Correction Bac Blanc

Exercice 1

1.a - La requête de l'énoncé affiche la liste de tous les ordinateurs et affiche pour chacun sa marque et sa salle associée. C'est une relation de deux attributs. Elle produit l'affichage de la table suivante :

salle	marque_ordi
012	HP
114	Lenovo
223	Dell
223	Dell
223	Dell

1.b - La requête de l'énoncé affiche la liste des noms et des salles de tous les ordinateurs reliés à un vidéoprojecteur. Elle produit l'affichage de la table suivante :

nom_ordi	salle
Gen-24	012
Tech-62	114
Gen-132	223

2 - La requête donnant tous les attributs des ordinateurs correspondant aux années supérieures ou égales à 2017 ordonnées par dates croissantes est :

```sql
SELECT * FROM Ordinateur
WHERE anne >= 2017
ORDER BY annee ASC;
```

3.a - Pour des raisons de contrainte d'intégrité, l'attribut `salle` ne peut pas être une clé primaire. En effet, la clé primaire de chaque élément de la relation `Ordinateur` doit être **unique** ce qui n'est pas le cas de l'attribut proposé.

3.b - En respectant les notations de l'énoncé, la relation `Imprimante` se définit :

```
Imprimante(
    nom_imprimante: String,
    nom_ordi: String,
    marque_imp: String,
    modele_imp: String,
    salle: String)
```

`nom_ordi` est une clé étrangère de la relation `Imprimante` car c'est une clé primaire de la relation `Ordinateur`.

4.a - Pour insérer le vidéoprojecteur de l'énoncé en salle 315 il faudra écrire la requête :

"'sql INSERT INTO Videoprojecteur(salle, marque_video, modele_video, tni) VALUES ('315', 'NEC', 'ME402X', false);

4.b - La requête nécessite une jointure :

"'sql SELECT o.salle, o.nom_ordi, v.marque_video FROM Ordinateur AS o JOIN Videoprojecteur AS v ON o.salle = v.salle WHERE v.tni = false;

Exercice 2

1.a - L'abre possède 7 noeuds. Il a donc une taille égale à 7.

1.b - La hauteur de l'arbre est de 4. C'est le nombre de nœuds du plus grand chemin entre la racine et ses feuilles.

2 - L'arbre suivant possède les mêmes valeurs que celui de l'énoncé mais est bien construit :

```
     10
   /    \
  5     15
 / \   /  \
4   8 12   20
```

3 - Les instructions de l'énoncé produisent l'arbre suivant :

```
         10
       /      \
      /        \
    8           20
  /            /
 4            15
/ \          /  \
   5        12
  / \      / \
```

4 - La méthode `hauteur` de la classe `Noeud` renvoie la hauteur d'un noeud en partant de la racine. Ainsi une feuille a une hauteur de 1, un noeud relié à une feuille a une hauteur de 2 et ainsi de suite.

Pour implémenter la méthode hauteur de la classe arbre, il suffit de renvoyer la hauteur du nœud racine. Ce qui donne :

```python
class Arbre:

    ...

    def hauteur(self):
        return self.racine.hauteur()
```

5 - La méthode `taille` de la classe `Noeud` renvoie la taille du sous-arbre de racine le noeud. Elle peut être implémentée de la façon suivante :

```
class Noeud:

    ...

    def taille(self):
        # cas de base
        # le noeud est une feuille
        if self.gauche == None and self.droit == None:
            return 1

        if self.gauche == None:
            t_gauche = 0
        else:
            t_gauche = self.gauche.taille()

        if self.droit == None:
            t_droit = 0
        else:
            t_droit = self.droit.taille()

        return 1 + t_gauche + t_droit
```

On peut maintenant implémenter la méthode `taille` pour la classe `Arbre` en remarquant que la taille d'un arbre est égal à la taille de sa racine !

```
class Arbre:

    ...

    def taille(self):
```


grand que `nb_rouge` passé en argument, alors le contenu n'est pas correct et la fonction s'arrête en renvoyant `False`.

