

BACCALAURÉAT BLANC n°2

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

Lycée Emmanuel Mounier, Angers

24 mai 2024

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 16 pages numérotées de 1 sur 16 à 16 sur 16.

Le sujet est composé de trois exercices indépendants.

EXERCICE 1 (7 points)

Cet exercice porte sur la notion de listes, la récursivité, les algorithmes gloutons et la programmation dynamique.

On considère un tableau carré de taille n , c'est-à-dire un tableau de n lignes et n colonnes. Les lignes et les colonnes sont numérotées de 0 à $n - 1$. La case en haut à gauche est repérée par $(0, 0)$ et la case en bas à droite par $(n - 1, n - 1)$. On supposera que les tableaux ne contiennent que des nombres entiers.

On donne les définitions suivantes qui serviront tout au long de l'exercice :

- Un *chemin* est une succession de cases allant de la case $(0, 0)$ à la case $(n - 1, n - 1)$, en n'autorisant que des déplacements case par case : soit vers la droite, soit vers le bas.
- La *somme* d'un chemin est égale à la somme des entiers situés sur ce chemin.
- La *longueur* d'un chemin est égale au nombre de cases de ce chemin.

Par exemple, pour le tableau T suivant :

4	1	3	3
2	0	2	1
7	1	2	1
2	1	3	1

- Un chemin est $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 2)$, $(2, 2)$, $(2, 3)$, $(3, 3)$ (en gras sur le tableau) ;
- La somme de ce chemin est 14 ;
- La longueur de ce chemin est 7 ;
- $(0, 0)$, $(0, 2)$, $(2, 2)$, $(3, 3)$ n'est pas un chemin.

L'objectif de cet exercice est de déterminer la somme maximale des chemins allant de la case $(0, 0)$ à la case $(n - 1, n - 1)$. Plusieurs approches sont proposées.

Partie A : Recherche par force brute

1. Rappeler en quoi consiste la recherche par « force brute » pour résoudre un problème.
2. Donner tous les chemins possibles pour un tableau carré de **taille 3**. Expliquer pourquoi ils ont tous la même taille.

On donne ci-dessous la fonction `liste_chemins` qui renvoie une liste de tous les chemins possibles pour un tableau carré de taille n .

```
1 def liste_chemins(n):
2     resultats = []
3     parcours(..., ..., [(0, 0)], resultats, n)
4     return resultats
5
6 def parcours(i, j, chemin, resultats, n):
7     if i == ... and j == ...: # Si on atteint la case en bas à droite
8         resultats.append(...)
9     else:
10         if i < n - 1: # Se déplacer vers le bas si possible
11             parcours(i + 1, j, chemin + [(..., ...)], resultats, n)
12
13         if j < n - 1: # Se déplacer vers la droite si possible
14             parcours(..., ..., ..., resultats, n)
```

La fonction `liste_chemins` fait appel à la fonction `parcours` qui explore récursivement tous les chemins à partir de la case (i, j) dans un tableau de taille n , en ajoutant chaque position visitée au chemin courant `chemin`, et en enregistrant le chemin dans la liste `resultats` lorsque ce dernier a atteint la case finale.

Par exemple, l'appel `liste_chemins(2)` renvoie la liste

```
[[ (0, 0), (0, 1), (1, 1) ],
 [ (0, 0), (1, 0), (1, 1) ]]
```

3. Expliquer pourquoi la fonction `parcours` peut être qualifiée de fonction récursive.
4. Recopier et compléter les fonctions `liste_chemins` et `parcours`.
5. Écrire une fonction `somme_chemin` qui prend en paramètre un chemin `chemin` et un tableau `tab` et qui renvoie la somme de ce chemin dans `tab`.

Exemple :

```
>>> tab = [[4, 1, 3, 3],
           [2, 0, 2, 1],
           [7, 1, 2, 1],
           [2, 1, 3, 1]]
>>> somme_chemin([(0,0), (0,1), (0,2), (1,2), (2,2), (2,3), (3, 3)], tab)
14
```

6. Écrire une fonction `somme_max_force_brute` qui prend en paramètre un tableau `tab` et qui renvoie la somme maximale parmi tous les chemins possibles en adoptant une stratégie force brute. *On pourra utiliser les fonctions `liste_chemins` et `somme_chemin` même si ces dernières n'ont pas été écrites.*
7. Quel problème caractéristique de la stratégie de force brute, mis en évidence par l'évaluation ci-dessous, rend rapidement cette méthode inapplicable pour résoudre notre problème ?

```
>>> len(liste_chemins(13))
2704156
```

Partie B : Recherche par méthode gloutonne

Lorsqu'un algorithme de force brute n'est pas utilisable, on peut essayer d'appliquer une stratégie gloutonne pour résoudre le problème.

