Cours
Structures de données

2ème année SMI, Semestre 4

Département d'Informatique
Faculté des sciences de Rabat

Par
B. AHIOD
ahiod@fsr.ac.ma

2013-2014

Objectifs du Cours

- Résoudre des problèmes en programmant efficacement des structures de données utiles
- Etudier et Manipuler des structures de données de base :
  - *Structures linéaires (piles, files, listes) ;*
  - *Structures arborescentes (arbres binaires, arbres binaires de recherche, tas, arbres équilibrés) ;*
  - *Tables de hachage ;*
  - *Structures relationnelles (graphes)*
- Applications
- Langage de programmation utilisé : *C*

Contenu du Cours

- **Introduction générale (*Rappels*)**
- **Introduction à la complexité des algorithmes**
- **Algorithmes de tri**
- **Notion de Types Abstraits de Données**
- **Structures linéaires :**
  - **Piles, Files, Listes**
- **Structures arborescentes :**
  - **Arbres Binaires, de Recherche, Tas, Arbres équilibrés**
- **Tables de hachage**
- **Graphes**

Pré-requis

- **Notions de base d'algorithmique (*Info$_1$*)**
  - ***Structures de séquence, choix et répétition***
  - ***Analyse descendante***
  - ***Notion d'algorithme récursif***
  - ***…***
- **Programmation en langage C (*Info$_2$*)**
  - ***Programmation structurée et modulaire (compilation séparée)***
  - ***Notions de tableaux, structures et synonymes (`typedef`)***
  - ***Manipulation des pointeurs et allocation dynamique***
  - ***…***

Répartition & Evaluation

Semaine	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Cours	3h	3h	3h	3h	3h	3h		3h	3h	3h	3h	3h		
TDs		1h30	1h30	1h30	1h30	1h30		1h30	1h30	1h30	1h30	1h30	1h30	
TPs		2h	2h	2h	2h	2h		2h	2h	2h	2h	2h	2h	

Semaine éventuellement prévue pour contrôle continu

- **Modalités d'évaluation**
  - Une note de(s) contrôle(s) continu(s) : **CC**
  - Une note de TPs (et projet) : **TP**
  - Une note de contrôle final : **CF**
- **Note du module = ¼ (CC+TP)+1/2 CF**

Introduction générale

(*Rappels*)

Notion de programme (1)

Algorithmes + structures de données
=
Programme

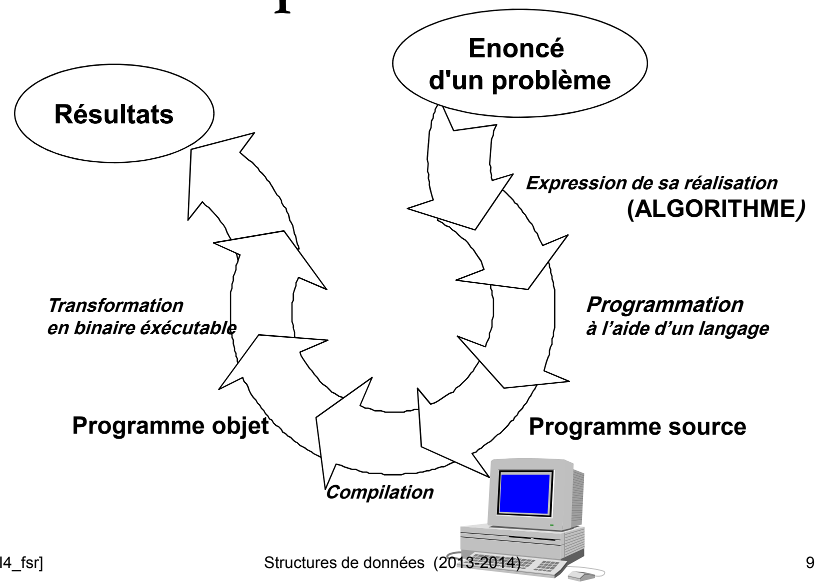
[*Wirth*]

- **Un programme informatique est constitué d'algorithmes et de structures de données manipulées par des algorithmes**

Notion de programme (2)

