Analyse de complexité

EXERCICE 1 – ÉVALUONS LA COMPLEXITÉ DE QUELQUES ALGORITHMES

Déterminer la complexité des algorithmes suivants :

```
tab = tableau de taille n
tab[0] = 1
tab[1] = 1
tab[2] = 1
Pour i allant de 3 à n−1
Faire   somme = 0
        Pour j allant de i−3 à i−1
        Faire somme = somme + tab[j]
        tab[i] = somme
```

Correction. La boucle dans la première boucle fait seulement 3 passages et elle n'effectue que des additions. Donc sa complexité est en $O(1)$. La première boucle qui imbrique l'autre fait $n-3$ passages.
La complexité est donc $O((n-3)\times 1) = O(n)$.

```
cpt = 0
Pour i allant de 1 à n
Faire   Pour j allant de 1 à n*n*n*n
        Faire   Pour k allant de 1 à racine_carree(n)
                Faire cpt = (cpt + 1) modulo 100
```

Correction. Trois boucles imbriquées : ce qui est à l'intérieur des 3 boucles est en $O(1)$, donc on multiple le nombre de passages de chacune d'entre elles (voir cours).
La complexité est donc $O(n \times n^4 \times \sqrt{n}) = O(n^5\sqrt{n})$.

```
Pour i allant de 1 à n
Faire   cpt = 0
        k = i
        Tant que k > 0
        Faire   cpt = cpt + (k modulo 2)
                k = k // 2
        Afficher "Nombre :", i, "Somme des bits :",cpt
```

Correction. Le nombre de passage de la boucle "tant que" est le nombre de bits de i. Vu que $i < n$, le nombre de bits dans i est $O(\log(n))$.
Donc la complexité de l'algorithme est $n \times O(\log(n)) = O(n\log(n))$.

```
tab = tableau de taille n
tab[0] = 1
tab[1] = 1
tab[2] = 1
Pour i allant de 3 à n−1
Faire   somme = 0
        Pour j allant de 0 à i−1
        Faire somme = somme + tab[j]
```

```
        tab[i] = somme
```

Correction. Cette fois-ci, la boucle de fin fait i passages, où i est le compteur de la première boucle.
Donc l'instruction `somme = somme + tab[j]` est répétée 3 fois pour $i = 3$, 4 fois pour $i = 4$, ainsi de suite... jusqu'à $n-1$ fois pour $i = n-1$. En tout cela fait donc

$$3 + 4 + \cdots + n - 1$$

fois, soit $1 + 2 + 3 + 4 + \cdots + n - 1 - 3 = n(n-1)/2 - 3$ fois.
L'addition se faisant en temps $O(1)$, la complexité du programme est $O(n^2)$. (Le reste est négligeable.)

```
Fonction Affichage_ZigZag(tab : tableau)
        Affiche_Bout(tab,VRAI)

Fonction Affiche_Bout(tab : tableau, on_affiche_minimum : booléen)
        Si longueur(tab) > 0
        Alors   Si (on_affiche_minimum)
                Alors m = minimum de tab
                Sinon m = maximum de tab
                Afficher m
                Supprimer m de tab
                Affiche_Bout(tab,non(on_affiche_minimum))
```

Correction. Si on omet l'appel récursif, la complexité de la fonction est en $O(n)$ (recherche d'un minimum ou maximum ; supprimer un élément d'un tableau), où n est la taille du tableau.
Donc si on appelle $C(n)$ la complexité de l'algorithme pour un tableau de taille n, on a la récurrence :

$$C(n) = C(n-1) + O(n)$$

On a donc

$$C(n) = O(n) + O(n-1) + \cdots + O(1) = O(n + (n-1) + \cdots + 1) = O\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

```
Fonction Puissance2modulo100(n : entier positif)
        Si n == 0
        Alors Renvoyer 1
        Sinon Renvoyer (Puissance2modulo100(n-1)+Puissance2modulo100(n-1)) modulo 100
```

Correction. Si on omet les appels récursifs, la complexité de la fonction est en $O(1)$ (addition de petits nombres, modulos).
Donc si on appelle $C(n)$ la complexité de l'algorithme pour un entier de taille n, on a la récurrence :

$$C(n) = C(n-1) + C(n-1) = 2C(n-1)$$

On a donc

$$C(n) = 2C(n-1) = 4C(n-2) = \cdots = 2^k C(n-k) = \cdots = O(2^n).$$

EXERCICE 2 – DES FONCTIONS MYSTÈRE CLASSIQUES !

