DIU EIL Algorithmique : Tri partition/fusion (merge sort) tri par segmentation (quicksort)

Concept : Diviser pour régner; Analyse de coût
Méthode : Décomposition récursive

Le tri fusion est un algorithme de tri utilisant le principe de "diviser pour régner", inventé par John von Neumann en 1945. Il effectue au plus $O(n \log n)$ opérations, qui est la complexité en temps dans le pire cas optimale parmi les algorithmes de tri à comparaison. Le tri fusion (ou ses variantes, comme Timsort) est utilisé par des librairies standards de plusieurs langages, comme Python, Java ou Perl.
Dans ce TD, on propose d'étudier la complexité du tri fusion, et d'une variante moins connue : le tri fusion en place.

Exercice 1: Tri fusion "classique"

L'algorithme de tri fusion est un algorithme récursif qui consiste à trier les deux moitiés du tableau séparément puis à fusionner les deux sous-tableaux triés ainsi obtenus. Pour trier un tableau de taille n, il s'opère récursivement de la façon suivante :

— On trie le sous tableau constitué des $\lceil n/2 \rceil$ premiers éléments du tableau.
— On trie le sous tableau constitué des $\lfloor n/2 \rfloor$ derniers éléments du tableau.
— On fusionne les deux tableaux triés en un seul tableau trié.

1. On veut trier le tableau de caractères "AZERTYUI".
   a) Écrire les valeurs des paramètres et les valeurs de retour de tous les appels récursifs à la fonction `tri_fusion`.
   b) Compter le nombre de comparaisons qu'effectue cet algorithme sur cet exemple.
2. Écrire (en pseudo-code) une fonction `fusion(T1, T2)` qui prend en entrée deux tableaux triés par ordre croissant (de taille n_1 et n_2) et rend un nouveau tableau de taille $n_1 + n_2$ contenant les éléments de T1 et T2 triés par ordre croissant.
   a) Quel est la complexité de votre algorithme ?
   b) Si ce n'est pas le cas : améliorer votre algorithme pour qu'il effectue au plus $n_1 + n_2$ opérations.
3. Écrire (en pseudo-code), une fonction récursive `tri_fusion(T)` qui effectue le tri de T par ordre croissant.
4. Soit $C(n)$ le temps que met le tri fusion pour trier un tableau de taille n. Donner une formule de récurrence pour $C(n)$ et la résoudre.

Rappel : L'étude de la complexité d'algorithmes de type *diviser pour régner* implique souvent des formule de récurrences reliant $C(n)$ à $C(n/b)$. Le résultat suivant est alors souvent utile :

Théorème 1 (Master theorem, Theorem 4.1 du livre de Cormen et. al)**.** *Soit $a \geq 1$ et $b > 1$ deux constantes. Soit $f(n)$ une fonction et $C(n)$ définit par la récurrence :*

$$C(n) = aC\left(\frac{n}{b}\right) + f(n).$$

$C(n)$ a les bornes asymptotiques suivantes :

$C(n) = \Theta(n^{\log_b a})$ *si* $f(n) = O(n^{\log_b a - \varepsilon})$ *pour un* $\varepsilon > 0$;
$C(n) = \Theta(n^{\log_b a} \log n)$ *si* $f(n) = \Theta(n^{\log_b a})$
$C(n) = \Theta(f(n))$ *si* $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *pour un* $\varepsilon > 0$ *et si* $f(n) \geq acf(n/b)$ *pour* $c \geq 1$.

Le tri rapide (ou quicksort) est un algorithme de tri inventé par Hoare en 1961. Il utilise la méthode de diviser pour régner. *Bien que sa complexité dans le pire des cas soit de $O(n^2)$, il a une complexité moyenne de $O(n \log n)$. C'est probablement l'algorithme de tri le plus utilisé. Dans ce TP, nous allons étudier sa complexité en temps et en espace, en faisant le lien avec les arbres binaires de recherche.*
Les arbre binaires de recherche ont été introduits par plusieurs personnes de façon indépendante à la fin des années 50. Cette structure de donnée dispose des opérateurs de recherche, du minimum, du maximum, d'insertion et et de suppression en un temps linéaire en la hauteur. En pratique, on utilise souvent des variations de ces arbres qui ont une garantie de performance, comme les arbres AVL (introduits par Adelson-Velski and Landis en 1962) ou les arbres 2-3 (introduits par Hopcroft en 1970).

L'algorithme du tri rapide fonctionne de la façon suivante :
- On choisit un indice du tableau `k`, le pivot, `T[k]` est la valeur du pivot
- On partitionne les éléments en deux sous-tableaux ;
  - On place tous les éléments plus petits que la valeur du pivot en début de tableau ;
  - On place tous les éléments plus grands que la valeur du pivot en fin de tableau ;
  - On le insère le pivot entre les deux ;
- On trie récursivement les deux sous-tableaux.

