Algorithmes Avancés
–
IUFM

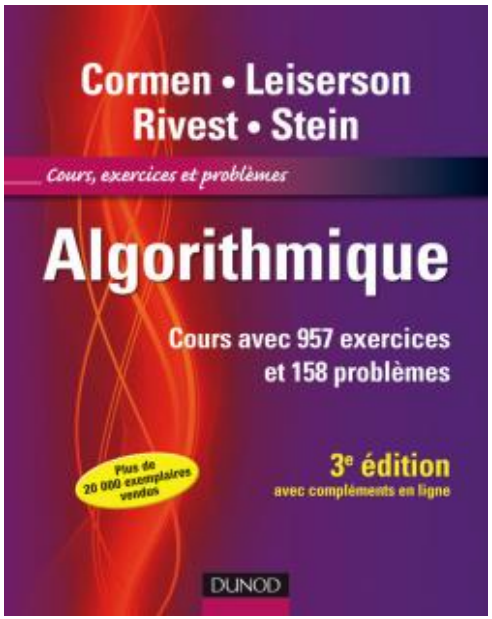
Vincent Limouzy
Limos - Université Blaise Pascal

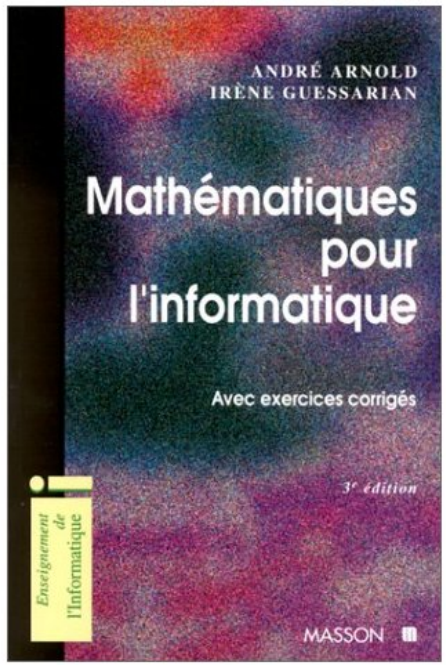
LIMOS

5 Avril 2012

1. Rappels
2. Complexité
3. Correction
4. Algorithmes de tri
   - Tri par sélection
   - Tri fusion
   - Tri rapide : QuickSort
   - Dichotomie
5. Algorithmes gloutons

Bibliographie





Definition

Algorithme : Un algorithme est une séquence finie d'opérations qui répond à une question donnée et qui s'exécute en un temps fini (sur une instance fini).

Propriétés fondamentales d'un algorithme :

- La correction
- L'arrêt
- La complexité

Complexité algorithmique

Definition

Complexité d'un Algorithme : La complexité d'un algorithme $\mathscr{A}$ est la mesure qui permet de quantifier les ressources consommées par l'algorithme sur un instance donnée.

Les ressources qui sont habituellement considérées sont le **temps** et **l'espace mémoire** nécessaire à l'algorithme.

Paramètres

On mesurera la complexité de l'algorithme en fonction de la **taille** de l'instance. La taille de l'instance sera généralement représentée par la variable n.

Afin de mesurer la complexité d'un algorithme, on doit évaluer l'espace nécessaire et le nombres d'instructions effectuées.

Espace :

Pour mesurer l'espace, il suffit en général de considérer les variables et les structures de données utilisées.

Nombre d'instructions = Temps :

Le temps d'exécution d'un algorithme dépend fortement du nombre d'instructions à exécuter. Chaque instruction (opérations arithmétiques, affectations,...) s'exécute en un temps fini que l'on peut borner par une constante.

Comptage des instructions

Pour évaluer finement un algorithme on peut compter exactement le nombre d'instructions vraiment effectués.

Cependant, cette approche peut être fastidieuse et n'est pas forcément nécessaire. On peut se contenter de mesurer le comportement asymptotique.

