Exercice

1) On obtient la matrice suivante.

	0	1	2	3	4
0	0	0	0	0	0
1	∞	4	1	1	1
2	∞	∞	7	-2	-2
3	∞	-1	-1	-1	-1
4	∞	∞	1	1	1

Par exemple $d(1,2) = 1$. En effet, on considère les arêtes qui arrivent au sommet 1 qui sont les arêtes $(0,1)$ de poids 4 et $(3,1)$ de poids 2. On en déduit que

$$d(1,2) = \min\left(d(1,1), d(0,1)+4, d(3,1)+2\right) = \min(4,4,1) = 1$$

De même, pour calculer $d(2,3)$ on considère les arêtes qui arrivent à 2 qui sont les arêtes $(1,2)$, $(3,2)$ et $(4,2)$. On en déduit que

$$d(2,3) = \min\left(d(2,2), d(1,2)+3, d(3,2)+10, d(4,2)-3\right) = \min(7,7,9,-2) = -2$$

2) Lors de l'initialisation, pour tout sommet j, $d(j,0)$ contient bien le poids minimal d'un chemin de longueur 0 allant de 0 à j.
Soit $k \in \mathbb{N}$. On suppose qu'après être passé dans la boucle principale (lignes $5-12$) pour l'indice k, la case $d(i,k)$ contient le poids minimal d'un chemin de longueur au plus k allant de 0 à i. Pour tout sommet i, le poids minimal d'un chemin de longueur au plus $k+1$ allant de 0 à i est soit
— $d(i,k)$ si on ne peut pas trouver un chemin de poids inférieur en autorisant un sommet de plus
— $d(u,k)+w(u,i)$ s'il existe un chemin de plus faible poids dont la dernière arête est (u,i)
À la fin du passage pour $k+1$, les cases $d(i,k+1)$ contiennent les valeurs voulues (et ne seront plus modifiées).
De ce fait, à la fin de l'algorithme, la case $d(i,n-1)$ contient le poids des chemins de poids le plus faible allant de s à i et ayant moins de $n-1$ arêtes. Or, si un chemin de plus faible poids entre deux sommets avait plus de n arêtes, il passerait deux fois par le même sommet. En enlevant la boucle (qui ne peut pas être de poids négatif par hypothèse), on obtient un nouveau chemin de poids minimal. En réitérant l'opération on obtient un chemin ayant au plus $n-1$ arêtes.

3) La première boucle nécessite n opérations.
On passe n fois dans la boucle principale (lignes $5-12$). Dans cette boucle, la ligne 7 est faite pour tous les sommets donc n fois et la ligne 9 est faite pour toutes les arêtes donc p fois. On en déduit que la complexité totale est

$$O(n+n(n+p)) = O(n(n+p)$$

Dans le cas d'un graphe connexe, $n = O(p)$ on obtient donc $O(np)$.

4)

```
let add x y = match x, y with
    | Entier (a), Entier (b) -> Entier (a + b)
    | _ -> Infini;;
```

```
let mini x y = match x, y with
    | Infini, _ -> y
    | Entier (a), Infini -> Entier (a)
    | Entier (a), Entier (b) -> Entier (min a b);;
```

5) a) Le tableau des prédécesseurs est
`[|[];[(0,4);(3,2)];[(1,3);(3,10);(4,-3)];[(0,-1);(1,3)];[(3,2)]|]`

b) On commence par construire le tableau `prec` qui sera renvoyé à la fin. La fonction `parcours : (int*'a) list -> int -> ()` sert à parcourir une liste d'adjacence. En parcourant la liste d'adjacence du sommet i, tout élément (u, w) de cette liste permet de créer un couple (i, w) dans la liste des prédécesseurs du sommet u. On appelle cette fonction sur toutes les listes d'adjacences.

De cette sorte, toutes les arêtes sont considérées une fois et une seule. La complexité des donc linéaire en le nombre d'arêtes du graphe.

```
let succ_prec g =
  let n = Array.length g in
  let prec = Array.make n [] in
  let rec parcours l i = match l with
      | [] -> ()
      | (u, w) :: q -> prec.(u) <- (i, w) :: prec.(u); parcours q i
  in for i = 0 to (n - 1) do
    parcours g.(i) i
  done;
  prec;;
```

