UNIVERSITE MOHAMED KHIDER – BISKRA
FACULTE DES SCIENCES EXACTES, DES SCIENCES DE LA NATURE ET DE LA VIE
DEPARTEMENT D'INFORMATIQUE

Complexité et Optimisation

Support de cours et TD

Master d'informatique

Réalisé par : Dr. S. SLATNIA

Version amélioée

05/01/2016

CHAPITRE 1.

COMPLEXITE DES ALGORITHMES ET MESURE DE PERFORMANCE

Partie 1.

THÉORIE DE LA COMPLEXITÉ

La théorie de la complexité à pour but de classifier des problèmes en fonction de la complexité du meilleur algorithme pour les résoudre.

- Certains problèmes sont durs**.**
  - Il peut ne pas exister d'algorithme efficace pour les réoudre,
- Certains problèmes sont faciles.
  - Connaît-on le meilleur algorithme?

Le domaine de l'analyse d'algorithmique permet de présenter un ensemble de concepts de bases pour mesurer la complexité et les performances des algorithmes.

Généralités

L'objectif de ce chapitre est de présenter les grands principes de la complexité algorithmique et les qualités peut-on demander à un algorithme ou à un programme. Ceci nous amène à donner les outils permettant de répondre aux questions suivantes :

- Le programme s'arrête-t-il de l'exécution ?
- Est-ce qu'il réalise effectivement la tâche qu'on lui a demandé? C'est à dire, est-il correct?
- Il faut qu'il soit bien écrit, compréhensible en vue d'une maintenance ou d'une amélioration ce qui nous conduit à présenter le critère d'évaluation d'efficacité entre les algorithmes suivant :
- L'algorithme doit être performant, pour cela on distingue deux critères essentiels :
  - Le temps de calcul nécessaire, directement lié au nombre d'opérations élémentaires qu'effectué l'algorithme.
  - La quantité d'occupation mémoire nécessaire.

Le premier principe s'appelle la terminaison de l'algorithme, le deuxième sa correction et le dernier sa complexité**.** Nous donnons ces principes comme des critères de qualités d'un bon algorithme.

Pour décrire un algorithme, on lui donne en général un nom, on précise quels sont les paramètres qu'il peut prendre en entrée et le résultat qu'il est sensé renvoyer. On Utilise des spécifications de l'algorithme (des modifications de la mémoire).

Si ces opérations s'exécutent en séquence, on parle d'algorithme séquentiel. Si les opérations s'exécutent sur plusieurs processeurs en parallèle, on parle d'algorithme parallèle. Un algorithme est donc un moyen de calculer une solution au problème posé, le calcul étant décomposé en un nombre fini d'instructions et devant être exécutable en un temps fini, quelle que soit la donnée en entrée.

On décrit un algorithme souvent à l'aide d'un pseudo-langage naturel très simplifié, à la fois lisible et formel. Les données, résultats, et variables intermédiaires sont clairement déclarés, chaque calcul est précisé de façon complète.

Définition 1. Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.

En fin, l'algorithme est traduit sous forme d'un langage de programmation adapté. Il est clair que les ordinateurs ne savent faire que des opérations extrêmement élémentaires : accès mémoire et arithmétique essentiellement. C'est donc le travail de l'informaticien d'écrire une solution à un problème complexe sous forme d'une suite d'instructions élémentaires.

Terminaison

Pour montrer qu'un algorithme termine quel que soit le paramètre passé en entrée respectant la spécification, il faut montrer que chaque bloc élémentaire décrit ci-dessus termine, les boucles pour (for) et les instructions conditionnelles terminent forcément. Le seul souci pourrait venir d'une boucle tantque (while).

Un algorithme comporte :

- Une partie temporelle : séquence d'instructions en principe savamment organisée,
- Une partie spatiale : ensemble de données plus ou moins structurées (nombres, vecteurs, matrices, listes...)**.**

Problème : donnée → *l'algorithme fournir* (nombre fini d'étapes) → réponse

Programme doit finir par s'arrêter

Formellement

Un problème abstrait = l'ensemble < E, A, S > où

E est une entrée (un des cas du problème),

A l'algorithme,

S la sortie (résultat).

Un algorithme = < C entraînent → I >,

C : clauses (règles)

I : instructions (traitements).
Ces règles forment une partition de l'univers**:**
- **U** des règles = tous les cas possibles.
- ∩ =∅

→ *L'algorithme ne doit pas oublier un cas, et on ne peut pas être dans deux cas à la fois.*

Exemple 1 : trouver le maximum d'un tableau T

A. Façon abstraite d'un algorithme

Une fois qu'on a compris ce qu'on vient de faire, on peut l'abstraire :
- On considère le premier élément du tableau comme maximum,
- Si on trouve un élément plus grand que lui, c'est notre nouveau maximum
- Une fois que nous avons tout parcouru, nous renvoyons notre dernier maximum : rien trouvé de plus grand

B. Les règles trouvées

- Max(T, n, i, max) = Max(T, |T|, 2, T[1]))}
- i ≤ n ^ T[i] > max → Max(T, n, i, max) = Max(T, n, i+1, T[i])
- Si je n'ai pas parcouru tout le tableau et que l'élément actuel est supérieur au max, c'est le nouveau max.
- i ≤ n ^ T[i] ≤ max → Max(T, n, i, max) = Max(T, n, i+1, max)
- Si mon élément actuel n'est pas supérieur au max, alors je continue simplement de parcourir.
- i > n → Max(T, n, i, max) = max Si j'ai parcouru tout le tableau, je renvoie le max.

C. Ces règles forment une partition de l'univers

- L'union de ces règles, donne tous les cas possible :
(i > n) ∪ (i ≤ n ^ T[i] > max) ∪ (i ≤ n ^ T[i] ≤ max) = {i, T[i]}
- L'intersection des règles deux à deux, donne l'ensemble vide :
(i ≤ n ^ T[i] > max) ∩ (i ≤ n ^ T[i] ≤ max) = {}, l'élément ne peut pas être à la fois plus petit et plus grand.
- (i > n) ∩ (i ≤ n ^ T[i] ≤ max) = {}, on ne peut pas être à la fois dans le tableau et hors du tableau.

