Aix*Marseille université

Portail René Descartes

Science informatique : Partiel 1h30

Année 2023–24

Documents et calculatrice interdits

Tous les exercices sont indépendants et peuvent donc être traités dans n'importe quel ordre. Rédigez vos réponses pour obtenir tous les points.

Exercice 1 (Terminaison, correction, complexité) On considère différents algorithmes pour le problème de la division euclidienne. Plus précisément, on considère le problème de prendre en entrée un nombre entier positif `a` et un nombre entier strictement positif `b` et de renvoyer le quotient `q=a//b` et le reste `r=a%b` dans la division euclidienne de `a` par `b`. On rappelle que `q` et `r` sont les uniques entiers tels que `a = b*q+r` et `0<=r<b`.

Par exemple, pour `a=7` et `b=3`, on doit renvoyer `q=2` et `r=1` et pour `a=8` et `b=2`, on doit renvoyer `q=4` et `r=0`.

Pour chacun des algorithmes ci-dessous :

— indiquez si l'algorithme termine ou non et expliquez brièvement pourquoi ;

— indiquez si l'algorithme est correct ou non et si ce n'est pas le cas, donnez un exemple d'entrée pour laquelle la sortie de l'algorithme est incorrecte (on ne demande pas de justification dans le cas où vous pensez que l'algorithme est correct) ;

— indiquez l'ordre de grandeur de la complexité algorithmique dans le pire des cas de l'algorithme en fonction de `a//b` (on pourrait par exemple avoir une complexité en $O((\texttt{a//b})^2)$ ou en $O((\texttt{a//b})\log_2(\texttt{a//b}))$. On ne demande pas de justification détaillée.

```
def diviser1(a, b):
    q = 0
    while q*b<=a:
        q = q+1
    r = a-q*b
    return q, r
```

```
def diviser2(a, b):
    r = a
    q = 0
    while r>=b:
        r = r-b
        q = q+1
    return q, r
```

```
def diviser3(a, b):
    r = a
    q = 0
    while r>=b:
        r = r+b
        q = q+1
    return q, r
```

```
def diviser4(a, b):
    r = a
    q = 0
    while r>=b:
        p = 1
        while r-2*p*b>=0:
            p = 2*p
        r = r-p*b
        q = q+p
    return q, r
```

Exercice 2 (Coloration de graphes) Soit G un graphe *non orienté* et *connexe* (en un seul morceau) de n sommets, représenté par sa matrice d'adjacence. On veut colorier chaque sommet de G avec une couleur (représentée par un entier naturel ≥ 0) de telle façon que deux sommets adjacents (reliés par une arête) soient toujours de deux couleurs différents. Les couleurs des sommets seront stockées dans un tableau `couleur` où la case `couleur[u]` représente la couleur du sommet u.

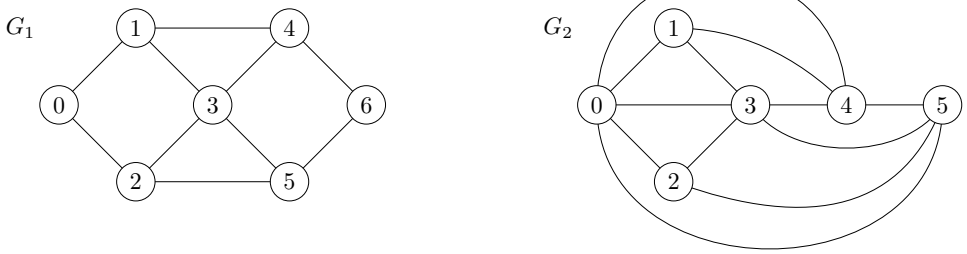
L'algorithme suivant vérifie s'il y a un conflit qui concerne le sommet u, c'est-à-dire si u est de la même couleur qu'un de ses voisins dans le graphe G :

```
def conflit(G, u, couleur):
    n = len(G)
    for v in range(n):
        if G[u][v] == 1 and couleur[u] == couleur[v]:
            return True
    return False
```

