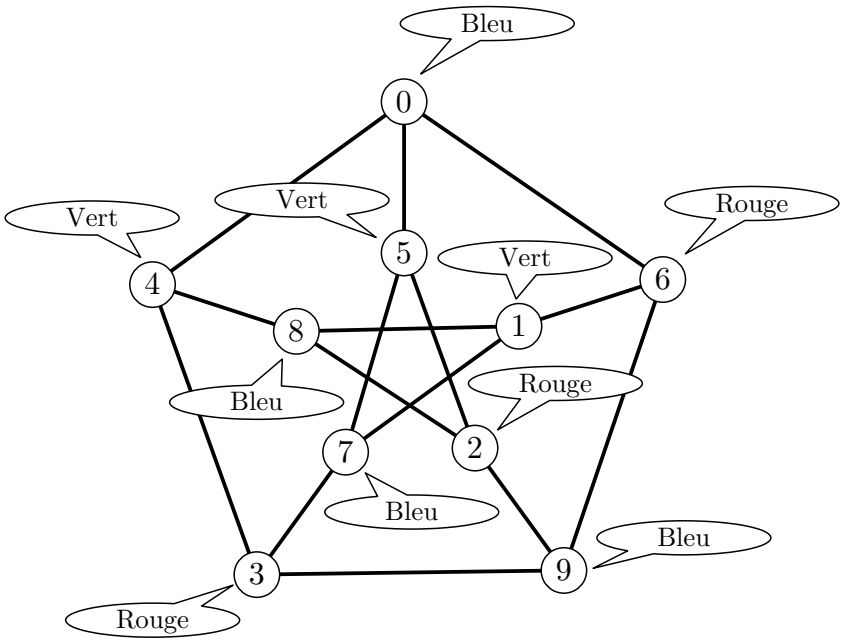
```
(* Les deux types suivants sont copiés-collés depuis la page 2 de l'énoncé *)
type graphe = bool array array;;
type etiquetage = int array;;
```

```
(* En haut de la page 3, l'énoncé évoque des fonctions Caml Light. En Ocaml, ces
   fonctions peuvent être définis par: *)
let make_vect = Array.make;;
let vect_length = Array.length;;
let list_length = List.length;;
let vect_of_list = Array.of_list;;
let range (n: int): int array =  Array.init n (fun i -> i);;
```

**Question 1** – Le graphe de gauche n'est pas colorié. En effet, les deux sommets rouges sont voisins et ne respectent donc pas la condition (1) de l'énoncé. En revanche, le graphe de droite est colorié car il respecte la condition (1) de l'énoncé.

**Question 2** – Voici une 3-coloration possible :




De plus, ce graphe ne peut pas être 2-colorié. En effet, si on essaie de le colorier avec du Bleu et du Rouge, alors le cycle $(0, 5, 2, 9, 6, 0)$ pose problème. Si 0 est colorié en Bleu (le cas où 0 est colorié en Rouge se traite de manière similaire), alors 5 doit être Rouge, 2 doit être Bleu, 9 doit être Rouge et 6 doit être Bleu. Les sommets 0 et 6 sont voisins et ont la même couleur ce qui contredit la condition (1) de l'énoncé. Finalement :

Le nombre chromatique du graphe de Petersen est 3.

**Question 3** –

```
1  let est_col (gphe: graphe) (etiq: etiquetage): bool =
2    if Array.length gphe > Array.length etiq then false else begin
3        let n = Array.length gphe in
4        let res = ref true in
5        for i = 0 to n-1 do
6          for j = 0 to n-1 do
7            if gphe.(i).(j) && etiq.(i) = etiq.(j) then res := false
8          done;
9        done;
10       !res
11     end;;
```

Pour la complexité :

→ Lignes 5 et 6 : on a deux boucles `for` imbriquées chacune faisant $n$ tours. Chaque tour s'exécute en temps constant.

→ Toutes les autres opérations s'exécutent en temps constant.

Ainsi, le temps d'exécution est bien quadratique en le nombre de sommets du graphe.

