CHAPITRE 4

LE PASSAGE DES PARAMETRES ET LA RECURSIVITE

Ce chapitre est le plus important de la première partie, on y expose les notions de variables globales et locales ainsi que de paramètres passés par valeur ou par adresse, notions qui sont extrêmement mal comprises des débutants.

Le douloureux problème du passage des paramètres

Une procédure est un sous-programme réalisant certaines actions suivant les valeurs de ses paramètres. Ainsi peut-on décrire le travail à effectuer pour afficher une table de multiplication, et lorsqu'on en désirera une, il sera évidemment nécessaire de préciser laquelle, la table de 8 par exemple. On aura donc une procédure à un seul paramètre. Pour programmer le travail en question, ce paramètre devra avoir un nom, S par exemple, et lors de "l'appel", S prendra la valeur 8 (transmission par valeur). Lors d'un autre appel, S prendra une autre valeur. Si maintenant on désire produire une table de longueur variable, (jusqu'à 10 ou 12 ou 15 etc...), on construira une procédure à deux paramètres de façon à l'utiliser en donnant deux valeurs numériques, la table, et jusqu'où on veut aller.
Une fonction en Pascal est une procédure qui renvoie un résultat, ce n'est rien d'autre que la notion mathématique de fonction (si bien sûr il n'y a pas de vilains effets de bord). La différence visible entre la notion de fonction et celle de procédure réside dans le fait que dans l'en-tête de la seconde figure aussi la liste de ses paramètres avec leurs types, alors qu'en plus, pour une fonction, celle-ci est suivie de la déclaration de type de son résultat.
Il faut bien comprendre que les noms choisis pour les paramètres sont tout à fait arbitraires et n'ont à voir que fortuitement avec ceux choisis pour les variables ou constantes du programme. De façon plus imagée, on définit couramment en mathématique la fonction f(x) = x+3, il est évident que la fonction f(y) = y+3 est la même. Si on pose x = 5, f(x) aura pour valeur 8, x n'ayant pas été modifié ce qui n'aurait aucun sens.

La transmission par adresse (ou par référence) est fort différente, signalée en Pascal par le préfixe "var" devant le paramètre formel, elle indique que le paramètre réel (l'objet lors de l'appel) peut être altéré, voire créé (initialisé) lors de l'appel de la procédure. C'est ce que l'on appelle un paramètre accessible en lecture et en écriture, ou encore un paramètre donnée/résultat. Par exemple la fonction Pascal définie par :

```
f (var  x : integer) : integer ; begin   x := x + 3 ; f := x   end;
```

aura l'effet suivant, si on initialise x à 5 et y à 10, f(x) et f(y) auront bien pour valeurs respectives 8 et 13, mais après ces calculs, x et y conserveront les dites valeurs 8 et 13. On dit alors que la fonction f provoque un effet de bord et c'est ce qu'il faudra s'efforcer d'éviter.

Rappel : la notion de variable est celle qui désigne au contraire un objet physique, ainsi X := 3 affecte la valeur 3 à un objet nommé X qui n'a rien à voir avec l'objet Y ou Z ... Une variable est globale si elle est utilisable par l'ensemble du programme, locale à une procédure si elle n'a de sens que dans celle-ci. Ces 4 notions sont illustrée dans le programme du paragraphe suivant.

La récursivité

C'est une notion fondamentale que l'on peut décrire simplement en parlant de la faculté d'une procédure de s'appeler elle-même. On cite souvent la définition récursive du verbe "marcher" comme : "marcher" = "mettre un pied devant l'autre" puis "marcher", en programmation une telle définition bouclerait indéfiniment (exercice 3).
Le programme qui suit est destiné à montrer les différentes façons d'appeler une procédure ou une fonction, à bien marquer les distinctions entre variables globales et locales d'une part ; paramètres accessibles en lecture ou en écriture d'autre part ; et enfin à introduire la notion essentielle de récursivité. On se propose de programmer des choses très simples telles que les valeurs de 1+2+3+....+n , 1*2*3*....*n , 2^n etc...

```
program potpourri ;
                var X : real ; Y : integer ;        { Ce sont les variables globales du programme. }
function som (P : integer ) : integer;              { P est un paramètre formel }
                var I, J : integer ;
{ Sont des variables locales à la fonction "som" elles ne servent que temporairement au calcul du résultat.}
                begin J := 0 ;
                for I := 0 to P do J := J + I;      { Définition itérative de som }
                write ('La somme des entiers inférieurs vaut :');
{ Ceci est un effet de bord, c'est-à-dire que l'impression à l'écran de cette phrase ne se fera que si le programme
appelle som, mais elle se produira chaque fois qu'il l'appellera .}
                som := J end;
function fac (Q : integer ): integer ;
                begin if Q = 0 then fac := 1 else fac := Q*fac (Q-1) end;
{ Définition récursive de "fac", le programmeur n'a pas besoin de variables locales. "fac" s'appelle elle-même, le
même nom Q servant à désigner toute une suite d'instanciations, par exemple fac(2) appelle fac(1) qui appelle
fac(0). On constatera qu'une définition itérative impose toujours l'emploi d'une variable locale, ne serait-ce que
pour évaluer le test d'arrêt, alors qu'ici, ce test est décidé par la valeur d'un paramètre.}
function pui (R : integer ) : real ;
                function sgn (S : real ) : integer ;
                { Fonction locale à une autre fonction, "sgn" ne pourra être appelée en dehors de "pui" }
                begin   if S < 0        then sgn := -1
                                        else if S > 0      then sgn := 1
                                                           then sgn := 0
                end ;
begin { Début de la définition récursive de "pui" suivant le signe du paramètre }
                case  sgn (R)  of       1       : pui := 2*pui (R - 1);
                                        0       : pui := 1;
                                        -1      : pui := 1/ pui (-R) end {du "case"}
end ;
procedure mul (S : integer ) ;
{ Contrairement aux fonctions, les procédures ne renvoient pas un résultat. On désire simplement faire
apparaître une table de multiplication. }
                var T : integer ;
                begin for T := 1 to 12  do writeln (T, ' * ', S ,' = ', T*S); end ;

procedure lec (var U : real ; var V : integer );
{U et V sont des paramètres accessibles en écriture, ce qui est précisé par le mot "var". Le but de cette procédure
est seulement d'obtenir un entier. Si U et V avaient des valeurs avant l'appel de "lec", celles-ci sont modifiées,
sinon elles sont crées.}
                begin write ('Donnez un nombre '); readln (U); V := round (U) end;

{Ici commence le programme proprement dit,  qui, comme la plupart des programmes Pascal, doit se contenter
d'appeler ses sous-programmes. }
begin lec (X, Y) ; if Y > 0 then writeln (som (Y),'  factorielle : ' , fac (Y) ) ;
writeln ('exponentielle en base deux : ', pui (Y)) ; mul (Y) end.
```

Ce programme demande encore quelques commentaires; un effet de bord ou effet marginal, a-t-on dit, est une action quelconque (affectation , impression ...) produite lors d'un appel de fonction, il est recommandé de les éviter sauf naturellement si on souhaite expressément de telles actions, ce qui est le cas des procédures "mul" et "lec".

