Algorithmes et Pensée Computationnelle

Algorithmes de tri et Complexité - exercices basiques

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

Pour chacun des programmes ci-dessous, donnés à chaque fois en Python et Java, 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**

Quelle est la complexité du programme ci-dessous ?
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;
}
```

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

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**

Quelle est la complexité du programme ci-dessous ?

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;
}
```

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

Solution

L'algorithme est composé d'une boucle `while` qui va parcourir une liste `L` jusqu'à trouver une valeur qui est 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**

Quelle est la complexité du programme ci-dessous ?

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;
    }
}
```

1. $O(n^2)$
2. $O(n + m)$
3. $O(n)$

4. $O(2^n)$

>_ Solution

L'algorithme est composé de 2 boucles. La première parcourt une liste **L** et multiplie par 2 les éléments de la liste. L'autre parcourt une liste **M** et assigne à chaque élément le reste de la division euclidienne de l'élément par 2. Soient n et m les tailles respectives de **L** et de **M**, on obtient une complexité de $O(n + m)$. 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**

Quelle est la complexité du programme ci-dessous ?

Python :

```python
#Entrée: L est une liste de nombre entiers
def algo4(L):
    n = len(L)
    i = 0
    s = 0
    while i < math.log(n):
        s += L[i]
        i += 1
    return s

```

Java :

```java
    import java.lang.Math;

    public static void algo4(int[] L) {
        int n = L.length;
        int s = 0;
        for (int i=0; i < Math.log(n); i++){
            s += L[i];
        }
    }

```

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

>_ Solution

L'algorithme est composé d'une boucle qui va itérer sur $\log(n)$ éléments et va calculer la somme de ces éléments. Ainsi, l'algorithme a une complexité de $O(\log n)$. Le temps d'exécution de ce programme peut être visualisé sur la courbe jaune du graphe ci-dessous (**??**).

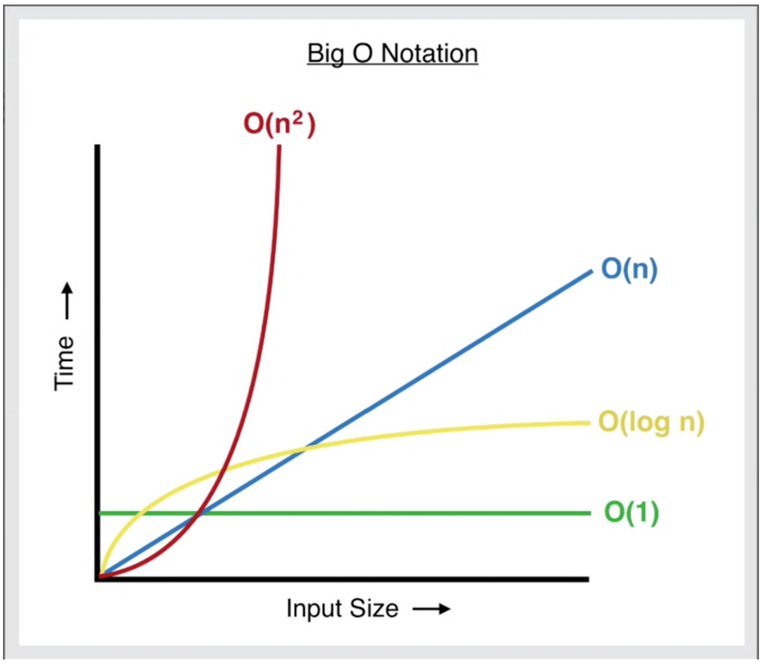

FIGURE 1 – Représentation de complexités temporelles

Question 5: (*10 minutes*) **Complexité**

Quelle est la complexité du programme ci-dessous ?

Python :

```python
# Entrée: n un nombre entier
def algo5(n):
    m = 0
    for i in range(n):
        for j in range(i):
            m += i+j
    return m
```

Java :

```java
public static int algo5(int n) {
    int m = 0;
    for (int i=0; i < n; i++){
        for (int j=0; j < i; j++){
            m += i+j;
        }
    }
    return m;
}
```

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

Solution

L'algorithme est composé de 2 boucles **imbriquées** suivant une suite définie par $\frac{n(n-1)}{2}$. L'algorithme va additionner les index i et j à chaque itération et les rajouter à m. 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 de $O(n^2)$.

2 Récursivité

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

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 :

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

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 $O()$.

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

Solution

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

3 Algorithmes de Tri

Question 7: (*20 minutes*) **Tri à bulles (Bubble Sort) en python**

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 ?

— **Python :**

```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(l)
```

Solution

Python :

```python
def tri_bulle(l):
    n = len(l)

    for i in range(n):
        # 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(l)
```

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

Question 8: (*10 minutes*) **Tri par insertion - 1 (Python)**

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

```python
def insertion_entier(liste, number):
    # TODO : Compléter ici

print(insertion_entier([2, 4, 6], 1))
```

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

```python
def insertion_entier(liste, number):
    # ajoute un élément à la liste
    liste.append(number)
    n = len(liste) - 1
    while n > 0 and liste[n - 1] > number:
        liste[n] = liste[n - 1]
        n -= 1
    liste[n] = number
    return liste

print(insertion_entier([2, 4, 6], 1))
```

Question 9: (*20 minutes*) **Tri fusion (Merge Sort) en python**

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 ?

— **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))
```

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

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))
```