Exercice 2: Complexité en temps de Quicksort : pire cas et moyenne

On s'intéresse dans cet exercice au nombre de comparaisons qu'il faut pour effectuer le tri d'un tableau `T` de n éléments distincts. On note P_n le nombre de comparaisons nécessaires pour trier un tableau de taille n dans le pire des cas et M_n le nombre moyen de comparaisons. On admettra que l'opération de partitionnement coûte $n-1$ comparaisons.

1. On choisit pour pivot le premier élément d'un tableau (l'élément 1).
   a) Dérouler l'algorithme sur le tableau $\texttt{T}_1 = [A, B, C, D, E, F, G]$. Combien de comparaisons effectue cet algorithme pour trier ce tableau ?
   b) Dérouler l'algorithme sur le tableau $\texttt{T}_2 = [D, B, F, G, A, E, C]$. Combien de comparaisons effectue cet algorithme pour trier ce tableau ?
2. *(Complexité en pire cas)* Montrer que $P_n \leq n^2$.
3. *(Complexité en moyenne)* On considère maintenant que le pivot est choisit aléatoirement, de manière uniforme, à chaque étape.
   a) Quelle est la probabilité que ce pivot correspond à l'élément de rang i dans le tableau ?
   b) En déduire que
   $$M_n = (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(M_i + M_{n-i-1}) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} M_i.$$
   Que vous rappelle cette formule ? Quelles sont les valeurs initiales de cette suite récurrente ?
   c) Montrer que $M_n = 2n\log(n) + \mathcal{O}(n)$ et conclure sur le coût moyen de l'algorithme lorsque le choix du pivot est aléatoire et uniforme.
   Indications : (1) calculer $nM_n - (n-1)M_{n-1}$. (2) En déduire que $M_n/(n+1) - M_{n-1}/n \leq 2/n$. (3) Utiliser le fait que $1 + 1/2 + 1/3 + \cdots + 1/n = \log n + \mathcal{O}(1)$)

Exercice 3: Tri fusion en place

Au prix d'une complexité plus grande, on peut effectuer le tri fusion sans utiliser de tableau extérieur. Pour cela, on modifie l'opération de fusion. L'idée est la suivante : pour fusionner deux sous tableaux V et W, on découpe chaque sous-tableau en deux parties de même taille (pour simplifier, on suppose pour l'instant que $n = 2^k$). Ces parties sont notées V_1, V_2 et W_1, W_2. On effectue alors les opérations de fusion suivantes (voir Figure 1) :

— fusion(V_1,W_1)
— fusion(V_2,W_2)
— fusion(V_2,W_1)

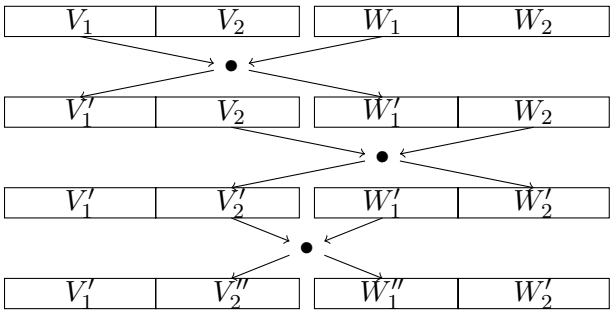


FIGURE 1 – Les opérations de fusion du tri de fusion en place

1. Soit $D(n)$ le nombre de comparaisons qu'effectue l'algorithme de fusion en place pour fusionner deux tableaux de taille $n/2 = 2^{k-1}$ et $n/2 = 2^{k-1}$.
   a) Écrire une formule de récurrence pour $D(n)$.
   b) En déduire la complexité de la fusion en place lorsque $n = 2^k$.
   c) En déduire la complexité du tri en place lorsque $n = 2^k$.
2. Montrer que l'opération de fusion est correcte.
3. Écrire la fonction `fusion_en_place` en pseudo-code.
4. Écrire la fonction `tri_en_place` en pseudo-code.
5. Généraliser l'algorithme et l'étude de la complexité à des tableaux de taille quelconque (différent de 2^k).

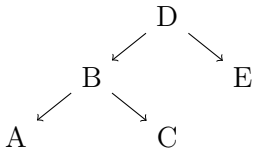
Exercice 4: Mélanger pour mieux trier ? (tri par segmentation)

1. Pourquoi si avant de commencer l'algorithme les éléments sont dans un ordre aléatoire (*i.e.*, correspondant à une permutation aléatoire), alors l'algorithme qui choisit pour pivot l'indice 1 a la même complexité moyenne que celui qui choisit un pivot aléatoirement ?
2. Écrire une procédure qui tire une permutation aléatoire d'un tableau de taille n en temps $O(n)$.
3. Pourquoi certaines implémentations de quicksort mélangent le tableau avant de le trier ?

