Chapitre 1 : Algorithmique et complexité

1. Introduction

Un algorithme est une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème. Le temps d'exécution d'un algorithme dépend des facteurs suivants :

- **Les données du programme:** généralement lorsque la taille de données traitées par le programme augmente son temps d'exécution augmente aussi.

  Exemple: le tri d'un tableau de 10 éléments prend un temps inferieur au temps du tri d'un tableau de 100 éléments.

- **La qualité du compilateur (langage utilisé):**

  Exemple: les programmes écrits en Java sont généralement plus lents que ceux écrits en C ou en C++.

- **La machine utilisée:** la vitesse de processeur, la taille du mémoire, mémoire cache..etc**.**
- **La complexité de l'algorithme lui-même :** le facteur le plus important

Par le calcul de la complexité, on cherche à mesurer la vitesse ou le temps d'exécution d'un algorithme indépendamment de la machine et du langage utilisés, mais uniquement en fonction de la taille des données que l'algorithme doit traiter.

2. Notion de la complexité

2.1. Définition:

La complexité d'un algorithme est la mesure du nombre ***d'opérations élémentaires*** qu'il effectue sur ***un jeu de données***. La complexité est exprimée comme une fonction (f(N), C(N),…) de la taille du jeu de données.

- ***Opération élémentaire:*** Une opération élémentaire est une opération dont le temps d'exécution est indépendant de la taille n des données tel que l'affectation, la lecture, l'écriture, la comparaison (<, >, =,…), les opérations arithmétiques ...etc.
- ***La taille de données N:*** est le nombre des éléments traités par l'algorithme.

  Exemples:
  - Dans le cas de tri d'un tableau, ***N*** **est le nombre des éléments du tableau.**

- Dans le cas de calcul d'un terme d'une suite, **N** est **l'indice du terme calculé.**
- Dans le cas de calcul de la somme des éléments d'une matrice de n*m éléments, **la taille de données est n*m.**

2.2. Objectif

L'objectif est de trouver la fonction (f(N), C(N), T(N)…) qui exprime le nombre d'opérations effectuées par l'algorithme en fonction de la taille de données N.

2.3. Exemple illustratif

Soit l'algorithme somme suivant qui calcule la somme des N premiers entiers.

Algorithme: somme N, i, S : entier	La taille des données est N.
Début	L'algorithme somme effectue:
i ←1;	1 affectation (i ← 0)
S ←0;	1 affectation (S←0)
Tantque i ≤N faire	N + 1 comparaison (≤N);
S ← S+i;	N additions (s+i); N affectations (S← S+i);
i ← i+1;	N additions (i+1); N affectations (i←i+1);
Fintantque	
Ecrire (s);	1 affichage (écrire (s)):
fin	**Total : 5N+4 opérations élémentaires**
	f(n) = 5N+4

2.4. Pourquoi calculer la complexité des algorithmes

1. Calculer le temps et l'espace mémoire nécessaire pour l'exécution d'un programme.
2. Vérifier si un algorithme est exécutable dans un temps raisonnable.
3. Comparer deux algorithmes résolvant le même problème pour choisir le meilleur.
4. Vérifier si un algorithme est optimal. C.-à-d. vérifier s'il existe ou non un algorithme plus performant.

3. Règles de calcul de la complexité d'un algorithme

3.1. Instruction simple: la complexité d'une instruction simple est le nombre d'opérations élémentaires dans l'instruction.

Exemple: dans l'instruction y← 3+5*x; nous avons 3 opérations élémentaires, une affectation, une addition et une multiplication.

3.2. Algorithmes sans structures de contrôle (séquence d'instructions): La complexité d'une séquence d'instructions égale la somme des complexités des instructions de la séquence.

Exemple:

Algorithme somme

x, y, z : entiers

Début

Lire (x);
Lire (y);
z←x+y;
Écrire (z);

} **5 opérations**

Fin

3.3. Le cas des structures conditionnelles : La complexité d'une instruction <si – alors – sinon> égale la somme de la complexité de l'évaluation de la condition et la plus grande complexité entre le bloc alors et le bloc sinon.

