Université Constantine 2
Faculté des Nouvelles Technologies de l'Information et de la Communication
Département Technologie du Logiciel et Systèmes d'Information
ASD – Algorithmique et Structures de Données
Semestre I, Année Universitaire 2014-2015

Chapitre 1 : Notion de complexité algorithmique

Sommaire :

1. Rappel

> Algorithms + Data structures = Programs (Niklaus Wirth)

Un algorithme est une suite finie et non-ambiguë d'opérations ou d'instructions permettant de résoudre un problème. Il provient du nom du mathématicien persan Al-Khawarizmi, le père de l'algèbre.
Un problème algorithmique est souvent formulé comme la transformation d'un ensemble de valeurs, d'entrée, en un nouvel ensemble de valeurs, de sortie.

Exemples d'algorithmes :

1. Une recette de cuisine (ingrédients → plat préparé)
2. La recherche dans un dictionnaire (mot → définition)
3. La division entière (deux entiers → leur quotient)
4. Le tri d'une séquence (séquence → séquence ordonnée)

Un algorithme peut être spécifié de différentes manières :

- en langage naturel,
- graphiquement,
- en pseudo-code,
- par un programme écrit dans un langage informatique

...
La seule condition est que la description soit précise.

On étudiera essentiellement les algorithmes corrects. Un algorithme est (totalement) correct lorsque pour chaque instance, il se termine en produisant la bonne sortie. Il existe également des algorithmes partiellement corrects dont la terminaison n'est pas assurée mais qui fournissent la bonne sortie lorsqu'ils se terminent. Il existe également des algorithmes approximatifs qui fournissent une sortie inexacte mais néanmoins proche de l'optimum.

2. Introduction

Etant donné un problème à résoudre algorithmiquement, une des questions la plus immédiate est de trouver un algorithme efficace, si possible. La question suivante est, bien sûr, de pouvoir décider qu'un algorithme est meilleur qu'un autre ou que tous les autres. On pourrait penser que les performances croissantes des ordinateurs permettent de négliger de plus en plus cet aspect des choses. En fait, il n'en est rien, et il existe même des problèmes, convenablement analysés, pour lesquels on a des algorithmes, mais qui restent pratiquement insolubles, du simple fait que les dits algorithmes prendraient des millions d'années sur les ordinateurs les plus rapides.
De ce point de vue, trois questions récurrentes face à un algorithme :

1. Mon algorithme est-il correct, se termine-t-il ?

2. Quelle est sa vitesse d'exécution ?

3. Y-a-t'il moyen de faire mieux ?

Quel que soit le type auquel se rattache le problème à traiter, le but de l'analyse de la complexité est de quantifier la rapidité de calcul et l'occupation mémoire pour qu'on puisse déterminer :

- Un ordre de grandeur du temps ou d'espace nécessaire à l'exécution du programme, indépendamment de l'ordinateur utilisé.
- Si un algorithme est meilleur qu'un autre pour résoudre le problème considéré quelque soit le langage de programmation et l'ordinateur utilisé.
- Si un certain algorithme donné est "optimal", c'est à dire s'il existe ou non des algorithmes plus performants que l'algorithme donné.

Nous nous intéressons dans la suite de ce chapitre à l'étude de la complexité en temps.

3. Complexité en temps

Définition :
On appelle complexité d'un algorithme A pour une donnée d de taille n la durée d'exécution de l'algorithme A pour cette donnée : $Coût_A(d)$

On cherche à déterminer une mesure en temps exprimant la complexité intrinsèque des algorithmes indépendamment de leur implémentation. On se donne pour réaliser cela:

1. **Une mesure de taille de donnée** : qui sera souvent définie cas par cas.

Exemples :

- Pour un problème de manipulation de matrices, on pourra prendre la dimension maximale ou bien la somme, ou le produit des dimensions.
- Pour un problème de tri d'un tableau, on pourra prendre le nombre d'éléments à trier.

2. **Une unité d'évaluation :** on peut mettre en évidence une ou plusieurs opérations jugées fondamentales au sens où le temps d'exécution d'un algorithme est toujours proportionnel au nombre de ces opérations.

