

L'utilisation de la calculatrice est interdite pour ce devoir.

Ce devoir est d'une durée de 4h, et est composé de 2 parties indépendantes :

- la partie 1 porte sur l'étude d'un réseau social, et est à traiter dans le langage C ;
- la partie 2 porte sur l'étude d'une relation d'ordre, et est à traiter dans le langage OCaml.

Le barème tient compte de la clarté et de la présentation de votre copie. On prendra garde à indiquer clairement le numéro des questions, et, autant que possible, à rédiger les questions dans l'ordre. Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Complexité. La complexité, ou le temps d'exécution, d'un programme P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, ...) nécessaires à l'exécution de P . Lorsque cette complexité dépend de plusieurs paramètres n et m , on dira que P a une complexité en $\mathcal{O}(\varphi(n, m))$ lorsqu'il existe trois constantes A , n_0 et m_0 telles que la complexité de P soit inférieure ou égale à $A \times \varphi(n, m)$ pour tout $n \geq n_0$ et $m \geq m_0$.

Lorsqu'il est demandé de donner une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

1 Étude de réseaux sociaux

Remarque. Dans cette partie, les fonctions demandées sont à écrire dans le langage C.

Objectif. Le but de cette partie est de regrouper des personnes par affinité dans un réseau social. Pour cela, on cherche à répartir les personnes en deux groupes de sorte à minimiser le nombre de liens d'amitié entre les deux groupes. Une telle partition s'appelle une **coupe minimale** du réseau.

1.1 Réseaux sociaux

Structure de données. Nous supposons que les individus sont numérotés de 0 à $n - 1$, où n est le nombre total d'utilisateurs du réseau social.

Nous supposons également que les liens d'amitié du réseau sont symétriques : i est un ami de j si et seulement si j est un ami de i ; et qu'un individu i ne peut pas être ami avec lui-même.

```
type reseau
struct reseau {
    int nb_users;
    int** liens;
};
typedef struct reseau reseau;
```

Un réseau social R entre n individus sera représenté par la structure ci-dessus, où :

- `R.nb_users` contient le nombre n d'individus appartenant au réseau ;
- `R.liens` est un tableau de longueur n tel que :
 - pour $i \in \llbracket 0, n-1 \rrbracket$, `R.liens[i]` est un tableau d'entiers de longueur n représentant l'**ensemble des amis** de i , avec les conventions suivantes :
 - `R.liens[i][0]` contient le nombre d'amis de i ;
 - $\forall k \in \llbracket 1, R.liens[i][0] \rrbracket$, `R.liens[i][k]` contient une valeur j telle que i et j sont amis ;
 - les autres cases de `R.liens[i]` contiennent des valeurs arbitraires.

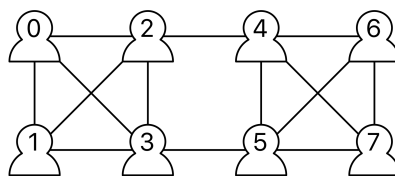
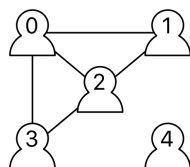


FIGURE 1 – Un exemple de réseau social

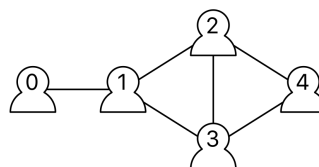
La figure 1 donne l'exemple d'un réseau social à 8 individus et ayant 14 liens d'amitié, dont le tableau `R.liens` est le suivant :

$$\begin{pmatrix} 3 & 1 & 2 & 3 & \cdot & \cdot & \cdot & \cdot \\ 3 & 0 & 3 & 2 & \cdot & \cdot & \cdot & \cdot \\ 4 & 3 & 0 & 1 & 4 & \cdot & \cdot & \cdot \\ 4 & 1 & 2 & 0 & 5 & \cdot & \cdot & \cdot \\ 4 & 5 & 6 & 7 & 2 & \cdot & \cdot & \cdot \\ 4 & 4 & 7 & 6 & 3 & \cdot & \cdot & \cdot \\ 3 & 7 & 4 & 5 & \cdot & \cdot & \cdot & \cdot \\ 3 & 5 & 6 & 4 & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

