Analyse d’algorithmes et programmation

- Les algorithmes de tri -

Kahina BOUCHAMA

Université de Versailles, Saint-Quentin en Yvelines

Février 2023

Table des matières

Principe *diviser pour régner*

Il s'agit d'une méthode de conception des algorithmes impliquant trois étapes:

- **Diviser:** consiste à découper le problème initial en sous-problèmes de taille inférieure;
- **Régner:** il s'agit de résoudre chaque sous-problème récursivement;
- **Combiner:** il s'agit de recomposer les solutions des sous-problèmes de sorte à produire la solution du problème original.

Très utilisé en algorithmiques. Exemple, en:

- recherche dichotomique;
- tri par fusion;
- multiplication des grands nombres (algorithme de Karatsuba);
- ...

Son principe:

Le tri par fusion ou merge sort est un algorithme de tri très efficace de nature récursive. Le tri par fusion est basé sur le principe diviser pour régner. Il opère comme suit:

1. On divise le tableau de n éléments en deux sous-tableaux de taille $\approx n/2$ chacun.
2. On trie les deux sous-tableaux récursivement en utilisant le tri par fusion. La récursion s'arrête si les tableaux sont de taille 1, et donc trivialement triés.
3. On fusionne les deux sous-tableaux triés en un seul tableau trié.

Algorithme du tri par fusion:

Algorithm 1 Tri-fusion ($L[0, \ldots, n-1]$)

Entrées: Un tableau L à trier, de taille n.
Sortie: Le tableau L trié.

if $n \leq 1$ **then**
  Renvoyer L
end if
$m = \lfloor \frac{n}{2} \rfloor$
Renvoyer Fusion(Tri-fusion($L[0, \ldots, m-1]$), Tri-fusion($L[m, \ldots, n-1]$))

La procédure de Fusion

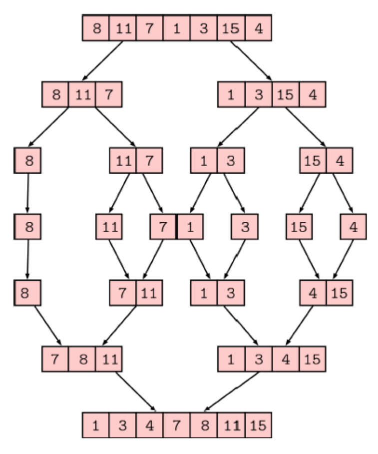
Algorithm 2 Procédure Fusion

Entrées: Deux tableaux L_1 et L_2 triés.
Sortie: Un nouveau tableau trié L trié qui est le résultat de la fusion des deux tableaux L_1 et L_2.

```
L = [0, ..., 0], L est de longueur len(L1) + len(L2)
i = 0, j = 0, k = 0
while (i < len(L1)) et (j < len(L2)) do
    if L1[i] < L2[j] then
        L[k] = L1[i]
        i = i + 1
    else
        L[k] = L2[j]
        j = j + 1
    end if
    k = k + 1
end while
if i = len(L1) then
    for l de j à len(L2) − 1 do
        L[k] = L2[l]
        k = k + 1
    end for
else
    for l de i à len(L1) − 1 do
        L[k] = L1[l]
        k = k + 1
    end for
end if
Renvoyer L
```

Exemple:

Soit **L = [8, 11, 7, 1, 3,15, 4]** , le tableau à trier.



Justesse du tri par fusion

Pour vérifier que l'algorithme de tri par fusion est correcte, on va s'intéresser à la boucle centrale de la procédure *Fusion*. En analysant par la méthode de l'invariant de boucle, on aura:

Invariant de boucle (IDB): Au début de chaque itération k de la boucle centrale, le tableau L contient les k plus petits éléments des tableaux L_1 et L_2 en ordre triée.

Initialisation: Avant la première itération de la boucle, L est vide, il contient donc les $k = 0$ plus petits éléments des tableaux L_1 et L_2, d'où IDB est vrai.

