

Séance 3: Algorithmique

Philipp Ahrendt

December 30, 2025

Contents

1 Généralités d’algorithmique et de complexité	1
1.1 Complexité	2
1.2 Paradigmes algorithmiques	4
2 Récursion	5
3 Diviser pour régner	8
4 Programmation dynamique	10

1 Généralités d’algorithmique et de complexité

En algorithmique, on a deux objectifs principaux:

1. Trouver des procédures *efficaces* pour résoudre des problèmes
2. *Analyser* l’efficacité d’une solution

Pour faire cela, il y a deux questions importantes à se poser:

1. Comment peut-on *mesurer* l’efficacité d’un algorithme?
 - *Théorie de la complexité*
2. De quelles méthodes dispose-t-on pour trouver, décrire des algorithmes?
 - *Paradigmes algorithmiques*

1.1 Complexité

Mesure les ressources requises par un algorithme pour fournir son résultat.

En général, on considère deux types de complexité:

Complexité temporelle	durée du calcul/nombre d'opérations à effectuer
Complexité spatiale	espace (en kB) requis

Nous allons nous concentrer principalement sur la **complexité temporelle**.

On s'intéresse au nombre "d'opérations de base" effectuées par l'algorithme

- Affectations
- Comparaisons
- Opérations arithmétiques
- etc. . .

On veut estimer comment le nombre d'opérations évolue en fonction de la taille de la donnée d'entrée. C'est-à-dire on cherche la performance *asymptotique* de l'algorithme.

Exemple: Dans un algorithme de tri, combien de comparaisons entre éléments sont nécessaires en fonction de la longueur de la liste?

En général, le nombre d'opérations à faire dépend de l'entrée (par exemple, la liste d'entrée peut déjà être triée, presque triée, triée à l'envers etc).

Le plus souvent, les grandeurs intéressantes sont:

- La performance **worst case**
- La performance **en moyenne**

Les performances "worst case" sont généralement plus simples à analyser. On va se limiter à cela dans ce cours.

Exemple: Analysons le **tri par insertion** (voir feuille d'exercices 2).

```
def insertion(L):  
    for i in range(len(L)):  
        a = i  
        tmp = L[i]  
        while a > 0 and L[a-1] > tmp:
```

```

L[a] = L[a-1]
L[a-1] = tmp
a = a - 1

```

- *Quel est le “worst case” pour cet algorithme?*

A l'étape i , on “propage en arrière” l'élément i jusqu'à sa place parmi les éléments précédents. Au pire, on est obligé de parcourir tous les indices avant i , ce qui arrive quand $L[i]$ est plus petit que tous les éléments avant. On peut donc voir que le pire des cas est de trier une liste qui est triée à l'envers au départ.

- *Asymptotiquement, combien de comparaisons entre éléments faut-il faire au pire des cas?*

On veut estimer le nombre d'opérations en fonction de la longueur n de la liste à trier. On voit que l'algorithme a deux boucles “imbriquées”. La boucle extérieure fait n itérations. A l'intérieur de cette boucle, il y a quelques opérations simples, et une boucle **while**. On vient de voir que, à l'itération i , la boucle intérieure peut faire jusqu'à i itérations, et faire un nombre constant d'opérations à chaque fois. On peut négliger le comptage exact des opérations et se concentrer sur le fait que le nombre d'opérations augmente linéairement avec chaque itération. Cela suffit pour voir que le nombre d'itérations est $\simeq \sum_{0 \leq i \leq n} i$. On pourrait aussi donner un argument mathématiquement “propre”, du style

- K_1 = opérations avant la boucle **for**
- K_2 = opérations avant la boucle **while**
- K_3 = opérations dans la boucle **while**

$C(n) = K_1 + \sum_{0 \leq i \leq n} (K_2 + \sum_{0 \leq j \leq i} K_3) = K_3 \sum_{0 \leq i \leq n} i + K_2 n + K_1 \sim \sum_{0 \leq i \leq n} i$ (les autres termes vont être petits en comparaison quand $n \rightarrow \infty$)

Pour estimer $\sum_{0 \leq i \leq n} i$, on peut soit se rappeler que cette somme vaut $n(n-1)/2 \simeq n^2/2$, soit utiliser des arguments encore plus élémentaires, comme,

$$\left(\frac{n}{2}\right)^2 = \sum_{n/2 \leq i \leq n} i \leq \sum_{n/2 \leq i \leq n} i \leq \sum_{0 \leq i \leq n} i \leq \sum_{i \leq 0 \leq n} i = n^2$$

En tout cas, la complexité estimée est de l'ordre de n^2 .