- **Synonymes**
  - Programme, application, logiciel
- **Objectifs des programmes**
  - Utiliser l'ordinateur pour traiter des données afin d'obtenir des résultats
  - Abstraction par rapport au matériel
- **Un programme est une suite logique d'instructions que l'ordinateur doit exécuter**
  - Chaque programme suit une logique pour réaliser un traitement qui offre des services (*obtention des résultats souhaités à partir de données*)
  - Le processeur se charge d'effectuer les opérations arithmétiques et logiques qui transformeront les données en résultats
- **Programmes et données sont sauvegardés dans des fichiers**
  - Instructions et données doivent résider en mémoire centrale pour être exécutées

Résolution informatique d'un problème



Notion d'algorithme

- Le terme algorithme vient du nom du mathématicien Al-Khawarizmi (*820 après J.C.)*
- Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données
- Le rôle d'un algorithme est fondamental ; sans algorithme il n'y aurait pas de programme
- Un algorithme est indépendant à la fois de l'ordinateur qui l'exécute ; du langage dans lequel il est énoncé et traduit

Spécifier/Exprimer/Implémenter un algorithme

- Spécification d 'un algorithme :
  - *ce que fait l'algorithme*
  - *cahier des charges du problème à résoudre*
- Expression d 'un algorithme :
  - *comment il le fait*
  - *texte dans un pseudo langage*
- Implémentation d 'un algorithme :
  - *traduction du texte précédent*
  - *dans un langage de programmation réel*

Exemple : Recherche d'un élément

Spécification d'un algorithme

```
/* Cet algorithme recherche la place d'un élément val
   dans un tableau tab contenant n éléments */
```

```
Algorithme recherche_sequentielle(tab: entier[]; n, val: entier) : entier
 entrées : tab, n et val
 sortie : indice de val dans le tableau tab, sinon -1
Début
  variables locales : i: entier
  i ← 0;                                      // 1
  tant que ((i<n) ou (tab[i] <> val)) faire   // 2
       i ← i+1                                // 3
  ftq                                         // 4
  si (i = n) alors retourner -1               // 5
  sinon retourner i                           // 6
Fin
```

L'algorithme traduit en C

L'algorithme en pseudo code

```
int recherche_sequentielle(int *tab, int n, int val) {
 int i;                                      // 1
 i = 0;                                      // 2
 while ((i<n) && (tab[i] != val))            // 3
    i ++;                                    // 4
 if (i == n)                                 // 5
   return(-1);                               // 6
 else return(i);                             // 7
}
```

Analyse descendante

- **Consiste à décomposer un problème en sous problèmes, eux-mêmes, à décomposer en sous problèmes, et ainsi de suite jusqu'à descendre à des actions dites primitives**
  - ***Les étapes successives de décomposition donnent lieu à des sous algorithmes pouvant être considérés comme des actions dites intermédiaires***
  - ***Ces étapes sont appelées fonctions ou encore procédures***

Notion d'algorithme récursif

- **Un algorithme est dit *récursif* lorsqu'*il s'appelle lui même* de façon directe ou indirecte.**

- **Pour trouver une solution récursive d'un problème, on cherche à le décomposer en plusieurs sous problèmes de même type, mais de taille inférieure.**

  On procède de la manière suivante :
  - **Rechercher un** (*ou plusieurs*) **cas de base** et sa (*ou leur*) solution (*évaluation sans récursivité*)
  - **Décomposer le cas général** en cas plus simples eux aussi décomposables pour *aboutir au cas de base*.

Algorithme récursif
Avantages & Inconvénients

- Avantages
  - Un problème pouvant être décrit de manière récursive est souvent plus simple et plus concis à formaliser
  - Algorithme beaucoup plus facile à comprendre et à maintenir

- Inconvénients
  - Temps d'exécution peut être plus long
  - Possibilité de grande occupation de la mémoire

Exemple d'algorithme itératif

```
/* Calcul de la somme des carrés des entiers entre m et n
   (version itérative) */
Algorithme SommeCarres_iter(m: entier; n: entier) : entier
 entrées : m et n
 sortie : somme des carrés des entiers entre m et n inclus,
          si m<=n, et 0 sinon
Début
  variables locales : i, som: entier

  som ← 0;                                  // 1
  pour i de m à n faire                     // 2
       som ← som + (i*i)                    // 3
  fpour                                     // 4
  retourner som                             // 5
Fin
```

Exemple d'algorithme récursif

```
/* Calcul de la somme des carrés des entiers entre m et n
   (version récursive) */
Algorithme SommeCarres_rec(m: entier; n: entier) : entier
 entrées : m et n
 sortie : somme des carrés des entiers entre m et n
 pré-condition : m<=n
Début
  si (m<>n) alors                                  // 1
      retourner ((m*m)+SommeCarres_rec(m+1,n)      // 2
  sinon                                            // 3
      retourner (m*m)                              // 4
  fsi
Fin
```

