DS 4

le 26/03/2026 – Durée : 2h

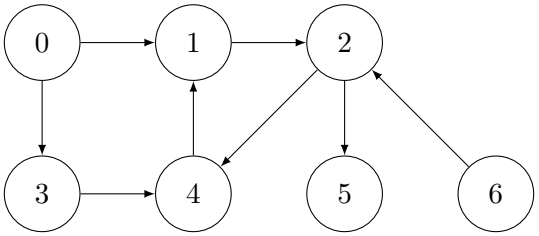
- L'usage de la calculatrice n'est pas autorisé.
- La clarté et la précision des raisonnements interviendront pour une grande part dans la notation. La présentation (en particulier les indentations) est également essentielle.
- Le résultat d'une question peut être admis afin de traiter une question suivante ; une fonction peut être utilisée dans la suite d'un exercice même si elle n'a pas été écrite.
- Le barème tiendra compte de la longueur du devoir.

Exercice 1

Hormis les questions 3 et 4, les questions de cet exercice sont indépendantes.

Dans cet exercice (informatique du tronc commun), les graphes ont leurs sommets numérotés à partir de 0 et ils sont orientés. On les représente par un dictionnaire d'adjacence.

Par exemple, le graphe:



est représenté par le dictionnaire:

```
d = { 0: [1, 3], 1: [2], 2: [4, 5], 3: [4], 4: [1], 5: [], 6: [2]}
```

1. Écrire en langage `Python` une fonction `degreMax(d: dict) -> int` qui reçoit en entrée un dictionnaire d'adjacence représentant un graphe orienté et renvoie le degré sortant maximal parmi tous les degrés sortants des sommets du graphe.

   Si `G` est un graphe orienté, on appelle graphe inverse de `G` le graphe possédant les mêmes sommets ainsi que les mêmes arêtes mais en sens inverse par rapport à celles de `G`.

2. Représenter le graphe inverse du graphe orienté donné en introduction.

   Écrire en langage `Python` une fonction `grapheInv(d: dict) -> dict` qui renvoie un dictionnaire d'adjacence du graphe inverse du graphe représenté par `d`.

   On souhaite colorier notre graphe orienté. Les couleurs sont représentées par des entiers naturels. La coloration du graphe est modélisée par une liste `L` telle que `L[s]` est égale à la couleur attribuée au sommet `s`.

   Deux sommets du graphe reliés par une arête ne doivent pas être de la même couleur (coloration du graphe valide).

3. Écrire en langage `Python` une fonction `colorationValide(d: dict, L: list) -> bool` qui renvoie `True` si la coloration `L` du graphe représenté par `d` est valide et `False` dans le cas contraire.

4. Donner la complexité dans le pire des cas de la fonction précédente en fonction du nombre N de sommets et du nombre M d'arêtes. Justifier votre réponse.

On considère deux tables: `FILMS` et `LOCATIONS`. La première contient des informations sur des films et la seconde des informations sur des location de films par les clients.

La table `FILMS` contient les attributs suivants:

- `codefilm`: code d'un film (entier), clé primaire;
- `nomfilm` (chaîne de caractères).

La table `LOCATIONS` contient les attributs suivants:

- `codecli`: code du client (entier), clé primaire avec l'attribut `codefilm`;
- `codefilm`: code du film (entier), clé primaire avec l'attribut `codecli`;
- `datedebut`: date de début de la location (chaîne de caractères);
- `duree`: durée de la location (flottant).

5. Écrire une requête SQL permettant de connaître la plus grande durée de location parmi tous les films.

6. Écrire une requête SQL permettant d'extraire le code du film, le nom du film et la durée moyenne de location des films qui ont été en moyenne loués moins de 2 jours. Le résultat doit être classé dans l'ordre décroissant des durées moyennes de location.

Exercice 2

L'objet de cette partie est l'étude de certains algorithmes sur le groupe des permutations d'un ensemble fini.
Pour $n \in \mathbb{N}^*$, on note S_n le groupe des permutations de l'ensemble $[\![0, n-1]\!]$. Une permutation de S_n sera représentée en Python par une liste, dont l'élément d'indice `i` est l'image de `i` par cette permutation. Par exemple, la liste $[3, 1, 0, 2]$ représente la permutation $\sigma \in S_4$ définie par $\sigma(0) = 3$, $\sigma(1) = 1$, $\sigma(2) = 0$ et $\sigma(3) = 2$.
Dans tout l'exercice, on pourra utiliser librement les tests Python du type `x in L` (respectivement `x not in L`) permettant de vérifier si `x` est présent dans la liste `L` (respectivement de vérifier si `x` n'est pas présent dans la liste `L`).

1. Si `s`, est une liste Python représentant une permutation de S_4, quelle instruction Python permet de trouver l'image de `1` par cette permutation ?

   Quelle liste Python représente la transposition $\begin{pmatrix} 2 & 3 \end{pmatrix} \in S_4$?

2. Écrire une fonction Python `comp(s1,s2)` prenant en entrée deux listes représentant des permutations σ_1 et σ_2 du même groupe de permutations et renvoyant la liste représentant la permutation $\sigma_1 \circ \sigma_2$.

3. Écrire une fonction Python `inv(s)` prenant en entrée une liste représentant une permutation σ et renvoyant la liste représentant σ^{-1}.

4. On souhaite tester si un sous-ensemble G de S_n est ou non un sous-groupe de S_n. Écrire une fonction Python `groupe(G)`, prenant en entrée une liste de listes, où chaque sous-liste représente une permutation de S_n et renvoyant `True`, s'il s'agit bien d'un sous-groupe de S_n, `False` sinon.

