Lycée Faidherbe, 2020-2021

D.S. d'informatique

MP1 option informatique

Version du 21 mars 2021

Table des matières

Devoir A

DS1 Centrale 2018

À travers l'étude d'un jeu de société, ce sujet s'intéresse aux mouvements de robots, qui possèdent des capacités limitées de localisation. Avec le développement de la robotique, plusieurs problèmes de ce type font l'objet de nombreuses recherches : parcours minimum pour examiner une surface donnée, stratégies collectives avec plusieurs robots en interaction proche, nombre de robots nécessaires pour que tous les points d'une surface avec obstacles soient accessibles, etc.
Ce sujet porte sur la résolution de la situation pratique du jeu Ricochet Robots créé par Alex Randolph en 1999. Ce jeu se déroule sur un plateau de 16×16 cases, avec 4 robots (un robot principal et 3 autres indiscernables) et des murs. À chaque mouvement, le joueur choisit un robot qu'il déplace, dans une des quatre directions, jusqu'à ce qu'il rencontre un obstacle (un mur ou autre robot). Le but est de trouver le nombre de coups minimal pour déplacer le robot principal de son point de départ jusqu'à une case précise du plateau. Un exemple en 8 coups est donné ci-dessous.

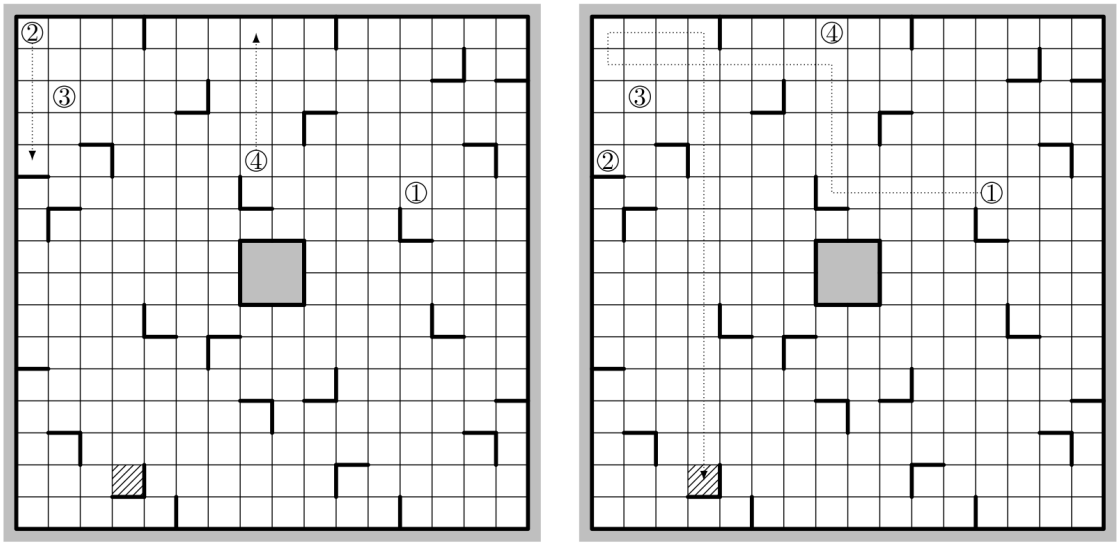


Figure A.1 – Le jeu des robots : le but est d'amener le robot 1 sur la case hachurée.
À gauche : deux déplacements des robots 2 et 4 ; à droite : six déplacements du robot 1.
Le jeu est résolu en 8 mouvements (solution optimale)

On rappelle la définition des fonctions suivantes, disponibles dans la bibliothèque standard :

- `Array.copy : 'a array -> 'a array` telle que l'appel `Array.copy v` renvoie un nouveau tableau contenant les valeurs contenues dans `v` ;
- `Array.make : int -> 'a -> 'a array` telle que l'appel `Array.make n x` renvoie un nouveau tableau de longueur `n` initialisé avec des éléments égaux à `x` ;
- `Array.make_matrix : int -> int -> 'a -> 'a array array` telle que l'appel `Array.make_matrix p q x` renvoie une nouvelle matrice à `p` lignes et `q` colonnes initialisée avec des éléments égaux à `x`.

1 Déplacement d'un robot dans une grille

On considère pour le moment une grille sans robots du jeu Ricochet Robots. Notons N le nombre de cases par ligne et colonne de la grille (16 dans le jeu originel). *Dans les fonctions demandées, on supposera que N est une variable globale.* On numérote chaque case par un couple (a, b) de $\{0, 1, \ldots, N-1\}^2$, correspondant à la ligne a et à la colonne b. On numérote également les lignes horizontales et verticales séparant les cases à l'aide d'un entier de $\{0, 1, \ldots, N\}$, de sorte que la case (a, b) est délimitée par les lignes horizontales a (au dessus) et $a+1$ (en dessous), de même que par les lignes verticales b (à gauche) et $b+1$ (à droite).

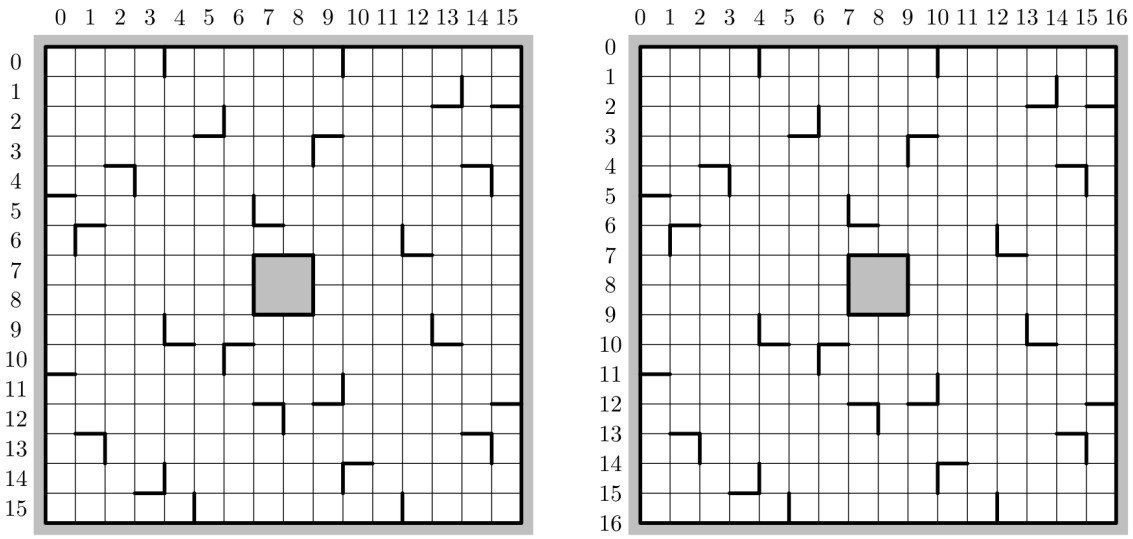


Figure A.2 – À gauche : numérotation des cases par ligne/colonne ; à droite : numérotation des lignes horizontales et verticales

Pour représenter en OCaml la grille avec ses obstacles, on se donne deux tableaux de taille N. Le premier contient les obstacles verticaux sur chacune des lignes, le second contient les obstacles horizontaux sur chacune des colonnes. Un obstacle est donné par le numéro de la ligne (verticale ou horizontale) auquel il appartient. Les obstacles sur une ligne (ou colonne) sont donnés sous la forme d'un tableau ordonné dans l'ordre croissant. Par exemple, la représentation du plateau ci-dessus serait

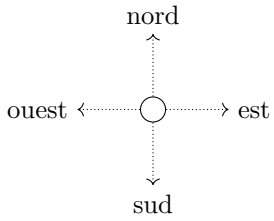
```
let obstacles_lignes =
  [| [|0; 4; 10; 16|]; [|0; 14; 16|]; [|0; 6; 16|];
     [|0; 9; 16|]; [|0; 3; 15; 16|]; [|0; 7; 16|];
     [|0; 1; 12; 16|]; [|0; 7; 9; 16|]; [|0; 7; 9; 16|];
     [|0; 4; 13; 16|]; [|0; 6; 16|]; [|0; 10; 16|];
     [|0; 8; 16|]; [|0; 2; 15; 16|]; [|0; 4; 10; 16|];
     [|0; 5; 12; 16|] |];;

let obstacles_colonnes =
  [| [|0; 5; 11; 16|]; [|0; 6; 13; 16|]; [|0; 4; 16|];
     [|0; 15; 16|]; [|0; 10; 16|]; [|0; 3; 16|];
     [|0; 10; 16|]; [|0; 6; 7; 9; 12; 16|];
     [|0; 7; 9; 16|]; [|0; 3; 12; 16|]; [|0; 14; 16|];
     [|0; 16|]; [|0; 7; 16|]; [|0; 2; 10; 16|];
     [|0; 4; 13; 16|]; [|0; 2; 12; 16|] |];;
```

Notez que les bordures de la grille sont considérées comme des obstacles. Ainsi, les entiers 0 et N sont présents dans les tableaux associés à chaque ligne/colonne.

Question 1
Écrire une fonction `dichotomie a t` *de signature* `int -> int array -> int` *telle que si* `t` *est un tableau d'entiers strictement croissants et* `a` *un élément supérieur ou égal au premier élément du tableau et strictement inférieur au dernier, la fonction renvoie l'unique indice* `i` *tel que* `t.(i)` $\leq$ `a` $<$ `t.(i+1)`. *La fonction doit avoir une complexité logarithmique en la taille du tableau.*

On considère un robot positionné en (a, b), avec $0 \leq a, b < N$. Il peut se déplacer dans les quatre directions cardinales ouest/est/nord/sud représentées ci-dessous



Question 2
Écrire une fonction `deplacements_grille (a,b)` *de signature* `int * int -> (int * int) array` *fournissant les 4 cases atteintes par les déplacements en question, sous forme d'un tableau à 4 éléments (ouest/est/nord/sud). Si le robot ne peux pas bouger dans une direction donnée (car il est contre un obstacle), on considérera que le résultat du déplacement dans cette direction est la case* (a, b) *elle-même.*
Les deux tableaux `obstacles_lignes` *et* `obstacles_colonnes` *sont des variables globales.*

Question 3
Écrire une fonction `matrice_deplacements ()`*, de type* `unit -> (int * int) array array array` *produisant une matrice* `m` *telle que* `m.(a).(b)` *contienne le arrayeur des déplacements possibles pour un robot depuis la case* (a, b)*, et ce pour tous* $0 \leq a, b < N$. *Donner la complexité de création de la matrice.*

On cherche maintenant à intégrer les positions d'autres robots dans le déplacement d'un robot. On utilise la fonction précédente pour créer une matrice `mat_deplacements` que l'on considérera comme globale.

Question 4
Écrire une fonction `modif t (a,b) (c,d)` *de signature*
`(int * int) array -> int * int -> int * int -> unit`
telle que si `t` *est le tableau de taille 4 donnant les déplacements ouest/est/nord/sud d'un robot placé en* (a, b) *dans la grille ne contenant pas d'autres robots, et* (c, d) *la position d'un autre robot, alors la fonction modifie si nécessaire le tableau* `t` *en prenant en compte le robot en* (c, d)*.*

On s'intéresse maintenant au déplacement d'un robot situé en (a, b) dans la grille, avec d'autres robots éventuellement présents, dont les positions sont stockées dans une liste.

Question 5
Déduire des questions précédentes une fonction `deplacements_robots (a,b) q` *de signature*
`int * int -> (int * int) list -> (int * int) array`
donnant les déplacements ouest/est/nord/sud d'un robot situé en (a, b) *dans la grille, les positions des autres robots étant stockées dans la liste* `q`*. On ne modifiera pas la matrice* `mat_deplacements` *: on souhaite une copie modifiée de* `mat_deplacements.(a).(b)`*.*

Question 6
Si on suppose que la solution optimale demande au plus k *mouvements, une solution possible pour résoudre le jeu Ricochet Robots consiste à générer toutes les suites possibles de* k *déplacements. Avec 4 robots en tout, estimer la complexité d'une telle approche (on utilisera la notation* $\mathcal{O}$*).*

La suite du problème a pour objet de proposer une solution plus efficace pour la résolution du jeu Ricochet Robots.

2 Quelques fonctions utilitaires

2.1 Une fonction de tri

Question 7
Écrire une fonction `insertion x q` *de signature* `'a -> 'a list -> 'a list` *prenant en entrée un élément* `x` *et une liste* `q` *triée dans l'ordre croissant, et renvoyant une liste triée dans l'ordre croissant, constituée des éléments de* `q` *et* `x`*.*

Question 8
En déduire une fonction `tri_insertion q` *de signature* `'a list -> 'a list` *permettant de trier une liste dans l'ordre croissant.*

Question 9
Rappeler la complexité de ce tri dans le pire et le meilleur cas. Que peut-on dire de la complexité si dans la liste `q`*, tous les éléments excepté peut-être un sont dans l'ordre croissant ?*

2.2 Quelques fonctions sur les listes

Question 10
Écrire une fonction `mem1 x q` *de signature* `'a -> ('a * 'b) list -> bool` *testant l'appartenance d'un couple dont le premier élément est* `x` *dans la liste* `q`*.*

Question 11
Écrire une fonction `assoc x q` *de signature* `'a -> ('a * 'b) list -> 'b` *renvoyant, s'il existe, l'élément* `y` *du premier couple* `(x,y)` *appartenant à la liste* `q`*.*

3 Tables de hachage

Dans l'optique de résoudre le problème du jeu des robots, nous allons travailler sur un graphe dont les sommets seront étiquetés par les positions des robots. Le nombre de sommets possibles étant élevé, il est nécessaire d'utiliser une structure de données adaptée pour travailler sur ce graphe. Nous allons donc réaliser une structure de dictionnaire permettant, en particulier, de tester facilement si un sommet a déjà été vu ou non et d'associer un sommet à chaque sommet découvert.
Une structure de dictionnaire est un ensemble de couples (clé, élément), les clés (nécessairement distinctes) appartenant à un même ensemble K, les éléments à un ensemble E. La structure doit garantir les opérations suivantes :

- recherche d'un élément connaissant sa clé ;
- ajout d'un couple (clé, élément) ;
- suppression d'un couple connaissant sa clé.

Une structure de dictionnaire peut-être réalisée à l'aide d'une table de hachage. Cette table est implantée dans un tableau de w listes (appelées *alvéoles*) de couples (clé, élément). Ce tableau est organisé de façon à ce que la liste d'indice i contienne tous les couples (k, e) tels que $h_w(k) = i$ où $h_w : K \mapsto \{0, 1, \ldots, w-1\}$ s'appelle *fonction de hachage*. On appelle w la *largeur de la table* de hachage et $h_w(k)$ le *haché* de la clé k.
Ainsi pour rechercher ou supprimer l'élément de clé k, on commence par calculer son haché qui détermine l'alvéole adéquate et on est alors ramené à une action sur la liste correspondante. De même pour ajouter un nouvel élément au dictionnaire on l'ajoute à l'alvéole indiquée par le haché de sa clé.

3.1 Une famille de fonctions h_w

Nous commençons par nous doter d'une famille de fonctions h_w, pour les listes de couples de $\{0, 1, \ldots, N-1\}^2$. Un hachage naturel d'une liste comportant les couples $(a_i, b_i)_{0 \leq i < p}$ avec $0 \leq a_i, b_i < N - 1$ est donnée par :

$$P_w(N) = \left(\sum_{i=0}^{p-1} (a_i + b_i N) N^{2i} \right) \text{ modulo } w$$

Autrement dit, on évalue le polynôme dont les coefficients sont donnés par les a_i et b_i en N, et on ne considère que le reste dans la division euclidienne par w. On rappelle qu'on a supposé que N est une variable globale.

Question 12
Écrire une fonction récursive `hachage_liste w q` *de signature* `int -> (int * int) list -> int` *calculant la quantité précédente.*

3.2 Tables de hachage de largeur fixée

choix pour w serait par exemple un nombre premier ni trop petit, ni trop grand, comme 997. Pour les listes, on considérerait alors la fonction de hachage h_{997} donnée en OCaml par `hachage_liste 997`. On définit en toute généralité le type suivant :

```
type ('a, 'b) table_hachage = {
  hache: 'a -> int;
  donnees: ('a * 'b) list array;
  largeur: int};;
```

Implantation de la structure de dictionnaire

Question 13
Écrire une fonction `creer_table h w` *de signature* `('a -> int) -> int -> ('a, 'b) table_hachage` *telle que* `creer_table h w` *renvoie une nouvelle table de hachage vide de largeur* `w` *munie de la fonction de hachage* `h`.

Question 14
Écrire une fonction `recherche t k` *de signature* `('a, 'b) table_hachage -> 'a -> bool` *renvoyant un booléen indiquant si la clé k est présente dans la table t. On pourra utiliser les fonctions de la partie* **2**.

Question 15
Écrire une fonction `element t k` *de signature* `('a, 'b) table_hachage -> 'a -> 'b` *renvoyant l'élément e associé à la clé k dans la table t, si cette clé est bien présente dans la table.*

Question 16
Écrire une fonction `ajout t k e` *de signature* `('a, 'b) table_hachage -> 'a -> 'b -> unit` *ajoutant l'entrée (k, e) à la table de hachage t. On n'effectuera aucun changement si la clé est déjà présente.*

Question 17
Écrire enfin une fonction `suppression t k` *de signature* `('a, 'b) table_hachage -> 'a -> unit` *supprimant l'entrée de la clé k dans la table t. On n'effectuera aucun changement si la clé n'est pas présente.*

Étude de la complexité de la recherche d'un élément

Nous étudions ici la complexité de la recherche d'une clé dans une table de hachage. Dans le pire cas, toutes les clés sont hachées vers la même alvéole, ainsi la complexité de la recherche d'une clé dans une table de hachage n'est pas meilleure que la recherche dans une liste. Cependant, si la fonction de hachage h_w est bien choisie, on peut espérer que les clés vont se répartir de façon apparemment aléatoire dans les alvéoles, ce qui donnera une complexité bien meilleure.
Nous faisons donc ici l'hypothèse de *hachage uniforme simple* : pour une clé donnée, la probabilité d'être hachée dans l'alvéole i est $1/w$, indépendante des autres clés. On note n le nombre de clés stockées dans la table et on appelle $\alpha = n/w$ le *facteur de remplissage* de la table. On suppose de plus, que le calcul du haché d'une clé se fait en temps constant.

Question 18
On se donne une clé k non présente dans la table. Montrer que l'espérance de la complexité de la recherche de k dans la table est un $\mathcal{O}(1+\alpha)$.

Question 19
On prend au hasard une clé présente dans la table ; toutes les clés sont équiprobables. Montrer qu'alors la recherche de la clé se fait en $\mathcal{O}(1+\alpha)$, en moyenne sur toutes les clés présentes.

3.3 Tables de hachage dynamique

Les deux questions précédentes montrent que l'on peut assurer une complexité moyenne constante pour la recherche dans une table de hachage, sous réserve que le facteur de remplissage α soit borné. Il en va de même des opérations d'insertion et de suppression, pour peu que les clés à ajouter/supprimer vérifient des hypothèses d'indépendance. Bien souvent, et cela va être le cas dans notre problème, on ne sait pas à l'avance quel sera le nombre de clés à stocker dans la table, et on préfère ne pas surestimer ce nombre pour garder un espace mémoire linéaire en le nombre de clés stockées. Ainsi, il est utile de faire varier la largeur w de la table de hachage : si le facteur de remplissage devient trop important, on réarrange la table sur une largeur plus grande (de même, on peut réduire la largeur de la table lorsque le facteur de remplissage devient petit). On parle alors de tables de hachage dynamiques pour ces tables à largeur variable.
À une table de hachage dynamique est associée une *famille de fonctions de hachage* (h_w). Par exemple, pour les listes de couples de $\{0, 1, \ldots, N-1\}^2$, la fonction `hachage_liste` précédemment écrite fournit une telle famille. On définit en toute généralité le type suivant :

```
type ('a,'b) table_dyn =
     {hache: int -> 'a -> int;
      mutable taille: int;
      mutable donnees: ('a * 'b) list array;
      mutable largeur: int};;
```

On notera trois différences par rapport au type précédent :

- la fonction `hache` possède un paramètre supplémentaire qui est la largeur de hachage, elle correspond maintenant à la famille de fonctions de hachage (h_w) ;
- on a rendu les champs `donnees` et `largeur` modifiables ;
- un champ `taille` (modifiable) est rajouté, il doit à tout moment contenir le nombre de clés présentes dans la table.

Question 20
Écrire une fonction `creer_table_dyn h` *permettant de créer une table de hachage dynamique initialement vide, avec la famille de fonctions de hachage* `h` *et la largeur initiale 1.*

On admet avoir écrit deux fonctions `recherche_dyn t k` et `element_dyn t k`, variantes des fonctions `recherche` et `element` précédentes, basées sur le même principe. On va maintenant développer une stratégie pour maintenir à tout moment un facteur de remplissage borné.

Question 21
Écrire une fonction `rearrange_dyn t w2` *prenant en entrée une table de hachage dynamique et une nouvelle largeur de hachage* `w2`*, qui réarrange la table sur une largeur* `w2`*. En supposant que le calcul des valeurs de hachage se fasse en temps constant, la complexité doit être en* $\mathcal{O}(n + w + w_2)$ *où* n *est le nombre de clés présentes dans la table (sa taille),* w *est l'ancienne largeur de la table,* w_2 *la nouvelle.*

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné, tout en garantissant une bonne répartition des clés dans le cas des listes de couples à valeurs dans $\{0, 1, \ldots, N-1\}$ avec $N = 16$, est d'utiliser les puissances de 3 comme largeurs de hachage. Après ajout d'un élément à la table, si celle-ci est de taille strictement supérieure à trois fois sa largeur w, on la réarrange sur une largeur $w' = 3w$.

Question 22
Écrire une fonction `ajout_dyn t k e` *ajoutant le couple* (k, e) *à la table de hachage (si la clé* k *n'est pas présente), en réarrangeant si nécessaire la table, en suivant le principe ci-dessus.*

Dans l'hypothèse que chaque ajout se fait en temps $\mathcal{O}(1 + \alpha)$, où α est le facteur de remplissage de la table, on peut montrer qu'une série de p ajouts dans une table initialement vide prend un temps $\mathcal{O}(p)$.

On pourrait écrire de même une fonction de suppression dynamique, de sorte de maintenir un facteur de remplissage de la table borné, et qu'une série de p opérations licites d'insertion/suppression dans la table prenne un temps $\mathcal{O}(p)$.

4 Solutions

Solution de l'exercice 1 - La fonction demandée s'appelle `dichotomie` et on demande une complexité logarithmique : on va utiliser une recherche par dichotomie. L'idée est de travailler avec deux indices, i et j tels que `t.(i) <= a < t.(j)` : c'est un invariant de boucle.

```
let dichotomie a t =
  let n = Array.length t in
  let i = ref 0 in
  let j = ref (n -1) in
    while !i + 1 < !j do
      let c = (!i + !j)/2 in
      if a < t.(c) then j := c
                   else i := c done;
    !i;;
```

On peut aussi donner une écriture récursive

```
let dichotomie a t =
  let rec aux i j =
    if j = i + 1
    then i
    else let c = (i+j)/2 in
         if a < t.(c)
         then aux i c
         else aux c j
  in aux 0 (Array.length t  - 1);;
```

On suppose qu'on a $j - i \leqslant 2^p$ on a alors $i + 2^{p-1} \leqslant c \leqslant j - 2^{p-1}$, les nouvelles valeurs de i et j, i' et j', vérifient alors $j' - i' \leqslant 2^{p-1}$. Si on a $2^{N-1} < n - 1 - 0 \leqslant 2^N$ alors, après au plus N passages dans la boucle on a $j - i \leqslant 1$ et la boucle termine.
La complexité est donc majorée par N qui est un $\mathcal{O}\big(\log(n)\big)$.

Solution de l'exercice 2 - On considère la case (a, b).
Sur la ligne d'ordonnée b les obstacles sont aux positions dans `obstacles_ligne.(b)`, noté `t`.
Si p est la longueur de `t` on a `t.(0) = 0 <= a < n = t.(p-1)` ; on peut calculer `dichotomie a t`.
On trouve un indice i tel que `t.(i) <= a < t.(i+1)`. Ainsi `t.(i)` est le dernier obstacle à gauche de a et `t.(i+1)` est le premier obstacle à droite de a (sur la ligne b).
On en déduit que le déplacement à gauche va s'arrêter à la case `t.(i)` et le déplacement à droite va s'arrêter à la case `t.(i+1) - 1`. De même pour les déplacement nord-sud.

```
let deplacement_lignes (a, b) =
  let t = obstacles_lignes.(b) in
  let i = dichotomie a t in
  let ouest = t.(i) in
  let est = t.(i+1) -1 in
  let s = obstacles_colones.(a) in
  let j = dichotomie b t in
  let nord = s.(j) in
  let sud = s.(j+1) -1 in
  [| (ouest, b); (est, b); (a, nord); (a, sud) |];;
```

Comme on a utilisé deux fois `dichotomie` donc la complexité est en $\mathcal{O}\big(\log(n)\big)$.

Solution de l'exercice 3 - On construit une matrice vide qu'on remplit case par case avec la fonction précédente

```
let matrice_deplacements () =
  let m = Array.make_matrix n n [||] in
  for a = 0 to n - 1 do
    for b = 0 to n - 1 do
      m.(a).(b) <- deplacements_grille (a, b) done done;
  m;;
```

En raison des deux boucles imbriquées, la complexité est en $\mathcal{O}\big(n^2 log(n)\big)$.

Solution de l'exercice 4 - Un robot en (c, d) devient un obstacle possible en c pour les déplacements vers l'ouest des points (a, d) avec $a > c$ et un obstacle possible en c pour les déplacements vers l'est des points (a, d) avec $a < c$. On suppose ici que les deux points sont distincts.
On a choisi de faire 4 tests séparés pour que la structure reste lisible.

```
let modif t (a, b) (c, d) =
  if b = d && a > c
  then t.(0) <- (max (c+1) (snd t.(0)), b);
  if b = d && a < c
  then t.(1) <- (min (c-1) (snd t.(1)), b);
  if a = c && b > d
  then t.(2) <- (a, max (d+1) (snd t.(2)));
  if a = c && b < d
  then t.(3) <- (a, min (d-1) (snd t.(3)));;
```

Solution de l'exercice 5 - On commence par copier le vecteur de déplacements de la matrice de déplacements. On traite la liste à l'aide d'une fonction récursive auxiliaire.

```
let deplacements_robots (a, b) q =
  let t = Array.copy mat_deplacements.(a).(b) in
  let rec aux q =
    match q with
    |[] -> ()
    |(c, d)::reste -> modif t (a, b) (c, d); aux reste
  in maj q;   t;;
```

Solution de l'exercice 6 - À chaque itération, on peut déplacer chacun des 4 robots dans une des 4 directions : cela fait un total de 16 déplacements possibles.
On en déduit que le nombre de suites de k déplacements est 16^k.
La complexité est donc un $\mathcal{O}(16^k)$: elle n'est possible que pour des valeurs très petites de k.

Solution de l'exercice 7 -

```
let rec insertion x liste =
  match liste with
  |[] -> [x]
  |y::reste when x <= y -> x::liste
  |y::reste -> y::(insertion x reste);;
```

Solution de l'exercice 8 -

```
let rec tri_insertion liste =
  match liste with
  |[] -> []
  |t::reste -> insertion t (tri_insertion reste);;
```

Solution de l'exercice 9 - Dans le pire des cas, la complexité est quadratique (par exemple pour une liste décroissante).
Dans le meilleur des cas, la complexité est linéaire (pour une liste triée).
Soit $L = (a_0, \ldots, a_k, x, a_{k+1}, \ldots a_{n-1})$ une liste, telle que seul l'élément x n'est pas à sa place.
le tri se fait en commençant par la fin.

- Le tri de la sous-liste triée $(a_{k+1}, \ldots, a_{n-1})$ est en $O(1)$ pour chaque élément.
- Si $x > a_{k+1}$, alors l'insertion de x se fera en l'insérant progressivement à droite jusqu'à sa position. Cette insertion est alors en $O(n)$, et les insertions suivantes se feront alors en $O(1)$ car la liste est triée. Au final, la complexité est en $O(n)$.
- Si $x < a_k$, alors l'insertion de x se fera en $O(1)$. Les éléments qui précèdent sont, pour certains, plus grands que x : ils seront insérés derrière x, ce qui se fera avec deux appels à la fonction `insertion`, donc en $O(1)$. Les autres se feront en un seul appel.
  Au final, la complexité est en $O(N)$.

Quand un seul élément n'est pas à sa place, la complexité est en $O(N)$.

Solution de l'exercice 10 -

```
let rec mem1 x liste =
  match liste with
  |[] -> false
  |(a, b)::reste when x = a -> true
  |t::reste-> mem1 x reste;;
```

Solution de l'exercice 11 -

```
let assoc x liste =
  match liste with
  |[] -> failwith "Element non present"
  |(a, b)::reste when x = a -> b
  |t::reste-> assoc x reste;;
```

Solution de l'exercice 12 - Il n'y a pas de fonction puissance en Caml. On va utiliser la règle de Hörner pour faire le calcul : elle évite de calculer les puissances de N.

```
let rec hachage_liste w liste =
  match liste with
  |[] -> 0
  |(a, b)::reste ->
    ((hachage_liste w reste)*n*n + a + b*n) mod w;;
```

Solution de l'exercice 13 - On n'a pas besoin des type `'a` et `'b` : on remplit le tableaux de listes vides.

```
let creer_table h w =
  let tab = Array.make w [] in
  {hache = h; donnees = tab; largeur = w};;
```

On notera que `largeur` est inutile, c'est la longueur de `donnees`.