Fonctions asymptotiques : O, Ω, Θ

Pire des cas

On cherche à trouver la fonction $g(n)$ la plus petite telle que le temps d'exécution est inférieur ou égal à $g(n) \times c$. Pour cela on utilise la notation **O**.

$$\mathbf{O(g(n))} = \begin{array}{c} \{f(n) : \exists c > 0 \text{ une constante et } n_0 \text{ tels que} \\ 0 \leq f(n) \leq c \cdot g(n), \quad \forall n \geq n_0\} \end{array}$$

Fonctions asymptotiques : O, Ω, Θ

Meilleur des cas

On cherche à trouver une fonction $g(n)$ la plus grande possible telle que le temps d'exécution de l'algorithme est supérieur ou égal à $c \times g(n)$.
On utilise la notation $\mathbf{\Omega}$.

$$\mathbf{\Omega(g(n))} = \begin{array}{c} \{f(n) : \exists c > 0 \text{ une constante et } n_0 \text{ tels que} \\ 0 \leq c \cdot g(n) \leq f(n), \quad \forall n \geq n_0\} \end{array}$$

Analyse en moyenne

Pour exprimer le temps moyen nécessaire pour traiter une instance on utilise la notation Θ.

$$\mathbf{\Theta(g(n))} = O(g(n)) \cap \Omega(g(n))$$

Exemples

Quelle est la complexité de cette fonction ?

Function `df`(*float f*)

```
1 float z := f
2 z := z * 2
3 return z
```

Exemples

Quelle est la complexité de cette fonction ?

Function `df` (*float f*)

```
1 float z := f
2 z := z * 2
3 return z
```

Réponse :

O(1)

Exemples

Quelle est la complexité de cette fonction ?

Function `df` (*float f*)

```
1 float z := f
2 z := z * 2
3 return z
```

Réponse :

O(1)

Explication

Le temps d'exécution dépend du nombre d'instructions, et ici le nombre d'instructions, quelle que soit la valeur de f, ne varie pas.
On dit que la complexité de cette fonction est constante.

Exemples

Exemple : Racine carrée

Function `Racine`(*int n*)

```
1 i := 0
2 while i × i ≤ n do
3     i := i + 1
4 return i − 1
```

Quelle est la complexité de cette fonction ?

Exemples

Exemple : Racine carrée

Function `Racine`(*int n*)

```
1 i := 0
2 while i × i ≤ n do
3     i := i + 1
4 return i − 1
```

Quelle est la complexité de cette fonction ?

Réponse :

$O(\sqrt{n})$

Algorithmes polynomiaux

On dit qu'un algorithme $\mathscr{A}$ a une **complexité polynomiale** (ou est dit polynomial) si il existe une constante positive k telle que pour tout instance $\mathcal{I}$ de taille n l'algorithme s'exécute en temps $\mathbf{O(n^k)}$.

Algorithmes polynomiaux

On dit qu'un algorithme $\mathscr{A}$ a une **complexité polynomiale** (ou est dit polynomial) si il existe une constante positive k telle que pour tout instance $\mathcal{I}$ de taille n l'algorithme s'exécute en temps $\mathbf{O(n^k)}$.

Question :

Est-ce que la fonction `Racine(int n)` est un algorithme polynomial ?

Afin de montrer qu'une fonction correspond aux définitions d'un algorithme on doit s'assurer :

- qu'il se termine
- et qu'il est correct.

Pour montrer qu'un algorithme est valide on utilise les notions d'invariants.

Ressources

Logiciel Tripatouille crée par R. Malgouyres
`http://www.malgouyres.fr/tripatouille/`

La documentation associée :
`http://www.irem.univ-bpclermont.fr/spip.php?rubrique32`

Les tableaux

Notations

- *int* $T[n]$: T est un tableau d'entiers qui comporte n éléments.
- $T[0]$: le premier élément.
- $T[n-1]$: le dernier élément.
- Un élément d'un tableau est comme une variable :
  $T[k] := T[k-1] + T[0] * 2$
- L'accès à $T[0]$, $T[i]$ ou $T[n-1]$ prend le même temps : $\mathbf{O(1)}$(Modèle RAM).