1. Représenter le tableau des liens d'amitié pour chacun des deux réseaux sociaux ci-dessous :



Réseau A



Réseau B

- Écrire une fonction `reseau creer_reseau_vide(int n)`; prenant en entrée un entier n , et qui crée, initialise, et renvoie la représentation d'un réseau social à n individus n'ayant aucun lien d'amitié.
- Écrire une fonction `bool sont_amis(reseau R, int i, int j)`; prenant en entrée un réseau R et deux individus i et j , et renvoie `true` si i et j sont amis (et `false` sinon).
- Écrire une fonction `void ajouter_amis(reseau R, int i, int j)`; prenant en entrée un réseau R et deux individus i et j , et modifie R par effets de bords pour rajouter les liens d'amitiés entre i et j . **Attention** : on ne fera aucune modification si i et j sont déjà amis, et on prendra garde à rajouter les deux liens $i \rightarrow j$ et $j \rightarrow i$ sinon.
- Écrire une fonction `void free_reseau(reseau R)`; prenant en argument un réseau R et libérant toute la mémoire allouée sur le tas par R .

Dans la suite, il sera plus pratique d'avoir à notre disposition la liste de tous les liens d'amitié (i, j) du réseau tels que $i < j$ (pour ne pas les avoir en double), stockés dans un tableau de couples de taille **adéquate**. On introduit la structure suivante pour représenter un couple :

type couple

```
struct couple {
    int fst;
    int snd;
};
typedef struct couple couple;
```

- Écrire une fonction `int nb_liens(reseau R)`; prenant en entrée un réseau R , et renvoyant le nombre de couples (i, j) tels que $i < j$ et i est ami avec j dans le réseau R .
- Écrire une fonction `couple* liste_des_liens(reseau R)`; prenant en entrée un réseau R , et renvoyant un tableau contenant tous les couples (i, j) tels que $i < j$ et i est ami avec j dans le réseau R .

1.2 Partitions

Partition. Une partition en k groupes d'un ensemble A à n éléments consiste en k sous-ensembles **disjoints non vides** A_1, \dots, A_k de A dont l'union est A , i.e. $A = A_1 \uplus \dots \uplus A_k$.

Par exemple, $A_1 = \{1, 8, 9, 11, 13, 14, 15\}$, $A_2 = \{2, 3, 4, 12\}$, $A_3 = \{10\}$, et $A_4 = \{0, 5, 6, 7\}$ forment une partition de $\llbracket 0, 15 \rrbracket$.

Structure Union-Find. Dans cette partie, nous implémentons une structure de données très efficace pour coder des partitions d'ensembles de la forme $\llbracket 0, n-1 \rrbracket$: la structure **Union-Find**. Le principe de cette structure de données est que les éléments de chaque groupe sont structurés par une relation filiale : chaque élément a un (unique) parent choisi dans le groupe et l'unique élément du groupe qui est son propre parent est appelé le **représentant** du groupe. On s'assure par construction que chaque élément i du groupe a bien pour ancêtre le représentant du groupe.

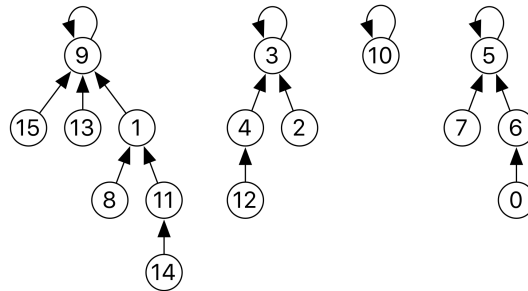


FIGURE 2 – Exemple de structure **Union-Find** représentant une partition de $\llbracket 0, 15 \rrbracket$.

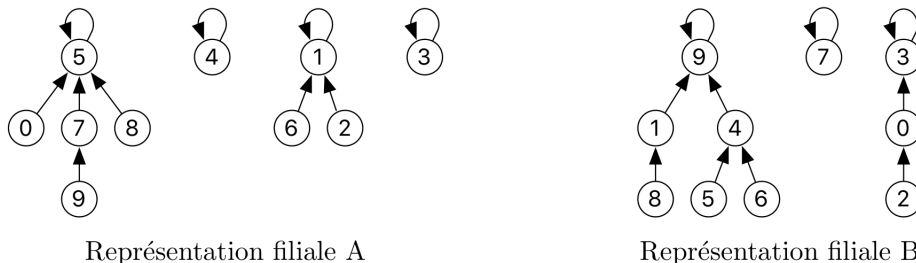
La relation filiale est symbolisée par une flèche allant de l'enfant au parent dans la figure 2 qui présente un exemple de cette structure de données représentant la partition de $\llbracket 0, 15 \rrbracket$ décrite ci-dessus. Dans l'exemple de cette figure, 14 a pour parent 11 qui a pour parent 1 qui a pour parent 9 qui est son propre parent. Ainsi, 9 est le représentant du groupe auquel appartiennent 14, 11, 1 et 9. Notons que ce groupe contient également 8, 13 et 15.

