Introduction à l'informatique
Tri d'un tableau

Benjamin Monmege

2019/2020

On a vu précédemment que pour rechercher un élément dans un tableau, la complexité était bien meilleure dès lors que le tableau était trié (c'est-à-dire que ses éléments sont rangés dans un ordre croissant). Il est donc naturel de se demander comment faire en sorte de trier un tableau et de la complexité nécessaire pour cela. Ce problème très naturel de tri d'éléments apparaît à de nombreuses reprises en informatique : par exemple, lorsqu'on a voulu visualiser la liste des restaurants à proximité par note moyenne décroissante, c'est un tri (par ordre décroissant, plutôt que croissant...) que l'on exécute.

1 Tri par insertion

Commençons par considérer la situation où l'on cherche à ranger dans sa main des cartes à jouer suivant un ordre précis (afin de séparer les couleurs, puis de ranger les cartes par ordre croissant dans chaque couleur, par exemple). Pour simplifier, considérons un cas particulier où nous n'avons que des cartes de carreau. Prenez une dizaine de cartes et rangez-les par ordre croissant : analysez alors votre façon de faire...

Je décris ici ma façon de faire lorsqu'il s'agit de trier un nombre conséquent de cartes. Je laisse le tas de cartes à trier face contre la table et prend les cartes les unes après les autres. Je prends la première carte dans ma main. Je retourne ensuite la seconde carte que je viens placer au bon endroit (avant ou après la première carte) dans ma main. Je retourne alors la troisième carte que je dois de même insérer au bon endroit dans ma main. Lorsque j'arrive à la treizième carte, il devient plus difficile de savoir où je dois l'insérer : si je décompose à nouveau ma routine, je vois que je *scanne* la suite des cartes de droite à gauche jusqu'à trouver l'endroit où je dois insérer la nouvelle carte. Essayez d'imiter ma méthode afin de classer un paquet de treize cartes de carreau, en décomposant bien chaque étape en opérations les plus simples possibles.

Cette procédure de tri s'appelle le *tri par insertion* du fait qu'on ne fait qu'insérer les éléments les uns après les autres au bon endroit dans la portion triée. On peut également l'appliquer pour trier un tableau. On suppose donc qu'on a en entrée de l'algorithme un tableau non trié, par exemple, le tableau $[10, 8, 2, 5, 13]$. On considère les éléments de gauche à droite, en cherchant à chaque fois à les placer au bon endroit dans la portion triée qui sera la partie gauche du tableau. Initialement, on considère donc le premier élément du tableau (10) qui est bien placé puisque c'est le seul élément considéré jusqu'alors. La portion triée est donc $[10, ...$ et il reste le tableau $...8, 2, 5, 13]$ à considérer. On regarde ensuite le second élément (8) puis on le compare à l'élément à sa gauche, afin de l'insérer au bon endroit dans la portion

triée : ici $8 < 10$ donc il faut échanger les deux éléments, ce qui termine l'insertion de 8 à sa place. On se retrouve alors avec la portion triée $[8, 10, ...$ et le reste du tableau $...2, 5, 13]$. Au coup suivant, on doit considérer l'élément 2 : il est inférieur à 10, donc on doit échanger sa place avec 10 (temporairement on obtient donc le tableau $[8, 2, 10, 5, 13]$), puis on le compare avec l'élément à sa gauche (8) pour arriver à la situation où la portion triée est $[2, 8, 10, ...$ et le reste $...5, 13]$. L'insertion de 5 se fait alors en deux étapes : on échange les éléments 10 et 5, puis les éléments 8 et 5, avant de voir que $2 < 5$ ce qui achève l'insertion de l'élément 5. Finalement, on considère l'élément 13 qui est directement supérieur à l'élément à sa gauche (10) stoppant dès le début son insertion. On termine donc avec le tableau trié $[2, 5, 8, 10, 13]$.

Voici une écriture sous forme de pseudo-code de cet algorithme :

```
fonction trier_par_insertion(tableau) :
    n := longueur(tableau)
    Pour i de 1 à n−1 faire
        x := tableau[i]
        # insérer x parmi les i premiers éléments
        j := i
        Tant que ((j > 0) et (x < tableau[j−1])) faire
            # décaler d'un élément
            tableau[j] := tableau[j−1]
            j := j−1
        FinTantQue
        # ici, x ≥ tableau[j−1] ou bien j = 0
        tableau[j] := x
    FinPour
    # le tableau est trié !
```

Exécutez l'algorithme sur le tableau $[8, 2, 10, 5, 13]$ pour bien comprendre comment il fonctionne : vous verrez qu'il ne fait pas exactement ce qui est dit au-dessus en s'épargnant des échanges de cases inutiles...