Solution de l'exercice 14 - On utilise les fonctions de la partie précédente pour chercher dans la liste définie par la fonction de hachage.

```
let recherche table k =
  let ind = table.hache k in
  let liste = table.donnees.(ind) in
  mem1 k liste;;
```

Solution de l'exercice 15 -

```
let element table k =
  let ind = table.hache k in
  let liste = table.donnees.(ind) in
  assoc k liste;;
```

Solution de l'exercice 16 -

```
 let ajout table k e =
  if not (recherche table k)
  then let ind = table.hache k in
       table.donnees.(ind) <- (k, e) :: table.donnees.(ind);;
```

Solution de l'exercice 17 - On écrit une fonction auxiliaire qui supprime un élément d'une liste.

```
let suppression table k =
  let rec supp liste =
    match liste with
    |[] -> []
    |(a, b)::reste when a = k -> reste
    |x::reste -> x::(supp reste) in
  let ind = ind = table.hache k in
  table.donnees.(ind) <- supp t.donnees.(ind);;
```

Solution de l'exercice 18 - La complexité est donc la somme de la complexité du calcul du hache et de la recherche dans l'alvéole. Pour une clé qui n'est pas trouvée, la complexité de la recherche est proportionnelle à la longueur de la liste.
On note L_i, la longueur de la liste dans l'alvéole d'indice i.
La complexité moyenne d'une recherche quand le hache est i est donc majorée par $K(1+L_i)$, où K majore la complexité du calcul de l'indice et celle d'une comparaison. La complexité moyenne vérifie donc

$$C(n) \leqslant \sum_{i=0}^{w-1} \frac{1}{w} K(1+L_i) = \frac{K}{w}\left(\sum_{i=0}^{w-1} 1 + \sum_{i=0}^{w-1} L_i\right) = \frac{K}{w}(w+n) = K(1+\alpha)$$

Solution de l'exercice 19 - L'équi-probabilité porte ici sur les n valeurs dans la table, pas sur les w alvéoles.
On s'intéresse aux éléments insérés dans la table selon leur ordre d'insertion : $x_1, x_2, \ldots, x_n$.
On note X_k la variable aléatoire donnant le rang de l'élément x_k dans la liste, à partir de 0 ; X_k vaut p si p élément sont placés dans la même alvéole **après** x_k, c'est-à-dire aux rangs $k+1$ à n.
On a donc $P(X_k = p) = \binom{n-k}{p}\left(\frac{1}{w}\right)^p\left(\frac{w-1}{w}\right)^{n-k-p}$.
S'il y a p autres éléments dans la liste avant x_k, sa recherche va demander $p+1$ comparaisons donc l'espérance du nombre de comparaisons pour la recherche de x_k est $1+\mathrm{E}(X_k) = 1+\frac{n-k}{w}$.

L'espérance du nombre de recherche d'un élément présent est donc

$$R(n) \leqslant \sum_{k=1}^{n} \frac{1}{n}\left(1+\frac{n-k}{w}\right) = 1+\frac{1}{nw}\sum_{k=1}^{n}(n-k) = 1+\frac{1}{nw}\sum_{i=0}^{n-1} i = 1+\frac{n-1}{2w} = 1+\frac{\alpha}{2}-\frac{1}{w}$$

On a donc $C(n) \leqslant K\left(1+1+\frac{\alpha}{2}-\frac{1}{w}\right) \leqslant K\left(2+\frac{\alpha}{2}w\right) \leqslant 2K(1+\alpha)$

Solution de l'exercice 20 -

```
let creer_table_dyn h =
  let tab = Array.make 1 [] in
  {hache = h; taille = 0; donnees = tab; largeur = 1};;
```

Solution de l'exercice 21 - On crée une fonction auxiliaire `ajout` qui ajoute tous les éléments d'une liste à un tableau de liste créé précédemment, en hachant chaque clé avec la nouvelle fonction h_{w_2}. On applique cette fonction aux listes de la table de hachage.

```
let rearrange_dyn table w2 =
  let d2 = Array.make w2 [] in
  let h2 = table.hache w2 in
  let rec ajout liste =
    match liste with
    |[] -> ()
    |(a, b)::reste -> d2.(h2 a) <- (a, b)::(d2.(h2 a); ajout
       reste
  in for i = 0 to t.largeur - 1 do
      ajout table.donnees.(i) done;
     t.donnees <- d2;
     t.largeur <- w2;;
```

Solution de l'exercice 22 -

```
let ajout_dyn table k e =
let w  = table.largeur in
  if not (recherche_dyn table k)
  then begin let ind = table.hache w k in
         table.donnees.(ind) <- (k, e)::(table.donnees.(ind));
         table.taille <- table.taille + 1 end;
  if table.taille > 3*w then rearrange_dyn t (3*w);;
```

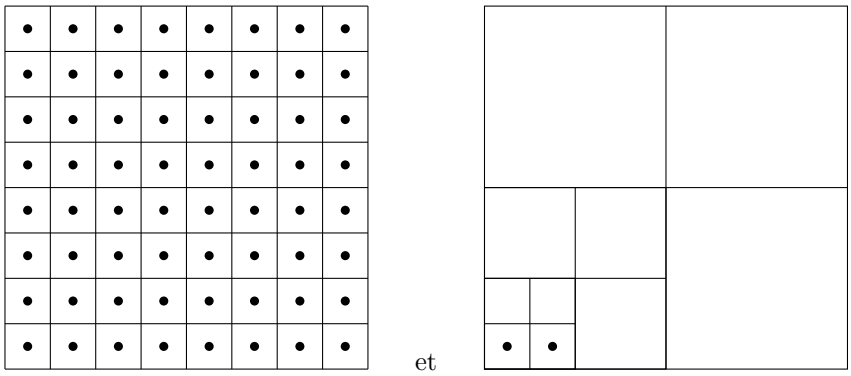
Devoir B

DS1+ X 1997

1 Quadtrees

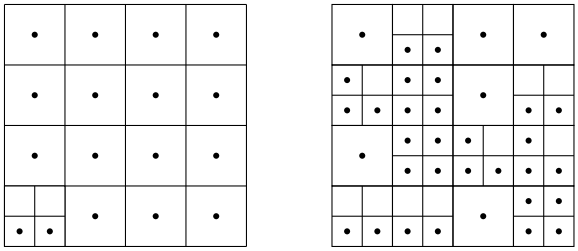
Question 1 — Propriétés des quadtrees

a) On peut proposer



et

b) Un quadtree de profondeur p contient au plus 4^p feuilles donc $4^p \geqslant N$,
c'est-à-dire $p \geqslant \lceil \log_4(N) \rceil = f(N)$.
Pour tout p_0 on peut construire un quadtree à $N_0 = 4^{p_0}$ corps de profondeur p_0 obtenu, comme ci-dessus en subdivisant l'univers en 4^{p_0} carrés, chacun contenant un corps ; on a bien $p_0 = f(N_0)$.
Mais la question est ambigüe, on peut construire un quadtree à N_0 corps de profondeur p_0 pour tout N_0 tel que $f(N_0) = p_0$. En effet, on a alors $4^{p_0-1} < N_0 \leqslant 4^{p_0}$. On construit un quadtree de profondeur $p_0 - 1$ complet comme ci-dessus et place les N_0 corps, entre 1 et 4 par nœud. Comme il y a au moins $4^{p_0-1} + 1$ corps on devra subdiviser au moins un nœud et on arrive ainsi à une profondeur de p_0.

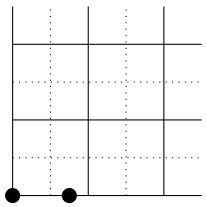


c) À partir d'un quadtree t_p contenant N corps avec $N \geqslant 2$ et de profondeur p, on peut

construire un quadtree t_{p+1} de profondeur $p+1$ avec N corps en joignant t_p et 4 nœuds vides. On peut donc construire des quadtrees à N corps de profondeur quelconque : il n'y a pas de majorant fonction de N de la profondeur.

d) Dans un quadtree de profondeur p il y a au moins un carré au niveau $p-1$ contenant 2 corps. La distance entre deux corps distincts inclus dans ce carré est majorée par la longueur du côté, $\frac{D_\mathcal{U}}{2^{p-1}}$ d'où, pour le minimum des distances, $\delta \leqslant \frac{D_\mathcal{U}}{2^{p-1}}$.
On en déduit $p \leqslant 1 + \log_2\left(\frac{D_\mathcal{U}}{\delta}\right)$, puis, comme p est entier, $p \leqslant 1 + \left\lfloor \log_2\left(\frac{D_\mathcal{U}}{\delta}\right)\right\rfloor$.
Inversement, si on pose $p_0 = 1 + \left\lfloor \log_2\left(\frac{D_\mathcal{U}}{\delta}\right)\right\rfloor$,
on a $\frac{D_\mathcal{U}}{2^p} < \delta \leqslant \frac{D_\mathcal{U}}{2^{p-1}}$. On peut alors placer deux corps en $(0,0)$ et $(\delta,0)$ et un quadtree qui les contient devra être de profondeur p_0 pour les séparer.
La borne supérieure de la profondeur est bien

$$1 + \left\lfloor \log_2\left(\frac{D_\mathcal{U}}{\delta}\right)\right\rfloor$$



e) Le plus petit nombre strictement positif représentable est 2^e donc $\delta \geqslant 2^e$.
Le plus grand nombre représentable est $2^e(2^m-1)$ donc $D_\mathcal{U} \leqslant 2^e(2^m-1)$.
On en déduit $p \leqslant 1 + \left\lfloor \log_2\left(\frac{D_\mathcal{U}}{\delta}\right)\right\rfloor \leqslant 1 + \lfloor \log_2(2^m-1)\rfloor = m$.

2 Construction du quadtree

Question 2 — Opérations élémentaires

a) Ne pas oublier les symboles des opérations sur les flottants.

```
let add v1 v2 = {x = v1.x +. v2.x; y = v1.y +. v2.y};;
```

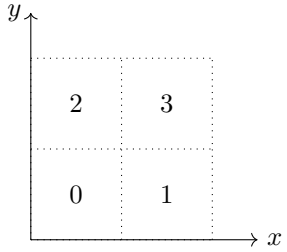
```
let sub v1 v2 = {x=v1.x-.v2.x; y=v1.y-.v2.y};;
```

b) Même remarque

```
let scal m u  = {x = m*.u.x; y = m*.u.y};;
```

c) L'exponentiation demande des termes flottants, même pour l'exposant.
let carre u = u.x**2.0 +. u.y**2.0 ; ;

Question 3 — Indices



On choisit l'ordre suivant

a) On place les nœuds en fonction de leur position par rapport au centre.

```
let indice_fille p_c p =
    (if p_c.y > p.y then 0 else 2)
  + (if p_c.x > p.x then 0 else 1);;
```

b) La taille de la sous-cellule est divisée par 2 et le centre est à la moitié de la taille.

```
let position_fille p_c taille i =
    let t = taille/.4.0 in
    match i with
    | 0 -> add p_c {x =   t; y =   t}
    | 1 -> add p_c {x =   t; y = -.t}
    | 2 -> add p_c {x = -.t; y =   t}
    | _ -> add p_c {x = -.t; y = -.t};;
```

Le dernier motif est laissé ouvert pour que le pattern matching soit exhaustif.

Question 4 — Arbre-univers

a) Si on insère dans un arbre vide, on obtient une feuille,
si on insère dans une feuille, on crée un Noeud et on place le nouveau corps, ainsi que celui qui était dans la feuille, le centre est donné par le vecteur reçu,
si on insère dans un nœud, on insère dans le fils correspondant, on transmet la nouvelle position du centre.
On commence par une fonction qui factorise l'ajout d'un corps dans un tableau, elle sert 3 fois.

```
let rec insere_corps corps arbre p_c taille =
    let ajoute corps filles centre taille =
      let k= indice_fille p_c corps.pos in
      let new_c = position_fille p_c taille ind in
      let t = taille /. 2.0 in
      filles.(k) <- insere_corps corps filles.(k) new_c t in
    match arbre with
   | Vide -> Feuille(corps)
   | Feuille(c) -> let filles = Array.make 4 Vide in
      ajoute c filles p_c taille;
      ajoute corps filles p_c taille;
      Noeud({cm_mass = 0.0;
             cm_pos = vecteur_nul;
             filles = filles})
   | Noeud(cellulle) ->
      ajoute corps cellulle.filles p_c taille;
      arbre;;
```

b) On suppose que tous les corps à insérer ont des positions appartenant au carré $[-\mathcal{D}_\mathcal{U}; \mathcal{D}_\mathcal{U}]^2$.
On suppose aussi que les positions des corps à insérer sont distinctes et, de plus, séparées au sens de la norme introduite en **1.d)** d'au moins 2^e, voir le **1. e)**.
Pour démontrer la terminaison, on va aussi prouver l'algorithme en ce sens qu'il place les corps géométriquement dans des carrés qui les contiennent. On considère la propriété suivante :

- pour tout arbre ne contenant que des corps dont les positions sont dans le carré $[\texttt{p\_c - taille/2; p\_c + taille/2}]^2$,
- pour tout corps `corps` de position dans le carré $[\texttt{p\_c - taille/2; p\_c + taille/2}]^2$,
- l'appel de `insere_corps corps arbre p_c taille` termine.

Le résultat de l'appel fournira un arbre qui, lui aussi, ne contient que des corps dont les positions sont dans le carré $[\texttt{p\_c - taille/2; p\_c + taille/2}]^2$.

On démontre la propriété par induction structurelle.

- Si l'arbre est vide, la fonction construit une feuille et termine.
- Si l'arbre est une feuille, celle-ci contient un unique corps. L'algorithme construit un nœud et place ce corps dans une des branches en remplaçant une fille vide. Il place alors le nouveau corps
  soit dans une autre branche et l'algorithme termine,
  soit dans la même branche avec une taille diminuée de moitié. Cette situation ne peut que se reproduire qu'un nombre fini de fois car sinon, on parviendrait à une taille strictement majorée par 2^e et deux corps dans un carré de côté $c < 2^e$ ce qui a été supposé impossible.
  L'algorithme termine.
- Si `arbre` est un nœud, on suppose que l'algorithme termine si on l'applique aux filles. `indice_fille` et `position_fille` sélectionnent la bonne cellule fille et la fonction `ajoute` effectue un appel récursif sur l'une des filles donc termine.

c) La fonction `insere_corps` effectue un nombre fini d'opérations car elle suit une branche de l'arbre puis, eventuellement prolonge une branche mais la profondeur maximale est bornée d'après la question **1. e)**.
La complexité de l'insertion de N corps est donc linéaire en N.

3 Calculs des forces

Question 5 — Barycentres
La fonction auxiliaire calcule la masse et la somme des positions récursivement et on modifie l'arbre. On calcule la somme plutôt que le barycentre pour ne pas avoir à re-multiplier par les masses. Comme on veut un résultat `unit`, l'appel final doit être encapsulé.

```
let barycentres arbre =
  let rec masse_somme arbre =
    match arbre with
    |Vide -> (0.0, vecteur_nul)
    |Feuille(c) -> (c.mass, scal c.mass c.pos)
    |Noeud(cellule)
          -> let m = ref 0.0 and s = ref vecteur_nul in
             for i = 0 to 3 do
                let mi, si = masse_somme cellule.filles.(i) in
                m := !m +. mi;
                s := add !s si done;
             cellule.cm_mass <- !m;
             cellule.cm_pos <- scal (1.0 /. !m) !s;
             !m, !s
  in let _ = masse_somme arbre in ();;
```

Question 6 — Gravitation

a) On commence par un calcul de la distance entre deux points et un calcul du vecteur accélération, si la distance entre les corps est nulle, c'est qu'on cherche l'accélération d'un corps sur lui-même, elle est nulle.

```
let distance a b =
   let vec = sub a b in
   (carre vec)**0.5 ;;
```

```
let accel point orig masse =
   let vec = sub orig point in
   let r = distance point orig in
   if r = 0.0
   then vecteur_nul
   else scal (masse /. r**3.0) vec;;
```

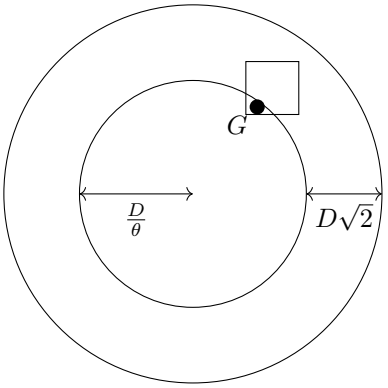
```
let rec grav_approx pos arbre taille =
   match arbre with
   |Vide -> vecteur_nul
   |Feuille(c)  -> accel pos c.pos c.mass
   |Noeud(cl)
     -> let r = distance pos cl.cm_pos in
        if taille < r *. theta
        then accel pos cl.cm_pos cl.cm_mass
        else begin
            let acc = ref vecteur_nul
            and t = taille /. 2.0 in
            for i=0 to 3 do
                acc := add !acc
                           (grav_approx pos cl.filles.(i) t)
            done;
            !acc end ;;
```

b) Le barycentre peut se situer en tout point du carré. Pour qu'une cellule de taille D soit subdivisée, il faut que son centre de masse soit situé à une distance inférieure à $\frac{D}{\theta}$ du corps c ; cette cellule doit donc être entièrement inclus dans le disque de centre c et de rayon $\frac{D}{\theta} + D\sqrt{2}$. Comme toutes les cellules de même taille sont deux à deux disjointes, la somme des aires des cellules divisables est majorée par l'aire du disque d'où, si N est le nombre de ces cellules,

$N.D^2 \leqslant \pi \left(\frac{D}{\theta} + D\sqrt{2}\right)^2$.

Le nombre est donc majoré par

$$K(\theta) = \pi \left(\frac{1}{\theta} + \sqrt{2}\right)^2$$



c) Un appel `grav_approx c arbre taille` engendre au plus $K(\theta)$ appels récursifs pour des nœuds de taille $2^{-k}\times$ `taille` pour tout entier k d'après la question ci-dessus. Comme la profondeur de l'arbre est majorée par m, le nombre total d'appels est majoré par $mK(\theta)$ qui est une constante. Pour N points la complexité est donc un $\mathcal{O}(n)$.

Devoir C

DS2 : Logique

1 CCINP 2016, partie I

Nous nous intéressons dans cet exercice à l'étude de quelques propriétés de la logique propositionnelle tri-valuée. En plus des deux valeurs classiques **vrai** notée $\top$ et **faux** notée $\bot$ que peuvent prendre les variables et les expressions, la logique propositionnelle tri-valuée introduit une troisième valeur **indéterminé** notée ?.
$\mathcal{V}$ est l'ensemble des variables propositionnelles et $\mathcal{F}$ l'ensemble des formules construites sur $\mathcal{V}$. Pour $A, B \in \mathcal{V}$, les tables de vérités des opérateurs classiques dans cette logique propositionnelle sont les suivantes :

A	B	$A \wedge B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$
$\bot$	?	$\bot$
?	$\top$	?
?	$\bot$	$\bot$
?	?	?

A	B	$A \vee B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	?	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$
$\bot$	?	?
?	$\top$	$\top$
?	$\bot$	?
?	?	?

A	B	$A \Rightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	?	$\top$
?	$\top$	$\top$
?	$\bot$	?
?	?	$\top$

A	$\neg A$
$\top$	$\bot$
$\bot$	$\top$
?	?

Définitions et notations

1. Une tri-valuation est une fonction f de $\mathcal{V}$ vers $\{\top, \bot, ?\}$.
2. On étend de manière usuelle une tri-valuation sur l'ensemble des formules en une fonction $\widehat{f}$ de $\mathcal{F}$ vers $\{\top, \bot, ?\}$.
3. Une tri-valuation $\widehat{f}$ satisfait une formule ϕ si $\widehat{f}(\phi) = \top$.
   On notera alors $\widehat{f} \vdash_3 \phi$.
4. Une formule ϕ est une conséquence d'un ensemble de formules $\mathcal{X}$ si toute tri-valuation qui satisfait toutes les formules de $\mathcal{X}$ satisfait ϕ.
   On notera dans ce cas $\mathcal{X} \Vdash_3 \phi$.
5. Une formule ϕ est une une tautologie si pour toute tri-valuation $\widehat{f}$, $\widehat{f}(\phi) = \top$.
   On notera dans ce cas $\Vdash_3 \phi$.

Question 1
Montrer que $A \vee \neg A$ n'est pas une tautologie.

Question 2
Proposer une tautologie simple dans cette logique.

On pose $\top = 1$, $\bot = 0$ et $? = 0,5$.

Question 3
Proposer un calcul arithmétique simple permettant de trouver la table de vérité de $A \wedge B$ en fonction de A et B. Même question pour $A \vee B$.

Question 4
En logique bi-valuée classique, les propositions $\neg A \vee B$ et $A \Rightarrow B$ sont équivalentes.
Qu'en est-il dans le cadre de la logique propositionnelle tri-valuée ?

Question 5
En écrivant les tables de vérité, indiquer si les propositions $\neg B \Rightarrow \neg A$ et $A \Rightarrow B$ sont équivalentes.

Question 6
Donner la table de vérité de la proposition $F = \big((A \Rightarrow B\ ) \wedge (\neg A \Rightarrow B)\big) \Rightarrow B$.
Cette proposition est-elle une tautologie ?

Un nouvel opérateur d'implication, noté $\rightarrow$, est défini par sa table de vérité.

A	B	$A \rightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$?$	$?$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	$?$	$\top$
$?$	$\top$	$\top$
$?$	$\bot$	$?$
$?$	$?$	$?$

Question 7
$A \rightarrow A$ est-il une tautologie ?

Question 8
Montrer qu'il n'existe aucune tautologie pour les formules construites avec les opérateurs $\neg$, $\vee$, $\wedge$ et $\rightarrow$.

Je ne reproduis pas la question suivante qui n'a pas de sens, le rapport parle de "*Question facile mais dont la formulation semble avoir dérouté la majorité des candidats*". De la pure langue de bois.

2 Mines 2014, partie II

Recommandations

- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.
- Tout résultat fourni dans l'énoncé peut être utilisé pour les questions ultérieures même s'il n'a pas été démontré.
- Il ne faut pas hésiter à formuler les commentaires qui semblent pertinents même lorsque l'énoncé ne le demande pas explicitement.
- Lorsque le candidat écrira une fonction ou une procédure, il pourra faire appel à une autre fonction ou procédure définie dans les questions précédentes ; il pourra aussi définir une procédure ou une fonction auxiliaire.
- Si les paramètres d'une fonction ou d'une procédure à Écrire sont supposés vérifier certaines hypothèses, il ne sera pas utile, dans l'écriture de cette fonction ou de cette procédure, de tester si les hypothèses sont bien vérifiées.
- On ne se préoccupera pas d'un éventuel dépassement du plus grand entier codable dans le langage de programmation.
- Dans les énoncés du problème, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (n) et du point de vue informatique pour celle caractères de machine à Écrire (`n`).

Rappel

Si n et p sont des entiers, l'instruction

```
Array.make_matrix n p false
```

permet de construire une matrice carrée booléenne `A` à `n` lignes et `p` colonnes, et dont les cases sont initialisées à `false`.

Notations et définitions

- On considère un ensemble fini Π de n propositions logiques distinctes : $\Pi = \{P_0, P_1, \ldots, P_{n-1}\}$.
- On suppose qu'un ensemble d'implications entre ces propositions, appelé *ensemble des implications initiales* et noté I, a déjà été établi.
- On peut en général déduire d'autres implications à partir de l'ensemble des implications initiales en utilisant la transitivité des implications.
- Pour deux propositions P et Q appartenant à Π, si Q se déduit de P à l'aide d'une suite d'implications appartenant à I, on dit que P **implique** Q, ce que l'on note $P \Rightarrow Q$.
- Dans toute la suite, on s'intéresse aux implications que l'on peut déduire de I.
- Pour tout P dans Π, on suppose que I contient l'implication $P \Rightarrow P$; une telle implication, nommée **boucle**, est notée $P \Rightarrow_0 P$.
- Si P et Q sont dans Π, la notation $P \Rightarrow_1 Q$ signifie que l'implication $P \Rightarrow Q$ appartient à I (pour tout P dans Π, on a donc aussi $P \Rightarrow_1 P$).
- Pour P et Q dans Π, P implique Q signifie qu'il existe un entier $k \geqslant 0$ et $k+1$ propositions $P_{i_0}, P_{i_1}, \ldots, P_{i_j}, P_{i_{(j+1)}}, \ldots, P_{i_k}$ appartenant à Π tels que l'on ait :
  - $P_{i_0} = P$,
  - $P_{i_k} = Q$,
  - pour j tel que $0 \leqslant j \leqslant k-1$, l'implication $P_{i_j} \Rightarrow P_{i_{(j+1)}}$ appartient à I.
- Avec les notations ci-dessus, on dit alors qu'il existe une **preuve de longueur k de l'implication** $P \Rightarrow Q$, ce que l'on note $P \Rightarrow_k Q$. Les implications de I sont donc les preuves de longueur 0 (si $P = Q$) ou 1.

Exemple

$I_1 = \{P_0 \Rightarrow_0 P_0, P_1 \Rightarrow_0 P_1, P_2 \Rightarrow_0 P_2, P_3 \Rightarrow_0 P_3, P_0 \Rightarrow_1 P_2, P_2 \Rightarrow_1 P_3, P_3 \Rightarrow_1 P_0, P_3 \Rightarrow_1 P_1\}$.
On a $P_0 \Rightarrow_2 P_3$ car on a $P_0 \Rightarrow_1 P_2$ et $P_2 \Rightarrow_1 P_3$; on a aussi $P_0 \Rightarrow_3 P_3$ car on peut ajouter une boucle en considérant les trois implications $P_0 \Rightarrow_0 P_0$, $P_0 \Rightarrow_1 P_2$ et $P_2 \Rightarrow_1 P_3$.
En revanche, on n'a pas : $P_0 \Rightarrow_2 P_1$.
Les implications qui peuvent être prouvées mais qui n'appartiennent pas à I sont :
$P_0 \Rightarrow P_1$, $P_0 \Rightarrow P_3$, $P_2 \Rightarrow P_0$, $P_2 \Rightarrow P_1$, $P_3 \Rightarrow P_2$.

2.1 Implications

On se donne l'ensemble d'implications initiales pour 4 propositions.
$I_2 = \{P_0 \Rightarrow_0 P_0, P_1 \Rightarrow_0 P_1, P_2 \Rightarrow_0 P_2, P_3 \Rightarrow_0 P_3, P_0 \Rightarrow_1 P_1, P_1 \Rightarrow_1 P_2, P_2 \Rightarrow_1 P_1, P_3 \Rightarrow_1 P_2\}$.

Question 9
Donner la liste des implications qui peuvent être prouvées mais qui n'appartiennent pas à I_2.

Question 10
Soient P et Q dans Π ; soient h et k deux entiers positifs ou nuls vérifiant : $h \leqslant k$.
Montrer que si on a $P \Rightarrow_h Q$, alors on a aussi $P \Rightarrow_k Q$.

Question 11
Soient P et Q dans Π.
Montrer qu'il existe une implication $P \Rightarrow Q$ si et seulement si on a $P \Rightarrow_{n-1} Q$.

2.2 Matrices booléennes

Une matrice booléenne est une matrice dont les coefficients prennent uniquement les valeurs **faux** et **vrai** (`false` et `true` en langage de programmation). Le produit de matrices booléennes s'obtient selon la formule habituelle en prenant comme somme de deux valeurs booléennes le **ou** logique (disjonction, notée $\vee$) et comme produit le **et** logique (la conjonction, notée $\wedge$) ; le produit de deux matrices A et B est noté $A \times B$.

Par exemple, si on considère les deux matrices :
$A_0 = \begin{pmatrix} \textbf{vrai} & \textbf{vrai} \\ \textbf{faux} & \textbf{vrai} \end{pmatrix}$ et $B_0 = \begin{pmatrix} \textbf{faux} & \textbf{vrai} \\ \textbf{vrai} & \textbf{faux} \end{pmatrix}$, le produit $A_0 \times B_0$ vaut $\begin{pmatrix} \textbf{vrai} & \textbf{vrai} \\ \textbf{vrai} & \textbf{faux} \end{pmatrix}$.
On ne s'intéressera dans ce problème qu'à des matrices carrées ; la dimension d'une matrice carrée est son nombre de lignes (et donc de colonnes). Si k est un entier strictement positif, on obtient la matrice A^k en multipliant $k-1$ fois la matrice A par elle-même.

Question 12
Écrire une fonction nommée `mult` *telle que, si* `a` *et* `b` *codent A et B, alors* `mult A B` *renvoie une matrice codant le produit $A \times B$.*

On considère un ensemble Π des n propositions logiques $P_0, P_1, \ldots, P_{n-1}$ et un ensemble I d'implications initiales entre ces propositions. On leur associe une matrice A carrée booléenne de dimension n définie de la façon suivante :

- les lignes et les colonnes de A sont indicées de 0 à $n-1$;
- soient i et j deux entiers vérifiant $0 \leqslant i, j \leqslant n-1$, en notant $A[i,j]$ le coefficient de A situé sur la ligne d'indice i et la colonne d'indice j, $A[i,j]$ vaut **vrai** si et seulement si l'implication $P_i \Rightarrow P_j$ appartient à I.

Ainsi, les matrices A_1 et A_2 correspondant respectivement aux ensembles d'implications initiales I_1 et I_2 sont :

$$A_1 = \begin{pmatrix} \textbf{vrai} & \textbf{faux} & \textbf{vrai} & \textbf{faux} \\ \textbf{faux} & \textbf{vrai} & \textbf{faux} & \textbf{faux} \\ \textbf{faux} & \textbf{faux} & \textbf{vrai} & \textbf{vrai} \\ \textbf{vrai} & \textbf{vrai} & \textbf{faux} & \textbf{vrai} \end{pmatrix} \quad A_2 = \begin{pmatrix} \textbf{vrai} & \textbf{vrai} & \textbf{faux} & \textbf{faux} \\ \textbf{faux} & \textbf{vrai} & \textbf{vrai} & \textbf{faux} \\ \textbf{faux} & \textbf{vrai} & \textbf{vrai} & \textbf{faux} \\ \textbf{faux} & \textbf{faux} & \textbf{vrai} & \textbf{vrai} \end{pmatrix}$$

Question 13
Montrer que, pour i et j vérifiant $0 \leqslant i, j \leqslant n-1$ et pour tout k strictement positif, le coefficient $A^k[i,j]$ vaut **vrai** *si et seulement si on a $P_i \Rightarrow_k P_j$.*

Question 14
Montrer que, pour tout $k \geqslant n-1$, on a $A^k = A^{n-1}$.