**Question 4** – Soit $G$ un graphe avec $n$ sommets. On remarque que si $G$ contient une boucle, il n'est pas coloriable quel que soit le nombre de couleurs. Sinon, le nombre chromatique de $G$ est au plus $n$ (on obtient une $n$-coloration en associant une couleur différente à chaque sommet). Soit $k \leqslant n$. Pour déterminer si $G$ peut être colorié avec $k$ couleurs, on peut adopter une stratégie force-brute consistant à générer les $k^n$ étiquetages à $k$ couleurs et à vérifier pour chacun d'entre eux s'il s'agit d'une $k$-coloration (en temps quadratique avec la fonction de la question précédente).

Si $n \in \mathbb{N}^*$ et $k \in \mathbb{N}^*$ sont fixés, il est possible de générer un tableau « `toutes_etiq: int array array` » de taille $k^n$ contenant tous les tableaux « `etiq: int array` » de taille $n$ dont les éléments appartiennent à $[\![0; k-1]\!]$. En effet, pour chaque `i` $\in [\![0; \texttt{k}^{\texttt{n}} - 1]\!]$, il suffit de stocker dans `toutes_etiq.(i)` la décomposition de `i` en base `k`. La génération de `toutes_etiq` se fait en temps $\mathcal{O}(nk^n)$. Dans la suite, on aura $k \leqslant n$ et donc la génération de `toutes_etiq` se fera en temps $\mathcal{O}(n^{n+1})$.

Voici le pseudo-code pour calculer le nombre chromatique d'un graphe : :

→ Soit $G$ un graphe avec $n \in \mathbb{N}^*$ sommets et représenté en OCaml par « `gphe: graphe` ».

→ Si $G$ contient une boucle, il n'est pas coloriable.

→ Sinon, pour $k$ variant de 1 à $n$ (le nombre chromatique est au plus $n$) :

- On génère le tableau `toutes_etiq` associé à $n$ et $k$.
- Pour chaque élément `etiq` de `toutes_etiq` :
  - Si « `est_col gphe etiq` » s'évalue en `true`, on arrête la fonction et on déclare que le nombre chromatique est $k$.
  - Sinon, on passe au tour de boucle suivant.

Le temps d'exécution de cette procédure est en $\mathcal{O}\Big(n \times (n^{n+1} + n^n.n^2)\Big) = \mathcal{O}(n^{n+3})$. Or $n^{n+3} = 2^{g(n)}$ avec $g(n) = (n+3)\log_2(n) = \mathcal{O}(n^2)$. La fonction $g$ étant polynomiale, le temps d'exécution est bien exponentiel en $n$.

**Question 5** – On le montre par double implication :

($\Rightarrow$) Supposons que $G$ soit biparti : $G = (T \uplus U, A)$. Soit :

$$L : \begin{cases} T \uplus U \to \{0,1\} \\ s \mapsto 0 \text{ si } s \in T \\ s \mapsto 1 \text{ si } s \in U \end{cases}$$

Comme $G$ est biparti, chacune de ses arêtes a une extrémité dans $T$ et une extrémité dans $U$, donc deux sommets voisins ne peuvent pas être de la même couleur. Ainsi, $L$ est un coloriage de $G$ avec 2 couleurs.

($\Leftarrow$) Si $L : S \to \{0, 1\}$ est un coloriage de $G$ avec 2 couleurs, on pose :

$$T = \Big\{ s \in S : L(s) = 0 \Big\} \qquad\qquad U = \Big\{ s \in S : L(s) = 1 \Big\}$$

Alors $S = T \uplus U$. Comme $L$ est un coloriage, deux sommets de $T$ (resp. $U$) ne peuvent pas être voisins. En d'autres termes, chaque arête a une extrémité dans $T$ et l'autre dans $U$. Donc $G$ est biparti.

**Question 6** –

```
1  (* Dans la fonction parc_prof:
2     - s est le prochain sommet à parcourir
3     - c = 0 ou c = 1 est la couleur à utiliser pour colorer s
4     Si le graphe n'est pas 2-coloriable, la fonction déclenche une erreur *)
5  let deux_col (gphe: graphe): etiquetage =
6    let n = Array.length gphe in
7    let etiq = Array.make n (-1) in
8    let rec parc_prof (s: int) (c: int): unit = match etiq.(s) with
9      | -1 -> etiq.(s) <- c;
10             for t = 0 to n-1 do
11               if gphe.(s).(t) then parc_prof t (1-c)
12             done;
13     | c2 when c2 = c -> ()
14     | c2 -> failwith "Le graphe n'est pas 2-coloriable"
15   in
16   for s = 0 to n-1 do
17     if etiq.(s) = -1 then parc_prof s 0
18   done;
19   etiq;;
```

