LYCEE CHARLEMAGNE
Mardi 14 mai
M.P.S.I.2

2023 2024

ITC1

Document réponse	Nom :	Prénom :

COLORER UN GRAPHE

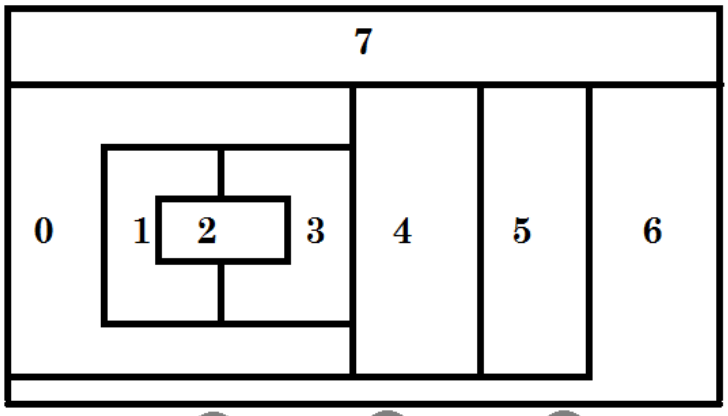
La coloration d'une carte de pays consiste à attribuer une couleur à chacun des pays de manière à ce que deux pays voisins soient de couleurs différentes. Étudier ces techniques de coloration revient de façon plus abstraite à travailler sur des graphes. Le champ d'applications de la coloration de graphes 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.
Quelques rappels de syntaxe `Python` *figurent dans l'annexe 2, et quelques cartes en annexe 3.*

DES ALGORITHMES POUR COLORER UN GRAPHE

Introduction sur un exemple 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 sont 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 que l'on représentera par le graphe `Gex` *donné en figure 2.*
Ainsi, sur le graphe de la figure 2, deux pays sont voisins si et seulement si les sommets correspondants sont reliés par une arête.

Figure 1



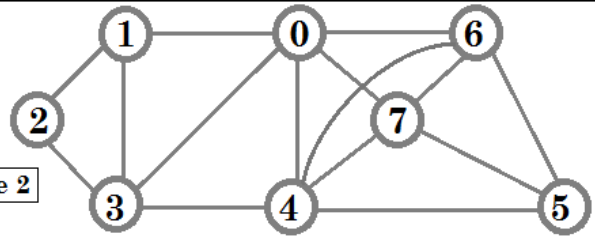
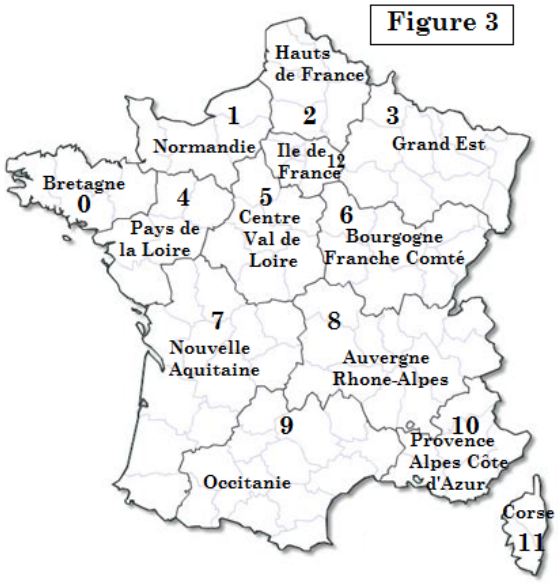


Figure 2

Figure 3



Q1. Un graphe peut être représenté par une matrice d'adjacence. Le DR contient la matrice d'adjacence pré-remplie du graphe `Gex`. Complétez la et expliquez le processus de construction.

Donnez également la matrice d'adjacence des treize régions de France Métropolitaine. [1]

Gex :

0	1	0	1	1			
	0						
		0					
			0				
				0			
0					0		
						0	
							0

Régions :

0												
	0											
		0										
			0									
				0								
					0							
						0						
							0					
								0				
									0			
										0		
											0	
												0

Q2. Une autre manière de représenter en mémoire un graphe est d'utiliser une liste d'adjacence. La liste d'adjacence d'un graphe (constitué de sommets numérotés de `0` à `n - 1`) est une liste `LA`, telle que `LA[i]` est la liste des sommets voisins du sommet i. Par convention, les voisins énumérés dans `LA[i]` seront listés dans l'ordre croissant. Donnez la liste d'adjacence du graphe `Gex` présenté en exemple.

.

Même question pour la carte de France

.

Q3. Donnez un avantage et un inconvénient d'une représentation par matrice d'adjacence.

.

Donnez un avantage et un inconvénient d'une représentation par liste d'adjacence.

.

Q4. On rappelle que le degré d'un sommet `s` d'un graphe `G` est le nombre de voisins du sommet `s`, c'est-à-dire le nombre de sommets reliés à `s`. Complétez le le tableau des degrés des différents sommets des deux graphes.

Gex	Sommet	0	1	2	3	4	5	6	7
	Degré								

France	Sommet	0	1	2	3	4	5	6	7	8	9	10	11	12
	Degré													

Montrez que la somme des degrés des sommets du graphe est un entier pair.

1. vous avez de la chance, quand j'avais votre âge, il y en avait 22, la réforme qui a fusionné par exemple deux Normandies en une date de 2010

.

Montrez que le nombre de sommets dont le degré est impair est pair.

.

TESTER SI UNE COLORATION EST VALIDE

Q5. Écrivez une fonction `voisins` avec trois arguments, deux nombres entiers distincts `i` et `j` et une liste d'adjacence `LA` représentant un graphe, qui renvoie `True` si les sommets numérotés `i` et `j` sont reliés par une arête et `False` sinon.

```
def voisins(i, j, LA) : #int, int, list of list of int -> boolean
....
....
....
....
....
....
```

On considère une carte de pays, représentée par un graphe `G` pour laquelle on a une proposition de coloration donnée par une liste `C`. Ainsi, `C[i]` donne le numéro de la couleur attribuée au sommet `i`. 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).

Q6. Écrivez la fonction `coloration_valide` avec pour arguments une liste d'adjacence `LA` et une liste de couleurs `C`, qui renvoie `True` si la coloration est valide, `False` sinon.

```
def coloration_valide(LA, C) : #list of list of int, list of int -> boolean
....
....
....
....
....
....
....
....
```

Q7. Pour un graphe comportant `n` sommets, quelle est la complexité temporelle dans le pire des cas de la fonction `coloration_valide` ?

.

UN ALGORITHME INTUITIF DE COLORATION

L'attribution des couleurs à chaque sommet est caractérisée par une liste `C` où `C[i]` est la couleur attribuée au sommet `i` ; `C[i]` vaut `-1` si la couleur n'est pas encore attribuée. La liste `C` ne contient que des `-1` au départ et ses valeurs sont modifiées progressivement au fur et à mesure que les couleurs sont attribuées.

Q8. On suppose (dans cette question seulement) que `n` est une constante déjà définie. Écrivez la ou les instruction(s) permettant de créer une liste initiale `C` composée de `n` éléments valant `-1`.

```
.
```

Q9. Complétez la fonction `colore_sommet` ayant trois arguments, la liste `C` des couleurs attribuées, le numéro `s` du sommet à colorer et la liste d'adjacence `LA` caractérisant le graphe. Cette fonction ne renvoie rien mais modifie la liste `C` en donnant à `C[s]` la plus petite couleur possible, en fonction des couleurs des sommets voisins qui sont déjà colorés. [2]

```
def colore_sommet(C,s,LA) : #spécifications
....#on détermine la liste des couleurs des voisins de s déjà colorés
....coul_vois = [ ]
....
....
....
....
....
....#coul_vois est maintenant déterminée et on recherche la
....#plus petite couleur, notée num_coul, absente de coul_vois
....
....
....
....
....
....#la valeur num_coul trouvée devient la couleur du sommet s
....
....
....
```

Q10. À l'aide de Q9, écrivez une fonction `colorer1` avec pour argument une liste `LA` caractérisant un graphe, qui crée et renvoie la liste `C` des numéros des couleurs attribuées en colorant les sommets un par un par ordre croissant de leurs numéros. [3]

```
def colorer1(LA) : #list of list of int -> list of int
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
```

Q11. L'ordre de coloration imposé à la question précédente est arbitraire. On souhaite maintenant colorer le graphe en traitant les sommets selon un ordre arbitraire donné en argument. Écrivez une fonction `colorer2` analogue à `colorer1` et avec un argument supplémentaire, une liste `ordre` fixant l'ordre de coloration des pays. [4]

2. Par exemple, pour le graphe `Gex`, prenons `C=[0,1,-1,-1,-1,-1,-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,LA)` modifie la liste `C` en `C=[0,1,0,-1,-1,-1,-1,-1]`. 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`.

