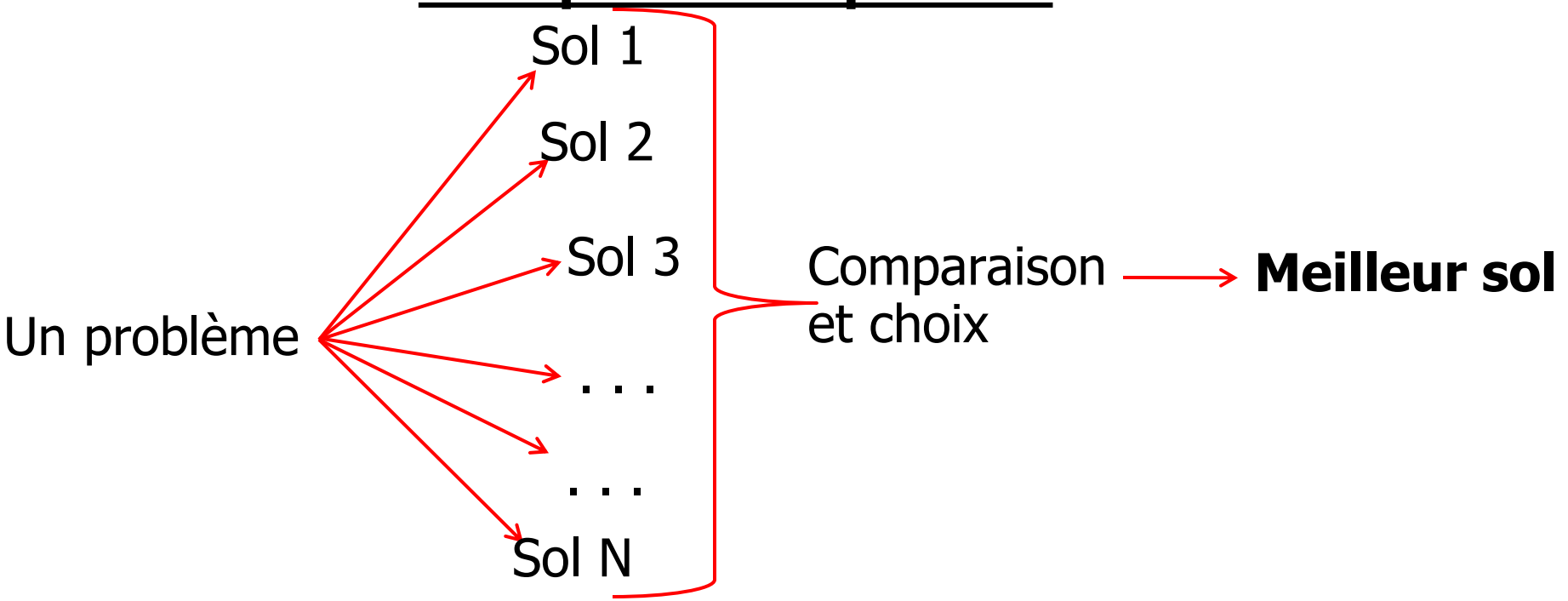
Chapitre II :

Analyse des algorithmes et

Complexité

Pourquoi la complexité?



Meilleur sol est l'algorithme le plus efficace.

L'efficacité= MOINS de temps d'exécution, moins de place mémoire.

Objectif

Estimer le «Temps d'exécution» pour comparer plusieurs algorithmes pour le même problème.

Mesure de la complexité en temps

Définition : Une opération **OP** est **fondamentale** pour un algorithme A si le nombre de OP influe directement sur le temps d'exécution de l'algorithme A.

Exemples d'opérations fondamentales

Algorithme	Opérations fondamentales
Recherche d'un élément dans une liste en mémoire centrale	comparaison entre l'élément cherché et les composants de la liste
Trier une liste d'éléments	- comparaison entre deux éléments - déplacement des éléments
multiplier deux matrices de nombres	- multiplication - addition

Complexité d'une séquence d'instructions

S = Begin

i_1 ; i_2 ; . . . ; i_n

End

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

Nb (i_k) = le nombre d'opérations fondamentales dénotées *Opf* contenues dans l'instruction algorithmique i_k.