Pour la complexité, on remarque que pour chaque sommet $s$, le test de la ligne 9 ne peut être vrai qu'une seule fois (pour $s$) au cours de l'exécution. Ainsi, la ligne 11 est exécutée au plus $n^2$ fois avec $n$ le nombre de sommets. La fonction `parc_prof` est donc appelée au plus $n^2 + n = \mathcal{O}(n^2)$ fois (ligne 11 + ligne 17). Ainsi, chaque ligne de la fonction `parc_prof` est exécutée $\mathcal{O}(n^2)$ fois.

En résumé, on a un appel à `Array.make` (ligne 7) en temps $\mathcal{O}(n)$ ; toutes les autres opérations sont en temps constant et s'exécutent $\mathcal{O}(n^2)$ fois chacune. Donc :

| Le temps d'exécution total est bien en $\mathcal{O}(n^2)$ |
|---|

**Question 7** –

Pour `num1`, on utilise trois couleurs :

| $s$ | $C(s)$ | $c$ |
|---|---|---|
| 1 | {} | 0 |
| 3 | {} | 0 |
| 4 | {0} | 1 |
| 0 | {1} | 0 |
| 2 | {} | 0 |
| 6 | {0} | 1 |
| 5 | {0} | 1 |
| 9 | {0; 1} | 2 |
| 8 | {0; 1} | 2 |
| 7 | {0; 1} | 2 |

| $s$ | $L(s)$ |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1 |
| 6 | 1 |
| 7 | 2 |
| 8 | 2 |
| 9 | 2 |

Pour `num2`, on utilise quatre couleurs :

| $s$ | $C(s)$ | $c$ |
|---|---|---|
| 0 | {} | 0 |
| 7 | {} | 0 |
| 2 | {} | 0 |
| 5 | {0} | 1 |
| 4 | {0} | 1 |
| 6 | {0} | 1 |
| 8 | {0; 1} | 2 |
| 1 | {0; 1; 2} | 3 |
| 3 | {0; 1} | 2 |
| 9 | {0; 1; 2} | 3 |

| $s$ | $L(s)$ |
|---|---|
| 0 | 0 |
| 1 | 3 |
| 2 | 0 |
| 3 | 2 |
| 4 | 1 |
| 5 | 1 |
| 6 | 1 |
| 7 | 0 |
| 8 | 2 |
| 9 | 3 |

**Question 8 –**

```
1  (* col.(i) vaut true si la couleur i a été utilisée *)
2  let min_couleur_possible (gphe: graphe) (etiq: etiquetage) (s: int): int =
3    let n = Array.length gphe in
4    let col = Array.make n false in
5    for t = 0 to n-1 do
6      if gphe.(s).(t) && etiq.(t) >= 0 then col.(etiq.(t)) <- true
7    done;
8    let c = ref 0 in
9    while !c < Array.length col && col.(!c) do
10     incr c
11   done;
12   !c;;
```

Pour la complexité, on remarque que la boucle `for` fait exactement $n$ tours et que la boucle `while` en fait au plus $n$. De plus, la ligne 4 s'exécute en temps $\mathcal{O}(n)$ et les autres lignes en temps constant. Donc :

On a bien une complexité en $\mathcal{O}(n)$

**Question 9 –**

```
1  let glouton (gphe: graphe) (num: int array): etiquetage =
2    let n = Array.length gphe in
3    let etiq = Array.make n (-1) in
4    for k = 0 to n-1 do
5      let s = num.(k) in
6      etiq.(s) <- min_couleur_possible gphe etiq s;
7    done;
8    etiq;;
```

Pour la complexité :

→ La ligne 3 s'exécute en temps $\mathcal{O}(n)$

→ La ligne 6 s'exécute $n$ fois et chaque exécution est en temps $\mathcal{O}(n)$.

→ Les autre lignes s'exécutent en temps constant.

