Corrigé Bac Blanc novembre 2025

Exercice 1

1. Pour la classe Chemin,
   - `itineraire`, `longueur`, `largeur` et `grille` sont des attibuts des instances de la classe.
   - `remplir_grille` et le constructeur `__init__` en sont des méthodes.
2.
   - `a` vaut 4 car `DDBDBBDDDDB` contient 4 B.
   - `b` vaut 7 car ‘`DDBDBBDDDDB` contient 7 D.
3.

```
def remplir_grille(self):
    i, j = 0, 0
    self.grille[0][0] = 'S'
    for direction in self.itineraire:
        if direction == 'D':
            j = j + 1
        elif direction == 'B':
            i = i + 1
        self.grille[i][j] = '*'
    self.grille[self.largeur][self.longueur] = 'E'
```

4.

```
def get_dimensions(self):
    return (self.longueur, self.largeur)
```

5.

```
def tracer_chemin(self):
    for ligne in self.grille:
        ligne_affichee = ''
        for cellule in ligne:
            ligne_affichee += ' ' if cellule == '.' else cellule
        print(ligne_affichee)
```

6.

```
def itineraire_aleatoire(m, n):
    itineraire = ''
    i, j = 0, 0
    while i != m and j != n:
        dir = random.choice(['D', 'B'])
        itineraire = itineraire + dir
        if dir == 'D':
            j = j + 1
        else :
```

```
            i = i + 1
    if i == m:
        itineraire = itineraire + 'D'*(n-j)
    if j == n: # un else aurait suffit du fait de la condition du while
        itineraire = itineraire + 'B'*(m-i)
    return itineraire
```

7.

$N(1,n)$ correspond aux nombres de chemins avec un déplacement vers le bas et n déplacements vers la droite, il y a donc plusieurs chemins possibles comme le permet de constater les appels à `itineraire_aleatoire(1, 5)` par exemple pour `n = 5`.

C'est $N(0,n) = N(m,0) = 1$ la réponse attendue.

Compte tenu de l'erreur dans l'ennoncé, cette question est comptée en point bonus.

8.

Pour arriver à la case de coordonnée (m,n), il faut passer par la case (m - 1,n) en arrivant du haut ou par la case $(m,n-1)$ en arrivant de la gauche. Comme ce sont les seules possibiltés, nous avons bien le nombre de chemins $N(m,n)$ qui correspond à la somme des nombres de chemins $N(m-1,n)$ et $N(m,n-1)$ par disjonction des cas.

9.

Nous allons écrire une fonction **récursive**, cas le plus adapté ici.

```
def nombre_chemins(m: int, n: int) -> int:
    assert n >= 0 and m >= 0, "les arguments sont des entiers positifs"
    if n == 0 or m == 0:
        return 1
    else:
        return nombre_chemins(m - 1, n) + nombre_chemins(m, n - 1)
```

Exercice 2

1.

```
def echange(tab: list, i: int, j: int):
    tab[i], tab[j] = tab[j], tab[i]
```

ou avec une variable temp comme dans d'autres langages de plus bas niveau.

```
def echange(tab, i, j):
    tmp = tab[i]
    tab[i] = tab[j]
    tab[j] = tmp
```

2.

On applique la méthode décrite dans l'ennoncé.

```
def triStooge(tab, i, j):
    if tab[i] > tab[j]:
        echange(tab, i, j)
    if (j - i) > 1:
        k = (j - i + 1)//3
        triStooge(tab, i, j-k)
```

```
triStooge(tab, i+k, j)
triStooge(tab, i, j-k)
```

3.

L'algorithme est récursif. La fonction triStooge s'appelle elle-même 3 fois. (et aussi, la condition d'arrêt sera atteinte en diminuant strictement l'écart entre `i` et `j`).

4.

Lors du premier appel, `k`va être calculé à partir de `i` et `j`

- `i = 0`
- `j = 5`

Alors `k = (5 - 0 + 1) // 3 = 6 // 3 = 2`

5.

La fonction appelée que l'on ne compte pas effectue 3 appels, chacun de ces appels en effectue 3, et ces 9 appels en effectuent aussi 3 chacun.

Nous avons donc `3 + 9 + 27 = 39` appels.

Cela correspond au nombre de case de la figure.

6.

- case 1 : triStooge(A, 1, 3)
- case 2 : triStooge(A, 2, 3)
- case 3 : triStooge(A, 0, 3)

7.

Ce tableau permet de constater les évolutions de la liste `A` avant et après les appels suivants : appel global et du premier niveau de récursivité.

Dans le tableau ci-dessous la première ligne correspond à l'appel global et les 3 lignes suivantes aux 3 appels du premier niveau récursif.

Attention : la valeur de `A` est bien `[5,6,4,2]` ce qui correspond au sous-arbre complètement à gauche de la question 4. Et non `[5,4,6,2]` comme indiqué dans la phrase de la question.

Appel	Valeur de A vant l'appel	Valeur de `A` après l'appel
`triStooge(A,0,3)`	`[5,6,4,2]`	`[2,4,5,6]` **résultat trié**
`triStooge(A,0,2)`	`[2,6,4,5]`	`[2,4,6,5]`
`triStooge(A,1,3)`	`[2,4,6,5]`	`[2,4,5,6]`
`triStooge(A,0,2)`	`[2,4,5,6]`	`[2,4,5,6]`

8.

