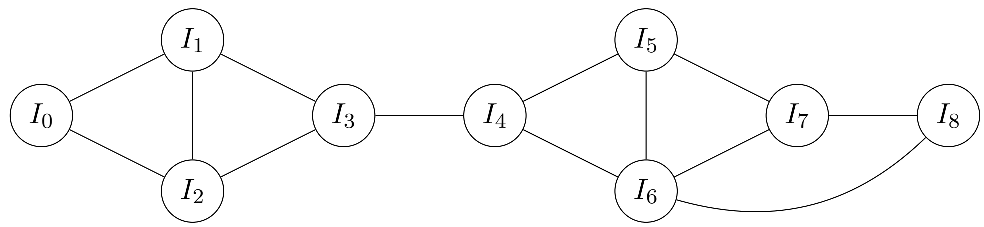
*I.A.*1) On note `I = (ai,bi)` et `J = (aj,bj)`.
Les intervalles I et J ne sont pas en conflit si et seulement si (`aj > bi` ou `ai > bj`).
En prenant le contraire on obtient

```
let conflit (ai,bi) (aj,bj) = (bi >= aj) && (bj >= ai);;
```

*I.C.*1) Le problème b donne le graphe



*I.C.*2) On teste tous les couples (i, j) avec $0 \leq i < j \leq n-1$ où n est le nombre d'intervalles. Si I_i et I_j sont en conflits, on ajoute i à la liste des voisins de j et on ajoute j à la liste des voisins de i.

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

*I.D.*1) Pour le problème a de la figure 1. Il faut au moins trois couleurs car les intervalles I_0, I_1 et I_2 ne peuvent pas être de la même couleur. On peut alors prendre $(0, 1, 2, 0, 1, 0, 0)$ comme coloriage. Pour le problème b de la figure 1. Il faut au moins trois couleurs car les intervalles I_0, I_1 et I_2 ne peuvent pas être de la même couleur. On peut alors prendre $(0, 1, 2, 0, 1, 0, 2, 1, 0)$ comme coloriage.

I.D.2.a) Fonction classique.

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

I.D.2.b) On ne demande pas de complexité particulière, on peut écrire une fonction de complexité quadratique qui utilise la fonction `appartient`. La fonction auxiliaire `aux : int list -> int -> int` est écrite pour que `aux l n` renvoie le plus petit entier supérieur à n absent de la liste :

```
let plus_petit_absent l =
  let rec aux l n =
    if appartient l n then
      aux l (n + 1)
    else
      n
  in aux l 0;;
```

*I.D.*2.*c*) L'énoncé ne précise pas s'il y a lieu ou pas d'éliminer les éventuels doublons dans la liste renvoyée. On décide de les laisser. Le principe est de parcourir les sommets voisins de i avec la fonction auxillaire `parcours` et d'ajouter la couleur des sommets qui sont déjà colorés.

```
let couleurs_voisins aretes couleurs i =
  let rec parcours l accu =
    match l with
      | [] -> accu
      | t :: q -> if couleurs.(t) <> - 1 then parcours q (couleurs.(t) :: accu)
           else parcours q accu
  in
    parcours aretes.(i) [];;
```

*I.D.*2.*d*) On utilise les deux fonctions précédentes.

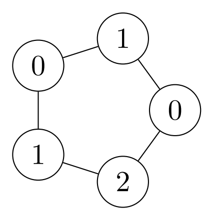
```
let couleur_disponible aretes couleurs i =
    plus_petit_absent (couleurs_voisins aretes couleurs i);;
```

*I.E.*1.*a*) Si le graphe ne possède pas d'arêtes. On peut colorer tous les sommets avec la même couleur et donc $\chi(G) = 1$. Une clique ne peut pas avoir plusieurs sommets (car deux sommets ne sont jamais reliés). On a donc $\omega(G) = 1$.

*I.E.*1.*b*) Si G est le graphe complet a n sommets. Aucun sommet ne peut porter la même couleur donc $\chi(G) = n$. De plus S est une clique donc $\omega(G) = n$.

*I.E.*2) On a $\chi(G) \geq \omega(G)$. En effet soit C une clique telle que $|C| = \omega(G)$. Les sommets de la clique ne peuvent pas être coloriés avec une même couleur.

Il se peut que $\chi(G) > \omega(G)$. Par exemple le graphe suivant vérifie $\omega(G) = 2$ et $\chi(G) = 3$.



*I.E.*3) Pour déterminer si la liste `xs` est une clique on utilise une fonction recursive.

— Si la liste est vide c'est une clique.

— Si `xs = t :: q`. Pour ce que soit une clique il faut que `t` soit relié à tous les sommets de la liste `q` et que `q` soit une clique.

