Mémo n. 5

Analyse de la complexité des algorithmes

I32 Preuves et Analyses d'algorithmes

P. Véron

Objectif : Obtenir des résultats sur les principales caractéristiques d'un algorithme à partir desquels on pourra dériver une estimation précise des ressources nécessaires (disque, temps CPU) pour l'exécution de cet algorithme sur une certaine machine. L'analyse des algorithmes permet notamment de :

- pouvoir comparer entre eux différents algorithmes résolvant le même problème,
- déterminer si un algorithme donné est exécutable en un temps acceptable sur des données d'une certaine taille,
- fournir une borne supérieure de la taille des données pour lesquelles l'algorithme est utilisable.

1. Une analyse, pouquoi faire ?

Soit T un tableau de n entiers que l'on désire trier dans l'ordre croissant. On suppose que l'on dispose de deux algorithmes permettant de réaliser ce travail :

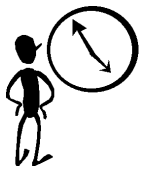
- l'algorithme 1 triant le tableau en n^2 opérations,
- l'algorithme 2 triant le tableau en $n \log_2 n$ opérations,

Pour exécuter nos deux algorithmes , on dispose de deux machines :

- M_1 capable d'effectuer 2^{29} opérations par seconde (Pentium III 450),
- M_2 capable d'effectuer 2^{25} opérations par seconde (486 DX 33),

L'algorithme 1 est exécuté sur M_1 et l'algorithme 2 sur M_2. Le tableau à traiter comporte 2^{20} entiers. Les temps d'exécution sont les suivants :

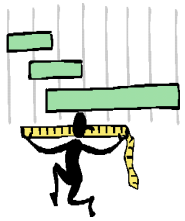
. Algo 1 / M_1 : $(2^{20})^2/2^{29}$ secs. $=$ 2^{11} secs. $\simeq$ 34 minutes
. Algo 2 / M_2 : $(2^{20} \times 20)/2^{25}$ secs. $=$ 0.625 secondes !



Moralité : il ne suffit pas d'avoir une machine puissante, encore faut-il développer de **bons** algorithmes !

2. Une analyse, comment faire ?

Si vous aviez à calculer le résultat de la somme des entiers 5 et 3, il vous suffirait d'une seule addition pour effectuer ce travail. Supposons à présent que l'on vous demande d'ajouter les entiers 9863 et 7498. A moins d'être un génie du calcul mental, vous allez vous ramener à une suite d'additions entre chacun des chiffres composant ces 2 entiers afin d'obtenir le résultat. Ainsi, les données à traiter étant de taille trop "'importantes" pour effectuer le calcul en une seule addition, vous les avez naturellement découpées en données de taille plus petite sur lesquelles il vous était plus facile de réaliser une addition. Le calcul initial vous aura donc demandé au moins 4 additions (sans compter la gestion de la retenue). Il apparaît donc que selon la taille des données à traiter, certaines opérations sont pour vous (ou ne sont pas) "élementaires".



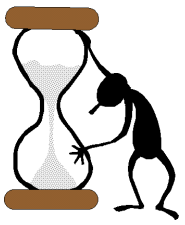
Pour analyser correctement un algorithme, il faut tout d'abord déterminer quelles opérations sont considérées comme élémentaires et surtout pour quelle taille de données. On pourra alors exprimer la complexité de l'algorithme en fonction du nombre d'opérations élémentaires à réaliser sur des données de taille élémentaire (cf. l'exemple précédent où nous avons exprimé que le calcul de $9863 + 7498$ nécessite pour un être humain au moins 4 additions sur des données de taille 1).

Pour les algorithmes étudiés en I32, on supposera que les opérations $+$, $-$, $\times$, $/$ sont des opérations élémentaires sur toutes données de type entier ou réel. Ainsi le calcul de $a+b$ (a et b deux entiers) ne nécessite qu'une addition. Ce résultat devient faux dès lors que l'on considère un programme C réalisant la même opération. En effet, pour ce langage, cette opération est élémentaire si a, b et $a+b$ sont inférieurs au plus grand entier IMAX représentable en machine. Si on doit ajouter des entiers qui sont supérieurs à IMAX, il va falloir les décomposer dans une certaine base et les représenter par un tableau contenant les symboles de leur décomposition dans la base chosie.
Exemple : Considérons $a = 5000090000$ et $b = 6000910000$, ces deux entiers ne peuvent pas être stockés dans une variable de type `unsigned int` en C. Choisissons pour base $B = 1000000$, dans cette base a est représenté par $(5000, 90000)$ et b par $(6000, 910000)$, chacun de ces entiers pouvant être stockés dans une variable de type `unsigned int`. On peut donc représenter a et b par deux tableaux à 2 éléments et le cacul de $a+b$ reviendra à effectuer au moins 4 additions sur des variables de type `unsigned int`.

