

Informatique Tronc Commun
Devoir surveillé n°4
Epreuve du Concours blanc

Lycée Fabert de Metz, MPSI – PCSI (2025 – 2026)

Durée de l'épreuve : 2h00

Consignes :

- La correction tient compte du soin apporté à la **présentation** et à la **rédaction** du document-réponse. Les bouts de code devront être écrits en **langage Python** en respectant la syntaxe inhérente à ce langage. On veillera notamment à faire apparaître clairement les **indentations** qui structurent le code.
- Lorsque l'énoncé impose des contraintes sur le type et la valeur des arguments d'une fonction, on supposera que ces contraintes sont effectivement vérifiées lors de l'appel de la fonction. Il n'est donc pas nécessaire d'ajouter des tests dans la fonction pour vérifier ces contraintes.
Exemple : si l'énoncé demande de définir une fonction $f(x)$ avec x un entier positif, il n'est pas nécessaire dans la définition de la fonction f de contrôler le type et le signe de x .
- L'usage des **calculatrices** et de **tout autre document** n'est **pas autorisé**.

Les questions abordées dans ce problème ne sont pas toutes indépendantes : pour y répondre, on pourra faire appel aux fonctions des questions précédentes, même si celles-ci n'ont pas été traitées.

Principales opérations sur les dictionnaires	
Création d'un dictionnaire ➤ Création d'un dictionnaire vide : <code>dico = {}</code> ➤ Par donnée explicite des éléments : <pre>dico = {"k1": 0, "k2": 0, "k3": 0}</pre> In : dico Out: {'k1': 0, 'k2': 0, 'k3': 0} ➤ Par compréhension : <pre>L_k = ["k1", "k2", "k3"]</pre> <pre>dico = {k: 0 for k in L_k}</pre> In : dico Out: {'k1': 0, 'k2': 0, 'k3': 0}	
Test de présence d'une clé <code>key</code> dans un dictionnaire <code>dico</code> : <pre>In : "k2" in dico</pre> Out: True <pre>In : "k4" in dico</pre> Out: False	<code>dico[key]</code> renvoie la valeur associée à une clé valide <code>key</code> du dictionnaire : <pre>In : dico["k2"]</pre> Out: 0 <pre>In : dico["k4"]</pre> KeyError: 'k4'
Instruction <code>dico[key] = val</code> : ➤ si <code>key</code> est déjà une clé : remplace par <code>val</code> la valeur associée à cette clé ➤ si <code>key</code> n'est pas une clé : ajoute le couple <code>key:val</code> au dictionnaire <code>dico</code> <pre>In : dico["k1"] = 1</pre> <pre>In : dico["k4"] = 2</pre> In : dico Out: {'k1': 1, 'k2': 0, 'k3': 0, 'k4': 2}	Taille d'un dictionnaire : <code>len(dico)</code> Itération sur les clés du dictionnaire : <pre>for cle in dico.keys() :</pre> <pre> print(cle, end = " ; ")</pre> Out: k1 ; k2 ; k3 ; k4 ;



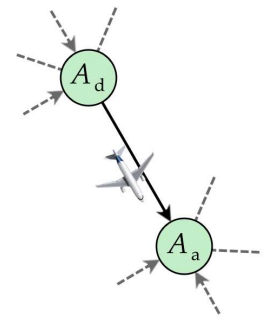
Graphe des liaisons aériennes

A. Implémentation du graphe

On suppose que l'on dispose d'une liste Python `Airports` répertoriant $n \approx 4000$ aéroports commerciaux, chaque aéroport étant désigné par une chaîne de trois caractères (le code 'CDG' désigne par exemple l'aéroport de « Paris – Roissy Charles de Gaulle »). On affiche par exemple ci-dessous les dix premiers éléments de cette liste :

```
In : Airports[0:10]
Out: ['CDG', 'LGW', 'DEL', 'DOH', 'DSM', 'EWR', 'EYW', 'HND', 'ICN', 'JFK']
```

Les liaisons aériennes entre ces aéroports sont toutes contenues dans une liste Python `Routes` dont les p éléments sont des tuples (A_d, A_a) où A_d et A_a sont les codes de deux aéroports répertoriés dans la liste `Airports` : chaque tuple (A_d, A_a) désigne un arc orienté $A_d \rightarrow A_a$ traduisant l'existence d'une liaison aérienne avec décollage de l'aéroport A_d et atterrissage à l'aéroport A_a (sans présumer de l'existence d'une liaison aérienne dans le sens inverse). On affiche par exemple ci-dessous les cinq premiers éléments de cette liste :



```
In : Routes[0:5]
Out: [('DSM', 'ORD'), ('ORD', 'BGI'), ('BGI', 'LGA'), ('SIN', 'CDG'), ('CDG', 'SIN')]
```

