Devoir Surveillé d'ITC N°2
le 23/05/2025
MPSI & PCSI – Durée : 2 heures

Consignes

- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La numérotation des exercices (et des questions) doit être respectée et mise en évidence. Les résultats (hors questions purement informatiques) doivent être encadrés proprement.
- Il est important de numéroter correctement les pages des copies qui seront données à la correction. Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page.
- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il convient de le signaler sur la copie et de poursuivre la composition en expliquant les raisons des initiatives qui ont été prises.
- Les candidats ne doivent avoir aucune communication entre eux ou avec l'extérieur durant l'épreuve. Aussi, l'utilisation des téléphones portables et, plus largement, de tout appareil permettant des échanges ou la consultation d'informations, est interdite.
- À l'issue de la durée prévue pour cette épreuve, les candidats doivent déposer le stylo et ne sont plus autorisés à écrire quoi que ce soit sur leur copie. Tout retard donne lieu à une pénalité sur la note finale.
- **L'usage de la calculatrice est interdit.**

- Les signatures des fonctions devront toutes être écrites et reprises sur votre copie. Une docstring devra être indiquée lorsque l'énoncé le demande.
- Vous pouvez (et même devez) utiliser les fonctions des questions précédentes, même si vous ne les avez pas toutes implémentées. Le barème en tiendra compte.

Problème **Des algorithmes pour colorer un graphe** La coloration d'une carte de pays consiste à attribuer une couleur à chacun des pays de manière à ce que deux pays voisins soient de couleurs différentes.

Étudier ces techniques de coloration revient de façon plus abstraite à travailler sur des graphes. Le champ d'applications de la coloration de graphes est très vaste et couvre des domaines aussi variés que le problème de l'attribution de fréquences dans les télécommunications, la conception de puces électroniques ou l'allocation de registres en compilation. Soulignons que tous les graphes considérés dans ce sujet sont *non-orientés*.

PARTIE I — REPRÉSENTATION EN MACHINE D'UN GRAPHE

On cherche à colorer une carte de pays avec comme seule contrainte que deux pays ayant une frontière commune ne peuvent être de la même couleur. Comme les pays, les couleurs seront numérotées à partir de zéro. À titre d'exemple, on considèrera la carte suivante (Figure 1), comportant 8 pays numérotés de 0 à 7 :

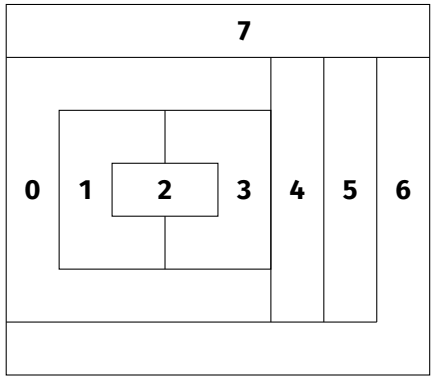

FIGURE 1. – Exemple d'une carte de pays

Cette carte peut être représentée par un graphe G_{ex} donné en Figure 2 :

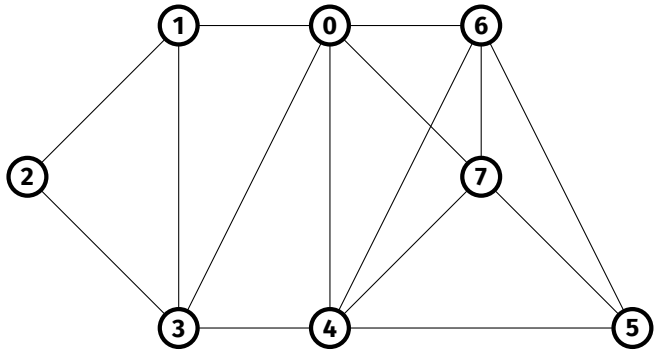

FIGURE 2. – Graphe G_{ex} associé à la carte

Ainsi, sur le graphe de la Figure 2, deux pays sont voisins si et seulement si les sommets correspondants sont reliés par une arête.

1. Un graphe peut être représenté par une *matrice d'adjacence*. Rappeler brièvement ce que contient la matrice d'adjacence d'un graphe dont les sommets sont des entiers positifs numérotés à partir de zéro. Préciser sur votre copie la taille de la matrice d'adjacence M_{ex} du graphe G_{ex}, et écrire le contenu des 4 premières lignes.
2. Un graphe peut être représenté par un *dictionnaire des voisins* (appelé aussi *dictionnaire d'adjacence*). Donner le dictionnaire des voisins `Gex` du graphe G_{ex}.
3. Recopier et compléter le tableau ci-dessous, donnant le degré de chaque sommet dans G_{ex}.

