Cnam Hauts de France

# *Complexité des algorithmes*

**Rédigé par** : Rachik FETTACHE

**Unité d'enseignement** : RCP101

# Sommaire

# Introduction

Quand on parle de compléxité algorithique , on pourrait penser qu'on parle d'une difficulté de compréhension , cependant cela est trompeur , on parlera plus d'efficacité.C est en ce terme qu'on pourra comparer des algorithmes .
On pourrait ainsi définir la compléxité comme une sorte de quantification de la performance d'un algorithme.

Cette notion de complexité algoritmique a été formalisée et a pris une définition plus rigoureuse grace à l'informatique théorique et aux progrès technologiques,elle est aujourd'hui tres utile dans beaucoup de domaines.

Mais jusque récemment , on ne pouvait pas faire grand chose car on était bloque par la mesure de la complexité , on peut désormais estimer cette compléxité, cela a permis de débloquer un grand nombre de problèmes.

Le but de ce rapport est d'essayer de synthétiser au maximum les choses : on verra comment on peut mesurer l'efficacité d'un algorithme, comparer ces mesures. Puis qu'on peut distinguer différentes mesures et classifier les complexités en différentes classes.

# 1- Historique

Quand les scientifiques ont voulu énoncer formellement et rigoureusement ce qu'est l'*efficacité* d'un algorithme ou au contraire sa *complexité*, ils se sont rendu compte que la comparaison des algorithmes entre eux était nécessaire et que les outils pour le faire à l'époque1 étaient primitifs.

Dans la préhistoire de l'informatique (les années 1950), la mesure publiée, si elle existait, était souvent dépendante du processeur utilisé, des temps d'accès à la mémoiré vide et de masse et de, du langage de programmation et du compilateur utilisé.

Une approche indépendante des facteurs matériels était donc nécessaire pour évaluer l'efficacité des algorithmes.

Donald Knuth fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa série The Art of Computer Programming. Il complétait cette analyse de considérations propres à la théorie de l'information : celle-ci par exemple, combinée à la formule de Stirling, montre que, dans le pire des cas, il n'est pas possible d'effectuer, sur un ordinateur classique, un tri général (c'est-à-dire uniquement par comparaisons) de N éléments en un temps croissant avec N moins rapidement que $N \ln N$.

# 2- Quelques définitions

Algorithme (source wikipédia) : Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, ont donné cette définition :

*"Procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie."*du nom du mathématicien perse Al Khuwarizmi (780 – 850)

*Compléxité d'un algorithme* :a complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.Elle est souvent déterminée à travers une description mathématique du compportement de cet algorithme.

*Complexité d'un problème:*La Compléxité d un problème est le meilleur résultat qui résout ce problème

*Efficacité d'un algorithme* : on considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur.

*Algorithme optimal*:Un algorithme est dit optimal si sa compléxité est la compléxité minimale parmi les algorithmes de sa classe.

# 3- Mesure asymptotique

## 3.1- Complexité asymptotique

La compléxité est une mesure du comportement asymptotique de l'algorithme, cela signifie que lorsque la valeur des entrées tend vers l'infini l'exécution de l'algorithme devient de plus en plus lent. Cette mesure asymptotique indique donc que les écarts entre deux complexités se font de plus en plus importants quand la taille de l'entrée augmente.
On s'intéresse donc plus à l'ordre de grandeur :'l'asymptotique.