- on fait de même avec les éléments `"vert"`, puis avec les éléments `"jaune"`
- si la fonction arrive à passer les trois tests précédents, alors c'est qu'elle est correcte et elle s'arrête en renvoyant `True`.

[11]:
```
def verifier_contenu(F, nb_rouge, nb_vert, nb_jaune):
    rouge_reel = nb_elements(F, "rouge")
    if rouge_reel > nb_rouge:
        return False

    vert_reel = nb_elements(F, "vert")
    if vert_reel > nb_vert:
        return False

    jaune_reel = nb_elements(F, "jaune")
    if jaune_reel > nb_jaune:
        return False

    return True
```

– défiler F et mettre chaque élément dans une pile temporaire et dans une file temporaire
– remettre en état la file F en parcourant/vidant la file temporaire
– dépiler la pile temporaire dans la pile finale afin de remettre les éléments dans le bon ordre
– renvoyer la pile finale

[14]:
```python
def former_pile(F):
    F_temp = creer_file_vide()
    P_temp = creer_pile_vide()

    while not est_vide(F):
        element = defiler(F)
        enfiler(F_temp, element)
        empiler(P_temp, element)

    # remise en état de F
    while not est_vide(F_temp):
        enfiler(F, defiler(F_temp))

    # inversion de la pile P_temp
    # dans la pile P à renvoyer
    P = creer_pile_vide()
    while not est_vide(P_temp):
        empiler(P, depiler(P_temp))

    return P
```

3 - L'algorithme proposé ressemble beaucoup à celui de `taille_fle`. Cette fois-ci, on ajoute 1 au total seulement si l'élément défilé est égal à l'élément passé en argument de la fonction.

[15]:
```python
def nb_elements(F, elt):
    total = 0

    F_temp = creer_file_vide()
    while not(est_vide(F)):
        elt_courant = defiler(F)
        if elt_courant == elt:
            total = total + 1
        enfiler(F_temp, elt_courant)

    # remise en état de F
    while not(est_vide(F_temp)):
        enfiler(F, defiler(F_temp))

    return total
```

4 - L'implémentation proposée est la suivante :

– compter le nombre d'élément `"rouge"` de la file. Si ce nombre est plus


```python
        return self.racine.taille()
```

6.a - Un arbre binaire de recherche est *bien construit* s'il n'est pas possible de le *réduire* à un arbre de hauteur $h-1$ car sinon, la propriété *bien construit* ne serait pas vérifiée. Donc un tel arbre doit avoir une taille supérieure aux arbres de hauteur $h-1$, c'est-à dire que sa taille doit être supérieure à $2^{h-1}-1$.

On a donc l'encadrement suivant pour un arbre bien construit :

$2^{h-1}-1 < t \leq 2^h - 1.$

La deuxième partie de l'inégalité est vraie pour tout ABR, mais la première partie est caractéristique des ABR bien construit.

6.b - Implémentons la méthode `bien_construit` qui s'appuie sur une telle propriété :

"'python class Arbre :

```
...

def bien_contruit(self):
    t = self.taille()
    h = self.hauteur()

    if  t > 2 ** (h-1) - 1:
        return True
    else:
        return False
```

Exercice 3

1.a - Il faut implémenter une fonction renvoyant la somme des éléments d'un tableau donné en argument.

[2]:
```
def total_hors_reduction(tab):
    total = 0
    n = len(tab)
    for i in range(n):
        total = total + tab[i]
    return total


tab = [30.5, 15.0, 6.0, 20.0, 5.0, 35.0, 10.5]
print(total_hors_reduction(tab))
```

```
122.0
```

1.b - Voici la version complète de la fonction donnée dans l'énoncé :

[4]:
```
def offre_bienvenue(tab: list) -> float:
    somme = 0
    longueur = len(tab)
    if longueur > 0:
        somme = tab[0] * 0.8
    if longueur > 1:
        somme = somme + tab[1] * 0.7
    if longueur > 2:
        for i in range(2, longueur):
            somme = somme + tab[i]
    return somme


print(offre_bienvenue(tab))
```

```
111.4
```