Cas de base (ou cas trivial)

Qualités d'un algorithme

- **Clarté** : **un algorithme doit être structuré, indenté, modulaire, avec des commentaires pertinents, etc.**
- **Terminaison : le résultat doit être atteint en un nombre fini d'étapes. (*techniques de preuves de terminaison de la théorie des programmes)***
- **Validité : le résultat doit répondre au problème demandé. (*méthodes des assertions (Hoare, Floyd)*)**
- **Efficacité : étude du coût (*complexité*) en temps et en mémoire.**

Introduction à la complexité des algorithmes

Motivation

- **Pour un problème donné, il peut exister plusieurs algorithmes pour le résoudre. Par exemple :**
  - ***Problème de tri*** → algorithmes possibles (*tri par sélection, tri par insertion, tri à bulles, tri rapide, tri par fusion, …*)
  - ***Problème de recherche*** → algorithmes possibles (*recherche séquentielle, recherche dichotomique, …*)

- ***Il faut donc une analyse de la complexité des algorithmes !***

Analyse de Complexité Algorithmique

- **Consiste à évaluer les ressources nécessaires à l'exécution d'un algorithme**

- **Deux ressources critiques :**
  - ***le temps d'exécution***
  - ***la place mémoire***

- **Le but est de choisir (*dans des conditions de réalisation égales*) l'algorithme, le mieux adapté et le plus efficace (*le moins coûteux*) pour résoudre un problème.**

- **On s'intéresse à l'évaluation du temps d'exécution d'un algorithme : la complexité temporelle**

Comment Evaluer ? (1)

- **De façon empirique (*expérimentale*) :**
  - on programme l'algorithme
  - on exécute le programme sur des ensembles de données variés
  - on mesure le temps d'exécution

- **Ceci dépend de plusieurs facteurs :**
  - la machine utilisée
  - le langage de programmation
  - le compilateur
  - les données
  - …

- **On veut pouvoir dire :**
  - **Sur toute machine, quel que soit le langage de programmation, l'algorithme A1 est meilleur que l'algorithme A2 pour les données de grande taille**

Comment Evaluer ? (2)

- De façon formelle (*théorique*) : on compte le nombre d'opérations caractéristiques de l'algorithme

- On se base sur un modèle de machine abstraite sur laquelle l'algorithme sera implémenté (*sous forme de programme*)

- On prendra comme référence *un modèle de machine à accès aléatoire* (*RAM*), munie d'une mémoire et d'un processeur unique, où les instructions sont exécutées l'une après l'autre, sans opérations simultanées

Notion de Taille de Données

- **Le temps d'exécution d'un algorithme doit être défini comme une fonction de la taille des données**

  - **Exemples :**
    - **le nombre d'éléments à trier,**
    - **le nombre de chiffres dans l'écriture des nombres à multiplier,**
    - **la dimension des matrices à multiplier,**
    - **…**

Représenter le Temps d'Exécution

- **Le temps d'exécution (*ou complexité algorithmique*) désigne le nombre d'opérations élémentaires effectuées par un algorithme**
- **Il s'exprime en fonction de la taille n des données. On note : T(n)**
- **T(n) donne une évaluation du nombre d'opérations élémentaires à exécuter pour des données de taille n**

Notion d'Opération Elémentaire

- **Une opération est élémentaire lorsque son temps d'exécution est constant, c'est-à-dire ne dépend pas de la taille des données**
- **Exemples :**
  - Les opérations suivantes sont élémentaires : ***Appel et retour d'une fonction***, ***Effectuer une opération arithmétique***, ***Comparer deux nombres***, ***etc.***
  - **Le test d'appartenance d'un élément à un ensemble n'est pas une opération élémentaire parce que son temps d'exécution dépend de la taille de l'ensemble**

Différentes Complexités (1)

- **Dans la pratique, le temps d'exécution d'un algorithme dépend non seulement de *la taille des données*, mais aussi de *leurs valeurs précises***

- **Exemple :**
  - ***Pour la recherche séquentielle d'un élément dans une liste de n éléments*, le temps d'exécution varie selon que l'élément se trouve en première position, ou que l'élément n'est pas présent dans la liste**

  - ***Pour le tri d'une liste*, le temps d'exécution varie selon que la liste est déjà triée ou non**

Différentes Complexités (2)

- **Complexité dans le pire cas (T_{max} (n)) :** temps d'exécution maximal d'un algorithme pour des données de taille n

- **Complexité moyenne (T_{moy}(n)) :** temps moyen d'exécution d'un algorithme sur tous les jeux de données de taille n

- **Complexité dans le meilleur cas (T_{min}(n)) :** temps d'exécution minimal d'un algorithme pour des données de taille n

Complexité dans le Pire Cas ?

