Devoir d'informatique 3

D.Malka – MPSI 2016-2017 – Lycée Saint-Exupéry

13.01.2017

Exercice 1 – Exponentiation naïve

On redonne l'algorithme d'exponentation naïve de k par un entier naturel n.

```
1 def expo(k,n):
2     p=1
3     c=n
4     while c>0:
5         p=p*k
6         c=c-1
7     return p
```

Figure 1 – Algorithme 1

1. Variant de boucle $u_i = n - c_i$. Voir cours.
2. Invariant de boucle $p_i = k^i$. Voir cours.
3. Unité de mesure : on suppose que les opérations arithmétiques de base (addition, multiplication, soustraction) sont réalisées en temps constant. Dans ces conditions la complexité de l'algorithme est $O(n)$.

Exercice 2 – Tranche de somme minimale d'un tableau

Soit T un tableau d'entiers relatifs. La tranche de somme minimale de T est le sous-tableau de T dont la somme des éléments est minimale. Par exemple, la tranche de somme minimale du tableau $[12, -4, -5, 2, -1]$ est $[-4, -5]$ (la somme vaut alors -9). Les tranches $[-5, 2, -1]$ ou encore $[12, -4]$ ne sont pas de somme minimale. Il arrive qu'il existe plusieurs tranches de somme minimale au sein d'un tableau.

1. La tranche de somme minimale du tableau $[-5, -2, 9, -4, -4, 3]$ est $[-4, -4]$ de somme -8.
2. On propose, figure 2, un premier algorithme naïf déterminant la tranche de somme minimale d'un tableau.

```
1  def tranche_min(T):
2      n=len(T)
3      g=0;d=0
4      s_min=T[0]
5      for i in range(n):
6          for j in range(i,n):
7              s=somme(T,i,j)
8              if s<s_min:
9                  s_min=s;g=i;d=j
10     return T[g:d+1],s_min
```

Figure 2 – Algorithme 2

2.1 Fonction `somme` : fig.3.
On suppose que l'addition est réalisée en temps constant. Le corps de boucle est exécuté $j - i + 1$ fois. Dans ces conditions, la complexité de la fonction est $O(j - i)$.

```python
def somme(T,i,j):
    """
    Calcule et renvoie la somme des elements du sous-tableau T[i:j+1]
    d'un tableau d'entiers ou de flottants
    T : liste d'entiers relatifs ou de flottants; type : list
    i,j : int, bornes du sous-tableau
    s : somme des elements du tableau T; type : int ou float
    """
    n=len(T)
    s=0
    for k in range(i,j+1):
        s=s+T[k]
    return s
```

FIGURE 3 – Algorithme 3

2.2 L'algorithme fig.2 évalue la somme de toutes les tranches possibles du tableau T et stocke dans les variables d et g à chaque itération les bornes de la tranche de somme minimale parmi celles explorées. A la sortie des deux boucles, la tranche $T[g, d+1]$ est la tranche de somme minimale du tableau.

2.3 L'instruction la plus coûteuse à chaque itération est l'appel à la fonction somme qui est $O(j-i)$. On peut donc estimer la fonction coût $C(n)$ de la façon suivante :

$$C(n) = \sum_{i=0}^{n}\sum_{j=i}^{n}(j-i)$$

$$C(n) = \sum_{i=0}^{n}\left(\frac{(n-i)(n-i+1)}{2} - i(n-i+1)\right)$$

$$C(n) = \sum_{i=0}^{n}\frac{n^2}{2} - \sum_{i=0}^{n}2in + \sum_{i=0}^{n}\frac{3}{2}i^2 + \sum_{i=0}^{n}\frac{n}{2} - \sum_{i=0}^{n}\frac{i}{2} - \sum_{i=0}^{n}1$$

— Les termes $\sum_{i=0}^{n}\frac{n^2}{2}$, $\sum_{i=0}^{n}2in$, $\sum_{i=0}^{n}\frac{3}{2}i^2$ sont de l'ordre de n^3.

