Algorithmes et Pensée Computationnelle

Algorithmes de tri et Complexité

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.

1 Complexité (30 minutes)

Pour chacun des programmes 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 des cas**. 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: L est une liste de nombres 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 supérieure à `M`. Ainsi, dans le pire des cas, 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 une liste `L` et l'autre qui parcourt une liste `M`. Soient n et m les tailles respectives de `L` et de `M`, on obtient une complexité de $O(n)+O(m) = O(\max\{n, m\})$. Ainsi, l'élément ayant la plus grande complexité sera utilisé pour déterminer la complexité de l'algorithme dans son ensemble.

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 Récursivité (15 minutes)

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

Question 5: (*5 minutes*) **Somme des chiffres**

Écrivez un algorithme récursif en Python ou en Java qui prend un nombre et retourne la somme des chiffres dont il est composé. Par exemple, la somme des chiffres de 126 est : 1+2+6 = 9.

Conseil

Pour obtenir les chiffres qui composent un nombre, utilisez l'opérateur `%` (modulo - `https://fr.wikipedia.org/wiki/Modulo_(op%C3%A9ration)`).
Pour obtenir le nombre 12 à partir du nombre 126, il vous suffit de faire la division entière par 10. En Python, on utilise l'opérateur \\ : 126 \\ 10 = 12. En Java, la division entre deux variables de type `int` est entière, et vous n'aurez ainsi qu'à utiliser l'opérateur de division normal \ : 126 \ 10 = 12

Solution

Python :

```python
def sum_digits(number):
    if number == 0:
        return 0
    else:
        return (number % 10) + sum_digits(number // 10)

print(sum_digits(126))
```

Java :

```java
public class Main {
    public static int sum_digits(int number) {
        if(number == 0){
            return 0;
        } else{
            return (number%10) + sum_digits(number/10);
        }
    }

    public static void main(String[] args){
        System.out.println(sum_digits(126));
    }
}
```

Question 6: (*10 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 :

```
9   public static int fibonacci(int n) {
10    if(n == 0 | n == 1){
11      return n;
12    } else{
13      return fibonacci(n−1) + fibonacci(n−2);
14    }
15  }
```

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 esssayer de trouver la complexité en terme de $O()$.

Solution

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

3 Algorithmes de Tri (60 minutes)

Question 7: (*10 minutes*) **Tri par insertion - 1 (Python)**

Soit un nombre entier **n**, et une liste triée **l**. Ecrivez un programme qui insère la valeur **n** dans la liste **l** tout en s'assurant que la liste **l** reste triée.

Exemple

En passant les arguments suivants à votre programme : n=5 et l=[2,4,6]. Ce dernier devra retourner **l =[2,4,5,6]**

Solution

```
1   def insertion_entier(liste, number):
2     # ajoute un élément à la liste
3     liste.append(number)
4     n = len(liste) − 1
5     while n > 0 and liste[n − 1] > number:
6       liste[n] = liste[n − 1]
7       n −= 1
8     liste[n] = number
9     return liste
10
11  print(insertion_entier([2, 4, 6], 1))
```

Question 8: (*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 :**

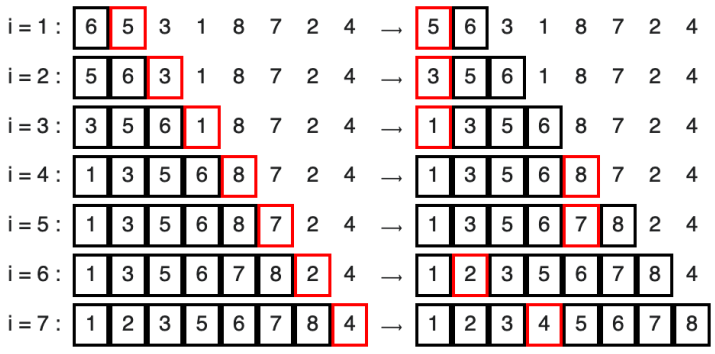
```java
public class Main {
    public static void tri_insertion(int[] l) {
        for (int i = 1; i < l.length; i++){
            //TODO: Code à compléter
        }
    }

    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.



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 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] + " ");
    }

    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 9: (*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 étapes à suivre à implémenter l'algorithme sont les suivants :

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

— **Python :**

```
35          def merge(partie_gauche, partie_droite):
36              #TODO: Code à compléter
37          def tri_fusion(l):
38              #TODO: Code à compléter
39
40          if __name__ == "__main__":
41              l = [38, 27, 43, 3, 9, 82, 10]
42              print(tri_fusion(l))
43
```

— **Java :**

```
53          public class Main {
54              // Fusionne 2 sous-listes de arr[].
55              // Première sous-liste est arr[l..m]
56              // Deuxième sous-liste est arr[m+1..r]
57              public static void merge(int arr[], int l, int m, int r) {
58                  //TODO: Code à compléter
59              }
60
61              // Fonction principale qui trie arr[l..r] en utilisant
62              // merge()
63              public static void tri_fusion(int arr[], int l, int r){
64                  //TODO: Code à compléter
65              }
66
67              public static void printArray(int l[]){
68                  int n = l.length;
69                  for (int i = 0; i < n; ++i)
70                      System.out.print(arr[i] + " ");
71
72                  System.out.println();
73              }
74
75
76              public static void main(String[] args){
77                  int[] l = {38, 27, 43, 3, 9, 82, 10};
78                  tri_fusion(l);
79                  printArray(l);
80              }
81          }
82
```

Conseil

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

Solution

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):
    # complèter 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))
```

Solution

Java :

```java
// Solution question 9 - 1/2
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 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 9 − 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 $O(n \log n)$.