Avec les 3 appels récursifs à chaque niveau, le coût de cet algorithme est particulièrement mauvais.

Il est pire que les tris par **insertion** et **sélection** et même **à bulle**.

Les tri **fusion** et **rapide** ont une complexité en $\mathcal{O}(n.log(n))$.

Exerice 3

1.

Le caractère espace (noté `_` dans l'arbre) sera codé `010`

2.

Pour décoder le mot, on descend dans l'arbre jusqu'à arriver à une feuille, puis on recommence avec la suite du mot binaire.

```
Ainsi le mot binaire '000111010111111100011001' peut être découpé en

'00 011 1010 1111 11001 1001'
'e  s   p    i    o     n'

Le texte codé est : 'espion'
```

3.

Un **parcours en largeur d'abord** permet de parcourir les feuilles par ordre de profondeur croissante. C'est donc celui qui permet ici d'obtenir les symboles par taille d'encodage croissante.

Partie B

4.

Le total des occurrences est égal au nombre de symboles de la chaîne (22 dans l'exemple de la figure 2.) Pour avoir deux sous-groupes avec le nombre d'occurrences les plus proches possible, il faut que ces nombres soient le plus proche de la moitié (ici 11).

Dans le cas de la Figure 2., les deux sous-groupes ont un nombre d'occurrences égal à 11, c'est donc le partage optimal.

De plus l'ordre des symboles est bien conservé entre les sous-groupes et au sein de chaque sous groupe.

5.

La profondeur de la racine étant 0, la profondeur maximale est atteinte pour les feuilles `o` et `d` et elle est de 5. La hauteur de l'arbre de la Figure 3. est 5. Dans le contexte du codage de Shannon-Fano cela correspond donc à la taille maximale en bits du codage d'un symbole.

6.

En ASCII, la chaîne de caractères `je pense, donc je suis` est encodée avec 22 octets, c'est à dire 22 x 8 = 176 bits.

Le codage de Shannon-Faro demanderait ici beaucoup moins de bits :

- 4 bits pour chacun des caractères `i u c p` qui apparaissent chacun 1 fois : 16 bits
- 5 bits pour chacun des caractères `o d` qui apparaissent chacun 1 fois : 10 bits
- 4 bits pour chacun des caractères `n j` qui apparaissent chacun 2 fois : 16 bits
- 3 bits pour le caractère `s` qui apparait 3 fois : 9 bits
- 3 bits pour le caractère `_` qui apparait 4 fois : 12 bits
- 2 bits pour le caractère `e` qui apparait 4 fois : 8 bits

Au total le codage de Shannon-Fano utilise 16 + 10 + 16 + 9 + 12 + 8 = 71 bits. C'est environ deux fois moins que le codage ASCII.

> Cette performance du codage de Shannon-Fano s'explique autant par l'intérêt d'un codage préfixe qui utilise des codages plus courts pour les caractères les plus fréquents que par le nombre réduits de caractères codés dans le codage de Shannon-Fano : 12 au lieu des 128 de l'ASCII.

7. Dessin

Partie C

8.

```
def creer_dico_occ(texte: str) -> dict:
    dico = {} # dictionnaire vide
    for symbole in texte:
        if symbole in dico:
            dico[symbole] = dico[symbole] + 1
        else:
            dico[symbole] = 1 # première apparation du symbole
    return dico
```

9.

```
def somme_occ(tab: list) -> int:
    somme = 0
    for symbole, occ in tab:
        somme += occ
    return somme
```

ou par compréhension:

```
def somme_occ(tab: list) -> int:
    return sum([t[1] for t in tab])
```

10.

```
def separe(tab: list) -> (list,list):
    moitie = somme_occ(tab) // 2
    somme = 0
    i = 0
    while moitie > somme:
        somme = somme + tab[i][1] # incrémentation du nombre d'occurrence du i-ème symbole
        i = i + 1
    tab1 = [ tab[k] for k in range(0,i)]
    tab2 = [ tab[k] for k in range(i,len(tab))]
    return tab1, tab2
```

Fonction `shannon` complétée :

```
def shannon(symbole, tab):
    if len(tab) == 1:
        return ""
    else:
        t1, t2 = separe(tab)
        if symbole in [elt[0] for elt in t1]:
            return "1" + shannon(symbole, t1)
        else:
            return "0" + shannon(symbole, t2)
```

11.

La fonction récursive `shannon` se termine car elle possède une condition d'arrêt (len(tab) == 1) et que les appels récursifs se font sur des tableau de tailles qui décroissent strictement, au dernier appel, le tableau ne contient que le caractère et a donc pour taille 1.

12.

```python
def encode_shannon(texte: str) -> str:
    '''Retourne l'encodage binaire de texte avec le codage de Shannon-Fano'''
    dico = creer_dico_texte(texte) # création du dictionnaire
    tab = creer_tab_trie(dico) # liste triée suivant les nombres d'occurrences
    codage = {} # dictionnaire des codages des caractères

    # ajout des codages des caractères au dictionnaire codage
    for symbole in dico:
        codage[symbole] = shannon(symbole, tab)


    encodage_binaire = "" # chaine pour contenir le codage binaire

    for s in texte:
        encodage_binaire += codage[s]

    return encodage_binaire
```

On peut envisager une fonction sans le dictionnaire codage. Cependant ce dernier est utile pour ne pas avoir à rappeler la fonction `shannon` a chaque occurrence d'un symbole.