Gestion des données : tris itératifs et tris récursifs

Le travail sur les listes ou les tableaux est fondamental dans la gestion des données. En particulier, il existe de nombreux algorithmes de tris qu'ils soient itératifs ou récursifs. A l'instar de ce que nous avons vu dans les problèmes d'approximation numérique, certaines méthodes seront même plus efficaces que d'autres : c'est pour cela qu'on essaiera de retenir l'importance de la complexité en nombre d'opérations.

1 Les premiers tris utiles

Parmi les tris classiques, on peut considérer :

- le **tri à bulles** : pour une liste donnée de nombres réels, on parcourt le tableau plusieurs fois et à chaque étape, on échange les éléments adjacents afin de les remettre dans l'ordre. Ainsi, comme des bulles, et à chaque passage, les plus "grands" éléments se placent à la fin de la liste... jusqu'à ce que la liste soit enfin triée.

  Par exemple, si $L = [10, 5, 8, 1]$, alors le programme modifie la liste de sorte que :

  $$L \to [5, 8, 1, 10] \to [5, 1, 8, 10] \to [1, 5, 8, 10]$$

- le **tri par sélection** : pour une liste donnée de nombres réels, on extrait à chaque passage le plus petit élément qu'on place dans une nouvelle liste. D'ailleurs, ce programme repose sur une fonction secondaire : **la sélection**.

  Par exemple, si $L = [10, 5, 8, 1]$, alors le programme modifie la liste de sorte que :

  $$1 \to [1], 5 \to [1, 5], 8 \to [1, 5, 8], 10 \longrightarrow [1, 5, 8, 10]$$

- le **tri par insertion** : pour une liste donnée de nombres réels, on prend une valeur à chaque passage puis on l'insère à la bonne place dans une nouvelle liste. D'ailleurs, ce programme repose sur une fonction secondaire : **l'insertion**.

  Par exemple, si $L = [10, 5, 8, 1]$, alors le programme modifie la liste de sorte que :

  $$10 \to [10], 5 \to [5, 10], 8 \to [5, 8, 10], 1 \longrightarrow [1, 5, 8, 10]$$

Remarques

1. Si on note n la taille de la liste initiale, alors en comptant le nombre d'opérations (comparaisons et échanges) pour trier une telle liste, on peut montrer que pour chacun de ces tris itératifs, on a en notant $C(n)$ le coût en nombre d'opérations :
   $$C(n) = O(n^2)$$
2. Attention, ne croyez pas que ces tris itératifs sont grossiers. D'une part, ils ont l'avantage d'être faciles à mettre en oeuvre et d'autre part, on peut en construire des variantes fort intéressantes... par exemple, on peut faire du tri par insertion, en utilisant des curseurs dichotomiques pour trouver la place du nombre à insérer : c'est le **tri par insertion dichotomique** qui est beaucoup plus rapide et on peut même **estimer sa complexité** de sorte que :
   $$C(n) = \Theta(n \ln(n))$$

2 Un exemple de tri récursif : le tri fusion ou *merge sort*

Le **tri fusion** est un algorithme récursif qui va nous permettre de diviser notre problème de taille n en deux sous-problèmes de taille environ $n//2$. Pour cela, on considère encore une liste L constituée de n nombres réels et on note $m = n//2$.
Puis,

- on trie la première liste $L1$ constituée des éléments de la liste L d'indices 0 à $m-1$,
- on trie la seconde liste $L2$ constituée des éléments de la liste L d'indices m à $n-1$,

avant de **fusionner** les listes obtenues. Cela signifie que ce programme reposera d'abord sur une fonction auxiliaire $fusion$ permettant de fusionner deux listes déjà triées.

Remarque Si on note $C(n)$ la complexité en nombre d'opérations, alors celle-ci sera évidemment récursive puisqu'elle dépend directement de la complexité pour trier les sous-listes obtenues.
On peut alors chercher à **estimer la complexité** par des études empiriques, ou bien par encadrement en considérant des tableaux de taille $n = 2^k$ ou 2^{k+1}, car dans ce cas particulier, il est souvent plus facile d'obtenir une forme explicite de $C(n)$... on donnera un exemple de ce calcul à la fin du TD.

Exercice 1 (tri fusion ou *merge sort*)**.** [ ]

On cherche à construire l'algorithme décrit précedemment. Attention, comme il s'agit d'un algorithme récursif, **il faudra traiter la condition d'arrêt de ces appels récursifs**, c'est à dire lorsqu'une telle liste n'a pas d'élément ou n'est constituée que d'un seul élément.

1. Dans le langage Python, construire le programme $fusion : (L1 : list, L2 : list) \longrightarrow list$ qui renvoie la fusion de deux listes $L1$ et $L2$ **déjà triées**. *On cherchera à comparer les premiers éléments (les plus petits) de chacune des listes et il faudra tenir compte des tailles n_1 et n_2 associées aux listes $L1$ et $L2$.*

2. En déduire le programme $trifusion : (L : list) \longrightarrow list$ qui renvoie les éléments de L triés dans l'ordre croissant, et cela en faisant appel à la fonction $fusion$.

3. Justifier que le nombre d'appels récursifs est nécessairement fini, ce qui assurera la terminaison de notre programme.

3 Applications : programmation des tris classiques

Exercice 2 (tri à bulles). [ ]

1. Expliquer à quoi correspond cette instruction : `a,b=b,a`.
2. Dans le langage Python, construire le programme $tribulles(L : list) \longrightarrow list$ qui pour toute liste L donnée, renvoie une liste triée contenant les valeurs de L.

