Informatique tronc commun
Exercices d'algorithmique

Corrigé

29 septembre 2006

1 Calculs récursifs : quelques fonctions

1.

```
factorielle := proc (n)
    if (n=0) then
        1 ;
    else
        n*(factorielle(n-1)) ;
    fi
end ; ;
```

2.

```
deriven := proc (f,n)
local g ;
    if (n=0) then
        f ;
    else
        g :=deriven(f,n-1) ;
        D(g) ;
    fi
end ; ;
```

3.

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

4.

```
somme_tableau := proc (t,n)
local res,i ;
    # H(k) : "res contient la somme
    # des k premiers termes du
    # tableau t"
    res :=0 ;
    # H(0) est vraie
    for i from 1 to (n) do
        # si H(i-1) est vraie
        res := res + t[i] ;
        # alors H(i) est vraie
    od
    # H(n) est vraie
end ; ;
```

5.

```
somme_liste := proc (l,n)
    if n=0 then
        0 ;
    else
        somme_liste(l,n-1)+l[n] ;
    fi
end ; ;
```

1. La validité du programme se prouve en énonçant la récurrence qui définit la facorielle, et su laquelle est calquée le programme. La complexité de ce programme est de n multiplications pour le calcul de $n!$.
2. De même, il suffit de donner la définition de la dérivée n-ième pour justifier de la validité du programme. La complexité de ce programme est celle du calcul de n dérivations pour le calcul de $f(n)$. On remarquera qu'il est impossible de quantifier la complexité de la dérivation (et à fortiori notre programme de calcul de dérivée n-ième)en nombre d'opérations sans connaître la fonction f à laquelle elle s'applique.
3. L'énoncé donne lui-même une justification de la validité de notre programme. Par contre, sa complexité est de n applications de f pour la calcul de u_n. Là encore, impossible de raisonner en nombre d'opérations, on prend cette application elle-même comme unité.
4. La validité du programme est justifiée par la validité de H_n en sortie de boucle. La complexité du programme est en $O(n)$ additions (une addition par passage dans la boucle, n passages).
5. Soit $S(n)$ la somme que l'on souhaite calculer. Alors, $S(0) = 0$ et pour tout n strictement positif, $S(n) = S(n-1) + T(n)$ où $T(n)$ est la valeur de la case en position n du tableau T. On montrerait par récurrence (triviale) que cette fonction effectue n additions.
6. Encore une fois, les commentaires du programme justifient de sa validité. On remarquera qu'il faut ici compter les exponentiations effectuées, plus coûteuses que les multiplications, pour déterminer la complexité du programme, qui est linéaire dans les deux cas.

Attention, même si les fonctions demandées dans les questions 4 et 5 pouvaient être les mêmes (dans ce corrigé sont présentes une version itérative et une version récursive) **il ne faut pas confondre une liste maple et un tableau maple.** Une des différences fondamentales entre une liste et un tableau maple est que les fonctions d'accès par les crochets ne fonctionnent pas de la même manière. Par exemple :

```
>t :=array(1..4,[1,4,9,16]) ;
                          t := [1, 4, 9, 16]
>t ;
                                t
>nops(t) ;
                                1
                                >t[1..2] ;
                             t[1 .. 2]
>l :=[1,4,9,16] ;
                          l := [1, 4, 9, 16]
>l ;
                            [1, 4, 9, 16]
>nops(l) ;
                                4
>l[1..2] ;
                                [1, 4]
```

6.

```
with(linalg);;

eval_poly := proc(p,x)
local res,n,i;
    n := rowdim(p);
    # H(k) : "res contient la somme
    # des k premiers monomes de P"
    res := 0;
    # H(0) est vraie
    for i from 1 to n do
        # si H(i-1) est vraie
        res :=res+(p[i]*(x^(n-i)));
        # alors H(i) est vraie
    od
    # H(n) est vraie
end;;
```

```
eval_poly2 := proc(p,x)
local res,n,i,y;
    n := rowdim(p);
    # H(k) : "res contient la somme
    # de k monomes distincts de p"
    res := 0;
    # H(0) est vraie
    for i from 1 to n do
        # si H(i-1) est vraie
        y :=p[i];
        res :=res+(y[2]*(x^(y[1])));
        # alors H(i) est vraie
    od
    # H(n) est vraie
end;;
```

