

Nom et Prénom : _____ Groupe : _____

Examen INF TC1 – Algorithmique et structure de données

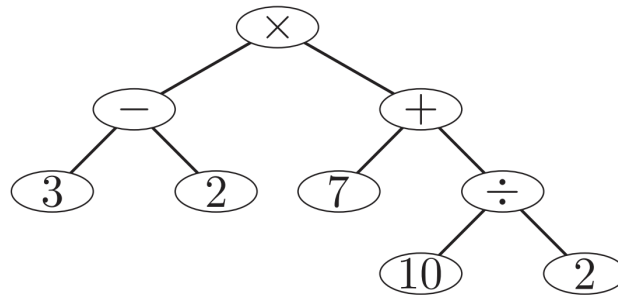
Les réponses seront faites directement dans les emplacements réservés à cet effet sur le sujet qui sera rendu à la fin de l'examen. Aucun document (support de cours, TD, etc...) n'est admis (sauf dictionnaires de langues pour les étrangers). Les ordinateurs et calculatrices sont également interdits.

Problème 1 : Tri d'une liste (4pts)

```
def tri(L) :  
    n = len(L)  
    for p in range(n-1) :  
        pmin = p  
        for j in range(p,n) :  
            if L[j] < L[pmin]:  
                pmin = j  
        L[pmin], L[p] = L[p], L[pmin]  
    return L
```

1. (2 points) Que réalise le tri ci-dessus appliqué à une liste d'entiers ? Donnez un exemple avec une liste de votre choix. Quelle est la complexité de ce tri dans le pire des cas ?

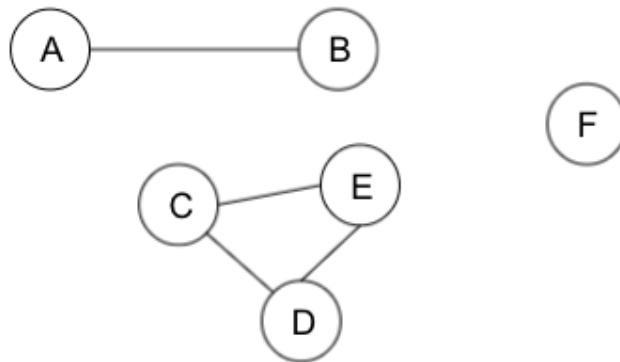
2. (2 points) Proposez une adaptation du code de tri ci-dessus afin de permettre le tri de tuples d'entiers, où le tri sera effectué par ordre croissant pour la première valeur du tuple. Dans le cas où deux tuples ont une première valeur identique, trier par ordre décroissant en fonction de la deuxième valeur.

Problème 2 : Validation d'un arbre syntaxique (10 pts)

Une expression arithmétique telle que $(3 - 2) * (7 + 10/2)$ peut être stockée sous forme d'arbre binaire (comme dans l'exemple ci-dessus). Cet arbre stocke les opérateurs (+, -, / et *) au sein de noeuds internes (qui ont toujours deux enfants). Les valeurs numériques de l'expression étant stockées dans les feuilles de l'arbre (noeuds qui n'ont pas d'enfant). Dans cet exercice nous considérons que les valeurs numériques sont des entiers positifs, et donc nous ne considérons pas les cas où - est unaire, comme -1 .

3. (2 points) Proposez une structure de données d'arbre binaire en Python qui permet de stocker une expression arithmétique telle que définie dans l'énoncé du problème. Illustrez son utilisation avec l'expression arithmétique fournie également dans l'énoncé. Attention cette structure sera utilisée dans les questions suivantes, donc lire toutes les questions de ce problème dans un premier temps.