On colorie les sommets de G dans l'ordre correspondant à un parcours en largeur à partir du sommet 0. Initialement, on attribue à tous les sommets la couleur -1, pour indiquer qu'ils ne sont pas encore coloriés. Ensuite, dès qu'on découvre un sommet pas encore colorié, on lui attribue la plus petite couleur ≥ 0 qui ne crée pas de conflit avec ses voisins :

```
1  def colorier(G):
2      n = len(G)
3      F = nouvelle_file()
4      couleur = [-1] * n
5      enfiler(F, 0)
6      couleur[0] = 0
7      while not est_vide(F):
8          u = défiler(F)
9          for v in range(n):
10             if G[u][v] == 1 and couleur[v] == -1:
11                 enfiler(F, v)
12                 couleur[v] = 0
13                 while conflit(G, v, couleur):
14                     couleur[v] = couleur[v] + 1
15     return couleur
```

1. Appliquez l'algorithme `colorier` aux graphes G_1 et G_2 suivants, en montrant les couleurs des sommets au début (après avoir exécuté la ligne 6), ainsi qu'à à chaque tour de la boucle `while` des lignes 7–14 (y compris la coloration finale).



2. Est-ce que le graphe G_1 et/ou le graphe G_2 sont planaires ? Pourquoi ?
3. Est-ce que les colorations trouvées par l'algorithme à la question 1 sont optimales ? Pourquoi ? Fournir une coloration optimale pour le ou les graphes pour lequel ou lesquels on n'en a pas trouvé une optimale (justifiez votre réponse).

Exercice 3 (Tri) On veut développer un nouvel algorithme pour le tri de tableaux d'entiers basé sur le comptage du nombre d'occurrences de chaque valeur, plutôt que sur les comparaisons entre éléments du tableau.

Pour les trois questions suivantes, supposez toujours que A est *un tableau d'entiers naturels* (donc supérieurs ou égaux à 0). Remarquez qu'un tableau A peut être vide (c'est-à-dire, de longueur 0). Écrivez les algorithmes demandés sous la forme de fonctions Python.

1. Écrivez un algorithme `dupliquer(A)` qui renvoie un tableau B contenant $A[0]$ fois l'entier 0, suivi par $A[1]$ fois l'entier 1, ..., suivi par $A[n-1]$ fois l'entier $n-1$, où n est la longueur de A. Par exemple :
   — `dupliquer([0, 2, 0, 4, 1])` renvoie `[1, 1, 3, 3, 3, 3, 4]` ;
   — `dupliquer([0, 0, 5])` renvoie `[2, 2, 2, 2, 2]` ;
   — `dupliquer([1, 1, 1, 1])` renvoie `[0, 1, 2, 3]` ;
   — `dupliquer([])` renvoie `[]`.
2. Écrivez un algorithme `compter(A)` qui renvoie un tableau C tel que, pour tout indice i de 0 à M, où M est le maximum des éléments de A, la valeur de $C[i]$ est le nombre d'occurrences de l'entier i dans A. Si A est le tableau vide, alors le résultat devra être aussi vide. Par exemple :
   — `compter([3, 0, 1, 0, 0, 1, 0])` renvoie `[4, 2, 0, 1]` ;
   — `compter([4, 3, 2, 1, 0])` renvoie `[1, 1, 1, 1, 1]` ;
   — `compter([2, 2, 2, 2, 2])` renvoie `[0, 0, 5]` ;
   — `compter([0, 0, 0, 0])` renvoie `[4]` ;
   — `compter([])` renvoie `[]`.
3. Écrivez un algorithme `trier(A)` qui renvoie une version triée du tableau A. L'algorithme proposé doit se baser uniquement sur des appels aux algorithmes `compter` et `dupliquer`. (Vous pouvez répondre à cette question même si vous n'avez pas répondu à une ou deux des précédentes ; supposez tout simplement que des fonctions Python `compter` et `dupliquer` sont disponibles.)