Fermeture transitive

On appelle **fermeture transitive** de A et on note $FT(A)$ la matrice A^{n-1}.

Question 15
Écrire une fonction `fermeture` *telle que, si* `a` *code une matrice booléenne A, alors* `fermeture a` *renvoie la matrice $FT(A)$. On pourra supposer qu'on a $n \geqslant 2$.*

Question 16
Écrire une fonction `deduction` *telle que si :*

- `a` *code A, matrice booléenne associée à un ensemble des implications initiales,*
- `i` *est un entier compris entre 0 et $n-1$,*

alors `deduction a i` *renvoie un tableau de booléens de longueur n tel que, pour j compris entre 0 et $n-1$, la valeur d'indice* `j` *de ce tableau vaut* `true` *si et seulement si on a $P_i \Rightarrow P_j$.*
ATTENTION *: On exige que la complexité de cette fonction soit un $\mathcal{O}(n^2)$. On utilisera pour cela la récursivité. On ne justifiera pas la complexité de la fonction qui sera écrite.*

Question 17
En utilisant la fonction `deduction`*, écrire une fonction* `fermeture_bis` *telle que si* `a` *code une matrice booléenne A, alors* `fermeture_bis A` *renvoie la matrice $FT(A)$.*

2.3 Propositions équivalentes

Dans toute la suite la fermeture transitive $B = FT(A)$ de la matrice A représentant un ensemble d'implications initiales I sera appelée **matrice des implications**.
Soient P et Q deux propositions appartenant à Π.
On dit que les propositions P et Q sont **équivalentes** si on a : $P \Rightarrow Q$ et $Q \Rightarrow P$.

Axiomes

Soit P appartenant à Π. On dit ici que P est un axiome si on a la propriété suivante : quelle que soit la proposition Q appartenant à Π, si on a $Q \Rightarrow P$, alors on a aussi $P \Rightarrow Q$ et donc P et Q sont équivalentes ; autrement dit, P est équivalente à toute proposition qui l'implique.

Question 18
Donner tous les axiomes dans les exemples donnés par I_1 et I_2 ci-dessus.

Question 19
Écrire une fonction `est_axiome` *telle que si* `b` *code une matrice des implications et si* `i` *est un entier compris entre 0 et $n-1$, alors* `est_axiome b i` *renvoie la valeur* `true` *si P_i est un axiome et la valeur* `false` *sinon.*

On appelle suite unidirectionnelle une suite $(Q_0, Q_1, \ldots, Q_h)$ de propositions de Π telle que :
1. pour i vérifiant $0 \leqslant i \leqslant h-1$, $Q_i \Rightarrow Q_{i+1}$,
2. pour i vérifiant $0 \leqslant i \leqslant h-1$, Q_{i+1} n'implique pas Q_i.

Question 20
Montrer que les propositions d'une suite unidirectionnelle sont deux à deux distinctes.

Question 21
Soit Q une proposition appartenant à Π. Montrer qu'il existe un axiome P avec $P \Rightarrow Q$.

Classes d'équivalence

La relation est une relation d'équivalence. On peut donc partitionner Π en sous-ensembles de sorte que deux propositions de Π soient équivalentes si et seulement si elles appartiennent au même sous-ensemble. On appelle **classes d'équivalence** ces sous-ensembles.
Pour I_1, il y a deux classes d'équivalence : $\{P_0, P_2, P_3\}$ et $\{P_1\}$.

Question 22
Donner les classes d'équivalence pour I_2.

Question 23
On considère une classe d'équivalence C contenant un axiome.
Montrer que toutes les propositions contenues dans C sont des axiomes.

On dit qu'une classe d'équivalence est une **classe source** si elle contient un axiome.
Une partie X de Π est une **axiomatique** si, quelle que soit la proposition $Q \in \Pi$, il existe une proposition P appartenant à X vérifiant $P \Rightarrow Q$.

Question 24
Montrer qu'on obtient une axiomatique de cardinal minimum en choisissant une et une seule proposition dans chacune des classes sources.

Question 25
Écrire une fonction `axiomatique` *telle que, si* `B` *code une matrice des implications, alors* `axiomatique B` *renvoie une liste d'entiers contenant les indices de propositions de Π formant une axiomatique de cardinalminimum.*

On pourra utiliser un tableau pour éliminer, lorsqu'on a choisi un axiome, les axiomes qui lui sont équivalents.

3 Solutions

Solution de l'exercice 1 - Si une valuation f vérifie $f(A) =?$ alors $\widehat{f}(A \vee \neg A) = . \vee \neg ? =?$: $A \vee \neg A$ n'est pas satisfaite par f, ce n'est pas une tautologie.

Solution de l'exercice 2 - On vérifie que $A \Rightarrow A$ est une tautologie.

Solution de l'exercice 3 - On peut généraliser les formule de la logique classique :
$\widehat{f}(A \wedge B) = \min\big(f(A), f(B)\big)$ et $\widehat{f}(A \vee B) = \max\big()f(A), f(B)\big)$.

Solution de l'exercice 4 - Pour une valuation f telle que $f(A) = f(B) =?$, on a
$\widehat{f}(\neg A \vee B) =?$ et $\widehat{f}(AQ \Rightarrow B) = \top$.

Solution de l'exercice 5 -

A	B	$A \Rightarrow B$	$\neg A$	$\neg B$	$\neg B \Rightarrow \neg A$
$\top$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$
$\top$	?	?	$\bot$	?	?
$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\bot$	?	$\top$	$\top$	?	$\top$
?	$\top$	$\top$	?	$\bot$	$\top$
?	$\bot$	?	?	$\top$	?
?	?	$\top$	?	?	$\top$

On a bien l'égalité des évaluations.

Solution de l'exercice 6 -

A	B	$A \Rightarrow B$	$\neg A \Rightarrow B$	$(A \Rightarrow B\ ) \wedge (\neg A \Rightarrow B)$	F
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\top$	?	?	$\top$	?	$\top$
$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\bot$	?	$\top$	?	?	$\top$
?	$\top$	$\top$	$\top$	$\top$	$\top$
?	$\bot$	?	?	?	?
?	?	$\top$	$\top$	$\top$	?

En raison des deux dernières lignes, ce n'est pas une tautologie.

Solution de l'exercice 7 - Si $f(A) =?$ alors $\widehat{f}(A \to A) =?$, $A \to A$ n'est pas une tautologie.

Solution de l'exercice 8 -
On voit que, si $\widehat{f}(\phi) = \widehat{f}(\psi) =?$ alors $\widehat{f}(\neg\phi) = \widehat{f}(\phi \wedge \psi) = \widehat{f}(\phi \vee \psi) = \widehat{f}(\phi \to \psi) =?$
On choisit la valuation telle que $f(P) =?$ pour toute variable alors, par induction structurelle, $\widehat{(}\phi) =?$ pour toute formule. Ainsi aucune formule n'est une tautologie.

Solution de l'exercice 9 - $P_0 \Rightarrow P_2$ et $P_3 \Rightarrow P_1$ sont les implications prouvables qui n'appartiennent pas à I_2.

Solution de l'exercice 10 - Si on a une preuve de longueur h de $P \Rightarrow Q$,
$P \Rightarrow P_{i_1} \Rightarrow P_{i_2} \Rightarrow \cdots \Rightarrow P_{i_{h-1}} \Rightarrow Q$,
on peut la prolonger en une preuve de longueur $h+1$,
$P \Rightarrow P_{i_1} \Rightarrow P_{i_2} \Rightarrow \cdots \Rightarrow P_{i_{h-1}} \Rightarrow Q \Rightarrow Q$.
Ainsi $P \Rightarrow_h Q$ implique $P \Rightarrow_{h+1} Q$ donc, par transitivité, $P \Rightarrow_h Q$ implique $P \Rightarrow_k Q$ pour $k \geqslant h$.

Solution de l'exercice 11 -
Sens réciproque Si on a $P \Rightarrow_{n-1} Q$ alors il existe une preuve de l'implication $P \Rightarrow Q$.
Sens direct On suppose qu'il existe une preuve de l'implication $P \Rightarrow Q$
On considère la preuve de longueur minimale k de cette implication :
$P = P_{i_0} \Rightarrow P_{i_1} \Rightarrow P_{i_2} \Rightarrow \cdots \Rightarrow P_{i_{k-1}} \Rightarrow P_{i_k} = Q$.
Les $k+1$ propositions P_{i_r} sont des éléments de Π de cardinal n.
Si on avait $k \geqslant n$ alors deux de ces propositions devraient être égales, $P_{i_r} = P_{i_s}$ avec $r < s$. On pourrait alors supprimer les implications de I entre r et s pour obtenir une preuve de longueur $r + k - s < k$ $P = P_{i_0} \Rightarrow P_{i_1} \Rightarrow \cdots \Rightarrow P_{i_r} = P_{i_s} \Rightarrow cdots \Rightarrow P_{i_k} = Q$.
Ceci contredit la minimalité de k donc on doit avoir $k \leqslant n-1$.
D'après l'exercice précédent on déduit de $P \Rightarrow_k Q$ qu'on a $P \Rightarrow_{n-1} Q$.

Solution de l'exercice 12 -

```
let mult a b =
   let n =Array.length a in
   let c = Array.make_matrix n n false in
   for i = 0 to n-1 do
      for j = 0 to n-1 do
         for k = 0 to n-1 do
            c.(i).(j) <- c.(i).(j) || (a.(i).(k) && b.(k).(j))
         done;
      done;
   done;
   c;;
```

Solution de l'exercice 13 - On montre le résultat par récurrence sur k :
$\mathcal{P}(k)$: le coefficient $A^k[i,j]$ vaut **vrai** si et seulement si on a $P_i \Rightarrow_k P_j$.
$\mathcal{P}(1)$ est vraie par définition de la matrice A.
On suppose que $\mathcal{P}(k)$ est vérifiée.
On a $A^{k+1}[i,j] = \bigvee_{r=0}^{n-1} A^k[i,r] \wedge A[r,j]$

- Si $A^{k+1}[i,j]$ vaut **vrai** alors il existe (au moins) un entier r tel que $A^k[i,r] \wedge A[r,j]$ vaut **vrai** d'où $A^k[i,r]$ et $A[r,j]$ valent **vrai**.
  En utilisant $\mathcal{P}(k)$ et la définition de A on en déduit qu'on a $P_i \Rightarrow_k P_r$ et $P_r \Rightarrow P_j \in I$.
  En concaténant on en déduit une preuve de longueur $k+1$, $P_i \Rightarrow_{k+1} P_j$.
- Si on a $P_i \Rightarrow_{k+1} P_j$, on en écrit une preuve :
  $P_i = P_{i_0} \Rightarrow P_{i_1} \Rightarrow P_{i_2} \Rightarrow \cdots \Rightarrow P_{i_k} \Rightarrow P_{i_{k+1}} = P_j$.
  Le début fournit une preuve de longueur k : $P_i \Rightarrow_k P_{i_k}$ donc $A^k[i,i_k]$ vaut **vrai** d'après l'hypothèse de récurrence. De plus $P_{i_k} \Rightarrow P_j$ appartient à I donc $A[i_k,j]$ vaut **vrai**.
  On en déduit que $A^k[i,i_k] \wedge A[i_k,j]$ vaut **vrai** puis $A^{k+1}[i,j] = \bigvee_{r=0}^{n-1} A^k[i,r] \wedge A[r,j]$ aussi.

On a donc prouvé l'équivalence souhaitée au rang $k+1$: $\mathcal{P}(k+1)$ est vérifiée.

Solution de l'exercice 14 - On utilise l'équivalence ci-dessus.
D'après la question 3), $A^k[i,j]$ vaut **vrai** implique $A^{n-1}[i,j]$ vaut **vrai** pour tout k.
D'après la question 2), $A^{n-1}[i,j]$ vaut **vrai** implique $A^k[i,j]$ vaut **vrai** pour tout $k \geqslant n-1$.
Ainsi, pour $k \geqslant n-1$, $A^k[i,j]$ vaut **vrai** si et seulement si $A^{n-1}[i,j]$ vaut **vrai**
donc $A^k[i,j] = A^{n-1}[i,j]$ pour tous i et j : $A^k = A^{n-1}$.

Solution de l'exercice 15 -

```
let fermeture a =
   let n = Array.length a in
   let res = ref a in
   for i = 1 to (n-2) do
```

```
      res := mult !res a done;
   !res;;
```

Solution de l'exercice 16 - Pour chaque proposition P qu'on a pu déduire, on peut déduire aussi les conséquence de P dans I. Le tableau de booléens sert aussi à tester si une proposition a déjà été vue. On utilise une fonction auxiliaire.

```
let deduction a i =
   let n = Array.length a in
   let deduit = Array.make n false in
      let rec voir k =
         if not deduit.(k)
         then begin
            deduit.(k) <- true;
            for j = 0 to n-1 do
               if a.(k).(j) then voir j done
         end in
    voir i;
  deduit;;
```

Solution de l'exercice 17 -

```
let fermeture_bis a =
   let n = Array.length a in
   let res = Array.make  n [||] in
   for i = 0 to n-1 do
      res.(i) <- deduction a i done;
   res;;
```

Solution de l'exercice 18 - Les axiomes de I_1 sont P_0, P_2 et ceux de I_2 sont P_0 et P_3.

Solution de l'exercice 19 -

```
let est_axiome b i =
   let res = ref true in
   let n = Array.length b in
      for j = 0 to n-1 do
         if b.(j).(i) then res := !res && b.(i).(j) done;
    !res;;
```

Solution de l'exercice 20 -
On considère une suite unidirectionnelle de propositions $(Q_0, Q_1, ..., Q_h)$.
S'il existait $i < j$ tels que $Q_i = Q_j$ alors la suite d'implications $Q_{i+1} \Rightarrow \cdots \Rightarrow Q_j = Q_i$ donnerait l'implication $Q_{i+1} \Rightarrow Q_i$, ce qui est exclu.
Ainsi les propositions d'une suite unidirectionnelle sont distinctes.

Solution de l'exercice 21 - Soit $Q \in \Pi$.
Les suites unidirectionnelles qui aboutissent à Q ont des éléments distincts donc elle doivent être de longueur $n-1$ au plus. Comme il existe au moins la suite unidirectionnelle de longueur 0, (Q), il existe une suite unidirectionnelle qui aboutit à Q de longueur maximale : $(Q_0, ..., Q_p)$ avec $Q_p = Q$.
Si Q_0 n'était pas un axiome, il existerait une proposition p telle que $P \Rightarrow Q_0$ et Q_0 n'implique pas P. On pourrait alors définir une suite unidirectionnelle plus longue : $(P, Q_0, ..., Q_p)$ ce qui est impossible.
Ainsi Q_0 est un axiome et Q est impliquée par un axiome..

Solution de l'exercice 22 -
Les classes d'équivalence pour I_2 sont $\{P_0\}$, $\{P_1, P_2\}$ et $\{P_3\}$.

Solution de l'exercice 23 - P est un axiome et C est sa classe d'équivalence.
Pour toute proposition $Q \in C$, on considère Q' tellle que $Q' \Rightarrow Q$.
Comme on a $Q \Rightarrow P$, on a aussi $Q' \Rightarrow P$ puis, comme P est un axiome, $P \Rightarrow Q'$.
En utilisant $Q \Rightarrow P$, on en déduit $Q \Rightarrow Q'$. Ainsi Q est un axiome.

Solution de l'exercice 24 - On décompose en plusieurs étapes. X est une axiomatique

1. Si P est un axiome, il existe $Q \in P$ tel que $Q \Rightarrow P$ donc, d'après la définition Q est équivalent à P. Ainsi X doit contenir un élément de la classe de P.
   Toute axiomatique contient au moins un élément de chaque classe source.
2. Si X contient deux propositions d'une même classe source, P_1 et P_2, alors $X' = X \setminus \{P_2\}$ est encore une axiomatique. En effet, toute proposition déduite de P_2 est aussi déduite de P_1. **Toute axiomatique minimale contient au plus un élément de chaque classe source.**
3. Si X contient une proposition P qui n'est pas un axiome alors P est déduite d'un axiome Q donc, comme X contient un élément de la classe de Q, P est déduit d'un élément de X. On peut alors retirer $P : X \setminus \{P\}$ est encore une axiomatique. **Toute axiomatique minimale ne contient que des axiomes.**

Ainsi tout axiomatique minimale est formée d'axiomes avec un axiome unique par classe source.
Inversement en choisissant un élément unique dans chaque classe source on obtient une axiomatique car tout élément est déduit d'un axiome que l'ont peut choisir indistinctement dans sa classe et le premier point montre que cette axiomatique est minimale.

Solution de l'exercice 25 -
On parcourt les proposition. Pour chaque proposition non encore déduite qui est un axiome, on l'ajoute aux axiomes et et marque comme déduites toutes ses conséquences.

```
let axiomatique b =
   let n = Array.length b in
   let ax = Array.make n false in
   let deduit = Array.make n false in
      for i = 0 to n-1 do
         if (est_axiome b i) && not(deduit.(i))
         then begin
            ax.(i) <- true;
            for j = 0 to (n-1) do
               if b.(i).(j) then deduit.(j) <- true done
         end done;
    ax;;
```

Devoir D

DS2+ : X-ENS 2013

On étudie dans ce problème un formalisme logique, la logique temporelle, permettant de définir des formules auxquelles sont associés des langages de mots. Les mots décrivent l'évolution dans le temps d'un phénomène. Ainsi, pour toute formule φ de la logique temporelle et pour tout mot u on définira la propriété que le mot u satisfait la formule φ, et à toute formule φ on associera l'ensemble L_φ des mots qui satisfont φ.
L'objet principal de ce problème est de s'intéresser aux propriétés de ces langages L_φ.
La partie I introduit la logique temporelle et donne des exemples de formules.
La partie II introduit une forme normale pour les formules.
La partie III est consacrée à montrer que pour toute formule l'ensemble de mots associés est un langage rationnel.
Enfin, la partie IV étudie d'une part le problème de la satisfiabilité d'une formule (étant donnée une formule φ, existe-t-il un mot satisfaisant φ ?) et d'autre part l'expressivité des formules.
Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes.

1 Préliminaires

- Un **alphabet** est un ensemble fini $\mathcal{A}$ dont les éléments sont appelés **lettres**.
- Un sur un alphabet $\mathcal{A}$ est une suite finie d'éléments de $\mathcal{A}$.
- On notera ε le mot vide (c'est-à-dire la suite vide).
- On définira la longueur $|u|$ d'un mot non vide $u = a_0 a_1 \cdots a_{n-1}$ comme valant n. La longueur de ε vaut 0.
- Si $\mathcal{A}$ est un alphabet, on notera $\mathcal{A}^*$ l'ensemble des mots sur $\mathcal{A}$ et $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$ l'ensemble des mots non vides sur $\mathcal{A}$.
- Dans la suite, les lettres d'un mot de longueur p sont indicées de 0 à $p-1$.
- En OCaml, nous représenterons les mots à l'aide du type `string`.
- Si mot est de type `string` et i est de type `int` alors `mot.[i]` est de type `char` et donne la lettre d'indice i de mot .
  Par exemple `"bonjour".[3]` renvoie `'j'`.
- La fonction `String.length` renvoie la longueur d'un mot.

Dans tout le problème on se fixe un alphabet fini $\mathcal{A}$ (on pourra imaginer qu'il s'agit des lettres minuscules de l'alphabet usuel).
On souhaite définir des ensembles de mots sur l'alphabet $\mathcal{A}$ à l'aide de formules logiques. Pour cela, on définit, pour chaque lettre $a \in \mathcal{A}$, un prédicat p_a, qui permettra de tester si la lettre à une position qui sera donnée est un a. Pour construire des formules à partir de ces prédicats, on utilise les connecteurs booléens $\vee$ (ou), $\wedge$ (et) et $\neg$ (non) ainsi que les connecteurs temporels **X** (juste après), **G** (désormais), **F** (un jour) et **U** (jusqu'à).
Les formules de la logique temporelle sont alors construites par induction comme suit. Si p_a est un prédicat, et si φ et ψ sont des formules de la logique temporelle, toutes les formules seront

construites selon la syntaxe suivante :

1. **vrai**
2. p_a
3. $(\neg\varphi)$
4. $(\varphi \vee \psi)$
5. $(\varphi \wedge \psi)$
6. $(\mathbf{X}\varphi)$
7. $(\mathbf{G}\varphi)$
8. $(\mathbf{F}\varphi)$
9. $(\varphi\mathbf{U}\psi)$

Toutes les formules seront construites de la façon précédente, en omettant les parenthèses si celles-ci sont inutiles. Par exemple, $\mathbf{X}p_b$, $p_a\mathbf{U}p_b$ et $\mathbf{F}(\mathbf{G}p_a)$ sont des formules.

Nous allons maintenant définir le sens (ou la sémantique) des formules.
Soit un mot u et soit une formule de la logique temporelle φ. On définit, pour tout $i \geqslant 0$, la propriété "le mot u satisfait la formule φ à la position i", ce qui sera noté $(u, i) \vDash \varphi$, comme suit.

- Si $i \geqslant |u|$, on n'a pas $(u, i) \vDash \varphi$: une formule ne peut être vraie qu'à une position du mot ; en particulier le mot vide ne satisfait aucune formule (pas même la formule **vrai**).
- Si $i \leqslant |u| - 1$, on note $u = a_0a_1 \cdots a_{|u|-1}$ et on raisonne par induction sur la structure de φ.
  1. $(u, i) \vDash$ **vrai**.
  2. $(u, i) \vDash p_a$ si et seulement si $a_i = a$.
  3. $(u, i) \vDash (\neg\varphi)$ si et seulement si on n'a pas $(u, i) \vDash \varphi$.
  4. $(u, i) \vDash (\varphi \vee \psi)$ si et seulement si $(u, i) \vDash \varphi$ ou $(u, i) \vDash \psi$.
  5. $(u, i) \vDash (\varphi \wedge \psi)$ si et seulement si $(u, i) \vDash \varphi$ et $(u, i) \vDash \psi$.
  6. $(u, i) \vDash (\mathbf{X}\varphi)$ si et seulement si $(u, i + 1) \vDash \varphi$.
     Notez que si $i = |u| - 1$, alors on ne peut avoir $(u, i) \vDash (\mathbf{X}\varphi)$.
  7. $(u, i) \vDash (\mathbf{G}\varphi)$ si et seulement si $(u, j) \vDash \varphi$ pour tout j tel que $i \leqslant j < |u|$.
  8. $(u, i) \vDash (\mathbf{F}\varphi)$ si et seulement si il existe j tel que $i \leqslant j < |u|$ et $(u, j) \vDash \varphi$.
  9. $(u, i) \vDash (\varphi\mathbf{U}\psi)$ si et seulement si il existe j tel que $i \leqslant j < |u|$, $(u, j) \vDash \psi$ et $(u, k) \vDash \varphi$ pour tout $k \in \{i, i + 1, \ldots, j - 1\}$.

On notera $(u, i) \vDash \varphi$ si et seulement si l'on n'a pas $(u, i) \vDash \varphi$.
On notera $u \vDash \varphi$ (et on dira alors que φ est vraie pour u) le fait que $(u, 0) \vDash \varphi$. On notera $u \nvDash \varphi$ le fait que $(u, 0) \nvDash \varphi$.

Exemples

- $(aaabcbab, 2) \vDash (\mathbf{X}p_b)$ car la lettre d'indice 3 de *aaabcbab* est un b.
- $aaabcbab \vDash (p_a\mathbf{U}p_b)$ car la lettre d'indice 3 est un b et n'est précédée que de a.
- $aaabcbab \vDash \mathbf{F}(\mathbf{G}p_a)$ car il n'existe pas d'indice à partir duquel il n'y ait plus que des a (en effet, la dernière lettre est un b).

Question 1

On pose $u = bbbcbbaa$. Pour chacun des énoncés ci-dessous dire s'il est vrai en justifiant brièvement votre réponse.

1. $(u, 4) \vDash \mathbf{G}(p_a \vee p_b)$
2. $(u, 2) \vDash \mathbf{X}(\mathbf{G}(p_a \vee p_c))$
3. $(u, 1) \vDash \mathbf{F}(\mathbf{G}(p_a \vee p_b))$
4. $u \vDash (p_a \vee p_b)\mathbf{U}(p_a \vee p_c)$

Question 2
Écrire une formule φ telle que $u \vDash \varphi$ si et seulement si u contient un a suivi plus tard d'un b. Par exemple on aura $ccacccba \vDash \varphi$ tandis que $ccacccaa \nvDash \varphi$.

Question 3
Écrire une formule **fin** *telle que $(u, i) \vDash$* **fin** *si et seulement si $i = |u|-1$ (c'est-à-dire si et seulement si i est l'indice de la dernière lettre de u).*

Dans la suite, on pourra utiliser **fin** comme une boîte noire.

Question 4
Écrire une formule φ telle que $u \vDash \varphi$ si et seulement si u se termine par un a.

Question 5
Écrire une formule φ telle que $u \vDash \varphi$ si et seulement si $u = ababab \cdots ab$, c'est-à-dire si et seulement s'il existe $k \geqslant 1$ tel que $u = u_0u_1 \cdots u_{2k-1}$ et, pour tout i tel que $0 \leqslant i < 2k$ on a $u_i = a$ si i est pair et $u_i = b$ si i est impair.

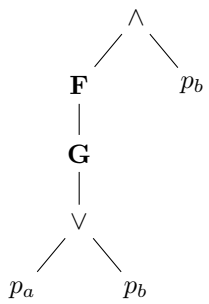
Question 6 — Pour les cubes seulement
Pour cette question, on pose $\mathcal{A} = \{a, b, c\}$. Soit $\varphi = \mathbf{F}\big(p_a \wedge \mathbf{X}\big(\mathbf{G}(\neg p_a)\big) \wedge \mathbf{F}(p_b \wedge \mathbf{X}p_c)\big)$. Décrire un automate fini (pouvant être non-déterministe) reconnaissant le langage $L_\varphi = \big\{u \in \mathcal{A}^+ \;\big|\; u \vDash \varphi\big\}$.

Deux formules φ et ψ sont **équivalentes**, ce que l'on note $\varphi \equiv \psi$, si pour tout mot u, on a $u \vDash \varphi$ si et seulement si $u \vDash \psi$. Autrement dit, les formules φ et ψ sont vraies pour les mêmes mots.

Question 7
Soient φ et ψ deux formules quelconques. Montrer que l'on a $\varphi \mathbf{U} \psi \equiv \psi \vee \big(\varphi \wedge \big(\mathbf{X}(\varphi \mathbf{U} \psi)\big)\big)$.

2 Normalisation de formules

Afin de manipuler des formules de la logique temporelle, on va représenter ces dernières par des arbres. Outre les prédicats $(p_a)_{a\in A}$ et la formule **vrai** nous avons des connecteurs unaires ($\neg, \mathbf{X}, \mathbf{G}$ et $\mathbf{F}$) et des connecteurs binaires ($\vee, \wedge$ et $\mathbf{U}$). À la manière des expressions arithmétiques, on représente une formule de la logique temporelle par un arbre dont les feuilles sont étiquetées soit par un prédicat p_a soit par **vrai**, et dont les nœuds internes sont étiquetés par un connecteur unaire (un tel nœud aura alors un seul fils) ou par un connecteur binaire (un tel nœud aura alors deux fils). Dans la suite, on confondra fréquemment une formule avec sa représentation par un arbre.
Par exemple, la formule $\big(\mathbf{F}(\mathbf{G}(p_a \vee p_b))\big) \wedge p_b$ sera représentée par l'arbre ci-dessous :



On définit le type suivant pour les formules

```
type formule =  VRAI
              |Predicat of char
              |NON of formule
              |ET of formule * formule
              |OU of formule * formule
              |X of formule
              |G of formule
              |F of formule
              |U of formule * formule ;;
```

On définit la taille d'une formule par le nombre de nœuds (y compris les feuilles) de l'arbre qui la représente.
En particulier, la formule **vrai** et les formules réduites à un prédicat p_a sont de taille 1.

Question 8
Écrire une fonction `taille` *qui prend en argument une formule et renvoie sa taille.*

```
taille: formule -> int
```

Question 9
Donner, pour toute formule φ n'utilisant pas le connecteur **F**, *une formule équivalente à* $(\mathbf{F}\varphi)$ *qui n'utilise pas le connecteur temporel* **F**.
En déduire une fonction `normaliseF` *qui prend en entrée une formule quelconque et renvoie une formule équivalente qui n'utilise pas le connecteur* **F**.
Quelle est la complexité de votre algorithme ?

```
normaliseF: formule -> formule
```

Une formule qui utilise comme seuls connecteurs temporels le **X** et le **U** sera qualifiée de **normalisée**.

Question 10
Montrer que toute formule est équivalente à une formule normalisée. Donner une fonction `normalise` *de complexité linéaire qui prend en entrée une formule quelconque et renvoie une formule équivalente normalisée.*

```
normalise: formule -> formule
```

Dans toute la suite du problème, on supposera que l'on travaille avec des **formules normalisées**.

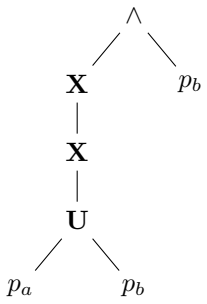
Question 11
Écrire une fonction récursive `veriteN` *qui prend en argument un mot u, un indice i et une formule (normalisée) φ et détermine si $(u, i) \vDash \varphi$. Justifier que votre fonction termine et indiquer si elle est exponentielle ou polynomiale en la taille de ses arguments.*

```
veriteN: string -> int -> formule -> bool
```

3 Rationalité des langages décrits par des formules

Le but de cette partie est de montrer qu'étant donnée une formule φ, l'ensemble des mots pour lesquels φ est vraie est un langage rationnel.