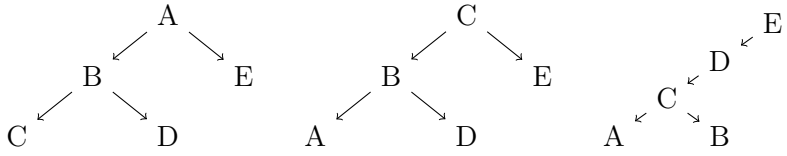
Exercice 5: Complexité en espace et hauteur d'un arbre

La complexité en espace d'un algorithme est la taille de la mémoire qu'il faudrait réserver pour pouvoir exécuter notre algorithme. Dans le cas de Quicksort, cette quantité est liée à la hauteur de l'arbre des appels récursifs.

On peut représenter les appels récursifs à la fonction Quicksort par un arbre d'appel, dans lequel l'étiquette d'un noeud est le contenu du pivot. Par exemple, si l'on trie le tableau $\{D, B, C, A, E\}$ en choisissant pour pivot le premier élément du tableau, on obtient l'arbre d'appel ci-contre.

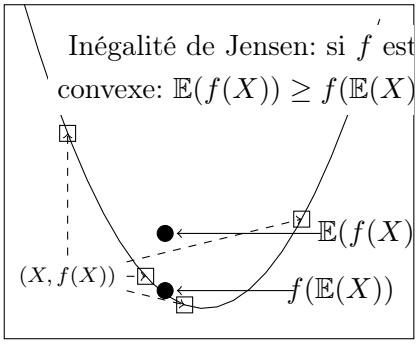


1. On suppose que la hauteur de l'appel récursif est h. Justifier pourquoi la complexité en espace de Quicksort est $O(h)$.
2. On suppose que le pivot est le premier élément du tableau. Dessiner les arbres d'appels correspondants au tri des tableaux de l'exercice précédents :
   a) $\mathtt{T}_1 = [A, B, C, D, E, F, G]$
   b) $\mathtt{T}_2 = [D, B, F, G, A, E, C]$.
   Quelle est la hauteur de ces arbres ?
3. Justifier pourquoi l'arbre d'appel est un arbre binaire de recherche, c'est à dire un arbre binaire tel que si un nœud a une clé k, alors :
   — les clés des nœuds du sous-arbre gauche sont inférieures ou égales à k.
   — les clés des nœuds du sous-arbre droit sont supérieures ou égales à k.
4. Parmi les arbres suivants :



   Lesquels sont des arbres binaires de recherche (*i.e.*, peuvent correspondre à un arbre d'appel de la fonction Quicksort) ?
5. Quelle est la hauteur maximale d'un arbre de taille n ? Quelle est sa hauteur minimale ?
6. On effectue le tri d'un tableau de taille n en choisissant le pivot aléatoirement. On note X_n la hauteur de l'arbre d'appel et on appelle $Y_n = 2^{X_n}$ la hauteur exponentielle de cet arbre. Soit Y_i^g (respectivement Y_i^d) la hauteur exponentielle du sous-arbre gauche (respectivement droit) de l'arbre lorsque ce sous-arbre est de taille i. On note Z_i une variable qui vaut 1 lorsque le sous-arbre gauche a i nœuds et 0 sinon.
   a) Montrer que $Y_n = 2\sum_{i=0}^{n-1} Z_i \max(Y_i, Y_{n-i-1})$.
   b) En déduire que

$$\mathbb{E}(Y_n) \leq 2\sum_{i=0}^{n-1} \frac{1}{n}(\mathbb{E}(Y_i) + \mathbb{E}(Y_{n-i-1})) = \frac{4}{n}\sum_{i=0}^{n-1} \mathbb{E}(Y_i)$$



   c) Calculer $n\mathbb{E}(Y_n) - n\mathbb{E}(Y_{n-1})$ et en déduire que

$$\mathbb{E}(Y_n) \leq \frac{n+3}{n}\mathbb{E}(Y_{n-1}) \leq \frac{(n+3)(n+2)(n+1)}{6}\mathbb{E}(Y_0) = \mathcal{O}(n^3).$$

   d) En déduire que $\mathbb{E}(T_n) \leq 3\log_2 n + \mathcal{O}(1)$.
7. Conclure sur la complexité en mémoire moyenne et en pire cas et comparer avec la complexité en temps.