Exercice 1

1. Que se passe-t-il si on remplace la troisième ligne de l'algorithme par `Pour i de 0 à n−1 faire` ?
2. Montrons que cet algorithme termine. La seule raison pour laquelle il pourrait ne pas terminer est la boucle `Tant que`, comme dans l'algorithme de recherche dichotomique. Montrer que cette boucle termine bien dans tous les cas.

Notons également que cet algorithme ne retourne aucun résultat : on a effectivement choisi de faire un tri *en place*, c'est-à-dire qu'on ne renvoie pas un nouveau tableau, mais qu'on a modifié le tableau donné en entrée directement... Il est donc inutile de le retourner, puisqu'on a directement modifié le tableau dans la mémoire. Notons qu'en Python, le passage d'un tableau en argument d'une fonction se fait bien « par référence » et donc la modification au sein de la fonction se répercute bien sur la mémoire globale : attention, ce n'est pas le cas dans tous les langages de programmation !

Pour mieux comprendre cet algorithme (puis le comparer avec d'autres algorithmes, dans la suite), estimons sa complexité dans le pire des cas. Comme pour les algorithmes de recherche étudiés auparavant, il faut donc comptabiliser les opérations élémentaires.

Vu l'imbrication de boucles de cet algorithme, il convient de commencer par considérer les boucles les plus internes, c'est-à-dire le code qui est le plus « décalé à droite ». Considérons

donc d'abord la boucle `Tant que` qui exécute deux opérations élémentaires à chaque itération (une affectation dans `tableau[j]` et une décrémentation de `j`), auxquelles il faut ajouter les opérations élémentaires nécessaires pour tester si l'on doit continuer la boucle ou pas : ce test (`(j> 0) et (x<tableau[j−1])`) requiert deux comparaisons qu'on comptabilise donc comme deux opérations élémentaires. Une itération réclame donc 4 opérations élémentaires. Il faut aussi considérer le dernier test qui fait sortir de la boucle `Tant que`, qui nécessite aussi 2 tests dans le pire des cas. Pour totaliser la complexité de cette boucle interne, il nous suffit donc de savoir le nombre de fois que cette boucle s'exécute. Dans le pire des cas, l'élément `x` est inférieur à tous les éléments de la portion triée, ce qui pousse alors à le décaler jusqu'au tout début du tableau : la variable `j` prend alors toutes les valeurs de `i` à 1, avant d'être égale à 0, auquel cas on sort de la boucle. Dans le pire des cas, on exécute donc `i` fois la boucle, générant donc au total $4\texttt{i} + 2$ opérations élémentaires.

On peut ensuite passer à la boucle `Pour` :

— on y affecte la variable `x`

— ainsi que la variable `j` ;

— on exécute la boucle `Tant que`, coûtant donc $4\texttt{i} + 2$ opérations élémentaires dans le pire des cas ;

— finalement, on modifie la valeur de `tableau[j]`.

Lors de l'itération correspondant à une valeur particulière de `i`, on exécute donc $1+1+4\texttt{i}+2+1 = 4\texttt{i} + 5$ opérations élémentaires. Ce nombre dépend de l'itération `i` : on ne peut donc pas simplement multiplier le nombre d'opérations par le nombre d'itérations. À la place, on fait la somme de toutes les contributions des itérations : le nombre total d'opérations élémentaires exécutées au sein de la boucle `Pour` vaut donc

$$(4 \times 1 + 5) + (4 \times 2 + 5) + \cdots + (4 \times (n-1) + 5)$$

qu'on peut écrire sous la forme d'une somme qu'on décompose en deux sommes indépendantes :

$$\sum_{i=1}^{n-1}(4i+5) = \sum_{i=1}^{n-1}(4i) + \sum_{i=1}^{n-1} 5 = 4\sum_{i=1}^{n-1} i + 5 \times (n-1)$$

Il ne reste plus qu'à calculer la somme de gauche, qui n'est rien d'autre que la somme des $n-1$ premiers entiers dont on sait par ailleurs qu'elle vaut $(n-1)n/2$. On obtient donc un nombre d'opérations élémentaires égal à $2(n-1)n + 5(n-1) = 2n^2 + 3n - 5$.

Finalement, il faut aussi comptabiliser les opérations élémentaires en dehors de la boucle `Pour` : il n'y en a qu'une, l'affectation de la variable `n`. In fine, on exécute donc, dans le pire des cas, $2n^2 + 3n - 4$ opérations élémentaires. L'ordre de grandeur est donc en $O(n^2)$ (pour s'en convaincre ici, il suffit de remarquer que pour tout n supérieur à 3, $2n^2 + 3n - 4 \leq 2n^2 + 3n \times n = 5n^2$). L'algorithme de tri par insertion est donc un algorithme de complexité *quadratique* en la longueur du tableau à trier.

