Nom : Prénom : Classe :

Devoir 03						
Thèmes abordés :	Date :	8 / 12 / 2020	Insuf.	Frag.	Satisf.	TrèsB
Structures de données abstraites						
Parcours des arbres binaires						
Récursivité						
Complexité d'un algorithme						

Numérique et Sciences Informatiques

Niveau terminale

Durée prévue : 1h50 minutes

Le sujet comporte 8 pages numérotées de 2 à 8. Les 8 pages sont à rendre. Vous répondrez directement sur ce document ainsi que sur une copie pour l'exercice 3.
Vous pouvez joindre des feuilles supplémentaires en annexe avec vos identifiants.

Aucun document personnel, livre ou notice n'est autorisé

Aucun matériel n'est autorisé :
- ni calculatrice, ni règle ou autre matériel.

Nécessaire :
- Une feuille de brouillon
- Copie supplémentaire pour l'exercice 3

<u>Exercices proposés :</u>

Exercice 1 : Structures d'ordonnancement de données : / 12 points

Exercice 2 : Arbre et labyrinthe : / 20 points

Exercice 3 : Les tours de Hanoï : / 10 points *(+ 5 pts bonus)*

Exercice 4 : Lecture et interprétation de code : tri de liste / 16 points

/ 60

Appréciation de la copie : lisibilité, orthographe, indications facilitant la compréhension du code … / 2 points

Rappel :

Tout début de raisonnement, réflexion sera pris en compte dans l'évaluation à condition que la rédaction soit argumentée avec un vocabulaire précis et adapté.

Exercice 1 – Structures d'ordonnancement de données / 12

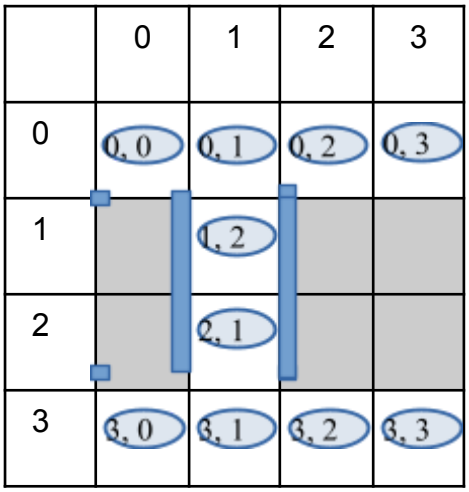
Chaque projet informatique nécessite de traiter des données ou des informations. Pour développer les algorithmes de traitement des données ou des informations, il est donc nécessaire d'avoir recours à des structures abstraites d'ordonnancement.
Dans chacun des cas suivants, quelle serait la structure la mieux adaptée ?

Situation	**Structure abstraite** d'ordonnancement et **justification**. *Attention à bien faire la distinction entre le type list de Python et les listes.*
Gestion des fiches de commandes dans une pizzeria	file : car le premier qui commande sera le premier servit.
Répertoire téléphonique *sans base de données* avec uniquement Nom et n° de téléphone	dictionnaire : le dictionnaire permet de structurer des données avec des clés ("Nom" et "n° de téléphone)
Saisie des numéros de dossards à l'arrivée d'une course	pile : car le premier arrivé sera le premier à sortir
Gestion des numéros de caisses entreposées sur un espace de stockage entre deux postes sur une ligne de montage.	pile : car le premier numéros est mis à la fin
Gestion des numéros de lots pour de la mise en rayon de denrées périssables. *(date limite de consommation)*	file : car la date de péremption la plus proche sera prise avant les autres
Gestion des noms des visiteurs dans la perspective de faire des classements alphabétiques.	liste : car on peut la trié plus facilement