Analyse de la liste `Routes` :

A.1. Ecrire une fonction `degre_In(Routes, A)` :

- qui prend en arguments la liste `Routes` et une chaîne de caractères `A` correspondant au code d'un aéroport
- qui renvoie le degré entrant du sommet `A` (nombre de prédécesseurs de l'aéroport de code `A`).

Exemple d'application :

```
In : degre_In(Routes, 'CDG')
Out: 311
```

Préciser la complexité de cette fonction `degre_In` (à l'aide de la notation $\mathcal{O}$), en justifiant succinctement.

A.2. Ecrire une séquence d'instructions permettant d'afficher dans la console le code de l'aéroport le plus « accueillant », c'est-à-dire celui possédant le degré entrant le plus élevé (on admettra qu'il y a unicité de ce maximum). Préciser la complexité de cette séquence d'instructions (à l'aide de la notation $\mathcal{O}$), en justifiant succinctement.

Construction du graphe des liaisons aériennes :

L'association des données précédentes forme un graphe orienté G de n sommets (les aéroports) et p arcs (les liaisons aériennes). On choisit d'implémenter ce graphe sous la forme d'un dictionnaire `G` :

- chaque clé désigne le code `A` d'un aéroport (chaîne de 3 caractères)
- la valeur `G[A]` associée à un aéroport `A` est la liste de ses « successeurs », c'est-à-dire la liste des aéroports directement accessibles en décollant depuis l'aéroport `A`.

Par exemple, pour l'aéroport de Metz-Nancy-Lorraine (code 'ETZ'), on obtient :

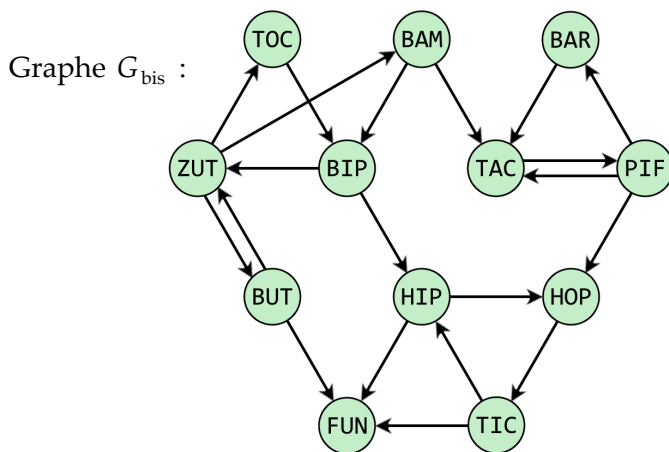
```
In : G['ETZ']  
Out: ['ALG', 'CMN', 'CZL', 'LYS', 'MRS', 'TLS']
```

A.3. Ecrire une séquence d'instructions permettant de créer le dictionnaire G à partir des listes Python Airports et Routes.

B. Parcours en largeur

Pour un aéroport A_{start} donné, on souhaite identifier tous les aéroports qui sont atteignables en décollant de A_{start} , quel que soit le nombre de liaisons empruntées.

Pour simplifier, raisonnons sur le graphe G_{bis} ci-dessous, un graphe fictif de taille fortement réduite par rapport au graphe G des liaisons aériennes, mais implémenté sous la forme d'un dictionnaire G_{bis} possédant les mêmes caractéristiques que le dictionnaire G :



```
G_bis = {'TOC': ['BIP'],  
         'BAM': ['BIP', 'TAC'],  
         'BAR': ['TAC'],  
         'ZUT': ['BAM', 'BUT', 'TOC'],  
         'BIP': ['HIP', 'ZUT'],  
         'TAC': ['PIF'],  
         'PIF': ['BAR', 'HOP', 'TAC'],  
         'BUT': ['FUN', 'ZUT'],  
         'HIP': ['FUN', 'HOP'],  
         'HOP': ['TIC'],  
         'FUN': [],  
         'TIC': ['FUN', 'HIP']}
```

Pour ce graphe G_{bis} , l'aéroport 'HIP' est par exemple accessible en décollant de $A_{\text{start}} = \text{'PIF'}$, en empruntant trois liaisons aériennes : $\text{'PIF'} \rightarrow \text{'HOP'} \rightarrow \text{'TIC'} \rightarrow \text{'HIP'}$.

