Corrigé

option informatique

Coloration d'un graphe

Question 1.

```
let coloration_valide g couleur =
  let rec aux = function
    | i when i = vect_length g -> true
    | i -> for_all (function j -> couleur.(j) <> couleur.(i)) g.(i) && aux (i+1)
  in aux 0 ;;
```

Cette fonction est de coût $O(n+p)$: chaque liste d'adjacence est parcourue une fois.

Question 2.

```
let biparti g =
  let n = vect_length g in
  let couleur = make_vect n (-1) in
  let rec aux c i = match couleur.(i) with
    | -1           -> couleur.(i) <- c ;
                      do_list (aux (1-c)) g.(i)
    | k when k = c -> ()
    | _            -> failwith "biparti"
  in for i = 0 to n-1 do
       if couleur.(i) = -1 then aux 0 i
     done ;
     couleur ;;
```

Pour les mêmes raisons, cette fonction est de coût $O(n+p)$.

Question 3.

```
let glouton g =
  let n = vect_length g in
  let couleur = make_vect n (-1) in
  let rec aux i = function
    | c when for_all (function j -> couleur.(j) <> c) g.(i) -> c
    | c                                                     -> aux i (c+1)
  in for i = 0 to n-1 do
       couleur.(i) <- aux i 0
     done ;
     couleur ;;
```

La fonction `aux i` recherche la plus petite couleur admissible pour colorer le sommet i.
Appliquée au graphe de la figure 1, cette fonction retourne la 3-coloration `[|0; 0; 0; 1; 2; 1; 0; 1|]` qui n'est pas optimale puisque le graphe est biparti.

Question 4.

a) Lorsqu'un sommet v_i reçoit la couleur k par l'algorithme glouton, c'est que l'extrémité gauche a_i de l'intervalle v_i appartient à au moins $k+1$ intervalles distincts. Mais ces $k+1$ intervalles sont deux à deux sécants (ils contiennent tous a_i), donc le sous-graphe restreint à ces intervalles et complet et nécessite au moins $k+1$ couleurs distinctes pour être colorié.

b) Considérons une k-coloration optimale c de G, puis ordonnons les sommets de G en numérotant d'abord les sommets de couleur 0, puis les sommets de couleur 1, etc. Alors l'algorithme glouton appliqué à G retourne la coloration c.

Question 5. La fonctionnelle **exists** est de type *('a -> bool) -> 'a list -> bool*.

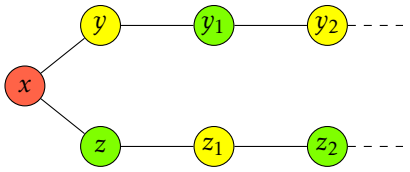
```
let ds g couleur v =
    let rec aux = function
    | [] -> 0
    | t::q when couleur.(t) = -1                                    -> aux q
    | t::q when exists (function j -> couleur.(j) = couleur.(t)) q -> aux q
    | _::q                                                          -> 1 + aux q
    in aux g.(v) ;;
```

```
let satmax g couleur =
  let rec aux acc = function
    | i when i = vect_length g    -> acc
    | i when couleur.(i) >= 0     -> aux acc (i+1)
    | i when ds g couleur i > acc -> aux i (i+1)
    | i                           -> aux acc (i+1)
  in aux (-1) 0 ;;
```

```
let dsatur g =
  let n = vect_length g in
  let couleur = make_vect n (-1) in
  let rec aux i = function
    | c when for_all (function j -> couleur.(j) <> c) g.(i) -> c
    | c                                                     -> aux i (c+1)
  in for i = 0 to n-1 do
       let s = satmax g couleur in
       couleur.(s) <- aux s 0
     done ;
     couleur ;;
```

Question 6.

a) Considérons un graphe biparti et supposons que l'algorithme DSATUR retourne une 3-coloration. Considérons le premier sommet x tel que $d_s(x) = 2$. Ce sommet est donc au moins de degré 2 et a deux voisins y et z de couleurs différentes, qui ont été colorés avant x. Compte tenu de l'ordonnancement des sommets, ils sont tous deux de degré au moins égal à 2, ce qui permet de proche en proche de construire deux chaînes issues de x et de couleurs alternées.



Le graphe n'ayant qu'un nombre fini de sommet, ces deux chaînes se rejoignent, ce qui construit un cycle d'ordre impair dans G, ce qui n'est pas possible dans le cas d'un graphe biparti.

b) Appliqué au graphe de l'énoncé l'algorithme DSATUR retourne la 4-coloration `[|0; 1; 1; 0; 2; 3; 2; 2|]` qui n'est pas optimale puisqu'une 3-coloration existe :