CHAPITRE I

Introduction à la Complexité de Calcul

L'art de concevoir un algorithme

Plusieurs algorithmes pour un même énoncé de problème

Considérons l'exemple de calcul des fréquences de nombres entiers positifs fournis dans un tableau. Les fréquences calculées seront affichées pour chacun de ces nombres. Les trois algorithmes suivants sont développés pour solutionner le problème.

Algorithme Calcul-Fréquences1 *;*
entrée : *un tableau T[1..n] d'***entiers** *;*
sortie : *le nombre d'occurrences de chaque élément du tableau ;*

```
const   n=500 ;
var     i, j, fréquences : entiers ;
        trouve : booléen ;

début
        pour    i := 1 à n faire
         début  j :=1 ;
                trouve := faux ;
                tant que (j<i) et (non trouve) faire
                        si T[i] = T[j] alors trouve := vrai
                                       sinon j := j+1 ;
                si (non trouve) alors
                  début  fréquences := 1 ;
                        pour j := i+1 à n faire
                          si T[i] = T[j] alors fréquences := fréquences + 1 ;
                        écrire ('fréquence de' T[i] '=' fréquences) ;
                 fin
          fin
fin
```

```
Algorithme Calcul-Fréquences2 ;
entrée : un tableau T[1..n] d'entiers ;
sortie : le nombre d'occurrences de chaque élément du tableau ;

const   n=500 ;
var     i, j, fréquences : entiers ;

début
        pour    i := 1 à n faire
                si T[i] <> -1 alors
                  début  fréquences := 1 ;
                         pour j := i+1 à n faire
                           si T[i] = T[j] alors
                                  début fréquences := fréquences + 1 ;
                                        T[j] := -1 ;
                                  fin ;
                        écrire ('fréquence de' T[i] '=' fréquences) ;
                fin ;
fin
```

```
Algorithme Calcul-Fréquences3 ;
entrée : un tableau T[1..n] d'entiers ;
sortie : le nombre d'occurrences de chaque élément du tableau ;

const   max = 500 ;
var     fréquences : tableau[1..max] d'entiers ;
        i : 1..max ;

début
        pour    i := 1 à max faire fréquences[i] := 0 ;
        pour i := 1 à n faire fréquences[T[i]] := fréquences[T[i]] + 1 ;
        pour    i := 1 à max faire
         si fréquences[i] <> 0 alors écrire ('fréquence de', i, '=', fréquences[i]) ;
fin
```

Analyse et calcul de la complexité par l'exemple

Nous calculons dans ce qui suit la complexité de chacun des trois algorithmes en considérant la donnée extrême qui nécessite le plus de temps d'exécution. Ce cas est appelé 'cas le plus défavorable' ou 'pire cas'. Pour le premier algorithme par exemple, le pire cas correspond à la donnée où tous les nombres fournis en entrée sont distincts les uns des autres.

Algorithme Calcul-Fréquences1

Le principe de l'algorithme est de considérer chaque élément du tableau un par un et de :

- vérifier d'abord en consultant les éléments qui se trouvent à gauche de l'élément courant si ce dernier existe, auquel cas il a déjà été traité et donc sa fréquence a déjà été calculée.
- ensuite dans le cas où l'élément courant n'existe pas à gauche, parcourir tous les éléments qui se trouvent à droite pour calculer sa fréquence.

L'algorithme possède trois boucles : deux boucles internes séquentielles, les deux imbriquées dans une boucle externe. L'algorithme procède à un balayage du tableau T *n* fois et pour chaque élément du tableau, dans le pire des cas il parcourt tout le tableau une deuxième fois pour les *(n-1)* autres éléments. Le nombre total d'exécutions des instructions des boucles internes est donc égal à $n*(n-1)= n^2-n$. La complexité est de l'ordre de n^2 et est notée $O(n^2)$.

