```
######################## PROBLÈME 1: #########################
# solution récursive
u := proc(n)
  options remember ;  # nécessaire!
  111 - 1130/u(n-1) + 3000/(u(n-2)*u(n-1)) ;
end:
u(0) := 11/2:
u(1) := 61/11:
# solution itérative
u2 := proc(n)
  local t, i, n_1, n_2 ;
  n_2 := u2(0) ;
  n_1 := u2(1) ;
  for i from 2 to n do
    t := 111 - 1130/n_1 + 3000/(n_2*n_1) ;
    n_2 := n_1 ;
    n_1 := t ;
  od ;
  t ;
end:
u2(0) := 11/2:
u2(1) := 61/11:
# solution approchée (utilise la variable globale d, supposée entière)
d := 20:
u3 := proc(n)
  global d ;
  options remember ;
  evalf(111 - 1130/u3(n-1) + 3000/(u3(n-2)*u3(n-1)), d) ;
end:
u3(0) := evalf(11/2, d):
u3(1) := evalf(61/11, d):
# solution approchée (propre, évite de modifier/restaurer Digits)
ua := proc(n, d)
  local u ;
  u := proc(n)
    options remember ;
    evalf(111 - 1130/u(n-2) + 3000/(u(n-2)*u(n-1)), d) ;
  end ;
  u(0) := evalf(11/2, d) ;
  u(1) := evalf(61/11, d) ;
  u(n) ;
end:
```

```
eq := (111 - 1130/x + 3000/(x*x) = x):
solve (eq, x) ;

####################### PROBLÈME 2: #######################
# a)
plist := proc(n)
  local i, L ;
  L := 2 ;
  for i from 3 to n by 2 do
    if isprime(i) then L := L,i ; fi ;
  od ;
  [L] ;
end ;

# autre:
plist := n -> select( isprime, [seq(i, i=1..n)] ) ;

# b)
plist2 := proc(n)
  local i,j, L, T ;
  T := array([true$n]) ;
  L := NULL ;
  for i from 2 to n/2 do # si 2i > n la boucle en j est vide
    if T[i] then
      L := L,i ;
      for j from 2*i to n by i do T[j] := false ; od
    fi ;
  od ;
  [L] ;
end ;

# un peu plus efficace (construit la liste à la fin):
plist3 := proc(n)
  local i,j, L, T ;
  T := array([true$n]) ;
  for i from 2 to n/2 do
    if T[i] then
      for j from 2*i to n by i do T[j] := false ; od
    fi ;
  od ;
  # ne marche apparemment pas dans la version Etudiant, mais est correct
  select(i->T[i], [ seq(i, i=1..n) ]) ;
end ;

####################### PROBLÈME 3: #######################
#  a)
```

```
F[2*n]    := F[n]^2 + F[n-1]^2 ;
F[2*n-1] := 2*F[n]*F[n-1] - F[n-1]^2 ;

F[2*n+1] := F[2*n] + F[2*n-1] ;
F[2*n+2] := F[2*n+1] + F[2*n] ;
F[n-1]   := F[n+1] - F[n] ;

simplify(F[2*n+1]) ;
simplify(F[2*n+2]) ;

#  b)

# récursion dyadique
F := proc(k)
  local n,a,b ;
  options remember ;

  n := ceil(k/2) ;
  if type(k, even) then
    F(n)^2 + F(n-1)^2 ;
  else
    F(n-1) * (2*F(n) - F(n-1)) ;
  fi ;
end ;
F(0) := 1 ;
F(1) := 1 ;

# une variante plus efficace avec un seul appel récursif
# T(n)[1] calcule F_n
T := proc(n)
  local q,r,t,a,b ;
  q := iquo(n, 2, 'r') ;
  t := T(q) ; a := t[1] ; b := t[2] ;
  if (r = 1) then [a^2 + b^2, b*(2*a + b)] ;
             else [a*(2*b - a), a^2 + b^2] ;
  fi ;
end ;
T(0) := [0,1] ;
T(1) := [1,1] ;

# récursion directe, trop profonde si n est trop grand
F2 := proc(n)
  options remember ;
  F2(n-1) + F2(n-2) ;
end ;
F2(0) := 1 ;
F2(1) := 1 ;