Q1. Qu'effectue la fonction "mystère" suivante ?

```
Fonction mystere ( x : flottant , n : entier )
        res = 1
        Pour i allant de 1 à n
        Faire   res = res * x
        Renvoyer res
```

Correction. Il s'agit de la fonction *puissance* : `mystere(x,n)` renvoie x^n.

Q2. Quelle est la complexité (en temps) de cette fonction `mystere` ?

Correction. Dans la fonction mystère, on retrouve une boucle qui fait $n-1$ passages. Si on estime que le coût pour une multiplication entre deux flottants est constant (ce qui n'est pas déconnant), alors la complexité de la fonction est $n \times O(1) = O(n)$: elle est linéaire.

Q3. Quelle est sa complexité en espace ?

Correction. On alloue seulement 4 variables (les deux paramètres, `res` et `x`) donc la complexité en $O(1)$.

Considérons maintenant la fonction suivante :

```
Fonction mystere2 ( x : flottant , n : entier )
        b = tableau de 0 et de 1 codant n en binaire
        res = x
        Pour i allant de 1 à (taille de b − 1)
        Faire   Si b[i] == 0
                Alors res = res * res
                Sinon res = res * res * x
        Renvoyer res
```

Pour la première ligne, on supposera que `b[0]` est le bit de poids fort (donc toujours égal à 1), et `b[taille de b - 1]` le bit de poids faible. Par exemple, si `n` vaut `18`, alors `b` sera le tableau `[1,0,0,1,0]`.

Q4. Calculez `mystere2(2.0,11).`

Correction. Ca vaut 2048.

Q5. Quelle est la complexité en temps de `mystere2` ?

Correction. Cette fois la boucle principale a autant d'itérations que la taille de `b`. La complexité de l'algorithmique est ainsi proportionnelle au nombre de bits de n. Or le nombre de bits dans un entier `n` n'est d'autre que le logarithmique en base de 2 de n (pris en l'entier supérieur). Donc la complexité de `mystere2` est en $O(\log n)$.

Q6. Quelle est la complexité en espace de `mystere2` ?

Correction. On a alloué un tableau d'une taille égale au nombre de bits de n. La complexité en espace est donc aussi en $O(\log n)$.

Q7. Prouvez que `mystere2` calcule la même chose que `mystere`. Indication : Il faut prouver par récurrence sur i qu'au bout de la i-ième itération, `res` vaut x^m où m est le nombre obtenu en convertissant en binaire le sous-tableau `b[0..i]`.

Correction. Cet algorithme s'appelle l'algorithme d'exponentiation rapide (très utilisé).

Prouvons donc la propriété de l'énoncé par récurrence sur i.

Cas de base. Quand i vaut 0, on n'est pas encore rentré dans la boucle principale. La variable `res` est initialisée à x. De plus, m vaut 1 car le tableau `b` commence forcément par 1 et `b[0..0]` n'est que le tableay `[1]`. On a bien $res = x = x^m$.