Complexité d'une instruction conditionnelle

Cond = if Expr then E_1 else E_2 ;

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

Complexité d'une itération finie bornée

Iter = Itération Expr(i)
S
finItér

Donc : Nb(Iter) = [Nb(S) + Nb (Expr(i))] x Nb_itérations

Par exemple dans le cas d'une boucle For :

Iter = For I := a to b do
Begin
i_1 ; i_2 ; . . . ; i_n
End;

Donc :

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

Complexité de la condition de continuité I<=b

Complexité des appels de sous-programmes :

Sans récursivité :

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

Sous-pgme A

I_1;
I_2;
Appel de B;
I_3;

En cas de récursivité, calculer Nb(A) donne lieu à la résolution d'une relation de récurrence.

Exemple:

```
Function fact(n : integer) : integer;
  Begin
    If n=0 then fact:=1 else fact:=n*fact(n-1);
  End;
```

Le nombre T(n) d'opérations fondamentales (*) vérifie :
$T(0)=0$ et $T(n)=T(n-1)+1$; pour $n\geq 1$

Par une résolution directe on obtient : $T(n)=n$.

Exemple d'indication :

Soit l'algorithme de recherche séquentielle d'un élément X dans une liste L.

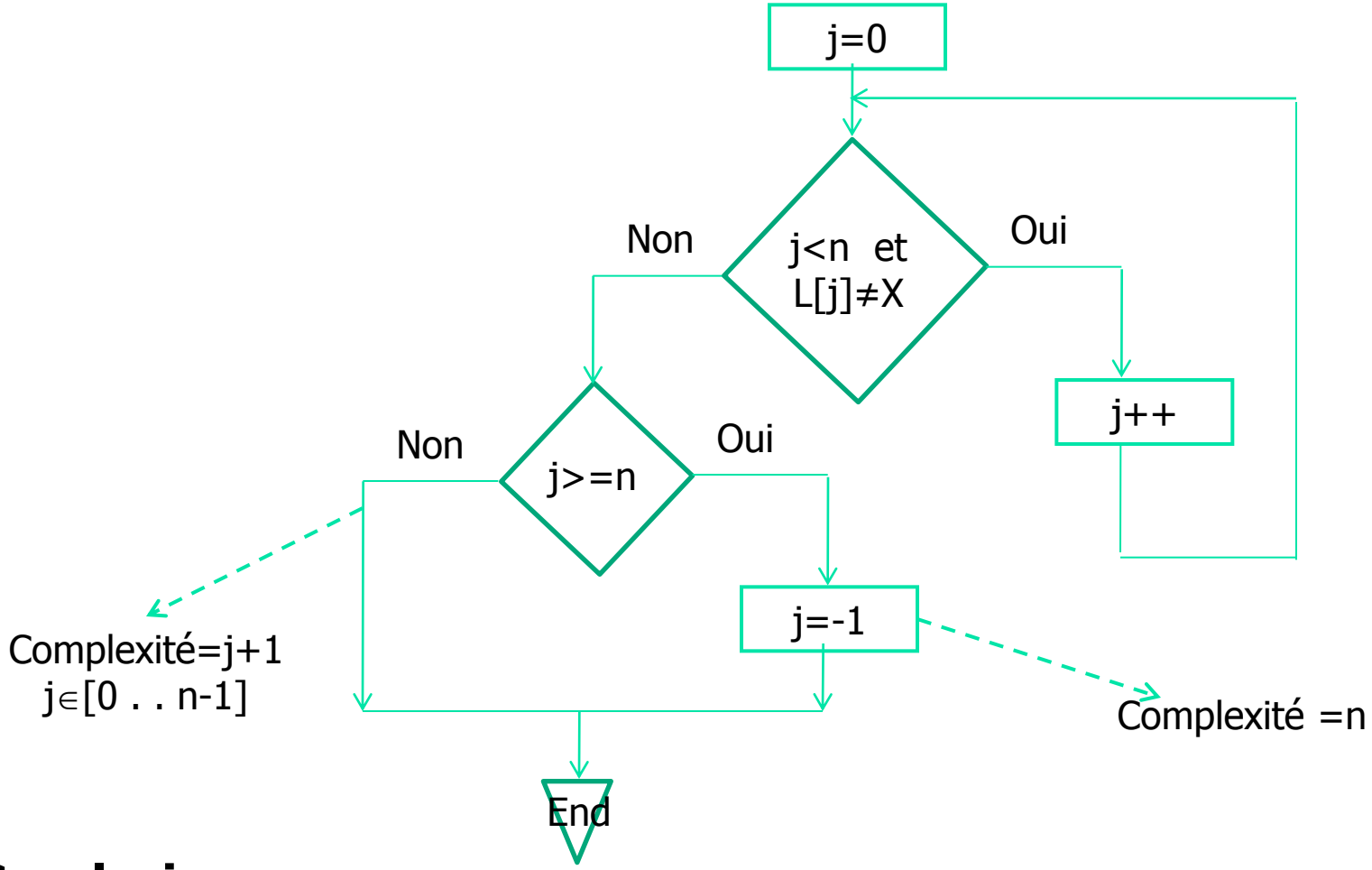
```
int Chercher (Element L[n] , Element X)
{
(1)       int j=0;
(2)       while (j<n and L[j]!=X)    j++;
(3)       if (j>=n) j=-1;
          return j;
}
```

