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. les algorithmes de tri,
4. une mise en pratique des notions de récursivité.

Les exercices sont disponibles en Python et en Java.

1 Les fonctions (Java ou Python)

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 affiche **"Hello World"** sur l'écran une fois la fonction invoquée.

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 d'afficher la phrase **"Hello World"**. Puis nous appelons cette fonction :

1.2.2 Java

En Java, les fonctions ont la structure suivante :
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 :
Question 1: (🕓 *2 minutes*) **Python ou Java**
Créez une fonction du nom de votre choix qui affiche votre prénom et appelez cette fonction.
Solutions :

1. **Python :**
2. **Java :**

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.

1.3.2 Java

Nous reprenons l'exemple précédent et le réécrivons en Java :
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**
>
> N'oubliez pas de typez vos arguments en Java !

Solutions :

1. **Python :**
2. **Java :**

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 :
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 :
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.
Nous convertissons le code Python ci-dessus en code Java :
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**, **print y**.

> **Conseil**

Solutions :

1. **Python :**
2. **Java :**

1.5 Arguments par défaut (Python uniquement)

Une fonction peut avoir des arguments par défaut, c'est-à-dire des arguments optionnels qui prennent soit une valeur par défaut, soit une valeur donnée par l'utilisateur. Par exemple, la fonction suivante a un argument par défaut de nom **name**, qui vaut **"Bjarne"** par défaut.
La première fois, nous appelons la fonction avec **"Ken"** comme argument, nous affichons donc **Mon nom est Ken**, la deuxième fois nous ne donnons par d'argument, l'argument par défaut est donc utilisé et nous affichons **Mon nom est Bjarne**.

Question 4: (*5 minutes*) **Python** Ecrivez une fonction `f` qui prend un argument `x` avec une valeur par défaut `0` et qui retourne `x` au carré. Appelez cette fonction avec un argument puis sans.

> **Conseil**

Solutions :

1.6 Exercices d'applications

Question 5: (*5 minutes*) **Python ou Java** Complètez la fonction ci-dessous afin qu'elle retourne le cube de l'argument `x`.
Python :
Java :

> **Conseil**
>
> La librairie java.lang.Math peut vous être utile pour la version Java.

Solutions :
- **Python :**
- **Java :**

Question 6: (*5 minutes*) **Python ou Java** Complétez la fonction ci-dessous afin qu'elle affiche les nombres de 0 à 10.
Python :
Java :

> **Conseil**

Solutions :
- **Python :**
- **Java :**

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 élèments.
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 élements à rajouter entre crochets
- **Suppression avec indexation** : **my_list[0:2] = []**, sans rien entre les crochets

Question 7: (*5 minutes*) **Comportement des listes en Python**
Après l'exécution du code ci-dessous, à quoi va ressembler **my_list** ?

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

Solutions :

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 spécifiez le contenu de la liste à l'initialisation. Après cela, la taille de la liste ne pourra être modifiée. 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**. 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 : 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 :

2.2.2 Java

3 Complexité

Question 8: (*20 minutes*) **Complexité**
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.

1. **Python :**
   Java :
2. **Python :**
   Java :
3. **Python :**
   Java :
4. **Python :**
   Java :

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 : $O(2n^2 + 5n + 50) = O(n^2)$

Solutions :

4 Récursion

Le but principal de la récursion est de résoudre un gros problème en le divisant en plusieurs petites parties à résoudre.
Pour vous donnez 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 courier 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.
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 à 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 pensez que chaque stagiaire devra délivrer le courrier à une des 4 maisons de la liste. Cependant, ne connaissant pas le

nombre de stagiaire travailleurs nécessaires, il est plus simple de commencer par un délégateur et continuez à ajouter des délégateurs 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.

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 disctinte 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 9: (🕓 *10 minutes*) **Fibonacci**
La suite de Fibonacci est définie récursivement par :

- si n est 0 ou 1 : fibo(0) = fibo(1) = 1
- si n au moins égal à 2, alors ; fibo (n) = fibo(n - 1) + fibo(n - 2)

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

Solutions :

- **Python :**
- **Python :**
  La complexité de cet algorithme est $O(2^n)$.

5 Algorithmes de Tri

Question 10: (🕓 *10 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 :**
- **Java :**

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

Solutions :

Python :

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

Question 11: (🕓 *10 minutes*) **Tri par insertion (Insertion Sort)** Compléter le code suivant pour trier la liste **l** définie ci-dessous en utilisant un tri par insertion. Combien d'itérations effectuez-vous ?

— **Python :**
— **Java :**

Conseil

Solutions :
— **Python :**
— **Java :**
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 12: (*20 minutes*) **Tri fusion (Merge Sort)** 4.3 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.

Source : https ://fr.wikipedia.org/wiki/Tri$_{fusion}$
Soit la liste **l** suivante, trier les éléments de la liste suivante en utilisant un tri à bulles. Combien d'itération effectuez-vous ?
— **Python :**
— **Java :**

Conseil

Solutions : Python :
Java :
Le tri fusion est un algorithme récursif. Ainsi, nous pouvons exprimer la complexité temporelle via une relation de récurence : $T(n) = 2T(n/2) + O(n)$. 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 $O(n \log n)$.