



Objectifs

Le candidat doit choisir 3 exercices qu'il traitera sur les 5 exercices proposés.

- Exercice 1 : Notion de Pile et programmation Python.
- Exercice 2 : Programmation et récursivité.
- Exercice 3 : Arbres binaires et les arbres binaires de recherche.
- Exercice 4 : Bases de données relationnelles et le langage SQL.
- Exercice 5 : Réseaux en général et les protocoles RIP et OSPF en particulier.

Exercice 1 : Notion de Pile et programmation Python

On munit la structure de données Pile de quatre fonctions primitives définies dans le tableau ci-dessous. :

- `creer_pile_vide` : $\emptyset \rightarrow \text{Pile}$
`creer_pile_vide()` : renvoie une pile vide
- `est_vide` : $\text{Pile} \rightarrow \text{Booléen}$
`est_vide(pile)` : renvoie True si pile est vide, False sinon
- `empiler` : $\text{Pile}, \text{Élément} \rightarrow \text{Rien}$
`empiler(pile, element)` : ajoute element au sommet de la pile
- `depiler` : $\text{Pile} \rightarrow \text{Élément}$
`depiler(pile)` : renvoie l'élément au sommet de la pile en le retirant de la pile

Question 1

On suppose dans cette question que le contenu de la pile P est le suivant (les éléments étant empilés par le haut). Quel sera le contenu de la pile Q après exécution de la suite d'instructions suivante

```
1 Q = creer_pile_vide ()
2 while not est_vide(P):
3     empiler(Q, depiler(P))
```

Avant	4
	2
	5
	8

Après	8
	5
	2
	4

On dépile la pile P et on empile la pile Q.

Question 2

1. On appelle hauteur d'une pile le nombre d'éléments qu'elle contient. La fonction `hauteur_pile` prend en paramètre une pile `P` et renvoie sa hauteur. Après appel de cette fonction, la pile `P` doit avoir retrouvé son état d'origine. Recopier et compléter sur votre copie le programme Python suivant implémentant la fonction `hauteur_pile` en remplaçant les ??? par les bonnes instructions.

```

1  def hauteur_pile(P):
2      Q = creer_pile_vide ()
3      n = 0
4      while not(est_vide(P)):
5          n=n+1 # c'est l'incrément du compteur
6          x = depiler(P)
7          empiler(Q,x)
8      while not(est_vide(Q)):
9          x=depiler(Q) # On récupère l'élément au sommet de la pile Q
10         empiler(P, x)
11     return n

```

2. Créer une fonction `max_pile` ayant pour paramètres une pile `P` et un entier `i`. Cette fonction renvoie la position `j` de l'élément maximum parmi les `i` derniers éléments empilés de la pile `P`. Après appel de cette fonction, la pile `P` devra avoir retrouvé son état d'origine. La position du sommet de la pile est 1.

```

1  def max_pile(P,i):
2      '''In : P pile et i entier <= hauteur_Pile
3          Out : renvoie la position j de l'élément maximum parmi les i
4              derniers éléments empilés de la pile P'''
5      Q=creer_pile_vide()
6      n=1
7      x=depiler(P)
8      empiler(Q,x)
9      maximum=x
10     rang=1
11     while not(est_vide(P)) and n<i:
12         n=n+1
13         x = depiler(P)
14         empiler(Q,x)
15         if x>maximum:
16             maximum=x
17             rang=n
18     while not(est_vide(Q)):
19         x=depiler(Q)
20         empiler(P, x)
21     return rang

```

Question 3

Créer une fonction retourner ayant pour paramètres une pile P et un entier j. Cette fonction inverse l'ordre des j derniers éléments empilés et ne renvoie rien. On pourra utiliser deux piles auxiliaires.

```
def retourner(P, j):
    '''In : P pile et j entier <= hauteur_Pile
    Out : None.
    Cette fonction inverse l'ordre des j derniers éléments empilés
    et ne renvoie rien'''
    Q = creer_pile_vide ()
    K = creer_pile_vide ()

    # on dépile les j derniers éléments de P que l'on empile dans Q
    # (ordre inversé)
    n=0
    while n<j and not(est_vide(P)):
        empiler(Q,depiler(P)) # comme dans la question 1
        n=n+1

    # on dépile les j derniers éléments de Q que l'on empile dans K
    # (ordre inversé) donc l'ordre redevient celui initial
    n=0
    while n<j and not(est_vide(Q)):
        empiler(K,depiler(Q))
        n=n+1

    # on dépile les j derniers éléments de K que l'on empile dans P
    # (ordre inversé)
    n=0
    while n<j and not(est_vide(K)):
        empiler(P,depiler(K))
        n=n+1
```

Question 4

L'objectif de cette question est de trier une pile de crêpes. On modélise une pile de crêpes par une pile d'entiers représentant le diamètre de chaque crêpe. On souhaite réordonner les crêpes de la plus grande (placée en bas de la pile) à la plus petite (placée en haut de la pile). On dispose uniquement d'une spatule que l'on peut insérer dans la pile de crêpes de façon à retourner l'ensemble des crêpes qui lui sont au-dessus. Le principe est le suivant :

