

Algorithmes et Pensée Computationnelle

Algorithmes et Complexité

Le but de cette séance est d'aborder divers concepts de langages de programmation. En effet, la série d'exercices porte sur :

1. l'usage des fonctions, des listes et des dictionnaires,
2. la complexité des algorithmes,
3. une introduction à la récursion et
4. les algorithmes de tri

Les exercices sont disponibles en Python et en Java.

1 Les fonctions (Java ou Python) (30 minutes)

1.1 Rôle

Les `fonctions` permettent d'enregistrer du code dans une variable afin de réutiliser celui-ci à plusieurs endroits, et ainsi éviter de devoir le réécrire. Celles-ci sont définies une fois et peuvent être réutilisées autant de fois que l'on veut par la suite.

Les `fonctions` permettent de *factoriser* le code, offrant ainsi une structure plus générale à celui-ci et le rendant plus facilement modifiable.

Un exemple simple de fonctions est de créer une fonction qui imprime "Hello World" sur l'écran en l'appelant.

1.2 Syntaxe

1.2.1 Python

Pour définir une fonction, nous utilisons le mot `def` suivi du nom de notre `fonction`, puis des parenthèses `()`. Ces parenthèses peuvent contenir ou non des noms d'`arguments`, mais nous reviendrons dessus plus tard. Pour appeler une fonction, il suffit d'écrire son nom suivi de parenthèses `()`.

Pour reprendre l'exemple mentionné ci-dessus, nous déclarons une fonction du nom `print_hello` qui a pour unique utilité de `print` la phrase "Hello World". Puis nous appelons cette fonction

```
def print_hello():  
    print("Hello World")  
  
print_hello()
```

1.2.2 Java

En Java, les fonctions ont la structure suivante :

```
TypeDeRetour nomDeLaMethode() {  
    liste d'instructions  
}
```

Notez le typage dans les fonctions en Java. Si une fonction ne retourne rien, nous utilisons le type `void`. Une fonction dans Java doit être dans une classe, et pour exécuter le code de notre fonction, nous devons l'appeler dans la méthode `main`. L'exemple en Python ci-dessus peut être réécrit de la façon suivante :

```
public class Main {  
    public static void print_hello() {  
        System.out.println("Hello World!");  
    }  
    public static void main(String[] args){  
        print_hello();  
    }  
}
```

Question 1: (🕒 5 minutes) Python ou Java

Créez une fonction du nom de votre choix qui affiche votre prénom et appelez cette fonction.

💡 Conseil

Réutilisez les mêmes structures que ci-dessus. En Python, utilisez la fonction `print()` pour afficher du texte dans la console.
En Java, utilisez la fonction `System.out.println()`.

Solutions :

1. Python :

```
def print_name():  
    print("Je m'appelle Yasser")  
  
print_name()
```

2. Java :

```
public class Main {  
    public static void print_name() {  
        System.out.println("Je m'appelle Yasser");  
    }  
    public static void main(String[] args){  
        print_name();  
    }  
}
```

1.3 Arguments

Comme dit précédemment, une fonction peut avoir un ou plusieurs *arguments*. Comme en maths, les arguments sont des valeurs que l'on passe à notre fonction et c'est avec ces valeurs que la fonction va effectuer ses opérations.

Par exemple en maths, lorsqu'on écrit $f(x) = x+2$, l'argument de la fonction f est x , je peux maintenant simplement écrire $f(2)$, ce qui signifie **remplacer x par 2 dans la fonction f** .

1.3.1 Python

En Python, les arguments fonctionnent de la même façon. Dans l'exemple suivant, nous créons une fonction du nom `print_name` qui prend un *argument* que nous appelons `name`. Nous nous servons de cet argument pour faire `print("Mon nom est", name)`. Nous appelons ensuite cette fonction avec un argument.

```
def print_name(name):  
    print("Mon nom est", name)  
  
print_name("Yasser")
```

1.3.2 Java

Nous reprenons l'exemple précédent et le réécrivons en Java :

```
public class Main {  
    public static void print_name(String name) {  
        System.out.println("Mon nom est" + name);  
    }  
    public static void main(String[] args){  
        print_name("Yasser");  
    }  
}
```

Question 2: (🕒 5 minutes) Python ou Java

Créez une fonction du nom de votre choix qui prend un argument x et qui `print(x+1)`.

Conseil

En Python, utilisez la fonction `print()` pour afficher du texte dans la console.
En Java, utilisez la fonction `System.out.println()`. N'oubliez pas de typer vos arguments !