4. (4 points) Proposez une méthode **valide** dont le paramètre d'entrée est un arbre syntaxique stocké au format que vous avez proposé dans la question précédente (question 3), et renvoie **True** ou **False** si il correspond à une expression arithmétique valide ou non. Autrement dit si les noeuds internes ont bien 2 enfants, les noeuds terminaux 0 enfant, les étiquettes des noeuds non-terminaux sont des opérateurs arithmétiques valides, et celles des noeuds terminaux sont des entiers (avec le test `isinstance(n, int)`).
5. (4 points) Proposez une méthode **evaluation** qui évalue l'arbre et renvoie la valeur numérique correspondante à l'équation. En cas de division par zéro, la méthode déclenchera une exception **ZeroDivisionError**.

Problème 3 : Composantes connexes de graphes (6 pts)

Considérons le graphe ci-dessus, qui est un graphe non orienté et non pondéré. Ce graphe sera dit connexe si tous les sommets sont reliés par un chemin. Dans l'exemple ci-dessus le graphe n'est pas connexe et possède 3 *composantes connexes* ($\{A, B\}$, $\{C, D, E\}$, et $\{F\}$), à savoir trois groupes de noeuds qui sont tous accessibles entre eux, au sein d'une même composante, mais pas accessibles entre composantes.

6. (3 points) Proposez une structure de données Python de graphe et une méthode de parcours en profondeur qui renvoie la liste des sommets accessibles depuis un sommet donné en paramètre (pour l'instant il est acceptable de ne pas parcourir toutes les composantes si le graphe n'est pas connexe).

7. (3 points) Proposez un algorithme en Python qui renvoie le nombre de composantes connexes d'un graphe. Nous vous suggérons d'utiliser la fonction introduite dans la question précédente.

CORRECTION:Proposition de correction

Question 1

Tri par selection, complexite n^2 dans le pire des cas

Question 2

```
def tri_tuples(L):
    n = len(L)
    for p in range(n-1):
        pmin = p
        for j in range(p+1, n):
            if L[j][0]<L[pmin][0] or (L[j][0]==L[pmin][0] and L[j][1]>L[pmin][1]):
                pmin = j
        L[pmin], L[p] = L[p], L[pmin]
    return L
```

Question 3

Arbre binaire implicite de type tas

```
exp = ['*', '-', '+', 3, 2, 7, '/', None, None, None, None, None, None, 10, 2]
```

Question 4

```
def valide(arbre):
    for i, noeud in enumerate(arbre):
        gauche = arbre.get(i*2+1, None)
        droite = arbre.get(i*2+2, None)
        if gauche!=None and droite!=None:
            if noeud not in ('+', '-', '/', '*'):
                return False
        elif gauche==droite==None:
            if not isinstance(noeud, int):
                return False
        else:
            return False
    return True
```

Question 5

```
def evaluation(arbre, i=0):
    noeud = arbre[i]
    if noeud == '+':
        return evaluation(arbre, i*2+1) + evaluation(arbre, i*2+2)
    elif noeud == '-':
        return evaluation(arbre, i*2+1) - evaluation(arbre, i*2+2)
    elif noeud == '/':
        diviseur = evaluation(arbre, i*2+2)
```

```
        if diviseur == 0:
            raise ZeroDivisionError()
        return evaluation(arbre, i*2+1) / diviseur
    elif noeud == '*':
        return evaluation(arbre, i*2+1) * evaluation(arbre, i*2+2)
    else:
        return noeud
```

Question 6

```
g = {'A': ['B'],
     'B': ['A'],
     'C': ['D', 'F'],
     'D': ['C', 'F'],
     'E': [],
     'F': ['C', 'D']}
```

```
def parcours_profondeur(graphe, noeud, visites=set(), resultat=[]):
    visites.add(noeud)
    resultat.append(noeud)
    for voisin in graphe[noeud]:
        if voisin not in visites:
            parcours_profondeur(graphe, voisin, visites, resultat)
    return resultat
```

Question 7

```
def compte_composantes_connexes(graphe):
    visites = set()
    compte = 0
    for noeud in graphe:
        if noeud not in visites:
            compte += 1
            parcours_profondeur(graphe, noeud, visites)
    return compte
```