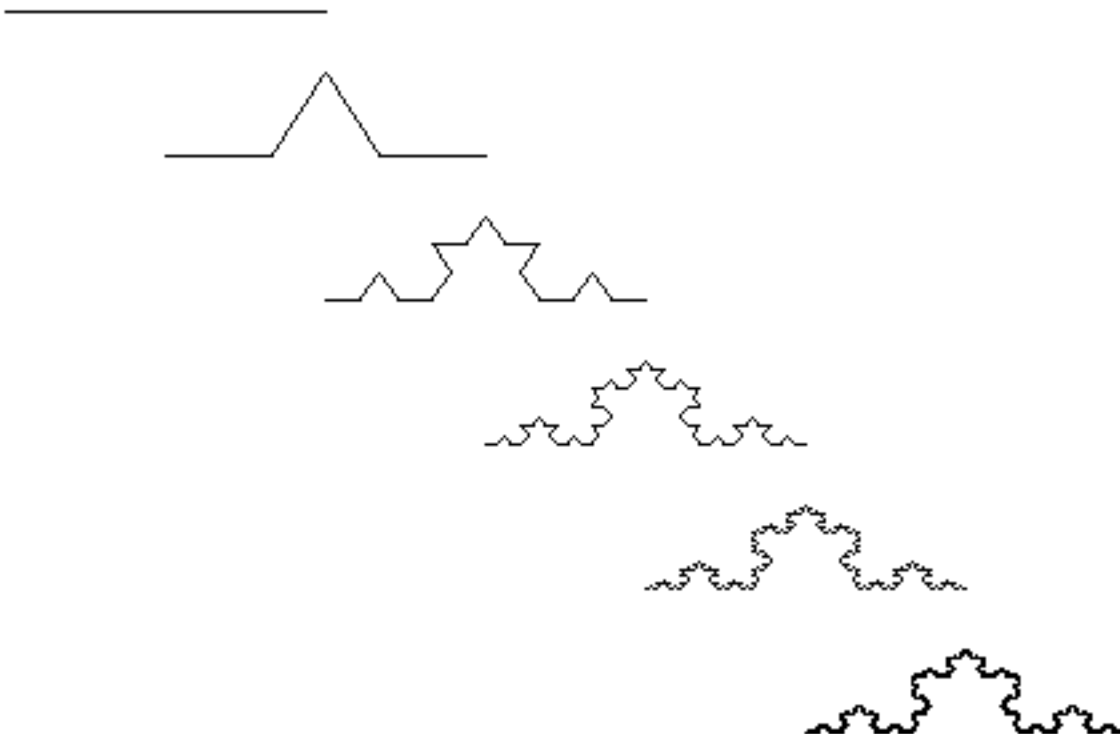
Récursivité terminale

Supposons que la fonction f soit définie par f(x) = si R(x) alors g(x) sinon h(x, f(k(x))) où x est éventuellement un n-uplet de valeurs. C'est l'expression générale d'une fonction récursive "non terminale" : si la relation R est satisfaite, alors le calcul se termine avec celui de l'évaluation de la fonction g, par contre si R n'est pas satisfaite, alors au moyen d'une transformation k, il faut faire un autre appel de f (appel récursif) qui lorsqu'il sera terminé n'achevera pas le calcul initial car il faut avoir conservé la valeur de x pour exécuter h (x, ...).
L'appel n'est pas terminal, signifie du point de vue informatique que chaque appel doit avoir une zone propre en mémoire où il conserve tout son contexte : les paramètres qu'on lui a passé et surtout la marque (si plus d'un appel) de l'endroit où il a été interrompu afin de pouvoir "terminer" son travail.
Tout autre est la définition f(x) = si R(x) alors g(x) sinon f(k(x)) car, hormis le cas où l'appel termine tout de suite quand R(x) est vrai, l'appel suivant pour la valeur k(x) termine en même temps l'appel précédent. La récursivité terminale n'a pas besoin d'une occupation de la mémoire qui soit proportionnelle au nombre d'appels, elle est constante tout comme pour un calcul itératif.
Exemple, la fonction factorielle de façon récursive terminale :

```
function facbis (n, r : integer) : integer;
        begin if n = 0   then facbis := r { cas où on termine vraiment, le résultat est r }
                         else facbis (n - 1, n*r)
        end;
function fac (n : integer) : integer;
        begin fac := facbis (n, 1) end; { c'est un simple appel à la fonction générale }
```

Récursivité et graphique

Les courbes fractales constituent un bon exemple de programme graphiques récursifs, très longs et difficiles à programmer sans récursivité. Prenons le cas des courbes de Von Koch où un segment va être divisé en n (ici n = 4) segments chacun égaux à 1/r du segment initial (ici r est égal à 3 et la profondeur c est imagée de 0 à 5)

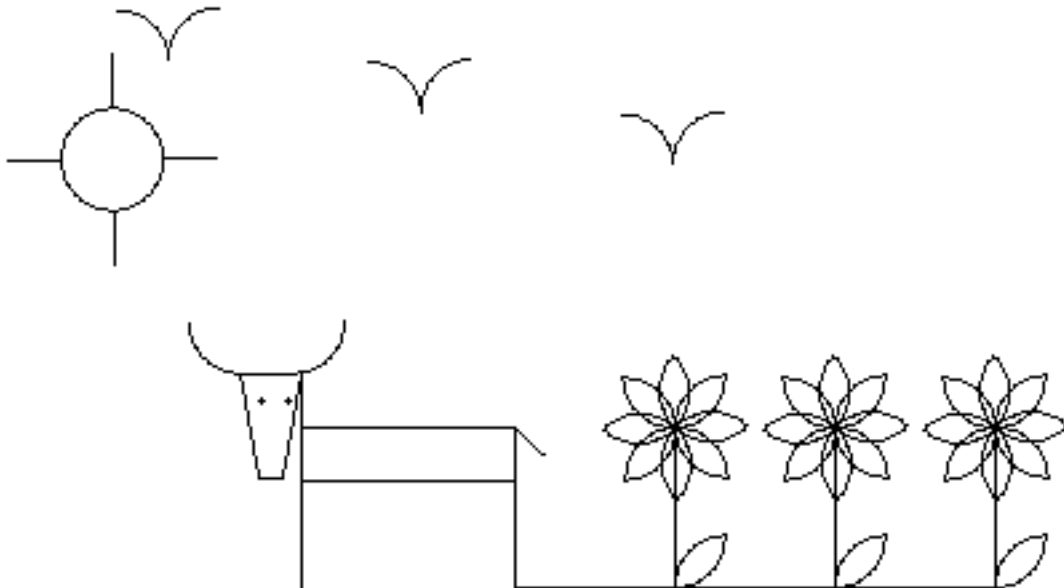


Plus généralement si le segment sert de premier vecteur d'une base orthonormée du plan, et que l'on découpe en n segments non nécessairement égaux, on pourra placer les coordonnées des
points de jointure dans deux tableaux u et v avec u(0) = v(0) = 0 et u(n) = 1, v(n) = 0. L'algorithme est donc le suivant :

```
dragon (c, n : entiers x1, y1, x2, y2 : réels u, v : tableaux)
                {va développer la fractale d'ordre c sur le segment joignant (x1, y1) et (x2, y2) }
si  c = 0       alors on place (x1,y1) et on trace (x2, y2)
                sinon pour i allant de 0 à n-1 on éxecute le dragon (c-1, n) du point
                        x1+u[i]*(x2-x1)+v[i]*(y1-y2), y1+u[i]*(y2-y1)+v[i]*(x2-x1) vers le point
                        x1+u[i+1]*(x2-x1)+v[i+1]*(y1-y2), y1+u[i+1]*(y2-y1)+v[i+1]*(x2-x1)
                        avec les mêmes tableaux u et v
```

Dimension d'une courbe fractale : par analogie avec les surfaces et volumes on dit que si c'était une surface, d'une étape à l'autre on en a n fois plus mais r fois plus petit, on devrait avoir $n = r^2$ et si c'était un volume $n = r^3$. Or ici on a $n = r^d$ qui peut donner la dimension d = ln(n) / ln(r) soit ln(4) / ln(3) pour la courbe de Von Koch. Si on exerce sur un motif du dessin une similitude de rapport r, et que l'on obtient n de ces mêmes motifs alors d est le rapport des log de n et de d.

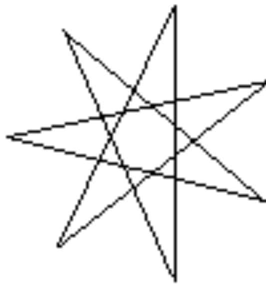
Manipulation de la tortue



Il s'agit d'une bibliothèque de fonctions présente dans le turbo-pascal mais surtout connue au travers du langage LOGO lui même dérivé du LISP, dans laquelle un mobile appelé "tortue" peut être déplacé grâce à des instructions du genre "avancer", "reculer", "tourner à droite", "lever le crayon", "gommer", etc...

Par exemple la construction d'un polygone régulier étoilé ou non se fait très simplement par:

```
program figures; {turbo-pascal sur Mac}
                uses memtypes, quickdraw , turtle;

procedure poly (n, C : integer);                {polygone à n côtés valant tous C}
                var i : integer;
                begin for i := 1 to n do  begin forwd (C); turnleft (360 div n)  end end;

procedure etoile (n, p, C : integer);           {étoile à n sommets joints p à p, de côté C}
                var i : integer;
                begin for i := 1 to n do  begin forwd (C); turnleft ((360*p) div n ) end end;

begin etoile (7, 3, 100) end.
```



Le repère est centré sur l'écran et sur le Macintosh est [-240, 240]*[-140, 140], les instructions sont :

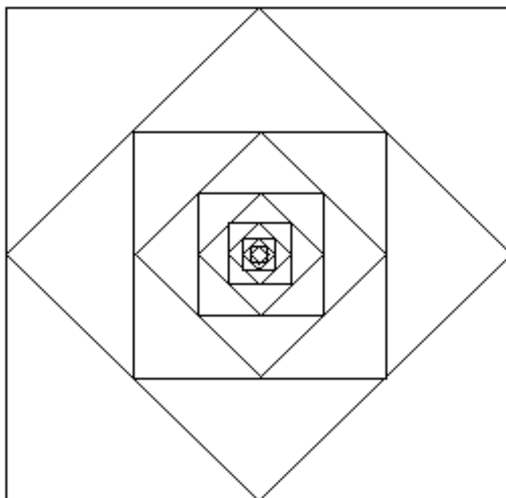
clear; efface l'écran et remet la tortue au centre
home; remet la tortue au centre dirigée vers le haut (north)
setposition(..., ...); force la position de la tortue dans le repère
setheading(...); force sa direction en degrés
penup; ne trace pas lors des déplacements
pendown; trace à l'écran
forwd (...); avance de la distance spécifiée qui doit être un nombre entier
back (...); recule
turnleft (...); fait tourner à gauche de l'angle spécifié en degrés
turnright (...); à droite

On peut utiliser en outre, les constantes "north" = 90°, "south", "east" et "west", et les fonctions "xcor", "ycor", "heading" donnent respectivement l'abscisse, l'ordonnée et la direction de la position actuelle de la tortue.

