TD n°08 : La complexité temporelle

2 heures

Rédigé par Pascal Delahaye

12 décembre 2015

1 Exemple 1 : Fibonacci

La suite de fibonnacci est définie par : (u_n) telle que $\begin{cases} u_0 = 0 \\ u_1 = 1 \end{cases}$ et pour tout $n \in \mathbb{N}^*$: $u_{n+2} = u_{n+1} + u_n$.

Pour calculer le terme u_n, nous disposons des 2 algorithmes suivants :

```
def u1(n) :                          |          def u2(n) :
    S1, S2 = 0, 1                    |              if n == 0 : return 0
    for i in range(2,n+1):           |              if n == 1 : return 1
        G = S2                       |              else :   return u2(n-1) + u2(n-2)
        S2 = S1 + S2                 |
        S1 = G                       |          # Ceci est un algorithme récursif #
    return S2                        |
```

Nous allons évaluer le temps de calcul de chacun des algorithmes de calcul précédents de façon expérimentale et théorique.

Etude expérimentale :

En Python, il est possible de déterminer le temps de calcul d'un algorithme grâce aux instructions suivantes :

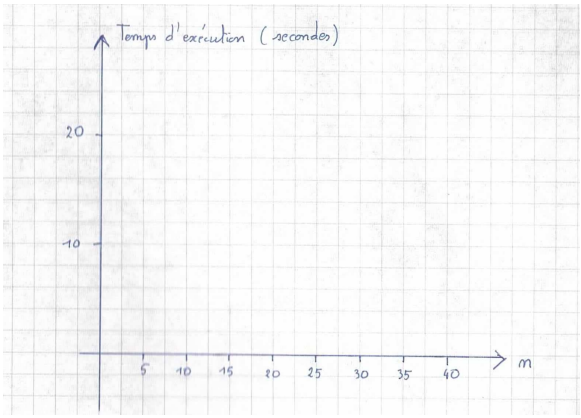
```
from time import clock
t1 = clock()
... instructions du programme dont on veut évaluer la durée ...
t2 = clock()
print(t2-t1)
```

Concevoir un programme permettant de représenter le temps d'exécution de chacune des deux fonctions suivantes en fonction de l'indice n du terme de la suite à calculer. On choisira $n \in [\![0, 30]\!]$ par pas de 2.

Pour cela, on utilisera les fonctions suivantes :

- La fonction `plot()` de la bibliothèque `matplotlib.pylab` en imposant une couleur à chaque courbe
- La fonction `arange()` de la bibliothèque `matplotlib.pylab`

- La fonction `clock()` de la bibliothèque `time`



Courbes de complexité des deux algorithmes

1. Quel est le programme le plus rapide ?
2. Quelle conjecture peut-on faire sur la nature de la complexité de chacun des deux algorithmes ?
3. Eliminer la courbe de complexité de u_2 afin d'observer plus précisément la courbe de complexité de u_1.
   Vous prendrez alors $n \in [\![0, 1000]\!]$ *avec un pas de* 1.
   Commentez les résultats observés.

Etude théorique : Donner sous la forme d'un "grand O" la complexité des deux algorithmes précédents.

On considèrera :

1. La taille de la donnée : n
2. L'opération significative : l'addition

Cette étude théorique vient-elle confirmer les conjectures précédentes ?

2 Exemple 2 : Polynômes

Soit $P \in \mathbb{R}[X]$ et $\alpha \in \mathbb{R}$. Les deux algorithmes suivants proposent deux méthodes de calcul de $P(\alpha)$.
Nous représenterons un polynôme P par la liste de ses coefficients :

$$P = a_0 + a_1 X + \cdots + a_n X^n \quad \text{est alors repésenté en machine par :} \quad p = [a_0, \ldots, a_n]$$

Calcul naïf

La première idée pour le calcul de $P(\alpha)$ est tout simplement de calculer $P(\alpha) = \sum_{k=0}^{n} a_k \alpha^k$ à l'aide d'une boucle FOR.

L'algorithme de Horner

Cet autre algorithme propose de calculer $P(\alpha)$ en procédant de la façon suivante :

Si $\alpha \in \mathbb{K}$, l'idée de l'algorithme d'Hörner est de déterminer $P(\alpha)$ en effectuant les calculs suivants :

1. a_n
2. $a_n \times \alpha + a_{n-1}$
3. $(a_n \times \alpha + a_{n-1}) \times \alpha + a_{n-2}$
4. ...etc ...

Après la ième itération on obtient l'expression :

$$a_n \alpha^i + a_{n-1} \alpha^{i-1} + \cdots + a_{n-i}$$

En poursuivant ainsi jusqu'à a_0, on obtient la valeur de $P(\alpha)$.

L'algorithme se présente alors de la façon suivante :

```
Arguments : p      : liste des coefficients du polynôme
            alpha  : flottant

Variables locales : valeur : flottant
                    n      : entier

Initialisation :
 * n      <- longueur de la liste p
 * valeur <- p[n]

Boucle :  Pour i allant de 1 à n-1 faire valeur <- valeur * alpha + p[n - i]

Fin : retourner valeur
```

Travail

Etude expérimentale :

1. Construire une procédure CALC d'arguments P et α et calculant $P(\alpha)$ selon la méthode naïve.
2. Construire une procédure HORNER d'arguments P et α et calculant $P(\alpha)$ avec l'algorithme de Horner.
3. Grâce à la fonction `clock()` de Python, évaluer sur un exemple le temps de calcul de $P(\alpha)$ par chacune des procédures précédentes. Vous prendrez un polynôme de degré très important et par exemple $\alpha = 1000$!.
4. Nous souhaitons représenter sur un même graphe, l'évolution des temps de traitement expérimentaux des deux algorithmes précédents en fonction du degré n de la fonction polynômiale. Le polynôme utilisé pour ces expérimentations sera `range(1,n)` où n pourra varier de 1 à 30.
   (a) Programmer la fonction factorielle ou importez-la depuis la bibliothèque *sympy*.
   (b) Concevoir un programme dont le but est de représenter sur un même graphe les courbes permettant de visualiser le temps d'évolution de chacun des deux algorithmes en fonction de n.

Etude théorique :

Expliquez les deux courbes observées expérimentalement en évaluant la complexité des deux algorithmes.