Exemple :

- Pour le problème de recherche d'un élément dans une table, l'unité pourra être la durée d'exécution d'une instruction élémentaire (si on suppose que les instructions élémentaires ont toutes le même temps d'exécution).
- On peut prendre comme unité (c'est le plus souvent) le coût d'une comparaison de élément recherché à un élément de la table.

Pour certains algorithmes, le temps exécution ne dépend que de la taille des données; mais la plupart du temps, la complexité varie aussi en fonction de la configuration de la donnée.

4. Complexité asymptotique

En fait, on s'intéresse bien souvent à des ordres de grandeurs et non à des valeurs exactes. On utilise très souvent la notation $g = \theta(f)$ pour dire que la fonction g est de l'ordre de f.

Définition :
Soient deux algorithmes A et B de complexité respectives f(n) et g(n), on dit que:
$f = \theta(g)$ si et seulement si il existe un entier n_0 et $c \in R^{++}$ tel que : $\forall n > n_0$, on a: $f(n) < c.g(n)$

On dit que f est dominée **asymptotiquement** par g.
Les ordres de grandeurs qu'on rencontre le plus souvent en algorithmique sont : Log(n), n^2, n^3, n^4, n^n, 2^n, n!
Pour fixer les idées, supposons qu'un algorithme A prenne, une fois programmé dans un langage donné sur un ordinateur donné un temps $coût_A(100)$ pour s'exécuter avec n=100. Le tableau suivant indique, selon l'ordre de grandeur $\theta(\ )$ jusqu'à quelle valeur N de n, on peut faire exécuter cet algorithme sans dépasser un temps cent fois plus grand.

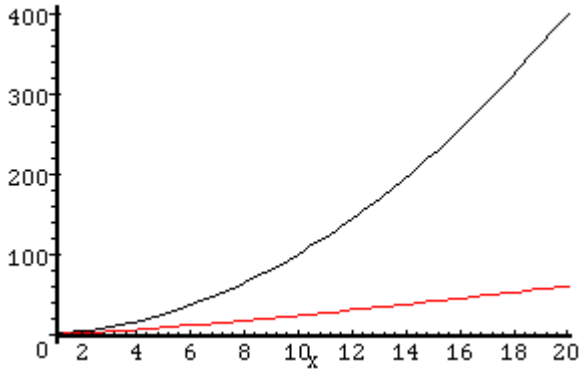
$\theta(\ )$	Log(n)	n	nLog(n)	n^2	n^3	n^4	2^n	n^n
N	100^{100}	10^4	4500	10^3	463	316	107	101

On voit donc que l'ordre de grandeur de l'algorithme utilisé va être le facteur prépondérant pour déterminer quelle taille maximum du problème on pourra traiter dans un temps donné.
Un algorithme est dit :

- ***En temps constant*** si sa complexité dans le pire des cas est bornée par une constante.
- ***linéaire*** (resp linéairement borné) si sa complexité (dans le pire des cas) est en (n) (resp en θ (n)).
- *Sub linéaire* si sa complexité (dans le pire des cas) est en ($\log_2(n)$) (resp en θ ($\log_2(n)$)).
- ***Quadratique*** si sa complexité (dans le pire des cas) est en (n^2) (resp. en θ (n^2)).
- ***Polynômial*** ou polynômialement borné, si sa complexité (dans le pire des cas) est en $\theta(n^p)$ pour un certain p.
- ***Exponentiel*** si sa complexité est en $\theta(2^n)$.

Un polynôme est de l'ordre de son degré. On distingues les fonctions linéaires (en $\theta(n)$), les fonctions quadratiques (en $\theta(n^2)$) et les fonctions cubiques (en $\theta(n^3)$). Les fonctions d'ordre exponentiel sont les fonctions en $\theta(a^n)$ ou a >1. Les fonctions d'ordre logarithmique sont les fonctions en $\theta(\log(n))$ (Remarquons que peu importe la base du logarithme). Une classe intéressante d'algorithme est en nlog(n).

Comparaison de nlog(n) et de n^2.



Le tableau ci-dessous montre le temps d'exécution de quatre algorithmes selon leur comportement et la taille des données; on suppose que la durée d'une instruction élémentaire est de l'ordre de la µs.