Exercice 2

On vient de voir que, dans le pire des cas, la complexité du tri par insertion est en $O(n^2)$. On peut raisonnablement se poser la question de savoir si on a surestimé le nombre d'opérations élémentaires. En fait, il n'en est rien. Trouver donc une suite de tableaux $(t_n)_{n\in\mathbf{N}}$ avec t_n un tableau de longueur n telle que le nombre C_n d'opérations élémentaires effectuées lors du tri du tableau t_n est de la forme $an^2 + bn + c$ avec a, b, c des constantes et $a > 0$.

2 Tri par fusion

Proposons maintenant une autre façon de trier un tableau, utilisant une méthode tout à fait différente, consistant à diviser-pour-régner. Illustrons-la sur l'exemple du tableau

$$[3, 16, 14, 1, 12, 7, 10, 4, 5, 11, 15]$$

On va diviser le problème en deux sous-problèmes de la même forme : ici, il suffit de couper le tableau en deux, pour obtenir deux sous-tableaux $[3, 16, 14, 1, 12, 7]$ et $[10, 4, 5, 11, 15]$ de taille (presque) identique. Imaginons qu'on sache trier ces deux sous-tableaux : on obtient alors les tableaux $[1, 3, 7, 12, 14, 16]$ et $[4, 5, 10, 11, 15]$. Pour obtenir le grand tableau trié, il suffit donc de fusionner ces deux tableaux triés. Mais ceci est très facile à faire (imaginez que vous avez deux jeux de cartes triés et que vous voulez les fusionner en conservant le tri...). Il suffit de remarquer que le plus petit élément du grand tableau trié doit forcément être 1 ou 4 (les premiers éléments des deux petits tableaux triés), c'est donc 1. Pour obtenir la suite du grand tableau trié, on continue ce même raisonnement avec les deux petits tableaux $[3, 7, 12, 14, 16]$ et $[4, 5, 10, 11, 15]$: c'est 3 le plus petit élément des deux qui doit donc être à la suite de 1 dans le tableau. Puis 4 vient ensuite : on se retrouve alors avec deux petits tableaux de la forme $[7, 12, 14, 16]$ et $[5, 10, 11, 15]$. On continue ainsi jusqu'à avoir complètement fusionné les deux petits tableaux triés et obtenu le tableau

$$[1, 3, 4, 5, 7, 10, 11, 12, 14, 15, 16]$$

La question reste cependant entière : comment fait-on pour trier les deux petits tableaux $[3, 16, 14, 1, 12, 7]$ et $[10, 4, 5, 11, 15]$? La réponse est simple : « on recommence ! ». En effet, la tactique décrite ci-dessus pour trier le grand tableau peut être aussi utilisée pour traiter ces deux petits tableaux, de manière indépendante. On peut ainsi visualiser dans la FIGURE 1 le cheminement complet pour trier le grand tableau : la division en deux sous-tableaux est marquée par l'éclair rouge et les deux flèches bleues, puis la fusion des deux tableaux triés est représentée par les flèches orange.

Comment écrire dans du pseudo-code la formule magique « on recommence ! » ?!? La manière la plus simple de faire est la suivante :

```
fonction tri_fusion(tableau) :
    n := longueur(tableau)
    Si n ≤ 1 alors
        retourner(tableau)
    Sinon
        gauche := tableau[0..⌊n/2⌋]
        droite := tableau[⌊n/2⌋ + 1..n−1]
        gauche_trié := tri_fusion(gauche)
        droite_trié := tri_fusion(droite)
        retourner( fusionner(gauche_trié, droite_trié) )
    FinSi
```

On a utilisé la notation `tableau[0..⌊n/2⌋]` pour représenter la portion gauche du tableau constituée des cases d'indice $0, 1, \ldots, \lfloor n/2 \rfloor$ du tableau. Contrairement au tri par insertion, cette fonction retourne un tableau : elle n'est pas *en place*. On a également utilisé une fonction `fusionner` dont on ne fournit pas le pseudo-code : c'est la fonction qui prend deux petits tableaux triés en entrée et doit les fusionner pour produire un grand tableau trié, comme expliqué ci-dessus. Finalement, le « on recommence ! » est écrit par l'appel de la fonction