Algorithme Calcul-Fréquences2

Cet algorithme est une version améliorée du premier algorithme. A chaque fois qu'un élément est traité, sa valeur est mise à -1. De cette manière, la recherche de l'élément courant à gauche ne sera plus nécessaire et par conséquent la première boucle interne est supprimée. Seule la deuxième boucle interne reste et est exécutée (n-i) fois à chaque itération de la boucle externe. La complexité est donc égale à $\sum_{i=1}^{i=n}(n-i)= \frac{n(n-1)}{2}= \frac{n^2-n}{2}$ et est donc en $O(n^2)$. Faisons remarquer que la complexité a été réduite de moitié mais que l'ordre est le même que celui de l'algorithme précédent.

Algorithme Calcul-Fréquences3

Dans cet algorithme, nous utilisons un deuxième tableau appelé *fréquences* dans lequel nous calculons les fréquences des nombres. Nous supposons que les nombres sont compris entre 1 et *max*, la valeur maximale que peut avoir un élément du tableau et que $max > n$. L'algorithme commence par mettre à 0 tous les éléments du tableau *fréquences*, puis il parcourt le tableau *T* en incrémentant la fréquence de l'élément courant *i* qui se trouve dans le tableau fréquences à la position *T[i]*.

Il est clair que la complexité dans ce cas est $O(max)$ du moment que l'algorithme contient trois boucles séquentielles (une pour l'initialisation du tableau *fréquences* de taille *max*, une deuxième pour le calcul des fréquences avec un nombre d'itérations égal à *n* et la dernière pour l'impression des résultats) avec un nombre d'itérations égal à *max*.

En conclusion, nous avons développé trois algorithmes différents pour le même problème avec des complexités différentes, les deux premières solutions ont une complexité du même ordre à savoir $O(n^2)$ et la troisième une complexité plus réduite qui est $O(max)$. Si max est très grand par rapport à n, l'espace mémoire

requis pour le tableau des fréquences sera très important. On peut donc faire remarquer que la complexité temporelle de l'algorithme a été réduite au détriment de l'espace mémoire.

Problèmes et instances de problèmes

Description d'un problème

Un exemple simple de problème qui sera traité dans ce cours est le problème de tri défini comme suit :

Donnée : un ensemble de *n* nombres entiers.

Question : trier les *n* nombres entiers par ordre croissant.

Un autre exemple concerne le problème du voyageur de commerce (Traveling Salesman Problem en anglais ou TSP en abrégé). Etant donnée une carte géographique de villes et des distances les reliant, un voyageur a pour objectif de visiter toutes les villes une et une seule fois de façon que la distance parcourue au total n'excède pas une constante égale à *k*. La description formelle du problème est comme suit :

Donnée : un ensemble de *n* villes et un ensemble de distances des routes reliant certaines villes entre elles.

Question : Existe-il un chemin englobant toutes les villes une et une seule fois tel que la somme des routes constituant le chemin est inférieure ou égale à *k* ?

Instance d'un problème

Une instance de problème est une donnée concrète du problème respectant la spécification de la description du problème. Une instance du problème de tri peut être la suivante :

Donnée : 5 nombres entiers, en l'occurrence : 12, 43, 28, 63, 7.

Question : trier les 5 nombres par ordre croissant.

Une instance du problème TSP est comme suit :

Donnée : 7 villes : Alger, Oran, Constantine, Annaba, Chlef, Tamanrasset et Adrar et la matrice des distances montrée sur la table 1.1.

	Alger	**Oran**	**Constantine**	**Annaba**	**Chlef**	**Tamanrasset**
Oran	420					
Constantine	390	790				
Annaba	540	940	160			
Chlef	200	215	580	730		
Tamanrasset	1920	2050	2050	2200	1960	
Adrar	1420	1250	1560	1700	1350	1020

Table 1.1. Distances entre des villes d'Algérie

Taille d'un problème

A chaque problème est donc associée une taille qui mesure la quantité des données en entrée. Elle est exprimée à l'aide d'une ou de plusieurs variables ou paramètres.

Algorithme et complexité

L'approche **théorique** présentée dans cet ouvrage, consiste à déterminer mathématiquement la quantité de ressources (en temps et/ou en espace) nécessaire à l'exécution des algorithmes en fonction de la taille des données considérées.

Complexité théorique

Elle permet d'éviter l'effort de programmer inutilement un algorithme inefficace, comme elle peut renseigner sur l'efficacité des algorithmes étudiés quel que soit la taille des entrées.

