Devoir en temps limité n°6 – 4h

Le code doit être commenté dès qu'il dépasse les 5 lignes ou comprend un concept compliqué. Il est possible (et recommandé) d'utiliser des fonctions auxiliaires en Ocaml, en expliquant leur rôle.

1 Exercice 0 : Tas

Dans cet exercice, on considère le tableau suivant : $t = [2; 5; 10; 23; 7; 9; 3]$. On va faire quelques manipulations simples sur les tas max.

1. **En utilisant la méthode de complexité linéaire vue en cours**, construire un tas max qui contient les valeurs de ce tableau. On fera apparaitre des étapes intermédiaires (pas forcément toutes), en dessinant les arbres.
2. Ajouter à votre tas la valeur 12. On fera apparaitre toutes les étapes intermédiaires, en dessinant les arbres.
3. Effectuer le tri par tas sur le tableau t originel. On fera apparaitre des étapes intermédiaires, sous forme d'arbre ou de tableau.

2 Exercice 1 : Dictionnaires

La sécurité sociale souhaite réaliser une étude sur le médicament X, médicament délivré sur ordonnance pour un mois de traitement. Le but est de savoir en moyenne combien de mois de traitement sont nécessaires pour guérir un patient.

Pour ce faire, on va utiliser un dictionnaire dont les clés sont les numéros de sécurité sociale des patients et les valeurs sont le nombre de mois durant lesquels les patients ont pris le médicament (il suffit d'incrémenter à chaque fois que le patient revient à la pharmacie pour obtenir son traitement).

Le dictionnaire sera implémenté par une table de hachage et la sécurité sociale estime que le médicament X sera pris par maximum $m = 200003$ patients sur la durée de l'étude. (c'est un nombre premier)

Hachage

Dans un premier temps, on veut trouver deux fonctions de hachage pour les numéros de sécurité sociale.

On rappelle que le numéro de sécurité sociale est construit ainsi (en ignorant les nombreux cas particuliers) :

- 1 chiffre pour le sexe : 1 ou 2.
- 2 chiffres pour l'année de naissance : 06 pour 2006 par exemple.
- 2 chiffres pour le mois de naissance.
- 2 chiffres pour le département de naissance (ou 99 pour l'étranger).
- 3 chiffres pour le numéro de la commune de naissance.
- 3 chiffres pour l'ordre de naissance cette année là, ce mois là, dans cette commune là.
- 2 chiffres de vérification, calculés selon la formule 97 - (le nombre formé par les 13 chiffres précédents % 97).

Par exemple un petit Devoir, né le 17 mai 2025 à Dijon, qui serait le 96e bébé ce mois-ci aurait comme numéro 1 25 05 21 231 096 39.

Pour éviter des calculs avec des grandes puissances, on représentera la clé en C par un tableau de taille 15 (il y a toujours 15 chiffres). Pour l'exemple précédent, le tableau est `{1,2,5,0,5,2,1,2,3,1,0,9,6,3,9}`

1. Dans le contexte des tables de hachage, qu'est-ce qu'une fonction de hachage ?
2. Pourquoi la fonction de hachage qui consiste à prendre le premier chiffre modulo m est elle mauvaise ?

On propose une fonction de hachage

$$h_1 : tab \mapsto \sum_{i=7}^{14} tab[i] * 10^{i-8} \mod m$$

qui exploite les 3 dernières informations (sachant que les chiffres de vérification contiennent de l'information à propos des autres chiffres). Cette fonction répartit bien les données car elle peut prendre toutes les valeurs entre 0 et $m - 1$ de manière assez uniforme.

3. Proposer une autre fonction de hachage h_2 qui répartit bien les données.
   Vous pouvez vous inspirer de h_1, mais elles ne doivent pas être liées par une relation linéaire.

La table

On représentera les couples (clé,valeur) en C par le type suivant :

```
typedef struct couple {int* cle;
                       int valeur} couple;
```

On représentera la table en C par le type suivant :

```
typedef struct tableHachage {couple* donnees;
                             int remplissage;} tableHachage;
```

`donnees` est le tableau qui stocke les données, donc les couples (clé,valeur). Il est de taille m, variable globale.

`remplissage` indique combien de cases du tableau `donnees` sont effectivement utilisées.

La table fonctionnera avec un adressage ouvert à deux fonctions, en utilisant h_1 et h_2.

