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.
Le temps indiqué (🕓) est à titre indicatif.

1 Complexité (30 minutes)

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 (1).

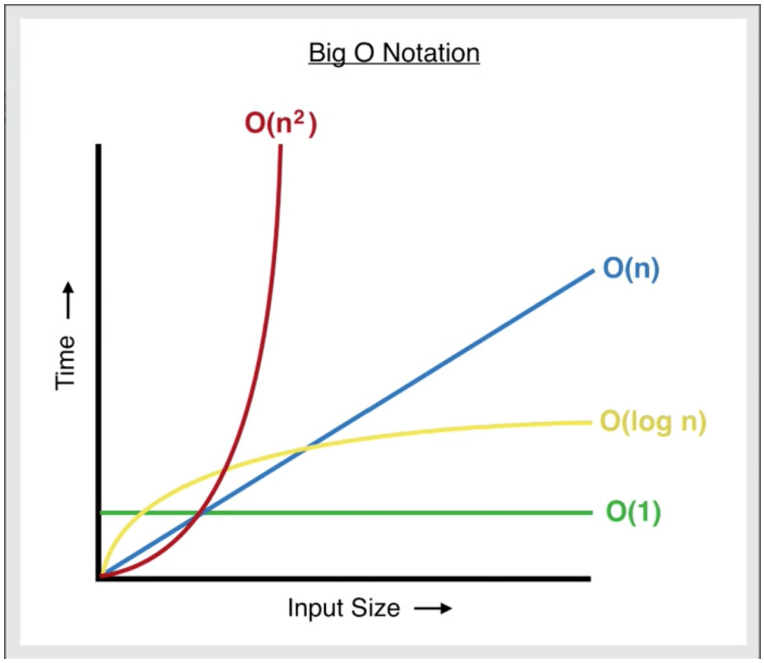

FIGURE 1 – Représentation de complexités temporelles

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

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 :

```
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 $O()$.

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 (60 minutes)

Question 7: (*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.

```
1  def insertion_entier(liste, number):
2    # TODO : Compléter ici
3
4  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

```
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))
```