Taille / Comportement	**$\log_2 n$**	**n**	**n^2**	**2^n**
10	3µs	10µs	100µs	1000µs
100	7µs	100µs	1/100s	1014 siècles
1000	10µs	1000µs	1s	astronomique
10000	13µs	1/100s	1, 7mn	astronomique
100000	17µs	1/10s	2, 8h	astronomique

5. Calcul de complexité

Déterminer la complexité d'un algorithme A revient à trouver un ordre de grandeur pour la fonction $coût_A(n)$ représentant le temps d'exécution de cet algorithme en fonction de la taille n des données d'entrée et les opérations fondamentales.

Instruction simple

La complexité d'une instruction d'affectation (sans appel de fonction), de lecture ou d'écriture peut en général se mesurer par $\theta(1)$ (indépendant de n).

Séquence

La complexité d'une séquence d'instructions est égale à la somme des complexités de chaque instruction.

Le choix

La complexité d'une instruction « Si-sinon finsi » est égale à la somme de la complexité d'évaluation de la condition et la plus grande complexité entre la séquence d'instructions exécutée si la condition est vraie et celle exécutée si la condition est fausse.

Répétition

La complexité d'une boucle est égale à la somme de la complexité de la séquence d'instructions qui constitue le corps de la boucle et celle de l'évaluation de la condition de sortie.

Sous-programme

Pour les appels de procédures/fonctions, s'il n'y a pas récursivité, la complexité d'un sous-programme est égale à la somme des complexités de chacune de ses instructions. Pour les procédures et fonctions récursives, l'analyse donne en général lieu à la résolution de relations de récurrences.

Exemples

Déterminons la complexité (trouver une fonction $coût_A(n)$) de quelques algorithmes classiques afin d'illustrer les différents cas vus précédemment à travers les exemples suivants.

Exemple 1 :
Calculez le nombre d'actions exécutées par le segment du code suivant:

```
pour I←2 jusqu'à n pas 1 faire
    somme← somme + I
fin pour
```

1. Le nombre d'itérations d'une boucle **pour** est égal à : la valeur finale de l'indice de parcours, moins sa valeur initiale, plus 1. Dans notre cas, nous avons: $n - 2 + 1 = n - 1$. Le corps de la boucle a une complexité égale à 1. La condition de sortie ($i > n$) et évaluée n fois. Donc, nous avons en tout $(n-1) *1+n$ actions, c-à-d. $(2n-1)$ actions.

2. Nous pouvons voir intuitivement que la valeur de cette expression est proportionnelle à n. On dit alors qu'elle a un "ordre de grandeur (ou taux d'augmentation) de n ". Nous dénotons ceci en utilisant la notation θ (n). Or, cette expression n'est autre que le temps d'exécution de cet algorithme en fonction du nombre d'actions et l'entier n (donné), donc $coût_A(n) = \theta(n)$.

Exemple 2 :
Reprenons la même question pour l'algorithme suivant:

Algorithme moyenne_de_n_nombre
Déclaration
n, somme, i, nombre : entier
Moyenne : réel
Début
1. Lire(n)
2. somme ← 0
3. i← 1
4. **Tant que** i <= n **faire**
5. lire(nombre)
6. somme← somme + nombre
7. i ← i+1
fin tanque
8. moyenne← somme/n
fin

Chacune des actions 1, 2, et 3 sera exécutée une seule fois.
Chacune des actions 5, 6 et 7 sera exécutée n fois.
L'action 4 qui contrôle la boucle sera exécutée n + 1 fois (une exécution supplémentaire est exigée),
Et l'action 8 sera exécutée une seule fois. Cela est résumé ci-dessous:

Actions	**Nombre de fois**
1	1
2	1
3	1
4	n + 1
5	n
6	n
7	n
8	1

Donc, le temps d'exécution pour cet algorithme sera :
$coût_A(n) = 1+1+1+(n+1)+n+n+n+1 = 4n + 5 = \theta(n)$.