Donc l'algorithme ne peut pas être dans deux cas à la fois.

Quelques problèmes algorithmiques classiques

– **Le problème de tri** d'un ensemble d'éléments selon une relation d'ordre (croissant ou décroissant), avec un milliard d'éléments.
– **Le problème de la Recherche** pertinente dans des grandes bases de données.

– **Réalisation d'une Optimisation multi objectifs,** trouver une combinaison des meilleurs critères qui minimisent une fonction coût (temps de calcul et espace mémoire).

Qualités d'un bon algorithme

- **Correct:** Il faut que le programme exécute correctement les tâches pour lesquelles il a été conçu.
- **Complet:** Il faut que le programme considère tous les cas possibles et donne un résultat dans chaque cas..
- **Efficace:** Il faut que le programme exécute sa tâche avec efficacité c'est à dire avec un coût minimal. Le coût pour un ordinateur se mesure en termes de temps de calcul et d'espace mémoire nécessaire.

Pour chaque problème, nous trouvons des différentes variantes définis le corps de l'algorithme. Ceci nous conduit à la nécessité d'estimer le coût d'un algorithme avant de l'écrire et l'implémenter.

Analyse et notions de complexité

L'analyse d'algorithmes permet de donner un outil d'aide à la comparaison des performances des algorithmes. Le mot complexité (latin : *complexus enlacé, embrassé* signifiant un objet constitué de plusieurs éléments ou plusieurs parties).

Cet outil permet de mesurer les ressources critiques (coûteuses) utilisées par les algorithmes.
Les ressources coûteuses :

- Le temps d'exécution
- Programmation
- La mémoire
- Les processeurs
- Les messages
- Les implémentations matérielles d'un algorithme
- La surface du circuit ou le nombre de portes logiques
- L'énergie consommée
- Les radiations émises.

La conception de certains ordinateurs mobiles permet de présenter le problème suivant :

Problème : l'énergie pour la conception de certains ordinateurs mobiles (la consommation électrique) induite par l'émission réception.

Solution : évaluer les performances de l'algorithme en fonction de cette ressource (la taille de cette ressource utilisée par l'algorithme).

Critères d'évaluation de performance de l'algorithme

L'efficacité d'un algorithme est désignée sous le terme de « complexité ». On la mesure, sur une taille d'entrée n donnée, par deux fonctions : *le nombre d'opérations* (complexité en temps) et *la place mémoire* (complexité en espace). Déterminer ces fonctions est le domaine de « l'analyse de complexité ».

Définition 3. La complexité est la mesure de l'efficacité d'un programme pour un type de ressources :
— Complexité temporelle (temps de calcul CPU) : nombre d'opérations élémentaires réalisés par le processeur, ce qui est lié directement au temps de calcul.
— Complexité spatiale (mémoire nécessaire) : espace mémoire RAM ou sur disque dur requis par le calcul.

Calcul de nombre d'opérations

Plusieurs façons d'écrire la performance d'un algorithme, avec des méthodes plus ou moins efficaces :

unité de mesure = l'opération élémentaire, propre à chaque algorithme
En additionnant toutes les opérations élémentaires
→ une bonne estimation du coût (coût est faible).
→ algorithme meilleur

La performance de l'algorithme dépend de ses entrées.

Opérations élémentaires

L'exécution d'un algorithme est une séquence d'opérations nécessitant plus ou moins de temps. Pour mesurer ce temps, il faut préciser quelles sont les opérations élémentaires à prendre en compte.

Pour un algorithme numérique ce sont les opérations arithmétiques de base (addition, multiplication, soustraction, division...), pour un algorithme de tri, ce sont les comparaisons entre éléments à trier (opérations plus coûteuses). lire ou modifier un élément d'un tableau, ajouter un élément à la fin d'un tableau, affecter un entier ou un flottant. Les transferts de valeurs ou affectations sont souvent négligées**.**

Mesure d'espace mémoire

Il est particulièrement important de mesurer la mémoire utilisée, car c'est une ressource qui peut-être rare ou dont l'utilisation peut s'avérer très coûteuse du point de vue énergétique (imaginons les satellites…). Pour un problème donné, on peut donc être

amené à choisir entre un algorithme rapide mais utilisant beaucoup de mémoire, et un algorithme plus lent qui utilise la mémoire de façon modérée.

Il faut examiner la *complexité* des algorithmes, *la nature des données, et éventuellement la machine cible.*

Coût d'un algorithme

Définition 4.
L'algorithme a un coût qui est lié :
- au nombre d'opérations effectuées (opérations arithmétiques, logiques, transferts...),
- à l'espace mémoire occupé par les données.

Evaluer ce coût revient à mesurer ce qu'on appelle la complexité de l'algorithme, en temps comme en espace. Il s'agit donc de dénombrer des opérations et des octets.

Dans la procédure d'évaluation de complexité, la complexité temporelle est plus importante que la complexité spatiale. L'étude de la complexité d'une fonction consiste à estimer son coût en ressource en fonctions des entrées. Pour différencier deux entrées entre elles, on compare en général leur taille. Les entrées seront constituées d'entiers, de flottants ou de tableaux. Pour les tableaux, la donnée pertinente est la taille. Pour les entiers, cela dépend du contexte. Pour un entier n, on peut en effet exprimer la complexité d'une fonction dépendant de n en fonction :
— de l'entier n lui-même.
— ou de son nombre de chiffres (sa taille), correspondant à $\log_2(n)$.

Définition 5.
Coût de A sur n: l'exécution de l'algorithme A sur la donnée n requiert $C_A(n)$ opérations élémentaires.

Estimer le coût d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée. La complexité en mémoire consiste à estimer la mémoire nécessaire à une fonction pour son exécution.

Complexité d'un algorithme

Définition 6. On appelle complexité de l'algorithme A pour la ressource R la fonction :
T(A,R, n) = max{ ressource R utilisée par A sur l'ensemble des données de taille n}

Définition 7. La taille d'une donnée est simplement la taille d'un bon codage en mémoire de cette donnée exprimée en nombres de bits.