3. Temps d'exécution d'un algorithme

Généralement on analyse les algorithmes afin de pouvoir prédire leur temps d'exécution. Pour cela on adopte la stratégie suivante : on considère que chaque ligne de l'algorithme s'exécute en un temps constant indépendant de la taille de la donnée. On note c_i le temps d'exécution de la ligne numéro i. Chaque ligne de l'algorithme est suivi de son coût c_i ainsi que du nombre de fois où l'instruction est exécutée. Le temps total d'exécution d'un algorithme traitant une donnée de taille n, noté $T(n)$ est égal alors à la somme des coûts pondérés par le nombre d'exécution de chaque ligne.
Exemple : On veut calculer la somme des éléments d'un tableau A de n entiers. La donnée à traiter est donc de taille n puisqu'on sait additionner de façon élémentaire deux éléments du tableau.

		Coût	fois
1	Algo somme		
2	**données**		
3	A : *tableau de n entiers*		
4	s, i : *entiers*		
5	**début**		
6	*lire(A)*	c_1	1
7	$s \leftarrow 0$	c_2	1
8	$i \leftarrow 1$	c_3	1
9	**tant que** $i \leqslant n$ **faire**	c_4	$n+1$
10	$s \leftarrow s + A[i]$	c_5	n
11	$i \leftarrow i + 1$	c_6	n
12	**fintq**		
13	*ecrire(s)*	c_7	1
14	**fin**		



Dans ce cas on a

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_7 + (c_4 + c_5 + c_6)n \\ &= \alpha n + \beta \end{aligned}$$

où α et β sont deux variables indépendantes de n.

4. Algorithme du tri insertion

Principe : On dispose d'un tableau A de n entiers que l'on veut ordonner. A la $j^{\text{ème}}$ étape $(2 \leqslant j \leqslant n)$ on suppose que les éléments $A[1], \ldots A[j-1]$ sont rangés dans l'ordre croissant. On cherche alors où insérer l'élément $A[j]$ en le comparant successivement aux éléments $A[j-1], \ldots, A[1]$ de façon à conserver cet ordre. Ce principe correspond exactement à la méthode que l'on adopte lorsque l'on dispose dans une main un certain nombre de cartes à jouer ordonnées et que l'on tire une nouvelle carte que l'on doit ranger.

```
1  Algo tri-insertion                                          Coût    fois
2  données
3    A : tableau de n entiers
4    garde, i, j : entiers
5  début
6      lire(A)                                                 c1      1
7      j ← 2                                                   c2      1
8      tant que j ⩽ n faire                                    c3      n
9              garde ← A[j]                                    c4      n − 1
10             i ← j − 1                                       c5      n − 1
11             tant que (i > 0) et (A[i] > garde) faire        c6      ∑_{j=2}^{n} t_j
12                     A[i + 1] ← A[i]                         c7      ∑_{j=2}^{n} (t_j − 1)
13                     i ← i − 1                               c8      ∑_{j=2}^{n} (t_j − 1)
14             fintq
15             A[i + 1] ← garde                                c9      n − 1
16             j ← j + 1                                       c10     n − 1
17     fintq
18 fin
```

t_j correspond au nombre de fois où le test $(i > 0)$ et $(A[i] > garde)$ est évalué pour un j donné. On en déduit alors :

$$\begin{aligned} T(n) &= c_1 + c_2 - c_4 - c_5 - c_9 - c_{10} + (c_3 + c_4 + c_5 + c_9 + c_{10})n + c_6 \textstyle\sum_{j=2}^{n} t_j + \\ &\quad (c_7 + c_8) \textstyle\sum_{j=2}^{n} (t_j - 1) \\ &= \alpha + \beta n + \gamma \textstyle\sum_{j=2}^{n} t_j + \delta \sum_{j=2}^{n} (t_j - 1) \end{aligned}$$

$\alpha, \beta, \gamma, \delta$ étant des constantes. Contrairement à l'algorithme *somme*, il nous est impossible de conclure quant à la valeur de $T(n)$ en fonction de n, le paramètre t_j étant inconnu, i.e. à j fixé il est impossible de savoir exactement combien de fois est exécuté le test ceci dépendant essentiellement :

- de la nature du tableau,
- de la case en cours.

Nous devons alors **dans cette situation** réaliser une étude de cas nous permettant d'encadrer la valeur de $T(n)$.

5. Cas favorables, cas défavorables

Soit $\mathcal{A}$ un algorithme traitant une donnée de taille n. Notons D_n l'ensemble de toutes les données de taille n pouvant être traitées par $\mathcal{A}$. Soit $d \in D_n$ on note $T_d(n)$ le temps d'exécution nécessaire au traitement de la donnée d.

