Correction du devoir du Vendredi 5 Mai

Exercice 1 (Nombre d'occurrences d'un élément dans un ensemble)

1. (a)

```
let occurences1 a x =
  let n = Array.length a and compt = ref 0 in
    for k = 0 to (n-1) do
      if a.(k) = x then compt := !compt+1
    done;
  !compt
;;
```

(b) Proposons l'invariant de boucle : Soit $k \in [\![1, n]\!]$,

($\mathcal{H}_k$) "A la fin de la k-ième itération, *compt* contient le nombre de valeurs égales à x parmi $\{a.(0), \dots, a.(k-1)\}$."

• ($\mathcal{H}_1$) : A la première itération, on teste si $a.(0) = x$. Si c'est le cas, la variable *compt*, qui était initialisée à 0, prend la valeur $0 + 1 = 1$. Sinon, la variable *compt* reste égale à 0. Dans tous les cas, à la fin de la première itération, *compt* contient le nombre de valeurs égales à x parmi $\{a.(0)\}$.

• Soit $k \in [\![1, n]\!]$. Supposons ($\mathcal{H}_{k-1}$), c'est-à-dire qu'au début de la k-ième itération, *compt* contient le nombre de valeurs égales à x parmi $\{a.(0), \dots, a.(k-2)\}$. On teste alors si $a.(k-1) = x$. Si c'est le cas, la variable *compt* prend la valeur *compt* + 1. Sinon, la variable *compt* reste inchangée. Dans tous les cas, à la fin de la k-ième itération, *compt* contient bien le nombre de valeurs égales à x parmi $\{a.(0), \dots, a.(k-1)\}$. Donc ($\mathcal{H}_k$) est vraie.

• Par principe de récurrence, pour tout k appartenant à $[\![1, n]\!]$, ($\mathcal{H}_k$) est vraie. Notre algorithme est donc correct.

(c) On effectue n itérations avec à chaque fois une comparaison. On donc une complexité $C(n) = n = O(n)$ qui est linéaire.

2. (a)

```
let rec occurences2 a x = match a with
  | [] -> 0
  | t::q when (t=x) -> 1+(occurences2 q x)
  | t::q -> occurences2 q x
;;
```

(b) Nous proposons comme suite strictement décroissante la suite "longueur de la liste a". Si a est de longueur $n \in \mathbb{N}^*$, `occurences2 a x` amène un appel récursif sur la queue de a de longueur $n - 1 < n$.

L'ordre usuel sur $\mathbb{N}$ étant bien fondé, notre algorithme se termine.

(c) Choisissons comme taille de données n la longueur de la liste et comme opération fondamentale la comparaison. En notant $C(n)$ la complexité au pire, on obtient :

$$C(0) = 0 \text{ et } \forall n \in \mathbb{N}^*,\ C(n) = 1 + C(n-1).$$

Donc $C(n) = n = O(n)$. La complexité est linéaire.

3.
```
let occurrence3 a x =
  let rec aux a x acc = match a with
    | [] -> acc
    | t::q when (t=x) -> aux q x (acc+1)
    | t::q -> aux q x acc
  in aux a x 0
;;
```

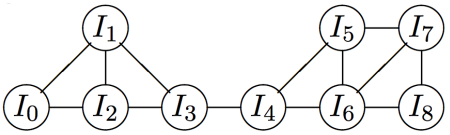
Exercice 2

1. (a) Voici la fonction demandée :

```
let conflit (a,b) (c,d) = not ((b<c)||(d<a));;
```

(b) /

(c) i. Voici le graphe d'intervalles pour le problème B de la figure 1 :



ii. On initialise un tableau `g` de bonne taille avec des listes vides. A l'aide d'une double boucle, on considère tous les couples (I_i, I_j) d'intervalles avec $i < j$. Quand on détecte un conflit, on construit une arête, ce qui revient à ajouter i à la liste numéro j et j à la liste numéro i (ce qui est possible puisque la structure de tableau est mutable). Voici la fonction demandée :

```
let construire_graphe t =
  let n = Array.length t in
    let g = Array.make n [] in
      for i = 0 to n-2 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;;
```