Pour identifier de façon systématique ce type de chemin, on décide de mettre en œuvre un parcours en largeur du graphe (parcours « BFS » pour *Breadth First Search*). Pour stocker les sommets découverts en attente de traitement, on utilise pour cela une structure de donnée de type « file ».

B.1. Quelle stratégie est exploitée lors d'un tel parcours en largeur ?

Stratégie « LIFO » (*Last In, First Out*) : l'élément extrait de la file est systématiquement le dernier à avoir été ajouté ?

ou

Stratégie « FIFO » (*First In, First Out*) : l'élément extrait de la file est systématiquement le plus ancien à avoir été ajouté ?

B.2. A la suite d'un parcours en largeur partant d'un sommet A_{start} , on parvient à identifier un chemin $A_{\text{start}} \rightarrow \dots \rightarrow A$ entre le sommet de départ A_{start} et n'importe quel autre sommet accessible A . Quelle est la propriété particulière de chacun de ces chemins ?

On considère le sommet 'PIF' comme sommet de départ du parcours en largeur du graphe G_{bis} :

- au début de l'algorithme de parcours en largeur, ce sommet est le seul présent dans la file d'attente
- à la première itération :
 - le sommet 'PIF' est retiré de la file d'attente
 - puis on y place chacun de ses successeurs 'BAR', 'HOP' et 'TAC'.



Premières étapes d'évolution de la file d'attente

B.3. Poursuivre l'algorithme de ce parcours en largeur en représentant l'évolution de la file d'attente jusqu'à la fin du parcours partant du sommet 'PIF'. On impose pour cela :

- que les listes d'adjacence soient systématiquement parcourues par ordre alphabétique (*cette contrainte n'est effective que sur cette question, il ne sera pas nécessaire de la traduire lors des implémentations suivantes en langage Python*)
- que seuls des sommets non encore découverts peuvent être introduits dans la file d'attente.

Tous les sommets du graphe G_{bis} sont-ils accessibles en partant du sommet 'PIF' ?

B.4. Ecrire une fonction `BFS(G, A_start)` :

- qui prend en arguments le dictionnaire `G` (associé au graphe des liaisons aériennes) et une chaîne de caractères `A_start` correspondant à une clé de ce dictionnaire (supposée valide)
- qui renvoie un dictionnaire `Pere` :
 - ce dictionnaire vaut `{A_start:None}` au début du parcours en largeur
 - à la fin de l'algorithme BFS, les clés de ce dictionnaire (autres que `A_start`) sont tous les sommets `A` accessibles depuis `A_start` et chaque valeur `Pere[A]` correspond au « parent » du sommet `A` dans l'arborescence du parcours (c'est-à-dire le sommet qui a permis de découvrir `A`).
- Pour stocker les sommets en attente, on utilisera une structure de données `File_attente` de type « file », dont on rappelle ci-dessous les principaux outils :

Application au graphe G_{bis} :

```
In : Pere = BFS(G_bis, 'PIF')
In : Pere['HOP']
Out: 'PIF'
In : Pere['ZUT']
KeyError: 'ZUT'
```

Structure de données deque de type « file »	Observation dans la console
<pre># Importation d'un module dédié from collections import deque # Création d'une file vide F = deque() # Ajout d'éléments dans la file F F.append('AAA') F.append('BBB') F.append('CCC')</pre>	<pre>In : F Out: deque(['AAA', 'BBB', 'CCC'])</pre> <u>Extraction et renvoi de l'élément prioritaire</u> <pre>In : S = F.popleft() In : S Out: 'AAA' In : F Out: deque(['BBB', 'CCC'])</pre> <u>Taille de la file</u> <pre>In : len(F) Out: 2</pre>

B.5. On s'intéresse à la terminaison de la fonction BFS précédente. Pour prouver la terminaison d'une boucle, préciser le type de propriété que l'on cherche à identifier :

- ☐ un **invariant** de boucle ☐ un **variant** de boucle ☐ un **covariant** de boucle.

En justifiant votre réponse, indiquer si la valeur de `len(File_attente)` peut être utilisé directement pour prouver la terminaison de l'algorithme. Même question pour `len(Pere)`.