Complexité en temps

Si un algorithme s'exécute en un temps $c*n^2$ pour une entrée de taille n, nous dirons que l'algorithme a une complexité de l'ordre de n^2 que l'on note $O(n^2)$. Plus précisément, un algorithme qui s'exécute en un temps égal à $g(n)$ a une complexité temporelle en $O(f(n))$ s'il existe une constante c telle que $g(n) <= c*f(n)$ pour tout n. De cette manière, on voit que la complexité n'est pas une mesure exacte mais exprime le **comportement asymptotique** du temps pour les données de très grande taille en utilisant la notation O dite de Landau.

Complexité en espace

Tout comme la complexité temporelle, la complexité spatiale est une mesure théorique mais qui permet d'estimer en fonction de la taille du problème l'espace mémoire requis par l'algorithme pendant son exécution.

Comparaison entre ordres de complexités

Effet de la performance de la machine

Considérons six algorithmes A1-A6 pour résoudre un même problème, avec leur complexité respective exprimée en nombre d'unités nécessaires pour l'exécution d'une entrée de taille *n*. L'effet d'une multiplication de la vitesse d'exécution par un facteur de 10 est donné par la table 1.2 :

Algorithme	**Complexité temporelle**	**Taille maximale traitée par M1**	**Taille maximale traitée par M2**
A1	$O(\log_2 n)$	s_1	s_1^{10}
A2	$O(\sqrt{n})$	s_2	$100s_2$
A3	O(n)	s_3	$10s_3$
A4	$O(n^2)$	s_4	$3.16s_4$
A5	$O(n^3)$	s_5	$2.15s_5$
A6	$O(2^n)$	s_6	$s_6 + 3.3$

Table 1.2. Effet de la performance de la machine sur la quantité de données traitées

Si l'on prend l'exemple de l'algorithme A_1, la taille s_1 est calculée par M_1 en un temps égal à $(log_2 s_1)$. Puisque M_2 est 10 fois plus rapide que M_1, pendant ce temps, M_2 traite une taille x en un temps égal à $(log_2 x)/10$. On a alors :

$\log_2 s_1 = (\log_2 x)/10$, d'où $x = {s_1}^{10}$.

Le tableau montre la taille des problèmes qu'on peut résoudre avec une telle vitesse. Notons que la taille pour l'algorithme A6 n'augmente que de 3.3 alors que celle de l'algorithme A4 est multipliée par 3.16.

Effet de l'amélioration de l'algorithme

Au lieu d'augmenter la performance de la machine, considérons maintenant une seule machine et étudions l'effet des différentes complexités d'algorithmes sur la taille traitée pendant le même intervalle de temps et prenons comme unité de temps un millième de secondes. La table 1.3 montre les tailles maximales traitées par la machine respectivement pendant une seconde, une minute puis une heure pour chacun des algorithmes A1 à A6.

Ces résultats sont plus impressionnants que ceux obtenus avec un changement de calculateur d'une vitesse plus grande. En l'occurrence, en prenant comme unité de comparaison la minute, l'algorithme A3 traite 245 fois plus de données que l'algorithme A4, qui lui traite 6 fois plus que A5. L'écart est encore plus significatif lorsque la durée de temps est plus grande.

Nous en concluons que l'étude de la complexité des algorithmes est une mesure importante de l'efficacité d'un algorithme.

Algorithme	**Complexité temporelle**	**Taille traitée pendant :**		
		1seconde	**1minute**	**1heure**
A1	$O(\log_2 n)$	2^{1000}	>>>	>>>
A2	$O(\sqrt{n})$	10^6	>>>	>>>
A3	O(n)	1000	$6*10^4$	$3.6*10^6$
A4	$O(n^2)$	31	244	1897
A5	$O(n^3)$	10	39	153
A6	$O(2^n)$	9	15	21

Table 1.3. Effet de la complexité de calcul sur la quantité de données traitées

Notion d'efficacité d'algorithmes

Il est clair que les algorithmes ayant des complexités exponentielles sont incapables de résoudre un problème lorsque la taille de ce dernier est importante.