Exemple: Recherche “naïve” vs. recherche dichotomique

- *Lequel des deux algorithmes est plus efficace?*

Sans faire une analyse formelle, on peut voir que la recherche naïve dans une liste triée parcourt, au pire des cas, toute la liste, et la moitié de la liste en moyenne. En revanche, dans la recherche dichotomique, chaque étape réduit à moitié la partie de la liste dans laquelle on cherche. Cela veut dire qu'une grande partie de la liste (bien plus que la moitié) n'est jamais regardée. Cela indique déjà clairement que la recherche dichotomique est plus efficace. L'analyse des complexités dans la question suivante montre à quel point elle est plus efficace.

- *Quelles sont les complexités asymptotiques?*

Pour la recherche naïve c'est simple: on parcourt un par un les éléments de la liste jusqu'à ce qu'on trouve le bon élément. Au pire on regarde chacun des n éléments. La complexité est de l'ordre de n .

Pour la recherche dichotomique, chaque étape réduit à moitié la partie considérée de la liste. Au pire cela continue jusqu'à ce qu'il reste un seul élément dans cette partie. Si on suppose que la liste contient $n = 2^k$ éléments, la première étape réduit le champ de recherche à $n/2 = 2^{k-1}$ éléments, la deuxième étape réduit cela à $n/4 = 2^{k-2}$ et ainsi de suite. On peut répéter cela au plus k fois. Donc pour $n = 2^k$, la complexité est de l'ordre de k . Pour n quelconque, on peut déjà deviner que la complexité est de l'ordre de $\log(n)$. L'argument mathématique est simple: on peut trouver un k tel que

$$2^k \leq n \leq 2^{k+1} \Rightarrow k \leq \log(n) \leq k+1$$

On sait que la complexité pour n est entre k et $k+1$, et les inégalités ci-dessus montrent que k est la partie entière de $\log(n)$. Donc asymptotiquement la complexité est $\log(n)$.

1.2 Paradigmes algorithmiques

Un *paradigme algorithmique* est une méthodologie générique pour construire des algorithmes.

Un paradigme peut être appliquée à un grand nombre de problèmes, et fournir des algorithmes qui partagent des principes clés dans leur structure.

Lorsqu'on essaye de résoudre un problème algorithmique, il est souvent utile de raisonner en termes de paradigmes. Cela aide à *structurer* le problème, et trouver une solution efficace.

Un même problème peut être résolu en utilisant des paradigmes différents, mais souvent il y en a un qui est plus efficace que l'autre.

Dans ce cours, on va voir deux paradigmes importants:

- **Diviser pour régner**
- **Programmation dynamique**

2 Récursion

Le principe de **récursion** est un outil important pour analyser des problèmes algorithmiques. Il consiste à décrire la solution du problème dans un cas complexe en termes de ses solutions dans des cas simples.

Cas typique: un problème sur les entiers naturels peut se formuler sous forme d'une **réurrence** mathématique.

Exemple: calcul de factoriel

Le factoriel $n!$ d'un entier n est le produit $1 \times 2 \times 3 \dots \times (n-1) \times n$. On peut écrire la suite des $n!$ comme une récurrence:

- $0! = 1$
- Pour $n \geq 1$, $n! = n \times (n-1)!$

On peut interpréter cela comme une instruction pour calculer $n!$ de n'importe quel n . Le cas "simple" est $0!$, pour lequel on a directement la réponse. Ensuite, calculer $n!$ pour un n plus grand revient à multiplier par n le résultat (trouvé auparavant) pour $n-1$. Au final, tout se ramène au cas $n = 0$. On peut implémenter ce raisonnement dans un code python:

```
def factoriel(n):  
    if n == 0:  
        return 1 # Cas de base  
    else:  
        return n * factoriel(n-1) # Appel récursif
```