La taille d'un codage d'un entier n sera :
[(log2(n + 1)], et non pas n.
∄ d'ambiguïté, T(A,R, n)= T(n).
⇨ la mesure du plus mauvais cas, car elle garantit que tout comportement de l'algorithme A utilisera au plus T(A,R, n) éléments de la ressource R.

Définitions 8.
On définit le Cas le pire et le cas moyen**.**
n désigne la taille de la donnée à traité.

- **Dans le pire des cas :** $C_A(n) := \max_{x|x|=n} C_A(x)$
- **En moyenne :** $C^{Moy}_A(n) := \sum_{x|x|=n} p_n(x)*C_A(x)$

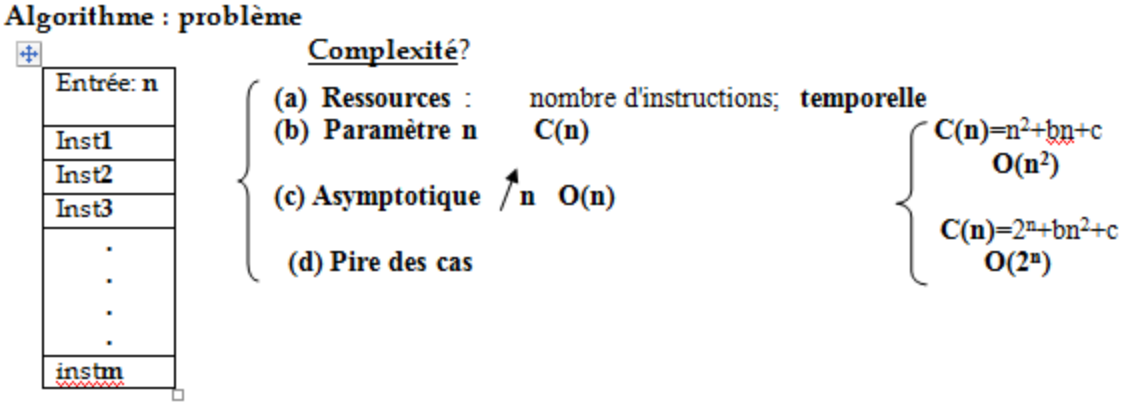
p_n : distribution de probabilité sur les données de taille n.

Complexité d'un problème

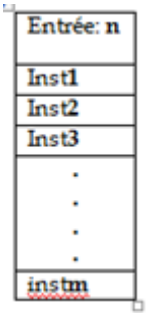
L'analyse d'un algorithme :
- Exprimer quelques choses sur le problème associé.
- La résolution d'un problème → tout algorithme n'a pas une complexité minimale.
≡ C'est la question des *bornes inférieures* de complexité.

Problème : les calculs des fonctions de complexité sont difficiles à faire de manière exacte.
Solution : procède par approximations des notations (O, Ω et Θ).

Comment calculer la complexité d'un algorithme ?



Algorithme: **Problème**
Complexité?



$$\begin{cases} C(n+1)=C(n) & O(1) \\ C(n+1)=C(n)+1 & O(n) \\ C(n+1)=C(n)+\varepsilon & O(\log_2(n)) \\ \quad 2n \quad C(n)+1 & \\ C(n+1)=C(n)+n & O(n^2) \\ C(n+1)=2*C(n) & O(2^n) \end{cases}$$

Type de complexité

A. Complexités temporelle et spaciale d'un algorithme

A.1. Complexité en temps d'un algorithme

Définition 9.
I : l'ensemble des données d'instances d'un problème abstrait Π.
In : les données de taille n (le coût dépend de la donnée)
c(i) : le coût de l'algorithme résolvant le problème Π pour une donnée i∈In.
On définit 3 types de complexité :
→ Le coût dans le pire des cas (l'algorithme est le moins performant) :

$$\mathbf{W_{algorithme}(n) = max\{c(i) \mid i \in In\}}$$

→ Le coût dans le meilleur des cas (l'algorithme est le plus performant) :

$$\mathbf{B_{algorithme}(n) = min\{c(i) \mid i \in In\}}$$

→ Le coût dans cas moyen :

$$\mathbf{A_{algorithme}} = \sum_{i \in In} \mathbf{p(i)} * \mathbf{c(i)}$$

En général : $\mathbf{A_{algorithme}} = \frac{1}{|In|} \sum_{i \in In} \mathbf{c(i)}$

A.2. Complexité en espace d'un algorithme

Permet de définir l'espace mémoire requis par le calcul.

B. Complexités pratique et théorique

Définitions 10.
La complexité pratique est une mesure précise des complexités temporelles et spatiales pour un modèle de machine donné.

La complexité théorique est un ordre de grandeur de ces couts, exprimé de manière la plus indépendante possible des conditions pratiques d'exécution.

Notion d'optimalité

Définition 11.
Un algorithme est dit optimal (performant) si sa complexité est la complexité minimale (la borne inférieure de complexité) parmi les algorithmes de sa classe.

Exemple 2 :
On peut montrer que tout algorithme résolvant le problème du tri a une complexité dans le pire des cas en Ω(nlgn). Le tri par fusion est en O(nlgn) dans le pire des cas : il est donc optimal.

Partie 2.

GRANDEUR DES FONCTIONS, LA COMPLEXITE ASYMPTOTIQUE ET CALCUL DE COMPLEXITE DES ALGORITHMES ITERATIFS ET RECURSIFS

I. Optimisation d'exécution de la récursivité

1. Structures de données

Une structure de données indique la manière d'organisation des données dans la mémoire. Le choix d'une structure de données adéquate dépend généralement du problème à résoudre.
Deux types de structures de données :

1. Statiques : Les données peuvent être manipulées dans la mémoire dans un espace statique alloué dès le début de résolution du problème. Exemple : les tableaux
2. Dynamiques : On peut allouer de la mémoire pour y stocker des données au fur et à mesure des besoins de la résolution du problème. Exemple: liste chainée, pile, file,
    - Notion des pointeurs,
    - Nécessité de la gestion des liens entre les données d'un problème en mémoire.

2. Qu'est ce que la récursivité ?

Qu'est-ce que la récursion?
Le mot «récursion» vient du latin recursare qui signifie courir en arrière, revenir.
L'idée de réapparition, de retour, est donc étymologiquement liée à la récursivité.