8. Rappeler en quoi consiste un algorithme glouton et pourquoi cette stratégie évite le problème évoqué pour la méthode « force brute » à la question précédente.

La stratégie gloutonne pour calculer la somme maximale des chemins possibles est simple : à chaque étape, lorsqu'il y a deux choix possibles pour la case suivante, on choisit celle ayant la plus grande valeur. En cas d'égalité, on choisira toujours celle de droite. Si on est arrivé sur la dernière ligne ou la dernière colonne, on poursuit avec la seule case suivante possible.

Par exemple, pour le tableau T ci-dessous, à la première étape on est sur la case $(0, 0)$ et on choisit $(1, 0)$ comme case suivante car $2 > 1$. En poursuivant ainsi, cette méthode gloutonne donne le chemin représenté en gras dans le tableau T :

4	1	3	3
2	0	2	1
7	1	2	1
2	1	3	1

9. Recopier et compléter la fonction `solution_gloutonne` qui calcule et renvoie la somme du chemin obtenu selon la stratégie gloutonne dans un tableau `tab`.

Exemple :

```
>>> t = [[4, 1, 3, 3],
          [2, 0, 2, 1],
          [7, 1, 2, 1],
          [2, 1, 3, 1]]
>>> solution_gloutonne(t)
20
```

```
1 def solution_gloutonne(tab):
2     n = len(tab)
3     somme = tab[0][0]
4     i = 0 # indice de ligne
5     j = 0 # indice de colonne
6     # on n'a atteint aucun des bords droit ou bas,
7     # on choisit la meilleure option :
8     while i < n - 1 and j < ...:
9         a_droite = tab[i][j+1] # case de droite
10        en_bas = ... # case du dessous
11        if a_droite >= en_bas:
12            somme = ...
13            j = ...
14        else:
15            ...
16            ...
17        # on a atteint le bas, on va à droite :
18        if i == n - 1:
19            for k in range(..., ...):
20                somme = somme + tab[i][k]
21        # ou alors on a atteint le bord droit, on descend :
22        else:
23            for k in ...:
24                ...
25        return somme
```

10. Donner un tableau qui prouve que la stratégie gloutonne ne donne pas toujours la somme maximale des chemins. Expliquer.
11. Justifier que la longueur des chemins dans un tableau de taille n est toujours égale à $2n - 1$.
12. En déduire le nombre d'étapes nécessaires (qui est aussi le nombre total des itérations des boucles de notre fonction) pour construire la solution gloutonne selon l'algorithme proposé puis donner le coût d'exécution de la fonction `solution_gloutonne`.

Partie C : Recherche par programmation dynamique

13. Rappeler en quoi consiste la méthode algorithmique de « programmation dynamique » pour résoudre un problème.

On s'autorise dans cette dernière partie à parler de « chemin » même lorsque la case d'arrivée n'est pas $(n - 1, n - 1)$.

Pour résoudre notre problème en utilisant la programmation dynamique, on va créer et utiliser un tableau `s` de n lignes et n colonnes, où chaque élément `s[i][j]` contient la somme maximale pour un chemin allant de la case $(0, 0)$ à la case (i, j) .

Pour la question suivante, `tab` désigne le tableau :

4	1	3	3
2	0	2	1
7	1	2	1
2	1	3	1

Voici une version incomplète du tableau `s` associé au tableau `tab` :

$s =$	4	5	8	?
	6	6	10	12
	13	?	16	17
	15	16	?	20

Dans ce tableau `s`, on voit par exemple que la somme maximale des chemins de `tab` entre les cases $(0, 0)$ et $(1, 2)$ est égale à 10.

14. Recopier et compléter le tableau `s` associé au tableau `tab`.

15. En partant d'un tableau `s` vide, expliquer comment remplir la première ligne et la première colonne de `s` en utilisant les valeurs de `tab` et `s`.

