Chapitre 2

Algorithmes de tri

2.1 Premier exemple de tri d'une liste d'entiers : le tri par insertion

2.1.1 Problématique du tri

Une tri sur une séquence linéaire de nombres entiers (une liste d'entiers) se définit comme suit :

Entrée : la séquence de n nombres, a_1, ..., a_n.

Sortie : une permutation, a'_1, ..., a'_n, de la séquence d'entrée, telle que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

2.1.2 Principe du tri par insertion

De manière répétée, on retire un nombre de la séquence d'entrée et on l'insère à la bonne place dans la séquence des nombres déjà triés (ce principe est le même que celui utilisé pour trier une poignée de cartes).

2.1.3 Algorithme

TRI-INSERTION

Pour $j \leftarrow 2$ **à** n **faire**
  clé $\leftarrow$ A[j] — *On retire un nombre de la séquence d'entrée*
  $i \leftarrow j-1$ — *Les $j-1$ premiers éléments de A sont déjà triés.*
  tant que $i > 0$ **et** A[i] $>$ clé **faire** — *Tant que l'on n'est pas arrivé au début du tableau, et que l'élément courant est plus grand que celui à insérer.*
    A[$i+1$] $\leftarrow$ A[i] — *On décale l'élément courant (on le met dans **la** place vide).*
    $i \leftarrow i-1$ — *On s'intéresse à l'élément précédent.*
  A[$i+1$] $\leftarrow$ clé — *Finalement, on a trouvé où insérer notre nombre.*

2.1.4 Exemple

Les différentes étapes de l'exécution de l'algorithme TRI-INSERTION sur le tableau $[5;2;4;6;1;3]$ sont présentées figure 2.1.

2.1.5 Etude de la Complexité

Nous passons en revue les différentes étapes de notre algorithme afin d'évaluer son temps d'exécution. Pour ce faire, nous attribuons un coût en temps à chaque instruction, et nous comptons le nombre d'exécutions de chacune des instructions. Pour chaque valeur de $j \in [2,n]$, nous notons t_j le nombre d'exécutions de la boucle `tant que` pour cette valeur de j. Il est à noter que la valeur de t_j **dépend des données**...

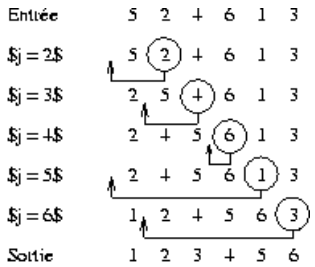

FIGURE 2.1 – Action de TRI-INSERTION sur le tableau $[5;2;4;6;1;3]$; l'élément à insérer est entouré par un cercle.

TRI-INSERTION	*Coût*	*Nombre d'exécutions*
Pour $j \leftarrow 2$ **à** n **faire**	c_1	n
clé $\leftarrow$ A[j]	c_2	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
tant que $i > 0$ **et** A[i] $>$ clé **faire**	c_4	$\sum_{j=2}^{n} t_j$
A[$i+1$] $\leftarrow$ A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[$i+1$] $\leftarrow$ clé	c_7	$n-1$

Le temps d'exécution total de l'algorithme est alors :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Complexité au meilleur : le cas le plus favorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié, comme le montre le cas $j = 4$ de la figure 2.1. Dans ce cas $t_j = 1$ pour tout j.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

$T(n)$ peut ici être écrit sous la forme $T(n) = an + b$, a et b étant des *constantes* indépendantes des entrées, et $T(n)$ est donc une **fonction linéaire** de n.

Le plus souvent, comme c'est le cas ici, le temps d'exécution d'un algorithme est fixé pour une entrée donnée ; mais il existe des algorithmes « aléatoires » intéressants dont le comportement peut varier même pour une entrée fixée. Nous verrons un algorithme de ce style au chapitre 2 : une version « aléatoire » du *tri rapide*

Complexité au pire : le cas le plus défavorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié dans l'ordre inverse, comme le montre le cas $j = 5$ de la figure 2.1. Dans ce cas $t_j = j$ pour tout j.

Rappel : $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$. Donc $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ et $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

$T(n)$ peut ici être écrit sous la forme $T(n) = an^2 + bn + c$, a, b et c étant des constantes, et $T(n)$ est donc une **fonction quadratique** de n.

Complexité en moyenne : supposons que l'on applique l'algorithme de tri par insertion à n nombres choisis au hasard. Quelle sera la valeur de t_j ? C'est-à-dire, où devra-t-on insérer A[j] dans le sous-tableau A[$1..j-1$] ? En moyenne, pour moitié les éléments de A[$1..j-1$] sont inférieurs à A[j], et pour moitié supérieurs. Donc $t_j = j/2$. Si l'on reporte cette valeur dans l'équation définissant $T(n)$, on obtient, comme dans le pire cas, une fonction quadratique en n.

