Informatique tronc commun
Algorithmes usuels et correction des boucles

Corrigé

26 octobre 2007

Introduction

Que renvoie un appel de `g(a,b)` ?

La procédure `f` renvoie l'inverse multiplicatif de tout entier strictement positif. Donc `g` renvoie $\Sigma_{k=a}^{b} \frac{1}{k}$.

1 Un premier exemple

1. Lorsque l'on appelle `calcul(5)`, Maple affiche

```
> calcul (5);
calcul (5);
           "je vais calculer u", 5

           "je vais calculer u", 4

           "je vais calculer u", 3

           "je vais calculer u", 2

           "je vais calculer u", 1

           "je vais calculer u", 0

          "je viens de calculer u", 0

          "je viens de calculer u", 1

          "je viens de calculer u", 2

          "je viens de calculer u", 3

          "je viens de calculer u", 4

          "je viens de calculer u", 5

                      156
```

2.

3. À chaque appel récursif, la procédure fait une multiplication, et diminue son argument de 1. Elle s'arrête lorsque son argument vaut 0, et fait donc n multiplications.

2 Le calcul des puissances entières

2.1 Algorithme naïf itératif

a On a $u_n(x) = xu_{n-1}(x)$ et $u_0 = x^0 = 1$.

b Pour avoir H_0 vraie, il suffit d'initialiser `u` à 1. Si on suppose H_i, il suffit de mettre à jour la valeur de `u` à xu pour avoir H_{i+1} vraie. Pour calculer x^n, on effectue ceci pour $1 \leq i \leq (n-1)$

c
```
puissance := proc(x,n)
    local i,u;
    u := 0;
    # H0 est vraie
    for i from 1 to (n-1) do
        #si Hi vraie
        u := x * u;
        #alors H(i+1) vraie
    od
    #Hn vraie
    u;
end;;
```

d La fonction `puissance` fait une multiplication par passagedans la boucle indicée par `i`, soit n multiplications.

2.2 Algorithme naïf récursif

a Pour avoir H_0 vraie, il suffit d'initialiser u à 1. Pour avoir H_n vraie sachant H_{n-1}, il suffit de multiplier u par x.

b
```
puissance2 := proc(x,n)
    if n=0 then 1
    else x*puissance2 (x,n-1)
    fi
end;;
```

L'arrêt et la validité se prouvent simultanément par récurrence sur n au fil des appels récursifs.

c La fonction fait n appels récursifs en incluant l'appel initial de `puissance2 (x,n)`, puisque n diminue de 1 à chaque appel, et fait une multiplication par appel, pour un total de $\mathcal{O}(n)$ multiplications.

2.3 Algorithme rapide de calcul des puissances

a Le quotient et le reste dans la division euclidienne de n par m s'obtiennent respectivement par `iquo(n,m)` et `irem(n,m)`.

On remarque par ailleurs que

$$u_n(x) = u_{2p+r}(x) = u_{2p}(x)u_r(x) = u_p(xx)u_r(x)$$

b En supposant que H_{n-1} est vraie, on ne fait en fait pas grand-chose. Mes excuses pour cette erreur.

On prend plutôt ici pour $H(n)$ la propriété "quel que soit x, $\texttt{puissance}_{rapide}\texttt{(x,k)}$ s'arrête et renvoie $u_k(x)$". On suppose la propriété vraie à tous les rangs $k < n$ (récurrence forte). Pour avoir $H(n)$, en supposant p et r définis comme ci-dessus, il suffit alors de retourner le produit de $\texttt{puissance}_{rapide}\texttt{(x*x,p)}$, et de x ou 1, selon que r vaut respectivement 0 ou 1.

c
```
puissance_rapide := proc (x,n)
local p,r,u;
    if n = 0 then 1;
    else
        p := iquo (n,2);
        r := irem (n,2);
        u := puissance_rapide (x * x, p);
        if r = 0 then
  u;
        else
  u * x;
        fi
    fi
end;;
```

La correction et l'arrêt de la fonction sont justifiés à la quesiton précédente, à l'exception du cas d'arrêt : lorsque $n = 0$, point vers lequel convergent les appels récursifs, d'après l'algorithme d'Euclide, on renvoie directement 1.

d La fonction fait un nombre constant de multiplications par appel récursif, et $ln_2(n)$ appels récursifs, donc un nombre $\mathcal{O}(n)$ multiplications.

2.4 Le calcul de la multiplication par un entier

On remarque qu'il s'agit simplement d'utiliser les lois d'un anneau abélien. Il suffit donc de remplacer, dans les raisonnements et programmes qui précèdent, toutes les occurences de la loi multiplicative (exp,^) par $\times$,*, toutes les occurences de la loi additive ($\times$,*) par +,+ et toutes les occurence du neutre pour l'ancienne loi de groupe (1) par le neutre de la nouvelle loi, 0.

