Algorithmes de tri interne (5) [tr]
Méthodes par partitions

Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner

Unisciel algoprog Version 21 mai 2018

Table des matières

1 Tri par fusion **3**
1.1 Principe du tri par fusion . . . 3
1.2 Algorithme du tri par fusion . . . 3
1.3 Algorithme fusionner . . . 4
1.4 Complexité du tri par fusion . . . 5

2 Tri rapide **6**
2.1 Principe du tri rapide . . . 6
2.2 Algorithme du tri rapide . . . 7
2.3 Principe de l'algorithme de partitionnement . . . 7
2.4 De la bonne écriture du partitionnement . . . 8
2.5 Complexité du tri rapide . . . 9
2.6 Conclusion . . . 10

3 Complexités du tri rapide **11**
3.1 Pire cas . . . 11
3.2 Meilleur cas . . . 11
3.3 Complexité en moyenne . . . 12

4 Rendre le tri rapide efficace **14**
4.1 Amélioration de la complexité spatiale . . . 14
4.2 Choix du pivot . . . 15
4.3 Cas des petits-tableaux . . . 15

Algorithmes de tri interne (5)

Méthodes par partitions

Les **méthodes par partitions** sont des algorithmes récursifs basés sur le paradigme :

- **Diviser** le tableau en deux sous-tableaux
- **Régner** en triant récursivement les sous-tableaux
- **Combiner** les sous-tableaux

Il existe deux tris par partitions fondamentaux :

- Le « tri rapide » : l'intelligence du tri se trouve au niveau de la *division* du tableau (par partitionnement).
- Le « tri fusion » : l'intelligence du tri se trouve au niveau de la *combinaison* des deux sous-tableaux.

1 Tri par fusion

Nom anglais : *merge sort*

Propriétés : tri interne et externe, ne trie pas sur place, stable

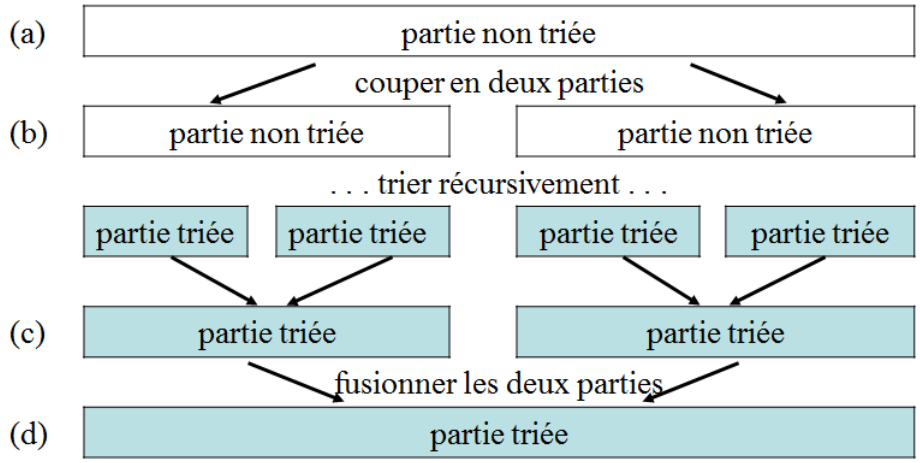
Visualisation : http://www.sorting-algorithms.com/merge-sort

1.1 Principe du tri par fusion

Le **tri par fusion** reprend l'idée du tri par insertion en lui appliquant le principe d'*équilibrage*. Plutôt que de séparer le tableau en un élément A[n]| et un sous-tableau \lstinlineA[1..n-1]|, on le coupe en deux parties à peu près égales.

Pour trier complètement le tableau, il suffit alors de :

- Diviser le tableau de taille n en deux sous-tableaux de taille $n/2$.
- Trier récursivement les deux sous-tableaux.
- Fusionner les deux sous-tableaux pour produire le tableau complet trié.