B.6. Ecrire une fonction `distance(G, A_start, A_end)` :

- qui prend en arguments le dictionnaire `G` et deux chaînes de caractères distinctes `A_start` et `A_end` (clés valides du dictionnaire `G`)
- qui commence par créer le dictionnaire `Pere` associé au parcours en largeur du graphe `G` en partant du sommet `A_start`
- qui renvoie :
 - `False` si l'aéroport `A_end` n'est pas accessible par liaisons aériennes en partant de `A_start`
 - sinon, le nombre de liaisons aériennes (c'est-à-dire le nombre d'arcs) du chemin identifié menant de `A_start` à `A_end`.

Application au graphe G_{bis} :

```
In : distance(G_bis, 'PIF', 'HIP')
Out: 3
In : distance(G_bis, 'PIF', 'ZUT')
Out: False
```

C. Identification des composantes fortement connexes

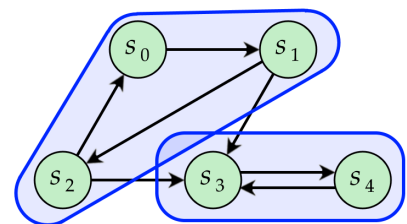
1. Composantes fortement connexe et algorithme de KOSARAJU

Définition d'une composante fortement connexe :

En théorie des graphes, une **composante fortement connexe** d'un graphe orienté est un sous-graphe possédant la propriété suivante et qui est *maximal* * pour cette propriété : pour tout couple de sommets (A, B) de ce sous-graphe, B est accessible depuis A et A est accessible depuis B .

* c'est-à-dire qu'on ne peut pas prendre un sommet de plus tout en conservant cette propriété

On illustre cette définition sur le graphe ci-contre où les composantes fortement connexes ont été identifiées. Les sommets s_0, s_1, s_2 sont mutuellement accessibles, donc ils font partie de la même composante fortement connexe $\{s_0, s_1, s_2\}$. Le sommet s_3 est accessible en partant de s_0 , mais il n'existe pas de chemin inverse : les sommets s_3 et s_0 ne font donc pas partie de la même composante fortement connexe.



Identification des composantes fortement connexes d'un graphe orienté

L'algorithme de KOSARAJU est un algorithme d'identification de toutes les composantes fortement connexes d'un graphe orienté G . Cet algorithme s'effectue en deux étapes.

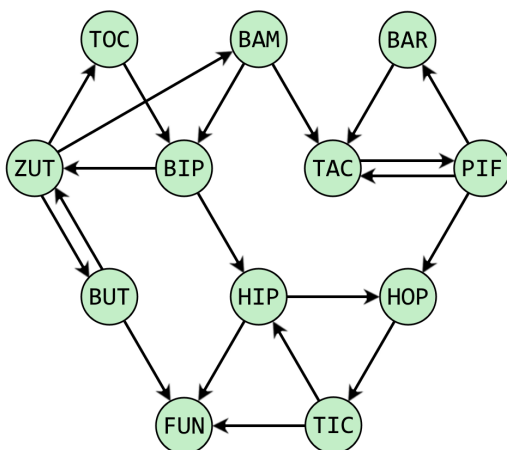
Etape n°1 ► On exécute un parcours en profondeur sur l'ensemble du graphe G (on poursuit le parcours tant que tous les sommets n'ont pas été traités), en identifiant l'ordre de traitement complet de tous les sommets. On mémorise alors cet ordre après l'avoir inversé.

Etape n°2 ► On exécute ensuite un parcours en profondeur sur le graphe transposé G^T en suivant l'ordre donné par la première étape. Les arbres produits par ce deuxième parcours sont alors les composantes fortement connexes recherchées.

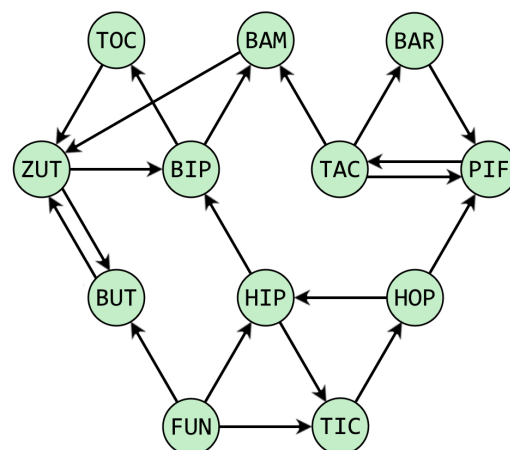
Définition d'un graphe transposé :

Le graphe transposé G^T d'un graphe orienté G est le graphe possédant les mêmes sommets que G mais dont tous les arcs ont été inversés.

