

[← Index des sujets 2021](#)

21-NSIJ2PO1 : Corrigé

Année : **2021**

Centre : **Polynésie**

Jour : **2**

Enoncé : [PDF](#)

1. Exercice 1

algorithmique et programmation (algorithmes de tri)

1.1. Partie A

1. Les affichages obtenues seront :

- `8` car `len(notes)` est le nombre d'éléments de la liste `notes`
- `[8, 7, 18, 16, 12, 9, 17, 3]` car on affiche la liste de départ en ayant remplacé la valeur située à l'indice 3 par 16.

2. Pour afficher les éléments d'indice 2 à 4 de la liste on peut écrire :

Script Python

```
for i in range(2,5):
    print(note[i])
```

Note

L'utilisation d'une boucle ne s'impose pas, on aurait pu écrire `print(note[2], note[3], note[4])` ou encore utiliser les *slices* : `print(note[2:5])` (bien que cette solution affiche une *liste* composée des trois éléments demandés)

1.2. Partie B

1. Code complété :

Script Python

```
1 def tri_insertion(liste):
2     """ trie par insertion la liste en paramètre """
3
4     for i in range(1, len(liste)):
5         key = liste[i]
6         j = i - 1
7
8         while j >= 0 and liste[j] > key:
9             liste[j + 1] = liste[j]
10            j -= 1
11
12        liste[j + 1] = key
```

```

3     for indice_courant in range(1,len(liste)):
4         element_a_inserer = liste[indice_courant]
5         i = indice_courant - 1
6         while i >= 0 and liste[i] > element_a_inserer :
7             liste[i+1] = liste[i]
8             i = i - 1
9         liste[i + 1] = element_a_inserer

```

2. Après le premier passage : [7, 8, 18, 14, 12, 9, 17, 3]

3. Après le troisième passage : [7, 8, 14, 18, 12, 9, 17, 3]

Note

On rappelle que le principe de l'algorithme est d'insérer au passage n le n -ième élément de la liste dans le début de la liste (déjà triée).

1.3. Partie C

1. Cet algorithme est itératif car un tri fusion fait appel lui-même
2. Si les deux tas sont déjà triés, il suffit de comparer les cartes situées sur le dessus de chaque tas et de prendre la plus petite.
3. Code complété :

Script Python

```

1 from math import floor
2
3 def tri_fusion (liste, i_debut, i_fin):
4     """ trie par fusion la liste en paramètre depuis i_debut jusqu'à i_fin """
5     if i_debut < i_fin:
6         i_partage = floor((i_debut + i_fin) / 2)
7         tri_fusion(liste, i_debut, i_partage )
8         tri_fusion(liste, i_partage+1 , i_fin)
9         fusionner(liste, i_debut , i_partage , i_fin)

```

Note

L'utilisation de la fonction `floor` ne s'impose pas, puisque'on travaille sur des entiers `i_partage` se définit sans recours à la bibliothèque `math` avec `i_partage = (i_debut+i_fin)//2`.

4. Cette ligne permet d'importer la méthode `floor` à partir du module `math`.

1.4. Partie D

1. C'est l'algorithme du tri fusion qui a été utilisée. Chaque étape représente la fusion de listes déjà triées.
2. Le tri par insertion a une complexité en $O(n)$ dans le pire des cas et le tri fusion une complexité en $O(n \log_2(n))$.
3. Dans un tri par insertion, on effectue un maximum de n insertions demandant chacune au plus n opérations. Cette algorithme a donc une complexité quadratique. Pour le tri fusion, on note $C(n)$ le coût en nombre d'opérations pour trier une liste de taille n . Pour trier une liste de taille n , séparer les deux listes (coût de n opérations), trier deux listes de taille $\frac{n}{2}$ et les fusionner (coût de n opérations). Donc $C(n) = 2C(\frac{n}{2}) + 2n$, on montre que cela implique un coût en $O(n \log_2(n))$.

Attention

Question difficile et à la limite du programme de NSI qui par ailleurs parle de *coût* plutôt que de *complexité*.

2. Exercice 2

données en table, bases de données

3. Exercice 3

arbres binaires de recherche et programmation orientée objet

3.1. Partie A : Etude d'un exemple

1. Le noeud racine a pour valeur 5, et ses fils sont 2 et 7.
2. Ce sont les noeuds 5,2 et 3.
3. Arbre obtenu après l'ajout de la valeur 6 :

```
graph TD
    N5["5"] --> N2["2"]
    N5 --> N7["7"]
    N2 --> V1[" "]
    N2 --> N3["3"]
    N7 --> N6["6"]
    N7 --> N8["8"]
    style V1 fill:#FFFFFF, stroke:#FFFFFF
    style N6 fill:#DD4444
    linkStyle 2 stroke:#FFFFFF,stroke-width:0px
    linkStyle 4 stroke:#FF0000
```

3.2. Partie B : Implémentation en Python

- La fonction `__init__` permet de créer un objet de type ABR, par défaut c'est l'arbre binaire vide (`valeur=None`) mais on peut préciser une valeur pour le noeud racine en modifiant ce paramètre.
- Si on ajoute un élément déjà présent dans l'arbre, alors il ne se passe rien. En effet, dans la méthode `insereElement` le cas `e==self.valeur` n'est pas traité.

3. Script Python

```

1 arbre = ABR(5)
2 arbre.insereElement(2)
3 arbre.insereElement(3)
4 arbre.insereElement(7)
5 arbre.insereElement(8)

```

3.3. Partie C : Tri par arbre binaire de recherche

- C'est le parcours infixé dans lequel on liste la valeur d'un noeud *entre* les valeurs de sons sous arbre gauche et les valeurs de son sous arbre droit.
- On sait que les tri par insertion et par sélection ont tous les deux une complexité quadratique. Dans ce nouvel algorithme :
 - l'insertion d'une valeur dans l'arbre a une complexité logarithme (semblable à celle d'une recherche dichotomique)
 - donc l'insertion des n valeurs a une complexité en $O(n \log(n))$
 - Une fois les insertions effectuées le parcours a une complexité linéaire La complexité de ce nouvel algorithme est donc en $O(n \log(n))$ (parfois appelé complexité pseudo linéaire) et est donc meilleur que la complexité quadratique des tris par selection ou par insertion.

4. Exercice 4

routage, architecture matérielle

5. Exercice 5

données en table, bases de données