Opérations sur les graphes T.D.5.

On définit un graphe par ses listes d'adjacence ; à chaque sommet, on associe la liste ordonnée dans l'ordre strictement croissant de ses voisins. Sauf mention contraire, on considère des graphes orientés.

```
type graphe = int list array;;
```

1. Le graphe orienté associé au graphe représenté est défini par :

```
let (g : graphe) = [|[1;2];[2];[3];[4];[1;2;5];[]|];;
```

et le graphe non orienté associé par :

```
let (gd : graphe) = [|[1;2];[0;2;4];[0;1;3;4];[2;4];[1;2;3;5];[4]|];;;;
```

2. Les prédécesseurs du sommet `k` dans le graphe `g` sont les sommets `i` pour lesquels `k` est un voisin de `i` . On effectue donc une boucle sur `i` en commençant par la plus grande valeur possible pour `i` afin que les prédecesseurs de `k` se retrouvent classés dans l'ordre croissant.

```
let predecesseurs (g : graphe) k = let n = Array.length g  and l = ref [] in
    for i = n-1 downto 0 do
        if List.mem k g.(i) then l :=  i::!l
    done;
    !l;;
```

Si `g1` est le graphe obtenu à partir du graphe orienté `g` en retournant toutes les arêtes, on a pour `i,j` sommets l'équivalence :
(`j` voisin de `i` dans `g1`) si et seulement si (`j` prédecesseur de `i` dans `g`) d'où

```
let transposition (g:graphe) =
    let n = Array.length g in
    let (g1:graphe) = Array.make n [] in
    for k = 0 to n-1 do
        g1.(k) <- predecesseurs g k
    done;
    g1;;
```

3. On écrit tout d'abord une fonction qui étant donné un tableau de couleurs (`f.(i)` contient la couleur attribuée au sommet `i`) et une liste de sommets retourne la liste des couleurs des sommets de cette liste :

```
let rec applique f l = match l with
    |[] -> []
    |t::q -> f.(t)::(applique f q);
```

On en déduit la fonction qui teste si un tableau de couleurs `f` est un coloriage du graphe `g` :

```
let coloriage (f : int vect) (g : graphe) =
    let n = Array.length g and ok = ref true and i = ref 0 in
    while !i < n && !ok do
        if List.mem f.(!i) (applique f g.(!i)) then ok := false; incr i
    done;
    !ok;;
```

4. Redonnons tout d'abord un fonction insérant un entier `i` dans une liste strictement croissante d'entiers

```
let rec insere l i = match l with
    |t::q when t < i -> t::insere q i
    |t::q when t = i -> l
    |_               -> i::l;;
```

Notons que le dernier cas du filtrage correspond à une liste vide ou une liste dont tous les éléments sont strictement supérieurs à `i` .
La fonction `aux` ajoute le sommet `i` à toutes les listes d'adjacence du graphe `g` des sommets de la liste `l` :

```
let rec aux (g : graphe)  l i = match l with
    |[] -> ()
    |t::q -> g.(t)<-insere g.(t) i; aux g q i;;
```

La fonction `desoriente` convertit un graphe orienté en le graphe non orienté correspondant :

```
let desoriente (g : graphe)  =
    let n = Array.length g in
    for i = 0 to n-1 do aux g g.(i) i done;;
```

5. La fonction auxiliaire `totalite` est telle que `totalite i j g.(j)` retourne un booléen indiquant si pour tout $k \in [\![0, j]\!] \setminus \{i\}$ il existe un arc de k vers i dans G.

```
let puits g =
    let rec totalite i j lj = match lj with
        | _ when i = j      -> i=0 || totalite i (j-1) g.(j-1)
        | k :: q when k = i -> j=0 || totalite i (j-1) g.(j-1)
        | k :: q when k < i -> totalite i j q
        | _                 -> false
    in let n = Array.length g and b = ref false in
    for i = 0 to n-1 do
        b := b || g.(i) = [] && totalite i (n-1) g.(n-1)
    done;
    !b;;
```

6. On peut par exemple tester si j est dans la liste d'un parcours du graphe en largeur à partir du sommet i.

```
let mcc g i j =
    let n = Array.length g in
    let t = Array.make n false in
    let rec bfs li = match li with
        | [] -> []
        | x :: q when t.(x) -> bfs q
        | x :: q            -> t.(x) <- true; x :: bfs (q @ g.(x))
    in
    List.mem j bfs [j];;
```