Exemple: Nombres de fibonacci

Les nombres de fibonacci sont les nombres de la suite

- $f_0 = 0$
- $f_1 = 1$
- Pour $n \geq 2$, $f_n = f_{n-1} + f_{n-2}$

Voici les 10 premiers nombres de fibonacci: 0,1,1,2,3,5,8,13,21,34

Avec la définition ci-dessus, on a un algorithme récursif naturel pour calculer la suite de fibonacci

```
def fibonacci(n):
    if n == 0:
        return 0

    elif n == 1:
        return 1

    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

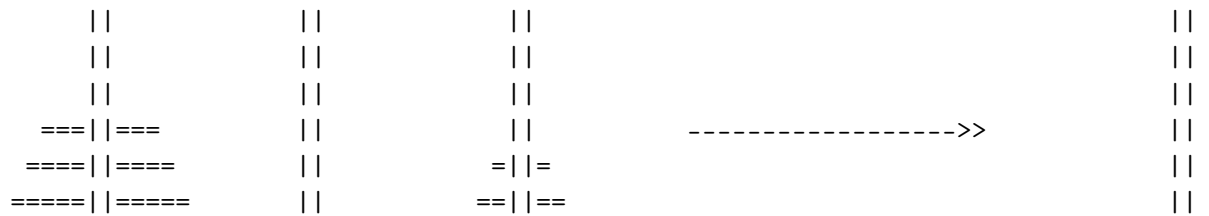
Exemple: Les tours de Hanoi

Dans le jeu des *tours de Hanoi*, on a 3 piliers, notés A,B et C, et n disques troués de différentes tailles, que l'on peut empiler sur les piliers. Au départ, tous les disques sont empilés, du plus grand au plus petit, sur l'un des piliers. Le but du jeu est de déplacer les disques, un par un, jusqu'à ce qu'ils soient empilé sur un autre pilier, du plus grand au plus petit. Cela en suivant les règles suivantes.

- On ne peut déplacer une pile que lorsqu'elle se trouve en haut d'une pile
- On ne peut pas placer un disque au-dessus d'un disque plus petit.

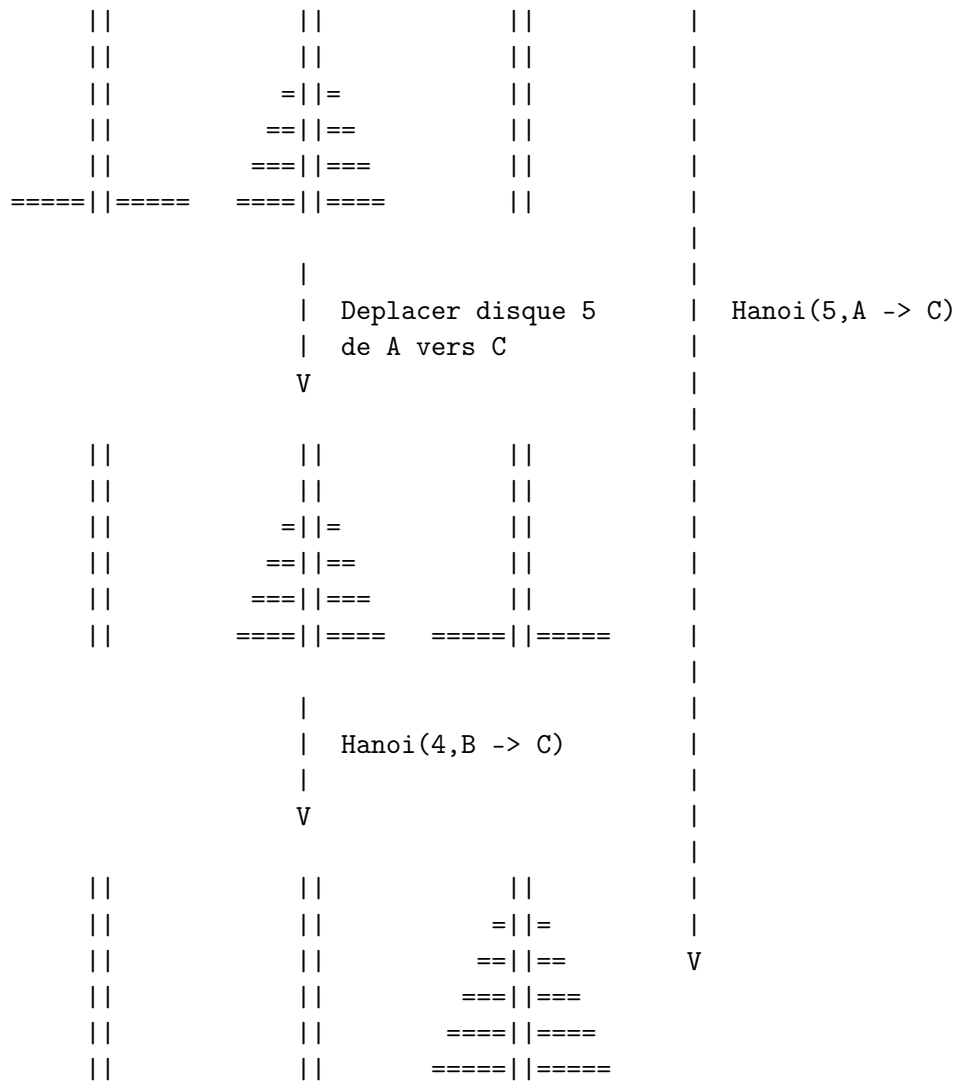
Voici une illustration d'une suite d'emplacement que l'on peut faire, dans le but de déplacer la pile du pilier A au pilier C.

A	B	C
= =		
== ==		
=== ===		
==== ====		
===== =====		



Idée: Il y a une manière de formuler la solution sur n disques en termes de la solution sur $n-1$ disques. En voici une illustration:





Cela donne un algorithme récursif pour les tours de Hanoi (voir dans les exercices).

3 Diviser pour régner

L'idée du paradigme *diviser pour régner* est la suivante:

- Étant donné un cas particulier d'un problème algorithmique à résoudre,
- On *décompose* le problème en des sous cas du même problème, que l'on peut résoudre indépendamment.

- On *combine* les solutions pour retrouver la solution du problème de départ.

Ce principe est appliqué **récursivement**:

- On fixe des cas de base, suffisamment simples pour être résolus directement.
- Les cas plus complexes sont décomposés successivement jusqu'à ce qu'on tombe sur un cas de base, et ensuite combinés en remontant.

Tous les algorithmes récursifs de la partie précédente sont des implémentations de ce paradigme.

