Contenu de la matière

Chapitre 0 : Rappels

Chapitre 1 : Complexité algorithmique

1- Introduction

2-calcul de la complexité

Chapitre 2: Algorithmes de tri

1- présentation

2- tri à Bulles, Par sélection, par Insertion, Tri Rapide,

Chapitre 3: Les Arbres

1- Introduction

2- Définition

3- Arbre Binaire, passage n-aire binaire, Représentation chainée, parcours d'un Arbre, Arbre Binaire particuliers(complet, de recherche, Tas)

Chapitre 4: Les Graphes

1- Définitions 2- Représentation des Graphes 3- Parcours des Graphes

Chapitre 1
La complexité des algorithmes

1-Introduction

Tous les algorithmes ne sont pas équivalents. On les différencie selon 2 critères

1. **Temps de calcul : lents vs rapides**
2. **Mémoire utilisée : peu vs beaucoup**
3. **On parle de complexité en temps (vitesse) ou en espace (mémoire utilisée)**

But de la complexité est:

- **avoir une idée de la difficulté des problèmes.**
- **donner une idée du temps de calcul ou de l'espace nécessaire pour résoudre un problème.**

➢ **La complexité va permettre de comparer les algorithmes Entre eux.**

➢ **Elle est Exprimée en fonction du nombre de données et de leur taille.**

A défaut de pouvoir faire mieux :

- **On considère que les opérations sont toutes équivalentes**
- **Seul l'ordre de grandeur nous intéresse.**
- **On considère uniquement le pire des cas**
- **Pour n données on aura : O(n) linéaire, O(n2) quadratique O(n log(n)),…**

Qu'est-ce que la complexité des algorithmes ?

- Ce que l'on entend par complexité des algorithmes est une évaluation du coût d'exécution d'un algorithme en termes de :

- temps : **complexité temporelle,**
- d'espace mémoire : **complexité spatiale.**

- But de la complexité est de comparer entre eux des algorithmes différents qui résolvent le même problème.
- Ce qui suit traite essentiellement de la complexité temporelle, mais les mêmes notions permettent de traiter de la complexité spatiale.

Objectif de l'analyse de la complexité

Quantifier les 2 grandeurs physiques :

- temps d'exécution
- place mémoire

Définition 1. La complexité d'un algorithme est
en temps, le nombre d'opérations élémentaires effectuées pour traiter une donnée de taille n,
en mémoire, l'espace mémoire nécessaire pour traiter une donnée de taille n.

Comment quantifier ?

On pourrait compter :

- Pour le temps : le nombre d'opérations effectuées par le programme et le temps nécessaire pour chaque opération.
- Pour la place : le nombre d'instructions, le nombre de données du programme et le nombre de variables de travail.

Ce type d'analyse conduit à des énoncés comme :

« l'algorithme A implémenté par le programme P sur l'ordinateur O et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire. »

Comment quantifier ?

Les mesures étant dépendantes :

- ✓ du processeur utilisé,
- ✓ des temps d'accès à la mémoire vive et de masse,
- ✓ du langage de programmation,
- ✓ de la qualité du code produit par le compilateur utilisé,

etc.

une telle démarche rendrait difficile la comparaison des algorithmes entre eux.

D’où une approche indépendante de la machine sur laquelle s’exécute l’algorithme et de la traduction de l’algorithme en langage exécutable par la machine est nécessaire pour évaluer l'efficacité des algorithmes.

Le but étant d’établir des résultats plus généraux permettant **d’estimer l’efficacité de l’algorithme.**

Le type d'énoncé que l'on souhaite produire est donc :

« Entre différents algorithmes réalisant une même tâche, quel est le plus rapide, indépendamment de l'implantation, et dans quelles conditions ? »

2-Calcul de la Complexité temporelle d’un algorithme

De quoi dépend la mesure de la complexité ?

A priori, on peut distinguer 2 facteurs déterminants pour le calcul de la complexité en temps d'un algorithme :

1. **Une ou plusieurs opérations élémentaires** pertinentes par rapport au problème, au sens où le temps d'exécution d'un algorithme résolvant ce problème est toujours proportionnel au nombre de ces opérations.

On les appellera **opérations fondamentales**

2. La **taille des entrées.**

En pratique, pour évaluer la taille des données d'un algorithme on choisit la ou les dimensions les plus significatives.