Tri par sélection : Principe

25	12	3	9	-12	10

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

10	-12	3	9	12	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

10	-12	3	9	12	**25**

10	-12	3	9	**12**	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

10	-12	3	9	12	**25**

10	-12	3	9	**12**	**25**

9	-12	3	10	**12**	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

10	-12	3	9	12	**25**

10	-12	3	9	**12**	**25**

9	-12	3	10	**12**	**25**

9	-12	3	**10**	**12**	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

10	-12	3	9	12	**25**

10	-12	3	9	**12**	**25**

9	-12	3	10	**12**	**25**

9	-12	3	**10**	**12**	**25**

3	-12	9	**10**	**12**	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

25	12	3	9	-12	10

10	12	3	9	-12	25

10	12	3	9	-12	**25**

10	-12	3	9	12	**25**

10	-12	3	9	**12**	**25**

9	-12	3	10	**12**	**25**

9	-12	3	**10**	**12**	**25**

3	-12	9	**10**	**12**	**25**

3	-12	**9**	**10**	**12**	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

9	-12	3	10	**12**	**25**

25	12	3	9	-12	10

9	-12	3	**10**	**12**	**25**

10	12	3	9	-12	25

3	-12	9	**10**	**12**	**25**

10	12	3	9	-12	**25**

3	-12	**9**	**10**	**12**	**25**

10	-12	3	9	12	**25**

-12	3	**9**	**10**	**12**	**25**

10	-12	3	9	**12**	**25**

Tri par sélection : Principe

25	12	3	9	-12	10

9	-12	3	10	**12**	**25**

25	12	3	9	-12	10

9	-12	3	**10**	**12**	**25**

10	12	3	9	-12	25

3	-12	9	**10**	**12**	**25**

10	12	3	9	-12	**25**

3	-12	**9**	**10**	**12**	**25**

10	-12	3	9	12	**25**

-12	3	**9**	**10**	**12**	**25**

10	-12	3	9	**12**	**25**

-12	**3**	**9**	**10**	**12**	**25**

Fonctions Annexes

Function `Echange`(*int T[n], int i, int j*)

```
1 int z := T[i]
2 T[i] := T[j]
3 T[j] := z
4 return T
```

Fonctions Annexes (suite)

Function `IMax(int T[n], int j)`

```
1 int indice = 0
2 if j > n − 1 then
3     return -1
4 for i := 0 to j do
5     if T[i] > T[indice] then
6         indice := i
7 return indice
```

TriSelection

Function `TriSelection(`*int T[n]*`)`

```
1 int i := n − 1;
2 int IndiceMax;
3 while i > 0 do
4 |   IndiceMax = IMax(T[n],i)
5 |   Echange(T[n],i, IndiceMax)
6 |   i := i − 1
7 return T
```

Complexité

Echange

La complexité de la fonction `Echange(int T[n], int i, int j)` est $O(1)$.

Complexité

Echange

La complexité de la fonction `Echange(int T[n], int i, int j)` est $O(1)$.

IndiceMax

La complexité de la fonction `IndiceMax` dépend de j (avec $j \leq n-1$)
Chaque opération dans la boucle `For` (comparaison et affectation) a pour complexité $O(1)$. On va faire exactement $j+1$ itérations : on va examiner toutes les cases du tableau comprises entre 0 et j.
On fait donc $j+1$ fois des opérations à complexité constante : La complexité globale de la fonction est $O(j)$.

Complexité (suite)

TriSelection

La fonction `TriSelection` va effectuer $n-1$ itérations. à chaque itération :

- On appelle la fonction `IMax` sur le tableau $T[0, i]$
- On appelle la fonction `Echange`
- On décrémente la valeur de i.

Appeler la fonction `Echange` et décrémenter la valeur de i se fait en temps constant : $O(1)$.
L'appel à la fonction `IMax` est plus problématique : en effet à chaque itération le coût de le fonction `IMax` n'est pas le même.
On pourrait faire l'hypothèse qu'un appel à la fonction `IMax` est toujours en $O(n)$...
... mais ça peut aboutir à un estimation de la complexité trop imprécise !

Complexité (suite)

On doit donc estimer le comportement global des différents appels à cette fonction.
Si on reprend l'exemple : pour trouver le plus grand élément à la première itération on doit consulter tous les éléments du tableau. A l'itération suivante, on doit tous les consulter sauf un etc...

Complexité (suite)

On doit donc estimer le comportement global des différents appels à cette fonction.
Si on reprend l'exemple : pour trouver le plus grand élément à la première itération on doit consulter tous les éléments du tableau. A l'itération suivante, on doit tous les consulter sauf un etc...