2 Itération, récursion : calcul de terme de suite

Version itérative

L'arrêt est trivial, la validité est justifiée en commentaire. La complexité en nombre d'applications de f est linéaire.

```
terme_suite_iterative := proc(f,a,b,n)
    local u,v,w,i;
        # H(i) :"u = u_i"
        u :=a;
        v :=b;
        w :=a;
        # H(0) vraie
        for i from 0 to (n-1) do
            #si H(i) vraie
            w :=v;
            v :=f(v,u);
            u :=w;
            #alors H(i+1) vraie
        od;
        # H(n) vraie
        u;
    end;;
```

Version récursive multiple

L'arrêt et la validité sont justifiés par la récurrence donnée par l'énoncé. La complexité d'un appel de cette fonction est, en nombre d'appels récursifs, exponentielle.

```
terme_suite := proc(f,a,b,n)
      local y,z;
      if n=0 then
          a;
      elif n=1 then
          b;
      else
          z :=(terme_suite(f,a,b,n-
2));
          y :=(terme_suite(f,a,b,n-
1));
          f(y,z);
      fi
  end;;
```

Version récursive avec astuce

On calcule à chaque appel de `terme_suite_decalage(f,a,b,n)`, une autre suite v telle que $\forall n \geq 2\ v_n = f(v_{n-1}, v_{n-2})$, $v_0 = b$ et $v_1 = f(a,b)$. On montre aisément que $\forall n \in \mathbb{N}\ u_{n+1} = v_n$, ce qui suffit à prouver la validité de l'algorithme. L'arrêt se prouve en montrant par récurrence qu'au $k-ime$ appel récursif, on calcule une suite w telle que $\forall n \in \mathbb{N}\ u_{n+k} = w_n$, et en remarquant que le calcul s'arrête pour $k = n_0$, où n_0 est l'indice de l'appel initial à `terme_suite_decalage`, puisqu'il suffit de calculer w_0. On en tire également que la complexité de cet algorithme est linéaire en appels récursifs.

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

Version récursive avec stockage

Il s'agit ici d'une simple application de la récurrence proposée par l'énoncé, ce qui assure la validité et l'arrêt de l'algorithme. Par contre, le calcul de la complexité (linéaire) s'effectue en montrant que chaque terme n'a été calculé qu'une seule fois, par étude de cas, à l'aide des conditions de branchement du choix `if` dans le programme `terme_suite_aux`.

```
terme_suite_aux := proc (f,a,b,n)
           global valeurs,indixes ;
                if n=0 then
                    valeurs[1] := a ;
                    indixes[1] := true ;
                elif n=1 then
                    valeurs[2] := b ;
                    indixes[2] := false ;
                elif evalb(indixes[i]=true) then
                    # les terme d'ordre (i-1),(i-2)
                    # ont été calculés,
                    calcul direct
                    valeurs[i+1] :=f(valeurs[i],valeurs[i-1]) ;
                    indixes[i+1] :=true ;
                else
                    # le terme d'ordre (i-1) n'a pas
                    # été calculé, appel récursif simple
                    terme_suite_aux(f,a,b,(i-1))
                    valeurs[i+1] :=f(valeurs[i],valeurs[i-1]) ;
                    indixes[i+1] :=true ;
                fi
            end;;

terme_suite_stockage := proc(f,a,b,n)
            local valeurs,indixes ;
                valeurs :=array(1..n) ;
                indixes :=array(1..n) ;
                terme_suite_aux(f,a,b,n) ;
                valeurs[n] ;
            end;;
```

3 Récursion simple, diviser pour régner

3.1 Exponentiation

Version itérative

L'arrêt et la validité sont justifiés en commentaire.
L'algorithme a une complexité linéaire.

```
exponentiation_iterative := proc(x,n)
       local res,i ;
           # H(i) :"res=x^i"
           res := 1 ;
           # H(0) vraie
           for i from 0 to (n-1) do
               # si H(i) est vraie
               res := res*x ;
               # alors H(i+1) est vraie
           od ;
           res ;
       end;;
```

Version diviser-pour-régner

Version récursive simple

On justifie l'algorithme avec la récurrence $a^n = aa^{n-1}$ qui donne un algorithme linéaire.

```
    exponentiation_recursive := proc(x,n)
                if n=0 then
                    1 ;
                else
                    x*
                    (exponentia-
tion_recursive(x,n-1)) ;
                fi
            end;;
```

On va utiliser la récurrence suivante :

$$\begin{aligned} a^{2n} &= (a^2)^n \\ a^{2n+1} &= (a^n)^2 a \end{aligned}$$

Ici, cela tourne en $\ln n$ (on divise la taille de l'argument par 2 à chaque fois). Cet algorithme est appelé « exponentiation rapide ».