# version itérative
```

```
F3 := proc(n)
  local n_1, n_2, i, t ;
  n_2 := F3(0) ;
  n_1 := F3(1) ;
  for i from 2 to n do
    t := n_2 + n_1 ;
    n_2 := n_1 ;
    n_1 := t ;
  od ;
  t ;
end ;
F3(0) := 1 ;
F3(1) := 1 ;
# c)
FF := n-> matrix([[F[n+2],F[n+1]], [F[n+1],F[n]]]) ;
A := matrix([[1,1], [1,0]]) ;
F4 := proc(n)
  evalm(A^n)[1,1] ;
end ;
simplify(evalm(FF(n-1)^2)) ;
evalm(FF(n) &* A) ; # soit FF(n) = A^(n+2)
# d)
simplify(evalm(FF(n-1)^3)) ;
# récursion triadique (plus lent que dyadique)
F5 := proc(k)
  local n, r ;
  options remember ;
  r := irem(k,3) ;
  if r = 0 then
    n := (k-3)/3 ;
    F5(n+1)^3 + 3*F5(n+1)*F5(n)^2 - F5(n)^3 ;
  elif r = 2 then
    n := (k-2)/3 ;
    3*F5(n)*F5(n+1)^2 - 3*F5(n)^2*F5(n+1) + 2*F5(n)^3 ;
  else   #  r = 1
    n := (k-1)/3 ;
    F5(n+1)^3 - 3*F5(n)*F5(n+1)^2 + 6*F5(n)^2*F5(n+1) - 3*F5(n)^3 ;
  fi ;
end ;
F5(0) := 1 ;
```

```
F5(1) := 1 ;
# e)
# précalculs: phi et a,b
z := solve(x^2 = x + 1, x) ;
if (evalf(z[1]) < 1) then
  phi := z[2] ;
else
  phi := z[1] ;
fi ;
z := solve({a + b = 1, a * phi - b / phi = 1}, {a,b}) ;
a := subs(z, a) ;

F6 := proc(n)
  local d ; # = précision nécessaire (10 chiffres de rab)
  d := ceil(evalf(ln(a) + n * ln(phi) / ln(10))) + 10 ;
  round(evalf(a * phi^n, d)) ;
end ;

######################## PROBLÈME 4: ########################
# a)
fact := proc(n)
  n * fact(n-1) ;
end ;
fact(0) := 1 ;

fact2 := proc(n)
  local i,N ;
  N := 1 ;
  for i from 2 to n do
    N := N*i
  od ;
  N ;
end ;

# b)
# multiplication des éléments d'une liste par "divide and conquer"
# 1ère version
mymul := proc(L)
  local i, x, k, lx ; # x = liste des éléments a multiplier
                      # lx= nombre d'éléments de x
  lx := nops(L) ;
  if   (lx = 0) then RETURN(1) ;
  elif (lx = 1) then RETURN(L[1]) ;
  fi ;
  x := array(L) ; # copie locale. Ses éléments sont remplacés par
```

```
                    # les produits 2 a 2 successifs (array car une liste
                    # serait inefficace)
  k := lx ;
  while (k > 1) do
    lx := k ; k := 0 ;
    for i to lx-1 by 2 do
      k := k+1 ;
      x[k] := x[i] * x[i+1] ;
    od ;
    if (i = lx) then
      k := k+1 ;
      x[k] := x[i] ;
    fi
  od ;
  x[1] ;
end ;
# c)
# 2eme version:
mymul2 := proc(L)
  local i, x, k, lx ; # x = liste des elements a multiplier
                      # lx= nombre d'elements de x
  lx := nops(L) ;
  if   (lx = 0) then RETURN(1) ;
  elif (lx = 1) then RETURN(L[1]) ;
  fi ;
  x := array(L) ;
  k := lx ;
  while (k > 1) do
    lx := k ; k := 0 ;
    for i to lx/2 do
      k := k+1 ;
      x[k] := x[i] * x[lx+1-i] ;
    od ;
    if (i = (lx+1)/2) then
      k := k+1 ;
      x[k] := x[i] ;
    fi
  od ;
  x[1] ;
end ;
# d)
myfact := n -> mymul( [ seq(i, i=1..n) ] ) ;
```

```
# légèrement plus rapide (produits plus équilibrés)
myfact2 := n -> mymul2( [ seq(i, i=1..n) ] ) ;
```