Algorithmique

Trier et Trouver

Florent Hivert

Mél : `Florent.Hivert@lri.fr`
Page personnelle : `http://www.lri.fr/~hivert`

Algorithmes et structures de données

La plupart des bons algorithmes fonctionnent grâce à une méthode astucieuse pour organiser les données. Par exemple, on sait très bien, intuitivement, que pour retrouver une carte dans un jeu, il est très utile que le jeu soit trié.

Trouver et Trier :

- Donald E. Knuth, *The Art of Computer Programming (TAOCP), Volume 3 : Sorting and Searching*, Addison-Wesley, 1998.

Algorithmes et structures de données

La plupart des bons algorithmes fonctionnent grâce à une méthode astucieuse pour organiser les données. Par exemple, on sait très bien, intuitivement, que pour retrouver une carte dans un jeu, il est très utile que le jeu soit trié.

Trouver et Trier :

- Donald E. Knuth, *The Art of Computer Programming (TAOCP), Volume 3 : Sorting and Searching*, Addison-Wesley, 1998.

Recherche dans un tableau, dichotomie

Algorithme de recherche d'un élément dans un tableau

Algorithme

- ***Entrée :*** *un tableau* `tab` *de taille* `taille` *et un élément* `e`*.*
- ***Sortie :*** `i` *tel que* `tab[i] = e` *ou* `NonTrouvé` *(ex : -1).*

```
pour i de 0 à taille-1 faire
    si tab[i] = e alors
        retourner i
retourner NonTrouvé
```

$\Rightarrow$ *Complexité :* $O(\texttt{taille}) \cap \Omega(1)$.

Recherche d'un élément dans un tableau

La complexité précédente est trop élevée, surtout sachant que la recherche dans un tableau est une opération de base utilisée dans de nombreux algorithmes.

Pour aller plus vite, on peut utiliser les **tableaux triés** et la **dichotomie** (méthode «diviser pour régner») :

Retenir (Idée)

Si le tableau `tab` *est trié, pour tout indice* `i`*,*

- *les éléments* $e \leq$ `tab[i]` *sont d'indice* $\leq$ `i` *;*
- *les éléments* $e >$ `tab[i]` *sont d'indice* $>$ `i`*.*

On essaye avec `i` au milieu du tableau.

Recherche dichotomique

Algorithme (`RechDichoRec` : recherche dans un tableau trié)

- ***Entrée :*** *un tableau* ***trié*** `tab`, *un intervalle* $[\mathtt{min}, \mathtt{max}]$ *avec* $0 \leq \mathtt{min} \leq \mathtt{max} < \mathtt{taille}$ *et un élément* `e`.
- ***Sortie :*** `i` *tel que* $\mathtt{tab[i]} = \mathtt{e}$ *ou* `NonTrouvé` *(ex :* -1*).*

```
si min = max alors
   si tab[min] = e alors retourner min
   sinon                 retourner NonTrouvé
mid <- (min + max) / 2
si tab[mid] < e alors
    retourner RechDichoRec(tab, mid+1, max, e)
sinon
    retourner RechDichoRec(tab, min, mid, e)
```

⇒ *Complexité :* $\Theta(\log_2(\mathtt{taille}))$.

Recherche dichotomique itérative

Remarque : La recherche dichotomique est récursive terminale.

Algorithme (`RechDichoIt` recherche dichotomique itérative)

```
min <- 0;
max <- taille - 1
tant que min < max faire
    mid <- (min + max) / 2
    si tab[mid] < e alors
        min <- mid+1
    sinon
        max <- mid
si tab[min] = e alors retourner min
sinon                 retourner NonTrouvé
```

$\Rightarrow$ *Complexité :* $\Theta(\log_2(\texttt{taille}))$.

On peut stopper la recherche plus tôt si l'on a trouvé !

Algorithme (Recherche dichotomique variante)

```
min <- 0;
max <- taille - 1
tant que min < max faire
    mid <- (min + max) / 2
    si tab[mid] = e alors retourner mid
    sinon si tab[mid] < e alors
              min <- mid+1
           sinon
              max <- mid-1
si tab[min] = e alors retourner min
sinon                 retourner NonTrouvé
```

$\Rightarrow$ *Complexité :* $O(\log_2(\mathtt{taille})) \cap \Omega(1)$.

Autre application de la recherche dichotomique

- Jeu du nombre inconnu où l'on répond soit «plus grand» soit «plus petit» soit «gagné».
- Calcul d'une racine d'une fonction croissante (exemple : $\sqrt{x}$).
- Algorithme de pointage, de visée.
- Recherche de l'apparition d'un bug dans l'historique d'un programme (commandes `hg bisect`, `git-bisect`...)