Définition 1.
Lorsqu'un système contient une autoréférence (ou une copie de lui même), on dit que ce système est récursif**.**

Définition 2.
Un algorithme est dit récursif si l'expression qui le définit fait appel à lui-même. On qualifiera de récursif, un appel à une fonction f provoqué par l'évaluation d'un autre appel à f.

Une fonction récursive est caractérisée par une, ou plusieurs, *relations de récurrence***.**
La fonction récursive se rappelle jusqu'à arriver sur :
→ Cas d'arrêt (cas de base),
→ Quand on construit des fonctions récursives, le raisonnement adopté est en général:
- Comment traiter le cas le plus simple ?
- Comment me ramener d'un cas plus compliqué au cas simple ?

Exemple 1 : La fonction factorielle
$n > 1 \rightarrow f(n) = n*f(n-1)$
$n = 0 \rightarrow f(0) = 1$
Le cas de base est $f(0) = 1$.

3. Types de récursivité

Récursivité simple

Une récursivité simple contient un seul appel récursif à P dans le corps d'une procédure récursive P.

Récursivité multiple

Une récursivité est multiple si il y a plusieurs appels récursifs à P dans le corps d'une procédure récursive P.

Récursivité mutuelle

Une récursivité est mutuelle ou croisée quand une procédure P appelle une procédure Q qui déclenche un appel récursif à P.

Récursivité imbriquée

Une récursivité est imbriquée si une procédure récursive P contient un appel imbriqué.

Récursivité terminale

La récursivité est terminale si l'appel récursif est la dernière instruction et elle est isolée.

La récursivité n'est pas terminale si l'appel récursif n'est pas la dernière instruction et/ou elle n'est pas isolée.

4. Critères de terminaison pour un bon algorithme récursif

Deux règles sont à respecter impérativement.

Règle 1 : Un algorithme récursif doit être défini par une expression conditionnelle dont l'un au moins des cas mène à une expression évaluable sans appel récursif. Une telle expression est appelée condition d'arrêt ou test d'arrêt ou clause terminale ou cas de base ou encore cas trivial.

Règle 2 : Il faut s'assurer que pour toute valeur du (ou des) paramètre(s), il suffira d'un nombre fini d'appels récursifs pour atteindre la condition d'arrêt.

5. Comment gérer la récursivité ?

Quand la fonction est récursive, l'ordinateur est bien obligé de *stocker les calculs quelque part*. Il utilise pour cela une pile; il s'agit d'un objet défini par deux opérations :

- Empiler (ajouter un élément au sommet),
- Dépiler (prendre l'élément qui est au sommet).

Exemple 2 :

1. Gestion de la pile pour la factorielle

Algorithme pour le calcul de la factorielle :

F(n) = F(m) (toujours une initialisation)

n > 1 → F(n) = n*F(n-1)

n = 0 → F(0) = 1

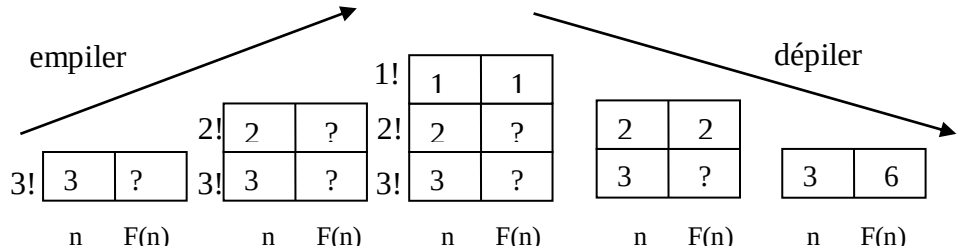


Figure 1. Calcul factorielle, f(3).

2. Une factorielle qui n'a pas besoin de pile

Algorithme pour la factorielle utilisant une variable acc, qui accumule :

F(n,acc) = F(m, 1)

n > 1 → F(n, acc) = F(n-1, n*acc)

n = 0 → F(n, acc) = acc

Il n'y a pas besoin ici de mettre des calculs en attente : on peut les enchaîner. Ainsi, on se passe de la pile.

5.1. Gestion de la récursivité sans la pile : Récursivité Terminale (RT)

La récursivité terminale est une notion qui peut améliorer nettement les performances de vos algorithmes. L'exécution d'un algorithme utilisant la récursivité terminale est

transformée en général en algorithme itératif (plus rapide et moins gourmand en mémoire) par le compilateur.

Définitions 3.
Un algorithme est récursif terminal si son appel est le dernier. Une fonction est récursive terminale si elle renvoie, sans autre calcul, la valeur obtenue par son appel récursif.

Une fonction est de type récursivité terminale, si la fonction est du type
F(x1, …, xn) = G(y1, …, yn).

⇨ Il n'y a alors pas besoin de pile pour continuer le calcul.

5.2. Gestion de la récursivité avec la pile : Récursivité Enveloppée (RE)

Définition 4.
Une fonction est de type récursivité enveloppée, lorsqu'il y a quelque chose autour de la fonction G, Il y a des opérations qui enveloppent la fonction.

⇨ Il y a alors besoin de pile pour continuer le calcul.

6. Optimisation d'exécution de la récursivité

Optimisation d'exécution de la récursivité par la transformation de RE vers RT, En remplaçant son enregistrement au lieu d'en empiler un autre au-dessus de lui, l'utilisation de la pile est considérablement réduite, ce qui, en pratique, se traduit par de meilleures performances.

On doit donc transformer les fonctions récursives en fonctions récursives terminales à chaque fois que cela est possible.

Les langages exécutent un programme à récursivité terminale comme s'il était itératif (en espace constant). Sinon, il est facile de transformer une définition récursive terminale en itération pour optimiser l'exécution.

6.1. Comparaison d'algorithmes

Exemple 3 :

1. Gestion de la récursivité pour le calcul de la factorielle :

→ **RT :**

- F(n, acc) = F(n-1, n*acc), on peut identifier $y_1 = x_1 - 1$, $y_2 = n*x_2$ et G = F.
- F(1, acc) = acc. On peut identifier $y_1 = 0$, $y_2 = x_2$ et $G = y_2$.

