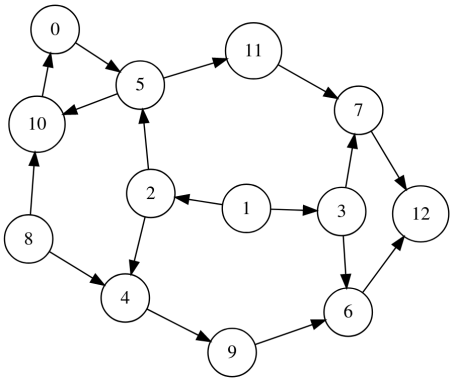
TP 7 : Parcours en largeur

Dans ce TP nous allons travailler l'implémentation en OCaml du parcours en largeur d'un graphe.

I. Parcours en largeur

On fixe les types suivants pour représenter des graphes sous forme de liste d'adjacence :

```
type sommet = int
type graphe = sommet list array
```



> **Question 1**. Définir le graphe ci-contre en OCaml.

> **Question 2**. Écrire une fonction `parcours_en_largeur : graphe → sommet → sommet list` qui effectue un parcours en largeur du graphe depuis un sommet donné. Cette fonction renvoie les sommets dans l'ordre dans lequel ils ont été visités.

L'utilisation la plus classique du parcours en largeur est de trouver un chemin de longueur minimale dans un graphe.

> **Question 3**. Adapter votre fonction de parcours en largeur pour écrire une fonction `distances : graphe → sommet -> int option list`, telle que l'appel `distances g s` renvoie un tableau `d` tel que la case d'indice `t` du tableau contient :
> - `Some n` s'il existe un chemin de `s` à `t`, et que le chemin le plus court est de longueur `n` ;
> - `None` s'il n'en existe pas.
>
> Par exemple, l'appel de la fonction `distances` sur le graphe ci-dessus depuis le sommet 9 doit renvoyer un tableau contenant `Some 0` à l'indice 9, `Some 1` à l'indice 6, `Some 2` à l'indice 12, et `None` dans toutes les autres cases.

Une autre utilisation du parcours en largeur est la détection de cycle (notons néanmoins que le parcours en profondeur rempli cette tâche tout aussi bien).

> **Question 4**. Adapter votre fonction de parcours en largeur pour écrire une fonction `: graphe → sommet → bool` qui effectue un parcours en largeur depuis le sommet passé en paramètre, et renvoie `true` si ce parcours détecte un cycle, `false` sinon.
>
> Par exemple, depuis le sommet 0, la fonction doit renvoyer `true`, mais depuis le sommet 9 elle doit renvoyer `false`.