En fait, on peut également partir du bas (en considérant la décomposition de n en base 2), et ce faisant gagner un peu de mémoire. Enfin, notons que cet algorithme n'est pas spécifique à l'exponentiation au sens strict : il s'applique en remplaçant la multiplication par n'importe quelle opération associative sur un ensemble quelconque (en particulier l'exponentiation de matrice).

```
exponentiation_dpr := proc(x,n)
           local y;
               if n=0 then
                   1;
               elif irem(n,2)=0 then
                   y :=(exponentiation_dpr(x,n/2));
                   y*y;
               else
                   y :=(exponentiation_dpr(x,(n-1)/2));
                   y*y*x;
               fi
           end;;
```

3.2 Multiplication

Version itérative

L'arrêt et la validité sont justifiés en commentaire.
L'algorithme a une complexité linéaire.

```
multiplication_iterative := proc(x,n)
      local res,i;
           # H(i) :"res=x*i"
           res := 0;
           # H(0) vraie
           for i from 0 to (n-1) do
               # si H(i) est vraie
               res := res+x;
               # alors H(i+1) est vraie
           od;
           # H(n) vraie
           res;
       end;;
```

Version récursive

On justifie l'algorithme avec la récurrence $an = a + a(n-1)$, qui donne un algorithme linéaire.

```
multiplication_recursive := proc(x,n)
       if n=0 then
           0;
       else
           x+(multiplication_recursive(x,n-
1));
       fi
   end;;
```

Version diviser-pour-régner

On utilise la récurrence suivante :

$$\begin{aligned} a(2n) &= (a+a)n \\ a(2n+1) &= a(2n)+a \end{aligned}$$

De même qu'à l'exercice précédent, on aboutit à une complexité logarithmique (en nombre d'additions, cette fois).

```
multiplication_dpr := proc(x,n)
               local y;
                   if n=0 then
                       0;
                   elif irem(n,2)=0 then
                       y :=(multiplication_dpr(x,n/2));
                       y+y;
                   else
                       y :=(multiplication_dpr(x,(n-1)/2));
                       y+y+x;
```

```
            fi
        end;;
```

4 Produits avancés

4.1 Produit de matrices

On découpe les matrices A et B de taille 2^n de la façon suivante :

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} B = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}$$

Puis on se sert de la formule de multiplication par blocs :

$$AB = \begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{bmatrix}$$

... avant d'appeller récursivement les quatre produits de matrices de taille 2^{n-1}. On arrive ainsi à un algorithme en n^3. En effet :

$$T(2^0) = 1$$
$$T(2^n) = 8T(2^{n-1})$$

Ce qui permet de montrer par récurrence :

$$T(2^n) = 8^n = (2^n)^3$$

```
produit_matrices := proc(A,B)
        local n,m,A1,A2,A3,A4,B1,B2,B3,B4,res1,res2,res3,res4,haut,bas,res;
            n :=rowdim(A);
            if n=1 then
                matrix(A[1]*B[1]);
            else
                m :=(n/2);
                A1 :=submatrix(A,1..m,1..m);
                A2 :=submatrix(A,1..m,(m+1)..n);
                A3 :=submatrix(A,(m+1)..n,1..m);
                A4 :=submatrix(A,(m+1)..n,(m+1)..n);
                B1 :=submatrix(B,1..m,1..m);
                B2 :=submatrix(B,1..m,(m+1)..n);
                B3 :=submatrix(B,(m+1)..n,1..m);
                B4 :=submatrix(B,(m+1)..n,(m+1)..n);
                res1 :=(produit_matrices(A1,B1)&+produit_matrices(A2,B3));
                res2 :=(produit_matrices(A1,B2)&+produit_matrices(A2,B4));
                res3 :=(produit_matrices(A3,B1)&+produit_matrices(A4,B3));
                res4 :=(produit_matrices(A3,B2)&+produit_matrices(A4,B4));
                haut :=(augment(res1,res2));
                bas :=(augment(res3,res4));
                res :=(transpose(augment(transpose(haut),transpose(bas))));
            fi
end;;
```

4.2 Produit de polynomes

Ici, on découpe les polynome P et Q, de degré 2^n, de la façon suivante :

$$P(X) = P_1(X) + XP_2(X)Q(X) = Q_1(X) + XQ_2(X)$$

Où P_1 est le polynome des termes de P de degré inférieur ou égal à 2^{n-1}, et $XP_2(X) = (P(X) - P_1(X))$ (et respectivement pour Q,Q_1,Q_2).