  Exemple : 100 modifications, 10 minutes de tests pour chaque modifications. L'algorithme naif demande 1000 min≈16h40 au lieu de 70min≈1h10 par dichotomie.

Tableaux triés, algorithmes de tris

Insertion dans un tableau trié

Algorithme (`Insert`)

- ***Entrées :***
  - *Tableau* `tab`, `max_taille` *éléments alloués*
    éléments $0 \leq i <$ `taille` $<$ `max_taille` *initialisés.*
  - *un élément* `e`*.*
- ***Précondition :*** `tab` *est trié* ($\texttt{tab}[i] \leq \texttt{tab}[i+1]$)*.*
- ***Effet :*** `e` *ajouté à* `tab` *trié.*

```
i <- taille
tant que i > 0 et tab[i-1] > e faire
    tab[i] <- tab[i-1]
    i <- i-1
tab[i] <- e
taille <- taille + 1
```

$\Rightarrow$ *Complexité :* $O(\texttt{taille})$

Tri par insertion

Algorithme (`InsertSort`)

- ***Entrée :*** *Tableau* `T` *de taille* `taille`.
- ***Effet :*** `T` *trié.*

```
pour i de 1 à taille-1 faire
    e <- t[i]
    // Insérer e à sa place dans T[0], ..., T[i-1]
    j <- i
    tant que j > 0 et T[j-1] > e faire
        t[j] <- t[j-1]
        j <- j-1
    T[j] <- e
```

$\Rightarrow$ *Complexité :* $O(\mathtt{taille}^2)$

Algorithmes plus efficaces : Diviser pour régner

Diviser pour régner

Du latin « Divide ut imperes » (Machiavel)

On divise un problème de grande taille en plusieurs (deux) sous-problèmes analogues. Différentes stratégies :

1. **récursivité sur les données :** on sépare les données en deux parties arbitraires, puis on résout les sous-problèmes, pour enfin combiner les résultats.
2. **récursivité sur le résultat :** on effectue un pré-traitement pour bien découper les données, afin que, après avoir résolu les sous-problèmes, les sous-résultats se combinent d'eux-mêmes.

Diviser pour régner

Du latin « Divide ut imperes » (Machiavel)

On divise un problème de grande taille en plusieurs (deux) sous-problèmes analogues. Différentes stratégies :

1. **récursivité sur les données :** on sépare les données en deux parties arbitraires, puis on résout les sous-problèmes, pour enfin combiner les résultats.
2. **récursivité sur le résultat :** on effectue un pré-traitement pour bien découper les données, afin que, après avoir résolu les sous-problèmes, les sous-résultats se combinent d'eux-mêmes.

Récursivité sur les données :

On sépare les données en deux parties arbitraires, puis on résout les sous-problèmes, pour enfin combiner les résultats.

Comment obtenir un tableau trié, si l'on sait trier chaque moitié ?

Fusion de tableaux trié !

Récursivité sur les données :

On sépare les données en deux parties arbitraires, puis on résout les sous-problèmes, pour enfin combiner les résultats.

> Comment obtenir un tableau trié, si l'on sait trier chaque moitié ?

Fusion de tableaux trié !

Récursivité sur les données :

On sépare les données en deux parties arbitraires, puis on résout les sous-problèmes, pour enfin combiner les résultats.

> Comment obtenir un tableau trié, si l'on sait trier chaque moitié ?

Fusion de tableaux trié !

Fusion de deux tableaux triés

Algorithme (`Fusion de tableaux triée`)

- ***Entrée :*** *Tableaux* `T1`, `T2` *triés de taille* $t1$, $t2$,
  Tableau `T` *alloué de taille* $t = t1 + t2$
- ***Sortie :*** `T` *avec les contenus* `T1` *et* `T2` *trié*

```
i1 <- 0; i2 <- 0; i <- 0
tant que i1 < t1 et i2 < t2 faire
    si T1[i1] < T2[i2] alors
        T[i] <- T1[i1]; i++; i1++
    sinon
        T[i] <- T2[i2]; i++; i2++
si i1 < t1 alors
    tant que i1 < t1 faire
        T[i] <- T1[i1]; i++; i1++
sinon
    tant que i2 < t2 faire
        T[i] <- T2[i2]; i++; i2++
```

$\Rightarrow$ *Complexité :* $\Theta(t)$

Variantes et applications de la fusion

Opérations ensemblistes sur les tableaux trié :

- inclusion ;
- intersection, réunion ;
- différence et différence symétrique.