On vous donne la fonction suivante pour créer un dictionnaire, on initialise les éléments du tableau à `(NULL,0)`. :

```
tableHachage* nouveau_dict(){
  tableHachage* res = malloc(sizeof(tableHachage)); //Pointeur de table de hachage pour la mutabilité
  res->donnees = malloc(m*sizeof(couple)); //Tableau de couples
  for(int i=0;i<m;i+=1){
    res->donnees = {cle=NULL;valeur=0}; //Les données sont des couples (et pas des pointeurs de couples)
  }
  res->remplissage = 0;
  return res;
}
```

4. Implémenter la fonction `void ajoute(tableHachage* t, int* cle)` qui ajoute un nouveau patient à la table. Un nouveau patient est un patient qui achète son premier mois de traitement.
   Si le patient ne peut pas être ajouté avec le méthode décrite, une erreur sera levée.
5. Implémenter la fonction `void modifie(tableHachage* t, int* cle)` qui ajoute un mois au patient donné. Si le patient n'est pas trouvé, une erreur sera levée.
   Il vous faudra probablement une fonction auxiliaire.

La fonction pour retirer une clé est très similaire, donc on ne va pas l'écrire.

6. Pour finir, on suppose que le dictionnaire a été rempli et mis à jour durant toute la durée de l'étude, écrire une fonction `float resultat_etude(tableHachage* t)` qui calcule le temps moyen de traitement (en mois).

3 Exercice 2 : Logique

Dans un jeu vidéo vous devez vous échapper d'un labyrinthe avec l'aide de deux compagnons sphinx, qui vous aident à progresser dans le jeu via des énigmes. Les énigmes suivent une règle que chaque Sphinx respecte scrupuleusement (individuellement) :

Dans les énigmes, ils peuvent soit dire la vérité, soit mentir. Mais, pour une énigme donnée, la première et la dernière de leurs affirmations seront de la même nature (soit vérité, soit mensonge) ; et toutes les autres affirmations seront de la nature opposée (si la première et dernière sont vraies alors ce sont des mensonges et respectivement).

1. Considérons que l'un des Sphinx fait une suite de n déclarations (représentées par des formules A_i, numérotées dans l'ordre à partir de 0) dans une même énigme, proposer une formule du calcul des propositions qui représente la règle qu'il respecte.

Vous vous retrouvez face à trois escaliers, l'un à gauche, l'autre à droite et le dernier au milieu entre les deux autres. Le premier Sphinx, nommé Phinx, énonce les affirmations suivantes :

- l'escalier de gauche est sûr,
- l'escalier du milieu est sûr ou celui de droite n'est pas sûr.

Le second Sphinx, nommé Shinx, énonce les affirmations suivantes :

- ni l'escalier de gauche, ni celui du milieu ne sont sûrs,
- si les escaliers de gauche ou de droite sont sûrs, alors l'escalier du milieu est sûr.

Nous noterons g, m et d les variables propositionnelles associées au fait que les escaliers de gauche, du milieu et de droite sont sûrs.

Nous noterons P_1 et P_2, respectivement S_1 et S_2, les formules propositionnelles associées aux déclarations du premier Sphinx, respectivement du second Sphinx.

2. Représenter les déclarations des deux Sphinx sous la forme de formules du calcul des propositions P_1, P_2, S_1 et S_2 dépendant des variables g, m et d.
3. Appliquer la règle respectée par les Sphinx , comme vous l'avez proposé pour la question 1.
   Nous noterons P , respectivement S, la formule du calcul des propositions qui correspond au respect de la règle par le premier Sphinx, respectivement le second Sphinx, dans cette énigme. Nous noterons R la formule du calcul des propositions qui décrit le respect global des règles par les deux Sphinx dans cette énigme.
   Note : les formules ont le droit de dépendre d'autres formules déjà écrites
4. En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer quel est (ou quels sont) le (ou les) escalier(s) qui est (ou sont) sûr(s) ? Vous indiquerez explicitement les résultats intermédiaires correspondant aux formules P_1, P_2, P , S_1, S_2 et S.
   Note : dans cette question il est interdit de faire autre chose qu'une table de vérité.

Vous vous retrouvez plus tard face à trois portes de couleurs rouge, verte et bleue. Seul le premier Sphinx s'exprime et énonce les affirmations suivantes :

- la porte rouge n'est pas sûre ou la porte verte est sûre,
- si les portes rouge et verte sont sûres alors la porte bleue n'est pas sûre,
- la porte verte n'est pas sûre mais la porte bleue est sûre.

Nous noterons P_3, P_4 et P_5 les formules propositionnelles associées aux déclarations du premier Sphinx.

Nous noterons r, v et b les variables propositionnelles associées au fait que les portes rouge, verte et bleue sont sûres.