Exemple: Recherche dichotomique (voir feuille d'exercices 1)

On a vu un cas particulier de la recherche dichotomique. Le problème général résolu par une recherche dichotomique est le suivant:

Étant donnée une liste *triée* L de taille $\text{len}(L)$, et un objet x , déterminer si x est dans L , et donner sa position s'il l'est.

On peut formuler la recherche dichotomique sur L de la manière suivante:

`def dichot(L):`

- **Cas de base:** $\text{len}(L) \leq 1$

– L contient zéro ou un élément. Si elle contient un élément, on regarde si c'est x . Dans ce cas, x est en position 0

- **Cas "complexe":** $\text{len}(L) \geq 2$

– `med = L[len(L)/2], L_inf = L[0:len(L)/2], L_sup = L[len(L)/2+1:len(L)]`

- * Si `x == med`: x est dans L , en position $\text{len}(L)/2$
- * Sinon, si `x < med`: faire `dichot(L_inf)`, si x est dans L_{inf} en position i , il est dans L en position i
- * Sinon, si `x > med`: faire `dichot(L_sup)`, si x est dans L_{sup} en position i , il est dans L en position $i + \text{len}(L)/2 + 1$

La complexité d'un algorithme de ce type s'exprime en termes de:

- Nombre de sous-problèmes à résoudre.
- Rapport de taille entre le problème initial et les sous-problèmes.
- Nombre d'opérations lors de la division et de la combinaison.

Regardons ce que cela donne pour la recherche dichotomique:

- Le problème est divisé en deux sous-problèmes, mais en réalité, on ne calcule jamais qu'un seul des sous-problème. On peut donc dire que le nombre de sous-problèmes est 1.
- Pour un problème de taille n (longueur de la liste), le sous-problème est de taille $n/2$.
- Le nombre d'opérations pour décomposer et recombinaison est toujours le même:
 - Calculer `len(L)/2`
 - Évaluer `med = L[len(L)/2]`
 - Comparer `med` avec `x`
 - Pour combiner, décaler l'indice `i` trouvé si besoin

Il suffit de dire que c'est une constante K .

Cela donne une formule récursive: si $C(n)$ est le nombre d'opérations (au pire des cas) en taille n , cela donne

$$C(n) \simeq 1 \times C(n/2) + K$$

Cela permet d'estimer l'ordre de complexité à $C(n) = O(\log(n))$

Exemple: Tri fusion

Voir exercices

4 Programmation dynamique

Regardons de plus près l'algorithme récursif, de type "diviser pour régner", pour calculer la suite de fibonacci.

