Algorithmes et Pensée Computationnelle

Algorithmes de tri 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. la complexité des algorithmes,
2. une introduction à la récursion et
3. les algorithmes de tri

Les exercices sont disponibles en Python et en Java.

1 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 1: (*10 minutes*) **Complexité**
Python :

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

Java :

```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 2: (*10 minutes*) **Complexité**
Python :

```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 :

```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 supérieur 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 3: (*10 minutes*) **Complexité**

Python :

```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 :

```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 4: (*10 minutes*) **Complexité (Optionnel)**

Python :

```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 :

```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 $O(n^2)$.

2 Récursion (15 minutes)

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.

```python
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 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.

```python
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 délégateur divisant sa tâche en deux
    else:
        mid = len(maisons) // 2
        first_half = maisons[:mid]
        second_half = maisons[mid:]

        # Stagiaire délégateur délégant ses deux parties
        # de tâche à deux autres stagiaires
        delivrer_courrier_recursively(first_half)
        delivrer_courrier_recursively(second_half)

delivrer_courrier_recursively(maisons)
```

2.1 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 5: (*15 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.

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 **fibonacci(n)**. Ainsi, $T(n) = T(n-1) + T(n-2) + c$, c étant une constante. Vous pouvez alors énumérer le nombre d'opérations pour **fibonacci(3)**, **fibonacci(4)**... et esssayer de trouver la complexité en terme de $O()$.

Solutions :
- **Python :**

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

- **Python :**

```java
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 $O(2^n)$.

3 Algorithmes de Tri (60 minutes)

Question 6: (*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 élements 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 élement
            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 7: (🕓 *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 :**

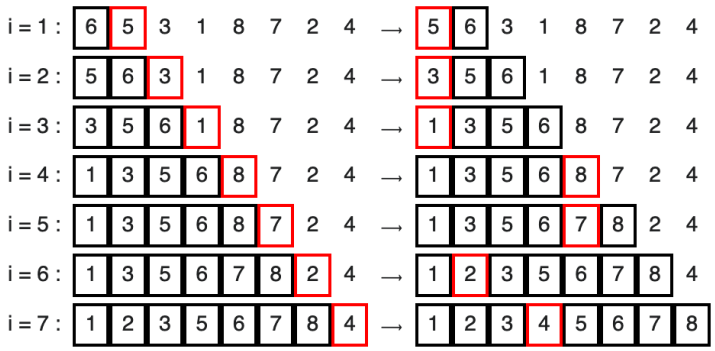
```
def tri_insertion(l):
    for i in range(1, len(l)):
        #TODO: Code à compléter

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 :**

```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 :**

```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 8: (*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 :**

```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 :**

```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 :

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

        /* Copier les élements 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 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 $O(n \log n)$.