Numérique et Sciences Informatiques

Epreuve de BAC Blanc

Février 2021

Durée de l'épreuve : 3 heures

Le sujet comporte 13 pages.

L'annexe des pages 12 et 13 est à rendre avec votre copie.

Exercice 1 : le codage de Huffman (8 points) **1**
Exercice 2 : Base de données dans un hôpital (6 points) **5**
Exercice 3 : Récursivité (4 points)..... **6**
Exercice 4 : Graphes (4 points)..... **7**
Exercice 5 : Evaluer une expression numérique avec une pile (4 points)..... **8**
Exercice 6 : Protocoles de routage RIP et OSPF (4 points)..... **10**

Le barème indicatif est donné sur 30 points.

L'usage de la calculatrice est autorisé.

Exercice 1 : le codage de Huffman (8 points)

Ce sujet propose d'étudier une méthode de compression de données inventée par David Albert Huffman en 1952, qui permet de réduire la longueur du codage d'un alphabet et qui repose sur la création d'un arbre binaire.

On appelle *alphabet* l'ensemble des symboles (caractères) composant la donnée de départ à compresser. Dans la suite, nous utiliserons un alphabet composé seulement des 8 lettres A, B, C, D, E, F, G et H.

Partie A

1) On cherche à coder chaque lettre de cet alphabet par une séquence de chiffres binaires.
   a) Combien de bits sont nécessaires pour coder chacune des 8 lettres de l'alphabet.
   b) Quelle est la longueur en octets d'un message de 1 000 caractères construit sur cet alphabet ?
2) Proposer un code de taille fixe pour chaque caractère de l'alphabet de 8 lettres.
3) On considère maintenant le codage suivant, la longueur du code de chaque caractère étant variable.

Lettre	A	B	C	D	E	F	G	H
Code	10	001	000	1100	01	1101	1110	1111

   Ce type de code est dit *préfixe*, ce qui signifie qu'aucun code n'est le préfixe d'un autre (le code de A est 10 et aucun autre code ne commence par 10, le code de B est 001 et aucun autre code ne commence par 001). Cette propriété permet de séparer les caractères de manière non ambiguë.
   a) En utilisant la table précédente, donner le code du message : CACHE.
   b) Quel est le message correspondant au code 001101100111001.
4) Dans un texte, chacun des 8 caractères a un nombre d'apparitions différent. Cela est résumé dans le tableau suivant, construit à partir d'un texte de 1 000 caractères.

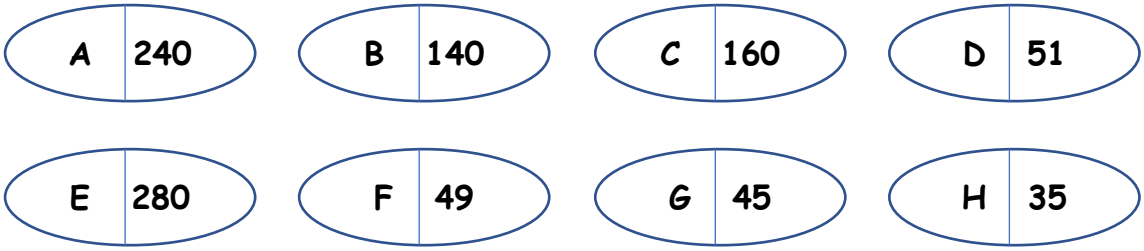
Lettre	A	B	C	D	E	F	G	H
Nombre	240	140	160	51	280	49	45	35

   a) En utilisant le code de taille fixe proposé à la question 2., quelle est la longueur en bits du message contenant les 1 000 caractères énumérés dans le tableau précédent ?
   b) En utilisant le code de la question 3., quelle est la longueur du même message en bits.

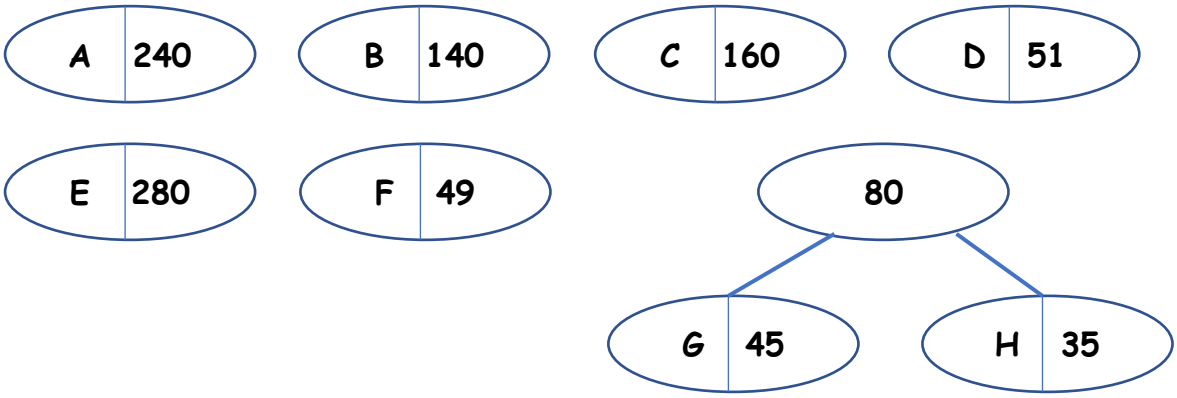
Partie B

1) L'objectif du codage de Huffman est de trouver le codage proposé en A 3., qui minimise la taille en nombre de bits du message codé en se basant sur le nombre d'apparitions de chaque caractère (un caractère qui apparaît souvent aura un code plus court).

Pour déterminer le code optimal, on considère 8 arbres, chacun réduit à une racine, contenant le symbole et son nombre d'apparitions.



Puis on fusionne les deux arbres contenant **les plus petits nombres d'apparitions** (valeur inscrite sur la racine), et on affecte à ce nouvel arbre la somme des nombres d'apparitions de ses deux sous-arbres. Lors de la fusion des deux arbres, le choix de mettre l'un ou l'autre à gauche n'a pas d'importance. Nous choisissons ici de mettre le plus fréquent à gauche (s'il y a un cas d'égalité, nous faisons un choix arbitraire).

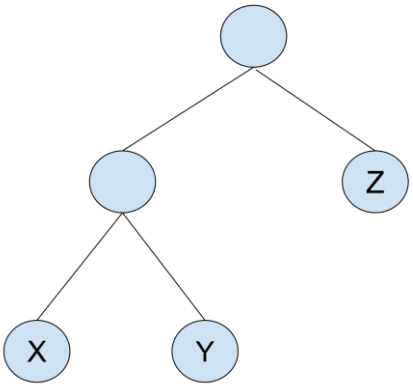


On recommence jusqu'à ce qu'il n'y ait plus qu'un seul arbre.

Combien d'étapes (combien de fusions d'arbres) sont nécessaires pour que cet algorithme se termine ?

2) En suivant l'algorithme précédent, construire l'arbre de Huffman.
3) Le code à affecter à chaque lettre est déterminé par sa position dans l'arbre. Précisément, le code d'un symbole de l'alphabet décrit le chemin de la racine à la feuille qui le contient : un 0 indique qu'on descend par le fils gauche et un 1 indique qu'on descend par le fils droit.

Dans le cas de l'arbre ci-contre, le code de X est 00 (deux fois à gauche), le code de Y est 01, et celui de Z est 1.



Quel est le code de F ?

Partie C

En annexe, vous trouverez un code Python qui permet, à partir d'un fichier nommé *texte.txt*, de construire l'arbre de Huffman puis un dictionnaire qui associe à chaque caractère du fichier d'entrée son code sous forme d'une séquence de bits (liste de 0 et de 1).

Compléter le code en indiquant ce qui manque dans les zones rectangulaires.

Et rendre l'annexe avec votre copie en indiquant votre nom.

Exercice 2 : Base de données dans un hôpital (6 points)

On veut créer une base de données relative à la gestion d'un hôpital qui contient les 3 tables suivantes.

Patients
id : entier
nom : texte
prenom : texte
genre : texte
Annee_naissance : entier

Ordonnances
code : entier
id_patient : entier
matricule_medecin: entier
Date_ordonnance : date
medicaments : texte

Medecins
matricule : entier
prenom : texte
nom : texte
specialite : texte
telephone : texte

On suppose que les dates sont données sous la forme jj-mm-aaaa.

Exemple : 14-11-2020 pour le 14 novembre 2020.