Par exemple dans le cas de :

- la recherche dans un vecteur ou le tri d'un vecteur : la taille est le nombre d'éléments du vecteur
- le produit de 2 matrices carrées : la taille est la dimension de la matrice

Détermination des opérations fondamentales

La nature du problème fait que certaines opérations sont fondamentales et que leur nombre intervient principalement dans l'étude de la complexité de l'algorithme.

Exemple d'opérations fondamentales :

- Pour la recherche d'un élément e dans un vecteur : le nombre de comparaisons entre e et les entrées du vecteur;
- Pour la recherche d'un élément sur mémoire secondaire : le nombre d'accès à la mémoire secondaire;
- Pour le tri d'un vecteur, on peut considérer 2 opérations fondamentales :
  - le nombre de comparaisons
  - le nombre de déplacements (transferts)
- Pour la multiplication de 2 matrices : le nombre de multiplications et le nombre d'additions.

Calcul du nombre d’opérations fondamentales

- Après avoir déterminé les opérations fondamentales, il faut compter le nombre d’opérations de chaque type.

- S’il existe plusieurs opérations fondamentales, il faudra les décompter séparément.
- De plus, si besoin est, il faudra affecter, à chacune d’entre elles, un poids qui tient compte des temps d’exécution différents.
- Précisons qu’il n’existe pas de système complet de règles permettant de compter le nombre d’opérations en fonction de la syntaxe des algorithmes.

On peut cependant citer quelques règles.

➢Pour mesurer la complexité temporelle d'un algorithme on établie une relation (appelée fonction de complexité) entre la durée d'exécution de l'algorithme (exprimée en termes de nombres d'opérations élémentaires) et la taille des données à traiter .

➢Soit la fonction : $T_A : D_n \rightarrow R^+$ tel que $T_A(n)$ est Appelée :

- Complexité temporelle de l'algo A
- Temps d'exécution de l'algo A pour une donnée de taille n
- Nombre d'opérations élémentaires nécessaire à l'exécution de l'algo A sur une donnée de taille n.

➢**Remarque**: Il n'existe pas de système complet de règles permettant de compter le nombre d'opérations élémentaires en fonction de la syntaxe des algorithmes.

Nous essayerons de fixer certaines règles pour aider à l'évaluation de la complexité temporelle d'un algorithme.

Règle 1 : nombre d'opérations d'une séquence

Lorsque les opérations sont dans une séquence d'instructions, leur nombre s'ajoutent.

Si on note par $T_{Ij}(n)$ le nombre d'opérations de l'instruction Ij avec une entrée de taille n, on a :

$T_{I1+I2+\ldots+Im}(n) = T_{I1}(n) + T_{I2}(n) + \ldots + T_{Im}(n)$

Règle 2 : nombre d'opérations d'une alternative

Comme il est difficile, voir impossible, de déterminer la branche de l'alternative qui est exécutée, la règle généralement adoptée consiste à majorer le nombre d'opérations par le nombre maximum d'opérations entre les 2 branches.

Si I est une alternative de la forme :

si (condition) alors I1 sinon I2 fsi

Alors:

$T_I(n) = T_{condition}(n) + \max\{T_{I1}(n), T_{I2}(n)\}$

Règle 3 : nombre d’opérations d’une boucle

Le nombre d’opérations dans la boucle est : $\sum Ti(n)$ avec,

- i : variable de contrôle de la boucle
- Ti(n) : nombre d’opérations fondamentales lors de l’exécution de la ième itération.

Pour évaluer les bornes de i, il faut connaître le nombre d’itérations.

Dans le cas d’une boucle « pour » ce nombre est défini dans l’algorithme.

Pour les autres boucles, il doit être calculé à partir de l’algorithme.

Ce calcul peut s’avérer difficile. On se contente, parfois d’un majorant.

R3-1 Boucle pour

Soit A une action de la forme : pour i de Ideb à Ifin faire A1 fpr ;

Nb itérations est: NB=Ifin-Ideb+1

$$T_A(n) = \sum_{i}^{nb}(1affec+1add+1comp+T_{A1}(i,n))+1affec+1add+1comp$$

$$= NBT_{A1}(n)+(NB+1)affec + (NB+1)comp + NBadd$$

Remarque :

Une pratique courante consiste à négliger (lorsque cela ne change pas la complexité) les opérations implicites telles que l'affectation, la comparaison et l'incrémentation.