6) La fonction suit l'algorithme proposé :

```
let bf g =
  let n = Array.length g in
  let prec = succ_prec g in
  let d = Array.make_matrix n n Infini in
  d.(0).(0) <- Entier (0);
  let dprime = ref Infini in
  for k = 1 to (n - 1) do
    for i = 0 to (n - 1) do
      dprime := d.(i).(k - 1);
      let rec parcours l = match l with
          | [] -> ()
          | (u, w) :: q -> dprime := mini !dprime (add d.(u).(k - 1) (
             Entier(w))); parcours q
      in
      parcours prec.(i);
      d.(i).(k) <- !dprime
    done;
  done;
  let res = Array.make n Infini in
  for i = 0 to (n - 1) do
    res.(i) <- d.(i).(n - 1)
  done;
  res;;
```

Problème

Partie I - Coloriages

1) La matrice d'adjacence est :

$$\begin{pmatrix} \texttt{false} & \texttt{false} & \texttt{true} & \texttt{true} & \texttt{false} \\ \texttt{false} & \texttt{false} & \texttt{false} & \texttt{true} & \texttt{true} \\ \texttt{true} & \texttt{false} & \texttt{false} & \texttt{false} & \texttt{true} \\ \texttt{true} & \texttt{true} & \texttt{false} & \texttt{false} & \texttt{false} \\ \texttt{false} & \texttt{true} & \texttt{true} & \texttt{false} & \texttt{false} \end{pmatrix}$$

Le premier étiquetage n'est pas coloriage car les sommets 1 et 4 sont de couleur 1 alors qu'ils sont voisins.

Le deuxième étiquetage est un coloriage car 0 et 4 ne sont pas voisins et 1 et 2 ne sont pas voisins.

2) On peut colorier le graphe avec trois couleurs avec l'étiquetage suivant (par exemple) :
`[|0;1;0;1;2;1;2;2;0;0|]`

Par contre, on ne peut pas le colorier qu'avec deux couleurs, en effet, on peut considérer le cycle $(0, 5, 7, 2, 1)$. Si on ne veut utiliser que deux couleurs 0 et 1 et que l'on colorie le sommet 0 en couleur 0, le sommet 5 doit être de couleur 1, le sommet 7 de couleur 0, le sommet 2 de couleur 1 et le sommet 1 de couleur 0. Ce n'est pas correct car 0 et 1 qui sont voisins sont de la même couleur.

3) Les graphes qui sont coloriables qu'avec une seule couleur sont les graphes sans arêtes.

Si G est un graphe avec n sommets et que l'on dispose de n couleurs on peut affecter une couleur différente à chaque sommet. C'est un bon coloriage.

Le graphe complet a n sommets où tout couple de sommets (s, t) avec $s \neq t$ est relié par une arête a un nombre chromatique égal à n car chaque sommet doit avoir une couleur différente de tous les autres sommets.

4) Pour déterminer si un étiquetage est un coloriage, on parcours toute la matrice d'adjacence par une double boucle for (on peut se restreindre au cas ou $j > i$ car le graphe étant non orienté la matrice est symétrique) et, à chaque arête (i, j) dans le graphe, on teste si `etiq.(i)` est égal à `etiq.(j)`. Si c'est le cas, la balise booléenne passe à `false` car le graphe n'est pas coloriable.

```
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;;
```

5) Soit $k \in \mathbb{N}^*$. Un étiquetage avec au plus k couleurs et une application du nombre de sommets dans un ensemble ayant k éléments. Il y en a donc k^n.

Un graphe de n sommet possède un n-coloriage d'après la question 3). 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.

D'après ce qui précède, il existe une constante A telle que si on ne donne un étiquetage, on peut tester si c'est un coloriage en moins de An^2 opérations.
La complexité est alors majorée par

$$An^2 \left(\sum_{k=2}^{n-1} k^n \right) \leq An^2.n.n^n = Ae^{(n+3)\ln(n)} \leq Be^{n^2}$$

la complexité est exponentielle.

Partie II - 2-Coloriages

6) 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 couleur 1 (ou 2), et les arêtes vont donc d'un sommet de T vers un sommet de U.

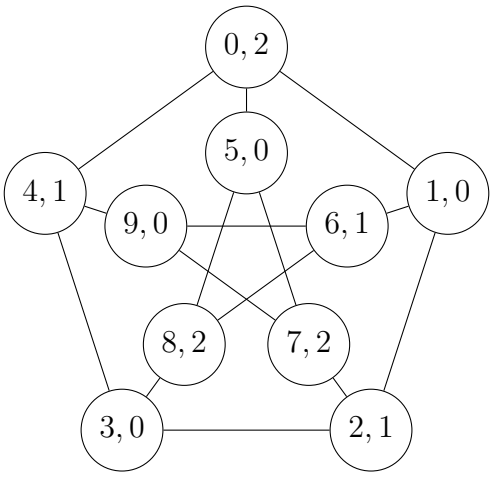
7) On porcède comme suggéré dans le texte. La fonction auxiliaire `parcours : int -> int -> unit` permet de parcourir les voisins d'un sommet `i` qui serait de couleur `coul` en parcourant la ligne correspondante dans la matrice d'adjacence. Le fait de ne relancer la fonction que si `etiq.(j) = -1` permet d'éviter de faire une boucle infinie.