1.2 Algorithme du tri par fusion

Algorithme trFusion

```
Action trFusion ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  trFusionRec ( A , 1 , n )
Fin
Action trFusionRec ( DR A : Element [ NMAX ] ; g : Entier ; h : Entier )
Début
  |  Si ( g < h ) Alors
  |     |  milieu <- DivEnt ( g + h , 2 ) // Ou une Autre valeur entre g Et h
  |     |  trFusionRec ( A , g , milieu )
  |     |  trFusionRec ( A , milieu + 1 , h )
  |     |  fusionner ( A , g , milieu , h )
  |  FinSi
Fin
```

1.3 Algorithme fusionner

Le principe de cette fusion est simple : à chaque étape, on compare les éléments minimaux des deux sous-tableaux triés, le plus petit des deux étant l'élément minimal de l'ensemble on le met de côté et on recommence. On conçoit ainsi un algorithme `fusionner` qui prend en entrée un tableau `A` et trois entiers, `g`, `m` et `h`, tels que $g \leq m < h$ et tels que les tableaux `A[g..m]` et `A[m+1..h]` soient triés.

Procédure fusionner

```
Action fusionner ( DR A : Element [ NMAX ] ; g , m , h : Entier )
Variable C : Element [ NMAX ]
Début
  |  k <- 1
  |  TantQue ix <= m Et jx <= h Faire
  |    |  Si ( C [ ix ] < C [ jx ] ) Alors
  |    |    |  C [ k ] <- A [ ix ]
  |    |    |  ix <- ix + 1
  |    |  Sinon
  |    |    |  C [ k ] <- A [ jx ]
  |    |    |  jx <- jx + 1
  |    |  FinSi
  |    |  k <- k + 1
  |  FinTantQue
  |  TantQue ix <= m Faire
  |    |  C [ k ] <- A [ ix ]
  |    |  ix <- ix + 1
  |    |  k <- k + 1
  |  FinTantQue
  |  TantQue jx <= h Faire
  |    |  C [ k ] <- A [ jx ]
  |    |  jx <- jx + 1
  |    |  k <- k + 1
  |  FinTantQue
  |  n <- h - g + 1
  |  Pour k <- 1 à n Faire
  |    |  A [ g + k - 1 ] <- C [ k ]
  |  FinPour
Fin
```

Complexité

Étudions les différentes étapes de l'algorithme :

- Les initialisations ont un coût constant $\Theta(1)$.
- La boucle `TantQue` de fusion s'exécute au plus $h - g$ fois, chacune de ses itérations étant de coût constant, d'où un coût total en $O(h - g)$.
- Les deux boucles `TantQue` complétant `C` ont une complexité respective au pire de $h - g + 1$ et de $h - g$, ces deux complexités étant en $O(h - g)$.
- La recopie finale coûte $\Theta(h - g + 1)$.

Par conséquent, l'algorithme de fusion a une complexité en $\Theta(h - g)$.

1.4 Complexité du tri par fusion

Complexité temporelle

Pour déterminer la formule de récurrence qui nous donnera la complexité de l'algorithme `trFusion`, étudions les trois phases de cet algorithme « diviser pour régner » :

1. **Diviser :** cette étape se réduit au calcul du milieu de l'intervalle $[g..h]$
2. **Régner :** l'algorithme résout récursivement deux sous-problèmes de tailles respectives $n/2$.
3. **Combiner :** la complexité de cette étape est celle de l'algorithme de fusion qui est de $\Theta(n)$ pour la construction d'un tableau solution de taille n.

Par conséquent, la complexité du tri par fusion est donnée par la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sinon} \end{cases}$$

Pour déterminer la complexité du tri par fusion, nous utilisons le Master-Théorème : ici $a = 2$ et $b = 2$ donc $\log_b a = 1$, et nous nous trouvons dans le cas 2 du théorème : $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$. Par conséquent :

$$T(n) = \Theta(n \lg n)$$

Complexité spatiale

La complexité spatiale est en $O(n)$ pour l'opération `fusionner`. Ce type de tri est donc surtout intéressant pour des structures de données **autres** que le tableau.

