Option Informatique

TD $n°01$ – Implémentations de divers algorithmes de tris pour une liste

Dans cette feuille on s'intéresse aux algorithmes de tri basés sur le principe de la comparaison des éléments (il existe d'autres façon d'obtenir des algorithmes de tri, bien que non intuitives).

On peut être amené à trier diverses structures de données. Ici on s'intéresse uniquement au tri des *listes*. Une différence fondamentale entre cette structures de données et la structure de donné *tableau* en OCaml est que les listes sont des structures *immuables* alors que les tableaux sont des structures *modifiables* : on ne peut pas modifier une liste, on peut seulement créer une nouvelle liste à partir d'une liste donnée ; par contre, on peut modifier les éléments d'un tableau, de la même façon qu'on peut modifier une référence. Par conséquent, on considèrera :

- qu'un algorithme de tri d'une liste (les seuls qu'on examine dans cette feuille) est un algorithme qui retourne une nouvelle liste comportant les mêmes éléments que la première, mais triée ;
- alors qu'un algorithme de tri d'un tableau ne retourne rien mais a pour effet de bord de modifier le tableau donné en argument en le triant.

L'unité de mesure de toutes les complexités calculées sera le nombre de comparaisons d'éléments effectuées. Bien sûr, tous les algorithmes de tris nécéssitent d'autres types d'opérations (accès à un élément d'un tableau ou écriture dans un tableau dans le cas du tri d'un tableau ; accès à l'élément de tête d'une liste ou cons d'un élément et d'une liste dans le cas du tri d'une liste... pour ne citer que ceux-là), mais on ne s'intéressera ici qu'aux comparaisons.

Exercice 1. Tri par insertion (version enveloppée)

Pour trier une liste ℓ par insertion, on procède comme suit :

- Si ℓ est vide, alors elle est triée.
- Sinon, ℓ est de la forme $h :: t$. On trie récursivement t et on insère h à sa place dans la liste triée.

1. Écrire en OCaml une fonction `insere : 'a →'a list→'a list` telle que, si `l` est une liste triée alors `insere x l` retourne une liste **triée** ayant les mêmes éléments de `l`, plus l'élément `x` (à sa place).

2. En utilisant la fonction précédente, implémenter l'algorithme du tri par insertion sur les listes en OCaml : écrire une fonction `tri1 : 'a list→'a list`.

3. Montrer que dans le pire des cas, le nombre de comparaisons effectuées est de l'ordre de n^2. Qu'en est-il dans le meilleur des cas ?

Exercice 2. Tri par insertion (version terminale)

Une variante du tri par insertion est la suivante : pour trier une liste ℓ :

- On considère deux listes : la liste ℓ_1 des éléments qui sont encore à tirer, initialement ℓ, et la liste ℓ_2 des élément déjà triés, initialement [].
- On prend la tête de ℓ_1 et on l'insère à sa place dans ℓ_2 ;
- On recommence jusqu'à que ce ℓ_1 soit vide.

Bien sûr, dans la description précédente, il n'y a pas de réelle modification des deux listes, simplement des appels récursifs sur des listes modifiées.

4. Implémenter cette version de l'algorithme en OCaml. On utilisera la fonction `insere` vue dans l'exercice précédent, voire d'autres fonctions auxilliaires pour obtenir une fonction `tri2 : 'a list→'a list`.

Exercice 3. Tri par fusion

Le tri par fusion consiste, pour trier une liste, à :

- Diviser la liste en deux moitiés ;
- Trier récursivement chacune des deux moitiés ;
- Fusionner les deux moitiés triées pour reconstituer la liste triée.

Le cas de base étant celui des listes de longueur $\leqslant 1$.

Sur les listes, le tri par fusion est un algorithme fondamental du programme de l'option.

5. On souhaite écrire une fonction `scinder : 'a list → ('a list * 'a list)`, qui prend comme argument une liste ℓ de longueur n et retourne un couple (ℓ_1, ℓ_2) de listes de longueurs $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$ telles que ℓ aie les mêmes éléments que $\ell_1 @ \ell_2$.
   (a) Écrire une telle fonction `scinder` en partitionnant suivant les indices pairs et impairs. La tester et vérifier que `scinder [8;5;4;3;1]` retourne bien `([8;4;1],[5;3])`.
   (b) Variante : écrire une fonction `scinder2` qui renvoie les listes formées respectivement des $\lfloor \frac{n}{2} \rfloor$ premiers et des $\lceil \frac{n}{2} \rceil$ derniers éléments de la liste initiale. La tester et vérifier que `scinder [8;5;4;3;1]` retourne bien `([8;5],[4;3;1])`.