Si (*condition)* alors

…….… } bloc alors

Sinon

…… } bloc sinon

Finsi

La complexité = C_1+ max (C_2, C_3)

Où :

- **C_1** est le nombre d'opérations exécutées pour l'évaluation de la condition.
- **C_2** est le nombre d'opérations exécutées si la condition est vraie (le bloc alors)
- **C_3** est le nombre d'opérations exécutées si la condition est faux (le bloc sinon)

Exemple:

Algorithme val_abs

x: entiers

Début

Lire (x) ;
Si x<0 alors
x← -x ;
Ecrire (x) ;
Sinon
Ecrire (x) ;
Finsi
Ecrire (x) ;

Fin

La complexité = C_1+ max (C_2, C_3)

Où :

- **C_1=1** (comparaison x<0)
- **C_2**=3 (-x,x←-x , écrire)
- **C_3**= 1 (écrire)

La complexité = 1+ max (3, 1)=4

3.4. Le cas des structures itératives : la complexité d'une boucle égale la somme, sur toutes les itérations, de la complexité de l'évaluation de la condition de sortie et la complexité du bloc d'instructions du corps de la boucle.

Exemple

Tantque i<=N faire

S←s+i;

I←i+1;

Fin tantque

- Dans boucle tantque nous avons *N* itérations
- La comparaison i<=N s'effectue N+1 fois (N itérations de la boucle plus une fois pour sortir de la boucle)
- Dans le bloc d'instruction nous avons 4 opérations chacune s'effectue N fois. Donc 4*N.
- La complexité de la boucle est donc 5N+1

3.5. Sous programmes (fonctions et procédures) : La complexité de l'appel d'un sous programme égale la somme des complexités de ses instructions.

Exemple:

Algorithme SommeTab
T : Tableau de N entier
N,S: entiers
Procedure lireVe (var T : tableau d'entier, N : entier)
i: entier ;
Début
Pour i allant de 1 à N faire
Lire (T[i]) ;
Fin pour
} Complexité de lireVe est : 3N+2
Fin
Fonction somme (T : tableau d'entier, N : entier) : entier
i, S: entier ;
Début
S←0 ;
Pour i allant de 1 à N faire
S←S+T[i] ;
Fin
Retourne (S) ;
Fin
} Complexité de somme est : 4N+4

Début
Lire (N) ; ----------> 1
LireVe (T, N) ; ----------> 3N+2
S←Somme(T,N) ; ----------> 1+4N+4
Ecrire (S) ; ----------> 1
Fin ***f*(N) = 7N+9**

3.6. Les algorithmes récursifs : La complexité d'un algorithme récursif se fait par raisonnement et démonstration par récurrence.

Exemple : Soit la fonction récursive suivante :

```
Fonction factorielle (n : entier): entier
Début      Op1
        Si n=0 alors
                retourne 1;    Op2
        Sinon    Op3      Op4          Op5
                retourne (n* factorielle(n-1))
        fin si
Fin
```

Posons C(n) le temps d'exécution nécessaire pour un appel à factorielle (n).

C (0) = Op1+ Op2 = 2

C(1) = Op1 +Op3+ Op4+ Op5+ C(0) = b + C(0) (b= Op1 +Op3+ Op4+ Op5+ Op6)

C(2) = Op1 +Op3+ Op4+ Op5+ + C(1) = b + b + C(0) = 2b+ C(0)

C(3) = Op1 +Op3+ Op4+ Op5+ C(2) = 3b+ C(0)

.

.

C(n-1)= (n-1) b +C(0)

C(n) = Op1 +Op3+ Op4+ Op5+C(n-1)=

= b + (n-1) b +C(0) = nb +C(0)= 4n+2

4. Exemple de calcule de la complexité

```
Algorithme SommeTab
        T : Tableau de N entier
        N,S: entiers
Procedure lireVe (var T : tableau d'entier, N : entier)
        i: entier ;
Début
    Pour i allant de 1 à N faire   Op1, Op2, Op3
        Lire (T[i]) ;   Op4
    Fin pour
Fin
Fonction somme (T : tableau d'entier, N : entier) : entier
        i, S: entier ;
Début
        S←0 ;   Op1
  Pour i allant de 1 à N faire      Op2, Op3, Op4
        S←S+T[i] ;   Op5, Op6
  Fin
        Retourne (S) ;   Op7
Fin

Début                    Op1
        Lire (N) ;
        LireVe (T, N) ;   Appel lireVe
        S←Somme(T,N) ;   Op2, Appel Somme
        Ecrire (S) ;   Op3
Fin
```

Opération	Nombre
Op1	1
Op2	N+1
Op3	N
Op4	N
Total	3N+2

Opération	Nombre
Op1	1
Op2	1
Op3	N+1
Op4	N
Op5	N
Op6	N
Op7	1
Total	4N+4

Opération	Nombre
Op1	1
Appel lireVe	3N+2
Op2	1
Appel Somme	4N+4
Op3	1
Total	7N+9

5. Complexité au mieux et au pire

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer:

- La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.
- La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).
- La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, **on utilise la complexité au pire**, car on veut borner le temps d'exécution.