Itération :	1	2	...	$n-2$	$n-1$
N° Opérations :	n	$n-1$	...	3	2

Complexité (suite)

Nombre total d'opérations

Le nombre total des opérations est la somme des opérations effectuées lors de chaque itération :

$$n + n - 1 + \ldots + 2 = \sum_{k=2}^{n} k \tag{1}$$

$$\sum_{k=2}^{n} k = (\sum_{k=1}^{n} k) - 1 \tag{2}$$

$$(\sum_{k=1}^{n} k) - 1 = \frac{n \times (n-1)}{2} - 1 \tag{3}$$

$$(\frac{n \times (n-1)}{2} - 1) \in O(n^2) \tag{4}$$

Correction

Une fois que la complexité de l'algorithme est trouvé, on doit s'assurer que l'algorithme fournit une réponse correcte pour toutes les instances potentielles.
Pour ce faire on doit montrer que toutes les fonctions utilisées sont correctes : à savoir :

- `Echange`
- `IMax`
- `TriSelection`

Echange

Pour la fonction `Echange` il est facile de s'assurer que la fonction est correcte.

Correction (IMax)

La fonction `IMax` prend en paramètre un tableau $T[n]$ et un indice j, avec $j \leq n-1$. Nous devons montrer que la fonction trouve l'indice de l'élément maximum dans le tableau qui commence à l'indice 0 et qui finit à l'indice j.

Invariant (P)

À l'étape i, *indice* représente l'indice de l'élément maximum du tableau $T[0, i]$.

Correction (IMax)

La fonction `IMax` prend en paramètre un tableau $T[n]$ et un indice j, avec $j \leq n-1$. Nous devons montrer que la fonction trouve l'indice de l'élément maximum dans le tableau qui commence à l'indice 0 et qui finit à l'indice j.

Invariant (P)

À l'étape i, *indice* représente l'indice de l'élément maximum du tableau $T[0, i]$.

Preuve

Par récurrence : Quand i vaut 0, le tableau $T[0, i]$ n'a qu'un seul élément $T[0]$ et comme indice =0, cette variable représente bien l'indice de l'élément maximum.

$P_i \Rightarrow P_{i+1}$ Il y a deux possibilités :

1. $T[i+1] < T[indice]$
2. $T[i+1] \geq T[indice]$

Correction (IMax)

(1) Si $T[i+1] < T[indice]$, par hypothèse de récurrence la propriété (P_i) est vrai pour le tableau $T[0-i]$, ce qui signifie que *indice* représente l'indice de la valeur maximum dans $T[0-i]$, or si on est dans le cas(1), cela signifie $T[indice]$ est également l'élément maximum pour le tableau $T[0-i+1]$.

(2)Si $T[i+1] \geq T[indice]$, cela signifie que $T[i+1]$ est plus grand que tous les éléments de $T[0,i]$. En effet par hypothèse $T[indice]$ représente l'élément maximum de $T[0,i]$, or si on trouve un élément plus grand en position $i+1$, cet élément devient le maximum pour le tableau $T[0,i+1]$. La fonction `IMax` est donc correcte. □

Correction (TriSelection)

Nous devons maintenant montrer que la fonction `TriSelection` est correcte. On doit pour cela trouver des invariants.

Invariant 1 (I1)

Le tableau $T[i, n-1]$ est trié.

Correction (TriSelection)

Nous devons maintenant montrer que la fonction `TriSelection` est correcte. On doit pour cela trouver des invariants.

Invariant 1 (I1)

Le tableau $T[i, n-1]$ est trié.

Invariant 2 (I2)

L'élément maximum contenu dans $T[0, i-1]$ est inférieur ou égal à l'élément minimum contenu dans $T[i, n-1]$.

Correction (TriSelection)

Nous devons maintenant montrer que la fonction `TriSelection` est correcte. On doit pour cela trouver des invariants.

Invariant 1 (I1)

Le tableau $T[i, n-1]$ est trié.

Invariant 2 (I2)

L'élément maximum contenu dans $T[0, i-1]$ est inférieur ou égal à l'élément minimum contenu dans $T[i, n-1]$.

