Algorithmes et Pensée Computationnelle

Algorithmes de tri et Complexité - exercices avancés

Le but de cette série d'exercices est d'aborder les notions présentées durant la séance de cours. Cette série d'exercices sera orientée autour des points suivants :

1. la complexité des algorithmes,
2. la récursivité et
3. les algorithmes de tri

Les languages de programmation qui seront utilisés pour cette série d'exercices sont Java et Python.
Le temps indiqué (🕓) est à titre indicatif.

1 Récursivité (15 minutes)

Question 1: (🕓 *5 minutes*) **Fibonacci**
La suite de Fibonacci est définie récursivement par les propriétés suivantes :

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

Voici son implémentation en Java :

```
1  public static int fibonacci(int n) {
2    if(n == 0 | n == 1){
3      return n;
4    } else{
5      return fibonacci(n−1) + fibonacci(n−2);
6    }
7  }
8
9
```

Quel est la complexité de l'algorithme ci-dessus ?

Conseil

Aidez-vous d'un exemple (**fibonacci(3)**, **fibonacci(4)**,...)
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 essayer de trouver la complexité en terme de $O()$.

1. $O(n^2)$
2. $O(n)$
3. $O(\log(n))$
4. $O(2^n)$

Solution

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

Question 2: (🕓 *15 minutes*) - Fibonacci et utilisation de la mémoire (Python)
Le calcul des éléments de la suite de Fibonacci en utilisant l'algorithme de la question 1 est fastidieux. Cela est dû au fait que certaines opérations sont effectuées plusieurs fois. Vous pouvez le vérifier en passant un grand nombre à la fonction **fibonacci**. Par exemple **fibonacci(50)** peut prendre plusieurs heures avant de donner un résultat !

En Python, simplifiez cet algorithme en écrivant une fonction **fibonacci** qui prend en entrée un nombre **n** et un dictionnaire **previous_fibonacci**. Le dictionnaire stockera chacune des opérations des appels récursifs. Ce dictionnaire aura pour clé un nombre **n** et pour valeur le résultat de l'appel de la fonction **fibonacci(n)**.
Avant de faire appel à la fonction de façon récursive, vérifiez que le nombre passé en paramètre n'existe pas dans le dictionnaire **previous_fibonacci**.

Conseil

Au lancement du programme, **previous_fibonacci** aura pour valeur **{0:0, 1:1}** ce qui correspond aux conditions de sortie des appels récursifs.
La fonction **fibonacci** retournera le dernier élément du dictionnaire **previous_fibonacci**.

Solution

```
counter = 0
previous_fibonacci = {0:0, 1:1}

def fibonacci(number, previous):
    global counter
    counter += 1
    if number not in previous:
        previous[number] = fibonacci(number - 1, previous) + fibonacci(number - 2, previous)
        print(previous)

    return previous[number]


print(fibonacci(50, previous=previous_fibonacci))
```

2 Algorithmes de Tri (40 minutes)

Question 3: (*20 minutes*) **Tri à bulles (Bubble Sort) - Java** Optionnel

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 = [1, 2, 4, 3, 1]**, triez les éléments de la liste en utilisant un tri à bulles. Combien d'itérations effectuez-vous ?

— **Java :**

```
public class question3 {
    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(l[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 que vous nommerez **temp** afin de faire l'échange de valeur entre deux éléments de la liste.

Solution

Java :

```java
public class question3 {
    public static void tri_bulle(int[] l) {
        int n = l.length;
        for (int i = 0; i < n - 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(l[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 4: (*20 minutes*) **Tri par insertion - 2 (Insertion Sort)**

Dans l'algorithme de tri par insertion, 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 le précède. 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 :**

```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 :**

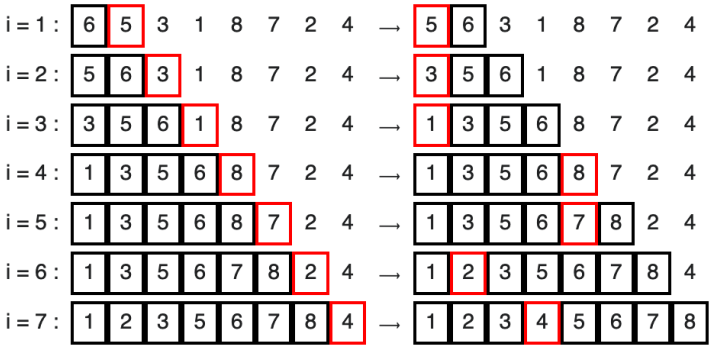
```
public class question4 {
    public static void tri_insertion(int[] l) {
        for (int i = 1; i < l.length; i++){
            //TODO: Code à compléter
        }
    }

    public static void printArray(int l[]){
        int n = l.length;
        for (int i = 0; i < n; ++i)
            System.out.print(l[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.
Référez vous aussi aux diapositives 18 à 72 du cours.



Solution

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 question4 {
    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] + " ");
    }

    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 5: (*20 minutes*) **Tri fusion (Merge Sort) - java**

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 étapes à suivre pour implémenter l'algorithme sont les suivantes :

1. Si le tableau n'a qu'un élément, il est déjà trié.
2. Sinon, séparer le tableau en deux parties plus ou moins égales.
3. Trier récursivement les deux parties avec l'algorithme de tri fusion.
4. Fusionner les deux tableaux triés en un seul tableau trié.

Soit la liste `l` suivante [38, 27, 43, 3, 9, 82, 10], triez les éléments de la liste en utilisant un tri fusion. Combien d'itération effectuez-vous ?

— **Java :**

```java
public class question5 {
    // 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(l[i] + " ");

        System.out.println();
    }


    public static void main(String[] args){
        int[] l = {38, 27, 43, 3, 9, 82, 10};
        tri_fusion(l, 0, l.length - 1);
        printArray(l);
    }
}
```

Conseil

- L'algorithme est récursif.
- Revenez à la visualisation de l'algorithme dans les diapositives 83 à 111 pour comprendre comment marche concrètement le tri fusion.

Solution

Java :

```java
// Solution question 5 – 1/2
public class question5 {
    // 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 les 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);
        }
    }
```

Solution

```
1  // Solution question 5 – 2/2
2  public static void affiche_liste(int l[]) {
3     int n = l.length;
4     for (int i = 0; i < n; ++i)
5        System.out.println(l[i] + " ");
6  }
7
8
9  public static void main(String[] args) {
10    int[] l = {38, 27, 43, 3, 9, 82, 10};
11    tri_fusion(l, 0, l.length – 1);
12    affiche_liste(l);
13 }
14 }
```

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

1. "Divide Step", qui divise les listes en deux sous-listes, et cela prend un temps constant
2. "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. La dernière étape consiste à fusionner les listes, sa complexité est de $O(n)$.

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