Définition : On appelle cas favorable toute donnée $d' \in D_n$ telle que

$$T_{d'}(n) = \min\{T_d(n),\ d \in D_n\} \overset{(notation)}{=} T_{\min}(n)$$

Définition : On appelle cas défavorable toute donnée $d' \in D_n$ telle que

$$T_{d'}(n) = \max\{T_d(n),\ d \in D_n\} \overset{(notation)}{=} T_{\max}(n)$$

Proposition : Pour toute donnée $d \in D_n$, on a $T_{\min}(n) \leqslant T_d(n) \leqslant T_{\max}(n)$.

Nous allons donc essayer de déterminer pour l'algorithme du tri-insertion les valeurs de $T_{\min}(n)$ et $T_{\max}(n)$ afin d'encadrer pour tout $d \in D_n$ la valeur de $T_d(n)$ (que nous noterons dorénavant $T(n)$).

Cas favorable

Le cas favorable correspond à la situation où l'on exécute un nombre d'opérations minimal dans l'algorithme pour un n donné. Or mis à part les opérations intervenant dans la deuxième boucle, toutes les autres sont effectuées un nombre fixe de fois en fonction de n. Le cas favorable correspond donc à la situation où pour toute valeur de j le test de la deuxième boucle est faux dès sa première évaluation (i.e on ne passe jamais dans la deuxième boucle), ce qui revient à dire que $\forall j, 2 \leqslant j \leqslant n$, $A[j-1] \leqslant A[j]$, ou encore $\forall j, 2 \leqslant j \leqslant n$, $t_j = 1$. Remarquez que ceci signifie que le tableau est déjà ordonné dans l'ordre croissant ! Dans ce cas on a $T_{\min}(n) = \alpha - \gamma + (\beta + \gamma)n$.

Cas défavorable

Ce cas correspond à la situation où l'on exécute un nombre d'opérations maximal dans l'algorithme pour un n donné. Ce qui revient ici à exécuter systématiquement pour tout j les instructions de la deuxième boucle pour toutes les valeurs de i, i.e $\forall j, 2 \leqslant j \leqslant n$, $t_j = j$. Ce qui se traduit aussi par $\forall j, 2 \leqslant j \leqslant n,\ A[j] < A[i]$, $i = 1, \ldots, j-1$. Remarquez qu'une telle condition implique que le tableau d'origine était ordonné dans l'ordre décroissant. On obtient alors :

$$T_{\max}(n) \quad = \quad \alpha - \gamma + (\beta + \gamma/2 - \delta/2)n + \tfrac{\gamma+\delta}{2} n^2$$

On en conclut donc que pour l'algorithme du tri-insertion, il existe des constantes α, β, γ et δ telles que :

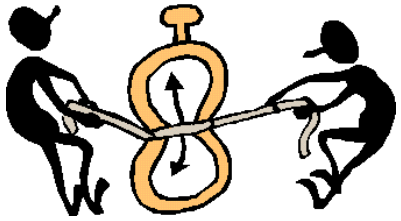
$$\alpha - \gamma + (\beta + \gamma)n \leqslant T(n) \leqslant \alpha - \gamma + (\beta + \gamma/2 - \delta/2)n + \frac{\gamma + \delta}{2} n^2$$

6. Analyse en fonction du nombre d'opérations

Analyser le temps d'exécution d'un algorithme aboutit souvent à des calculs assez lourds étant donné qu'il faut attribuer à chaque ligne de l'algorithme un coût c_i. Or dans un programme, certaines opérations sont plus exigentes que d'autres au niveau des ressources machines nécessaires (par exemple, une multiplication nécessite plus de cycles CPU qu'une addition). Pour analyser un algorithme, on met alors en évidence une (ou plusieurs) opération(s) dite(s) "significative(s)" et on compte en fonction de la taille de la donnée combien d'opérations significatives sont exécutées. Nous noterons cette quantité $Op(n)$.

Exemple : Considérons l'algorithme calculant le produit de la matrice $A(n, n)$ avec un vecteur x de longueur n.

```
1  Algo produit
2  données
3     A : tableau de (n, n) entiers
4     y, x : tableaux de n entiers
5     s, i, j : entiers
6     début
7        lire(A)
8        lire(x)
9        i ← 1
10       tant que i ⩽ n faire
11                s ← 0
12                j ← 1
13                tant que j ⩽ n faire
14                          s ← s + A[i, j] × x[j]
15                          j ← j + 1
16                fintq
17                y[i] ← s
18                i ← i + 1
19       fintq
20    fin
```



Considérons comme opération significative la multiplication. A i et j fixés, on effectue une seule multiplication pour la mise à jour de s. La variable j variant de 1 à n, on effectue à i fixé, $\sum_{j=1}^{n} 1 = n$ multiplications. L'entier i variant de 1 à n, on a $Op(n) = \sum_{i=1}^{n} n = n^2$ multiplications.

