

- 1/ Rappels et compléments Java.
- 2/ Tableaux, boucles et invariants.
- 3/ Notions élémentaires de complexité. **ICI**
- 4/ Récursion.
- 5/ Structures de données et introduction aux types abstraits de données.
- 6/ Quelques compléments Java.

3/ Notions élémentaires de complexité.

Evaluer un algorithme

Quels sont les critères pour caractériser un **bon algorithme** , un **bon programme** ?

Evaluer un algorithme

Quels sont les critères pour caractériser un **bon algorithme**, un **bon programme** ?

Réponses intuitives

- L'algorithme est **correct** et est bien programmé (sans bugs, bien commenté, lisible, modulable, ...).
- Il peut s'exécuter **rapidement** ou en un temps **raisonnable** en utilisant **peu de ressource en mémoire** même si on augmente la taille des données à traiter : ce sont des thèmes centraux en **théorie de la complexité** des algorithmes.

Evaluer un algorithme

Quels sont les critères pour caractériser un **bon algorithme**, un **bon programme** ?

Réponses intuitives

- L'algorithme est **correct** et est bien programmé (sans bugs, bien commenté, lisible, modulable, ...).
- Il peut s'exécuter **rapidement** ou en un temps **raisonnable** en utilisant **peu de ressource en mémoire** même si on augmente la taille des données à traiter : ce sont des thèmes centraux en **théorie de la complexité** des algorithmes.

Deux critères en complexité

En fonction de la taille des données en entrée, on a deux questions fondamentales.

1/ Temps d'exécution. Quel est le temps d'exécution de l'algorithme?

2/ Place mémoire. Quelle est la place mémoire nécessaire pour son exécution?

Un programme de recherche d'un élément

```
public static int recherche(int[] tab, int x) {  
    /** méthode pour rechercher l'indice d'un élément  
    * ENTRÉES : tab est un tableau d'entiers distincts,  
    *           x est un entier  
    * SORTIE : renvoie i si  $\text{tab}[i] == x$ ,  
    *           -1 si  $x$  n'est pas dans le tableau  
    */  
    int i=0;  
    while(i<tab.length && x!=tab[i])  
        /* A1 :  $\forall 0 \leq j \leq i, \text{tab}[j] \neq x$  */  
        i++;  
    if (i==tab.length)  
        /* A2 :  $x$  n'est pas dans le tableau. */  
        return -1;  
    else return i; /* A3 :  $x$  est à l'indice  $i$  */  
}
```

- A1 est vrai en rentrant dans la boucle ($i = j = 0$).

L'algorithme de l'exemple est correct

- A1 est vrai en rentrant dans la boucle ($i = j = 0$).
- A1 est vrai à chaque passage dans la boucle car on rentre quand $\text{tab}[i] \neq x$ et i a été incrémenté.

L'algorithme de l'exemple est correct

- **A1 est vrai** en rentrant dans la boucle ($i = j = 0$).
- **A1 est vrai** à chaque passage dans la boucle car on rentre quand $\text{tab}[i] \neq x$ et i a été incrémenté.
- **A2 est vrai** puisque A1 est vrai à chaque étape de la boucle si $i == \text{tab.length}$ alors $\neg \exists j \in [0, \text{tab.length} - 1] : \text{tab}[j] = x$, donc x n'est pas dans le tableau.

L'algorithme de l'exemple est correct

- **A1 est vrai** en rentrant dans la boucle ($i = j = 0$).
- **A1 est vrai** à chaque passage dans la boucle car on rentre quand $\text{tab}[i] \neq x$ et i a été incrémenté.
- **A2 est vrai** puisque A1 est vrai à chaque étape de la boucle si $i == \text{tab.length}$ alors $\neg j < \text{tab.length} \vee \text{tab}[j] \neq x$, donc x n'est pas dans le tableau.
- **A3 est vrai** car si $i < \text{tab.length}$ alors on est sorti de la boucle avec $x == \text{tab}[i]$ et donc le x recherché est à l'indice i .