— Les termes $\sum_{i=0}^{n}\frac{n}{2}$ et $\sum_{i=0}^{n}\frac{i}{2}$ sont de l'ordre de n^2.

— Le terme $\sum_{i=0}^{n}1 = n+1$ est de l'ordre de n.

Si on suppose raisonnablement que les termes $\sim n^3$ ne se compensent pas (le calcul le montrerait), on conclut que la complexité de l'algorithme est $O(n^3)$.

3. On propose, figure 4, un second algorithme.

3.1 Appliquons l'algorithme fig.4 au tableau $[-5,-2,6,-4,-4,3]$.

g	d	k	j	s	s_{min}	$T[g:d+1]$
0	0	0	0	0	-5	[-5]
0	0	0	1	-5	-5	[-5]
0	1	0	2	-7	-7	[-5, -2]
0	1	0	3	2	-7	[-5, -2]
0	1	3	4	-4	-7	[-5, -2]
3	4	3	5	-8	-8	[-4, -4]
3	4	3	6	-5	-8	[-4, -4]

3.2 Montrons que :

Invariant 1 : *Le sous tableau `T[k,j+1]` est la tranche finissant en `j` de somme minimale.*

Invariant 2 : *Le sous tableau `T[g,d+1]` est la tranche de somme minimale du sous tableau `T[0:j+1]`.*

sont des invariants de boucle.

— **Initialisation**. $k=0$, $j=0$, $g=0$, $d=0$.
  — *invariant 1.* $T[0:1]$ est la seule et donc la tranche de somme minimale terminant en j.
  — *invariant 2.* $T[0:1]$ est l'unique tranche du tableau $T[0:1]$ donc est sa tranche minimale.
  .

— **Hérédité**.

```
def tranche_min_2(T):
    n=len(T)
    g=0;d=0;
    s_min=T[0]
    k=0;j=0;
    s=0
    while j<n:
        if s+T[j]<T[j]:
            s=s+T[j]
        else:
            s=T[j]
            k=j
        if s<s_min:
            g=k
            d=j
            s_min=s
        j=j+1
    return T[g:d+1],s_min
```

FIGURE 4 – Algorithme 4

— *invariant 1.* $T[k : j+1]$ est la plus tranche finissant en j de somme minimale. Si on ajoute un élément, l'hypothèse de récurrence impose que $\forall p \neq k$, $somme(T[p : j+1]) < somme(T[k : j+1])$ ce qui implique que $\forall p \neq k$ $somme(T[p : j+1])+T[j+2] < somme(T[k : j+1])+T[j+2])$ soit $\forall p \neq k$ $somme(T[p : j+2]) < somme(T[k : j+2])$. Donc $T[k : j+2]$ est la plus tranche finissant en $j+1$ de somme minimale sauf si la $T[j+2 < somme(T[k : j+2])]$ ce qui est traité par la disjonction de cas de l'algorithme.

— *invariant 2.*A l'itération $j+1$, par hypothèse de récurrence les seuls nouvelles tranches de somme potentiellement plus petite que $T[g : d+1]$ sont $T[k : j+2]$ et $T[j+2]$. D'après l'algorithme, si $sommme(T[k : j+2]) > somme(T[g : d+1])$ et $T[j+2] > somme(T[g : d+1])$ alors $T[g : d+1]$ est la tranche de somme minimale du sous tableau $T[0 : j+2]$ sinon $T[g : d+1]$ devient la tranche de somme minimale parmi $T[k : j+2]$ et $T[j+2]$ et donc $T[g : d+1]$ est la tranche de somme minimale du sous tableau $T[0 : j+2]$.

— **Sortie de boucle**. $i = n$ et l'invariant de boucle de boucle est toujours valable : $T[g : d+1]$ est la tranche de somme minimale du sous tableau $T[0 : n]$ c'est-à-dire du tableau entier.