Un autre exemple (récursif) peut être donné par une suite de carrés emboités :

```
procedure carres (C : integer);
        begin
        if C > 5 then   begin poly (4, C); forwd (C div 2);
                        turnleft (45); carres (round (C / sqrt(2)) )
                        end
        end;
```

On appelera la procédure avec par exemple : setposition (240, -140); carres (250);



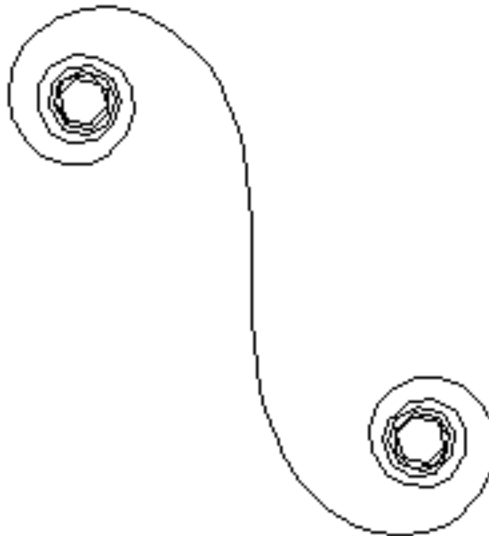
Naturellement, la géométrie de la tortue doit être utilisée à propos, ainsi si une courbe n'est donnée que par son équation intrinsèque c'est à dire par une équation donnant le rayon de courbure $R = ds/d\theta$ où s est la longueur parcourue sur la courbe, et θ l'angle polaire de la tangente, alors on peut faire tracer la courbe par une succession de petits pas où le virage à

prendre dϕ (variation de tangente) est calculé en fonction de ds. Le meilleur exemple est quand même la spirale de Cornu où R est proportionnel à s, ici on a pris ds = 9 et dϕ = 0,1s. Le dessin sera demandé par le programme :

```
begin  home; cornu (0,0); home; setheading (south); cornu (0,0) end.
```

utilisant le sous-programme :

```
procedure cornu (L, A: integer);
        var da : integer;
        begin if L < 600 then    {l'arrêt se fait sur une longueur maximale de 600}
                begin da := round (L*0.1); turnleft (da); forwd (9); cornu (L+9, A+da) end
        end;
```



Les essais successifs ou "backtracking", exemple des reines de Gauss

Nous exposerons cet algorithme sur un exemple bien connu : les reines de Gauss. Etant donné un échiquier 8*8, on cherche à placer 8 reines qui ne soient pas en position de prises mutuelles, cela oblige à avoir une reine sur chaque ligne, une sur chaque colonne, et jamais deux sur toute transversale.
L'algorithme général d'un tel problème est :

Back(i) donne vrai si c'est possible de continuer avec le choix numéro i arrêté, faux sinon.
Si i est le dernier choix à faire alors c'est fini, on renvoie vrai.
Sinon, on examine toutes les possibilités de choix satisfaisant aux contraintes du problème, pour l'étape suivante, en stoppant cet examen dès que l'une indique que back (i+1) est vrai, et en ce cas back(i) sera vrai.
Si tous ces appels répondent faux, alors back(i) sera faux.
Le problème général est étudié au chapitre sur les pointeurs.

Pour les dames de Gauss, on utilisera une table t, telle que t[i] contienne le numéro de colonne de la reine située dans la ligne i, t sera une variable globale.

```
program gauss;
const n = 8;
var t : array [1..n] of integer ;

procedure afficher (m : integer) ; {affiche le tableau t de dimensions m*m}
        var i, j : integer;
        begin   for  i := 1 to m  do        begin for j := 1 to t[i-1] do write ('-': 3); write ('R': 3);
                                            for j := t[i+1] to m do write ('-': 3); writeln
                                            end end;
```

```
function nonprise (i, j, k, l : integer) : boolean;
{indique avec i < k que les reines située en i, j et en k, l ne sont pas en prise}
                    begin if j = l    then nonprise := false
                                      else if abs (j-l) = k-i     then nonprise := false
                                                                  else nonprise := true
                    end;

function caselibre (i, j : integer) : boolean;
{vérifie que la position i, j  est possible par rapport à toutes les lignes précédentes}
var l : integer; {sera un numéro de ligne}          drap : boolean; {sera le résultat }
begin l := 1; drap := true;
while drap and (l < i) do         begin drap := nonprise (l, t[l], i, j);
                                  l := l+1
                                  end;
caselibre := drap
end;

function reine (i : integer) : boolean;
{répond vrai si, t étant bien rempli jusqu'à i compris, il est possible de continuer le remplissage du tableau}
var k : integer; poss : boolean;
begin if i=n      then      reine := true
                            else      begin poss := false; k := 1;
                                      repeat   if  caselibre (i+1, k)      then     begin t[i+1] := k;
                                                                                    poss := reine (i+1)
                                                                                    end ;
                                               k := k+1
                                      until poss or (k > n)
                                      reine := poss
                                      end;
end; {bien regarder la structure de ce programme et indenter de cette façon}

begin {programme} if reine (0) then afficher (n) end.
{la procédure "afficher (n)" devant faire un affichage du tableau "reine" jusque n*n}
```

R							
						R	
				R			
							R
	R						
			R				
					R		
		R					

Voici le résultat pour n = 8, on pourra maintenant modifier le programme afin d'afficher toutes les solutions. Ce problème sera réexaminé au chapitre 7. Maintenant il n'est pas difficile d'obtenir toutes les solutions (n est toujours une constante) grâce à :

```
procedures dames (i : integer) ;
                    var k : integer;
begin if i = n      then afficher (n)
                    else for k := 1 to n do if caselibre (i+1, k) then   begin t[i+1] := k;
                                                                         dames (i+1)
                                                                         end
end;
```

Le programme serait alors simplement lancé par "dames (0)".

4-1° Ecrire "som" récursivement, puis par la formule 1+2+...n = n(n+1)/2 .

4-2° Que se passe-t-il si on introduit dans "fac" un effet de bord analogue à celui de "som" ?

4-3° Que se passe-t-il si on ne définit la fonction fac (N) seulement que par N*fac (N - 1)?

4-4° Ecrire "fac" itérativement.

4-5° Suite de Syracuse. On considère la suite dont le premier terme est un entier positif rentré au clavier, et dont chaque terme est ensuite obtenu comme la moitié du terme précédent si celui-ci est pair , ou bien le successeur de son triple dans le cas contraire. Ecrire un programme donnant tous les termes de la suite jusqu'à ce que l'un d'entre eux soit égal à 1. (Faire des essais à la main avec tout entier de départ, pour constater que le processus s'arrête toujours.) Proposer d'abord une solution itérative, puis une fonction récursive provoquant l'impression d'un terme de la suite comme effet marginal. Il existe un prédicat Pascal "odd (n)" qui est vrai si et seulement si n est impair.

4-6° Construire récursivement en pascal la fonction h définie sur N* par :

$$h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots\ldots + \frac{1}{n}$$

4-7° Construire itérativement en pascal la fonction w définie sur N* par :

$$w(n) = \frac{1*3*\ldots*(2n+1)}{2*4*\ldots(2n)} \qquad \text{exemple : } w(3) = \frac{1*3*5*7}{2*4*6} = \frac{105}{48}$$

4-8° Soit la suite u définie par son terme de rang 0 égal à 7, et sinon u_{n+1} calculé comme la racine de $2+u_n$

a) Construire un programme donnant les termes de la suite jusqu'à ce que $|u_{n+1}-u_n| < 10^{-3}$, grâce à la boucle "repeat".

b) Même question mais en utilisant une procédure récursive à un paramètre réel.

Solution : `u := 7; repeat v := u; write (u,'/'); u := sqrt (2+u) until abs (u-v) < 0.001`

ou bien :

```
procedure suite (u : real);
        var v : real;
        begin
        write (u,'/'); v := sqrt (2+u) ;
        if abs (u-v) > 0.001 then suite (v)
        end;
```

4-9° Quelles sont dans l'ordre les affichages à l'écran que va provoquer ce programme ?

```
program truc;
        var A, B : integer;
procedure machin ; begin A := A + B end;
procedure bidule ;
        var A, B : integer;
        begin A := 2 ; B := 4; machin ; write (A) end;
begin A := 3; B := 1; machin ; write (A); bidule ; write (A) end.
```

Réponse : 4 2 5

4-10° Donner toutes les explications nécessaires, et l'effet produit par le programme suivant :

```
program khasthaite;
        var x, y : integer;
function f (x, y : integer): integer;
        var z : integer;
        begin z := x+y; f := z*z*z end;
procedure m (x, y : integer; var a : integer);
        var z : integer;
        begin z := f (x-y, x+y) ; write (z,'/'); a := f (a,1); writeln (a) end;
begin y := -2; for x := 0 to 3 do
        m (x-1, (exp (x) + atan (x+2*y)) / sqrt (sin (pi*x)+2), y) ;write (y);
end.
```

Réponse : On voit tout de suite que $f(x,y) = (x+y)^3$, et que m est une procédure qui écrit $8x^3$ puis qui modifie a en $(a+1)^3$
Le résultat à l'écran est -8/-1 puis à la ligne 0/0 puis 8/1, 64/8 et enfin 2.

4-11° Quelles sont dans l'ordre les affichages à l'écran que va provoquer ce programme ?

```
program amstramgram;
        var Z : integer;
function colegram (X : integer) : integer;
        var Z : integer;
        begin  Z := 2*(X+1) ; write (X) ; colegram := Z*(Z + 3)  end;
begin for Z := 1 to 5 do writeln (Z , colegram (Z))  end.
```

Réponses : 1 1 28 / 2 2 54 / 3 3 88 / 4 4 130 / 5 5 180 /, car colegram $(x) = 4x^2 + 14x + 10$

4-12° Si "trunc" désigne la fonction "partie entière d'un réel", si (N div P) désigne la quotient entier de N par P, et si (N mod 10) désigne le reste de la division entière de N par 10, expliquer le rôle joué par la fonction F définie par :

```
function F (N : integer) : boolean;
begin
if N < 0                then F := F(- N)
else if N = 0           then F := true
else if N < 10          then F := false
else if (N mod 10) = 0  then F := true
else F := F(N div 10)
end;
```

Réponse : vrai dès qu'un nombre entier N comporte au moins un zéro dans son développement décimal, et faux sinon. Prendre des exemples.