L'algorithme de l'exemple est correct

- **A1 est vrai** en rentrant dans la boucle ($i = j = 0$).
- **A1 est vrai** à chaque passage dans la boucle car on rentre quand $\text{tab}[i] \neq x$ et i a été incrémenté.
- **A2 est vrai** puisque A1 est vrai à chaque étape de la boucle si $i == \text{tab.length}$ alors $\neg j < \text{tab.length} \wedge \text{tab}[j] \neq x$, donc x n'est pas dans le tableau.
- **A3 est vrai** car si $i < \text{tab.length}$ alors on est sorti de la boucle avec $x == \text{tab}[i]$ et donc le x recherché est à l'indice i .
- ET le programme est **bien commenté**.

L'algorithme termine!

Au pire i croît de 0 à `tab.length` qui est une **valeur finie**.

L'algorithme termine!

Au pire i croît de 0 à `tab.length` qui est une **valeur finie**.

Temps d'exécution : cas pire, analyse en moyenne et meilleur des cas

- Dans le pire des cas, on doit parcourir tout `tab`, n itérations.
- Dans le meilleur des cas, on trouve x de suite, 1 itération.
- Si p est la probabilité que x se trouve dans le tableau, le nombre moyen d'itérations sera (sous l'hypothèse que tous les nombres sont équiprobables)

$$n(1 - \frac{p}{2}) + \frac{p}{2}.$$

L'algorithme termine!

Au pire i croît de 0 à `tab.length` qui est une **valeur finie**.

Temps d'exécution : cas pire, analyse en moyenne et meilleur des cas

- Dans le pire des cas, on doit parcourir tout `tab`, n itérations.
- Dans le meilleur des cas, on trouve x de suite, 1 itération.
- Si p est la probabilité que x se trouve dans le tableau, le nombre moyen d'itérations sera (sous l'hypothèse que tous les nombres sont équiprobables)

$$n(1 - \frac{p}{2}) + \frac{p}{2}.$$

En effet, ce nombre moyen est donné par $\underbrace{(1-p)n}_{x \in \text{tableau}} + p \underbrace{\sum_{i=1}^n \frac{i}{n}}_{x \in \text{tableau}}.$

Mémoire

La place nécessaire pour stocker x , `tab` et l'indice i : $n + 2$ entiers.

Un second exemple

Dans le premier programme, on a pris comme hypothèse des *entiers distincts*. Dans ce second exemple, on suppose de plus que le tableau `tab` donné en entrée est **déjà trié!** Il s'agit toujours de savoir si l'élément `x` est dans `tab`.

Un second exemple

Dans le premier programme, on a pris comme hypothèse des *entiers distincts*. Dans ce second exemple, on suppose de plus que le tableau `tab` donné en entrée est **déjà trié**! Il s'agit toujours de savoir si l'élément `x` est dans `tab`.

Solution force brute

On utilise l'algorithme précédent et on trouve soit l'indice de `x` soit `-1` si `x` n'est pas dans `tab`.

Un second exemple

Dans le premier programme, on a pris comme hypothèse des *entiers distincts*. Dans ce second exemple, on suppose de plus que le tableau **tab** donné en entrée est **déjà trié!** Il s'agit toujours de savoir si l'élément **x** est dans **tab**.

Solution force brute

On utilise l'algorithme précédent et on trouve soit l'indice de x soit -1 si x n'est pas dans **tab**.

Solution dichotomique

On profite du fait que les éléments sont **ordonnés** pour appliquer le paradigme "DIVISER POUR RÉGNER". Le principe est le suivant:

- on compare x à l'élément m qui est celui du milieu du (sous-)tableau considéré.
- Si $x = m$ alors renvoyer l'indice de m .
- Si $x < m$ alors on va travailler dans la partie gauche du tableau (la partie $< m$).
- Si $x > m$ alors *idem* mais avec la partie droite.

Un programme de recherche dichotomique

```
/** ENTRÉES : tab est un tableau d'entiers
 * distincts ordonnés par ordre croissant, x est un entier
 * SORTIE : renvoie i si  $\text{tab}[i] == x$ ,
 *          -1 si  $x$  n'est pas dans le tableau */
public int rechercheDichotomique(int[] tab, int x) {
    int gauche = 0;
    int droite = tab.length - 1;
    int milieu;
    while (gauche <= droite) {
        /* A1 :  $\forall j < \text{gauche} \text{ tab}[j] \neq x \ \forall j > \text{droite} \text{ tab}[j] \neq x$  */
        milieu = (gauche + droite) / 2 ;
        if (x==tab[milieu])
            /* A2 : x est à l'indice milieu */
            return milieu;
        if (x<tab[milieu]) droite = milieu - 1;
        else gauche = milieu + 1;
    }
    /*
```

L'algorithme termine!