Sommet	0	1	2	3	4	5	6	7
Degré								

4. Écrire une fonction `degre(G:dict)->dict` qui a pour argument le dictionnaire des voisins d'un graphe quelconque, qui renvoie un dictionnaire de clefs les sommets du graphe, et valeurs le degré d'un sommet. Par exemple, pour le graphe représenté par :
```
G = {0:[1,2], 1:[0,2,3], 2:[0,1,3], 3:[1,2,4], 4:[3]}
```
la fonction renverra `{0:2, 1:3, 2:3, 3:3, 4:1}`.
5. Écrire une fonction `est_voisins(s, v, G:dict)->bool` avec trois arguments, deux sommets `s` et `v` et un dictionnaire `G` représentant un graphe, qui renvoie `True` si les sommets `s` et `v` sont reliés par une arête, et `False` sinon.

Dans toute la suite, on utilisera la représentation des graphes à l'aide d'un dictionnaire.

Partie II — Tester si une coloration est valide

On considère une carte de pays, représentée par un graphe `G` pour laquelle on a une proposition de coloration donnée par un dictionnaire `C` de clefs les sommets, et valeur un numéro de couleur associée. Ainsi, `C[s]` donne le numéro de la couleur attribuée au sommet `s`. On souhaite déterminer si la coloration proposée est valide (deux sommets du graphe reliés par une arête ne peuvent pas être de la même couleur).

6. On propose les deux listes ci-dessous (relativement au graphe G_{ex}). Préciser le caractère valide ou non des deux dictionnaires de couleurs `C1`, `C2` définies ci-dessous :
```
C1 = {0:0, 1:1, 2:0, 3:2, 4:1, 5:0, 6:2, 7:3}
C2 = {0:0, 1:3, 2:0, 3:1, 4:3, 5:2, 6:1, 7:0}
```
7. Écrire la fonction `coloration_valide(G:dict, C:dict)->bool` avec pour arguments un dictionnaire `G` codant un graphe et un dictionnaire des couleurs `C`, qui renvoie `True` si la coloration est valide, `False` sinon.
8. Pour un graphe comportant n sommets et m arêtes, quelle est la complexité temporelle asymptotique dans le pire des cas de la fonction `coloration_valide` ? *On comptera les tests et affectations.*

Partie III — Fonction auxiliaire

Avant de rentrer dans le coloriage à proprement parler, on commence par une fonction préliminaire. On souhaite écrire dans cette partie une fonction `premierNonPresent(L:list)->int` renvoyant, pour une liste `L` formée d'entiers naturels, le plus petit entier positif non présent dans `L`.

9. **[Méthode 1]** On décide dans cette première méthode (naïve) d'utiliser la fonction ci-dessous :
```
def premierNonPresent(L:list)->int:
    """
    renvoie le plus petit entier naturel non présent
    dans la liste d'entiers naturels L
    """
    i = 0
    while i in L:
        i += 1
    return i
```
On note i_k le contenu de la variable `i` à la fin de l'itération k.

9.1) Comment tester avec une ou plusieurs commandes `assert` le fait que `L` est composée d'entiers naturels ? Vous indiquerez sur votre copie la ou les lignes en question (inutile de recopier le reste).

9.2) Prouver la terminaison de la fonction `premierNonPresent`. On notera alors dans la suite N le nombre d'itérations, de sorte que la suite $(i_k)_{k\in\{0,\dots,N\}}$ est une suite finie.

9.3) Dans cette question, pour tout couple $(a,b)\in\mathbb{N}^2$, $[\![a,b]\!]$ désigne $[a,b]\cap\mathbb{N}$ avec pour convention $[\![a\,,\,b]\!]=\emptyset$ lorsque $a>b$. Justifier que :

$$\mathscr{P}(k)\quad\text{«}[\![0\,,\,k-1]\!]\subset \mathrm{L}\ \underline{\textbf{et}}\ i_k=k\text{»}\quad\text{pour } k\in\{0,\dots,\mathrm{N}\}$$

est un invariant de boucle `while`. En déduire la correction de la fonction.