3. Par exemple, l'application de la fonction colorer1 au graphe `Gex` renverra la liste de couleurs `[0,1,0,2,1,0,2,3]`.

4. Par exemple `colorer2([0,2,4,6,1,3,5,7], LA)` colorera le graphe `Gex` en commençant d'abord par le sommet `0`, puis en continuant par les sommets `2`, `4`, `6`...

```
def colorer2(ordre, LA) : #list of int, list of list of int -> list of int
....
....
....
....
....
....
....
....
....
....
....
....
....
```

Q12. Donnez la liste des couleurs renvoyée par `colorer2` pour colorer le graphe `Gex` donné en exemple en prenant `ordre=[7,6,5,4,3,2,1,0]`. Combien de couleurs ont elles été utilisées ?
Même question avec `France` en prenant pour ordre les régions dans l'ordre alphabétique : `ordre = [8, 6, 0, ..., 1, 7, ..., 10]`

Gex	0	1	2	3	4	5	6	7

France	0	1	2	3	4	5	6	7	8	9	10	11	12

La méthode que nous venons de décrire est rapide 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 sous-parties suivantes est d'affiner la stratégie pour mieux choisir cet ordre de coloration.

VARIANTE DE WELSH-POWELL

Une alternative est donnée par la variante de Welsh-Powell[5]*. 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).*

Q13. Écrivez une fonction `degre` avec pour argument la liste d'adjacence `LA` d'un graphe quelconque, qui renvoie la liste des degrés des sommets du graphe. [6]

```
def degre(LA) : #list of list of int -> list of int
....
....
....
....
....
....
```

Q14. Écrivez une fonction `init` avec pour argument un entier `n`, qui renvoie une liste de listes `R` de taille `n`, telle que `R[i]` soit une liste vide. Par exemple, `init(3)` renverra `[[ ], [ ], [ ]]`.

```
def init(n) : #int -> list of list
....
....
....
....
....
....
```

5. *détail amusant : quand j'ai cherché l'algorithme de Welsh-Powell sur internet, on m'a proposé l'algorithme de Gallois-Powell, do you understand why ?*

6. Par exemple, pour un graphe de liste d'adjacence `LA = [[1,2], [0,2,3], [0,1,3], [1,2,4], [3]]`, la fonction degre renverra la liste `[2,3,3,3,1]`.

Q15. Écrivez une fonction `ranger` avec pour argument une liste d'adjacence `LA`, qui renvoie une liste `R` de même taille que `LA`, telle que `R[i]` soit la liste des sommets de degré `i` [7]

```
def ranger(LA) : #list of list of int -> list of list of int
....
....
....
....
....
....
```

Indiquez la réponse pour `ranger(France)`.

.

Q16. Écrivez une fonction `renverse` avec pour argument une liste `L`, qui crée et renvoie une nouvelle liste obtenue en lisant `L` dans l'ordre inverse. [8]

```
def renverse(L) : #list -> list
....
....
....
....
```

Q17. Écrivez une fonction `trier_sommets` avec pour argument une liste d'adjacence `LA`, qui renvoie la liste des sommets triés dans l'ordre décroissant de leur degré. [9]

```
def trier_sommets(LA) : #list of list of int -> list of int
....
....
....
....
....
....
....
....
....
....
```

Q18. Pour un graphe à `n` sommets, quelle est la complexité temporelle de la fonction `trier_sommets` dans le pire des cas ?

.

Q19. Écrivez la fonction `colorer3` avec pour argument une liste d'adjacence `LA`, qui crée et renvoie une liste de couleurs `C`, telle que `C[i]` soit la couleur à attribuer au sommet numéro `i`, les sommets étant colorés dans l'ordre décroissant de leur degré. Quelle est la complexité de `colorer3` dans le pire des cas pour un graphe à `n` sommets ?

7. Ainsi, pour l'exemple de la question Q13, l'appel `ranger(LA)` renverra la liste `[[ ], [4], [0], [1,2,3], [ ]]`.
8. Par exemple, `renverse([1,2,3,4])` renverra `[4,3,2,1]`.
9. Par exemple, pour un graphe de liste d'adjacence `LA=[[1,2],[0,2,3],[0,1,3],[1,2,4],[3]]`, la fonction `trier_sommets` renverra la liste de sommets `[1,2,3,0,4]`.

```
def colorer3(LA) : #list of list of int -> list of int
....
....
....
....
....
....
....
....
....
....
```

Q20. Pour les graphes `Gex` et `France`, donnez la liste `C` des couleurs renvoyée par la fonction `colorer3`.

Gex	0	1	2	3	4	5	6	7

France	0	1	2	3	4	5	6	7	8	9	10	11	12

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 Danier 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.

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.

Q21. Écrivez une fonction `degre_satur` avec 3 arguments, une liste d'adjacence `LA`, un sommet `s` du graphe, une liste `C` de couleurs. Cette fonction renvoie le degré de saturation du sommet `s`. On rappelle que le sommet `i` est coloré si et seulement si `C[i]` est différent de `-1`.

```
def degre_satur(LA, s, C) : #list of list of int, int, list of int -> int
....
....
....
....
....
....
....
....
....
....
```

Q22. Écrivez une fonction `liste_satur` avec deux arguments, une liste d'adjacence `LA`, la liste `C` des couleurs des sommets, qui renvoie la liste des sommets non colorés du graphe ayant un degré de saturation maximum 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.

```
def liste_satur(LA, C) : #list of int, list of int -> list of int
....
....
....
....
....
....
....
....
....
....
```

Q23. Écrire une fonction `pas_fini` avec pour argument une liste `C`, qui renvoie `True` si cette liste contient la valeur `-1`, `False` sinon.

```
def pas_fini(C) : #list of int -> boolean
....
....
....
....
....
....
....
```

Q24. Compléter la fonction `colorer4` ayant pour argument une liste d'adjacence `LA`, qui renvoie une liste `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 parmi ceux-ci qui soit de degré maximal ;
– colorer le sommet choisi en lui attribuant la couleur disponible ayant la plus petite valeur.

```
def colorer4(LA) : #spécifications
....n =
....#nombre de sommets du graphe
....D =
....#liste des degrés des sommets du graphe
....C =
....#initialisation de la liste des couleurs
....while .......... :
........#liste des sommets non colorés de degré de saturation maximal
........Ls =
........#en cas d'égalité, recherche du/d'un sommet de degré maximal
........
........
........
........
........
........#coloration du sommet prioritaire
........
........
....return C
```

Il n'est pas facile d'être certain d'avoir utilisé le nombre minimum de couleurs. La sous-partie suivante propose une piste.

UN MINORANT DU NOMBRE DE COULEURS NÉCESSAIRES

Considérons un graphe `G` et un sous-ensemble `K` de sommets de ce graphe. On dit que `K` forme une clique si et seulement si pour chaque paire de sommets de `K`, il existe une arête les reliant. Par exemple, sur le graphe `Gex` de la figure 2, `(4, 5, 6, 7)` constitue une clique de 4 sommets et `(4, 5, 6)` une clique de 3 sommets. Par contre, `(0, 4, 5, 6, 7)` ne forme pas une clique puisque le sommet `0` n'est pas relié au sommet `5`. Enfin, on

note **nc** *le cardinal de la plus grande clique de* **G**.

Q25. Justifiez que le nombre minimum de couleurs pour colorer un graphe est supérieur ou égal au nombre **nc**.

.

Montrez également que : $nc \leqslant 1 + max\{deg(s),\ s \in G\}$ où $deg(s)$ désigne le degré du sommet s dans le graphe G.

.

Dans ce qui suit, on suppose toujours que les sommets d'un graphe à n sommets sont numérotés de 0 à n − 1.

Q26. Écrivez une fonction **est_clique** avec deux arguments, la liste d'adjacence **LA** d'un graphe et un tuple **K** de sommets de ce graphe, qui renvoie **True** si **K** est une clique, **False** sinon.

```
def est_clique(LA, K) : #list of list of int, list -> boolean
....
....
....
....
....
```

Q27. La fonction **combinations** du module **itertools**, présentée à l'annexe 2, fournit toutes les combinaisons de taille fixée d'une liste. Compléter la fonction **minoration_nb_couleurs** ayant pour argument la liste d'adjacence **LA** d'un graphe, qui renvoie le cardinal de la plus grande clique du graphe considéré.

```
from itertools import combinations
def minoration_nb_couleurs (LA) :
....n =
....#nombre de sommets du graphe
....S = [k for k in range(n)]
....#liste des sommets du graphe
....i =
....test = True
....while test :
........for K in combinations (S,i) :
............
............
............
............
............
............
............
........i = i-1
....return
```

Ainsi, si le nombre de couleurs utilisées pour colorer le graphe est égal à **nc**, on est sûr d'avoir utilisé le nombre minimum de couleurs.