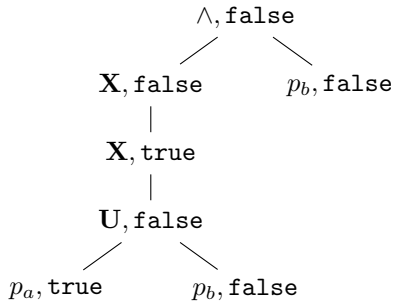
Soit φ une formule représentée par un arbre t_φ. Une **sous-formule** de φ est une formule représentée par un sous-arbre de t_φ. Par exemple si l'on considère la formule $\varphi = (\mathbf{X}(\mathbf{X}(p_a \mathbf{U} p_b))) \wedge p_b$, représentée par l'arbre ci-dessous,



l'ensemble de ses sous-formules est :

$$\big\{(\mathbf{X}(\mathbf{X}(p_a \mathbf{U} p_b))) \wedge p_b,\ \mathbf{X}(\mathbf{X}(p_a \mathbf{U} p_b)),\ \mathbf{X}(p_a \mathbf{U} p_b),\ p_a \mathbf{U} p_b,\ p_b,\ p_a\big\}$$

On souhaite représenter des ensembles de sous-formules d'une formule donnée. Pour cela, on va utiliser des arbres dont les nœuds sont étiquetés (en plus des symboles de la logique) par des booléens. Par exemple pour la formule $(\mathbf{X}(\mathbf{X}(p_a \mathbf{U} p_b))) \wedge p_b$ représentée ci-dessus, l'ensemble de sous-formules $\{p_a,\ \mathbf{X}(p_a \mathbf{U} p_b)\}$ sera représenté par l'arbre ci-dessous.



Une sous-formule ψ appartient à l'ensemble représenté par un arbre de type `ensemble` s'il existe un nœud de l'arbre étiqueté par `true` et dont le sous-arbre «correspond» à la formule ψ (c'est-à-dire qu'après suppression des booléens, le sous-arbre est égal à l'arbre codant ψ). Le type `ensemble` est défini comme suit.

```
type ensemble = eformule * bool
 and eformule =  AVRAI
               |APredicat of char
               |ANON of ensemble
               |AET of ensemble * ensemble
               |AOU of ensemble * ensemble
               |AX of ensemble
               |AU of ensemble * ensemble ;;
```

On rappelle que l'on ne travaille plus qu'avec des formules normalisées.

Si a est une lettre et si u est un mot, on note $a \cdot u$ le mot obtenu en ajoutant a en tête de u. Si $u = a_0 a_1 \cdots a_{p-1}$, alors $v = a a_0 a_1 \cdots a_{p-1}$.

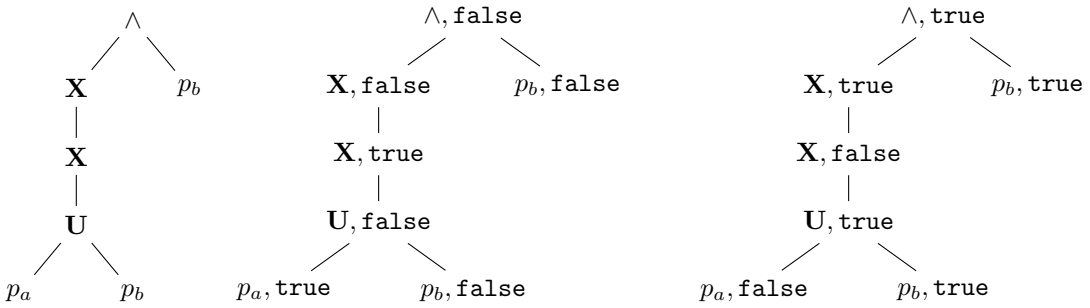
Question 12
Écrire une fonction `initialise` *qui prend en argument une formule φ et renvoie un ensemble représentant l'ensemble vide de sous-formules de φ.*

```
initialise: formule -> ensemble
```

Question 13
Soient u un mot, φ une formule et $\mathcal{S}$ l'ensemble des sous-formules de φ vraies pour u. Soit a une lettre. Montrer que l'ensemble $\mathcal{S}'$ des sous-formules de φ vraies pour $a \cdot u$ peut être déterminé uniquement en fonction de a et de $\mathcal{S}$ (en particulier indépendamment de u).

Question 14
En vous basant sur la question 13, écrire une fonction `maj` *qui prend en argument un* `ensemble` *$\mathcal{S}$ (représentant un ensemble de sous-formules d'une formule φ) et une lettre a et renvoie un* `ensemble` *$\mathcal{S}'$ tel que : pour tout mot u si $\mathcal{S}$ représente l'ensemble $\{\psi \mid u \vDash \psi$ et ψ sous-formule de $\varphi\}$ des sous-formules de φ vraies pour u alors $\mathcal{S}'$ représente l'ensemble $\{\psi \mid a \cdot u \vDash \psi$ et ψ sous-formule de $\varphi\}$ des sous-formules de φ vraies pour $a \cdot u$. Justifier la terminaison et préciser la complexité (dans la taille de la formule) de votre fonction.*

Par exemple si l'on reprend l'exemple de la formule $\varphi = (\mathbf{X}(\mathbf{X}(p_a \mathbf{U} p_b))) \wedge p_b$ (arbre à gauche ci-dessous) et de l'`ensemble` $\mathcal{S}$ (au centre ci-dessous), alors `maj S 'b'` devra renvoyer l'ensemble $\mathcal{S}'$ (à droite ci-dessous).




```
maj: ensemble -> char -> ensemble
```

Question 15
Écrire une fonction `sousFormulesVraies` *qui prend en argument une formule φ et un mot et renvoie un* `ensemble` *décrivant les sous-formules ψ de φ telles que $u \vDash \psi$. Votre fonction devra avoir une complexité polynomiale dans la taille de la formule et dans la taille du mot. Justifier la terminaison et préciser la complexité (dans la taille de la formule et la taille du mot).*

```
sousFormulesVraies: formule -> string -> ensemble
```

Question 16
En déduire une fonction `veriteBis` *qui teste si une formule donnée est vraie à la première position d'un mot donné.*

```
veriteBis: formule -> string -> bool
```

Soit φ une formule. On associe à φ un langage de mots $L_\varphi \subseteq A^+$ en posant

$$L_\varphi = \{u \in A^+ \mid u \vDash \varphi\}$$

Soit un mot $u = a_0 \cdots a_{\ell-1}$. On note $\tilde{u}$ le mot miroir de u : $\tilde{u} = a_{\ell-1} a_{\ell-2} \cdots a_0$. Soit un langage $L \subseteq A^+$, on notera $\widetilde{L} = \{\tilde{u} \mid u \in L\}$.

À partir de ce moment, les questions ne sont pas à traiter. Elles correspondent à une partie du cours non encore étudiée (comme la question 6). Bien entendu les cubes véloces peuvent les traiter.

Question 17
En vous inspirant de la fonction `maj` *de la* **question 14**, *montrer que pour toute formule φ, le langage $\widetilde{L}_\varphi$ est reconnu par un automate déterministe complet. Donner un majorant du nombre d'états d'un tel automate.*

Question 18
En déduire que pour toute formule φ, le langage L_φ est reconnu par un automate fini. Donner un majorant du nombre d'états d'un tel automate (automate qui pourra être pris non-déterministe).

4 Satisfiabilité et expressivité

On rappelle que désormais les formules considérées sont normalisées (elles n'utilisent donc ni le **F** ni le **G**). Dans toute cette partie, on considérera que $A = \{a, b\}$ sauf mention explicite.

Le but de cette partie est dans un premier temps d'écrire un programme qui prend en entrée une formule φ et renvoie `true` si et seulement s'il existe $u \in A^+$ tel que $u \vDash \varphi$. Dans un second temps, on montre qu'il existe un langage accepté par un automate fini qui ne peut être décrit par aucune formule de la logique temporelle.

Question 19
Soit $\mathcal{A}$ un automate fini déterministe sur l'alphabet A. Décrire informellement un algorithme qui calcule la liste des états atteignables depuis l'état initial (c'est-à-dire l'ensemble des états q tels qu'il existe un mot qui lu depuis l'état initial termine dans q).

Question 20
Soit φ une formule satisfiable, c'est-à-dire pour laquelle existe un mot $u \in A^+$ tel que $u \vDash \varphi$. Soit u_{min} un plus court u tel que $u \vDash \varphi$. Majorer la longueur de u_{min} en fonction de la taille de φ.

Pour la question suivante, on pourra utiliser des listes de couples formés d'un ensemble et d'une chaîne de caractères. En CAML, une telle liste aura le type `(ensemble * string) list`.

Question 21
Écrire une fonction `satisfiable` *qui prend en argument une formule φ et renvoie la chaîne*

`"Formule non satisfiable"`

s'il n'existe pas de $u \in A^+$ tel que $u \vDash \varphi$ et sinon renvoie un $u \in A^+$ tel que $u \vDash \varphi$. La complexité de votre fonction devra être en $\mathcal{O}(2^{\alpha|\varphi|^\beta})$ où α et β sont deux constantes que vous préciserez.

```
satisfiable : formule -> string
```

Il est permis de décomposer la réponse en quelques fonctions auxiliaires. On rappelle ici que l'on suppose que $A = \{a, b\}$. Donner dans un premier temps les idées clés de l'algorithme avant d'écrire le code.

Question 22
Pour cette question, on pose $A = \{a\}$ et on note a^i le mot formé de i lettres a, par exemple $a^4 = aaaa$. Montrer qu'il n'existe pas de formule φ telle que $L_\varphi = \left\{a^{2i} \;\middle|\; i \geqslant 1\right\}$. On pourra montrer que pour toute formule φ, il existe $N \geqslant 0$ tel que l'on ait l'une des deux situations suivantes :

- *pour tout $n \geqslant N$, $a^n \vDash \varphi$;*
- *pour tout $n \geqslant N$, $a^n \nvDash \varphi$.*

5 Solutions

Solution de l'exercice 1 -

1. $(u,4) \vDash \mathbf{G}(p_a \vee p_b)$ est vrai car les lettres sont a ou b à partir du rang 4.
2. $(u,2) \vDash \mathbf{X}\big(\mathbf{G}(p_a \vee p_c)\big)$ signifie $(u,3) \vDash \mathbf{G}(p_a \vee p_c)$ qui est faux car u_4 est b.
3. Comme $(u,4) \vDash \mathbf{G}(p_a \vee p_b)$ est vrai alors $(u,2) \models \mathbf{F}\big(\mathbf{G}(p_a \vee p_b)\big)$ est vrai.
4. $u \vDash (p_a \vee p_b)\mathbf{U}(p_a \vee p_c)$ est vrai car $(u,j) \vDash p_b$ pour $j \leqslant 2$ et $(u,3) \models p_c$.

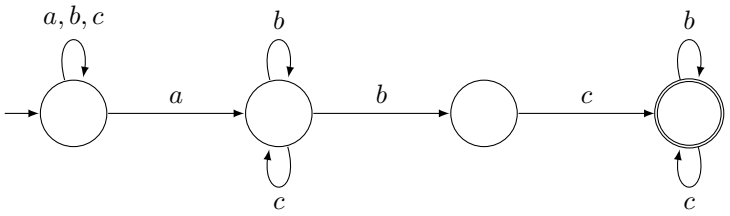
Solution de l'exercice 2 - La formule $\mathbf{F}\big(p_a \wedge \mathbf{F}(p_b)\big)$ convient.

Solution de l'exercice 3 - Un indice i est valide pour un mot u si et seulement si $(u,i) \vDash \mathbf{vrai}$. Le dernier indice valide d'un mot u est l'indice valide tel que son successeur ne l'est pas, il est donc caractérisé par $\mathbf{fin} = \mathbf{vrai} \wedge \neg\mathbf{X}(\mathbf{vrai})$

Solution de l'exercice 4 - u se termine par a si et seulement si la dernière lettre est a. Il doit donc exister une lettre qui est finale et qui vaut a : $\mathbf{F}(\mathrm{Fin} \wedge p_a)$.
On peut aussi exprimer qu'il y un a qui suit chaque position, en particulier la dernière.
On aboutit à la formule $\mathbf{G}\big(\mathbf{F}(p_a)\big)$.

Solution de l'exercice 5 - La périodicité s'exprime par le fait que, pour toute lettre, selon qu'elle vaille a ou b la suivante vaut b ou a ; chaque lettre vérifie $(p_a \wedge \mathbf{X}(p_b))$ ou $(p_b \wedge \mathbf{X}(p_a))$. La dernière lettre, qui est un b ne vérifie pas $(p_b \wedge \mathbf{X}(p_a))$, il faut ajouter la condition $p_b \wedge \mathrm{Fin}$.
De plus, la première lettre doit être un a.
Le mot doit donc vérifier $p_a \wedge \mathbf{G}\big((p_a \wedge \mathbf{X}(p_b)) \vee (p_b \wedge \mathbf{X}(p_a)) \vee (p_b \wedge \mathrm{Fin})\big)$.

Solution de l'exercice 6 - On veut les mots qui contiennent un a suivi uniquement de b et c, avec dans cette suite l'apparition d'un bc.
Une expression régulière associée est $(a+b+c)^*.a.b^*.b.c.(b+c)^*$.
Un automate qui reconnaît le langage peut être



Solution de l'exercice 7 - On définit les nouveaux connecteurs U_j par la sémantique $(u,i) \vDash \varphi\mathbf{U_j}\psi$ si et seulement si $(u,j) \vDash \psi$ et $(u,k) \vDash \varphi$ pour $i \leqslant k < j$.
On remarque que $(u,i) \vDash \varphi\mathbf{U_i}\psi$ se traduit par $(u,i) \vDash \psi$ et, pour $i < j$,
$(u,i) \vDash \varphi\mathbf{U_j}\psi$ se traduit par $(u,i) \vDash \varphi$ et $(u,i+1) \vDash \varphi\mathbf{U_j}\psi$, c'est-à-dire, $(u,i) \vDash \mathbf{X}(\varphi\mathbf{U_j}\psi)$.

Comme la définition de $(u,i) \vDash \varphi U\psi$ peut se lire sous la forme
$(u,i) \vDash \varphi U_i\psi$ ou il existe j tel que $i+1 \leqslant j < |u|$ et $(u,i) \vDash \varphi U_j\psi$ on obtient

$$\varphi U\psi \equiv \psi \vee \big(\varphi \wedge \mathbf{X}(\varphi U\psi)\big)$$

Solution de l'exercice 8 - On doit séparer selon les motifs :

```
let rec taille form =
   match form with
   |VRAI -> 1
   |Predicat(_) -> 1
   |NON f -> 1 + taille f
   |ET (fg, fd) -> 1 + taille fg + taille fd
   |OU (fg, fd) -> 1 + taille fg + taille fd
   |X f -> 1 + taille f
   |G f -> 1 + taille f
   |F f -> 1 + taille f
   |U (fg, fd) -> 1 + taille fg + taille fd;;
```

Dans la suite on notera $|\varphi|$ la taille d'une formule φ.

Solution de l'exercice 9 - $(u,i) \vDash \mathbf{F}(\varphi)$ signifie qu'il existe un entier $j \geqslant i$, valide pour u, tel que $(u,j) \vDash \varphi$. Pour $i \leqslant k < j$ on a $(u,i) \vDash \mathbf{vrai}$ donc $\mathbf{F}(\varphi) \equiv \mathbf{vrai}\,\mathbf{U}\varphi$.

```
let rec normaliseF form =
   match form with
   |VRAI -> VRAI
   |Predicat k -> Predicat k
   |NON f -> NON(normaliseF f)
   |ET (fg,fd) -> ET(normaliseF fg, normaliseF fd)
   |OU (fg,fd) -> OU(normaliseF fg, normaliseF fd)
   |X f -> X (normaliseF f)
   |G f -> G (normaliseF f)
   |F f -> U (VRAI, normaliseF f)
   |U (fg,fd) -> U (normaliseF fg, normaliseF fd);;
```

L'algorithme termine car chaque appel récursif est fait sur une formule de taille strictement inférieure : on doit aboutir aux cas élémentaires.
La complexité est linéaire car les formules ci-dessus permettent de majorer la complexité par la taille ; elle est un $\mathcal{O}\big(|\varphi|\big)$.

Solution de l'exercice 10 -
On peut remarquer qu'on a $\mathbf{G}(\varphi) \equiv \varphi\,\mathbf{U}(\varphi \wedge \mathrm{Fin}) \equiv \varphi\,\mathbf{U}(\varphi \wedge \mathbf{vrai} \wedge \mathbf{X}(\neg\mathbf{vrai}))$.
On peut trouver plus simple en remarquant que $\mathbf{G}$ est une double négation de $\mathbf{F}$: $\mathbf{G}(\varphi) \equiv \neg\big(F(\neg\varphi)\big)$, avec la formule ci-dessus on obtient

$$\mathbf{G}(\varphi) \equiv \neg\big(\mathbf{vrai}\mathbf{U}(\neg\varphi)\big)$$

On peut utiliser ces formules pour éliminer les $\mathbf{G}$ puis éliminer les $\mathbf{F}$ en même temps.

```
let rec normalise form =
   match form with
   |VRAI -> VRAI
   |Predicat k-> Predicat k
   |NON f -> NON (normalise f)
   |ET (fg, fd) -> ET (normalise fg, normalise fd)
   |OU (fg, fd) -> OU(normalise fg, normalise fd)
   |X f -> X (normalise f)
   |G f -> NON(U (VRAI, NON (normalise f)))
   |F f -> U (VRAI, normaliseF f)
   |U (fg, fd) -> U (normalise fg, normalise fd);;
```

Comme ci-dessus la terminaison et la complexité linéaire proviennent de la majoration du nombre d'instruction par la taille de l'arbre.

Solution de l'exercice 11 - On utilise la question **7**.

```
let rec veriteN u i form =
   let n = String.length u in
   if i >= n || i < 0
   then false
   else match form with
      |VRAI -> true
      |Predicat k -> u.[i] = k
      |NON f -> not (veriteN u i f)
      |ET (fg, fd) -> (veriteN u i fg) && (veriteN u i fd)
      |OU (fg, fd) -> (veriteN u i fg) || (veriteN u i fd)
      |X f -> veriteN u (i+1) f
      |U (fg, fd) ->   (veriteN u i fd)
                    || (veriteN u i fg) && (veriteN u (i+1)
                       form)
      |_ -> failwith "Formule non normalisée";;
```

Dans chaque cas on appelle récursivement `veriteN` à une ou plusieurs formules qui sont soit de taille strictement inférieure, soit avec un mot de longueur strictement inférieure en considérant que `(u,i)` représente en fait le suffixe de u obtenu en lui enlevant i lettres. Ainsi la somme $|u|+|\varphi|$ est strictement décroissante donc on aboutit donc à une formule atomique ou à un mot vide et, dans ces deux cas, la récursivité s'arrête. L'algorithme termine.
On note $C(p,\varphi)$ la complexité du calcul de `veriteN u i phi` avec $|u| = p + i$.
Si $\varphi = \varphi_1 \,\mathbf{U}\, \varphi_2$ on a $C(p,\varphi) = C(p,\varphi_1) + C(p,\varphi_2) + C(p-1,\varphi) + 2$.

Comme $C(0,\varphi) = 0$ on en déduit que $C(p,\varphi) = \sum_{k=1}^{p} \big(C(k,\varphi_1) + C(k,\varphi_2) + 2\big)$.

On définit ψ_r par $\psi_1 = p_a$ et $\psi_r = p_a \,\mathbf{U}\, \psi_{r-1}$ et $\gamma(p,r) = C(p,\psi_r)$.

La formule ci-dessus donne $\gamma(p,r) = \sum_{k=1}^{p} \big(\gamma(k,r-1) + 3\big)$ avec $\gamma(p,1) = 1$.

On a donc $\gamma(p,2) = 4p$, $\gamma(p,3) = 2p^2 + 5p$ et on montre par récurrence que $\gamma(p,r)$ est un polynôme de de degré $r-1$ en p. Comme la taille de ψ_r est $2r-1$ on a une complexité équivalente à $Kp^{(|\psi_r|-1)/2}$ exponentielle en la taille.
Dans le cas général la complexité est donc au moins exponentielle en la taille.

Solution de l'exercice 12 - Il suffit de convertir l'arbre de la formule en arbre-ensemble en marquant par faux chaque nœud.

```
let rec initialise form =
   match form with
   |VRAI -> AVRAI, false
   |Predicat k -> APredicat k, false
   |NON f -> ANON(initialise f), false
   |ET (fg, fd) -> AET (initialise fg, initialise fd), false
   |OU (fg, fd) -> AOU (initialise fg, initialise fd), false
   |X f -> AX (initialise f), false
   |U (fg, fd) -> AU (initialise fg, initialise fd), false
   |_ -> failwith "Formule non normalisée";;
```

Solution de l'exercice 13 - Soit une formule normalisée φ ; on montre par induction structurelle que l'appartenance de φ à $\mathcal{S}'$ dépend uniquement de a et de $\mathcal{S}$.
C'est vrai pour les formules "terminales" :
- Si $\varphi = \mathbf{vrai}$, alors φ appartient toujours à $\mathcal{S}'$ car $a \cdot u$ n'est pas vide.
- Si $\varphi = p_x$ alors φ appartient à $\mathcal{S}'$ si et seulement si $a = x$.

On suppose que l'appartenance de φ_1, φ_2 et ψ à $\mathcal{S}'$ ne dépend que a et de $\mathcal{S}$.
- Si $\varphi = \neg\psi$ alors φ appartient à $\mathcal{S}'$ si et seulement si ψ n'appartient pas à $\mathcal{S}'$.

- Si $\varphi = \psi_1 \wedge \psi_2$, alors φ appartient à $\mathcal{S}'$ si et seulement si ψ_1 et ψ_2 appartiennent à $\mathcal{S}'$.
- Si $\varphi = \psi_1 \vee \psi_2$, alors φ appartient à $\mathcal{S}'$ si et seulement si ψ_1 ou ψ_2 appartient à $\mathcal{S}'$.
- Si $\varphi = \mathbf{X}\psi$, alors φ appartient à $\mathcal{S}'$ si et seulement si ψ appartient à $\mathcal{S}$.
- si $\varphi = \psi_1 \mathbf{U} \psi_2$, alors φ appartient à $\mathcal{S}'$ si et seulement ψ_2 appartient à $\mathcal{S}'$ ou ψ_1 appartient à $\mathcal{S}'$ et φ appartient à $\mathcal{S}$.

Dans tous les cas on voit que l'appartenance de φ à $\mathcal{S}'$ ne dépend que de a et de $\mathcal{S}$.

Solution de l'exercice 14 - On agit selon la première composante du couple

```
let rec maj ens a =
   match (fst ens) with
   |AVRAI -> AVRAI, true
   |APredicat k -> APredicat k, a = k
   |ANON f -> let s, b = maj f a in ANON (s, b), not b
   |AET (fg, fd) -> let sg,bg = maj fg a and sd,bd = maj fd a
                    in AET ((sg, bg), (sd, bd)), bg && bd
   |AOU (fg, fd) -> let sg,bg = maj fg a and sd,bd = maj fd a
                    in AET ((sg, bg), (sd, bd)), bg || bd
   |AX f -> let s = maj f a in AX s, (snd ens)
   |AU (fg, fd) -> let sg,bg = maj fg a and sd,bd = maj fd a
                   in AU ((sg, bg), (sd, bd)),
                         (bd || bg && (snd ens)) ;;
```

Solution de l'exercice 15 - On construit l'ensemble à partir de l'ensemble associé au mot vide, celui construit à la question 12, puis en ajoutant les lettres une-par-une depuis la dernière. On doit donc lire le mot de droite à gauche.

```
let sousFormulesVraies form u =
  let ens = ref (initialise form) in
  let n = String.length u in
  for i = (n-1) downto 0 do
    ens := maj !ens (u.[i]) done;
  !ens;;
```

La fonction `maj` termine et a une complexité en $\mathcal{O}\big(|\varphi|\big)$. On en fait l'appel $|u|$ fois donc `sousFormulesVraies` termine et a une complexité en $\mathcal{O}\big(|u|.|\varphi|\big)$.

Solution de l'exercice 16 - La vérité de la formule est le booléen associé à la racine.

```
let veriteBis form u = snd (sousFormulesVraies form u);;
```

Solution de l'exercice 17 - La question 14 montre que les lettres agissent sur les `ensembles` associés à une formule en modifiant les booléens de chaque nœud.
On peut donc définir un automate par

- les états sont les objets de type `ensemble` associés à la formule
- le résultat de $\delta(e, x)$ est `maj e x`
- l'état initial est celui associé au mot vide, `initialise form`
- les états finaux sont ceux pour qui la racine a le booléen `true`.

Cet automate comporte $2^{|\varphi|}$ états.
Par construction un mot $u = u_0 u_1 \cdots u_{p-1}$ est reconnu par l'automate si et seulement on arrive à un état final avec le trajet $u_{p-1} u_{p-2} \cdots u_1 u_0$, c'est-à-dire $\widetilde{u} \vDash \varphi$.
Le langage reconnu est bien $\widetilde{L_\varphi}$.

Solution de l'exercice 18 - Si on retourne l'automate ci-dessus on obtient un automate non-déterministe à $2^{|\varphi|}$ états qui reconnaît L_φ.

Solution de l'exercice 19 - Les états accessibles dans un automate déterministe peuvent être calculés par un algorithme de parcours de graphe : on cherche les sommets accessibles depuis l'état initial.

Solution de l'exercice 20 - $u \vDash \varphi$ équivaut à $u \in L_\varphi$ donc équivaut à $\widetilde{u} \in \widetilde{L_\varphi}$.
Si un état est accessible dans un automate à N élément alors il existe un mot de taille au plus $N-1$ qui y accède ; tout mot de longueur supérieure passerait deux fois par un même état, il suffit de supprimer alors la boucle pour trouver un mot plus court.
Ainsi, si $\widetilde{L_\varphi}$ est non vide, il contient un mot de longueur $2^{|\varphi|}-1$ au plus donc φ est satisfiable par un mot de longueur $2^{|\varphi|}-1$ au plus.

Solution de l'exercice 21 - La remarque précédente indique qu'il suffit de tester tous les mots de longueur au plus $2^{|\varphi|}-1$ pour savoir si une formule φ est satisfiable. Il serait sans doute improductif de tester tous les mots : il y an a $2^{2^{|\varphi|}}-1$!
On revient à l'idée du parcours de graphe.
Ici les voisins sont simplement les transformés de l'ensemble des sous-formules par a et par b.
On a besoin de gérer les ensembles de sous-formules déjà traités, le sujet suggère une liste. Il faudra chercher une élément (un ensemble de sous-formule) dans une liste.
On va donc gérer une liste de couples (ensemble, mot) à explorer en partant de la liste réduite à l'état initial couplé avec le mot vide.
À chaque étape

- on prend le premier couple
- si l'ensemble n'a pas été vu on l'ajoute dans la liste des vus
- on calcule sa mise-à-jour par a,
- si elle est terminale on sort le résultat
- sinon on l'ajoute dans la liste
- on fait de même pour b.
- On termine si la longueur du mot est trop grande.

On devrait ajouter les lettres à gauche pour garder le mot de l'automate puis inverser le mot mais, comme la fonction de mise-à-jour ne dépend pas du mot on construit directement le mot en ajoutant les lettres à droite.
On ne fera pas plus de passages qu'il n'y a de transition depuis les sommets : $2.(2^{|\varphi|}-1)$.
Pour chaque sommet de `aVoir` on parcourt la liste `vus` qui est de longueur $2^{|\varphi|}-1$ au plus et chaque comparaison compare deux arbres de taille $|\varphi|$; on effectue alors un nombre fini d'opérations.
Ainsi la complexité est de l'ordre de $2.(2^{|\varphi|}-1).\big((2^{|\varphi|}-1)|\varphi|+k\big) = \mathcal{O}\big(|\varphi|2^{2|\varphi|}\big)$.

```
let final ens = snd ens;;

let rec appartient x liste =
  match liste with
  |[] -> false
  |t::q when t = x -> true
  |t::q -> appartient x q;;
```

```
let satisfiable form =
  let rec test vus aVoir =
    match aVoir with
    |[] -> "Formule non satisfiable"
    |(ens, u)::q when not (appartient ens vus)
                -> let ensa = maj ens 'a' in
                   let ua = u^"a" in
                   let ensb = maj ens 'b' in
                   let ub = u^"b" in
                   let vus1 = ens::vus in
                   let aVoir1 = (ensa, ua)::(ensb, ub)::q in
                   if final ensa
                   then ua
                   else if final ensb
                        then ub
                        else test vus1 aVoir1
    |_::q -> test vus q
  in test [] [(initialise form, "")];;
```

Solution de l'exercice 22 - On note $E(\varphi) = \{n \in \mathbb{N}^* \ ; \ a^n \vDash \varphi\}$. On veut montrer que, pour tout φ, il existe un entier N (ou N_φ) tel que $E(\varphi) \subset [1;N]$, $(N-)$, ou $[N;+\infty[\subset E(\varphi)$, $(N+)$.
On procède par induction structurelle

- $E(\mathbf{vrai}) = \mathbb{N}^*$ et $E(p_a) = \mathbb{N}^*$.
- $E(\neg\varphi) = \mathbb{N}^* \setminus E(\varphi)$ et $E\big(\mathbf{X}(\varphi)\big) = \{1+n \ ; \ n \in E(\varphi)\}$.
- On détermine les ensembles pour les lois binaires

$E(\varphi_1)$	$E(\varphi_2)$	$E(\varphi_1 \wedge \varphi_2)$	$E(\varphi_1 \vee \varphi_2)$	$E(\varphi_1 \,\mathbf{U}\, \varphi_2)$
N_1-	N_2-	$\max(N_1,N_2)-$	N_1-	N_2-
N_1-	N_2+	N_2+	N_1-	N_2+
N_1+	N_2-	N_1+	N_2-	N_2-
N_1+	N_2+	N_1+	$\max(N_1,N_2)+$	N_1+

On en déduit que $E(\varphi)$ ne peut être égal à $2\mathbb{N}^*$.

Devoir **E**

DS3 : Mines 2016
Graphe du WEB

Préliminaire concernant la programmation

Il faudra coder des fonctions à l'aide du langage de programmation OCaml, tout autre langage étant exclu. Lorsque le candidat écrira une fonction, il pourra faire appel à d'autres fonctions définies dans les questions précédentes ; il pourra aussi définir des fonctions auxiliaires. Quand l'énoncé demande de coder une fonction, il n'est pas nécessaire de justifier que celle-ci est correcte, sauf si l'énoncé le demande explicitement. Enfin, si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien vérifiées.