```
def fibonacci(n):
    if n == 0:
        return 0

    elif n == 1:
        return 1

    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

- *Quelle est la complexité de cet algorithme?*

Un problème de taille n est divisé en un sous-problème de taille $n-1$ et un sous-problème de taille $n-2$, avec un nombre constant K d'opérations pour décomposer et combiner. La formule récursive pour la complexité est donc

$$C(n) = C(n-1) + C(n-2) + K$$

Pour faire simple, on peut raisonner sur les inégalités

$$2C(n-2) + K \leq C(n) \leq 2C(n-1) + K$$

La formule à droite est assez standard, et on peut la développer en

$$C(n) \leq C(0) \times 2^n + K \times \sum_{0 \leq i < n} 2^i = C(0) \times 2^n + K \times (2^n - 1)$$

Ce qui est de l'ordre de 2^n . (En bref, c'est la partie $2C(n-1)$ qui compte ici)

La formule à gauche peut être transformée en la première en prenant $C'(n) = C(2n)$, on a alors

$$C'(n) \geq 2C'(n-1) + K$$

Donc $C'(n)$ est plus grand qu'une suite d'ordre 2^n , et donc $C(n) = C'(n/2)$ est au moins d'ordre $2^{n/2} = (2^{1/2})^n$ (strictement on n'a montré cela que pour les n pairs, mais on a quelque chose de similaire pour les n impairs). On pourrait raffiner l'analyse, mais l'important est de voir que cet algorithme a une complexité exponentielle. Il devient donc rapidement très coûteux!

Problème: On recalcule plusieurs fois les mêmes valeurs, ce n'est pas nécessaire!

On peut éviter cela en faisant des **sauvegardes** des solutions déjà trouvées. Au lieu de recalculer des solutions, on accède à des valeurs existants.

Voici une implémentation itérative des nombres de fibonacci qui suit ce principe

```
def fibonacci(n):
    a,b = 0,1

    if n == 0:
        return a

    else:
        for i in range(1,n):
```

$a, b = b, a+b$

return b

- *Quelle est la complexité de cet algorithme?*

Cet algorithme a une seule boucle à n itérations, avec un nombre constant d'opérations à chaque itération. On a donc une complexité de n , bien meilleure que la version "diviser pour régner" de l'algorithme!

Dans cet, exemple, on garde toujours une valeur précédente en mémoire, à savoir **a**. Lorsqu'on en a besoin, il suffit d'accéder à la valeur sauvegardée.

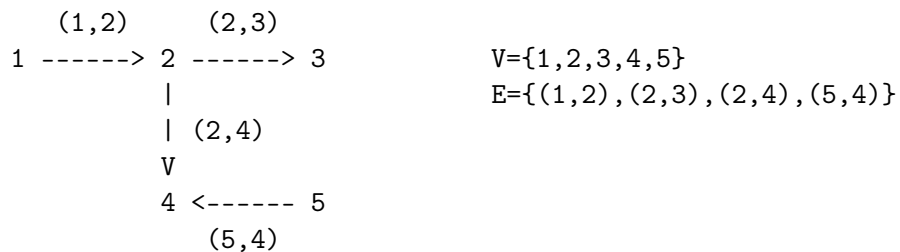
C'est le principe de la *programmation dynamique*. Dans ce paradigme, le principe est toujours de diviser un problème en sous-parties, mais par rapport à "diviser pour régner", on change l'ordre des opérations:

- On commence par les cas les plus "petits", et on monte jusqu'à la solution finale.
- Entretemps, on garde en mémoire des solutions déjà obtenues, que l'on utilise après dans les cas plus complexes.

Exemple: Chemin le plus court dans un graphe

Un *graphe* $G(V,E)$ est donné par:

- Des nœuds $V = \{1, \dots, n\}$
- Des arêtes $E \subset V \times V$. Une arête $e = (i,j)$ relie les nœuds i et j



Un *chemin de longueur* k dans un graphe est une suite d'arêtes $(i_0, i_1), (i_1, i_2), \dots, (i_{k-2}, i_{k-1}), (i_{k-1}, i_k)$ reliant deux nœuds i_0 et i_k . Les i_j , $0 < j < k$, sont les *nœuds intermédiaires* du chemin.