Cependant , on constate que lorsque n devient grand ,le rapport des complexités va tendre vers une constante .
Si on prend 2 algorithmes A et B , de compléxité en temps défni par respectivement cA(n) = n et cB (n)=3n + 100 , on s'aperçoit que les les algorithmes ont un comportement identique.

| exemplaire de petite taille | | | | exemplaire de grande taille | | | |
|---|---|---|---|---|---|---|---|
| n | A | B | A/B | n | A | B | A/B |
| 10 | 10 | 130 | 0.07 | $10^6$ | 1000000 | 3000100 | 0.33 |
| 100 | 100 | 400 | 0.25 | $10^9$ | 1000000000 | 3000000100 | 0.33 |

*Figure 1 : Tableau représentant un exemple d'algorithme.*

## 3.2- Notation « grand O »

La complexité prend en réalité qu'un ordre de grandeur du nombres d'opérations.
Pour représenter cette compléxité, on utilise une notion spécifique la notation O .
On différencie_nombres d'opérations différentes et ordre de grandeur , par exemple des algorithmes effectuants environ N opérations:2*N+5 opérations ou N opérations auront tous la meme meme compléxité , on la O(N) (lire « grand O de N »).De meme un algorithme en 2*N2 + 3*N + 5 aurant une compléxité de O(N2),

Pour etre plus précis d'un point de vue mathématique ,si f(N) désigne une fonction mathématique dépendant de la variable N,O(f(N) désigne la compléxité des algorithmes s'éxécutant en enviton f(N)opérations.

La notation O est aussi appelé notation de landau.

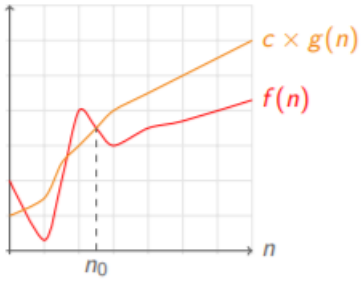


*Figure 2 : Interprétation graphique de la notion Grand O.*

## 3.3 Notation Grand Omega Ω

On dit qu 'une fonction est grand Omega d'une fonction g si et seulement si :

∃c>0, ∃n0>0 tel que ∀n>n0, c×g(n)<f(n)

On note alors f(n)=Ω(g(n)).

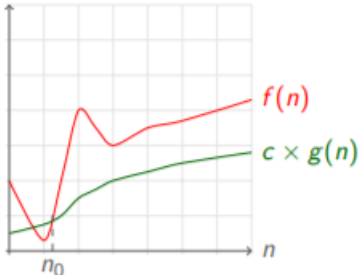


*Figure 3 :Interprétation graphique de la notion Grand Omega.*

## 3.4- Notation Grand Théta Θ

On dit qu'une fonction f est un grand Théta d'une fonction g si et seulement si :

∃c1>0, ∃c2>0, ∃n0>0 tel que ∀n>n0, c1×g(n)<f(n)<c2×g(n)

On note alors f(n)=Θ(g(n)).

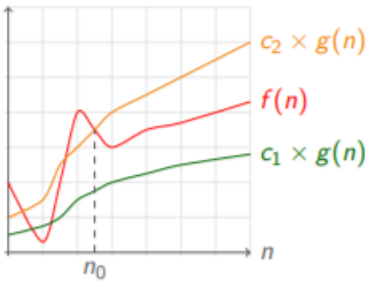


*Figure 4 : Interprétation graphique de la notion Grand Théta.*

### 3.5 D'équivalence

On dit qu'une fonction f est équivalente à une fonction g si et seulement si

$\lim_{n\to\infty} f(n)g(n)=1$

On note alors $f(n)\sim g(n)$.

# 4- Les types de complexités

## 4.1- Introduction

Pour rappel , le but de la compléxité est de comparer l efficacité des algorithmes mais ceci indépendamment des ressources matérielles utilisées ou d'un langage de programmation.
A noter que par exemple les programmeurs ne s'intéressent pas uniquement au temps d'exécution de leurs algorithmes Ils peuvent en mesurer d'autres caractéristiques , la plus courante étant la consommation mémoire.

## 4.2- Complexité en temps

Réaliser un calcul de complexité en temps revient à décompter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison…) effectuées par l'algorithme.Elle s'exprime en fonction de taille n des données .
Pour rendre ce calcul réalisable, on part du principe que que toutes les opérations élémentaires sont à égalité de coût. Cependant en pratique ce n'est pas tout à fait exact ,
il faut estimer que le temps d'exécution de l'algorithme est proportionnel au nombre d'opérations élémentaires.