16. Justifier que si i et j sont différents de 0, alors :

$$s[i][j] = \text{tab}[i][j] + \max(s[i-1][j], s[i][j-1]).$$

17. Recopier et compléter la fonction `somme_max_dynamique` ci-dessous qui prend en paramètre un tableau `tab`, et qui renvoie la somme maximale pour un chemin dans `tab` en utilisant la programmation dynamique. L'idée est de construire le tableau `s` puis de le remplir progressivement avec les sommes maximales pour arriver jusqu'à chaque case (i, j) .

```
1 def somme_max_dynamique(tab):
2     # création et initialisation du tableau
3     n = len(tab)
4     s = [[None] * n for i in range(n)]
5     s[0][0] = tab[...][...]
6     # remplissage du reste de la première ligne
7     for j in range(1, n):
8         s[0][j] = ...
9     # remplissage du reste de la première colonne
10    for i in ...:
11        s[i][0] = ...
12    # remplissage du reste du tableau
13    for i in ...:
14        for j in ...:
15            s[i][j] = ...
16    # renvoi de la somme maximale
17    return ...
```

18. Montrer que le coût d'exécution de cette fonction est quadratique en n pour un tableau carré de taille n . Pour cela, on pourra commencer par exprimer, en fonction de n , le nombre total d'affectations d'une valeur à une case du tableau `s` (lignes 5, 8, 11 et 15).

EXERCICE 2 (9 points)

Cet exercice porte sur les graphes, les algorithmes sur les graphes, les algorithmes de tri, les bases de données et les requêtes SQL.

Une équipe d'informaticiens développe une application permettant de proposer des visites culturelles. Cette application est basée sur un service de cartographie et permet aux utilisateurs d'obtenir des parcours de visites pour découvrir des points d'intérêts culturels dans les différentes régions françaises.

L'équipe de développeurs choisit d'utiliser la carte de France ci-dessous représentant les 12 régions métropolitaines (sans la Corse pour des raisons de simplification du problème).

Ils choisissent de modéliser la carte des régions par un graphe. Chaque région sera un sommet du graphe et, si deux régions sont limitrophes, il existera alors une arête entre ces deux sommets. Pour nommer les sommets ils utilisent les codes ci-dessous.



Code	Région
BRE	Bretagne
PLO	Pays de la Loire
NAQ	Nouvelle Aquitaine
OCC	Occitanie
PAC	Provence-Alpes-Côte d'Azur
ARA	Auvergne-Rhône-Alpes
BFC	Bourgogne Franche-Comté
CVL	Centre-Val de Loire
IDF	Île de France
GES	Grand Est
HDF	Hauts-de-France
NOR	Normandie

Dans la suite, on supposera que les graphes sont implémentés par des dictionnaires dont les clés sont les sommets et leurs valeurs sont des listes contenant les voisins du sommet considéré. Ainsi, le graphe des régions de France est mémorisé dans le dictionnaire suivant :

```
regions = {
  'BRE': ['NOR', 'PLO'],
  'PLO': ['BRE', 'CVL', 'NAQ', 'NOR'],
  'NAQ': ['ARA', 'CVL', 'OCC', 'PLO'],
  'OCC': ['ARA', 'NAQ', 'PAC'],
  'PAC': ['ARA', 'OCC'],
  'ARA': ['BFC', 'CVL', 'NAQ', 'OCC', 'PAC'],
  'BFC': ['ARA', 'CVL', 'GES', 'IDF'],
  'CVL': ['ARA', 'BFC', 'IDF', 'NAQ', 'NOR', 'PLO'],
  'IDF': ['BFC', 'CVL', 'GES', 'HDF', 'NOR'],
  'GES': ['BFC', 'HDF', 'IDF'],
  'HDF': ['GES', 'IDF', 'NOR'],
  'NOR': ['BRE', 'CVL', 'HDF', 'IDF', 'PLO']
}
```

Partie A : Création d'un parcours de visites

1. Représenter sur la copie le graphe correspondant à la carte "Régions de France" en utilisant les codes pour les sommets.

2. Écrire en Python une fonction `voisins` qui prend en paramètres un graphe `G` et un sommet `s` de ce graphe, et qui renvoie la liste des voisins du sommet `s`.

Exemple :

```
>>> voisins(regions, 'PLO')
['BRE', 'CVL', 'NAQ', 'NOR']
```

L'application dispose d'une interface graphique permettant aux utilisateurs de sélectionner la région qu'ils souhaitent visiter et de choisir un nombre pour étendre le parcours de visites aux régions limitrophes jusqu'à une certaine distance. Par exemple, si un utilisateur choisit la région « Bretagne » et le nombre 2, le parcours de visite proposé par l'application s'étendra jusqu'aux régions à une distance de 2 de la région « Bretagne ».