La complexité de cet algorithme=Fonction (nombre d'itérations, nombre d'opérations par itération).

3 opérations fondamentales candidates dans la ligne (2).

Opération fondamentale à retenir = comparaison (L[j]!=x) dans la ligne (2).

Les 2 autres opérations dépendant de la programmation.



<u>Conclusion :</u>

La complexité (nbre de comparaisons L[j]≠x) dépend de la taille des données.

Complexité au pire = n

Complexité au meilleur = 1

Complexité moyenne comprise entre 1 et n

Pour une taille fixée, elle dépend des différentes données possibles.

Complexité en moyenne et au pire :

Soit $coût_A(d)$ =complexité de l'algo A pour la donné d.

L'ensemble des données de taille n est noté D_n.

La complexité dans le meilleur des cas

$$Min_A(n) = \min \{coût_A(d) ; d \in Dn\}$$

La complexité dans le pire des cas

$$Max_A(n) = \max \{coût_A(d) ; d \in Dn\}$$

La complexité moyenne

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

où P(d) est la probabilité d'avoir la donnée d en entrée de l'algorithme

Remarque:

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

Exercices du TD N° 02

Exercice 01: Déterminer la complexité moyenne, minimale et maximale, en nombre de comparaisons, de l'algo de recherche séquentielle d'un élément dans une liste.

```
int Chercher (Element L[n] , Element X)
{

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

Exercice 02: Soient les matrices réelles n x n : $A=(a_{ij})$, $B=(b_{ij})$ et $C=(c_{ij})$. Etudier la complexité de l'algo suivant qui calcule les coefficients (c_{ij}) de la matrice produit C= A x B selon la formule classique :

$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$ pour i, j compris entre 1 et n

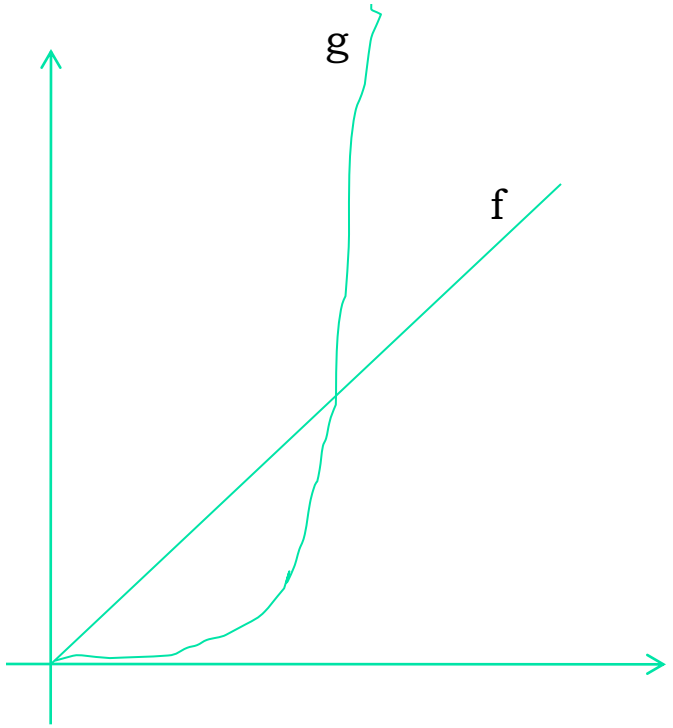
```
Const int n=8;
typedef float matrice[n][n];

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];
                                }

