M1 Informatique - Algorithmique et Complexité

TP 1 : Algorithmes de tri

L'objectif de ce premier TP est de réussir à (re)prendre en main la programmation fonctionnelle et le langage OCaml. Pour cela, on implémente et teste le bon fonctionnement d'algorithmes normalement bien connus : les algorithmes de tri.

Tri par insertion

Question 1. L'implémentation en OCaml du tri par insertion est donnée ci-après. Expliquez chacune des lignes du code en une phrase.

```
let rec insere_dans_liste_triee element liste =
  match liste with
  | [] -> [element]
  | elt1::reste ->
    if element < elt1 then element::liste
    else elt1::(insere_dans_liste_triee element reste) ;;

let rec tri_insertion liste =
  match liste with
  | [] -> []
  | elt1::reste -> insere_dans_liste_triee elt1 (tri_insertion reste) ;;
```

Question 2. On souhaite rendre le code précédent plus générique pour permettre le tri croissant/décroissant ou le tri de listes plus complexes (de coordonnées de points, etc.) avec une seule et même fonction.

Pour cela, la programmation fonctionnelle permet de passer en paramètre la fonction de comparaison de deux éléments de la liste (voir code ci-dessous).

```
let rec insere_dans_liste_triee element liste fct_comparaison =
  match liste with
  | [] -> [element]
  | elt1::reste ->
    if fct_comparaison element elt1
    then element::liste
    else elt1::(insere_dans_liste_triee element reste fct_comparaison) ;;

let rec tri_insertion liste fct_comparaison =
  match liste with
  | [] -> []
  | elt1::reste -> insere_dans_liste_triee elt1 (tri_insertion reste
   fct_comparaison) fct_comparaison ;;
```

1) Écrivez une fonction de comparaison de deux entiers (*cmp_croissant*) pour que *tri_insertion* retourne une liste triée par ordre croissant.
2) Même consigne pour obtenir une liste d'entiers triée par ordre décroissant (*cmp_decroissant*).
3) Écrivez une fonction de comparaison de deux coordonnées (*cmp_coo_croissant (x1,y1) (x2,y2)*) pour que *tri_insertion* retourne une liste triée par ordre croissant selon la coordonnée x.

Question 3. Vérifiez que votre code fonctionne en exécutant les tests ci-dessous.

```
let liste_a_trier = [9;0;5;9;6] ;;
let liste_coo_a_trier = [(9,1);(0,2);(5,3);(9,4);(6,5)] ;;

tri_insertion liste_a_trier cmp_croissant ;;
tri_insertion liste_a_trier cmp_decroissant ;;
tri_insertion liste_coo_a_trier cmp_coo_croissant ;;
```

Résultats attendus :

```
- : int list = [0; 5; 6; 9; 9]
- : int list = [9; 9; 6; 5; 0]
- : (int * int) list = [(0,2); (5,3); (6,5); (9,4); (9,1)]
```

Tri fusion / Merge sort

Le tri fusion (merge sort) exploite le fait que, quand deux listes sont déjà triées, il suffit de récupérer un par un le plus petit des deux premiers éléments pour obtenir une troisième liste triée. La procédure à suivre pour trier une liste est la suivante :

1) Si la liste est vide, elle est déjà triée ;
2) Si la liste contient un seul élément, elle est déjà triée ;
3) Sinon ...
   (a) on coupe la liste en deux ;
   (b) on trie chaque sous-liste ;
   (c) on les fusionne.

Pour que le tri fusion fonctionne, il est donc nécessaire de savoir : couper une liste en deux, trier une liste, fusionner deux listes. On remarque facilement l'aspect récursif de l'algorithme.

Question 4. Complétez le code OCaml des fonctions *coupe_en_2* et *fusionne* ci-dessous.

```
let rec coupe_en_2 liste =
  match ___ with
    | [] -> ___, ___
    | [_] -> ___, ___
    | elt1::elt2::reste ->
        let partie1, partie2 = ___ in
            elt1::partie1, elt2::partie2 ;;
```

```
let rec fusionne liste1 liste2 fct_comparaison =
  match liste1, liste2 with
    | [], _ -> ___
    | _, [] -> ___
    | ___, ___ ->
        if ___
        then tete1::(fusionne queue1 liste2 fct_comparaison)
        else tete2::(fusionne liste1 queue2 fct_comparaison) ;;
```

Question 5. Créez une fonction *tri_fusion* qui prend en entrée une liste (*liste*) et une fonction de comparaison (*fct_comparaison*) et qui implémente la procédure décrite précédemment.

Question 6. En vous inspirant de la question 3, vérifiez le bon fonctionnement de votre code.