- On recherche la plus grande crêpe.
- On retourne la pile à partir de cette crêpe de façon à mettre cette plus grande crêpe tout en haut de la pile.
- On retourne l'ensemble de la pile de façon à ce que cette plus grande crêpe se retrouve tout en bas.
- La plus grande crêpe étant à sa place, on recommence le principe avec le reste de la pile.

```
def tri_crepe(Pile):
    '''In : Pile
       out : None. Cette fonction va trier la pile'''
    n=hauteur_pile(Pile)
    while n!=0:
        k=max_pile(Pile,n)
        retourner(Pile,k)
        retourner(Pile,n)
        n=n-1
```

Exercice 2 : Programmation et récursivité

Question 1

On considère tous les chemins allant de la case (0, 0) à la case (2, 3) du tableau T donné en exemple.

Exemple avec $n = 3$ lignes et $p = 4$ colonnes.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)



Remarque historique

Compter ce nombre de chemins avec ce déplacement imposé, dans un tableau avec n lignes et p colonnes, est un problème historique. On nomme cela les chemins de Pascal, du nom de l'illustre mathématicien français Blaise Pascal (1623-1662).

En fait on peut facilement montrer que

- Le nombre de cases pour passer de la case (0,0) à la case $(n-1, p-1)$ est $n+p-1$ soit ici $3+4-1=6$.
- Le nombre de déplacements est $n+p-2$ soit $3+4-2=5$ ici, avec $n-1=2$ déplacements vers le bas et $p-1=3$ déplacements vers la droite.
- Par ailleurs le nombre de chemins pour passer de la case (0,0) à la case $(n-1, p-1)$ est :

$$\binom{n+p-2}{p-1} = \binom{n+p-2}{n-1}$$

Cela revient à choisir parmi les $n+p-2=5$ déplacements possibles, les $n-1=2$ déplacements vers le bas ou les $p-1=3$ déplacements vers la droite.

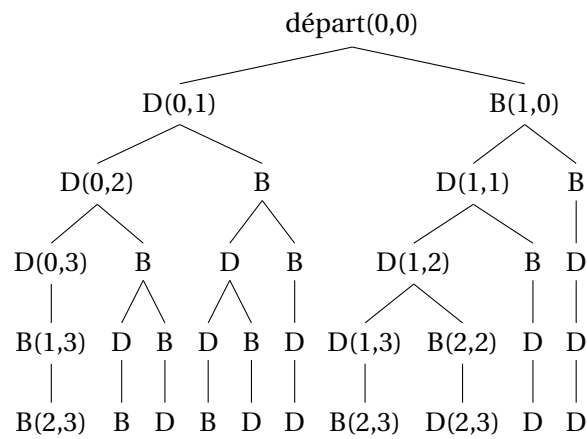
$$\binom{n+p-2}{p-1} = \binom{5}{3} = \frac{5!}{3!2!} = 10 \quad \text{ou} \quad \binom{n+p-2}{n-1} = \binom{5}{2} = \frac{5!}{2!3!} = 10$$

1. **Un tel chemin comprend nécessairement 3 déplacements vers la droite. Combien de déplacements vers le bas comprend-il ?**

Il comprend 2 chemins vers le bas.

2. **La longueur d'un chemin est égal au nombre de cases de ce chemin. Justifier que tous les chemins allant de (0, 0) à (2, 3) ont une longueur égale à 6.**

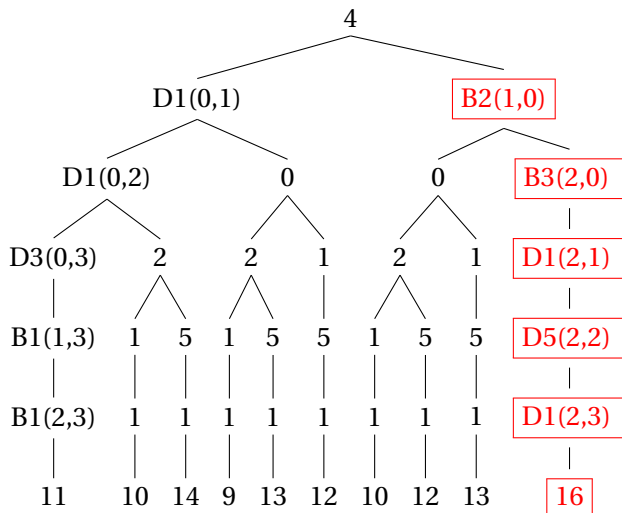
- Pour chaque déplacement Droite ou Bas, on arrive sur une nouvelle case. On ne peut faire que 3 déplacements vers la droite et 2 vers le bas. Ce qui fait 5 cases accessibles plus la case de départ donc un chemin est de longueur 6.
- On peut aussi faire un arbre de tous les chemins possibles (il y en a 10) et on s'aperçoit que leur longueur est 6 car chaque noeud de l'arbre correspond à une case.