Cet algorithme de KOSARAJU (dont on admettra la correction) s'appuie notamment sur le fait que les composantes fortement connexes d'un graphe sont les mêmes que celles de son graphe transposé. On illustre ci-dessous l'application de cet algorithme au graphe G_{bis} .



Graphe orienté G_{bis}



Graphe transposé G_{bis}^T

En appliquant l'étape n°1 au graphe G_{bis} , les sommets sont traités complètement dans l'ordre suivant :

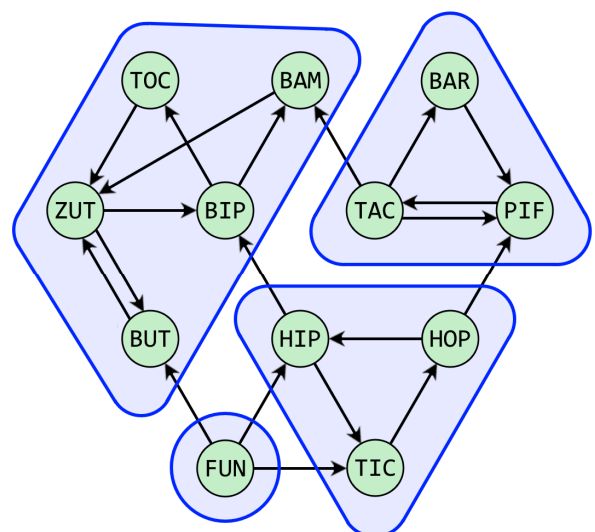
'FUN' ▷ 'TIC' ▷ 'HOP' ▷ 'HIP' ▷ 'BAR' ▷ 'PIF' ▷ 'TAC' ▷ 'BAM' ▷ 'BUT' ▷ 'ZUT' ▷ 'BIP' ▷ 'TOC'

On mémorise alors l'ordre inverse :

'TOC' ▷ 'BIP' ▷ 'ZUT' ▷ 'BUT' ▷ 'BAM' ▷ 'TAC' ▷ 'PIF' ▷ 'BAR' ▷ 'HIP' ▷ 'HOP' ▷ 'TIC' ▷ 'FUN'

On construit alors le graphe transposé G_{bis}^T afin de mettre en œuvre l'étape n°2.

- En démarrant le parcours en profondeur du sommet 'TOC', on identifie la composante fortement connexe { 'TOC', 'BIP', 'ZUT', 'BUT', 'BAM' }.
- En partant du sommet 'TAC', on identifie ensuite la composante fortement connexe { 'TAC', 'PIF', 'BAR' }.
- En partant du sommet 'HIP', on identifie ensuite la composante fortement connexe { 'HIP', 'HOP', 'TIC' }.
- Le dernier sommet restant 'FUN' est finalement seul dans sa composante fortement connexe { 'FUN' }.



Mise en évidence des composantes fortement connexes

On identifie finalement bien les quatre composantes fortement connexes des graphes G_{bis} et G_{bis}^T .

2. Fonctions utiles

Avant d'implémenter l'algorithme de KOSARAJU, commençons par écrire quelques fonctions utiles.

Manipulation de listes :

C.1. Ecrire une fonction `inverse(L)` :

- qui prend en argument une liste Python `L` de chaînes de caractères
- qui renvoie une nouvelle liste dont les éléments sont énumérés dans l'ordre inverse (la liste `L` placée en argument ne doit pas être modifiée).

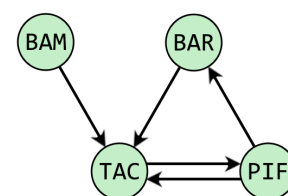
Exemple d'application :

```
In : L = ['a', 'b', 'c', 'd', 'e']
In : inverse(L)
Out: ['e', 'd', 'c', 'b', 'a']
In : L
Out: ['a', 'b', 'c', 'd', 'e']
```

L'utilisation de méthodes natives du type `reverse` n'est pas autorisée pour traiter cette question.