Pour tout n , il existe k tel que $2^k \leq n < 2^{k+1}$. Intuitivement, dans le pire des cas on va toujours diviser la taille du (sous)tableau à traiter en 2 jusqu'à ce que la condition de la boucle `gauche <= droite` ne soit plus satisfaite.

Temps d'exécution et place mémoire pour le second exemple

L'algorithme termine!

Pour tout n , il existe k tel que $2^k \leq n < 2^{k+1}$. Intuitivement, dans le pire des cas on va toujours diviser la taille du (sous)tableau à traiter en 2 jusqu'à ce que la condition de la boucle **gauche** \leq **droite** ne soit plus satisfaite.

Temps d'exécution : cas pire, analyse en moyenne et meilleur des cas

- Dans le pire des cas, on doit toujours traiter tous les sous-tableaux. Soit k tel que $2^k \leq n < 2^{k+1}$ on va traiter en gros au plus $k + 1 = \lfloor \log_2 n \rfloor + 1$ sous-tableaux! On parle de complexité **logarithmique!**
- Dans le meilleur des cas, on trouve x de suite, 1 itération.
- Si p est la probabilité que x se trouve dans le tableau, le nombre moyen d'itérations sera (sous l'hypothèse que tous les nombres sont équiprobables) aussi **logarithmique** : $(\lfloor \log_2 n \rfloor + 1)(1 - p) + \underbrace{\dots}_{\text{peu importe}}$

Temps d'exécution et place mémoire pour le second exemple

L'algorithme termine!

Pour tout n , il existe k tel que $2^k \leq n < 2^{k+1}$. Intuitivement, dans le pire des cas on va toujours diviser la taille du (sous)tableau à traiter en 2 jusqu'à ce que la condition de la boucle **gauche** \leq **droite** ne soit plus satisfaite.

Temps d'exécution : cas pire, analyse en moyenne et meilleur des cas

- Dans le pire des cas, on doit toujours traiter tous les sous-tableaux. Soit k tel que $2^k \leq n < 2^{k+1}$ on va traiter en gros au plus $k + 1 = \lfloor \log_2 n \rfloor + 1$ sous-tableaux! On parle de complexité **logarithmique!**
- Dans le meilleur des cas, on trouve x de suite, **1** itération.
- Si p est la probabilité que x se trouve dans le tableau, **le nombre moyen d'itérations** sera (sous l'hypothèse que tous les nombres sont équiprobables) aussi **logarithmique** : $(\lfloor \log_2 n \rfloor + 1)(1 - p) + \underbrace{\dots}_{\text{peu importe}}$

Mémoire : $n + 4$ entiers pour tab, x, milieu, gauche et droite

Attention!!!

En algorithmique, on s'intéresse aux ordres de grandeur du temps d'exécution et/ou de l'espace mémoire nécessaire à un algorithme en fonction de la taille des données en entrée n . On s'intéresse aux grandes valeurs de n ($n \rightarrow \infty$).

Attention!!!

En algorithmique, on s'intéresse aux ordres de grandeur du temps d'exécution et/ou de l'espace mémoire nécessaire à un algorithme en fonction de la taille des données en entrée n . On s'intéresse aux grandes valeurs de n ($n \rightarrow \infty$).

Soit f et g deux fonctions.

- **Grand O.** On note $f(n) = O(g(n))$ si il existe une constante $c > 0$ et un entier n_0 tel que

$$\forall n \geq n_0, |f(n)| \leq cg(n).$$

On dit alors que g **domine** f à partir d'un certain rang (**à une constante multiplicative près!!!**). Par exemple, si $g(n) = n$ et $f_1(n) = \sqrt{n}$ et $f_2(n) = 2n$ alors $f_1(n) = O(g(n))$ et $f_2(n) = O(g(n))$.

Attention!!!

