TP – CHAPITRE 12

Sommaire

TP1 - Tri Fusion

- On considère la liste suivante : [8, 5, 6, 10, 2, 7, 12, 9, 4, 1, 3, 11, 0, 13] .
- Sur une feuille de brouillon, détailler toutes les étapes du tri fusion pour trier cette liste.

Lever la main pour valider ce TP.

TP2 - Recherche linéaire d'un élément dans une liste triée

- Dans **Thonny** créer un fichier nommé **TP2 .py** dans lequel, vous allez rédiger une fonction **rechercher1(e, liste)** . Cette fonction doit renvoyer **True** si l'élément e est dans la liste et **False** sinon, en utilisant **l'algorithme linéaire**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    liste1 = [i for i in range(100)]
    print('recherche1 :', rechercher1(48, liste1))
    print('recherche2 :', rechercher1(150, liste1))
```

- Retours attendus dans la console :

```
>>> test()
  recherche1 : True
  recherche2 : False
```

Lever la main pour valider ce TP.

TP3 - Recherche dichotomique récursive d'un élément dans une liste triée

- Dans **Thonny** créer un fichier nommé **TP3 .py** dans lequel, vous allez rédiger une fonction **rechercher2(e, liste)** . Cette fonction doit renvoyer **True** si l'élément e est dans la liste et **False** sinon, en utilisant **l'algorithme dichotomique récursif**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    liste1 = [i for i in range(500)]
    print('recherche1 :', rechercher2(4080, liste1))
    print('recherche2 :', rechercher2(450, liste1))
```

- Retours attendus dans la console :

```
>>> test()
  recherche1 : False
  recherche2 : True
```

Lever la main pour valider ce TP.

TP4 - Recherche dichotomique itérative d'un élément dans une liste triée

- Dans **Thonny** créer un fichier nommé **TP4 .py** dans lequel, vous allez rédiger une fonction **rechercher3(e, liste)** . Cette fonction doit renvoyer **True** si l'élément e est dans la liste et **False** sinon, en utilisant **l'algorithme dichotomique itératif**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    liste1 = [i for i in range(-200, 200)]
    print('recherche1 :', rechercher3(-380, liste1))
    print('recherche2 :', rechercher3(150, liste1))
    print('recherche3 :', rechercher3(450, liste1))
```

- Retours attendus dans la console :

```
>>> test()
  recherche1 : False
  recherche2 : True
  recherche3 : False
```

Lever la main pour valider ce TP.

TP5 - Recherche linéaire d'un maximum

- Dans **Thonny** créer un fichier nommé **TP5 .py** dans lequel, vous allez rédiger une fonction **maximum1(liste)** . Cette fonction doit renvoyer la valeur maximum contenue dans la liste ou **None** si la liste est vide, en utilisant **l'algorithme linéaire**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    liste1 = [-2, 5, 1, -8, 14, 7, 6, -3, 0, 9]
    liste2 = [i%20 for i in range(50)]
    liste3 = []
    print('maximum1 :', maximum1(liste1))
    print('maximum2 :', maximum1(liste2))
    print('maximum3 :', maximum1(liste3))
```

- Retours attendus dans la console :

```
>>> test()
  maximum1 : 14
  maximum2 : 19
  maximum3 : None
```

Lever la main pour valider ce TP.

TP6 - Recherche récursif d'un maximum avec diviser pour régner

- Dans **Thonny** créer un fichier nommé **TP6 .py** dans lequel, vous allez rédiger une fonction **maximum2(liste)** . Cette fonction doit renvoyer la valeur maximum contenue dans la liste ou **None** si la liste est vide, en utilisant **l'algorithme récursif diviser pour régner**.

- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    liste1 = [-2, 5, 1, -8, 14, 7, 6, -3, 0, 19]
    liste2 = [i%30 for i in range(100)]
    liste3 = []
    print('maximum1 :', maximum2(liste1))
    print('maximum2 :', maximum2(liste2))
    print('maximum3 :', maximum2(liste3))
```

- Retours attendus dans la console :

```
>>> test()
  maximum1 : 19
  maximum2 : 29
  maximum3 : None
```

Lever la main pour valider ce TP.

TP7 - Calcul linéaire d'une puissance

- Dans **Thonny** créer un fichier nommé **TP7 .py** dans lequel, vous allez rédiger une fonction **puissance1(x, n)** . Cette fonction doit renvoyer la valeur de x^n, en utilisant **l'algorithme linéaire**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    print('puissance1 :', puissance1(2, 10))
    print('puissance2 :', puissance1(10, 6))
    print('puissance3 :', puissance1(-3, 5))
    print('puissance4 :', puissance1(7, 0))
```

- Retours attendus dans la console :

```
>>> test()
  puissance1 : 1024
  puissance2 : 1000000
  puissance3 : -243
  puissance4 : 1
```

Lever la main pour valider ce TP.