(d) i. Le graphe associé au problème A ne peut être colorié avec moins de trois couleurs du fait de la présence du triangle reliant I_0, I_1, I_2. Une 3-coloration sera donc optimale si on en trouve une. On peut choisir

$$(0, 1, 2, 0, 1, 0, 0).$$

De même, on a besoin d'au moins 3 couleurs pour le graphe du problème B (triangle I_0, I_1, I_2). Une coloration optimale convenable est

$$(0, 1, 2, 0, 1, 2, 0, 1, 2).$$

ii. Pour la fonction `appartient` :

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

On remarque que cette fonction est récursive terminale grâce à l'évaluation paresseuse (si le test `t=x` vaut `true`, l'appel récursif n'est pas effectué).

iii. Pour la fonction `plus_petit_absent` :
On teste successivement tous les entiers en incrémentant une référence tant que sa valeur est présente.

```
let plus_petit_absent l =
  let i = ref 0 in
    while (appartient l !i) do
      i := !i+1
    done;
  !i;;
```

iv. Pour la fonction `couleurs_voisins` :
La fonction auxiliaire locale `construit : int list -> int list` prend en argument une liste de sommets et renvoie la liste des couleurs de ceux-ci (ceux qui sont coloriés). Il suffit de l'appeler avec la liste des voisins de i. Cette fonction étant locale, elle connaît le tableau `couleurs`. L'énoncé ne précise pas si la liste résultat peut comporter des doublons (rien n'empêche un sommet d'avoir plusieurs voisins de la même couleur). La fonction suivante impose une liste résultat sans doublon (d'où le test d'appartenance).

```
let couleur_voisins aretes couleurs i =
  let rec construit l = match l with
    | [] -> []
    | j::q -> let listcoul = construit q in
          if couleurs.(j) <> (-1) &&
                    not (appartient listcoul couleurs.(j))
            then couleurs.(j)::listcoul
            else listcoul
  in construit aretes.(i);;
```

v. Pour la fonction `couleur_disponible` :
Il suffit de combiner les deux fonctions précédentes.

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

(e) i. Si G ne possède pas d'arêtes, alors (de façon immédiate)

$$\chi(G) = 1 \qquad \text{et} \qquad \omega(G) = 1.$$

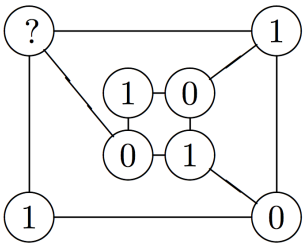
Si G est complet à n sommets alors (de façon immédiate)

$$\chi(G) = n \qquad \text{et} \qquad \omega(G) = n.$$

ii. Une bonne coloration d'un graphe G induit une bonne coloration de tout sous-graphe de G. Avec la question précédente, on en déduit que :

$$\omega(G) \leqslant \chi(G).$$

L'égalité est fausse en général. Dans le graphe suivant, il n'y a pas de triangles et donc pas de clique de taille 3. Cependant, on vérifie aisément que trois couleurs sont indispensables pour le colorier correctement. Pour le voir, on donne la couleur 0 à un sommet et on complète de manière obligatoire en n'utilisant que 0 ou 1 pour tomber sur une contradiction.



iii. Il s'agit de voir si, pour tout élément `i` de `xs`, la liste `aretes.(i)` contient bien tous les autres éléments de `xs`. On utilise pour cela deux fonctions auxiliaires locales (qui connaîtront donc `aretes`) :
— `tester : int list -> int -> bool` : dans l'appel `tester l i` on regarde si tous les éléments de `l` différents de `i` sont voisins de `i`.
— `parcourir : int list -> bool` : dans l'appel `parcourir xs` on teste si chaque élément de `xs` est relié à tous les autres.

```
let est_clique aretes xs =
  let rec tester l i = match l with
    | [] -> true
    | j::q -> if j=i then tester q i
                else (appartient aretes.(i) j) && (tester q i)
  in
    let rec parcourir xs = match xs with
      | [] -> true
      | i::q -> tester xs i
    in
      parcourir xs ;;
```