Tri rapide / Quicksort

Le tri rapide (quicksort) consiste à réorganiser récursivement une liste par rapport à un élément donné, qu'on appelle le pivot. La procédure exacte pour un tri rapide croissant est la suivante :

1) Si la liste est vide, elle est déjà triée ;
2) Si la liste contient un seul élément, elle est déjà triée ;
3) Sinon ...
   (a) choisir un pivot (ici on prendra le premier élément de la liste) ;
   (b) identifier tous les éléments de la liste qui sont plus petits que le pivot ;
   (c) identifier tous les éléments de la liste qui sont égaux au pivot ;
   (d) identifier tous les éléments de la liste qui sont plus grands que le pivot ;
   (e) trier les listes des éléments plus petits et plus grands ;
   (f) concaténer à la suite les éléments plus petits triés, les éléments égaux et les éléments plus grands triés.

Question 7. En vous inspirant du travail effectué sur les tris précédents, implémentez le tri rapide en OCaml. La fonction de comparaison entre 2 éléments doit être fournie en paramètre.

Question 8. En vous inspirant de la question 3, vérifiez le bon fonctionnement de votre code.

Tri par arbre binaire

Un arbre binaire de recherche est une structure récursive composée de noeuds. Soit un noeud est vide, soit il est associé à une valeur et possède exactement deux fils (appelés fils gauche et fils droit) qui sont eux-mêmes des noeuds et peuvent donc être vides. Un noeud dont les deux fils sont des noeuds vides est une feuille de l'arbre. Le tri par arbre binaire transforme une liste en arbre binaire de recherche puis parcourt la structure de manière préfixe (de gauche à droite) pour récupérer les éléments triés. En OCaml, cela donne :

```
1 type 't arbre = Vide | Noeud of 't arbre * 't * 't arbre ;;
2
3 let tri_arbre liste fct_comparaison =
4     let arbre_binaire = List.fold_left (inserer fct_comparaison) Vide liste in
5     lire_prefixe arbre_binaire [] ;;
```

Sur la ligne 1, on déclare un nouveau type appelé *arbre* qui contient des valeurs d'un type générique *t*. Les valeurs autorisées pour les éléments de type *arbre* sont *Vide* ou les triplets de la forme *Noeud(a,b,c)* où *a* est un arbre, *b* est une valeur de type *t* et *c* est aussi un arbre.

Sur la ligne 4, la fonction *List.fold_left* permet d'appliquer la fonction *inserer* sur chaque élément de la liste en la parcourant de gauche à droite. À chaque appel, l'élément courant et le résultat du dernier appel sont passés en paramètres à la fonction *inserer*, qui reçoit donc : la fonction de comparaison, l'arbre résultat de l'appel précédent (ou un arbre vide pour le premier appel) et l'élément courant de la liste à insérer dans l'arbre.

L'opérateur *match ... with* d'OCaml peut être utilisé pour étudier la composition d'un arbre de la même manière que pour une liste :

```
let fonction_qui_utilise_un arbre =
    match arbre with
    | Vide -> (* Les instructions si l'arbre est un noeud vide. *)
    | Noeud(fils_gauche, valeur, fils_droit) -> (* Les instructions si l'arbre
   n'est pas un noeud vide. *)
```

Pour instancier un nouvel arbre :

```
(* Arbre vide. *)
let arbre_vide = Vide ;;
(* Arbre d'entiers compose d'un seul element (= feuille). *)
let arbre_feuille = Noeud(Vide, 1, Vide) ;;
(* Arbre d'entiers compose d'un seul noeud avec un fils. *)
let arbre_un_fils = Noeud(Vide, 1, Noeud(Vide, 2, Vide)) ;;
(* Arbre d'entiers compose d'un seul noeud avec deux fils. *)
let arbre_deux_fils = Noeud(Noeud(Vide, 0, Vide), 1, Noeud(Vide, 2, Vide)) ;;
```

Question 9. À partir des éléments précédents, proposez une implémentation pour la fonction *inserer* utilisée par *tri_arbre*. Dans un arbre binaire de recherche, un fils gauche est toujours associé à une valeur plus petite que son noeud parent, et inversement pour un fils droit qui est toujours associé à une valeur plus grande ou égale. La notion de "plus petit" ou "plus grand" est définie par la fonction de comparaison.

Question 10. Proposez une implémentation de la fonction *lire_prefixe* utilisée par *tri_arbre*.

Question 11. En vous inspirant de la question 3, vérifiez le bon fonctionnement de votre code.

Pour aller plus loin

Question 12. Écrivez une fonction *liste_aleatoire* qui prend en entrée deux paramètres *taille* et *max* et qui génère une liste aléatoire d'entiers compris entre 0 et *max* (en utilisant la fonction *Random.int* d'OCaml).

Question 13. Écrivez une fonction *est_triee* qui vérifie qu'une *liste* passée en paramètre est bien triée selon une fonction de comparaison *fct_comparaison*, elle aussi passée en paramètre.

Question 14. Vérifiez que toutes les fonctions de tri développées dans ce TP fonctionnent bien sur de grandes listes (taille $> 10\,000$).