4-13° Calculer la somme des inverses des entiers de 1 à N en sautant ceux comportant au moins un zéro parmi leurs chiffres. (Se servir de la fonction précédente. Cette série converge très lentement vers 23,10345....)

4-14° Soit la fonction F définie par :

```
function F (N : integer) : integer;
        begin if abs (N) <= 2     then F := 3
                                  else if N > 0     then F := F(N-1) + 2*F(1-N)
                                                    else  F := 3*F(N+1) - F(-N)
        end;
```

Que vaut F(5) et combien d'appels de la fonction ont été nécessaires à son calcul ?

Réponse F(5) = -9 lui-même plus 24 autres, obtenus en développant l'arbre F(4) est calculé 2 fois, F(3) 4 fois.

4-15° Permutations d'un tableau de m entiers où m est fixé.

Solution, en supposant écrite une procédure d'échange et d'affichage, on considère une procédure de paramètres n, qui doit afficher toutes les permutations du tableau qui laissent invariants les éléments après n.

```
procedure perm (var t : tableau; n : integer); {affiche les permutations de t entre 1 et n}
            var j : integer;
begin       if n=1  then afficher (t, m)
                    else   begin perm (t, n-1);
                           for j := 1 to n-1 do    begin echange (t[n], t[j]);
                                                   perm (t, n-1);
                                                   echange (t[n], t[j])
                                                   end;
end;
```

4-16° Tri-bulle boustrophédon : le parcours d'une liste à trier se fait alternativement de gauche à droite et de droite à gauche avec la méthode du tri bulle. En remarquant que chaque balayage a pour effet de pousser les éléments extrêmes à leurs places définitives, écrire une autre version du programme de tri-bulle où l'étendue du parcours est diminuée à chaque fois. Se servir pour celà de deux procédures mutuellement récursives réalisant les balayages en avant et en arrière.

```
program boustro;
            type liste = array[1..100] of real;
procedure echange (var x,y : real);
            var t : real; {tampon}   begin t := x; x := y; y := t end;
procedure arriere (var L : tab; d,f : integer); forward;
procedure avant (var L: tab; d,f : integer);
{réalise le balayage dans la sens des indices d à f croissants}
            var drap : boolean; i : integer;
            begin drap := false;
            for i := d to f-1 do if L[i] > L[i+1] then begin echange (L[i], L[i+1]); drap := true end;
            if drap then arriere(L, f-1, d)
            end;
procedure arriere ; {réalise le balayage dans la sens des indices d à f décroissants}
            var drap : boolean; i : integer;
            begin drap := false;
            for i := d downto f+1 do if L[i] < L[i-1] then begin echange (L[i], L[i-1]); drap := true end;
            if drap then avant (L, f+1, d)
            end;
procedure tri (var L : tab; n : integer);{réalise le tri du tablau L des indices 1 à n}
            begin avant (L, 1, n) end; { On fera suivre ceci d'un programme d'essai. }
```

4-17° Disposition traditionnelle de la multiplication en Russie (et en Inde), sur deux colonnes, on double le facteur de gauche tout en divisant par deux celui de droite (quotient entier) jusqu'à obtenir 1 à droite. la somme des termes de la première colonne, qui sont placés à gauche d'un nombre impair, donne alors le résultat. On construira une fonction à trois paramètres : u, v, et r le résultat partiel.

```
function multiplic (u, v, r : integer) : integer;
begin if v = 0   then multiplic := r
            else if odd(v)   then multiplic := multiplic (2*u, v div 2, r + u)
                             else multiplic := multiplic (2*u, v div 2, r)
end;
```

```
фунштион мултиплиш (у, в, р : интегер) : интегер;
бегин иф в = 0        then мултиплиш := р
                      елсе иф одд (в)      then мултиплиш := мултиплиш(2*у, в див 2, р + у)
                                           елсе мултиплиш := мултиплиш(2*у, в див 2, р)
енд;
```

4-18° Palindromes, reconnaître si les éléments d'un tableau de caractères forment un palindrome c'est à dire un mot symétrique. Exemples (sans tenir compte des espaces, accents ou majuscules) "Esope reste et se repose", "élu par cette crapule", "Eric, notre valet, alla te laver ton ciré".

```
function palind (t : tableau; i, j : integer) : boolean;
begin if i >= j  then palind := true
                 else if t[i]=t[j]  then palind := palind (t, i+1, j-1)
                                    else palind := false
end;
```

Si le tableau est indicé de 1 à n, on demandera la valeur de palind (t, 1, n).

4-19° Dichotomie. Reprendre le problème de la recherche d'une solution d'une équation à une inconnue f(x) = 0, dont on est sûr de l'existence dans un intervalle [a, b]. On utilise l'algorithme de dichotomie consistant à couper l'intervalle en deux, et à recommencer dans celle des deux moitiés qui contient la solution.
a) Comment savoir si la racine se situe dans la première ou seconde moitié de [a, b] ?
b) Sur quel critère devrait-on stopper ce découpage ?
c) Ecrire une fonction récursive des trois variables a , b , ε (précision), renvoyant la valeur de la racine à ε près.
d) On appliquera la dichotomie pour résoudre $x^3 - x - 1 = 0$ puis, trois fois afin de réaliser la fonction :

$$\phi(p, q) = \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}$$

Quand $4p^3 + 27q^2 > 0$ c'est la seule solution de $x^3 + px + q = 0$
e) Etablir que la complexité de la dichotomie en fonction de ε et de |a-b| est donnée par un nombre d'appels approximativement $(\ln(|a-b| - \log_{10}(eps)) / \ln(2)$

4-20° Soit la procédure récursive Tros (I_0, J_0) définie par :
$S \leftarrow A[I_0]$; $T \leftarrow A[J_0]$; $I \leftarrow I_0$; $J \leftarrow J_0$
tant que $I \neq J$ faire si $S > T$ alors $A[I] \leftarrow T$; I incrémenté ; $T \leftarrow A[I]$
sinon $A[J] \leftarrow T$; J décrémenté; $T \leftarrow A[J]$
$A[I] \leftarrow S$
si $I > I_0+1$ alors tros (I_0, J - 1)
si $J < J_0-1$ alors tros (I + 1, J_0)

Pour un tableau A dont les éléments sont initialement (3 5 7 1 9 6 4), développer les appels de TROS en précisant les valeurs I_0, J_0 d'entrée, la valeur S, celles de T, et l'état de A à chaque sortie. (I, J, S et T sont des variables locales).
Essayer de nouveau pour A = (3 2 0 5 8 1 1 0 4 6). Que fait la procédure ? En quoi le terme de segmentation est-il bien adapté ?

Réponse :

I0	J0	S	valeurs successives de T	I=J final
1	7	3	4 6 9 1 5 7 5	2
3	7	7	4 5 9 6	6
3	5	4	6 5 4	3
4	5	5	6	4

A chaque appel, le I=J final détermine l'indice d'un élément de A qui est à sa place définitive, c'est à dire que tous les éléments de A de rang inférieur lui sont plus petits, sans être obligatoirement rangés entre eux, et tous les éléments de A de rang supérieur lui sont plus grands. Les deux dernières instructions consistent à appeler le tri pour chacun de ces deux segments. (Tri par segmentation.)

4-21° Tri d'un tableau t par segmentation, entre des indices g et d, on part de l'élément x situé "au milieu" d'indice p = (g+d) div 2, et on rétrécit l'encadrement [g, d] en intervertissant à l'occasion ses extrêmités de façon à déterminer l'indice p (pivot de la partition) où x sera à sa place définitive, c'est à dire que tous les éléments à gauche de x lui seront inférieurs et chaque élément à droite lui sera supérieur. Ceci ne voulant pas dire que les deux segments partagés ainsi sont triés. Ecrire une fonction "partition" calculant ce p en modifiant t, puis une procédure de tri récursive réalisant cet algorithme.

```
function part (var t : tableau; g,p,d : integer) : integer;
begin
if  t[g] < t[p]    then part  :=  part (t, g+1, p, d)
else               if t[p] < t[d]     then  part  := part (t, g, p, d-1)
                   else     begin echange (t[g], t[d]);
                            if  (g < p) and (p < d) then part :=part (t, g+1, p, d-1)
                            else     if g < p then part := part (t, g, g, d-1)
                                     else     if p < d then part := part (t, g+1, d, d)
                                              else part := p
                            end
end; {on a indenté les if avec les else pour des raisons de place }

procedure tri (var t : tableau; deb, fin : integer);
var pivot : integer;
begin              pivot := part (t, deb, (deb + fin) div 2, fin);
                   if deb < pivot - 1 then tri (t, deb, pivot - 1);
                   if pivot + 1 < fin then tri (t, pivot + 1, fin)
end;
```

4-22° Complexité du tri rapide : trouver le nombre T_n de tests à effectuer en moyenne pour trier n éléments.