Algorithme (`Inclusion de tableau trié`)

- ***Entrée :*** *Tableaux* `T1`, `T2` *triés de taille* $t1$, $t2$,
- ***Sortie :*** `Vrai` *si* `T1` $\subset$ `T2`

```
i1 <- 0; i2 <- 0
tant que i1 < t1 et i2 < t2 faire
    si       T1[i1] = T2[i2] alors i1++; i2++
    sinon si T1[i1] > T2[i2] alors i2++
    sinon retourner Faux
retourner i1 = t1
```

Tri par fusion (MergeSort)

Algorithme (`TriFusion`)

- ***Entrée :*** *Tableaux* `T` *de taille* t, $0 \leq \min \leq \max < t$
  Tableau `Tmp` *alloué de taille* t
- ***Sortie :*** `T` *trié.*

```
si min <> max alors
    mid <- (min+max) / 2
    TriFusion(T, min, mid)
    TriFusion(T, mid+1, max)
    Fusion(T[min..mid], T[mid+1..max], Tmp)
    Copie de Tmp dans T[min..max]
```

$\Rightarrow$ *Complexité :* $\Theta(t \log(t))$

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si T est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par Fusion (1)

Pour simplifier, on suppose que la taille du tableau est une puissance de 2.

On note $c_k = d_n$ le nombre de copies d'éléments si `T` est de taille $n = 2^k$. On trouve

- $c_0 = d_1 = 0$
- $c_1 = d_2 = 2 + 2$ (fusion + copie)
- $c_2 = d_4 = 2c_1 + 4 + 4 = 16$ (rec + fusion + copie)
- $c_3 = d_8 = 2c_2 + 8 + 8 = 48$ (rec + fusion + copie)
- $c_4 = d_{16} = 2c_3 + 16 + 16 = 128$ (rec + fusion + copie)
- $c_5 = d_{32} = 2c_4 + 32 + 32 = 320$ (rec + fusion + copie)
- $c_6 = d_{64} = 2c_5 + 64 + 64 = 768$ (rec + fusion + copie)
- $c_7 = d_{128} = 2c_6 + 128 + 128 = 1792$ (rec + fusion + copie)

Complexité du tri par fusion (2)

Proposition

Le nombre c_k de copies d'éléments effectuées par le tri par fusion d'un tableau de $n = 2^k$ éléments vérifie :

$$c_0 = 0 \qquad et \qquad c_k = 2(c_{k-1} + 2^k)\,.$$

$$c_k = 2^{k+1}k = 2n\log_2(n)\,.$$

Preuve par récurrence :

- vrai pour $k = 0$
- si $c_k = 2^{k+1}k$ alors

$$c_{k+1} = 2(c_k + 2^{k+1}) = 2(2^{k+1}k + 2^{k+1}) = 2^{k+2}(k+1)$$

Complexité du tri par fusion (2)

Proposition

Le nombre c_k de copies d'éléments effectuées par le tri par fusion d'un tableau de $n = 2^k$ éléments vérifie :

$$c_0 = 0 \qquad \text{et} \qquad c_k = 2(c_{k-1} + 2^k)\,.$$

$$c_k = 2^{k+1}k = 2n\log_2(n)\,.$$

Preuve par récurrence :

- vrai pour $k = 0$
- si $c_k = 2^{k+1}k$ alors

$$c_{k+1} = 2(c_k + 2^{k+1}) = 2(2^{k+1}k + 2^{k+1}) = 2^{k+2}(k+1)$$

Complexité du tri par fusion (2)

Proposition

Le nombre c_k de copies d'éléments effectuées par le tri par fusion d'un tableau de $n = 2^k$ éléments vérifie :