Puis on calcule le produit $P(X)Q(X)$:

$$P(X)Q(X) = P_1(X)Q_1(X) + X(P_1(X)Q_2(X)) + X(P_2(X)Q_1(X)) + X^2(P_2(X)Q_2(X))$$

Cette méthode nous permet de calculer ce produit en quatre appels de produits de polynomes de degré 2^{n-1}. Ceci produit un algorithme quadratique :

$$T(2^n) = 4T(2^{n-1})$$

Donc par récurrence :

$$T(2^n) = 4^n = (2^n)^2$$

```
produit_polys := proc (P,Q)
          local n,m,P1,P2,Q1,Q2,res1,res2,res3,res4,res ;
          n :=vectdim(P) ;
          if n=1 then
              vector([P[1]*Q[1]]) ;
          else
              m :=(n/2) ;
              P1 :=vector([seq(P[i],i=1..m)]) ;
              P2 :=vector([seq(P[i],i=(m+1)..n)]) ;
              Q1 :=vector([seq(Q[i],i=1..m)]) ;
              Q2 :=vector([seq(Q[i],i=(m+1)..n)]) ;
              res1 :=produit_polys(P1,Q1) ;
              res2 :=produit_polys(P1,Q2) ;
              res3 :=produit_polys(P2,Q1) ;
              res4 :=produit_polys(P2,Q2) ;
              res :=vector([seq(res1[i],i=1..m),
                           seq(res1[i+m]+res2[i]+res3[i],i=1..m),
                           seq(res2[i+m]+res3[i+m]+res4[i],i=1..m),
                           seq(res4[i+m],i=1..m)]) ;
          fi
      end; ;
```

5 Récursion mutuelle

```
calcul_u := proc (a,b,n)
     local i,j ;
         if n = 0 then
             a ;
         else
             i := calcul_u (a,b,n-1) ;
             j := calcul_v (a,b,n-1) ;
             i^2+2*j ;
         fi
     end; ;
```

```
calcul_v := proc (a,b,n)
       local i,j ;
           if n = 0 then
               b ;
           else
               i := calcul_v (a,b,n-1) ;
               j := calcul_u (a,b,n-1) ;
               i+3*j ;
           fi
       end; ;
```

Ici on observe la récurrence croisée suivante :

$$\begin{cases} C_u(n) &= C_u(n-1) + C_v(n-1) + 2 \\ C_v(n) &= C_v(n-1) + C_u(n-1) + 2 \end{cases}$$

Par équivalence des rôles joués par u et v, on peut poser $\forall n \geq 0\ C_u(n) = C_v(n)$. On en tire aisément $C_u(n) = 2(C_u(n-1)+1)$, d'où :

$$C_u(n) = C_v(n) = 2^n + \frac{2^{n+1}-1}{2-1} = O(2^n)$$

6 Translation de polynomes

On connaît l'algorithme de Horner, qui permet d'évaluer un polynome P en un point z. On notera :

$$P(X) = \sum_{K=0}^{n} \alpha_k X^k$$

Ici on veut calculer les coefficients du polynome Q tel que $\forall X Q(X) = P(X+a)$.

$$
\begin{aligned}
Q(X) &= \sum_{k=0}^{n} \alpha_k (X+a)^k \\
&= \sum_{k=0}^{n} \alpha_k \sum_{i=0}^{k} \binom{k}{i} X^i a^{k-i} \\
&= \sum_{i=0}^{n} X^i \sum_{k=i}^{n} \alpha_k \binom{k}{i} a^{k-i} \\
&= \sum_{i=0}^{n} X^i \sum_{k=0}^{n-i} \alpha_{k+i} \binom{k+i}{i} a^k
\end{aligned}
$$