- **Donne idée de borne supérieure du temps d'exécution pour une entrée quelconque**
- **De nombreux algorithmes fonctionnent assez souvent dans le pire cas. Par exemple, la recherche d'une information dans une base de données**
- **Le pire cas est d'importance cruciale pour certaines applications (par exemple, contrôle aérien, chirurgie, gestion de réseaux)**
- **Le meilleur cas apporte peu d'informations car beaucoup d'algorithmes font exactement la même chose dans une telle situation**
- **La complexité en moyenne est en général plus difficile à calculer, car il faut que soit précisée une distribution de probabilité pour les données de taille n**

Règles de Calcul de Complexité

- ***Les opérations élémentaires*** **ont un temps d'exécution constant**
- ***La*** **complexité d'une** ***séquence d'instructions*** **est la somme des complexités des instructions qui la composent**
- ***La complexité d'une instruction conditionnelle*** **est égale à la somme de la complexité du test et du maximum des complexités des deux alternatives**
- ***La complexité d'une instruction répétitive*** **est la somme sur toutes les répétitions, de la somme de la complexité du corps de la boucle et de la complexité de la condition de boucle**
- ***La complexité d'une fonction*** **est déterminée par celle de son corps :**
  - ***pour une fonction récursive*****, elle est exprimée comme une équation de récurrence ;**
  - ***pour une fonction itérative*****, elle se calcule en sachant que l'appel à une fonction prend un temps constant.**

Exemple

	Nombre d'opérations élémentaires (pire cas)
`/* Recherche du maximum d'un tableau d'entiers */`	
`Algorithme MaxTab(T: entier[]; n: entier) : entier`	
`entrées : un tableau T de n >= 1 entiers`	
`sortie : l'élément maximum de T`	
`Début`	
`variables locales : i, max: entier`	
`max ← T[0];   // 1`	1+1
`pour i de 1 à n-1 faire   // 2`	1+n+2(n-1)
`si (T[i]>max) alors   // 3`	2(n-1)
`max ← T[i]   // 4`	2(n-1)
`fsi   // 5`	
`fpour   // 6`	
`retourner max   // 7`	1
`Fin`	

Donc T(n) = 7 n – 2 (pire cas)

Complexité Asymptotique

- **On s'intéresse à une "*estimation asymptotique*" (*i.e. pour n grand*) du temps d'exécution des algorithmes**
- **On évalue l'efficacité d'un algorithme en donnant un "*ordre de grandeur*" du nombre d'opérations T(n) qu'il effectue lorsque la taille n du problème qu'il résout augmente**
- **On ne calcule ni le temps réel d'exécution sur une machine précise, ni le nombre exact d'instructions exécutées**
- **Pour exprimer cet ordre de grandeur, on utilise *les notations de Landau.* La plus fréquente, la notation O (*grand O*) introduite par la suite, donne une majoration de l'ordre de grandeur**

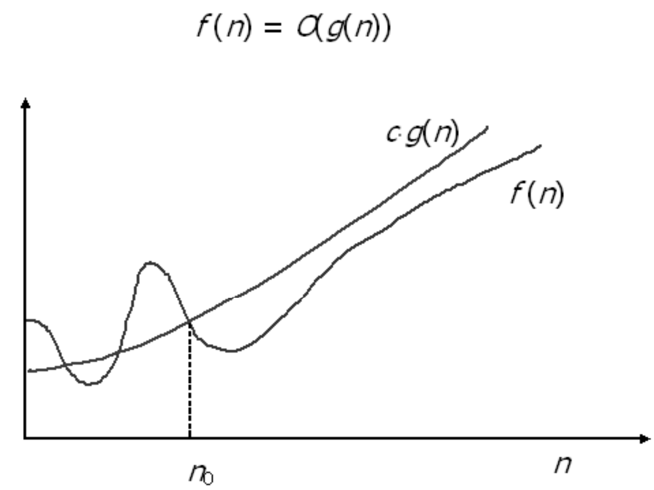
Notation O (grand O)