2 Tri rapide

Nom anglais : *quick sort*

Propriétés : tri interne, sur place, non stable

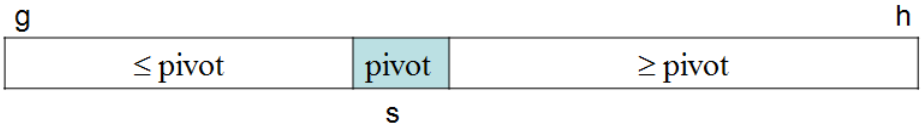
Visualisation : http://www.sorting-algorithms.com/quick-sort

2.1 Principe du tri rapide

Le **tri rapide**, proposé en 1962 par C.A.R. Hoare, est une application systématique des principes « diviser pour régner » et « équilibrage » au tri par échanges. L'idée est d'éviter la dispersion entraînée par le tri bulles en divisant l'ensemble des éléments à trier en deux parties homogènes.

Méthode

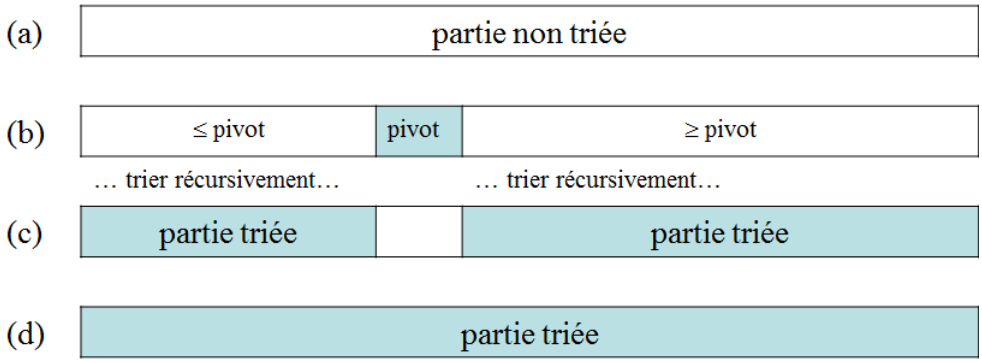
Supposons pour cela qu'un algorithme `partitionner(A,g,h)` puisse trouver un élément « pivot » dans le tableau `A[g..h]` et réorganise le tableau de telle sorte que l'élément `pivot` se retrouve à un certain indice `s`, tous les éléments inférieurs ou égaux à `pivot` étant placés à gauche et tous les éléments supérieurs ou égaux à `pivot` à sa droite :



Pour trier complètement le tableau, il suffit alors de :

- ne plus toucher au pivot, qui est à sa place
- trier (récursivement) le sous-tableau de gauche `A[g..s-1]`
- trier (récursivement) le sous-tableau de droite `A[s+1..h]`

Initialement on considère le tableau entier `A[1..n]`.



2.2 Algorithme du tri rapide

Algorithme trRapide

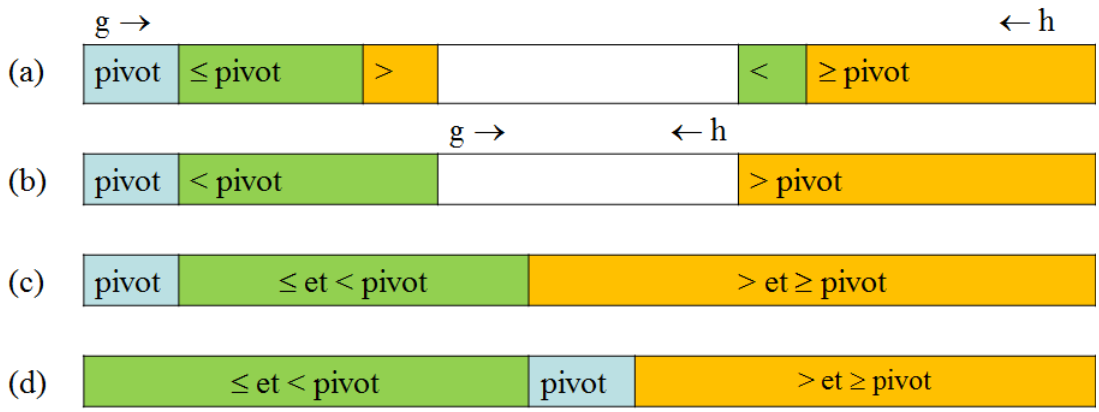
```
Action trRapide ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  trRapideRec ( A , 1 , n )
Fin
Action trRapideRec ( DR A : Element [ NMAX ] ; g , h : Entier )
Début
  |  Si ( g < h ) Alors
  |     |  s <- partitionner ( A , g , h )
  |     |  trRapideRec ( A , g , s - 1 )
  |     |  trRapideRec ( A , s + 1 , h )
  |  FinSi
Fin
```