Pour coder cette structure, on utilise un tableau d'entiers **parent** de longueur n tel que **parent**[i] contient le numéro du parent de i .

Par exemple, les valeurs du tableau **parent** encodant la structure **Union-Find** de la figure 2 sont :

i :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
parent [i] :	6	9	3	3	3	5	5	5	1	9	10	1	4	9	11	9

8. Donner les valeurs du tableau **parent** encodant les représentations filiales des deux partitions de $\llbracket 0, 9 \rrbracket$ ci-dessous, et préciser les représentants de chaque groupe :



Représentation filiale A

Représentation filiale B

Initialement, chaque élément de $\llbracket 0, n-1 \rrbracket$ est son propre représentant, et la partition consiste en n groupes contenant chacun un individu. Ainsi, initialement, **parent**[i] = i pour $i \in \llbracket 0, n-1 \rrbracket$.

9. Écrire une fonction **int*** **creer_partition_en_singletons**(**int** n); prenant en argument un entier n et qui alloue sur le tas et renvoie un tableau **parent** à n éléments dont les valeurs sont initialisées de sorte à encoder $\llbracket 0, n-1 \rrbracket$ en n groupes d'un seul élément.

Nous sommes intéressés par deux opérations sur les partitions :

- Trouver le représentant du groupe d'un élément i ;
- Fusionner deux groupes pour n'en faire plus qu'un. Dans ce cas, si r_i et r_j sont les représentants de i et j , on effectue l'opération $\text{parent}[\text{r}_i] = \text{r}_j$;

La figure 3 présente la structure **Union-Find** obtenue après la fusion des groupes contenant 6 et 14 de la figure 2.

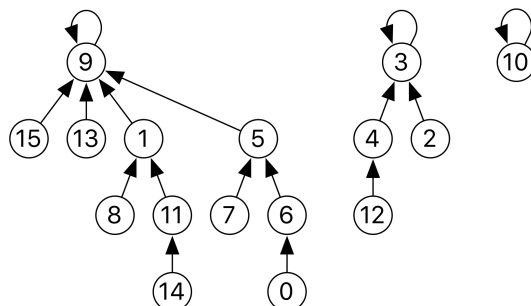


FIGURE 3 – Structure Union-Find obtenue après une fusion de 6 et 14.

- (a) Écrire une fonction `int representant(int* parent, int i)` ; qui utilise (sans le modifier) le tableau `parent` pour trouver et renvoyer l'indice du représentant du groupe auquel appartient i dans la partition encodée par le tableau `parent`.
(b) Quelle est la complexité de votre fonction dans le pire des cas, en fonction du nombre total n d'éléments ?
Donner un exemple de tableau `parent` à n éléments qui atteigne cette complexité dans le pire des cas.
- Écrire une fonction `void fusion(int* parent, int i, int j)` ; qui modifie le tableau `parent` par effets de bords pour fusionner les deux groupes contenant i et j respectivement.
- Proposer une suite de $(n - 1)$ fusions dont l'exécution à partir de la partition en n singletons de $\llbracket 0, n - 1 \rrbracket$ nécessite de l'ordre de n^2 opérations élémentaires.

Compression de chemins. Cette dernière question montre que la structure de données n'est pour l'instant pas très efficace. Pour remédier à cette mauvaise performance, une astuce consiste à **compresser** la relation filiale après chaque appel à la fonction `representant(parent, i)`. L'opération de compression consiste à faire la chose suivante : si r_i est le résultat de l'appel à la fonction `representant(parent, i)`, modifier le tableau `parent` par effets de bords de façon à ce que chaque ancêtre de i , y compris i , ait pour parent direct r_i (avant de bien sûr renvoyer r_i).

Ainsi, si l'on reprend l'exemple de la figure 2, le résultat de l'appel `representant(parent, 14)` est 9, et la compression de chemin mène à la structure représentée dans la figure 4.