Solutions :

1. Python :

```
def add_one(x):  
    print(x+1)  
  
add_one(5)
```

2. Java :

```
public class Main {  
    public static void add_one(int x) {  
        System.out.println(x+1);  
    }  
    public static void main(String[] args){  
        add_one(1);  
    }  
}
```

1.4 Return

Il est très commun que nous voulions enregistrer le résultat d'une fonction dans une variable. Par exemple, en maths, si nous avons une fonction $f(x) = x + 15$, nous pouvons faire $y = f(10)$ et nous savons donc que y vaut 25.

1.4.1 Python

En Python, si nous écrivons :

```
def f(x):  
    x + 15  
  
y = f(10)  
print(y)
```

Le `print(y)` va afficher `None`, car le résultat de `f(10)` ne vaut rien.

Pour résoudre ce problème, nous avons le mot-clef `return`, celui-ci permet de retourner une valeur de la fonction pour permettre d'enregistrer le résultat dans une variable. Pour reprendre l'exemple précédent :

```
def f(x):  
    return x + 15  
  
y = f(10)  
print(y)
```

Cette fois-ci y vaut bien 25, car nous avons fait `return x + 15`.

1.4.2 Java

En Java, nous utilisons le mot-clef `return` aussi pour retourner le résultat. De plus, il faut spécifier le type de la variable que nous retournons.

```
TypeDeRetour nomDeLaMethode(type1 argument1) {  
    liste d'instructions  
    return variable;  
}
```

Nous convertissons le code Python ci-dessus en code Java :

```
public class Main {
    public static int f(int x) {
        return x + 15;
    }
    public static void main(String[] args){
        int y = f(10);
        System.out.println(y);
    }
}
```

Question 3: (🕒 5 minutes) Python ou Java

Écrivez une fonction de nom `f` qui prend un argument `x` et qui retourne $x * 10 + 2$, appelez cette fonction et enregistrer le résultat de celle-ci dans une variable `y`, puis `print` `y`.

💡 Conseil

Solutions :

1. Python :

```
def f(x):
    return (x*10)+2
y=f(2)
print(y)
```

2. Java :

```
public class Main {
    public static int f(int x) {
        return (x * 10) + 2;
    }
    public static void main(String[] args){
        int y = f(2);
        System.out.println(y);
    }
}
```

1.5 Exercices d'applications (Optionnels)

Question 4: (🕒 5 minutes) Python ou Java

Complétez la fonction ci-dessous afin qu'elle retourne le cube de l'argument `x`.

Python :

```
def cube(x):
    # Complétez ici

y = cube(2)
print(y)
```

Java :

```
public class Main {
    public static int cube(int x) {
        // Complétez ici
    }
    public static void main(String[] args){
        y = cube(2);
        System.out.println(y);
    }
}
```

💡 Conseil

En Python, utilisez l'opérateur `**` afin de faire des puissances.
La librairie `java.lang.Math` peut vous être utile pour la version Java.

Solutions :

— Python :

```
def cube(x):  
    return x**3
```

```
y = cube(2)  
print(y)
```

— Java :

```
import java.lang.Math;  
  
public class Main {  
    public static int cube(int x) {  
        return Math.pow(x, 3);  
    }  
    public static void main(String[] args){  
        y = cube(2);  
        System.out.println(y);  
    }  
}
```

Question 5: (🕒 5 minutes) **Python ou Java** Complétez la fonction ci-dessous afin qu'elle affiche les nombres de 0 à 10.

Python :

```
def counting():  
    i = 0  
    # Complétez ici
```

```
counting()
```

Java :

```
public class Main {  
    public static void counting() {  
        int i = 0;  
        // Complétez ici  
    }  
    public static void main(String[] args){  
        counting();  
    }  
}
```

Solutions :

— Python :

```
def counting():  
    i = 0  
    while i <= 10:  
        print(i)  
        i += 1
```

```
counting()
```

— Java :

```
public class Main {  
    public static void counting() {  
        int i = 0;  
        while (i <= 10){  
            System.out.println(i);  
            i += 1;  
        }  
    }  
    public static void main(String[] args){  
        counting();  
    }  
}
```

2 Listes et dictionnaires (Java ou Python)

2.1 Listes

2.1.1 Python

Les **listes** permettent de stocker plusieurs éléments, et sont immuables. Nous pouvons ainsi modifier leur contenu ou retirer et rajouter dynamiquement des éléments.