Exemple illustratif: soit la fonction de recherche séquentielle d'un élément dans un tableau de n entiers

```
Fonction recherche (n, x: entier, tab: tableau d'entier): booléan
        i: entier; b: booléan;
début
    i ← 1;  b←faux;
    tantque (i <= n) et (b=faux) faire
     si (tab[i] = x) alors
              b=vrai;
    finsi
              i←i+1;
    fintantque
    Retourne b;
Fin
```

- Le paramètre de complexité est la taille du tableau d'entrée.
- Le nombre de tours de boucles varie selon que x est dans le tableau ou pas, et selon l'endroit où x est présent.
  - Si x est dans la première case du tableau : 1 tour de boucle avec la condition b=vraie
  - Si x est dans la deuxième case du tableau : 1 tour de boucle avec la b= faux et 1 tour de boucle avec b= vrai

    . . .
  - Si x est dans dernière case du tableau : N-1 tours de boucle avec la b= faux et 1 tour de boucle avec b= vrai. Donc N tours
  - Si x n'est pas dans le tableau : N tours de boucle avec la condition b= faux.

6. Comportement asymptotique et notation landau (*O)*

Le but de cette partie va être de comparer les complexités calculées avec des **fonctions de référence** (puissance, logarithme, exponentielle, etc.) en utilisant la notion borne supérieure et la notation landau (*O*).

Borne supérieure asymptotique

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

$$\exists c>0,\ \exists n0>0 \text{ tel que } \forall n>n_0,\ f(n) < c \times g(n)$$

Cela signifie qu'à partir d'un certain rang la fonction f est majorée par une constante fois la fonction g. Il s'agit donc d'une situation de domination de la fonction f par la fonction g.

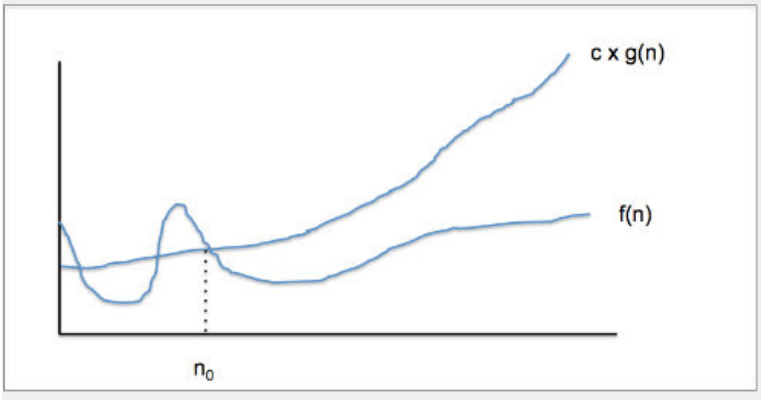


Figure 1.1. Interprétation graphique de la notion de grand O

Exemple

- Si T(n)=4 alors T(n)=O(1)T(n)=O(1). Pour le prouver, prendre par exemple c=5 et n_0=1.
- Si T(n)=3n+2 alors T(n)=O(n)T(n)=O(n). Pour le prouver, prendre par exemple c=4 et n0=2.
- Si T(n)=2n+3 alors T(n)=O(n2). Pour le prouver, prendre par exemple c=3c=3 et n0=1n0=1.

La notation *O* donne donc une majoration du nombre d'opérations exécutées (temps d'exécution) par le programme P.

7. Classes de complexité algorithmique

Les **complexités** algorithmiques que nous allons calculer doit être exprimées comme des **grand *O*** de **fonctions de références**. Cela va nous permettre de les **classer**. Les algorithmes appartenant à une **même classe** seront alors considérés comme de **complexité équivalente**. Cela signifiera que l'on considèrera qu'ils ont la **même efficacité**. Sous-dessous les classe de **complexités de référence**. Ces complexités sont rangées dans l'ordre croissant.

1) **T(n) = *O*(1), temps constant :** temps d'exécution indépendant de la taille des données à traiter.
2) **T(n) = *O*(log(n)), temps logarithmique:** on rencontre généralement une telle complexité lorsque l'algorithme casse un gros problème en plusieurs petits, de sorte que la résolution d'un seul de ces problèmes conduit à la solution du problème initial.