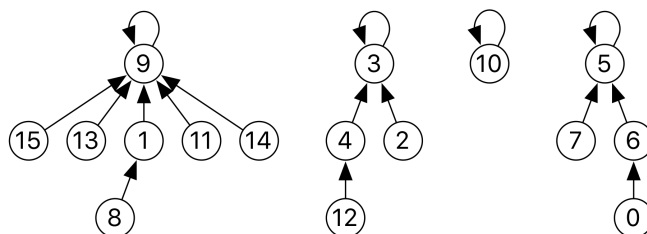


FIGURE 4 – Exemple de compression à partir de 14.

- (a) Proposer une nouvelle fonction `int representant(int* parent, int i)` ; implémentant cette optimisation.
(b) En quoi cette optimisation de la structure filiale peut-elle être considérée comme “gratuite” du point de vue de la complexité ?

1.3 Algorithme randomisé pour la coupe minimum

Revenons à présent à notre objectif principal : trouver une partition des individus d'un réseau social en deux groupes qui minimise le nombre de liens d'amitiés entre les deux groupes. Pour résoudre ce problème, nous allons utiliser l'algorithme randomisé suivant :

Algorithme 1 : Algorithme randomisé pour la coupe minimum

Données : Un réseau R à n individus

Résultat : Une partition des individus de R en deux groupes

Étape 1 : créer une partition **parent** en n singletons ;

Étape 2 : initialement, aucun lien n'est marqué ;

Étape 3 : tant que la partition **parent** contient au moins 3 groupes et qu'il reste des liens d'amitié non marqués dans R faire

Étape 3(a) : choisir un lien (i, j) uniformément au hasard parmi les liens non marqués du réseau ;

Étape 3(b) : si i et j n'appartiennent pas au même groupe dans **parent** alors
 fusionner les groupes de i et j ;

Étape 3(c) : marquer le lien (i, j) ;

Étape 4 : si **parent** contient $k \geq 3$ groupes alors

 faire $k - 1$ fusions pour obtenir 2 groupes ;

Étape 5 : renvoyer **parent** ;

Un exemple d'une exécution de cet algorithme sur le réseau de la figure 1 est présentée en figure 5, où les liens sélectionnés aléatoirement sont dans l'ordre : $(1, 2)$, $(4, 5)$, $(5, 6)$, $(0, 3)$, $(0, 2)$, $(2, 3)$, et $(5, 7)$. Les liens représentés en noir épais sont les liens sélectionnés au hasard à l'étape courante ; les liens épais en gris sont les liens marqués par l'algorithme ; les ronds représentent la partition à l'étape courante.

On suppose qu'on dispose d'une fonction `int randint(int i, int j)` ; qui renvoie un entier dans $\llbracket i, j \rrbracket$ avec probabilité uniforme.

14. (a) Écrire une fonction `int* coupe_minimum_randomisee(reseau R)` ; implémentant l'algorithme décrit ci-dessus.

Indication : au lieu de marquer explicitement les liens déjà vus, on pourra avantageusement les positionner à la fin du tableau des liens du réseau, et ainsi pouvoir tirer simplement les liens au hasard parmi ceux placés au début du tableau.

- (b) Quelle est la complexité de votre fonction, en fonction de n (nombre d'individus), m (nombre de liens d'amitiés), et $\alpha(n)$ où $\alpha(n)$ est la complexité d'un appel à la fonction `representant` ?
15. Écrire une fonction `int taille_coupe(reseau R, int* parent)` ; qui calcule le nombre de liens entre les différents groupes de la partition représentée par **parent** dans le réseau R .

On peut démontrer que cet algorithme renvoie une coupe de taille minimum avec une probabilité supérieure à $1/n$, ce qui fait que la meilleure parmi n exécutions indépendantes de cet algorithme est effectivement minimum avec probabilité supérieure à $1/e \approx 0.36787\dots$

Remarque. La structure de données filiale avec compression pour les partitions est particulièrement efficace aussi bien en pratique qu'en théorie. En effet, la complexité de k opérations est de $\mathcal{O}(k\alpha(k))$ opérations élémentaires où $\alpha(k)$ est l'inverse de la fonction d'Ackermann, une fonction qui croît **extrêmement** lentement vers $+\infty$ (par exemple $\alpha(10^{80}) = 5$).