Aix*Marseille université

Portail René Descartes

Science informatique : Partiel 1h30

Année 2023–24

Documents et calculatrice interdits

Tous les exercices sont indépendants et peuvent donc être traités dans n'importe quel ordre. Rédigez vos réponses pour obtenir tous les points.

Exercice 1 (Terminaison, correction, complexité) On considère différents algorithmes pour le problème de la division euclidienne. Plus précisément, on considère le problème de prendre en entrée un nombre entier positif `a` et un nombre entier strictement positif `b` et de renvoyer le quotient `q=a//b` et le reste `r=a%b` dans la division euclidienne de `a` par `b`. On rappelle que `q` et `r` sont les uniques entiers tels que `a = b*q+r` et `0<=r<b`.

Par exemple, pour `a=7` et `b=3`, on doit renvoyer `q=2` et `r=1` et pour `a=8` et `b=2`, on doit renvoyer `q=4` et `r=0`.

Pour chacun des algorithmes ci-dessous :

— indiquez si l'algorithme termine ou non et expliquez brièvement pourquoi ;

— indiquez si l'algorithme est correct ou non et si ce n'est pas le cas, donnez un exemple d'entrée pour laquelle la sortie de l'algorithme est incorrecte (on ne demande pas de justification dans le cas où vous pensez que l'algorithme est correct) ;

— indiquez l'ordre de grandeur de la complexité algorithmique dans le pire des cas de l'algorithme en fonction de `a//b` (on pourrait par exemple avoir une complexité en $O((\texttt{a//b})^2)$ ou en $O((\texttt{a//b})\log_2(\texttt{a//b}))$. On ne demande pas de justification détaillée.

```
def diviser1(a, b):
    q = 0
    while q*b<=a:
        q = q+1
    r = a-q*b
    return q, r
```

```
def diviser2(a, b):
    r = a
    q = 0
    while r>=b:
        r = r-b
        q = q+1
    return q, r
```

```
def diviser3(a, b):
    r = a
    q = 0
    while r>=b:
        r = r+b
        q = q+1
    return q, r
```

```
def diviser4(a, b):
    r = a
    q = 0
    while r>=b:
        p = 1
        while r-2*p*b>=0:
            p = 2*p
        r = r-p*b
        q = q+p
    return q, r
```

Solution : L'algorithme `diviser1` termine. En effet, le seul élément susceptible de prévenir la terminaison est la présence d'une boucle `while`. La condition de cette boucle est équivalente à `a-q*b>=0`. L'algorithme terminera donc toujours si on peut montrer que le variant de boucle `v=a-q*b` devient négatif après un nombre fini d'itérations de la boucle. C'est bien le cas, car `v` vaut `a` lors de la première itération et à chaque itération subséquente il est décrémenté d'au moins 1 (en effet, `q*b` est incrémenté d'un entier strictement positif à chaque itération, puisque `q` est incrémenté de 1 et que `b`, par hypothèse, est strictement positif).
L'algorithme est incorrect, car il incrémente `q` une fois de trop. Par exemple `diviser1(0,1)`) renvoie `1,-1` au lieu de renvoyer `0, 0`.
La complexité dans le pire des cas de l'algorithme est en $O(\texttt{a//b})$, qu'on obtient en remarquant, d'une part, que `q` vaut initialement 0 et est incrémenté de 1 à chaque tour de boucle et, d'autre part, que la première valeur de `q` invalidant la condition de boucle, c'est à dire telle que `q*b > a`, est par définition `a//b`+1. Il y a donc `a//b`+1 tours de boucle effectués.

