TP 4 : OCaml. Programmation récursive

Exercice 1 – Fonctions simples en OCaml

▶ **Question 1** Définir des fonctions répondant aux spécifications suivantes :

1. `puissance : int -> int -> int` tel que `puissance x n` renvoie x^n pour $n \geq 0$.
2. `somme_n_puissance_k : int -> int -> int` tel que `somme_n_puissance_k n k` est évalué en $\sum_{j=0}^{n} j^k$. *On supposera que* $k > 0$.
3. `binom : int -> int -> int` tel que `binom k n` renvoie le coefficient binomial $\binom{k}{n}$, avec $0 \leq k \leq n$. *On se contentera d'une version naïve implémentant la formule du triangle de Pascal pour l'instant, on verra bientôt comment faire mieux* 😄
4. `somme_list : float list -> float` qui renvoie la somme d'une liste de flottants
5. `croissant : 'a list -> bool` déterminant si une liste est croissante (au sens de l'opérateur <=)
6. `compare_tailles : 'a list -> 'b list -> int` qui renvoie un entier strictement positif si la deuxième liste est plus grande que la première, 0 si elles sont de même taille et un entier négatif sinon

Exercice 2 – Algorithmes de tri — tri par insertion

L'un des problèmes fondamentaux de l'informatique est le problème du tri, énoncé comme suis :

Problème du tri

Entrée : une séquence d'éléments $T = [T_0, \dots, T_{n-1}]$
Sortie : une permutation de la séquence T triée dans l'ordre croissant

Remarquez que l'on ne précise ni le type des éléments dans la séquence ni l'ordre avec lequel on veut les trier : c'est parce que les premiers algorithmes de tris que l'on va étudier sont généraux, dans le sens où ils fonctionnent pour des données entières, flottantes, des couples, des structures plus complexes...La seule exigence est que cette relation d'ordre soit totale, c'est-à-dire qu'on a soit $x = y$, soit $x < y$, soit $x > y$. Dans ce TP, on s'intéresse à deux tris "généraux" : le tri par insertion et le tri fusion. On représentera pour l'instant les séquences par des listes, et on utilisera la relation d'ordre prédéfinie en OCaml (par les opérateurs `<= >= = <>`).

▶ **Question 1** Écrire une fonction `insere : 'a list -> 'a -> 'a list` telle que si `l` est une liste triée, `insere l x` est une liste où `x` a été inséré à la bonne place pour qu'elle soit triée. *Par exemple, l'appel* `insere [1;5;6;10] 4` *sera évalué en* `[1;4;5;6;10]`.

▶ **Question 2** Écrire une fonction `tri_insertion : 'a list -> 'a list` triant la liste en entrée `l` selon la stratégie suivante :

— Si `l = []`, elle est déjà triée et on renvoie `[]`
— Si `l = h :: t`, on trie récursivement `t` puis on insère `h` dans `t`

▶ **Question 3** Dans le pire cas et dans le meilleur cas, combien de comparaisons entre éléments sont faites par la fonction `insere` ? Par la fonction `tri_insertion` ? *On ne compte pas les filtrages comme des comparaisons.*

Exercice 3 – Algorithmes de tri — tri fusion

Le tri fusion est un tri particulièrement adapté à être appliqué à des listes, la version pour tableaux est moins naturelle. Le principe est décrit dans la figure suivante :

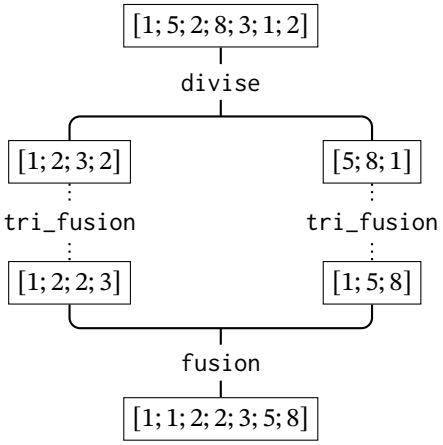

FIGURE 1 – Tri fusion appliqué à [1; 5; 2; 8; 3; 1; 2]

On commence par diviser la liste [1; 5; 2; 8; 3; 1; 2] en deux listes de taille similaires. Ensuite, on utilise deux appels récursifs à `tri_fusion` pour trier les deux sous-listes obtenues, et enfin on fusionne les deux listes pour retrouver la liste triée.

▶ **Question 1** Écrire une fonction `divise : 'a list -> ('a list * 'a list)` telle que `diviser l` renvoie un couple `(lg, ld)` de liste d'éléments de `l` telles que `|l| = |lg| + |ld|` et $||lg| - |ld|| \leq 1$. Par exemple, une valeur acceptable pour l'appel `divise [1;2;3;4;5]` serait `([5;2;4],[3;1])`.