3 Suites récurrentes du premier ordre

3.1 une version itérative

a
```
calculu := proc(a,f,n)
local i,u;
    if n = 0 then a;
    else
        u := a;
        #H0 vraie
        for i from 1 to n do
            #si H(i-1) vraie
            u := f(u);
            #alors H(i) vraie
            od;
            #H(n) vraie
            u
    fi
end;;
```

b Cette procédure fait un appel de `f` par passage dans la boucle, soit $\mathcal{O}(n)$ appels.

c On a $E = \mathbb{N}$, $a = 1$, $f : x \mapsto 2 + x$.
```
> calculu (1,x → 2+x,10);;
                                        21

> calculu (1,x → 2+x,100);;
                                        201
```

d On a $E = \mathbb{R}$, $a = 1$, `f = x → evalf(sqrt(2+x))`. notez l'importance de `evalf`, dans ces conditions !
```
afficheu := proc(n)
    local i;
    for i from 0 to n do
        print(calculu (1,x → evalf(sqrt(2+x)),n));
    od
end;;
```

Cette suite est constante au bout d'un certain rang, puisqu'on sait que 2 est point fixe de f. On obtient le graphe de ses termes avec la commande :
```
plot ([seq ([n,calculu (1,x → evalf(sqrt(2+x)),n)],
                                n=1..100)],style=line);;
```

e
```
afficheu := proc(n)
    local i;
    for i from 1 to n do
        print(calculu ("",,x →"ab"||x||"ba",n));
    od
end;;
```

t_n est la concaténation de n chaînes "ab" suivie de la concaténation de n chaînes "ba".

3.2 une version récursive

a
```
calculu2 := proc(a,f,n)
    if n = 0 then a else f(calculu2 (a,f,n-1)) fi
end;;
```

b Cette fonction appelle `f` une fois par appel récursif, soit $\mathcal{O}(n)$ fois au total.

c

4 Suites récurrentes du second ordre

4.1 une version itérative

a
```
calculu := proc (a,b,f,n)
    local i,u,v,tmp;
    if n = 0 then a;
    else
        if n = 1 then b;
            else
            u := a;
            v := b;
```

```
            #H0 vraie
            for i from 1 to n do
                #si H(i-1) vraie
                tmp := u ;
                u := v ;
                v := f(tmp,u) ;
                #alors H(i) vraie
            od ;
            #H(n) vraie
            u ;
        fi
    fi
end ; ;
```

b Cette fonction appelle f une fois par passage dans la boucle indicée par `i`, soit $\mathcal{O}(n)$ appels récursifs.

c $E = \mathbb{N}$, $a = b = 1$, $f : (x, y) \mapsto x + y$. On reconnaît la suite de Fibonacci.

```
> calculu (1,1, (x,y) → x+y, 10) ; ;
                                        89

> calculu (1,1, (x,y) → x+y, 100) ; ;
                              573147844013817084101
```

4.2 une version récursive

a Donner une fonction récursive `calcul_u2` telle que `calcul_u2(a,b,f,n)` renvoie la valeur de u_n. On prouvera la validité et l'arrêt de cette fonction.

b Déterminer le nombre d'appels de f effectués par cette procédure.

c Tester cette fonction sur la suite v définie précédemment. Tentez de calculer v_{100}.

d Proposez une fonction récursive qui résoud le problème rencontré.

a
```
calculu2 := proc (a,b,f,n)
    if n = 0 then a ;
    else
        if n = 1 then b ;
        else
            f(calculu2(a,b,f,n-1),calculu2(a,b,f,n-2)) ;
        fi
    fi
end ; ;
```

b La fonction fait deux appels de `f` par appel récursif. On a ainsi $C_n = C_{n-1} + C_{n-2} + 2$. La complexité de cette fonction est de l'ordre de 2^n. Plus précisément, si on n'est intéressé que par une borne asymptotique du résultat, on peut considérer : $C_n \begin{cases} 1 & n < 2 \\ C_{n-1} + C_{n-2} + 1 & n \geq 2 \end{cases}$ On remarque qu'il s'agit précisément de la récurrence de Fibonacci. Or, on prouve par induction que $v_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2}^n - \frac{1-\sqrt{5}}{2}^n)$ (formule de Binet). On obtient ainsi $v_n \geq \frac{1+\sqrt{5}}{2}^n - 1$, et quelques secondes avec une calculatrice devraient vous convaincre qu'on a affaire à une complexité de l'ordre de $(3/2)^n$.

c v_{100} n'est pas calculable en un temps raisonnable :
```
> calculu2 (1,1, x,y → x+y, 10) ; ;
Error, (in calculu2) too many levels of recursion
```

d
```
calculu3 := proc (a,b,f,n)
    if n = 0 then a ;
    else
        if n = 1 then b ;
        else
            calculu3(b,a+b,f,n-1) ;
        fi
    fi
end ; ;
```

5 Recherche d'un zéro par dichotomie

a
```
dichotomie := proc (u,v,f,c)
    local m ;
    m := evalf((u+v)/2) ;
    if evalf((v-u)) <= evalf(2*c) then m
    else
        if (f(u)*f(m)) <= 0 then dichotomie (u,m,f,c)
        else
            dichotomie (m,v,f,c)
        fi
    fi
end ; ;
```

b
```
> dichotomie (0,Pi,cos,1/(10^6)) ; ;
                                    1.570795952

> evalf(Pi/2) ; ;
                                    1.570796327
```

c (a) La procédure fait moins de $ln_2(10^6/2) \simeq 19$ étapes.

(b) D'après les indications de l'énoncé, la procédure fait au plus $ln_2(\frac{b-a}{2e})$.

(c) La procédure est de complexité logarithmique, soit très rapide.

(d)
```
dichotomie := proc (u,v,f,c)
    local m,a,b ;
    m := evalf((u+v)/2) ;
    if evalf((v-u)) <= evalf(2*c) then m,1
    else
        if (f(u)*f(m)) <= 0 then
            a,b := dichotomie (u,m,f,c) ;
            (a,(b+1)) ;
        else
            a,b := dichotomie (m,v,f,c) ;
            (a,(b+1)) ;
        fi
    fi
end ; ;

> dichotomie(0,Pi,cos,1/(10^6)) ; ;
                                    1.570795578, 22
```