Exemple 3:
Soit la boucle intérieure d'un algorithme de tri (dit tri par sélection). Trouvez le nombre d'actions exécutées et la complexité de cet algorithme?

```
1. Pour i←1 jusqu'à n-1 faire
2.      min_pos ← i;
3.      min ← t[min_pos];
4.      pour j← i+1 jusqu'à n faire
5.          si (t[j] < min) alors
6.            min_pos ← j
7.           min ← t[min_pos]
            Finsi
        Finpour
8.      t[min_pos] ← t[i];
9.      t[i] ← min
   Finpour
```

1. L'action 1 est exécutée n fois (n − 1 + 1);
2. Les actions 2, 3, 8, 9 (chacune des actions a une complexité de l'ordre de θ(1)) sont exécutées (n − 1) fois, une fois sur chaque itération de la boucle externe.
3. A la première itération (i = 1):
   - l'action 4 est exécutée n fois;
   - l'action 5 est exécutée n − 1 fois,
   - et en supposant le plus mauvais cas où les éléments de la collection sont dans un ordre descendant, les actions 6 et 7 (chacune en θ (1)) sont exécutées (n−1) fois.
4. A la deuxième itération (i = 2),
   - l'action 4 est exécutée n − 1 fois
   - et les actions 5, 6 et7 sont exécutées n-2 fois, etc.,
   - Donc, l'action 4 est exécutée (n) + (n − 1) + ... + 2 = n(n+1)/2 fois
   - et les actions 5, 6, 7 sont exécutés (n − 1)+(n − 2)+...+2+1 = n(n−1)/2 fois.

D'où le temps d'exécution total est:

$coût_A = (n) + 4(n - 1) + n(n+1)/2 + 3[n(n-1)/2]$
$= n + 4n - 4 + (n^2+n)/2 + (3n^2-3n)/2$
$= 5n - 4 + 2n^2 - n$
$= 2n^2 + 4n - 4$
$= \theta (n^2)$

Exemple 4:

Analyser le programme suivant :

```
Procédure foo(donnée/sortie x, n : entier) ;
Déclaration
i : entier ;
Début
1. Pour i ← 1 jusqu'à n faire
2.       x ← x + bar(i, n)
  finpour
 Fin
Fonction bar(donnés x, n : entier) : entier
Déclaration
i : entier ;
Début
3. Pour i =1 jusqu 'à n faire
```

4. x ← x + i
finpour
5. retourner(x)
Fin
Algorithme principal
Déclaration
un, n : entier
Début
6. lire(n)
7. un ← 0 ;
8. foo(un, n) ;
9. écrire(bar(n, n))
Fin

1- Nous commençons par analyser la fonction bar qui ne fait aucun appel à d'autres sous-programmes.
La boucle **pour** ajoute à x la somme des entiers de 1 à n. Donc, la fonction bar(x, n) calcule l'expression : x+n(n+1)/2. Le temps d'exécution est analysé comme suit :

- Les actions 3 et 4 sont exécutées n fois.
- L'action 5 est exécutée une seule fois.

Par conséquent, le temps d'exécution de la fonction bar est de l'ordre de $\theta(n) + \theta(1) = \theta(n)$.

2- Ensuite, nous analysons la procédure foo :

- L'action 2 a normalement une complexité de l'ordre de $\theta(1)$. Mais il y a un appel à la fonction bar, donc le temps d'exécution de l'action 2 est : $\theta(1) + \theta(n) = \theta(n)$.
- Les ligne 1 et 2 se répètent n fois,

Donc la complexité de la procédure est de l'ordre de $(n* \theta(n)) = \theta(n^2)$.

3- Finalement, nous analysons l'algorithme principal.

- La complexité de chacune des actions 6 et 7 est de l'ordre $\theta(1)$.
- La complexité de l'appel à foo (à la ligne 8) est de l'ordre de $\theta(n^2)$.
- La complexité de l'action écrire (de ligne 9) égale (à la complexité de l'opération de l'écriture + la complexité de l'appel de fonction bar) : $\theta(1) + \theta(n)$.

Donc, le temps d'exécution total pour l'algorithme principal est : $\theta(1) + \theta(1) + \theta(n^2) + \theta(n) \cong \theta(n^2)$

6. Différents types de complexité

Il faut remarquer qu'un algorithme peut avoir des comportements différents suivant les données à traiter. Imaginons que l'on cherche un nom dans un annuaire téléphonique et que notre algorithme consiste à lire tous les noms depuis le début jusqu'à ce qu'on trouve le bon.

- Si on a beaucoup de chance, le nom cherché est le premier.
- Si, au contraire, on n'a pas de chance du tout, le nom cherché se trouvera en dernier.
- En moyenne, on trouvera le nom après avoir parcouru environ la moitié des pages de l'annuaire.