Complexité empirique

La complexité empirique mesure le temps réel nécessaire à l'exécution d'un programme. Elle dépend des caractéristiques de la machine sur laquelle s'exécute le programme et du langage de programmation utilisé.

Modèles de calcul

Modèle universel de Von Neumann

La plupart des calculateurs repose sur l'architecture de Von Newman dont les principales composantes sont :

- Une mémoire centrale pour stocker le programme et ses données avant son exécution.
- Une unité d'exécution des instructions.

- Des périphériques d'entrée/sortie :
    - Des unités de lecture des données (clavier, souris, scanner, …).
    - Des unités d'affichage ou d'impression des résultats (écran, imprimante, table traçante …).

Le modèle de machine à accès aléatoire (Random Access Memory ou RAM), à processeur unique où les instructions sont exécutées de manière séquentielle est considéré comme référence.

Pour calculer la complexité d'un programme, deux hypothèses peuvent être considérées. La première appelée **critère du coût uniforme,** repose sur les hypothèses suivantes :

- le temps d'exécution d'une instruction coûte 1 unité de temps.
- un mot de la mémoire coûte 1 unité d'espace.

La deuxième hypothèse plus réaliste et appelée **critère du coût logarithmique,** tient compte de la représentation des instructions et des données en mémoire et plus précisément du nombre de bits les représentant. (*|Log n| +1)* bits sont requis pour représenter un entier *n* en mémoire, d'où l'appellation de coût logarithmique.

Langage évolué

Considérons l'exemple 1.2 du calcul de la suite de Fibonnacci.

Exemple 1.2 : Soit la suite de Fibonnacci donnée par la formule suivante :

$$\begin{cases} f(0) = 0 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2) \end{cases}$$

Nous nous intéressons à l'écriture d'un programme en pseudo-Pascal pour calculer la suite de Fibonnacci. Le langage Pascal est utilisé ici car il est proche de l'écriture algorithmique. Une version simple du programme correspond à un programme récursif déduit de la définition et qui se présente comme suit :

```
Programme Fibonnacci;

fonction f ;
entrée : n: entier ;
sortie : la valeur f(n) ;

début
        si (n=0) alors f := 0
                sinon si (n=1)  alors f := 1
                                sinon f := f (n-1) + f(n-2);
fin ;

début
        écrire (f(30)) ;
fin.
```

La fonction *f* est récursive et nécessite beaucoup de calcul. Le calcul de *f(n)* nécessite à chaque fois 2 appels de la même fonction *f* jusqu'à ce que les paramètres d'appel soient égaux à 1 et 0. Le nombre d'appels est à chaque fois multiplié par 2. La complexité est donc en $0(2^n)$. La démonstration peut se faire par récurrence. Pour plus de précision, la relation de Fibonnacci peut être transformée par la formule de Binet pour déduire la complexité de calcul de *f(n)*.

La version itérative du programme est plus efficace et s'écrit comme suit :

```
Programme Fibonnacci ;
var     T[1..n] : tableau d'entiers ;
        i: entier;

début   n := 30;
        T[1] := 0;
        T[2] := 1;
        pour (i = 3 à n) faire  T[i] := T[i-2] + T[i-1];
        écrire (T[n];
fin.
```

Il est facile de voir que la complexité de ce programme est *O(n)*.

Considérons maintenant l'exemple 1.3 du programme de reconnaissance du langage $L= \{0^k1^k,\ \ k$ entier positif ou nul$\}$.

Exemple 1.3 :

Le programme qui reconnaît le langage L est le suivant :

```
programme L;
var c, compteur : entier ;

début
        lire(c);
        compteur := 0 ;
        tant que ( c= 0) faire
                début
                        compteur := compteur + 1 ;
                        lire(c) ;
                fin ;
        tant que (c = 1) faire
                début
                        compteur := compteur - 1 ;
                        lire(c) ;
                fin ;
        si (compteur = 0) alors signaler ('succès')
                          sinon  signaler (' échec')
fin.
```

La complexité du programme *L* selon toujours le critère du coût uniforme, est égale au nombre total d'instructions exécutées pour un mot de longueur *n* pour le pire cas. Ces instructions se composent de :

- 2 instructions d'initialisation
- 2 instructions pour la première boucle
- 2 instructions pour la deuxième boucle
- 1 instruction de fin de programme

En tout, ce nombre est égal à : $2 + 2(\frac{n}{2}) + 2(\frac{n}{2}) + 1 = 3 + 2n$. La complexité est donc *O(n)*.