5. Représenter les déclarations du Sphinx sous la forme de formules du calcul des propositions P_3, P_4 et P_5 dépendant des variables r, v et b.
6. Appliquer la règle respectée par le premier Sphinx que vous avez proposée pour la question 1. Nous noterons P', la formule du calcul des propositions qui correspond au respect de la règle par le premier Sphinx dans cette énigme.
7. En utilisant le calcul des propositions (résolution avec les formules de De Morgan), déterminer quelle est (ou quelles sont) la (ou les) porte(s) qui est (ou sont) sûre(s).
   Note : dans cette question il est obligatoire de raisonner par équivalents sémantiques.

4 Problème : Graphes planaires

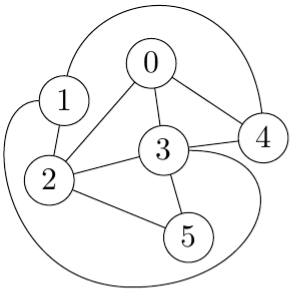

FIGURE 1

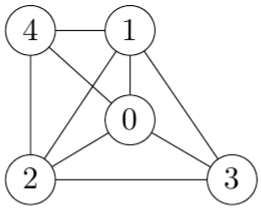

FIGURE 2

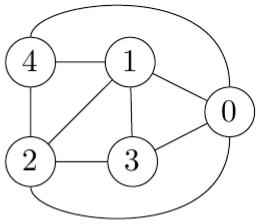

FIGURE 3

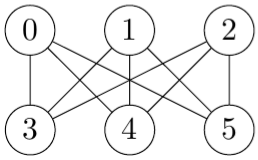

FIGURE 4

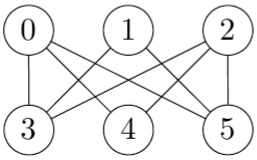

FIGURE 5

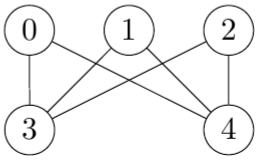

FIGURE 6

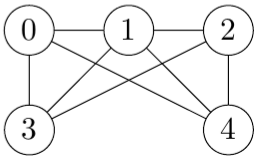

FIGURE 7

1. Préambule

Tous les graphes considérés sont non orientés et sans boucles (pas d'arête de x à x).

1. Donner la définition formelle d'un graphe non orienté.

Dans la suite on considèrera que les sommets d'un graphe sont toujours étiquetés dans $[\![0, n-1]\!]$ avec n le nombre de sommets.

On représentera les graphes en Ocaml sous forme de liste d'adjacence, donc par le type `int list array`, dont on donne un alias `graphe`.

2. Donner, en Ocaml, les listes d'adjacence associées aux graphes des figures 2 et 3.

Définition (informelle). Un graphe G est dit planaire s'il peut être être dessiné dans le plan sans que deux arêtes ne se croisent.

Cette définition a du sens car un graphe peut être dessiné de plusieurs manières, en plaçant les sommets à différents endroits.

Par exemple, le graphe de la figure 1 est planaire et, contrairement aux apparences, le graphe de la figure 2 est planaire. En effet, les figures 2 et 3 représentent le même graphe. Puisque le graphe de la figure 3 est planaire, le graphe de la figure 2 l'est aussi.

En revanche, le graphe de la figure 4 n'est pas planaire, même en le dessinant autrement il y aura deux arrêtes qui s'intersectent.

3. Montrer que le graphe de la figure 5 est planaire.

2. Transformation de graphes

Opérations sur les listes

4. Écrire une fonction `supprimer : int list -> int -> int list` qui prend en entrée une liste l et un élément i_0 et renvoie une liste contenant les mêmes éléments que l où i_0 a été supprimé (toutes ses occurences). Par exemple, avec $l = [4; 5; 1; 4; 9]$ et $i_0 = 4$, on obtient $[5; 1; 9]$.
5. Écrire une fonction `decaler : int list -> int -> int list` qui prend en entrée une liste l et un élément i_0 et renvoie une liste contenant les mêmes éléments que l en remplaçant tous les éléments $i > i_0$ par $i - 1$. Par exemple avec $l = [5; 1; 4; 9]$ et $i_0 = 4$, on obtient $[4; 1; 4; 8]$.
6. Écrire une fonction `supprime_doublons : int list -> int list` qui prend en entrée une liste l et renvoie une liste qui est similaire à l mais où aucun élément n'est présent en double. Par exemple avec $l = [4; 8; 2; 8; 3; 6; 4; 0; 3; 4]$, on doit obtenir $[4; 8; 2; 3; 6; 0]$.
   La complexité doit être linéaire en la taille n de l, sachant qu'on garantit que les éléments dans la liste sont tous inférieurs à n.