Méthodes principales :

- **Création** : `ma_liste = [1, 2, 3, 4]`
- **Modification** : `ma_liste[2] = 0`
- **Ajout** : `ma_liste.append(10)`
- **Suppression** : `ma_liste.pop()` pour enlever le dernier élément dans la liste ou `ma_liste.remove(10)` pour enlever 10 de la liste
- **Slicing / Indexation** : `my_list[0:2]` pour prendre les 2 premiers éléments de la liste [i (inclus) : j (exclu)]
- **Ajout avec indexation** : `my_list[0:2] = [4]` avec les éléments à rajouter entre crochets
- **Suppression avec indexation** : `my_list[0:2] = []`, sans rien entre les crochets

Question 6: (🕒 5 minutes) Comportement des listes en Python

Après l'exécution du code ci-dessous, à quoi va ressembler `my_list` ?

```
my_list = [1, 3, 5, 7, 11, 12]
print(my_list[0:3])

my_list.append(15)
my_list.pop()
my_list.remove(12)

my_list += [17, 30]
my_list[0:2] = [4]
my_list[0:3] = []
```

💡 Conseil

Écrivez le contenu de la liste après chaque opération pour pouvoir mieux comprendre le déroulement du programme.

Solutions :

```
[11, 17, 30]
```

2.1.2 Java

En Java, nous faisons la distinction entre **Array**, une liste à dimension fixe, et **ArrayList**, une liste à dimension variable.

1. **Array** : La taille de la liste doit être déclarée à l'initialisation, ou vous pouvez directement spécifier le contenu de la liste à l'initialisation. Après cela, la taille de la liste ne pourra être modifiée.

```
int[] mon_array = new int[5];
int[] mon_array1 = {1, 2, 3};
```

On utilise des accolades pour initialiser un **Array** avec des valeurs. Méthodes principales :

- (a) **Accès** : `ma_liste[0]`
 - (b) **Modification** : `ma_liste[1] = 10`
 - (c) **Slicing / Indexation** : `int[] newArray = Arrays.copyOfRange(oldArray, startIndex, endIndex);`
2. **ArrayList** : Plusieurs options s'offrent à nous quant à l'initialisation d'une **ArrayList**.

```
import java.util.ArrayList;
import java.util.List;

1. ArrayList liste = new ArrayList();
2. List<Integer> nombres = new ArrayList<>(6); // Dimension = 6

3. Collection elements = ...;
   List<Integer> nombres = new ArrayList<>(elements);
   // ArrayList contenant la collection elements
```

La première méthode est **déconseillée**, car elle ne spécifie pas explicitement le type des valeurs contenue dans l'`ArrayList`. Ainsi, nous devons toujours spécifier clairement le type pendant l'initialisation :

```
List<TypeDesValeurs> nomDeLaListe = new ArrayList<>();
```

Les méthodes principales pour les `ArrayList` :

- **Accès** : `ma_liste.get(0);`
- **Modification** : `ma_liste.set(index, value);`
- **Ajout** : `ma_liste.add(10);`
- **Suppression** : `ma_liste.remove("Java");` pour enlever "Java" de la liste ou `ma_liste.remove(10);` pour enlever l'élément à l'index 10.
- **Slicing / Indexation** : `ma_liste.sublist(startIndex, endIndex);`

2.2 Dictionnaires

Les `dictionnaires` sont des listes associatives, c'est-à-dire des listes qui relient une valeur à une autre.

2.2.1 Python

Dans un dictionnaire Python, on parle d'une relation **clef**, **valeur**. La **clef** étant le moyen de "*retrouver*" notre **valeur** dans notre **dictionnaire**.

Méthodes principales :

- Ajout de la **clef** "**Clef**" avec valeur "**Valeur**" : `my_dict["Clef"] = "Valeur"`
- Suppression d'une relation **clef-valeur** :
 - `my_dict.pop("Clef", None)` au cas où on sait pas si la **clef** en question est présente dans le dictionnaire
 - `del my_dict["Clef"]` si vous êtes sûrs que la **clef** en question est dans le dictionnaire

Vous pouvez trouver ci-dessous un exemple d'utilisation des dictionnaires :

```
annuaire = {"Shioban": 111, "Tyson": 222, "Shawn": 333 }
annuaire["Steven"] = 444
del annuaire["Tyson"]
print(annuaire.keys())
print(annuaire.values())
```

2.2.2 Java

En Java, les dictionnaires sont appelés des **Map**. La structure pour initialiser une **Map** est la suivante :

```
Map<Type de la clef, Type des éléments> dictionnaire = new HashMap<>();
```

HashMap est une des implémentations de **Map** les plus utilisées, et une des plus performantes.