9.4) Justifier rapidement que la complexité dans le pire des cas de cette fonction est en $O(n^2)$ où n est la taille de la liste. *On comptera les tests et affectations, et on supposera que l'instruction* `in` *est en* $O(n)$. *Vous compterez également les* `assert`*, s'ils ont été complétés.*

10. **[Méthode 2]** On décide d'utiliser désormais le principe suivant :

- on commence par définir une liste VU (comme *valeurs utilisées*) initialisée d'abord avec $n+1$ fois la valeur 0 (où n désigne la taille de L),
- puis parcourir une à une chaque valeur e de L, et si $e \leqslant n$, placer VU[e] à 1.
- Une fois ce parcours achevé, il reste à parcourir la liste VU à la recherche de la première position contenant encore 0 (on est sûr qu'il y en a au moins une puisque la liste VU ayant plus d'éléments que la liste L, toutes les positions de VU n'ont pas pu être mises à 1). Cette première position contenant 0 est bien le premier entier naturel non présent dans L.

10.1) Donner la liste VU1 correspondant à la liste `L1 = [2, 0, 5, 2]`, puis VU2 correspondant à la liste `L2 = [2, 0, 1, 3]`.

10.2) Donner une nouvelle version de la fonction `premierNonPresent2(L : list)->int` en utilisant cet algorithme. *On se passera de vérification avec* `assert` *dans cette question.*

10.3) Justifier que la complexité dans le pire des cas de `premierNonPresent2` est en $O(n)$. *On comptera les tests et affectations, et l'instruction* `[0]*(n+1)` *pour n opérations.*

PARTIE IV — UN ALGORITHME INTUITIF DE COLORATION

Au cours de l'algorithme présenté dans cette partie, l'attribution des couleurs à chaque sommet est toujours caractérisée par un dictionnaire C où `C[s]` est la couleur attribuée au sommet `s` ; si `s` n'est pas une clef de C, cela signifiera que le sommet `s` n'a pas encore été colorié. Le dictionnaire C est vide au départ et des éléments `sommet:couleur` sont ajoutés progressivement au fur et à mesure que les couleurs sont attribuées.

11. Recopier et compléter le code de la fonction `colore_sommet(C:dict, s:int, G:dict)->None` ayant trois arguments, le dictionnaire C des couleurs attribuées, le numéro `s` du sommet à colorer et le dictionnaire des voisins G caractérisant le graphe. *Cette fonction agira par effets de bord ; elle ne renvoie rien mais modifie le dictionnaire C en donnant à `C[s]` la plus petite couleur possible, en fonction des couleurs des sommets voisins qui sont déjà colorés.*
Par exemple, pour le graphe G_{ex}, prenons : `C = {0:0, 1:1}`. Les sommets 0 et 1 ont donc déjà été colorés avec les couleurs C[0] = 0 et C[1] = 1. L'appel `colore_sommet(C, 2, G_ex)` modifie le dictionnaire C en : `C = {0:0, 1:1, 2:0}`. Cela veut dire que le sommet 2, adjacent au sommet 1 mais pas au sommet 0, a reçu la même couleur que le sommet 0.

```
def colore_sommet(C:dict, s:int, G:dict)->None:
    # Construction de la liste U des couleurs déjà utilisées \
    ↪ par les voisins de s
    U = []
    for v in _______________ :  # pour chaque voisin de s
        if _______________ :  # s'il a déjà une couleur
            _____________________  # on ajoute cette couleur \
            ↪ à la liste U
    C[s] = premierNonPresent(U) # on colorie le sommet s
```

12. À l'aide de la question 11, écrire une fonction `colorer1(G:dict)->dict` avec pour argument un dictionnaire des voisins G caractérisant le graphe, qui crée et renvoie le dictionnaire C des numéros des couleurs attribuées en colorant les sommets dans un ordre arbitraire. Par exemple, l'application de la fonction `colorer1` au graphe G_{ex} renverra le dictionnaire de coloriage :

`{0:0, 1:1, 2:0, 3:2, 4:1, 5:0, 6:2, 7:3}`

13. L'ordre de coloration imposé à la question précédente est un exemple arbitraire d'ordre. On souhaite maintenant colorer le graphe en traitant les sommets selon un autre ordre arbitraire donné en argument. Écrire une fonction `colorer2(S:list, G:dict)->dict` analogue à `colorer1` et avec un argument supplémentaire S, une liste fixant l'ordre de coloration des sommets. Par exemple `colorer2([0, 2, 4, 6, 1, 3, 5, 7], G)` colorera le graphe G_{ex} en commençant d'abord par le sommet 0, puis en continuant par les sommets 2, 4, 6… *etc.*

14. Donner le dictionnaire de coloriage renvoyé par `colorer2` pour colorer le graphe G_{ex} donné en exemple en prenant l'ordre `[7, 6, 5, 4, 3, 2, 1, 0]`. Combien de couleurs ont-elles été utilisées ?