Remarques

I1 est suffisant pour montrer qu'on obtient un tableau trié. En effet quand i vaut 0 (càd après la dernière itération), on a $T[i, n-1] = T[0, n-1] = T$ qui est trié. I2, nous dit qu'une fois qu'on a traité l'itération i, les éléments du tableau $T[i, n-1]$ sont à leur place définitives.

Correction (TriSelection)

I1

Il est facile de voir $T[i, n-1]$ est trié. En effet en sélectionnant l'élément maximum de tout le tableau et en le plaçant à la fin. Et ensuite en procédant sur le tableau restant...

Correction (TriSelection)

I1

Il est facile de voir $T[i, n-1]$ est trié. En effet en sélectionnant l'élément maximum de tout le tableau et en le plaçant à la fin. Et ensuite en procédant sur le tableau restant...

I2

Le même type d'argument est employé pour I2, on s'appuie sur les propriétés de `IMax`

Tri Fusion

Afin d'obtenir des meilleurs performances en termes de complexité, l'algorithme du tri Fusion (*Merge sort*) s'appuie sur deux paradigmes :

- Diviser pour régner,
- Travailler avec des données structurées.

Tri Fusion

Afin d'obtenir des meilleurs performances en termes de complexité, l'algorithme du tri Fusion (*Merge sort*) s'appuie sur deux paradigmes :

- Diviser pour régner,
- Travailler avec des données structurées.

Question 1 :

La question clé : soient T_1 et T_2 deux tableaux de n_1 et n_2 entiers respectivement. On souhaite fusionner T_1 et T_2 en un seul tableau T. On veut en plus que T soit trié. Quelle sera la complexité de cette fonction ?

Tri Fusion

Afin d'obtenir des meilleurs performances en termes de complexité, l'algorithme du tri Fusion (*Merge sort*) s'appuie sur deux paradigmes :

- Diviser pour régner,
- Travailler avec des données structurées.

Question 1 :

La question clé : soient T_1 et T_2 deux tableaux de n_1 et n_2 entiers respectivement. On souhaite fusionner T_1 et T_2 en un seul tableau T. On veut en plus que T soit trié. Quelle sera la complexité de cette fonction ?

Question 2

Est-ce que ça change quelque chose si T_1 et T_2 sont triés ou, si on ne fait aucune hypothèse sur les tableaux ?

Fusion

Réponse Q2 : Oui !

Principe :

Pour fusionner deux tableaux triés, il suffit de parcourir en même temps les deux tableaux en partant du plus petit élément (i.e. $T_1[0]$ et $T_2[0]$) jusqu'à ce qu'un des deux tableaux soit complètement parcouru.

Function `Fusion`(*int T[n], U[m]*)

```
1  int V[n + m]; int i := 0; int j := 0; int z := 0
2  while (i < n and j < m) do
3  |   if T[i] < U[j] then
4  |   |   V[z] := T[i]; i := i + 1
5  |   else
6  |   |   V[z] := U[j]; j := j + 1
7  |   z := z + 1
8  while (i < n) do
9  |   V[z] := T[i]; i := i + 1
10 |   z := z + 1
11 while (j < m) do
12 |   V[z] := U[j]; j := j + 1
13 |   z := z + 1
14 return V
```