1) Madame Anne Wizeunid née en 2000 doit être enregistrée comme patiente. Donner la commande SQL correspondante.
2) Le patient numéro 100 a changé de genre et est maintenant une femme. Donner la commande SQL modifiant en conséquence ses données.
3) Par souci d'économie, la direction décide de se passer des médecins spécialisés en épidémiologie. Donner la commande SQL permettant de supprimer leur fiche.
4) Donner la requête SQL permettant d'obtenir la liste des prénoms et noms des patients qui sont de sexe féminin triée dans l'ordre croissant des âges.
5) Donner la requête SQL qui donne la liste des patient(e)s ayant été examiné(e)s par un(e) psychiatre le 1er avril 2020.
6) Que fait la requête suivante :

```
SELECT m.prenom, m .nom
FROM Medecins as m
JOIN Ordonnances AS o
ON m.matricule = o.matricule_medecin
WHERE specialite = "ophtalmologie"
```

Exercice 3 : méthode du paysan pour la multiplication Récursivité (4 points)

La méthode du paysan russe est un très vieil algorithme de multiplication de deux nombres entiers déjà écrit, sous une forme légèrement différente, sur un papyrus égyptien rédigé autour de 1650 avant J.-C. Il s'agissait de la principale méthode de calcul en Europe avant l'introduction des chiffres arabes.

Les premiers ordinateurs l'ont utilisé avant que la multiplication ne soit directement intégrée dans le processeur sous forme de circuit électronique.

Sous une forme moderne, il peut être décrit ainsi :

Fonction **Multiplication (x,y) :**

```
p =0
TANT QUE x > 0 :
    Si x est impair :
        p = p + y
    x = x // 2
    y = y + y
retourner p
```

On rappelle que l'opérateur x // 2 donne le quotient entier de la division euclidienne de x par 2.

1) Appliquer cette fonction pour effectuer la multiplication de 105 par 253. Détailler les étapes en remplissant le tableau suivant :

x	y	p
105	253	…
…	…	…

2) On admet que cet algorithme repose sur les relations suivantes :

$$x \times y = \begin{cases} \boldsymbol{0 \; si \; x \; = \; 0} \\ \boldsymbol{(x//2) * (y + y) \; si \; x \; est \; pair} \\ \boldsymbol{(x//2) * (y \; + \; y) \; + \; y \; si \; x \; est \; impair} \end{cases}$$

Proposer une fonction équivalente selon la méthode récursive.

Exercice 4 : Graphes (4 points)

On considère un groupe de dix personnes présentes sur un réseau social, le tableau suivant indique les paires de personnes qui ont une relation d'amitié dans ce réseau social.

i	Ami de i
1	3, 6, 7
2	6, 8
3	1, 6, 7
4	5, 10
5	4, 10
6	1, 2, 3, 7
7	1, 3, 6
8	2
9	
10	4,5

1) Représenter cette situation par un graphe dans lequel une arête montre le lien d'amitié.
2) Ce graphe est-il connexe ? Si non, donner ses composantes connexes.
3) Donner les parcours en largeur et en profondeur de ce graphe à partir de la personne numéro 1.
4) L'adage « les amis de mes amis sont nos amis » est-il vérifié ?
   Si non, que faudrait-il faire pour qu'il le soit ?

Exercice 5 : Evaluer une expression numérique avec une pile (4 points)

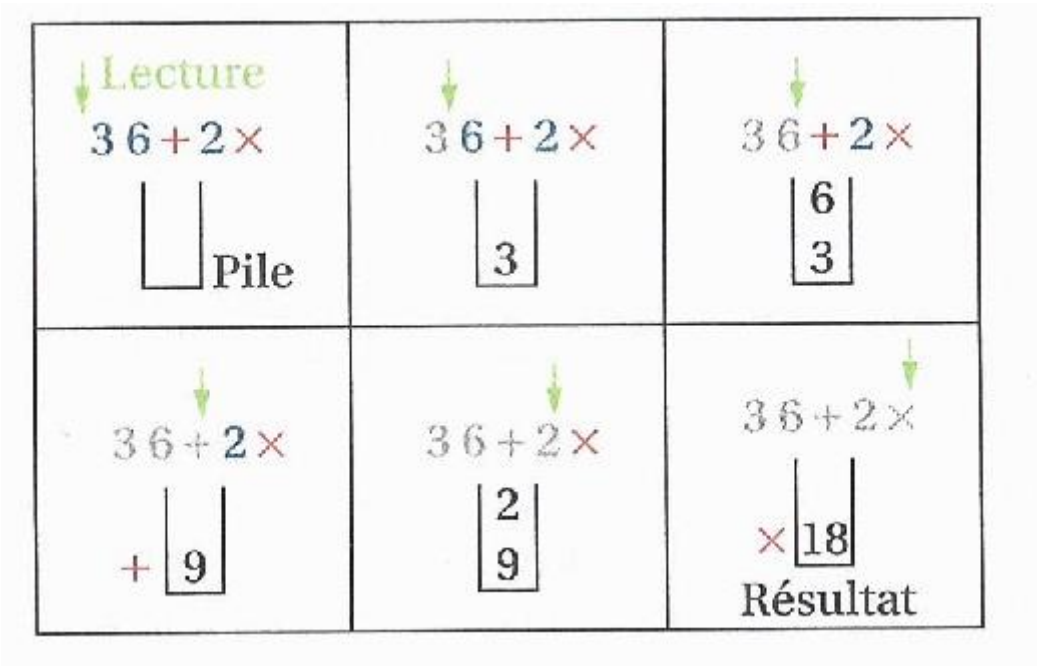
Il existe plusieurs façons de noter les expressions arithmétiques :

- la notation infixe (usuelle), utilisant des parenthèses, l'opérateur est indiqué **entre** les opérandes : (3 + 6) × 2 ;
- la notation postfixe, qui ne nécessite pas de parenthèse, l'opérateur est mentionné **après** les opérandes : 3 6 + 2 ×.

Une expression en notation postfixe peut facilement être évaluée à l'aide d'une **pile**, en même temps qu'elle est analysée. Pour cela, on lit l'expression de gauche à droite (voir figure) :

- si c'est un nombre, on l'empile ;
- si c'est un opérateur, on l'applique aux deux nombres qui sont au sommet de la pile et on empile le résultat

A la fin de la lecture de l'expression, la pile ne contient qu'un élément le résultat.



1) Coder les fonctions Python suivantes qui permettent de modéliser une pile par une liste avec les fonctions suivantes :

- ***creer_pile_vide()*** : crée et retourne une pile vide
- ***est_vide()*** : renvoie True si la pile est vide et False sinon
- ***empiler(P,e)*** : empile l'élément e au sommet de la pile P.
- ***depiler(P)*** : dépile et renvoie l'élément au sommet de la pile si la pile n'est pas vide et **None** si la pile est vide.

2) Coder la fonction ***calcul(P,op)*** qui :prend en paramètre une pile ***P*** et une opération ***op*** donnée sous la forme d'un caractère : +, -, *, /, dépile les deux éléments au sommet de la pile, leur applique l'opération et empile le résultat.
   Exemple :
   Calcul([14,11],'+') doit retourner 25.

3) Coder la fonction ***evaluation(exp)*** qui prend en paramètre une expression contenant des nombres et des symboles sous la forme d'une chaîne de caractères et évalue le résultat.
   Cette fonction devra utiliser une pile, utiliser la fonction précédente ***calcul(P,op)***.
   Exemples :
   - evaluation ('3 5 *') doit retourner $3 \times 5 = 15$
   - evaluation ('3 5 * 2 +') doit retourner $(3 \times 5) + 2 = 17$
   - evaluation ('8 1 1 + /') doit retourner $8 / (1 + 1) = 4$

   Indications :

   - On suppose que l'expression postfixée ***exp*** passée en paramètre est bien « formée » : on ne demande pas de gérer les cas d'erreur où l'expression n'est pas conforme.
   - Pour transformer la chaine de caractères ***exp*** en une liste, on pourra utiliser la fonction ***split()***
     Exemple :
     Si exp = '3 5 *' alors exp.split() contient la liste ['3','5','*'].
   - On pourra utiliser une fonction (que l'on ne demande pas de coder ici) nommée ***is_float(e)*** qui retourne True si ***e*** est un nombre flottant et False sinon.
     Exemples :
     is_float('1.2') retourne True
     is_float('14') retourne True
     is_float('+') retourne False

Exercice 6 : Protocoles de routage RIP et OSPF (4 points)

On considère un réseau composé de 6 routeurs

On donne les tables de routages selon le protocole RIP des routeurs A, B et C.