Remarque
L'algorithme est valide quel que soit l'ordre des deux appels récursifs, propriété que nous utiliserons pour améliorer le tri rapide.

2.3 Principe de l'algorithme de partitionnement

On peut partitionner le tableau `A[g..h]` autour d'un élément pivot en considérant chaque élément une fois ou deux seulement, si l'on « brûle la chandelle par les deux bouts ». La recherche de l'élément pivot fonctionne comme suit :

- Prendre pour pivot un élément de `A[g..h]`, par exemple celui en `g`.
- Explorer le tableau à partir de la gauche et à partir de la droite, jusqu'à trouver, à gauche, un élément de clé supérieure à celle du pivot, et à droite un élément de clé inférieure : (a).
- On peut alors échanger ces éléments (et supprimer une inversion). On continue la double exploration, à partir des positions atteintes à gauche et à droite : (b).
- Lorsque les indices se rejoignent, le tableau est partitionné et leur valeur commune est `s` : (c).
- Terminer par l'échange de `A[g]` (le pivot) et `A[s]` : (d).



2.4 De la bonne écriture du partitionnement

Il existe de nombreuses versions de `partitionner`. Elles sont cependant loin d'être équivalentes du point de vue de la simplicité et de l'efficacité. Pour ce dernier critère, deux buts seront recherchés :

1. Essayer d'accélérer les boucles internes.
2. Préserver le caractère « aléatoire » du tableau, c.-à-d. ne pas introduire par inadvertance un certain ordre, qui serait préjudiciable au rendement global de tri rapide. L'algorithme `partitionner` doit se garder de la tentation de trier !

C'est ce dernier critère qui est le plus difficile à assurer.

Méthode de Sedgewick

La méthode suivante due à SEDGEWICK semble meilleure que les méthodes habituellement publiées :

1. Placer le pivot à gauche en `g` par un échange éventuel avec l'élément en `g`.
2. Partitionner le sous-tableau `A[g+1..h]` autour de la valeur-pivot `A[g]| en laissant \lstinline`A[g]@ à sa place : on obtient une séparation autour d'une position intermédiaire `s = g = h`.
3. Échanger les éléments en `g` et `s`. La valeur retournée par l'algorithme est `s`.

Fonction partitionner

```
Fonction partitionner ( DR A : Element [ NMAX ] ; g0 , h0 : Entier ) : Entier
Variable g , h : Entier
Variable pivot : Element
Début
  |  g <- g0
  |  h <- h0
  |  pivot <- A [ g ]
  |  TantQue ( g <> h ) Faire
  |    |  TantQue ( g < h Et A [ g ] <= pivot ) Faire
  |    |    |  g <- g + 1
  |    |  FinTantQue
  |    |  TantQue ( g < h Et pivot <= A [ h ] t ) Faire
  |    |    |  h <- h - 1
  |    |  FinTantQue
  |    |  Si ( g < h ) Alors
  |    |    |  permuterTab ( A , g , h )
  |    |    |  g <- g + 1
  |    |    |  h <- h - 1
  |    |  FinSi
  |  FinTantQue
  |  Si ( g <> g0 ) Alors
  |    |  permuterTab ( A , g0 , g )
  |  FinSi
  |  Retourner ( g )
Fin
```

Complexité
Cet algorithme est clairement de complexité `O(h-g)` : en effet, la boucle externe `TantQue` examine chaque élément de `A[g+1..h]` au plus une fois, et le reste demande un temps fixe.