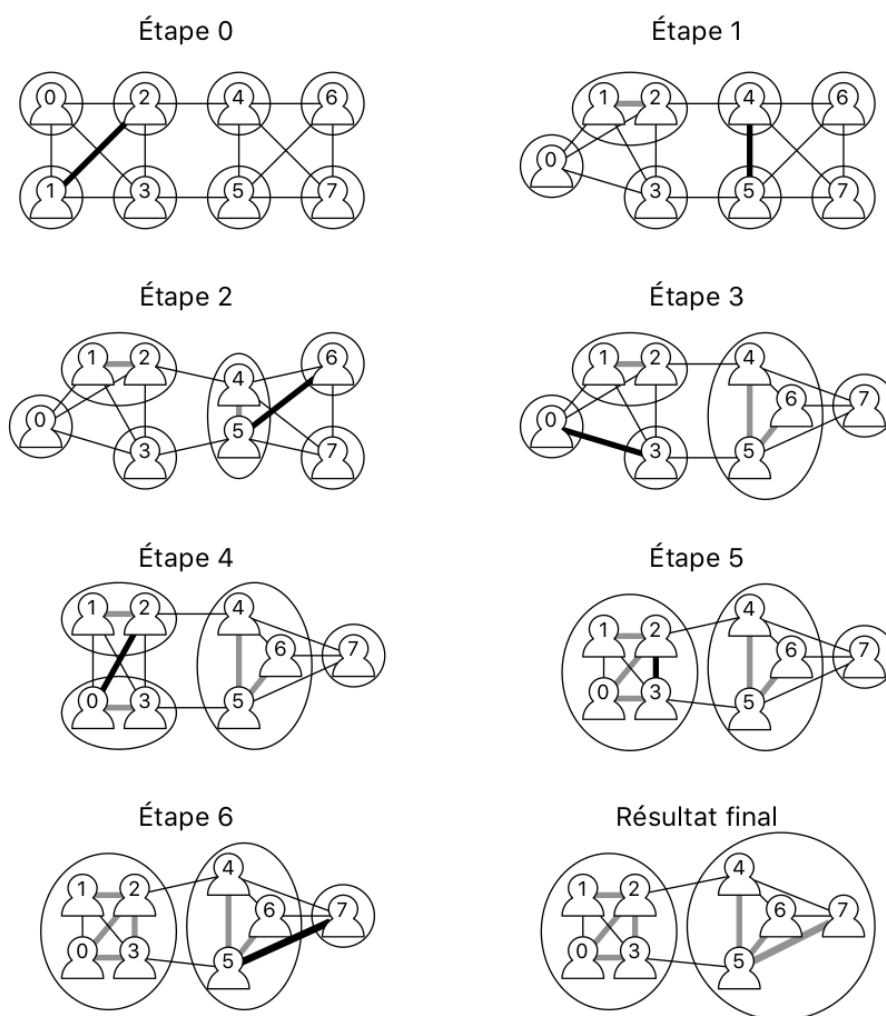


FIGURE 5 – Un exemple d'exécution de l'algorithme randomisé.

2 Étude de relations d'ordre

Remarque. Dans cette partie, les fonctions demandées sont à écrire dans le langage OCaml.

Dans cette partie, on fixe un ensemble fini V , et on note $n = |V|$ son cardinal.

On note systématiquement v_0, v_1, \dots, v_{n-1} les éléments distincts de V .

Définition 2.1 : relation d'ordre

Une **relation d'ordre** sur V est une relation binaire \leq qui est :

- **réflexive** : $\forall i \in \llbracket 0, n-1 \rrbracket, v_i \leq v_i$;
- **antisymétrique** : $\forall (i, j) \in \llbracket 0, n-1 \rrbracket^2$, si $v_i \leq v_j$ et $v_j \leq v_i$ alors $v_i = v_j$;
- **transitive** : $\forall (i, j, k) \in \llbracket 0, n-1 \rrbracket^3$, si $v_i \leq v_j$ et $v_j \leq v_k$, alors $v_i \leq v_k$.

Si un ensemble fini V est muni d'une relation d'ordre \leq , on dit que (V, \leq) est un **ensemble ordonné**.

Notation

Dans la suite, on réserve le symbole \leq pour désigner l'ordre usuel sur \mathbb{N} , et on utilise \leq pour désigner une relation d'ordre quelconque sur V .