2 - Pour implémenter la fonction demandée, il faut différencier tous les cas possibles :

[7]:
```
def prix_solde(tab):
    longueur = len(tab)
    total = total_hors_reduction(tab)
    if   longueur >= 5:
        return total * 0.5
    elif longueur == 4:
        return total * 0.6
    elif longueur == 3:
        return total * 0.7
    elif longueur == 2:
        return total * 0.8
    elif longueur == 1:
        return total * 0.9
    else:
        return 0


print(prix_solde(tab))
```

```
61.0
```


- F : enfilement $\rightarrow\rightarrow$ défilement
- P : empilement/dépilement $\leftrightarrow$ "rouge" "vert" "jaune" "rouge" "jaune"

1.b - Pour déterminer la taille, nous allons vider la file originale dans une file temporaire, initialement vide. À chaque défilement, nous incrémentons le compteur permettant un dénombrement.

Ensuite, pour remettre en état la file originale F, on défile la file temporaire dans la F qui était devenue vide.

[10]:
```
def taille_file(F):
    taille = 0

    F_temp = creer_file_vide()
    while not est_vide(F):
        taille = taille + 1
        enfiler(F_temp, defiler(F))

    while not est_vide(F_temp):
        enfiler(F, defiler(F_temp))

    return taille
```

2 - L'idée de l'algorithme est la suivante :

1. vider la file F dans une pile temporaire (comme le fait la question 1.a)
2. retourner/inverser la pile temporaire en la dépilant dans une deuxième pile
3. renvoyer la deuxième pile qui contient les mêmes éléments que F, dans le bon ordre.

[13]:
```
def former_pile(F):
    # pile temporaire qui contiendra les valeurs de F
    # mais dans l'ordre inversé
    P_temp = creer_pile_vide()
    while not est_vide(F):
        empiler(P_temp, defiler(F))

    # retourner/inverser la pile temporaire
    P = creer_pile_vide()
    while not est_vide(P_temp):
        empiler(P, depiler(P_temp))

    return P
```

Voici une deuxième implémentation qui va remettre en état la file F (au lieu de la laisser vide). L'algorithme est le suivant :

B.1 - Un algorithme de tri qui a une complexité meilleure que quadratique est le tri **fusion** ou le tri **rapide**. Ces deux ont des complexité quasi-linéaire en $n\log(n)$.

B.2 -

[7]:
```
def moitie_gauche(tab):
    if tab == []:
        return []

    n = len(tab)
    milieu = (n-1) // 2

    m_gauche = [None] * (milieu+1)
    for i in range(milieu + 1):
        m_gauche[i] = tab[i]

    return m_gauche
```

B.3 - Implémentation :

```
def nb_inversions_rec(tab):
    tab_g = moitie_gauche(tab)
    nb_inv_gauche = nb_inversions_rec(tab_g)

    tab_d  = moitie_droit(tab)
    nb_inv_droit = nb_inversions_rec(tab_d)

    tab_g_trie = tri(tab_g)
    tab_d_trie = tri(tab_d)
    nb_inv_tries = nb_inv_tab(tab_g_trie, tab_d_trie)

    nb_inv = nb_inv_gauche + nb_inv_droit + nb_inv_tries
    return nb_inv
```

Exercice 5

1.a - À la fin de l'exécution, la file F est vide et la pile P contient le contenu de F initial inversé :

3.a - Implémentation de la fonction qui renvoie la valeur minimale d'un tableau :

[4]:
```
def minimum(tab):
    mini = tab[0]
    longueur = len(tab)
    for  i in range(1, longueur):
        if tab[i] < mini:
            mini = tab[i]
    return mini


print(minimum(tab))
```

```
5.0
```

3.b - Utilisons la fonction `minimum()` créée à la question précédente :

[5]:
```
def offre_bon_client(tab):
    total = total_hors_reduction(tab)
    longueur = len(tab)
    if longueur > 1:
        mini = minimum(tab)
        total = total - mini
    return total


print(offre_bon_client(tab))
```

```
117.0
```

4.a - Si on permute les articles à `6.0` et `20.0`, alors on obtient le tableau `[30.5, 15.0, 20.0, 6.0, 5.0, 35.0, 10.5]`. Pour ce tableau, les articles à `15.0` et `5.0` sont offerts. Le prix après promotion est donc différent de 111 euros.