On remarque donc que le calcul d'un coefficient de Q revient, à un coefficient binomial près, au calcul d'un polynome en a. Supposons que l'on dispose des coefficients du polynome P rangé dans un tableau `T` :

```
with(linalg) ; ;

# calcul_coeff calcule le coefficient de degré i de Q, étant donnés les
# coefficients de P (dans le tableau T), et la valeur de a.
calcul_coeff := proc (T,i,a)
       local n,k,res ;
           # n le degré de P
           n := vectdim(T)-1 ;
           res := 0 ;
           for k from 0 to (n-i) do
               #attention T[i] → terme de degré i-1
               res := a * res + binomial(k+i,i)*T[k+i+1] ;
           od ;
           res ;
       end ; ;

#translate renvoie les coefficients de Q étant données ceux de P et la
# valeur de a.
translate := proc(T,a)
local n,i,res ;
    # n le degré de P
    n := vectdim(T)-1 ;
    res :=array(0..n) ;
    for i from 0 to n do
        # ici, i est le degré du terme dont on
        # calcule le coefficient
        res[i] := (calcul_coeff(T,i,a)) ;
    od ;
    eval(res) ;
end ; ;
```

7 Décodage de nombres entiers

1.

```
valeur := proc(s)
local y,n,tete,queue ;
    n := length(s) ;
    fin := substring(s,n) ;
    if fin="0" then
        y :=0 ;
    elif fin="1"then
        y :=1 ;
    elif fin="2" then
        y :=2 ;
    elif fin="3" then
        y :=3 ;
    elif fin="4" then
        y :=4 ;
    elif fin="5" then
        y :=5 ;
    elif fin="6" then
        y :=6 ;
    elif fin="7" then
        y :=7 ;
    elif fin="8" then
        y :=8 ;
    else
        #fin="9"
        y :=9 ;
    fi ;
    if n=1 then
        y ;
    else
        debut := substring(s,1..(n-1)) ;
        y+10*(valeur(debut)) ;
    fi ;
end ; ;
```

2.

```
    # cette fonction s'assure simple-
ment que l'écriture décimale de s et t 'est
    # pas précédée de carac-
tères "0" inutiles.
    clean :=proc(s)
        local n,debut ;
        n :=length(s) ;
        if n<=1 then
            s ;
        else
            debut :=substring(s,1) ;
            if debut = "0" then
                clean(substring(s,2..n)) ;
            else
                s ;
            fi ;
        fi ;
    end ; ;

    compare := proc(a,b)
    local n,p,debut_s,debut_t,fin_s,fin_t ;
        s :=clean(a) ;
        t :=clean(b) ;
        n :=length(s) ;
        p :=length(t) ;
        if n>p then
            true ;
        elif p<n then
            false ;
        # ici, les deux nombres sont de même lon-
gueur (n=p)
        elif n=1 then
            (valeur(s)>=valeur(t)) ;
        else
            debut_s := substring(s,1) ;
            debut_t := substring(t,1) ;
            if valeur(debut_s)>valeur(debut_t) then
                true ;
            elif valeur(debut_s)<valeur(debut_t) then
                false ;
            else
                # les pre-
miers chiffres de s & t sont identiques
                fin_s := substring(s,2..n) ;
                fin_t := substring(t,2..n) ;
                compare(fin_s,fin_t) ;
            fi
        fi
    end ; ;

    > compare("12345678998765432112345678998765432 1",
              "14725836996385274114725836996385274 1") ;
                                        false
```

8 Suite de Fibonacci

1.

```
calcul_fibo := proc (n)
    if n<=1 then
        1;
    else
        calcul_fibo(n-1)+calcul_fibo(n-
2);
    fi
end;;
```

2.

```
calcul_fibo_double := proc(n)
    local a,b;
        if n=0 then
            1,1;
        else
            (a,b) :=calcul_fibo(n-1);
            a+b;
        fi
    end;;
```

3.

```
calcul_fibo_dpr := proc(n)
    if n<=1 then
        1
    else
        if (n mod 2) = 0 then
            calcul_fibo_dpr(n/2)^2+calcul_fibo_dpr(n/2-1)^2;
        else
            p :=(n-1)/2;
            # 2(n-1)/2+1 = n
            q :=p+1;
            # appels récursifs
            calcul_p_moins_1 := calcul_fibo_dpr(p-1);
            calcul_q_moins_1 := calcul_fibo_dpr(q-1);
            # (q-1) = p
            calcul_p := calcul_q_moins_1;
            # q = (p+1)
            calcul_q := calcul_p + calcul_p_moins_1;
            calcul_p*calcul_q+calcul_p_moins_1*calcul_q_moins_1;
        fi;
    fi;
end;;
```