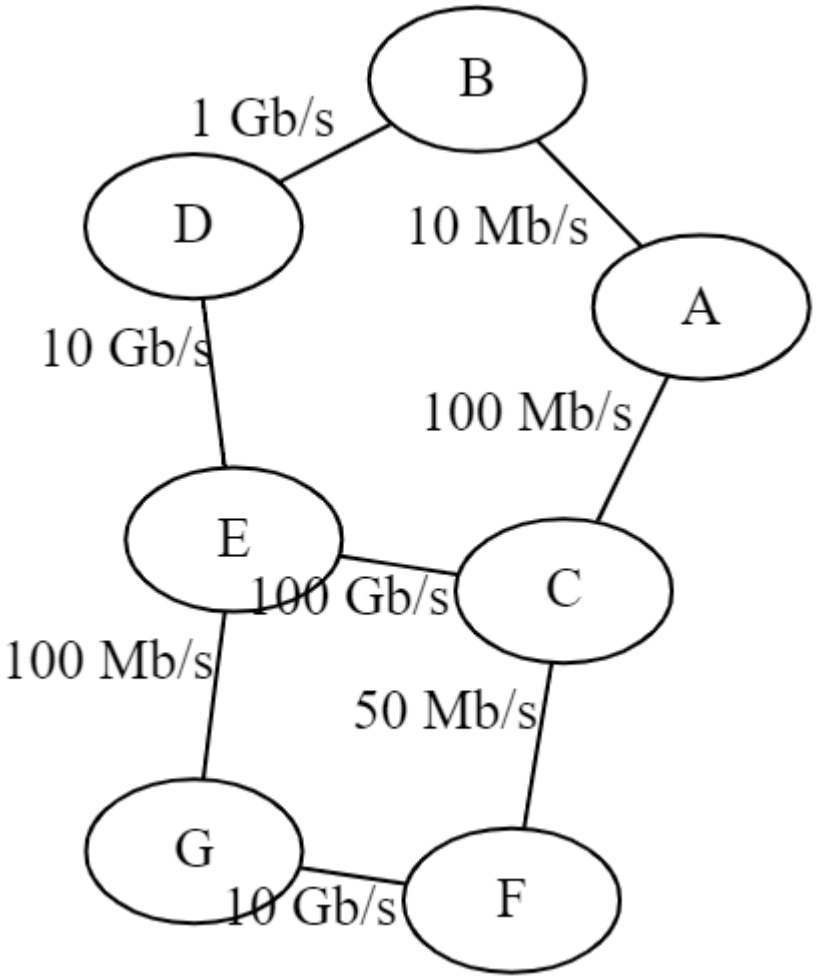
Table de routage du routeur A		
Destination	Routeur suivant	Distance
B	B	1
C	C	1
D	D	1
E	D	2
F	D	2

Table de routage du routeur B		
Destination	Routeur suivant	Distance
A	A	1
C	C	1
D	A	2
E	A	3
F	C	2

Table de routage du routeur C		
Destination	Routeur suivant	Distance
A	A	1
B	B	1
D	A	2
E	A	3
F	F	1

1) Représenter ce réseau sous la forme d'un graphe.
2) Donner la table de routage du routeur D.
3) Donner la route pour un message transmis entre le routeur C et E.

4) On considère le réseau de routeurs suivants :



Sur la liaison entre deux routeurs est indiquée la vitesse de transmission des données.

a) Indiquer le principe de fonctionnement du protocole de routage OSPF.

b) En utilisant le protocole OSPF, donner la route empruntée pour transmettre des données du routeur A au routeur G.
Justifiez votre réponse en utilisant le coût d'une liaison donnée par la formule :

$$\text{coût} = \frac{\mathbf{10^8}}{\boldsymbol{d}} \text{ où d est le débit de la liaison en bits/s.}$$

On donnera le coût de la route utilisée .

NOM :	Prénom :

Annexe de l'exercice 1 : (à rendre avec votre copie)

```
import bisect

class ArbreHuffman:
    def __init__(self,lettre,nbocc,g=None,d=None):
        self.lettre = lettre
        self.nbocc = nbocc
        self.gauche = g
        self.droite = d

    def est_feuille(self):
        return ____________________________________


    def __lt__(self, other):
        """
        Un arbre A est strictement inférieur à un arbre B
        si le nombre d'occurrences indiqué dans A est
        strictement supérieur à celui de B
        """
        return self.nbocc > other.nbocc

def parcours(arbre,chemin_en_cours,dico):
    if arbre is None:
        return
    if arbre.est_feuille():
        dico[arbre.lettre] = chemin_en_cours
    else:
        parcours(arbre.gauche,chemin_en_cours + [0],dico)
        ____________________________________________

def fusionne(gauche,droite):
    nbocc_total = ____________________________________
    return ArbreHuffman(None,nbocc_total,gauche,droite)
```

```
def compte_occurences(texte):
    """
    Renvoie un dictionnaire avec chaque caractère du texte comme clé
et le nombre d'apparitions
    de ce caractère dans le texte comme valeur
    >>> compte_occurences("AABCECA")
    {"A":3,"B":1,"C":2;"E":1}
    """
    occ = dict()
    for car in texte:
        if car not in occ :
            _____________
        occ[car] = occ[car] + 1
    return _______

def construit_liste_arbres(texte):
    """
    Renvoie une liste d'arbres de Huffmann, chacun réduit à une
feuille
    """
    dic_occurences = compte_occurences(texte)
    liste_arbres = []
    for lettre,occ in dic_occurences.items():
        liste_arbres.append(ArbreHuffman(lettre,occ))
    return liste_arbres

def codage_huffman(texte):
    """
    Codage de Huffman optimal à partir d'un texte
    >>> codage_huffman("AAAABBBBBCCD")
    {'A': [0, 0], 'C': [0, 1, 0], 'D': [0, 1 , 1], 'B': [1]}
    """
    liste_arbres = construit_liste_arbres(texte)
    #Tri par nombres d'occurences décroissants
    liste_arbres.sort()
    # Tant que tous les arbres n'ont pas été fusionnés
    while len(liste_arbres) > 1:
        # Les deux plus petits nombres d'occurrences sont à la fin
de la liste
        droite = liste_arbres.pop()
        gauche = liste_arbres.pop()
        nouvel_arbre = fusionne(gauche,droite)
        # Le module bisect permet d'insérer le nouvel
        # arbre dans la liste, de manière à ce que la liste reste
triée
        bisect.insort(liste_arbres,nouvel_arbre)
    # Il ne reste plus qu'un arbre dans la liste, c'est l'arbre de
Hufman
    arbre_huffman = liste_arbres.pop()
    #Parcours de l'arbre pour relever les codes
    dico = {}
    parcours(arbre_huffman,[],dico)
    return dico

# Script principal
with open("texte.txt") as f:
    texte = f.read()
print(codage_huffman(texte))
```

Exercice 1 : le codage de Huffman (8 points)

Ce sujet propose d'étudier une méthode de compression de données inventée par David Albert Huffman en 1952, qui permet de réduire la longueur du codage d'un alphabet et qui repose sur la création d'un arbre binaire.

On appelle *alphabet* l'ensemble des symboles (caractères) composant la donnée de départ à compresser. Dans la suite, nous utiliserons un alphabet composé seulement des 8 lettres A, B, C, D, E, F, G et H.

Partie A

1) On cherche à coder chaque lettre de cet alphabet par une séquence de chiffres binaires.
   a) Combien de bits sont nécessaires pour coder chacune des 8 lettres de l'alphabet.
   Pour coder les 8 lettres, il faut 3 bits car $2^3 = 8$.

   b) Quelle est la longueur en octets d'un message de 1 000 caractères construit sur cet alphabet ?

1000 caractères correspondent à $1000 \times 3 = 3000$ bits
1 octet = 8 bits
Donc $\frac{3000}{8} = 375$ octets.
Il faut 375 octets pour représenter les 1000 caractères.

2) Proposer un code de taille fixe pour chaque caractère de l'alphabet de 8 lettres.
   Proposition d'un code de taille fixe pour lequel chaque caractère est représenté par 3 bits.

Lettre	A	B	C	D	E	F	G	H
Code	000	001	010	011	100	101	110	111

3) On considère maintenant le codage suivant, la longueur du code de chaque caractère étant variable.

Lettre	A	B	C	D	E	F	G	H
Code	10	001	000	1100	01	1101	1110	1111

Ce type de code est dit *préfixe*, ce qui signifie qu'aucun code n'est le préfixe d'un autre (le code de A est 10 et aucun autre code ne commence par 10, le code de B est 001 et aucun autre code ne commence par 001).

Cette propriété permet de séparer les caractères de manière non ambiguë.

a) En utilisant la table précédente, donner le code du message : CACHE.
CACHE est codé par 00010000111101.

b) Quel est le message correspondant au code 001101100111001.
On peut décomposer le message comme suit : 001-10-1100-1110-01
Soit le mot : BADGE

4) Dans un texte, chacun des 8 caractères a un nombre d'apparitions différent. Cela est résumé dans le tableau suivant, construit à partir d'un texte de 1 000 caractères.

Lettre	A	B	C	D	E	F	G	H
Nombre	240	140	160	51	280	49	45	35

a) En utilisant le code de taille fixe proposé à la question 2., quelle est la longueur en bits du message contenant les 1 000 caractères énumérés dans le tableau précédent ?
La longueur serait de 1000×3 = 3000 bits.