La complexité est bien en $\mathcal{O}(n^2)$

**Question 10 –**

**Remarque.** L'énoncé semble supposer implicitement qu'il n'y a pas de boucle dans le graphe (arête qui relie un sommet à lui-même). En effet, si $G$ contient une boucle, alors il n'existe pas de coloriage pour $G$ et le résultat de la question 10 est faux. À partir de maintenant, on suppose qu'il n'y a pas de boucle dans les graphes considérés (je ne crois pas que cette hypothèse apparaisse dans l'énoncé).

Pour montrer la correction de la fonction, on utilise un invariant de boucle. Les tours sont indicés par la variable $\texttt{k} \in [\![\texttt{0}\,;\texttt{n}-\texttt{1}]\!]$. Pour chaque `k`, on pose :

$$(\mathcal{P}_{\texttt{k}}) : \begin{cases} \text{À la fin du tour de boucle d'indice } \texttt{k}, \text{ si on note } \texttt{T} = \left\{\texttt{num.(i)} : \texttt{i} \in [\![\texttt{0}, \texttt{k}]\!]\right\}, \\ \text{alors les sommets de } \texttt{T} \text{ sont coloriés avec des couleurs de } [\![\texttt{0}, \texttt{d}]\!] \text{ et deux} \\ \text{sommets voisins n'ont pas la même couleur.} \end{cases}$$

Montrons $\mathcal{P}_k$ par itération finie.

Initialisation. Pour $k = 0$, à la fin du premier tour de boucle, le sommet `num.(0)` a la couleur `0` et les autres sommets n'ont pas de couleur. Donc $\mathcal{P}_0$ est vrai.

Hérédité. Soit $k \in [\![1; n-1]\!]$. On suppose $\mathcal{P}_{k-1}$ et on montre $\mathcal{P}_k$. Lors du tour de boucle d'indice $k$, on colorie le sommet `s = num.(k)`. On remarque que lorsque la fonction `min_couleur_possible` est appliquée à `s` :

→ Elle renvoie une couleur qui n'apparaît pas chez les voisins de `s`. Ainsi, `s` est étiqueté par une couleur qui n'apparaît pas chez ses voisins.

→ Elle renvoie une couleur $\leqslant d$ puisque `s` a au plus $d$ voisins qui ne peuvent donc pas utiliser toutes les couleurs de $[\![0; d]\!]$.

→ Étant donné qu'il n'y a pas de boucle dans le graphe, on a $d \leqslant n-1$ et donc l'étiquetage donné en entrée est bien à valeur dans $\{-1, \dots, n-1\}$ (ainsi, l'hypothèse de l'énoncé de la question 8 est bien vérifiée).

En combinant ces remarques avec $(\mathcal{P}_{k-1})$, on obtient $(\mathcal{P}_k)$.

Conclusion. Par le principe de récurrence, $(\mathcal{P}_k)$ est vraie pour tout $k \in [\![0; n-1]\!]$. En particulier, à la fin du dernier tour de boucle, $(\mathcal{P}_{n-1})$ indique qu'on obtient un coloriage utilisant au plus $d+1$ couleurs.

**Question 11** –

• Soit $L$ un coloriage de $G$. On numérote les sommets $s$ de $G$ par ordre croissant de $L(s)$. En d'autres termes, on écrit :

$$S = \{s_0, s_1, \dots, s_{n-1}\} \qquad \text{avec} \qquad L(s_0) \leqslant L(s_1) \leqslant \dots \leqslant L(s_{n-1})$$

On donne la liste `[s₀; s₁; ...; sₙ₋₁]` en entrée de la fonction `glouton` qui renvoie un coloriage $L'$. Montrons par itération finie sur $k \in [\![0, n-1]\!]$ que $L'(s_k) \leqslant L(s_k)$.