L'équipe a développé la fonction suivante pour implémenter cette fonctionnalité :

```
1 def sommets_a_distance(G, depart, n):
2     """Renvoie la liste de tous les sommets de G à une distance
3     inférieure ou égale à n du sommet de départ."""
4
5     visites = {depart: 0} # départ est à distance 0 de lui-même
6     file = [depart]
7     sommets_a_proximite = [...] # initialisation de la liste à renvoyer
8     while file != []:
9         s = file.pop(0)
10        for voisin in voisins(G, s):
11            if voisin not in visites:
12                distance_voisin = visites[s] + ...
13                if distance_voisin <= ...:
14                    file.append(voisin)
15                    visites[...] = ...
16                    sommets_a_proximite.append(voisin)
17    return sommets_a_proximite
```

La fonction `sommets_a_distance` prend en paramètres un graphe `G`, un sommet de départ `depart` et un entier positif `n`. Cette fonction renvoie une liste Python contenant tous les sommets situés à une distance inférieure ou égale à `n` du sommet `depart`. L'idée est de parcourir en largeur le graphe `G` à partir du sommet `depart` jusqu'à arriver à la distance souhaitée. Le dictionnaire `visites` permet de marquer les sommets visités en leur associant leur distance par rapport au sommet `depart`.

On rappelle que si `L` est une liste, `L.pop(0)` retire et renvoie le premier élément de `L`.

Ainsi, on a par exemple :

```
>>> sommets_a_distance(regions, 'BRE', 2)
['BRE', 'NOR', 'PLO', 'CVL', 'HDF', 'IDF', 'NAQ']
```

3. Justifier que la fonction `sommets_a_distance` met en œuvre un parcours en largeur du graphe `G`.
4. Recopier et compléter la fonction `sommets_a_distance`. *Il n'est pas utile de recopier la chaîne de documentation et les commentaires.*
5. Dans cette fonction, on utilise un dictionnaire `visites` pour mémoriser les sommets déjà visités. On aurait également pu utiliser un tableau (une liste Python) pour effectuer cette tâche. Expliquer pourquoi l'exécution de la ligne 11 de la fonction `sommets_a_distance` est plus efficace (en temps) si `visites` est un dictionnaire plutôt qu'un tableau.

Partie B : Coloration de la carte

Sur l'application, la carte des régions est affichée. Pour des raisons esthétiques, l'équipe de développement veut colorier la carte avec les contraintes suivantes :

- utiliser un minimum de couleurs ;
- faire en sorte que deux régions limitrophes (ayant une frontière commune) soient coloriées de deux couleurs différentes.

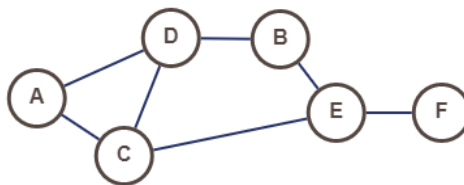
Une première approche pour colorer le graphe est de prendre ses sommets les uns après les autres afin de leur affecter une couleur, tout en veillant à ce que deux sommets adjacents n'aient jamais la même couleur : c'est *l'algorithme de coloration séquentielle*.

La fonction ci-dessous implémente cet algorithme, en utilisant la fonction `voisins` écrite à la question 2 :

```
1 def coloration(G) :
2     """Renvoie une coloration du graphe G.
3     Version séquentielle, limitée à 6 couleurs."""
4
5     liste_couleurs = ['Rouge', 'Bleu', 'Vert', 'Jaune', 'Noir', 'Blanc']
6     coloration_sommets = {s: None for s in G}
7     for s in G:
8         voisins_s = voisins(G, s)
9         couleurs_voisins_s = [coloration_sommets[v] for v in voisins_s]
10        k = 0
11        while liste_couleurs[k] in couleurs_voisins_s:
12            k = k + 1
13        coloration_sommets[s] = liste_couleurs[k]
14    return coloration_sommets
```

Code 1 – Coloration séquentielle d'un graphe

On considère le graphe G1 représenté ci-dessous :



Graphe G1

6. Donner les couleurs que va attribuer la fonction `coloration` à chaque sommet du graphe G1. Expliquer.

On considère que la boucle principale va envisager les sommets dans l'ordre lexicographique : tout d'abord A, puis B, puis C, etc.

7. On appelle la fonction `coloration` sur un nouveau graphe G2 comme indiqué ci-dessous. Représenter ce graphe sur la copie et montrer qu'il existe une solution avec moins de couleurs.

```
>>> G2 = {"A": ["C"], "B": ["D"], "C": ["A", "D"], "D": ["B", "C"]}
>>> coloration(G2)
{'A': 'Rouge', 'B': 'Rouge', 'C': 'Bleu', 'D': 'Vert'}
```


On voit que l'algorithme de coloration séquentielle n'est pas optimal. Dans la suite de cette partie, on va s'intéresser à l'*algorithme de Welsh et Powell* qui propose de colorer séquentiellement le graphe mais en visitant les sommets par ordre de degré décroissant. L'idée est que les sommets ayant beaucoup de voisins sont plus difficiles à colorer : il faut donc les colorer en premier.