}
```

Ordre de grandeur :

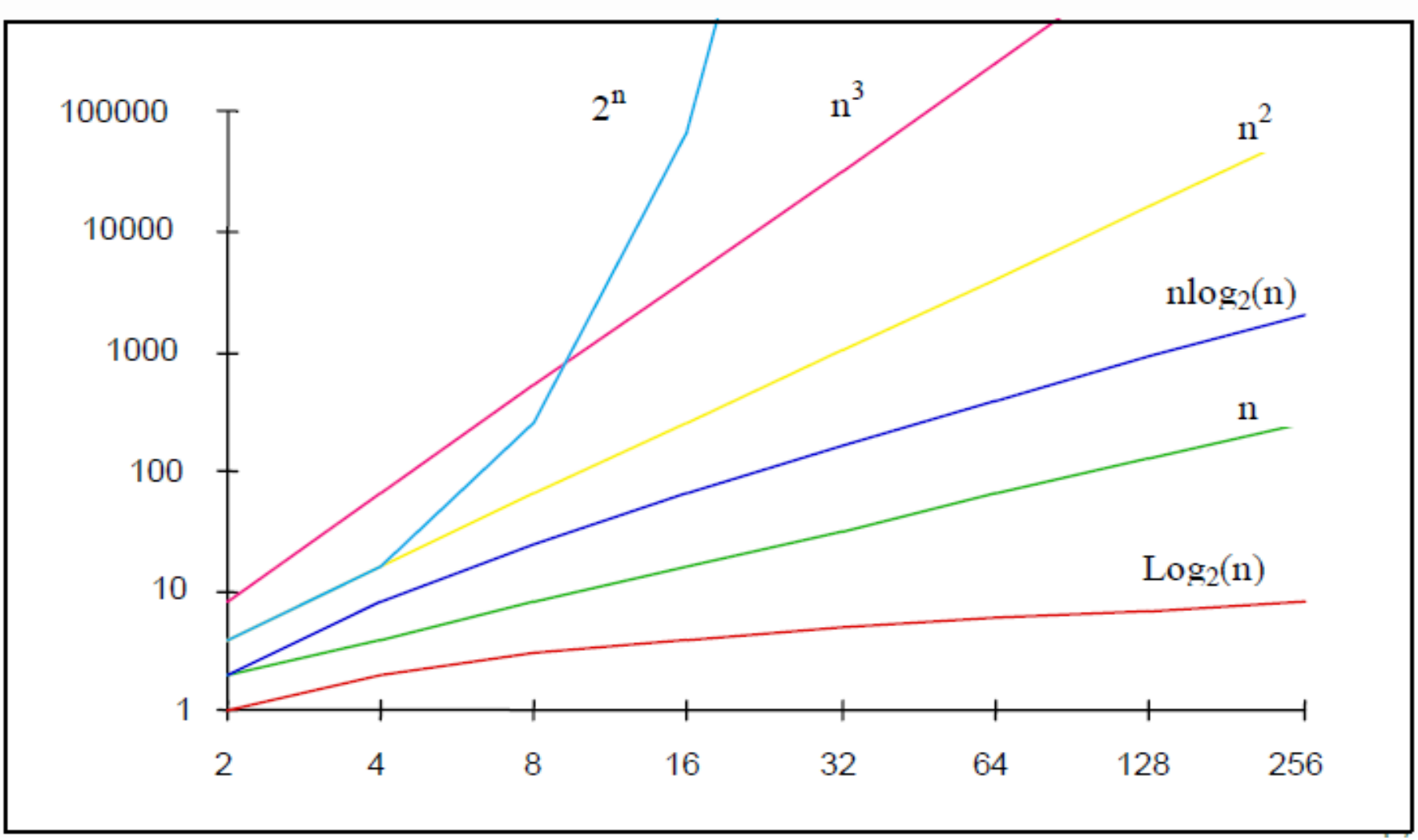
On veut comparer l'algo A_1 de complexité $M_1(n)=n^2$ et A_2 de complexité $M_2(n)=2n$.

A_2 est meilleur que A_1 pour n>2. car *g(n)=n²* croit plus vite que *f(n) =n*

puisque $\lim_{n\to\infty}\left(f(n)/g(n)\right)=0$



L'ordre de grandeur asymptotique de $g(n)=n^2$ est plus grand que celui de $f(n)=n$.



Les fonctions $\log_2 n$, n , $n\log_2 n$, n^2, n^3, 2^n forment une échelle de comparaison

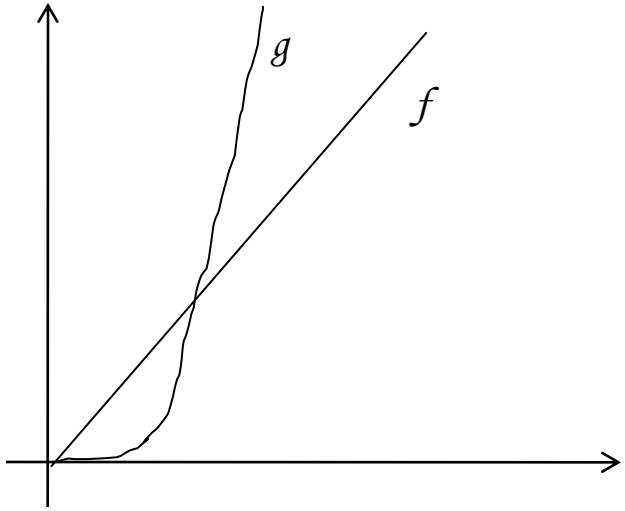
Définition 1 :

Soient f et g deux fonctions de N dans R^+

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

Exemple:

1) f(n)=2n g(n)=n^2 f=O(g) car f(n) ≤ g(n) pour n >1



Définition 2 :

f=θ(g) SSI f =**O**(g) et g=**O**(f) c.à.d

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

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

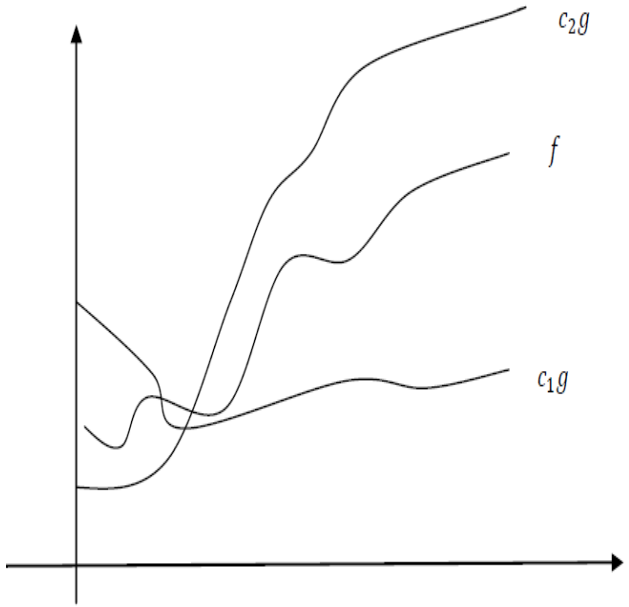


Tableau A : Estimation du temps d'exécution de certains algos pour différentes tailles n de données d'un pbme sur un ordinateur effectuant 10^6 opérations par seconde. On voit que, plus la taille des données est grande, plus les écarts entre les différents temps d'exécution se creusent.

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