Solution :
On doit d'abord considérer qu'au pire, le pivot est toujours le plus petit élément, le nombre de tests pour une table de n éléments est alors (n+1) + n + (n - 1) + ... + 3, soit de l'ordre de n^2.
Mais en moyenne, on dit que si le pivot vient occuper la place p (les places étant équiprobables), la fonction "partition" qui détermine cette place, demande n + 1 tests, et donc, on a : $T_0 = T_1 = 0$ $T_2 = 2$, $T_n = n + 1 + \sum_{0 \le p \le n} (T_p + T_{n-p}) / n$, on en déduit que :
$n(T_n - n - 1) = 2(\sum_{1 \le p \le n-1} T_p)$ en écrivant la même relation pour n - 1 on obtient une récurrence simple.

$$nT_n = (n + 1)T_{n-1} + 2n \text{ d'où } \frac{T_n}{n+1} = \frac{T_{n-1}}{n} + \frac{2}{n+1} \text{ donc :}$$

$$\frac{T_n}{n+1} = 2\left(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{n}\right) = 2\left(\frac{1}{n+1} - \frac{3}{2} + H_n\right)$$ où H_n est la somme harmonique jusqu'à n.

$T_n = 2 + 2(n + 1)H_n - 3(n + 1)$ or H_n est équivalent à l'infini à C + ln(n) où C est la constante d'Euler, donc T_n est de l'ordre de n.ln(n).

Ce résultat est à rappocher de celui du tri-bulle qui est en n^2, et de celui du tri par fusion qui est aussi en n.ln (n) mais au pire.

4-23° Les tours de Hanoï : sur trois emplacements : gauche, milieu, droite, on peut empiler n disques de diamètres différents. La règle est de toujours mettre un disque plus petit sur un disque qui lui est de taille supérieure. Grâce à une procédure élémentaire "déplacer un disque (a, b)" signifiant que le disque supérieur en a est placé sur la pile déjà en b, et à une procédure récursive "mouvement (n, a, b, c)" où n est le nombre de disques et a, b, c les noms de trois empacements, on écrira la procédure principale "hanoi (n)" qui ne sera lancée que pour n relativement petit.

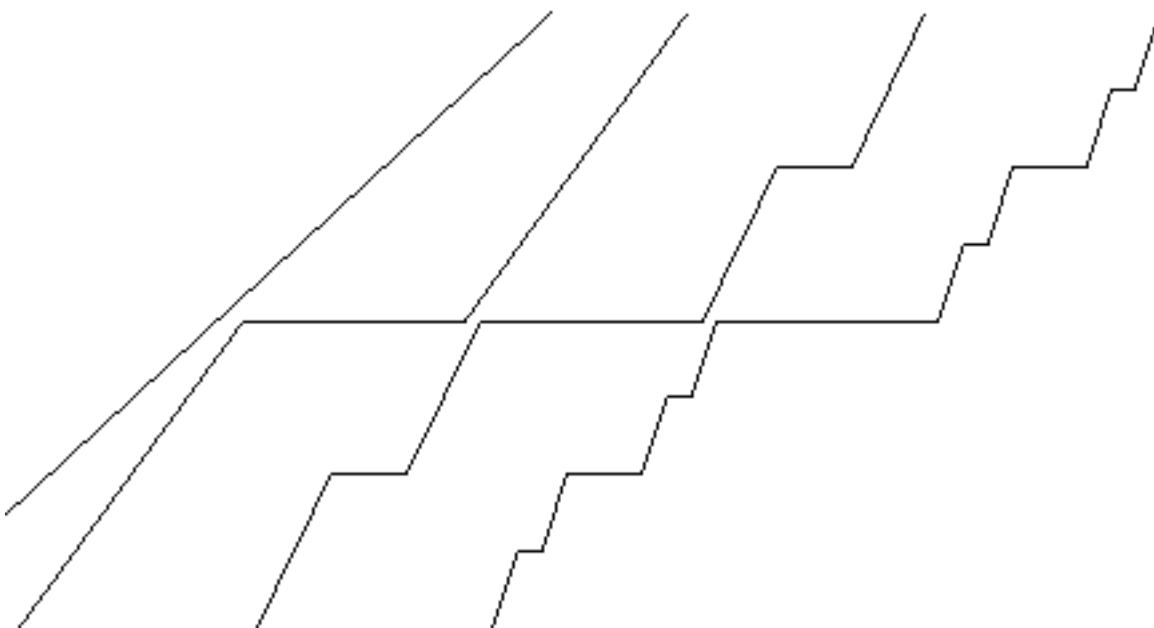
4-24° Courbes de Cantor

a) Sur l'intervalle [0, 1], en partant de la fonction $f_0(x) = x$, on coupe en 3 cet intervalle et on définit f_1 comme constante égale à 1/2 sur [1/3, 2/3] égale à f_0 en 0 et en 1, et affine par morceaux, f_2 est définie de même à partir de f_1, et ainsi de suite les paliers étant conservés. On montre que la suite des fonctions ainsi définie converge uniformément vers une fonction croissante, continue, de longueur totale 2, d'intégrale 1/2, et ayant en tout point de K (triadiques de Cantor) une dérivée à droite et une dérivée à gauche distinctes.
Faire le programme capable de tracer l'une quelconque des f_n

b) On définit cette fois f_1 en lui donnant la valeur 2/3 en 1/3 et 1/3 en 2/3, la suite converge alors vers une fonction continue non dérivable à droite ni à gauche en tout point de [0, 1]. Faire le programme traçant les représentations de f_0 à f_5.

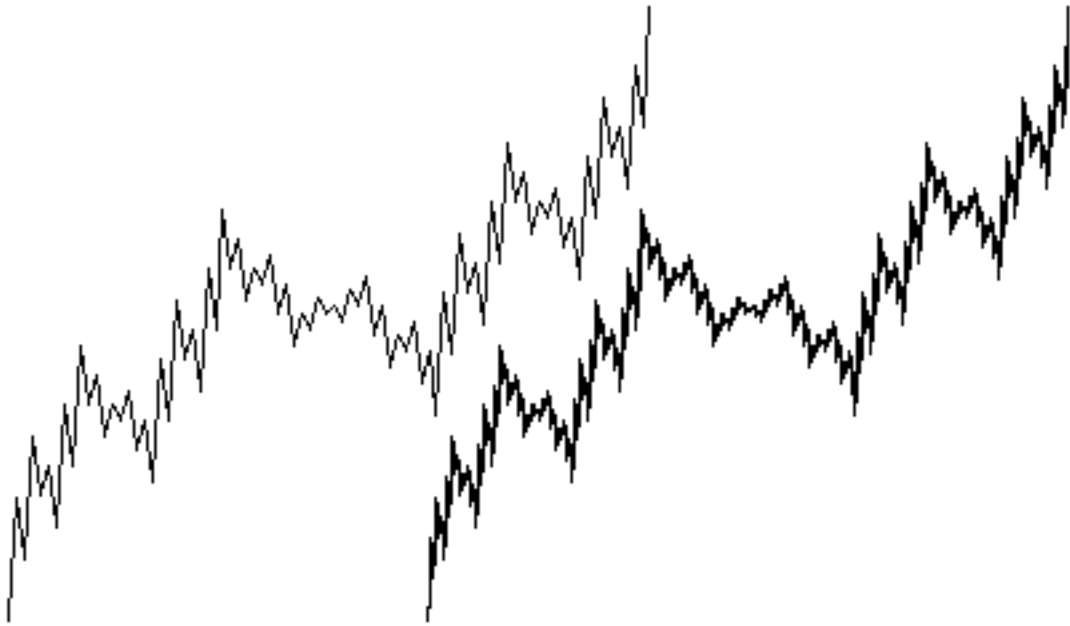
```
program fractal;       { Courbes de Cantor  turbo-pascal sur mac }
                uses memtypes, quickdraw;
const A = -2; B = 2; C = -1; D = 1.5; L = 480; H = 270;{ Pour x entre A et B, et y entre C et D les coordonnées
sur l'écran sont U=L*(X-A)/(B-A) et V=H*(Y-D)/(C-D)  }

procedure place (x, y :real);          {place le point de vraies coord x,y}
                begin moveto (round (L*(x-A)/(B-A)), round (H*(y-D)/(C-D))) end;

procedure trace (x, y :real);  {trace le segment vers le point de vraies coord x,y}
                begin lineto (round (L*(x-A)/(B-A)), round (H*(y-D)/(C-D))) end;

procedure cantor (m : integer ; a, b, c, d : real);
begin    if   m=0       then begin place (a,b); trace (c,d) end
                        else     begin
                                 cant (m -1, a, b, (2*a+c) / 3,(b+ d) / 2);
                                 trace ((a+2*c) / 3, ( b+d) / 2);
                                 cant (m -1 , (a+2*c) / 3, ( b+d) / 2, c, d)
                                 end
end;

begin {place (0, 0) peut être mis ici une seule fois} cantor (4, 0, 1, 1,1 ) end.
```

Les résultats pour m allant de 0 à 3 :

