IF111 - Algorithmes et structures de données
EI1 - Complexité et notation asymptotique

Rohan Fossé

rfosse@labri.fr

Dans les exercices, sauf mention contraire, on réalisera toutes les analyses dans le pire cas et on donnera toujours les complexités demandées sous la forme d'un équivalent simple en $\mathcal{O}$.

1 Complexité des algorithmes

Exercice 1.1

Pour chacune des fonctions suivantes, déterminer la complexité asymptotique dans la notation $\mathcal{O}$.

1. $T_1(n) = 6n^3 + 10n^2 + 5n + 2$
2. $T_2(n) = 3log_2 n + 4$
3. $T_3(n) = 2^n + 6n^2 + 7n$
4. $T_4(n) = 7k + 2$
5. $T_5(n) = 4log_2 n + n$
6. $T_6(n) = 2log_{10} k + kn^2$

Exercice 1.2

Considérons les deux algorithmes A_1 et A_2 avec leurs temps d'exécution $T_1(n) = 9n^2$ et $T_2(n) = 100n + 96$ respectifs.

1. Déterminer la complexité asymptotique des deux algorithmes. Quel algorithme a la meilleure complexité asymptotique ?
2. Calculer les temps maximales d'exécution des deux algorithmes pour $n = 1$, $n = 3$, $n = 5$, $n = 10$, $n = 14$.
3. Quel est l'intervalle des valeurs de n sur lequel l'algorithme A_1 est plus efficace que l'algorithme A_2 ?

Exercice 1.3

Considérer les deux matrices carrées A et B de taille n :

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

l'addition de ces deux matrices donne la matrice C quadratique de taille n:

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix}$$

avec

$$c_{ij} = a_{ij} + b_{ij} \forall i, \forall j$$

1. Écrire un algorithme qui effectue l'addition des deux matrices A et B et stocke les résultats dans C;
2. Déterminer la complexité $\mathcal{O}$ de l'algorithme pour des matrices de taille n.

2 Programmes itératifs

Exercice 2.1

Donner la complexité en temps du programme suivant en fonction du paramètre n :

```
i ← 0
j ← 0
while i < n do
    if i % 2 == 0 then
        j ← j + 1
    else
        j ← j/2
    i ← i + 1
```

Exercice 2.2

On rappelle que si la variable *tab* contient un tableau, alors *tab.length* donne la longueur du tableau et $tab[i]$ permet d'accéder à sa case numéro i (la numérotation commence à 0). Donner la complexité en temps du programme suivant en fonction de la longueur du paramètre *tab* :

```
x ← 0
i ← 1
j ← 0
for i < tab.length − 1 do
    for j < 3 do
        x ← x + tab[i − 1 + j] ∗ (j + 1)
```

Exercice 2.3

Soit le programme suivant, dépendant des tableaux *tab* et *filter* :

```
x ← 0
i ← 1
j ← 0
for i < tab.length − filter.length do
    for j < filter.length do
        x ← x + tab[i + j] ∗ filter[j]
```

Calculer la complexité en temps du programme en fonction des longueurs des deux tableaux.

Exercice 2.4

On cherche à déterminer le minimum et le maximum d'un tableau. On propose d'abord la solution suivante :

```
function Solution_1(tab)
    min ← tab[0]
    max ← tab[0]
    i ← 0
    for i < tab.length do
        if tab[i] < min then
            min ← tab[i]
        else
            if tab[i] > max then
                max ← tab[i]
```

1. Calculer le nombre exact de comparaisons effectuées dans le pire cas en fonction de la longueur du tableau. On ne comptera que les comparaisons entre les éléments du tableau et le contenu des variables *min* et *max*

2. Pour toute longueur n, donner un exemple de tableau de longueur n correspondant au cas le pire.

On propose la variante suivante :

```
function Solution_2(tab)
    if tab.length % 2 == 1 then
        min ← tab[0]
        max ← tab[0]
        start ← 1
    else
        if tab[0] < tab[1] then
            min ← tab[0]
            max ← tab[1]
        else
            min ← tab[1]
            max ← tab[0]
        start ← 2
    i ← start
    for i < tab.length do
        if tab[i] < tab[i + 1] then
            if tab[i] < min then
                min ← tab[i]
            if tab[i + 1] > max then
                max ← tab[i + 1]
        else
            if tab[i + 1] < min then
                min ← tab[i + 1]
            if tab[i] > max then
                max ← tab[i]
        i ← i + 2
```

Calculer le nombre exact de comparaisons effectuées en fonction de la longueur du tableau. On ne comptera que les comparaisons entre les éléments du tableau entre eux et avec le contenu des variables *min* et *max*.