2.5 Complexité du tri rapide

L'étude mathématique est en section suivante @[Complexités du tri rapide].

Complexité
Pour calculer la complexité de l'algorithme du tri rapide on part de la définition récursive du tri rapide. Soit $C(T)$ le nombre de comparaisons pour trier un tableau T et soit $P(T)$ le nombre de comparaisons pour partitionner T selon le pivot. Si la partition répartit les éléments dans deux tableaux T_1 et T_2, on obtient :

$$C(T) = P(T) + C(T_1) + C(T_2)$$

Si T est de taille n, le nombre de comparaisons pour réaliser la partition ne peut être plus petit que $n-1$ (il faut comparer chacun des autres éléments au pivot pour décider du sous-tableau dans lequel il doit se trouver). On supposera donc que $P(T) = n-1$. Dans l'algorithme `partitionner`, le nombre de comparaisons pour partitionner un tableau de taille n peut aller jusqu'à $n+1$ mais cela ne change pas l'ordre de grandeur du résultat final.

Mais pour ce qui est de la place du pivot, et donc des tailles de T_1 et T_2, la situation est très variable :

1. Si le pivot est le plus petit (ou plus grand) élément du tableau, l'un des deux tableaux de la partition aura une taille nulle et l'autre aura une taille de $n-1$.
2. Si le pivot est l'élément médian, les deux tableaux T_1 et T_2 auront pour taille (environ) la moitié de n.
3. En général si la place du pivot est k, il reste à traiter récursivement un tableau de taille $k-1$ et un tableau de taille $n-k$.

Complexité au pire
Le premier cas est le pire des cas possibles (et il se produit lorsque le tableau initial est trié – en ordre croissant ou décroissant). Dans ce cas, pour un tableau de taille n, le nombre de comparaison pour trier est :

$$C_{max}(n) = n - 1 + C_{max}(n-1)$$

Et avec le cas initial $C_{max}(1) = 0$, la récurrence se résout en :

$$C_{max}(n) = n(n-1)/2 = O(n^2)$$

Complexité au meilleur

Le second cas est le meilleur des cas possibles. En effet, s'il se répète à toutes les étapes récursives, on a finalement pour le nombre minimal de comparaisons pour trier un tableau de taille n :

$$C_{min}(n) = n - 1 + C_{min}(\lfloor n/2 \rfloor) + C_{min}(\lceil n/2 \rceil)$$

Et avec le cas initial $C_{min}(1) = 0$, cette récurrence se résout en :

$$C_{min}(n) \sim n \lg n = \Omega(n \lg n)$$

Complexité en moyenne

Pour traiter la complexité en moyenne, il faut considérer toutes les configurations possibles de placement du pivot, et faire la moyenne arithmétique des nombres de comparaisons :

$$C_{moy}(n) = n - 1 + \frac{1}{k} \sum_{k=1}^{n} (C_{moy}(k-1) + C_{moy}(n-k))$$

La résolution de cette récurrence donne :

$$C_{moy}(n) \sim 2n \lg n = \Theta(n \lg n)$$

2.6 Conclusion

Ainsi la complexité moyenne du tri rapide est en $\Theta(n \lg n)$. Par exemple pour un tableau d'un million d'éléments, le tri par insertion nécessite de un milliard de comparaisons en moyenne alors que le tri rapide n'en fera en moyenne qu'une vingtaine de millions ($10^6 \approx 2^{20}$ donc $\lg 10^6 \approx 20$).

Toutefois dans le pire cas, le tri rapide est en $O(n^2)$. Et cela est d'autant plus gênant que ce cas apparaît pour des tableaux triés. Il existe cependant plusieurs parades pour éviter le cas le pire, en faisant en sorte que le pivot se place toujours dans la région médiane du tableau pour avoir un tri en temps proportionnel à $n \lg n$ (voir plus loin @[Rendre le tri rapide efficace]).