ANNEXE 2
Rappels de syntaxe **Python**

Test d'appartenance

```
>>> 2 in [1, 2, 3, 4]
True
>>> 2 in [1, 3, 4]
False
```

Définir une liste

```
>>> L = [1, 2, 3]
>>> L[0]
1
```

Ajouter un élément à la fin d'une liste

```
>>> L = [1 ,2 ,3]
>>> L
[1, 2, 3]
>>> L.append(5)
>>> L
[1, 2, 3, 5]
```

Ajouter tous les éléments d'une liste `L1` à la fin d'une liste `L`

```
>>> L = [6, 8, 7]
>>> L1 = [-1, -2]
>>> L.extend(L1)
>>> L
[6, 8, 7, -1, -2]
```

Obtenir les combinaisons d'une taille donnée d'une liste

```
>>> from itertools import combinations
>>> L = [0, 1, 2, 3, 4]
>>> for C in combinations(L, 3) :
.....print(C)
(0, 1, 2)
(0, 1, 3)
(0, 1, 4)
(0, 2, 3)
(0, 2, 4)
(0, 3, 4)
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)
```

Le sujet se poursuivait avec des questions d'interrogation en SQL d'une base de données.

Je vous demande plutôt

un convertisseur qui prend une matrice d'adjacence et crée la liste adjacence

```
def matrix_to_list(M) : #spécifications
....
....
....
....
....
```

un convertisseur qui prend en entrée une liste d'adjacences et crée la matrice d'adjacence

```
def list_to_matrix(M) : #spécifications
....
....
....
....
....
```

une fonction combinaison qui prend en entrée une liste et un entier k (plus petit que la longueur n de la liste) et retourne les $\binom{n}{k}$ combinaisons sous forme de liste de listes.

```
def combinaisons(L, k) : #list, int
....
....
....
....
....
....
....
....
....
....
....
```

Annexe 3

Figure 1

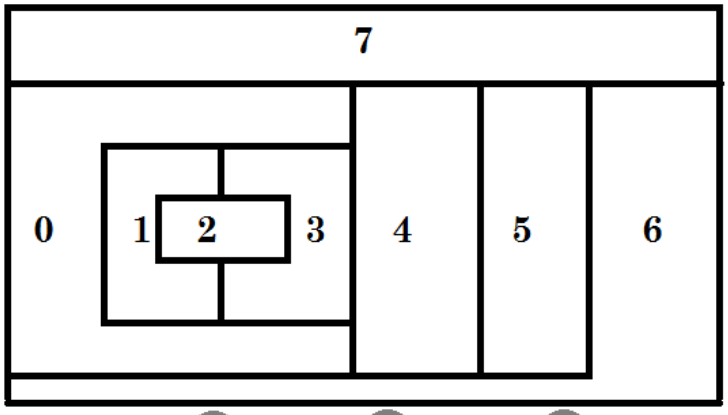


Figure 2

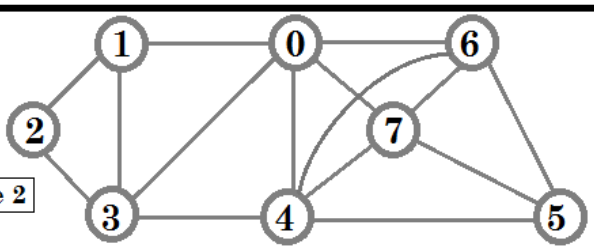


Figure 3

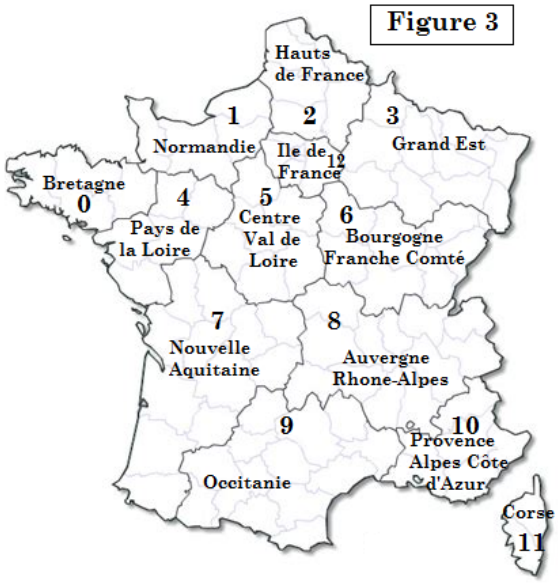


Figure 1

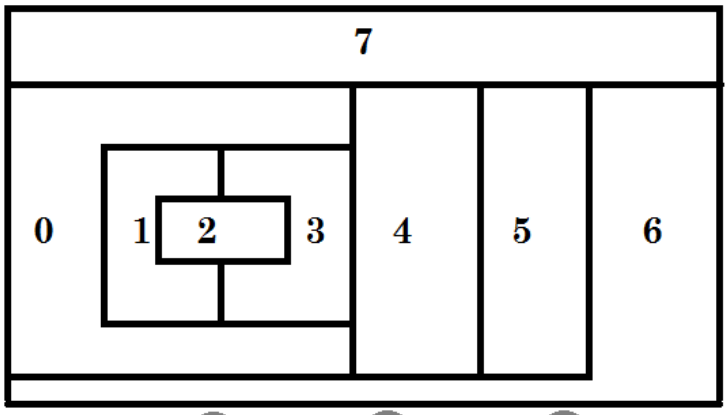


Figure 2

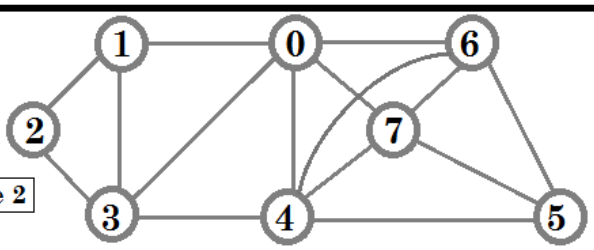


Figure 3

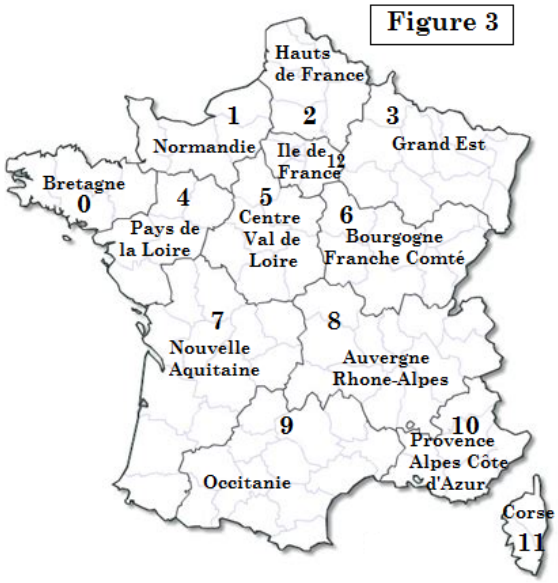


Figure 1

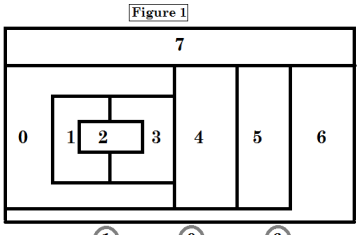


Figure 2

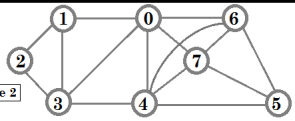


Figure 3

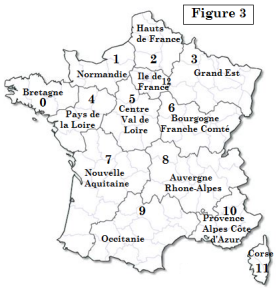


Figure 1

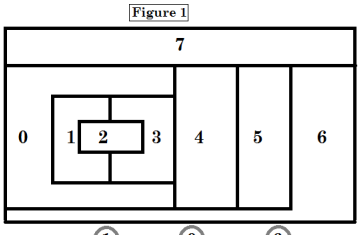


Figure 2

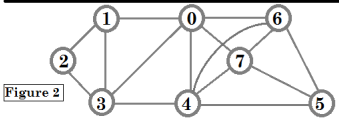
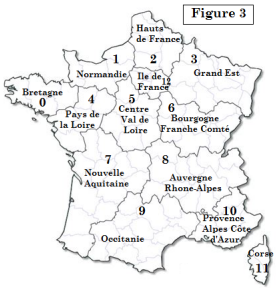


Figure 3



LYCEE CHARLEMAGNE
Mardi 14 mai
M.P.S.I.2

2023 2024
ITC1
CORRECTION

Q1. Un graphe peut être représenté par une matrice d'adjacence. Le DR contient la matrice d'adjacence pré-remplie du graphe `Gex`. Complétez la et expliquez le processus de construction.
Donnez également la matrice d'adjacence des treize régions de France Métropolitaine.