Exemple: **calculer la complexité de la partie d'algo suivant: (On suppose que l'opération fondamentale est l'+)**

```
Pour i de 1 à n faire
Res ← X+Y+Z+Res;
        Si (T[i]+K <B) alors
A1:  Pour j de 1 à n faire Res ← Res + T[j] fpr;
      Sinon
A2:   Res ← Res+T[I];
Fsi;
Fpr;
```

R3-2 Boucle tant que

Soit une action A de la forme : tant que (cond) faire A1 ftq;

En notons NB: le nombre d'itérations on a:

$T_A(n)={}_i^{nb}\sum Tcond(i,n)+T_{A1}(i,n)+T_{cond}(nb+1,n)$

Remarque:

-la condition cond est exécutée dans tant que une fois de plus que le corps A1 de la Boucle.

-Tcond n'est pas toujours constante, elle peut dépendre de de n et de l'itération i .

-Exemple:

Res ←0;

L← 2;

Tant que (L≤ 100) faire Res ← Res +2*Tab[L+1]

L ← L+2;

Ftq;

Règle 4 : nombre d'opérations d'un appel de procédure ou de fonction non récursives

Lorsqu'il n'y a pas de procédures ou de fonctions récursives, on peut toujours trouver une façon d'ordonner les procédures et les fonctions de telle sorte que chacune d'entre elles n'appelle que des procédures ou fonctions dont le nombre d'opérations fondamentales a déjà été évalué.

On évalue d'abord les fonctions ou procédures qui ne contiennent pas d'appels à d'autres fonctions et procédures puis celles qui contiennent des appels aux précédentes,...ect.

Si P1,P2,...,PK sont ordonnées alors

T(P1,P2,......,PK)= $\sum$ T(Pi) pour i=1,....,K

Exemple:

On veut calculer le nombre de multiplication T(n) pour N=n pour l'algorithme Essai.

```
Algorithme Essai
Début
C: Booleen;
N,R0,R1,I: Entier;
Lire (C); lire (N);
R0 ← 1;  R1 ← 1;
Si  (C=Vrai) alors
Pour I de 1 à N faire B(N) Fpr;
fsi;
A(N);
Fin.
Procédure A (↑↓R: Entier)
Début
I: Entier;   pour I de 1 à N faire  R← R*I fpr;
Fin.
Procédure B (↓↑ R: Entier)
Début
I,J: Entier
J ← 1;  pour I de 1 à N faire A(J); R ← R*J fpr;
Fin.
```

Règle 5 : nombre d’opérations d’un appel de procédure ou de fonction récursives

Pour les procédures ou fonctions récursives, compter le nombre d’opérations fondamentales donne en général lieu à la résolution de relations de récurrence.
Le nombre T(n) d’opérations dans l’appel de procédure avec un argument de taille n s’écrit selon la récursion, en fonction de divers T(k) , pour $k<n$.

Règle 5 : exemple

Fonction fact (n:Entier):Entier

Début

Si (n = 0) alors fact ← 1 sinon fact ← n* fact(n-1) fsi;

fin

Si on choisit comme opération fondamentale la multiplication de 2 entiers, le nombre T(n) d'opérations fondamentales vérifie :

- $T(0) = 0$,
- $T(n) = T(n-1) + 1$, pour $n \geq 1$,

D'où :

$T(n)=T(n-1)+1=(T(n-2)+1)+1 =((T(n-3)+1)+1)+1 =\ldots=$
$(\ldots(T(1)+1)+1+\ldots+1)+1= (\ldots((T(0)+1))+1+\ldots+1)+1$

Soit : $T(n) = n$

Détermination de la complexité : un exemple (1)

A titre d'exemple, nous allons analyser une fonction de recherche séquentielle d'1 entier x dans 1 vecteur v. Cette fonction renvoie le rang j de x si $x \in v$ et 0 si $x \notin v$.

Fonction rang(x, v[n] : entier) : entier

Début

j : entier ; j ← 1 ;

Tant que (j ≤n et v[j] ≠ x) faire j ← j+1 ftq ;

si j>n

alors

rang ← 0

sinon

Rang← j ;

Fsi;

Fin.

Analyse de la complexité

Les opérations fondamentales sont les comparaisons de x avec les éléments du vecteur. Il y en a une par itération.