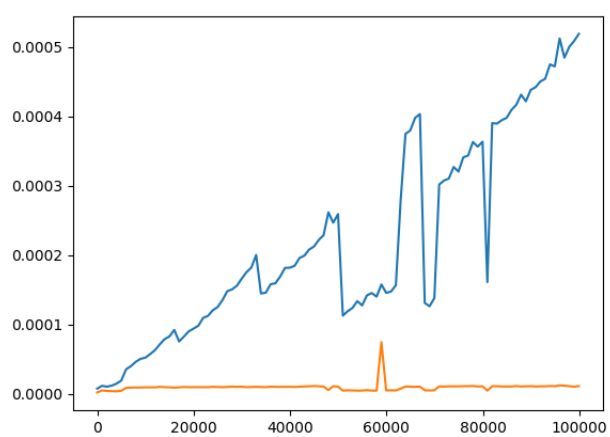
Hérédité. Par récurrence, on suppose qu'après la $i-1$-ième itération, donc juste avant la i-ième itération, `res` vaut $x^{\tilde{m}}$ où $\tilde{m}$ est le nombre binaire $b[0], \dots, b[i-1]$. Remarquons que si m est égal à $b[0], \dots, b[i]$, alors m est égal $2\tilde{m}$ si `b[i]` vaut 0, $2\tilde{m}+1$ sinon.
Or `res` vaut après la i-ième itération soit $x^{2\tilde{m}}$ si `b[i]` vaut 0, soit $x^{2\tilde{m}+1}$ si `b[i]` vaut 1. Dans les deux cas, cela vaut x^m, ce qui achève la récurrence.

Quand la boucle s'arrête, la fonction `mystere2` renvoie `res` qui contient $x^{b \text{ converti en binaire}}$, soit x^n.

J'ai codé `mystere2` en python :

```python
def mystere2(x,n):
    b = list(map(int,bin(n)[2:]))
    res = x
    for i in range(1,len(b)):
        if b[i] == 0:
            res = res * res
        else:
            res = res * res * x
    return res
```

et ai testé d'une part le temps d'exécution de `mystere2(2,n)`, et d'autre part le temps d'exécution de `mystere2(2.0,n)` en fonction de n. Voici les deux courbes que j'ai obtenues :



Q8. Pouvez-vous deviner à quelle courbe correspond le temps d'exécution de `mystere2(2,n)`, et celui de `mystere2(2.0,n)` ? A votre avis, pourquoi une telle différence entre les deux courbes ?

Correction. La courbe en bleu est celle de `mystere2(2,n)`. En effet, en python, les entiers ont une précision illimitée, contrairement aux flottants. Ainsi au bout de la i-ième itération, `res` sera codé sur m bits. Le coût d'une multiplication ne peut plus être considérée comme constante (elle est au moins linéaire en m) ! (La complexité est donc au moins $\theta(n \log n)$, peut-être plus !)
Quant aux flottants, ils sont codés sur un nombre limités de bits. Donc on procède à un arrondi à chaque étape, ce qui fait que le temps d'une multiplication reste constante, même quand `res` est très grand.

EXERCICE 3 – RECHERCHE D'UN MOTIF DANS UN TEXTE (COMPLEXITÉ À PLUSIEURS VARIABLES ?)

Q1. Écrire (de manière naïve !) une fonction qui prend en paramètre deux chaînes de caractère `texte` et `motif` et qui renvoie `VRAI` si `motif` est bien une sous-chaîne de `texte` et `FAUX` dans le cas contraire.

Correction.

```
Fonction recherche ( texte : chaîne de caractères, motif : chaîne de caractère )
        n = longueur(texte)
        m = longueur(motif)
        Pour i allant de 0 jusqu'à n − m                // Attention aux indices
        Faire   j = 0

                Tant que (j < m) et (texte[i+j] == motif[j])
                Faire   j = j + 1

                Si (j == m)
                Alors   Renvoyer VRAI

        Renvoyer FAUX
```

La boucle "tant que" peut être remplacée par une seconde boucle "pour" et un "break".

Q2. Exprimez la complexité en temps de votre fonction en fonction de la longueur de `texte`, notée `n`, et la longueur de `motif`, notée `m`.
Attention ! La complexité finale doit bien dépendre de m.

Correction. La boucle "Tant que" fait au plus m itérations, et la complexité à l'intérieur de cet boucle est en $O(1)$. Donc la complexité de ce qu'il y a à l'intérieur de la boucle "pour" est en $O(m)$. Cette boucle "pour" faisant au pire $n - m$ passages, la complexité de la fonction est en $O(m(n - m))$.
La complexité $O(n^2)$ est juste dans l'absolu, mais trop grossière, comme la question suivante le montre.