Exemple avec $n = 3$ lignes et $p = 4$ colonnes.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

Question 2

En listant tous les chemins possibles allant de (0, 0) à (2, 3) du tableau T, déterminer un chemin qui permet d'obtenir la somme maximale et la valeur de cette somme.



(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

Le chemin qui donne la somme maximale 16 est donc le chemin :

$(0,0) \rightarrow (1,0) \rightarrow (2,0) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (2,3)$

Question 3

On veut créer le tableau T' où chaque élément $T'[i][j]$ est la somme maximale pour tous les chemins possibles allant de (0, 0) à (i, j).

1. Compléter et recopier sur votre copie le tableau T' donné ci-dessous associé au tableau T.

$$T = \begin{array}{|c|c|c|c|} \hline 4 & 1 & 1 & 3 \\ \hline 2 & 0 & 2 & 1 \\ \hline 3 & 1 & 5 & 1 \\ \hline \end{array}$$

$$T' = \begin{array}{|c|c|c|c|} \hline 4 & 5 & 6 & 9 \\ \hline 6 & 6 & 8 & 10 \\ \hline 9 & 10 & 15 & 16 \\ \hline \end{array}$$

— Pour $T'[0][3] = 9$.

Il n'y a qu'un seul chemin possible, il est de somme $9 = 4 + 1 + 1 + 3$.

$(0,0) = 4 \rightarrow (0,1) = 1 \rightarrow (0,2) = 1 \rightarrow (0,3) = 3$ donc $S = 4 + 1 + 1 + 3 = 9$

— Pour $T'[1][1] = 6$.

Il n'y a que 2 chemins possibles de (0,0) à (1,1), de somme 6 et 5.

$(0,0) = 4 \rightarrow (1,0) = 2 \rightarrow (1,1) = 0$ donc $S = 4 + 2 + 0 = 6$

$(0,0) = 4 \rightarrow (0,1) = 1 \rightarrow (1,1) = 0$ donc $S = 4 + 1 + 0 = 5$

— Pour $T'[2][2] = 15$.

Il y a 6 chemins possibles de (0,0) à (2,2). Remarque $\binom{4}{2} = \frac{4!}{2!2!} = 6$.

- $(0,0) = 4 \rightarrow (1,0) = 2 \rightarrow (2,0) = 3 \rightarrow (2,1) = 1 \rightarrow (2,2) = 5$ donc $S = 4+2+3+1+5 = 15$;
- $(0,0) = 4 \rightarrow (0,1) = 1 \rightarrow (0,2) = 1 \rightarrow (1,2) = 2 \rightarrow (2,2) = 5$ donc $S = 4+1+1+2+5 = 13$;
- Les 4 autres sont de sommes : $4+2+0+1+5 = 12$; $4+2+0+2+5 = 12$; $4+1+0+2+5 = 12$ et $4+1+0+1+5 = 11$.

2. Justifier que si j est différent de 0, alors : $T'[0][j] = T[0][j] + T'[0][j-1]$.

(0,0)	...	(0, j-1)	(0, j)
(1,0)
(2,0)

Le seul chemin possible pour aller à la case (0, j) passe par la case (0, j-1).

De ce fait $T'[0][j]$ qui est la somme maximale pour tous les chemins possibles allant de (0,0) à (0, j) est la somme de :

- la valeur de la case (0, j) soit $T[0][j]$;
- la somme maximale pour tous les chemins possibles allant de (0, 0) à (0, j-1).

Soit

$$T'[0][j] = T[0][j] + T'[0][j-1]$$

Question 4

Justifier que si i et j sont différents de 0, alors : $T'[i][j] = T[i][j] + \max(T'[i-1][j], T'[i][j-1])$.

...
(i-1, 0)	...	(i-1, j-1)	(i-1, j)
(i, 0)	...	(i, j-1)	(i, j)

- Le seul chemin possible pour aller à la case (i, j) passe par la case (i, j-1) ou par la case (i-1, j).
- De ce fait $T'[i][j]$ qui est la somme maximale pour tous les chemins possibles allant de (0,0) à (i, j) est la somme de :
 - la valeur de la case (i, j) soit $T[i][j]$;
 - le maximum entre :
 - la somme maximale pour tous les chemins possibles allant de (0, 0) à (i, j-1) soit $T'[i][j-1]$;
 - la somme maximale pour tous les chemins possibles allant de (0, 0) à (i-1, j) soit $T'[i-1][j]$.

Soit

$$T'[i][j] = T[i][j] + \max(T'[i-1][j], T'[i][j-1])$$

Question 5

On veut créer la fonction récursive `somme_max` ayant pour paramètres un tableau `T`, un entier `i` et un entier `j`. Cette fonction renvoie la somme maximale pour tous les chemins possibles allant de la case (0, 0) à la case (i, j).