→ **RE :**

- F(n) = n*F(n-1), la fonction est enveloppée par une multiplication avec n.

2. Gestion de la récursivité pour faire la somme des éléments d'un tableau

- On dispose d'un tableau T d'entiers, et on veut faire la somme de tous ses éléments.

→ **RE :** les règles trouvées

1. $\Sigma(T, n) = \Sigma(T, |T|)$
2. $n \geq 1 \rightarrow \Sigma(T, n) = T[n] + \Sigma(T, n-1)$
3. $n < 1 \rightarrow \Sigma(T, n) = 0$

Problème : On remarque à la ligne 2 que l'appel est enveloppé par l'addition avec T[n] : récursivité enveloppée → utiliser la pile.

Solution : traduire ceci en une récursivité terminale.
Le but maintenant est de faire rentrer l'addition avec T[n] dans la fonction.

⇨ Utiliser un accumulateur : au départ on le met à 0, et à chaque appel on y ajoute T[n]. A la fin, il suffira de rendre la valeur accumulée :

→ **RT** : les règles trouvées ≡ algorithme avec une boucle

1. $\Sigma(T, n, acc) = \Sigma(T, |T|, 0)$
2. $n \geq 1 \rightarrow \Sigma(T, n, acc) = \Sigma(T, n-1, acc + T[n])$
3. $n < 1 \rightarrow \Sigma(T, n, acc) = acc$

→ **Traduction d'un algorithme récursif vers un algorithme itératif**

⇨ Traduction d'un algorithme de RT vers un algorithme avec une boucle

```
fonction F(T : tableau[1..n] d'éléments entier; n, acc : entier) : entier
Début
acc← 0;
  tant que (n ≥ 1) faire
     debut
     acc ← acc + T[n];
     n ← n – 1;
     fin;
  retourner(acc) ;
fin.
```

3. Gestion de la récursivité pour trouver le maximum d'un tableau d'entiers :

→ **RT :**

```
i←2 ;
max← T[1] ;
```

```
Fonction Max(T : tableau [1..n] d'entier ; n,i,max: entier) : entier
Début
  Si (i ≤ n et T[i] > max) alors
    Max(T, n, i+1, T[i]) ;
  sinon
     Si (i ≤ n)  alors
        Max(T, n, i+1, max) ;
     sinon
        retourner(max) ;
     finsi ;
 finsi ;
Fin.
```

→ **Traduction de d'un algorithme récursif vers un algorithme itératif**

⇨ Traduction d'un algorithme de RT vers un algorithme avec une boucle

```
1.  fonction Max(T : tableau[1..n] d'élément entier ; n :entier)
2.  Début
3.       max ← T[1] ;
4.       pour i de 2 à n faire
5.           si (T[i] > max) alors
6.               max ← T[i] ;
7.       finpour ;
8.       Retourner (max) ;
9.  Fin.
```

→ **Calculer le coût de l'algorithme**

Pour compter le nombre exacts d'opérations de cet algorithme :
La condition dépend des données :
- Si le maximum est au début du tableau → on a juste une affectation (ligne 3) et n – 1 comparaisons soit un total de n opérations.
- Si le tableau est trié dans l'ordre croissant → (n – 1)*2 + 1 = 2n – 1 opérations.
- Si le tableau est aléatoire → on a (n – 1)*(1 + ½) + 1 = (3*n – 1)/2 opérations.

⇨ Le coût de l'algorithme dépend de la taille (n) des données, mais peu aussi être dépendant de leur nature.

II. C'est quoi les notions de Landau ?

La plupart des programmes contient un nombre d'instructions très élevés, l'évaluation de la complexité de façon exact pour comparer deux algorithmes n'est pas facile parce que des critères matérielles sont introduite dans le calcul.

6.2. Classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité**.**

Quelques complexités de référence les plus utilisées: de la meilleure (algorithme le plus performant) à la pire (algorithme difficilement exploitable)**.**

Complexités de référence	Notation	Fonction	Définition
Complexité Constante	**O(1)**	**f(n)=1**	Accéder au premier élément d'un ensemble de données.
Complexité Logarithmique	**O(logn)**	**f(n)=log(n)**	Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales.
Complexité Linéaire	**O(n)**	**f(n)=n**	Parcourir un ensemble de données.
Complexité Quasi-linéaire	**O(nlogn)**	**f(n)=nlog(n)**	Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale.
Complexité Quadratique	**$O(n^2)$**	**$f(n)= n^2$**	Parcourir un ensemble de données en utilisant deux boucles imbriquées.
Complexité Cubique	**$O(n^3)$**	**$f(n)= n^3$**	Parcourir un ensemble de données en utilisant trois boucles imbriquées.
Complexité Polynomiale	**$O(n^P)$**	**$f(n)= n^p$**	Parcourir un ensemble de données en utilisant P boucles imbriquées.
Complexité Exponentielle	**$O(2^n)$**	**$f(n)= 2^n$**	Générer tous les sous-ensembles possibles d'un ensemble de données.

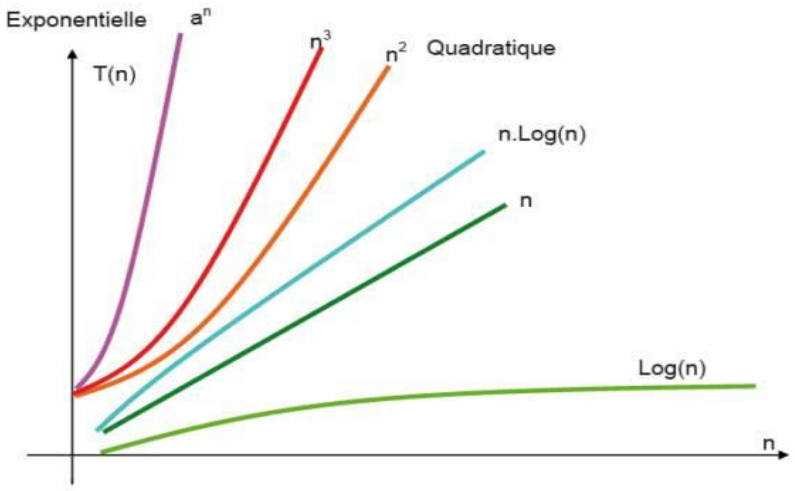


Figure 2. Classes de complexité

Algorithme par rapport à ces références : est-il au pire en n^2 ? Au mieux en n^2 ? Exactement en n^2 ? Pour cela, on a introduit et défini formellement les notations dites de **Landau** :

III. Classes de complexité et Notations asymptotique

Notations asymptotique

Domination asymptotique

f et g étant des fonctions, f = O(g) s'il existe des constantes c>0 et n0 telles que
f(x) < c*g(x) pour tout x > n0

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

Définition 5.
- Quand la complexité f(n) de l'algorithme est majorée par g(n), on dit qu'il est en O(g(n)).