Remarque : Tout comme pour le calcul de $T(n)$, il se peut qu'il soit impossible de calculer $Op(n)$ directement et qu'il soit nécessaire de passer par un encadrement en calculant $Op_{\min}(n)$ et $Op_{\max}(n)$. Par exemple, reprenez l'algorithme du tri-insertion et calculez $Op(n)$ en considérant comme opération significative la comparaison.

7. Classes de complexité

Nous noterons dorénavant $C(n)$ la complexité d'un algorithme exprimée en tant que temps d'exécution ou nombre d'opérations. C'est une fonction à valeurs dans $\mathbb{R}^+$.

$$\begin{array}{rcccc} C & : & \mathbb{N} & \to & \mathbb{R}^+ \\ & & n & \mapsto & C(n) \end{array}$$

Etudier la complexité d'un algorithme, revient à essayer d'exprimer cette complexité en la comparant à des fonctions connues telles que : n, $\log_2 n$, n^j, 2^n, ... Analyser un algorithme consiste donc à essayer de classifier la complexité de ce dernier. Nous supposons par la suite que toutes les fonctions mentionnées sont des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.

La classe Θ

Définition : Soit $g(n)$ une fonction, on note

$$\Theta(g(n)) = \{f(n) \mid \exists A, B \in \mathbb{R}^+\backslash\{0\},\ \exists n_0 \in \mathbb{N},\ \text{t.q.}\ \forall n \geqslant n_0, \quad 0 \leqslant Ag(n) \leqslant f(n) \leqslant Bg(n)\}$$

De façon peu formelle une fonction $f(n)$ appartient à l'ensemble $\Theta(g(n))$ s'il existe deux constantes A et B telles qu'à partir d'un certain rang $f(n)$ puisse être prise "en sandwich" entre $Ag(n)$ et $Bg(n)$. On dit aussi qu'à partir d'un certain rang, à un facteur multiplicatif près, on a "$f(n) = g(n)$". Par abus d'écriture on écrit $f(n) = \Theta(g(n))$ au lieu de $f(n) \in \Theta\ (g(n))$.

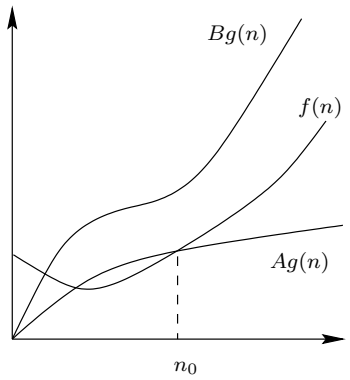


Fig. 1 $f(n) = \Theta(g(n))$

Exemple 1 : $n^2/2 - 3n = \Theta(n^2)$.
On cherche A, B et n_0 tels que

$$\begin{array}{rccccc} & An^2 & \leqslant & n^2/2 - 3n & \leqslant & Bn^2 \quad \forall n \geqslant n_0 \\ \Leftrightarrow & A & \leqslant & 1/2 - 3/n & \leqslant & B \quad \text{en supposant } n_0 > 0. \end{array}$$

Or $\forall n \geqslant 1$, $1/2 - 3/n < 1/2$. Donc en posant $B = 1/2$ et $n_0 = 1$, on obtient

$$\forall n \geqslant n_0 \quad n^2/2 - 3n \leqslant Bn^2$$

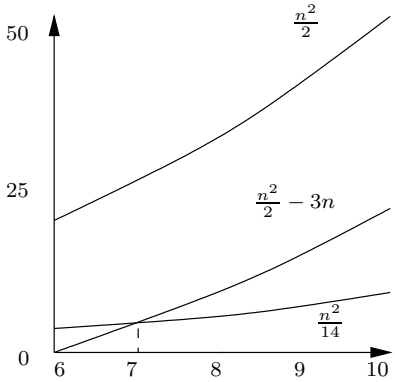
D'autre part, la fonction $1/2 - 3/n$ est croissante et strictement positive dès que $n > 6$. D'où $\forall n \geqslant 7$, $1/2 - 3/n \geqslant 1/2 - 3/7$. Ainsi en posant $n_0 = 7$ et $A = 1/14$, on obtient

$$\forall n \geqslant n_0 \quad An^2 \leqslant n^2/2 - 3n$$

D'où $\exists n_0 = 7$, $A = 1/14$, $B = 1/2$ tels que

$$\forall n \geqslant n_0, \quad An^2 \leqslant n^2/2 - 3/n \leqslant Bn^2$$

Ce qui montre que $n^2/2 - 3/n = \Theta(n^2)$.



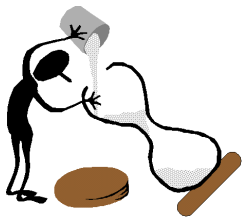


Fig. 2 $n^2/2 - 3n = \Theta(n^2)$

Exemple 2 : Pour l'algorithme *produit* on a $Op(n) = \Theta(n^2)$ (démontrez-le !).