Solution : L'algorithme `diviser2` termine. Comme précédemment, il suffit de justifier que la boucle `while` termine. On peut prendre `r` comme variant de boucle. La condition de boucle est invalidée quand `r` devient inférieur à l'entier `b`. Or `r` vaut initalement `a` et est décrémenté d'un entier strictement positif `b` à chaque tour de boucle, donc `r` finira bien par devenir inférieur à `b` après un nombre fini d'itérations. L'algorithme est correct. (On pourrait le démontrer en utilisant l'invariant de boucle `a=q*b+r` et `q` et `r` sont des nombres entiers.)
La complexité dans le pire des cas de l'algorithme est aussi en $O(\texttt{a//b})$. On le voit en remarquant qu'après `k` itérations de la boucle, `r` vaut `a-k*b` et en en déduisant que la condition de boucle `r>=b` sera invalidée après `a//b` itérations, par définition de `a//b`.

Solution : L'algorithme `diviser3` ne termine pas. En effet, dès que `a>=b`, la condition de la boucle `while` est valide la première fois qu'elle est évaluée (puisque `r` vaut `a` initialement) et dès lors `r` est incrémenté d'un nombre (strictement) positif `b` à chaque itération de la boucle, ce qui garantit que la condition de boucle reste valide indéfiniment et que l'algorithme ne termine pas.
Comme l'algorithme ne termine pas, il est incorrect et sa complexité est infinie dans le pire des cas.

Solution : L'algorithme `diviser4` termine. Pour le justifier, il faut montrer, d'une part, qu'à chaque tour de la boucle `while` externe, la boucle `while` interne termine et, d'autre part, que la boucle `while` externe termine. La boucle interne termine toujours car le variant de boucle `r-2*p*b` décroit strictement à chaque itération (`b` étant strictement positif par hypothèse et `p` étant un entier strictement positif qui est doublé à chaque itération) et que la boucle s'arrête quand il devient strictement négatif. La boucle externe termine parce que le variant de boucle `r` décroit strictement à chaque itération (`b` étant strictement positif par hypothèse et `p` étant positif tout au long et donc aussi à la fin de l'exécution de la boucle `while` interne) et que la boucle s'arrête quand il devient strictement inférieur à `b`.
L'algorithme est correct. (Pour le démontrer, on pourrait utiliser, pour la boucle extérieure, l'invariant de boucle `r = a-b*`$\left(\sum_{i=0}^{k-1} c_{\lfloor \log_2(a//b)\rfloor - i} 2^{\lfloor \log_2(a//b)\rfloor - i}\right)$ où k est le nombre d'itérations de la boucle extérieure effectuées et c_i est le i-ième chiffre dans l'écriture de `a//b` en base 2.)
La complexité de l'algorithme dans le pire des cas est en $O(\log_2(\texttt{a//b})^2)$. En effet, si nous notons $x := \log_2(\texttt{a//b})$, après k tours de la boucle externe, le nombre de tours de la boucle interne est d'au plus $\lfloor x \rfloor - k$. De plus, il y a au plus $\lfloor x \rfloor$ tours de la boucle externe. On a donc un total d'opérations en $O\left(\sum_{k=0}^{\lfloor x \rfloor - 1} \lfloor x \rfloor - k\right) = O(\lfloor x \rfloor^2 - \lfloor x \rfloor(\lfloor x \rfloor - 1)/2) = O(\lfloor x \rfloor^2) = O(x^2)$. Cette complexité est effectivement atteinte, par exemple pour n'importe quel choix de `b` et pour `a=b*`$\sum_{i=0}^{n} 2^i = b(2^{n+1} - 1)$.

Exercice 2 (Coloration de graphes) Soit G un graphe *non orienté* et *connexe* (en un seul morceau) de n sommets, représenté par sa matrice d'adjacence. On veut colorier chaque sommet de G avec une couleur (représentée par un entier naturel ≥ 0) de telle façon que deux sommets adjacents (reliés par une arête) soient toujours de deux couleurs différents. Les couleurs des sommets seront stockées dans un tableau `couleur` où la case `couleur[u]` représente la couleur du sommet u.