On suppose également écrite une fonction `remplacer : int list -> int -> int -> int list` qui prend en entrée une liste et deux entiers a et b et qui remplacent les occurences de a par des occurences de b.

Opérations sur les graphes

Dans cette partie, on s'intéresse à trois opérations :

- L'opération de suppression d'une arête.
  Par exemple, dans le graphe de la figure 4, supprimer l'arête $\{1, 4\}$ donne le graphe de la figure 5.
- L'opération de suppression d'un sommet.
  Soit $G_1 = (S, A)$ un graphe avec $n \in \mathbb{N}^*$ sommets. Supprimer un sommet $s_0 \in S$ consiste à construire un graphe G_2 où le sommet s_0 a été supprimé ainsi que toutes les arêtes adjacentes à s_0. Afin que l'ensemble des étiquettes de G_2 soit égal à $[\![0, n-2]\!]$, un sommet d'étiquette $s > s_0$ dans G_1 est ré-étiqueté par $s - 1$ dans G_2 .
  Par exemple, supprimer le sommet d'étiquette $i_0 = 4$ dans le graphe de la figure 4 donne le graphe de la figure 6. Notez qu'en particulier, le sommet d'étiquette 5 de la figure 4 vérifie $5 > i_0$ et est donc ré-étiqueté par 4 dans la figure 6.
- L'opération de contraction d'une arête.
  Soit $G_1 = (S, A)$ un graphe avec $n \geq 2$ sommets et s_1, s_2. Contracter l'arête $\{s_1, s_2\}$ consiste à construire un nouveau graphe G_2 où les sommets s_1 et s_2 sont remplacés par un nouveau sommet t. Les voisins de t sont tous les voisins de s_1 et tous les voisins de s_2 (sauf s_1 et s_2 eux-mêmes).
  Afin que l'ensemble des étiquettes dans G_2 soit égal à $[\![0, n-2]\!]$, le sommet t est étiqueté par $\min(s_1, s_2)$ et tout sommet d'étiquette $i > \max(s_1, s_2)$ dans G_1 est ré-étiqueté par $i - 1$ dans G_2. Par exemple, contracter l'arête $\{1, 4\}$ dans le graphe de la figure 4 donne le graphe de la figure 7.

On considère un graphe G_1. Soient s_0, s_1, s_2 trois sommets de G_1. Les fonctions écrites dans les questions qui suivent doivent renvoyer un nouveau graphe G_2 (une nouvelle représentation par listes d'adjacence) et **ne pas modifier** G_1.

De plus, vous devez faire en sorte que les sous-listes de G_2 ne contiennent pas d'élément en double (c'est déjà le cas pour les sous-listes de G_1). Pensez à utiliser les fonctions de la partie précédente.

7. Écrire une fonction `supprime_arrete : graphe -> int -> int -> graphe` qui prend en entrée G_1, s_1 et s_2 et renvoie le graphe obtenu après suppression de l'arête $\{s_1, s_2\}$. On pourra supposer que cette arête existe.
8. Écrire une fonction `supprime_sommet : graphe -> int -> graphe` qui prend en entrée G_1 et s_0 et renvoie le graphe obtenu après suppression du sommet s_0.
9. Écrire une fonction `contracte_arrete : graphe -> int -> int -> graphe` qui prend en entrée G_1, s_1 et s_2 et renvoie le graphe obtenu après contraction de $\{s_1, s_2\}$. On pourra supposer que cette arête existe.

3. Graphes complets

On rappelle qu'un graphe complet est un graphe $G = (S, A)$ dans lequel pour tout $x \neq y \in S$, l'arête $\{x, y\}$ est dans A.

Graphes non-bipartis

10. Écrire une fonction `make_complet : int -> graphe` qui prend en entrée n et renvoie le graphe complet à n noeuds (sous forme de liste d'adjacence).
11. Écrire une fonction `est_complet : graphe -> int -> bool` qui prend en entrée une graphe g et un entier n et indique si g est le graphe complet à n noeuds.
12. Quelle est la complexité de votre fonction ?

Graphes bipartis

On rappelle qu'un graphe biparti est un graphe $G = (S, A)$ tel qu'on peut séparer $S = S_1 \cup S_2$ avec $S_1 \cap S_2 = \emptyset$, avec la contrainte suivante : toute arrête $\{a, b\}$ a une extrémité dans S_1 et une extrémité dans S_2.

13. Parmi les graphes d'exemples, lesquels sont bipartis ? (sans justification)
14. Montrer que le graphe de la figure 1 n'est pas biparti.