▶ **Question 2** Écrire la fonction `fusionne : 'a list -> 'a list -> 'a list` suivant la stratégie décrite plus haut pour obtenir une liste triée contenant les éléments des deux arguments. *Par exemple, l'appel* `fusionne [1;5;6;10] [2;6;8]` *sera évalué en* `[1;2;5;6;6;8;10]`.

▶ **Question 3** Déduire de ces deux fonctions une fonction `tri_fusion : 'a list -> 'a list` renvoyant l'entrée triée avec la stratégie décrite plus haut. Combien de comparaisons sont nécessaires au mieux et au pire ?

Exercice 4 – Représentation des polynômes en OCaml (optionnel)

Le but de cet exercice est de représenter des polynômes en OCaml et de les manipuler. On se limitera ici aux polynômes à coefficients entiers.
Dans cet exercice, on choisit de représenter un monôme (de la forme pX^k) comme un type enregistrement et un polynôme comme une liste de monômes.

```OCaml
type monome = { coeff : int ; degre : int } (** représente coeff × X^degre *)
type polynome = monome list
```

La liste `[{ coeff = pk; degre = nk }; ... ; { coeff = p1; degre = n1 }; { coeff = p0; degre = n0 }]` représente le polynôme suivant :

$$P = p_k X^{n_k} + \cdots + p_1 X^{n_1} + p_0 X^{n_0} \tag{0.1}$$

dans lequel on suppose que pour tout $i \in [\![0, k]\!]$, $p_i \neq 0$ et $n_k > \cdots > n_1 > n_0 \geq 0$.
Par exemple, on représente le polynôme $1 - X + 3X^5$ par :

```OCaml
let exemple_1 = [
  {coeff = 3; degre = 5};
  {coeff = -1; degre = 1};
  {coeff = 1; degre = 0};
]
```

Le polynôme nul est simplement représenté par une liste vide `[]`. L'intérêt de cette représentation est qu'elle permet de représenter efficacement les polynômes creux, c'est-à-dire ayant très peu de coefficients non nuls.

▶ **Question 1** Représenter le polynôme $3X^2 + 2X - X^{10}$ par une variable de type `polynome`.

▶ **Question 2** Écrire une fonction `polynome_valide : polynome -> bool` vérifiant qu'un élément de type `polynome` est bien une représentation correcte d'un polynôme, avec les conditions décrites plus haut : les degrés sont positifs et décroissants, et les coefficients sont non nuls.

Dans la suite, on considèrera que les fonctions prennent en entrée des entrées valides. Pour les questions suivantes, je vous conseille d'appliquer la méthode décrite pendant le cours sur la récursivité : toujours essayer de décrire la valeur de retour attendue en fonction de solutions de sous-problèmes.

▶ **Question 3** Écrire une fonction déterminant le degré d'un polynôme `degre : polynome -> int`. Le degré du polynôme nul sera défini à -1.

▶ **Question 4** Écrire une fonction `evaluation : polynome -> int -> int` tel que pour un polynôme P représenté par `p` et un entier `x`, `evaluation p x` renvoie $P(x)$. *On pourra commencer par écrire une fonction* `puissance_entiere : int -> int -> int` *calculant la puissance de deux entiers.*

▶ **Question 5** Écrire une fonction `composition_carre : polynome -> polynome` qui à un polynôme P renvoie le polynôme $P(X^2)$. En reprenant l'exemple de l'Équation (0.1), cela donnera :

$$P(X^2) = p_k (X^2)^{n_k} + \cdots + p_1 (X^2)^{n_1} + p_0 (X^2)^{n_0} = p_k X^{2n_k} + \cdots + p_1 X^{2n_1} + p_0 X^{2n_0}$$

▶ **Question 6** Généraliser cette fonction en `composition : polynome -> int -> polynome` tel que `composition p n` renvoie le polynôme $P(X^n)$.

On va maintenant calculer les opérations élémentaires sur les polynômes : somme, produit, dérivée. *On fera attention à ce que le résultat soit bien un polynôme valide : pas de coefficients non nuls et les degrés sont positifs et décroissants.*

▶ **Question 7** Écrire une fonction `somme : polynome -> polynome -> polynome` calculant la somme de deux polynômes.

▶ **Question 8** Écrire une fonction `derivee : polynome -> polynome` calculant la dérivée d'un polynôme.

▶ **Question 9** Écrire une fonction `produit : polynome -> polynome -> polynome` calculant le produit de deux polynômes.