1. Quel est le cas de base, à savoir le cas qui est traité directement sans faire appel à la fonction `somme_max`? Que renvoie-t-on dans ce cas?

Le cas de base est le cas où le tableau ne contient qu'une seule case dans ce cas on renvoie `T[0][0]`.

2. À l'aide de la question précédente, écrire en Python la fonction récursive `somme_max`.

```
from numpy import *

# On utilise la formule de la question 4
# T'[i][j] = T[i][j] + max(T'[i-1][j], T'[i][j-1])

def somme_max(T, i, j):
    if i == 0:
        if j == 0:
            return T[0][0]
        else:
            return T[0][j] + somme_max(T, 0, j-1)
            # on calcule la somme sur la première ligne jusqu'à j
    else:
        if j == 0:
            return T[i][0] + somme_max(T, i-1, 0)
            # on calcule la somme sur la première colonne jusqu'à i
        else:
            return T[i][j] + max(somme_max(T, i-1, j), somme_max(T, i, j-1))
            # C'est la formule finale
```

3. Quel appel de fonction doit-on faire pour résoudre le problème initial?

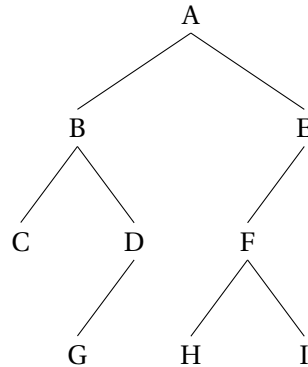
Il suffit d'appeler : `somme_max(T)`.

Exercice 3 : Arbres binaires et les arbres binaires de recherche

Dans cet exercice, on utilisera la convention suivante : la hauteur d'un arbre binaire ne comportant qu'un nœud est 1.

Question 1

Déterminer la taille et la hauteur de l'arbre binaire suivant :



Arbre binaire

1. On appelle **taille d'un arbre** le nombre de nœuds présents dans cet arbre.
2. Dans un **arbre binaire**, un nœud possède au plus 2 fils.
3. On appelle **profondeur d'un nœud** ou d'une feuille dans un arbre binaire le nombre de nœuds du chemin qui va de la racine à ce nœud. La racine d'un arbre est à une profondeur 1 (ici, mais cela dépend de la convention).
4. On appelle **hauteur d'un arbre** la profondeur maximale des nœuds de l'arbre.

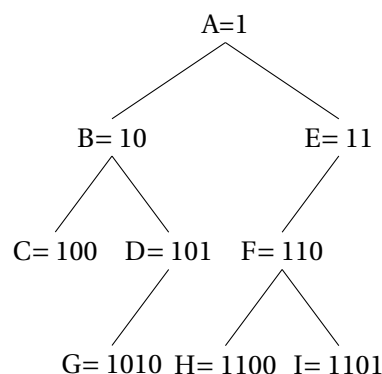
- Par définition, la taille d'un arbre est le nombre de nœud qu'il contient. Ici il y en a 9 donc la taille est 9.
- Par définition la hauteur est le nombre de nœuds du chemin le plus long dans l'arbre, ici les chemins les plus long sont ABDG, AEFH et AEFI. Comme la hauteur d'un arbre ne contenant qu'un nœud est 1, la hauteur de cet arbre est 4.

Question 2

On décide de numéroté en binaire les nœuds d'un arbre binaire de la façon suivante :

- la racine correspond à 1;
- la numérotation pour un fils gauche s'obtient en ajoutant le chiffre 0 à droite au numéro de son père;
- la numérotation pour un fils droit s'obtient en ajoutant le chiffre 1 à droite au numéro de son père.

1. Dans l'exemple précédent, quel est le numéro en binaire associé au nœud G?



On a donc $G : 1010$.

2. **Quel est le nœud dont le numéro en binaire vaut 13 en décimal ?**

$13_{10} = 1101_2$ or $I : 1101$ donc cela correspond au nœud I.

3. **En notant h la hauteur de l'arbre, sur combien de bits seront numérotés les nœuds les plus bas ?**

A chaque niveau de l'arbre on rajoute 1 bit donc les numéros des nœuds les plus bas (les feuilles) contiennent h bits.

4. **Justifier que pour tout arbre de hauteur h et de taille $n \geq 2$, on a : $h \leq n \leq 2^h - 1$.**

— Soit un arbre de hauteur h . Il contient un maximum de nœuds si il est complet. Or un arbre binaire complet de hauteur h contient $1 + 2 + 2^2 + \dots + 2^{h-1}$ nœuds. C'est la somme d'une suite géométrique de raison 2 donc :

$$1 + 2 + 2^2 + \dots + 2^{h-1} = \frac{2^h - 1}{2 - 1} = 2^h - 1$$

On a donc $n \leq 2^h - 1$.