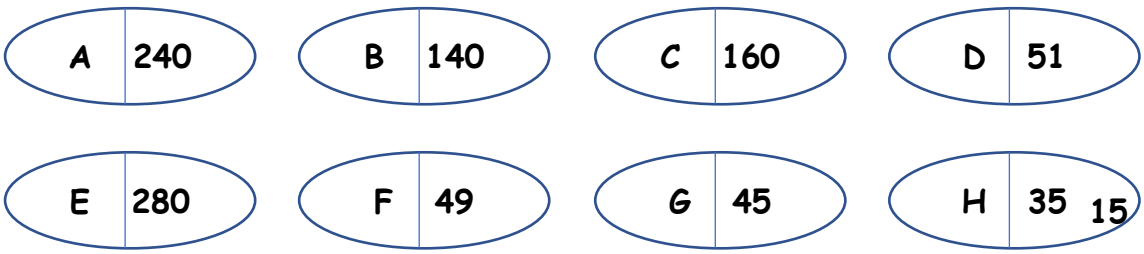
b) En utilisant le code de la question 3., quelle est la longueur du même message en bits ?

Lettre	A	B	C	D	E	F	G	H
Nombre	240	140	160	51	280	49	45	35
Longueur en bit du codage de la lettre	2	3	3	4	2	4	4	4

Avec le codage variable de la question 3, la longueur du message serait de : 240×2 + 140×3 + 160×3 + 51×4 + 280×2 + 49×4 + 45×4 + 35×4
Soit 2 660 bits.

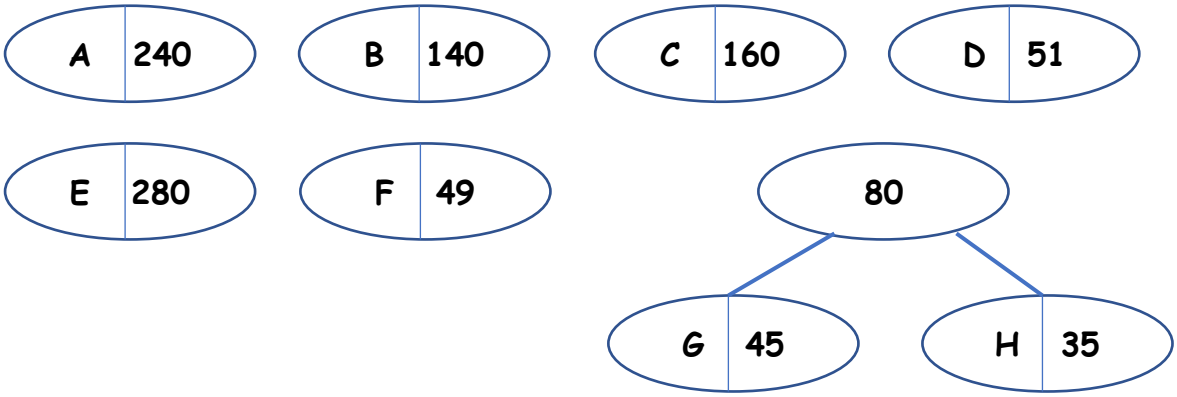
Partie B

1) L'objectif du codage de Huffman est de trouver le codage proposé en A 3., qui minimise la taille en nombre de bits du message codé en se basant sur le nombre d'apparitions de chaque caractère (un caractère qui apparaît souvent aura un code plus court).
Pour déterminer le code optimal, on considère 8 arbres, chacun réduit à une racine, contenant le symbole et son nombre d'apparitions.



Puis on fusionne les deux arbres contenant **les plus petits nombres d'apparitions** (valeur inscrite sur la racine), et on affecte à ce nouvel arbre la somme des nombres d'apparitions de ses deux sous-arbres. Lors de la fusion des deux arbres, le choix de mettre l'un ou l'autre à gauche n'a pas d'importance. Nous choisissons ici de mettre le plus fréquent à gauche (s'il y a un cas d'égalité, nous faisons un choix arbitraire).

,



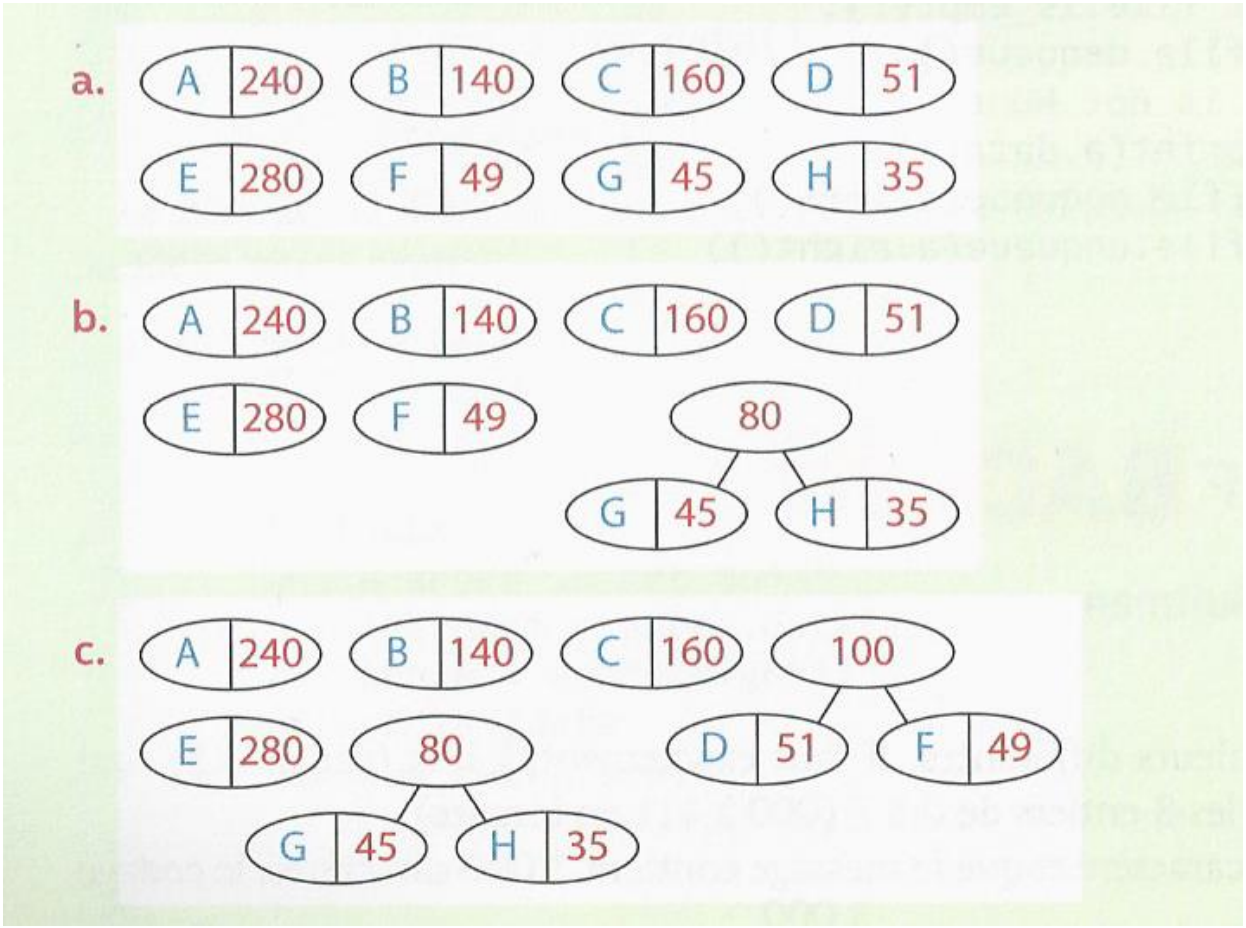
On recommence jusqu'à ce qu'il n'y ait plus qu'un seul arbre.

Combien d'étapes (combien de fusions d'arbres) sont nécessaires pour que cet algorithme se termine ?

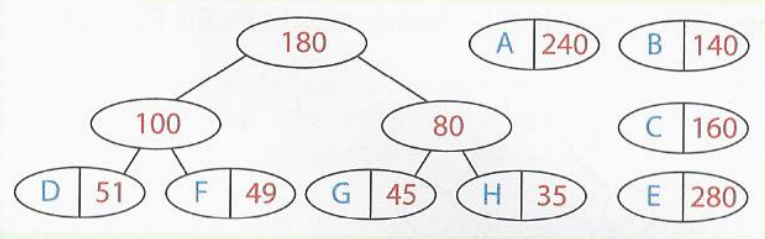
Il faudra 7 étapes puisqu'il y a à chaque étape un arbre de moins.

2) En suivant l'algorithme précédent, construire l'arbre de Huffman.