Dans les énoncés, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

1 Présentation

Le World Wide Web, ou Web, est un ensemble de pages (identifiées de manière unique par leurs adresses, ou URL pour Uniform Resource Locators, de la forme `http://mines-ponts.fr/index.php`) reliées les unes aux autres par des hyperliens. Le Web est souvent modélisé comme un graphe orienté dont les sommets sont les pages Web et les arcs les hyperliens entre pages. Le Web étant potentiellement infini, on s'intéresse à des sous-graphes du Web obtenus en naviguant sur le Web, c'est-à-dire en le parcourant page par page, en suivant les hyperliens d'une manière bien déterminée. Ce parcours du Web pour en collecter des sous-graphes est réalisé de manière automatique par des logiciels autonomes appelés Web crawlers ou crawlers en anglais, ou collecteurs en français.

2 Fonctions utilitaires

Nous allons tout d'abord coder certaines fonctions de manipulation de structures de données de base, qui seront utiles dans le reste de l'exercice.

Question 1
Écrire une fonction `renverse : 'a liste -> 'a list` *qui renvoie la liste passée en paramètre retournée ; on n'utilisera pas* `List.rev`. `renverse [4; 2; 3]` *renvoie* `[3; 2; 4]`.

Question 2
On part d'une liste `liste` *de type* `(a * a list) list`*, c'est une liste de couples* `[(x1, l1) ; ... ; (xn, ln)]`*, où chaque* `xi` *(lire x_i) est un élément de type* `'a`*, et* `li` *(lire l_i) une liste d'éléments de type* `'a` *de la forme* `[yi1 ; ... ;yipi]`. `yij` *se lit $y_{i,j}$ et* `yipi`*, y_{i,p_i}.*
Coder une fonction `aplatir : (a * a list) list -> a list`*, telle que* `aplatir liste` *est une liste d'éléments de type* `'a` *:*

```
[x1;y11;...;y1p1;x2;y21;...;y2p2;...;xn;yn1;...;ynpn]
```

Question 3
Coder une fonction `tri_fusion : (a * b) list -> (a * b) list` *triant une liste de couples (x, y) par ordre décroissant de la valeur de la seconde composante y de chaque couple. On devra utiliser l'algorithme de tri par fusion (aussi appelé « tri fusion »).*
Donner, sans démonstration, la complexité de cet algorithme.

2.1 Dictionnaires

On va utiliser dans la suite du problème un type de données `dictionnaire` qui permet de stocker des couples formés d'une chaîne de caractères (une clef) et d'un entier (une valeur). On dit que le dictionnaire associe la valeur à la clef. A chaque clef présente dans le dictionnaire est associée une seule valeur. Les fonctions suivantes sont supposées être prédéfinies :

- `dictionnaire_vide : unit -> dictionnaire`.
  l'appel `dictionnaire_vide ()` crée un nouveau dictionnaire vide.
- `ajoute : string -> int -> dictionnaire -> dictionnaire`.
  l'appel `ajoute clef valeur dict` renvoie un nouveau dictionnaire identique au dictionnaire `dict`, sauf qu'un couple `(clef, valeur)` y a été ajouté.
- `contient : string -> dictionnaire -> bool`.
  l'appel `contient clef dict` renvoie un booléen indiquant s'il y a un couple dont la clef est `clef` dans le dictionnaire `dict`.
- `valeur : string -> dictionnaire -> int`.
  l'appel `valeur clef dict` renvoie la valeur associée à la clef `clef` dans le dictionnaire `dict`. Cette fonction ne peut être appelée que si la clef est présente dans le dictionnaire.

Question 4
Écrire les fonctions ci-dessus si on implémente le dictionnaire par des listes de couples :

```
type dict = (string * int) list;;
```

Quelle la complexité des fonctions écrites en fonction du nombre n d'entrées dans le dictionnaire ?

Les chaînes de caractères sont comparables par les opérateurs classiques, `<, >, <=, >=`.
On peut implémenter aussi les dictionnaires par des arbres binaires :

```
type arbre = Vide | Noeud of arbre * string * int * arbre;;

let dictionnaire_vide () = Vide;;
```

Question 5
Quelle propriété doivent vérifier les chaînes des nœuds de l'arbre pour obtenir des dictionnaires dont la complexité, que l'on donnera sans démonstration, est meilleure que dans la question précédente ? On supposera que les arbres restent équilibrés.

Question 6
Écrire la fonction `contient`.
Expliquer simplement comment modifier la fonction `contient` *en la fonction* `valeur`.

On supposera écrite la fonction `ajoute`.

Dans toute la suite on supposera que les complexités des fonctions `ajoute`, `contient` et `valeur` est en $\mathcal{O}\big(\log(n)\big)$ où n est le nombre d'entrées du dictionnaire.

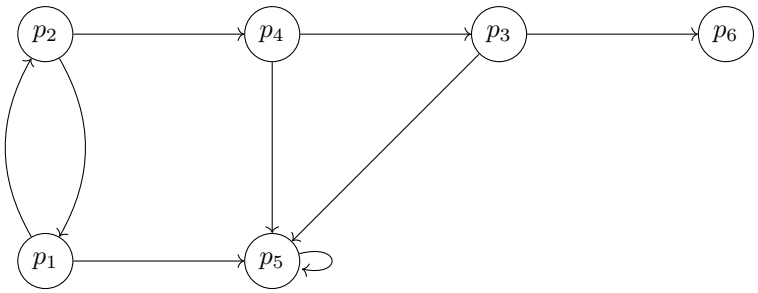
Question 7
Coder `unique : string list -> string list * dictionnaire`*, qui est telle que l'appel* `unique liste` *renvoie un couple* `(liste', dict)` *où* `liste'` *est la liste des chaînes de caractères de liste distinctes (dans l'ordre de leur première occurrence dans liste) et où* `dict` *associe à chaque chaîne de caractères dans* `liste'` *sa position dans* `liste'` *(en numérotant à partir de 0).*

Exemple : l'appel `unique ["x"; "zz"; "x"; "x"; "zz"; "yt"]` renvoie un couple formé de la liste `["x"; "zz"; "yt"]` et d'un dictionnaire associant à `"x"` la valeur `0`, à `"zz"` la valeur `1` et à `"yt"` la valeur `2`.

Question 8
Quelle est la complexité de la fonction `unique` *en terme de la longueur n de la liste liste en argument et du nombre m d'éléments distincts dans la liste liste ? Justifier la réponse.*

3 Crawler simple

Nous allons maintenant implémenter un crawler simple en OCaml.
On suppose fournie une fonction `recupere_liens : string -> string list` prenant en argument l'URL d'une page Web p et renvoyant la liste des URL des pages q pour lesquelles il existe un hyperlien de p à q, dans l'ordre lexicographique.
Pour illustrer le comportement de cette fonction, nous considérons un exemple de mini-graphe du Web à six pages et neuf hyperliens comme suit :



Dans cette représentation, p_1, p_2, etc., sont les URL de pages Web (simplifiées pour l'exemple), et les arcs représentent les hyperliens entre pages Web.
Dans ce mini-graphe, un appel à `recupere_liens "p1"` retourne la liste `["p2"; "p5"]`.
Un crawler est un programme qui, à partir d'une URL et d'un entier n, parcourt le graphe du Web en visitant progressivement les pages dont les liens sont présents dans chaque page rencontrée, en suivant une stratégie de parcours de graphe (par exemple, largeur d'abord, ou profondeur d'abord). À chaque nouvelle page, si celle-ci na pas déjà été visitée, tous ses hyperliens sont récupérés et ajoutés à une liste de liens à traiter.
N.B. Le sujet demande donc d'utiliser une liste pour implémenter une pile ou une file dans les parcours ci-dessous. On notera que, selon que l'on adjoint la liste des voisins (avec l'opérateur `@`) d'un coté ou de l'autre, on construit une file ou une pile.

Le processus s'arrête quand Le crawler a visité n pages distinctes et donc appelé n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter).
Les fonctions `crawler` renvoient en sortie une liste de longueur au plus n de couples `(v, l)` où `v` est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et `l` la liste des liens récupérés sur la page `v`. Ce résultat est appelé un *crawl*.
On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.

Question 9
Coder `crawler_bfs : int -> string -> (string * string list) list` *qui prend en entrée un nombre n de pages et une URL* `u` *et renvoie un crawl. On demande que* `crawler_bfs` *parcoure le graphe du Web en suivant une stratégie en largeur d'abord (breadth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus tôt dans l'exploration.*

Par exemple, sur le mini-graphe, `crawler_bfs 4 "p1"` pourra renvoyer le résultat :
`["p1", ["p2"; "p5"]; "p2", ["p1"; "p4"]; "p5", ["p5"]; "p4", ["p3"; "p5"]]`

Question 10
Coder `crawler_bfs : int -> string -> (string * string list) list` *qui prend en entrée un nombre n de pages et une URL* `u` *et renvoie un crawl. On demande que*`crawler_dfs` *parcoure le graphe du Web en suivant une stratégie en profondeur d'abord (depth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus récemment dans l'exploration.*

Par exemple, sur le mini-graphe, `crawler_dfs 4 "p1"` pourra renvoyer le résultat :
`["p1", ["p2"; "p5"]; "p2", ["p1"; "p4"]; "p4", ["p3"; "p5"]; "p3", ["p5"; "p6"]]`

Question 11
Coder une fonction OCaml `construit_graphe` *de signature*

```
(string * string list) list -> string list * int vect vect
```

telle que si `crawl` *est le résultat renvoyé par un crawler alors* `construit_graphe crawl` *est un couple* `(l, G)` *où* `l` *est une liste de toutes les URL de pages contenues dans la liste* `crawl` *et* `G` *est la matrice d'adjacence du sous-graphe partiel du Web restreint aux pages de la liste* `l` *: `G.(i).(j)` est le nombre de liens découverts dans le crawl de la page d'indice i dans* `l` *vers la page d'indice j dans l.*

Par exemple, sur le mini-graphe, si `crawl` est une variable contenant le résultat de l'appel `crawler_bfs 4 "p1"`, alors `construit_graphe crawl` doit renvoyer :

```
["p1"; "p2"; "p5"; "p4"; "p3"], [|[|0; 1; 1; 0; 0|];
                                  [|1; 0; 0; 1; 0|];
                                  [|0; 0; 1; 0; 0|];
                                  [|0; 0; 1; 0; 1|];
                                  [|0; 0; 0; 0; 0|]|]
```

En particulier :
- p_3 apparaît même s'il na pas été visité dans le crawl ;
- p_6 n'apparaît pas car il na pas été découvert dans le crawl ;
- l'hyperlien de p_3 à p_5 n'apparaît pas car p_3 na pas été visité.

4 Calcul de PageRank

PageRank est une manière d'affecter un score à l'ensemble des pages du Web, imaginée par Sergey Brin et Larry Page, les fondateurs du moteur de recherche Google. L'introduction de PageRank a révolutionné la technologie des moteurs de recherche sur le Web. Nous allons maintenant implémenter le calcul de PageRank.
Étant donnée une partie du Web (où l'ensemble des pages est indexé entre 0 et $n-1$), la matrice de surf aléatoire dans cette partie du Web est la matrice M de taille $n \times n$ définie comme suit :
- s'il n'y a aucun lien depuis une page Web d'indice i, alors pour tout j, $M_{ij} := 1/n$.

- Sinon, s'il y a k_i liens depuis la page Web d'indice i, alors pour tout j, on a $M_{ij} := (1-d) \times G_{ij}/k_i + d/n$, où G_{ij} est le nombre de liens depuis la page d'indice i vers la page d'indice j et d est un nombre réel fixé appartenant à $[0,1]$ (on prend souvent $d = 0,15$).

Cette matrice peut être vue comme décrivant la marche aléatoire d'un surfeur sur le Web. à chaque fois que celui-ci visite une page Web :

- Si cette page ne comporte aucun lien, il visite une page Web arbitraire, choisie aléatoirement de façon uniforme.
- Si cette page comporte au moins un lien, il visite avec une probabilité égale à $1/d$ un des liens sortants de cette page, et avec une probabilité égale à d une page Web arbitraire, choisie aléatoirement de façon uniforme.

Question 12
Coder `surf_aleatoire : float -> int vect vect -> float vect vect`
telle que si d est un nombre entre 0 et 1, et si G est la matrice d'adjacence d'un sous-graphe partiel du Web, alors `surf_aleatoire d G` *renvoie la matrice M de surf aléatoire dans ce sous-graphe. On convertit un entier en flottant avec* `float_of_int` *ou* `Float.of_int`

Question 13
Coder `multiplie : float vect -> float vect vect -> float vect`,
une fonction prenant en argument un vecteur ligne v de taille n et une matrice M de taille $n \times n$ et renvoyant le vecteur ligne w de taille n résultant du produit de v par la matrice M : $w = v.M$. En d'autres termes, pour tout j, $w_j = \sum_i v_i M_{ij}$.

Le PageRank des pages d'un sous-graphe du Web à n pages se calcule par des multiplications successives d'un vecteur ligne par la matrice de surf aléatoire M de ce sous-graphe.
Plus précisément, soit θ un nombre réel strictement positif (par exemple, $\theta = 10^{-4}$) et soit $v^{(0)}$ le vecteur ligne de taille n dont toutes les composantes valent $1/n$.
On pose pour un entier naturel p arbitraire $v^{(p)} := v^{(0)}.M^p$.
L'algorithme de PageRank calcule la suite des $v^{(p)}$ pour $p = 0, 1, \ldots$ jusqu'à ce que
$\|v^{(p+1)} - v^{(p)}\|_1 \leqslant \theta$ et renvoie alors le vecteur $v^{(p+1)}$, considéré comme le vecteur des scores de PageRank. On peut montrer (à l'aide du théorème de Perron-Frobenius) que l'algorithme termine dès lors que d est strictement positif.

PageRank est utilisé pour affecter un score d'importance aux pages du Web. Le vecteur de scores v retourné par l'algorithme de PageRank donne dans v_i le score d'importance de la page d'indice i. Les pages de plus haut score de PageRank sont considérées comme les plus importantes.

Question 14
Coder `pagerank : float -> float vect vect -> float vect`, *une fonction prenant en argument un nombre $\theta > 0$ et une matrice M de surf aléatoire d'un sous-graphe du Web et renvoyant le vecteur des scores de PageRank pour θ et M. La fonction* `pagerank` *devra faire appel à la fonction* `multiplie` *précédemment codée.*
La valeur absolue d'un flottant se calcule avec `Float.abs`.

Question 15
Coder `calcule_pagerank` *de signature*

```
float->float->(string*string list) list->(string*float) list
```

telle que `calcule_pagerank d theta crawl` *renvoie une liste de couples* `(u, s)`, *un couple pour chaque URL découverte dans le crawl, triée par valeur décroissante de* `s`, *où* `u` *est l'URL de cette page et* `s` *son score de PageRank. Ici, d et θ sont les deux paramètres nécessaires au calcul de la matrice de surf aléatoire et du PageRank respectivement. On fera appel à la fonction* `tri_fusion`.

5 Solutions

Solution de l'exercice 1 - La récursivité terminale produit un résultat souvent inversé.

```
let renverse liste =
   let rec aux reste fait =
       match reste with
       |[] -> fait
       |t::q -> aux q (t :: fait)
    in aux liste [];;
```

Solution de l'exercice 2 -

```
let rec aplatir liste =
   match liste with
   |[] -> []
   |(w,l)::reste -> (w::l) @ (aplatir reste);;
```

Solution de l'exercice 3 - On commence par découper la liste en deux parties presque de même longueur.

```
let rec decouper liste =
   match liste with
   |x::y::q -> let (l1, l2)=decouper q in
               x::l1, y::l2
   |_ -> liste, [] ;;
```

On doit savoir fusionner deux listes triées

```
let rec fusion l1 l2 =
   match l1, l2 with
   |[], _ -> l2
   |_, [] -> l1
   |(a1, b1)::q1, (a2, b2)::q2 when b1 < b2 -> (a2, b2)::(
      fusion l1 q2)
   |t1::q1, t2::q2 -> t1::(fusion q1 l2);;
```

Il reste à tout combiner

```
let rec tri_fusion liste =
   match liste with
   |[] -> []
   |[t] -> [t]
   |_ -> let l1, l2 = decouper liste in
         fusion (tri_fusion l1) (tri_fusion l2);;
```

Solution de l'exercice 4 - On commence par deux fonctions de complexité constante

```
let dictionnaire_vide () = [];;

let ajoute cle k dico = (cle, k) :: dico;;
```

Les deux suivantes, semblables, sont de complexité en $\mathcal{O}(n)$.

```
let rec contient cle dico =
   match dico with
   |[] -> false
   |(ch, k)::q when ch = cle -> true
   |_::q -> contient cle q;;

let rec valeur cle dico =
   match dico with
   |[] -> failwith "Clé non présente"
   |(ch, k)::q when ch = cle -> k
   |_::q -> valeur cle q;;
```

Solution de l'exercice 5 - On doit avoir une structure d'arbre binaire de recherche : pour tout nœud n_0, les chaînes des nœuds du fils gauche (resp. du fils droit) doivent être inférieures (resp.supérieures) à la chaîne de n_0. La complexité est lors un $\mathcal{O}(h)$ où h est la hauteur.
Dans le cas d'arbres équilibrés la hauteur est une $\mathcal{O}\big(\log(n)\big)$.

Solution de l'exercice 6 -

```
let rec contient cle dico =
   match dico with
   |Vide -> false
   |Noeud(g, ch, k, d) when ch = cle -> true
   |Noeud(g, ch, k, d) when ch > cle -> contient cle g
   |Noeud(g, ch, k, d) -> contient cle d;;
```

Pour la fonction `valeur` on renvoie une erreur dans le premier cas, on renvoie k dans le second et on remplace `contient` par `valeur` dans les appels récursifs des autres cas.

Solution de l'exercice 7 - On utilise une fonction auxiliaire dont les paramètres sont la liste restant à traiter, la liste déjà construite, le dictionnaire déjà construit et l'indice du premier terme à ajouter

```
let unique liste =
   let rec aux reste fait dico k =
       match reste with
       |[] -> renverse fait, dico
       |cle::q when contient cle dico -> aux q fait dico k
       |cle::q -> aux q (cle::fait) (ajoute cle k dico) (k +
          1)
   in aux liste [] (dictionnaire_vide ()) 0;;
```

Solution de l'exercice 8 -
On peut majorer le coût des fonctions `contient` et `ajoute` par un $\mathcal{O}\big(\log(m)+1\big)$ car le dictionnaire contient au plus m éléments ; on ajoute 1 car m peut rester égal à 1. Ainsi la complexité de `aux` est majorée par un $\mathcal{O}\big(n\log(m)+n\big)$; la fonction `renverse` n'ajoutant qu'un $\mathcal{O}(n)$.

Solution de l'exercice 9 -

```
let crawler_bfs n page =
   let rec aux pages vus n liste =
      match n, liste with
      |0, _ -> renverse pages
      |_, [] -> renverse pages
      |_, p::q when contient p vus -> aux pages vus n q
      |_, p::q -> let l = recupere_liens p in
                  aux ((p, l)::pages) (ajoute p 17 vus) (n-1)
                     (q @ l)
   in aux [] (dictionnaire_vide ()) n [page];;
```

Les 4 paramètres de la fonction auxiliaire sont

1. les pages déjà visitées, dans l'ordre inverse,
2. le dictionnaire des pages vues, avec une valeur inutilisée,
3. le nombre de pages restant à découvrir,
4. la liste des pages en attente.

Les deux premiers cas du pattern-matching correspondent à la fin de la recherche, soit on a vu assez de pages, soit il n'y ap lus de nouvelles pages à visiter.
Le troisième cas passe une page déjà visitée.
Le dernier cas traite une nouvelle page : on l'ajoute au dictionnaire et aux pages vues et on ajoute ses liens **à droite** des pages à voir pour obtenir un parcours en largeur.
On notera qu'on n'a pas optimisé la fonction :

- on ne teste pas si une page a déjà été visitée avant de l'ajouter dans la file
- l'ajout a une complexité linéaire à cause de la concaténation alors que la complexité peut être constante (en moyenne) dans un type bien écrit.

Solution de l'exercice 10 - On peut écrire une fonction analogue en adjoignant en tête

```
let crawler_dfs n page =
   let rec aux pages vus n liste =
      match n, liste with
      |0, _ -> renverse pages
      |_, [] -> renverse pages
      |_, p::q when contient p vus -> aux pages vus n q
      |_, p::q -> let l = recupere_liens p in
                  aux ((p, l)::pages) (ajoute p 17 vus) (n-1)
                     (l @ q)
   in aux [] (dictionnaire_vide ()) n [page];;
```

Solution de l'exercice 11 -

```
let construit_graphe crawl =
   let sommets, dico = unique (aplatir crawl) in
   let n = List.length s in
   let g = Array.make_matrix n n 0 in
   let rec miseajour i l =
      match l with
      |[] -> ()
      |cle::q -> let j = valeur cle dico in
                 g.(i).(j) <- g.(i).(j) + 1;
                 miseajour i q in
   let rec parcours crawl =
      match crawl with
      |[] -> ()
      |(cle, liste)::q -> let i = valeur cle dico in
                          miseajour i liste;
                          parcours q in
   parcours crawl;
   sommets, g;;
```

Solution de l'exercice 12 - Des flottants ! Donc des opérateurs avec un point.

```
let surf_aleatoire d g =
   let n = Array.length g in
   let m = Array.make_matrix n n (1.0 /. (float_of_int n)) in
   for i = 0 to (n-1) do
      let k0 = ref 0 in
      for j = 0 to (n-1) do k0 := !k0 + g.(i).(j) done;
      let k = float_of_int !k0 in
      if k > 0.0
      then for j = 0 to (n-1)
           do m.(i).(j) <- (1. -. d) *. (float_of_int g.(i).(j
              )) /. k
                          +. d *. m.(i).(j) done done;
   m;;
```

Solution de l'exercice 13 -

```
let multiplie v m =
   let n = Array.length v in
   let w = Array.make n 0.0 in
   for j = 0 to n-1 do
      for i = 0 to n-1
      do w.(j) <- w.(j) +. v.(i)*.m.(i).(j) done done;
   w;;
```

Solution de l'exercice 14 - On définit tout d'abord une fonction qui calcule la distance entre deux vecteurs de même taille pour la norme $\|.\|_1$.

```
let distance v w =
   let n = Array.length v in
   let s = ref 0.0 in
   for i = 0 to (n-1)
   do s := !s +. Float.abs (v.(i) -. w.(i)) done;
!s ;;
```

```
let pagerank theta m =
   let n = Array.length m in
   let rec aux v =
      let w = multiplie v m in
      if distance v w <= theta
      then w
      else aux w in
   aux (Array.make n (1.0 /. (float_of_int n))) ;;
```

Solution de l'exercice 15 -

```
let calcule_pagerank d theta crawl =
   let s, g = construit_graphe crawl in
   let m = surf_aleatoire d g in
   let v = pagerank theta m in
   let rec construireliste sommets i =
      match sommets with
      |[] -> []
      |s::reste -> (s, v.(i))::(construireliste reste (i+1))
         in
   tri_fusion (construireliste s 0);;
```

Devoir F

DS3 CCINP 2020

Les questions posées sont beaucoup plus facilement résolues en Python grâce à la structure hybride des listes python : on les construit par adjonction, comme les listes OCaml mais on les lit séquentiellement, comme les tableaux Caml.

Question 10
Pierre peut connaître immédiatement x et et y si et seulement si Π est le produit de deux nombres premiers distincts.

On a la majoration $x + y = S \leqslant n$. On en déduit $P = x.y \leqslant x.(n - x) = \frac{n^2}{4} - \left(x - \frac{n}{2}\right)^2 \leqslant \frac{n^2}{4}$.
Quand on cherchera un nombre de produits on pourra classer les produits comme les indices d'un tableau de taille $\left\lfloor \frac{n^2}{4} \right\rfloor + 1$.

Question 11

```
def CoupleProd(n):
    m = (n**2)//4+1
    nb_prods = [0]*m
    for x in range(2, n):
        for y in range(x+1, n-x+1):
            p = x*y
            nb_prods[p]  += 1
    listeP = []
    for p in range(m):
        if nb_prods[p] > 1:
            listeP.append(p)
    return listeP
```

On peut simplifier les 5 dernières lignes en

```
    return [p for p in range(m) if nb_prods[p] > 1]
```

Question 12
On cherche les produits de la forme $x.(S - x)$ pour $x \geqslant 2$ et $x < S - x$ donc $x < \frac{S}{2}$.
Or $x \mapsto x.(S - x)$ est strictement croissante sur $[-\infty; \frac{S}{2}[$ donc les valeurs sont distinctes : on n'a pas besoin de tester les doublons.
Je n'ai pas compris l'utilité de la variable n, on suppose $S \leqslant n$ donc n ne sert pas.

```
def Prod(S, n):
    prods = []
    x = 2
    while 2*x < S:
        prods.append(x*(S-x))
        x = x + 1
    return prods
```

Question 13
La valeur minimale d'une somme admissible est $S = 2 + 3$.

```
def inclus(liste1, liste2):
    for x in liste1:
        if not x in liste2:
            return False
    return True

def Candidat_S(n):
    listeP = CoupleProd(n)
    cand = []
    for S in range(5, n+1):
        if inclus(Prod(S, n), listeP):
            cand.append(S)
    return cand
```

Question 14
Je ne comprends pas bien l'énoncé, il semble en fait parler de Sophie plutôt que Pierre. Pour que Pierre puisse connaître Σ, il suffit que, parmi les sommes possibles associées au produit Π, il y en ait une seule appartenant à `Candidat_S(n)`. La question est utile en fait pour Sophie, même s'il serait plus simple d'écrire une fonction `Unique_P`.
On reprend le principe de la question `CoupleProd`.

```
def Double_p(n):
    m = (n*n)//4+1
    nb_prods = [0]*m
    for S in Candidat_S(n):
        for P in Prod(s, n):
            nb_prods[P] += 1
    return  [k for k in range(m) if np_prods[k] > 1]
```

Question 15
On écrit ce qui est demandé bien que cela semble inutile
On commence par calculer l'intersection.

```
def inter(liste1, liste2):
    int = []
    for x in liste1:
        if x in liste2:
            int.append(x)
    return int
```

```
def Reste_S(n):
    reste = []
    db = Double_P(n)
    for S in Candidat_S(n):
        int = inter(d, Prod(S, n))
        if len(int) == 1:
            reste.append(S)
    return reste
```

Question 16

```
def Reste_P(S, n):
    db = Double_P(n)
    reste = []
    for P in Prod(S, n):
        if not(P in db):
            reste.append(P)
    return reste
```

Question 17
Il s'agit de la résolution d'une équation de degré 2 ; à quoi sert n ?

```
def Solution(P, S, n):
    d = (s**2 - 4*p)**0.5
    return (s-d)/2, (s+d)/2
```

Devoir G

DS3+ X-ENS 2017

1 Jeu à un joueur, parcours en largeur

Question 1
On peut lister les nombres qu'on peut atteindre avec une profondeur donnée : on simule en fait le parcours en largeur en ajoutant à chaque niveau les valeurs calculées à partir du niveau précédent qui n'ont pas encore été atteintes.

- 0 : 1
- 1 : 2
- 2 : 3, 4
- 3 : 5, 6, 8
- 4 : 7, 9, 10, 12, 16
- 5 : 11, 13, 14, 17, 18, 20, 24, 32
- 6 : 15, 19, 21, 22, 25, 26, 28, 33, 34, 36, 40, 48, 64
- 7 : 23, 27, 29, 30, 35, 37, 38, 41, 42, 44, 49, 50, ...

On voit que 42 apparaît pour la première fois à la profondeur 7 avec la solution optimale

1, 2, 4, 5, 10, 20, 21, 42

Question 2
Invariant On propose l'invariant $\mathcal{P}(p)$ suivant :
lors du passage de la boucle `tant que` pour la valeur p,

- A ne contient que des états accessibles,
- ces états sont de profondeur au plus p,
- A contient tous les états de profondeur p.

$\mathcal{P}(0)$ est vraie car A contient initialement e_0, seul état accessible de profondeur 0.
Si la propriété $\mathcal{P}(p)$ est vérifiée et qu'aucun état de A n'appartient à F alors on place dans B les voisins des états de A :

Démonstration de l'invariant

- ces voisins d'états accessibles sont donc accessibles,
- ils sont voisins d'états de profondeur p au plus sont de profondeur $p+1$ au plus,
- tout état s de profondeur $p+1$ est le voisin d'un état s' de profondeur p, or s' appartient à A donc s appartient à B.

Après avoir remplacé p par $p+1$ et A par B on voit donc que $\mathcal{P}(p+1)$ est vérifiée.
On a donc prouvé par récurrence que $\mathcal{P}(p)$ est vérifiée pour tout p qui correspond à un passage de la boucle `tant que`.

Preuve Ainsi, si le parcours renvoie VRAI, c'est qu'on a trouvé un état accessible dans F donc qu'il existe solution. S'il existe une solution, il en existe une qui est optimale.
Inversement s'il existe une solution (optimale) alors il existe un état t accessible dans F.
Si t est à la profondeur p_0, il sera découvert lors du passage pour la valeur p_0 pour p, sauf si un autre sommet accessible a été découvert auparavant. Dans les deux cas l'algorithme renvoie VRAI.

Question 3

On note A_p l'ensemble A lors du passage de la boucle pour la valeur p.