Exercice 3 (tri par sélection). [ ]

1. Dans le langage Python, construire la fonction $selectmin(L : list) \longrightarrow tuple$ qui pour toute liste L donnée, renvoie la valeur du minimum m ainsi que le plus grand indice i contenant m.
2. En déduire le programme $triselection(L : list) \longrightarrow list$ qui pour toute liste L donnée, renvoie une liste triée contenant les valeurs de L.

Exercice 4 (tri par insertion). [ ]

1. Dans le langage Python, construire la fonction $insertion(L : list, x : float) \longrightarrow list$ qui pour tout couple (L, x) donné, insère l'élément x dans la liste **déjà triée** L.
2. En déduire le programme $triinsertion(L : list) \longrightarrow list$ qui pour toute liste L donnée, renvoie une liste triée contenant les valeurs de L.

Exercice 5 (tri rapide ou *quick sort*). [ ]

Le principe du tri rapide, c'est qu'il repose sur le principe de **diviser pour régner** : on divise la tâche en appelant notre algorithme sur des sous-listes de ses données.

Concrètement, considérons une liste L de n nombres réels. On choisit un élément pivot, par exemple $L[m]$ avec $m = n//2$, de la liste initiale, de l'enlever, puis de constituer deux sous-listes :

- $L1$ constituée des éléments de L inférieurs ou égaux à $L[m]$
- $L2$ constituée des éléments de L strictement plus grands que $L[m]$

On trie alors récursivement chacune des sous-listes et on rassemble le tout.

Attention, comme il s'agit d'un algorithme récursif, **il faudra traiter la condition d'arrêt de ces appels récursifs**, c'est à dire lorsqu'une telle liste n'a pas d'élément ou n'est constituée que d'un seul élément.

1. Dans le langage Python, construire le programme $trirapide : (L : list) \longrightarrow list$ qui renvoie la liste des éléments de L triée par ordre croissant.
2. Justifier que le nombre d'appels récursifs est nécessairement fini, ce qui assurera la terminaison de notre programme.

Remarque On pourra retenir que la complexité des tris récursifs présentés dans ce TD (***quick sort*** et ***merge sort***) ne sont pas simples à calculer en raison de ce principe récursif. Par exemple, on présente ici **une estimation de la complexité en nombre d'opérations pour le tri rapide** :

on note encore $C(n)$ le nombre de comparaisons nécessaires pour trier la liste L de longueur n. Dans ce cas, pour un pivot donné, il y a $n-1$ comparaisons pour constituer les listes $L1$ et $L2$, puis on trie ces nouvelles listes...

- **Dans le pire des cas**, on peut considérer que ce pivot ne nous a pas permis de diviser notre problème, c'est à dire que tous les autres éléments ont été placés dans une des deux listes. Et ainsi, on a dans le pire des cas :

  $$C(n) = n - 1 + C(n-1)$$

  Avec cette relation de récurrence et la condition initiale $C(1) = 0$, on en déduit en sommant les égalités et par télescopage, que pour tout $n \in \mathbb{N}^*$,

  $$C(n) = \frac{(n-1)n}{2} = O(n^2)$$

- **Dans le meilleur des cas**, on a réussi avec le pivot choisi à séparer "équitablement" les éléments de la liste initiale. Ainsi,
  * si $n = 2p+1$, alors on a $C(n) = n - 1 + 2C(p)$.
  * si $n = 2p$, alors on a $C(n) = n - 1 + C(p) + C(p-1)$.

Ici, il n'est pas simple d'obtenir le nombre exact de comparaisons. Cependant, on peut montrer qu'il existe un plus grand entier k tel que :

$$2^k \leq n < 2^{k+1} \quad (*)$$

et ainsi, **on va estimer $C(n)$ par encadrement à l'aide des coûts $C(2^k)$ et $C(2^{k+1})$.**

On a pour tout $k \in \mathbb{N}^*$,

$$C(2^k) = 2^k - 1 + C(2^{k-1}) + C(2^{k-1} - 1) \simeq 2^k - 1 + 2.C(2^{k-1})$$

et en notant $x_k = C(2^k)$, on peut voir ici une relation de la forme : $x_k = 2^k - 1 + 2x_{k-1}$. Par récurrence, on peut montrer alors que pour tout $k \in \mathbb{N}^*$,

$$x_k = (k-1)2^k + 1$$

Finalement, on obtient un encadrement de la complexité :

$$C(2^k) \leq C(n) < C(2^{k+1}) \Rightarrow (k-1)2^k + 1 \leq C(n) < k2^{k+1} + 1$$

et donc, quand $n \longrightarrow +\infty$, $k \longrightarrow +\infty$ de sorte que $C(n) = O(k2^k)$.

Pour finir, l'encadrement $(*)$ nous permet d'en déduire que $k = E(\frac{\ln(n)}{\ln(2)}) \sim \frac{\ln(n)}{\ln(2)}$ et on peut admettre grossièrement que dans le meilleur des cas :

$$C(n) = \Theta(n \ln(n))$$

Dans le pire des cas, ce tri rapide est donc aussi performant que la plupart des tris itératifs déjà rencontrés, mais à chaque étape, on peut aussi rencontrer un pivot qui accélère le tri ! Culturellement, on retiendra donc que dans le meilleur des cas, on a les complexités suivantes :

tri	complexité $C(n)$
à bulles	$O(n^2)$
par sélection	$O(n^2)$
par insertion	$O(n^2)$
par insertion dichotomique	$\Theta(n \ln(n))$, *car on a dû faire une estimation*
rapide	$\Theta(n \ln(n))$, *car on a dû faire une estimation*
fusion	$\Theta(n \ln(n))$, *car on a dû faire une estimation*

Et ainsi, on comprend mieux pourquoi ces deux derniers tris sont souvent les plus utilisés !