La méthode que nous venons de décrire est rapide, gloutonne, et fonctionne plutôt bien. Cependant, si on cherche à utiliser le nombre minimum de couleurs, l'efficacité de l'algorithme proposé ci-dessus dépend en grande partie de l'ordre dans lequel on choisit de colorer les sommets du graphe.

L'objectif des deux parties suivantes est d'affiner la stratégie pour mieux choisir cet ordre de coloration.

PARTIE V — VARIANTE DE WELSH-POWELL

Une alternative est donnée par la variante de *Welsh-Powell*. L'idée est de parcourir l'ensemble des sommets du graphe par ordre décroissant de leurs degrés. Comme le degré d'un sommet est un entier positif, il est possible d'écrire un algorithme de tri efficace (dit par *répartition* ou par *comptage*).

15. [Préliminaires sur les listes]

15.1) Écrire une fonction `renverse(L:list)->list` avec pour argument une liste L, qui crée et renvoie une nouvelle liste obtenue en lisant L dans l'ordre inverse.

Par exemple, `renverse([1, 2, 3, 4])` renverra `[4, 3, 2, 1]`. *Dans cette question, on s'interdira l'utilisation de `reversed` ou toute autre fonction renvoyant une telle liste.*

15.2) Écrire une fonction `init(n:int)->list` avec pour argument un entier n, qui renvoie une liste de listes R de taille n, telle que $R[i]$ (pour $0 \leqslant i \leqslant n-1$) soit une liste vide. Par exemple, `init(3)` renverra `[[], [], []]`.

15.3) Écrire une fonction `ranger(G:dict)->[list]` qui a pour argument le dictionnaire des voisins d'un graphe quelconque, et qui renvoie une liste `R` de longueur le nombre de sommets, et telle que `R[i]` (pour $0 \leqslant i \leqslant n-1$) soit la liste des sommets de degré `i` de `G`. Ainsi, pour le graphe associé à :

```
G = {0:[1,2], 1:[0,2,3], 2:[0,1,3], 3:[1,2,4], 4:[3]}
```

l'appel `ranger(G)` renverra la liste `[[], [4], [0], [1,2,3]]`

16. Écrire une fonction `trier_sommets(G:dict)->list` avec pour argument un dictionnaire des voisins `G`, et qui renvoie la liste des sommets triés dans l'ordre **décroissant** de leur degré. Par exemple, pour le graphe associé à :

```
G = {0:[1,2], 1:[0,2,3], 2:[0,1,3], 3:[1,2,4], 4:[3]}
```

la fonction `trier_sommets` renverra la liste de sommets `[1,2,3,0,4]`. *Plusieurs listes peuvent convenir comme résultat, vous vous assurerez que votre fonction renvoie l'une d'entre elles.*

17. Écrire la fonction `colorer3(G:dict)->dict`, utilisant `colorer2` appliquée à une liste de sommets `S` bien choisie, avec pour argument un dictionnaire des voisins `G` et renvoyant un dictionnaire de coloriage `C`, telle que `C[s]` soit la couleur à attribuer au sommet numéro `s`, les sommets étant colorés dans l'ordre **décroissant** de leur degré.

18. Pour le graphe G_{ex} de la Figure 2, donner le dictionnaire de coloriage `C` des couleurs renvoyé par la fonction `colorer3`.

L'amélioration proposée donne de bons résultats dans un grand nombre de cas. Cependant, il reste quelques cas où la coloration obtenue utilise trop de couleurs.

La méthode suivante, proposée en 1979 par Daniel BRÉLAZ de l'École Polytechnique Fédérale de Lausanne, raffine la détermination de l'ordre de coloration. La priorité de coloration est ainsi recalculée après chaque traitement d'un sommet et non plus une fois pour toute au départ. Au final, cette approche fournit rapidement une coloration optimale dans un très grand nombre de cas.

PARTIE VI — ALGORITHME DSATUR

Lorsque l'on colore un graphe, le *degré de saturation* d'un sommet est le nombre de couleurs différentes déjà attribuées à ses différents sommets voisins. Évidemment, ce degré de saturation est susceptible d'évoluer à chaque fois qu'un nouveau sommet est coloré.

