

L'évaluation attachera une grande importance à la clarté des justifications.  
Vous pouvez utiliser les fonctions du module **List**. Le barème est indicatif.

**Exercice 1 — 6pts.**

1. Écrire un type couleur ayant trois valeurs : Blanc, Rouge et Rose.
2. On définit le type region comme suit :

```
type region = Medoc | Graves | Alsace | Beaujolais | Touraine | Bourgogne
```

En utilisant les types couleur et region, définir un type vin permettant de représenter un vin par

- sa région,
  - sa couleur, **et**
  - son millésime (c'est-à-dire, une année, de type int).
3. Écrire une fonction `bordeaux` qui prend en paramètre une liste de vins et renvoie la sous-liste des vins de Bordeaux (c'est-à-dire produits dans le Médoc ou dans les Graves) qu'elle contient.
  4. Écrire une fonction `millesimes` qui prend en paramètres une liste de vins `l` et un prédicat `p` de type `vin -> bool`, et qui renvoie la liste des millésimes des vins de `l` qui satisfont `p`. Donner un exemple d'appel de votre fonction, ainsi que le type et la valeur de son retour.

**Exercice 2 — 7pts.**

1. Écrire une fonction `distribute` de type `'a -> 'b list -> ('a * 'b) list` telle que l'appel `distribute e l`, pour un élément `e` et une liste `l = [e1; ...; en]`, retourne la liste des couples `[(e, e1); (e, e2); ...; (e, en)]`.

Exemples :

```
distribute 0 [];;
- : (int * 'a) list = []
distribute 0 [1; 2; 3];;
- : (int * int) list = [(0, 1); (0, 2); (0, 3)]
```

2. Soit maintenant la fonction `cartesian_product` définie ci-dessous :

```
let rec cartesian_product l1 l2 =
  match l1 with
    [] -> []
  | hd::tl -> (distribute hd l2) @ (cartesian_product tl l2)
```

Que retournent les appels suivants ? On indiquera le **type** et la **valeur** affichés.

- a. `cartesian_product`;;
  - b. `cartesian_product [] []`;;
  - c. `cartesian_product [1; 2; 3] [4; 5]`;;
3. De manière, générale, si `l1 = [x1; ...; xn]` et `l2 = [y1; ...; yp]` que retourne `cartesian_product l1 l2` ?
  4. Écrire une version récursive terminale de la fonction `cartesian_product`.

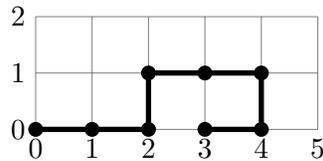
**Exercice 3 — 7pts.** Pour représenter des chemins dans le plan partant de l'origine et faisant des pas d'une unité vers le Nord (N), le Sud (S), l'Est (E) ou l'Ouest (O), on définit les types suivants :

```

type direction = N | E | S | O
type chemin = direction list
type point = int * int

```

Un chemin commence au point de coordonnées (0,0), et passe donc par des points de coordonnées entières. Par exemple [E;E;N;E;E;S;O] passe par (0,0), (1,0), (2,0), (2,1), (3,1), (4,1), (4,0), (3,0).



1. Écrire une fonction `arrivee` de type `chemin -> int * int` qui retourne le point d'arrivée du chemin passé en paramètre. Par exemple, `arrivee []` doit retourner (0,0) et `arrivee [E;E;N;E;E;S;O]` doit retourner (3,0).
2. Écrire une fonction `coordonnees` de type `chemin -> point list` qui renvoie la liste des points visités le long du chemin. Notez que cette liste a un élément de plus que le nombre de pas dans le chemin. En particulier `coordonnees []` renvoie la liste [(0, 0)]. De même, `coordonnees [E;E;N;E;E;S;O]` renvoie la liste [(0, 0); (1, 0); (2, 0); (2, 1); (3, 1); (4, 1); (4, 0); (3, 0)].
3. En utilisant la fonction `coordonnees` (même si vous le l'avez pas écrite), écrire une fonction `se_recoupe` de type `chemin -> bool` qui teste si un chemin rencontre deux fois le même point. Par exemple, `se_recoupe [N;N;E;S]` renvoie `false`, alors que `se_recoupe [E;N;E;S;O;S]` renvoie `true` (on passe deux fois par le point (1,0)) et `[N;S]` renvoie aussi `true` (on passe deux fois par (0,0)).
4. Écrire une fonction `symmetrie_centrale` de type `chemin -> chemin` qui renvoie le chemin symétrique de son paramètre par rapport à l'axe des abscisses : les pas N sont changés en S, les pas S sont changés en N, et les deux autres pas sont inchangés.

Une *transformation* du plan associe à un point  $(x, y)$  un autre point  $(x', y')$ . Une translation de vecteur  $(v_x, v_y)$  qui à un point  $(x, y)$  associe le point  $(x + v_x, y + v_y)$  est un exemple de transformation. On utilise des fonctions de type `point -> point` pour représenter les transformations du plan.

5. Implémenter la fonction `translation` de type `int -> int -> point -> point` telle que `translation vx vy` retourne la translation de vecteur  $(v_x, v_y)$ .

Exemples :

```

translation 2 1;;
- : point -> point = <fun>
(translation 2 1) 3 1;;
- : point = (5, 2)
(translation 2 1) 3 4;;
- : point = (5, 5)

```

6. Écrire une fonction `liste_points` de type `chemin -> int -> int -> point list` telle que `liste_points c vx vy` retourne la liste de points de `(coordonnees c)`, chacun translaté par le vecteur  $(v_x, v_y)$ .