L3 SID APU
Cours 3
Méthodes de conception d'algorithmes
Application aux algorithmes de Tri
(suite)

Thomas Pellegrini, équipe SAMoVA, IRIT,
thomas.pellegrini@irit.fr

IRIT - UPS

2016-2017

Approches diviser-pour-régner pour le tri

Exemples communs

- Tri par tas
- Tri rapide
- **Tri fusion**

Approches diviser-pour-régner pour le tri

Exemples communs

- Tri par tas
- Tri rapide
- Tri fusion

Approches diviser-pour-régner pour le tri

Exemples communs

- Tri par tas
- Tri rapide
- **Tri fusion**

Méthode diviser-pour-régner

algorithmes qui utilisent une **récursion**

- **Diviser** le problème en un certain nombre de sous-problèmes qui sont des instances plus petites du même problème
- **Régner** sur les sous-problèmes en les résolvant de manière récursive
- **Combiner** les solutions des sous-problèmes pour produire la solution du problème originel

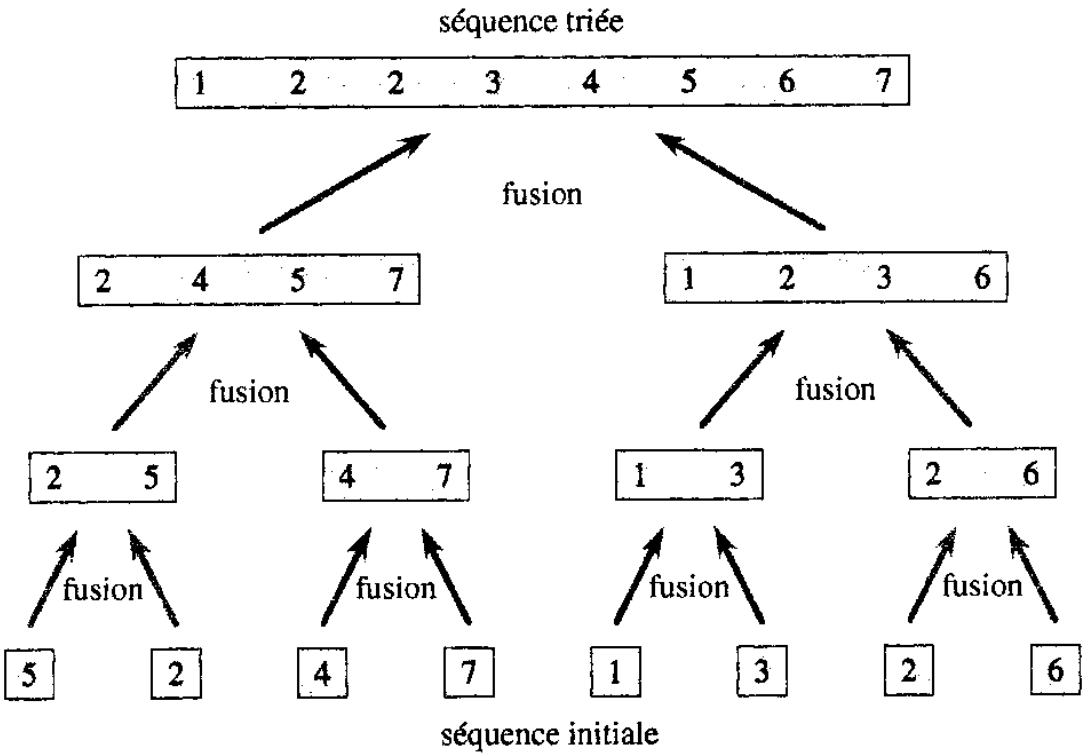
TRI-FUSION

Méthode diviser-pour-régner

- **Diviser :** diviser la suite de n éléments en deux sous-suites de $n/2$ éléments chacun
- **Régner :** Trier les deux sous-suites de manière récursive en utilisant le tri par fusion (récursion)
- **Combiner :** Fusionner les sous-suites triés deux par deux pour produire la suite originelle triée

$\Rightarrow$ Nous allons utiliser une fonction qui réalise l'opération de fusion : FUSION(A, p, q, r) avec A la suite (un tableau), et p, q, r trois entiers tels que $p \leq q < r$.
FUSION suppose que les sous-tableaux $A[p..q]$ et $A[q+1..r]$ sont triés. Elle les fusionne en un sous-tableau trié $A[p..r]$.