Conservation: On suppose que $L_1[i] \leq L_2[j]$. Alors, $L_1[i]$ est le plus petit élément qui n'a pas encore été copié dans L. En début de boucle, L contient les k plus petits éléments de L_1 et L_2. Une fois la boucle terminée, $L_1[i]$, qui est plus grand que tout élément de L, sera copié dans L. Le tableau L est alors trié après cette opération et il contiendra les $k + 1$ plus petits éléments des deux tableaux. A la fin de la boucle, on incrémente k d'un pas, ce qui recrée l'invariant pour l'itération suivante.

Terminaison: A la fin de la boucle, on a $k = len(L_1) + len(L_2)$. Par l'invariant de boucle, L contient les $len(L_1) + len(L_2)$ plus petits éléments de L_1 et L_2 en triés, d'où la justesse de l'algorithme de tri par fusion.

Complexité de la procédure *Fusion*

La complexité est de $\Theta(n)$, où $n = len(L_1) + len(L_2)$ est le nombre total d'éléments fusionnés.

Preuve:

- On suppose que L_1 et L_2 sont virtuellement collé en un tableau L_{init}.
- A chaque étape, on enlève un élément du tableau L_{init}, en faisant une comparaison entre deux éléments. Comme il y a n éléments au total, on effectue n étapes.

Soit $T(n)$ le temps d’exécution d’un problème de taille n.

- Si la taille du problème est suffisamment petite $n \leq c$ pour une constante c qui dépend du problème étudié, la solution prend un temps constant $\Theta(1)$.
- On divise le problème en t sous-problèmes de taille $n = m$ (on peut avoir $t = m$).
- $D(n)$ le temps nécessaire pour diviser le problème en sous-problèmes
- $C(n)$ le temps nécessaire pour combiner les solutions des sous-problèmes afin de construire la solution finale.

Nous avons alors:

$$T(n) = \begin{cases} \Theta(1), & \text{Si } n \leq c, \\ tT(n/m) + D(n) + C(n), & \text{sinon.} \end{cases} \quad (1)$$

Complexité d'un algorithme de type *diviser pour régner*

Cas du tri par fusion: On a $c = 1$, $t = m = 2$ et $D(n) = \Theta(1)$ pour le calcul du milieu du tableau. On a aussi $C(n) = \Theta(n)$ qui est la complexité de la procédure de Fusion. On obtient alors:

$$T(n) = \begin{cases} \Theta(1), & \text{Si } n \leq 1, \\ 2T(n/2) + \Theta(n), & \text{sinon.} \end{cases} \tag{2}$$

Prouvons que $T(n) = n \log(n)$

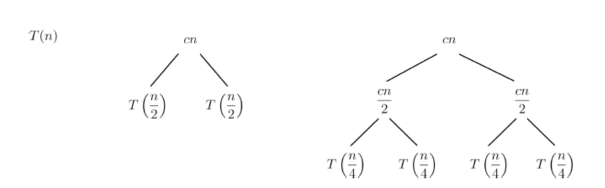
La complexité du tri par fusion est quasi-linéaire. En effet, on utilisera quelques simplifications:

- On réécrit l'équation (3) en utilisant une constante c au lieu des notations asymptotiques;
- Sans nuire à l'analyse, on peut utiliser le même symbole de constante pour désigner à la fois le temps de résolution dans le cas $n = 1$ et le temps par élément du tableau dans On obtient alors:

$$T(n) = \begin{cases} c, & \text{Si } n \leq 1, \\ 2T(n/2) + cn, & \text{sinon.} \end{cases} \tag{3}$$

- Encore plus de simplification et sans nuire à la preuve, on supposera que $n = 2^i$.

Complexité d'un algorithme de type *diviser pour régner*



Construction d'un arbre récursif pour la récurrence $T(n) = 2T(n/2) + cn$:

- On développe $T(n)$ comme un arbre de racine cn;
- On génère deux sous-arbres, chacun correspondant à $T(n/2)$;
- On remplace chacun des $T(n/2)$ par un arbre de racine $cn/2$ ayant lui-même deux sous-arbres $T(n/4)$;
- On continue à développer de la même manière jusqu'à atteindre les feuilles;
- Les feuilles sont des problèmes de taille 1, ayant une complexité de résolution c.

Nous avons au total $log_2 n + 1$ niveaux, chacun de complexité cn.