Initialisation et Hérédité. Soit $k \in [\![0, n-1]\!]$. On suppose $L'(s_{k'}) \leqslant L(s_{k'})$ pour tout $k' < k$ et on montre $L'(s_k) \leqslant L(s_k)$. Lors du tour de boucle d'indice $k$, on s'intéresse à l'ensemble :

$$C'(s_k) = \Big\{ L'(t) \mid t \in V(s_k) \cap \{s_0, s_1, \dots, s_{k-1}\} \Big\}$$

ou bien à $C'(s_k) \cup \{-1\}$ si au moins un voisin de $s_k$ n'a pas encore de couleur. Dans les deux cas, on obtient :

$$L'(s_k) = \min\Big(\mathbb{N} \setminus C'(s_k)\Big)$$

Si $C'(s_k) = \varnothing$, alors $L'(s_k) = 0 \leqslant L(s_k)$. Sinon, soit $t_0 \in V(s_k) \cap \{s_0, s_1, \dots, s_{k-1}\}$ tel que $L'(t_0) = \max\Big(C'(s_k)\Big)$. Alors :

$$L'(s_k) \leqslant L'(t_0) + 1.$$

Comme $t_0 \in \{s_0, s_1, \dots, s_{k-1}\}$, on a $L(t_0) \leqslant L(s_k)$. Par hypothèse de récurrence, on a aussi $L'(t_0) \leqslant L(t_0)$. De plus, comme $t_0$ est un voisin de $s_k$, on a $L(t_0) \neq L(s_k)$ et donc $L(t_0) < L(s_k)$. Finalement :

$$L'(s_k) \leqslant L'(t_0) + 1 \leqslant L(t_0) + 1 < L(s_k) + 1$$

Les quantités considérées sont des entiers donc $L'(s_k) \leqslant L(s_k)$. D'où le résultat.

• Soit $L$ un coloriage optimal et $k$ le nombre de couleurs utilisées. Quitte à renommer les couleurs, on peut supposer que les couleurs utilisées sont $0, 1, \dots, k-1$ (chaque couleur étant utilisée au moins une fois).

D'après ce qui précède, il existe un ordre de numérotation des sommets tel que le coloriage glouton $L'$ associé vérifie $L'(s) \leqslant L(s) \leqslant k-1$ pour tout sommet $s$ de $G$. Le nombre de couleurs utilisées dans $L'$ noté $k'$ vérifie donc $k' \leqslant k$. Par optimalité de $L$, on a $k' = k$, d'où le résultat.

**Question 12** –

```
(* Renvoie le degre de s *)
let degre (gphe: graphe) (s: int): int =
  let n = Array.length gphe in
  let d = ref 0 in
  for t = 0 to n-1 do
    if gphe.(s).(t) then incr d
  done;
  !d;;
```

```
(* deg_inv.(d) contient la liste de tous les sommets de degré d *)
let degres_inverse (gphe: graphe): int list array =
  let n = Array.length gphe in
  let deg_inv = Array.make n [] in
  (* downto pour que les listes soient dans l'ordre croissant (facultatif) *)
  for s = n-1 downto 0 do
    let d = degre gphe s in
    deg_inv.(d) <- s :: deg_inv.(d)
  done;
  deg_inv;;
```

```
let tri_degre (gphe: graphe): int array =
  let n = Array.length gphe in
  let deg_inv = degres_inverse gphe in
  let num = Array.make n (-1) in
  let k = ref 0 in
  for d = n-1 downto 0 do
    List.iter (fun s -> num.(!k) <- s;
                        incr k)
      deg_inv.(d);
  done;
  num;;
```

On a utilisé un algorithme de tri par paquets puisque c'est un algorithme de tri simple et efficace (lorsqu'on peut l'appliquer). Ici il est possible d'utiliser cet algorithme car on sait que le degré est compris entre 0 et $n-1$ avec $n$ le nombre de sommets.

```
let welsh_powell (gphe: graphe): etiquetage =
  let num = tri_degre gphe in
  glouton gphe num;;
```

**Question 13** – On suppose que $G$ est $(k+1)$-coloriable. Soit $L : V \to [\![0, k]\!]$ une $(k+1)$-coloration de $G$ et $s$ un sommet. On s'intéresse à $G'$ le graphe induit par $V(s)$ et à $L'$ la restriction de $L$ à $V(S)$. On vérifie facilement que $L'$ est une coloration de $G'$. Comme $L$ est une coloration, pour tout $t \in V(s)$ :

$$L'(t) = L(t) \neq L(s).$$

Ainsi, $L'$ est à valeurs dans $[\![0, k]\!] \setminus \{L(t)\}$, c'est donc une $k$-coloration de $G'$ qui est $k$-coloriable.