3.3 L'algorithme 3 est clairement plus efficace que l'algorithme 2. La durée d'exécution augmente linéairement avec la taille de l'entrée ce qui signe une complexité linéaire ($O(n)$). Cette complexité est conforme à l'analyse de l'algorithme qui montre que le tableau n'est parcouru qu'une seule fois et qu'à chaque itération les opérations sont réalisées en temps constant. La durée d'exécution de l'algorithme 2 croît environ d'un facteur $1000 = 10^3$ lorsque la taille de l'entrée est multipliée par 10 et environ d'un facteur $30 \approx 3^3$ lorsque la taille de l'entrée passe de 1000 à 3000. Ce comportement signe une complexité cubique conformément à l'analyse réalisée préalablement.

n	10	100	1000	3000
algo 2	$81\,\mu s$	$18\,ms$	$13\,s$	$384\,s$
algo 3	$6\,\mu s$	$30\,\mu s$	$250\,\mu s$	$846\,\mu s$

Exercice 3 – Suite ultimement périodique

On considère un système dynamique discret, c'est-à-dire une suite d'éléments d'un ensemble E définie par le relation $x_{n+1} = f(x_n)$ dont $x_0 \in E$ est le premier terme et $f : E \to E$. On dit qu'une telle suite d'élément est ultimement périodique de période p s'il existe un entier N tel que $\forall\, n \geq N$, $x_{n+p} = x_n$.

1. la suite est périodique de période p si il existe p tel que $\forall\, n$, $x_{n+p} = x_n$. Si on suppose que la suite est périodique, on peut naïvement calculer tous les termes de la suite jusqu'au terme x_p telle que $x_p = x_0$. La période vaut alors p.

```
def periode_suite(f,x0):
    xp=f(x0)
    p=1
    while xp!=x0:
```

```
        xp=f(xp)
        p=p+1
    return p
```

2. A présent, la suite est seulement ultimement périodique. L'algorithme de Floyd que nous étudions :
   — prend en entrée la fonction $f : E \to E$ et l'élément $x_0 \in E$ définissant une suite récurrente ultimement périodique ;
   — renvoie un entier t tel que $x_{2t} = x_t$.

```
def Floyd(f,x0):
    t=1
    x=f(x0);y=f(f(x0))
    while y!=x:
        x=f(x)
        y=f(f(y))
        t=t+1
    return t
```

2.1 A chaque itération, on applique deux fois f à y et une seule fois à x, au bout de $t-1$ itérations, x contient x_t et y contient $x(2t)$.

2.2 Si la suite est ultimement périodique alors il existe un entier N tel que $\forall n \geq N$, $x_{n+p} = x_n$. Dans la boucle t croît par pas de 1 donc t atteint N au bout de $t-1$ itération. L'écart d'indice entre x et y vaut $2t - t = t$ et croît donc par pas de 1. Après au moins $N+1$ itération, il existe une valeur de t_f de t tel que $2t_f = t_f + \alpha p$ avec $\alpha \in \mathbb{N}^*$ et p la période de la suite. Alors $x = x_{t_f}$ et $y = x_{2t_f} = x_{t_f+\alpha p} = x_{t_f} = x$. Donc l'algorithme termine.

2.3 La démonstration précédente montre alors que $t_f = \alpha p$. L'algorithme renvoie un multiple de la période de la suite si elle est périodique.

2.4 Fonction permettant de déterminer la période p de la suite. On supposera définie et déclarée la fonction `diviseur` qui renvoie sous forme de liste l'ensemble des diviseurs de l'entier naturel passé en argument.

```
def Floyd(f,x0):
    t=1
    x=f(x0);y=f(f(x0))
    while y!=x:
        x=f(x)
        y=f(f(y))
        t=t+1
    return t
```

2.5 L'algorithme de Floyd ne peut pas démontrer l'ultime périodicité d'une suite car on ne peut pas exclure qu'il existe un rang n_0 particulier d'une suite quelconque $(x_n)_{n\in\mathbb{N}}$ tel que $x_{n_0} = x_{2n_0}$. L'algorithme terminerai et renverrai $t = n_0$ alors que la suite n'est ultimement périodique.