19. [Préliminaire sur les listes] Écrire une fonction `sans_doublon(L:list)->bool` qui étant donnée une liste `L` renvoie une liste contenant tous les éléments de `L` apparaissant une unique fois. Par exemple, `sans_doublon([2, 3, 2, 2, 1, 3, 0]` renverra `[2, 3, 1, 0]`. *Bien sûr, plusieurs listes conviennent (égales à permutation près) ; on s'assurera que la fonction renvoie bien l'une d'entre elles. On pourra utiliser la commande `in` en cas de besoin.*

20. Écrire le code de la fonction `degre_satur(C:dict, s:int, G:dict)->int` ayant trois arguments, le dictionnaire `C` des couleurs attribuées, le numéro `s` du sommet à colorer et le dictionnaire des voisins `G` caractérisant le graphe, et renvoyant le degré de saturation du sommet `s`. On rappelle qu'un sommet `v` est coloré si et seulement si `v` est une clef de `C`.

21. Écrire une fonction `liste_satur(C:dict, G:dict)->list` avec deux arguments, un dictionnaire des voisins `G`, le dictionnaire `C` des couleurs des sommets, renvoyant la liste des sommets non colorés du graphe ayant un degré de saturation maximal parmi les sommets non colorés. On notera qu'il s'agit d'une liste car plusieurs sommets peuvent avoir le même degré de saturation. On supposera de plus qu'il reste au moins un sommet non coloré lorsque l'on fait appel à cette fonction.

22. Écrire le code de la fonction `colorer4(G:dict)->dict` ayant pour argument un dictionnaire des voisins `G`, qui renvoie un dictionnaire de coloriage `C` constituant une coloration du graphe. Cette fonction procède de la façon suivante, tant qu'il reste un sommet non coloré :

- déterminer parmi les sommets non colorés ceux de degré de saturation maximale ;
- si plusieurs sommets non colorés ont un degré de saturation maximale, en choisir un arbitrairement parmi ceux-ci qui soit de degré maximal ;
- colorer le sommet choisi en lui attribuant la couleur disponible ayant la plus petite valeur.

On notera qu'il n'est pas possible dans cette méthode de déterminer *a priori* l'ordre de parcours du graphe puisque le choix du futur sommet dépend de l'état du dictionnaire de coloriage `C`.

Correction
du Devoir Surveillé d'ITC N°2
MPSI & PCSI

Solution

PARTIE I — REPRÉSENTATION EN MACHINE D'UN GRAPHE

1. La matrice M associée à un graphe G est la matrice telle que $M_{i,j}$ soit 1 (ou `True`) s'il existe un arc $i \to j$, et 0 (ou `False`) sinon. Elle est de format $n \times n$ où n est le nombre de sommets du graphe. Dans le cas de M_{ex}, la taille est 8×8 puisqu'il y'a 8 sommets. Les 4 premières lignes sont : $\begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$.

2.
```
G_ex = {
    0:[1, 3, 4, 6, 7], \
    1:[0, 2, 3], \
    2:[1, 3], \
    3:[0, 1, 2, 4], \
    4:[0, 3, 5, 6, 7], \
    5:[4, 6, 7], \
    6:[0, 4, 5, 7], \
    7:[0, 4, 5, 6]}
```

3. Recopier et compléter le tableau ci-dessous, donnant le degré de chaque sommet dans G_{ex}.

Sommet	0	1	2	3	4	5	6	7
Degré	5	3	2	4	5	3	4	4

4.
```
def degre(G:dict)->dict:
    D = {}
    for s in G:
        D[s] = len(G[s])
    return D
```
```
>>> G = {0:[1,2], 1:[0,2,3], 2:[0,1,3], 3:[1,2,4], 4:[3]}
>>> degre(G)
{0: 2, 1: 3, 2: 3, 3: 3, 4: 1}
```

5.
```
def est_voisins(s, v, G:dict)->bool:
    return v in G[s]
```
```
>>> est_voisins(0, 3, G)
False
>>> est_voisins(0, 2, G)
True
```

PARTIE II — TESTER SI UNE COLORATION EST VALIDE

6. La première est valide, en revanche la deuxième non puisque les sommets 0 et 7 sont colorés de la même couleur et sont pourtant voisins.