Machine de Turing

Une machine de Turing est un outil théorique élémentaire utilisé dans la théorie de la calculabilité et également dans la théorie de la complexité. Elle est définie formellement comme un septuplé :

(Q, T, I, D, b, q_0, q_f) où

Q est l'ensemble des états de la machine,

T est l'alphabet du langage reconnu par la machine,

I est l'alphabet des données,

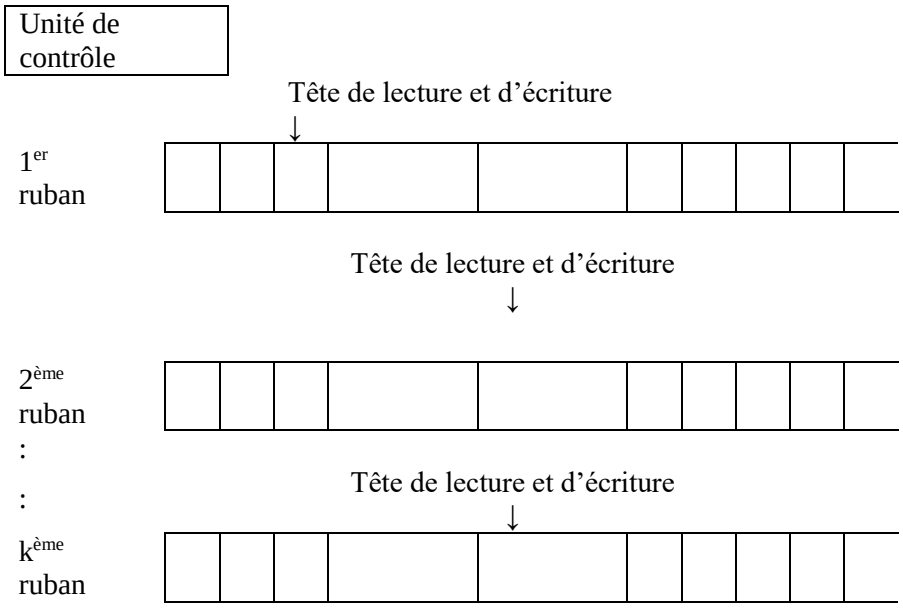
D est la fonction de transition,

b est le caractère blanc,

q_0 est l'état initial et

q_f est un état final. La machine peut avoir plusieurs états finaux.

La machine de Turing est schématisée comme suit :



Pour illustrer le fonctionnement de la machine de Turing, considérons l'exemple 1.4 qui construit une machine de Turing pour le langage 0^k1^k de l'exemple 1.1.

Exemple 1.4 :

Pour le développement de la machine de Turing, il faut définir l'ensemble des états Q, l'alphabet T, l'alphabet I, la fonction de transition, l'état initial et les états finaux.

1ère solution : utilisation d'un seul ruban

Rôles des états de la machine

q0 : état initial, se déplacer à droite et remplacer 0 par b

q1 : se déplacer vers l'extrémité droite

q2 : se déplacer d'un cran vers la gauche et mettre le dernier 1 à blanc

q3 : se déplacer vers l'extrémité gauche

q4 : état final signalant un échec : un 1 est rencontré à l'extrémité gauche

q5 : état final signalant un échec : un 0 est rencontré à l'extrémité droite

q6 : état final signalant l'appartenance du mot au langage

(q0, 0) → (q1, b, R)
(q0, 1) → (q4, b, R)
(q1, 0) → (q1, 0, R)
(q1, 1) → (q1, 1, R)
(q1, b) → (q2, b, L)
(q2, 0) → (q5, b, L)
(q2, 1) → (q3, b, L)
(q2, b) → (q6, 1, S)
(q3, 1) → (q3, 1, L)
(q3, 0) → (q3, 0, L)
(q3, b) → (q0, b, R)
(q4, 1) → (q4, b, R)
(q4, 0) → (q4, b, R)
(q4,b) → (q4, 0, S)
(q5,1) → (q5, b, L)
(q5,0) → (q5, b, L)
(q5,b) → (q5, 0, S)