Exercice 2 – Structures d'ordonnancement de données : les arbres / 20

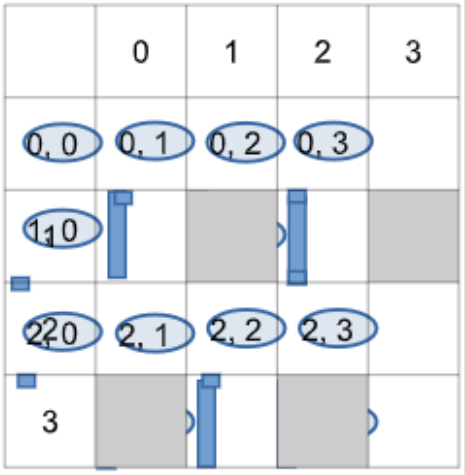
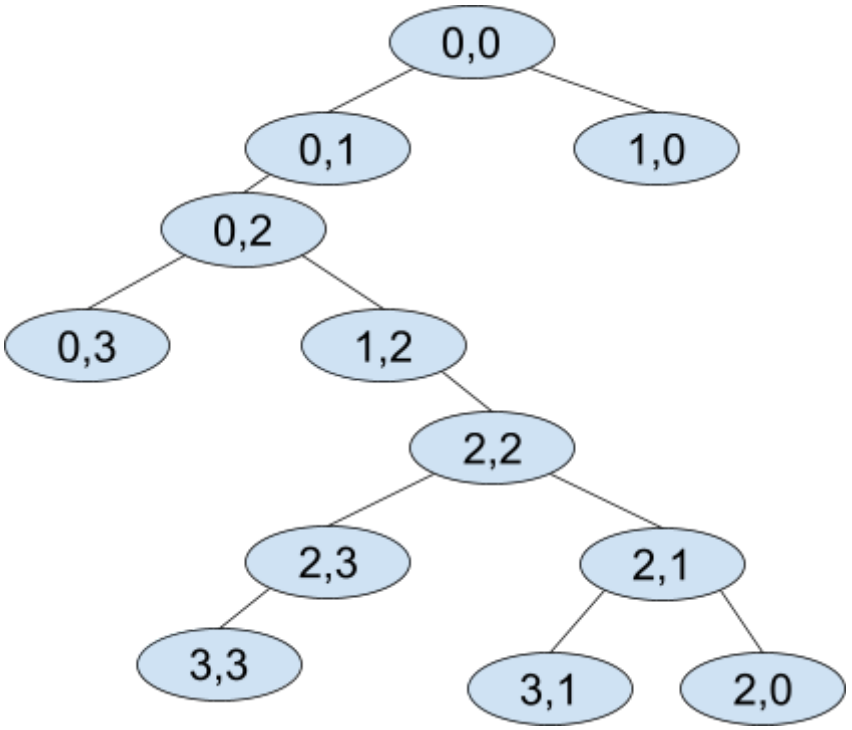
On souhaite mettre en place une structure d'ordonnancement de données qui permette de mémoriser les déplacements possibles dans un labyrinthe.

Chaque case est repérée par son numéro de ligne et de colonne.
On donne l'implémentation de cette structure avec les listes en Python

<table>
<tr><th>Plan du labyrinthe</th><th>Structuration abstraite des données</th></tr>
<tr><td>DÉPART : (0, 0)
 

ARRIVÉE : (3, 3)</td><td>Note : pour simplifier la structure du labyrinthe on considérera que chaque branche peut se prolonger par un embranchement de deux branches maximum, lesquelles peuvent se prolonger par un embranchement de deux branches au maximum ...</td></tr>
<tr><td>Implémentation :</td><td colspan="2">[(0,0), (0,1) , [(0,2), (0,3)], [(1,2), (2,1), (3,1) , [(3,0)] , [(3,2), (3,3)]]]</td></tr>
</table>

On donne le labyrinthe ci-dessous. Proposer la structuration abstraite des déplacements ainsi que l'implémentation en mémoire à l'aide de listes en Python

Plan du labyrinthe	Structuration abstraite des données (/2)
DÉPART : (0, 0)  ARRI Note : Le passage n'est pas possible depuis 1, 0 vers 2, 0	
Implémentation : (/2)	**[(0,0),[(0,1),(0,2),[(1,2),(2,2),[(2,1),[(2,0)],[(3,1)],[(2,3),(3,3)]]],[(0,3)]],[(1,0)]]**

Rédiger les instructions Python qui permettraient de coder les fonctions ci-dessous.