Principe de fonctionnement

Décomposition

25	12	3	9	-12	10

Principe de fonctionnement

Décomposition

25	12	3	9	-12	10

25	12	3

9	-12	10

Principe de fonctionnement

Décomposition

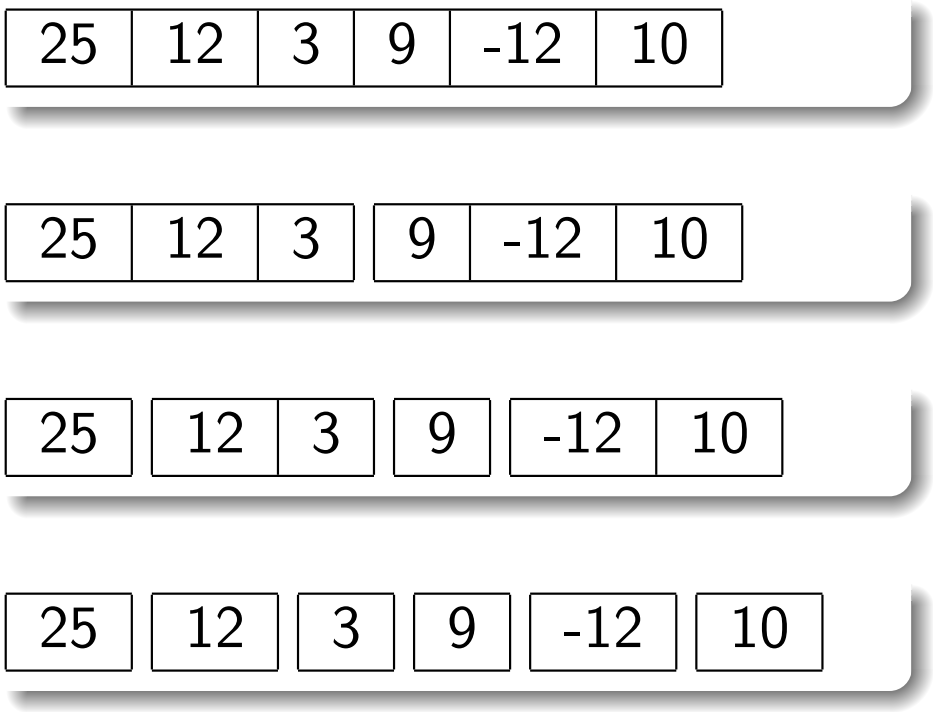
25	12	3	9	-12	10

25	12	3	9	-12	10

25	12	3	9	-12	10

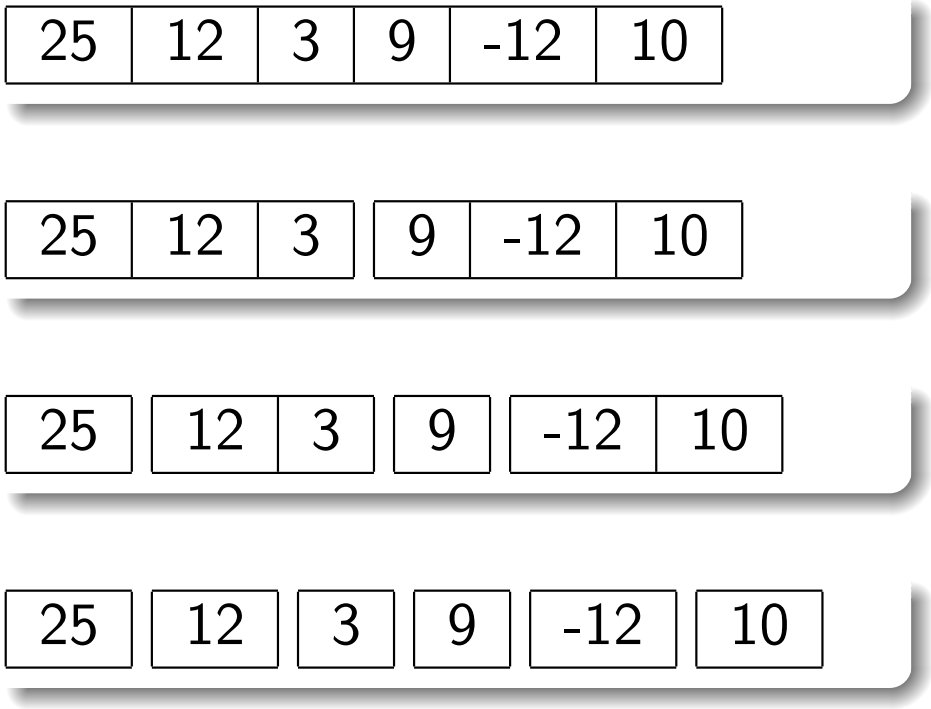
Principe de fonctionnement

Décomposition



Principe de fonctionnement

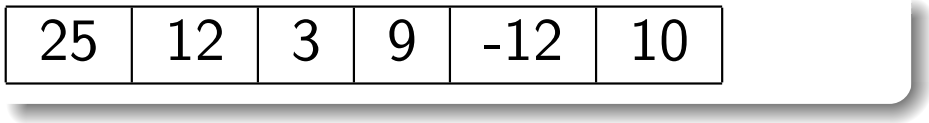
Décomposition

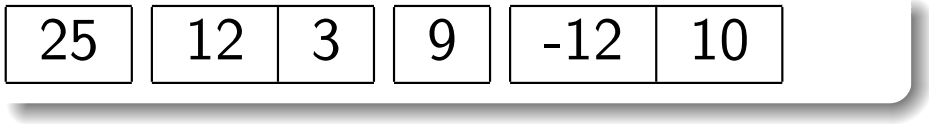


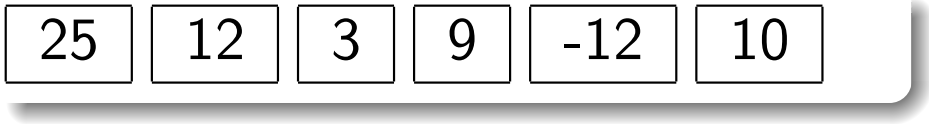
Fusion

Principe de fonctionnement

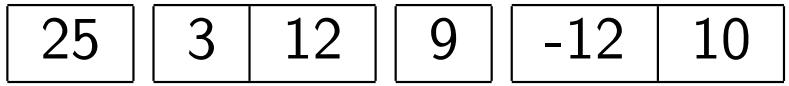
Décomposition







Fusion



Principe de fonctionnement

Décomposition

25	12	3	9	-12	10

25	12	3	9	-12	10

25	12	3	9	-12	10

25	12	3	9	-12	10

Fusion

25	3	12	9	-12	10

3	12	25	-12	9	10

Principe de fonctionnement

Décomposition

25	12	3	9	-12	10

25	12	3	9	-12	10

25	12	3	9	-12	10

25	12	3	9	-12	10

Fusion

25	3	12	9	-12	10

3	12	25	-12	9	10

-12	3	9	10	12	25

Principe

Le principe de l'algorithme tri fusion consiste à découper le tableau jusqu'à obtenir des tableaux triés.

Une façon de s'assurer qu'un tableau est trié est de considérer les tableaux à un seul élément.

Une fois que nous avons des tableaux triés on peut les fusionner deux à deux en temps.

L'algorithme

Function `TriFusion`(*int T[n]*)