$$c_0 = 0 \qquad et \qquad c_k = 2(c_{k-1} + 2^k)\,.$$

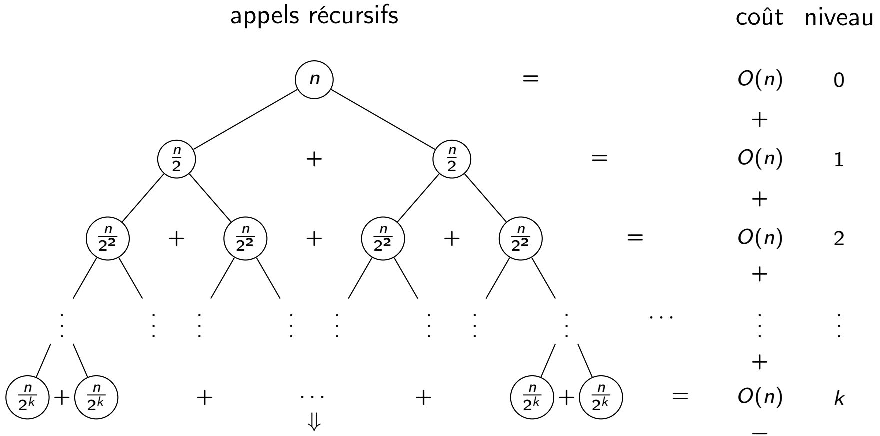
$$c_k = 2^{k+1}k = 2n\log_2(n)\,.$$

Preuve par récurrence :

- vrai pour $k = 0$
- si $c_k = 2^{k+1}k$ alors

$$c_{k+1} = 2(c_k + 2^{k+1}) = 2(2^{k+1}k + 2^{k+1}) = 2^{k+2}(k+1)$$

Complexité du tri par fusion (3)



$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow \boxed{O(n \cdot \log n)}$$

Tri par fusion / insertion

Nombre de copie d'éléments (cas le pire) :

k	0	1	2	3	4	5	6	...	k
n	1	2	4	8	16	32	64	...	2^k
insert	0	3	12	42	150	558	2142	...	$\frac{(n+1)(n+2)}{2} - 3$
fusion	0	4	16	48	128	320	768	...	$2^{k+1}k$

Retenir

Pour les petites valeurs de n, le tri par fusion fait plus de copies que le tri par insertion.

De plus, les appels récursifs induise des coûts supplémentaires. . .

Tri par fusion / insertion

Nombre de copie d'éléments (cas le pire) :

k	0	1	2	3	4	5	6	...	k
n	1	2	4	8	16	32	64	...	2^k
insert	0	3	12	42	150	558	2142	...	$\frac{(n-1)(n+4)}{2}$
fusion	0	4	16	48	128	320	768	...	$2^{k+1}k$

Retenir

Pour les petites valeurs de n, le tri par fusion fait plus de copies que le tri par insertion.

De plus, les appels récursifs induise des coûts supplémentaires. . .

Tri par fusion / insertion

Nombre de copie d'éléments (cas le pire) :

k	0	1	2	3	4	5	6	...	k
n	1	2	4	8	16	32	64	...	2^k
insert	0	3	12	42	150	558	2142	...	
fusion	0	4	16	48	128	320	768	...	$2^{k+1}k$

Retenir

Pour les petites valeurs de n, le tri par fusion fait plus de copies que le tri par insertion.

De plus, les appels récursifs induise des coûts supplémentaires...

Tri par fusion / insertion

Retenir (Tri mixte fusion / insertion)

Pour des petits tableaux le tri par insertion est plus rapide. Il vaut mieux l'utiliser comme cas de base de la récursion :

```
si max - min < SEUIL alors
    InsertSort(T[min..max])
sinon
    mid <- (min+max) / 2
    TriFusion(T, min, mid)
    TriFusion(T, mid+1, max)
    Fusion(T[min..mid], T[mid+1..max], Tmp)
    Copie de Tmp dans T
```

La valeur de `SEUIL` est déterminé expérimentalement.

Le tri par fusion n'est pas en place

On utilise un tableau de taille t supplémentaire (en fait en étant astucieux, un tableau de taille $\frac{t}{2}$ suffit).

Définition

On dit qu'un tri est en place, s'il utilise un emplacement mémoire constant ($O(1)$) en plus du tableau pour stocker ses éléments.

- Les tris par bulles et insertions sont en place ;
- Le tri par fusion n'est pas en place.

On aimerait bien avoir un tri en $n\log(n)$ en place...

Le tri par fusion n'est pas en place

On utilise un tableau de taille t supplémentaire (en fait en étant astucieux, un tableau de taille $\frac{t}{2}$ suffit).

Définition

On dit qu'un tri est en place, s'il utilise un emplacement mémoire constant ($O(1)$) en plus du tableau pour stocker ses éléments.

- Les tris par bulles et insertions sont en place ;
- Le tri par fusion n'est pas en place.

On aimerait bien avoir un tri en $n \log(n)$ en place...

Le tri par fusion n'est pas en place

On utilise un tableau de taille t supplémentaire (en fait en étant astucieux, un tableau de taille $\frac{t}{2}$ suffit).