Résultat Fondamental : *Soit* $p(n) = p_j n^j + p_{j-1} n^{j-1} + \cdots p_1 n + p_0$ *un polynôme quelconque définie sur* $\mathbb{N}$ $(p_j > 0)$ *et à valeurs dans* $\mathbb{R}^+$, *on a* $p(n) = \Theta(n^{deg\ p})$

La classe $\mathcal{O}$

Définition : Soit $g(n)$ une fonction, on note

$$\mathcal{O}(g(n)) = \{f(n) \ \mid \ \exists B \in \mathbb{R}^+ \setminus \{0\},\ \exists n_0 \in \mathbb{N},\ \text{t.q. } \forall n \geqslant n_0, \quad 0 \leqslant f(n) \leqslant Bg(n)\}$$

On dit que g "domine" f, et on note par abus d'écriture $f(n) = \mathcal{O}(g(n))$.

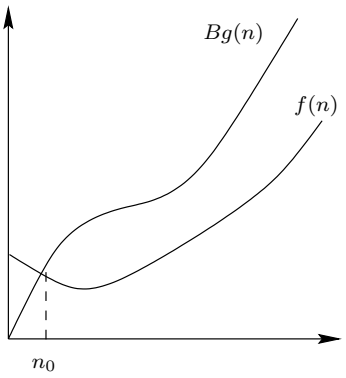


Fig. 3 $f(n) = \mathcal{O}(g(n))$

Classifier la complexité $C(n)$ d'un algorithme pour une instance quelconque à l'aide de la notation $\mathcal{O}$ permet de donner une borne supérieure générale sur le temps d'exécution (ou le nombre d'opérations) de l'algorithme quelle que soit l'instance traitée. Remarquons alors qu'il suffit de pouvoir classifier $C_{\max}(n)$ dans une classe $\mathcal{O}$ pour pouvoir affirmer que pour n'importe quelle instance d, $C_d(n)$ est dans cette même classe.

Exemple : Pour l'algorithme du *tri-insertion*, on a obtenu le résultat suivant :

$$\exists \alpha', \beta', \gamma', \quad T_{\max}(n) = \alpha' + \beta' n + \gamma' n^2$$

Or $\forall n \geqslant 1$, $\alpha' + \beta' n + \gamma' n^2 \leqslant (\alpha' + \beta' + \gamma') n^2$. D'où $\exists B = \alpha' + \beta' + \gamma'$, $\exists n_0 = 1$ tels que

$$\forall n \geqslant n_0, \quad 0 \leqslant T_{\max}(n) \leqslant Bn^2$$

Or, par définition, $\forall d \in D_n$, $0 \leqslant T_d(n) \leqslant T_{\max}(n)$, donc $\forall d \in D_n$, $T_d(n) = \mathcal{O}(n^2)$.

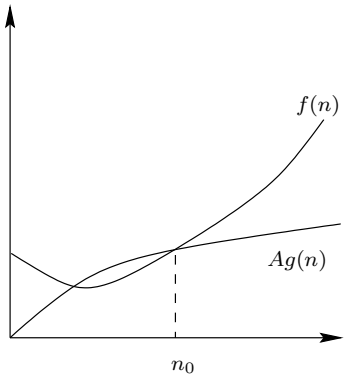
Remarque : On a aussi $T_{\max}(n) = \Theta(n^2)$ (montrez-le !).

La classe Ω

Définition : Soit $g(n)$ une fonction, on note

$$\Omega(g(n)) = \{f(n) \ \mid \ \exists A \in \mathbb{R}^+ \setminus \{0\}, \ \exists n_0 \in \mathbb{N}, \ \text{t.q. } \forall n \geqslant n_0, \quad 0 \leqslant Ag(n) \leqslant f(n)\}$$

Classifier la complexité $C(n)$ d'un algorithme pour une instance quelconque à l'aide de la notation Ω permet de donner une borne inférieure générale sur le temps d'exécution (ou le nombre d'opérations) de l'algorithme quelle que soit l'instance traitée. Remarquons alors qu'il suffit de pouvoir classifier $C_{\min}(n)$ dans une classe $\otimes$ pour pouvoir affirmer que pour n'importe quelle instance d, $C_d(n)$ est dans cette même classe.