TP8 - Calcul récursif d'une puissance avec diviser pour régner

- Dans **Thonny** créer un fichier nommé **TP8 .py** dans lequel, vous allez rédiger une fonction **puissance2(x, n)** . Cette fonction doit renvoyer la valeur de x^n, en utilisant **l'algorithme récursif diviser pour régner**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    print('puissance1 :', puissance2(2, 8))
    print('puissance2 :', puissance2(10, 3))
    print('puissance3 :', puissance2(-3, 7))
    print('puissance4 :', puissance2(5, 0))
```

- Retours attendus dans la console :

```
>>> test()
  puissance1 : 256
  puissance2 : 1000
  puissance3 : -2187
  puissance4 : 1
```

Lever la main pour valider ce TP.

TP9 - * La multiplication récursive de Karatsuba

Algorithme de Karatsuba : Pour multiplier deux nombres a et b de n chiffres, la méthode naïve multiplie chaque chiffre du multiplicateur par chaque chiffre du multiplicande. Cela exige donc n^2 produits de deux chiffres. Le temps de calcul est en $O(n^2)$.
En 1960, Karatsuba remarque que pour tout k, le calcul naîf d'un produit :

$$(a_0 \times 10^k + a_1)(b_0 \times 10^k + b_1) = a_0 b_0 \times 10^{2k} + (a_0 b_1 + a_1 b_0) \times 10^k + a_1 b_1$$

qui semble nécessiter les quatre produits $a_0 b_0$, $a_1 b_1$, $a_0 b_1$ et $a_1 b_0$, peut en fait être effectué seulement avec les trois produits $a_0 b_0$, $a_1 b_1$ et $(a_0 - a_1)(b_0 - b_1)$ en regroupant les calculs de la façon suivante :

$$(a_0 \times 10^k + a_1)(b_0 \times 10^k + b_1)$$
$$= a_0 b_0 \times 10^{2k} + ((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) \times 10^k + a_1 b_1$$

Pour de grands nombres et en prenant $k = n/2$, la méthode peut être appliquée de manière récursive pour les calculs de $a_0 b_0$, $a_1 b_1$ et $(a_0 + a_1)(b_0 + b_1)$ en scindant à nouveau a_0, a_1, b_0 et b_1 en deux et ainsi de suite. C'est un algorithme de type **diviser pour régner**.
La multiplication par la base de numération (10 dans l'exemple précédent, mais 2 pour les machines) correspond à un décalage de chiffre, et les additions sont peu coûteuses en temps ; ainsi, le fait d'être capable de calculer les grandeurs nécessaires en 3 produits au lieu de 4 mène à une amélioration de complexité.

- Dans **Thonny** créer un fichier nommé **TP9 .py** dans lequel, vous allez rédiger une fonction **récursive** **multiplication(a, b)** . Cette fonction doit renvoyer la valeur de $a \times b$, en utilisant **l'algorithme récursif diviser pour régner de Karatsuba**.
- Rédiger une fonction **test( )** avec le code ci-dessous :

```
def test():
    print('multiplication1 :', multiplication(48, 12))
    print('multiplication2 :', multiplication(1014, 3875))
    print('multiplication3 :', multiplication(45087158, 15975347))
```

- Retours attendus dans la console :

```
>>> test()
  multiplication1 : 576
  multiplication2 : 3929250
  multiplication3 : 786897405093826
```

Lever la main pour valider ce TP.

TP10 - La rotation de 90° d'une image carrée dans le sens horaire

- Récupérer en ressources les fichiers **TP10 .py** et **Joconde_256.png** .
- Ouvrir le fichier **TP10.py** avec **Thonny** et rédiger du code à la place des trois **pass** pour que la fonction **rotation( )** applique une rotation de 90° dans le sens horaire à l'image **Joconde_256.png** selon le principe diviser pour régner.
- Ouvrir le fichier **Joconde2.png** et vérifier que la rotation de 90° a bien été appliquée dans le sens horaire.

Lever la main pour valider ce TP.

TP11 - * La rotation de 90° d'une image carrée dans le sens anti-horaire

- Dupliquer le fichier **TP10 .py** en **TP11 .py** .
- Modifier le code de la fonction **rotation( )** pour que la rotation de 90° de l'image s'effectue dans le sens anti-horaire. Modifier le nom du fichier où l'image sera tournée en **Joconde3.png** .
- Ouvrir le fichier **Joconde3.png** et vérifier que la rotation de 90° a bien été appliquée dans le sens anti-horaire.

Lever la main pour valider ce TP.

TP12 - * La rotation de 180° d'une image carrée

- Dupliquer le fichier **TP11 .py** en **TP12 .py** .
- Modifier le code de la fonction **rotation( )** pour que la rotation de l'image soit une rotation de 180° et modifier le nom du fichier image obtenue en **Joconde4.png** .
- Ouvrir le fichier **Joconde4.png** et vérifier que la rotation de 180° a bien été appliquée.

Lever la main pour valider ce TP.