TD : La complexité temporelle

2 heures

Rédigé par Pascal Delahaye

24 novembre 2017

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):           |              elif n == 1 : return 1
        S1, S2 = S2, S1 + S2         |              else :       : return u2(n-1) + u2(n-2)
    return S2                        |          # Ceci est un algorithme récursif #
```

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

1) Etude expérimentale

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

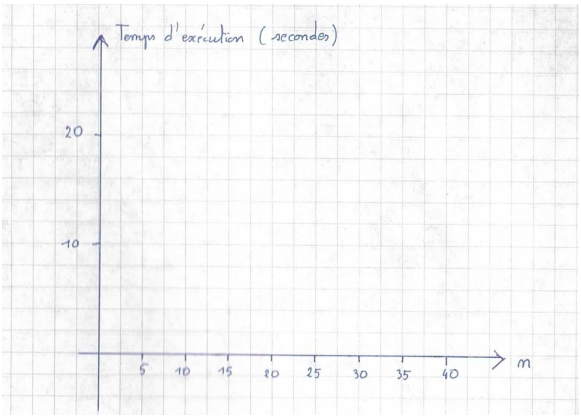
```
from time import clock
t1 = clock()
... instructions du programme dont on veut évaluer la durée ...
t2 = clock()
print(t2-t1)
```

1. Concevoir deux fonctions `tp1(n)` et `tp2(n)` donnant le temps d'exécution des instructions `u1(n)` et `u2(n)`.

2. En déduire deux fonctions `graphe1(N,p)` et `graphe2(N,p)` permettant de représenter le temps d'exécution de chacune des deux fonctions u_1 et u_2 pour n allant de 0 à N par pas de $p \in \mathbb{N}^*$.

   Lancer alors les instructions `graphe1(100,1)` et `graphe2(35,3)`.

3. Répondez alors aux questions suivantes :



Courbes de complexité des deux algorithmes

(a) Quel est le programme le plus rapide ?
(b) Quelle conjecture peut-on faire sur la nature de la complexité de chacun des deux algorithmes ?
(c) Comment interpréter :
   i. les valeurs qui semblent aberrantes ?
   ii. les différences obtenues lorsqu'on lance deux fois la même instruction ?
   iii. les discontinuités qui apparaissent éventuellement ?

2) 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 ?

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 :

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

1. Echauffement

Construire une fonction `somme(P,Q)` qui renvoie la somme de deux polynômes.

2. 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.

3. 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. $S = a_n$ (initialisation)
2. $S = S \times \alpha + a_{n-1} = a_n.\alpha + a_{n-1}$ (i=1)
3. $S = S \times \alpha + a_{n-1} = (a_n \times \alpha + a_{n-1}) \times \alpha + a_{n-2} = a_n.\alpha^2 + a_{n-1}.\alpha + a_{n-2}$ (i=2)
4. ...etc ...

Après la ième itération on obtient l'expression : $S = a_n\alpha^i + a_{n-1}\alpha^{i-1} + \cdots + a_{n-i}$

En poursuivant ainsi jusqu'à $i = n$, 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 : S : flottant stockant les valeurs successives du calcul
                    n      : entier représentant le degré du polynôme

Initialisation :
 * n <- degré du polynome p          ATTENTION au sens de n !!
 * S <- p[n]

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

Fin : retourner S
```

Travail

Etude expérimentale :

1. Construire une fonction `CALC(P,alpha)` calculant $P(\alpha)$ selon la méthode naïve.

2. Construire une fonction `HORNER(P,alpha)` calculant $P(\alpha)$ avec l'algorithme de Horner.

3. En utilisant la fonction `clock()` de Python, déterminer deux fonctions `tp1(P,alpha)` et `tp2(P,alpha)` qui renvoient le temps de calcul de $P(\alpha)$ par chacune des deux fonctions précédentes.

   Vous appliquerez ces deux fonctions en prenant pour arguments un polynôme de degré variable et $\alpha = 1000$! (oui, je sais : c'est très grand !)

4. Construire une procédure `graphe(N,alpha)` qui affiche le graphe donnant l'évolution des temps d'execution des deux algorithmes précédents en fonction du degré $n \in [\![1, N]\!]$ de la fonction polynômiale.

   Vous prendrez pour polynômes `range(1,n)` où n pourra varier de 1 à N=30 et de nouveau $\alpha = 1000$!. Commenter les résultats obtenus.

Etude théorique :

A l'aide des deux courbes observées expérimentalement, conjecturer la complexité des deux algorithmes.
Valider votre conjecture à l'aide d'une étude théorique de complexité.

Comparaison expérimentale de la complexité de différents algorithmes

-

Solutions

Python

```
########################################################################
# Comparaison de deux algorithmes de calcul de la suite de Fibonacci#
########################################################################

def u1(n):                          # Algorithme usuel (complexité linéaire)
    u0 = 0
    u1 = 1
    for k in range(2,n+1):
        u0, u1 = u1, u0 + u1
    return u1

def u2(n):                          # Algorithme récursif (complexité exponentielle)
    if n == 0   : return 0
    elif n == 1 : return 1
    else : return u2(n-1)+u2(n-2)

def tp1(n):
    from time import clock
    t1 = clock()
    A = u1(n)
    t2 = clock()
    return t2 - t1

def tp2(n):
    from time import clock
    t1 = clock()
    A = u2(n)
    t2 = clock()
    return t2 - t1

def G1(N,p):                                    # Complexite de l'algorithme usuel
    from matplotlib.pylab import plot, arange
    X = arange(1,N+1,p)
    Y = [tp1(x) for x in X]
    plot(X,Y)

def G2(N,p):                                    # Complexite de l'algorithme récursif
    from matplotlib.pylab import plot, arange
    X = arange(1,N+1,p)
    Y = [tp2(x) for x in X]
    plot(X,Y)
```

Python

```
#######################################################
# Comparaison de 2 algorithmes de calcul de P(alpha) #
#######################################################

from sympy import factorial

def alg1(P,alpha):              # algorithme naïf (complexité quadratique)
    n = len(P)
    S=0
    for k in range(0,n):
        S = S + P[k]*alpha**k
    return S

def alg2(P,alpha):              # algorithme de Horner (complexité linéaire)
    n = len(P)
    S = P[n-1]
    for k in range(1,n):
        S = S*alpha + P[n-1-k]
    return S

def tpalg1(n,alpha):            # durée d'execution de l'alg. naïf
    from time import clock
    P = range(2,n)
    t1 = clock()
    A = alg1(P,alpha)
    t2 = clock()
    return t2 - t1

def tpalg2(n,alpha):            # durée d'execution de l'alg. de Horner
    from time import clock
    P = range(2,n)
    t1 = clock()
    A = alg2(P,alpha)
    t2 = clock()
    return t2 - t1

def graphe(N,p,alpha):          # graphes de comparaison des deux complexité
    from matplotlib.pylab import plot, arange
    X = arange(3,N+1,p)
    Y1 = [tpalg1(x,alpha) for x in X]
    Y2 = [tpalg2(x,alpha) for x in X]
    plot(X,Y1)
    plot(X,Y2)
```