```
1 if n == 1 then
2     return T
3 int A[n/2]
4 int B[n − n/2]
5 for i = 0 to n/2 − 1 do A[i] = T[i]
6 for i = n/2 to n − 1 do B[i − (n/2)] = T[i]
7 A := TriFusion(A)
8 B := TriFusion(B)
9 return Fusion (A,B)
```

Complexité

Exercice 1 :

Quelle est la complexité de la fonction `Fusion` ?

Exercice 2 :

Quelle est la complexité de la fonction `TriFusion` ?

Correction

Lemme

La fonction `Fusion` est valide.

Théorème

La fonction `TriFusion` est valide.

Correction

Lemme

La fonction `Fusion` est valide.

Preuve

Par hypothèse T et U sont triés on parcours simultanément T et U, et à chaque itération, on transfère un élément de T ou de U qui est le plus petit dans V .
La propriété suivante est vérifiée : à l'étape z (et les valeurs correspondantes de i et j), pour tous les éléments de $V[0, z]$ on a :

- $V[0, z]$ est trié par odre croissant
- Tous les éléments de $V[0, z]$ sont inférieurs ou égaux à tous les éléments contenus dans $T[i, n-1]$ et $U[j, m-1]$.

Théorème

La fonction `TriFusion` est valide.

Correction (suite)

Remarque

Par construction, la fonction `Fusion` dans la fonction `TriFusion` ne sera appelée qu'avec, pour arguments, des tableaux triés.

Drapeau Hollandais

Le problème :

Étant donné un tableau T de n éléments où tous les éléments sont des éléments de $\{1, 2, 3\}$.