Complexité spatiale Chaque élément de A_p admet deux voisins donc la taille de A_{p+1} est au plus le double de celle de A_p : $|A_{p+1}| \leqslant 2|A_p|$. On en déduit $A_p \leqslant 2^p$.
On a $B = B_1 \cup B_2$ avec $B_1 = \{x+1\ ; x \in A\}$ et $B_2 = \{2x\ ; x \in A\}$.
B_1 et B_2 on le même cardinal que A_p et sont formés chacun d'éléments distincts.
Touts les éléments de B_2 sont pairs donc la moitié, au moins, des éléments de B sont pairs.
Ainsi la moitié, au moins, des éléments de A_{p+1} sont pairs pour tout p.
Les élément pairs de A_p vont donner dans B_1 des éléments impairs qui seront donc distincts des éléments de B_2 : on en déduit qu'on a $|A_{p+1}| \geqslant \dfrac{|A_p|}{2} + |A_p|$ pour $p \geqslant 1$. Comme on a $|A_1| = 2 \geqslant \frac{3}{2}$ on en déduit $|A_p| \geqslant \left(\frac{3}{2}\right)^p$.
Ainsi la complexité spatiale lors du passage pour p, qui est la somme des taille de A et de B, $|A_p| + |A_{p+1}|$, est encadrée par deux exponentielles :

$$\left(\frac{3}{2}\right)^p \leqslant \left(\frac{3}{2}\right)^p + \left(\frac{3}{2}\right)^{p+1} \leqslant |A_p| + |A_{p+1}| \leqslant 2^p + 2^{p+1} \leqslant 2^{p+2}$$

Complexité temporelle La complexité temporelle d'une étape est minorée par la taille de B car il faut au moins une instruction par élément ajouté.
Pour chaque x de A on fait un nombre fini d'opération élémentaires puis on ajoute chacun des deux voisins : pour effectuer une union il faut, au pire tester tous les éléments de B pour ne pas ajouter de doublon donc on effectue au plus $a_p(C + 2a_{p+1})$ instructions.
La complexité temporelle $C(p)$ est donc encadrée

$$\left(\frac{3}{2}\right)^p \leqslant |A_{p+1}| \leqslant \sum_{k=0}^{p} |B_p| \leqslant C(p)$$

$$C(p) \leqslant \sum_{k=0}^{p} a_p(C + 2a_{p+1}) \leqslant \sum_{k=0}^{p} 2^{k+2}(C + 2.2^{k+3}) \leqslant C2^{p+3} + 2^{2p+6} \leqslant C'.4^p$$

Les deux complexités sont bien encadrées par des exponentielles.

Question 4

```
let bfs () =
  let rec successeurs A B p =
    match A with
    |[] -> successeurs B [] (p+1)
    |(t::q) -> if final t
               then p
               else successeurs q ((suivants t) @ B) p
    in successeurs [initial] [] 0 ;;
```

La fonction `successeurs` prend en entrée l'état des variables `A`, `B` et `p` décrites dans le sujet, et renvoie la solution. Le cas `A = []` correspond à la fin de la boucle **pour tout**.

Question 5
On a montré à la question 2 que si l'algorithme termine en renvoyant VRAI au passage pour la valeur p alors l'état appartenant à F considéré est à la profondeur p au plus.
Il ne peut pas être à une profondeur $k < p$ car sinon il était dans l'ensemble A_k lors du passage pour la valeur k et l'algorithme aurait terminé lors de ce passage.
Ainsi la valeur renvoyée est la profondeur d'une solution.
Si ce n'était pas la profondeur d'une solution optimale alors un état correspondant à une solution optimale aurait appartenu à A lors d'une valeur antérieure de p ce qui aurait fait terminer l'algorithme plus tôt : c'est impossible.
L'algorithme renvoie la profondeur d'une solution optimale.

2 Parcours en profondeur

Question 6
Sens direct Supposons qu'il existe une solution de profondeur inférieure ou égale à m : $(e_0, \dots, e_p)$.
e_p est gagnant donc DFS(m, e_p, p) renvoie VRAI.
e_p est un voisin de e_{p-1} donc DFS$(m, e_{p-1}, p-1)$ va renvoyer VRAI, car au moins un de ses suivants, e_p ou un autre élément de F, sera tel que
On prouve ainsi, par récurrence descendante, que DFS(m, e_k, k) renvoie VRAI pour tout $k \in \{0, \dots, p\}$ donc, en particulier, DFS$(m, e_0, 0)$ renvoie VRAI.
Réciproque Si DFS$(m, e_0, 0)$ renvoie VRAI alors soit e_0 est gagnant,
soit il existe un état e_1 suivant de e_1 tel que DFS$(m, e_1, 1)$ renvoie VRAI.
Tant qu'on n'a pas trouvé un état gagnant, on construit ainsi des états e_k tels que DFS(m, e_k, k) renvoie VRAI et $e_k \in s(e_{k-1})$.
La séquence doit s'arrêter car, si on parvient à e_m, DFS(m, e_m, m) renvoie VRAI seulement si e_m est gagnant. Il existe donc une solution de profondeur m au plus.

Question 7
Test d'une liste On commence par une fonction de test d'une fonction booléenne sur une liste qui renvoie `true` si et seulement si il existe au moins un élément de la liste qui vérifie la condition.

```
let rec tester f liste =
  match liste with
  |[] -> false
  |t::q -> (f t) || (tester f q);;
```

Cette fonction existe : `List.exists`.
Parcours limité On suit alors l'algorithme proposé

```
let rec dfs m e p =
  if p > m
  then false
  else if final e
       then true
       else tester (fun x -> dfs m x (p+1)) (suivants e);;
```

Recherche du niveau Pour trouver la profondeur minimale on teste par valeur de m croissante

```
let ids () =
  let rec aux m =
    if dfs m initial 0
    then m
    else aux (m+1) in
  aux 0;;
```

Question 8
Si une solution existe on note m la profondeur optimale d'une solution.
L'algorithme `ids` va tester `dfs k initial 0` pour $k < m$ qui renvoie Faux d'après la question 6
On teste ensuite `dfs m initial 0` , la question 6 indique que la fonction `dfs` renvoie VRAI donc `ids` renverra m : la valeur trouvée est donc bien optimale.

Question 9

Un état à chaque profondeur Quand il y a exactement un état à chaque profondeur p, le parcours en largeur n'aura qu'un élément dans A (et dans B) à chaque passage, donc une complexité spatiale en $\mathcal{O}(1)$. Si on note m la profondeur optimale, le calcul de B (et de A) se fait en temps $\mathcal{O}(m)$. Le parcours en profondeur n'utilise aussi que des liste de taille 1.
Cependant il faut ajouter la pile des appels récursif, en $\mathcal{O}(m)$.
On explore aux profondeurs 1, puis 2, puis etc. jusqu'à arriver à m.
Chaque exploration demande un temps linéaire en la profondeur souhaitée, soit une complexité totale en $\mathcal{O}\big(m^2\big)$.

2^p états à la profondeur p S'il y a exactement 2^p états à la profondeur p, le parcours en largeur aura besoin de stocker les 2^p à chaque étape, soit une complexité spatiale en $\mathcal{O}\big(2^m\big)$ pour la dernière étape.
Le temps de calcul sera également un $\mathcal{O}\big(2^m\big)$. Le parcours en profondeur n'utilisera lui que la taille de sa pile d'appels, en $\mathcal{O}(m)$.
Le temps de calcul sera ici aussi un $\mathcal{O}\big(2^m\big)$.

3 Parcours en profondeur avec horizon

Question 10
Pour initialiser un minimum, on lui donne une valeur infinie, c'est-à-dire `let mini = ref max_int` pour les entiers. Comme il n'y a pas de `return` en Caml, on l=peut le remplacer par une erreur que l'on rattraperait ou, ici, utiliser la condition de sortie de la boucle.

```
let idastar () =
   (* Initialisation de min *)
   let mini = ref max\_int in
   (* La fonction DFS* *)
   let rec dfsstar m e p =
      let c = p + (h e) in
      if c > m
      then begin if c < !mini
                 then mini := c;
                 false end
      else begin if final e
                 then true
                 else tester (fun x-> dfsstar m x (p+1))
                             (suivants e) end           in
   (* On suit l'énoncé, en incluant une réponse entière *)
   let m = ref (h initial) in
   let reponse = ref (-1) in
   while !m < max_int do
      mini := max_int;
      if dfsstar !m initial 0
      then begin reponse := !m;
                 m := max_int end
      else m := !mini done;
   !reponse;;
```

Question 11
Pour le jeu (1), un état q accessible à partir de p par un chemin de longueur k vérifie $q \leqslant 2^k.p$.
Pour tout p il existe un entier k tel que $2^{k-1}.p < t \leqslant 2^k.p$ où t est l'objectif : il faut au moins jouer encore k coups à partir de p avant de trouver une solution.
$h : p \mapsto \lceil \log_2(t/p) \rceil$ est donc une fonction admissible.

Question 12
Preuve de DFS* Si $(e_0, \ldots, e_p)$ est une solution avec $p \leqslant m$.
Pour tout k on doit avoir $h(e_k) \leqslant p - k$ car la distance de e_k à F est au plus $p - k$.
On a alors $e_p \in F$ et $p + h(e_p) = p \leqslant m$ donc DFS*(m, e_p, p) renvoie VRAI.
$p-1+h(e_{p-1}) \leqslant p-1+1 \leqslant m$ et e_p est un voisin de e_{p-1} donc DFS*$(m, e_{p-1}, p-1)$ renvoie VRAI.
On prouve ainsi, par récurrence descendante, que DFS*(m, e_k, k) renvoie VRAI pour tout $k \in \{0, \ldots, p\}$ donc, en particulier, DFS*$(m, e_0, 0)$ renvoie VRAI.

La réciproque se démontre comme dans l'exercice 6.

Preuve de `idastar` Une solution doit avoir une profondeur au moins $h(e_0)$: c'est la première tentative faite.
Si DFS*$(m, e_0, 0)$ a renvoyé FAUX alors, lors du calcul on a ramené min à la plus petite valeur pour laquelle il pourrait exister une solution ayant cette profondeur. Cela démontre qu'il n'y a pas de solution de profondeur $k < min$.
On essaye alors avec DFS*$(min, e_0, 0)$.
Dès que DFS*$(m, e_0, 0)$ renvoie VRAI, m est bien la profondeur optimale.

4 Application au jeu de taquin

Question 13
Chaque état est une permutation des 16 éléments (case vide comprise) de l'état final : il y en a donc 16!. Un entier de $\{0, 1, 2, \ldots, 15\}$ est représentable avec 4 bits donc chaque état est représentable avec 64 bits, 8 octets. L'ensemble des états demandera donc 16!.8 octets. Comme on a $16! > \left(\frac{16}{e}\right)^{16} > 5^{16}$, l'ensemble des états demande plus de $8.5^{16} > 10^{12}$ octets.
Chaque état admet au moins deux voisins donc, à la profondeur 49, on a un ensemble de voisins atteints de l'ordre de 2^{49}, qui est supérieur à 16!. La taille ci-dessus est un ordre de grandeur atteint par la liste A.
On atteint une taille supérieure à un téra-octet, non réaliste.

Question 14
Dans l'état final, l'entier $k \in \{0, 1, \ldots, 14\}$ se trouve à la ligne `k/4` et à la colonne `k mod 4`. Chaque mouvement modifie soit la ligne soit la colonne d'une unité donc le nombre d'étapes pour remettre en place l'entier k est au moins $|e_k^i - k/4| + |e_k^j - (k \text{mod} 4)|$.
Ainsi $h(e)$ minore le nombre de coups à jouer à partir de e.

Question 15
On modifie la valeur de `h` en ajoutant la valeur pour la nouvelle position `(li, lj)` et en enlevant la valeur pour l'ancienne position `(i, j)` devenue vide.

```
let move i j =
  let k = grid.(i).(j) in
  grid.(!li).(!lj) <- k;
  h := !h - abs (i - k/4) - abs (j - k mod 4)
          + abs (!li - k/4) + abs (!lj - k mod 4);
  li := i;
  lj := j;;
```

Question 16
On a besoin de tester si une direction est possible ; on peut factoriser ce test.
Lors de la définition de la fonction la variable globale `reponse` doit exister : on doit donner une définition **avant** les fonctions.

```
let solution = ref[];;
```

On devra ré-initialiser cette variable avant chaque appel de la fonction finale. Si on la définit dans la fonction elle devient locale alors que les fonctions appelées utilisent une fonction globale.

```
let tente_gauche () =
  if !lj < 3 && (List.hd !solution) <> Droite
  then begin
         gauche ();
         solution := Gauche::!solution;
         true
       end
  else false ;;
```

Question 17
La position finale est reconnue par le fait que la valeur de `!h` est zero.
Il ne faut pas oublier de revenir à la position initiale si un mouvement était possible mais n'aboutit pas ; on écrit donc une fonction d'annulation.

```
let annule ()
   match List.hd !solution with
   |Gauche -> droite ()
   |Droite -> gauche ()
   |Haut -> bas ()
   |Bas -> haut ();
   solution := List.tl !solution;;
```

Comme on n'a pas de liste de voisins, il faut tester les 4 mouvements. Cela occasionne deux tests : "*le mouvement est-il possible ?* et, s'il l'est, "*que renvoie dfs ?*". Pour ne pas répéter les conditionnelles, on utilise la fonction `tester` qui porte sur les 4 fonctions `tente_dir`.

```
let tests = [tente_gauche;tente_droite;tente_bas;tente_haut];;
```

La fonction qui sera testée dépend des fonctions de la liste ci-dessus.

```
let mini = ref max_int;;
let rec dfs m p =
   let direction f =
      if f()
      then begin if dfs m p
                 then true
                 else (annule(); false) end
      else false in
  let c = p + ! h in
  if c > m
  then (if c < !mini
        then mini := c;
        false)
  else if !h = 0
       then true
       else tester direction tests;;
```

Question 18
On commence par l'initialisation de `h` en fonction de la grille initiale.

```
let h = ref 0;
for i = 0 to 3 do
  for j = 0 to 3 do
    if i <> !li || j <> !lj
    then let k = grid.(i).(j) in
        h := !h + abs (i - (k/4)) + abs (j - (k mod 4)) done
           done;;
```

On peut alors écrire la fonction

```
let taquin () =
   let m = ref !h in
   let reponse = ref (-1) in
   while !m <> max_int do
      mini := max_int;
      if dfs !m  0
      then begin reponse := !m;
                 m := max_int end
      else m := !mini done;
   List.rev !solution;;
```

Devoir H

DS4 : Centrale 2015

1 Graphes d'intervalles

On considère le problème concret suivant : des cours doivent avoir lieu dans un intervalle de temps précis (de 8h à 9h55, ...), et on cherche à attribuer une salle à chaque cours. On souhaite qu'à tout moment une salle ne puisse être attribuée à deux cours différents et on aimerait utiliser le plus petit nombre de salles possibles.
Ce problème d'allocation de ressources (ici les salles) en fonction de besoins fixes (ici les horaires des cours) intervient dans de nombreuses situations très diverses (allocation de pistes d'atterrissage aux avions, répartition de la charge de travail sur plusieurs machines, ...).

1.1 Représentation du problème

On modélise le problème ainsi :

- chaque besoin est représenté par un segment $[a; b]$ où $a, b \in \mathbb{N}$ et $a \leqslant b$.
- deux besoins I et J sont en conflit quand $I \cap J \neq \emptyset$.

La donnée du problème est une suite finie $(I_0, \ldots, I_{n-1})$ de n segments où $n \in \mathbb{N}$.

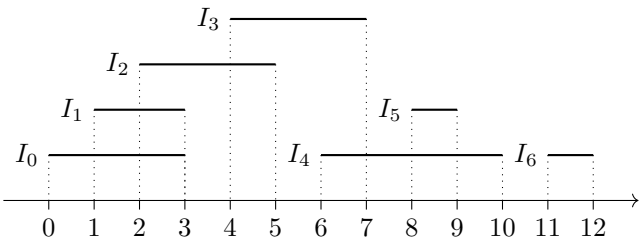


Figure H.1 – Problème (a)

On représente un segment en Caml par un couple d'entiers, la donnée du problème est une valeur du type `(int*int) array`. Le problème (a) de la figure H.1 est représenté par le tableau

```
[|(0,3); (1,3); (2,5); (4,7); (6,10); (8,9); (11,12)|]
```

Question 1
Écrire une fonction `conflit` *telle que* `conflit I J` *renvoie* `true` *si et seulement si I et J sont en conflit.*

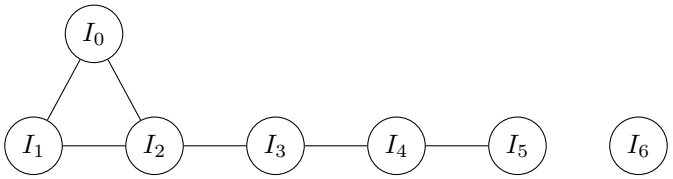
```
conflit : int * int -> int * int -> bool
```

1.2 Graphe d'intervalles

À toute suite finie de segments, $\overline{I} = (I_0, \ldots, I_{n-1})$ on associe un graphe non orienté, le **graphe d'intervalles** $G(\overline{I}) = (S, A)$

- dont les sommets sont les segments I_k, $0 \leqslant k < n$,
- où, pour $i, j \in \{0, \ldots, n-1\}$ avec $i \neq j$, (I_i, I_j) et (I_j, I_i) sont des arêtes (on dit que I_i et I_j sont reliés) si et seulement si I_i et I_j sont en conflit.

Le graphe d'intervalles qui correspond au problème a admet la représentation graphique suivante.



Question 2
Donner une représentation graphique du graphe d'intervalles pour le problème (b) de la figure H.2.

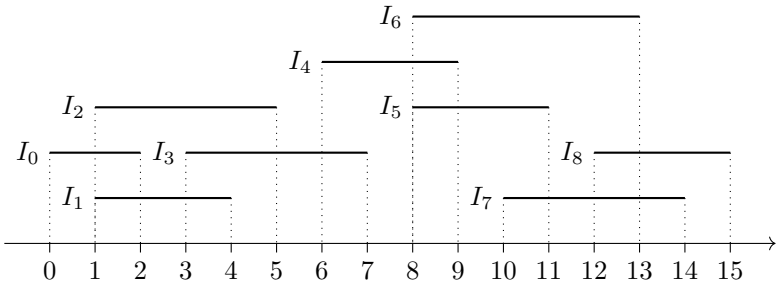


Figure H.2 – Problème (b)

Un graphe est implémenté par le tableau `g` de listes d'adjacence, de type `int list array`, où `g.(i)` est une liste contenant les entiers `j` tels que I_j est relié à I_i..
Par exemple le problème (a) donne un graphe implémenté par

```
[|[1; 2; 3] ; [0; 2; 3] ; [0; 1; 3; 4] ; [0; 1; 2] ; [2]|]
```

Question 3
Écrire une fonction `construit_graphe` *qui, étant donné un problème $\overline{I}$, renvoie la représentation de $G(\overline{I})$ où les intervalles sont numérotés par leur ordre dans le tableau.*

```
construit_graphe : int * int array -> int list array
```

1.3 Coloration

Une **coloration** d'un graphe non orienté $G = (S, A)$ avec $S = (x_0, x_1, \ldots, x_{n-1})$ est une suite $(c_0, c_1, \ldots, c_{n-1})$ telle que $c_i \neq c_j$ pour toute arête (x_i, x_j).
Par exemple $(0, 1, 2, 3, 0, 1, 0)$ est une coloration du graphe du problème (a).
L'entier c_i est appelé la couleur du sommet x_i et la condition se traduit ainsi : deux sommets reliés ont des couleurs distinctes. Dorénavant, le terme couleur sera synonyme d'entier naturel.

Une coloration qui utilise le plus petit nombre de couleurs distinctes possibles, est dite **optimale**. Ce nombre minimum de couleurs est noté $\chi(G)$, c'est le **nombre chromatique** de G.
En associant une salle à chaque couleur, on peut répondre au problème initial à l'aide d'une coloration de son graphe d'intervalles associé.

Question 4
Déterminer des colorations optimales pour les graphes d'intervalles associés aux deux problèmes (a) et (b). On attribuera à chaque fois la couleur 0 *à l'intervalle* I_0

Question 5
Écrire une fonction `appartient` *telle que l'appel* `appartient l x` *renvoie* `true` *si et seulement si l'entier* x *est présent dans la liste* `l`*.*

```
appartient : int list -> int -> bool
```

Question 6
Écrire une fonction `plus_petit_absent` *telle que l'appel* `plus_petit_absent l` *renvoie le plus petit entier naturel non présent dans* `l`*.*

```
plus_petit_absent : int list -> int
```

On considère ici une coloration progressive des sommets d'un graphe. Pour cela, une coloration partielle est un tableau d'entiers `col` tel que `col.(i)` contient la couleur de i s'il est coloré et -1 sinon, ce qui ne pose pas de problème car les couleurs sont toujours positives.

Question 7
Écrire une fonction `couleurs_voisins` *telle que l'appel* `couleurs_voisins g col i` *renvoie la liste des couleurs des voisins colorés du sommet d'indice* i *dans le graphe décrit par* `g` *où le tableau* `col` *décrit une coloration partielle.*

```
couleurs_voisins :
   int list array -> int array -> int -> int list
```

Question 8
En déduire, une fonction `couleur_disponible` *telle que l'appel* `couleur_disponible g col i` *renvoie la plus petite couleur pouvant être attribuée au sommet* i *afin qu'il n'ait la couleur d'aucun de ses voisins dans le graphe décrit par* `g`*.*

```
couleur_disponible : int list array-> int array-> int -> int
```

1.4 Cliques

Un sous-ensemble $C \subset S$ est une **clique** du graphe $G = (S, A)$ lorsque deux sommets de C sont toujours reliés : $\forall x, y \in C,\ x \neq y \Rightarrow (x, y) \in A$.
Le cardinal de la plus grande (en nombre d'éléments) clique de G est notée $\omega(G)$.

Question 9
Déterminer $\chi(G)$ *et* $\omega(G)$ *lorsque* G *ne possède pas d'arête (c'est à dire* $A = \emptyset$*) puis lorsque* G *est un graphe complet (pour tous* $u, v \in S$ *distincts,* $(u, v) \in A$*).*

Question 10
Comparer $\chi(G)$ et $\omega(G)$ pour un graphe G quelconque.

Question 11
Écrire une fonction `est_clique` *telle que* `est_clique g xs` *renvoie* `true` *si et seulement si la liste* `xs` *est une liste d'indices de sommets formant une clique dans le graphe décrit par* `g`.

```
est_clique : int list array -> int list -> bool
```

2 Algorithme glouton pour la coloration

Étant donnée une liste de segments $\overline{I} = (I_0, \ldots, I_{n-1})$ de longueur $n \geqslant 1$, on se propose de déterminer une coloration optimale de son graphe d'intervalles associé.
On appelle coloration de $\overline{I}$ une coloration de $G(\overline{I})$.
On suppose dans cette partie que les segments $I_k = [a_k, b_k]$, pour $k \in \{0, \ldots, n-1\}$, sont énumérés dans l'ordre croissant de leur extrémités gauches, c'est-à-dire que $a_0 \leqslant a_1 \leqslant \cdots \leqslant a_{n-1}$.
On propose l'algorithme suivant :

```
Pour k variant de 0 à n-1
    colorer l'intervalle Ik
        avec la plus petite couleur non encore utilisée
        dans la coloration des intervalles Ij, 0 <= j < k,
        qui ont une intersection non vide avec Ik.
```

Ainsi, l'intervalle I_0 est toujours coloré avec la couleur 0, l'intervalle I_1 reçoit la couleur 0 si $I_0 \cap I_1 = \emptyset$, et la couleur 1 sinon, etc.

Question 12 — Un exemple
Déterminer la coloration renvoyée par l'algorithme pour le problème (b).

Question 13
Écrire une fonction `coloration` *telle que l'appel* `coloration seg`*, où* `seg` *est un tableau contenant des segments triés par ordre croissant de leurs extrémités gauches, renvoie la coloration obtenue avec l'algorithme ci-dessus.*

```
coloration : (int * int) array -> int list array -> int array
```

2.1 Analyse

On se propose maintenant de démontrer que l'algorithme ci-dessus fournit une coloration optimale de l'ensemble de segments. Soit k un entier entre 0 et $n-1$. On suppose qu'à la k-ième étape de l'algorithme, le segment I_k reçoit la couleur c.

Question 14
L'extrémité gauche du segment I_k appartient à un certain nombre de segments parmi $I_0, \ldots, I_{k-1}$. Combien au moins ?

Question 15
Prouver que l'ensemble constitué de I_k et de ses voisins d'indice inférieur à k constitue une clique de taille au moins $c+1$ dans le graphe d'intervalles associé.

Question 16
En déduire que le nombre de couleurs nécessaires à une coloration de l'ensemble des segments est au moins égal à $c+1$.

Question 17
Conclure.

Question 18 — Complexité
Déterminer la complexité de la fonction `coloration` *en fonction du nombre m d'arêtes du graphe d'intervalles associé à la liste $\overline{I}$.*

3 Graphes munis d'un ordre d'élimination parfait

Si $S' \subset S$ est un ensemble de sommets de G, le **sous-graphe de G induit par S'** est le graphe $G' = (S', A')$ où $A' \subset A$ est l'ensemble des arêtes de G dont les extrémités appartiennent à S'.
Soient $G = (S, A)$ un graphe et $(x_0, \dots, x_{n-1})$ une énumération des sommets de G.
Pour tout $i \in \{0, \dots, n-1\}$, on note $G_i = (S_i, A_i)$ sous-graphe de G induit par $S_i = \{x_0, \dots, x_i\}$.
Une énumération $(x_0, \dots, x_{n-1})$ des sommets de G est appelée un **ordre d'élimination parfait** si pour tout $i \in \{0, \dots, n-1\}$, les voisins de x_i d'indices inférieurs à i forment une clique.

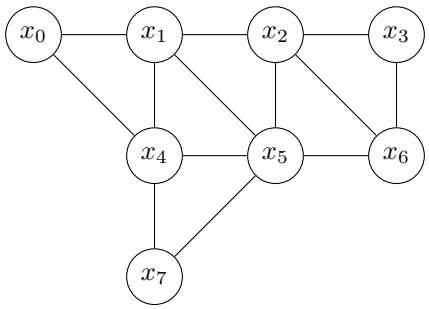


Figure H.3 – Graphe G_0

Question 19 — Un exemple
Déterminer un ordre d'élimination parfait pour le graphe G_0 de la figure H.3.

3.1 Vérification

Question 20
Écrire une fonction `voisins_inferieurs` *telle que* `voisins_inferieurs g x` *renvoie la liste des voisins du sommet d'indice x dont l'indice est strictement inférieur à x.*

```
voisins_inferieurs : int list array -> int -> int list
```

Question 21
Écrire une fonction `est_ordre_parfait` *telle que* `est_ordre_parfait g` *renvoie* `true` *si et seulement si l'énumération associée au graphe représenté par* `g` *est un ordre d'élimination parfait.*

```
est_ordre_parfait : int list array -> bool
```

Question 22 — Cas des graphes d'intervalles
Montrer que l'énumération des segments $(I_0, \dots, I_{n-1})$ obtenue en les triant par leurs extrémités gauches en ordre croissant est un ordre d'élimination parfait de leur graphe d'intervalles.

3.2 Coloration

On considère un graphe dont $(x_0, \dots, x_{n-1})$ est une énumération des sommets. On colore ce graphe à l'aide l'algorithme suivant :

```
Pour k variant de 0 à n-1
    colorer le sommet xk
    avec la plus petite couleur non encore utilisée
    par un de ses voisins déjà colorés
```

Question 23
Appliquer cet algorithme de coloration au graphe G_0 de la figure H.3 muni
1. *de l'ordre $(x_0, \dots, x_7)$,*
2. *d'un ordre d'élimination parfait.*

Question 24
Soit $(c_0, \dots, c_{n-1})$ la coloration obtenue par cet algorithme pour un graphe G dont l'énumération des sommets est un ordre d'élimination parfait.
1. *Montrer que pour tout $i \in \{0, \dots, n-1\}$, on a $\chi(G) \geqslant 1 + c_i$.*
2. *En déduire que l'algorithme de coloration renvoie une coloration optimale.*

4 Ordre d'élimination parfait pour un graphe cordal

Un graphe G est dit **cordal** lorsque pour tout cycle $C = (v_0, v_1, \dots, v_{n-1}, v_0)$ de G de longueur $n \geqslant 4$, il existe i, j distincts entre 0 et $n-1$ tels que les sommets v_i et v_j soient reliés dans le graphe G mais non successifs dans le cycle. Une telle arête $\{v_i, v_j\}$ est appelée une **corde** du cycle C. Autrement dit, le graphe G est cordal lorsque tout cycle de G de longueur supérieure ou égale à 4 possède une corde.

4.1 Cycles de longueur 4 dans un graphe d'intervalles

Soit G un graphe d'intervalles.
Dans cette question, on se propose de démontrer par l'absurde que tout cycle de longueur 4 de G possède une corde. On suppose à cet effet que G contient un 4-cycle sans corde.
On dispose donc de quatre segments I_0, I_1, I_2, I_3 tels que

$$I_0 \cap I_1 \neq \emptyset,\ I_1 \cap I_2 \neq \emptyset,\ I_2 \cap I_3 \neq \emptyset,\ I_3 \cap I_0 \neq \emptyset,\ I_0 \cap I_2 = \emptyset,\ I_1 \cap I_3 = \emptyset$$

On supposera pour simplifier que les extrémités des segments sont toutes distinctes.

Question 25
Montrer qu'aucun des segments I_k, $k = 0, 1, 2, 3$, n'est inclus dans un autre de ces segments.

Question 26
On a donc par exemple $\min I_0 < \min I_1 < \max I_0 < \max I_1$.
Montrer que $\min I_1 < \min I_2 < \max I_1 < \max I_2$ *et de même pour I_2 et I_3.*

Question 27
Conclure à une contradiction.

Question 28 — Cordalité des graphes d'intervalles
Montrer plus généralement que tout graphe d'intervalles est cordal.

4.2 Une enquête policière

Six personnes sont entrées dans la bibliothèque le jour où un livre rare y a été volé. Chacune d'entre elles est entrée une seule fois dans la bibliothèque, y est restée un certain temps, puis elle en est sortie. Si deux personnes étaient ensemble dans la bibliothèque à un instant donné, alors au moins l'une des deux a vu l'autre. A l'issue de l'enquête, les témoignages recueillis sont les suivants : Albert dit qu'il a vu Bernard et Edouard dans la bibliothèque. Bernard a vu Albert et Isabelle. Charlotte affirme avoir vu Didier et Isabelle. Didier dit qu'il a vu Albert et Isabelle. Edouard certifie avoir vu Bernard et Charlotte. Isabelle dit avoir vu Charlotte et Edouard.