Attention

Une relation d'ordre n'est pas forcément **totale** : il se peut qu'il existe $(v_i, v_j) \in V^2$ tels qu'on n'ait pas $v_i \leq v_j$ ni $v_j \leq v_i$ (v_i et v_j ne sont alors **pas comparables**).

2.1 Relation d'ordre et tri topologique

```
type order = bool array array
```

On se propose de représenter une relation \leq sur un ensemble V par le type ci-dessus.

si V est de cardinal n , \leq est représentée par une matrice de booléens \mathbf{r} de taille $n \times n$ telle que $\forall (i, j) \in \llbracket 0, n-1 \rrbracket^2$, $\mathbf{r}.(i).(j)$ contient **true** si $v_i \leq v_j$ (et **false** sinon).

16. (a) Écrire une fonction `check_reflexivity : order -> bool` prenant en argument une matrice représentant une relation \leq et renvoyant **true** si \leq est réflexive (et **false** sinon).
Votre fonction devra relever du paradigme de **programmation impérative**.
- (b) Écrire une fonction `check_reflexivity_bis : order -> bool` prenant en argument une matrice représentant une relation \leq et renvoyant **true** si \leq est réflexive (et **false** sinon).
Votre fonction devra relever du paradigme de **programmation fonctionnelle**.

Dans le reste du sujet, le choix du paradigme de programmation est libre.

17. Écrire une fonction `check_antisymmetry : order -> bool` prenant en argument une matrice représentant une relation \leq et renvoyant **true** si \leq est antisymétrique (et **false** sinon).
18. Écrire une fonction `check_transitivity : order -> bool` prenant en argument une matrice représentant une relation \leq et renvoyant **true** si \leq est transitive (et **false** sinon).
19. En déduire une fonction `is_order : order -> bool` prenant en argument une matrice représentant une relation \leq et renvoyant **true** si \leq est une relation d'ordre (et **false** sinon).
20. Écrire une fonction `nb_sup : order -> int array` prenant en argument une matrice représentant un ensemble ordonné (V, \leq) et renvoyant un tableau d'entiers \mathbf{d} de longueur n tel que $\forall i \in \llbracket 0, n-1 \rrbracket$, $\mathbf{d}.(i)$ contienne le nombre d'éléments $v_j \in V$ tels que $v_i \leq v_j$.
21. Écrire une fonction `argmin : 'a array -> int` prenant en argument un tableau \mathbf{d} de longueur ℓ et renvoyant un indice $i \in \llbracket 0, \ell-1 \rrbracket$ tel que $\forall j \in \llbracket 0, \ell-1 \rrbracket, \mathbf{d}.(i) \leq \mathbf{d}.(j)$.
On supposera sans le vérifier que $\ell > 0$.

Définition 2.2 : tri topologique

Soit (V, \preceq) un ensemble ordonné de cardinal n . On appelle **tri topologique** de V une permutation $\tau \in \mathfrak{S}_{[0, n-1]}$ telle que $\forall (i, j) \in [0, n-1]^2$, si $v_i \preceq v_j$, alors $\tau(i) \leq \tau(j)$. Dans ce cas, $\forall i \in [0, n-1]$, on dit que v_i est de **numéro** $\tau(i)$.

22. Dans cette question, on prouve que tout ensemble ordonné fini (V, \preceq) possède un tri topologique.
- (a) Montrer que si $V \neq \emptyset$, alors $\exists v_i \in V$ tel que $\forall v_j \neq v_i$, on n'a pas $v_i \preceq v_j$.
 - (b) Montrer par récurrence sur n le cardinal de V que tout ensemble ordonné fini (V, \preceq) possède un tri topologique.

On représente une telle permutation τ en OCaml par la liste $[\tau^{-1}(0); \tau^{-1}(1); \dots; \tau^{-1}(n-1)]$.