Caveat : ce raisonnement est partiellement faux ; un raisonnement précis doit bien évidemment tenir compte des *valeurs* des éléments déjà triés. Pour un calcul précis, voir KNUTH [**?**, p. 82]. CORI et LÉVY [**?**, p. 26] font un autre raisonnement et trouve un autre résultat (de même ordre de grandeur). Les deux sont justes : tout dépend de l'hypothèse que l'on prend sur les jeux de données. Ainsi [**?**] suppose que les permutations sont équiprobables, et [**?**] que les valeurs à trier sont équiprobables...

Ordre de grandeur

Ce qui nous intéresse vraiment, c'est l'ordre de grandeur du temps d'exécution. Seul le terme dominant de la formule exprimant la complexité nous importe, les termes d'ordres inférieurs n'étant pas significatifs quand n devient grand. On ignore également le coefficient multiplicateur constant du terme dominant. On écrira donc, à propos de la complexité du tri par insertion :

meilleur cas : $\Theta(n)$.

pire cas : $\Theta(n^2)$.

en moyenne : $\Theta(n^2)$.

En général, on considère qu'un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur.

Classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :
— Les algorithmes sub-linéaires dont la complexité est en général en $O(\log n)$.
— Les algorithmes linéaires en complexité $O(n)$ et ceux en complexité en $O(n \log n)$ sont considérés comme rapides.
— Les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents
— Les algorithmes exponentiels (par exemple $O(n!)$ ou $O(k^n), k > 1$ dont la complexité est supérieure à tout polynôme en n) sont tout simplement impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

2.2 Tri par fusion

2.2.1 Principe

L'algorithme de tri par fusion est construit suivant le paradigme « diviser pour régner » :

1. Il divise la séquence de n nombres à trier en deux sous-séquences de taille $n/2$.
2. Il trie récursivement les deux sous-séquences.
3. Il fusionne les deux sous-séquences triées pour produire la séquence complète triée.

La récursion termine quand la sous-séquence à trier est de longueur 1... car une telle séquence est toujours triée.

2.2.2 Algorithme

La principale action de l'algorithme de tri par fusion est justement la fusion des deux listes triées.

La fusion

Le principe de cette fusion est simple : à chaque étape, on compare les éléments minimaux des deux sous-listes triées, 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, p, q et r, tels que $p \leq q < r$ et tels que les tableaux $A[p..q]$ et $A[q+1..r]$ soient triés. L'algorithme est présenté figure 2.2.

```
FUSIONNER(A, p, q, r)
  i ← p                                  indice servant à parcourir le tableau A[p..q]
  j ← q+1                                indice servant à parcourir le tableau A[q+1..r]
  Soit C un tableau de taille r−p+1      tableau temporaire dans lequel on construit le résultat
  k ← 1                                  indice servant à parcourir le tableau temporaire
  tant que i ≤ q et j ≤ r faire          boucle de fusion
    si A[i] < A[j] alors C[k] ← A[i]
                         i ← i+1
                   sinon C[k] ← A[j]
                         j ← j+1
    k ← k+1
  tant que i ≤ q faire C[k] ← A[i]       on incorpore dans C les éléments de A[p..q]
                       i ← i+1             qui n'y seraient pas encore ; s'il y en a,
                       k ← k+1             les éléments de A[q+1..r] sont déjà tous dans C
  tant que j ≤ r faire C[k] ← A[j]       on incorpore dans C les éléments de A[q+1..r]
                       j ← j+1             qui n'y seraient pas encore ; s'il y en a,
                       k ← k+1             les éléments de A[p..q] sont déjà tous dans C
  pour k ← 1 à r−p+1 faire               on recopie le résultat dans le tableau originel
      A[p+k−1] ← C[k]
```

FIGURE 2.2 – Algorithme de fusion de deux sous-tableaux adjacents triés.

Complexité de la fusion

Étudions les différentes étapes de l'algorithme :
— les initialisations ont un coût constant $\Theta(1)$;
— la boucle *tant que* de fusion s'exécute au plus $r-p$ fois, chacune de ses itérations étant de coût constant, d'où un coût total en $O(r-p)$;
— les deux boucles *tant que* complétant C ont une complexité respective au pire de $q-p+1$ et de $r-q$, ces deux complexités étant en $O(r-p)$;
— la recopie finale coûte $\Theta(r-p+1)$.