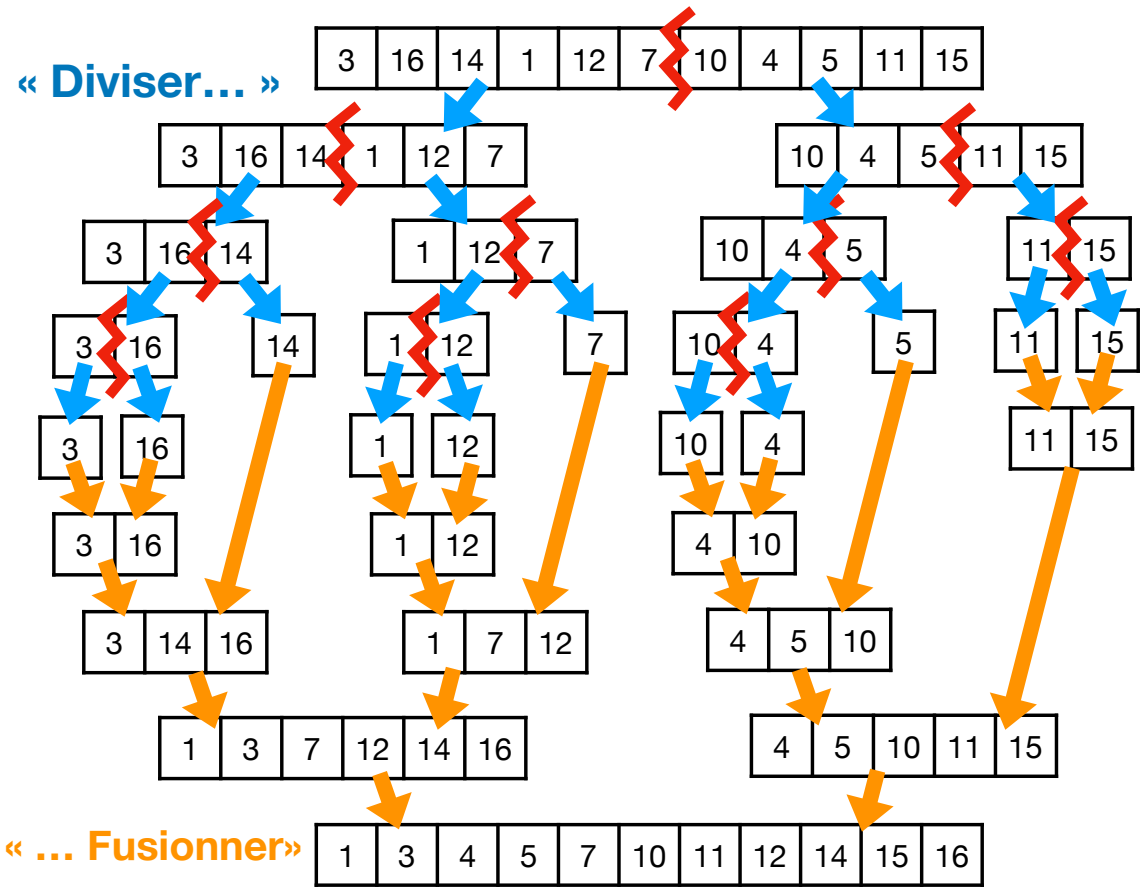

FIGURE 1 – Représentation graphique du tri par fusion

`tri_fusion` elle-même au sein de sa propre définition : on appelle cela des *appels récursifs* ; la fonction elle-même s'appelle une *fonction récursive*. Nous reverrons plus loin ce mécanisme.

En terme de complexité, il n'est pas très difficile de se convaincre que la fusion de deux tableaux triés peut s'exécuter en temps linéaire ($O(n)$) en la taille du grand tableau. Par ailleurs, comme on peut le voir en FIGURE 1, on fait assez peu d'appels récursifs finalement : comme pour la recherche dichotomique, l'intérêt de la méthode est qu'à chaque appel récursif, on a divisé par deux la longueur du tableau à trier. On ne peut donc faire qu'au plus $O(\log_2 n)$ appels récursifs imbriqués. Ceci explique pourquoi la complexité du tri par fusion est en $O(n \log_2 n)$, ce qu'on admet dans ce cours.

On est donc passé d'un tri de complexité $O(n^2)$ à un tri de complexité $O(n \log_2 n)$: c'est évidemment bien mieux de la même façon que la recherche dichotomique de complexité $O(\log_2 n)$ est bien meilleure que la recherche séquentielle de complexité $O(n)$. Mais peut-on encore mieux faire ? En fait, on peut montrer qu'il n'est pas possible de mieux faire, dès lors qu'on ne considère que des tris qui procèdent par comparaison des éléments deux par deux, comme c'est le cas pour les tris qu'on a étudiés jusque-là.