- **Soient T(n) et f(n), deux fonctions réelles positives du paramètre entier n**
- **On dit que T(n) est en O(f(n)) (*en grand O de f(n)*) si et seulement si il existe un entier n_0 et une constante c > 0 tels que : $T(n) \leq c\, f(n)$, pour tout $n \geq n_0$**
- **On dit aussi que T(n) est de l'ordre de f(n). Par abus de notation, on écrit : T(n) = O(f(n))**
- **Soit maintenant A un algorithme de temps d'exécution T(n). On dira que A est de complexité O(f(n))**

Exemples

- **$T(n) = (n+1)^2$ est en $O(n^2)$. Pour le démontrer, prendre $n_0 = 1$ et c = 4**
- **$T(n)=3(n^3)+2(n^2)$ est en $O(n^3)$. Pour le démontrer, prendre $n_0 = 0$ et c = 5**
  - **Si T(n) est le temps d'exécution d'un algorithme A, A est donc $O(n^3)$. On pourrait aussi dire que A est $O(n^4)$, mais, ce serait un énoncé plus faible**
- **$T(n) = 3^n$ n'est pas en $O(2^n)$**

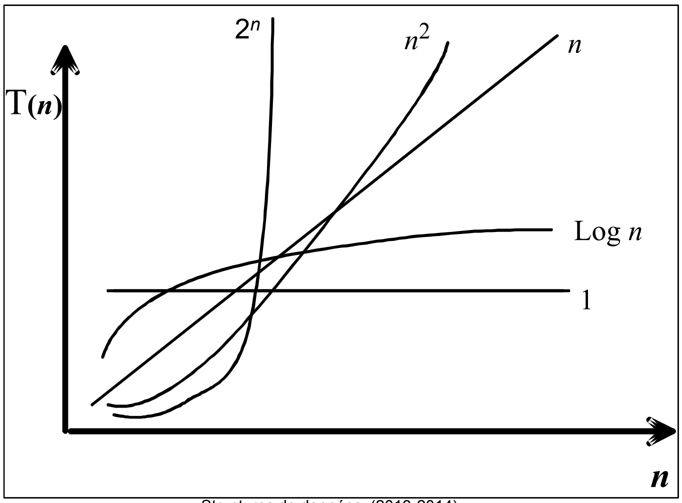
Illustration du grand O



Grandes Classes de Complexité

Grand O	Complexité	Exemples d'algorithmes
O(1)	**Constante**	*Renvoi du premier élément d'une liste de longueur n*
O(log(n))	**Logarithmique**	*Recherche dichotomique dans un tableau trié de taille n*
O(n)	**Linéaire**	*Recherche séquentielle dans un tableau de taille n*
O(n log(n))	**n log n**	*Tri rapide, tri par tas d'une liste de longueur n*
O(n^2)	**Quadratique**	*Tris « naïfs » d'une liste de longueur n*
O(n^3)	**Cubique**	*Multiplication de deux matrices carrées d'ordre n*
O(2^n)	**Exponentielle**	*Tours de Hanoi avec n disques*

Ordres de Grandeur (*exemples*)
1, log n, n, n^2, n^3, 2^n



Quelques Règles de Simplification

- **Si T(n) = O(k f(n)) où k > 0, alors T(n) = O(f(n))**
  - ***Exemple :*** 2n+10 **est O(n), par contre n^2 n'est pas O(n)**
- **Dans un polynôme, seul le terme de plus haut degré compte**
  - ***Exemple :*** **3(n^3)+20(n^2)+5 est O(n^3)**
- **Une exponentielle "l'emporte" sur une puissance, et cette dernière sur un log**
  - ***Exemple :*** (2^n)+(n^{100}) **est O(2^n) et** 100log(n)+n **est O(n)**
- **Si $T_1(n)$ = O(f(n)) et $T_2(n)$ = O(g(n)) alors**
  $T_1(n)+T_2(n)$ = O(Max(f(n),g(n)))
  - ***Exemple :*** **$T_1(n)$ = n et $T_2(n)$ = n^2 décrivent deux tâches qui s'exécutent l'une à la suite de l'autre**
- **Si $T_1(n)$ = O(f(n)) et $T_2(n)$ = O(g(n)) alors $T_1(n).T_2(n)$ = O(f(n).g(n))**
  - ***Exemple :*** **une tâche provoque l'exécution d'une autre plusieurs fois dans chacune de ses itérations (*boucles imbriquées*)**