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

Le tri

Écrire l'algorithme de tri par fusion est maintenant une trivialité (cf. figure 2.3).

```
TRI-FUSION(A, p, r)
  si p < r alors q ← ⌊(p+r)/2⌋
                 TRI-FUSION(A, p, q)
                 TRI-FUSION(A, q+1, r)
                 FUSIONNER(A, p, q, r)
```

FIGURE 2.3 – Algorithme de tri par fusion.

2.2.3 Complexité

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

Diviser : cette étape se réduit au calcul du milieu de l'intervalle $[p;r]$, sa complexité est donc en $\Theta(1)$.

Régner : l'algorithme résout récursivement deux sous-problèmes de tailles respectives $\frac{n}{2}$, d'où une complexité en $2T(\frac{n}{2})$.

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 :

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

Un théorème, que nous ne détaillons pas ici, permet de déduire que :

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

Pour des valeurs de n suffisamment grandes, le tri par fusion est donc nettement plus efficace que le tri par insertion dont le temps d'exécution est en $\Theta(n^2)$.

2.3 Tri rapide (*Quicksort*)

2.3.1 Principe

Le tri rapide est fondé sur le paradigme « diviser pour régner », tout comme le tri fusion, il se décompose donc en trois étapes :

Diviser : Le tableau $A[p..r]$ est partitionné (et réarrangé) en deux sous-tableaux non vides, $A[p..q]$ et $A[q+1..r]$, tels que chaque élément de $A[p..q]$ soit inférieur ou égal à chaque élément de $A[q+1..r]$. L'indice q est calculé pendant la procédure de partitionnement.

Régner : Les deux sous-tableaux $A[p..q]$ et $A[q+1..r]$ sont triés par des appels récursifs.

Combiner : Comme les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner, le tableau $A[p..r]$ est déjà trié !

2.3.2 Algorithme

TRI-RAPIDE(A, p, r)

si $p < r$ **alors** $q \leftarrow$PARTITIONNEMENT(A, p, r)
TRI-RAPIDE(A, p, q)
TRI-RAPIDE(A, $q+1$, r)

L'appel TRI-RAPIDE(A, 1, *longueur*(A)) trie le tableau A. Le point principal de l'algorithme est bien évidemment le partitionnement qui réarrange le tableau A sur place :

PARTITIONNEMENT(A, p, r)

$x \leftarrow A[p]$
$i \leftarrow p-1$
$j \leftarrow r+1$
tant que VRAI **faire**
  répéter $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$
  répéter $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$
  si $i < j$ **alors** échanger $A[i] \leftrightarrow A[j]$
    sinon renvoyer j

Exemple de partitionnement :

1. Situation intiale :

1	2	3	4	5	6	7
4	3	6	2	1	5	7

   Nous avons donc $x = 4$, $i = 0$ et $j = 8$.
2. On exécute la boucle « **répéter** $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$ » et on obtient $j = 5$.
3. On exécute la boucle « **répéter** $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$ », et on obtient $i = 1$.

4. Après l'échange on obtient le tableau :

1	2	3	4	5	6	7
1	3	6	2	4	5	7

5. On exécute la boucle « **répéter** $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$ » et on obtient $j = 4$.
6. On exécute la boucle « **répéter** $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$ », et on obtient $i = 3$.
7. Après l'échange on obtient le tableau :

1	2	4	3	5	6	7
1	3	2	6	4	5	7

8. On exécute la boucle « **répéter** $j \leftarrow j-1$ **jusqu'à** $A[j] \leq x$ » et on obtient $j = 3$.
9. On exécute la boucle « **répéter** $i \leftarrow i+1$ **jusqu'à** $A[i] \geq x$ », et on obtient $i = 3$.
10. Comme $i = j$, l'algorithme se termine et renvoie la valeur « 3 ».

2.3.3 Complexité

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'est-à-dire qu'il existe une constante c telle que $T(n) \leq c \times n^2$. Si $T(n)$ est la complexité au pire :

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n),$$

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

$$T(n) \leq \max_{1 \leq q \leq n-1} \left(cq^2 + c(n-q)^2\right) + \Theta(n).$$

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

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

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 $\Theta(n^2)$.

Meilleur cas

On subodore que le meilleur cas apparaît quand la procédure de partitionnement produit deux régions de taille $\frac{n}{2}$. La récurrence est alors :

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

ce qui, d'après le cas intermédiaire du même théorème qu'utilisé pour le tri fusion, nous donne

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

Complexité en moyenne

Le calcul et la démonstration de la complexité dans le cas moyen dépasse le cadre de ce cours. Nous nous contentons simplement de dire ici que le temps d'exécution moyen du tri rapide est également $O(n \log n)$.