$O(g(n)) = \{ f, \exists n0, \exists c>0, \forall n>n0, f(n) \leq c.g(n) \}$

Il existe un certain nombre à partir duquel f (n) est toujours inferieur à g(n)*constante.

La notation O, dite notation de Landau, vérifie les propriétés suivantes :
- si f=O(g) et g=O(h) alors f=O(h)
- si f=O(g) et k un nombre, alors k*f=O(g)
- si f1=O(g1) et f2=O(g2) alors f1+f2 = O(g2+g2)

- si f1=O(g1) et f2=O(g2) alors f1*f2 = O(g2*g2)

Notation Ω

f = Ω(g) s'il existe des constantes c>0 et n0 telles que f(x) ≥ c*g(x) pour tout x ≥ n0

Définition 6.

- Quand la complexité f(n) de l'algorithme est minorée par g(n), on dit qu'il est en Ω(g(n)).

$\Omega(g(n)) = \{ f, \exists n0, \exists c>0, \forall n>n0, f(n) \geq c.g(n) \}$

Il existe un certain nombre à partir duquel g(n) est toujours supérieur à f(n)*constante.

Equivalence asymptotique

f et g étant des fonctions, f = Θ(g) s'il existe des constantes c1, c2, strictement positives et n0 telles que c1*g(x) ≤ f(x) ≤ c2*g(x) pour tout x ≥ n0

Définition 7.

- Quand la complexité f(n) de l'algorithme est exactement en g(n), on dit qu'il est en Θ(g(n)).

S'il est exactement en Θ(g(n)), cela veut dire qu'il est borné par deux multiples de g(n). Il peut-être minoré et majoré par des multiples ; c'est donc l'intersection des classes de complexité précédentes : Θ(f(n)) = O(f(n)) ∩ Ω(f(n))

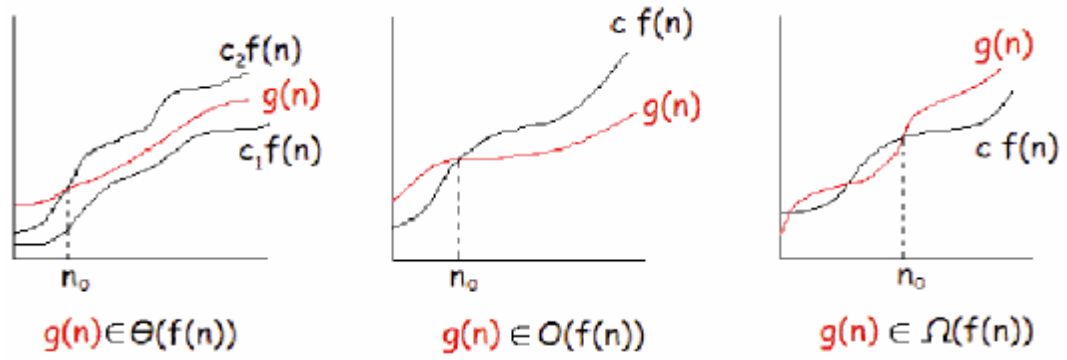


Figure 3. Les notations de Landau**.**

Calcul de complexité asymptotique

Nous concentrons sur le coût des actions résultant de l'exécution de l'algorithme, en fonction d'une "taille" n des données traitées. Ceci permet en particulier de comparer deux algorithmes traitant le même calcul. Nous sommes plus intéressés par un

comportement asymptotique, que se passe -t'il quand n tend vers l'infini? que par un calcul exact pour n fixé. Le temps d'exécution dépend de la nature des données.

Définitions 8.

Borne supérieure non asymptotiquement approchée (négligeable).

g(n) est une borne supérieure non asymtotiquement approchée de f(n) ou que f est négligeable devant g.

Si $\lim_{n\to\infty} f(n)/g(n) = 0$ alors On note $f(n) = o(g(n))$

Propriétés

- si $f(n) = \Theta(g(n))$ alors $f(n) = O(g(n))$
- si $f(n) = o(g(n))$ alors $f(n) = O(g(n))$
- Θ et O sont des relations réflexives
- et o sont des relations transitives
- Θ est une relation d'´equivalence
- pour tout $K \in IR$, $\Theta(K + f(n)) = \Theta(f(n))$
- pour tout $C \in IR$, $\Theta(C\, f(n)) = \Theta(f(n))$
- pour tout $j < k$, $\Theta(n^k + n^j) = \Theta(n^k)$

Propriétés.

soient f et g deux fonctions.

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = 0 \text{ implique } f \in O(g) \text{ et } g \notin O(f) \text{ et } \Theta(f) < \Theta(g).$$

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = 1 \text{ implique } f \sim g.$$

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = c \neq 0 \text{ implique } f \in O(g) \text{ et } g \in O(f) \text{ et } f \in \Theta(g),\ g \in \Theta(f)$$

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = \infty \text{ implique } f \notin O(g) \text{ et } g \in O(f) \text{ et } \Theta(f) > \Theta(g).$$

$$f(n) = O(g(n)) \Leftrightarrow \begin{array}{l}\text{il existe des constantes stt positives } c \text{ et } n_0 \\ \text{telles que } 0 \leq f(n) < c.g(n) \text{ pour tout } n \geq n_0\end{array} \Leftrightarrow \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$$

$$f(n) = \Omega(g(n)) \Leftrightarrow \begin{array}{l}\text{il existe des constantes stt positives } c \text{ et } n_0 \\ \text{telles que } 0 \leq c.g(n) \leq f(n) \text{ pour tout } n \geq n_0\end{array} \Leftrightarrow \liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0$$

$$f(n) = \Theta(g(n)) \Leftrightarrow \begin{array}{l}\text{il existe des constantes stt positives } c_1,\ c_2 \text{ et un } n_0 \\ \text{tels que } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \text{ pour tout } n \geq n_0\end{array}$$

