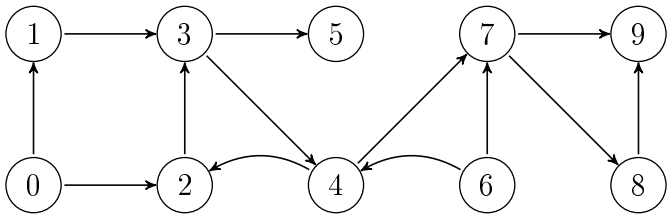
TD 5

Parcours d'un graphe

On considère le graphe suivant :



On donne les listes suivants :

```
let sommets_exemple = [0;1;2;3;4;5;6;7;8;9] ;;
let arcs_exemple = [(0,1);(1,3);(0,2);(2,3);(3,5);(3,4);(4,2)
;(4,7);(6,4);(6,7);(7,8);(7,9);(8,9)] ;;
```

1 Construction du graphe

On utilise les deux types utilisés au TD précédent :

```
type graphe_1 = bool array array ;;
type graphe_2 = int list array ;;
```

1. Écrire la fonction `construction_1` qui prend en arguments la liste des sommets et la listes des arcs et renvoies le graphe correspondant codé avec le premier type.

   `construction_1 : 'a list -> (int * int) list -> graphe_1`

2. Écrire la fonction `construction_2` qui prend en arguments la liste des sommets et la listes des arcs et renvoies le graphe correspondant codé avec le second type.

   `construction_2 : 'a list -> (int * int) list -> graphe_2`

2 Fonctions utiles

1. Écrire la fonction `appartient` qui prend en argument un élément et une liste, et renvoie le booléen correspondant à l'appartenance de l'élément à la liste.

   `appartient : 'a -> 'a list -> bool`

   (Remarque : cette fonction correspond à la fonction `List.mem`)

2. Écrire la fonction `retourne` qui prend en argument une liste, et renvoie la liste retournée.

   `retourne : 'a list -> 'a list`

   (Remarque : cette fonction correspond à la fonction `List.rev`)

3 Parcours en largeur

On utilisera le `type graphe_2` dans la suite du TD.
Le parcours en largeur (appelé BFS, pour Breadth First Search) consiste à traiter le sommet puis tous ses sommets voisins en évitant de traiter deux fois un sommet.

1. Donner la liste des sommets accessibles en partant du sommet 3 sur le graphe exemple.

2. Écrire la fonction `bfs` qui prend en argument un graphe et une liste de sommets déjà vus et une liste de sommets à voir, et renvoie la liste des sommets parcourus.

   ```
   bfs : graphe_2 -> int list -> int list -> int list
   ```

3. En déduire la fonction `parcours` qui prend en argument un graphe et un sommet, et renvoie la liste de sommets accessibles dans un parcours en largeur.

   ```
   parcours : graphe_2 -> int -> int list
   ```

4. Le parcours en largeurs traite les sommets dans l'ordre de leur distance au sommet de départ. En notant (s_i^j) les sommets du parcours où j est la distance de s_i à d, on a :

$$\left[d, \underbrace{s_1^1, s_2^1, ..., s_n^1}_{distance=1}, \underbrace{s_{n+1}^2, ..., s_m^2}_{distance=2}, ..., \underbrace{s_{m+1}^k, ..., s_p^k}_{distance=k}\right]$$

   On cherche à déterminer le chemin le plus court entre deux sommets d et a du graphe. Si a est un élément du parcours, en notant k la distance entre d et a, on a :

$$\left[d, \underbrace{s_1^1, s_2^1, ..., s_n^1}_{distance=1}, \underbrace{s_{n+1}^2, ..., s_m^2}_{distance=2}, ..., \underbrace{s_{m+1}^k, ..., a, ..., s_p^k}_{distance=k}...\right]$$

   Il suffit donc de reconstituer le parcours en partant du sommet d'arrivée.

   (a) Écrire la fonction `liste_num` qui prend en argument une liste et un entier n et renvoie la liste des couples constituer des éléments de la liste, et de l'entier n.

   ```
   liste_num : 'a list -> 'b -> ('a * 'b) list
   ```

   Exemple :

   ```
   liste_num ['a'; 'b' ; 'c'] 3;;
   - : (char * int) list = [('a', 3); ('b', 3); ('c', 3)]
   ```

   (b) Écrire la fonction `appartient_couple` qui prend un élément e et une liste de couple et renvoie le booléen correspondant au fait que e est le première élément d'un couple de la liste.

   ```
   appartient_couple : 'a -> ('a * 'b) list -> bool
   ```

   Exemple :

```
appartient_couple 'c' [('a',2) ; ('b',5);('c',2)];;
- : bool = true
```

(c) Modifier la fonction `bfs`, en fonction `bfs_distance`, en y ajoutant la distance parcourue.

```
bfs_distance :
  graphe_2 -> (int * int) list -> (int * int) list -> (int * int)
list
```

(d) Écrire la fonction `parcours_distance` qui renvoie la liste des sommets accessibles, doublés de la distance.

```
parcours_distance : graphe_2 -> int -> (int * int) list
```

(e) Écrire la fonction `chemin` qui renvoie la liste des sommets entre d et a, représentant le plus court chemin.

```
chemin : graphe_2 -> int -> int -> (int * int) list
```

4 Parcours en profondeur

Le parcours en profondeur (appelé DFS, pour Depth First Search) consiste à parcourir le graphe jusqu'au bout, puis de revenir aux sommets non traités.

1. Donner la liste des sommets accessibles en partant du sommet 0 sur le graphe exemple.
2. Écrire la fonction `dfs` qui prend en argument une graphe une liste de sommets déjà vus et une liste de sommets à voir, et renvoie la liste des sommets parcourus.

   ```
   dfs : graphe_2 -> int list -> int list -> int list
   ```

3. En déduire la fonction `parcours_profond` qui prend en argument un graphe et un sommet, et renvoie la liste de sommets accessibles dans un parcours en long.

   ```
   parcours_profond : graphe_2 -> int -> int list
   ```