## 4.3 Complexité en espace mémoire

La complexité en espace est quand à elle proportionnelle à la taille de la mémoire nécessaire pour stocker les différentes structures de données utilisées lors de l'exécution de l'algorithme(quantité d 'espace occupé pendant l'execution).

Ce type de compléxité est moins un problème aujourd'hui qu'au début de l'informatique ou la mémoire était restreinte et très couteuse.

# 5- Les mesures possibles

## 5.1- Introduction

Quand on parle de complexité « dynamique », on veut évaluer les ressources utilisées d un algorithme Plus précisément on veut evaluer :
-le cout d'un algorithme
-la quantité de ressources utilisées (dépendant de la taille des données , les ressources peuvent etre de plusieurs types : le temps , la mémoire , le matériel

Avant d estimer la compléxite , on doit préciser :

-La taille des données
-La notion de cout , on définit pour une donnée d , le cout de l algorithme A pour cette donné d : coutA(d).

En général quand on parle d'une complexité sans la définir , on parle de la complexité dans le pire des cas.
Pour deux données de meme taille , l execution d'un algorithme peut utiliser des ressources différentes , ainsi on peut définir trois mesures de complexités :compléxité dans le meilleure des cas , compléxité dans le pire des cas et complexité en moyenne,

## 5.2- Complexité dans le meilleur des cas

N est pas tres souvent utilisée,consiste à prendre le minimum des couts et est définie par :

$$Inf_A(n) = inf\{cout_A(d)/d\ de\ taille\ n\}$$

Exemple:Recherche dans une liste ordonnée ,le meilleur des cas est celui ou le nombre recherché est en premier , il est alors trouvé instantanément.Une erreur fréquente est de considérer n comme valant 1 ce qui est faux , on utilise systématiquement la notation grand O car on s'intéresse au comportement de l'algorithme en fonction de n.

## 5.3- Complexité dans le pire des cas

Le nombre d 'operations effectu ées par un algorithme va dépendre des conditions de départ ,
Si on prend l exemple de recherche d'un element dans une liste , on va comparer ici chaque element de la liste avec celui qu'on cherch : on a effectué X comparaisons , on dit que l alogithme a une complexité de O(L)(on parle aussi de complexité en temps linéaire car sa progressuion dépend fortement des conditions de depart,

On parle ainsii de pire des cas lorsqu'on sait que l entrée est la pire possible pour notre algorithme ,elle est définie par la formule suivante :

$$Sup_A(n) = sup\{cout_A(d)/d\ de\ taille\ n\}$$

En résumé ,La complexité dans le pire des cas permet de comparer l'efficacité de deux algorithmes. Elle s'oppose à la compléxité moyenne.

### 5.4- Complexité moyenne

C'est d'une certaine façon celle qui révèle le mieux le comportement "réel" de l'algorithme à elle est souvent dure à calculer, même de façon approximative! Son calcul peut nécessiter la mise en oeuvre de techniques mathématiques non élémentaires.Elle est définie par la formule suivante :

$$Moy_A(n) = \sum_{d\ de\ taille\ n} p(d) * cout(d)$$

## 6- Les classes de complexité

### 6.1- Les différentes classes

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.(source)

→ **Les algorithmes de complexité constante O(1)** : opération élémentaire , affectation , comparaison : *L'exécution ne dépend pas du nombre d'éléments en entrée mais s'effectue toujours en un nombre constant d'opérations*

→ **Les algorithmes sub-linéaires, dont la complexité est en général en *O*(log *n*)**. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal *n*
*La durée d'exécution croît légèrement avec n. Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.Des exemples comme l'algorithme de dichotomie ou de puissance en sont le parfait reflet.*