Le nombre d'itérations est égal à :

- j : rang de la 1ère occurrence de x si x est dans le vecteur v.
- n (taille du vecteur) : si x n'est pas dans v,

Conclusion

L'exemple analysé met en évidence 3 points essentiels :

1. Le choix de (ou des) l'opération(s) que l'on prend en compte doit être établi avant toute analyse et précisé dans le résultat.
2. La complexité dépend de la taille des données (ici n),
3. La complexité dépend pour une taille fixée, des différentes entrées possibles.

Dans le cas de l'exemple,

pour les données de taille n, la complexité en nombre de comparaisons varie de 1 à n selon les données x et v :

Les données où x apparaît au rang i de v correspondent à une complexité i,

Celles où x n'apparaît pas dans v, correspondent à une complexité n.

Exercice :

Calculer la complexité de l’algorithme suivant :

```
I← 0 ;
J ← 0 ;
Tant que (i<n) faire
Si (i mod2= 0) alors J ← J+1
                    Sinon J ← J/2 fsi ;
I ← i+1 ;
Ftq;
```

Complexité en moyenne au mieux et au pire

L’exemple de la recherche séquentielle a mis en évidence , que le temps d’exécution d’un algorithme dépend de la donnée sur laquelle il opère.

Plusieurs mesures peuvent donc être nécessaires pour analyser le comportement d’un algorithme sur un ensemble de données de taille fixée.

Le plus souvent on s’intéresse à 3 mesures :

- La complexité dans le meilleur des cas,
- La complexité dans le pire des cas,
- La complexité en moyenne

La complexité dans le meilleur des cas

La complexité dans le meilleur des cas, d'un algorithme A sur l'ensemble Dn des différentes données possibles de taille n, donne une indication sur la borne minimale de la complexité de A sur les données de taille n.
Elle est déterminée sur la base d'une construction de données d particulière correspondant à la configuration la plus favorable pour la résolution de l'algorithme A.
TminA(n) = min { TA(d) ; d ϵ Dn }

Exemple:

Pour l'exemple de la recherche séquentielle, la complexité minimale est 1.
Elle correspond au jeu de données d qui est tel que x apparaît au rang 1 du tableau v.

La complexité dans le pire des cas

La complexité dans le pire des cas, d'un algorithme A sur l'ensemble Dn des différentes données possibles de taille n, donne une borne supérieure du temps d'exécution de A. Elle est particulièrement utile car elle donne une estimation maximale des données qui pourront être traitées par A.

$TmaxA(n) = \max \{ TA(d) ; d \in Dn \}$

Exemple:

Pour l'exemple de la recherche séquentielle, la complexité maximale est n.

Elle correspond au jeu de données d qui est tel que x n'apparaît pas dans le tableau v.

La complexité en moyenne

- Les cas extrémaux ne sont pas les plus fréquents. Dans la pratique on cherche à savoir quel est « en général » le comportement de l'algorithme. Cette information est fournie par la complexité en moyenne.

$$Tmoy_A(n)=\sum p(d)*T_A(d) \;/ d\in D$$

Où p(d) est la probabilité que l'on ait la donnée d en entrée de l'algorithme.

- La complexité en moyenne nécessite une connaissance de la distribution probabliliste des données.
- La complexité en moyenne est souvent difficile à calculer car il n'est pas toujours facile de déterminer un modèle de probabilité adéquat au problème. L'analyse est mathématiquement difficile .

Comparaison de deux algorithmes
Ordre de grandeur

- Si l’on souhaite comparer les performances d’algorithmes, on peut considérer une
mesure basée sur leur temps d’exécution.
- Cette mesure est appelée la complexité
en temps de l’algorithme.
- On utilise la notion dite « de Landau » qui traite de l’ordre de grandeur du nombre d’opérations effectuées par un algorithme donné.
- on utilise la notation « O » qui donne une majoration de l’ordre de grandeur du nombre d’opérations.

Ordre de grandeur asymptotique

- Pour n grand, il est secondaire de savoir si un algorithme comporte n ou n+50 opérations.
- Les constantes multiplicatives ont, elles aussi, peu d'importance. Si on doit comparer les algorithmes :

A1 de complexité : n^2

A2 de complexité : 2n

On voit que A2 est meilleur que A1 dès que n>2. De plus, quelles que soient les constantes c1 et c2; si :

A1 est de complexité : $c1.n^2$

A2 est de complexité : c2.2n

A2 est toujours meilleur que A1 à partir d'un certain n, car $f(n)=n^2$ croît plus vite que g(n)=2n.

On dit alors que **l'ordre de grandeur asymptotique de f(n) est** strictement plus grand que celui de g(n).

