{ij}). Study the complexity of the following algorithm which calculates the coefficients (cij) of the product matrix C= A x B according to the classical formula:

$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj} \quad \text{for i, j between 1 and n}$$

```
Const int n=8;
typedef float matrice[n][n];

void multimat(matrice a, matrice b, matrice c)
{for (int i=0; i<n; i++)
                    for (int j=0; j<n; j++)
                            { c[i][j]=0;
                                      for (int k=0; k<n; k++)
                                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
                            }
}
```

Exercise 02: Let the algorithm that calculates the value of the polynomial at a given point x : $P(x,n) = \sum_{i=0}^{n} a_i x^i$

We have 3 versions of this algorithm.

a)
```
p=a[0];
for (i=1; i<=n;i++){        xpi=puissance(x, i);
                            p=p+a[i]*xpi;}
```

b)
```
p=a[0]; xpi=1;
for (i=1; i<=n;i++){        xpi=xpi*x;
                            p=p+a[i]*xpi;}
```

c) Horner's method
```
p=a[n];
for (i=n-1; i>=0;i--){      p=p*x+a[i];
```

Calculate the complexity of each version.

Exercise 03: Calculate the complexity of the following 2 algorithms for the Fibonacci sequence.

a)
```
int Fib (int n)
{ int tab[n+1];
tab[0]=1;     tab[1]=1;
for (int i=2; i<=n; i++) tab[i]=tab[i-1]+tab[i-2];
return tab[n];}
```

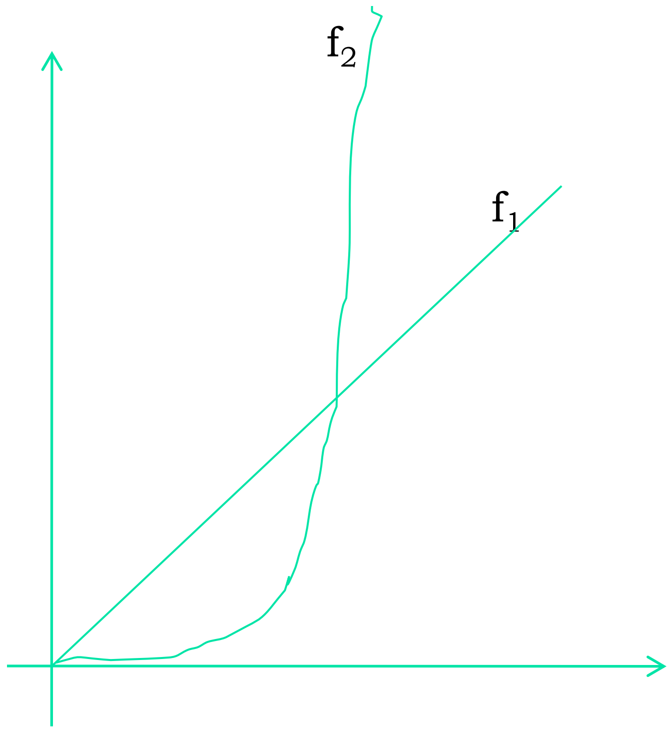
b)
```
int Fib (int n)
{ int tab[2];
tab[0]=1;     tab[1]=1;
for (int i=1; i<=n/2; i++) {tab[0]=tab[0]+tab[1];
                            tab[1]=tab[0]+tab[1];
if (pair(n)) return tab[0]; else return tab[1];}
```

Magnitude Order :