→ **Les algorithmes linéaires en complexité *O*(*n*)** sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de *n* symboles ou les algorithmes optimaux de tri.*C'est typiquement le cas d'un programme avec une boucle de 1 à net le corps de la boucle effectue un travail de durée constante et indépendante de n.*

→ **Les algorithmes de compléxité n-logarithmique** $O(n\log(n))$ *:Se rencontre dans les algorithmes où à chaque itération la taille du problème est divisée par une constante avec à chaque fois un parcours linéaire des données ,un exemple typique de ce genre de complexité est l'algorithme de tri "quick sort".*

→ **Les algorithmes de complexité située entre $O(n^2)$:quadratique et $O(n^3)$ cubique** , plus lents c'est le cas de la multiplication des matrices et du parcours dans les graphes.

→ **Compléxité polynomiale** $O(n^k)$ pour $k > 3$ sont considérés comme lents.

→ **Compléxité exponentielle** $O(2^n)$ (dont la complexité est supérieure à tout polynôme en $n$) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.Ces algorithmes sont dit « naifs »

Ci-dessous, un tableau récapitulatif des complexités et des cas d'algorithmes ou elles sont utilisées :

| **Complexité** | **Classe** | **Exemple** |
|---|---|---|
| 0(1) | Constante | Acces a un element de tableau |
| O (Log ((n)) | Logarithmique | Recherche dichotomique |
| O(n) | Linéaire | Recherche dans un tableau non trié |
| O (n.log(n)) | Quasi-linéaire | Tri rapide |
| O ($n^2$) | Quadratique | Tri à bulles |
| O ($n^3$) | Cubique | Multiplication de matrices |
| O ($2^n$) | Exponentielle | Algorithme du voyageur de commerce |

## 6.2- Analyse du comportement en fonction des différentes classes

On peut voir dans le tableau ci-dessous , le nombre d'opérations qu'il faut pour éxécuter un algorithme en fonction de sa compléxité et de la taille des entrées de l algorithme traité :

| n | 10 | 100 | 1000 | 1E+6 | 1E+9 |
|---|---|---|---|---|---|
| $\log_2(n)$ | 3,32 | 6,64 | 9,97 | 19,93 | 29,9 |
| n | 10 | 100 | 1000 | 1E+06 | 1E+09 |
| $n.\log_2(n)$ | 33,22 | 664,39 | 1E+04 | 2E+07 | 3E+10 |
| $n^2$ | 100 | 1E+04 | 1E+06 | 1E+12 | 1E+18 |
| $n^3$ | 1000 | 1E+06 | 1E+09 | 1E+18 | 1E+27 |
| $n^5$ | 1E+05 | 1E+10 | 1E+15 | 1E+30 | 1E+45 |
| $2^n$ | 1024 | 1,27E+030 | 1,07E+301 | | |

*Figure 5 :Nombre d'operations pour éxecuter un algorithme en fonction de sa compléxité et la taille des entrées.*

On distingue également dans le graphique qui suit , les différentes courbes des fonctions utilisées en rapport avec leur classe de compléxité :