3) **T(n) = *O*(n), temps linéaire:** cette complexité est généralement obtenue lorsqu'un travail en temps constant est effectué sur chaque donnée en entrée.

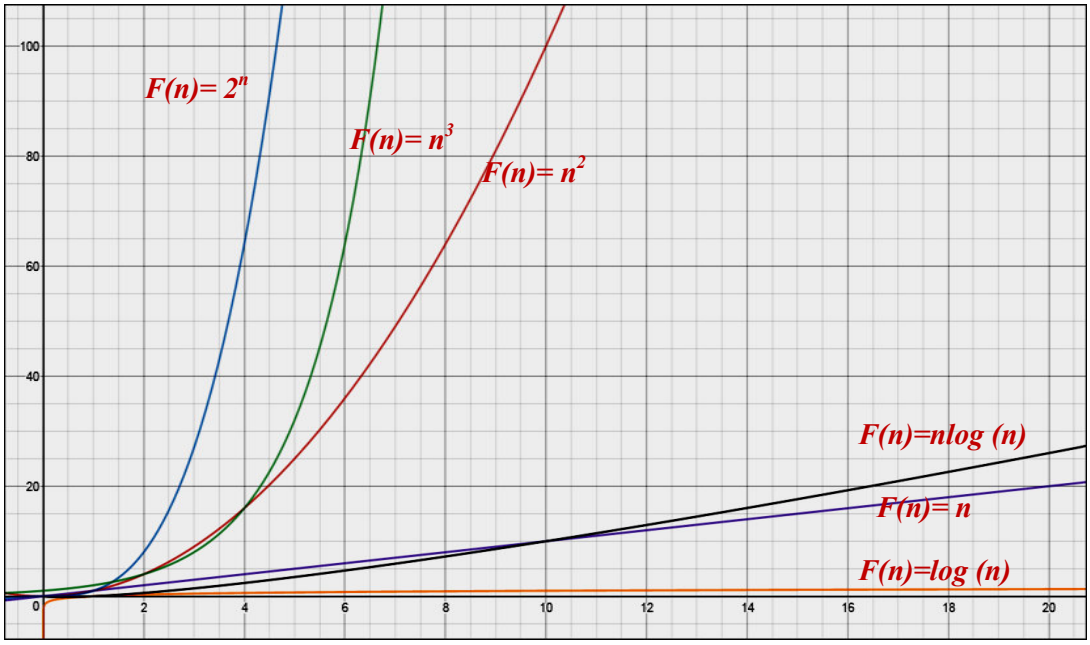
4) **T(n) = *O* (n log(n)):** l'algorithme scinde le problème en plusieurs sous-problèmes plus petits qui sont résolus de manière indépendante. La résolution de l'ensemble de ces problèmes plus petits apporte la solution du problème initial.

5) **T(n) = *O*(n^2), temps quadratique :** apparaît notamment lorsque l'algorithme envisage toutes les paires de données parmi les n entrées (ex. deux boucles imbriquées)

   Remarque : *O* (n^3) temps cubique, *O*(n^k)polynomial

6) **T(n) = *O* (2^n), temps exponentiel :** souvent le résultat de recherche brutale d'une solution.

Pour bien fixer les idées sur le comportement de ces fonctions, voici le tracé de leurs courbes.



Les temps d'exécution selon la taille de donnée

temps	constant $\theta(1)$	logarithmique $\theta(\log_2 N)$	linéaire $\theta(N)$	$\theta(N \log_2 N)$	polynomial $\theta(N^k)$ quadratique ($k=2$)	$k=3$	exponentiel $\theta(2^N)$
$N = 100$	$1\mu s$	$6.6\mu s$	0.1 ms	0.6 ms	10 ms	1s	10^{16} ans
$N = 1000$	$1\mu s$	$9.9\mu s$	1 ms	9.9 ms	1 s	16.6 min	∞
$N = 10^4$	$1\mu s$	$13.3\mu s$	10 ms	0.1 s	100 s	11.5 jours	∞
$N = 10^5$	$1\mu s$	$16.6\mu s$	0.1 s	1.6 s	2.7 heures	31 ans	∞
$N = 10^6$	$1\mu s$	$19.9\mu s$	1 s	19.9 s	11.5 jours	$3\ 10^4$ ans	∞