7.
```
def coloration_valide(G:dict, C:dict)->bool:
    for s in G:
        c_s = C[s] # couleur de s
        for v in G[s]:
            c_v = C[v]
            if c_v == c_s:
                return False # deux couleurs identiques sur deux sommets adj.
    return True
```
```
>>> G_ex = {0:[1, 3, 4, 6], 1:[0, 2, 3], 2:[1, 3], 3:[0, 1, 2, 4], 4:[0, 3, 5, 6, 7], 5:[4, 6, 7], 6:[0, 4, 5, 7], 7:[0, 4, 5, 6]}
>>> C1 = {0:0, 1:1, 2:0, 3:2, 4:1, 5:0, 6:2, 7:3}
>>> C2 = {0:0, 1:3, 2:0, 3:1, 4:3, 5:2, 6:1, 7:0}
>>> coloration_valide(G_ex, C1)
True
>>> coloration_valide(G_ex, C2)
False
```

8. Dans le pire des cas, le `return False` n'arrête jamais les boucles (cas où les sommets ont des couleurs 2 à 2 distinctes), dans ce cas on a :

$$C_{\text{pire}}(n,m) = \sum_{s\in S}\left(1 + \sum_{v\in V(s)} 2\right) = n + \sum_{s\in S}\sum_{v\in V(s)} 2,$$

où S désigne l'ensemble des sommets de G, et $V(s)$ l'ensemble des voisins d'un sommet $s \in S$. La somme double comporte un nombre $2m$ de termes car chaque arête est parcourue deux fois. Ainsi,

$$C_{\text{pire}}(n,m) = n + 4m = \boxed{O(n+m)}.$$

PARTIE III — FONCTION AUXILIAIRE

9. [Méthode 1]

9.1)

```
1  def premierNonPresent(L:list)->int:
2      """
3      renvoie le plus petit entier naturel non présent
4      dans la liste d'entiers naturels L
5      """
6      for x in L:
7          assert(type(x) == 'int' and x >= 0)
8      i = 0
9      while i in L:
10         i += 1
11     return i
```

9.2) Supposons par l'absurde que la fonction ne termine pas. Alors $(i_k)_{k\in\mathbb{N}}$ est une suite infinie, elle est de plus strictement croissante car $i_{k+1} = i_k + 1$ pour tout $k \in \mathbb{N}$. Donc il existe un rang k_0 à partir duquel $i_{k_0} > \max \mathrm{L}$: c'est une contradiction puisqu'on devrait avoir $i_k \in \mathrm{L}$ pour tout entier $k \in \mathbb{N}$.
Conclusion : la fonction `premierNonPresent` termine.

9.3) Montrons que

$$\mathscr{P}(k) \quad \text{«}[\![0\,,\,k-1]\!] \subset \mathrm{L} \text{ et } i_k = k\text{»} \quad \text{pour } k \in \{0,\dots,\mathrm{N}\}$$

est un invariant de boucle `while`.

Initialisation. $\mathscr{P}(0)$ «$\varnothing \subset \mathrm{L}$ et $i_0 = 0$» est bien vérifié.

Hérédité. Soit $k \in [\![0, \mathrm{N}-1]\!]$ tel que $\mathscr{P}(k)$ soit vraie. Montrons que $\mathscr{P}(k+1)$ est vraie aussi.

- On a $i_{k+1} = i_k + 1 = k + 1$ d'après $\mathscr{P}(k)$.
- Puisqu'il y'a une itération $k+1$, c'est que $i_k = k \in \mathrm{L}$. Mais d'après l'invariant, $[\![0\,,\,k-1]\!] \subset \mathrm{L}$, donc $[\![0\,,\,k]\!] \subset \mathrm{L}$.

Ceci prouve $\mathscr{P}(k+1)$.

Correction. D'après $\mathscr{P}(\mathrm{N})$, on a $[\![0\,,\,\mathrm{N}-1]\!] \subset \mathrm{L}$ et $i_\mathrm{N} = \mathrm{N}$. Comme il n'y a pas d'itération N + 1, c'est que $i_\mathrm{N} = \mathrm{N} \notin \mathrm{L}$. Ainsi :

$$[\![0\,,\,\mathrm{N}-1]\!] \subset \mathrm{L}, \quad \mathrm{N} \notin \mathrm{L}, \quad \text{la fonction renvoie } i_\mathrm{N} = \mathrm{N}$$

c'est bien le premier entier positif non présent dans L.