Le tri rapide est considéré en général comme le tri le plus efficace et c'est par exemple le tri utilisé dans la plupart des systèmes d'exploitation.

3 Complexités du tri rapide

Cette section réalise l'étude mathématique du tri rapide.

3.1 Pire cas

Le pire cas intervient quand le partitionnement produit une région à $n-1$ éléments et une à un élément, comme nous le montrerons ci-après. Comme le partitionnement coûte $\Theta(n)$ et que $T(1) = \Theta(1)$, la récurrence pour le temps d'exécution est :

$$T(n) = T(n-1) + \Theta(n)$$

D'où par sommation :

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

Pour montrer que cette configuration est bien le pire cas, montrons que dans tous les cas $T(n) = O(n^2)$, c.-à-d. qu'il existe une constante c telle que $T(n) \leq c \times n^2$. Si $T(n)$ est la complexité au pire :

$$T(n) = \max_{1 \leq s \leq n-1} \left(T(s) + T(n-s)\right) + \Theta(n)$$

où le paramètre s est dans l'intervalle $[1..n-1]$ puisque la procédure `partitionner` génère deux régions de tailles chacune au moins égale à un. D'où :

$$T(n) \leq \max_{1 \leq s \leq n-1} \left(cs^2 + c(n-s)^2\right) + \Theta(n)$$

Or l'expression $s^2 + (n-s)^2$ atteint son maximum à l'une des extrémités de l'intervalle (dérivée négative puis positive). D'où

$$T \max_{1 \leq s \leq n-1} \left(s^2 + (n-s)^2\right) = 1^2 + (n-1)^2 = n^2 - 2(n-1)$$

et

$$T(n) \leq cn^2 - 2c(n-1) + \Theta(n) \leq cn^2$$

puisque l'on peut choisir la constante c assez grande pour que le terme $2c(n-1)$ domine le terme $\Theta(n)$. Du coup, le temps d'exécution du tri rapide (dans le pire cas) est $O(n^2)$.

3.2 Meilleur cas

Le meilleur cas apparaît quand la procédure de partitionnement produit deux régions de taille $n/2$. La récurrence est alors :

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

ce qui, d'après le cas 2 du Master-Théorème nous donne :

$$T(n) = \Theta(n \log n)$$

3.3 Complexité en moyenne

On suppose que le tableau `A` ne contient pas deux fois le même élément.

Version stochastique du tri rapide

Un algorithme est dit **stochastique** si son comportement est déterminé non seulement par son entrée mais aussi par les valeurs produites par un **générateur de nombres aléatoires**. On modifie la procédure `partitionner` pour qu'elle est un comportement stochastique en utilisant une fonction `hasard(a,b)` qui renvoie de manière équiprobable un entier de l'intervalle `[a..b]`.

Fonction partitionnerStochastique

```
Fonction partitionnerStochastique ( DR A : Element [ NMAX ] ; g , h : Entier ) : Entier
Début
  |  s <- hasard ( g , h )
  |  permuterTab ( A , g , s )
  |  Retourner partitionner ( A , g , h )
Fin
```

Le but ici est de faciliter l'analyse de l'algorithme et de minimiser l'influence des configurations pathologiques.

Analyse du partitionnement

La valeur `s` renvoyée par `partitionner` ne dépend que du rang de `x = A[g]` parmi les éléments de `A[g..h]` (le **rang** d'un nombre dans un ensemble étant le nombre d'éléments qui lui sont inférieurs ou égaux). Du fait de l'encapsulation de `partitionner` dans `partitionnerStochastique` et de l'interversion de `A[g]` et d'un élément aléatoire de `A[g..h]`, on a $rang(x) = i$ pour $i = 1, 2, ..., n$ avec une probabilité $\frac{1}{n}$ en posant $n = h - g + 1$ (c'est le nombre d'éléments de l'intervalle $[g..h]$).