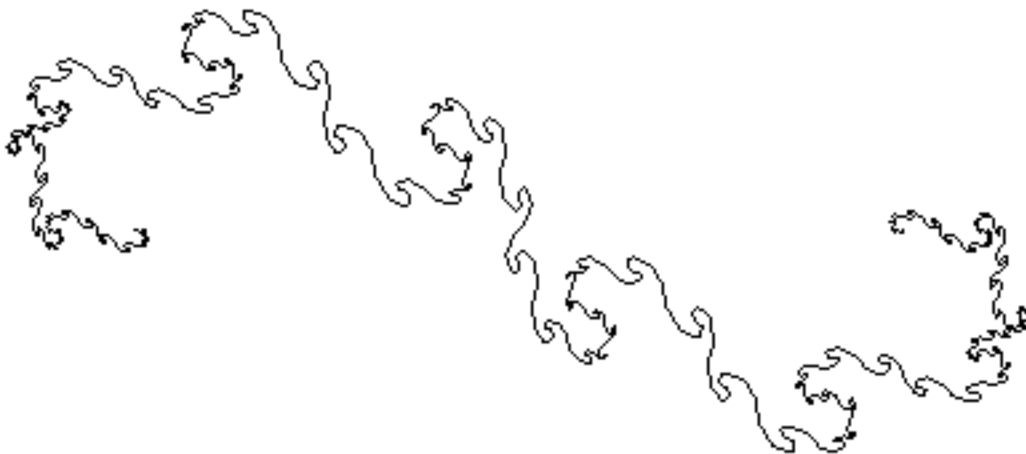


Pour l'autre courbe, il suffit de modifier un peu la procedure :

```
procedure diable (m : integer ; a, b, c, d : real);
begin   if   m = 0      then begin place (a, b); trace (c, d) end
                        else    begin
                                diable (m-1, a, b, (2*a+c)/3, (b+2*d)/3);
                                diable (m-1 , (2*a+c)/3, (b+2*d)/3 , (a+2*c)/3, (2*b+d)/3);
                                diable (m-1 , (a+2*c)/3, (2*b+d)/3, c, d)
                                end
end;
{Pour les étapes 4 et 8 uniquement : } var i : integer ;
begin  for i:=1 to 8 do diable (i, 0, 0, 1, 1 ) {si on les veut superposées} end.
```



4-25° Courbes fractales Réaliser les courbes de Von Koch, à plat puis sur un triangle équilatéral. Imaginer d'autres découpage, comme par exemple :

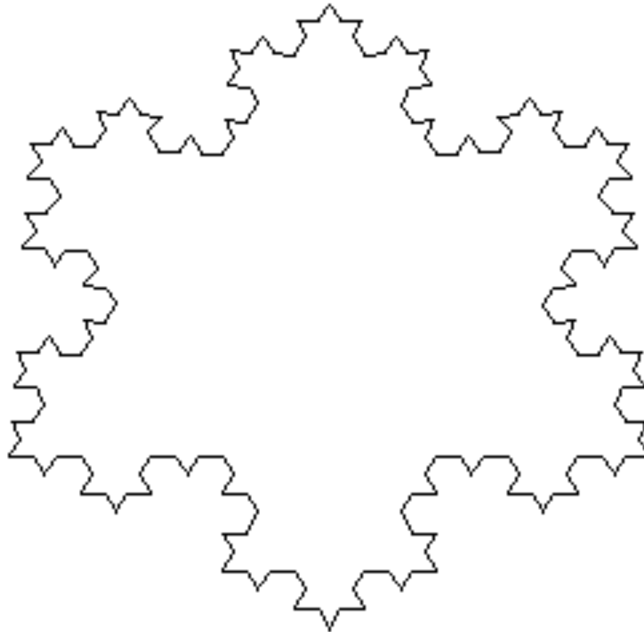


```
program fractal;      { Courbes de Von Koch  turbo-pascal sur mac }
uses memtypes, quickdraw;
const A = -2; B = 2; C = -1; D = 1.1; L = 480; H = 270;
type tab = array[0..10] of real;
var i : integer;
procedure place (x,y : real); begin moveto (L*(x-A) div (B-A),  H*(y-D) div (C-D)) end;
procedure trace (x,y : real); begin lineto (L*(x-A) div (B-A),  H*(y-D) div (C-D)) end;
```

```
procedure dragon (c, n : integer; x1, y1, x2, y2 : real; u, v : tab);
        {développe la fractale d'ordre c sur le segment}
        var i : integer ;
        begin
        if c = 0        then begin place (x1,y1); trace (x2,y2) end
                        else for i :=0 to n-1 do
                        dragon (c-1,n,x1+u[i]*(x2-x1)+v[i]*(y1-y2), y1+u[i]*(y2-y1)+v[i]*(x2-x1),
                        x1+u[i+1]*(x2-x1)+v[i+1]*(y1-y2), y1+u[i+1]*(y2-y1)+v[i+1]*(x2-x1),u,v)
        end;

procedure choix (c, k : integer);
        {c est l'ordre, k le type de dessin : triangles, cristaux, arbres, nuages, peano, petits carrés...}
        var i,n : integer; u,v : tab;
        begin u[0] := 0; v[0] := 0; case k of
        1:begin n:=4;u[1]:=1/3;u[2]:=0.5;u[3]:=2/3; v[1]:=0;v[2]:=-sqrt(3)/6;v[3]:=0 end;
        2:begin n:=4;u[1]:=0.5;u[2]:=0.5;u[3]:=0.5; v[1]:=0;v[2]:=0.3;v[3]:=0 end;
        3:begin n:=4;u[1]:=0.25;u[2]:=0.35;u[3]:=0.5; v[1]:=0;v[2]:=0.5;v[3]:=0 end;
        4:begin n:=4;u[1]:=0.1;u[2]:=0.3;u[3]:=0.5; v[1]:=0;v[2]:=0.3;v[3]:=0 end;
        5:begin n:=5;u[1]:=0.2;u[2]:=0.4;u[3]:=0.8; u[4]:=0.6;v[1]:=0;v[2]:=0.8;v[3]:=0.6;v[4]:=0.2 end;
        6:begin n:=9;u[1]:=1/3;u[2]:=1/3;u[3]:=2/3; u[4]:=2/3;u[5]:=1/3;u[6]:=1/3;u[7]:=2/3;u[8]:=2/3;
                v[1]:=0;v[2]:=1/3;v[3]:=1/3;v[4]:=0;v[5]:=0;v[6]:=-1/3;v[7]:=-1/3;v[8]:=0 end ;
        7:begin n:=5;u[1]:=1/3;u[2]:=1/3;u[3]:=2/3; u[4]:=2/3;v[1]:=0;v[2]:=1/3;v[3]:=1/3;v[4]:=0 end
end;
                u[n] := 1; v[n] := 0;
        dragon(c, n, 0, 0, 1, 1, u, v)
        end;
begin {du programme} choix (5, 2) end.
```

Pour l'étoile de Von koch, on fera :

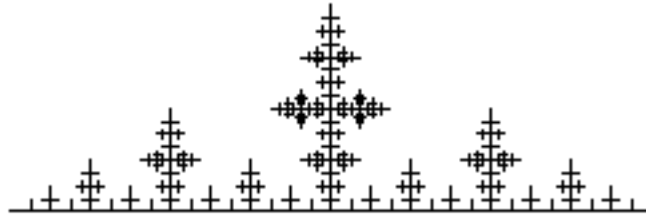
```
dragon (c, n, 0, 1, 1, -0.5, u, v) ; dragon (c, n, -1, -0.5, 0, 1, u, v) ; dragon (c, n, 1, -0.5, -1, -0.5, u, v)
```



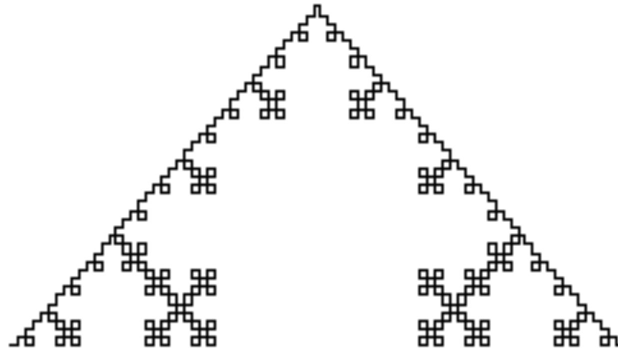
Anecdote pour l'étoile de Von Koch [Mandelbrot], si on part d'un triangle équilatéral de côté 1, donc de périmètre $p_0 = 3$ et d'aire $s_0 = \sqrt{3}/4$ pour le premier développement, on a 3 nouveaux triangles donc $3*s_0/9$ d'aire supplémentaire, soit $s_1 = 4s_0/3$.

La longueur du côté est $a_n = 1/3^n$, le nombre de côtés est $c_n = 3*4^n$, et donc, si le périmètre tend vers l'infini lorsque n tend vers l'infini, par contre, l'aire vérifie $s_n = s_{n-1} + 3s_0/4(4/9)^n$ d'où on déduit $s_n = s_0(1+3/5(1-(4/9)^n))$ dont la limite est $8s_0/5$.