Méthodes principales :

- Initialisation d'un dictionnaire avec comme **clef** des **String** et comme valeurs des **Integer** : `Map<String, Integer> age = new HashMap<>();`
- Ajout de la **clef** "**Justine**" avec valeur 22 : `age.put("Justine", 22)`
- Accès : `age.get("Justine")`
- Suppression d'une relation **clef-valeur** : `age.remove("Justine")`

3 Complexité (30 minutes)

Pour chaque algorithme ci-dessous, indiquez en une phrase, ce que font ces algorithmes et calculez leur complexité temporelle avec la notation $O()$. Le code est écrit en Python et en Java.

Question 7: (🕒 10 minutes) Complexité

Python :

```
# Entrée: n un nombre entier
def algo1(n):
    s = 0
    for i in range(10*n):
        s += i
    return s
```

Java :

```
public static int algo1(int n) {
    int s = 0;
    for (int i=0; i < 10*n; i++){
        s += i;
    }
    return s;
}
```

💡 Conseil

Rappelez vous que la notation $O()$ sert à exprimer la complexité d'algorithmes dans le **pire scénario**. Les règles suivantes vous seront utiles. Pour n étant la taille de vos données, on a que :

1. Les constantes sont ignorées : $O(2n) = 2 * O(n) = O(n)$
2. Les termes dominés sont ignorés : $O(2n^2 + 5n + 50) = O(n^2)$

>_ Solution

L'algorithme est composé d'une boucle qui incrémente une variable s . Il effectue $10*n$ l'opération et par conséquent a une complexité de $O(n)$.

Question 8: (🕒 10 minutes) Complexité

Python :

```
# Entrée: liste de nombre entiers et M un nombre entier
def algo2(L, M):
    i = 0
    while i < len(L) and L[i] <= M:
        i += 1
    s = i - 1
    return s
```

Java :

```
public static int algo2(int[] L, int M) {
    int i = 0;
    while (i < L.length && L[i] <= M) {
        i += 1;
    }
    int s = i - 1;
    return s;
}
```

>_ Solution

L'algorithme est composé d'une boucle **while** qui va parcourir une liste L jusqu'à trouver une valeur qui soit supérieure à M . Ainsi, dans le pire scénario, l'algorithme parcourt toute la liste, et a donc une complexité de $O(n)$, n étant la taille de la liste.

Question 9: (🕒 10 minutes) Complexité

Python :

```
#Entrée: L et M sont 2 listes de nombre entiers
def algo3(L, M):
    n = len(L)
    m = len(M)
    for i in range(n):
        L[i] = L[i]*2
    for j in range(m):
        M[j] = M[j]%2
```

Java :

```
public static void algo3(int[] L, int[] M) {
    int n = L.length;
    int m = M.length;
    for (int i=0; i < n; i++){
        L[i] = L[i]*2;
    }
    for (int j=0; j < m; j++){
        M[j] = M[j]%2;
    }
}
```

>_ Solution

L'algorithme est composé de 2 boucles, une qui parcourt **L** et l'autre qui parcourt **M**. Ainsi pour n et m étant les tailles respectives de **L** et de **M**, on a que la complexité est $O(n) + O(m) = O(\max\{n, m\})$. Ainsi la taille la plus grande domine la taille la plus petite.

Question 10: (🕒 10 minutes) Complexité (Optionnel)

Python :

```
# Entrée: n un nombre entier
def algo4(n):
    m = 0
    for i in range(n):
        for j in range(i):
            m += i+j
    return m
```

Java :

```
public static int algo4(int n) {
    int m = 0;
    for (int i=0; i < n; i++){
        for (int j=0; j < i; j++){
            m += i+j;
        }
    }
    return m;
}
```

>_ Solution

L'algorithme est composé de 2 boucles **imbriquées**. Cela veut dire que nous parcourons la liste un maximum de $n \times n$ fois, n étant la taille de la liste. La complexité de l'algorithme est ainsi $O(n^2)$.

4 Récursion (10 minutes)

Le but principal de la récursion est de résoudre un gros problème en le divisant en plusieurs petites parties à résoudre.