La complexité temporelle se déduit par le nombre de transitions effectuées pendant le traitement d'une donnée. Le pire cas se produit lorsque la chaine présentée en entrée est traitée complètement. Dans cette situation, les transitions se décomposent comme suit :

- n+1 déplacements vers la droite
- n déplacements vers la gauche
- n-1 déplacements vers la droite
- …
- 1 déplacement

Le nombre total de transitions est donc évalué à :

$$\sum_1^{n+1} i = (n+1)(n+2)/2 = (n^2+3n+2)/2.$$

La complexité est donc en $O(n^2)$.

La complexité spatiale s'exprime par le nombre de cases utilisées au total. Ce nombre est égal à *n* et sert à stocker le mot du langage à vérifier. Aucune autre case n'est exploitée par la suite. La complexité spatiale est donc $O(n)$.

2ème solution : utilisation de deux rubans

On suppose que le mot se présente initialement sur le premier ruban. Le deuxième ruban sert à copier la séquence des 0, tout en remplaçant ceux du premier ruban par des blancs. La machine passe ensuite dans une seconde étape qui consiste à comparer les longueurs respectives des sous chaines des 0 et celles des 1. Nous aurons besoin de définir 5 états avec les rôles suivants :

- q0 : état initial.
- q1 : sert à reporter la séquence des 0 sur le 2ème ruban et de préparer la comparaison avec la séquence des 1.
- q2 : sert à comparer la longueur des séquences respectives des 0 et des 1.
- q3 : état final servant à déclarer un résultat négatif.
- q4 : état final servant à annoncer le succès du traitement.

La fonction de transition est alors la suivante :

(q0, 0, b) → (q1, (0, S), (b, R))
(q0, 1, b) → (q3, (1, S), (b, S))
(q1, 0, b) → (q1, (b, R), (0, R))
(q1, b, b) → (q3, (0, S), (b, S))
(q1, 1, b) → (q2, (1, S), (b, L))
(q2, 1, 0) → (q2, (1, R), (0, L))
(q2, 1, b) → (q3, (b, R), (b, S))
(q2, b, 0) → (q3, (0, S), (b, S))
(q2, 0, b) → (q3, (0, S), (b, S))
(q2, b, b) → (q4, (1, S), (b, S))
(q3, 1, b) → (q3, (b, R), (b, S))
(q3, 0, b) → (q3, (b, R), (b, S))
(q3, b, b) → (q3, (0, S), (b, S))

La complexité temporelle de la machine se mesure par le nombre de transitions effectuées pendant le traitement de la donnée. Dans le cas de notre exemple, le pire cas se produit lorsque le mot est traité en entier, deux cas se présentent : le mot est correct ou bien le mot contient plus de 1 que de 0. Dans ces deux situations, les transitions effectuées sont :

- 1 transition pour marquer l'extrémité droite du 2ème ruban, nécessaire pour la suite du traitement.
- *n0* transitions pour copier la séquence des 0, *n0* étant la longueur de la séquence des 0.
- *n1* transitions pour comparer la séquence des 1 avec la séquence des 0.
- 1 transition pour mettre le résultat sur le 1er ruban.

Au total, le nombre de transitions est égal à $n0 + n1 + 2 = n + 2$, où n est la longueur du mot présenté en entrée. La complexité est donc $O(n)$.

Le nombre de cases utilisées au total étant de : $n+k0$ où $k0$ est le nombre de cases du 2ème ruban occupées par les 0 copiés à partir du 1er ruban. La complexité spatiale est $O(n)$ puisque $k0<=n$ et donc $n+k0 <= 2n$.

Nous remarquons que lorsque l'on utilise un seul ruban, la complexité temporelle est $O(n^2)$ alors qu'elle est O(n) lorsque 2 rubans sont utilisés. Ce résultat est prévisible de par l'énoncé du théorème suivant :