Définition

On dit qu'un tri est en place, s'il utilise un emplacement mémoire constant ($O(1)$) en plus du tableau pour stocker ses éléments.

- Les tris par bulles et insertions sont en place ;
- Le tri par fusion n'est pas en place.

On aimerait bien avoir un tri en $n\log(n)$ en place...

Le tri par fusion n'est pas en place

On utilise un tableau de taille t supplémentaire (en fait en étant astucieux, un tableau de taille $\frac{t}{2}$ suffit).

Définition

On dit qu'un tri est en place, s'il utilise un emplacement mémoire constant ($O(1)$) en plus du tableau pour stocker ses éléments.

- Les tris par bulles et insertions sont en place ;
- Le tri par fusion n'est pas en place.

On aimerait bien avoir un tri en $n \log(n)$ en place...

Récursivité sur le résultat

On effectue un pré-traitement pour bien découper les données, puis à résoudre les sous-problèmes, pour que les sous-résultats se combinent d'eux-mêmes à la fin.

> Comment séparer les élements d'un tableau en deux pour que si l'on trie chaque partie le résultat soit triée ?

Partition d'un tableau !

Récursivité sur le résultat

On effectue un pré-traitement pour bien découper les données, puis à résoudre les sous-problèmes, pour que les sous-résultats se combinent d'eux-mêmes à la fin.

> Comment séparer les élements d'un tableau en deux pour que si l'on trie chaque partie le résultat soit triée ?

Partition d'un tableau !

Problème de partition d'un ensemble

Retenir

Soit un ensemble E et un prédicat $P(e)$ sur les élements de E :
Alors

$$E = PE \cup \overline{P}E \qquad \text{et} \qquad PE \cap \overline{P}E = \emptyset$$

où

- $PE := \{e \in E \mid P(e) \text{ est vrai } \}$
- $\overline{P}E := \{e \in E \mid P(e) \text{ est faux } \}$

*On dit que $(PE, \overline{P}E)$ est une **partition** de E.*

Exemple : $E = \{1, 2, \ldots 10\}$, $P(e) =$ «e est pair».

Partition et tris

Pour obtenir un algorithme de tri on peut :

1. effectuer une partition du tableau en plaçant les petits éléments au début et les grands à la fin.

2. trier chacune des parties indépendamment.

$\Rightarrow$ Tri rapide (QuickSort) !

Partition et tris

Pour obtenir un algorithme de tri on peut :

1. effectuer une partition du tableau en plaçant les petits éléments au début et les grands à la fin.
2. trier chacune des parties indépendamment.

$\Rightarrow$ Tri rapide (QuickSort) !

Partition et tris

Pour obtenir un algorithme de tri on peut :

1. effectuer une partition du tableau en plaçant les petits éléments au début et les grands à la fin.
2. trier chacune des parties indépendamment.

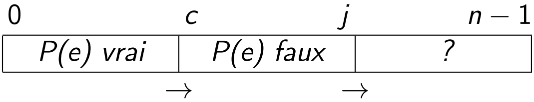
$\Rightarrow$ Tri rapide (QuickSort) !

Partition d'un tableau (1)

Algorithme (`PartitionnerPredicat`)

- ***Entrée :*** *Tableaux* `tab[taille]`*, prédicat* `P` *sur les éléments de* `tab`*.*
- ***Sortie :*** *entier* $0 \leq c <$ `taille`
- ***Effet :*** *échange les éléments de telle sorte que :*
  - *les élements du tableau qui vérifient P sont dans* $\texttt{tab}[0] \dots \texttt{tab}[c-1]$ *;*
  - *les élements du tableau qui ne vérifient pas P sont dans* $\texttt{tab}[c] \dots \texttt{tab}[\texttt{taille}-1]$*.*

Partition d'un tableau (2)



Algorithme (PartitionnerPredicat)

```
c <- 0
tant que c < taille et P(tab[c]) faire
    c <- c + 1
pour j de c+1 à taille-1 faire
    si P(tab[j]) alors
        échanger tab[c] et tab[j]
        c <- c+1
retourner c
```

$\Rightarrow$ *Complexité :* $\Theta(\texttt{taille})$.

Partition d'un tableau (2)

0	c	j $\quad\quad\quad\quad\quad\quad$ $n-1$
P(e) vrai	*P(e) faux*	*?*
	$\rightarrow$	$\rightarrow$

Algorithme (`PartitionnerPredicat`)

```
c <- 0
tant que c < taille et P(tab[c]) faire
    c <- c + 1
pour j de c+1 à taille-1 faire
    si P(tab[j]) alors
        échanger tab[c] et tab[j]
        c <- c+1
retourner c
```

$\Rightarrow$ *Complexité :* $\Theta(\texttt{taille})$.

