Chapitre 8

Tris et complexité

Table des matières

1 Notion de complexité

Définition 1 :

Soit $\mathcal{A}$ un algorithme et d une donnée d'entrée de $\mathcal{A}$. On appelle complexité de $\mathcal{A}$ pour d (notée $C_{\mathcal{A}}(d)$) le nombre de fois que l'opération élémentaire est exécutée lors de l'exécution de $\mathcal{A}$ sur d.

Exemples 1 : Déterminer le nombre d'opérations algébriques dans l'algorithme suivant :

```
s = 5
p = 6
q = (s+p)*(2*s-p)
p = 2*s
s= 3*p-2*q
```

Exemples 2 : Déterminer le nombre d'opérations algébriques dans l'algorithme suivant :

```
s = 0
for i in range(10) :
  s = 2*s - 3*i

print(s)
```

Exemples 3 : Déterminer le nombre d'opérations algébriques dans l'algorithme suivant la valeur de n :

```
def somme (n) :
  s = 0
  for i in range(n) :
    s = 2*s - 3*i
  return s
```

Exercice 1 :

On considère la fonction `recherche_zero` suivante :

```
def recherche_zero (liste) :
  for e in liste :
    if e == 0 :
      return True
  return False
```

1. Tester la fonction avec les listes suivantes :

```
liste_1 = [ 8,6,4,1,0,2,4,6,0,4]
liste_2 = [ 0,1,2,3,4,5]
liste_3 = [ 8,7,9,4,5,1,0]
liste_4 = [ 7,9,4,5,13,4,9,7]
```

2. Déterminer le nombre de comparaisons de la fonction pour les quatre listes précédentes.
3. Pour quelles sortes de listes, le nombre de comparaison est-il maximum ?
4. Pour quelles sortes de listes, le nombre de comparaison est-il minimum ?

Définition 2 :

Complexité dans le pire des cas :
On appelle complexité de $\mathcal{A}$ dans le pire des cas (notée $C_{\mathcal{A},\text{pire}}(n)$ pour une entrée de taille n) la complexité de cet algorithme dans le cas le plus défavorable :

$$C_{\mathcal{A},\text{pire}}(n) = \max_{d \in D(n)} (C_{\mathcal{A}}(d))$$

Définition 3 :

Complexité dans le meilleur des cas :
On appelle complexité de $\mathcal{A}$ dans le meilleur des cas (notée $C_{\mathcal{A},\text{meilleur}}(n)$ pour une entrée de taille n) la complexité de cet algorithme dans le cas le plus favorable :

$$C_{\mathcal{A},\text{meilleur}}(n) = \min_{d \in D(n)} (C_{\mathcal{A}}(d))$$

2 Ordre de grandeur

Le choix de l'opération élémentaire dans le calcul de la complexité d'un algorithme n'est pas un réel choix. La complexité est principalement liée au nombre de boucles.

Exemples 4 : Déterminer le nombre d'opérations algébriques dans les algorithmes suivant la valeur de la taille de la liste :

```
def fonction_1 (liste) :
  s = 0
  n = len(liste)
  for i in range(n) :
    s =  liste[i] + s
  return s

def fonction_2 (liste) :
  s = 0
  n = len(liste)
  for i in range(n) :
    s = 2*s - 3 * liste[i]
  return s
```

Définition 4 :

Domination asymptotique : Le grand O
Soit deux fonctions $f : \mathbb{N}^* \to \mathbb{R}$ et $g : \mathbb{N}^* \to \mathbb{R}$.
On dira que f est dominée asymptotiquement par g si :

$$\exists c \in \mathbb{R}_+^* \text{ , } \exists n_0 \in \mathbb{N}^* \text{ tels que } \forall n \in \mathbb{N}^* \text{ , } n > n_0 \Rightarrow f(n) \leqslant c.g(n)$$

On note $f = O(g)$.

3 Type de complexité :

$O(1)$	complexité constante (indépendante de la taille de la donnée)
$O(\log(n))$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n\log(n))$	complexité quasi-linéaire
$O(n^2)$	complexité quadratique
$O(n^p)$	complexité polynomiale
$(O(2^n))$	complexité exponentielle
$(n!)$	complexité factorielle

4 Estimation du temps de calcul en fonction de la complexité :

Taille n / $C(n)$	$\log(n)$	n	$n\log(n)$	n^2	n^3	2^n
10^2	$6,64\ ns$	$0,1\ \mu s$	$0,66\ \mu s$	$10\ \mu s$	$1\ ms$	4.10^{13} années
10^3	$9,96\ ns$	$1\ \mu s$	$9,96\ \mu s$	$1\ s$	$16,6\ min$	$3,4.10^{284}$ années

Pour éviter le choix arbitraire de l'opération élémentaire à dénombré, on poursuivra donc le calcul de la complexité en donnant le nombre de boucles (les plus intérieurs) dans l'algorithme, et on donnera ensuite la complexité en grand O de cette valeur.