Exemple 2.1

Si $v_2 \preceq v_1 \preceq v_5$ et $v_0 \preceq v_3 \preceq v_4$, et qu'il n'y a pas d'autre relation de comparaison, alors il y a plusieurs tri topologiques, comme par exemple :

```
let tri_topologique = [2; 0; 1; 3; 5; 4]
```

23. Écrire une fonction `topological_sort : order -> int list` prenant en argument une matrice représentant un ensemble ordonné (V, \preceq) et renvoyant une liste représentant un tri topologique de (V, \preceq) .
Indication : on pourra utiliser un tableau de booléens `deja_ajoute` tel que `deja_ajoute.(i)` indique si on a déjà ajouté i dans la liste représentant la permutation, et s'inspirer de la question 22a pour trouver le prochain élément à rajouter dans la liste.
24. Énoncer un ensemble d'invariants qui démontre que votre fonction `topological_sort` est correcte. Il pourra s'agir d'**invariants de boucle** dans le cas d'un programme impératif ou de **postconditions** dans le cas d'un programme récursif.
25. Exprimer la complexité temporelle de votre fonction `topological_sort` en fonction de la dimension n de la matrice prise en argument.

2.2 Chaines et antichaines**Définition 2.3 : chaîne**

Soit (V, \preceq) un ensemble ordonné. On appelle **chaîne** une partie $C \subseteq V$ totalement ordonnée, c'est-à-dire telle que $\forall (c, c') \in C^2$, $c \preceq c'$ ou $c' \preceq c$.

Dans la suite, on représente une telle chaîne C par le type `int list`.

Rappel

La fonction `for_all` du module `List` possède la documentation suivante :

```
val for_all : ('a -> bool) -> 'a list -> bool
for_all f [a1; ...; an] checks if all elements of the list satisfy the predicate f.
That is, it returns (f a1) && (f a2) && ... && (f an) for a non-empty list and
true if the list is empty.
```

On donne la fonction suivante :

```
let rec is_chain (r:order) (c:int list) : bool = match c with
| [] -> true
| v::cc -> List.for_all (fun x -> x > v || x < v) cc
    && is_chain r cc
```


26. Confirmer ou réfuter la spécification suivante : la fonction `is_chain r c` codée comme ci-dessus renvoie `true` si la liste de sommets `c` est une chaîne au sens de la relation d'ordre `r` (et `false` sinon). Dans le premier cas, on fournira une démonstration ; dans le second cas, on modifiera le code de la fonction `is_chain` afin qu'elle soit correcte.
27. Déterminer la complexité temporelle de la fonction `is_chain r` dans le pire des cas, en fonction de la longueur γ de la liste `c`.

Définition 2.4 : antichaine

Soit (V, \preceq) un ensemble ordonné. On appelle **antichaine** une partie $A \subseteq V$ dont les éléments sont deux à deux **incomparables**, c'est-à-dire telle que $\forall (a, a') \in A^2$, aucune des relations $a \preceq a'$ ou $a' \preceq a$ n'est vérifiée.

Dans la suite, on représente une telle antichaine A par le type `int list`.

28. Écrire une fonction `is_antichain : order -> int list -> bool` prenant en arguments une matrice représentant un ensemble ordonné (V, \preceq) et une liste d'entiers représentant une partie $A \subseteq V$, et renvoyant `true` si A est une antichaine (et `false` sinon).
29. Quelle est la complexité temporelle de votre fonction `is_antichain` dans le pire des cas ?

2.3 Couverture par des chaînes

Définition 2.5 : couverture

Soit (V, \preceq) un ensemble ordonné. On dit qu'une famille $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$ de chaînes disjointes de V est une **couverture** de V si :

$$\bigcup_{\ell=1}^{\lambda} C_\ell = V$$

30. Soit $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$ une couverture de (V, \preceq) , et soit A une antichaine de (V, \preceq) de cardinal $|A| = \alpha$. Montrer que $\alpha \leq \lambda$.

Définition 2.6 : graphe des poursuites

Soit (V, \preceq) un ensemble ordonné de cardinal n .

On construit une copie V' de V avec la numérotation suivante : si $v_i \in V$, la copie de v_i dans V' est notée v_{n+i} . Ainsi, les copies des v_i sont numérotées dans le même ordre que les v_i , mais avec des numéros dans $\llbracket n, 2n - 1 \rrbracket$.

On note $\tilde{V} = V \cup V'$, et on définit sur \tilde{V} la relation \mathcal{R}_{gp} suivante :

$$\forall (i, j) \in \llbracket 0, n - 1 \rrbracket^2, \text{ si } i \neq j \text{ et } v_i \preceq v_j, \text{ alors } v_i \mathcal{R}_{gp} v_{n+j} \text{ et } v_{n+j} \mathcal{R}_{gp} v_i.$$

\mathcal{R}_{gp} est donc symétrique, et met en relation des éléments de V avec des éléments de V' .