6. Écrire une fonction `fusionner : 'a list →'a list→'a list`, qui prend comme arguments deux listes supposées triées ℓ_1 et ℓ_2 et retourne comme résultat la liste triée ayant les même éléments que $\ell_1 @ \ell_2$.

   Par exemple `fusionner [5;7;11] [2;4;8;10]` doit retourner `[2;4;5;7;8;10;11]`.

   Comportement arbitraire si ℓ_1 ou ℓ_2 n'est pas triée.

7. Finalement, écrire une fonction `tri3 : 'a list→'a list`, qui trie une liste en utilisant l'algorithme du tri par fusion. On conseille évidemment d'utiliser les fonctions précédentes.

8. Dans le pire des cas, combien de comparaisons réalise l'algorithme pour trier une liste de longueur n ?

Exercice 4. Tri par pivot

On rappelle le principe du tri par pivot : on prend le premier élément du tableau ou de la liste à trier qu'on appelle le pivot, on parcourt le reste du tableau ou de la liste pour déterminer s'ils sont plus petits ou plus grands que le pivot, on trie récursivement la liste des éléments plus petits et celle des éléments plus grands, et enfin on raccorde les trois morceaux dans l'ordre.

9. Écrire une fonction `partition : 'a → 'a list → 'a list * 'a list`, telle que `partition k l` retourne un couple `(l1,l2)` tel que les éléments de l_1 sont les éléments de l strictement inférieurs à k et les éléments de l_2 sont les éléments de l supérieurs à k.

10. Écrire une fonction `tri4 : 'a list → 'a list` implémentant l'algorithme de tri par pivot sur les listes. On pourra sans scrupules utiliser l'opérateur de concaténation qui permet de concaténer deux listes.

Exercice 5. qui explique pourquoi le tri par pivot est appelé "tri rapide".

11. Quelle est la complexité du tri rapide si la liste à trier est déjà triée, ou si elle est triée par ordre décroissant ? (On doit trouver la même chose dans les deux cas.) Comparer à la complexité obtenue pour le tri par fusion : n'est-il pas louche que ce soit le tri par pivot qu'on appelle "tri rapide" ?

On souhaite maintenant calculer la complexité en moyenne du tri rapide. Le modèle choisi est le suivant : on imagine qu'on souhaite trier une permutation de $[1; 2; \dots; n]$ et que toutes les $n!$ permutations possibles de cette liste sont équiprobables. La complexité moyenne est donc

$$c_n = \frac{1}{n!} \sum_{s \in S_n} \mathcal{C}(s)$$

où $\mathcal{C}(s)$ est la complexité de l'algorithme lorsqu'on l'applique à $[s(1); s(2); \dots; s(n)]$.

12. Calculer C_0, C_1, C_2.

13. Dès C_3, ça devient pénible. Pour C_3, on y arrive encore : la calculer.

Pour C_n, on va devoir réfléchir un peu. On note $c_n(k)$ la complexité moyenne lorsque le premier pivot (c'est-à-dire le premier élément de la liste de départ) vaut k.

14. Pour tout entier n, exprimer C_n en fonction de n et des $c_n(k)$.

15. Pour tout entier n, exprimer $c_n(k)$ en fonction de n, de C_{k-1} et de C_{n-k}.

16. En déduire qu'on a, pour tout entier n, $nC_n = n^2 - n + 2\sum_{k=1}^{n-1} C_k$.

On pose, pour tout entier n, $u_n = \dfrac{C_n}{n+1}$.

17. Calculer $(n+1)C_{n+1} - nC_n$ et en déduire qu'on a, pour tout entier n, $u_{n+1} = u_n + \dfrac{2n}{(n+1)(n+2)}$.

18. En déduire un équivalent simple de u_n, puis un équivalent simple de C_n, puis une domination de C_n.

On pourra faire une décomposition en éléments simples.