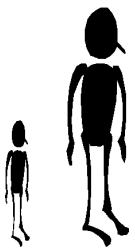


Fig. 4 $f(n) = \Omega(g(n))$

Exemple : Pour l'algorithme du *tri-insertion*, on a obtenu le résultat suivant :

$$\exists \alpha', \beta', \quad T_{\min}(n) = \alpha' + \beta' n$$

avec $\beta' > 0$. Or $\forall n \geqslant \lfloor -\alpha'/(\beta-1) \rfloor + 1$, on a $\alpha' + \beta' n \geqslant n$. D'où $\exists A = 1$, $\exists n_0 = \max(1, \lfloor -\alpha/(\beta-1) \rfloor + 1)$ tels que

$$\forall n \geqslant n_0, \quad 0 \leqslant An \leqslant T_{\min}(n)$$

Or, par définition, $\forall d \in D_n$, $0 \leqslant T_{\min}(n) \leqslant T_d(n)$, donc $\forall d \in D_n$, $T_d(n) = \otimes(n)$.

Remarque : On a aussi $T_{\min}(n) = \Theta(n)$ (montrez-le en utilisant par exemple le résultat fondamental).

Proposition : $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ et $f(n) = \Omega(g(n))$
Démonstration : à faire !

Les classes $\mathcal{O}$ et Ω donnent une borne supérieure et inférieure sur les valeurs de $f(n)$ lorsque n croit mais elles ne fournissent aucun renseignement sur la "qualité" de la borne proposée. La fonction $f(n)$ est-elle proche de la borne ? ou bien s'éloigne-t-elle de plus en plus de la borne lorsque n croit ?

Exemple 1 : $n^2/2 + n = \mathcal{O}(n^2)$ (montrez-le !)
Exemple 2 : $n = \mathcal{O}(n^2)$ (montrez-le !)

La classe o

La classe $\mathcal{O}$ fournit une borne supérieure sur les valeurs prises par une fonction f, cette borne peut être plus ou moins approchée de la fonction f. Pour distinguer le cas où la fonction f est proche asymptotiquement de sa borne supérieure du cas où cette dernière s'éloigne de f (i.e. f devient négligeable devant sa borne) on utilise la notation o.

Définition : Soit $g(n)$ une fonction, on note

$$o(g(n)) = \{f(n) \mid \forall B \in \mathbb{R}^+ \setminus \{0\}, \ \exists n_0 \in \mathbb{N}, \text{ t.q. } \forall n \geqslant n_0, \quad 0 \leqslant f(n) < Bg(n)\}$$

Une fonction $f(n)$ qui appartient à $o(g(n))$ (on écrira $f(n) = o(g(n))$) est une fonction qui n'approche pas asymptotiquement sa borne supérieure $g(n)$ en effet remarquez que

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n \to +\infty} f(n)/g(n) = 0$$

Exemple : on a $n = o(n^2)$, $n^2/2 \neq o(n^2)$, $n^2/2 = \mathcal{O}(n^2)$ (montrez-le !). Ci-dessous une interprétation "graphique" de cet exemple.

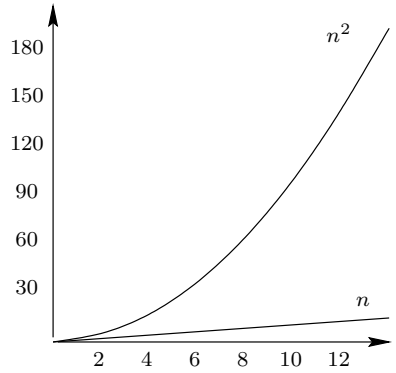


Fig. 5 $n = o(n^2)$

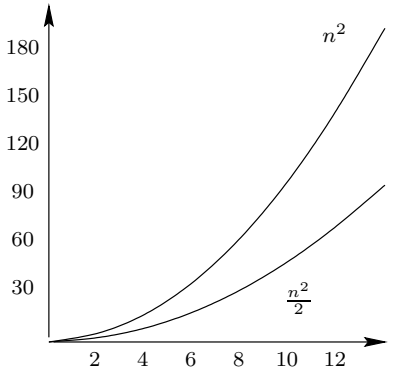


Fig. 6 $n^2/2 \neq o(n^2)$, $n^2/2 = \mathcal{O}(n^2)$

La classe ω

De même que pour les bornes supérieure, on distingue les fonctions qui s'approchent ou non de leur borne inférieure grace à la classe ω.

Définition : Soit $g(n)$ une fonction, on note

$$\omega(g(n)) = \{f(n) \mid \forall A \in \mathbb{R}^+ \setminus \{0\}, \ \exists n_0 \in \mathbb{N}, \text{ t.q. } \forall n \geqslant n_0, \quad 0 \leqslant Ag(n) < f(n)\}$$

Une fonction $f(n)$ qui appartient à $\omega(g(n))$ (on écrira $f(n) = \omega(g(n))$) est une fonction qui n'approche pas asymptotiquement sa borne inférieure $g(n)$ en effet remarquez que

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \to +\infty} f(n)/g(n) = +\infty$$

Exemple : on a $n^2/2 = \omega(n)$, $n^2/2 \neq \omega(n^2)$, $n^2/2 = \Omega(n^2)$ (montrez-le !).