9.4) On compte dans le pire des cas :

- lignes #6 : une boucle `for` de longueur n, et 2 tests ligne #7
- en ligne #8 : 1 opération ;
- en lignes #9 et #10, on boucle au maximum pour n itérations, avec pour chacune de ces itérations : n opérations pour le test `i in L` et 2 opérations pour `i += 1`, puis une dernière fois n opérations pour dernier test `i in L` qui fait quitter la boucle ;

donc : $\mathrm{C}_n = 2n + 1 + n(n+2) + n = n^2 + 3n + 1 = \boxed{\mathrm{O}(n^2)}$.

10. [Méthode 2] On décide d'utiliser désormais le principe suivant :

- on commence par définir une liste `VU` (comme *valeurs utilisées*) initialisée d'abord avec $n+1$ fois la valeur 0 (où n désigne la taille de `L`),
- puis parcourir une à une chaque valeur e de L, et si $e \leqslant n$, placer VU[e] à 1.
- Une fois ce parcours achevé, il reste à parcourir la liste `VU` à la recherche de la première position contenant encore 0 (on est sûr qu'il y en a au moins une puisque la liste `VU` ayant plus d'éléments que la liste `L`, toutes les positions de `VU` n'ont pas pu être mises à 1). Cette première position contenant 0 est bien le premier entier naturel non présent dans L.

10.1)

```
L1 = [2, 0, 5, 2]
VU1 = [1, 0, 1, 0, 0]
L2 = [2, 0, 1, 3]
VU2 = [1, 1, 1, 1, 0]
```

10.2)

```
1  def premierNonPresent(L : list)->int :
2      """
3      renvoie le plus petit entier naturel non présent \
       ↪ dans la liste d'entiers naturels L
4      """
5      n = len(L)
6      VU = [0]*(n+1)
7      for e in L :
8          if e <= n :
9              VU[e] = 1
10     i = 0
11     while VU[i] == 1 :
12         i += 1
13     return i
```

```
>>> premierNonPresent(L1)
1
>>> premierNonPresent(L2)
4
```

10.3) On compte dans le pire des cas :

- en lignes #5 et #6 : $n+2$ opérations (2 affectations + la création de `[0]*(n+1)`),
- en lignes #7, #8 et #9, on boucle pour n itérations, avec pour chacune de ces itérations au pire 2 opérations ;
- en lignes #10 : 1 opération ;
- en lignes #11 et #12, on boucle pour au pire n itérations, avec pour

chacune de ces itérations 2 opérations, et 1 dernière opération pour quitter la boucle ;
donc $C_n = n+2+2n+1+2n+1 = 5n+4 = \boxed{O(n)}$.

PARTIE IV — UN ALGORITHME INTUITIF DE COLORATION

11.
```
def colore_sommet(C:dict, s:int, G:dict)->None:
    # Construction de la liste U des couleurs déjà utilisées \
    ↪ par les voisins de s
    U = []
    for v in G[s] :  # pour chaque voisin de s
        if v in C :  # s'il a déjà une couleur
            U.append(C[v]) # on ajoute cette couleur à la \
            ↪ liste U
    C[s] = premierNonPresent(U) # on colorie le sommet s
```

```
>>> C = {0:0, 1:1}
>>> colore_sommet(C, 2, G_ex)
>>> C
{0: 0, 1: 1, 2: 0}
```

12.
```
def colorer1(G:dict)->dict:
    C = {}
    for s in G:
        colore_sommet(C, s, G)
    return C
```

```
>>> colorer1(G_ex)
{0: 0, 1: 1, 2: 0, 3: 2, 4: 1, 5: 0, 6: 2, 7: 3}
```

13.
```
def colorer2(S:list, G:dict)->dict:
    C = {}
    for s in S:
        colore_sommet(C, s, G)
    return C
```

```
>>> S = [0, 2, 4, 6, 1, 3, 5, 7]
>>> C = colorer2(S, G_ex)
>>> C
{0: 0, 2: 0, 4: 1, 6: 2, 1: 1, 3: 2, 5: 0, 7: 3}
>>> max(list(C.values()))+1 # le nombre de couleurs
4
```

14. On répond ici à la question avec Python :

```
>>> C = colorer2([7, 6, 5, 4, 3, 2, 1, 0], G_ex)
>>> C
{7: 0, 6: 1, 5: 2, 4: 3, 3: 0, 2: 1, 1: 2, 0: 4}
>>> max(list(C.values()))+1 # le nombre de couleurs
5
```

Il y a ici 5 couleurs utilisées (numérotées 0, 1, 2, 3, 4).