Question 29
Seul le coupable a menti. Qui est-il ?

4.3 Sommets simpliciaux

Un sommet v d'un graphe G est **simplicial** si l'ensemble des voisins de v dans G est une clique.
On remarque que, si un graphe G admet admet un ordre d'élimination parfait $(x_0, x_1, \ldots, x_{n-1})$; alors x_{n-1} est simplicial car tous ses voisins sont d'indices inférieurs à $n-1$ et ils forment donc une clique.

On représente en Caml le sous-graphe induit de G induit par S' par le couple `(g, sg)` de type `int list array * bool array` où `g` est une description du graphe G et `sg` est un tableau de booléens de taille n tel que `sg.(i)` vaut `true` si et seulement si i appartient à S'.

Question 30
Écrire une fonction `simplicial` *telle que l'appel* `simplicial (g, sg) k`*, où le sommet d'indice k est supposé appartenir au sous-graphe induit G' décrit par* `(g, sg)`*, renvoie* `true` *si le sommet d'indice k est simplicial dans H et* `false` *sinon. Déterminer sa complexité.*

```
simplicial : (int list array * bool array) -> int -> bool
```

Question 31
Écrire une fonction `trouver_simplicial` *telle que l'appel* `trouver_simplicial (g, sg)` *renvoie, s'il en existe, un sommet simplicial du sous-graphe induit décrit par* `(g, sg)`.
Déterminer la complexité de cette fonction.

```
trouver_simplicial : (int list array * bool array) -> int
```

Question 32
G admet un ordre d'élimination parfait $(x_0, x_1, \ldots, x_{n-1})$ et x_i est simplicial. Montrer que le graphe G' induit par $\{x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}\}$ admet un ordre d'élimination parfait.

Question 33
Écrire une fonction `ordre_parfait` *telle que l'appel* `ordre_parfait g` *renvoie un ordre d'élimination parfait du graphe décrit par* `g`*, s'il en existe un.*
Déterminer la complexité de cette fonction.

```
ordre_parfait : int list array -> int list
```

4.4 Coupures minimales dans un graphe cordal

Étant donné un graphe G on appelle **coupure** de G tout ensemble $C \subset S$ de sommets de G,tel que certains sommets reliés par un chemin dans le graphe G ne le sont plus dans le sous-graphe H de G induit par $S \setminus C$.
On se donne dans cette question un graphe cordal $G = (S, A)$.
Soient C une coupure de G de cardinal minimal et H le sous-graphe de G induit par $S \setminus C$.

a et b sont deux sommets de G déconnectés par la coupure, S_1 et S_2 sont les composantes connexes de a et b dans le graphe H et G_1 et G_2 sont les graphes induits par S_1 et S_2.

Question 34
Montrer que tout point de C est voisin dans G d'un sommet de S_1 et d'un sommet de S_2.

x et y sont deux sommets distincts de C (s'il en existe).

Question 35
Montrer qu'il existe un chemin $P_1 = (x, a_1, \ldots, a_p, y)$ (resp. $P_2 = (x, b_1, \ldots, b_q, y)$) dont tous les sommets hormis x et y appartiennent à S_1 (resp. à S_2).

Question 36
On prend deux tels chemins P_1 et P_2 de longueur minimale. En considérant un cycle formé à partir des chemins P_1 et P_2, montrer que x et y sont voisins dans le graphe G.

Question 37
Montrer que C est une clique du graphe G.

4.5 Sommets simpliciaux dans un graphe cordal

On se propose de montrer la propriété $\mathcal{P}(G)$: *tout graphe cordal G possède un sommet simplicial, et même deux sommets simpliciaux non voisins si G n'est pas complet.*
On se donne dans toute la suite un graphe cordal G.

Question 38
Montrer que si G est complet alors tous ses sommets sont simpliciaux.

Question 39
Montrer que la propriété $\mathcal{P}(G)$ est vérifiée si G possède 1, 2 ou 3 sommets.

On suppose maintenant que G n'est pas complet, possède au moins trois sommets et que la propriété $\mathcal{P}(G)$ est vérifiée pour tous les graphes cordaux G' ayant strictement moins de sommets que G.
On choisit une une coupure C de G de cardinal minimal. On a joute aux définitions de la partie 4.4, H_1 (resp H_2) est le sous-graphe de G induit par $S_1 \cup C$ (resp. par $S_2 \cup C$).

Question 40
Justifier que le graphe H_1 est cordal.

Question 41
On suppose que H_1 est complet. Montrer que S_1 contient un sommet simplicial de G.

Question 42
On suppose que H_1 n'est pas complet. Montrer que $S_1 \cup C$ contient deux sommets simpliciaux non voisins du graphe H_1. Montrer que au moins l'un de ces deux sommets est dans S_1 et que ce sommet est un sommet simplicial de G.

Question 43
Montrer que la propriété $\mathcal{P}(G)$ est vérifiée.

Question 44
Montrer que tout graphe cordal possède un ordre d'élimination

5 Solutions

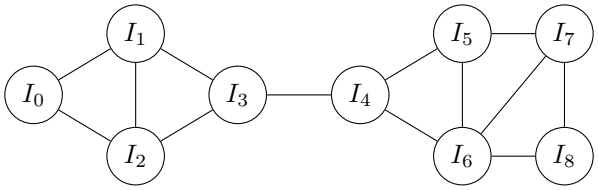
Solution de l'exercice 1 - Il y a conflit entre $[a;b]$ et $[c;d]$ lorsqu'il existe $x \in [a;b] \cap [c;d]$; cela implique $a \leqslant x \leqslant d$ et $c \leqslant x \leqslant b$.
Inversement si on a $a \leqslant d$ et $c \leqslant b$ alors,

- soit $c \leqslant a$, alors $a \in [c;d]$ car on a $a \leqslant d$,
- soit $c > a$ donc $a < c \leqslant b$ d'où $c \in [a;b]$.

Il y a donc conflit si et seulement si $a \leqslant d$ et $c \leqslant b$.

```
let conflit (a, b) (c, d) = (c <= b) && (a <= d);;
```

Solution de l'exercice 2 -



Solution de l'exercice 3 - Il suffit de balayer tous les couples (i,j) avec $0 \leqslant i < j \leqslant n-1$.

```
let construit_graphe seg =
   let n = Array.length seg in
   let g = Array.make n [] in
   for i = 0 to (n-2) do
      for j = (i+1) to (n-1) do
         if conflit seg.(i) seg.(j)
         then begin g.(i) <- j::g.(i);
                    g.(j) <- i::g.(j) end done done;
   g;;
```

Solution de l'exercice 4 - Dès qu'il y a conflit il faut au moins deux couleurs. S'il existe un cycle de 3 sommets dans le graphe alors il faut au moins 3 couleurs.
Les deux graphes demandent donc au moins 3 couleurs.
3 couleurs suffisent avec, par exemple,

- la coloration (0, 1, 2, 0, 1, 0, 0) pour le problème (a) et
- la coloration (0, 1, 2, 0, 1, 0, 2, 1, 0) pour le problème (b).

Solution de l'exercice 5 - Un grand classique ...

```
let rec appartient liste x =
   match liste with
   |[] -> false
   |t::q -> (t = x) || (appartient q x);;
```

Solution de l'exercice 6 -

```
let plus_petit_absent liste =
   let rec test i =
      if appartient liste i then test (i+1) else i
    in test 0;;
```

Solution de l'exercice 7 -

```
let couleurs_voisins g col i =
   let rec prendreCol liste =
      match liste with
      |[] -> []
      |t::q -> let color = col.(t) in
               if color <> -1
               then color :: (prendreCol q)
               else prendreCol q
   in prendreCol g.(i);;
```

Solution de l'exercice 8 - La question est un peu vide ...

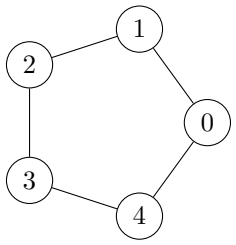
```
let couleur_disponible g col i =
   plus_petit_absent (couleurs_voisins g col i);;
```

Solution de l'exercice 9 -

1. Si G ne possède pas d'arête alors il ne peut exister de clique qui possède plus de un sommet donc $\omega(G) = 1$.
   De plus on peut colorer tous les sommets avec la même couleur donc $\chi(G) = 1$.
2. Si G est complet alors il ne peut pas exister deux sommets de la même couleur donc $\chi(G) = n$.
   Le graphe est alors une clique de n sommets donc $\omega(G) = n$.

Solution de l'exercice 10 - Dans une clique tous les sommets sont reliés entre eux donc doivent avoir des couleurs distinctes ; ainsi le nombre de couleur est minoré par le cardinal des cliques. Le nombre minimal de couleurs est donc minoré par la taille maximale des cliques : $\chi(G) \geqslant \omega(G)$.

Remarque Cette inégalité peut être stricte. Le graphe suivant ne contient pas de clique de cardinal 3 mais toute coloration demande au moins 3 couleurs.



Solution de l'exercice 11 -

```
let rec longueur liste =
  match liste with
  |[] -> 0
  |t::q -> 1 + longueur_liste q;;

let est_clique g xs =
   let p = longueur xs in
   let rec nb_voisins liste =
      match liste with
      |[] -> 0
      |t::q -> if appartient xs t
               then 1 + nb_voisins q
               else nb_voisins q in
   let rec tester liste =
      match liste with
      |[] -> true
      |t::q -> (nb_voisins g.(t) = p-1) && tester q in
  tester xs;;
```

Solution de l'exercice 12 -
La numérotation des intervalles vérifie la propriété de croissance de l'extrémité gauche.
On aboutit à la coloration (0, 1, 2, 0, 1, 0, 2, 1, 0).

Solution de l'exercice 13 - On utilise les fonctions de la première partie.

```
let coloration seg =
   let g = construit_graphe seg in
   let n = Array.length seg in
   let col = Array.make n (-1) in
   for i = 0 to (n-1) do
      col.(i) <- couleur_disponible g col i done;
   col;;
```

Solution de l'exercice 14 - Si I_k a reçu la couleur c alors il est lié avec des sommets qui on les couleurs 0 à $c-1$. Les seuls sommets colorés lors de la coloration de I_k sont les sommets I_0 à I_{k-1}. Ainsi I_k est lié avec au moins c intervalles d'indices inférieurs à k. Or $I_k = [a_k; b_k]$ et $I_p = [a_p; b_p]$ sont liés si on a $a_k \in I_p$ ou $a_p \in I_k$. Pour $p \leqslant k$ on a $a_p \leqslant a_k$ donc les intervalles I_p et I_k sont en conflit si et seulement si $a_k \in I_p$.
Ainsi a_k appartient à au moins c intervalles d'indices inférieurs à k.

Solution de l'exercice 15 - On note C l'ensemble des intervalles I_p tels que $p \leqslant k$ et $I_p \cap I_k \neq \emptyset$.
La question ci-dessus montre que C contient au moins $c+1$ intervalles car il contient I_k en plus de ses voisins.
Soient I_p et I_q appartenant à C. On a $a_k \in I_p$ et $a_k \in I_q$ donc $a_k \in I_p \cap I_q$: $I_p \cap I_q \neq \emptyset$.
On en déduit que (I_p, I_q) est une arête du graphe pour tous sommets I_p, I_q de C : C est une clique de taille au moins $c+1$.

Solution de l'exercice 16 -
La clique ci-dessus implique $\omega(G) \geqslant c+1$ donc, d'après la question 10, $\chi(G) \geqslant c+1$.

Solution de l'exercice 17 - Lors de la coloration d'un sommet par une couleur c, toutes les couleurs de 0 à $c-1$ ont été attribuées à un sommet. Si c_M est la couleur maximale utilisée par l'algorithme, le nombre de couleurs utilisées est donc $c_M + 1$. Ainsi $c_M + 1 \geqslant \chi(G)$.
Lors de la coloration d'un sommet par c_M la question ci-dessus montre qu'on a $\chi(G) \geqslant c_M + 1$.
On en déduit que $\chi(G) = c_M + 1$: la coloration de l'algorithme est optimale.

Solution de l'exercice 18 -

- La fonction `appartient` effectue au plus p comparaisons où p est la longueur de la liste passée en paramètre.
- La fonction `plus_petit_absent` effectue au plus $c+1$ appels à `appartient` où c est le résultat. Comme le nombre de couleurs est majoré par p, longueur de la liste, la complexité est majorée par $A.p.(p+1)$, c'est un $\mathcal{O}(p^2)$
- La fonction `couleurs_voisins` parcourt la liste des voisins d'un sommets, sa complexité est un $\mathcal{O}(m_i)$ où m_i est le nombre de voisins de i, elle produit une liste de taille m_i au plus.
- Ainsi la fonction `couleur_disponible` est de complexité en $\mathcal{O}(m_i^2) = \mathcal{O}(m^2)$.
- Enfin la fonction `coloration` fait appel n fois à la fonction `couleur_disponible` après avoir créé un tableau de taille n où n est le nombre de segments.
  Sa complexité est donc un $\mathcal{O}(n + nm^2)$.

Il semble difficile de donner une complexité ne faisant pas intervenir le nombre d'intervalles.

Solution de l'exercice 19 - L'ordre $(x_0, x_1, x_4, x_5, x_2, x_7, x_6, x_3)$ convient.
En effet les voisins placés avant pour chaque sommet forment les ensembles
$\emptyset$, $\{x_0\}$, $\{x_0, x_1\}$, $\{x_1, x_4\}$, $\{x_1, x_5\}$, $\{x_4, x_5\}$, $\{x_2, x_5\}$ et $\{x_2, x_6\}$
qui sont tous des cliques.

Solution de l'exercice 20 -

```
let voisins_inferieurs g x =
   let rec aux liste =
      match liste with
      |[] -> []
      |t::q when t < x -> t :: (aux q)
      |t::q -> aux q
  in aux g.(x);;
```

Solution de l'exercice 21 -

```
let est_ordre_parfait g =
   let n = Array.length g in
   let rec aux i =
      if i = n
      then true
      else let l = voisins_inferieurs g i in
           est_clique g l && aux (i+1)
  in aux 0;;
```

Solution de l'exercice 22 - L'ensemble des voisins de I_k est l'ensemble des intervalles en conflit avec I_k. On a vu à la question 15 que l'ensemble des voisins de I_k d'ordre inférieur formaient une clique (la question ajoutait I_k mais toute partie d'une clique est une clique).
On a donc bien un ordre d'élimination parfait.

Solution de l'exercice 23 -

Solution de l'exercice 24 - On reprend les arguments des **questions 16 et 17**.

Solution de l'exercice 25 -
Les intervalles jouent des rôles symétriques : supposons qu'on ait I_0 inclus dans I_k.
$I_0 \subset I_2$ est impossible car $I_0 \cap I_2 = \emptyset$.
Si on a $I_0 \subset I_1$ alors $I_0 \cap I_3 \subset I_1 \subset I_3 = \emptyset$ d'où $I_0 \cap I_3 = \emptyset$, ce qui est impossible.
De même $I_0 \subset I_3$ est impossible.

Solution de l'exercice 26 -
On commence par démontrer la propriété admise.
Soient $I = [a;b]$ et $I' = [a';b']$ en conflit.
Pour $x \in I \cap I'$, on a $a \leqslant x$ et $a' \leqslant x$ donc $\max\{a,a'\} \leqslant x$, de même $x \leqslant \min\{b,b'\}$.

- Si $\max\{a,a'\} = a$ et $\min\{b,b'\} = b$ alors $I = [a;b] \subset [a';b'] = I'$.
- Si $\max\{a,a'\} = a$ et $\min\{b,b'\} = b'$ alors $a' \leqslant a \leqslant x \leqslant b' \leqslant b$.
- Si $\max\{a,a'\} = a'$ et $\min\{b,b'\} = b$ $a \leqslant a' \leqslant x \leqslant b \leqslant b'$.
- Si $\max\{a,a'\} = a'$ et $\min\{b,b'\} = b'$ alors $I' = [a';b'] \subset [a;b] = I$.

Si on suppose de plus que les intervalles ne sont pas inclus l'un dans l'autre et que les extrémités sont toutes distinctes il ne reste que les possibilités $a' < a < b' < b$ ou $a < a' < b < b'$.

I_0 et I_1 sont en conflit, on suppose $a_0 < a_1 < b_0 < b_1$: $I_0 \cap I_1 = [a_1;b_0]$.
I_1 et I_2 sont en conflit aussi, si on avait $a_2 < a_1 < b_2 < b_1$ alors on aurait $I_1 \cap I_2 = [a_1;b_2]$ donc $a_1 \in I_2$, comme on avait aussi $a_1 \in I_0$, on aboutit à $a_1 \in I_0 \cap I_2$, ce qui est impossible.
On en déduit qu'on a $a_1 < a_2 < b_1 < b_2$.

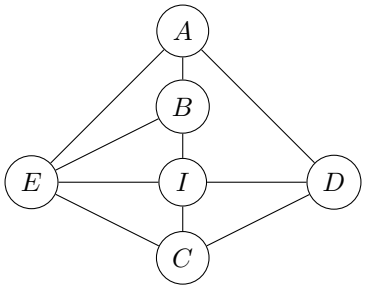
De même, on déduit de I_2 et I_3 en conflit qu'on a $a_2 < a_3 < b_2 < b_3$.

Solution de l'exercice 27 - On déduit de la question précédente qu'on a $a_0 < a_1 < a_2 < a_3$.
Comme I_0 et I_3 sont en conflits, cela impose $a_0 < a_3 < b_0 < b_3$ donc $b_0 \in I_3$.
Or an a aussi $I_0 \cap I_1 = [a_1; b_0]$ donc $b_0 \in I_1$ on aboutit à $I_1 \cap I_3 \neq \emptyset$ ce qui est impossible.

L'autre cas aboutit à une contradiction : il se ramène au premier cas en interchangeant I_0 et I_1 ainsi que I_2 et I_3.

Solution de l'exercice 28 - La démonstration est la même. I_0, I_1, ..., I_{p-1} est cycle : $p \geqslant 4$.
On suppose qu'on a $a_0 < a_1 < b_0 < b_1$: $b_0 \in I_0 \cap I_1$.
Comme ci-dessus, si (I_0, I_2) n'est pas une corde, on a $a_1 < a_2 < b_1 < b_2$.
On peut poursuivre, si (I_k, I_{k+2}) n'est pas une corde, pour tout k tel que $0 \leqslant k \leqslant p-3$, on a $a_i < a_{i+1} < b_i < b_{i+1}$ pour tout i tel que $0 \leqslant i \leqslant p-2$.
Sous ces conditions on en déduit $a_0 < a_{p-1}$ donc, comme I_0 et I_{p-1} sont en conflit, $a_0 < a_{p-1} < b_0 < b_{p-1}$. On conclut $b_0 \in I_1 \cap I_{p-1}$.
Il y a forcement une corde.

Solution de l'exercice 29 - On note A, B, C, D, E, I les intervalles de temps de présence des différents protagonistes.
Les affirmations des personnes, si elles ne mentent pas, la vision (d'au moins) une personne par une autre caractérise les intervalles en conflit.
On aboutit au graphe



On voit qu'il existe des cycle d'ordre 4 sans corde ce qui est impossible.
Les cycles sont (A, D, I, B), (A, D, I, E), (A, D, C, E),
Pour chacun de ces cycles, soit le cycle n'en est pas un, une des arêtes a été inventée soit il manque une corde, une des arêtes a été tue. Dans les deux cas le menteur est une des personnes du cycle.
Le menteur doit donc être A ou D.
Didier peut avoir menti en inventant avoir vu Albert.
Albert peut avoir menti et omis de dire qu'il a vu Isabelle et Charlotte.
On ne peut pas décider du coupable.

Solution de l'exercice 30 - Pour qu'un sommet soit simplicial il faut vérifier si l'ensemble de ses voisins appartenant à H forme une clique dans G', c'est-à-dire une clique dans G.

```
let simplicial (g, sg) k =
   let rec extraire liste  =
      match liste with
      |[] -> []
      |t::q when sg.(t) -> t::(extraire q)
      |t::q -> extraire q in
   est_clique g (extraire g.(k));;
```

Extraire les bons voisins se fait en parcourant la liste des voisins, de longueur p, sa complexité est donc un $\mathcal{O}(p)$.
Tester si un sous-ensemble forme un clique consiste, pour chaque élément, à compter le nombre de ses voisins dans l'ensemble : la complexité est donc un $\mathcal{O}(pn)$.
La complexité totale est donc un $\mathcal{O}\left(n^3\right)$ car on a $p \leqslant n$.

Solution de l'exercice 31 - On choisit de renvoyer -1 s'il n'y a pas de sommet simplicial.

```
let trouver_simplicial (g, sg) =
   let n = Array.length g in
   let simpl = ref (-1) in
   for k = 0 to (n-1) do
      if  sg.(k) && simplicial (g, sg) k
         then simpl := k done;
  !simpl;;
```

On répète n fois l'appel à simplicial : la complexité est un $\mathcal{O}(n^3)$.

Solution de l'exercice 32 - On va prouver que $(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{n-1})$ est un ordre d'élimination parfait.
On note V'_k l'ensemble des voisins de x_k d'indices inférieur à k dans G' et V_k l'ensemble des voisins de x_k d'indices inférieur à k dans G'

- Soit $V_k = V'_k$, dans ce cas V_k forme une clique dans G donc dans G'
- Soit $V_k = V'_k \cup \{x_i\}$, comme V_k est une clique dans G, V'_k en est une aussi donc c'est aussi une clique dans G'.

$(x_0, \dots, x_{i-1}, x_{i+1}, \dots, x_{n-1})$ est bien un ordre d'élimination parfait.

Solution de l'exercice 33 - Si x est un sommet simplicial d'un graphe G et si le sous-graphe G' obtenu en supprimant x admet un ordre d'élimination parfait alors G admet un ordre d'élimination parfait obtenu en ajoutant x à l'ordre d'élimination parfait dans G'.
Inversement si G admet un ordre d'élimination parfait on sait qu'il admet un sommet simplicial et que le graphe induit en enlevant ce sommet admet un ordre d'élimination parfait.
On peut donc construire un ordre d'élimination parfait récursivement :

- si le graphe est vide, son ordre d'élimination parfait est vide
- sinon, on détermine un sommet simplicial et on l'ajoute à un ordre d'élimination parfait du graphe induit en l'ôtant.

```
let ordre_parfait g =
   let n = Array.length g in
   let sg = Array.make n true in
   let rec aux_oep () =
      match trouver_simplicial (g, sg) with
      |-1 -> []
      |k -> sg.(k) <- false;
            k :: (aux_oep ()) in
   List.rev aux_oep ();;
```

On cherche n fois un simplicial : la complexité est un $\mathcal{O}(n^4)$

Solution de l'exercice 34 -
On considère l'ensemble C_1 des sommet de C qui sont voisins d'un sommet de S_1.
Tout chemin $a \to s_1 \to \dots \to s_{p-1} \to b$ contient (au moins) un sommet de C. Le premier sommet de C dans ce chemin est le successeur d'un sommet de S_1 donc appartient à C_1.
Ainsi toute chemin de a vers b contient un sommet de C_1 et C_1 est une coupure contenue dans C.
Par minimalité de C on en déduit qu'on a $C_1 = C$.
Tout sommet de C est voisin d'un sommet de S_1.
De même tout sommet de C est voisin d'un sommet de S_2.

Solution de l'exercice 35 - On considère un voisin de x dans G_1, a', et un voisin de y dans G_1, a''. a' et a'' appartiennent à S_1 qui est une composante connexe du graphe induit donc il existe un chemin de a' à a' dans G_1 que l'on note $(a_1, a_2, \dots, a_p)$.
Ainsi $(x, a_1, a_2, \dots, a_p, y)$ est chemin de G tel que demandé par l'énoncé.
De même il existe un chemin $(y, b_1, b_2, \dots, b_q, x)$ dans G avec $(b_1, b_2, \dots, b_q)$ dans G_2.

Solution de l'exercice 36 - On trouve alors un cycle $(x, a_1, a_2, \ldots, a_p, y, b_1, b_2, \ldots, b_q, x)$ de longueur au moins 4 dans G qui est cordal.
Il admet donc une corde (u, v).

- Si $u \in G_1$ et $v \in G_1$ on écrit $u = a_i$ et $v = a_j$, on suppose $j \geqslant i + 2$.
  Si on remplace $(a_i, a_{i+1}, \ldots, a_j)$ par (a_i, a_j), on obtient un cycle plus court avec les mêmes propriétés, ce qui est impossible.
- De même il est impossible d'avoir $u \in G_2$ et $v \in G_2$.
- La même démonstration montre qu'on ne peut pas avoir $u \in G_1$ ou $u \in G_2$ et $v = x$ ou $v = y$ ou l'inverse.

La seule possibilité est donc $(x, y) \in G$.

Solution de l'exercice 37 - Deux sommets quelconques de C sont voisins dans G donc C est une clique de G.

Solution de l'exercice 38 - Dans un graphe complets tous ses sommets sont voisins donc toute partie est une clique. Ce sera le cas pour les voisins d'un sommet.

Solution de l'exercice 39 -

1. Si G possède un sommet, il est complet et son seul sommet est simplicial.
2. Si G possède deux sommets, soit il est complet donc admet deux sommets simpliciaux soit il a deux sommets isolés qui sont simpliciaux et non adjacents.
3. Si G possède deux sommets il admet de 0 à 3 arêtes.
   (a) S'il n'admet aucune arête les 3 sommets sont simpliciaux et non adjacents.
   (b) S'il admet 1 arête alors il y a un sommet isolé donc simplicial qui n'est pas voisins aux deux autres qui sont aussi simpliciaux.
   (c) S'il admet 2 arêtes, le graphe est linéaire : a_1 est voisin de a_2 et a_2 est voisin de a_3. a_1 et a_3 sont alors simpliciaux et non voisins.
   (d) S'il admet 3 arêtes, G est complet : tous les sommets sont simpliciaux.

Solution de l'exercice 40 -

Solution de l'exercice 41 - Si H_1 est complet alors tout sommet de S_1 est simplicial pour H_1. Par exemple, a l'est. Mais les voisins de a dans G sont tous soit dans S_1 soit dans C. a est donc simplicial dans G.

Solution de l'exercice 42 - Si H_1 n'est pas complet, par hypothèse de récurrence $\mathcal{P}(H_1)$ est vraie car H_1 n'a pas b pour sommet. On peut trouver deux éléments de $S_1 \cup C$ simpliciaux pour H_1 et non voisins dans H_1. Comme ils ne sont pas voisins et que C est une clique, l'un des deux au moins est dans S_1. Comme ci-dessus, c'est alors un sommet simplicial de G.

Solution de l'exercice 43 - Dans tous les cas on trouve un sommet simplicial de G dans S_1 et de même on en trouve un dans S_2. Ces deux sommets ne peuvent pas être voisins dans G car sinon on aurait un chemin de a à b ne passant pas par C. On a donc montré que $\mathcal{P}(G)$ est vérifiée.
On a donc prouvé la récurrence sur la taille du graphe car la propriété est vraie pour les graphes de taille 3.

Solution de l'exercice 44 - Tout graphe cordal G admet au moins un sommet simplicial x.
Le graphe induit par $S \setminus \{x\}$, H, est encore cordal.
Par récurrence sur la taille on peut construire une suite de sommets qui créent un ordre d'élimination parfait.

Devoir I

DS4+ X-ENS 2018

Nombre chromatique et coloriage de graphe

Préliminaires

Complexité

Par *complexité en temps* d'un algorithme A, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaire à l'exécution de A dans le pire cas.

Lorsque la complexité en temps dépend d'un ou plusieurs paramètres $k_1, \ldots, k_r$, on dit qu'elle est en $O(f(k_1, \ldots, k_r))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_1, \ldots, k_r$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), la complexité est au plus $C \times f(k_1, \ldots, k_r)$.

On dit que la complexité en temps est *linéaire* quand f est une fonction linéaire des paramètres $k_1, \ldots, k_r$, *polynomiale* quand f est une fonction polynomiale des paramètres $k_1, \ldots, k_r$, et enfin *exponentielle* quand $f = 2^g$, où g est une fonction polynomiale des paramètres $k_1, \ldots, k_r$.

Les complexités (en temps) des algorithmes *devront être justifiées*.

Graphes

Rappelons qu'un graphe non-orienté est la donnée (S, A) de deux ensembles finis :

- un ensemble S de *sommets*, et
- un ensemble $A \subseteq S \times S$ d'*arêtes*, tel que pour tout couple de sommets (s, t), on a $s \neq t$ pour $(s, t) \in A$ et $(s, t) \in A$ si et seulement si $(t, s) \in A$.

Étant donné un graphe $G = (S, A)$, le *sous-graphe induit* par un ensemble de sommets $T \subseteq S$ est $(T, A \cap (T \times T))$.

Soit $G = (S, A)$ un graphe et soit $s \in S$ un sommet de G. Un *voisin* de s est un sommet t de G qui est relié à s par une arête, c'est-à-dire tel que $(s, t) \in A$. On note $V(s)$ l'ensemble des voisins de s. Le *degré* $d(s)$ de s est le cardinal de $V(s)$. Le *degré* $d(G)$ de G est le maximum des degrés de ses sommets. Un graphe est dit *étiqueté* lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers un ensemble non-vide arbitraire, que l'on appelle ensemble des étiquettes. Les étiquettes peuvent par exemple être des entiers, des listes ou des chaînes de caractères.

Coloriage

On dit qu'une fonction d'étiquetage L est un *coloriage* des sommets de $G = (S, A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées *couleurs*), c'est-à-dire lorsque L vérifie la condition (I.1) suivante :

$$\forall s, t \in S, (s, t) \in A \Rightarrow L(s) \neq L(t). \tag{I.1}$$

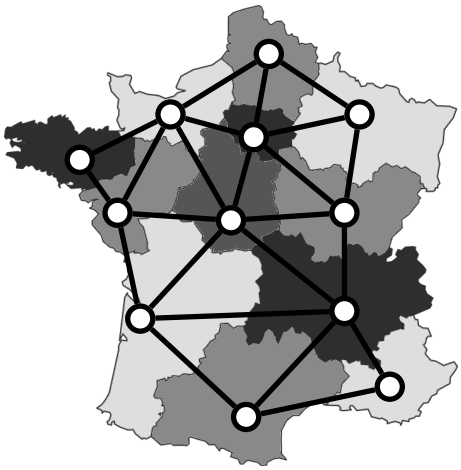


Figure I.1 – Exemple de graphe colorié : le graphe des régions métropolitaines françaises (hors Corse). Deux régions sont reliées par une arête dans le graphe si et seulement si elles sont voisines. Deux régions voisines sont de couleurs différentes.