Gex :

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
1	0	0	1	0	1	1	1
0	0	0	0	1	0	1	1
1	0	0	0	1	1	0	1
1	0	0	0	1	1	1	0

Régions :

0	1			1								
1	0	1		1	1							1
	1	0	1									1
		1	0			1						1
1	1			0	1		1					
	1			1	0	1	1	1				1
			1		1	0		1				1
				1	1		0	1	1			
					1	1	1	0	1	1		
							1	1	0	1		
								1	1	0		
											0	
	1	1	1		1	1						0

Pour la lisibilité, dans le seconde matrice, je n'ai mis que les 1, tout le reste, c'est des 0.
Il va de soi que ces matrices sont symétriques.
Enfin, on note le statut insulaire de la Corse, qui n'est reliée aucun autre sommet. Sa couleur pourra être choisie au hasard.
Je n'ai pas porté les régions ultramarines, pour lesquelles on a aussi des sous-graphes isolés.

Q2. Une autre manière de représenter en mémoire un graphe est d'utiliser une liste d'adjacence. La liste d'adjacence d'un graphe (constitué de sommets numérotés de `0` à `n - 1`) est une liste `LA`, telle que `LA[i]` est la liste des sommets voisins du sommet i. Par convention, les voisins énumérés dans `LA[i]` seront listés dans l'ordre croissant.
Donnez la liste d'adjacence du graphe `Gex` présenté en exemple.

```
LA = [ [1, 3, 4, 6, 7],  [0, 2, 3],  [1, 3],  [0, 1, 2, 4],  [0, 3, 5, 6, 7],
       [4, 6, 7],  [0, 4, 5, 7],  [0, 4, 5, 6]]
```

Et pour la carte de France

```
LA = [ [1, 4],  [0, 2, 4, 5, 12],  [1, 3, 12],  [2, 6, 12],  [0, 1, 5, 7],  [1, 4, 6, 7, 8, 12],  [3, 5, 8, 12],
       [4, 5, 8, 9],  [5, 6, 7, 9, 10],  [7, 8, 10],  [8, 9],  [ ],  [1, 2, 3, 5, 6]]
```

J'ai juste aéré un peu pour la lisibilité.
J'espère bien pas croiser d'erreurs sur cette question.

Q3. Donnez un avantage et un inconvénient d'une représentation par matrice d'adjacence.

On trouve en un temps O(1) si les deux sommets i et j sont en relation.
Mais ça prend de la place en mémoire si la graphe est grand $O(n^2)$.

Donnez un avantage et un inconvénient d'une représentation par liste d'adjacence.

Si le graphe n'est pas trop « dense », on n'occupe que peu de place en mémoire.
Mais pour savoir si i et j sont reliés, il faut accéder à la liste de i (direct temps $O(1)$), puis parcourir cette liste à la recherche de j (cette fois complexité $O(n)$).
Globalement, complexité $O(n)$.
Ou plutôt $O(\ln(n))$ car les listes sont triées).

Q4. On rappelle que le degré d'un sommet `s` d'un graphe `G` est le nombre de voisins du sommet `s`, c'est-à-dire le nombre de sommets reliés à `s`. Complétez le le tableau des degrés des différents sommets des deux graphes.

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

France	Sommet	0	1	2	3	4	5	6	7	8	9	10	11	12
	Degré	2	5	3	3	4	6	4	4	5	3	2	0	5

Montrez que la somme des degrés des sommets du graphe est un entier pair.

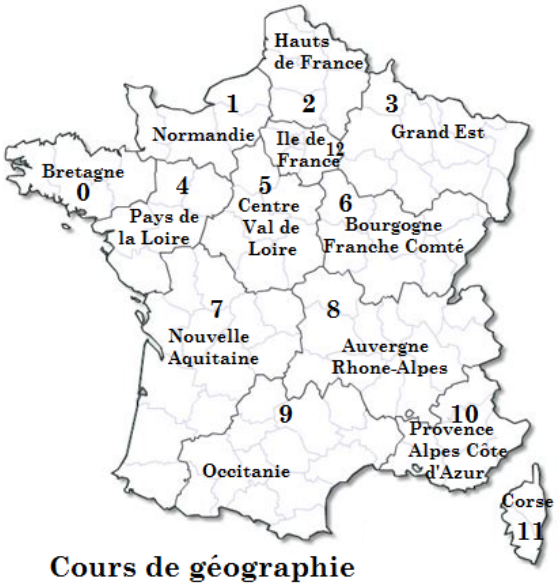
Le degré d'un sommet est le nombre d'arêtes issues de ce sommet.
En sommant sur tous les sommets, on compte toutes les arrêtes.
Mais chacune est comptée deux fois (une fois par sommet).
En pur matheux, on note $a_{i,j} = 1$ si il y a une arête entre i et j (et 0 sinon) On a donc $a_{i,i} = 0$ et $a_{i,j} = a_{j,i}$.
On a alors $\deg(i) = \sum_j a_{i,j}$ puis

$$\sum_i \deg(i) = \sum_i \sum_j a_{i,j} = \sum_{i,j} a_{i,j} = \sum_{i<j} a_{i,j} + \sum_{i=j} a_{i,j} + \sum_{j<i} a_{i,j} = \sum_{i<j} a_{i,j} + 0 + \sum_{j<i} a_{ji,i} = 2.\sum_{j<i} a_{ji,i}$$

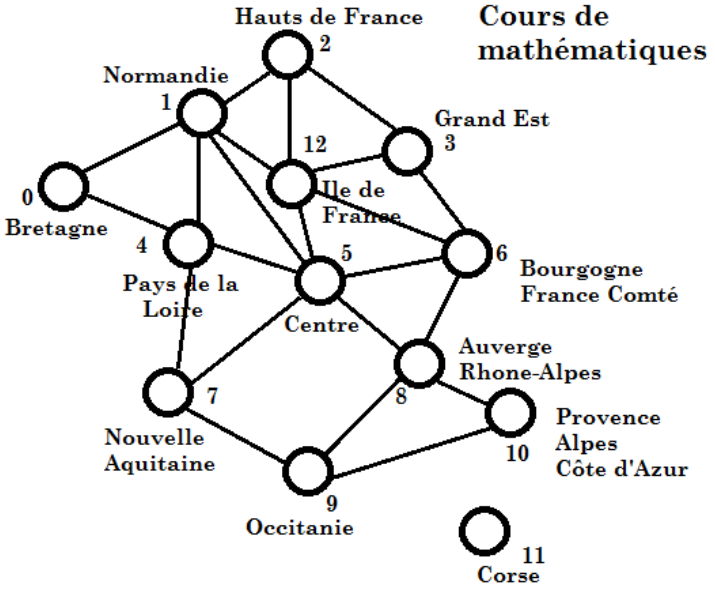
Montrez que le nombre de sommets dont le degré est impair est pair.

On sépare la somme $\sum_i \deg(i)$ en $\sum_{\deg(i)\ pair} \deg(i) + \sum_{\deg(i)\ impair} \deg(i)$.
La première somme est paire par construction, la somme totale aussi.
C'est donc que la seconde est paire.
Et pour qu'une somme de nombres impairs soit paire, il faut qu'il y ait un nombre pair de termes impairs.

Sinon, il est peut être encore plus simple de détruire $\sum_i \deg(i)$ modulo 2. Et dans $\sum_i (\deg(i)\%2)$ on a juste un compteur des sommets de degré impair.



Cours de géographie



TESTER SI UNE COLORATION EST VALIDE

Q5. Écrivez une fonction `voisins` avec trois arguments, deux nombres entiers distincts `i` et `j` et une liste d'adjacence `LA` représentant un graphe, qui renvoie `True` si les sommets numérotés `i` et `j` sont reliés par une arête et `False` sinon.