On rappelle que le degré d'un sommet est le nombre d'arêtes issues de ce sommet, c'est-à-dire son nombre de voisins.

8. La fonction `tri_sommets` donnée ci-dessous prend en paramètre un graphe `G` et renvoie la liste des sommets de `G` triés par degrés **décroissants**.

Recopier et compléter la fonction. *Il n'est pas utile de recopier la chaîne de documentation.*

```
1 def tri_sommets(G):
2     """Renvoie la liste des sommets de G, triée par degré
3     décroissant"""
4
5     sommets = [sommet for sommet in G]
6     for i in range(1, ...):
7         s = sommets[i]
8         j = i
9         while j > 0 and ... > len(G[sommets[j-1]]):
10             sommets[j] = sommets[...]
11             j = j - 1
12             sommets[j] = ...
13     return sommets
```

9. Quel type de tri est implémenté dans la fonction `tri_sommets` et quel est son coût en temps ? Citer un exemple de tri plus efficace en temps.
10. Donner la liste renvoyée par l'appel `tri_sommets(regions)`, où `regions` est le dictionnaire représentant le graphe des régions de France défini au début de l'exercice.
On considèrera que l'itération sur les clés d'un dictionnaire (ligne 5) se fait dans l'ordre d'apparition des clés (dans notre cas : 'BRE' puis 'PLO', etc.).
11. Écrire une autre version de la fonction `tri_sommets` qui utilise la fonction `sorted` de Python dont on donne en *annexe* des extraits de la documentation officielle de Python.
12. Quelle unique ligne de l'algorithme de coloration séquentielle (Code 1) faut-il modifier, et comment, pour implémenter l'algorithme de coloration de Welsh et Powell ?
13. Donner alors la couleur attribuée à chaque région par l'algorithme de Welsh et Powell. Combien de couleurs sont nécessaires ?

Partie C : Base de données de l'application

Pour fonctionner, l'application s'appuie principalement sur les données de lieux culturels à visiter. Les développeurs ont choisi de stocker les informations sur ces lieux dans une base de données relationnelle.

14. Expliquer en quoi le choix d'utiliser un système de gestion de base de données (SGBD) est plus pertinent que l'utilisation d'un simple fichier texte pour stocker les informations.

La base de données possède au départ les deux relations suivantes :

- `region(id, nom)`
- `lieu(id, nom, commune, latitude, longitude, type, #id_region)`

Dans ce schéma :

- la clé primaire de chaque relation est définie par les attributs soulignés
- un attribut précédé de # indique qu'il s'agit d'une clé étrangère.

Les identifiants (attributs `id`) sont des entiers, la `latitude` et la `longitude` sont des flottants, tous les autres attributs sont des chaînes de caractères.

Voici des *extraits* des tables `region` et `lieu` de la base de données :

Extrait de la table <code>region</code>	
<code>id</code>	<code>nom</code>
1	Nouvelle Aquitaine
2	Bretagne
3	Occitanie
4	Pays de la Loire
5	Provence-Alpes-Côte d'Azur

Extrait de la table <code>lieu</code>						
<code>id</code>	<code>nom</code>	<code>commune</code>	<code>latitude</code>	<code>longitude</code>	<code>type</code>	<code>id_region</code>
1	Les Machines de l'Ile	Nantes	47.204054	-1.566842	espace d'animation	4
2	La Cité du Vin	Bordeaux	48.852968	2.349902	musée	1
3	Musée des Beaux-Arts	Rennes	48.109444	-1.675000	musée	2
4	Château de Chambord	Chambord	47.616111	1.517222	château	7
5	Musée des Beaux-Arts	Angers	47.468889	-0.554722	musée	4
6	Palais de la Découverte	Paris	48.866200	2.21083	musée scientifique	9

15. Expliquer pourquoi l'attribut `nom` ne peut pas être choisi comme clé primaire de la table `lieu`.
16. Écrire une requête SQL permettant de lister les noms, latitudes et longitudes de tous les châteaux présents dans la table `lieu`.
17. Écrire une requête SQL permettant de lister par ordre alphabétique les noms de tous les musées de la région Nouvelle Aquitaine.