Pour vous donner une idée de ce qu'est la récursion, pensez au travail du facteur. Chaque matin, il doit délivrer le courrier à plusieurs maisons. Il a certainement une liste de toutes les maisons du quartier par où il doit passer dans l'ordre. Par conséquent, il se rend devant une maison, pose le courrier puis va à la prochaine maison figurant sur sa liste. Ce problème est itératif car nous pouvons l'exprimer avec la boucle `for` : Pour chaque maison de sa liste, le facteur dépose le courrier.

```
maisons = ["A", "B", "C", "D"]

def delivrer_courrier_iteratively():
    for maison in maisons:
        print("Courrier délivré à la maison ", maison)

delivrer_courrier_iteratively()
```

Maintenant, imaginons que des stagiaires viennent aider le facteur à délivrer le courrier. Par conséquent, le facteur peut diviser son travail entre ses stagiaires. Pour ce faire, il attribue tout le travail de livraisons à un seul stagiaire qui doit déléguer son travail à deux autres stagiaires. Ces deux autres stagiaires ayant deux maisons à délivrer peuvent également déléguer leur travail à deux autres nouveaux stagiaires. Ces derniers, n'ayant chacun qu'une seule maison à délivrer doivent effectuer cette tâche chacun de leur côté. Ainsi, le facteur a reçu l'aide de 7 stagiaires : 3 délégateurs et 4 travailleurs.

Vous pensez certainement que cette manière de réfléchir est bizarre car vous auriez directement pensé que chaque stagiaire devra délivrer le courrier à une des 4 maisons de la liste. Cependant, ne connaissant pas le nombre de stagiaires travailleurs nécessaires, il est plus simple de commencer par un délégué et continuer à ajouter des délégués jusqu'à ce qu'il ne reste plus que la tâche à faire.

L'algorithme récursif suivant donne le même résultat que la fonction `delivrer_courrier_iteratively`, mais est un peu plus rapide. En effet, le courrier est livré plus vite à chaque maison.

```
maisons = ["A", "B", "C", "D"]

def delivrer_courrier_recursively(maisons):
    # Stagiaire travailleur livrant
    if len(maisons) == 1:
        maison = maisons[0]
        print("Courrier livré à la maison", maison)

    # Stagiaire délégué divisant sa tâche en deux
    else:
        mid = len(maisons) // 2
        first_half = maisons[:mid]
        second_half = maisons[mid:]

        # Stagiaire délégué déléguant ses deux parties
        # de tâche à deux autres stagiaires
        delivrer_courrier_recursively(first_half)
        delivrer_courrier_recursively(second_half)

delivrer_courrier_recursively(maisons)
```

4.1 Différence entre Boucle et Récursion

Une boucle `for` sert principalement à itérer des séquences de données pour les analyser ou manipuler. Par séquence, on entend un string, une liste, un tuple, un dictionnaire ou autre. En d'autres termes, une boucle passe d'une donnée à l'autre et effectue une opération sur chaque donnée. Ainsi, la boucle `for` se termine à la fin de la séquence.

Maintenant, une fonction récursive peut faire la même chose mais de manière plus efficace pour les plus grandes données. La principale différence entre une boucle et une fonction récursive est la façon dont elles se terminent. Une boucle s'arrête généralement à la fin d'une séquence alors qu'une fonction récursive s'arrête dès que la "base condition" est vraie.

Le but est que la fonction récursive se rappelle à chaque fois avec de nouveaux arguments ou qu'elle retourne une valeur finale.

Question 11: (🕒 10 minutes) Fibonacci

La suite de Fibonacci est définie récursivement par :

- si n est 0 ou 1 : $\text{fibonacci}(0) = \text{fibonacci}(1) = 1$
- si n au moins égal à 2, alors ; $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

Écrivez une fonction récursive qui calcule une suite de Fibonacci selon un nombre n donné. Ensuite, calculez la complexité de l'algorithme.

💡 Conseil

Pour la complexité, aidez-vous d'un exemple.

Pour formaliser la formule de complexité, on peut poser que $T(n)$ énumère le nombre d'opérations requises pour calculer $\text{fibonacci}(n)$. Ainsi, $T(n) = T(n-1) + T(n-2) + c$, c étant une constante. Vous pouvez ainsi énumérer le nombre d'opérations pour $\text{fibonacci}(3)$, $\text{fibonacci}(4)$... et essayer de trouver la complexité en terme de $O()$.