**Question 14** –

• Soit $L : S \to \mathbb{N}$ la fonction obtenue par l'algorithme de Wigderson. Montrons que $L$ est bien une coloration. On remarque dans un premier temps que l'étape (3) garantie que $L(s)$ est définie pour tout sommet $s$. Soient $u$ et $v$ deux sommets voisins dans $G$, on veut montrer que $L(u) \neq L(v)$ :

→ Si $u$ et $v$ ont été coloriés à l'étape (3) alors la correction de l'algorithme glouton (question 10) garantit qu'il n'ont pas la même couleur (puisqu'ils sont aussi voisins dans le sous-graphe de l'étape (3)).

→ Si on est dans l'un des deux cas suivants :

- $u$ a été colorié à l'étape (2)($a$) et $v$ à l'étape (3) (ou inversement).
- $u$ et $v$ ont été coloriés à l'étape (2)($b$), mais pas lors du même tour de la boucle « Pour chaque ... » (de l'étape (2)).

Alors, l'étape (2)($b$) garantit que les couleurs utilisées pour $u$ et $v$ sont différentes.

→ Sinon, $u$ et $v$ ont été coloriés à la même étape (2)($b$). Alors, il existe $s \in S$ tel que $u$ et $v$ font parti du sous-graphe induit par $V(s)$ noté $G'$. Par la question 13, $G'$ est 2-coloriable. Comme les sommets $u$ et $v$ sont également voisins dans $G'$, les couleurs qui leur sont attribuées sont différentes.

Finalement :

| $L$ est bien un coloriage. |
|---|

• On remarque que lors d'un tour de boucle de l'étape (2), au moins $\sqrt{n}$ sommets sont coloriés. Si on note $k$ le nombre de tours, on obtient $k \times \sqrt{n} \leqslant n$ et donc $k \leqslant \sqrt{n}$. Finalement, lors de l'étape (2), on utilise au plus $2\sqrt{n}$ couleurs.

Lors de l'étape (3), tous les sommets considérés ont un degré inférieur ou égal à $d_0 = \lfloor \sqrt{n} \rfloor$ dans le graphe induit. D'après la question 10, l'algorithme glouton utilise au plus $d_0 + 1$ couleurs.

Finalement :

| Le nombre de couleurs utilisées est inférieur ou égal à $3\sqrt{n} + 1 = \mathcal{O}(\sqrt{n})$. |
|---|

**Question 15** –

```
let sous_graphe (gphe: graphe) (sg: int array): graphe =
  let m = Array.length sg in
  let sous_gphe = Array.make_matrix m m false in
  for s = 0 to m-1 do
    for t = 0 to m-1 do
      if gphe.(sg.(s)).(sg.(t)) then sous_gphe.(s).(t) <- true
    done;
  done;
  sous_gphe;;
```

**Question 16** –

```
let voisins_non_colories (gphe: graphe) (etiq: etiquetage) (s: int): int list =
  let n = Array.length gphe in
  let rec aux t = match () with
    | _ when t = n -> []
    | _ when gphe.(s).(t) && etiq.(t) = -1 -> t :: aux (t+1)
    | _ -> aux (t+1)
  in
  aux 0;;
```

```
let degre_non_colories (gphe: graphe) (etiq: etiquetage) (s: int): int =
  List.length (voisins_non_colories gphe etiq s);;
```

**Question 17 –**

```
let non_colories (gphe: graphe) (etiq: etiquetage): int list =
  let n = Array.length gphe in
  let rec aux t = match () with
    | _ when t = n -> []
    | _ when etiq.(t) = -1 -> t :: aux (t+1)
    | _ -> aux (t+1)
  in
  aux 0;;
```