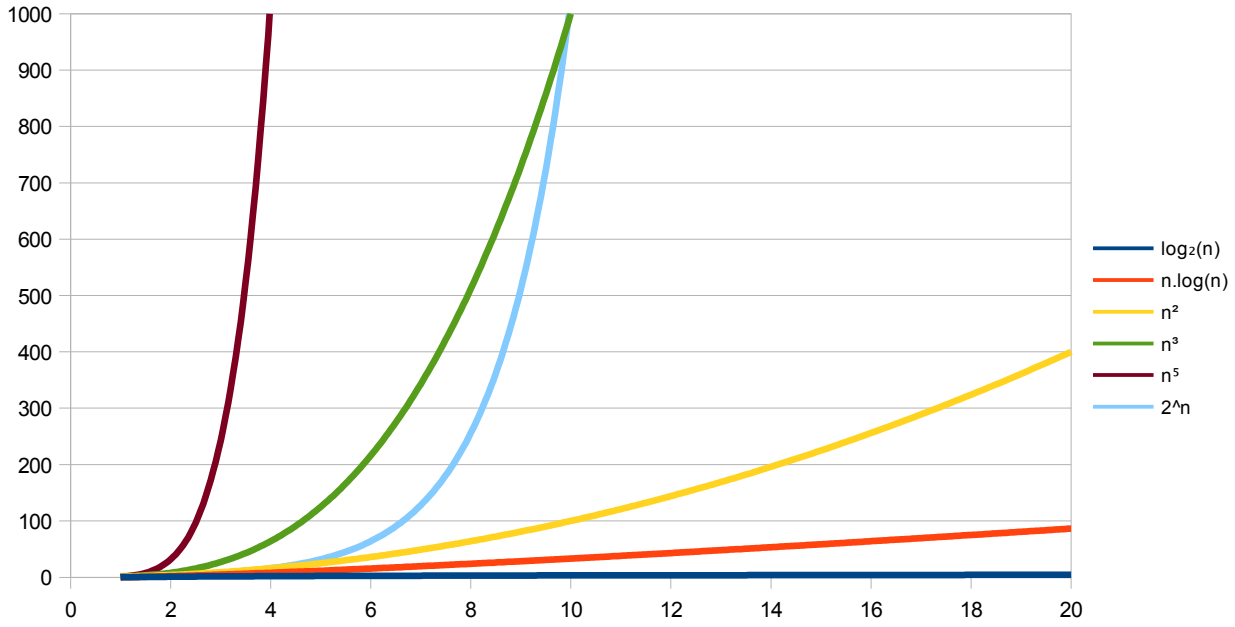


*Figure 6 :Fonctions complexité:croissances comparées.*

On peut également apercevoir les différences au niveau du temps nécessaire pour executer un algorithme en fonction de leurs compléxités et et la taille des données.

| n | 10 | | 100 | | 1 000 | | 1E+06 | | 1E+09 | |
|---|---|---|---|---|---|---|---|---|---|---|
| $Log_2(n)$ | 3,32E-009 | $10^{-9}$ s | 6,64E-009 | $10^{-9}$ s | 9,97E-009 | $10^{-8}$ s | 1,99E-008 | $10^{-8}$ s | 2,99E-008 | $10^{-8}$ s |
| n | 1,00E-008 | $10^{-8}$ s | 1,00E-007 | $10^{-7}$ s | 1,00E-006 | $10^{-6}$ s | 1,00E-003 | $10^{-3}$ s | 1,00E+000 | 1 s |
| n.log(n) | 3,32E-008 | $10^{-8}$ s | 6,64E-007 | $10^{-6}$ s | 9,97E-006 | $10^{-5}$ s | 1,99E-002 | $10^{-2}$ s | 2,99E+001 | 30 s |
| $n^2$ | 1,00E-007 | $10^{-7}$ s | 1,00E-005 | $10^{-5}$ s | 1,00E-003 | $10^{-3}$ s | 1,00E+003 | 17 min | 1,00E+009 | 32 ans |
| $n^3$ | 1,00E-006 | $10^{-6}$ s | 1,00E-003 | $10^{-3}$ s | 1,00E+000 | 1 s | 1,00E+009 | 32 ans | 1,00E+018 | $3.10^8$ siècles |
| $n^5$ | 1,00E-004 | $10^{-4}$ s | 1,00E+001 | 10 s | 1,00E+006 | 11,5 jours | 1,00E+021 | $3.10^{11}$ siècles | 1,00E+036 | |
| $2^n$ | 1,02E-006 | $10^{-6}$ s | 1,27E+021 | $4.10^{11}$ siècles | 1,07E+292 | | | | | |

*Figure 7 :Temps d 'execution d'un algorithme en fonction de la compléxité et taille des données en entreé.*