Problème: Pour un graphe $G(V,E)$, trouver, pour toute paire de nœuds i,j le chemin le plus court reliant ces nœuds (contenant le nombre minimal d'arêtes).

Explicitement, on veut une fonction `chemins(n,E)`, où n est le nombre de nœuds, et E est une liste de tuples (i,j) de nœuds entre 0 et n , qui donne comme résultat une matrice (liste de listes) C telle que $C[i][j]$ soit la longueur du chemin le plus court de i vers j , et $C[i][j] == \text{"infini"}$ sinon.

On peut associer à ce problème les sous-problèmes suivants:

Sous-problèmes: Pour $0 \leq k \leq n$, trouver, pour tout i,j , le chemin le plus court *en n'utilisant que les nœuds $1, \dots, k$ comme nœuds intermédiaires*. On va noter la solution `chemins_aux(n,E,k)`

C'est-à-dire:

- $k = 0$: aucun nœud intermédiaire, `chemins_aux(n,E,0)[i][j] == 1` si (i,j) est dans E , et `"infini"` sinon.
- $k = 1$: seulement le nœud 1 peut être intermédiaire. Les valeurs possibles dans `chemins_aux(n,E,1)` sont `"infini"`, 1 et 2.
- $k = 2$: le nœud 1, le nœud 2, ou les deux, peuvent être intermédiaires. Les valeurs possibles dans `chemins_aux(n,E,2)` sont `"infini"`, 1, 2 et 3.
- ...
- $k = n$: C'est le problème initial. `chemins_aux(n,E,n) == chemins(n,E)`.

Le cas $k = 0$ est simple, il suffit de tester si (i,j) est dans E pour calculer `chemins(n,E)[i][j]`.

A partir d'un cas $k \geq 0$, on peut construire le résultat pour $k+1$:

Pour deux nœuds i,j

- Si le chemin le plus court de i vers j , *en utilisant les nœuds intermédiaires $0, \dots, k+1$* , contient le nœud $k+1$
 - `chemins_aux(n,E,k+1)[i][j] == chemins_aux(n,E,k)[i][k+1] + chemins_aux(n,E,k)[k+1][j]`.
- Sinon
 - `chemins_aux(n,E,k+1)[i][j] == chemins_aux(n,E,k)[i][j]`

Pour tester si le chemin le plus court contient $k+1$, il suffit de comparer ces deux valeurs.

On a donc `chemins_aux(n,E,k+1)[i][j] == min(chemins_aux(n,E,k)[i][j], chemins_aux(n,E,k)[i][k+1] + chemins_aux(n,E,k)[k+1][j])`.

On trouve encore une forme récursive du problème. Mais c'est encore pire que pour les nombres de fibonacci: un algorithme "diviser pour régner" nécessiterait 3 sous-problèmes de taille $k-1$. La complexité serait de l'ordre de 3^n !

On fait de la programmation dynamique: on commence par la matrice `chemins_aux(n,E,0)`, et on calcule successivement les matrices `chemins_aux(n,E,k)` jusqu'au résultat final `chemins(n,E)`. A chaque étape, le calcul de `chemins_aux(n,E,k+1)[i][j]` consiste à chercher des valeurs sauvegardées de l'étape précédente.

L'algorithme que l'on obtient est:

`def chemins(n,E):`

- Construire `C = chemins(n,E,0)`
- Pour $0 \leq k \leq n$
 - Nouvelle matrice intermédiaire `C_tmp`, avec `C_tmp[i][j] = "infini"` partout
 - Pour $0 \leq i \leq n$
 - * Pour $0 \leq j \leq n$
 - `C_tmp[i][j] = min(C[i][j], C[i][k+1] + C[k+1][j])`
 - Mise à jour de `C`: `C = C_tmp`
- Rendre `C` en résultat

On peut "lire" directement la complexité de cet algorithme: On a 3 boucles imbriquées de n étapes, avec un nombre constant d'opérations dans chacune. La complexité est donc de l'ordre de n^3 : bien meilleure que pour l'approche "diviser pour régner"!