**Question 18 –**

```
(* Indique si 1 apparaît dans tab *)
let un_appartient (tab: int array): bool =
  let n = Array.length tab in
  let i = ref 0 in
  while !i < n && tab.(!i) <> 1 do
    incr i
  done;
  !i < n;;
```

```
(* Transfert les couleurs de etiq_sg vers etiq en ajoutant c à chaque couleur *)
let transfert (sg: int array) (etiq: int array) (etiq_sg: int array) (c: int): unit =
  let m = Array.length sg in
  for s = 0 to m-1 do
    etiq.(sg.(s)) <- etiq_sg.(s) + c;
  done;;
```

```
let wigderson_etape2 (gphe: graphe) (c: int ref) (etiq: etiquetage): unit =
  let n = Array.length gphe in
  let sqrt_n = int_of_float (ceil (sqrt (float_of_int n))) in
  for s = 0 to n-1 do
    if degre_non_colories gphe etiq s >= sqrt_n then begin
        let voisins_s = voisins_non_colories gphe etiq s in
        let sg = Array.of_list voisins_s in
        let sous_gphe = sous_graphe gphe sg in
        let etiq_sg = deux_col sous_gphe in
        transfert sg etiq etiq_sg !c;
        incr c;
        if un_appartient etiq_sg then incr c;
      end
  done;;
```

```
let wigderson (gphe: graphe): etiquetage =
  let n = Array.length gphe in
  let c = ref 0 in
  let etiq = Array.make n (-1) in
  wigderson_etape2 gphe c etiq;
  let sg = Array.of_list (non_colories gphe etiq) in
  let sous_gphe = sous_graphe gphe sg in
  let etiq_sg = glouton sous_gphe (range (Array.length sous_gphe)) in
  transfert sg etiq etiq_sg !c;
  etiq;;
```

Pour la complexité :

→ La fonction `sous_graphe` s'exécute en temps $\mathcal{O}(n^2)$. En effet, la ligne 3 est exécutée une fois en temps $\mathcal{O}(m^2) = \mathcal{O}(n^2)$. Il y a deux boucles imbriquées faisant chacune $n$ tours, chaque tour s'exécute en temps $\mathcal{O}(n)$, soit un temps total en $\mathcal{O}(n^2)$. Les autres opérations sont en temps constant.

→ La fonction `voisins_non_colories` s'exécute en temps $\mathcal{O}(n)$. En effet, dans la fonction `aux`, l'entier donné en entrée augmente de `1` à chaque appel récursif. Ainsi, le nombre d'appels à la fonction `aux` est au plus $n+1$. De plus, toutes les opérations s'exécutent en temps constant.

→ La fonction `degre_non_colories` s'exécute en temps $\mathcal{O}(n)$. En effet, on a un appel à la fonction `voisins_non_colories` et un appel à `List.length` qui s'exécute en temps linéaire en la taille de la liste en entrée.

→ La fonction `non_colories` s'exécute en temps $\mathcal{O}(n)$. L'argument est le même que pour la fonction `voisins_non_colories`.

→ La fonction `un_appartient` s'exécute en temps $\mathcal{O}(n)$. En effet, il y a au plus $n$ tours de boucle et toutes les opérations sont en temps constant.

→ La fonction `transfert` s'exécute en temps $\mathcal{O}(n)$ (même justification).

→ Pour la fonction `wigderson_etape2`, le temps d'exécution est en $\mathcal{O}(n^3)$. En effet, on a $n$ tour de boucle, les appels aux fonctions `degre_non_colories`, `voisins_non_colories`, `Array.of_list`, `transfert`, `un_appartient` s'exécutent en temps $\mathcal{O}(n)$, et les appels aux fonctions `sous_graphe`, `deux_col` s'exécutent en temps $\mathcal{O}(n^2)$. Toutes les autres opérations sont en temps constant.

→ Pour la fonction `wigderson`, on appelle les fonctions précédentes. On obtient une complexité en $\mathcal{O}(n^3)$.

| On obtient une complexité en $\mathcal{O}(n^3)$. |
|---|

**Question 19** – Lorsque le nombre chromatique du graphe est connu, on pourrait appliquer une variante récursive de l'algorithme de Wigderson. La fonction prendrait en entrée un argument supplémentaire $M$ tel que $M \geqslant \chi$ avec $\chi$ le nombre chromatique du graphe :

(I) Si $M \leqslant 2$, on 2-colorie le graphe avec l'algorithme de la partie 2.

(II) Sinon, on suit les étapes de l'algorithme de Wigderson :

(1) On se donne comme valeur 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 appelle récursivement la fonction sur le sous-graphe induit par l'ensemble des voisins de $s$ pas encore coloriés et sur $M-1$, en utilisant les couleurs $c, c+1, c+2, \ldots$

(b) On incrémente $c$ du nombre de couleurs utilisées en (a).

(3) On applique l'étape (3) de l'algorithme de Wigderson donnée dans l'énoncé.