On dit qu'un graphe G est biparti complet s'il est biparti et que toutes les arêtes possibles entre S_1 et S_2 existent.

Par exemple le graphe de la figure 4 est biparti complet.

15. Écrire une fonction `make_biparti_complet : int -> int -> graphe` qui prend en entrée deux entiers n_1 et n_2 et renvoie un graphe biparti complet avec $|S_1| = n_1$ et $|S_2| = n_2$.

Reconnaissance des graphes bipartis complets

On veut déterminer si un graphe $G = (S, A)$ est biparti complet en Ocaml. On note $n = |S|$ son nombre de sommets.

On propose l'algorithme suivant en pseudo-code pour déterminer si un graphe est biparti. On rappelle qu'un graphe est biparti si et seulement s'il est coloriable par deux couleurs, c'est à dire qu'on peut attribuer une couleur (0 ou 1) à chaque sommet de telle manière que deux sommets de même couleur ne partagent pas une arête.

- Créer une file vide f et y mettre le sommet 0.
- Créer un tableau $couleur$ de taille n initialisé à -1. Mettre $couleur[0]$ à 0.
- $res = true$.
- Tant que la file n'est pas vide et que res est vrai :
  - Défiler un élément s de f.
  - $c = couleur[s]$
  - Pour chaque voisin y de s :
    - Si $couleur[y] = c$: //Le voisin a la même couleur que s, le graphe n'est pas biparti
      - $res = false$
    - Sinon si $couleur[y] = -1$: //Le voisin n'est pas colorié, on le colorie mais il faut vérifier que cela ne crée pas de conflit
      - $couleur[y] = 1 - c$.
      - Enfiler y sur f.
- Renvoyer res et le tableau $couleur$.

16. Appliquer le pseudo code au graphe de la figure 1 et de la figure 5.
17. Une fois qu'on a les couleurs, comment déduire les deux parties de l'ensemble S ?

On rappelle que Ocaml propose une implémentation de file dans la bibliothèque `Queue`. Les primitives ont les noms suivants (en anglais), le type `'a t` désigne une file :

- `Queue.create : unit -> 'a t` qui crée une file vide

- `Queue.is_empty : 'a t -> bool` qui teste si une file est vide
- `Queue.push : 'a -> 'a t -> unit` qui ajoute un élément
- `Queue.pop : 'a t -> 'a` qui retire et renvoie l'élément le plus ancien
- `Queue.peek : 'a t -> 'a` qui renvoie sans retire l'élément le plus ancien

18. Implémenter le pseudo code en Ocaml.
19. En déduire une fonction `est_biparti_complet : graphe -> int -> int -> bool` qui prend en entrée un graphe g et deux entiers n_1 et n_2 et indique si le graphe est le graphe biparti complet avec $|S_1| = n_1$ et $|S_2| = n_2$.

4. Caractérisation des graphes planaires

- On appelle transformations les trois opérations vues dans la partie 2 (suppression d'une arête, suppression d'un sommet et contraction d'une arête).
- Soient $k \in \mathbb{N}$ et G_1, G_2 deux graphes. On dit que G_2 est un mineur d'ordre k de G_1, si G_2 peut être obtenu à partir de G_1 en appliquant une suite de k transformations.
- Soient G_1, G_2 deux graphes. On dit que G_2 est un mineur de G_1 s'il existe $k \in \mathbb{N}$ tel que G_2 est un mineur d'ordre k de G_1.
- On note K_1 le graphe complet à 5 sommets et K_2 le graphe biparti complet de la figure 4.

Théorème (admis) Un graphe $G = (S, A)$ est planaire si et seulement si l'ensemble des mineurs de G ne contient ni K_1, ni K_2.

20. Montrer qu'il est possible qu'un graphe biparti ait comme mineur K_1 et qu'il est possible qu'une graphe non-biparti ait comme mineur K_2.
21. En utilisant le théorème, répondre aux questions suivantes :
    (a) Soit $n \in \mathbb{N}$ et G un graphe complet à n sommets. Le graphe G est-il planaire ?
    (b) Soit $n_1, n_2 \in \mathbb{N}$ et G une graphe biparti complet dont les parties sont de taille n_1 et n_2. Le graphe G est-il planaire ?

Pour savoir si un graphe est planaire, on va générer tous ses mineurs et tester si K_1 ou K_2 en font partie.

22. Écrire une fonction `mineurs_ordre1 : graphe -> graphe list` qui prend en entrée G et renvoie la liste des mineurs d'ordre 1 de G.
23. En déduire une fonction `est_planaire: graphe -> bool` qui détermine si le graphe donné en entrée est planaire.