Exemples 5 : Déterminer la complexité de la fonction suivante :

```
def fonction_3 ( liste_1 , liste_2 ) :
  compteur = 0
  for i in liste_1 :
    for j in liste_2 :
      if i==j :
        compteur = compteur +1
  return compteur
```

5 Tris

5.1 Tri par sélection

Le tri par sélection (ou tri par extraction) est un des algorithmes de tri les plus triviaux. Il consiste en la recherche soit du plus grand élément (ou le plus petit) que l'on va replacer à sa position finale c'est-à-dire en dernière position (ou en première), puis on recherche le second

plus grand élément (ou le second plus petit) que l'on va replacer également à sa position finale c'est-à-dire en avant-dernière position (ou en seconde), etc., jusqu'à ce que le tableau, ou la liste, soit entièrement trié.

En pseudo-code, l'algorithme s'écrit ainsi, pour un tableau `t` de taille n en argument :

```
fonction tri_selection(t)
   n = taille (t)
   pour i de 0 a n - 2
          indice_mini = i
          pour j de i + 1 a n - 1
              si t[j] < t[indice_mini], alors indice_mini = j
          fin pour
          echanger t[i] et t[indice_mini]
    fin pour
  renvoyer t
```

La fonction `echange` permet de changer de place les éléments du tableau.

On obtient le tableau suivant dans l'exécution de la fonction avec le tableau `t` $= [6, 3, 1, 7, 2]$, avec donc $n = 5$:

Étapes	`i`	`j`	`mini` avant	t[j]<t[mini]	`mini` après	tableau `t`
1	0	1	6	Vrai	3	
2	0	2	3	Vrai	1	
3	0	3	1	Faux	1	
4	0	4	1	Faux	1	
						[1, 3, 6, 7, 2]
5	1	2	3	Faux	3	
6	1	3	3	Faux	3	
7	1	4	3	Vrai	2	
						[1, 2, 6, 7, 3]
8	2	3	6	Faux	6	
9	2	4	6	Vrai	3	
						[1, 2, 3, 7, 6]
10	3	4	7	Vrai	6	
						[1, 2, 3, 6, 7]

Exercice 2 : Déterminer, en fonction de la taille de la liste, le nombre de boucle de la fonction.

Propriété 1 :

Le tri par sélection a une complexité quadratique (en $O(n^2)$) et ceci quelque soit la liste en argument.

5.2 Tri par insertion

Le tri par insertion est un autre algorithme que l'on peut qualifier de naïf. Cet algorithme consiste à piocher une à une les valeurs du tableau et à les insérer, au bon endroit, dans le tableau trié constitué des valeurs précédemment piochées et triées. Les valeurs sont piochées dans l'ordre où elles apparaissent dans le tableau.
Soit p l'indice de la valeur piochée, les $(p-1)$ premières valeurs du tableau constituent le tableau trié dans lequel va être inséré la p-ième valeur.

En pseudo-code, l'algorithme s'écrit ainsi, pour un tableau `t` de taille n en argument :

```
pour i de 1 a n - 1
      temp = T[i]
      j = i
        tant que j > 0 et T[j - 1] > temp
               T[j] = T[j - 1]
               j = j - 1
        fin tantque
        T[j] = temp
fin pour
```

On obtient le tableau suivant dans l'exécution de la fonction avec le tableau `t` $= [8, 5, 3, 2, 1]$, avec donc $n = 5$:

Étapes	i	j	temp	tableau t
1	1	1	5	[8, 8, 3, 2, 1]
				[5, 8, 3, 2, 1]
2	2	2	3	[5, 8, 8, 2, 1]
3	2	1	3	[5, 5, 8, 2, 1]
				[3, 5, 8, 2, 1]
4	3	3	2	[3, 5, 8, 8, 1]
5	3	2	2	[3, 5, 5, 8, 1]
6	3	1	2	[3, 3, 5, 8, 1]
				[2, 3, 5, 8, 1]
7	4	4	1	[2, 3, 5, 8, 8]
8	4	3	1	[2, 3, 5, 5, 8]
9	4	2	1	[2, 3, 3, 5, 8]
10	4	1	1	[2, 2, 3, 5, 8]
				[1, 2, 3, 5, 8]

Dans le cas d'un tableau trié, par exemple pour le tableau $\texttt{t} = [\texttt{5}, \texttt{8}, \texttt{10}, \texttt{15}, \texttt{16}]$

Étapes	i	j	temp	tableau t
1	1	1	5	[5, 8, 10, 15, 16]
2	2	2	8	[5, 8, 10, 15, 16]
3	3	3	15	[5, 8, 10, 15, 16]
4	4	4	16	[5, 8, 10, 15, 16]

Dans ce cas là, la boucle while n'est jamais utilisée...

Propriété 2 :

Le tri par insertion a une complexité quadratique (en $O(n^2)$) dans le pire des cas, et linéaire dans le meilleur des cas.