On construit une fonction `sommet_relie_liste` qui renvoie un booléen selon qu'un sommet `x` est relié (ou non) à tous les sommets d'une liste `l`

```
let rec sommet_relie_liste x l aretes =
   match l with
     | [] -> true
     | t :: q -> appartient aretes.(t)  x && sommet_relie_liste x q aretes;;
```

On en déduit la fonction suivante

```
let rec est_clique aretes xs =
  match xs with
    |[] -> true
    |t :: q -> sommet_relie_liste t q aretes && est_clique aretes q;;
```

II.A) On obtient $(0, 1, 2, 0, 1, 0, 2, 1, 0)$

II.B) Il suffit d'utiliser les fonctions définies dans la partie I.

```
let coloration segments aretes =
  let n = Array.length aretes in
    let coul = Array.make n (- 1) in
      coul.(0) <- 0;
      for k = 1 to (n - 1) do
        coul.(k) <- couleur_disponible aretes coul k
      done;
      coul;;
```

*II.C.*1) Si le segment reçoit la couleur c, c'est que les couleurs $0, 1, \ldots, c-1$ sont des couleurs qui apparaissent pour des segments I_i avec $I_i \cap I_k \neq \emptyset$ ce qui signifie que l'extrémité gauche de I_k est dans ces segments. On en déduit qu'il y a au moins c tels segments.

*II.C.*2) On considère c segments $I_{i_1}, \ldots, I_{i_c}$ parmi ceux qui contiennent l'extrémité gauche de I_k (on a vu qu'il y en a au moins c). Les segments $I_{i_1}, \ldots, I_{i_c}, I_k$ forment une clique car l'extrémité gauche de I_k appartient à tous. Cette clique contient $c+1$ intervalles.

*II.C.*3) Le graphe contient une clique de cardinal $c+1$. Il faut donc au moins $c+1$ couleurs pour le colorer d'après *I.E.*2.

*II.C.*4) Pour $k \in \{1, \ldots, n-1\}$ on note c_k la couleur du segment I_k et on note c le maximum des c_k. Le coloriage obtenu a donc $c+1$ couleurs. En appliquant les trois précédentes questions pour k tel que $c_k = c$. On obtient bien que le coloriage est optimal.

II.D) Si on dispose d'un graphe avec m arêtes. On a vu que la fonction `plus_petit_absent` était quadratique en la longueur de la liste `l` qui est majorée par m (en effet la fonction `appartient` est linéaire en la longueur de la liste). Comme la fonction `couleurs_voisins` a une complexité $O(m)$ on en déduit que la complexité de `couleur_disponible` est $O(m^2)$ et donc la complexité de `coloration` est $O(m^3)$.

III.A) L'ordre $(x_5, x_6, x_2, x_4, x_1, x_7, x_3, x_0)$ est un ordre d'élimination parfait.

*III.B.*1) On a

```
let voisins_inferieurs aretes x =
  let rec aux l =
    match l with
      | [] -> []
      | t :: q -> if t < x then t :: aux q else aux q
  in
    aux aretes.(x);;
```

*III.B.*2) On ajoute les sommets dans l'ordre et on teste au fur et à mesure si les voisins d'indice inférieur forment une clique.

```
let est_ordre_parfait aretes =
  let n = Array.length aretes in
    let est_parfait = ref true in
      let k = ref 0 in
        while !est_parfait && !k < n do
          est_parfait := est_clique aretes (voisins_inferieurs aretes !k);
          k := !k + 1
        done;
        !est_parfait;;
```

III.C) Déjà fait en *II.C.*2)

*III.D.*1.*a*) On obtient la coloration $(0, 1, 0, 1, 2, 3, 2, 0)$.

*III.D.*1.*b*) Donnons la coloration obtenue en précisant les couleurs de $x_5, x_6, x_2 \ldots, x_0$ dans cet ordre : $(0, 1, 2, 1, 3, 2, 0, 0)$.

*III.D.*2) On procède comme le précédent algorithme de coloration en ne s'occupant que des voisins d'indice strictement inférieur.

```
let coul_voisins_inf aretes couleurs i =
  let rec parcours l accu =
    match l with
      | [] -> accu
      | t :: q -> couleurs.(t) :: accu
  in
    parcours (voisins_inferieurs aretes i) [];;

let colore aretes =
  let n = Array.length aretes in
    let coul = Array.make n (- 1) in
      coul.(0) <- 0;
      for k = 1 to (n - 1) do
        coul.(k) <- plus_petit_absent (coul_voisins_inf aretes coul k)
      done;
      coul;;
```

*III.D.*3) Comme dans *II.D*

*IV.A.*1) Supposons par l'absurde qu'un segment soit inclus dans un autre.