Par combinaison, on obtient:

$$T(n) = (logn + 1).cn = logn.cn + cn \tag{4}$$

On déduit que:

$$T(n) = \Theta(nlogn). \tag{5}$$

On conclut alors que la complexité est toujours de $\Theta(nlogn)$ dans toutes les situations (pire cas, cas moyen, meilleur cas).

Le tri rapide

Ses caractéristiques

- Appelé en anglais quicksort, il s'appuie sur le principe de diviser pour régner;
- Donne de bons résultats sur les tableaux très désordonnés;
- Son temps d'exécution dans le pire des cas est de $\Theta(n^2)$ pour un tableau de taille n. En pratique, il est souvent meilleur, car son temps d'exécution dans le cas moyen est très bon ($\Theta(nlogn)$);
- C'est un tri sur place.

Son principe

L'idée du tri rapide est de séparer le tableau à trier en deux. On choisit une valeur du tableau qu'on appelle *pivot*. On partitionne alors notre tableau en deux parties:

- la première partie contient les valeurs du tableau qui sont inférieures ou égales au pivot,
- la deuxième partie contient les valeurs supérieures au pivot.

Algorithme du tri rapide

Algorithm 3 Tri-rapide($L,\ l,\ r$)

if $l \leq r$ **then**
    $m =$ Partition(L, l, r)
    Tri-rapide($L, l, m - 1$)
    Tri-rapide($L, m + 1, r$)
end if

Algorithm 4 Partition($L,\ l,\ r$)

pivot $= L[r]$
$i = l - 1$
for j de l à r faire **do**
    if $L[j] \leq pivot$ **then**
        $i = i + 1$
        $L[i] \leftrightarrow L[j]$
    end if
end for$L[i + 1] \leftrightarrow L[r]$

Exemple de fonctionnement du tri rapide

Soit le tableau $L = [6, 5, 1, 3, 8, 4, 7, 9, 2]$. On choisit comme pivot le dernier élément du tableau. Voici l'état du tableau après chacune des itérations de l'algorithme:

6	5	1	3	8	4	7	9	2
6	5	1	3	8	4	7	9	2
6	5	1	3	8	4	7	9	2
1	5	6	3	8	4	7	9	2
1	2	6	3	8	4	7	9	5

6	3	8	4	7	9	5	$\rightarrow$	6	3	8	4	7	9	5	$\rightarrow$
3	6	8	4	7	9	5	$\rightarrow$	3	6	8	4	7	9	5	$\rightarrow$
3	4	8	6	7	9	5	$\rightarrow$	3	4	5	6	7	9	8	

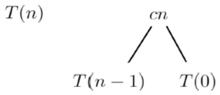
Analyse des performances du tri rapide

- Le temps d'exécution du tri rapide dépend du caractère équilibré ou non du partitionnement;
- La complexité temporelle de la procédure *Partition* est de $\Theta(n)$;

Partitionnement dans le cas le plus défavorable (tableau déjà trié): Il correspond au cas ou le résultat du partitionnement sont deux tableaux de taille respective $n-1$ et 0. Supposons que ce partitionnement survienne à chaque appel récursif.

- Pour le sous-tableau de taille 0, on a $T(0) = \Theta(1)$, la récurrence pour le temps d'exécution est donnée par:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + cn \end{aligned}$$



$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + 2cn \\ &= T(n-3) + 3cn \\ &\ldots \\ &= T(n-(n-1)) + (n-1)cn \\ &= T(0) + cn^2 - cn \\ &= \Theta(n^2) \end{aligned}$$

Le cas le plus favorable:

Ce cas survient lorsque le partitionnement est le plus équilibré possible, c'est-à-dire quand la procédure `Partition` produit à chaque étape deux sous-tableaux de taille $\frac{n}{2}$. Dans ce cas le temps d'exécution est :

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

Analysons cette équation :

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\
&= 2^2T\left(\frac{n}{4}\right) + 2cn \\
&= 2^2\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\
&= 2^3T\left(\frac{n}{8}\right) + 3cn \\
&\dots
\end{aligned}$$

Cette récurrence continuera jusqu'à $k = \log_2(n)$ (en supposant que $n = 2^k$).

En remplaçant $k = \log_2(n)$ on obtient $T(n) = nT(1) + cn\log(n) = \Theta(n\log(n))$.