# 7- Quelques exemples d'algorithme et calcul de complexité

Ces exemples d 'algorithme sont donnés en python ,Java

**-Algorithme simple sans structure itérative ou de contrôle**
Pour calculer la compléxité ici de l'algorithme on doit compter le nombre d'opérations successives le composant .
Si on prend , une fonction toute simple dont l'objectif est de transformer un nombre donné en seconde en un nombre converti en heures , minutes , secondes :

```
def convertir (secondes) :
heure= secondes //3600
minutes = (secondes- 3600*h) // 60
s = seconde% 60
  return heure,minutes,s
```

*Ici on a donc 5 opérations et trois affections : la complexité O(n) = 8 ;*

**-Algortihme simple avec structure conditionnelle**

Ici , on doit dénombrer le nombre de conditions du test.Puis on compte pour chaque opération élementaire de chaque alternative , on prendra le maximum pour obtenir la compléxité dans le pire des cas.Si dessous l'agoithme de calcul de puissance :

```
def puissanceMoinsUn(n):
  if n%2==0:
    res = 1
  else:
    res = -1
  return res
```

*On a une affectation pour chaque alternative , et chacun possédé une affection : la compléxité est donc O(n) = 3*

**-Rechercher si un element x est présent dans une Liste L de taille n :**

```
def present(x,L):
        for y in L:
                if x==y:
                        return True
         return False
```

*Dans le meilleur de cas : 0(1)*
*Dans le pire des cas : 0(n)*

**-Tri a bulles**

```
public static void tribulle(int[]t, int l,int r){
for(int i=l;i<=r;i++)
                    for(int j=r;j>i;j--)
                    if(t[j]<t[j-1])echange(t,j-1,j);
}
```

Ce tri fera  n *(n-1)/2comparaisons.

*Il sera de compléxité  O(n²) dans le pire cas, le meilleur cas et dans le cas moyen.*

**-Le tri rapide ou Quick Sort**

```
def trirapide(L):
    """trirapide(L): tri rapide (quicksort) de la liste L"""
    def trirap(L, g, d):
        pivot = L[(g+d)//2]
        i = g
        j = d
        while True:
            while L[i]<pivot:
                i+=1
            while L[j]>pivot:
                j-=1
            if i>j:
                break
            if i<j:
                L[i], L[j] = L[j], L[i]
            i+=1
            j-=1
        if g<j:
            trirap(L,g,j)
        if i<d:
            trirap(L,i,d)

    g=0
    d=len(L)-1
    trirap(L,g,d)
```

*O(n log n) est la complexité moyenne*
*O( n²) esl la complexité dans le pire des cas.*

# Conclusion

Choisir et appliquer de façon rigoureuse des méthodes pour définir un algorithime sont nécessaires pour en faire des algorithmes efficaces. Nous avons vu qu'il faut dissocier la performance absolue d'un programme au matériel , elle doit être dépendante de la façon dont on va réaliser et spécifier les algorithmes.

Les qualités d'un algorithme sont essentiellement sa maintenabilité et sa rapidité , cependant il faut mettre en avant la rapidité si et seulement si on y gagne en complexité.

De façon générale , la complexité d'un algorithme est le nombre d'étapes élémentaires de calcul indispensable pour calculer la sortie dans le pire des cas à partir d 'une entrée N.

Les informaticiens ont estimés que les algorithmes polynomiaux c'est à dire de compléxité polynomiale étaient les seuls raisonnables.

# Bibliographie

## ***Ressources internet :***

**[1]** wikipédia

**[2]** openclassrooms

**[3]** http://alexandre.boisseau.free.fr/Prive/WWW/InfoPCSI/resume10.pdf

**[4]** http://www.fil.univ-lille1.fr/~tison/AAC/C09/C1_2_res.pdf

**[5]** https://www.supinfo.com/cours/2ADS/chapitres/01-notion-complexite-algorithmique

**[6]** https://fr.slideshare.net/chahrawoods/cours-algorithmique-et-complexite-complet