Exemple 4 :

```
{début}
K ← 0 ;
I ← 1 ;
{#1}
TantQue I ≤ N {#2} Faire
    R ← R+T[I]; {#3}
    I ← I+1;{#4}
FinTantQue;
{fin}.
```

Le temps d'exécution t(n) de cet algorithme en supposant que:

- N=n
- t1 est le temps d'exécution entre le début et {#1}
- t2 est le temps d'exécution de la comparaison {#2}
- t3 est le temps d'exécution de l'action {#3}
- t4 est le temps d'exécution de l'action {#4}
- t1, t2, t3, t4 sont des constantes (ne dépendent pas de n)

$t(n)= t1 +\sum_{i=1}^{n} (t2+t3+t4) + t2$

et en définissant le temps tit d'exécution d'une itération (condition comprise),

tit= (t2+t3+t4) d'où t(n)= t1+ t2+n×tit

Ce qui signifie que le temps d'exécution dépend linéairement (plus précisément est une fonction affine) de la taille n.

Nous intéressions au comportement asymptotique:

$\lim_{n\to\infty} t(n)/tit\times n = 1$ autrement dit t(n) est équivalent à l'infini à tit*n : $t(n) \sim_{\infty} tit\times n$

⇨ L'algorithme est donc asymptotiquement linéaire en n.

Dans cet exemple simple l'évaluation en "pire des cas" est immédiate puisque t(n) ne dépend pas de la nature des données,

IV. Calcul de complexité des algorithmes itératifs et récursifs

Coût uniforme et coût logarithmique

A propos de la notation O

Cette notation fait abstraction des détails liés à la machine mais aussi de certaines opérations de programmation.

Exemple 1 : comment vérifier qu'un élément est dans un tableau.

```
Fonction test(T : tableau [1..n]d'élément entier ;x,n,i : entier) :booléen
Début
   i←1 ;
   Tant que  (i ≤ n) et (T[i] ≠ x) alors
      i←i+1;
   fintant que;
   si (i ≤ n) alors
     retourner(vrai) ;
   sinon
     retourner(faux) ;
   finsi ;
Fin.
```

→ La complexité est en O(n).

Si on ajoute x à la fin du tableau,

```
Fonction test(T : tableau [1..n]d'élément entier ;x,n,i : entier) :booléen
Début
   i←1 ;
  si (T[n] = x) alors retourner(vrai) ;
  sinon
       T[n] ← x;
  finsi;
   Tant que  (i ≤ n) et (T[i] ≠ x) alors
      i←i+1;
```

```
   fintant que;
   si (i ≤ n) alors
     retourner(vrai) ;
   sinon
     retourner(faux) ;
finsi ;
Fin.
```

→ La complexité est en O(n).

Les opérations de programmation qui peuvent avoir une certaine importance disparaissent avec cette notation asymptotique.

Calcul de complexité des algorithmes itératifs

Les étapes de calcul de complexité sont les suivantes:

Etape 1 : Sous quel aspect doit-on considérer la donnée d'entrée ?

- Soit comme le nombre d'objets donnés → analyse uniforme (classique).
- Soit comme la taille des bits nécessaires pour la représentation → analyse logarithmique.

Pour un nombre x, il faut $\lceil \log_2(x+1) \rceil$ bits, ou $\lfloor \log_2(x) \rfloor + 1$

Etape 2 : Quelles sont les opérations élémentaires, i.e. celles qui se font en temps constant ?

- En analyse uniforme → Opérations arithmétiques +, -, *, /, %.
- En analyse logarithmique → Manipulation de bits : décalage, test à zéro, additions de bits.

Etape 3 : On se donne une base de données de complexité élémentaires et des règles sur les opérations.

- w(i, J) = w(i) + w(I). Le coût d'instructions séquentielles est le coût de l'une plus celui de l'autre.
- w(x ← e) = cost(e). Le coût d'une affectation est le coût du calcul/manipulation de la donnée.
- w(if(c) S_1 else S_2) = w(c) + max(w(S_1), w(S_2)). On paye la condition puis le pire des cas.
- w(while (c), S) = $\Sigma_{\text{itérations}}$(w(c) + w(S)). On paye le test et les opérations à chaque passage.
- w(for(S_1 ; S_2 ; S_3), S_4) = w(S_1) + $\Sigma_{\text{itérations}}$(w($S_2$) + w($S_3$) + w($S_4$)). ↔ S_1 ; while(S_3) { S_2,S_4}.
- w(récursivité) : coût de la récursivité.

Formules utiles pour le calcul

- ln fonction logarithme népérien (ou naturel), de base e
- $\log_a$ fonction logarithme de base a : $\log_a(x) = \ln x / \ln a$
- log fonction logarithme sans base précise, à une constante multiplicative près
- $\log_2$ fonction logarithme binaire, de base 2 : $\log_2(x) = \ln x / \ln 2$

Logarithme : $\log_b n = x \leftrightarrow b^x = n$, pour $n > 0$ et $b > 1$.

$$\sum_{i=1}^{n}(C.Fi + Oi) = \sum_{i=1}^{n} CFi + \sum_{i=1}^{n} Oi$$

$\sum_{i=1}^{n} O[Fi] = O[\sum_{i=1}^{n} Fi]$

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} ix^i = \frac{1}{(1-x)^2} \quad \text{si } |x|<1$$

Problème :

- ⇨ O est un "pire des cas" : il n'indique donc pas que l'algorithme va prendre exactement ce temps, mais qu'il ne peut pas prendre davantage.
- ⇨ La notation O fait disparaître quelques constantes liées à la programmation dont les tests dans les boucles et de petites optimisations.