Graphe transposé :

C.2. Représenter la matrice d'adjacence $\mathcal{M}(H)$ du graphe orienté H ci-contre, en considérant ses sommets dans l'ordre 'BAM', 'BAR', 'TAC' puis 'PIF'. Représenter ensuite la matrice d'adjacence $\mathcal{M}(H^T)$ du graphe transposé H^T . Pourquoi parle-t-on ainsi de graphe « transposé » ?



Graphe orienté H

C.3. Pourquoi a-t-on choisi précédemment d'implémenter le graphe G des liaisons aériennes par listes d'adjacence et non pas par matrice d'adjacence ?

C.4. Ecrire une fonction `transpose(G)` :

- qui prend en argument le dictionnaire G associé au graphe des liaisons aériennes G
- qui renvoie un nouveau dictionnaire associé au graphe transposé G^T (sans modification de G).

3. Parcours en profondeur du graphe (étape n°1)

On commence par mettre en œuvre un parcours en profondeur du graphe G (parcours « DFS » pour *Depth First Search*), en implémentant cet algorithme de parcours sous forme récursive.

C.5. Ecrire une fonction récursive `explorer(G, A, DeJaVu, Visite)` :

- qui prend en arguments :
 - le dictionnaire G (implémentation du graphe orienté)
 - une chaîne de caractère A correspond à une clé de ce dictionnaire G (clé supposée valide)
 - un dictionnaire `DeJaVu` possédant comme clés tous les sommets du graphe G , chaque valeur étant un booléen `True` ou `False` indiquant si le sommet a déjà été découvert ou non
 - une liste `Visite`
- qui ne renvoie rien, mais engendre un parcours en profondeur du graphe G en partant du sommet A , en effectuant les opérations suivantes :
 - actualisation du dictionnaire `DeJaVu` pour le sommet A
 - pour tous les successeurs du sommet A qui n'ont pas encore été découverts, appel récursif de la fonction `explorer`
 - ajout du sommet A dans la liste `Visite` après avoir effectué tous les appels récursifs précédents.

C.6. Ecrire ensuite la fonction principale DFS(G) :

- qui prend en argument le dictionnaire G
- qui renvoie une liste correspondant à l'ordre inverse des éléments contenus dans la liste Visite (construite à l'issue du parcours en profondeur complet du graphe G).

Application au graphe G_{bis} :

```
In : DFS(G_bis)
Out: ['TOC', 'BIP', 'ZUT', 'BUT',
      'BAM', 'TAC', 'PIF', 'BAR',
      'HIP', 'HOP', 'TIC', 'FUN']
```

Cette fonction devra ainsi effectuer les opérations suivantes :

- initialisation du dictionnaire DeJaVu et de la liste Visite (aucun sommet n'a encore été découvert ou visité au début de l'algorithme)
- pour tous les sommets A du graphe G, mise en œuvre d'un parcours en profondeur du graphe en partant du sommet A si celui-ci n'a pas encore été découvert (appel de la fonction explorer)
- renvoi final de la liste attendue.

4. Algorithme de KOSARAJU (étapes n°1 et n°2)

On souhaite finalement écrire une fonction Kosaraju(G) :

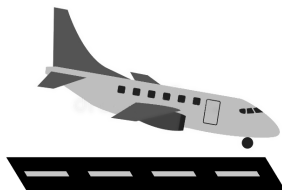
- qui prend en argument un dictionnaire G modélisant un graphe orienté G
- qui renvoie un dictionnaire CFC :
 - dont les clés sont les mêmes que celles du dictionnaire G
 - la valeur CFC[A] associée à un sommet A est un entier naturel correspondant au numéro de la composante fortement connexe dont fait partie ce sommet A.

Cette fonction Kosaraju(G) est écrite partiellement dans le document-réponse. La fonction explorer_K, que l'on supposera déjà implémentée, est très similaire à la fonction explorer définie précédemment, se chargeant de mettre en œuvre un parcours en profondeur du graphe transposé G^T , mais en remplissant également de façon adéquate le dictionnaire CFC.

C.7. Compléter le code du document-réponse (zones grisées) :

- afin que le dictionnaire G_t associé au graphe transposé G^T soit effectivement défini
- en faisant en sorte que les sommets A de ce graphe transposé soient énumérés dans l'ordre défini à l'issue de l'étape n°1.

C.8. Préciser la complexité de cette fonction Kosaraju, exprimée en fonction des paramètres n et p , en justifiant suffisamment votre réponse.



Fin de l'épreuve