Les cristaux (k = 5) :



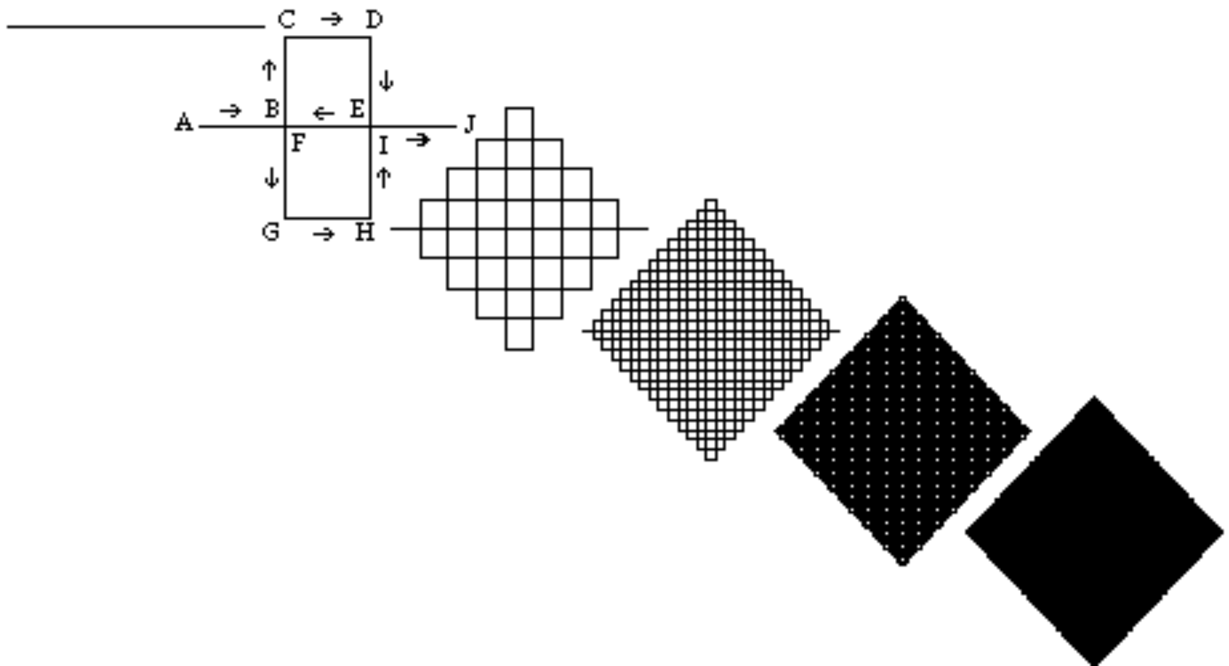
Les petits carrés avec u1 = u2 = u5 = u6 = 1/3, u3 = u4 = 2/3, v1 = v4 = 0, v2 = v3 = 1/3 :



Pour une succession d'étape, on pourra demander :

```
for i := 0 to 5 do dragon (i, n, -2+0.6*i, 1 -0.3*i, -1.2+0.6*i, 1 -0.3*i, u, v)
```

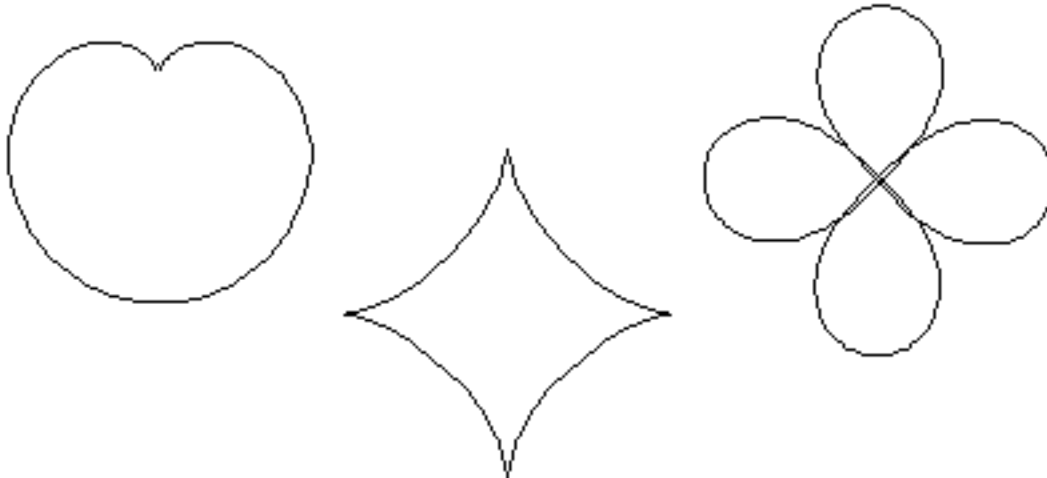
4-26° Tracer la courbe de Peano, obtenue en transformant chaque segment en une suite de 9 segments tous égaux au tiers du segment précédent (suivre les flèches du schéma ci-dessous B et I sont des points doubles). On obtient à la limite une fonction de [0, 1] dans $[0, 1]^2$ qui est continue et surjective, ce qui démontre l'équipotence de R et de R^2. D'ailleurs la dimension fractale de cette courbe est donnée par ln(n) / ln(r) = ln(9) / ln(3) = 2



Le choix k = 6 de la procedure choix ci-dessus.

4-27° Manipulation de la tortue, tracer en utilisant la tortue, la cardioïde dont l'équation intrinsèque, avec l'angle a de la tangente avec l'axe des x, a en radians, est ds = k * sin(a /3), l'astroïde d'équation intrinsèque ds = k*sin (a/2), la cycloïde ds = k sinϕ.dϕ, et enfin le trèfle d'équation ds = 1/√|cos(2*(a - pi/2)/3)|.

```
Program equintrinseques;
        uses memtypes, quickdraw , turtle; {tous les angles sont en degrés}

procedure cardioide ( A : integer);
        begin if A < 570 then {arrêt par trois demi-tours plus une petite marge}
              begin forwd (round (5*sin (a*pi/180/3))); turnleft (5); cardioide (A + 5) end
        end;

procedure astroide ( A : integer);
        begin if A < 570 then {arrêt défini de la même façon}
              begin forwd (round (8*sin ( a*pi/90 ))); turnleft (5); astroide (A + 5) end
        end;

procedure trefle ( A : integer);
        begin if A < 1200 then {arrêt défini expérimentalement}
        begin forwd (round (2/sqrt (abs (cos (2*(a*pi/180 - pi/2)/3 ))))); turnleft (5); trefle (A + 5) end
        end;

begin clear; cardioide(0); astroide (0); trefle (1) end.
```



Pour la cycloïde il suffit de remplacer ds = k sinϕ.dϕ dans le "forwd".

Essayer aussi la développée de la cardioïde :

```
procedure dev (n  : integer) ;
begin if n < 500 then
        begin forwd (round (4*sin (1/(3*n))));
        turnleft (90); forwd (round (40*sin (1/3/n)));     {on va au centre de courbure}
        back (round (40*sin (1/3/n)));                     {on en revient}
        turnright (90); turnleft (5);
        dev (n+2)
end;
```

Appel avec dev (0).

4-28° Problème du Professeur Facon : étant donné une suite de nombres entiers [e_1, e_2, ... e_n] et une somme S donnée, on souhaite réaliser, si cela est possible, la somme S avec des entiers pris dans la suite (il peut donc y avoir des répétitions dans la suite e).

On définit un tableau de booléen "choisi" tel que choisi (i) = vrai si l'entier d'indice i fait partie de la somme SP en construction.

```
function somme (SP, S : integer) : boolean;          {On suppose e, un tableau d'entiers positifs}
var i : integer; poss : boolean;
begin
if SP = S      then somme := true
               else     begin poss := false; i := 1;
                        repeat if not (choisi[i]) and (SP + e[i] <= S)
                                        then    begin choisi[i] := true;
                                                poss := somme (SP + e[i], S);
                                                if not (poss) then      begin choisi[i] := false;
                                                                        i := i+1
                                                                        end
                                        else    i := i + 1
                        until poss or (i > n);
                        somme := poss end
end;
```

Le programme consiste alors à initialiser n, S et le tableau e, puis "choisi" avec des "false", et si somme (0, S) est vrai alors afficher les entiers choisis sinon c'est impossible.
Exemple si S = 7 et e = {1, 2, 3, 4} alors pour l'appel somme (0, 7) on a choisi ={}, mais choisi (1) est mis à vrai, et l'appel somme (1, 7) met à son tour choisi (2) à vrai. L'étape suivante est l'appel de somme (3, 7) qui choisit 3 et appelle à son tour somme (6, 7) qui cette fois répond faux. Il y a alors depuis l'appel précédent qui était somme (3, 7) une "désaffectation" de choisi (3) pour choisir 4 et appeler somme (7, 7) puis remontée au premier appel qui termine.

4-29° Parcours du cavalier dans un échiquier, réaliser le parcours du cavalier dans un échiquier n*n où n et la position initiale du cavalier sont donnés. On rappelle que le cavalier se déplace en diagonale de tout rectangle de 2 sur 3.