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

Partie III - Algorithme glouton

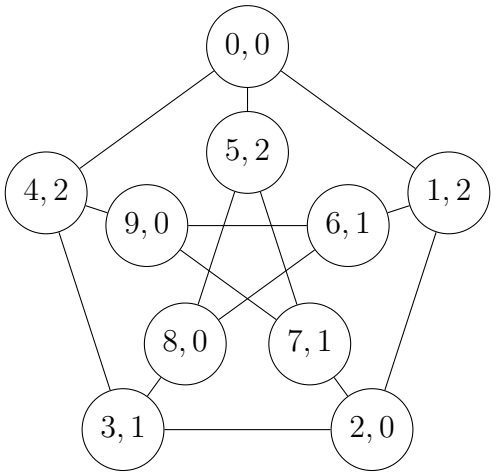
8) Avec le premier ordre on obtient un 3-coloriage (en précisant la couleur après la virgule)



C'est-à-dire :

sommet	0	1	2	3	4	5	6	7	8	9
couleur	2	0	1	0	1	0	1	2	2	0

Avec le deuxième ordre on obtient un 4-coloriage :



C'est-à-dire :

sommet	0	1	2	3	4	5	6	7	8	9
couleur	0	2	0	1	2	2	1	1	0	0

9) Comme suggéré dans l'enoncé, on commence en construisant un tableau `coul` tel que `coul.(i)` vaut `true` si et seulement si `i` est un voisin de `s` ayant déjà une couleur. Pour cela, on parcourt tous les sommets par une boucle `for` et on teste si `i` est un voisin de `s` (`gphe.(s).(i)`) et si le sommet `i` a déjà été coloré (`etiq.(i) <> -1`).
Il ne reste plus qu'à déterminer le plus petit entier c tel que `coul.(c)` est faux.

```
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;;
```

On parcourt tous les sommets du graphe pour construire la tableau `coul`. Pour déterminer la plus petite couleur disponible, on doit parcourir le tableau `coul` qui est de taille n. La complexité est bien en $O(n)$.

10) On applique l'algorithme glouton en considérant les sommets dans l'ordre imposé.

```
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;;
```

La fonction va appeler n fois la fonction de la question précédente qui avait une complexité $O(n)$. La complexité de `glouton` est donc $O(n^2)$.

11) 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, la couleur choisie est la plus petite parmi une ensemble d'au plus $d(G)$ entiers positif : elle est donc majorée par $d(G)$.

Ainsi la coloration construite admet au plus $d(G)+1$ couleurs.

12) 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 la fonction `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 $L'(s) \leq L(s)$ 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.

13) 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;;
```

La complexité de `degres` est en $O(n^2)$ car on parcourt la totalité de la matrice d'adjacence.

Il reste à trier les sommets en utilisant l'ordre proposé qui est donné par un tableau de degré. On écrit donc une fonction qui permet de trouver le minimum entre deux indices (en fonction des valeurs du tableau `deg` passé en paramètre).

```
let mini deg i j =
  if deg.(i) < deg.(j) then i else j;;
```

Ensuite on écrit une fonction qui permet de trouver l'indice du minimum des éléments entre `i` et la fin du tableau.

```
let rec minitab deg t i =
  let n = Array.length t in
  if i = n - 1 then n - 1
  else mini deg t.(i) (minitab deg t (i + 1));;
```

Il ne reste plus qu'a écrire la fonction de tri par insertion. Pour chaque `i` entre 0 et $n-1$ elle trouve l'indice du minimum dans la fin du tableau et le place en position i dans le tableau par un échange.

Le tri par insertion est de complexité quadratique. Ici ce n'est pas pénalisant car l'algorithme de Welsh-Powell est aussi de complexité quadratique.

```
let tri  deg t =
  let n = Array.length t in
  for k = 0 to (n - 2) do
    let i = minitab deg t k in
    let temp = t.(i) in
    t.(i) <- t.(k);
    t.(k) <- temp
  done;
  t;;
```

Il suffit alors de tout rassembler.

```
let welsh_powell gphe =
  let n = Array.length gphe in
  let id = Array.make n 0 in
  for i = 0 to (n - 1) do
    id.(i) <- i
  done;
  glouton gphe (tri (degres gphe) id);;
```