Exercice 1

On considère les deux problèmes suivants.

- ARRÊT :
  - **Instance** : le code source d'une fonction f et un argument x.
  - **Question** : est-ce que $f(x)$ termine ?
- $\text{ARRÊT}_\forall$:
  - **Instance** : le code source d'une fonction f.
  - **Question** : est-ce que $f(x)$ termine pour tout x ?

1. Montrer que $\text{ARRÊT} \leqslant_m \text{ARRÊT}_\forall$.
2. Que peut-on en déduire sur $\text{ARRÊT}_\forall$?

Corrigé

1. Supposons qu'on dispose d'une fonction `arret_tous` qui résout $\text{ARRÊT}_\forall$. Alors la fonction suivante résout ARRÊT :

```
let arret <f, x> =
    let g _ = universel <f, x> in
    arret_tous g
```

   On en déduit la réduction attendue.
2. Comme ARRÊT est indécidable, on en déduit que $\text{ARRÊT}_\forall$ est également indécidable.

Exercice 2

On considère des graphes représentés par tableau de listes d'adjacence en OCaml par le type :

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

En n'autorisant aucune autre fonction déjà implémentée en OCaml que `Array.length` et `Array.make`, écrire une fonction `transpose : graphe -> graphe` qui prend en argument un graphe G et renvoie le graphe transposé G^T.

1. Sans utiliser de boucle.
2. Sans utiliser de récursivité.

Corrigé

1. On peut ici envisager deux solutions :
   - une fonction récursive pour parcourir chaque case du tableau et une autre fonction récursive pour parcourir chaque liste d'adjacence. Pour chaque élément d'une liste d'adjacence, on crée l'arête dans le bon sens dans le graphe transposé. On écrit une première fonction `parcours_liste : int -> int list -> unit` qui prend en argument un sommet s et une liste de voisins de s et pour chaque sommet t dans cette liste, ajoute l'arête (t, s) au graphe transposé. On utilise ensuite une fonction `parcours_tableau : int -> unit` qui prend en argument un sommet s, parcourt la liste des voisins de s pour ajouter les bonnes arêtes dans le graphe transposé, puis passe au sommet suivant.

```
let transpose g =
    let n = Array.length g in
    let gt = Array.make n [] in
    let rec parcours_liste s = function
        | []     -> ()
        | t :: q -> gt.(t) <- s :: gt.(t);
                    parcours_liste s q in
    let rec parcours_tableau s =
        if s < n then (parcours_liste s g.(s); parcours_tableau (s+1)) in
    parcours_tableau 0;
    gt
```

– une autre façon de procéder est d'utiliser une seule fonction récursive qui se charge de passer au sommet suivant (ou précédent ici). On utilise une fonction `parcours_graphe : int -> int list -> graphe` qui prend en argument un sommet s et une liste de voisins de s et :

  * ajoute au graphe G^T les arêtes (t, s) pour chaque t dans la liste ;
  * relance un appel avec $s-1$ et les voisins de $s-1$ si la liste est vide ;
  * renvoie le graphe transposé lorsque toutes les listes d'adjacence ont été parcourues.

```
let transpose g =
    let n = Array.length g in
    let gt = Array.make n [] in
    let rec parcours_graphe s lst = match s, lst with
        | 0, []     -> gt
        | _, []     -> parcours_graphe (s - 1) g.(s - 1)
        | _, t :: q -> gt.(t) <- s :: gt.(t);
                       parcours_graphe s q in
    parcours_graphe (n - 1) g.(n - 1)
```

2. On utilise une boucle `for` pour parcourir chaque case du tableau. Ensuite, on utilise une référence de liste et une boucle `while` pour parcourir chaque liste d'adjacence.

```
let transpose g =
      let n = Array.length g in
      let gt = Array.make n [] in
      for s = 0 to n - 1 do
          let lst = ref g.(s) in
          while !lst <> [] do
              let t :: q =  !lst in
              lst := q;
              gt.(t) <- s :: gt.(t)
          done
      done;
      gt
```

OCaml écrirait un avertissement de filtrage non exhaustif pour la ligne `let t :: q = !lst in` car il ne sait pas que la liste est supposée non vide. On pourrait utiliser un filtrage pour éviter l'avertissement (avec un `assert false` pour le cas où la liste est vide).

Exercice 3

On considère des graphes représentés par tableau de listes d'adjacence en C par :

```
struct Liste {
    int val;
    struct Liste* suivant;
};

typedef struct Liste liste;

struct Graphe {
    int taille;
    liste** adj;
};

typedef struct Graphe graphe;
```

Pour un objet `G` de type `graphe*`, `G->adj` est un tableau de listes chaînées, chaque liste correspondant à une liste d'adjacence.

1. Écrire une fonction `graphe* init_graphe(int n)` qui prend en argument un entier n et crée et renvoie un pointeur vers un graphe d'ordre n sans arête.
2. Écrire une fonction `void ajout_arete(graphe* G, int s, int t)` qui prend en argument un pointeur vers un graphe G et deux entiers s et t correspondant à des sommets de G, et modifie G en rajoutant l'arête orientée (s, t). On supposera que l'arête n'existe pas déjà.
3. En déduire une fonction `graphe* transpose(graphe* G)` qui prend en argument un graphe G et renvoie le graphe transposé G^T.

Corrigé

1. Il faut allouer ici l'espace mémoire nécessaire au graphe, et celui au tableau de listes. On pense ensuite à initialiser chaque case du tableau (par des pointeurs `NULL` ici).

```
graphe* init_graphe(int n){
    graphe* G = malloc(sizeof(*G));
    liste** adj = malloc(n * sizeof(*adj));
    for (int i=0; i<n; i++){
        adj[i] = NULL;
    }
    G->taille = n;
    G->adj = adj;
    return G;
}
```

2. Il s'agit ici de rajouter un maillon à une liste chaînée. On fait l'allocation nécessaire, puis on modifie les pointeurs pour obtenir ce qu'on veut.

```
void ajout_arete(graphe* G, int s, int t){
    liste* lst = malloc(sizeof(*lst));
    lst->val = t;
    lst->suivant = G->adj[s];
    G->adj[s] = lst;
}
```

3. L'idée reste la même que dans la fonction OCaml : on parcourt chaque indice du tableau, puis la liste pour cet indice.

```
graphe* transpose(graphe* G){
    graphe* GT = init_graphe(G->taille);
    for (int s=0; s<G->taille; s++){
        liste* lst = G->adj[s];
        while (lst != NULL){
            ajout_arete(GT, lst->val, s);
            lst = lst->suivant;
        }
    }
    return GT;
}
```