Exemples (1)

- *Boucles simples*
  - `for (i=0; i<n; i++) { s; }`

    où `s` est en $O(1)$
  - la complexité en temps est en $O(n)$

- *Boucles imbriquées*
  - ```
    for(i=0; i<n; i++)
        for (j=0; j<n; j++) { s; }
    ```
  - la complexité est en $O(n^2)$

Exemples (2)

- *Boucles logarithmiques*
  - ```
    i=1;
    while (i<=n){
        s;
        i = 2 * i;
    }
    ```
  - i prend les valeurs 1, 2, 4, …, jusqu'à ce qu'elle dépasse n
  - il y a $1 + \log_2 n$ itérations
  - la complexité est en $O(\log n)$

- *L'indice de la boucle interne dépend de l'indice de la boucle externe*
  - ```
    for (j=0; j<n; j++)
      for (k=0; k<j; k++)
        s;
    ```
  - la boucle interne s'exécute : 1, 2, 3, …., n fois. En sommant sur j, pour j allant de 1 à n, on a : n (n+1)/2
  - la complexité est en $O(n^2)$
    ```

Temps d'Exécution pour différentes tailles des données

s : seconde
µs : 10^{-6} s
ms : 10^{-3} s

a : année
j : jour
h : heure
mn : minute

Taille des données	Complexité						
	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	1 µs	6.64 µs	0.1 ms	0.66 ms	10 ms	1 s	4.02×10^{16} a
$n = 10^3$	1 µs	9.97 µs	1 ms	9.97 ms	1 s	16.67 mn	∞
$n = 10^4$	1 µs	13.31 µs	10 ms	0.13 s	100 s	11.57 j	∞
$n = 10^5$	1 µs	16.61 µs	0.1 s	1.6 6s	2.78 h	31.71 a	∞
$n = 10^6$	1 µs	19.93 µs	1 s	19.93 s	11.57 j	31.7×10^3 a	∞

dépasse 10^{100}

On suppose une machine exécutant 10^6 opérations par seconde

Taille Maximum des Données pouvant être traitées en temps donné

Temps donné	Complexité						
	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1 s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 j	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

dépasse 10^{100}

Evolutions mutuelles du temps et de la taille des données

	Complexité						
	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	t + 3.32	10 x t	(10 + ε) x t	100 x t	1000 x t	t^{10}
Evolution de la taille quand le temps est multiplié par 10	∞	n^{10}	10 x n	(10 - ε) x n	3.16 x n	2.15 x n	n +3.32

dépasse 10^{100}

Comment Apprécier un Algorithme ?

- **Pour choisir un algorithme pour résoudre un problème, deux objectifs souvent contradictoires:**
  - algorithme facile à comprendre, coder et déboguer ;
  - algorithme rapide, qui utilise de manière efficace les ressources disponibles.
- **Il y a des situations où la “complexité asymptotique" n'est pas le problème le plus important. Par exemple :**
  - Un programme qui ne tournera que quelques fois. Inutile alors de passer des heures pour gagner quelques secondes ou minutes!
  - Un programme qui doit tourner sur de "petites" données. C'est le type de situation où un algorithme en $O(n^3)$ peut être plus efficace qu'un autre en $O(n^2)$ (*selon la valeur de "la constante"*).
  - Un algorithme efficace mais compliqué peut être indésirable si une personne autre que le programmeur doit ultérieurement en assurer la maintenance.
  - Un bon programmeur doit avoir le souci de l'efficacité, connaître les algorithmes efficaces pour certains problèmes courants, et les techniques qui permettent de les écrire.

Grand Omega (grand Ω)

- **Soient f(n) et g(n), deux fonctions réelles positives du paramètre entier n.**
- **On dit que f(n) est Ω(g(n)) si et seulement si il existe un entier n_0 et une constante $c > 0$ tels que: $f(n) \geq c\, g(n)$, pour tout $n \geq n_0$**
- **Exemple :** $f(n) = 60n^2 + 5n + 1$ est **Ω**(n^2).
  - Pour le voir, prendre $n_0=1$ et $c=60$

Grand Thêta (grand θ)

- **Soient f(n) et g(n), deux fonctions réelles positives du paramètre entier n**
- **On dit que f(n) est θ(g(n)) si et seulement si f(n) est à la fois O(g(n)) et Ω(g(n)), i.e. s'il existe un entier n_0 et deux constantes $c_1 > 0$ et $c_2 > 0$ tels que: $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$, pour tout $n \geq n_0$**
- **Exemple :** $f(n) = 60n^2 + 5n + 1$ est $\theta(n^2)$, car :
  - f(n) est **O**(n^2) (*prendre $n_0=1$ et $c_2=66$*)
  - f(n) est **Ω**(n^2) (*prendre $n_0=1$ et $c_1=60$*)

Exemples d’Analyse Asymptotique

Algorithmes itératifs

Produit de 2 Matrices A(n,p) et B(p,m)

- L'algorithme traduit en C :