L'ensemble $(\tilde{V}, \mathcal{R}_{gp})$ est appelé le **graphe des poursuites** de V .

```
type pursuit_graph = int list array
```

On représente un tel graphe des poursuites par un tableau `gp` de longueur $2n$ du type ci-dessus, tel que $\forall i \in \llbracket 0, 2n - 1 \rrbracket$, `gp.(i)` contienne la liste des numéros $j \in \llbracket 0, 2n - 1 \rrbracket$ des éléments tels que $v_i \mathcal{R}_{gp} v_j$.

31. Écrire une fonction `pursuit_graph_of_order : order -> pursuit_graph` prenant en argument une matrice représentant un ensemble ordonné (V, \preceq) et renvoyant un tableau de listes représentant son graphe des poursuites.

Définition 2.7 : couplage

Soit (V, \leq) un ensemble ordonné de cardinal n , et soit $(\tilde{V}, \mathcal{R}_{gp})$ son graphe des poursuites. On appelle **couplage** un ensemble M de couples de la forme $(v_i, v_{n+j}) \in V \times V'$ avec $v_i \mathcal{R}_{gp} v_{n+j}$ tel que $\forall v \in \tilde{V}$, v n'apparaît au plus qu'une fois dans M .

32. Pour tout couplage M du graphe des poursuites $(\tilde{V}, \mathcal{R}_{gp})$, on construit la partition $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$ de l'ensemble V telle que :
- $\forall (i, j) \in \llbracket 0, n-1 \rrbracket^2$, si (v_i, v_{n+j}) est un couple de M , alors $\exists \ell \in \llbracket 1, \lambda \rrbracket$ tel que $v_i \in C_\ell$ et $v_j \in C_\ell$;
 - le cardinal λ est aussi grand que possible.

Montrer que $\forall \ell \in \llbracket 1, \lambda \rrbracket$, l'ensemble C_ℓ est une chaîne de (V, \leq) .

On représente en OCaml un couplage M du graphe des poursuites $(\tilde{V}, \mathcal{R}_{gp})$ par un tableau `m` de type `int array` de longueur n tel que $\forall j \in \llbracket 0, n-1 \rrbracket$, `m.(j)` contient :

- i avec $i \in \llbracket 0, n-1 \rrbracket$ si $(v_i, v_{n+j}) \in M$;
- -1 si un tel i n'existe pas.

33. Écrire une fonction `chains_of_matching` : `int array` \rightarrow `int list list` prenant en argument un tableau représentant un couplage M du graphe des poursuites d'un ensemble ordonné (V, \leq) et renvoyant une liste de chaînes disjointes tirée du couplage M en respectant les contraintes de la question 32.
34. Soit $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$ une couverture fixée de V . Expliquer comment construire un couplage M tel que la construction de la question 32 conduit à la partition $(C_\ell)_{\ell \in \llbracket 1, \lambda \rrbracket}$.
35. Montrer que la construction de la question 32 fournit une couverture de l'ensemble V par un nombre minimum de chaînes si et seulement si elle s'applique à un couplage M de cardinal maximum.

On admet que la fonction suivante (que vous verrez en cours l'an prochain) permet de calculer un couplage de cardinal maximum à partir d'un graphe des poursuites $(\tilde{V}, \mathcal{R}_{gp})$ d'un ensemble ordonné (V, \leq) :

```

1 let maximum_matching (gp : pursuit_graph) : int array =
2   let n = (Array.length gp) / 2 in
3   let m = Array.make n (-1) in
4   let rec augment i visited =
5     let visit j' =
6       let j = j' - n in
7       match visited.(j) with
8       | true -> false
9       | false -> visited.(j) <- true ;
10      match m.(j) = -1 || augment m.(j) visited with
11      | false -> false
12      | true -> m.(j) <- i ; true
13    in List.exists visit gp.(i)
14  in
15  for i = 0 to n-1 do
16    let _ = augment i (Array.make n false) in ()
17  done ;
18  m

```

36. Quelle est la complexité temporelle dans le pire des cas de la fonction `maximum_matching` ?
37. Écrire une fonction `minimum_cover` : `order` \rightarrow `int list list` prenant en argument une matrice représentant un ensemble ordonné (V, \leq) et renvoyant une couverture de V ayant un nombre minimal de chaînes.
38. Quelle est la complexité temporelle dans le pire des cas de votre fonction `minimum_cover` ?