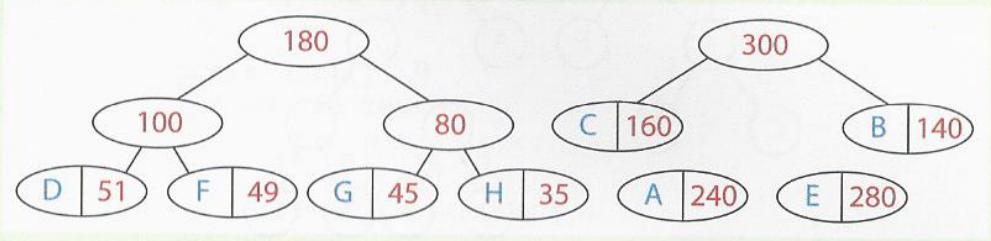
Voici les 7 étapes de construction de l'arbre de Huffman



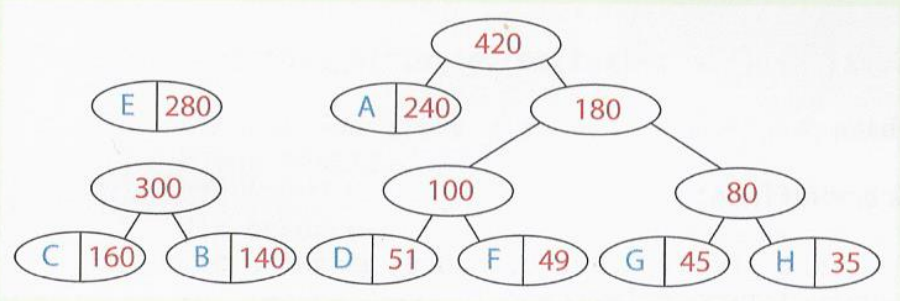
d.


180
100
80
D 51
F 49
G 45
H 35
A 240
B 140
C 160
E 280

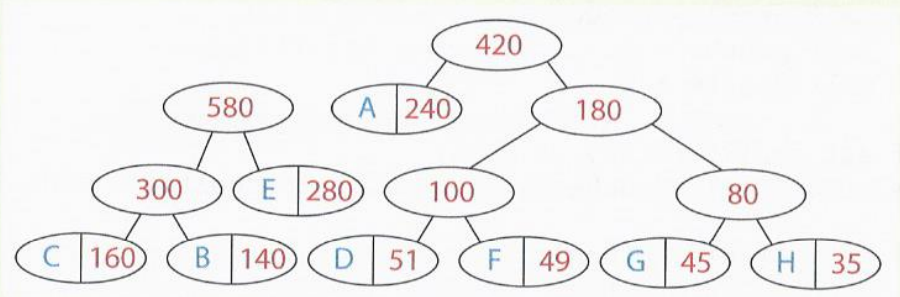
e.


180
100
80
D 51
F 49
G 45
H 35
300
C 160
B 140
A 240
E 280

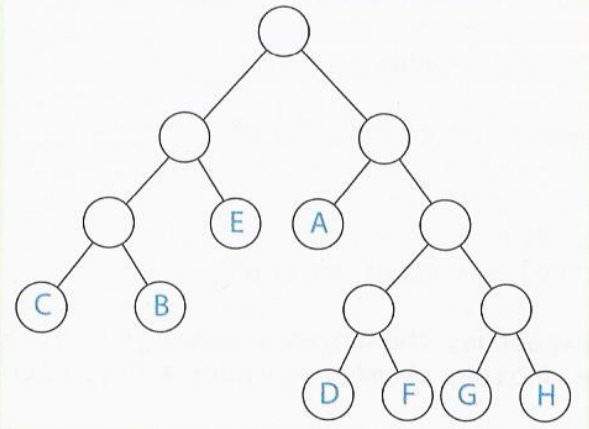
f.


420
E 280
A 240
180
300
100
80
C 160
B 140
D 51
F 49
G 45
H 35

g.

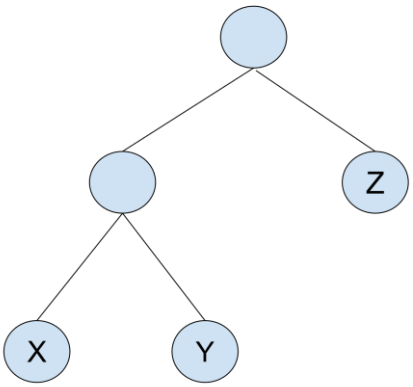

420
580
A 240
180
300
E 280
100
80
C 160
B 140
D 51
F 49
G 45
H 35

h.


E
A
C
B
D
F
G
H

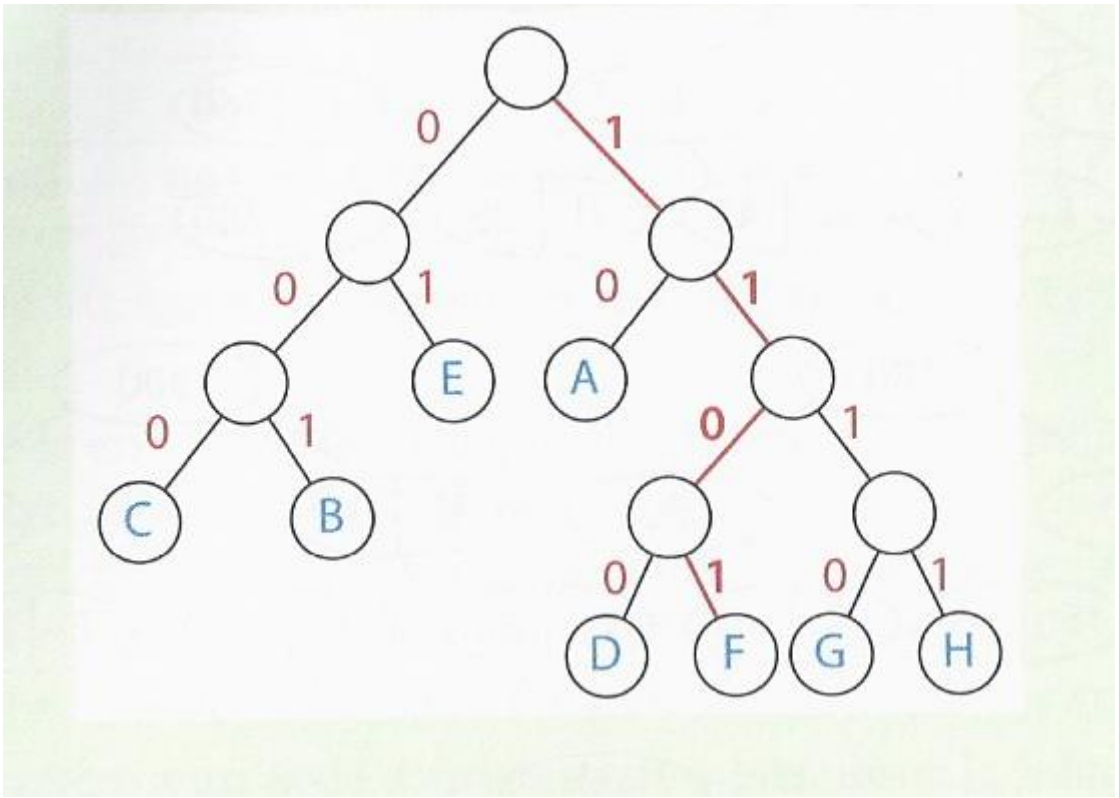
3) Le code à affecter à chaque lettre est déterminé par sa position dans l'arbre. Précisément, le code d'un symbole de l'alphabet décrit le chemin de la racine à la feuille qui le contient : un 0 indique qu'on descend par le fils gauche et un 1 indique qu'on descend par le fils droit.

Dans le cas de l'arbre ci-contre, le code de X est 00 (deux fois à gauche), le code de Y est 01, et celui de Z est 1.



Quel est le code de F ?

