Cette *dix-neuvième* colle vous fera écrire des fonctions sur des graphes représentés par des listes d'adjacence en OCaml, des calculs sur des entiers représentés en binaire et d'autres bases, et quelques fonctions très courtes sur des chaînes de caractères en C.

On travaillera depuis la machine virtuelle ClefAgreg2019, et on compilera les fichiers écrits avant d'exécuter les binaires produits.

Ex.1 Graphes orientés représentés par listes d'adjacence - OCaml (35 minutes)

On considère dans cet exercice des graphes *orientés* $G = (S, A)$, représentés par listes d'adjacence (comme vu en cours vendredi 04/03/2022).

1. Proposer un type (non récursif) en OCaml permettant de représenter un tel graphe orienté $G = (S, A)$ ayant $n = |S| \in \mathbb{N}$ sommets, numérotés dans l'ordre $S = [\![0; n-1]\!]$, et des arcs A.

Quelle est la taille en mémoire de cette représentation d'un graphe G, en fonction de $n = |S|$ et $m = |A|$?

Rappeler (sans l'implémenter) comment on peut adapter cette représentation si on veut aussi ajouter des étiquettes (de type `'a`) aux sommets.

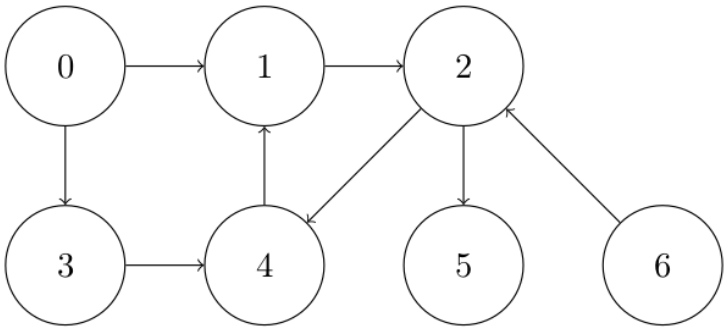

Figure 1 – L'exemple de graphe G_1 vu en cours vendredi dernier.

2. Implémenter en OCaml le graphe G_1 représenté dans la figure ci-dessus comme une variable `g1`, sur laquelle vous pourrez tester les questions suivantes.
3. Écrire une fonction `nombre_sommets : graphe -> int` qui calcule le nombre de sommets d'un graphe G représenté comme expliqué précédemment.
4. Même question pour une fonction `successeurs : graphe -> int -> int list` qui renvoie la liste des successeurs d'un sommet donné. Par exemple `successeurs g1 0` doit renvoyer la liste `[1;3]` (notez que leur ordre n'a pas d'importance).
5. Recopier le code suivant qui implémente le parcours en profondeur, comme vu en cours vendredi.

```
let parcours_profondeur g x =
  let n = nombre_sommets g in
  let vus = Array.make n false in
```



```
  let rec parcours y =
    if not vus.(y) then begin
      vus.(y) <- true;
      (* on parcours récursivement les successeur de y *)
      List.iter
        (fun z -> if vus.(z) then () else parcours z)
        (successeurs g y);
    end
  in
  parcours x;
  vus
;;
```

6. Justifier sa terminaison.
7. Quelle est sa complexité mémoire, en fonction de $n = |S|$ et $m = |A|$?
8. Quelle est sa complexité temporelle, en fonction de $n = |S|$ et $m = |A|$?
9. Au brouillon, quel sera le contenu du tableau de booléens `vus` renvoyés par `parcours_profondeur` sur le graphe `g1` en partant du sommet `0`. Vérifier avec le code.

Ex.2 Quelques calculs sur les représentations de nombres entiers (10 minutes)

Au brouillon, répondre aux questions suivantes.

1. Convertir votre année de naissance en base 2 (binaire), en hexadécimal et en base 8 (octale).
2. Calculer, en la posant, $1001\ 1100_2 + 1100\ 0010_2$. Vérifier en convertissant en décimal.
3. (bonus à faire à la fin) Calculer, en la posant, $1001\ 1100_2 \times 1100_2$. Vérifier en convertissant en décimal.

Ex.3 Deux petites fonctions en C (10 minutes)

On écrira un fichier C `colle19.c`, important `stdio.h` (pour `printf`), `assert.h` (pour `assert`) et `stdbool.h` (pour `true` et `false`).

On rappelle qu'on compile ce fichier avec `COMPILATEUR = gcc` ou `clang` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars, qui représentent le prompt de la ligne de commande du terminal) :

```
$ COMPILATEUR -O0 -Wall -Wextra -Werror -fsanitize=address
  -fsanitize=undefined -pedantic -std=c11 -o colle19.exe colle19.c
$ ./colle19.exe
```

Pour chacune des fonctions demandées, vous penserez évidemment à écrire dans votre fonction `main` au moins deux exemples, que vous testerez avec des `assert`, et afficher leurs résultats dans la console.

1. Écrire une fonction de prototype `int longueur(char* mot)` qui calcule la longueur de la chaîne `mot`, comme le fait la fonction `strlen` de la bibliothèque `string.h`.

On s'imposera évidemment de ne faire qu'une seule passe de lecture du tableau.

Quelle est sa complexité temporelle en fonction de $n = |\texttt{mot}|$ la longueur du mot ?

2. En se rappelant que dans le codage ASCII, le code des lettres `a..z` et `A..Z` et des chiffres `0..9` sont ordonnés consécutivement, écrire une fonction `bool est_chiffre(char symbole)` qui renvoie `true` si `symbole` est un des symboles entre `0` et `9`, et `false` sinon.
3. Utilisez les deux fonctions précédentes pour écrire `bool est_nombre(char* mot)` qui renvoie `true` si tous les caractères de `mot` sont des chiffres, et `false` sinon. Quelle est sa complexité temporelle en fonction de $n = |\texttt{mot}|$ la longueur du mot, dans le meilleur cas et dans le pire cas ?
4. Enfin, utilisez les mêmes deux fonctions précédentes pour écrire `int compte_chiffres(char* mot)` qui compte le nombre d'occurrence de symboles qui sont des chiffres.