Solutions :

— Python :

```
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

— Python :

```
public static int fibonacci(int n) {
    if (n == 0 | n == 1) {
        return n;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

La complexité de cet algorithme est $O(2^n)$.

5 Algorithmes de Tri (60 minutes)

Question 12: (🕒 20 minutes) Tri à bulles (Insertion Sort)

Le tri à bulles consiste à parcourir une liste et à comparer ses éléments. Le tri est effectué en permutant les éléments de telle sorte que les éléments les plus grands soient placés à la fin de la liste.

Concrètement, si un premier nombre x est plus grand qu'un deuxième nombre y et que l'on souhaite trier l'ensemble par ordre croissant, alors x et y sont mal placés et il faut les inverser. Si, au contraire, x est plus petit que y , alors on ne fait rien et l'on compare y à z , l'élément suivant.

Soit la liste l suivante, trier les éléments de la liste suivante en utilisant un tri à bulles. Combien d'itération effectuez-vous ?

— Python :

```
def tri_bulle(l):
    for i in range(len(l)):
        #TODO: Code à compléter

if __name__ == "__main__":
    l = [1, 2, 4, 3, 1]
    tri_bulle(l)
    print(liste_triee)
```

— Java :

```
public class Main {
    public static void tri_bulle(int[] l) {
        for (int i = 0; i < l.length - 1; i++) {
            //TODO: Code à compléter
        }
    }
}
```

```

    }

    public static void printArray(int l[]){
        int n = l.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    public static void main(String[] args){
        int[] l = {1, 2, 4, 3, 1};
        tri_bulle(l);
        printArray(l);
    }
}

```

Conseil

En Java, utilisez une variable temporaire **temp** afin de faire l'échange de valeur entre deux cases dans une liste.

Solutions :

Python :

```

def tri_bulle(l):
    for i in range(1, len(l)):
        # Les i derniers éléments sont dans leur bonne position
        for j in range(0, n-i-1):

            # parcourir la liste de 0 à n-i-1
            # Echanger si l'élément trouvé est supérieur
            # au prochain élément
            if l[j] > l[j+1] :
                l[j], l[j+1] = l[j+1], l[j]

if __name__ == "__main__":
    l = [1, 2, 4, 3, 1]
    tri_bulle(l)
    print(liste_triee)

```

Java :

```

public class Main {
    public static void tri_bulle(int[] l) {
        for (int i = 0; i < l.length - 1; i++) {
            for (int j = 0; j < n-i-1; j++) {
                if (l[j] > l[j+1]) {
                    // échange l[j+1] et l[i]
                    int temp = l[j];
                    l[j] = l[j+1];
                    l[j+1] = temp;
                }
            }
        }
    }

    public static void printArray(int l[]){
        int n = l.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    public static void main(String[] args){
        int[] l = {1, 2, 4, 3, 1};
        tri_bulle(l);
    }
}

```

```

        printArray(l);
    }
}

```

L'algorithme a une complexité de $O(n^2)$ car il contient deux boucles qui parcourent la liste.

Question 13: (🕒 20 minutes) Tri par insertion (Insertion Sort)

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le i -ème élément, les éléments qui le précèdent sont déjà triés. Pour faire l'analogie avec l'exemple du jeu de cartes, lorsqu'on est à la i -ème étape du parcours, le i -ème élément est la carte saisie, les éléments précédents sont la main triée et les éléments suivants correspondent aux cartes encore en désordre sur la table.

L'objectif d'une étape est d'insérer le i -ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

Compléter le code suivant pour trier la liste `l` définie ci-dessous en utilisant un tri par insertion. Combien d'itérations effectuez-vous ?

— Python :

```

def tri_insertion(l):
    for i in range(1, len(l)):
        #TODO: Code à compléter

if __name__ == "__main__":
    l = [2, 43, 1, 3, 43]
    tri_insertion(l)
    print(l)

```

— Java :

```

public class Main {
    public static void tri_insertion(int[] l) {
        for (int i = 1; i < l.length; i++) {
            //TODO: Code à compléter
        }
    }

    public static void printArray(int l[]){
        int n = l.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    public static void main(String[] args){
        int[] l = {2, 43, 1, 3, 43};
        tri_insertion(l);
        printArray(l);
    }
}

```

💡 Conseil

Référez vous à la figure du dessous pour un exemple de tri par insertion.