4.b - Pour avoir un prix le plus bas possible, je propose (par exemple) le panier suivant :

`[35.0, 30.5, 20.0, 6.0, 10.5, 15.0, 5.0]`

Le total de remise s'élève à $20,0 + 6,0 = 26,0$.

4.c - Pour minimiser le coût il faut maximiser la remise. Ainsi, pour un tableau donné, il faut arriver à mettre les articles les plus chers en remise. Une méthode systématique pour arriver à cela est d'ordonner les articles par **ordre décroissant**.

Exercice 4

1 - **réponse souvent incomplète sur les copies**

le couple $(1, 3)$ est une inversion pour le tableau `[4, 8, 3, 7]` car il respecte des deux propriétés :

- $1 < 3$
- `tab[1]` $= 8 >$ `tab[3]` $= 7$

2 - le couple $(2, 3)$ n'est pas une inversion car :

- $2 < 3$
- mais `tab[2]` $= 3 >$ `tab[3]` $= 7$ n'est pas vérifiée

A.1.a et A.1.b - La fonction `fonction1(tab, i)` compte le nombre d'inversion $(\mathtt{i}, j)$ que contient le tableau `tab` en partant du rang `i` fixé.

- `fonction1([1, 5, 3, 7], 0)` compte le nombre d'inversion de rang 0. Il n'y en a aucune donc la fonction renvoie `0`. En effet :
  - $1 < 5$ donc le couple $(0, 1)$ n'est pas une inversion
  - $1 < 3$ donc le couple $(0, 2)$ n'est pas une inversion
  - $1 < 7$ donc le couple $(0, 3)$ n'est pas une inversion
- `fonction1([1, 5, 3, 7], 1)` compte le nombre d'inversion de rang 1. Il y en a une seule car $5 > 3$ pour le couple $(1, 2). La fonction renvoie donc `1`.
- `fonction1([1, 5, 2, 6, 4], 1)` compte le nombre d'inversion de rang 1. Il y en a deux car $5 > 2$ pour le couple $(1, 2) et $5 > 4$ pour le couple $(1, 4)$. La fonction renvoie donc `2`.

[14]:
```
def fonction1(tab, i):
    nb_elem = len(tab)
    cpt = 0
    for j in range(i+1, nb_elem):
```



```
        if tab[j] < tab[i]:
            cpt += 1
    return cpt

print(fonction1([1, 5, 3, 7], 0))
print(fonction1([1, 5, 3, 7], 1))
print(fonction1([1, 5, 2, 6, 4], 1))
```

0
1
2

A.2 - Pour compter le nombre total d'inversions, on va accumuler toutes les inversions de rang 0, puis toutes celles de rang 1, et ainsi de suite jusqu'à l'avant dernière case case du tableau.

[20]:
```
def nombre_inversions(tab):
    total = 0
    longueur = len(tab)
    for i in range(longueur-1):
        total = total + fonction1(tab, i)
    return total

print (nombre_inversions([1, 5, 7]))
print (nombre_inversions([1, 6, 2, 7, 3]))
print (nombre_inversions([7, 6, 5, 3]))
```

0
3
6

A.3 - Soit `n` la longueur du tableau, la fonction `nombre_inversion` effectue `n` boucles.

Chaque boucle effectue `n` appels à `fonction1`, qui elle même effectue `i` tours de boucles.

La fonction `nombre_inversions` effectue donc $n+(n-1)+(n-2)+\ldots+2+1 = \frac{n(n+1)}{2}$ tests.

L'ordre de grandeur de la complexité en temps est donc n^2 avec n la longueur du tableau.

C'est une complexité quadratique.