— Supposons que $I_0 \subset I_1$. Alors $I_3 \subset I_0 \subset I_3 \subset I_1 = \emptyset$. C'est absurde. Par symétrie, on montre de même qu'il n'est pas possible que $I_i \subset I_{i+1}$ ou $I_{i+1} \subset I_i$ où les indices sont vus modulo 4.

— Il n'est pas possible que $I_0 \subset I_2$ car $I_0 \cap I_2 = \emptyset$. Là encore par symétrie on en déduit que tous les autres cas d'inclusion sont impossibles.

En conclusion, aucun segment n'est inclus dans un autre.

*IV.A.*2) On sait que $I_1 \cap I_2 \neq \emptyset$ et que les deux intervalles ne sont pas inclus l'un dans l'autre. De ce fait on a $\min I_1 < \min I_2 < \max I_1 < \max I_2$ ou $\min I_2 < \min I_1 < \max I_2 < \max I_1$. Dans le deuxième cas, $\min I_1 \in I_2$ or $\min I_1 \in I_0$ et $I_0 \cap I_2 = \emptyset$ donc c'est absurde. On a donc $\min I_1 < \min I_2 < \max I_1 < \max I_2$ On peut faire la même chose pour I_2 et I_3 et on obtient : $\min I_2 < \min I_3 < \max I_2 < \max I_3$.

*IV.A.*3) On a montré que $\min I_0 < \min I_1 < \min I_2$ mais on a aussi $I_2 \cap I_0 = \emptyset$ donc $\max I_0 < \min I_2$ et donc $\max I_0 < \min I_3$ ce qui est absurde car $I_0 \cap I_3 \neq \emptyset$ par hypothèses. On a obtenu une contradiction donc tout cycle de longueur 4 de G a une corde.

IV.B) On procède de même qu'en *IV.A*). On suppose que l'on a un cycle $(I_0, \ldots, I_{p-1})$ de longueur p avec $p \geq 4$. C'est-à-dire :

$$I_0 \cap I_1 \neq \emptyset, I_1 \cap I_2 \neq \emptyset, \ldots, I_{p-2} \cap I_{p-1} \neq \emptyset, I_{p-1} \cap I_0 \neq \emptyset.$$

On suppose que ce cycle n'a pas de corde, en particulier, I_i et I_{i+2} ne sont jamais reliés :

$$\forall i \in \{0, \ldots, p-3\}, I_i \cap I_{i+2} = \emptyset, I_{p-2} \cap I_0 = I_{p-1} \cap I_1 = \emptyset.$$

Deux intervalles ne peuvent donc pas être inclus l'un dans l'autre et si on suppose alors que $\min I_0 < \min I_1 < \max I_0 < \max I_1$ alors pour tout $i < p-1$,

$$\min I_i < \min I_{i+1} < \max I_i < \max I_{i+1}$$

De plus, comme il n'y a pas de cordes on obtient que $\max I_i < \min I_{i+2}$ ce qui est absurde car $I_0 \cap I_{p-1} \neq \emptyset$.

*IV.D.*1) On construit une fonction auxiliaire **aux** qui permet de ne garder que les voisins du sommet **k** qui sont dans le sous-graphe défini par **sg**. Il ne reste plus qu'à tester si cette liste désigne une clique.

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

L'appel à la fonction auxiliaire est de complexité $O(n)$ où n est le nombre de sommets du graphe. La fonction `est_clique` est de complexité $O(n^3)$. Notre fonction `simplicial` est donc elle aussi de complexité $O(n^3)$.

*IV.D.*2) On parcourt les sommets du graphe jusqu'à trouver un sommet qui soit dans le sous-graphe et qui soit simplicial (si le sommet n'est pas dans le sous-graphe on ne teste pas s'il est ou non simplicial). Si on ne trouve aucun sommet simplicial, la fonction renvoie -1.

```
let trouver_simplicial (aretes, sg) =
  let n = Array.length aretes in
    let trouve = ref false in
      let k = ref 0 in
        while !trouve = false && !k < n do
          if sg.(!k) && simplicial (aretes, sg) !k then trouve := true;
          k := !k + 1
        done;
        if !trouve then !k - 1 else (- 1);;
```

*IV.D.*3) Le principe de l'algorithme est le suivant.

— On cherche un sommet simplicial. On le place, à la fin de l'ordre.

— On recommence avec le graphe obtenu en supprimant le sommet que l'on vient de prendre.