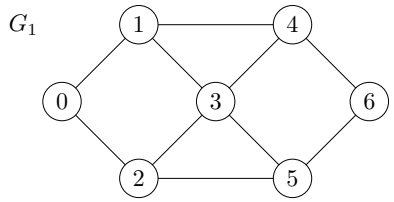
L'algorithme suivant vérifie s'il y a un conflit qui concerne le sommet u, c'est-à-dire si u est de la même couleur qu'un de ses voisins dans le graphe G :

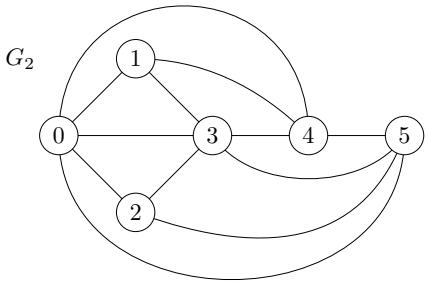
```
def conflit(G, u, couleur):
    n = len(G)
    for v in range(n):
        if G[u][v] == 1 and couleur[u] == couleur[v]:
            return True
    return False
```

On colorie les sommets de G dans l'ordre correspondant à un parcours en largeur à partir du sommet 0. Initialement, on attribue à tous les sommets la couleur -1, pour indiquer qu'ils ne sont pas encore coloriés. Ensuite, dès qu'on découvre un sommet pas encore colorié, on lui attribue la plus petite couleur ≥ 0 qui ne crée pas de conflit avec ses voisins :

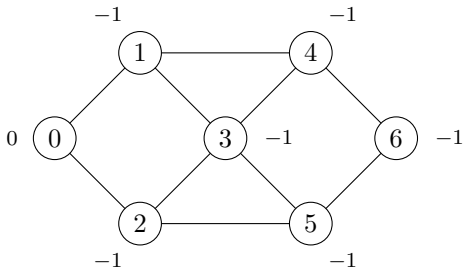
```
1  def colorier(G):
2      n = len(G)
3      F = nouvelle_file()
4      couleur = [-1] * n
5      enfiler(F, 0)
6      couleur[0] = 0
7      while not est_vide(F):
8          u = défiler(F)
9          for v in range(n):
10             if G[u][v] == 1 and couleur[v] == -1:
11                 enfiler(F, v)
12                 couleur[v] = 0
13                 while conflit(G, v, couleur):
14                     couleur[v] = couleur[v] + 1
15     return couleur
```

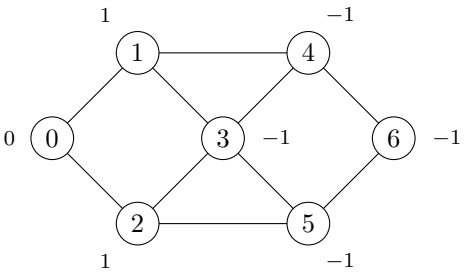
1. Appliquez l'algorithme `colorier` aux graphes G_1 et G_2 suivants, en montrant les couleurs des sommets au début (après avoir exécuté la ligne 6), ainsi qu'à à chaque tour de la boucle `while` des lignes 7–14 (y compris la coloration finale).