— Soit un arbre de hauteur h . La hauteur maximale que l'on peut obtenir avec un minimum de nœuds est le cas où chaque père n'a qu'un seul fils. Dans ce cas la hauteur est égale à n . Donc $h \leq n$.

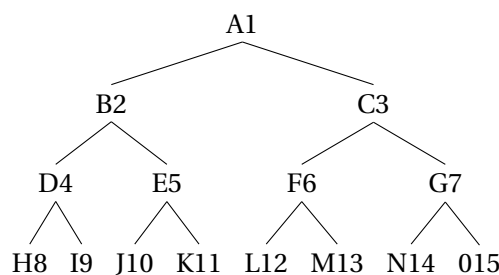
— Donc $h \leq n \leq 2^h - 1$

Question 3

Un arbre binaire est dit complet si tous les niveaux de l'arbre sont remplis. On décide de représenter un arbre binaire complet par un tableau de taille $n + 1$, où n est la taille de l'arbre, de la façon suivante :

- La racine a pour indice 1 ;
- Le fils gauche du nœud d'indice i a pour indice $2 \times i$;
- Le fils droit du nœud d'indice i a pour indice $2 \times i + 1$;
- On place la taille n de l'arbre dans la case d'indice 0.

1. **Déterminer le tableau qui représente l'arbre binaire complet de l'exemple précédent.**



On a donc le tableau

15, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O

2. **On considère le père du nœud d'indice i avec $i \geq 2$. Quel est son indice dans le tableau ?**

Le père d'un fils d'indice i a pour indice $i/2$ si i est pair et $(i - 1)/2$ sinon.

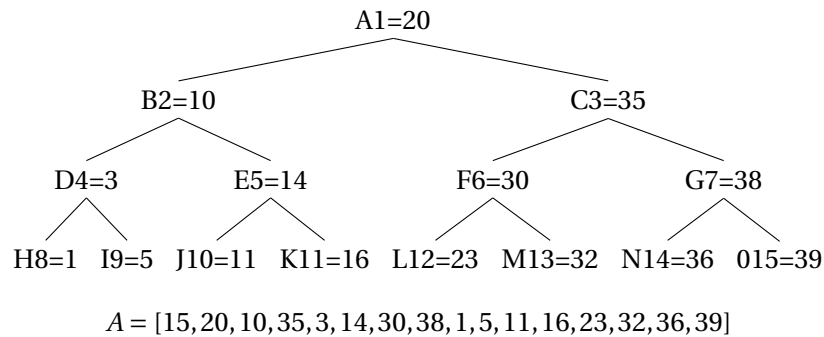
Sous python on peut dans ce cas utiliser la fonction `//`.

`a//b` donne le quotient de la division euclidienne de a par b .

Question 4

On se place dans le cas particulier d'un arbre binaire de recherche complet où les nœuds contiennent des entiers et pour lequel la valeur de chaque nœud est supérieure à celles des nœuds de son fils gauche, et inférieure à celles des nœuds de son fils droit.

On a par exemple un arbre de ce type :



Écrire une fonction *recherche* ayant pour paramètres un arbre *arbre* et un élément *element*. Cette fonction renvoie *True* si *element* est dans l'arbre et *False* sinon. L'arbre sera représenté par un tableau comme dans la question précédente.

```

def recherche(arbre, element):
    '''In : arbre et element entier
    Out : true si element est dans la liste'''
    taille = arbre[0]
    i=1
    while i<=taille:
        if element==arbre[i]:
            return True
        elif element>arbre[i]:
            i=2*i+1
        else:
            i=2*i
    return False
  
```

Exercice 4 : Bases de données relationnelles et le langage SQL

L'énoncé de cet exercice utilise les mots du langage SQL suivant : SELECT, FROM, WHERE, JOIN, INSERT INTO, VALUES, COUNT, ORDER BY.

Pour les besoins de l'organisation du lycée, le chef d'établissement exploite la base de données par des requêtes en langage SQL. Il a pour cela créé une table (ou relation) SQL dénommée *seconde* dans son système de gestion de bases de données dont la structure est la suivante :

seconde	Type
num_eleve (clef primaire)	entier (??)
langue1	CHAR
langue2	CHAR
option	CHAR
classe	CHAR

Question 1

1. Dans le modèle relationnel, quel est l'intérêt de l'attribut **num_eleve**.

La clé primaire d'une relation est un attribut qui permet de désigner d'une façon unique un uplet.

Par exemple l'attribut *num_eleve* permet d'identifier de façon unique les uplets de la table (ou relation) *seconde*. La seule connaissance de la clé primaire permet d'identifier toute ligne de la table.

Cet identifiant (souvent auto géré) unique est ici associé à un élève, par essence unique dans l'établissement.

