1 Courses en équipes

Une course a eu lieu entre trois équipes : l'équipe verte `"V"`, l'équipe jaune `"J"` et l'équipe rouge `"R"`. Les coureurs courent tous en même temps. Les équipes sont ensuite classées comme suit :

— chaque coureur obtient le nombre de points égal à son rang d'arrivée ;
— le score d'une équipe est la somme des points de tous ses coureurs ;
— la meilleure équipe est naturellement l'équipe qui a obtenu le moins de points.

On définit le type `coureur` comme suit :

```
type coureur = {dossard:int ; temps:float ; equipe:string};;
```

Une course est alors une liste de `coureur`.

1.1 À partir d'une course triée

Dans cette section, on suppose qu'on dispose d'une course triée dans l'ordre des arrivées, c'est-à-dire par temps de course croissants.

```
let course = [{dossard=4000;temps=32.45;equipe="V"};
              {dossard=4001;temps=33.5;equipe="J"};
              {dossard=4005;temps=35.6;equipe="R"};
              {dossard=4003;temps=37.75;equipe="V"};
              {dossard=4002;temps=37.95;equipe="R"};
              {dossard=4008;temps=38.45;equipe="V"};
              {dossard=4007;temps=39.5;equipe="J"};
              {dossard=4004;temps=41.45;equipe="R"};
              {dossard=4006;temps=42.45;equipe="J"}];;
```

1) Écrire une fonction `liste_equipes` de signature :
   `coureur list -> coureur list * coureur list * coureur list` qui renvoie le triplet des listes des coureurs des trois équipes (dans l'ordre "V,J,R").
2) Écrire une fonction `compter_points` de signature `coureur list -> int * int * int` qui compte le nombre de points de chaque équipe, c'est-à-dire qui somme les classements de chacun de ses coureurs.
3) Écrire une fonction `classement_equipe` de signature `coureur list -> (string * int) list` qui renvoie le classement des équipes sous forme d'une liste de couple (nom d'équipe,score).

1.2 Nouvelle course

Une seconde course a eu lieu. On va devoir mettre à jour les temps des coureurs. Dans cette section, on suppose que l'on dispose de la liste des nouveaux temps des coureurs couplés à leurs numéros de dossards.

```
let nt = [(4000,32.4);(4001,35.8);(4002,38.2);(4003,42.5);(4004,37.3);
                                                (4005,39.7);(4006,46.4);(4007,35.4);(4008,37.6)];;
```

4) Modifier le type `coureur` pour que l'on puisse modifier le temps de course d'un coureur.
5) Écrire une fonction `maj` de signature `coureur list -> (int * float) list -> unit` qui met à jour les coureurs en modifiant leurs temps de course.
6) Les coureurs dans la course ne sont plus dans l'ordre d'arrivée. Pour pouvoir appliquer les fonctions précédentes, nous allons écrire une fonction qui renvoie la liste des coureurs dans l'ordre d'arrivée. Procédons en deux étapes :
   a) Écrire une fonction `extrait_min` dont la signature est `coureur list -> coureur * coureur list` tel que `extrait_min course` renvoie le coureur `c` dont le temps de course est le plus petit et une liste contenant les mêmes éléments que `course` sauf le coureur `c`.
   Indication : un seul parcours de la liste peut être suffisant car l'ordre des éléments de la liste renvoyée n'est pas important.

b) Utiliser la fonction précédente pour écrire une fonction `trie` de signature `coureur list -> coureur list` qui renvoie la liste des coureurs dans l'ordre d'arrivée.

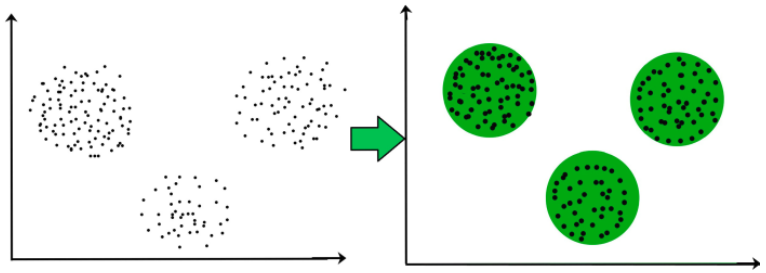
Indication : le principe est simple, à chaque étape, on met l'élément minimal en tête et on fait un appel récursif pour trier le reste de la liste.

7) En appliquant les fonctions de la section précédente, on trouve que deux équipes sont premières exæquo. Adapter ces fonctions pour départager les équipes au goal average, c'est-à-dire en sommant les temps de courses des membres de chaque équipe. Écrire une fonction `classement_equipe_bis` de signature `coureur list -> (string * (int * float)) list` qui renvoie le classement des équipes ainsi obtenu.

Indication : on pourra écrire une fonction `compter_temps` *de signature* `coureur list -> float * float * float` *qui compte les temps de course cumulés de chaque équipe. Puis on pourra utiliser l'ordre lexicographique sur les tuples en Ocaml.*

2 Algorithme des k-moyennes

On se propose d'implémenter un algorithme de clustering non-supervisé. L'objectif est le suivant : à partir d'un ensemble de données, identifier des critères pour classer les données par ressemblance.