1	4	9	18	21
10	17	20	3	8
5	2	13	22	19
16	11	24	7	14
25	6	15	12	23

Nous allons représenter le parcours par la suite numérotée des sauts du cavalier, la position 1 étant celle donnée par l'utilisateur. L'échiquier est naturellement représenté par un tableau n*n qui va donc, s'il y a une solution, être rempli par les entiers de 1 à n^2. Afin de regarder tous les sauts possibles à chaque étape, on les met en coordonnées relatives (1 ligne de moins, 2 colonnes de plus par exemple) dans un petit tableau TS contenant les 8 voisins possibles.

```
Program cavalier;             {uses crt en turbo 4 à 6 sur PC}
const m=10;
var ech : array [1..m,1..m] of integer; ts : array [1..8,1..2] of integer;
               l0, c0, n : integer;        {représentent la case de départ et la dimension n < 10}
function libre (l, c : integer) : boolean;
begin libre := (0 < l) and (l < n+1) and (0 < c) and (c < n+1) and (ech [l, c] = 0) end;
procedure aff (n : integer);    {affiche le tableau global "ech" sur n*n}
var i, j : integer;
               begin writeln;
               for  i := 1 to n  do        begin  for j := 1 to n do write (ech [i, j] : 4); writeln end end;
```

```
function cheval (i, l, c : integer) : boolean;
{la position i est déjà placée dans la case l, c lors de l'appel, la position éventuelle suivante est nommée
localement ls, cs, la fonction renvoie "vrai" s'il est possible}
                var res : boolean; k, ls, cs : integer;
begin if i = n*n then   begin aff (ech, n); cheval := true end
                else    begin k := 1; res := false;
                        repeat ls := l + ts [k, 1]; cs := c + ts [k, 2];
                                if libre (ls, cs)   then    begin ech [ls, cs] := i + 1;
                                                            res := cheval (i+1, ls, cs, n);
                                                            ech [ls, cs] := 0
                                                            end;
                                k := k + 1
                        until res or (k = 9);
                        cheval := res
                        end
end;

begin { programme } {initialisation du tableau des sauts relatifs possibles ts}
ts[1,1] := 1; ts[1,2] := 2 ts[2,1] := 2; ts[3,1] := 2; ts[3,2] := -1; ts[4,1] := 1; ts[4,2] := -2; ts[5,1] := -1;
ts[5,2] := -2; ts[6,1] := -2; ts[6,2] := -1; ts[7,1] := -2; ts[7,2] := 1; ts[8,1] := -1; ts[8,2] := 2;
write ('Quelle dimension du tableau ? '); readln (n);
write ('Ligne initiale ? '); readln (l0);
write ('Colonne initiale ? '); readln (c0);
ech [l0, c0] := 1;
if cheval (1, l0, c0) then write ('Voilà !') else write ('Pas de solution')
end.
```

Une solution avec retour pour n = 8, étudiée par Vandermonde en 1771 et Jordan en 1888

42	57	44	9	40	21	46	7
55	10	41	58	45	8	39	20
12	43	56	61	22	59	6	47
63	54	11	30	25	28	19	38
32	13	62	27	60	23	48	5
53	64	31	24	29	26	37	18
14	33	2	51	16	35	4	49
1	52	15	34	3	50	17	36

4-30° Le labyrinthe : on trace un carré avec des cases libres et d'autres occuppées par des murs ou des obstacles divers, en laissant une case libre sur le mur gauche (l'entrée) et une sur le mur droit (la sortie). On doit essayer de traverser si c'est possible, en visualisant la trace (la stratégie est d'essayer les quatre directions).

```
program laby;
const mur = '‡'; vide = ' '; entree = 'E' ; sortie = 'S';
var n {taille du tableau} : integer; ch : char; lab : array [1..25, 1..50] of char; indic : array[1..4] of char;

procedure affiche (n : integer);
                var i, j : integer;
                begin clearscreen; for i:= 1 to n do for j := 1 to 2*n do begin gotoxy(j, i); write(lab [i, j]) end end;

{La procédure de construction ci dessous n'a rien de satisfaisant, l'idéal est de créer une interface de façon à ce
que l'utilisateur puisse noircir des cases et corriger avec la souris, mais ceci entraîne de nombreux détails propres
aux systèmes utilisés }
```

```
procedure construc (var n : integer);
        var i, j : integer;
        begin write ('Quelle dimension ? '); readln (n); indic[1]:='<'; indic[2]:='v';
            indic[3]:='^'; indic[4]:='>';
        for i := 1 to n do for j:=1 to 2*n do lab[i,j]:= vide;
        for i := 1 to 2*(n div 3) do {On donne ici un exemple}
            begin lab [i, n div 3] := mur; lab [n-i, 2*(n div 3)]:= mur;
            lab [n div 4, 2+i+ n div 4] := mur;
            lab [5+ n div 2, i + n div 3] := mur; lab[i, n-i-3] := mur;
            lab [n div 2 +3, round (sqrt (i))] := mur;
            lab [i+2, n+2]:= mur; lab [n-i, 2*((2*n) div 3)] := mur; lab [i, 2*n - i+1] := mur end;
        for i := 1 to n do begin lab [i, 1] := mur; lab [i, 2*n] := mur end;
        for j := 1 to 2*n do begin lab [1, j] := mur; lab [n, j] := mur end;
        lab [n div 2, 1] := entree; lab [n div 2, 2*n] := sortie end;

function solution (x, y : integer) : boolean; {indique si étant en x, y, il est possible de gagner la sortie}
        var xs, ys, k : integer; {coordonnées de la case suivante} poss, impasse : boolean;
begin
if lab[x, y] = sortie   then    solution := true
                        else    begin poss := false; impasse := true; k := 1;
                                repeat  xs := (k mod 3) - 1 + x; ys := y + (k div 2) - 1;
                                        if lab[xs, ys] in [vide, sortie]
                                            then    begin impasse := false; lab [x, y] := indic[k];
                                                    poss := solution (xs, ys) end ;
                                        k := k+1
                                until poss or (k > 4);
                                if not (poss)   then if impasse then lab[x, y] := '*' else lab[x,y] := vide ;
                                solution := poss end
end;

begin {programme}
repeat construc (n); affiche (n);
        if solution (n div 2, 1) then affiche (n); gotoxy (1,24); write ('Recommencer ? '); ch := readchar
until ch = chr (13)
end.
```

Execution pour n = 23

```
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
▯>v   ▯          ▯>v>v*>>v>v>v**************▯▯
▯^v   ▯         ▯>^>^>v^▯v^v^v*************▯ ▯
▯^v   ▯        ▯*^<<<<v^▯v^v^v************▯>v▯
▯^v   ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯^v^▯v^v^v***********▯*^v▯
▯^v   ▯      ▯*>v>v>v^v^▯v^v^v  ********▯>v^v▯
▯^v   ▯     ▯>v^v^v^v^v^▯v^v^v* *******▯*^v^v▯
▯^v   ▯    ▯>^v^v^v^v^v^▯v^v^>v ******▯>v^v^v▯
▯^>v  ▯   ▯>^▯v^v^v^v^v^▯v^v^▯v *****▯*^v^v^v▯
▯^<v  ▯  ▯*^ ▯v^v^v^v^v^▯v^v^▯v ****▯>v^v^v^v▯
>v^v  ▯ ▯>v^ ▯v^v^v^v^v^▯v^v^▯v ***▯*^v^v^v^>S
▯v^v  ▯▯*^v^ ▯v^v^v^v^v^▯v^v^▯v **▯>v^v^v^v^ ▯
▯>^>v ▯>v^v^ ▯v^v^v^v^>^▯v^v^▯v *▯*^v^v^v^v^ ▯
▯▯▯▯v▯▯^v^v^ ▯v^v^v^v^< ▯v^v^▯v ▯>v^v^v^v^v^ ▯
▯v<<<*>^>^>^*▯>^>^>^>v^ ▯v^v^▯v>v^v^v^v^v^v^ ▯
▯v>v>v^▯▯▯▯▯▯▯▯▯▯▯▯▯▯v^*▯v^>^▯v^v^v^v^v^v^v^ ▯
▯v^v^v^      ▯v<<<<<<<^<v<^< ▯v^v^v^v^v^v^v^ ▯
▯v^v^v^      ▯v>v>v>v>v^v>v^ ▯v^v^v^v^v^v^v^ ▯
▯v^v^v^      ▯v^v^v^v^v^v^v^ ▯v^v^v^v^v^v^v^ ▯
▯v^v^v^      ▯v^v^v^v^v^v^v^ ▯v^v^v^v^v^v^v^ ▯
▯v^v^v^      ▯v^v^v^v^v^v^v^ ▯v^v^v^v^v^v^v^ ▯
▯>^>^>^      ▯>^>^>^>^>^>^>^*▯>^>^>^>^>^>^>^*▯
▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
Recommencer ?
```

4-31° Une courbe de Hilbert

```
program hilbert;
        uses memtypes, quickdraw;
        const n = 5; h0 = 256;
        var i, h, x, y, x0, y0 : integer;
procedure A (i : integer); forward;
procedure B (i : integer); forward;
procedure C (i : integer); forward;
procedure D (i : integer); forward;
procedure A;
begin if i > 0 then begin
        A(i - 1); x := x + h; y := y - h; lineto(x, y); B(i - 1); x := x + 2 * h; lineto(x, y);
        D(i - 1); x := x + h; y := y + h; lineto(x, y); A(i - 1) end end;
procedure B;
begin if i > 0 then
        begin B(i - 1); x := x - h; y := y - h; lineto(x, y); C(i - 1); y := y - 2 * h; lineto(x, y);
        A(i - 1); x := x + h;y := y - h; lineto(x, y); B(i - 1) end end;
procedure C;
begin if i > 0 then
        begin C(i - 1); x := x - h; y := y + h; lineto(x, y); D(i - 1); x := x - 2 * h; lineto(x, y);
        B(i - 1); x := x - h; y := y - h; lineto(x, y); C(i - 1) end end;
procedure D;
begin if i > 0 then
        begin  D(i - 1); x := x + h; y := y + h; lineto(x, y); A(i - 1); y := y + 2 * h; lineto(x, y);
        C(i - 1); x := x - h; y := y + h; lineto(x, y); D(i - 1) end end;
begin i := 0; h := h0 div 4; x0 := 2 * h; y0 := 3 * h;
for i := 1 to n do      begin x0 := x0 - h; h := h div 2; y0 := y0 + h; x := x0; y := y0; moveto(x, y);
                        A(i); x := x + h; y := y - h; lineto(x, y); B(i); x := x - h; y := y - h; lineto(x, y);
                        C(i); x := x - h; y := y + h; lineto(x, y); D(i); x := x + h;y := y + h; lineto(x, y) end;
end.
```

Que peut bien faire ce sacré programme ?

Réponse :