2. Écrire une requête SQL d'insertion permettant d'enregistrer l'élève ACHIR Mussa dans la table *seconde*. Les informations relatives à cet élève sont données dans la ligne 1 du fichier *seconde_lyc.csv*.

num_eleve	nom	prenom	datenaissance	langue1	langue2	option	classe
133310FE	ACHIR	Mussa	01/01/2005	anglais	espagnol		2A
156929JJ	ALTMAYER	Yohan	05/05/2005	allemand	anglais	théâtre	2D



Remarque

Un problème ici car la clé primaire proposée n'est pas un entier. Généralement cette clé est automatiquement attribuée par la bdd. On ne va prendre en compte que la partie de la clé composée de chiffres.

On ne complète pas la colonne **option** car l'élève n'en a pas. On suppose évidemment que cet attribut peut être NULL. cela doit être spécifié lors de la création de la table.

```
INSERT INTO seconde(num_eleve, langue1, langue2, classe)
VALUES (133310, 'anglais', 'espagnol', '2A') ;
```

3. Lors de l'insertion de l'élève ALTMAYER Yohan (ligne 2 du fichier *seconde_lyc.csv*), une erreur de saisie a été commise sur la première langue, qui devrait être allemand. Écrire une requête SQL de mise à jour corrigeant les données de cet élève.

```
UPDATE seconde
SET langue1 = 'allemand'
WHERE num_eleve = 156929 ;
```

Question 2

On suppose maintenant que la table *seconde* contient les informations issues de la figure 1 (ni plus, ni moins, même si la figure 1 n'est qu'un extrait du fichier *seconde_lyc.csv*).

1. Quel est le résultat de la requête `SELECT num_eleve FROM seconde;`?

```
SELECT num_eleve FROM seconde ;
```

Le résultat de la requête sera la colonne des clefs primaires *num_eleve* :

num_eleve (clef primaire)
133310
156929
...
666702

2. On rappelle qu'en SQL, la fonction d'agrégation `COUNT()` permet de compter le nombre d'enregistrements dans une table. Quel est le résultat de la requête `SELECT COUNT(num_eleve) FROM seconde;`?

```
SELECT COUNT(num_eleve) FROM seconde ;
```

Le résultat de la requête sera le nombre de lignes (de clefs primaires) de la table *seconde* donc ici 30.

3. Écrire la requête permettant de connaître le nombre d'élèves qui font allemand en *langue1* ou *langue2*.

```
SELECT COUNT(num_eleve) FROM seconde
WHERE langue1='allemand' OR langue2='allemand' ;
```

Question 3

Le chef d'établissement souhaite faire évoluer la structure de sa base de données. Pour ce faire, il crée une nouvelle table *eleve* dont la structure est la suivante :

eleve	Type
num_eleve (clef primaire, clef étrangère de la table <i>seconde</i>)	entier (??)
nom	CHAR
prenom	CHAR
datenaissance	CHAR

Là encore, l'attribut *num_eleve* est un entier, les autres sont des chaînes de caractère (le type CHAR).

1. Expliquer ce qu'apporte l'information clef étrangère pour l'attribut *num_eleve* de cette table en termes d'intégrité et de cohérence.

Les clés étrangères permettent de gérer des relations entre plusieurs tables, et garantissent la cohérence des données. On peut ainsi modifier des données d'un élève sans avoir à modifier plusieurs tables.

2. On suppose la table *eleve* correctement créée et complétée. Le chef d'établissement aimerait lister les élèves (nom, prénom, date de naissance) de la classe 2A. Écrire la commande qui permet d'établir cette liste à l'aide d'une jointure entre *eleve* et *seconde*.

seconde	Type
num_eleve (clef primaire)	entier (??)
langue1	CHAR
langue2	CHAR
option	CHAR
classe	CHAR

eleve	Type
num_eleve (clef primaire, clef étrangère de la table seconde)	entier (??)
nom	CHAR
prenom	CHAR
datenaissance	CHAR

```
SELECT nom, prenom ,datenaissance
FROM eleve
INNER JOIN seconde
ON seconde.num_eleve = eleve.num_eleve
WHERE seconde.classe='2A' ;
```

Question 4

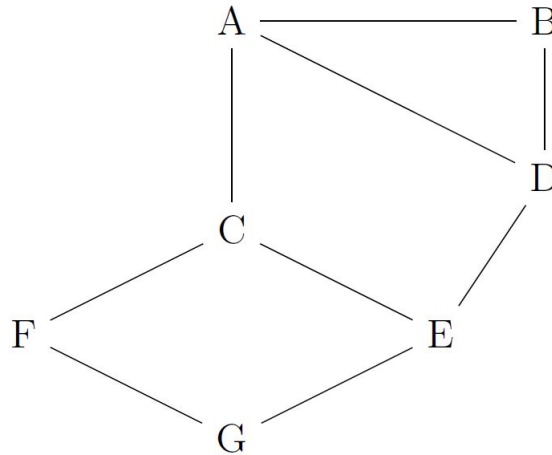
Proposer la structure d'une table *coordonnees* dans laquelle on pourra indiquer, pour chaque élève, son adresse, son code postal, sa ville, son adresse mail. Préciser la clef primaire et/ou la clé étrangère en vue de la mise en relation avec les autres tables.

coordonnees	Type
num_eleve (clef primaire, clef étrangère de la table seconde)	entier (??)
adresse	CHAR ou TEXT (pour des données plus longues)
codepostal	INT
ville	CHAR
email	CHAR

Exercice 5 : Réseaux en général et les protocoles RIP et OSPF en particulier

Le protocole RIP

Le protocole RIP permet de construire les tables de routage des différents routeurs, en indiquant pour chaque routeur la distance, en nombre de sauts, qui le sépare d'un autre routeur.



