Chapitre 7 - Listes, tris et hachage

Cette UE de Programmation fonctionnelle en Caml est aussi l'occasion (pour votre plus grand plaisir, oui...oui...) de revoir des concepts d'Algorithmique 1 & 2 et de Structures de données.

1. Algorithmes de tri

Le tri est un bon exemple de problème pour lequel il existe de nombreux algorithmes très différents des uns des autres. C'est parfait pour s'exercer à manipuler les listes en Caml !
On se limitera au cas des listes d'entiers.

1.1. Tri naïf (par sélection du minimum)

Cette première méthode consiste à calculer le minimum de la liste L pour venir le placer en tête de liste, et à trier récursivement le reste de la liste.

Exercice . TP - Exercice 1 - Tri naïf

1. Écrire une fonction `min : 'a list -> 'a` permettant de calculer le minimum d'une liste non vide.
2. Écrire une fonction `enleve : 'a * 'a list -> 'a list` qui enlève un élément d'une liste (sachant qu'il y est présent).
3. En déduire une fonction `naif : 'a list -> 'a list` qui met en œuvre l'algorithme de tri naïf.

1.2. Tri par insertion

C'est l'algorithme de tri du joueur de carte : on considère les éléments un par un et on les insère dans une liste annexe, initialement vide et qui doit toujours être maintenue triée.

Exercice . TP - Exercice 2 - Tri par insertion

1. Écrire une fonction `insertion : 'a * 'a list -> 'a list` permettant d'insérer un élément dans une liste triée.
2. Écrire la fonction `tri_insere : 'a list -> 'a list` qui met en œuvre l'algorithme de tri par insertion.

1.3. Tri par fusion

Le tri par fusion (ou *merge sort*) consiste à partager la liste à trier de taille n, en deux listes de taille $\frac{n}{2}$, à trier chacune d'elle et à fusionner les deux listes triées.
Quelle stratégie algorithmique est ici utilisée ?

Exercice . TP - Exercice 3 - Tri fusion

1. Écrire une fonction `divise : 'a list -> 'a list * 'a list` permettant de diviser une liste en deux.
2. Écrire une fonction `fusion : 'a list * 'a list -> 'a list` permettant de fusionner deux listes triées.
3. En déduire une fonction `tri_fusion` qui met en œuvre l'algorithme de tri fusion.

ALGO Quelle stratégie algorithmique utilise-t-on pour ce tri fusion ? Exprimer la complexité par une suite récurrente. Quel théorème permet de donner l'ordre de grandeur de cette complexité ?

1.4. Tri à bulle

Il s'agit de parcourir à plusieurs reprises la liste à trier par couple d'éléments : si un couple n'est pas ordonné, on échange les deux éléments. L'algorithme s'arrête lorsqu'un parcours n'a pas provoqué d'échange.
Bien plus naturel à écrire en programmation impérative... Tant pis pour nous !

Exercice . TP - Exercice 4 - Tri à bulle

1. Écrire une fonction `parcours : 'a list * 'b -> 'a list * bool` parcourant une fois toute la liste, inversant les couples qui sont mal ordonnés et mettant à jour un booléen *inversion* valant vrai si le parcours a provoqué au moins une inversion, faux sinon.
2. Écrire une fonction auxiliaire `bulle_aux : 'a list * bool -> 'a list` permettant de répéter les parcours tant que ceux-ci provoquent des inversions
3. En déduire la fonction `tri_bulle` mettant en œuvre l'algorithme du tri bulle en provoquant l'appel initial à `bulle_aux`.

1.5. Tri rapide

Le principe du tri rapide est le suivant : si la liste L possède au moins deux éléments, on considère son premier élément que l'on appelle le pivot p. On partitionne alors L en deux sous-listes L_1 contenant les éléments de L strictement inférieurs à p, et L_2 contenant les éléments de L supérieurs à p. On poursuit récursivement les tris de L_1 et L_2.

Exercice . TP - Exercice 5 - Tri rapide

1. Écrire une fonction `partition : 'a * 'a list -> 'a list * 'a list` qui, à un pivot et une liste donnés, associe les deux sous-listes définies comme décrit plus haut.
2. En déduire une fonction `quick : 'a list -> 'a list` mettant en œuvre l'algorithme de tri rapide.

ALGO De quelle stratégie algorithmique s'agit-il ici ?

2. Recherche efficace et table de hachage

Dans cette section, on s'intéressera aux méthodes de recherche dans une liste d'éléments. En pratique, une telle liste peut contenir énormément d'éléments et les opérations élémentaires de recherche, d'ajout, de suppression d'un élément doivent être effectuées le plus efficacement possible.