PARTIE V — VARIANTE DE WELSH-POWELL

15. [Préliminaires sur les listes]

15.1)
```
def renverse(L:list)->list:
    L_r = []
    n = len(L)
    for k in range(n):
        L_r.append(L[n-1-k])
    return L_r
```

```
>>> renverse([1, 2, 3])
[3, 2, 1]
```

15.2) Par exemple par compréhension.
```
def init(n:int)->list:
    return [[] for _ in range(n)]
```

```
>>> init(3)
[[], [], []]
```

15.3)
```
def ranger(G:dict)->[list]:
    D = degre(G) # dictionnaire des degrés
    # calcul du degré maximal possible
    n = -1
    for s in D:
        if D[s] >= n:
            n = D[s]
    # complétion de la liste
    R = init(n+1)
    for s in D:
        d_s = D[s]
        R[d_s].append(s)
    return R
```

```
>>> G = {0:[1,2], 1:[0,2,3], 2:[0,1,3], 3:[1,2,4], 4:[3]}
>>> ranger(G)
[[], [4], [0], [1, 2, 3]]
```

16.
```python
def trier_sommets(G:dict)->list:
    R = renverse(ranger(G))
    S_deg = []
    for L_deg_cst in R:
        S_deg += L_deg_cst
    return S_deg
```

```
>>> G = {0:[1,2], 1:[0,2,3], 2:[0,1,3], 3:[1,2,4], 4:[3]}
>>> trier_sommets(G)
[1, 2, 3, 0, 4]
```

17.
```python
def colorer3(G:dict)->dict:
    S = trier_sommets(G)
    return colorer2(S, G)
```

18.
```
>>> trier_sommets(G_ex)
[4, 0, 3, 6, 7, 1, 5, 2]
>>> colorer3(G_ex)
{4: 0, 0: 1, 3: 2, 6: 2, 7: 3, 1: 0, 5: 1, 2: 1}
```

Mais

```
{0:0, 4:1, 3:2, 6:2, 7:3,1:3,5:0,2:0}
```

est aussi accepté.

PARTIE VI — ALGORITHME DSATUR

Lorsque l'on colore un graphe, le *degré de saturation* d'un sommet est le nombre de couleurs différentes déjà attribuées à ses différents sommets voisins. Évidemment, ce degré de saturation est susceptible d'évoluer à chaque fois qu'un nouveau sommet est coloré.

19. **[Préliminaire sur les listes]**
```python
def sans_doublon(L:list)->list:
    """
    renvoie la liste des éléments de L, chaque élément \
    ↪ apparaissant une unique fois
    """
    M = []
    for x in L:
        if x not in M:
            M.append(x)
    return M
```

20.
```python
def degre_satur(C:dict, s:int, G:dict)->int:
    L_cou = [] # liste des couleurs des voisins de s
    for v in G[s]:
        if v in C: # v est donc déjà coloré
            L_cou.append(C[v]) # on ajoute la couleur dans \
            ↪ L_cou
    return len(sans_doublon(L_cou))
```

21.
```python
def liste_satur(C:dict, G:dict)->list:
    d_sat_max, L_sat_max = -1, [] # recherche de max sur le \
    ↪ degré de sat.
    for s in G:
        if s not in C:
            d_sat = degre_satur(C, s, G)
            if d_sat > d_sat_max:
                d_sat_max = d_sat
                L_sat_max = [s]
            elif d_sat == d_sat_max:
                L_sat_max.append(s)
    return L_sat_max
```

22.
```python
def colorer4(G:dict)->dict:
    C = {} # dictionnaire de coloriage
    n = len(G)
    while len(C) < n:
        L_sat_max = liste_satur(C, G)
        s = L_sat_max[0]
        colore_sommet(C, s, G)
    return C
```

```
>>> C = colorer3(G_ex)
>>> C
{4: 0, 0: 1, 3: 2, 6: 2, 7: 3, 1: 0, 5: 1, 2: 1}
>>> max(list(C.values()))+1 # le nombre de couleurs
4
>>> G
{0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3], 3: [1, 2, 4], 4: [3]}
>>> C = colorer3(G)
>>> C
{1: 0, 2: 1, 3: 2, 0: 2, 4: 0}
>>> max(list(C.values()))+1 # le nombre de couleurs
3
>>> C = colorer4(G)
>>> C
```

```
{0: 0, 1: 1, 2: 2, 3: 0, 4: 1}
>>> max(list(C.values()))+1 # le nombre de couleurs
3
```