Question 1

1. Le routeur A doit transmettre un message au routeur G, en effectuant un nombre minimal de sauts. Déterminer le trajet parcouru.

Il y a deux trajets possible ACFG et ACEG. La distance est de 3.

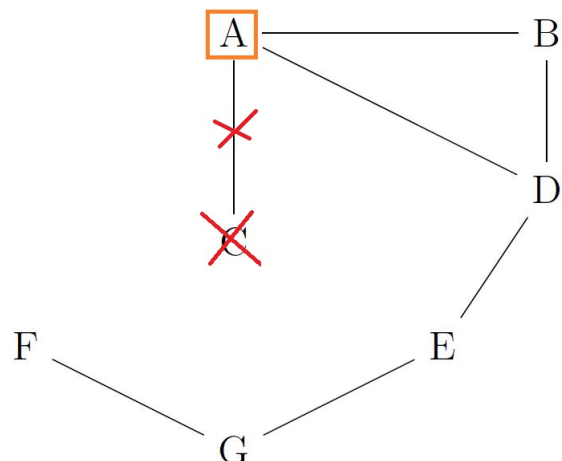
2. Déterminer une table de routage possible pour le routeur G obtenu à l'aide du protocole RIP.

Table de routage de G		
Destination	Routeur Suivant	Distance
A	E (ou F)	3
B	E	3
C	E (ou F)	2
D	E	2
E	E	1
F	F	1

Question 2

Le routeur C tombe en panne. Reconstruire la table de routage du routeur A en suivant le protocole RIP.

Table de routage de A		
Destination	Routeur Suivant	Distance
B	B	1
D	D	1
E	D	2
F	D	4
G	D	3

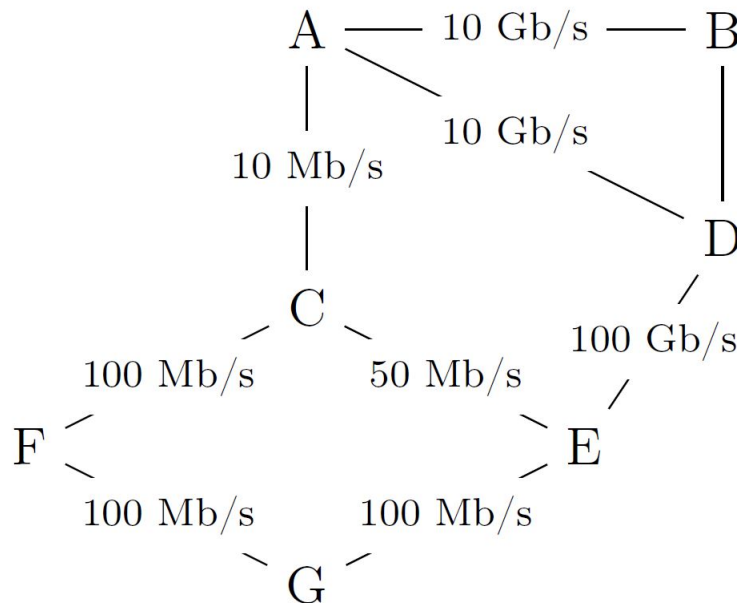


Le protocole OSPF

Contrairement au protocole RIP, l'objectif n'est plus de minimiser le nombre de routeurs traversés par un paquet. La notion de distance utilisée dans le protocole OSPF est uniquement liée aux coûts des liaisons. L'objectif est alors de minimiser la somme des coûts des liaisons traversées. Le coût d'une liaison est donné par la formule suivante :

$$\text{coût} = \frac{10^8}{d}$$

où d est la bande passante en bits/s entre les deux routeurs. On a rajouté sur le graphe représentant le réseau précédent les différents débits des liaisons. On rappelle que $1 \text{ Gb/s} = 1\,000 \text{ Mb/s} = 10^9 \text{ bits/s}$.