Partition d'un tableau (2 bis)

`PartitionnerPredicat` fait beaucoup de déplacement d'éléments.

$0 \qquad\qquad c \qquad\qquad j \qquad\qquad n-1$

P(e) vrai	*P(e) faux*	*?*

$\qquad\qquad\quad \rightarrow \qquad\qquad\quad \rightarrow$

En particulier dans certains cas dégénérés, il effectue `taille` échange là où un seul est nécessaire.

Par exemple, si `P(tab`[0]`)` est faux, mais `P(tab`[i]`)` est vrai pour tous les $i > 0$ alors `PartitionnerPredicat` décale tout le tableau alors qu'il suffirait d'échanger le premier avec le dernier élément.

Partition d'un tableau (variante)

0	i	j	$n-1$
$P(e)$ *vrai*	?	$P(e)$ *faux*	
	$\rightarrow$	$\leftarrow$	

Algorithme (PartitionnerPredicatDeux)

```
i <- 0
j <- taille-1
tant que i < j faire
    tant que P(tab[i]) faire i <- i+1
    tant que non P(tab[j]) faire j <- j-1
    si i < j alors
        échanger tab[i] avec tab[j]
        i <- i+1; j <- j-1
 retourner i
```

Partition d'un tableau (variante)

$0 \qquad\qquad\qquad i \qquad\qquad\qquad j \qquad\qquad\qquad n-1$

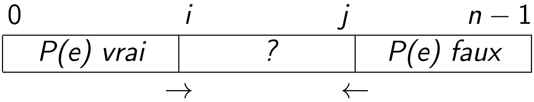
P(e) vrai	*?*	*P(e) faux*

$\rightarrow \qquad\qquad\qquad \leftarrow$

Algorithme (PartitionnerPredicatDeux)

```
i <- 0
j <- taille-1
tant que i < j faire
    tant que P(tab[i]) faire i <- i+1
    tant que non P(tab[j]) faire j <- j-1
    si i < j alors
        échanger tab[i] avec tab[j]
        i <- i+1; j <- j-1
 retourner i
```

Partition d'un tableau (variante)



Algorithme (PartitionnerPredicatDeux FAUX)

```
i <- 0
j <- taille-1
tant que i < j faire
    tant que P(tab[i]) faire i <- i+1
    tant que non P(tab[j]) faire j <- j-1
    si i < j alors
        échanger tab[i] avec tab[j]
        i <- i+1; j <- j-1
 retourner i
```

Partition d'un tableau (variante)

Attention ! L'algorithme `PartitionnerPredicatDeux` ne marche pas si tous ou aucun des éléments ne vérifient le prédicat P.

- En effet, si tous les éléments vérifient le prédicat, la ligne

  ```
  tant que P(tab[i]) faire i <- i+1
  ```

  sort du tableau.
- Inversement, si aucun des éléments ne vérifie le prédicat,

  ```
  tant que non P(tab[j]) faire j <- j-1
  ```

  sort du tableau.

Comment corriger l'algorithme ?

Partition d'un tableau (variante)

Attention ! L'algorithme `PartitionnerPredicatDeux` ne marche pas si tous ou aucun des éléments ne vérifient le prédicat P.

- En effet, si tous les éléments vérifient le prédicat, la ligne

  ```
  tant que P(tab[i]) faire i <- i+1
  ```

  sort du tableau.
- Inversement, si aucun des éléments ne vérifie le prédicat,

  ```
  tant que non P(tab[j]) faire j <- j-1
  ```

  sort du tableau.

Comment corriger l'algorithme ?

Partition d'un tableau (variante)

Attention ! L'algorithme `PartitionnerPredicatDeux` ne marche pas si tous ou aucun des éléments ne vérifient le prédicat P.

- En effet, si tous les éléments vérifient le prédicat, la ligne

  ```
  tant que P(tab[i]) faire i <- i+1
  ```

  sort du tableau.
- Inversement, si aucun des éléments ne vérifie le prédicat,

  ```
  tant que non P(tab[j]) faire j <- j-1
  ```

  sort du tableau.

Comment corriger l'algorithme ?