On dit que le clustering est non-supervisé parce qu'on n'a pas d'information particulière sur les données. Il n'y a pas de bonne réponse. Par exemple, lorsque l'on commence un puzzle qui contient beaucoup de pièces, il est souvent pertinent de regrouper les pièces en fonction de leurs couleurs respectives (après avoir construit le bord évidemment !). C'est du clustering non-supervisé.
Pour la suite, on s'intéresse au clustering non-supervisé de points du plan par l'algorithme des k-moyennes. Chaque donnée sera donc un couple de flottants.

2.1 Présentation de l'algorithme des k-moyennes

Entrées de l'algorithme : un ensemble de points du plan et un nombre k de clusters (groupes de données) à construire.

Ce que calcule l'algorithme : l'algorithme des k-moyennes détermine k points appelés centroïdes et représentant chacun un cluster. Un point sera alors considéré comme appartenant au cluster dont le centroïde est le plus proche de ce point.

Principe du calcul : l'algorithme fonctionne itérativement.

- **initialisation :** choisir k points dans l'ensemble des données. Ce sont les centroïdes provisoires.
- **itération :**
  - affecter chaque point au cluster du centroïde dont il est le plus proche
  - recalculer le centroïde de chaque cluster (moyenne des coordonnées des points du cluster)

Remarques :

1) plusieurs conditions d'arrêt sont possibles (nombre d'itérations, stabilisation des centroïdes, ...)
2) le résultat de l'algorithme peut être différent en fonction de l'initialisation des centroïdes.

2.2 Implémentation

On définit les types `point` et `centroide` ainsi :
```
type point = { x : float ; y : float } ;;
type centroide = { mutable xm : float ; mutable ym : float } ;;
```
Contrairement aux points de l'ensemble à classer, les centroïdes sont appelés à être déplacés. Les champs des centroïdes sont donc mutables.

Plusieurs tableaux vont être utilisés dans l'implémentation de l'algorithme.
- L'ensemble des points `e` donnés en entrée de l'algorithme est de type `point array`.
- L'ensemble des centroïdes `centroides` calculés itérativement est de type `centroide array` de longueur `k`.
- Un tableau d'entiers `clus` va également être utilisé pour stocker l'indice du cluster auquel chaque point appartient. (`clus.(i) = j` si `e.(i)` appartient au cluster `j` dont le centroïde est `centroides.(j)`).

1) Écrire une fonction `dist : point -> centroide -> float` qui calcule la distance euclidienne entre un point et un centroïde.
2) Écrire une fonction `centroides_init : point array -> int -> centroide array` telle que l'appel suivant `centroides_init e k` renvoie un tableau de centroïdes initialisé avec les coordonnées des `k` premiers points de `e`.
3) Écrire une fonction `maj_centroides : point array -> centroide array -> int array -> -> unit` telle que l'appel `maj_centroides e centroides clus` met à jour `centroides`, les coordonnées des centroides de chaque cluster, avec la moyenne des coordonnées des points de ces clusters. Attention à ne pas mettre à jour un centroïde qui correspond à un cluster vide.
4) Écrire une fonction `maj_clus : point array -> centroide array -> int array -> int -> unit` telle que l'appel `maj_clus e centroides clus` met à jour le tableau `clus` avec l'indice du centroïde dont il est le plus proche.
5) Écrire une fonction `kmoyennes : point array -> int -> int -> int array * centroide array` telle que l'appel `kmoyennes e k nb_it` applique l'algorithme des `k`-moyennes à l'ensemble des points stockés dans `e` avec `nb_it` itérations et renvoie le tableau `clus` et le tableau `centroides`.

   Tester la fonction avec `k=4` et l'ensemble suivant :

   `let e2 =` $[|\{x=3.;y=3.\};\{x=3.;y=-3.\};\{x=-3.;y=3.\};\{x=-3.;y=-3.\};$
   $\{x=3.;y=4.\};\{x=3.;y=-4.\};\{x=-3.;y=4.\};\{x=-3.;y=-4.\};$
   $\{x=4.;y=3.\};\{x=4.;y=-3.\};\{x=-4.;y=3.\};\{x=-4.;y=-3.\}|]$`;;`
6) Écrire une fonction `centroides_init_rand : point array -> int -> centroide array` telle que l'appel `centroides_init_rand e k` renvoie un tableau de centroïdes initialisé avec les coordonnées de `k` points distincts de `e` sélectionnés aléatoirement.