Un graphe G est dit k-coloriable s'il admet un coloriage avec au plus k couleurs. Un graphe est dit colorié s'il est muni d'un coloriage. Un exemple de graphe colorié est donné sur la figure I.1 ci-dessus. Le *nombre chromatique* d'un graphe non-orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable. Cet énoncé porte sur le calcul des nombres chromatiques et de coloriages.

Représentation des graphes étiquetés

On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence. On se fixe également comme convention que les étiquetages des graphes sont tous à valeurs entières. L'étiquetage d'un graphe sera donné par un tableau d'entiers. On utilisera les types Ocaml suivants :

```
type graphe = bool array array;;
type etiquetage = int array ;;
```

Il n'est pas nécessaire de forcer les types : on considérera qu'ils sont confondus dans la suite du sujet. Ainsi, si une fonction `graphe -> int` est attendue, alors une fonction `bool array array -> int` convient.
Un graphe non-orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par une valeur `gphe` de type `graphe` telle que pour tous sommets $i, j \in S$, on ait `gphe.(i).(j)= true` si et seulement si $(i, j) \in A$. Le graphe G étant supposé non-orienté, on a alors également par symétrie `gphe.(j).(i)= true`. Pour un étiquetage `etiq` de `gphe`, l'étiquette du sommet `i` de `gphe` est donnée par `etiq.(i)`.

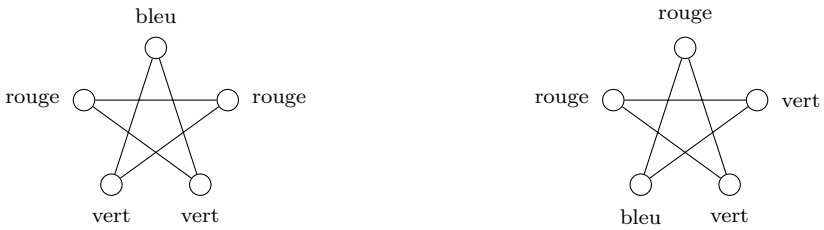
Fonctions OCaml

En plus des fonctionnalités de base du langage Ocaml, le/la candidat.e pourra utiliser les fonctions suivantes sans les programmer :

- `List.length : 'a list -> int`
  `List.length l` renvoie la longueur de `l`.
- `Array.of_list : 'a list -> 'a array`
  `Array.of_list l` renvoie un vecteur `t` de même longueur que `l`, qui contient les mêmes éléments que `l` et dans le même ordre.
- `range : int -> int array`
  `range n` renvoie le vecteur `[|0, ..., n-1|]`.

1 Coloriage

Question 1
Indiquer, pour chacun des graphes suivants, s'il est colorié.



Question 2
Donner le nombre chromatique, ainsi qu'un exemple de coloriage correspondant, pour le graphe de Petersen *représenté figure I.2*

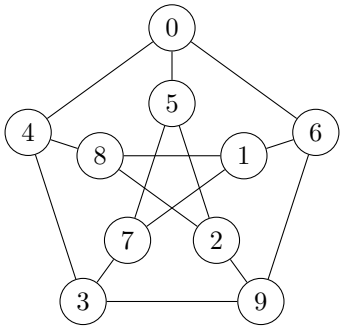


Figure I.2 – Le graphe de Petersen, de sommets 0, ..., 9.

La vérification de la propriété de coloriage est le problème suivant.

- *Entrée* : un graphe G et un étiquetage L de G.
- *Exercise* : L est-il un coloriage de G ?

Question 3
Écrire une fonction `est_col : graphe -> etiquetage -> bool`*, telle que* `est_col gphe etiq` *renvoie* `true` *si et seulement si* `etiq` *est un coloriage de* `gphe`*. Dans le cas où la taille de l'étiquetage est strictement inférieure au nombre de sommets du graphe, la fonction renvoie* `false`*. On demande une complexité quadratique en le nombre de sommets du graphe.*

Question 4
Démontrer que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets.

2 2-coloriage

Nous avons vu à la question 4 que le calcul du nombre chromatique peut s'effectuer en temps exponentiel en le nombre de sommets du graphe. Dans le cas général, on ne sait aujourd'hui pas faire mieux. Pour obtenir de meilleures bornes de complexité, il faut donc se limiter à des sous-problèmes. On considère dans cette partie le cas du 2-coloriage.

Graphe biparti. Un graphe G est *biparti* lorsque l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U, tels que chaque arête a une extrémité dans T et l'autre dans U.

Question 5
Démontrer qu'un graphe G est biparti si et seulement s'il est 2-coloriable.

On se propose de programmer la vérification de la 2-colorabilité des graphes en procédant comme suit. On effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe. On se donne pour ce faire trois étiquettes, disons -1, 0 et 1.
L'étiquetage est initialisé à -1 pour tous les sommets, et on teste la 2-colorabilité avec 0 et 1. Le principe de l'algorithme est le suivant :

1. On choisit un sommet s d'étiquette -1.
2. On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs 0 et 1 à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.
3. Enfin, s'il reste des sommets d'étiquette -1, alors on revient au point (1).

Question 6
Écrire une fonction `deux_col : graphe -> etiquetage` *telle que* `deux_col gphe` *renvoie un 2-coloriage de* `gphe` *si* `gphe` *est 2-coloriable. Le coloriage utilisera les couleurs 0 et 1. On demande une complexité quadratique en le nombre de sommets du graphe. Le comportement de la fonction est laissé au choix du/de la candidat.e lorsque* `gphe` *n'est pas 2-coloriable.*

3 Algorithmes gloutons

Dans cette partie, nous allons étudier deux algorithmes permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.

Ces deux algorithmes prennent en paramètre un ordre sur les sommets du graphe, que l'on appellera *ordre de numérotation.*
Par exemple, $1 < 3 < 4 < 0 < 2 < 6 < 5 < 9 < 8 < 7$ et
$0 < 7 < 2 < 5 < 4 < 6 < 8 < 1 < 3 < 9$ sont deux ordres de numérotation des sommets du graphe de Petersen (figure I.2).

Pour un graphe `gphe` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num.(k) = j` si et seulement si le sommet `j` apparaît en $(k+1)$-ème position dans l'ordre.

Nous commençons par l'*algorithme glouton* de coloriage. Cet algorithme construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :
On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné. Pour chaque sommet s parcouru :

1. On calcule l'ensemble $C(s) = \{L(t) | t \in V(s)\}$ des couleurs déjà données aux voisins de s.
2. On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.
3. On pose $L(s) = c$.

Question 7
Considérons le graphe de Petersen (figure I.2) et les deux ordres de numérotation :
`num1 = [|1;3;4;0;2;6;5;9;8;7|]` *et* `num2 = [|0;7;2;5;4;6;8;1;3;9|]`.

Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

Question 8
Écrire une fonction `min_couleur_possible : graphe -> etiquetage -> int -> int` *telle que pour un graphe* `gphe` *à n sommets, un étiquetage* `etiq` *à valeurs dans $\{-1, \dots, n-1\}$, et pour un sommet* `s` *de* `gphe`*, l'appel de* `min_couleur_possible gphe etiq s` *renvoie le plus petit entier naturel n'appartenant pas à l'ensemble* $\{$ `etiq.(t)` $\mid$ `t` $\in V($ `s` $)\}$.
On demande une complexité en $O(n)$.

Question 9
Écrire une fonction `glouton : graphe -> int array -> etiquetage`*, telle que, pour un graphe* `gphe` *et un ordre de numérotation* `num` *de ses sommets,* `glouton gphe num` *renvoie le coloriage glouton de* `gphe`*, avec au plus $d+1$ couleurs, où d est le degré de* `gphe`*. On demande une complexité en $O(n^2)$, où n est le nombre de sommets de* `gphe`*.*
Dans le cas où le tableau `num` *contient autre chose qu'un ordre de numérotation des sommets de* `gphe`*, le résultat de la fonction est laissé au choix du/de la candidat.e.*

Question 10
Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ couleurs, où d est le degré du graphe en entrée.

Question 11
Soit G un graphe. Montrer que pour tout coloriage L de G, il existe un ordre de numérotation des sommets tel que le coloriage glouton L' associé vérifie $L'(s) \leq L(s)$ pour tout sommet s de G. En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.

Les questions 7 à 11 indiquent que l'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe (dans notre cas les indices de la matrice d'adjacence, c'est-à-dire la permutation identité) est le plus simple à calculer, mais a peu de chances d'être efficace. A contrario, on pourrait essayer de déterminer l'ordre optimal, dont on a prouvé l'existence à la question 11, mais cela n'apporterait aucun bénéfice vis-à-vis de la complexité temporelle.
Une alternative est donnée par l'optimisation de Welsh-Powell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant. Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.

Question 12
Écrire une fonction `tri_degre : graphe -> int array`*, qui calcule le tableau des sommets d'un graphe trié par ordre décroissant de leurs degrés. En déduire une fonction* `welsh_powell : graphe -> etiquetage` *qui implémente l'optimisation de Welsh-Powell, et justifier le choix de votre algorithme de tri pour la fonction* `tri_degre`*.*

4 Algorithme de Wigderson

Considérons un graphe G avec n sommets. Supposons que G soit 3-coloriable, mais que l'on ait cette information sans pour autant effectivement disposer d'un 3-coloriage de G. Trouver un 3-coloriage de G pourrait prendre un temps exponentiel en n.
L'algorithme de Wigderson permet, pour un graphe G supposé 3-coloriable, de trouver en temps polynomial en n un coloriage de G en au plus $K\sqrt{n}$ couleurs couleurs où K est une constante.
Cet algorithme repose sur la propriété établie dans la question qui suit.

Question 13
Soit $k > 0$. Montrer que si G est $(k+1)$-coloriable, alors pour tout sommet s de G le sous-graphe induit par $V(s)$ est k-coloriable.

Voici le principe de l'algorithme de Wigderson. Soit G un graphe à n sommets, et tel que G est 3-coloriable.

1. On se donne comme couleur initiale $c = 0$.
2. Pour chaque sommet s de G pas encore colorié et ayant au moins $\sqrt{n}$ voisins pas encore coloriés :
   (a) On 2-colorie, avec les couleurs c et $c+1$, le sous-graphe induit par l'ensemble des voisins de s pas encore coloriés.
   (b) On incrémente c du nombre de couleurs utilisées en (a).
3. Enfin, on utilise l'algorithme glouton (avec un ordre de numérotation quelconque) pour colorier, avec des couleurs supérieures ou égales à c, le sous-graphe induit par l'ensemble des sommets par encore coloriés.

Question 14
Montrer que l'algorithme de Wigderson appliqué à un graphe 3-coloriable construit toujours un coloriage, et que ce coloriage utilise un nombre de couleur en $O(\sqrt{n})$, où n est le nombre de sommets du graphe.

Nous allons maintenant implémenter cet algorithme.
Commençons par programmer quelques fonctions auxiliaires simples.

Question 15
Écrire une fonction `sous_graphe : graphe -> int array -> graphe` *telle que pour* `gphe : graphe` *et* `sg : int array`*, si* `sg` *est à valeurs dans* $\{0, \dots,$ `Array.length gphe - 1`$\}$ *et sans répétition, alors* `sous_graphe gphe sg` *renvoie la matrice d'adjacence du graphe de sommets* $\{0, \dots,$ `Array.length sg - 1` $\}$*, et qui a une arête entre les sommets* `s` *et* `t` *si et seulement si* `gphe.(sg.(s)).(sg.(t))= true`*.*
Dans le cas où `sg` *a des valeurs hors de* $\{0, \dots,$ `Array.length gphe - 1` $\}$*, ou a des répétitions, le comportement de la fonction* `sous_graphe` *est laissé au choix du/de la candidat.e.*

Nous nous proposons d'utiliser pour les étiquetages la même convention que précédemment : on se donnera un étiquetage `etiq :etiquetage` de longueur `Array.length gphe`, initialisé à -1, et que l'on mettra à jour au fur et à mesure de l'algorithme.

Question 16
Écrire une fonction `voisins_non_colories : graphe -> etiquetage -> int -> int list` *telle que* `voisins_non_colories gphe etiq s` *renvoie la liste des voisins* `t` *de* `s` *non étiquetés, c'est-à-dire tels que* `etiq.(t)= -1`*.*
En déduire une fonction `degre_non_colories : graphe -> etiquetage -> int -> int` *telle que* `degre_non_colories gphe etiq s` *renvoie le nombre de voisins* `t` *de* `s` *.*

Question 17
Écrire une fonction `non_colories : graphe -> etiquetage -> int list` *telle que* `non_colories gphe etiq` *renvoie la liste des sommets* `s` *de* `gphe` *non étiquetés.*

Nous disposons maintenant de toutes les briques nécessaires à l'implémentation de l'algorithme de Wigderson.

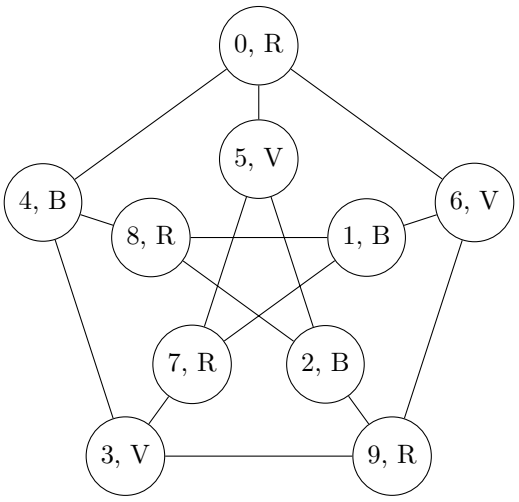
Question 18
Écrire une fonction `wigderson : graphe -> etiquetage` *telle que si* `gphe` *est 3-coloriable, alors* `wigderson gphe` *renvoie un coloriage de gphe obtenu par l'algorithme de Wigderson décrit plus haut. On demande une complexité polynomaile en le nombre de sommets de* `gphe`*. De plus, les propriétés sur le coloriage établies à la question 14 devront être respectées et justifiées.*
Le comportement de la fonction est laissé au choix du/de la candidat.e lorsque `gphe` *n'est pas 3-coloriable.*

Question 19
Comment pourrait-on étendre l'algorithme de Wigderson à des graphes de nombre chromatique connu et strictement supérieur à 3 ?

5 Solutions

Solution de l'exercice 1 - Le premier graphe admet deux sommets adjacent qui ont la même couleur ; l'étiquetage proposé n'est pas un coloriage.
Le second est colorié car l'étiquetage proposé est un 3-coloriage.

Solution de l'exercice 2 - Le graphe contient des cycles de longueur impaire, par exemple (0, 4, 3, 9, 6) : il n'est pas 2-coloriable. En effet il faudrait que les couleurs soient alternées dans le cycle et on arriverait à deux couleurs égales pour les sommets 0 et 6 qui sont reliés.
Par contre il est 3-coloriable :



Solution de l'exercice 3 -

```
let est_col gphe etiq =
   let n = Array.length gphe in
   let p = Array.length etiq in
   let coloriable = ref true in
   if n > p
   then coloriable := false
   else for i = 0 to (n-2) do
           for j = (i+1) to (n-1) do
              if gphe.(i).(j) && etiq.(i) = etiq.(j)
              then coloriable := false done done;
   !coloriable;;
```

Solution de l'exercice 4 - Un graphe de n sommet possède un n-coloriage, il suffit de donner une couleur distincte à chaque sommet.
Pour tester son nombre chromatique il suffit de tester, pour k variant de 2 à $n-1$ s'il admet un coloriage à k couleurs.
Pour cela on peut tester tous les coloriages : il y en a k^n.
La complexité est exponentielle car elle est majorée par

$$An^2\left(\sum_{k=2}^{n-1} k^n\right) \leqslant An^2.n.n^n = A2^{(n+3)\log_2(n)} \leqslant B2^{n^2}$$

Solution de l'exercice 5 - Si G est biparti, alors ses sommets peuvent se diviser en deux sous-ensembles T et U. On attribue alors la couleur 1 aux sommets de T, et 2 aux sommets de U. La propriété de coloration est alors instantanément vérifiée.

Inversement, si G possède une 2-coloration, alors on peut appeler T les sommets recevant la couleur 1 et U les sommets recevant la couleur 2. Aucune arête ne peut relier deux sommets de même couleur et les arêtes vont donc d'un sommet de T vers un sommet de U.

Solution de l'exercice 6 -

```
let deux_col gphe =
   let n = Array.length gphe in
   let etiq = Array.make n (-1) in
   let rec explo i k =
      etiq.(i) <- k;
      for j = 0 to n-1 do
          if gphe.(i).(j) && etiq.(j) = -1
          then explo j (1-k) done in
   for i = 0 to n-1 do
      if etiq.(i) = -1
      then explo i 0 done;
   etiq;;
```

Ici, si le graphe n'est pas 2-coloriable, alors le programme renvoie une coloration fausse (on ne détecte pas les erreurs si le sommet j est déjà colorié).
Le programme `explo` ne peut être lancé qu'une seule fois par sommet au maximum, et sa complexité est en $O(n)$. On arrive donc à une complexité en $O(n^2)$ dans le pire des cas.

Solution de l'exercice 7 - Avec le premier ordre, on trouve comme colorations pour les sommets : $(0; 0; 0; 0; 1; 1; 1; 2; 2; 2)$, donc trois couleurs.
Avec le second, on trouve comme colorations : $(0; 3; 0; 2; 1; 1; 1; 0; 2; 3)$, donc quatre couleurs.

Solution de l'exercice 8 -

```
let min_couleur_possible gphe etiq s =
   let n = Array.length gphe in
   let coul = Array.make n false in
   for i = 0 to (n-1) do
      if gphe.(s).(i) && etiq.(i) <> -1
      then coul.(etiq.(i)) <- true done;
   let c = ref 0 in
   while coul.(!c) do c := !c + 1 done;
   !c;;
```

Solution de l'exercice 9 -

```
let glouton gphe num =
   let n = Array.length gphe in
   let coul = Array.make n (-1) in
   for i = 0 to (n-1) do
      let k = num.(i) in
      coul.(k) <- min_couleur_possible gphe coul k done;
  coul;;
```

Solution de l'exercice 10 - La fonction `min_couleur_possible` renvoie une couleur qui n'a pas été affectée aux voisins du sommet passé en paramètre. Lorsqu'une couleur est affectée à un sommet elle ne pourra plus être affectée aux voisins qui suivent dans l'ordre de numération. De plus chaque sommet reçoit une couleur lorsque `num` est un ordre de numération.
On a bien construit un coloriage du graphe.

Dans l'algorithme glouton chaque sommet est colorié par une couleur non employée par ses voisins déjà coloriés, comme il admet au plus $d(G)$ voisins, l'ensemble $\{0, 1, \ldots, d(G)\}$ contient au moins

une valeur non atteinte par les voisins. La couleur choisie est donc dans cet ensemble de cardinal $1 + d(G)$ à chaque étape donc la coloration construite admet au plus $d(G) + 1$ couleurs.

Solution de l'exercice 11 - Soit L un coloriage de G. On considère une numération des sommets qui les classe par numéro de couleur croissante.
Comme deux sommets de même couleur ne sont pas adjacents, lors de l'appel de `min_couleur_possible` les seuls sommets voisins de s qui seront considérés auront une couleur strictement inférieure à celle de s, $L(s)$. La couleur choisie, $L'(s)$, ne pourra donc pas être strictement supérieure à $L(s)$: on a $L'(s) \leqslant L(s)$ pour tout s.
Si on part d'un coloriage optimal pour construire la numération des sommets on aboutit donc à un coloriage dont le maximum est majoré par celui du coloriage optimal : ce sera donc aussi un coloriage optimal.

Solution de l'exercice 12 - Comme l'algorithme `glouton` est de complexité quadratique on peut choisir un algorithme lui-même quadratique pour construire une numération des sommets sans augmenter la complexité.
On commence par construire un tableau avec les degrés.

```
let degres gphe =
   let n = Array.length gphe in
   let deg = Array.make n 0 in
   for i = 0 to (n-1) do
      for j = 0 to (n-1) do
         if gphe.(i).(j)
         then deg.(i) <- deg.(i) + 1 done done;
   deg;;
```

Les outils utilisés ensuite (dont `range`)

```
let echanger tab i j =
   let temp = tab.(i) in
   tab.(i) <- tab.(j);
   tab.(j) <- temp;;

let range n =
   Array.init n (fun x -> x);;
```

La création de la numération suit la méthode du tri par sélection

```
let tri_degre gphe =
   let n = Array.length gphe in
   let deg = degres gphe in
   let num = range n in
   for i = 0 to (n-2) do
      let max = ref i in
      let deg_max = ref deg.(num.(i)) in
      for j = (i+1) to (n-1) do
      let d =  deg.(num.(j)) in
         if d > !deg_max
         then (max := j; deg_max := d) done;
      echanger num i !max done;
   num;;
```

Il suffit alors de tout rassembler.

```
let welsh_powell gphe =
    glouton gphe (tri_degre gphe);;
```

Solution de l'exercice 13 - Supposons que le graphe G est $(k+1)$-coloriable, et soit s un de ses sommets. On considère une $(k+1)$-coloration de G. Alors aucun sommet de $V(s)$ n'est de la même couleur que s (ce serait contradictoire). Sur le sous-graphe induit par $V(s)$, cette coloration reste valide, et n'utilise donc que k couleurs au maximum (celle de s n'est pas employée).

Solution de l'exercice 14 - Fixons les notations.
On pose $G_0 = G$ de taille n.
L'étape (a) se décompose en étapes.

1. Si G_i admet un sommet de degré supérieur à $\sqrt{n}$,
2. on choisit un tel sommet, s_i, par exemple celui de degré maximum,
3. on colorie avec les couleurs $2i$ et $2i+1$ $V(s_i)$, ce qui est possible d'après la question précédente. En effet G_i est induit d'un graphe 3-coloriable donc est 3-coloriable d'où $V(s_i)$ est 2-coloriable.
4. On définit G_{i+1} comme le graphe induit de G_i en enlevant les sommets appartenant à $V(s_i)$.

Après au plus $\sqrt{n}$ itérations G_p n'admet plus de sommet de degré supérieur à $\sqrt{n}$ et on le colorie par l'algorithme glouton.
On a ainsi partitionné l'ensemble des sommets : $S = S_0 \cup S_1 \cup \cdots S_p$,
où $S_i = V(s_i)$ pour $i < p$ et S_p est l'ensemble des sommets de G_p.
Chaque S_i admet un coloriage :

1. pour $i < p$, $V(s_i)$ est colorié par $2i$ et $2i+1$,
2. G_p est colorié par q couleurs qui sont supérieures ou égales à $2p$.
   De plus, d'après la question **10**, on a $q \leqslant \sqrt{n} + 1$.

Comme chaque partie est coloriée par des couleurs distinctes on obtient un coloriage de G avec $2p + q$ couleurs qui vérifient $2p + q \leqslant 2\sqrt{n} + \sqrt{n} + 1 = \mathcal{O}\big(\sqrt{n}\big)$.

Solution de l'exercice 15 -

```
let sous_graphe gphe sg =
   let p = Array.length sg in
   let ss_gphe = Array.make_matrix p p false in
   for i = 0 to (p-1) do
      let si = sg.(i) in
      for j = 0 to (p-1) do
         let sj = sg.(j) in
         ss_gphe.(i).(j) <- gphe.(si).(sj) done done;
   ss_gphe;;
```

Solution de l'exercice 16 -

```
let voisins_non_colories gphe etiq s =
   let n = Array.length gphe in
   let vnc = ref [] in
   for j = 0 to (n-1) do
      if gphe.(s).(j) && etiq.(j) = -1
      then vnc := j :: !vnc done;
   !vnc;;

let degre_non_colories gphe etiq s =
    List.length (voisins_non_colories gphe etiq s);;
```

Solution de l'exercice 17 -

```
let non_colories gphe etiq =
   let n = Array.length gphe in
   let nc = ref [] in
   for i = 0 to (n-1) do
      if etiq.(i) = -1
      then nc := i :: !nc done;
   !nc;;
```

Solution de l'exercice 18 -
Pour ne pas écrire un programme démesurément long, on sort quelques fonctions.
La première détermine le degré en sommets non coloriés maximal. Le test de majoration de $\sqrt{n}$ est renvoyé par le type optionnel : la fonction renvoie `None` s'il n'existe pas de sommets avec suffisamment de voisins non coloriés et sinon elle renvoie un sommet k vérifiant cette proprité sous la forme `Some k`.

```
let sous_degre_maximum gphe etiq =
   let n = Array.length gphe in
   let r = int_of_float (sqrt (float_of_int n)) in
   let deg_max = ref (degre_non_colories gphe etiq 0) in
   let ind_max = ref 0 in
   for i = 1 to (n-1) do
      let d = degre_non_colories gphe etiq i in
      if  d > !deg_max
      then (deg_max := d; ind_max := i) done;
   if !deg_max > r then Some !ind_max else None;;
```

La seconde modifie le tableau des couleurs (`etiq`) à partir des couleurs calculées (`ss_etiq`) pour un sous-graphe (`sg`)

```
let ajouter_couleurs etiq sg ss_etiq couleur=
   let p = Array.length sg in
   for i = 0 to (p-1) do
      etiq.(sg.(i)) <- ss_etiq.(i) + couleur done;;
```

Le programme consiste alors à répéter la recherche de sommets avec suffisamment de voisins non coloriés tant qu'il en existe.

```
let wigderson g =
   let n = Array.length g in
   let e = Array.make n (-1) in
   let c = ref 0 in
   let rec aux () =
      match sous_degre_maximum g e with
      |None -> let nc = Array.of_list (non_colories g e) in
               let sg = sous_graphe g nc in
               let se = glouton sg (range (Array.length sg))
                  in
               ajouter_couleurs e nc se !c
      |Some i -> let nc = Array.of_list (voisins_non_colories
         g e i) in
                 let sg = sous_graphe g nc in
                 let se = deux_col sg in
                 ajouter_couleurs e nc se !c;
                 c := !c + 2;
                 aux ()
   in aux ();
```

```
e ; ;
```

Toutes les fonctions auxiliaires sont de complexité polynomiale en la taille du graphe et elles sont appelées au plus $\sqrt{n}$ fois : la complexité est polynomiale.
Je ne comprends pas la demande de justification des propriétés, c'est ce qui a été fait à la question **14**.

Solution de l'exercice 19 - Comme on sait définir un coloriage dans le cas d'un graphe 3-coloriable on peut poursuivre le raisonnement de manière semblable.
Si un graphe est 4-coloriable, les voisins d'un sommets sont 3-coloriables ; on va donc pouvoir définir un coloriage pour les "gros" voisinages puis conclure par un algorithme glouton.
On considère la propriété $\mathcal{P}(k)$: on peut définir un coloriage de $A_k.n^{a_k}$ couleurs d'un graphe k-coloriable avec $a_k < 1$.
$\mathcal{P}(2)$ est vraie avec $a_2 = 1$ et $A_2 = 2$.
$\mathcal{P}(3)$ est vraie avec $a_2 = \frac{1}{2}$ et $A_3 = 3$.
On suppose $\mathcal{P}(k)$ vraie. G est un graphe $k+1$-coloriable.
Pour chaque sommet s de G ayant au moins n^r voisins pas encore coloriés on colorie ceux-ci avec au plus $A_k.n^{a_k}$ couleurs non encore utilisées. On colorie le reste avec au plus $n^r + 1$ couleurs non utilisées.
Le nombre de sommets avec plus de n^r voisins est au plus $\frac{n}{n^r}$ donc on a utilisé au plus $n^{1-r}.A_k.n^{a_k} + n^r$ couleurs.
On peut choisir r tel que l'expression ci-dessus devienne homogène : $1 - r + a_k = r$ donc $r = \frac{1+a_k}{2}$.
On obtient ainsi un coloriage avec au plus $(A_k + 1).n^r$ couleurs.
La récurrence $a_{k+1} = \frac{1+a_k}{2}$ avec $a_2 = 0$ donne $a_k = 1 - 2^{2-k}$. donc
la propriété $\mathcal{P}(k)$ est donc vraie avec $A_k = k$ et $a_k = 1 - 2^{2-k}$.

Dans le calcul ci-dessus on n'a pas tenu compte du fait qu'on appliquait $\mathcal{P}(k)$ à des graphes de tailles inférieures à n.
Si les tailles auxquelles on applique $\mathcal{P}(k)$ sont m_1, m_2, …, m_p, le nombre de couleurs employées est en fait majoré par $\displaystyle\sum_{i=1}^{p} A_k.m_i^{a_k} + n^r$

La fonction $x \mapsto n^{a_k}$ est concave donc $\dfrac{\sum_{i=1}^{p} m_i^{a_k}}{p} \leqslant \left(\displaystyle\sum_{i=1}^{p} \frac{m_i}{p}\right)^{a_k}$.

Ainsi $\displaystyle\sum_{i=1}^{p} A_k.m_i^{a_k} + n^r \leqslant p^{1-a_k} A_k \left(\sum_{i=1}^{p} m_i\right)^{a_k} + n^r$.

Comme on a $\displaystyle\sum_{i=1}^{p} m_i \leqslant n$ et $p \leqslant n^{1-r}$, le nombre de couleurs utilisées est majoré par
$n^{(1-r)(1-a_k)} A_k n^{a_k} + n^r$.
On choisit $a_{k+1} = r$ tel que $(1-r)(1-a_k) + a_k = r$ d'où $r = \frac{1}{2-a_k}$.
$a_2 = 0$ donne bien $a_3 = \frac{1}{2}$.
On calcule ensuite $a_4 = \frac{2}{3}$, $a_5 = \frac{3}{4}$ et, par récurrence, $a_k = \frac{k-2}{k-1} = 1 - \frac{1}{k-1}$ qui donne moins de couleurs que la valeurs $1 - 2^{2-k}$.