TRI-FUSION


séquence triée
1 2 2 3 4 5 6 7
fusion
2 4 5 7
1 2 3 6
fusion
fusion
2 5
4 7
1 3
2 6
fusion
fusion
fusion
fusion
5
2
4
7
1
3
2
6
séquence initiale

TRI-FUSION

```
1:  FONCTION FUSION(A :
    Entier[ ], p, q, r : Entier)
2:  Entier i, j, k ;
3:  n1 : Entier ← q - p + 1 ;
4:  n2 : Entier ← r - q ;
5:  Entier L[n1] ;
6:  Entier R[n2] ;
7:  POUR i de 0 à n1-1 FAIRE
8:      L[i] ← A[p+i] ;
9:  POUR i de 0 à n2-1 FAIRE
10:     R[i] ← A[q+i+1] ;
11: L[n1]← ∞ ;
12: R[n2]← ∞ ;
13: i←1 ;
14: j←1 ;
15: POUR k de p à r FAIRE
16:     SI L[i] ≤ R[j] ALORS
17:         A[k] ← L[i] ;
18:         i ← i+1 ;
19:     SINON
20:         A[k] ← R[j] ;
21:         j ← j+1 ;
22: FIN FONCTION
```

Algorithme TRI-FUSION(A, p, r)

```
1: SI p < r ALORS
2:     q ← E((p+q)/2)
3:     TRI-FUSION(A, p, q) ;
4:     TRI-FUSION(A, q+1, r) ;
5:     FUSION(A, p, q, r) ;
6: FIN SI
```

Question !

Pourquoi la fonction FUSION ne renvoie pas le tableau A ?

Question !

Pourquoi la fonction FUSION ne renvoie pas le tableau A ?
$\Rightarrow$ comment les paramètres d'entrée sont-ils passés à une fonction ?

Question !

Pourquoi la fonction FUSION ne renvoie pas le tableau A ?
$\Rightarrow$ comment les paramètres d'entrée sont-ils passés à une fonction ?

Un paramètre peut être transmis à une fonction :

1. par valeur : la valeur est clonée et la fonction travaille sur une variable interne
2. par référence : la fonction travaille sur la case mémoire correspondant à la variable qui stocke la valeur

Question !

Pourquoi la fonction FUSION ne renvoie pas le tableau A ?
$\Rightarrow$ comment les paramètres d'entrée sont-ils passés à une fonction ?

Un paramètre peut être transmis à une fonction :

1. par valeur : la valeur est clonée et la fonction travaille sur une variable interne
2. par référence : la fonction travaille sur la case mémoire correspondant à la variable qui stocke la valeur

En algorithmique (et en Java) :

1. Types simples (Entier, Réel, Caractère, Booléen) : par valeur
2. Types composés (tableau, liste, pile, file, arbre) : par référence

Passage de paramètres en python

En python, tout se passe par référence

```
a = [1, 2, 3] # a est une liste (un tableau)
b = a # on ne copie pas la liste a,
# on copie de l'adresse de a (la reference)
b.append(4)
print(a) # resultat ?
```

Passage de paramètres en python

En python, tout se passe par référence

```
a = [1, 2, 3] # a est une liste (un tableau)
b = a # on ne copie pas la liste a,
# on copie de l'adresse de a (la reference)
b.append(4)
print(a) # resultat ?
```

Comment réaliser une vraie copie de a ?

Passage de paramètres en python

Comment réaliser une vraie copie de a ?

```
a = [1, 2, 3]
b = list(a) # on cree une vraie copie de la liste
# cette fois, grace a list()
b.append(4)
print(a) # resultat ?
```

Passage de paramètres en python

Comment réaliser une vraie copie de a ?

```
a = [1, 2, 3]
b = list(a) # on cree une vraie copie de la liste
# cette fois, grace a list()
b.append(4)
print(a) # resultat ?
```

Notion complémentaire en python : objets mutables et objets non mutables

1. Types simples (int, float, bool, str, tuple, etc) : non mutables
2. Types composés (list, dict, set, etc) : mutables

Passage de paramètres en python

```
def fonction_qui_sert_a_rien(L):
    L.append(157)

a = [1, 2, 3]
print(a) # resultat ?
fonction_qui_sert_a_rien(a)
print(a) # resultat ?
```

Passage de paramètres en python

```
def fonction_qui_sert_a_rien(L):
    L.append(157)

a = [1, 2, 3]
print(a) # resultat ?
fonction_qui_sert_a_rien(a)
print(a) # resultat ?
```

La liste a été modifiée ! Car la liste est passée par référence à la fonction. Toute modification de la liste dans la fonction est donc visible en dehors de la fonction.