On souhaite trier le tableau T en temps $\mathbf{O(n)}$

Drapeau Hollandais

Function `Ranger(int T[n])`

```
1  int i := 0
2  int j := 0
3  int k := n − 1
4  while j ≤ k do
5      if T[j] == 1 then
6          Echanger(T[n],i,j)
7          i := i + 1
8          j := j + 1
9      if T[j] == 2 then
10         j := j + 1
11     if T[j] == 3 then
12         Echanger(T[n],j,k)
13         k := k − 1
14 return T[n]
```

Pivot

Function `Pivot(int T[n], int p)`

```
1  Echange(T,p,0)
2  int i := 1
3  int f := n − 1
4  while i < f do
5      if T[i] ≤ T[0] then
6          i := i + 1
7      else
8          Echange(T,i,f)
9          f := f − 1
10 if T[i] < T[0] then
11     Echange(T,i,0) return i
12 else
13     Echange(T,i-1,0) return i-1
```

Pivot

Remarque

À l'issu de la fonction `Pivot` l'élément $T[p]$ choisi comme pivot est à sa place définitive dans un tableau trié.
Tous les éléments plus petits sont à gauche et tous les élément plus grand sont à droite.

Tri Rapide

Principe

Soit T notre tableau à trier :

- choisir un élément p comme pivot et utiliser la fonction `Pivot`.
- soit l la position de p après la fonction `Pivot`
- Relancer récursivement cette procédure sur $T[0, l-1]$ et $T[l+1, n-1]$

Complexité

Pivot

La complexité de la fonction Pivot est linéaire : $\mathbf{O(n)}$: à chaque étape de l'algorithme, on ajoute un élément dans la liste des plus petits ou celle des plus grands. Et comme tous les opérations au cours d'une itération sont en $\mathbf{O(1)}$, on obtient $\mathbf{O(n)}$.

TriRapide

Contrairement au TriFusion, la complexité peut varier.

- Quel est le paramètre qui peut faire varier la complexité ?
- Quel est le pire des cas ?
- Quel est le meilleur des cas ?
- Est ce que le tri pas sélection est un cas particulier ?

Principe

Propriétés

- Basé sur le principe de la recherche d'un mot dans un dictionnaire
- Consiste à diviser par 2, à chaque étape, l'espace de recherche.
- Permet d'obtenir des algorithmes avec pour complexité $\mathbf{O(\log(n))}$

Applications

- Calculer la racine carrée. (et autres fonctions...)
- Trouver la représentation binaire d'un nombre.
- Trouver un élément dans un tableau trié.
- ...

Un algorithme est dit glouton à partir du moment, où quand on décide d'inclure un élément dans la solution, son appartenance à la solution n'est jamais remise en cause.

Certains de ces algorithmes peuvent être regroupés sous un même formalisme celui des matroïdes pondérés.

Matroïdes

Un matroïde $\mathcal{M}$ est un couple $(E, \mathcal{I})$ qui vérifie les conditions suivantes :

1. E est un ensemble fini non vide.
2. $\mathcal{I}$ est une famille de sous-ensembles de E. telle que si $F \in \mathcal{I}$ et $H \subset F$ alors $H \in \mathcal{I}$.
3. (échange) Soient $F, H \in \mathcal{I}$ et $|H| < |F|$ alors il existe un élément e de $F - H$ tel que $H \cup \{e\} \in \mathcal{I}$.

Matroïdes

Propriétés

Les éléments de $\mathcal{I}$ sont appelés des ensembles indépendants.
Les ensembles indépendants maximaux sont appelés les bases.
Les bases ont toutes la même cardinalité.
La cardinalité d'une base est le rang du matroïde.

Matroïde pondéré

Un matroïde est pondéré si pour tous les éléments de E il existe une fonction $w : E \to \mathbb{R}$
Le poids d'un membre F de $\mathcal{I}$ est : $w(F) = \sum_{e \in F} w(e)$

Exemple

Arbre couvrant

Les sous-arbres (pas forcément connexes), vues comme une familles d'arêtes, d'un graphe forme un matroïde.
Les bases de ce matroïdes sont les arbres couvrants.

Algorithme

Problème

On cherche à trouver dans $\mathcal{M}$ un ensemble indépendant de poids maximum.

Algorithme

Function `Glouton(`M`,`w`)`

```
1 F := ∅
2 Trier E(M) par ordre décroissant selon w
3 foreach e ∈ E(M) par ordre décroissant do
4     if (F ∪ {e}) ∈ I then
5         F := F ∪ {e}
6     return F
```