L'arbre obtenu avec les arêtes pondérées en binaire est le suivant :



Le code de F est donc 1101

Partie C

```
import bisect

class ArbreHuffman:
    def __init__(self,lettre,nbocc,g=None,d=None):
        self.lettre = lettre
        self.nbocc = nbocc
        self.gauche = g
        self.droite = d

    def est_feuille(self):
        return self.gauche = None and self.droite = None


    def __lt__(self, other):
        """
        Un arbre A est strictement inférieur à un arbre B
        si le nombre d'occurrences indiqué dans A est
        strictement supérieur à celui de B
        """
        return self.nbocc > other.nbocc

def parcours(arbre,chemin_en_cours,dico):
    if arbre is None:
        return
    if arbre.est_feuille():
        dico[arbre.lettre] = chemin_en_cours
    else:
        parcours(arbre.gauche,chemin_en_cours + [0],dico)
        parcours(arbre.gauche,chemin_en_cours + [1],dico)

def fusionne(gauche,droite):
    nbocc_total = gauche.nbocc + droite.nbocc
    return ArbreHuffman(None,nbocc_total,gauche,droite)
```

```
def compte_occurences(texte):
    """
    Renvoie un dictionnaire avec chaque caractère du texte comme clé
et le nombre d'apparitions
    de ce caractère dans le texte comme valeur
    >>> compte_occurences("AABCECA")
    {"A":3,"B":1,"C":2;"E":1}
    """
    occ = dict()
    for car in texte:
        if car not in occ :
            occ[car] = 0
        occ[car] = occ[car] + 1
    return occ

def construit_liste_arbres(texte):
    """
    Renvoie une liste d'arbres de Huffmann, chacun réduit à une
feuille
    """
    dic_occurences = compte_occurences(texte)
    liste_arbres = []
    for lettre,occ in dic_occurences.items():
        liste_arbres.append(ArbreHuffman(lettre,occ))
    return liste_arbres

def codage_huffman(texte):
    """
    Codage de Huffman optimal à partir d'un texte
    >>> codage_huffman("AAAABBBBBCCD")
    {'A': [0, 0], 'C': [0, 1, 0], 'D': [0, 1 , 1], 'B': [1]}
    """
    liste_arbres = construit_liste_arbres(texte)
    #Tri par nombres d'occurences décroissants
    liste_arbres.sort()
    # Tant que tous les arbres n'ont pas été fusionnés
    while len(liste_arbres) > 1:
        # Les deux plus petits nombres d'occurrences sont à la fin
de la liste
        droite = liste_arbres.pop()
        gauche = liste_arbres.pop()
        nouvel_arbre = fusionne(gauche,droite)
        # Le module bisect permet d'insérer le nouvel
        # arbre dans la liste, de manière à ce que la liste reste
triée
        bisect.insort(liste_arbres,nouvel_arbre)
    # Il ne reste plus qu'un arbre dans la liste, c'est l'arbre de
Hufman
    arbre_huffman = liste_arbres.pop()
    #Parcours de l'arbre pour relever les codes
    dico = {}
    parcours(arbre_huffman,[],dico)
    return dico

# Script principal
with open("texte.txt") as f:
    texte = f.read()
print(codage_huffman(texte))
```

Exercice 2 : Base de données dans un hôpital (6 points)

On veut créer une base de données relative à la gestion d'un hôpital qui contient les 3 tables suivantes.

Patients
id : entier
nom : texte
prenom : texte
genre : texte
Annee_naissance : entier

Ordonnances
code : entier
id_patient : entier
matricule_medecin: entier
Date_ordonnance : date
medicaments : texte

Medecins
matricule : entier
prenom : texte
nom : texte
specialite : texte
telephone : texte

On suppose que les dates sont données sous la forme jj-mm-aaaa.

Exemple : 14-11-2020 pour le 14 novembre 2020.

1) Madame Anne Wizeunid née en 2000 doit être enregistrée comme patiente. Donner la commande SQL correspondante.
   INSERT INTO Patients(nom,prenom,genre,annee_naissance) VALUES ("Wizeunid","Anne","F",2020)
   Ou bien
   INSERT INTO Patients VALUES (1,"Wizeunid","Anne","F",2020)
2) Le patient numéro 100 a changé de genre et est maintenant une femme. Donner la commande SQL modifiant en conséquence ses données.
   UPDATE Patients SET genre = "F" where id = 100
3) Par souci d'économie, la direction décide de se passer des médecins spécialisés en épidémiologie. Donner la commande SQL permettant de supprimer leur fiche.
   DELETE FROM Medecins WHERE specialite = "épidémiologie"
4) Donner la requête SQL permettant d'obtenir la liste des prénoms et noms des patients qui sont de sexe féminin triée dans l'ordre croissant des âges.
   SELECT prenom,nom FROM Patients WHERE genre = "F" ORDER BY annee_naissance
5) Donner la requête SQL qui donne la liste des patient(e)s ayant été examiné(e)s par un(e) psychiatre le 1er avril 2020.
   SELECT Patients.prenom,Patients.nom FROM Patients
   INNER JOIN Ordonnances ON id_patient = id
   INNER JOIN Medecins ON matricule_medecin = matricule
   WHERE specialite = "psychiatrie" and date_ordonnance = "01-04-2020".

6) Que fait la requête suivante :

```
SELECT m.prenom, m .nom
FROM Medecins as m
JOIN Ordonnances AS o
ON m.matricule = o.matricule_medecin
WHERE specialite = "ophtalmologie"
```

Cette requête liste les prénom et nom des médecins ophtalmologistes.

Exercice 3 : méthode du paysan pour la multiplication Récursivité (4 points)

La méthode du paysan russe est un très vieil algorithme de multiplication de deux nombres entiers déjà écrit, sous une forme légèrement différente, sur un papyrus égyptien rédigé autour de 1650 avant J.-C. Il s'agissait de la principale méthode de calcul en Europe avant l'introduction des chiffres arabes.

Les premiers ordinateurs l'ont utilisé avant que la multiplication ne soit directement intégrée dans le processeur sous forme de circuit électronique.

Sous une forme moderne, il peut être décrit ainsi :

Fonction **Multiplication (x,y) :**

```
p =0
TANT QUE x > 0 :
    Si x est impair :
        p = p + y
    x = x // 2
    y = y + y
retourner p
```

On rappelle que l'opérateur x // 2 donne le quotient entier de la division euclidienne de x par 2.

1) Appliquer cette fonction pour effectuer la multiplication de 105 par 253. Détailler les étapes en remplissant le tableau suivant :

x	y	p
105	253	0
52	506	253
26	1012	253
13	2024	253
6	4048	2277
3	8096	2277
1	16192	10373
0	32384	26565

2) On admet que cet algorithme repose sur les relations suivantes :

$$x \times y = \begin{cases} \mathbf{0}\ \boldsymbol{si}\ \boldsymbol{x}\ =\ \mathbf{0} \\ (\boldsymbol{x}//\mathbf{2}) * (\boldsymbol{y}+\boldsymbol{y})\ \boldsymbol{si\ x\ est\ pair} \\ (\boldsymbol{x}//\mathbf{2}) * (\boldsymbol{y}\ +\ \boldsymbol{y})\ +\ \boldsymbol{y\ si\ x\ est\ impair} \end{cases}$$

Proposer une fonction équivalente selon la méthode récursive.

```
def multiplication_rec(x,y):
    """
    Implémente une version récursive de l'algorithme de multiplication du
paysan
    @param x: un nombre
    @param y: u nombre
    @return: le produit de x par y
    >>> multplication_rec(12,10)
    120
    """
    if x == 0:
        return 0
    if x % 2 == 0:
        return multiplication_rec(x //2,y+y)
    print(y)
    return multiplication_rec(x//2,y+y)+y
```

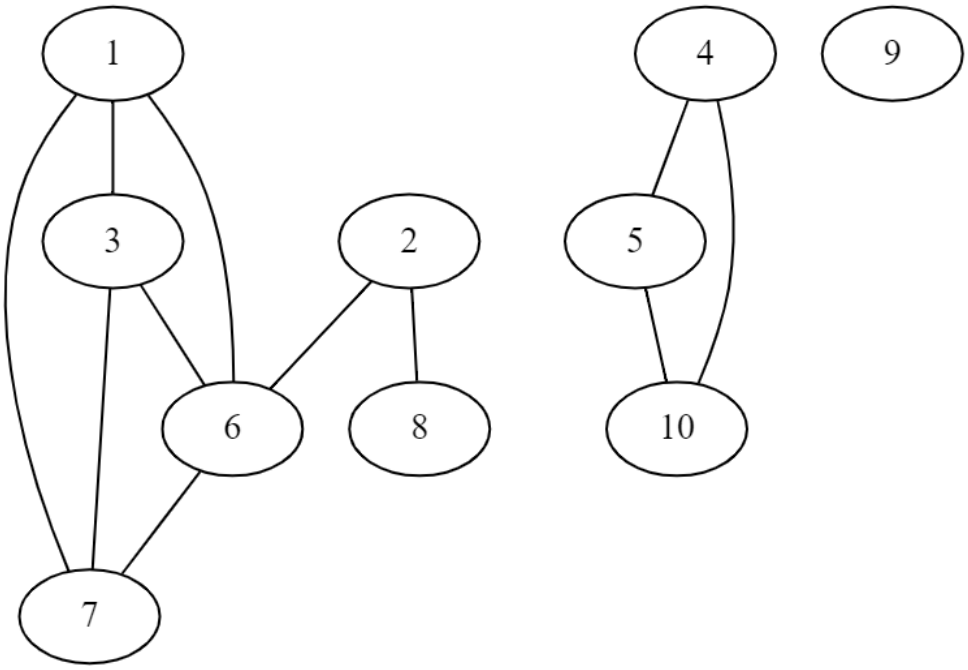
Exercice 4 : Graphes (4 points)

On considère un groupe de dix personnes présentes sur un réseau social, le tableau suivant indique les paires de personnes qui ont une relation d'amitié dans ce réseau social.

i	Ami de i
1	3, 6, 7
2	6, 8
3	1, 6, 7
4	5, 10
5	4, 10
6	1, 2, 3, 7
7	1, 3, 6
8	2
9	
10	4,5

1) Représenter cette situation par un graphe dans lequel une arête montre le lien d'amitié.
2) Ce graphe est-il connexe ? Si non, donner ses composantes connexes.
3) Donner les parcours en largeur et en profondeur de ce graphe à partir de la personne numéro 1.
4) L'adage « les amis de mes amis sont nos amis » est-il vérifié ? Si non, que faudrait-il faire pour qu'il le soit ?