Pour notre recherche dans l'annuaire, si on considère que lire un nom est une opération élémentaire, on aurait pour un annuaire de n noms:

- Premier cas : $\theta(1)$

- Deuxième cas : $\theta(n)$
- Troisième cas: $\theta(n/2)$

On peut donc distinguer trois types de complexité.
Etant donné le coût d'un algorithme A pour une donnée d de taille n (le nombre d'opérations élémentaires nécessaires au traitement de la donnée d), $coût_A(d)$, les différentes complexités sont :

1. Complexité dans le pire des cas :

$$Max_A(n) = \max\{coût_A(d), d \in D_n\}$$

D_n est l'ensemble des entrées de taille n.

2. Complexité dans le meilleur des cas :

$$Min_A(n) = \min\{coût_A(d), d \in D_n\}$$

3. Complexité en moyenne :

$$Moy_A(n) = \Sigma_{d \in Dn}\, p(d) \,.\, coût_A(d)$$

où p(d) est la probabilité d'avoir en entrée la donnée d parmi toutes les données de taille n.

Si toutes les données sont équiprobables, alors on a,

$$Moy_A(n) = 1/|Dn| \;\; \Sigma_{d \in Dn}\, coût_A(d)$$

Exemple
Soit T un tableau de taille n contenant des nombres entiers de 1 à k. Soit a un entier entre 1 et k. La fonction suivante retourne 1 lorsque l'un des éléments du tableau est égal à a, et 0 sinon.

```
Fonction proc(T[n], n, a :entiers ) : entier
Déclaration i: entier;
Début
  Pour i =1 jusqu'à n faire
    si T[i]=a alors Retourner(i) finsi;
  finpour
  Retourner(0);
Fin
```

Cas le pire : $\theta(n)$ (le tableau ne contient pas a)
Cas le meilleur : $\theta(1)$ (le premier élément du tableau est a)
Complexité moyenne : Si les nombres entiers de 1 à k apparaissent de manière équiprobable, on peut montrer que le coût moyen de l'algorithme est $(1-q/2).n + q/2$, sachant que $q=p[a \in T]$.
Si $q=1$ alors $Moy_A(n)=(n+1)/2$ et,
Si $q=1$ alors $Moy_A(n)=(3n+1)/4$

De fait les cas où l'on peut explicitement calculer la complexité en moyenne sont rares. Cette étude est un domaine à part entière de l'algorithmique que nous n'aborderons pas ici. Toutefois, il est indispensable, après avoir écrit un algorithme, de calculer sa complexité dans le pire des cas et dans le meilleur des cas.

Remarques

1. Les performances de la machine n'interviennent pas directement dans l'ordre de grandeur de la complexité.
2. La théorie de la complexité a pour but de donner un contenu formel à la notion intuitive de difficulté de résolution d'un problème.

7. NP-complétude

Généralement et comme vu précédemment, les principaux facteurs déterminant le temps nécessaire à un calcul sont la taille de l'entrée, l'ordinateur utilisé et le langage de programmation utilisé.
On veut développer une théorie robuste qui fait abstraction des deux derniers facteurs. Il est alors nécessaire de s'entendre sur des modèles de calcul.

Classes de complexité

Tous les problèmes résolubles avec une quantité de ressources donnée sont regroupés dans les trois classes de complexité suivantes :

- P: problèmes décidés par un algorithme *déterministe* résolubles en un temps polynomial.
- NP: problèmes résolubles en temps polynomial avec un algorithme non-déterministe (c.à.d, algorithme avec les instructions classiques + instruction "choisir entre deux instructions données" de façon non-déterministe).
- EXP: problèmes résolubles en temps exponentiel.

Pour placer un problème dans une classe de complexité donnée, il suffit de fournir un algorithme opérant dans les contraintes de ressources correspondantes.
Idéalement, on voudrait un moyen de placer chaque problème dans la *plus petite classe de complexité possible.* Cela correspondrait à trouver un *algorithme optimal*. Ceci peut être résolu en comparant la complexité de deux problèmes. Cette comparaison est basée sur le principe de *réduction* suivant:

Le problème A se réduit au problème B s'il existe un algorithme « efficace » qui permet de résoudre A en utilisant des appels procéduraux à B.