2. (a) Dans le problème B, certaines extrémités gauches de segments sont égales. On voit sur cet exemple que l'ordre des intervalles n'est donc pas parfaitement défini. En choisissant la numérotation proposée par l'énoncé, on obtient la coloration

$$(0, 1, 2, 0, 1, 0, 2, 1, 0).$$

(b) Notons que dans la fonction demandée, l'argument `segments` ne sert à rien puisque tous les renseignements sont contenus dans le graphe (alternativement, on pourrait se contenter de `segments` à partir de qui on sait construire le graphe).
Il nous suffit de créer un tableau de couleurs de bonne taille (initialisé avec l'absence de couleur -1) et de le remplir petit à petit grâce aux fonctions de 1.(d).

```
let coloration segments aretes =
  let n = Array.lentgh aretes in
    let couleur = Array.make n (-1) in
      couleurs.(0) <- 0;
      for i = 1 to n-1 do
        let c = couleur_disponible aretes couleurs i in
          couleurs.(i) <- c
      done;
    couleurs
;;
```

(c) i. Puisque I_k s'est vu attribuer la couleur c, il est en conflit avec des intervalles ayant reçu les couleurs $0, \dots, c-1$. Avec l'ordre choisi sur les extrémités gauches des segments, ceci signifie que a_k est inférieur ou égal à au moins c des entiers $b_0, \dots, b_{k-1}$. a_k appartient donc à coup sûr à au moins c des segments $I_0, \dots I_{k-1}$.

ii. Considérons l'ensemble C formé des intervalles I_i tels que $i \leqslant k$ et $a_k \in I_i$. On vient de voir que cet ensemble est au moins de cardinal $c+1$ (il y a I_k et c autres des précédents intervalles). Il forme une clique. En effet, considérons $i < j$ des éléments tels que $I_i, I_j \in C$. On a $a_i \leqslant a_j \leqslant a_k$ puisque les intervalles sont ordonnés selon la borne inférieure. De plus, $a_k \leqslant b_i$ (par définition de C) et donc $a_i \leqslant a_j \leqslant b_i$. Ainsi, I_i et I_j sont en conflit et donc reliés dans le graphe.

iii. La question 1.(e).(ii) indique alors que le nombre chromatique est plus grand que $c+1$.

iv. Quand on introduit une couleur, son numéro est toujours inférieur au nombre chromatique. Par ailleurs, la coloration partielle est toujours une bonne coloration (c'est un invariant d'itération évident puisqu'une couleur ajouté ne contredit pas la bonne coloration). On obtient donc finalement une coloration avec un nombre de couleurs inférieur au nombre chromatique. Il est égal à $\chi(G)$ par minimalité de $\chi(G)$ et c'est une coloration optimale.

(d) La fonction `appartient` est de complexité $O(p)$ où p est la longueur de la liste argument.

La fonction `plus_petit_absent` met en oeuvre une boucle effectuée au plus $p+1$ fois où p est la longueur de la liste argument (en effet, par le lemme des tiroirs, un des entiers $0, \dots, p$ est absent de la liste). Chaque itération est de complexité $O(p)$. Le coût total est donc $O(p^2)$.

La fonction `couleurs_voisins` parcourt la liste des voisins d'un sommet. A chaque élément rencontré, on effectue $O(n)$ opérations où n est un majorant du nombre de couleurs qu'on utilisera, par exemple le nombre de sommets (c'est l'appel à `appartient` qui induit ce coût). On a donc une complexité $O(m_i n)$ où m_i est le nombre de voisins de i et n le nombre total de sommets.

La fonction `couleur_disponible` a donc un coût $O(m_i n) + O(n)$ (comme on a formé une liste de couleurs sans doublon, on appelle `plus_petit_absent` avec une liste d'au plus n éléments) et donc $O((m_i + 1)n)$.

La fonction `coloration` initialise un tableau pour un coût $O(n)$ où n est le nombre de sommets. On fait un appel à `couleur_disponible` pour un coût global $\sum_{i=0}^{n-1} O((m_i+1)n) = O(mn) + O(n)$. Le coût global est donc $O(mn) + O(n)$.