Pour cause de travaux, le « Palais de la Découverte » s'installe temporairement dans un nouveau lieu nommé « Les Étincelles du Palais de la Découverte » situé également à Paris et ayant respectivement pour latitude et longitude les valeurs 48.84113 et 2.27966.

18. Écrire une requête SQL permettant d'effectuer les modifications nécessaires dans la table `lieu` pour le « Palais de la Découverte » (dont l'`id` est 6 dans la table).

La salle de cinéma sphérique « La Géode » située à Paris (latitude : 48.894444, longitude : 2.388611) va rouvrir ses portes en 2024.

19. Écrire la requête SQL permettant d'ajouter « La Géode » dans la table `lieu` avec l'identifiant 1253 et « cinéma » comme type.

20. Écrire une requête SQL permettant de lister les noms de tous les lieux situés dans la région Grand Est.

Les développeurs de l'application souhaitent mettre en place un système permettant aux utilisateurs de noter les différents lieux culturels enregistrés dans l'application. Chaque note, qui est un entier compris entre 0 et 5, doit être rattachée à un utilisateur et à un lieu. Pour cela, ils décident de créer deux tables supplémentaires appelées `utilisateur` et `note`.

21. Donner un schéma relationnel cohérent pour ces deux tables `utilisateur` et `note`. Vous indiquerez notamment quels attributs jouent le rôle de clé primaire et étrangère dans chacune des deux tables.

L'équipe de développement souhaite afficher sur l'interface graphique de l'application la note moyenne attribuée par les utilisateurs pour chaque lieu. Le serveur web utilisé supporte le langage Python, et une requête vers la base de données permet de récupérer l'ensemble des notes attribuées aux différents lieux sous la forme de la liste suivante :

```
notes = [  
    (2, "La Cité du Vin", 4),  
    (36, "Tour Eiffel", 4),  
    (17, "Château d'Angers", 5),  
    (1, "Les Machines de l'Ile", 5),  
    (2, "La Cité du Vin", 2),  
    (45, "Musée du Louvre", 5),  
    ...  
]
```

Les éléments de la liste `notes` sont des tuples de la forme (identifiant du lieu, nom du lieu, note).

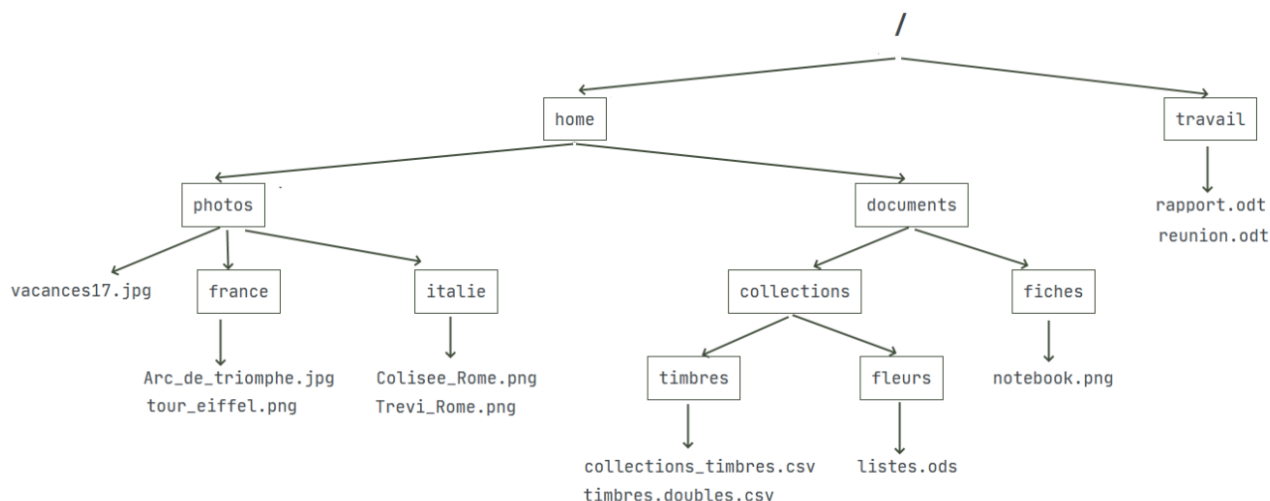
22. Comment, à partir de cette liste `notes`, les développeurs peuvent-ils procéder pour calculer la note moyenne de tous les lieux se trouvant dans cette liste. *Vous expliquerez vos idées et proposerez le code Python correspondant.*

EXERCICE 3 (4 points)

Cet exercice porte sur les systèmes d'exploitation, la gestion des processus et la programmation orientée objet.