5. Écrire une fonction Python `cyclique(s)`, prenant en entrée une liste `s` représentant une permutation σ de S_n et renvoyant le sous-groupe de S_n engendré par σ sous la forme d'une liste de listes.

Exercice 3

Dans cet exercice "Algorithme de décomposition primaire d'un entier" (*Informatique pour tous*), on se propose d'écrire un algorithme pour décomposer un entier en produit de nombres premiers. Les algorithmes demandés doivent être écrits en langage **Python**. On sera très attentif à la rédaction et notamment à l'indentation du code.

On définit la valuation p-adique [de n] pour p nombre premier et n entier naturel non nul.
Si p divise n, on note $v_p(n)$ le plus grand entier k tel que p^k divise n.
Si p ne divise pas n, on pose $v_p(n) = 0$.
L'entier $v_p(n)$ s'appelle la valuation p-adique de n.

1. Écrire une fonction booléenne `estPremier(n)` qui prend en argument un entier naturel non nul n et qui renvoie le booléen `True` si n est premier et le booléen `False` sinon. On pourra utiliser le critère suivant: un entier $n \geq 2$ qui n'est divisible par aucun entier $d \geq 2$ tel que $d^2 \leq n$, est premier.

2. En déduire une fonction `liste_premiers(n)` qui prend en argument un entier naturel non nul n et renvoie la liste des nombres premiers inférieurs ou égaux à n.

3. Pour calculer la valuation 2-adique de 40, on peut utiliser la méthode suivante:

   - 40 est divisible par 2 et le quotient vaut 20.
   - 20 est divisible par 2 et le quotient vaut 10.
   - 10 est divisible par 2 et le quotient vaut 5.
   - 5 n'est pas divisible par 2.

   La valuation 2-adique de 40 vaut donc 3.

   Écrire une fonction `valuation_p_adique(n,p)` **non récursive** qui implémente cet algorithme. Elle prend en arguments un entier naturel n non nul et un nombre premier p et renvoie la valuation p-adique de n. Par exemple, puisque $40 = 2^3 \times 5$, `valuation_p_adique(40,2)` renvoie 3, `valuation_p_adique(40,5)` renvoie 1 et `valuation_p_adique(40, 7)` renvoie 0.

4. Écrire une deuxième fonction cette fois-ci **récursive** `val_p_adique(n,p)` qui renvoie la valuation p-adique de n.

5. En déduire une fonction `decomposition_facteurs_premiers(n)` qui calcule la décomposition en facteurs premiers d'un entier $n \geq 2$.

   Cette fonction doit renvoyer la liste des couples $(p, v_p(n))$ pour tous les nombres premiers p qui divisent n.

   Par exemple, `decomposition_facteurs_premiers(40)` renvoie la liste `[[2, 3], [5, 1]]`.

Exercice 4

On se donne une suite de variables aléatoires $(U_n)_{n\geq 1}$ indépendantes identiquement distribuées, de loi définie par $\mathbb{P}(U_n = 1) = \mathbb{P}(U_n = -1) = 1/2$. Ces variables aléatoires sont définies sur un espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$ que l'on n'explicitera pas.
On appelle alors marche aléatoire sur $\mathbb{Z}$ le processus (suite de variables aléatoires) $(X_n)_{n\geq 0}$ défini par $X_0 = 0$ avec probabilité 1 et, pour $n \geq 1$, $X_n = X_{n-1} + U_n$.
On rappelle que, une fois le module random importé par `import random as rd` , la commande `rd.random()` renvoie une réalisation d'une V.A. de loi uniforme sur $[0; 1]$. On considèrera par ailleurs que la répétition de cette commande permet de simuler le tirage de variables aléatoires indépendantes.

1. Compléter la fonction `SimulU()` qui simule la réalisation d'une variable aléatoire distribuée comme les U_n :

```
def SimulU():
    x = rd.random()
    if x  ...
        return -1
    else :
        ...
```

2. Écrire une fonction `SimulX(n)` qui, un entier n étant donné, simule la réalisation d'une variable aléatoire X_n et renvoie sa valeur.

3. Adapter la fonction `SimulX(n)` en une fonction `SimulX_bis(n)` qui renvoie dans une liste la valeur simulée de X_n, le minimum atteint, le maximum, et le temps de premier retour à 0 avant n. Ce temps T_n de retour à 0 est défini comme le plus petit entier $0 < k \leq n$ tel que $X_k = 0$. Si aucun des X_k ne vaut 0, on affectera à T_n la valeur 0.

   Une des questions classiques en théorie des marches aléatoires est celle de la récurrence de la marche : une marche aléatoire est dite récurrente si la probabilité de retour en 0 en temps fini est égale à 1 (elle est dite transiente sinon). En notant :

$$T = \inf\{k > 0 : X_k = 0\} ,$$

   cet événement (retour en temps fini) est noté $(T < +\infty)$; on a donc

$$(T < +\infty) = \{\omega \in \Omega : \exists n \geq 1, X_n(\omega) = 0\} .$$

   On a également

$$(T < +\infty) = \bigcup_{n=1}^{+\infty} (T \leq n) .$$

   La suite d'événement $(T \leq n)_{n\geq 1}$ étant croissante (au sens de l'inclusion), on a :

$$\mathbb{P}(T < +\infty) = \lim_{n\to+\infty} \mathbb{P}(T \leq n) .$$

   On peut donc chercher à estimer $\mathbb{P}(T < +\infty)$ en estimant $\mathbb{P}(T \leq n)$ pour n grand.

4. Écrire une suite d'instruction qui renvoie, n étant donné, la probabilité empirique (fréquence) `freq` de l'événement $(T \leq n)$ sur un échantillon de taille N.