Passage de paramètres en python

```
def deuxieme_fonction_qui_sert_a_rien(a):
    a = a * 2
    return a

a = 4
print(a) # resultat ?
b = deuxieme_fonction_qui_sert_a_rien(a)
print(b) # resultat ?
print(a) # resultat ?
```

Passage de paramètres en python

```
def deuxieme_fonction_qui_sert_a_rien(a):
    a = a * 2
    return a

a = 4
print(a) # resultat ?
b = deuxieme_fonction_qui_sert_a_rien(a)
print(b) # resultat ?
print(a) # resultat ?
```

L'entier n'a pas été modifié ! Car même s'il est passé par référence à la fonction, il est non mutable.

Analyse de la complexité de TRI-FUSION

- Temps pris par FUSION ?
- Temps pris par TRI-FUSION ?

Analyse de complexité : notation asymptotique

- Notations pour décrire le temps d'exécution asymptotique d'un algorithme : par des fonctions définies sur $\mathbb{N} = \{1, 2, \ldots\}$

Analyse de complexité : notation asymptotique

- Notations pour décrire le temps d'exécution asymptotique d'un algorithme : par des fonctions définies sur $\mathbb{N} = \{1, 2, \ldots\}$
- $\ll$ asymptotique $\gg$: comportement quand n, la taille des entrées de l'algo, est très grand (n : taille d'un tableau pris en entrée par exemple)

Analyse de complexité : notation asymptotique

Notation O : borne asymptotique **supérieure**

- Pour une fonction f donnée définie sur $\mathbb{N}$, on note $O(f)$ ou $O(f(n))$ (prononcer « grand O de f ») l'ensemble des fonctions g suivant :

Analyse de complexité : notation asymptotique

Notation O : borne asymptotique **supérieure**

- Pour une fonction f donnée définie sur $\mathbb{N}$, on note $O(f)$ ou $O(f(n))$ (prononcer « grand O de f ») l'ensemble des fonctions g suivant :
- $O(f) = \{g$ telles qu'il existe des constantes positives c et n_0 telles que pour tout $n \geq n_0$, on a $0 \leq g(n) \leq cf(n)\}$

Analyse de complexité : notation asymptotique

Notation O : borne asymptotique **supérieure**

- Pour une fonction f donnée définie sur $\mathbb{N}$, on note $O(f)$ ou $O(f(n))$ (prononcer « grand O de f ») l'ensemble des fonctions g suivant :
- $O(f) = \{g$ telles qu'il existe des constantes positives c et n_0 telles que pour tout $n \geq n_0$, on a $0 \leq g(n) \leq cf(n)\}$
- f est un « majorant » des fonctions g (à une constante multiplicative près c et pour n suffisamment grand, i.e. n supérieur à un certain n_0)

Analyse de complexité : notation asymptotique

Notation O : borne asymptotique **supérieure**

- Pour une fonction f donnée définie sur $\mathbb{N}$, on note $O(f)$ ou $O(f(n))$ (prononcer « grand O de f ») l'ensemble des fonctions g suivant :
- $O(f) = \{g$ telles qu'il existe des constantes positives c et n_0 telles que pour tout $n \geq n_0$, on a $0 \leq g(n) \leq cf(n)\}$
- f est un « majorant » des fonctions g (à une constante multiplicative près c et pour n suffisamment grand, i.e. n supérieur à un certain n_0)
- Pour prouver que $g(n) = O(f(n))$: trouver des valeurs c de n_0 qui marchent

Analyse de complexité : notation asymptotique

Exemples :

- $T(n) = O(n^2)$ veut dire qu'un certain multiple constant de n^2 est un majorant de le temps d'exécution T

Analyse de complexité : notation asymptotique

Exemples :

- $T(n) = O(n^2)$ veut dire qu'un certain multiple constant de n^2 est un majorant de le temps d'exécution T
- TRI-SELECTION : les deux boucles POUR imbriquées donnent immédiatement un majorant en $O(n^2)$ pour le cas le plus défavorable

Analyse de complexité : notation asymptotique

Exemples :

- $T(n) = O(n^2)$ veut dire qu'un certain multiple constant de n^2 est un majorant de le temps d'exécution T
- TRI-SELECTION : les deux boucles POUR imbriquées donnent immédiatement un majorant en $O(n^2)$ pour le cas le plus défavorable
- Soit $T(n) = n^2 + 10n$, montrer que $T(n) = O(n^2)$?