Partie A

Dans un système d'exploitation de type UNIX, on considère l'arborescence des fichiers suivante dans laquelle on trouve des répertoires (dont les noms sont encadrés) et des fichiers (dont les noms ne sont pas encadrés).



On souhaite, grâce à l'utilisation du terminal de commande, explorer ou modifier les répertoires et fichiers présents.

On suppose que l'on se trouve actuellement à l'emplacement `/home/documents/collections`.

1. Parmi les quatre propositions suivantes, donner celle correspondant à l'affichage obtenu lors de l'utilisation de la commande `ls`.

Proposition 1 :

`timbres collections_timbres.csv timbres.doubles.csv fleurs listes.ods`

Proposition 2 : `timbres fleurs`

Proposition 3 : `collections`

Proposition 4 : `home travail`

2. Écrire la commande qui permet, à partir de cet emplacement, d'atteindre le répertoire `fiches`.

On suppose maintenant que l'on se trouve dans le répertoire `/travail`.

3. Écrire la commande qui permet de créer à cet emplacement un répertoire nommé `algorithmique`.
4. Écrire la commande qui permet, à partir de cet emplacement, de supprimer le fichier `rapport.odt`.

Partie B

Un système d'exploitation est chargé de la gestion des processus et des ressources. On rappelle qu'un processus est l'instance d'un programme en cours d'exécution. Il est identifié par un numéro unique appelé PID. L'ordonnanceur est la composante du système d'exploitation qui gère l'allocation du processeur entre les différents processus. Nous allons nous intéresser à l'algorithme

d'ordonnancement du tourniquet dont le fonctionnement est résumé ci-dessous :

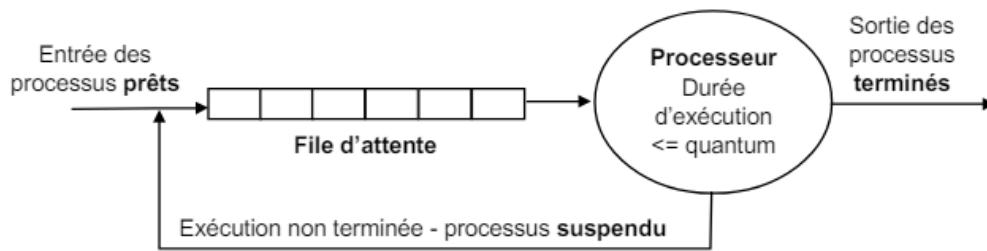


Schéma d'ordonnancement du tourniquet

- Les processus prêts à être exécutés sont placés dans une file d'attente selon leur ordre d'arrivée ;
- L'ordonnanceur alloue le processeur à chaque processus de la file d'attente un même nombre de cycles CPU, appelé quantum ;
- Si le processus n'est pas terminé au bout de ce temps, son exécution est suspendue et il est mis à la fin de la file d'attente ;
- Si le processus est terminé, il sort définitivement de la file d'attente.

On considère trois processus soumis à l'ordonnanceur **au même instant** pour lesquels on donne les informations ci-dessous :

PID	Durée (en cycles CPU)	Ordre d'arrivée
11	4	1
20	2	2
32	3	3

5. Si le quantum du tourniquet correspond à un seul cycle CPU, recopier et compléter la suite des PID des processus dans l'ordre de leur exécution : 11, 20, 32, 11,
6. Donner la composition de la suite des PID lorsque le quantum du tourniquet est de deux cycles CPU.
7. On considère deux processus P1 et P2, et deux ressources R1 et R2 mobilisables par P1 et P2. Décrire une situation qui conduit les deux processus P1 et P2 en situation d'interblocage.

Partie C

L'objectif de cette dernière partie est d'implémenter en langage Python l'algorithme du tourniquet.

Nous allons utiliser une liste pour simuler la file d'attente des processus et la classe `Processus` dont le constructeur est donné ci-dessous :

```
1 class Processus:
2     def __init__(self, pid, duree):
3         self.pid = pid
4         self.duree = duree
5         # Le nombre de cycles qui restent à faire :
6         self.reste_a_faire = duree
7         self.etat = "Prêt"
```

Les états possibles d'un processus sont : "Prêt", "En cours d'exécution", "Suspendu" et "Terminé".