Il suffit d'aller voir si l'indice `j` est dans la liste `LA[i]`. Et on a le `in` qui fait ça.

```
def voisins(i, j, LA) : #int, int, list of list of int -> boolean
....if j in LA[i] :
........return True
....else :
........return False
```

Et plus rapide encore car on est en salle 210

```
def voisins(i, j, LA) : #int, int, list of list of int -> boolean
....return j in LA[i] #retourne un booléen évalué
```

Q6. Écrivez la fonction `coloration_valide` avec pour arguments une liste d'adjacence `LA` et une liste de couleurs `C`, qui renvoie `True` si la coloration est valide, `False` sinon.

On va parcourir tous les couples de sommets (i, j).
Mais pour ne pas faire deux fois le même travail, on va imposer i < j (strictement, un sommet n'est pas relié à lui même).
Si les sommets sont voisins, on regarde si ils ont la même couleur.
En cas d'échec, on sort tout de suite.
En revanche, il faut attendre la fin de toutes les boucles pour répondre `True`.

```
def coloration_valide(LA, C) : #list of list of int, list of int -> boolean
....for j in range(len(LA)) :
........for i in range(j) :
............if voisins(i, j, LA) :
................if C[i] == C|[j] :
....................return False
....return True #je vais surveiller l'indentation ici
```

Si on ne veut pas sortir en cours de boucle (rigueur informatique ?), on crée un booléen initialisé à True, et en cas d'échec, on le met à False (et il le reste jusqu'à la fin).
Il suffit à la fin de retourner ce booléen.

Mais c'est bête de s'être fait influencer avec voisins.
Il suffit de prendre chaque sommet et de regarder ses voisins.

```
def coloration_valide(LA, C) : #list of list of int, list of int -> boolean
....for i in range(len(LA)) :
........for j in LA[i] : #on ne regarde que ses voisins
............if C[i] == C|[j] :
................return False #un voisin de la même couleur que i
....return True #tout s'est bien passé
```

Q7. Pour un graphe comportant `n` sommets, quelle est la complexité temporelle dans le pire des cas de la fonction `coloration_valide` ?

Si le graphe est une clique (tous les sommets sont reliés entre eux), on doit faire n.(n-1) tests.
On note que pour ne pas tester deux fois chaque arête, on peut se restreindre à j < i.

```
def coloration_valide(LA, C) : #list of list of int, list of int -> boolean
....for i in range(len(LA)) :
........for j in LA[i] : #on ne regarde que ses voisins
............if j> i :
................break #on quitte la boucle j et on passe au i suivant
............if C[i] == C|[j] :
................return False #un voisin de la même couleur que i
....return True #tout s'est passé sans anicroche
```

Quoi, il n'est pas belle la tournure « sans anicroche » ?

UN ALGORITHME INTUITIF DE COLORATION

Q8. On suppose (dans cette question seulement) que `n` est une constante déjà définie. Écrivez la ou les instruction(s) permettant de créer une liste initiale `C` composée de `n` éléments valant `-1`.

`C = [-1]*n` et c'est tout !

Et si on préfère éviter les confusions

```
C = [-1 for i in range(n)]
```

Les bourrins taperont

```
C = [ ]
for i in range(n) :
....C.append(-1)
```

Et si n n'est pas donné, on prend `len(LA)`.

Q9. Complétez la fonction `colore_sommet` ayant trois arguments, la liste `C` des couleurs attribuées, le numéro `s` du sommet à colorer et la liste d'adjacence `LA` caractérisant le graphe. Cette fonction ne renvoie rien mais modifie la liste `C` en donnant à `C[s]` la plus petite couleur possible, en fonction des couleurs des sommets voisins qui sont déjà colorés.

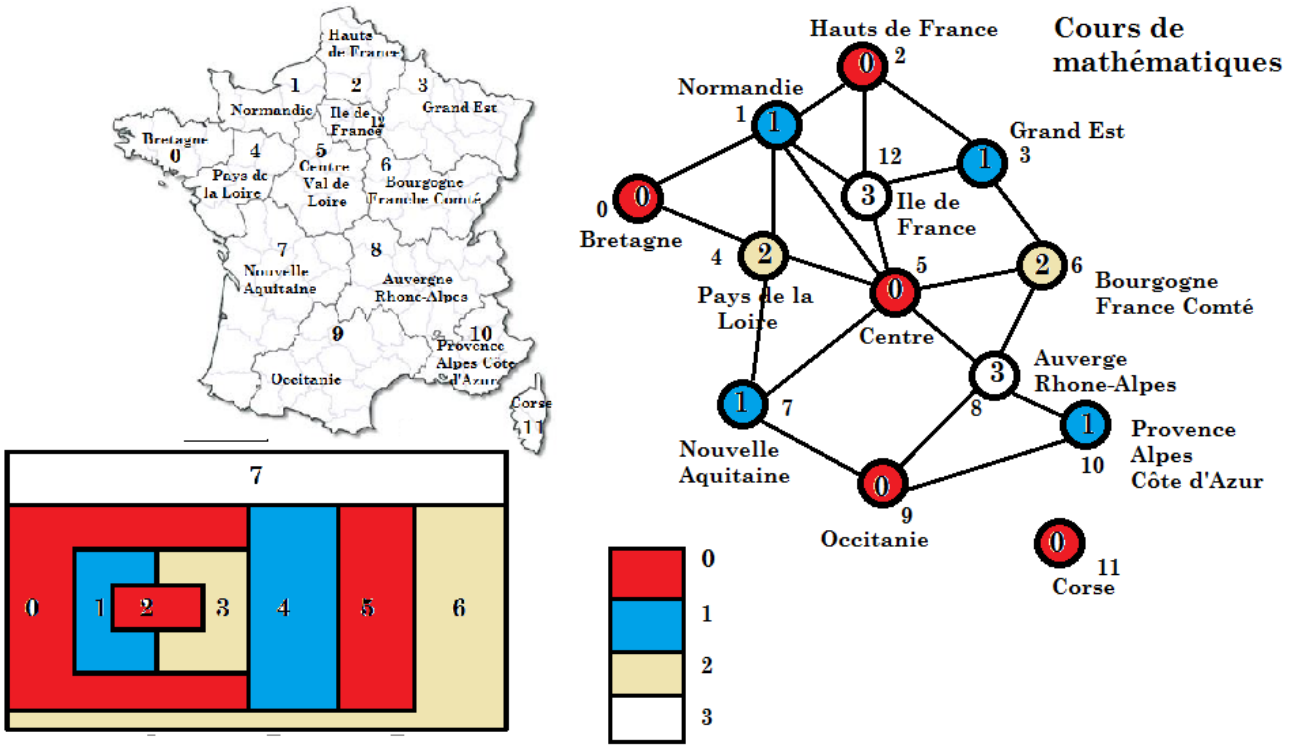
```
def colore_sommet(C, s, LA) : #spécifications
....#on détermine la liste des couleurs des voisins de s déjà colorés
....coul_vois = [ ] #d'abord une liste vide
....for j in LA[s] : #on regarde ses voisins un par un
........if C[j] != -1 : #si le voisin est coloré
............coul_vois.append(C[j]) #on ajoute sa couleur à la liste
.....................................#si la coloration est bonne, pas de doublon
....#coul_vois est maintenant déterminée et on recherche la
....#plus petite couleur, notée num_coul, absente de coul_vois
....num_coul = 0 #on va commencer par tester la couleur 0
....while num_coul in coul_vois : #tant qu'il s'agit d'une couleur prise
.........num_coul += 1 #couleur suivante à tester

....#la valeur num_coul trouvée devient la couleur du sommet s
....C[s] = num_coul
```

Q10. À l'aide de Q9, écrivez une fonction `colorer1` avec pour argument une liste `LA` caractérisant un graphe, qui crée et renvoie la liste `C` des numéros des couleurs attribuées en colorant les sommets un par un par ordre croissant de leurs numéros.

```
def colorer1(LA) : #list of list of int -> list of int
....n = len(LA) #pour être tranquille
....C = [-1]*n #inutile d'utiliser la procédure
....for i in range(n) : #on les prend un par un
........colore_sommet(C, i, LA) #là, on utilise la procédure
........#print(C) ce serait intéressant de voir la progression
....return C #c'est fini
```