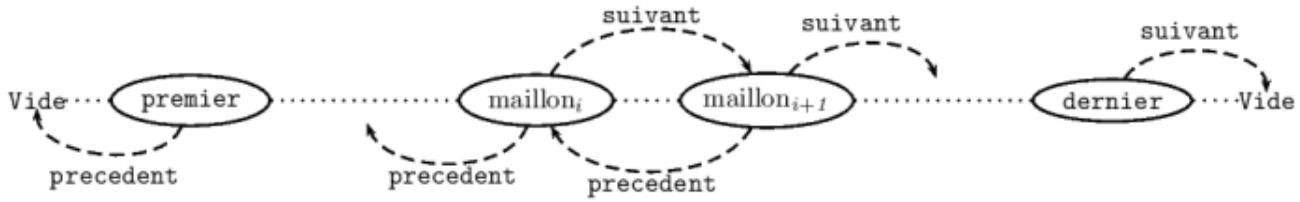
   Tester plusieurs fois la fonction `kmoyennes` avec cette initialisation aléatoire. Les résultats doivent parfois être différents.

3 Listes doublement chaînées

Dans cet exercice nous allons essayer de programmer une file de façon efficace : chaque opération élémentaire (`nouvelle_file`, `enfile`, `file_vide` et `defile` se faisant en temps constant).

Une *liste doublement chaînée* est une structure de données où on peut insérer des éléments (de type `'a`) séquentiellement. Elle peut servir à définir une structure de *file* (*FIFO : first in first out*) dans laquelle les éléments stockés sont extraits dans l'ordre où on les avait insérés.
Le but est d'obtenir sur une file les fonctionnalités suivantes :
- `est_vide : 'a file -> bool` qui permet de savoir si une file est vide
- `enfile : 'a -> 'a file -> unit` qui permet de stocker un nouvel élément dans une file (cette fonction est à effet de bord, c'est–à–dire qu'elle modifie la file)
- `defile : 'a file -> 'a` qui retourne l'élément en tête de file et modifie la file de façon à supprimer cet élément
- `cree_file : unit -> 'a file` qui crée une nouvelle file, initialement vide

Les éléments constitutifs d'une liste doublement chaînée sont définis par le type `maillon`.
Pour pouvoir gérer les extrémités de files, il existe deux types de maillons : les maillons définis par le constructeur `Interne` stockent les éléments dans le champ `etiquette`, ils possèdent un champ `precedent` qui pointe sur le maillon précédent dans la chaîne, et un champ `suivant` qui pointe sur le maillon suivant. Les maillons définis par le constructeur `Vide` se trouvent aux deux extrémités de la chaîne.
Enfin, le type `file` possède deux champs qui pointent respectivement sur le premier nœud interne et sur le dernier nœud interne de la file (ou qui pointent sur le nœud `Vide` si la file est vide).



On doit garantir que pour deux maillons internes `m1` et `m2`,

$$\texttt{m1.suivant} = \texttt{m2} \iff \texttt{m2.precedent} = \texttt{m1}$$

On définit donc les types :

```
type 'a interne = { mutable precedent : 'a maillon;
                    etiquette : 'a;
                    mutable suivant : 'a maillon }
and 'a maillon =  | Vide
                  | Interne of 'a interne
and 'a file = { mutable premier : 'a maillon;
                mutable dernier : 'a maillon };;
```

1) Écrire la fonction `cree_file : unit -> 'a file` qui crée une nouvelle file, initialement vide. On pourra compléter le code suivant :

```
let cree_file () = { premier = Vide;
                     ..... = ......};;
```

2) Écrire la fonction `est_vide : 'a file -> bool` qui permet de savoir si une file est vide.
Cette opération doit être effectuée en temps constant ($O(1)$).

3) Il s'agit maintenant d'écrire la fonction `enfile : 'a -> 'a file -> unit` qui permet de stocker un nouvel entier dans une file. Cette fonction modifie la file en ajoutant un nouveau maillon à la fin de la file. Il ne pas oublier le cas où la file initiale est vide. Cette opération doit être effectuée en temps constant ($O(1)$).

Voici une possibilité de la coder : compléter les parties manquantes.

```
let enfile x f =
  let avantdernier = f.dernier in
  let m = Interne {precedent = avantdernier; etiquette = x; suivant = Vide} in
    begin
      match avantdernier with
        | Vide -> f.premier <- .........
        | Interne i -> ..................
    end;
    f.dernier <- ..........;
    ()
;;
```

4) Écrire une fonction `affiche : int file -> unit` qui permet d'afficher les étiquettes des maillons de la file, dans le cas où ces étiquetees sont des entiers.
Par exemple si la file contient les maillons d'étiquette 3, 6, 2 l'affichage pourra être :

```
3 - 6 - 2 - Fin
```

On pourra utiliser les fonctions :

```
print_int : int -> unit          (* affichage d'un entier *)
print_string : string -> unit    (* affichage d'une chaine de caracteres *)
print_newline : string -> unit   (* affiche une chaine et retour a la ligne *)
```

Tester grâce à cette fonction les différentes fonctionnalités implantées.

5) Écrire la fonction `defile : file -> 'a` qui retire l'élément en tête de file (donc modifie la file) et retourne la valeur de l'étiquette du maillon qu'on vient de retirer. En cas de file initialement vide une exception sera levée.

   Cette opération doit être effectuée en temps constant ($O(1)$).

6) Tester les fonctions précédentes en créant une file vide, puis en enfilant successivement les entiers 5, 3, 8. Défiler ces entiers un à un.