1)



2) Ce graphe n'est pas connexe.
   Il est composé de 3 composantes connexes :
   - 1 – 2 – 3 – 6 – 7 – 8
   - 4 – 5 – 10
   - 9
3) Parcours en largeur à partir de 1 : 1 – 3 - 6 – 7 – 2 – 8
   Parcours en profondeur à partir de 1 : 1 – 3 - 7 – 6 – 2 – 8
   (Il y a plusieurs parcours possibles)
4) Non, l'adage « les amis de mes amis sont nos amis » n'est pas vérifié.
   Contre-exemple : 2 est l'ami de 6 et de 8, mais 6 et 8 ne sont pas amis.
   Pour que l'adage soit vérifié, il faudrait que la première composante connexe soit complète.
   C'est-à-dire qu'il existe une arête entre deux de ses sommets quelconques.

Exercice 5 : Evaluer une expression numérique avec une pile (4 points)

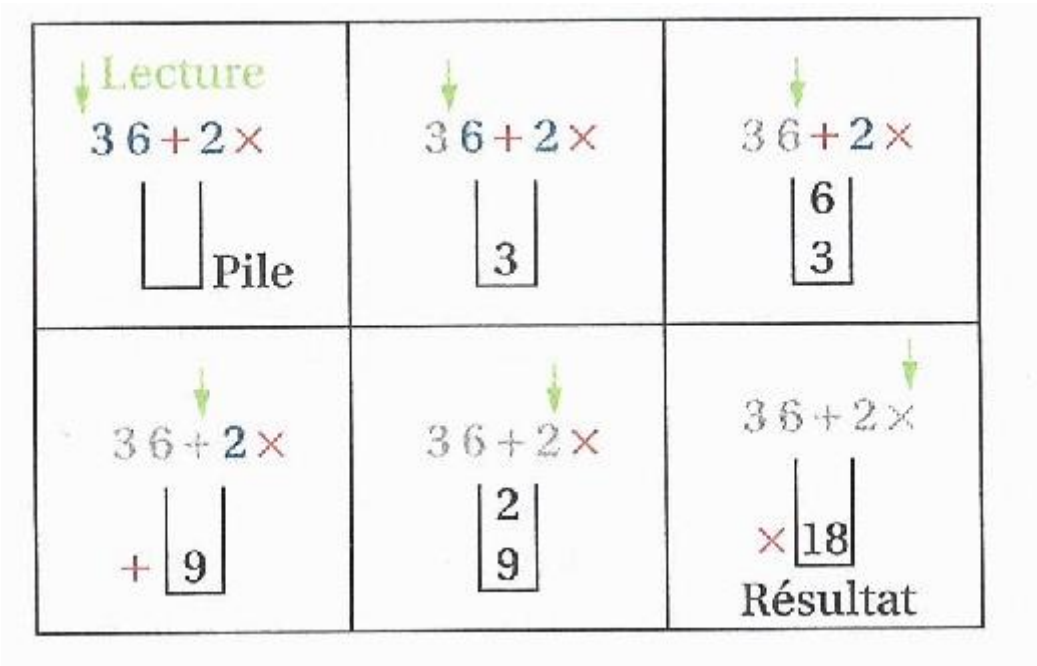
Il existe plusieurs façons de noter les expressions arithmétiques :

- la notation infixe (usuelle), utilisant des parenthèses, l'opérateur est indiqué **entre** les opérandes : (3 + 6) × 2 ;
- la notation postfixe, qui ne nécessite pas de parenthèse, l'opérateur est mentionné **après** les opérandes : 3 6 + 2 ×.

Une expression en notation postfixe peut facilement être évaluée à l'aide d'une **pile**, en même temps qu'elle est analysée. Pour cela, on lit l'expression de gauche à droite (voir figure) :

- si c'est un nombre, on l'empile ;
- si c'est un opérateur, on l'applique aux deux nombres qui sont au sommet de la pile et on empile le résultat

A la fin de la lecture de l'expression, la pile ne contient qu'un élément le résultat.



1) Coder les fonctions Python suivantes qui permettent de modéliser une pile par une liste avec les fonctions suivantes :

- ***creer_pile_vide()*** : crée et retourne une pile vide
- ***est_vide()*** : renvoie True si la pile est vide et False sinon
- ***empiler(P,e)*** : empile l'élément e au sommet de la pile P.
- ***depiler(P)*** : dépile et renvoie l'élément au sommet de la pile si la pile n'est pas vide et **None** si la pile est vide.

```
def creer_pile_vide():
    """
    retourne une pile vide (sans élément) représentée par une liste vide.
    @return: une liste vide
    >>> creer_pile_vide()
    []
    """
    return []

def est_vide(P):
    """
    Teste si la pile P passée en paramètre est vide
    @param P: la pile de type liste à tester
    @return: True si la pile est vide et False sinon
    >>> est_vide([])
    True
    >>> est_vide([1,2])
    False
    """
    return len(P) == 0

def empiler(P,e):
    """
    empile un élément  au sommet d'une pile
    @param P: la pile
    @param e: l'élément à empiler
    @return:
    """
    P.append(e)

def depiler(P):
    """
    Dépile et retourne l'élément au sommet d'une pile
    @param P: la pile
    @return: None si la pile P est vide et l'élement au sommet de la pile P
sinon
    >>> depiler([1,2,3])
    3
    """
    if est_vide(P):
        return None
    e = P.pop()
    return e
```

2) Coder la fonction ***calcul(P,op)*** qui :prend en paramètre une pile ***P*** et une opération ***op*** donnée sous la forme d'un caractère : +, -, *, /, dépile les deux éléments au sommet de la pile, leur applique l'opération et empile le résultat.

Exemple :

Calcul([14,11],'+') doit retourner 25.

```
def calcul(P,op):
    """
    Réalise le calcul d'une opération à deux opérandes à partir d'une pile
qui contient ces deux opérandes
    On commence par dépiler les deux opérandes du sommet de la pile
    On effectue le calcul à partir de ces deux opérandes et de l'opération
à effectuer.
    On empile le résultat calculé
    @param P: la pile contenant les deux opérandes de l'opération à
effectuer
    @param op: L'opération à effectuer : une chaine de caractères parmi
('+','*','/','-')
    @return: True si op est parmi ('+','*','/','-') et False sinon
    """
    operande1 = depiler(P)
    operande2 = depiler(P)
    if op in ('+','*','/','-'):
        if op == "+":
            empiler(P,operande2 + operande1)
        elif op == "*":
            empiler(P,operande2 * operande1)
        elif op == "/":
            empiler(P,operande2 / operande1)
        elif op == "-":
            empiler(P,operande2 - operande1)
        return True
    else:
        return False
```

3) Coder la fonction ***evaluation(exp)*** qui prend en paramètre une expression contenant des nombres et des symboles sous la forme d'une chaîne de caractères et évalue le résultat.

Cette fonction devra utiliser une pile, utiliser la fonction précédente ***calcul(P,op)***.