Exercice . TP - Exercice 6

Dans cet exercice toutefois, nous testerons nos fonctions sur une liste plutôt courte, représentant un annuaire téléphonique : les éléments sont donc des couples constitués d'une chaîne de caractère (le nom) et d'un entier (le numéro de la ligne).

```
#let annuaire = [("Claude", 1785) ; ("Andrée", 6949) ;("Antoine", 1386) ;
("Françoise", 2638) ; ("Pascal",1009) ; ("Jean",2066) ; ("Ginette", 5250) ;
("Julien", 8043) ;("Pierre",4773) ; ("Paul",3367) ; ("Cécile",5843)] ; ;
```

Recherche dans une liste quelconque

Commençons par considérer que la liste fournie n'est pas triée.

1. Écrire une fonction `cherche : string * (string*int) list -> string*int` permettant de trouver une personne dans la liste. On affichera son nom et son numéro de téléphone. Si cette personne n'est pas dans l'annuaire, un message d'erreur est renvoyé.

```
#cherche("Pierre", annuaire) ; ;
-  : string * int = "Pierre", 4773

#cherche("Brice", annuaire) ; ;
Exception : (Failure "Cette personne n'est pas dans l'annuaire")
```

2. Écrire une fonction `supprime : string * (string*int) list -> (string*int) list` permettant de supprimer une personne de la liste.

Tri d'une liste de couples

1. Écrire une fonction `insere : (string * int) * (string * int) list -> (string * int) list` permettant d'insérer un couple (nom, numéro) dans un annuaire supposé trié selon les noms.
2. En déduire une fonction `tri_insertion` mettant en œuvre le tri par insertion d'une liste représentant un annuaire.

```
# tri_insere (annuaire) ; ;
-  : (string * int) list = ["Andrée", 6949 ; "Antoine", 1386 ;
"Claude",1785 ; "Cécile", 5843 ; "Françoise",2638 ; "Ginette", 5250 ;
 "Jean", 2066 ; "Julien", 8043 ; "Pascal", 1009 ; "Paul", 3367 ;
  "Pierre", 4773]
```

Utilisation de tables de hachage Le principe du hachage permet d'accélérer le temps d'accès à un élément donné, ce qui est particulièrement utile lorsque la liste est de grande taille.
Le principe est le suivant : nous allons associer un entier appelé "clé" aux différents éléments de la liste, puis on va découper la liste en plusieurs listes regroupant des éléments ayant la même clé.
Nous appelons *table de hachage* le n-uplet $(L_0,L_1,...,L_p)$ constitué de ces différentes listes, où la liste L_i contient tous les éléments de clé i.
La recherche dans une table de hachage se fait alors en deux temps : si on cherche l'élément a dans la liste L, nous allons calculer la clé i de a, puis chercher a dans la liste Li (plus courte que L) des éléments de L de clé i.

1. Nous allons associer un entier (la clé) au nom de la personne de la manière suivante : On calcule la longueur du nom, puis on prend le reste modulo 4 de cette longueur. Ainsi, la clé est toujours un entier compris entre 0 et 3.
   Écrire la fonction `long : string -> int` donnant la longueur d'un mot.
2. En déduire la fonction `clé : string -> int` permettant de calculer la clé associée à un nom.
3. Écrire une fonction `ajoutn : int * 'a * 'a list list -> 'a list list` qui, à un entier n, un élément a et une table de hachage (représentée par une liste de listes), ajoute l'élément a à la liste d'indice n.
4. Écrire une fonction `ajoute : 'a * 'a list list -> 'a list list` qui à un élément a et une table de hachage, ajoute a à la liste correspondant à sa clé.

   ```
   let init=[[] ;[] ;[] ;[] ;[] ;[]] ; ;
   ajoute(("Pierre",6547),init) ; ;
   -  : (string * int) list list = [[] ; [] ; ["Pierre", 6547] ; []]
   ```

5. Écrire une fonction `hachage : (string * int) list -> (string * int) list list` qui, à partir d'un annuaire, construit la table de hachage correspondante.

   ```
   #let table=hachage(annuaire) ; ;
   -  : (string * int) list list = [["Jean", 2066 ; "Paul", 3367] ;
    ["Françoise", 2638] ; ["Claude", 1785 ; "Andrée", 6949 ; "Pascal", 1009 ;
     "Julien", 8043 ; "Pierre", 4773 ; "Cécile", 5843] ;
   ["Antoine", 1386 ; "Ginette", 5250]]
   ```

6. Écrire une fonction `rechercheTable : string * (string * int) list list -> bool` qui détermine si la personne p est présent dans la table de hachage. Vous pourrez écrire une ou plusieurs fonctions auxiliaires.

7. Écrire une fonction `rechercheValeur : string * (string * int) list list -> int` qui détermine le numéro d'une personne p que l'on suppose présente dans la table de hachage.