Et voici les valeurs de C au fil de l'exécution

```
                                  [0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
                                  [0, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
                                  [0, 1, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
                                  [0, 1, 0, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
[-1, -1, -1, -1, -1, -1, -1, -1] [0, 1, 0, 1, 2, -1, -1, -1, -1, -1, -1, -1, -1]
[0, -1, -1, -1, -1, -1, -1, -1]  [0, 1, 0, 1, 2, 0, -1, -1, -1, -1, -1, -1, -1]
[0, 1, -1, -1, -1, -1, -1, -1]   [0, 1, 0, 1, 2, 0, 2, -1, -1, -1, -1, -1, -1]
[0, 1, 0, -1, -1, -1, -1, -1]    [0, 1, 0, 1, 2, 0, 2, 1, -1, -1, -1, -1, -1]
[0, 1, 0, 2, -1, -1, -1, -1]     [0, 1, 0, 1, 2, 0, 2, 1, 3, -1, -1, -1, -1]
[0, 1, 0, 2, 1, -1, -1, -1]      [0, 1, 0, 1, 2, 0, 2, 1, 3, 0, -1, -1, -1]
[0, 1, 0, 2, 1, 0, -1, -1]       [0, 1, 0, 1, 2, 0, 2, 1, 3, 0, 1, -1, -1]
[0, 1, 0, 2, 1, 0, 2, -1]        [0, 1, 0, 1, 2, 0, 2, 1, 3, 0, 1, 0, -1]
[0, 1, 0, 2, 1, 0, 2, 3]         [0, 1, 0, 1, 2, 0, 2, 1, 3, 0, 1, 0, 3]
[0, 1, 0, 2, 1, 0, 2, 3]         [0, 1, 0, 1, 2, 0, 2, 1, 3, 0, 1, 0, 3]
```



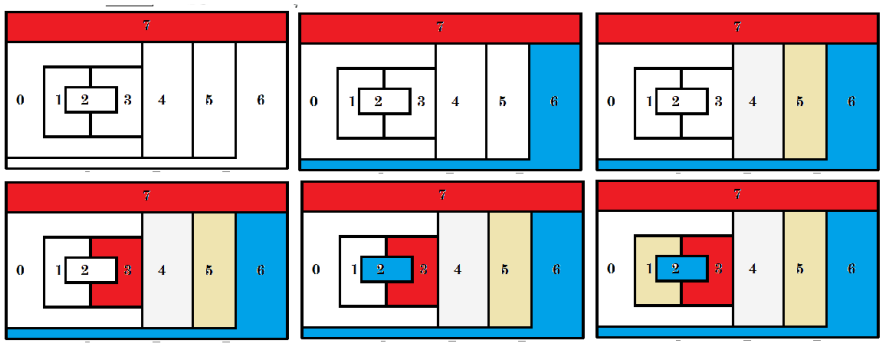
Q11. L'ordre de coloration imposé à la question précédente est arbitraire. On souhaite maintenant colorer le graphe en traitant les sommets selon un ordre arbitraire donné en argument. Écrivez une fonction `colorer2` analogue à `colorer1` et avec un argument supplémentaire, une liste `ordre` fixant l'ordre de coloration des pays.

```
def colorer2(ordre, LA) : #list of int, list of list of int -> list of int
....C = [-1]*len(LA)
....for i in ordre :
........colore_sommet(C, i, LA)
....return C
```

Q12. Donnez la liste des couleurs renvoyée par `colorer2` pour colorer le graphe `Gex` donné en exemple en prenant `ordre=[7,6,5,4,3,2,1,0]`. Combien de couleurs ont elles été utilisées ?

```
[-1, -1, -1, -1, -1, -1, -1, 0]
[-1, -1, -1, -1, -1, -1, 1, 0]
[-1, -1, -1, -1, -1, 2, 1, 0]
[-1, -1, -1, -1, 3, 2, 1, 0]
[-1, -1, -1, 0, 3, 2, 1, 0]
[-1, -1, 1, 0, 3, 2, 1, 0]
[-1, 2, 1, 0, 3, 2, 1, 0]
[4, 2, 1, 0, 3, 2, 1, 0]
[4, 2, 1, 0, 3, 2, 1, 0]
```

Cinq couleurs !
Mauvais plan !



Il faut une cinquième couleur pour la région 0

Même question avec `France` en prenant pour ordre les régions dans l'ordre alphabétique :
`ordre = [8, 6, 0, ..., 1, 7, ..., 10]`

```
[-1, -1, -1, -1, -1, -1, -1, -1, 0, -1, -1, -1, -1]
[-1, -1, -1, -1, -1, -1, 1, -1, 0, -1, -1, -1, -1]
[0, -1, -1, -1, -1, -1, 1, -1, 0, -1, -1, -1, -1]
[0, -1, -1, -1, -1, 2, 1, -1, 0, -1, -1, -1, -1]
[0, -1, -1, -1, -1, 2, 1, -1, 0, -1, -1, 0, -1]
[0, -1, -1, 0, -1, 2, 1, -1, 0, -1, -1, 0, -1]
[0, -1, 1, 0, -1, 2, 1, -1, 0, -1, -1, 0, -1]
[0, -1, 1, 0, -1, 2, 1, -1, 0, -1, -1, 0, 3]
[0, 4, 1, 0, -1, 2, 1, -1, 0, -1, -1, 0, 3]
[0, 4, 1, 0, -1, 2, 1, 1, 0, -1, -1, 0, 3]
[0, 4, 1, 0, -1, 2, 1, 1, 0, 2, -1, 0, 3]
[0, 4, 1, 0, 3, 2, 1, 1, 0, 2, -1, 0, 3]
[0, 4, 1, 0, 3, 2, 1, 1, 0, 2, 1, 0, 3]
[0, 4, 1, 0, 3, 2, 1, 1, 0, 2, 1, 0, 3]
```

Gex	4	1	2	3	4	5	6	7
	4	2	1	0	3	2	1	0

France	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	4	1	0	3	2	1	1	0	2	1	0	3

VARIANTE DE WELSH-POWELL

Q13. Écrivez une fonction `degre` avec pour argument la liste d'adjacence `LA` d'un graphe quelconque, qui renvoie la liste des degrés des sommets du graphe.

```
def degre(LA) : #list of list of int -> list of int
....return [len(liste) for liste in LA]
```

Q14. Écrivez une fonction `init` avec pour argument un entier `n`, qui renvoie une liste de listes `R` de taille `n`, telle que `R[i]` soit une liste vide. Par exemple, `init(3)` renverra `[[ ], [ ], [ ]]`.

```
def init(n) : #int -> list of list
....return [[ ] for i in range(n)]
```

On évitera `[ ]*n` qui crée juste une liste `[ ]` car on a beau faire, on n'ajoute rien.

On évitera `[[ ]]*n`.
Par exemple `L = [[ ]]*5` donne `[[ ], [ ], [ ], [ ], [ ]]` comme espéré.
Mais si j'exécute `L[2].append(4)`, je n'ai pas `[[ ], [ ], [4], [ ], [ ]]` comme j'ose l'espérer.
J'ai `[[4], [4], [4], [4], [4]]` car Python considère que toutes ces listes sont une seule et même liste vers laquelle pointent `L[0]` à `L[4]`.

Q15. Écrivez une fonction `ranger` avec pour argument une liste d'adjacence `LA`, qui renvoie une liste `R` de même taille que `LA`, telle que `R[i]` soit la liste des sommets de degré `i` [10]

```
def ranger(LA) : #list of list of int -> list of list of int
....R = [[ ]]*len(LA) #la liste de listes vides
....D = degre(LA) #la liste des degrés
....for i in range(len(D)) : #on la parcourt index par index
........R[d].append(i) #et on crée l'histogramme
....return R #la liste R est la bonne
```

Indiquez la réponse pour `ranger(France)`.

10. Ainsi, pour l'exemple de la question Q13, l'appel `ranger(LA)` renverra la liste `[[ ], [4], [0], [1,2,3], [ ]]`.

On rappelle :

France	Sommet	0	1	2	3	4	5	6	7	8	9	10	11	12
	Degré	2	5	3	3	4	6	4	4	5	3	2	0	5

On peut ensuite voir qu'il va y avoir des sommets de degré 0 à 12 (pas plus, un sommet ne peut pas avoir plus de voisins que n-1 (n étant le nombre total de sommets, et le -1 c'est parce qu'un sommet n'est pas son propre voisin).

France	Degré	0	1	2	3	4	5	6	7	8	9	10	11	12
	Liste	11		0, 10	2, 3, 9	4, 6, 7	1, 8, 12	5						

Vais je vérifier le bon nombre de listes vides au bout ?

```
[[11], [], [0, 10], [2, 3, 9], [4, 6, 7], [1, 8, 12], [5], [ ], [ ], [ ], [ ], [ ], [ ]]
```

La Corse est effectivement sans voisin, et la région Centre est celle qui a le plus de voisins.
On aura intérêt à commencer par elle.