Q3. Quelle est la complexité asymptotique en temps d'un appel de votre fonction lorsque :

1. le motif fait 10 lettres ? ($m = 10$)
2. le motif fait 3 lettres de moins que le texte ? ($m = n - 3$)
3. le motif a à peu près moitié moins de lettres que le texte ? ($m = n/2$)

Correction.

1. La complexité est en $O(m(n-m)) = O(10(n-10)) = O(10n) = O(n)$, donc linéaire.
2. La complexité est en $O(m(n-m)) = O((n-3)\times 3) = O(3n) = O(n)$, donc linéaire.
3. la complexité est en $O(m(n-m)) = O((n/2)^2) = O(n^2/4) = O(n)$, donc quadratique.

EXERCICE 4 – DEUXIÈME MINIMUM

Toutes les fonctions suivantes sont censées renvoyer le deuxième plus petit élément d'un tableau (si le plus petit élément du tableau apparaît plus qu'une fois, alors il renvoie ce plus petit élément).

```
Fonction deuxieme_min_1 ( tab : tableau d'entiers )
        m = minimum(tab)
        Supprimer m de tab
        m = minimum(tab)
        Renvoyer m
```

```
Fonction deuxieme_min_2 ( tab : tableau d'entiers )
        Trier tab
        Renvoyer ?????
```

```
Fonction deuxieme_min_3 ( tab : tableau d'entiers )
        n = longueur(tab)
        indice_minimum = 0
        Pour i allant de 1 jusqu'à n-1
        Faire    Si (tab[i] < tab[indice_minimum])
                 Alors    indice_minimum = i

        Si indice_minimum == 0
        Alors    indice_deuxieme_minimum = 0
        Sinon    indice_deuxieme_minimum = 1

        Pour j allant de 1 jusqu'à n-1
        Faire    Si (tab[j] < tab[indice_deuxieme_minimum]) et ?????
                 Alors    indice_deuxieme_minimum = j

        Renvoyer tab[indice_deuxieme_minimum]
```

Q1. Remplacez les ????? de sorte que les fonctions renvoient bien le deuxième plus élément.

Correction. Deuxième fonction → "`tab[1]`"
Troisième fonction → "`j` $\neq$ `indice_minimum`"

Q2. Quelle est la complexité asymptotique en temps de ses fonctions ?

Correction. **Première fonction :** Chacune des lignes prend un temps linéaire à être calculée. Donc le résultat est aussi linéaire. La complexité est en $O(n)$, où n est la longueur du tableau.
Deuxième fonction : L'instruction la plus coûteuse est évidemment celle du tri. Un tri prenant un temps $O(n \log n)$ à être exécuté, il s'agit aussi de la complexité de la fonction (n = taille du tableau).
Troisième fonction : Le corps de chacune des boucles (très similaires) s'exécute en temps $O(1)$ (lire une case d'un tableau, comparaison, affectation de variable). Or vu qu'on effectue $n-2$ passages dans chacune des boucles ; le temps d'exécution de l'une ou l'autre se fait en temps $O(n)$. Vu que ces deux boucles se suivent et ne sont pas imbriquées, on a une complexité linéaire en la taille du tableau : $O(n)$.

Q3. Qu'auriez-vous écrit comme fonction ?

Correction. Normalement le plus efficace est quelque chose qui s'approcherait d'un tri insertion où on ne garderait que les deux premiers éléments :

```
Fonction deuxieme_min(tab : tableau d'entiers)
        minimum = + infini
        deuxieme_minimum = + infini
        Pour i allant de O jusqu'à longueur(tab)
        Faire    x = tab[i]
                 Si x < deuxieme_minimum
                 Alors   Si x < minimum
                         Alors deuxieme_minimum = minimum
                               minimum = x
                         Sinon deuxieme_minimum = x
        Renvoyer deuxieme_minimum
```

Ici on ne fait qu'une seule passe.
Ensuite la deuxième fonction où on trie peut être également utilisée malgré une complexité en $O(n\log(n))$. La différence de temps d'exécution est normalement négligeable.