Ceci est dénoté $A \leq_T B$.

Les conséquences essentielles qui peuvent se dégager sont alors:

- Si $A <_T B$, et si on sait résoudre B, alors on sait résoudre A.
- Si $A <_T B$, et si A est difficile, alors B est aussi difficile.
- On peut avoir à la fois $A \leq_T B$ et $B \leq_T A$. Dans ce cas, on considère que A et B sont des problèmes de complexité équivalente. Si un problème qui nous intéresse est de

complexité équivalente à des problèmes pour lesquels aucun algorithme efficace n'est connu, on a une forte indication que ce problème est très complexe.

Complétude

Définition :
Un problème A est complet pour la classe de complexité **C** (P, NP, EXP, etc.) si

- $A \in \mathbf{C}$
- Pour tout $B \in \mathbf{C}$ on a $B \leq_T A$.

C'est-à-dire que A est le problème le plus dur de sa classe (A est un élément maximal dans le préordre).

Utilisation idéale.
Si A est complet pour la classe **C** alors :

- soit A n'admet pas d'algorithme efficace. (par exemple, si on prouve qu'un problème A est NP-complet. Donc probablement A n'admet pas d'algorithme polynomial et probablement P≠NP).
- soit il en admet un et alors *tous* les problèmes de **C** en admettent un également. (par exemple, si on arrive à trouver un algorithme polynomial pour un des milliers de problèmes NP-complets connus, alors P=NP et on en déduira des algorithmes polynomiaux pour tous ces problèmes !).
- Par contraposé, s'il existe un seul problème de **C** qui n'admet pas d'algorithme efficace, alors A n'admet pas d'algorithme efficace.

Propriétés : Deux éléments complets dans *C* sont équivalents (se réduisent mutuellement). Si A et complet dans *C*, B est dans *C*, et A<B, alors B est aussi complet.

La *NP-complétude* est un des outils les plus utilisés pour donner une indication forte qu'un problème n'admet pas d'algorithme efficace.

Exemple de problème: le voyageur de commerce (TSP).

Ce voyageur doit trouver le *chemin le plus court* pour visiter tous ses clients, localisés chacun dans une ville différente, et en passant une et une seule fois par chaque ville. On peut imaginer un algorithme naïf :

1. on envisage tous les chemins possibles,
2. on calcule la longueur pour chacun d'eux,
3. on conserve celui donnant la plus petite longueur.

Cette méthode « brutale » conduit à calculer un nombre de chemin «exponentiel» par rapport au nombre de villes. Il existe des algorithmes plus « fins » permettant de réduire le nombre de chemins à calculer et l'espace mémoire utilisé. **Mais** on ne connaît pas d'algorithme exact qui ait une complexité (temps de résolution) acceptable pour ce problème, *TSP est NP-complet*.

Pour les problèmes qui sont aussi «difficile» (satisfiabilité d'une formule Booléenne : SAT, couvertures de graphes, coloriage de graphe, etc.), on cherche des méthodes approchées, (heuristiques, probabilistes, etc).

Contourner la NP-complétude :

Dans certains cas réels on peut tout de même contourner la NP-complétude en utilsant plusieurs méthodes :

- Considérer des variantes plus simples du problème.
  - TSP est-il plus simple si on considère que la longueur entre les villes i et j : dij $\in \{0,1\}$?
  - TSP est-il plus simple si on considère que dij + djk ≤ dik?
- Utiliser des algorithmes d'approximation
  - Existe-t-il un algorithme efficace qui obtient une solution de TSP qui est au pire 50% plus chère que la solution optimale?
- Utiliser des algorithmes probabilistes
  - Peut-on trouver un algorithme efficace qui aura 99% de chances de trouver une solution optimale à chaque instance?
- Utiliser des heuristiques
  - Peut-on trouver un algorithme efficace qui trouve une solution optimale pour 99% des instances qui nous intéressent?

P = NP?

Beaucoup de problèmes sont NP-complets. Donc si P ≠ NP, aucun de ceux-ci n'admet d'algorithme qui s'exécute en temps polynomial. Malheureusement, on ne sait pas comment démontrer P ≠ NP. Et si P = NP?

Théorème
Si P= NP, il existe des problèmes dans NP et non complet.