En algorithmique, on s'intéresse aux ordres de grandeur du temps d'exécution et/ou de l'espace mémoire nécessaire à un algorithme en fonction de la taille des données en entrée n . On s'intéresse aux grandes valeurs de n ($n \rightarrow \infty$).

Soit f et g deux fonctions.

- **Grand O.** On note $f(n) = O(g(n))$ si il existe une constante $c > 0$ et un entier n_0 tel que

$$\forall n \geq n_0, |f(n)| \leq cg(n).$$

On dit alors que g **domine** f à partir d'un certain rang (**à une constante multiplicative près!!!**). Par exemple, si $g(n) = n$ et $f_1(n) = \sqrt{n}$ et $f_2(n) = 2n$ alors $f_1(n) = O(g(n))$ et $f_2(n) = O(g(n))$.

- **Theta.** On note $f(n) = \Theta(g(n))$ s'il existe deux constantes $c_1 > 0$ et $c_2 > 0$ telles que

$$c_1 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c_2$$

On dit alors que f et g sont du **même ordre** de grandeur.

Attention!!!

En algorithmique, on s'intéresse aux ordres de grandeur du temps d'exécution et/ou de l'espace mémoire nécessaire à un algorithme en fonction de la taille des données en entrée n . On s'intéresse aux grandes valeurs de n ($n \rightarrow \infty$).

Soit f et g deux fonctions.

- **Grand O.** On note $f(n) = O(g(n))$ si il existe une constante $c > 0$ et un entier n_0 tel que

$$\forall n \geq n_0, |f(n)| \leq cg(n).$$

On dit alors que g **domine** f à partir d'un certain rang (**à une constante multiplicative près!!!**). Par exemple, si $g(n) = n$ et $f_1(n) = \sqrt{n}$ et $f_2(n) = 2n$ alors $f_1(n) = O(g(n))$ et $f_2(n) = O(g(n))$.

- **Theta.** On note $f(n) = \Theta(g(n))$ s'il existe deux constantes $c_1 > 0$ et $c_2 > 0$ telles que

$$c_1 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c_2$$

On dit alors que f et g sont du **même ordre** de grandeur.

- **Omega.** On note $f(n) = \Omega(g(n))$ si $f(n)$ est minorée par $cg(n)$ avec $c > 0$ à partir d'un certain rang.

- Le Grand O est transitif : si $h(n) = O(g(n))$ et $g(n) = O(f(n))$ alors $h(n) = O(f(n))$.

- Le Grand O est transitif : si $h(n) = O(g(n))$ et $g(n) = O(f(n))$ alors $h(n) = O(f(n))$.
- La règle des sommes : si $g_1(n) = O(f_1(n))$ et $g_2(n) = O(f_2(n))$ alors $g_1(n) + g_2(n) = O(\max(f_1(n), f_2(n)))$.

- Le Grand O est transitif : si $h(n) = O(g(n))$ et $g(n) = O(f(n))$ alors $h(n) = O(f(n))$.
- La règle des sommes : si $g_1(n) = O(f_1(n))$ et $g_2(n) = O(f_2(n))$ alors $g_1(n) + g_2(n) = O(\max(f_1(n), f_2(n)))$.
- La règle des produits : si $g_1(n) = O(f_1(n))$ et $g_2(n) = O(f_2(n))$ alors $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.

- Le Grand O est transitif : si $h(n) = O(g(n))$ et $g(n) = O(f(n))$ alors $h(n) = O(f(n))$.
- La règle des sommes : si $g_1(n) = O(f_1(n))$ et $g_2(n) = O(f_2(n))$ alors $g_1(n) + g_2(n) = O(\max(f_1(n), f_2(n)))$.
- La règle des produits : si $g_1(n) = O(f_1(n))$ et $g_2(n) = O(f_2(n))$ alors $g_1(n)g_2(n) = O(f_1(n)f_2(n))$.
- Supposons que la limite $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$ existe avec $L \in \mathbb{R} \cup \{ \pm \infty \}$ alors
 - 1 si $L = 0$ alors $f(n) = o(g(n))$
 - 2 si $0 < L < \infty$ alors $f(n) = \Theta(g(n))$
 - 3 si $L = \pm \infty$ alors $f(n) = \Omega(g(n))$.