8. Analyse complète du tri à bulles

Le tri à bulles s'effectue en $n-1$ étapes (pour un tableau de n éléments). A la $i^{\text{ème}}$ étape on balaye le tableau en partant de la fin jusqu'à la case i et à chaque fois que l'élément courant est plus petit que son prédécesseur, on échange leur position. En considérant que chaque élément du tableau correspond au poids d'une bulle, cette méthode a pour effet de faire remonter au fur et à mesure les bulles les plus légères vers la surface.

Exemple : $T =$

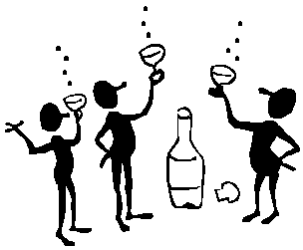
9	5	7	3	1

Itération 1 :

9	9	9	9	**1**
5	5	5	**1**	9
7	7	**1**	5	5
3	**1**	7	7	7
1	3	3	3	3

Itération 2 :

1	1	1	1
9	9	9	**3**
5	5	**3**	9
7	**3**	5	5
3	7	7	7



Itération 3 :

1	1	1
3	3	3
9	9	**5**
5	**5**	9
7	7	7

Itération 4 :

1	1
3	3
5	5
9	**7**
7	9

```
1  Algo tri_bulle
2  données
3     A : tableau de n entiers
4     i, j, n, aux : entiers
5     début
6        lire(A)
7        i ← 1
8        tant que i ⩽ n − 1 faire
9                 j ← n
10                tant que j ⩾ i + 1 faire
11                      si A[j] < A[j − 1] alors
12                                          aux ← A[j]
13                                          A[j] ← A[j − 1]
14                                          A[j − 1] ← aux
15                      finsi
16                      j ← j − 1
17                fintq
18                i ← i + 1
19       fintq
20    fin
```

Nous allons analyser la complexité $C(n)$ de cet algorithme en terme de nombre d'opérations effectuées par ce dernier. Pour cela on considère comme opération significative la comparaison. Il y a dans cet algoritme trois comparaisons : $i \leqslant n-1$, $j \geqslant i+1$, $A[j] < A[j-1]$.

$\underline{A[j] < A[j-1]}$: A i et j fixés la comparaison est effectuée une fois. A i fixé la comparaison est donc effectuée $\sum_{j=i+1}^{n} 1 = n - i$ fois. La variable i variant de 1 à $n-1$, la comparaison est effectuée $\sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = n(n-1)/2$ fois.

$\underline{j \geqslant i+1}$: A i fixé ce test de boucle est effectué une fois de plus que l'instruction $A[j] < A[j-1]$ soit $n - i + 1$ fois. La variable i variant de 1 à $n-1$, l'instruction est effectuée $\sum_{i=1}^{n-1} n - i + 1 = \sum_{i=2}^{n} i = n(n+1)/2 - 1$ fois.

$\underline{i \leqslant n-1}$: Ce test est effectué n fois.

On a donc $C(n) = n(n+1)/2 + n(n-1)/2 + n - 1 = 2n^2/2 + n - 1$. $C(n)$ est un polynôme en n de degré 2, on en conclut donc que $C(n) = \Theta(n^2)$.

Cependant, l'algorithme du tri à bulles tel qu'il est écrit n'est pas optimal. En effet, que se passe-t-il si à une étape i ($i < n-1$), le tableau est trié ? Il est facile de voir que pour toutes les étapes suivantes, l'algorithme parcourt les éléments du tableau sans réaliser un seul échange. On en déduit donc que lors d'une étape i, si aucun échange n'est effectué, le tableau est ordonné ; l'algorithme doit s'arrêter. On en déduit donc une nouvelle version du tri à bulles, dans laquelle on utilise un booléen `oncontinue` afin de savoir, si à une étape donnée, au moins un échange a eu lieu.

```
1  Algo tri_bulle version 2
2  données
3    A : tableau de n entiers
4    i, j, n, aux : entiers
5    oncontinue : booléen
6    début
7      lire(A)
8      i ← 1
9      oncontinue ← VRAI
10     tant que oncontinue faire
11               oncontinue ← FAUX
12               j ← n
13               tant que j ⩾ i + 1 faire
14                         si A[j] < A[j − 1] alors
15                                                oncontinue ← VRAI
16                                                aux ← A[j]
17                                                A[j] ← A[j − 1]
18                                                A[j − 1] ← aux
19                         finsi
20                         j ← j − 1
21               fintq
22               i ← i + 1
23     fintq
24   fin
```

L'initialisation de `oncontinue` à la constante booléenne `VRAI` nous permet d'exécuter au moins une fois la boucle principale. A chaque passage dans la boucle principale, `oncontinue` est initialisé à `FAUX` et sa valeur passe à `VRAI` si lors de l'étape en cours un échange est réalisé. Ainsi si aucun échange n'a lieu pour l'étape en cours, `oncontinue` conserve la valeur `FAUX` et la boucle principale se terminera.