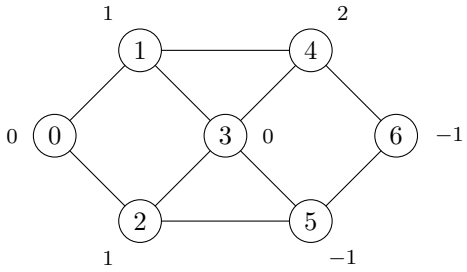


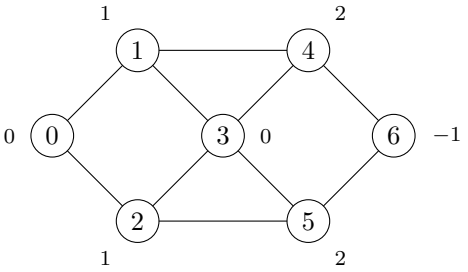


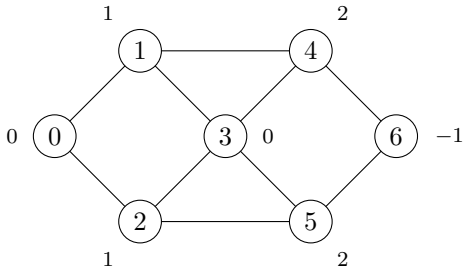
Solution : Pour le graphe G_1 on obtient une coloration à 3 couleurs :