Récurrence pour le cas moyen

Vu ce qui précède, le temps moyen requis pour le tri d'un tableau de n éléments vaut donc :

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{s=1}^{n-1}(T(s) + T(n-s))\right) + \Theta(n)$$

Comme $T(1) = \Theta(1)$ et $T(n-1) = O(n^2)$ (vue l'étude du pire cas), on a :

$$\frac{1}{n}(T(1) + T(n-1)) = \frac{1}{n}\left(\Theta(1) + O(n^2)\right) = O(n)$$

et ce terme peut être absorbé par le terme $\Theta(n)$ de la formule. Ainsi :

$$T(n) = \frac{1}{n}\sum_{s=1}^{n-1}(T(s) + T(n-s)) + \Theta(n) = \frac{2}{n}\sum_{s=1}^{n-1}T(s) + \Theta(n)$$

Résolution de la récurrence

On suppose par induction qu'il existe des constantes strictement positives a et b telles que

$$T(n) \leq a\, n \lg n + b$$

Si a et b sont tels que l'hypothèse est vraie pour $n = 1$ alors, si l'on suppose l'hypothèse vraie jusqu'au rang $n-1$, on a :

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{s=1}^{n-1} T(s) + \Theta(n) \\ &\leq \frac{2}{n}\sum_{k=1}^{n-1}(ak \lg k + b) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + \Theta(n) \end{aligned}$$

Si l'on sait que (cf. [Cormen-A1]) :

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$$

on obtient :

$$\begin{aligned} T(n) &\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\ &\leq an \lg n - \frac{a}{4}n + 2b + \Theta(n) \\ &= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \end{aligned}$$

d'où

$$T(n) \leq an \log n + b$$

puisque l'on peut choisir a suffisamment grand pour que $\frac{a}{4}n$ domine $\Theta(n) + b$. On en conclut que le temps d'exécution moyen du tri rapide est $\Theta(n \lg n)$.

4 Rendre le tri rapide efficace

Cette section discute d'améliorations pour rendre le tri rapide efficace.

4.1 Amélioration de la complexité spatiale

Dans l'algorithme du tri rapide, nous avons signalé que l'ordre relatif des deux appels récursifs était en principe quelconque. Considérons cependant une mise en oeuvre pratique de la récursion : les appels non encore satisfaits seront empilés. Dans un cas particulièrement défavorable, les partitions successives partagent le tableau en deux sous-tableaux l'un de taille vide et l'autre de taille $h - g$. A partir du tableau initial, de taille n, la profondeur des imbrications des appels récursifs pourra donc atteindre n.

Il existe un remède simple à ce phénomène : commencer systématiquement par le sous-tableau de plus petite taille. Celle-ci sera donc inférieure à la moitié de la taille du sous-tableau précédent : en d'autres termes, le nombre maximal d'éléments qui peuvent être présents dans la pile au même moment, $P(n)$, qui est aussi la profondeur maximale des appels récursifs, vérifiera :

$$P(n) < 1 + P\left(\lfloor n/2 \rfloor\right)$$

Comme $P(0) = 0$, on obtiendra $P(n) < \lg n + 1$. Cette méthode permet de ramener la complexité spatiale de tri rapide à $O(\lg n)$. Comme $\lg 10^6 \approx 20$, on peut considérer, pour toutes les applications pratiques, qu'une zone fixe de 20 éléments suffira à la représentation de la pile.

La modification de l'algorithme assurant cette propriété est alors :

Procédure trRapideRecCS

```
Action trRapideRecCS ( DR A : Element [ NMAX ] ; g , h : Entier )
Début
  |  Si ( g < h ) Alors
  |     |  s <- partitionner ( A , g , h )
  |     |  Si ( s - g <= h - s ) Alors
  |     |     |  trRapideCS ( A , g , s - 1 )
  |     |     |  trRapideCS ( A , s + 1 , h )
  |     |  Sinon
  |     |     |  trRapideCS ( A , s + 1 , h )
  |     |     |  trRapideCS ( A , g , s - 1 )
  |     |  FinSi
  |  FinSi
Fin
```

Complexité spatiale

La complexité spatiale étant ainsi ramenée à $O(\lg n)$ au maximum.