Partition d'un tableau (variante)

Algorithme (PartitionnerPredicatDeux)

```
i <- 0
j <- taille-1
tant que i < taille et P(tab[i]) faire i <- i+1
tant que j >= 0 et  non P(tab[j]) faire j <- j-1
tant que i < j faire
    tant que P(tab[i]) faire i <- i+1
    tant que non P(tab[j]) faire j <- j-1
    si i < j alors
        échanger tab[i] avec tab[j]
        i <- i+1; j <- j-1
 retourner i
```

Partition d'un tableau avec pivot

Adaptation du partitionnement au tri rapide

Algorithme (Spécification de `PartitionnerPivot`)

- ***Entrée :*** *un tableau* `tab`*, un intervalle* $[\texttt{min}, \texttt{max}]$ *avec* $0 \leq \texttt{min} \leq \texttt{max} < \texttt{taille}$
- ***Effet :*** *le tableau est réordonné de sorte que pour un certain* $\texttt{min} \leq c \leq \texttt{max}$*, on ait*
  - *si* $\texttt{min} \leq i < c$ *alors* $\texttt{tab}[i] \leq \texttt{tab}[c]$ *;*
  - *si* $c < i \leq \texttt{max}$ *alors* $\texttt{tab}[i] \geq \texttt{tab}[c]$*.*

min	*c*	*max*
$e \leq p$	p	$e \geq p$

- ***Sortie :*** *la position* c *du pivot*

Adaptation du partitionnement au tri rapide

Algorithme (`PartitionnerPivot`)

- ***Entrée :*** *un tableau* `tab`*, un intervalle* [`min`, `max`]
- ***Sortie :*** *la position du pivot*

```
pivot <- tab[max]
c <- min
tant que tab[c] < pivot faire c <- c+1
pour j de c+1 à max-1 faire
    si tab[j] < pivot alors
        échanger tab[c] et tab[j]
        c <- c+1
échanger tab[c] et tab[max]
retourner c
```

Partition d'un tableau avec pivot (variante)

min	i	j	$max-1$	max
$e \leq$ pivot	?	$e \geq$ pivot		pivot
	$\rightarrow$	$\leftarrow$		

Algorithme (PartitionnerPivotDeux)

```
pivot <- tab[max]
i <- min; j <- max-1
repeter
    tant que tab[i] < pivot faire i <- i+1
    tant que tab[j] > pivot faire j <- j-1
    si i < j alors
        échanger tab[i] avec tab[j]
        i <- i+1; j <- j-1
    sinon
        échanger tab[i] avec tab[max]
        retourner i
```

Partition d'un tableau avec pivot (variante)

min	i	j	$max - 1$	max
$e \leq$ pivot	?	$e \geq$ pivot		pivot
	$\rightarrow$	$\leftarrow$		

Algorithme (PartitionnerPivotDeux)

```
pivot <- tab[max]
i <- min; j <- max-1
repeter
    tant que tab[i] < pivot faire i <- i+1
    tant que tab[j] > pivot faire j <- j-1
    si i < j alors
        échanger tab[i] avec tab[j]
        i <- i+1; j <- j-1
    sinon
        échanger tab[i] avec tab[max]
        retourner i
```

Le tri rapide

Algorithme du tri rapide

Charles Antony Richard HOARE 1961.

0		$n-1$
$e \leq p$	p	$e \geq p$

Retenir (Idée)

- *choisir un élément p appelé **pivot** ;*
- *placer à gauche les éléments inférieur à p ;*
- *placer à droite les éléments supérieur à p ;*
- *trier récursivement la partie de droite et celle de gauche.*

Complexité du QuickSort

Retenir

La complexité du partitionnement d'un tableau de taille n est $\Theta(n)$.

Dans le cas le pire, c'est-à-dire

- si le pivot est le plus grand élément et donc est placé à la fin après le partitionnement,
- ou si le pivot est le plus petit élément et donc est placé au début après le partitionnement,

Retenir

Dans le cas le pire, la complexité du `QuickSort` *est $\Theta(n^2)$.*

Complexité du QuickSort

Retenir

La complexité du partitionnement d'un tableau de taille n est $\Theta(n)$.

Dans le cas le pire, c'est-à-dire

- si le pivot est le plus grand élément et donc est placé à la fin après le partitionnement,
- ou si le pivot est le plus petit élément et donc est placé au début après le partitionnement,

Retenir

Dans le cas le pire, la complexité du `QuickSort` *est $\Theta(n^2)$.*

Complexité du QuickSort

Retenir

La complexité du partitionnement d'un tableau de taille n est $\Theta(n)$.