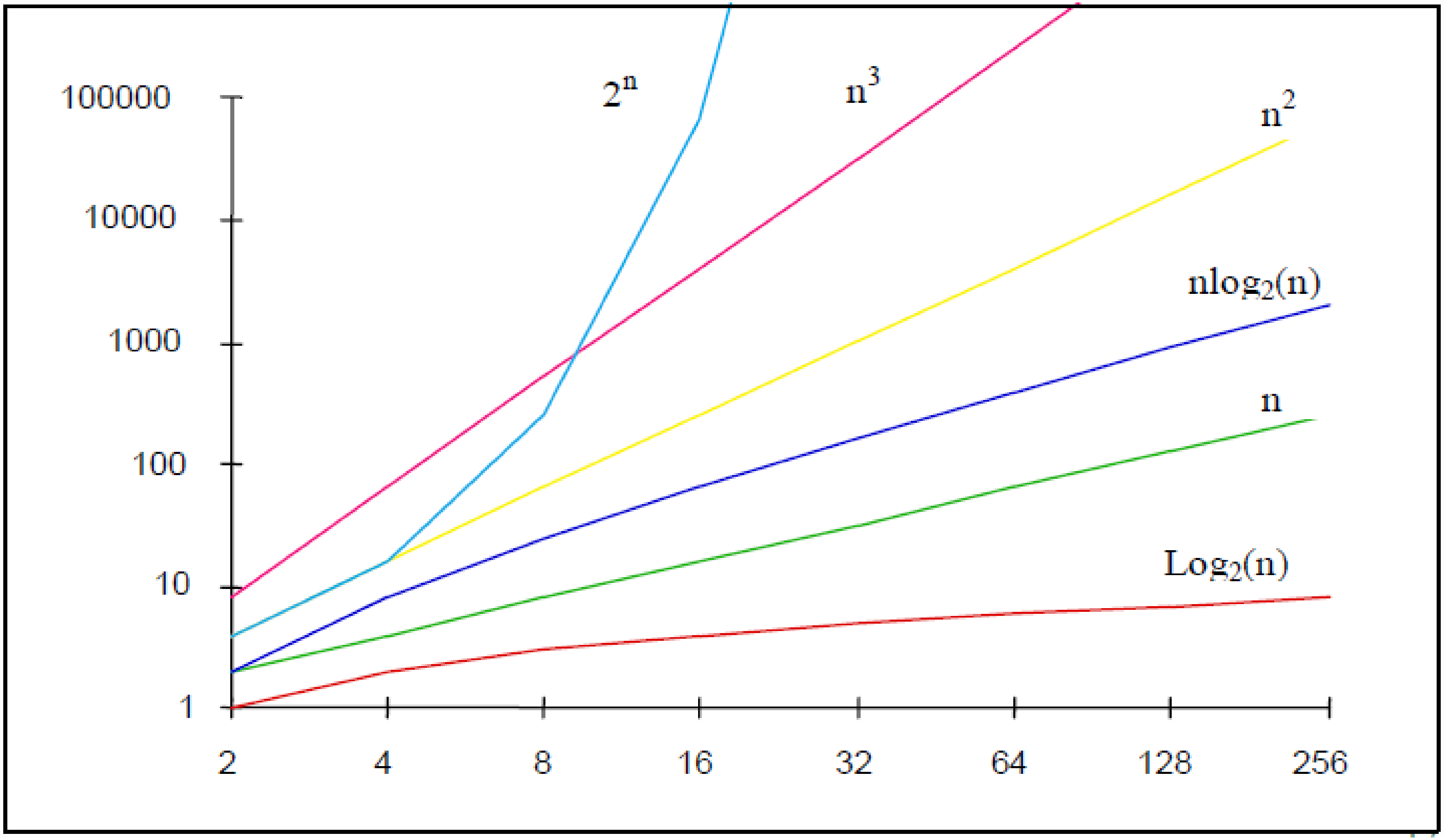
We want to compare algorithm A_1 with complexity $f_1(n)=2n$ and A_2 with complexity $f_2(n)=n^2$.

A_1 is better than A_2 for $n>2$, because $f_2(n)=n^2$ grows faster than $f_1(n)=2n$

since $\lim_{n\to\infty}\left(f_1(n)/f_2(n)\right)=0$



The asymptotic order of magnitude of $f_2(n)=n^2$ is larger than that of $f_1(n)=2n$.