L'analyse de la complexité de cette version du tri à bulles se fait en étudiant les cas favorables et défavorables (à cause du booléen !).

Exercice : Montrez que l'on a $C(n) = \mathcal{O}(n^2)$ et $C(n) = \Omega(n)$.

Un autre problème susbsiste sur cet algorithme. Comment prouve-t-on qu'il s'arrête, autrement dit comment démontrer qu'effectivement le booléen `oncontinue` garde à un certain moment la valeur `FAUX` ?

Exercice : Bien que cela soit moins évidemment que pour la première version de l'algorithme, la preuve d'arrêt se fait une fois encore en utilisant le variant $\{n + 1 - i \geqslant 0\}$.

9. Resultat général sur les tris comparatifs

Le tri par insertion et le tri à bulles font partie d'une même famille : les tris comparatifs. La méthode utilisée repose uniquement sur une comparaison entre deux éléments à chaque étape.

Théorème : *Tout tri comparatif effectue dans le pire des cas* $\Omega(n \log_2 n)$ *comparaisons pour trier une séquence de* n *éléments distincts.*

Nous ne donnerons ici qu'une ébauche de la preuve vue en cours :

1. Le déroulement d'un tri comparatif peut se représenter à l'aide d'un arbre de décision qui représente les différentes comparaisons effectuées. Chaque nœud de l'arbre représente une comparaison entre un élément a_i et un élement a_j du tableau A à trier. Si $a_i < a_j$ alors on part vers la gauche du nœud pour aboutir à un nouveau nœud sinon on part à droite. De chaque nœud il part au plus deux branches.

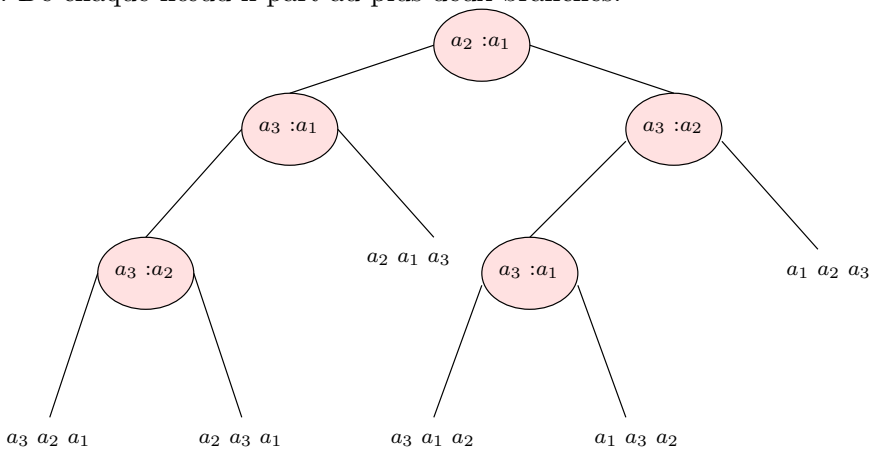


Fig. 7 Arbre du tri-insertion pour un tableau à 3 éléments

2. Trier les n éléments d'un tableau A revient à trouver la permutation π sur $\{1, \ldots n\}$ telle que les éléments $A[\pi(1)], \ldots A[\pi(n)]$ soient rangés dans l'ordre croissant. A chaque permutation π est associée une feuille de l'arbre de décision. **L'arbre de décision possède donc** $n!$ **feuilles**.
3. Un algorithme de tri part du haut de l'arbre et parcourt un chemin afin d'aboutir à la bonne feuille, ce chemin étant bien sur différent selon le tableau à trier. On évalue la longueur d'un chemin au nombre de nœuds rencontrés sur le chemin. Le chemin le plus long correspond au cas défavorable pour l'algorithme, i.e. celui où on effectue le plus de comparaisons. Le nombre de nœuds rencontrés dans ce cas là s'appelle la *hauteur* de l'arbre. Ainsi exhiber une borne inférieure sur la hauteur d'un arbre

de décision, revient à trouver une borne inférieure sur le nombre de comparaisons effectuées dans le pire des cas par un algorithme de tri comparatif.

4. **Proposition :** *Tout arbre binaire de hauteur h possède au plus 2^h feuilles.*
   Proposition : $n! \geqslant (n/e)^n$.
5. Soit h la hauteur de l'arbre de décision, on a donc

$$\begin{array}{rrl} & n! & \leqslant & 2^h \\ \Rightarrow & h & \geqslant & \log_2(n!) \\ \Rightarrow & h & \geqslant & \log_2(n/e)^n \end{array}$$

   Or $\log_2(n/e)^n = n\log_2 n - n\log_2 e = \Omega(n\log_2 n)$.