Analyse de complexité : notation asymptotique

Exemples :

- $T(n) = O(n^2)$ veut dire qu'un certain multiple constant de n^2 est un majorant de le temps d'exécution T
- TRI-SELECTION : les deux boucles POUR imbriquées donnent immédiatement un majorant en $O(n^2)$ pour le cas le plus défavorable
- Soit $T(n) = n^2 + 10n$, montrer que $T(n) = O(n^2)$?
- $n^2 + 10n \leq n^2 + 10n^2$ soit $n^2 + 10n \leq 11n^2$: on prend $c = 11$ et $n_0 = 1$ par exemple

Analyse de complexité : notation asymptotique

Exemples :

- $T(n) = O(n^2)$ veut dire qu'un certain multiple constant de n^2 est un majorant de le temps d'exécution T
- TRI-SELECTION : les deux boucles POUR imbriquées donnent immédiatement un majorant en $O(n^2)$ pour le cas le plus défavorable
- Soit $T(n) = n^2 + 10n$, montrer que $T(n) = O(n^2)$?
- $n^2 + 10n \leq n^2 + 10n^2$ soit $n^2 + 10n \leq 11n^2$: on prend $c = 11$ et $n_0 = 1$ par exemple
- $O(1)$: coût constant, c'est le coût des instructions élémentaires (assignation, test, etc)

Analyse de complexité : notation asymptotique

		n=65536, k=2 coût basique : 0,01s
$O(1)$	Temps constant, indépendant de la taille des données	
$O(\log_2(n))$	Temps logarithmique	0,16s
$O(n)$	Temps linéaire	655s
$O(n\log_2(n))$	10486s, environ 3h	
$O(n^2)$	Temps quadratique	497 jours
$O(n^k), k > 2$	Temps polynomial	
$O(k^n), k > 1$	Temps exponentiel	$6,17 * 10^{19717}$ millénaires

Analyse de la complexité de TRI-FUSION

- Temps pris par FUSION ? On fusionne deux tableaux de q-p+1 et r-q éléments soit $n = r - p + 1$ éléments. Les deux boucles POUR des lignes 7 et 9 prennent un temps $O(n_1 + n_2) = O(n)$ et il y a n itérations de la boucle POUR ligne 15 $\rightarrow O(n)$
- Temps pris par TRI-FUSION ?

Analyse de la complexité de TRI-FUSION

- Temps pris par FUSION ? On fusionne deux tableaux de q-p+1 et r-q éléments soit $n = r - p + 1$ éléments. Les deux boucles POUR des lignes 7 et 9 prennent un temps $O(n_1 + n_2) = O(n)$ et il y a n itérations de la boucle POUR ligne 15 $\rightarrow O(n)$
- Temps pris par TRI-FUSION ? $\rightarrow$ analyse des algos diviser-pour-régner

Analyse des algos diviser-pour-régner

- Lorsque l'algorithme contient un appel récursif, on décrit son temps d'exécution à l'aide d'une relation de récurrence

Analyse des algos diviser-pour-régner

- Lorsque l'algorithme contient un appel récursif, on décrit son temps d'exécution à l'aide d'une relation de récurrence
- Soit $T(n)$ le temps d'exécution d'un problème de taille n,

Analyse des algos diviser-pour-régner

- Lorsque l'algorithme contient un appel récursif, on décrit son temps d'exécution à l'aide d'une relation de récurrence
- Soit $T(n)$ le temps d'exécution d'un problème de taille n,
- Supposons que l'on divise le problème en a sous-problèmes de taille n/b,

Analyse des algos diviser-pour-régner

- Lorsque l'algorithme contient un appel récursif, on décrit son temps d'exécution à l'aide d'une relation de récurrence
- Soit $T(n)$ le temps d'exécution d'un problème de taille n,
- Supposons que l'on divise le problème en a sous-problèmes de taille n/b,
- Il faut un temps $T(n/b)$ pour résoudre un sous-problème, donc un temps $aT(n/b)$ pour résoudre a sous-problèmes.