The functions: $log_2 n$, n, $nlog_2 n$, n^2, n^3, 2^n form a comparaison scale

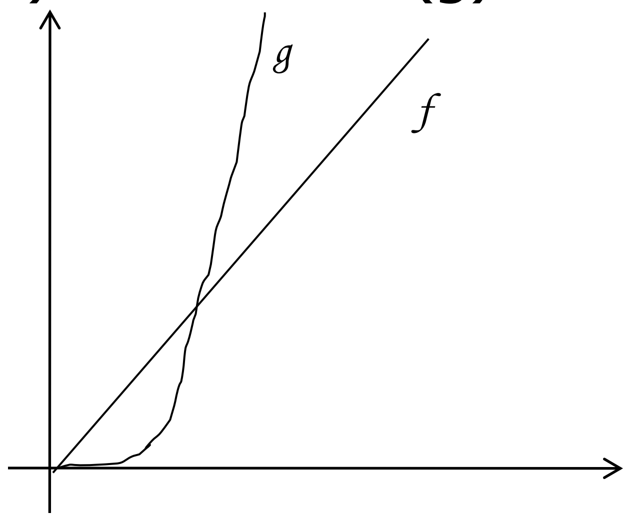
Definition 1 : Landou Notation

Let f and g be two functions from N to R^+

f = **O**(g) IOI $\exists$ c $\in R^{+,*}$, $\exists$ $n_0 \in$ N *such that* $\forall$ n > n_0, f(n) ≤ c.g(n)

Example:

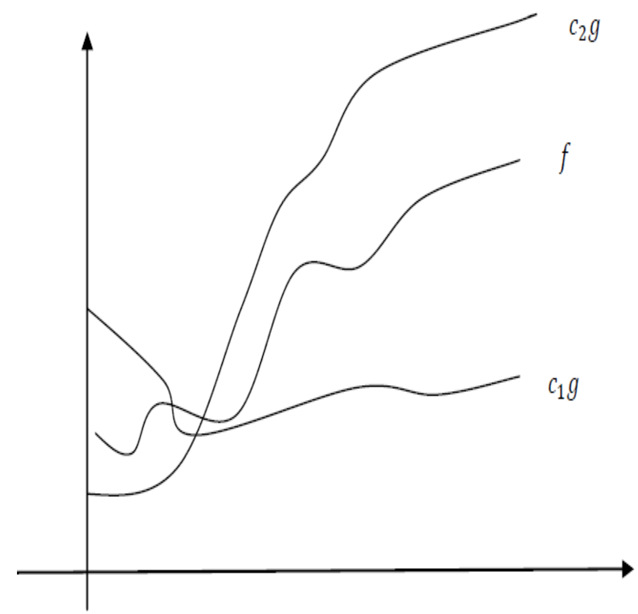
1) f(n)=2n   g(n)=n^2 f=O(g) because f(n) ≤ g(n) for n >1



Définition 2 :

f=θ(g) IOI f =**O**(g) and g=**O**(f) i.e.

$\forall$n >n_0, c_1.g(n) ≤ f(n)≤ c_2.g(n)



Example: $\frac{1}{2}n^2 - 3n = \theta(n^2)$

Table A : Estimation of the execution time of some algorithms for different data sizes n for a problem on a computer performing 10^6 operations per second. We see that, the larger the data sizes, the greater the gaps between the different execution times.

Complexité / Taille	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n=10^2$	≈1μs	6,6μs	0,1ms	0,6ms	10ms	1s	$4x10^{16}$ a
$n=10^3$	≈1μs	9,9μs	1ms	9,9ms	1s	16,6 mn	$\infty(>10^{100})$
$n=10^4$	≈1μs	13,3μs	10ms	0,1s	100s	11,5j	∞
$n=10^5$	≈1μs	16,6μs	0,1s	1,6s	2,7h	31,7a	∞
$n=10^6$	≈1μs	19,9μs	1s	19,9s	11,5j	$31,7x10^3$a	∞

A value greater than 10^{100} is denoted by ∞.
The algorithms that can be used are those that run in:

Constant, Logarithmic, Linear or n log n ;

Table B : Estimation of the maximum data size processed by certain algorithms in a fixed execution time on a computer performing 10^6 operations per second.

Infinite data processed/second

19 data processed/second

Complexity / Calculation time	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 day	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

Infinite data processed/day

36 data processed/day

Table C : Mutual evolutions of time and size

Complexity	1	Log $_2$n	n	nlog$_2$ n	n^2	n^3	2^n
Time Evolution when size is multiplied by 10	t	t+3,32	10xt	(10+ε)xt	100xt	1000xt	t^{10}
Size evolution when time is multiplied by 10	∞	n^{10}	10xn	(10-ε)xn	31,6xn	2,15xn	n+3,32

Exponential size

Size practically unchanged

Exponential time

Time practically unchanged

Therefore, it is always useful to look for efficient algorithms, even if technological advances increase hardware performance.