Complexité algorithmique — Corrigé des exos

Si vous ne pouvez pas battre votre ordinateur aux échecs, défiez-le au kick-boxing.

Exercice 1. — Calculer la complexité algorithmique d'un algorithme extrayant l'initiale d'un mot.

```
1  def Initiale(MOT):
2      return MOT[0]
```

➤ La ligne 2 est une "extraction" : elle compte donc pour une opération élémentaire.

Conclusion : la complexité algorithmique de ce programme est $C(n) = 1$ (avec n la taille du mot) : complexité constante.

Exercice 2. — Calculer la complexité algorithmique d'un algorithme comptant le nombre de voyelles d'un mot.

```
1  def Nvoy(MOT):
2      nbv =0
3      for k in len(MOT):
4          if L[k] in 'aeiouyAEIOUY':
5              nbv =nbv +1
6      return nbv
```

On détaille à présent le nombre d'opérations effectuées par l'algorithme ligne après ligne.

➤ La ligne 2 est une affectation : elle compte donc pour une opération élémentaire.

➤ La boucle est parcourue n fois (avec n la longueur du mot) : la ligne 3 compte pour une opération élémentaire ; la ligne 4 aussi ; et la ligne 5 comporte 2 opérations élémentaires.

Coût de la boucle : $4n$ opérations élémentaires.

Conclusion : la complexité algorithmique de ce programme est $C(n) = 4n + 1$ (avec n la taille du mot), donc $C(n) = O(n)$: complexité linéaire.

Exercice 3. — Calculer la complexité algorithmique d'un algorithme calculant la trace d'une matrice.

```
1  def trace(M):
2      TR =0
3      for i in range(n):
4          TR =TR +M[i][i]
5      return TR
```

On détaille à présent le nombre d'opérations effectuées par l'algorithme ligne après ligne.

➤ La ligne 2 est une affectation : elle compte donc pour une opération élémentaire.

➤ La boucle est parcourue n fois (avec n le nombre de lignes/colonnes de la matrice) : la ligne 3 compte pour une opération élémentaire ; la ligne 4 pour 4 opérations élémentaires (1 affectation, une addition et deux extractions).

Coût de la boucle : $5n$ opérations élémentaires.

Conclusion : la complexité algorithmique de ce programme est $C(n) = 5n+1$ (avec n le nombre de lignes/colonnes de la matrice), donc $C(n) = O(n)$: complexité linéaire.

Exercice 4. — Calculer la complexité algorithmique d'un algorithme du calcul de la somme de deux matrices (carrées de même taille).

```
1  def Mat_Som(A,B):
2      C =[ [0 for i in range(n)] for j in range(n) ]
3      for i in range(n):
4          for j in range(n):
5              C[i][j] =A[i][j] +B[i][j]
6      return C
```

On détaille à présent le nombre d'opérations effectuées par l'algorithme ligne après ligne.

➤ La ligne 2 compte pour n^2 opérations élémentaires.

➤ La première boucle est parcourue n fois (avec n le nombre de lignes/colonnes de la matrice) : à l'intérieur de cette boucle, on parcourt une seconde boucle n fois.*

Calculons le coût algorithmique de la seconde boucle : la ligne 4 compte pour une opération élémentaire ; la ligne 5 pour 6 opérations élémentaires (1 affectation, une addition et quatre extractions).

Coût de la boucle "j" : $7n$ opérations élémentaires.

Coût des boucles "i" et "j" : $7n^2+n$ opérations élémentaires.

Conclusion : la complexité algorithmique de ce programme est $C(n) = 8n^2+n$ (avec n le nombre de lignes/colonnes de chacune des deux matrices), donc $C(n) = O(n^2)$: complexité quadratique.

Exercice 5. — Calculer la complexité algorithmique d'un algorithme du calcul du produit de deux matrices (carrées de même taille).

```
1  def Mat_Prod(A,B):
2      C =[ [0 for i in range(n)] for j in range(n) ]
3      for i in range(n):
4          for j in range(n):
5              for k in range(n):
6                  C[i][j] =C[i][j] +A[i][k] * B[k][j]
7      return C
```

On se base sur les mêmes calculs que dans l'exemple précédent.

Conclusion : la complexité algorithmique de ce programme est $C(n) = 10n^3+2n^2+n$ (avec n le nombre de lignes/colonnes de chacune des deux matrices), donc $C(n) = O(n^3)$: complexité cubique.

Exercice 6. — Calculer la complexité algorithmique d'un algorithme déterminant si un entier supérieur à 4 est de Carmichael ou non.

Evidemment, les choses deviennent plus simples dès que l'on sait ce qu'est un entier de Carmichael...

Attention, cette notion nécessite des connaissances en arithmétique. *

Un entier $c \geqslant 2$ est un **entier de Carmichael** si c n'est pas premier, et si :

$$\forall n \in [\![0, c-1]\!], \qquad n^c \equiv n \ [c]$$

```
1  def Tprem_ou_pas(n):
2      TPREM =True
3      d =2
4      while (d * d <=n) and (TPREM ==True):
5          if n % d ==0:
6              TPREM =False
7          d =d +1
8      return TPREM
9
10 def Carmi_ou_pas(c):
11     TCARMI =True
12     if Tprem_ou_pas(c) ==True:
13         TCARMI =False
14     n =2
15     while (n <c) and (TCARMI ==True):
16         if (n**c - n) % c !=0:
17             TCARMI =False
18         n =n +1
19     return TCARMI
```

Conclusion : la complexité algorithmique de ce programme est de l'ordre de $C(n) = O(n)$ (avec n l'entier que l'on veut tester) : complexité linéaire.

∗. Si vous n'en avez pas encore fait, il faudra attendre la semaine de la rentrée.