Note : Pour ajouter des cases au labyrinthe, on appellera la fonction ajouter_case à laquelle on passera entre autre en argument, le tuple (ligne, colonne) qui correspond à la case.

Fonction déjà codée	objectif
parcours (...)	*Affiche les chemins d'un labyrinthe.*
Fonctions à coder	
creer_labyrinthe (...)	Retourne une liste vide à la création du labyrinthe
ajouter_case (...)	Ajoute les coordonnées d'une case sous la forme d'un tuple
embranchement (...)	Implémente un embranchement de deux listes dans la liste du labyrinthe
longueur (...)	Détermine la longueur du plus long parcours
nombre_cases (...)	Retourne le nombre d'éléments (cases) qui constitue le labyrinthe

```
def affichage_parcours ( lab:list, decal:int=3 ):
    """
    Pour centrer l'affichage de l'arbre, je considère une largeur de 7 rangées verticales au
    maximum 3+1+3. J'affiche donc la première case du graphe avec un décalage de 3
```

```
    tabulations '\t'.
    """
    longueur = len(lab) # nbre de cases + nbre de branches
    n = 0 # compteur
    while n < longueur : # parcourir toute la branche
        if type(lab[n]) == tuple : # affichage d'une case
            print ( decal*'\t', lab[n] )  # '\t' -> tabulation
            n += 1
        else : # embranchement
            affichage_parcours ( lab[n], decal - 1 ) # branche de gauche
            affichage_parcours ( lab[n+1], decal + 1 ) # branche de droite
            break; # la boucle while est terminée avec les deux embranchements
def creer_labyrinthe():
    return []

def ajouter_case(l,c):


# ================================================================================
lab = [(0,0),(0,1),[(0,2),(0,3)],[(1,2),(2,1),(3,1),[(3,0)],[(3,2),(3,3)]]]
affichage_parcours ( lab )
```

Console

```
>>> %Run affichage_parcours.py
                        (0, 0)
                        (0, 1)
                (0, 2)
                (0, 3)
                                (1, 2)
                                (2, 1)
                                (3, 1)
                        (3, 0)
                                        (3, 2)
                                        (3, 3)
```

Pour informations :

Console

```
>>> case = (1,1)
>>> lab = list(case)
>>> lab
[1, 1]
>>> lab = [case]
>>> lab
[(1, 1)]
```

Note : l'affichage pourrait être perfectible, mais ce serait au détriment de la lisibilité du code ...

Fonctions à coder		
creer_labyrinthe (...)	Retourne une liste vide à la création du labyrinthe	/2
ajouter_case (...)	Ajoute les coordonnées d'une case sous la forme d'un tuple	/2
embranchement (...)	Implémente un embranchement de deux listes dans la liste du labyrinthe	/4
longueur (...)	Détermine la longueur du plus long parcours	/ 4
nombre_cases (...)	Retourne le nombre d'éléments (cases) qui constitue le labyrinthe	/4

Exercice 3 – Les tours de Hanoï / 10 (+5)

(inventé par le mathématicien français Edouard Lucas)

L'objectif des tours de Hanoï est de déplacer les disques de la tour A vers la tour C en respectant deux règles :
1 – on ne peut déplacer qu'un disque à la fois
2 – on ne peut déplacer un disque sur une autre tour que si le disque à déposer est plus petit que le disque déjà empilé. *(Règle déjà respectée dans la position initiale)*

La résolution pour trois disques donnerait :