Pour comparer 2 algorithmes, il suffit de comparer l'ordre de grandeur asymptotique des 2 algorithmes.

Complexité asymptotique

Calculer la complexité de façon exacte n'est pas raisonnable vu la quantité d'instructions de la plupart des programmes et n'est pas utile pour pouvoir comparer deux algorithmes.

Première approximation : on ne considère souvent que la complexité au pire.

Deuxième approximation : on ne calcule que la forme générale de la complexité.

Troisième approximation : on ne regarde que le comportement asymptotique de la complexité.

Pour cela on utilise la notation en O (**Notation de Bachmann Landau)**

Notation de Bachmann Landau : « grand O »

Pour déterminer l’ordre de grandeur asymptotique on utilise la

notation de Landau en "grand O" définie comme suit :

Étant données 2 fonctions f et g de N dans R+ :

f = O(g) → ∃ c ϵR+*, ∃n_0 ϵ N, tels que:

pour tout n >n_0, f(n)≤ c.g(n)

Cette notion donne un majorant de l’ordre de grandeur de f.

f = O(g) signifie que f est en O(g) c’est-à-dire que :

l’ordre de grandeur asymptotique de f

≤

l’ordre de grandeur asymptotique de g

Notation en grand O :

Exemple1

Supposons que le temps d'exécution d'un programme, pour une entrée n donnée, est : $T(n) = (n+1)^2 = n^2 +2n+1$

Pour $n \geq 1$, on a : $n \leq n^2$ et $1 \leq n^2$

Ainsi $n^2 +2n+1 \leq n^2 +2n^2 +n^2 \leq 4n^2$

T(n) est donc en $O(n^2)$ car il existe un entier n0=1 et une constante c=4 tels que pour tout entier $n \leq 1$ on a $T(n) \leq 4n^2$ **Notation en grand O : Exemple**

En règle générale, si T(n) est polynomiale de la forme :

$akn^K + ak\text{-}1n^{K-1} + . . . + a2n^2 + a1n + a0$ avec ak>0

alors T(n) est en $O(n^K)$

Exemple2 : calculer la puissance n-ième d'un entier (n étant positif)

en C :

```
int puissance (int a, int n)
{
int i, x ;
x=1 ;
for (i = 0; i < n; i=i+1) // boucle de n iterations
x = x * a ; // opération fondamentale la multiplication
return x ;
}
```

On a une multiplication par itération

Donc pour n itérations on aura n multiplications

La complexité de cet algorithme est en temps linéaire O(n)