Solutions :

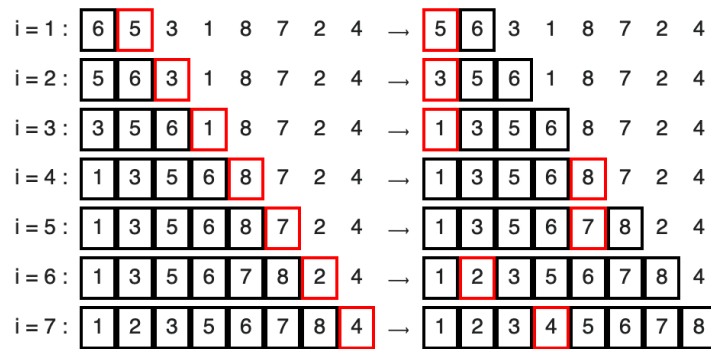
— Python :

```

def tri_insertion(l):
    for i in range(1, len(l)):
        key = l[i]
        j = i-1

        while j >= 0 and key < l[j] :
            l[j + 1] = l[j]

```



```

        j -= 1
        l[j + 1] = key

if __name__ == "__main__":
    l = [2, 43, 1, 3, 43]
    tri_insertion(l)
    print(l)

```

— Java :

```

public class Main {
    public static void tri_insertion(int[] l) {
        for (int i = 1; i < l.length; i++){
            int key = l[i];
            int j = i - 1;

            while (j >= 0 && l[j] > key) {
                l[j + 1] = l[j];
                j = j - 1;
            }
            l[j + 1] = key;
        }
    }

    public static void printArray(int l[]){
        int n = l.length;
        for (int i = 0; i < n; ++i)
            System.out.print(l[i] + " ");

        System.out.println();
    }

    public static void main(String[] args){
        int[] l = {2, 43, 1, 3, 43};
        tri_insertion(l);
        printArray(l);
    }
}

```

La complexité de l'algorithme est de $O(n^2)$ car nous utilisons 2 boucles imbriquées, qui dans le pire des cas, parcourent la liste deux fois.

Question 14: (🕒 30 minutes) Tri fusion (Merge Sort)

À partir de deux listes triées, on peut facilement construire une liste triée comportant les éléments issus de ces deux listes (leur *fusion*). Le principe de l'algorithme de tri fusion repose sur cette observation : le plus petit élément de la liste à construire est soit le plus petit élément de la première liste, soit le plus petit élément de la deuxième liste. Ainsi, on peut construire la liste élément par élément en retirant tantôt le premier élément de la première liste, tantôt le premier élément de la deuxième liste (en fait, le plus petit des deux, à supposer qu'aucune des deux listes ne soit vide, sinon la réponse est immédiate).

Les pas de l'algorithme sont comme suit :

1. Si le tableau n'a qu'un élément, il est déjà trié.
2. Sinon, séparer le tableau en deux parties à peu près égales.
3. Trier récursivement les deux parties avec l'algorithme du tri fusion.

4. Fusionner les deux tableaux triés en un seul tableau trié.

Soit la liste l suivante, trier les éléments de la liste suivante en utilisant un tri à bulles. Combien d'itération effectuez-vous ?

— Python :

```
def merge(partie_gauche, partie_droite):
    #TODO: Code à compléter
def tri_fusion(l):
    #TODO: Code à compléter

if __name__ == "__main__":
    l = [38, 27, 43, 3, 9, 82, 10]
    print(tri_fusion(l))
```

— Java :

```
public class Main {
    // Fusionne 2 sous-listes de arr[].
    // Première sous-liste est arr[l..m]
    // Deuxième sous-liste est arr[m+1..r]
    public static void merge(int arr[], int l, int m, int r) {
        //TODO: Code à compléter
    }

    // Fonction principale qui trie arr[l..r] en utilisant
    // merge()
    public static void tri_fusion(int arr[], int l, int r){
        //TODO: Code à compléter
    }

    public static void printArray(int l[]){
        int n = l.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    public static void main(String[] args){
        int[] l = {38, 27, 43, 3, 9, 82, 10};
        tri_fusion(l);
        printArray(l);
    }
}
```



Conseil

- L'algorithme est récursif.
- Revenez à la visualisation de l'algorithme dans les diapositives pour comprendre comment marche concrètement le tri fusion.