Calcul de complexité des algorithmes récursifs

- La complexité de l'algorithme récursif est donnée par son équation,
- La résoudre en utilisant un formulaire pour les cas standard.
- Le coût de la récursivité est donné par T(m) tel que :

$T(m) = \alpha$ cas de base

$T(m) = c.T(f(m)) + g(m)$ équation de récurrence

Les paramètres sont :

- α : temps d'exécution dans le cas de base. En général O(1).
- c : nombre de fois qu'on fait un appel récursif.

Exemple 2 : diviser pour régner
Deux appels récursifs (sur les éléments de la première moitié, et de la seconde).
- g(m), le coût d'un appel récursif
- f(m), la taille de la donnée m lors de l'appel récursif, (f(m) = m – 1 ou m – 2).

Formulaire des solutions d'équations de récurrence où T(1) = O(1) :
1. $T(n) = T(n/2) + O(1)$ $\rightarrow T(n) = O(\log n)$
2. $T(n) = T(n-1) + O(\log n)$ $\rightarrow T(n) = O(n.\log n)$
3. $T(n) = c.T(n-1) + O(n^k)$ $\rightarrow T(n) = O(c^k)$
4. $T(n) = c.T(n/d) + O(n^k)$ $\rightarrow$ si $c = d^k$, alors $T(n) = O(n^k.\log n)$
   $\rightarrow$ si $c < d^k$, alors $T(n) = O(n^k)$
   $\rightarrow$ si $c > d^k$, alors $T(n) = O(n^{\log \text{ base d de c}})$

3.1. Master théorème, Équations de récurrences

Théorème 1. Les équations de récurrence :

$$\begin{cases} T(1) = c, \\ T(n) = aT(n/b) + cn, \quad n \geq 2 \end{cases}$$

avec a,b,c > 0 et où n/b représente soit [n/b] soit [n/b], ont pour solution :
- $a < b \Rightarrow T(n) \in \Theta(n)$
- $a = b \Rightarrow T(n) \in \Theta(n\log(n))$
- $a > b \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$

Théorème 2 . Les équations de récurrence :

$$\begin{cases} T(1) = c \\ T(n) = aT(n/b) + c, \quad n \geq 2 \end{cases}$$

avec $a \geq 1, b > 1, c > 0$ et où n/b représente soit [n/b] soit [n/b] ont pour solution :
- $a < b$ si $a = 1 \Rightarrow T(n) \in \Theta(\log(n))$
  si $a > 1 \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$
- $a = b \Rightarrow T(n) \in \Theta(n)$
- $a > b \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$

Remarque 1. Le cas où a=1 et b=2 correspond au processus dichotomique classique. Pour démontrer les deux théorèmes précédents, on peut commencer par considérer les entiers $n = b^t$.

Théorème 3. Les équations de récurrence :

$$\begin{cases} T(1) = c \\ T(n) = aT(n/b) + f(n) , \ n \geq 2 \end{cases}$$

avec $a \geq 1, b > 1$ et où n/b représente soit [n/b] soit [n/b] ont pour solution :

- si $f(n) \in O(n^{\log_b(a)-\varepsilon})$ pour $\epsilon > 0 \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$
- si $f(n) \in \Theta(n^{\log_b(a)}) \Rightarrow T(n) \in \Theta(n^{\log_b(a)}\log(n)) = \Theta(f(n)\log(n))$
- si $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ pour $\epsilon > 0$ et si $af(n/b) \leq df(n)$ avec $d < 1, \Rightarrow T(n) \in \Theta(f(n))$

Remarque 2. Ce théorème ne couvre pas tous les cas possibles de la fonction f. Enfin terminons par les équations d'autres schémas récursifs.

Théorème 4. Les équations de récurrence :

$$\begin{cases} T(1) = a \\ T(n) = bT(n-1) + a, \ n \geq 2 \end{cases}$$

avec $a > 0, b \geq 1$ ont pour solution :

- $b = 1 \Rightarrow T(n) = an \in \Theta(n)$
- $b \geq 1 \Rightarrow T(n) \in \Theta(b^{n-1})$

Remarque 3. Le problème des tours de Hanoï correspond au cas où $a = 1, b = 2$, et $T(n) \in \Theta(2^{n-1})$, donc un algorithme récursif exponentiel.

Théorème 5. Les équations de récurrence :

$$\begin{cases} T(1) = a \\ T(n) = bT(n-1) + an \ , \ n \geq 2 \end{cases}$$

avec $a > 0$, $b \geq 1$ ont pour solution :

- $b = 1 \Rightarrow T(n) \in \Theta(n^2)$
- $b \geq 1 \Rightarrow T(n) \in \Theta(b^{n-1})$

Remarque 4. Le plus mauvais cas de l'algorithme Quicksort correspond au cas où $b = 1, a = 1$, cet algorithme a donc un comportement quadratique dans le pire des cas.

Propriétés.

1. $\begin{cases} T(1) = a \\ T(n) = T(n/5) + T(3n/10) + an, \ n \geq 2 \end{cases} \Rightarrow T(n) \in O(n)$
2. $\begin{cases} T(1) = a \\ T(n) = \sum_{i=1}^{i=k} ai \ T(\frac{n}{bi}) + an, \ n \geq 2 \end{cases} \Rightarrow$ si $\sum_{i=1}^{i=k} \frac{ai}{bi} < 1$ alors $T(n) \in O(n)$

Propriétés.

Soient f et g deux fonctions. Voici une synthèse du calcul de $\lim_{x\to+\infty}\frac{f(x)}{g(x)}$.

$\lim_{x\to+\infty}\frac{f(x)}{g(x)}=0$ implique $f\in O(g) et\, g\notin O(f)$ et $\Theta(f)<\Theta(g)$.

$\lim_{x\to+\infty}\frac{f(x)}{g(x)}=1$ implique $f\sim g$.

$\lim_{x\to+\infty}\frac{f(x)}{g(x)}=c\neq 0$ implique $f\in O(g) et\, g\in O(f)$ et $f\in\Theta(g)$, $g\in\Theta(f)$.

$\lim_{x\to+\infty}\frac{f(x)}{g(x)}=\infty$ implique $f\notin O(g) et\, g\in O(f)$ et $\Theta(f)>\Theta(g)$.