9 Une fonction mystérieuse

1. Montrons que la fonction f est bien définie par induction descendante sur x entier naturel.
   - Si $x \leq 1$ alors f est bien définie.
   - Si $x \geq 1$ alors :
     - Si x est pair alors $f(x) = 2f(x/2)$. Par ailleurs, $x/2 < x$, car $x \in \mathbb{N}^*$, donc $f(x/2)$ est bien défini par hypothèse d'induction. $f(x)$ est donc bien défini.
     - Si x est impair , alors $f(x) = 1 + f(x+1)$. Par ailleurs $x+1$ est pair, donc $f(x) = 1 + 2f((x+1)/2)$. Par ailleurs $(x+1)/2 < x$, car $x \in \mathbb{N}^*$ est impair, donc $f((x+1)/2)$ est bien défini par hypothèse d'induction. $f(x)$ est donc bien défini.
2. Montrons que $f(x) + x$ est une puissance de 2 par induction descendante sur x.
   - Si $x \leq 1$ alors $f(x) + x = 1$ ou $f(x) + x = 2$.
   - Si $x \geq 1$ alors :
     - Si x est pair, alors $f(x) + x = 2(f(x/2) + x/2)$, et comme $x > x/2$, par hypothèse d'induction, $f(x/2) + x/2$ est une puisssance de 2. $f(x) + x$ est donc une puissance de 2.
     - Si x est impair, alors $f(x) + x = 2(f((x+1)/2) + (x+1)/2)$, et comme $x > (x+1)/2$, par hypothèse d'induction, $f((x+1)/2) + (x+1)/2$ est une puissance de 2. $f(x) + x$ est donc une puissance de 2.
3. On montrerait aisément par une induction similaire à la question précédente, que :

$$f(n) = 2^{E(log_2(n)+1)} - n$$

... où E est la fonction partie entière.

10 Calcul du reste et du quotient dans la division euclidienne

1. `a=0 b=n`
2. La condition que l'on recherche est $b < p$, donc on effectue les affectations suivantes :
   ```
   a := a+1
   b := b-p
   ```
3.
```
DE := proc(n,p)
local a,b;
    a := 0;
    b := n;
    while (b>=p) do
        a :=(a+1);
        b :=(b-p);
    od;
    # H est vraie et b<p,
    # donc a=q, b=r
    (a,b);
end;;
```

```
DE_aux := proc (n,p,a,b)
       if b>=p then
           DE_aux(n,p,a+1,b-p);
       else
           (a,b);
       fi
   end;;

DE_rec := proc (n,p)
   DE_aux(n,p,0,n);
end;;
```

L'arrêt et la validité de cette procédure sont justifiés par la récurrence d'Euclide.

11 Calcul du PGCD

1. $n = \max(a,b)$ et $p = \min(a,b)$.
2. Si $p = 0$, alors $n \wedge p = n$.
3.
```
GCD := proc(a,b)
local n,p,r;
    n := max(a,b);
    p := min(a,b);
    if (p=0) then
        n;
    else
        while p<>0 do
            r :=(n mod p);
            n :=p;
            p :=r;
        od;
        n;
    fi;
end;;
```

```
GCD_rec := proc(a,b)
        if a<b then
            GCD_rec(b,a);
        elif b=0 then
            a;
        else
            GCD_rec(b,(a mod b));
        fi
    end;;
```

4. Montrons que si $a \geq b > 0$ et si `GCD_rec(a,b)` fait $n \geq 1$ appels récursifs, alors $a \geq f_n$ et $b \geq f_{k-1}$ par récurrence sur n. On fonde l'induction pour $n = 1$ en disant qu'alors $b \geq 1 = f_0$ et $a \geq 1 = f_1$.
   Comme $b > (a \mod b)$, dans chaque appel récursif l'hypothèse selon laquelle $a \geq b$ est conservée.
   Supposons que la propriété est vraie si $n-1$ appels sont effectués : puisque $n > 0$ on a $b \geq 1$ et `GCD_rec(a,b)` appelle `GCD_rec(b, a mod b)` récursivement, qui effectue à son tour $n-1$ appels récursifs. On a donc $b \geq f_n$ par hypothèse de récurrence, et $a \mod b \geq f_{n-1}$. On a $b + (a \mod b) = b + (a - \lfloor a/b \rfloor b) \leq a$.
   On en tire ainsi $a \geq f_{k-1} + f_k$, d'où le résultat.
5. Ceci se montre sans difficulté par récurrence sur n. On rappelle la formle de Binet :
$$f_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$
6. On tire le résultat des deux questions précédentes. La version itérative a la même complexité que la version récursive, si l'on considère le nombre d'affectations à la place du nombre d'appels récursifs comme mesure de complexité.