- $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$ et $g(n) = O(f(n))$

- $f(n) = \Theta(g(n))$, $f(n) = O(g(n))$ et $g(n) = O(f(n))$
- $f(n) = O(g(n))$ n'implique pas $f(n) = \Theta(g(n))$. Ex: $f(n) = \log n$ et $g(n) = n$.
- Les indications en "Theta" sont plus précises que les "Grand O".

Quand on écrit que le temps d'exécution $T(n) = \Omega(g(n))$ d'un algorithme avec des entrées de taille n , au delà du fait mathématique que " $T(n)$ est minorée à un facteur près par $g(n)$ ", en algorithmique cela a un sens fort.

Quand on écrit que le temps d'exécution $T(n) = \Omega(g(n))$ d'un algorithme avec des entrées de taille n , au delà du fait mathématique que " $T(n)$ est minorée à un facteur près par $g(n)$ ", en algorithmique cela a un sens fort.

Remarque

$g(n)$ est une borne inférieure (à un facteur constant près) de $T(n)$. Ce qui implique qu'il existe des instances sur lesquelles l'algorithme nécessitera un temps d'exécution d'au moins $g(n)$.

Quand on écrit que le temps d'exécution $T(n) = \Omega(g(n))$ d'un algorithme avec des entrées de taille n , au delà du fait mathématique que " $T(n)$ est minorée à un facteur près par $g(n)$ ", en algorithmique cela a un sens fort.

Remarque

$g(n)$ est une borne inférieure (à un facteur constant près) de $T(n)$. Ce qui implique qu'il existe des instances sur lesquelles l'algorithme nécessitera un temps d'exécution d'au moins $g(n)$.

Exemple

Le temps d'exécution d'une multiplication de deux matrices carrées d'ordre n nécessite $\Omega(n^2)$ opérations (au moins on doit calculer les n^2 termes de la nouvelle matrice)!

Que ce soit en **temps d'exécution** $T(n)$ et/ou en **espace mémoire** $M(n)$, on parle de complexité

- **logarithmique** si $T(n) = \Theta(\log n)$ (resp. pour la mémoire $M(n) = \Theta(\log n)$)
- **linéaire** si $T(n) = \Theta(n)$ (resp. pour la mémoire $M(n) = \Theta(n)$)
- **quadratique** si $T(n) = \Theta(n^2)$ (resp. pour la mémoire $M(n) = \Theta(n^2)$)
- **traitable** ou **polynomial** si $T(n) = O(n^\alpha)$ pour $\alpha > 0$ constante (idem pour $M(n)$)
- **intraitable** si $T(n) = \Theta(n^{\omega(n)})$ où $\lim_{n \rightarrow \infty} \omega(n) = +\infty$.

Important

Il faut toujours chercher à exprimer toute notation avec les **Grand-Oh**, **Omega** et **Theta** à l'aide de fonctions de référence (compréhensible par le plus grand nombre). Ce sont :

- les fonctions constantes
- les logarithmes
- les puissances (éviter les polynômes)
- les exponentiations

Algorithme linéaire

Si un algorithme linéaire s'exécutant en $\Theta(n)$ traite 10 données par minute, il peut en traiter 14 400 par jour, 5 260 000 par an ...

Algorithme linéaire

Si un algorithme linéaire s'exécutant en $\Theta(n)$ traite 10 données par minute, il peut en traiter 14 400 par jour, 5 260 000 par an ...

Intraitable

Pour un algorithme s'exécutant en $\Theta(2^n)$, il va aussi traiter 10 données en quelques minutes mais déjà il est limité à quelques 20 données par jour et à une trentaine par siècle !

Quelques ordres de complexité

Désignations usuelles	Vitesse réelle	Ordre de croissance	Formulation	Exemples
Factorielle	Trop lent	N^N	$N!$	Problèmes extrême (cf. L3/M1)
Exponentielle	Lent	constante ^N	$(1 + \varepsilon)^N$	Ecriture exhaustive de tous les mots de longueur N avec les lettres $\{a, b, c\}$
Polynomiale	Parfois acceptable	Dominé par $N^{\text{constante}}$		Multiplication de matrices
Quasi-linéaire	Suffisamment rapide	Entre linéaire et polynomiale	$N \log N$ $N \log \log N \dots$	Certains tris
Linéaire	Rapide	Proportionnel à N	N	Parcours tableau
Logarithmique	Très rapide	Proportionnel à $\log N$	$\log N$	Recherche dichotomique
Constant	Le plus rapide	Indépendant de N	constante	Accès à un élément d'un tableau

Un premier cas d'étude : LA MULTIPLICATION DE MATRICES

Un second cas d'étude : LE TRI FUSION (mergesort)

Nous avons en entrée un tableau T à trier. L'idée est d'appliquer le paradigme **diviser pour régner** récursivement :

Le tri fusion

Conditions d'arrêt. Si la taille du tableau T est 1 alors il est trié.