Q16. Écrivez une fonction `renverse` avec pour argument une liste `L`, qui crée et renvoie une nouvelle liste obtenue en lisant `L` dans l'ordre inverse. [11]

```
def renverse(L) : #list -> list
....R = [ ]
....for i in range(len(LA)) :
........R.append(LA[i-1])
....return R
```

Certes, il existe une méthode qui s'applique aux listes, mais la question sous-entend qu'on ne peut pas l'utiliser.

Q17. Écrivez une fonction `trier_sommets` avec pour argument une liste d'adjacence `LA`, qui renvoie la liste des sommets triés dans l'ordre décroissant de leur degré.

On tient une liste de listes. Mais dans le mauvais ordre.
On va la lire à l'envers (ou la renverser).
Et on va coller bout à bout les listes obtenues.

```
def trier_sommets(LA) : #list of list of int -> list of int
....R = renverse(range(LA))
....T = [ ]
....for liste in R :
........T.extend(liste)
....return T
```

J'ai accepté d'imbriquer leurs procédures, mais on peut faire ça un peu plus directement.

Q18. Pour un graphe à `n` sommets, quelle est la complexité temporelle de la fonction `trier_sommets` dans le pire des cas ?

On parcourt la liste LA une fois, en en lisant les longueur des éléments : $O(n)$.
On parcourt la nouvelle liste en plaçant ensuite les éléments à un emplacement imposé dans une liste : $O(n)$ encore.
On parcourt la nouvelle liste à l'envers : encore $O(n)$.
On fait des collages de n listes (chacune étant d'une certains longueur).
Mais la somme des longueurs est n.

Q19. Écrivez la fonction `colorer3` avec pour argument une liste d'adjacence `LA`, qui crée et renvoie une liste de couleurs `C`, telle que `C[i]` soit la couleur à attribuer au sommet numéro `i`, les sommets étant colorés dans l'ordre décroissant de leur degré. Quelle est la complexité de `colorer3` dans le pire des cas pour un graphe à `n` sommets ?

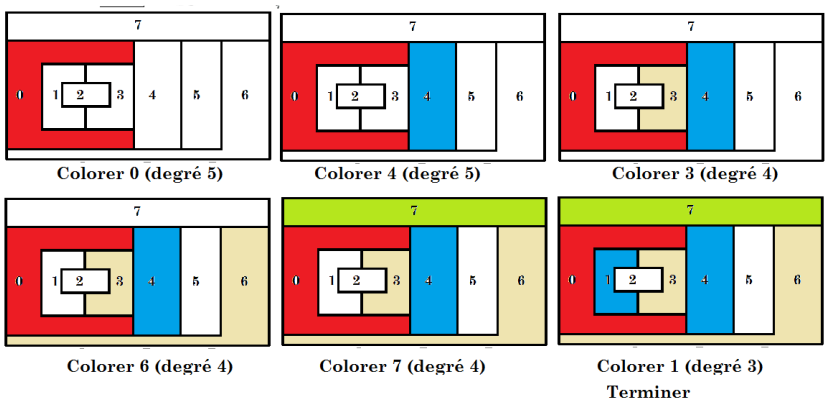
```
def colorer3(LA) : #list of list of int -> list of int
....return colorer2(trier_sommets(LA), LA)
```

11. Par exemple, `renverse([1,2,3,4])` renverra `[4,3,2,1]`.

Q20. Pour les graphes `Gex` et `France`, donnez la liste `C` des couleurs renvoyée par la fonction `colorer3`.

Pour `Gex`, la liste triée des sommets donne [0, 4, 3, 6, 7, 1, 5, 2].
Avec cet ordre de remplissage, ce choix des couleurs donne

```
[0, -1, -1, -1, -1, -1, -1, -1]
[0, -1, -1, -1, 1, -1, -1, -1]
[0, -1, -1, 2, 1, -1, -1, -1]
[0, -1, -1, 2, 1, -1, 2, -1]
[0, -1, -1, 2, 1, -1, 2, 3]
[0, 1, -1, 2, 1, -1, 2, 3]
[0, 1, -1, 2, 1, 0, 2, 3]
[0, 1, 0, 2, 1, 0, 2, 3]
[0, 1, 0, 2, 1, 0, 2, 3]
```



Gex	0	1	2	3	4	5	6	7
	0	1	0	2	1	0	2	3

France	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	1	0	1	2	0	3	3	1	0	2	0	2

Pour France, l'ordre des sommets à colorier est

[Centre, (6 voisins)

Normandie, (5 voisins)

Auvergne, Ile de France, Pays de Loire (4 voisins)

Bourgogne, Aquitaine, Hauts de France, Occitanie (3 voisins)

Bretagne, Rhône-Alpes (2 voisins),

Corse (pas de voisin)

France	0	1	2	3	4	5	6	7	8	9	10	11	12
						0							
		1				0							
		1				0			1				
		1				0			1				2
		1			2	0			1				2
		1			2	0	3		1				2
		1			2	0	3	3	1				2
		1	0		2	0	3	3	1				2
		1	0	1	2	0	3	3	1				2
		1	0	1	2	0	3	3	1	0			2
	et ainsi de suite												
	0	1	0	1	2	0	3	3	1	0	2	0	2

La priorité de coloration est 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.

ALGORITHME DSATUR

Q21. Écrivez une fonction `degre_satur` avec 3 arguments, une liste d'adjacence `LA`, un sommet `s` du graphe, une liste `C` de couleurs. Cette fonction renvoie le degré de saturation du sommet `s`. On rappelle que le sommet `i` est coloré si et seulement si `C[i]` est différent de `-1`.

```
def degre_satur(LA, s, C) : #list of list of int, int, list of int -> int
....coul_vois = [ ]
....for v in LA[s] : #on prend ses voisins
........cou = C[v] #couleur du voisin
........if cou != -1 and not(cou in coul_vois) : #est il coloré avec une couleur inédite
............cou_vois.append(coul) #nouvelle couleur
....return len(coul_vois)#combien de couleurs
```

Et en une seule ligne ?

Q22. Écrivez une fonction `liste_satur` avec deux arguments, une liste d'adjacence `LA`, la liste `C` des couleurs des sommets, qui renvoie la liste des sommets non colorés du graphe ayant un degré de saturation maximum 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.

```
def liste_satur(LA, C) : #list of int, list of int -> list of int
....record = 0 #on commence faible
....Lrecord = [ ] #personne n'a le record
....for s in range(len(LA)) : #on les prend un par un
........if C[s] == -1 : #sommet non coloré
............deg = degre_satur(LA, s, C) #on lit son degré
............if deg = record : #égalité avec les gagnants provisoires
................Lrecord.append(s) #il rejoint les gagnants
............if deg > record :
................recorc = deg #nouveau record
................Lrecord = [s] #nouveau vainqueur tout seul pour l'instant
....return Lrecord
```

Grand classique de recherche de maximum et de qui atteint ce maximum.
Les moins informaticiens le feront en deux temps.

```
def liste_satur(LA, C) : #list of int, list of int -> list of int
....Ldeg = [ ] #liste des degrés
....for s in range(len(LA)) : #on les prend un par un
........if C[s] != -1 : #si sommet non encore coloré
............Ldeg.append(degre_satur(LA, s, C))
....record = max(Ldeg) #on a droit à max ?
....Lrec = [ ] #allez, on va accueillir les vainqueurs
....for s in range(len(LA)) : #on les (re)prend un par un
........if C[s] == -1 and degre_satur(LA„ s, C) == record : #sommet non coloré
............Lrec.append(s)
....return Lrecord
```

Q23. Écrire une fonction `pas_fini` avec pour argument une liste `C`, qui renvoie `True` si cette liste contient la valeur `-1`, `False` sinon.

```
def pas_fini(C) : #list of int -> boolean
....return -1 in C
```

Des fois, c'est bien de lire le sujet pour voir où gagner un point facile.

Q24. Compléter la fonction `colorer4` ayant pour argument une liste d'adjacence `LA`, qui renvoie une liste `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 parmi ceux-ci qui soit de degré maximal ;
– colorer le sommet choisi en lui attribuant la couleur disponible ayant la plus petite valeur.

```
def colorer4(LA) : #spécifications
....n = len(LA)
....#nombre de sommets du graphe
....D = [degre(s) for s in range(n)]
....#liste des degrés des sommets du graphe
....C = [-1 for s in range(n)]
....#initialisation de la liste des couleurs
....while pas_fini(C) :
........#liste des sommets non colorés de degré de saturation maximal
........Ls = liste_satur(LA, C)
........#en cas d'égalité, recherche d'un sommet de degré maximal
........dmax = max([D[s] for s in Ls]) #
........for s in Ls :
............if D[s] == dmax
................winner = s #le dernier sera celui qu'on garde
........#coloration du sommet prioritaire
........colore_sommet(C, winner, LA)
....return C
```

Détail stupide : mon programme a tourner sans s'arrêter quand je l'ai testé sur le graphe France.
Pourquoi ? A cause de la Corse.
En effet, comme elle n'a pas de voisin, et comme j'avais une autre recherche de celui que j'ai appelé winner, elle n'était jamais colorée.
Et c'est ainsi que j'ai perdu dix minutes.