Dans le cas le pire, c'est-à-dire

- si le pivot est le plus grand élément et donc est placé à la fin après le partitionnement,
- ou si le pivot est le plus petit élément et donc est placé au début après le partitionnement,

Retenir

Dans le cas le pire, la complexité du `QuickSort` *est $\Theta(n^2)$.*

Influence du choix du pivot

Retenir

Pour avoir des bonnes performances dans `QuickSort`*, il est important que le pivot soit bien choisi.*

En pratique, sur une permutation au hasard, le pivot coupe le tableau à peu près en deux :

Retenir

En moyenne, le `QuickSort` *à une complexité de* $\Theta(n \log(n))$.

Influence du choix du pivot

Retenir

Pour avoir des bonnes performances dans `QuickSort`, *il est important que le pivot soit bien choisi.*

En pratique, sur une permutation au hasard, le pivot coupe le tableau à peu près en deux :

Retenir

En moyenne, le `QuickSort` *à une complexité de* $\Theta(n \log(n))$.

Influence du choix du pivot

Retenir

Pour avoir des bonnes performances dans `QuickSort`*, il est important que le pivot soit bien choisi.*

En pratique, sur une permutation au hasard, le pivot coupe le tableau à peu près en deux :

Retenir

En moyenne, le `QuickSort` *à une complexité de* $\Theta(n\log(n))$.

Influence du choix du pivot, en pratique

Retenir

Pour avoir des bonnes performances dans `QuickSort`, *il est important que le pivot soit bien choisi.*

Plusieurs stratégies sont proposées :

- choisir un élément au milieu du tableau ;
- choisir un élément au hasard dans le tableau ;
- choisir la médiane du premier, de celui du milieu et du dernier.

Bilan

Avantages du `QuickSort` :

- en place, avec une petite utilisation de pile pour les appels récursifs. ;
- bonne complexité en moyenne $n \log(n)$;
- boucle intérieur très courte ;
- très bien compris théoriquement.

Désavantages du `QuickSort` :

- récursif, difficile à implanter dans les environnements où la récursion n'est pas possible ;
- mauvaise complexité dans le cas le pire n^2 ;
- fragile : beaucoup de possibilité pour faire une erreur subtile en l'implantant.

Peut-on faire mieux que $O(n\log(n))$?

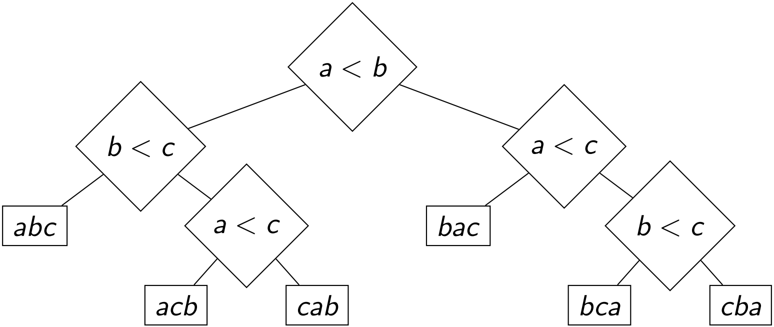
La complexité $O(n\log(n))$ est optimale !

Théorème

Pour tout algorithme de tri fonctionnant en comparant les données la complexité dans le cas le pire est $\Omega(n\log(n))$ (au moins $n\log(n)$).

Preuve : Arbres de décision binaire + formule de Stirling.

Arbre de décision binaire



Proposition

La profondeur de l'arbre est au moins $\lceil \log_2(n) \rceil$ où n est le nombre de feuille.

Permutations et formule de Stirling

Il y a $n!$ manières de permuter un tableaux de n nombres.

James Stirling (1692–1770) :

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left[1 + \frac{1}{12n} + \frac{1}{188n^2} + \frac{139}{51840n^3} + \cdots\right]$$

On en déduit

$$\log(n!) = n \cdot \log(n) - n + O(n)$$

Permutations et formule de Stirling

Il y a $n!$ manières de permuter un tableaux de n nombres.

James Stirling (1692–1770) :

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left[1 + \frac{1}{12n} + \frac{1}{188n^2} + \frac{139}{51840n^3} + \cdots\right]$$

On en déduit

$$\log(n!) = n \cdot \log(n) - n + O(n)$$

Permutations et formule de Stirling

Il y a $n!$ manières de permuter un tableaux de n nombres.

James Stirling (1692–1770) :

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left[1 + \frac{1}{12n} + \frac{1}{188n^2} + \frac{139}{51840n^3} + \cdots\right]$$

On en déduit

$$\log(n!) = n \cdot \log(n) - n + O(n)$$