8. Recopier et compléter l'instruction Python suivante permettant de créer la liste d'attente initiale des processus donnés dans le tableau précédent (le processus PID 11 est à l'indice 0 de la liste d'attente) :

```
liste_attente = [Processus(...,...), ..., ...]
```

9. Recopier (sans les commentaires) et compléter les trois méthodes suivantes de la classe `Processus` :

```
def execute_un_cycle(self):
    """Met à jour le reste à faire après l'exécution d'un
    cycle."""
    ...

def change_etat(self, nouvel_etat):
    """Change l'état du processus avec la valeur passée en
    paramètre."""
    ...

def est_termine(self):
    """Renvoie True si le processus est terminé, False sinon,
    en se basant sur le reste à faire."""
    ...
```

10. La fonction `tourniquet` ci-dessous implémente l'algorithme décrit dans l'exercice. Elle prend en paramètres une liste d'objets de la classe `Processus` donnés par ordre d'arrivée et un nombre entier positif `quantum` correspondant au quantum. La fonction renvoie la liste des PID dans l'ordre de leur exécution par le processeur.

Recopier et compléter sur la copie les lignes 8, 9, 12 et 16 avec le code manquant.

```
1 def tourniquet(liste_attente, quantum):
2     ordre_execution = []
3     while liste_attente != []:
4         # On extrait le premier processus
5         processus = liste_attente.pop(0)
6         processus.change_etat("En cours d'exécution")
7         compteur_tourniquet = 0
8         while ... and ...:
9             ordre_execution.append(...)
10            processus.execute_un_cycle()
11            compteur_tourniquet = compteur_tourniquet + 1
12        if ...:
13            processus.change_etat("Suspendu")
14            liste_attente.append(processus)
15        else:
16            processus.change_etat(...)
17    return ordre_execution
```

On rappelle que si `L` est une liste, `L.pop(0)` retire et renvoie le premier élément de `L`.

ANNEXE – Exercice 2 – Question 11

Extraits de la documentation officielle de Python

sorted(iterable, /, *, key=None, reverse=False)

Renvoie une nouvelle liste triée depuis les éléments d'*iterable*.

Possède deux arguments optionnels qui doivent être spécifiés par arguments nommés.

key spécifie une fonction à un argument utilisée pour extraire une clé de comparaison de chaque élément de l'itérable (par exemple, `key=str.lower`). La valeur par défaut est `None` (compare les éléments directement).

reverse est une valeur booléenne. Si elle est `True`, la liste d'éléments est triée comme si toutes les comparaisons étaient inversées.

Pour des exemples de tris et un bref tutoriel, consultez [Sorting Techniques](#).

Sorting Techniques

Auteur: Andrew Dalke et Raymond Hettinger

Les listes Python ont une méthode native [list.sort\(\)](#) qui modifie les listes elles-mêmes. Il y a également une fonction native [sorted\(\)](#) qui construit une nouvelle liste triée depuis un itérable.

Dans ce document, nous explorons différentes techniques pour trier les données en Python.

Les bases du tri

Un tri ascendant simple est très facile : il suffit d'appeler la fonction [sorted\(\)](#). Elle renvoie une nouvelle liste triée :

```
>>> sorted([5, 2, 3, 1, 4])  
[1, 2, 3, 4, 5]
```

Vous pouvez aussi utiliser la méthode [list.sort\(\)](#). Elle modifie la liste elle-même (et renvoie `None` pour éviter les confusions). Habituellement, cette méthode est moins pratique que la fonction [sorted\(\)](#) -- mais si vous n'avez pas besoin de la liste originale, cette technique est légèrement plus efficace.

```
>>> a = [5, 2, 3, 1, 4]  
>>> a.sort()  
>>> a  
[1, 2, 3, 4, 5]
```

Une autre différence est que la méthode [list.sort\(\)](#) est seulement définie pour les listes. Au contraire, la fonction [sorted\(\)](#) accepte n'importe quel itérable.

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})  
[1, 2, 3, 4, 5]
```

Fonctions clef

`list.sort()` et `sorted()` ont un paramètre `key` afin de spécifier une fonction (ou autre callable) qui peut être appelée sur chaque élément de la liste avant d'effectuer des comparaisons.

Par exemple, voici une comparaison de texte insensible à la casse :

```
>>> sorted("This is a test string from Andrew".split(), key=str.casefold) >>>
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

La valeur du paramètre `key` doit être une fonction (ou autre callable) qui prend un seul argument et renvoie une clef à utiliser à des fins de tri. Cette technique est rapide car la fonction clef est appelée exactement une seule fois pour chaque enregistrement en entrée.

Un usage fréquent est de faire un tri sur des objets complexes en utilisant les indices des objets en tant que clef. Par exemple :

```
>>> student_tuples = [ >>>
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2]) # tri par âge
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```