Solutions :

Python :

```
def merge(partie_gauche, partie_droite):
    # créer la liste qui sera retournée à la fin
    liste_fusionnee = []

    # définir un compteur pour l'index de la liste de gauche
    compteur_gauche = 0
    # pareil pour la liste de droite
    compteur_droite = 0

    longueur_gauche = len(partie_gauche)
    longueur_droite = len(partie_droite)

    # continuer jusqu'à ce que l'un des index (ou les deux) atteigne l'une des longueurs
    ↪ (ou les deux)
    while compteur_gauche < longueur_gauche and compteur_droite < longueur_droite:
        # comparer les éléments actuels, ajouter le plus petit à la liste fusionnée
        # et augmenter le compteur de cette liste
```

```

    if partie_gauche[compteur_gauche] < partie_droite[compteur_droite]:
        liste_fusionnee.append(partie_gauche[compteur_gauche])
        compteur_gauche += 1
    else:
        liste_fusionnee.append(partie_droite[compteur_droite])
        compteur_droite += 1

# s'il y a encore des éléments dans les listes, il faut les ajouter à la liste
↪ fusionnée
liste_fusionnee += partie_gauche[compteur_gauche:longueur_gauche]
liste_fusionnee += partie_droite[compteur_droite:longueur_droite]

return liste_fusionnee # retourner la liste fusionnée

def tri_fusion(l):
    # compléter la fonction
    longueur = len(l) # calculer la longueur de la liste
    # s'il n'y a pas plus d'un élément, retourner la liste
    if longueur == 1 or longueur == 0:
        return l
    # sinon, diviser la liste en deux
    elif longueur > 1:
        # convertir la variable en nombre entier (l'index ne peut pas être un nombre à
        ↪ virgule)
        index_milieu = int(longueur / 2)
        # la partie gauche va du 1er élément à celui du milieu
        partie_gauche = l[0:index_milieu]
        # la partie droite va du milieu à la fin de la liste
        partie_droite = l[index_milieu:longueur]

        # appeler la fonction tri_fusion à nouveau sur la partie gauche (récursivité)
        partie_gauche_triee = tri_fusion(partie_gauche)
        # même chose pour la partie droite
        partie_droite_triee = tri_fusion(partie_droite)

        liste_fusionnee = merge(partie_gauche_triee, partie_droite_triee) # enfin,
        ↪ joindre les 2 parties

        # retourner le résultat
        return liste_fusionnee

if __name__ == "__main__":
    l = [38, 27, 43, 3, 9, 82, 10]
    print(tri_fusion(l))

```

Java :

```

public class Main {
    // Fusionne 2 sous-listes de arr[].
    // Première sous-liste est arr[l..m]
    // Deuxième sous-liste est arr[m+1..r]
    public static void merge(int arr[], int l, int m, int r) {
        // Trouver la taille des deux sous-listes à fusionner
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Créer des listes temporaires */
        int L[] = new int[n1];
        int R[] = new int[n2];

        /*Copier les données dans les sous-listes temporaires */
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        /* Fusionner les sous-listes temporaires */

        // Indexes initiaux de la première et seconde sous-liste
        int i = 0, j = 0;

        // Index initial de la sous-liste fusionnée
        int k = l;
        while (i < n1 && j < n2) {

```



```

        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copier les éléments restants de L[] */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copier les éléments restants de R[] */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Fonction principale qui trie arr[l..r] en utilisant
// merge()
public static void tri_fusion(int arr[], int l, int r){
    if (l < r) {
        // Trouver le milieu de la liste
        int m = (l + r) / 2;

        // Trier la première et la deuxième parties de la liste
        tri_fusion(arr, l, m);
        tri_fusion(arr, m + 1, r);

        // Fusionner les deux parties
        merge(arr, l, m, r);
    }
}

public static void printArray(int l[]){
    int n = l.length;
    for (int i = 0; i < n; ++i)
        System.out.print(arr[i] + " ");

    System.out.println();
}

public static void main(String[] args){
    int[] l = {38, 27, 43, 3, 9, 82, 10};
    tri_fusion(l);
    printArray(l);
}
}

```

Le tri fusion est un algorithme récursif. Ainsi, nous pouvons exprimer la complexité temporelle via une relation de récurrence : $T(n) = 2T(n/2) + O(n)$. En effet, l'algorithme comporte 3 étapes :

1. la "Divide Step", qui divise les listes en deux sous-listes, et cela prend un temps constant
2. la "Conquer Step", qui trie récursivement les sous-listes de taille $n/2$ chacune, et cette étape est représentée par le terme $2T(n/2)$ dans l'équation.
3. l'étape où l'on fusionne les listes, qui prend $O(n)$.

La solution à cette équation est $O(n \log n)$.