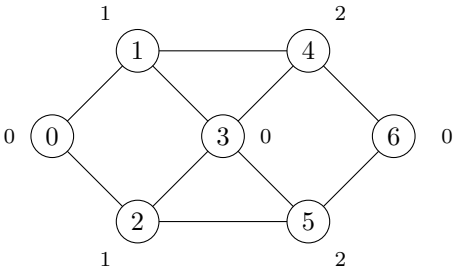


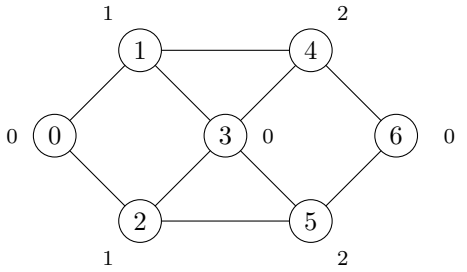


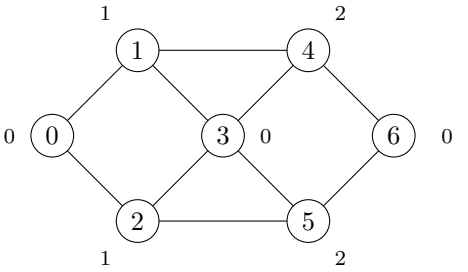




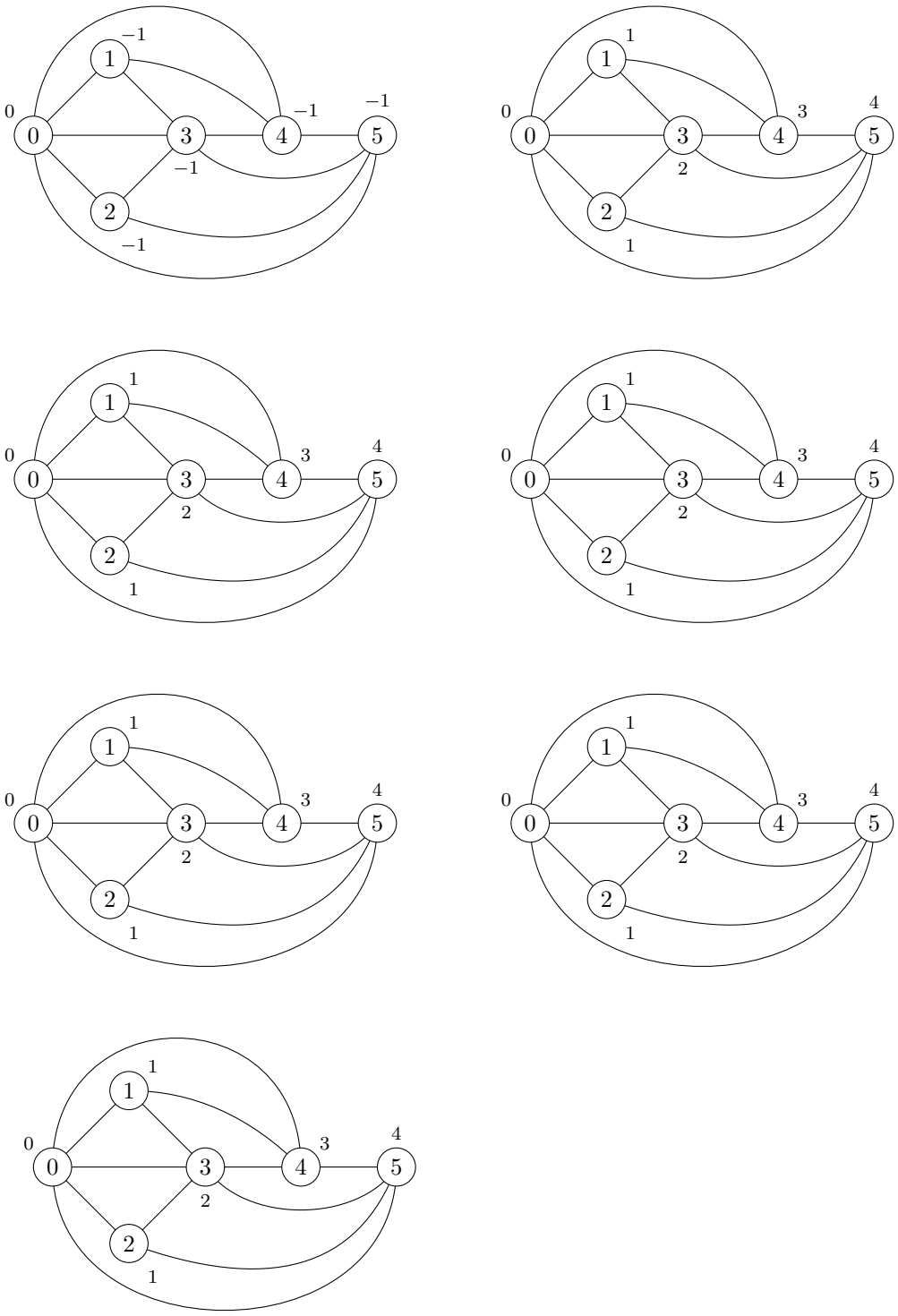








Solution : Pour le graphe G_2 on obtient une coloration à 5 couleurs :

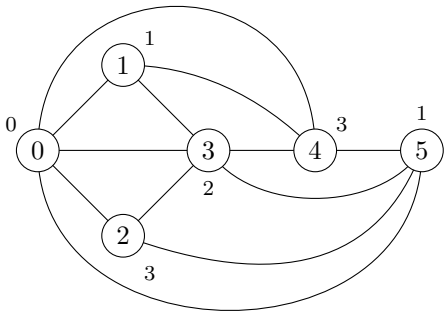


2. Est-ce que le graphe G_1 et/ou le graphe G_2 sont planaires ? Pourquoi ?

Solution : Comme on peut l'observer des illustrations, qui n'ont pas d'arête qui se croisent, les deux graphes sont planaires.

3. Est-ce que les colorations trouvées par l'algorithme à la question 1 sont optimales ? Pourquoi ? Fournir une coloration optimale pour le ou les graphes pour lequel ou lesquels on n'en a pas trouvé une optimale (justifiez votre réponse).

Solution : La coloration trouvée pour G_1 est optimale, puisque G_1 contient un triangle, c'est à dire un sous-graphe complet de taille 3 (par exemple celui formé par les sommets 1, 3 et 4) et que 3 couleurs sont nécessaires pour colorier un tel graphe.
En revanche, en étant planaire, G_2 admet une coloration à 4 couleurs grâce au théorème correspondant. Donc celle trouvé par l'algorithme n'est pas optimale. En voici une meilleure, à 4 couleurs :



Cette coloration est optimale puisque G_2 contient un sous-graphe complet de taille 4 (par exemple celui formé par les sommets 0, 1, 3 et 4) et que 4 couleurs sont nécessaire pour colorier un tel graphe.