Exemples :

- evaluation ('3 5 *') doit retourner $3 \times 5 = 15$
- evaluation ('3 5 * 2 +') doit retourner $(3 \times 5) + 2 = 17$
- evaluation ('8 1 1 + /') doit retourner $8 / (1 + 1) = 4$

Indications :

- On suppose que l'expression postfixée ***exp*** passée en paramètre est bien « formée » : on ne demande pas de gérer les cas d'erreur où l'expression n'est pas conforme.
- Pour transformer la chaine de caractères ***exp*** en une liste, on pourra utiliser la fonction ***split()***
  Exemple :
  Si exp = '3 5 *' alors exp.split() contient la liste ['3','5','*'].
- On pourra utiliser une fonction (que l'on ne demande pas de coder ici) nommée ***is_float(e)*** qui retourne True si ***e*** est un nombre flottant et False sinon.
  Exemples :
  is_float('1.2') retourne True
  is_float('14') retourne True
  is_float('+') retourne False

```
def evaluation(exp):
    """
    Effectue le calcul d'une expression arithmétique postfixée représentée
par une chaine de caractères (exp)
    et retourne le resultat du calcul
    La suite des calculs est effectuée à l'aide d'une pile
    Remarque : on suppose que l'expression passée en paramètre est bien
formée.
    (on ne gère pas ici les erreurs)
    @param exp: l'expression postfixée
    @return: le résultat du calcul
    >>> evaluation("3 5 *")
    15.0
    >>> evaluation("3 5 * 2 +")
    17.0
    >>> evaluation("8 1 1 + /")
    4.0
    """
    liste = exp.split()
    pile = creer_pile_vide()
    for e in liste:
        if is_float(e):
            pile.append(float(e))
        else:
            calcul(pile,e)
    return depiler(pile)
```

Exercice 6 : Protocoles de routage RIP et OSPF (4 points)

On considère un réseau composé de 6 routeurs

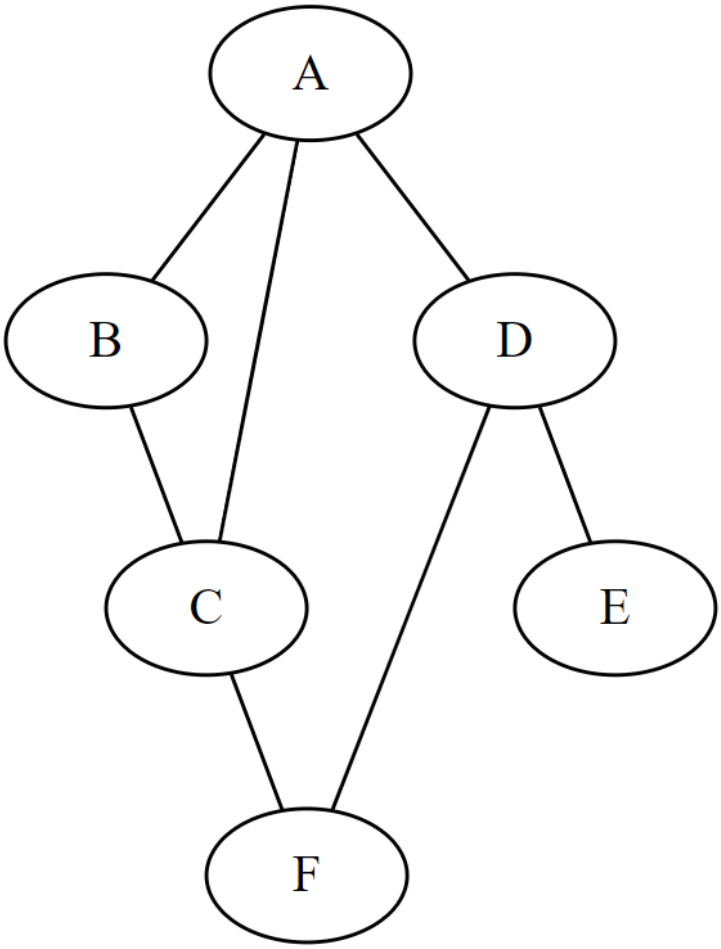
On donne les tables de routages selon le protocole RIP des routeurs A, B et C.

Table de routage du routeur A		
Destination	Routeur suivant	Distance
B	B	1
C	C	1
D	D	1
E	D	2
F	D	2

Table de routage du routeur B		
Destination	Routeur suivant	Distance
A	A	1
C	C	1
D	A	2
E	A	3
F	C	2

Table de routage du routeur C		
Destination	Routeur suivant	Distance
A	A	1
B	B	1
D	A	2
E	A	3
F	F	1

1) Représenter ce réseau sous la forme d'un graphe.



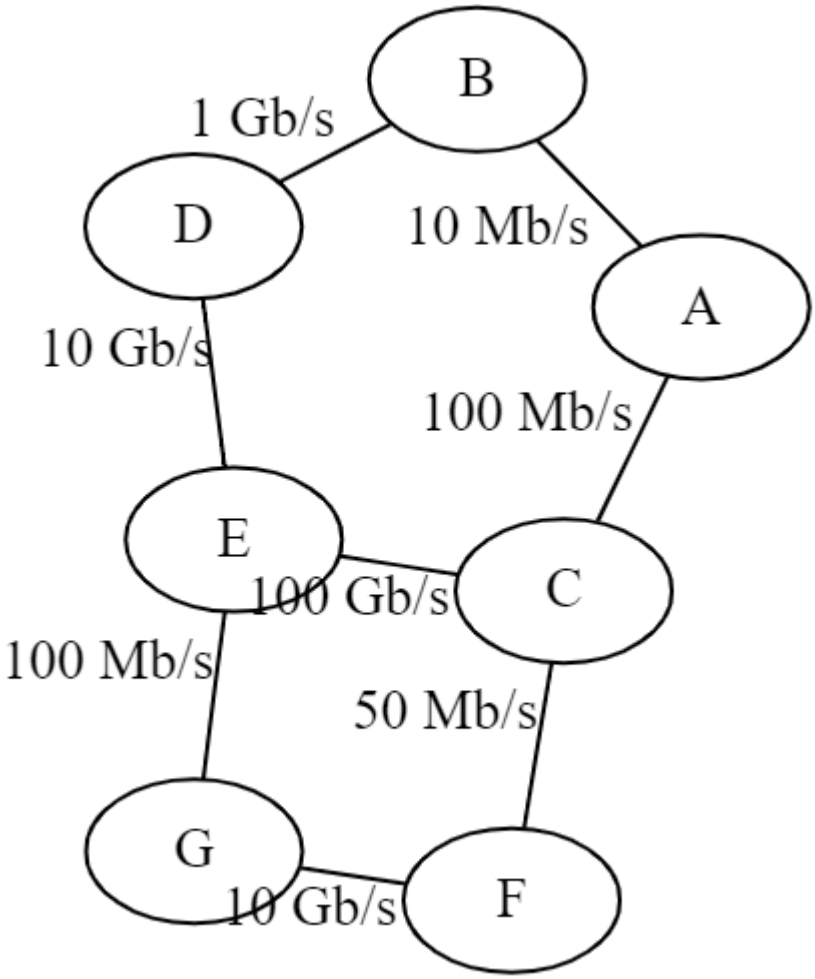
2) Donner la table de routage du routeur D.

Table de routage du routeur D		
Destination	Routeur suivant	Distance
A	A	1
B	A	2
C	A	2
E	E	1
F	F	1

3) Donner la route pour un message transmis entre le routeur C et E.

La route est C-A-D-E.

4) On considère le réseau de routeurs suivants :



Sur la liaison entre deux routeurs est indiquée la vitesse de transmission des données.

a) Indiquer le principe de fonctionnement du protocole de routage OSPF.

Dans OSPF, chaque routeur établit des relations d'adjacence avec ses voisins immédiats en envoyant des messages *hello* à intervalle régulier. Chaque routeur communique ensuite la liste des réseaux auxquels il est connecté par des messages *Link-state advertisements* (LSA) propagés de proche en proche à tous les routeurs du réseau. L'ensemble des LSA forme une base de données de l'état des liens *Link-State Database* (LSDB) pour chaque aire, qui est identique pour tous les routeurs participants dans cette aire. Chaque routeur utilise ensuite l'algorithme de Dijkstra, *Shortest Path First* (SPF) pour déterminer la route la plus rapide vers chacun des réseaux connus dans la LSDB.

b) En utilisant le protocole OSPF, donner la route empruntée pour transmettre des données du routeur A au routeur G.
Justifiez votre réponse en utilisant le coût d'une liaison donnée par la formule :

$$\text{coût} = \frac{10^8}{d}$$ où d est le débit de la liaison en bits/s.

On donnera le coût de la route utilisée .

Vitesse de transmission	coût
10 Mb/s	$\frac{10^8}{10 \times 10^6} = 10$
50 Mb/s	$\frac{10^8}{50 \times 10^6} = 2$
100 Mb/s	$\frac{10^8}{100 \times 10^6} = 1$
1 Gb/s	$\frac{10^8}{10^9} = 0{,}1$
10 Gb/s	$\frac{10^8}{10 \times 10^9} = 0{,}01$
100 Gb/s	$\frac{10^8}{100 \times 10^9} = 0{,}001$

Pour aller du routeur A au routeur G, la route la plus courte est celle qui passe par la liaison la plus rapide à 100 Gb/s.
Soit la route A – C – E – G dont le coût est : 1 + 0,001 + 1 = 2,001

Les autres routes possibles ont un coût supérieur :

- A – B – D – E – G : 10 + 0,1 + 0,01 + 1 = 11,11
- A – C – F – G : 1 + 2 + 0,01 = 3,01