<table>
<tr><td></td><td></td><td></td><td>Disque à déplacer</td><td>Déplacements :</td></tr>
<tr><td></td><td></td><td></td><td>1</td><td>Disque 1 sur position C</td></tr>
<tr><td></td><td></td><td></td><td>2</td><td>Disque 2 sur position B</td></tr>
<tr><td rowspan="2">A</td><td rowspan="2">B</td><td rowspan="2">C</td><td>1</td><td>Disque 1 sur position B</td></tr>
<tr><td>3</td><td>Disque 3 sur position C</td></tr>
<tr><td colspan="3" rowspan="3"></td><td>1</td><td>Disque 1 sur position A</td></tr>
<tr><td>2</td><td>Disque 2 sur position C</td></tr>
<tr><td>1</td><td>Disque 1 sur position C</td></tr>
</table>

Le nombre de disques est impair, dans ce cas, l'ordre de déplacement est le suivant :
- si le disque est pair, il suit la séquence de déplacements : A → B → C → A → B → C → …
- si le disque est impair, il suit suit la séquence : A → C → B → A → C → B → …

(répondre sur une copie simple supplémentaire. N'oubliez pas d'indiquer vos identifiants)

1 – Quelle **structure de données** permettrait d'associer une tour 'A', 'B' ou 'C' aux disques empilés sur celle-ci ? (/2)

2 – Comment serait alors **implémentée la condition initiale** avec cette structure, à savoir tous les disques empilés en ordre inverse de taille, du plus grand au plus petit sur la tour 'A'. (/2)

3 – Proposer la ou les instructions en Python qui permettraient d'obtenir cette structure dans sa configuration initiale. (/ 3)

4 – Quel serait l'affichage obtenu suite à l'appel de ces fonctions ? (/3)

```
def fonc_A ( n ):
  if n==1:
    print( n )
  else :
    fonc_A ( n-1 )
    print( n )

fonc_A ( 4 )
```

```
def fonc_B ( n=4 ):
  if n==1:
    print( n )
  else:
    print( n )
    fonc_B ( n-1 )

fonc_B ( 4 )
```

```
def fonc_C ( n=4 ):
  if n==1:
    print( n )
  else:
    fonc_C ( n-1 )
    print( n )
    fonc_C ( n-1 )

fonc_C ( 4 )
```

= POINTS BONUS = / 5

5 – Proposer **un algorithme** qui permettrait d'afficher l'ordre des déplacements à réaliser pour une tour de Hanoï de 3 disques comme indiqué dans la colonne [Déplacements].

Exercice 4 – Lecture et interprétation de code /16

Analyse d'un programme de tri de liste codé en Python.

```
 1- def trier_liste ( liste:list ) -> list :
 2-  nb_val = len(liste)
 3-  for n in range ( nb_val-1 ) :
 4-    val_min = liste[n]
 5-    index_min = n
 6-    for j in range ( n+1, nb_val ) :
 7-      if liste[j]<val_min :
 8-        val_min = liste[j]
 9-        index_min = j
10-    if val_min < liste[n] :
11-      liste[index_min], liste[n] = liste[n], liste[index_min]
12-  return liste
```

Soit une liste contenant les valeurs : [12, 24, 3, 2]

Valeur de n :	**0**	[12	, 24	, 3	, 2]
1) Donner la répartition des valeurs dans la liste pour chaque itération de la boucle for à la ligne n°3 (/ 2) *(toutes les lignes ne seront peut être pas nécessaires …)*	**1**				
	2				
	…				

2) Quel est l'ordre de grandeur de la complexité de cet algorithme ? (/ 4 : rep /2 + justif /2)
O(1) / O(n) / O(n2) ou O(n!) *Justifier votre réponse …*

3) Est-ce un algorithme de tri par sélection, bulle ou insertion ? (/ 4 : rep /2 + justif /2)
Quelles sont les lignes de code qui permettent de justifier votre choix et expliquer leur action dans le principe de classement de la liste.

4) Quelles lignes de code permettent de savoir si le tri se fera par ordre croissant ou décroissant ? (/ 2)

5) Proposer une autre version de ce code où le tri des valeurs se fera dans l'ordre inverse. Vous écrirez en bleu ou noir ce qui n'a pas été modifié et en vert vos modifications. (merci). (/ 4)