Exercice 3 (Tri) On veut développer un nouvel algorithme pour le tri de tableaux d'entiers basé sur le comptage du nombre d'occurrences de chaque valeur, plutôt que sur les comparaisons entre éléments du tableau.

Pour les trois questions suivantes, supposez toujours que A est *un tableau d'entiers naturels* (donc supérieurs ou égaux à 0). Remarquez qu'un tableau A peut être vide (c'est-à-dire, de longueur 0). Écrivez les algorithmes demandés sous la forme de fonctions Python.

1. Écrivez un algorithme `dupliquer(A)` qui renvoie un tableau B contenant $A[0]$ fois l'entier 0, suivi par $A[1]$ fois l'entier 1, ..., suivi par $A[n-1]$ fois l'entier $n-1$, où n est la longueur de A. Par exemple :
   — `dupliquer([0, 2, 0, 4, 1])` renvoie `[1, 1, 3, 3, 3, 3, 4]` ;
   — `dupliquer([0, 0, 5])` renvoie `[2, 2, 2, 2, 2]` ;
   — `dupliquer([1, 1, 1, 1])` renvoie `[0, 1, 2, 3]` ;
   — `dupliquer([])` renvoie `[]`.

Solution : On ajoute au tableau B, initialement vide, la valeur de i dupliquée $A[i]$ fois, pour tout $0 \leq i < n$.

```
def dupliquer(A):
    n = len(A)
    B = []
    for i in range(n):
        for j in range(A[i]):
            B.append(i)
    return B
```

2. Écrivez un algorithme `compter(A)` qui renvoie un tableau C tel que, pour tout indice i de 0 à M, où M est le maximum des éléments de A, la valeur de $C[i]$ est le nombre d'occurrences de l'entier i dans A. Si A est le tableau vide, alors le résultat devra être aussi vide. Par exemple :
   — `compter([3, 0, 1, 0, 0, 1, 0])` renvoie `[4, 2, 0, 1]` ;
   — `compter([4, 3, 2, 1, 0])` renvoie `[1, 1, 1, 1, 1]` ;
   — `compter([2, 2, 2, 2, 2])` renvoie `[0, 0, 5]` ;
   — `compter([0, 0, 0, 0])` renvoie `[4]` ;
   — `compter([])` renvoie `[]`.

Solution : On calcule d'abord le maximum M du tableau A, en utilisant $M = -1$ comme valeur initiale. Puis on crée un tableau C de longueur $M + 1$ (ce qui est 0 si A est vide) contenant initialement des 0. Enfin, pour chaque élément $A[i]$ de A on incrémente la case correspondante $C[A[i]]$.

```
def compter(A):
    n = len(A)
    M = -1
    for i in range(n):
        if A[i] > M:
            M = A[i]
    C = [0] * (M + 1)
    for i in range(n):
        C[A[i]] = C[A[i]] + 1
    return C
```

On peut également utiliser la fonction `max` de Python pour calculer le maximum de A s'il n'est pas vide, et gérer séparément le cas ou il l'est :

```
def compter(A):
    n = len(A)
    if n == 0:
        C = []
    else:
        M = max(A)
        C = [0] * (M + 1)
        for i in range(n):
            C[A[i]] = C[A[i]] + 1
    return C
```

3. Écrivez un algorithme `trier(A)` qui renvoie une version triée du tableau A. L'algorithme proposé doit se baser uniquement sur des appels aux algorithmes `compter` et `dupliquer`. (Vous pouvez répondre à cette question même si vous n'avez pas répondu à une ou deux des précédentes ; supposez tout simplement que des fonctions Python `compter` et `dupliquer` sont disponibles.)

Solution : Il suffit d'appliquer l'algorithme `compter` à A et d'appliquer `dupliquer` au résultat :

```
def trier(A):
    return dupliquer(compter(A))
```