On note par ∞ une valeur qui dépasse 10^{100}.

Les algorithmes utilisables sont ceux qui s'exécutent en temps:

Constant, Logarithmique, Linéaire ou n log n ;

Tableau B : Estimation de la taille maximale des données traitées par certains algos en un temps d'exécution fixé sur un ordinateur faisant 10^6 opérations par seconde.

Infinité de données traitées/ seconde

19 données traitées/ seconde

Complexité / Temps calcul	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	$63x10^3$	10^3	100	19
1 mn	∞	∞	$6x10^7$	$28x10^5$	$77x10^2$	390	25
1 h	∞	∞	$36x10^8$	$13x10^7$	$60x10^3$	$15x10^2$	31
1 jour	∞	∞	$86x10^9$	$27x10^8$	$29x10^4$	$44x10^2$	36

Infinité de données traitées/ jour

36 données traitées/ jour

Tableau C : Evolutions mutuelles du temps et de la taille

Complexité	1	Log $_2$n	n	nlog$_2$ n	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	t+3,32	10xt	(10+ε)xt	100xt	1000xt	t^{10}
Evolution de la taille quand le temps est multiplié par 10	∞	n^{10}	10xn	(10-ε)xn	31,6xn	2,15xn	n+3,32

Temps inchangé pratiquement

Taille exponentielle

Temps exponentiel

Taille inchangée pratiquement

Il est donc toujours utile de rechercher des algos efficaces, même si les progrès technologiques accroissent les performances du matériel.