Les questions *IV.F* et *IV.G* montrerons que cet algorithme donne toujours un ordre d'élimination parfait si le graphe est cordal.

```
let ordre_parfait aretes =
  let n = Array.length aretes in
    let sg = Array.make n true in
      let ordre = ref [] in
        for k = 1 to n do
          let x = trouver_simplicial (aretes, sg) in
            sg.(x) <- false;
            ordre := x :: !ordre;
        done;
        !ordre;;
```

*IV.E.*1) Les sommets a et b sont déconnectés par la coupure et x est un sommet de la coupure. Il existe donc un chemin allant de a à b et passant par $a \to x \to b$ car sinon $C \setminus \{x\}$ serait une coupure. On peut de même supposer que x est le seul sommet du chemin qui soit dans C pour la même raison. On en déduit que le sommet qui précède x dans le chemin est dans G_1 et celui qui succède à x est dans G_2. On fait de même pour y

*IV.E.*2) Soit a_1 un voisin de x dans G_1 et α un voisin de y dans G_1. Comme a_1 et α sont deux éléments de G_1 il existe un chemin inclus dans G_1 qui va de a_1 à α. On en déduit le chemin voulu. On peut obtenir de même le chemin P_2.

*IV.E.*3) On construit donc un cycle $x \to a_1 \to a_p \to y \to b_q \to b_1 \to x$. Supposons par l'absurde que x et y ne soient pas reliés dans le graphe. Dans ce cas $p \geq 1$ et $q \geq 1$. Le cycle a au moins 4 sommets. On en déduit qu'il admet une corde car le graphe est cordal.

— Supposons qu'il existe une corde $a_i \to b_j$. Dans ce cas $G_1 = G_2$ ce qui est absurde.

— Supposons qu'il existe une corde $a_i \to a_j$ (ou $b_i \to b_j$). Cela contredit le caractère minimal des chemins P_1 et P_2.

On a bien une absurdité. Donc x et y sont reliés dans le graphe.

*IV.E.*4) On a montré que deux sommets x et y de C étaient reliés. Donc C est une clique.

*IV.F.*1) Dans un graphe complet tous les sommets sont reliés à tous les sommets. Donc tout groupe de sommets est une clique. Tout sommet est donc simplicial.

*IV.F.*2) — Si $|G| = 1$, l'ensemble des voisins d'un sommet est vide. C'est une clique. Le sommet est simplicial. On a bien $P(G)$ car G est complet

— Si $|G| = 2$. Soit a un sommet. L'ensemble de ses voisins est de cardinal 0 ou 1. C'est une clique donc le sommet est simplicial. Les deux sommets sont bien simpliciaux. De plus si le graphe n'est pas complet ils ne sont pas voisins. On a bien $P(G)$.

— Si $|G| = 3$. Si le graphe est complet alors tous les sommets sont simpliciaux. S'il n'est pas complet, on peut regarder deux sommets a et b qui ne sont pas voisins. Ils ont chacun au plus 1 voisins. Ils sont donc simpliciaux. On a bien $P(G)$.

*IV.F.*3.*a*) Le graphe H_1 est un sous-graphe du graphe G qui est cordal, il est donc cordal.

*IV.F.*3.*b*) Comme H_1 est complet, tous ses sommets sont simpliciaux. En particulier les sommets de S_1 sont des sommets simpliciaux de H_1. Soit a un tel sommet. Ses voisins appartiennent à H_1 car les sommets de G_1 ne sont pas reliés aux sommets de G_2 (IV.E.3). Le sommet a est donc un sommet simplicial de G.

*IV.F.*3.*c*) Par recurrence, H_1 est cordal est il a strictement moins de sommets que G, comme il n'est pas complet il contient au moins deux sommets simpliciaux s_1 et s_2. Ils ne peuvent pas être tous les deux dans la coupure car c'est une clique (IV.E.4). L'un des deux sommets est donc dans S_1. Comme précédemment, c'est un sommet simplicial de G.

*IV.F.*3.*d*) On a vu que la propriété était vrai si $|G| \in \{1, 2, 3\}$. Soit n un entier au moins égal à 4. On supppose que $P(G)$ est vrai pour tout graphe cordal G tel que $|G| < n$. Maintenant, si G est complet alors il admet un sommet simplicial d'après (IV.F.1). Par contre, si G n'est pas complet, la question IV.F.3.c affirme qu'il existe un sommet simplicial de G dans S_1. En procédant de même on peut trouver un sommet simplicial dans S_2. Ces deux sommets ne sont pas reliés, donc $P(G)$ est vérifié.

IV.G) Tout graphe cordal admet un ordre d'élimination parfait. Il suffit de choisir un sommet simplicial (il en existe un d'après la question précédente) et de recommencer avec le graphe obtenu en enlevant ce sommet (qui reste cordal).