3 Appels de sous programmes

Exercice 3.1

On suppose que l'exécution de la fonction $f(n)$ prend un temps en $\mathcal{O}(n)$. En déduire la complexité du programme suivant en fonction de n :

```
i ← 0
for i < n do
    f(n - i)
```

Exercice 3.2

On suppose que l'exécution de la fonction $f(n)$ prend un temps en $\mathcal{O}(n)$. En déduire la complexité du programme suivant en fonction de n :

```
i ← 1
for i < n do
    f(i)
    i ← i * 2
```

Exercice 3.3

On considère la fonction suivante :

```
function pow(x, n)
    if n== 0 then return 1
    y ← x
    i ← 2
    for i < n do
        y ← y * x
    return y
```

1. Calculer le nombre exact d'opérations réalisées par un appel à $pow(x, n)$ en fonction de n, un entier positif ou nul.

   On utilise la fonction pow dans la fonction $variation_1$ suivante. Soit tab un tableau quelconque.

   ```
   function variation_1(x, tab)
       val ← 0
       i ← 0
       for i < tab.length do
           val ← val + tab[i] * pow(x, i)
   ```

2. Que fais la fonction ? Déduire de l'analyse précédente le nombre exact d'opérations réalisées par le programme en fonction de la longueur du tableau *tab*.

   On propose la variation suivante :

```
function variation_2(x, tab)
    val ← 0
    i ← 0
    y ← 1
    for i < tab.length do
        val ← val + tab[i] * y
        y ← y * x
```

3. Calculer le nombre exact d'opérations réalisées par le programme en fonction de la longueur du tableau *tab*.

   On propose enfin la fonction suivante :

```
function variation_3(x, tab)
    val ← 0
    i ← tab.length − 1
    for i >= 0 do
        val ← val + tab[i] * val
        i ← i − 1
```

4. Calculer le nombre exact d'opérations réalisées par le programme en fonction de la longueur du tableau *tab*.

Exercice 3.4

Calculer le nombre d'opérations et la complexité asymptotique de la fonction *mystère* suivante:

```
function mystere(n)
    m ← 0
    i ← 0
    j ← 0
    for i < n do
        for j < i do
            m ← m + i + j
```

4 Programmes récursifs

Exercice 4.1

On suppose qu'on dispose d'un tableau trié *tab* dans lequel on cherche la position d'une valeur x (avec la convention que la position est -1 si x n'apparaît pas dans *tab*). On propose la fonction de recherche suivante dans laquelle l'opération $tab[a : b]$ construit un tableau constitué des cases de *tab* d'indices compris au sens large entre a et b :

```
function search(x, tab)
    if tab.length == 0 then
        return -1
    if tab.length == 1 then
        if tab[0] == x then
            return 0
        else
            return -1
    else
        pos ← tab.length/2
        if tab[pos] == x then
            return pos
        if tab[pos] < x then
            subpos ← search(x, tab[(pos + 1) : (tab.length − 1)])
            if subpros >= 0 then
                return subpos + pos + 1
            else
                return -1
        else
            subpos ← search(x, tab[0 : (pos − 1)])
            if subpos >= 0 then
                return subpos
            else
                return -1
```

1. En supposant que l'opération $tab[a : b]$ peut être réalisée en temps constant, calculer la complexité en temps dans le cas le pire de ce programme en fonction de la longueur de *tab*.
2. En supposant que l'opération $tab[a : b]$ peut être réalisée en temps $\mathcal{O}(b - a + 1)$, calculer la complexité en temps dans le cas le pire de ce programme en fonction de la longueur de *tab*.

Exercice 4.2

On considère l'algorithme d'exponentiation rapide classique implémenté par la fonction suivante

```
function fastpow(x, n)
    if n == 0 then
        return 1
    if n == 1 then
        return x
    if n % 2 == 0 then
        returnfastpow(x ∗ x, n/2)
    else
        returnx ∗ fastpow(x ∗ x, n/2)
```

A l'aide du théorème maître, calculer la complexité en temps de cette fonction.