performances d'un algorithme

Soit deux fonctions f(x) et g(x) avec $x \geq 0$

On dit que f(x)=O(g(x)) si

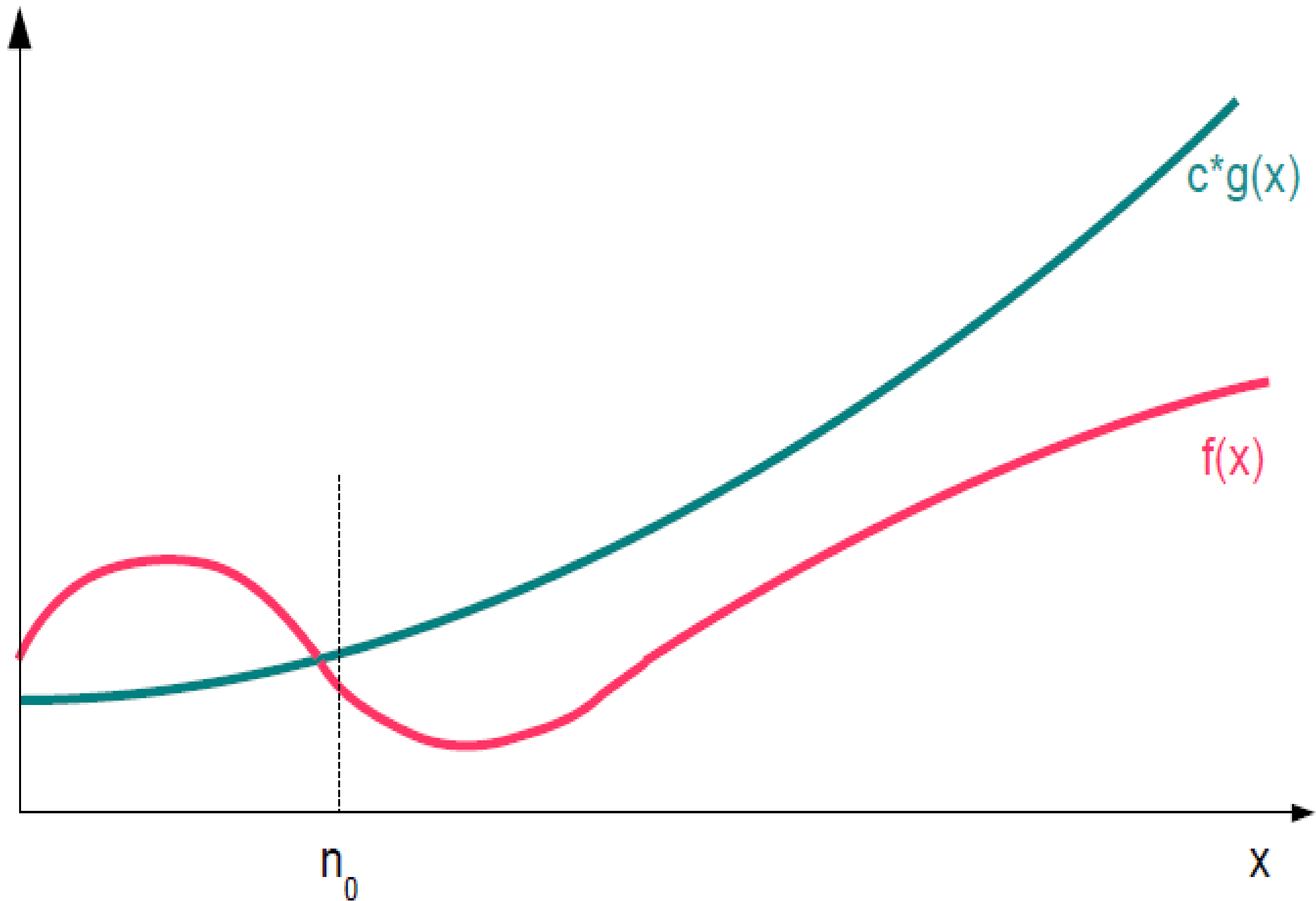
$\exists\ N0 \geq 0, \exists\ c \rangle 0, \forall\ N \geq N0\ f(N) \leq c \cdot g(N)$

A partir d'un certain rang, la fonction « g » majore la fonction « f » à un coefficient multiplicatif près.

On définie la performance d'un algorithme par le calcul de

$$\lim_{n \to \infty} f(n)/g(n) = c$$

Donc g est une borne supérieure de f.



f = O(g) signifie que f est dominée asymptotiquement par g.

Classes de complexité

$O(1)$: complexité constante, pas d'augmentation du temps d'exécution quand le paramètre croit

$O(\log(n))$: complexité logarithmique, augmentation très faible du temps d'exécution quand le paramètre croit. *Exemple : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).*

$O(n)$: complexité linéaire, augmentation linéraire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). *Exemple : algorithmes qui parcourent séquentiellement des structures linéaires.*

$O(n\log(n))$: complexité quasi-linéaire, augmentation un peu supérieure a $O(n)$. Exemple : algorithmes qui décomposent un problème en d'autres plus simples, traités indépendamment et qui combinent les solutions partielles pour calculer la solution générale. Exemple tri rapide

$O(n^2)$: complexité quadratique, quand le paramètre double, le temps d'exécution est multiplie par 4. *Exemple : algorithmes avec deux boucles imbriquées.*

$O(n^i)$: complexité polynomiale, quand le paramètre double, le temps d'exécution est multiplie par 2i. *Exemple : algorithme utilisant i boucles imbriquées.*

$O(2^n)$: complexité exponentielle, quand le paramètre double, le temps d'exécution est élevé a la puissance 2.

$O(n!)$: complexité factorielle.

Complexités les plus fréquentes

Les fonctions qui correspondent aux ordres de grandeurs les plus fréquemment rencontrés dans les calculs de complexité sont présentées par ordre croissant dans le tableau ci-dessous

Fonction	Nom
1	Constante (ne dépends pas de la taille des données)
$Log_2(n)$	logarithmique
n	linéaire
$nlog_2(n)$	N logarithmique
n^2	Quadratique ou polynomiale d'ordre 2
n^3	Cubique ou polynomiale d'ordre 3
2^n	exponentiel