4.2 Choix du pivot

On montre (voir [Knuth-AL1] ou [Sedgewick]) que la probabilité du cas le plus défavorable, en $O(n^2)$, est assez faible.

Ce résultat n'est cependant pas une consolation car le cas le plus défavorable est obtenu en particulier lorsque les données sont **déjà triées** initialement (ou dans l'ordre inverse). C'est le **paradoxe** de tri rapide : contrairement au tri par insertion ou même au tri bulles, il perd ses moyens face à un tableau partiellement ordonné. L'inconvénient est grave : le tri de données « presque » triées initialement n'est pas rare dans certains types d'application.

Pour utiliser le tri rapide en pratique, il faut donc obligatoirement imposer à l'algorithme `partitionner` des contraintes sur le choix du pivot, diminuant la probabilité que les deux longueurs de « segments » ne soient pas trop inégales. Deux stratégies sont possibles :

- « Tirer au hasard » à chaque exécution de partition la valeur de l'indice pivot parmi `[g..h]`.
- Éviter le cas le plus catastrophique en choisissant comme pivot dans `partitionner` l'élément médian d'un petit échantillon du tableau – c.-à-d. l'élément tel que l'échantillon considéré contienne (environ) autant de clés supérieures et de clés inférieures à la sienne. On espère ainsi, si l'échantillon est représentatif, obtenir un pivot qui se retrouvera à peu près au milieu du tableau. Le choix le plus simple pour l'échantillon, et sans doute le meilleur en général, est celui d'un échantillon comprenant les trois éléments d'indices `g`, `h` et `DivEnt(g+h,2)`.

L'emploi de l'une ou l'autre de ces méthodes diminue considérablement la probabilité du cas « catastrophique » en $O(n^2)$. Il importe cependant de signaler qu'elle n'en écarte pas définitivement la possibilité : **le tri rapide peut toujours dégénérer**.

Recherche du k-ème élément
La généralisation du problème de la recherche d'un élément médian, à savoir la recherche du k-ème élément, par ordre des clés, dans un tableau de n éléments, est traité par une variante du tri rapide ; voir exercice.

4.3 Cas des petits-tableaux

Pour devenir un algorithme réellement efficace, le tri rapide demande encore une amélioration. Dans la version présentée, il est évident que la gestion de la récursion et le choix du pivot devient trop lourde pour les petits sous-tableaux. **Le tri rapide n'est pas adapté aux petits tableaux** : il convient d'« arrêter » la récursion dès que la taille des sous-tableaux devient inférieure à une certaine constante `seuil`. On utilisera alors une méthode de tri simple dont l'efficacité est améliorée lorsque les données sont partiellement triées, par exemple le tri par insertion simple.

Une étude théorique de Knuth conduit à la valeur optimale `seuil = 9` (calculée à partir d'un modèle de machine assez typique des ordinateurs habituels). Dans les tests pratiques (voir [herve-R1]), les valeurs `seuil = 8` à 20 donnent généralement de bons résultats, l'optimum étant entre 14 et 16.

Avec cette nouvelle modification, l'algorithme devient :

Procédure trRapideRecOptm

```
Constante seuil <- ... // une valeur entre 8 Et 20
Action trRapideRecOptm ( DR A : Element [ NMAX ] ; g , h : Entier )
Début
  |  Si ( h - g > seuil ) Alors
  |    |  s <- partitionner ( A , g , h )
  |    |  Si ( s - g <= h - s ) Alors
  |    |    |  trRapideRecOptm ( A , g , s - 1 )
  |    |    |  trRapideRecOptm ( A , s + 1 , h )
  |    |  Sinon
  |    |    |  trRapideRecOptm ( A , s + 1 , h )
  |    |    |  trRapideRecOptm ( A , g , s - 1 )
  |    |  FinSi
  |  Sinon
  |    |  trInsertion ( A , g , h )
  |  FinSi
Fin
```