Analyse des algos diviser-pour-régner

- Lorsque l'algorithme contient un appel récursif, on décrit son temps d'exécution à l'aide d'une relation de récurrence
- Soit $T(n)$ le temps d'exécution d'un problème de taille n,
- Supposons que l'on divise le problème en a sous-problèmes de taille n/b,
- Il faut un temps $T(n/b)$ pour résoudre un sous-problème, donc un temps $aT(n/b)$ pour résoudre a sous-problèmes.
- Si l'on prend un temps $D(n)$ pour diviser le problème en sous-problèmes et un temps $C(n)$ pour construire la solution globale, on obtient la récurrence :

$$T(n) = \begin{cases} 0(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon} \end{cases}$$

Analyse des algos diviser-pour-régner

- Lorsque l'algorithme contient un appel récursif, on décrit son temps d'exécution à l'aide d'une relation de récurrence
- Soit $T(n)$ le temps d'exécution d'un problème de taille n,
- Supposons que l'on divise le problème en a sous-problèmes de taille n/b,
- Il faut un temps $T(n/b)$ pour résoudre un sous-problème, donc un temps $aT(n/b)$ pour résoudre a sous-problèmes.
- Si l'on prend un temps $D(n)$ pour diviser le problème en sous-problèmes et un temps $C(n)$ pour construire la solution globale, on obtient la récurrence :

$$T(n) = \begin{cases} 0(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon} \end{cases}$$

- Pour TRI-FUSION, $a = b = 2$

Analyse de la complexité de TRI-FUSION

- **Diviser :** l'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, donc $D(n) = O(1)$

Analyse de la complexité de TRI-FUSION

- **Diviser :** l'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, donc $D(n) = O(1)$
- **Régner :** on résoud deux sous-problèmes, chacun de taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution

Analyse de la complexité de TRI-FUSION

- **Diviser :** l'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, donc $D(n) = O(1)$
- **Régner :** on résoud deux sous-problèmes, chacun de taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution
- **Combiner :** c'est la procédure FUSION qui prend un temps $C(n) = O(n)$

Analyse de la complexité de TRI-FUSION

- **Diviser :** l'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, donc $D(n) = O(1)$
- **Régner :** on résoud deux sous-problèmes, chacun de taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution
- **Combiner :** c'est la procédure FUSION qui prend un temps $C(n) = O(n)$

- $D(n) + C(n) = O(1) + O(n) = O(n)$, terme linéaire domine le terme constant

Analyse de la complexité de TRI-FUSION

- **Diviser :** l'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant, donc $D(n) = O(1)$
- **Régner :** on résoud deux sous-problèmes, chacun de taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution
- **Combiner :** c'est la procédure FUSION qui prend un temps $C(n) = O(n)$

- $D(n) + C(n) = O(1) + O(n) = O(n)$, terme linéaire domine le terme constant
- on obtient la récurrence suivante :

$$T(n) = \begin{cases} 0(1) & \text{si } n = 1, \\ 2T(n/2) + O(n) & \text{si } n > 1 \end{cases}$$

Analyse de la complexité de TRI-FUSION

On ré-écrit la récurrence avec une constante c qui est le temps requis pour résoudre les problèmes de taille 1 :

$$T(n) = \begin{cases} 0(1) & \text{si } n = 1, \\ 2T(n/2) + cn & \text{si } n > 1 \end{cases}$$

Pour trouver $T(n)$, on construit un **arbre récursif**

Analyse de la complexité de TRI-FUSION

$$T(n) = \begin{cases} 0(1) & \text{si } n = 1, \\ 2T(n/2) + cn & \text{si } n > 1 \end{cases}$$

Analyse de la complexité de TRI-FUSION

$$T(n) = \begin{cases} 0(1) & \text{si } n = 1, \\ 2T(n/2) + cn & \text{si } n > 1 \end{cases}$$

$$\Rightarrow \text{résultat : } T(n) = cn(\log(n) + 1)$$

$$\text{Soit } T(n) = O(n \log(n))$$