**Question 3**

1. **Vérifier que le coût de la liaison entre les routeurs A et B est 0,01.**

Entre A et B la distance est de $d = 10 \text{ Gb/s} = 10^{10} \text{ bits/s}$. Donc le coût est :

$$\text{coût} = \frac{10^8}{10^{10}} = 10^{-2} = \underline{0,01}$$

2. **La liaison entre le routeur B et D a un coût de 5. Quel est le débit de cette liaison?**

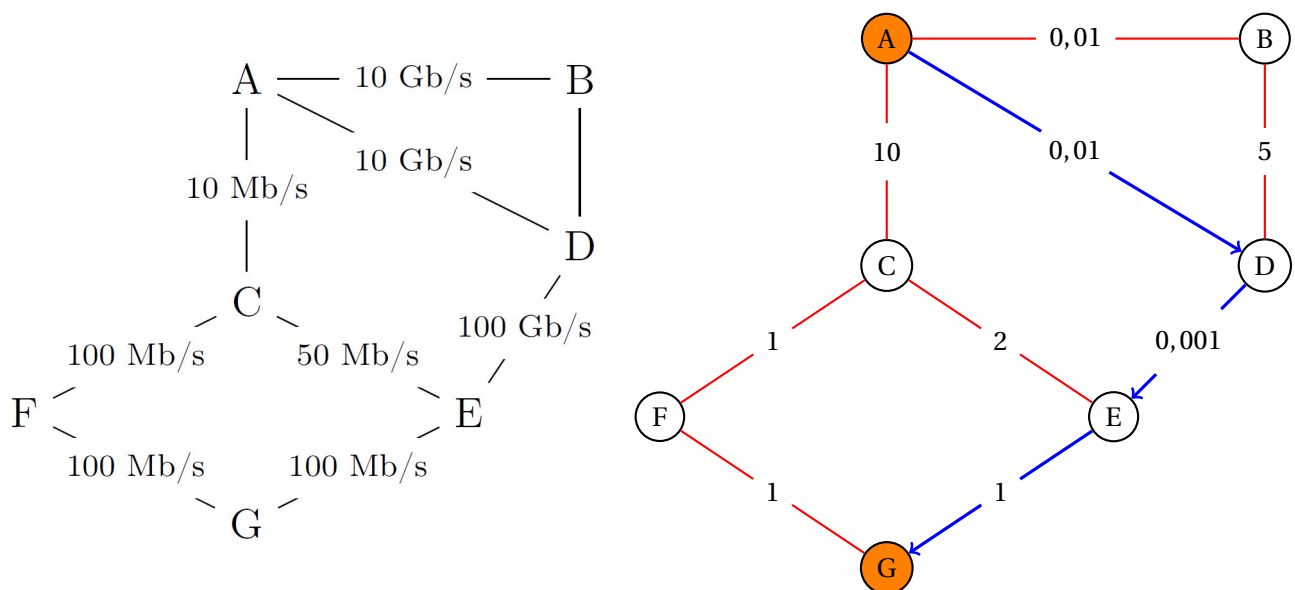
On a :

$$\text{coût} = \frac{10^8}{d} = 5 \iff d = \frac{10^8}{5} = 2 \cdot 10^7 \text{ bits/s} = \underline{20 \text{ Mb/s}}$$

Question 4

Le routeur A doit transmettre un message au routeur G, en empruntant le chemin dont la somme des coûts sera la plus petite possible. Déterminer le chemin parcouru. On indiquera le raisonnement utilisé. On construit le graphe en indiquant les coûts sur chaque arête.

Débit d bits/s	Coût = $\frac{10^8}{d}$
$d = 10\text{Gb/s} = 10^{10}\text{bits/s}$	coût = $10^8 \times 10^{-10} = 10^{-2} = 0,01$
$d = 100\text{Gb/s} = 10^{11}\text{bits/s}$	coût = $10^8 \times 10^{-11} = 10^{-3} = 0,001$
$d = 100\text{Mb/s} = 10^8\text{bits/s}$	coût = $10^8 \times 10^{-8} = 1$
$d = 50\text{Mb/s} = 5 \times 10^7\text{bits/s}$	coût = $0,2 \times 10^8 \times 10^{-7} = 2$
$d = 10\text{Mb/s} = 10^7\text{bits/s}$	coût = $10^8 \times 10^{-7} = 10$



On utilise l'algorithme de **Dijkstra** pour trouver le chemin avec le coût minimal.

Départ	A	B	C	D	E	F	G	Sommet choisi
	0	∞	∞	∞	∞	∞	∞	A
A(0)		0,01 (A)	10 (A)	0,01(A)	∞	∞	∞	B (A)
B(0,01)			10 (A)	0,01(A)	∞	∞	∞	D (A)
D(0,01)			10 (A)		0,011(D)	∞	∞	E(D)
E(0,011)			2,011(E)			∞	1,011(E)	G (E)
G(1,011)						2,011(G)		F (G)
F(2,011)			2,011(E)					C (E)

Le parcours avec un coût minimal pour aller de A à G est donc ADEG dont le coût est 1,011.

$$A \xrightarrow{0,01} D \xrightarrow{0,001} E \xrightarrow{1} G$$

↩ Fin du devoir ↩