```
void multiplier(int A[][p], int B[][m], int C[][m],
                              int n, int m, int p) {
 for (i = 0; i<n; i++)                          // 1
    for (j=0; j<m; j++){                        // 2
       S = 0;                                   // 3
       for (k =0; k<p; k++)                     // 4
          S = S + A[i][k]*B[k][j];              // 5
       C[i][j] = S;                             // 6
    }
}
```

- Après analyse, la complexité de l’algorithme est en O(n m p)

Recherche Séquentielle dans un Tableau de n entiers

- **L'algorithme traduit en C :**

```
int recherche_sequentielle(int *tab, int n, int val) {
 int i;                                          // 1
 i = 0;                                          // 2
 while  ((i<n)  &&  (tab[i] != val))             // 3
    i ++;                                        // 4
 if  (i == n)                                    // 5
   return(-1);                                   // 6
 else return(i);                                 // 7
}
```

- **Ici, la complexité de l'algorithme dépend des valeurs des données. Après analyse, la complexité est :**
  - **en O(1) dans le meilleur cas (*val en première position dans le tableau*) ;**
  - **en O(n) dans le pire cas (*val n'est pas dans le tableau*)**

Recherche Dichotomique dans un Tableau Trié de n entiers (1)

Algorithme traduit en C

```
int  recherche(int *tab, int n, int val){
    int sup, inf, milieu;                    // 1
    int trouve;                              // 2
    inf = 0; sup = n-1; trouve = false;      // 3
    while (sup >=inf && !trouve) {           // 4
           milieu = (inf + sup) / 2;         // 5
           if (val == tab[milieu])           // 6
               trouve = true;                // 7
           else if (val < tab[milieu])       // 8
                      sup = milieu -1;       // 9
                else inf = milieu + 1;       // 10
    if (!trouve)                             // 11
      return(-1);                            // 12
    return(milieu);                          // 13
 }
```

Recherche Dichotomique dans un Tableau Trié de n entiers (2)

- ***Meilleur cas :*** la valeur recherchée val est au milieu du tableau. L'algorithme est en O(1).

- ***Pire cas :*** la valeur recherchée val n'existe pas.
  Combien d'itérations sont nécessaires ? On constate qu'après chaque itération, l'ensemble de recherche est divisé par deux. Au départ, cet intervalle est égal à sup (= n-1) – inf (= 0) + 1 = n.

itération	0	1	2	3	…	i
Intervalle de recherche	n	n/2	n/4	n/8	…	$n/2^i$

On arrêtera les itérations de la boucle while dès que la condition suivante est vérifiée : $n/2^i = 1 \Rightarrow i = O(\log n)$

Après analyse, la complexité de cet algorithme dans le pire cas est en O(log(n)).

Exemples d'Analyse Asymptotique

Algorithmes récursifs (*simples*)

Calcul Récursif du Factoriel

- **L'algorithme traduit en C :**

```
long factoriel(int n) {
 if (n < 2) return 1;          // 1
 return n * factoriel(n-1);    // 2
}
```

- **Analyse de complexité :**

Soit **`T(n)`** la complexité de la fonction **`factoriel(n)`**. On a ***l'équation de récurrence*** suivante :

```
T(n) = T(n-1) + b; si n>=2     (a et b des constantes)
T(n) = a  sinon
```

Pour connaître **`T(n)`**, on résout l'équation comme suit :

```
T(n)   = T(n-1) + b
T(n-1) = T(n-2) + b
T(n-2) = T(n-3) + b
      ...
T(2)   = T(1) + b
```

En additionnant membre à membre, on obtient : **`T(n) = a + b(n-1)`**

Après analyse, la complexité du factoriel est en O(n)