UN MINORANT DU NOMBRE DE COULEURS NÉCESSAIRES

Q25. Justifiez que le nombre minimum de couleurs pour colorer un graphe est supérieur ou égal au nombre `nc`.

Dans une clique de taille p, chaque sommet doit avoir une couleur différente des autres.
Il faut donc p couleurs différentes.

Si on veut être rigoureux : si il y a moins de p couleurs, par principe du pigeonnier, il y a au moins deux sommets de la même couleur. Et ceci contredit la notion de coloriage.

Montrez également que : $nc \leqslant 1 + max\{deg(s),\ s \in G\}$ où $deg(s)$ désigne le degré du sommet s dans le graphe G.

Considérons un sommet s_0 de la plus grande clique (de cardinal nc).
Combien a-t-il de voisins ? Au moins ceux de la clique : il y en a $nc - 1$ (on ne le compte pas lui même).
On a donc $nc - 1 \leqslant \deg(s_0)$.
Mais par définition même du maximum : $nc - 1 \leqslant \deg(s_0) \leqslant max\{deg(s),\ s \in G\}$.

Q26. Écrivez une fonction `est_clique` avec deux arguments, la liste d'adjacence `LA` d'un graphe et un tuple `K` de sommets de ce graphe, qui renvoie `True` si `K` est une clique, `False` sinon.

```
def est_clique(LA, K) : #list of list of int, list -> boolean
....for s0 in K :
........for s1 in K :
............if not(est_voisin(s0, s1, LA)) :
................return False
....return True
```

L'idée semble bonne et classique. Si il existe deux points de la pseudo clique qui ne sont pas voisins, on sort.
Et si tout s'est bien passé, on répond favorablement.
Mais il y a un problème dans le cas $s_0 = s_1$. Un sommet n'est pas voisin de lui même dans le graphe.

Et à quoi bon tester s_1 et s_0 si on a déjà testé s_0 et s_1 ?

On va donc parcourir le t-uple par index avec $i < k$.

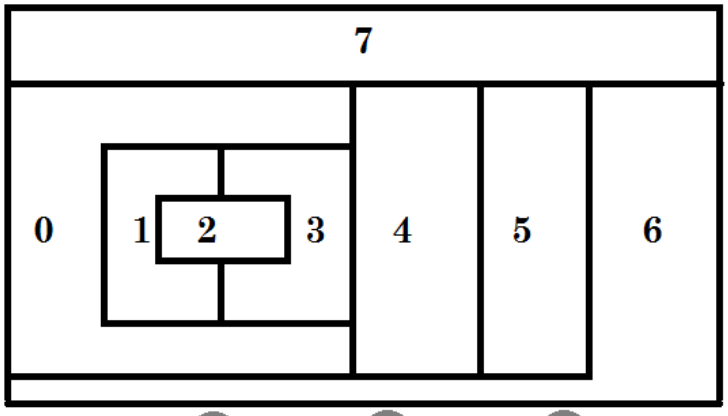
```
def est_clique(LA, K) : #list of list of int, list -> boolean
....for i in range(len(K)) :
........for j in range(i) :
............if not(est_voisin(K[i], K[j], LA)) :
................return False
....return True
```

Q27. La fonction `combinations` du module `itertools`, présentée à l'annexe 2, fournit toutes les combinaisons de taille fixée d'une liste. Compléter la fonction `minoration_nb_couleurs` ayant pour argument la liste d'adjacence `LA` d'un graphe, qui renvoie le cardinal de la plus grande clique du graphe considéré.

```
from itertools import combinations
def minoration_nb_couleurs (LA) :
....n = len(LA)
....# nombre de sommets du graphe
....S = [k for k in range(n)]
....# liste des sommets du graphe
....i = n
....test = True #indique qu'il n'y a pas de clique si grande que ça
....while test : #tant qu'on n'a pas détecté de clique
........for K in combinations (S, i) : #on teste pour toutes les combinaisons de taille i
............if est_clique(LA, K) : #a-t-on une clique
................test = False #on a détecté une clique on peut arrêter
........i = i-1 #on va tester les cliques plus petites
....return i+1 #car i a été décrémenté de une unité
```

Ainsi, si le nombre de couleurs utilisées pour colorer le graphe est égal à `nc`, on est sûr d'avoir utilisé le nombre minimum de couleurs.

Figure 1



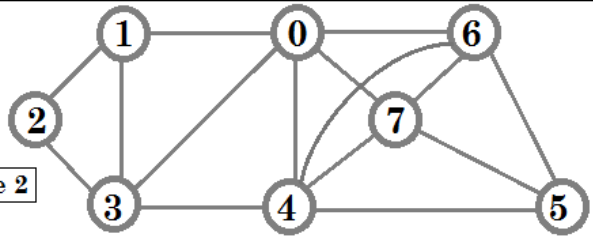
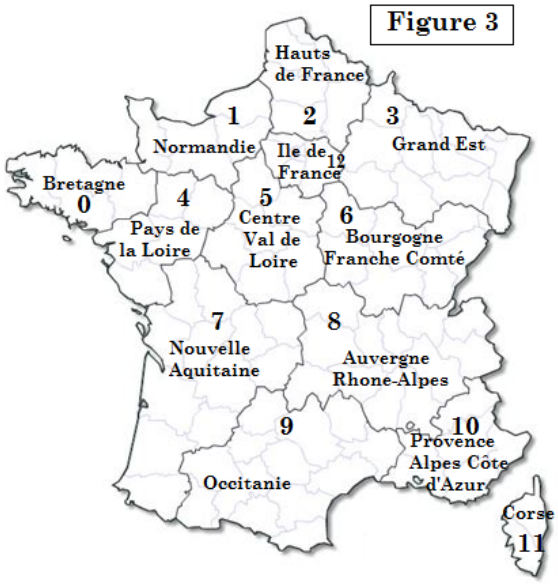


Figure 2

Figure 3



Un convertisseur qui prend une matrice d'adjacence et crée la liste d'adjacence

```
def convertisseur1(LA) : #list of list of int -> list of list of boolean
....n = len(LA)#pour être tranquille on le lit une fois
....return [[j in LA[i] for j in range(n)] for i in range(n)]

Et si vous voulez le bâtir par étapes
def convertisseur1(LA) : #list of list of int -> list of list of boolean
....n = len(LA)#il va servir plusieurs fois
....M = [ ] #création de la matrice
....for i in range(n) : #ligne par ligne
........L = [ ] #création d'une ligne vide
........for k in range(n) : #remplissage de la ligne
............L.append(k in LA[i])
........M.append(L) #et surtout pas extend
....return M
```

un convertisseur qui prend en entrée une liste d'adjacences et crée la matrice d'adjacence

une fonction combinaison qui prend en entrée une liste et un entier k (plus petit que la longueur n de la liste) et retourne les $\binom{n}{k}$ combinaisons sous forme de liste de listes.
On va faire appel à un programme récursif.
Si `k` est nul, on retourne une liste vide.
Ce sera donc `[[ ]]` (et pas `[ ]`).

Si k est plus grand que n (n est la longueur de la liste), on ne retourne rien.

Sinon, avec un ensemble à `n` éléments de L[0] à L[-1],
on prend les parties à k éléments sans L[-1] : il y en a $\binom{n-1}{k}$
et les parties à k éléments avec `L[-1]`.il y en a $\binom{n-1}{k-1}$

Pour le premier modèle, on va chercher la liste de listes `combi(L[:-1], k)`.
Pour le second modèle, on prend les `combi(L[: -1], k-1)` et on ajoute `L[-1]` à chacune.

Pour les parties

```
def combi(L, k) : #list, int -> list of list
....if k == 0 :
........return [[ ]]
....if k > len(L) :
........return [ ]
....Ck = combi(L[:-1],k) #les parties à k éléments de la liste raccourcie
....Ckm = combi(L[:-1],k-1) #les parties à k-1 éléments de la liste raccourcie
....Ckm = [liste+[L[-1]] for liste in Ckm] #on ajoute L[-1] à ces parties à k-1 éléments
(donc k éléments)
....return Ck+Ckm #reconnaissez vous (n k) + (n-1 k-1) ?
```

On peut évidemment raccourcir un peu le code,