Corps de la récursion. Sinon on le **partitionne** en **DEUX** (sous)tableaux T_1 et T_2 qu'on **trie** (**appels récursifs**) T_1 puis T_2 et on **fusionne** les deux tableaux triés obtenus.

Le tri-fusion a besoin d'un algorithme de fusion de deux tableaux T_1 et T_2 triés et dont le résultat (T_3) sera la fusion triée de ces deux tableaux.

Le tri-fusion a besoin d'un algorithme de fusion de deux tableaux T_1 et T_2 triés et dont le résultat (T_3) sera la fusion triée de ces deux tableaux.

Il suffit pour cela d'utiliser deux indices i et j indiquant respectivement les éléments de T_1 et de T_2 que l'on compare. L'élément le plus petit est copié dans T_3 et l'indice du tableau correspondant est incrémenté.

On utilise plusieurs faits :

- L'algorithme de **fusion** de deux tableaux T_1 et T_2 **triés** (de taille resp. n_1 et n_2) nécessite $n = n_1 + n_2$ itérations (quelque soit le cas).

On utilise plusieurs faits :

- L'algorithme de **fusion** de deux tableaux T_1 et T_2 **triés** (de taille resp. n_1 et n_2) nécessite $n = n_1 + n_2$ itérations (quelque soit le cas).
- Soit $T(n)$ le temps nécessaire pour l'exécution d'un tri fusion sur un tableau de taille n . Pour simplifier, on va supposer que n est une puissance de 2. On a

$$T(n) = n + 2T\left(\frac{n}{2}\right)$$

On utilise plusieurs faits :

- L'algorithme de **fusion** de deux tableaux T_1 et T_2 **triés** (de taille resp. n_1 et n_2) nécessite $n = n_1 + n_2$ itérations (quelque soit le cas).
- Soit $T(n)$ le temps nécessaire pour l'exécution d'un tri fusion sur un tableau de taille n . Pour simplifier, on va supposer que n est une puissance de 2. On a

$$T(n) = n + 2T\left(\frac{n}{2}\right)$$

- Donc

$$\frac{T(n)}{n} = 1 + 2\frac{T\left(\frac{n}{2}\right)}{n}$$

On utilise plusieurs faits :

- L'algorithme de **fusion** de deux tableaux T_1 et T_2 **triés** (de taille resp. n_1 et n_2) nécessite $n = n_1 + n_2$ itérations (quelque soit le cas).
- Soit $T(n)$ le temps nécessaire pour l'exécution d'un tri fusion sur un tableau de taille n . Pour simplifier, on va supposer que n est une puissance de 2. On a

$$T(n) = n + 2T\left(\frac{n}{2}\right)$$

- Donc

$$\frac{T(n)}{n} = 1 + 2\frac{T\left(\frac{n}{2}\right)}{n}$$

- Posons $u_n = \frac{T(n)}{n}$, on a donc $u_n = 1 + u_{n/2} = 2 + u_{n/4} = \dots$ et au final $T(n) = O(n \log n)$.

Robustesse

La notion de complexité en temps ou en espace est **robuste** :

- indépendante de l'ordinateur sur lequel l'algorithme est exécuté
- indépendante du compilateur ou du langage de programmation
- est exprimée en fonction de la **taille** de la donnée à traiter.

Robustesse

La notion de complexité en temps ou en espace est **robuste** :

- indépendante de l'ordinateur sur lequel l'algorithme est exécuté
- indépendante du compilateur ou du langage de programmation
- est exprimée en fonction de la **taille** de la donnée à traiter.

Unité de mesure

On mesure le nombre d' **opérations élémentaires** qui prennent en compte...