La récursivité pas à pas

Axel CHAMBILY – CASADESUS (www.chambily.com, axel@chambily.com)

Pétrut CONSTANTINE (petrut@wanadoo.fr)

Philippe ANGST (philprivatemail@free.fr)

Sommaire

- évaluer une chaîne de caractères contenant des entiers positifs et le signe "+"
- prolongement jusqu'à une calculatrice contenant les quatre signes

Chapitre II-5 : permutations des lettres d'une chaîne de caractères (anagrammes)

Echecs : Chapitre III

Chapitre III-1 : le problème du fou
Chapitre III-2 : le problème du cavalier sur un échiquier
Chapitre III-3 : le problème des huit reines sur un échiquier

Divers : Chapitre IV

Chapitre IV-1 : le triangle de Pascal
Chapitre IV-2 : tours de Hanoï
Chapitre IV-3 : le dessin de la maison sans lever le crayon
Chapitre IV-4 : trouver la sortie d'un labyrinthe
Chapitre IV-5 : le problème des chiens
Chapitre IV-6 : le solitaire
Chapitre IV-7 : le compte est bon
Chapitre IV-8 : fractales
Chapitre IV-9 : quick sort
Chapitre IV-10 : carrés magiques
Chapitre IV-11 : le problème des bouteilles d'eau
Chapitre IV-12 : les stations de métro
Chapitre IV-13 : écrire un nombre en toutes lettres

Programmation de contraintes : Chapitre V

Chapitre V-1 : le problème d'Einstein
Chapitre V-2 : les quatre musiciens

Récursivité croisée : Chapitre VI

Chapitre VI-1 : suites mathématiques An et Bn
Chapitre VI-2 : tri bulle boustrophedon

Les arbres : Chapitre VII

Chapitre VII-1 : les arbres binaires

- création
- affichage selon plusieurs parcours
- comparaison
- équilibrage

Chapitre VII-2 : morse
Chapitre VII-3 : créer l'arbre binaire d'une expression mathématique et l'évaluer
Chapitre VII-4 : les arbres n-aires

- codage d'un dictionnaire
- affichage d'un arbre n-aire
- recherche d'un mot dans l'arbre

Dérécursification : Chapitre VIII

Chapitre VIII-1 : dérécursifier la procédure de Fibonacci
Chapitre VIII-2 : dérécursifier un arbre binaire (dictionnaire)
Chapitre VIII-3 : dérécursifier le triangle de pascal
Chapitre VIII-4 : dérécursifier les tours de Hanoï
Chapitre VIII-5 : dérécursifier le problème des "nombres en lettres"

Les jeux de réflexion : Chapitre IX

Chapitre IX-1 : le morpion à trois pions
Chapitre IX-2 : le jeu d'othello

Introduction

"Une procédure récursive est une procédure récursive."

La récursivité est un domaine très intéressant de l'informatique, un peu abstrait, mais très élégant; elle permet de résoudre certains problèmes d'une manière très rapide, alors que si on devait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

La récursivité utilise toujours la pile du programme en cours.

On appelle "pile" une zone mémoire réservée à chaque programme; sa taille peut être fixée manuellement par l'utilisateur. Son rôle est de stocker les variables locales et les paramètres d'une procédure. Supposons que nous sommes dans une procédure "proc1" dans laquelle nous avons des variables locales. Ensuite nous faisons appel à une procédure "proc2"; comme le microprocesseur va commencer à exécuter "proc2" mais qu'ensuite il reviendra continuer l'exécution de "proc1", il faut bien stocker quelque part les variables de la procédure en cours "proc1"; c'est le rôle de la pile. Tout ceci est géré de façon transparente pour l'utilisateur. Dans une procédure récursive, toutes les variables locales sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement, et si on ne fait pas attention on arrive à un "débordement de pile". Ensuite, les variables sont désempilées (et on dit "désempilées", pas "dépilées").

Dans ce qui suit, nous allons utiliser le terme "procédure" pour désigner aussi bien une procédure qu'une fonction.

Une procédure récursive comporte un appel à elle-même, alors qu'une procédure non récursive ne comporte que des appels à d'autres procédures.

Toute procédure récursive comporte une instruction (ou un bloc d'instructions) nommée "point terminal" ou "point d'appui" ou "point d'arrêt", qui indique que le reste des instructions ne doit plus être exécuté.

Exemple :

```
procedure récursive(paramètres);
déclaration des variables locales
begin
if TEST_D'ARRET then
   begin
   instructions du point d'arrêt
   end
else  //----- exécution
   begin
   instructions;
   récursive(paramètres changés);  // appel récursif
   instructions;
   end;
end;
```

On constate, et il le faut, que les paramètres de l'appel récursif changent; en effet, à chaque appel, l'ordinateur stocke dans la pile les variables locales; le fait de ne rien changer dans les paramètres ferait que l'ordinateur effectuerait un appel infini à cette procédure, ce qui se

traduirait en réalité par un débordement de pile, et d'arrêt de l'exécution de la procédure en cours. Grâce à ces changements, tôt ou tard l'ordinateur rencontrera un ensemble de paramètres vérifiant le test d'arrêt, et donc à ce moment la procédure récursive aura atteint le "fond" (point terminal). Ensuite les paramètres ainsi que les variables locales sont désempilées au fur et à mesure qu'on remonte les niveaux.

Nous allons étudier en détail maintenant le mécanisme de la récursivité sur un exemple concret, la combinaison des lettres d'une chaîne de caractères (ou anagrammes). La réalisation de la procédure récursive faisant cela sera étudiée plus loin, nous allons décrire ici le mécanisme interne et le stockage des paramètres et des variables locales.

```
procedure combinaison2(st,tete : string);
var i : integer;
begin
if length(st)=1 then memo1.lines.add(tete,st)
   else
   for i:=1 to length(st) do
       begin
       combinaison1(copy(st,2,length(st)-1),tete+st[1]);
       st:=copy(st,2,length(st)-1)+st[1];
       end;
end;
```

voici l'appel de cette procédure : combinaison('abc','');

paramètres : st='abc' ; tete='' ;
i:=1 ;
combinaison('bc','a') ;

> paramètres : st='bc' ; tete='a'
> i:=1 ;
> combinaison('c','ab') ;
>
> > point d'arrêt :
> > on affiche 'ab'+'c' soit 'abc'
>
> st:='cb' ;
> i:=2 ;
> combinaison('b','ac') ;
>
> > point d'arrêt :
> > on affiche 'ac'+'b' soit 'acb'
>
> st:='bc' ; <— cette valeur n'a plus d'importance,elle sera perdue

st:='bca' ;
i:=2 ;
combinaison('ca','b') ;

> paramètres : st='ca' ; tete='b'
> i:=1 ;
> combinaison('a','bc') ;
>
> > point d'arrêt :
> > on affiche 'bc'+'a' soit 'bca'
>
> st:='ac' ;
> i:=2 ;
> combinaison('c','ba') ;
>
> > point d'arrêt :
> > on affiche 'ba'+'c' soit 'bac'
>
> st:='ca' ; <— cette valeur n'a plus d'importance,elle sera perdue

st:='ca'+'b'='cab' ;
i:=3 ;
combinaison('ab','c') ;

> paramètres : st='ab' ; tete='c'
> i:=1 ;
> combinaison('b','ca') ;
>
> > point d'arrêt :
> > on affiche 'ca'+'b' soit 'cab'
>
> st:='ba' ;
> i:=2 ;
> combinaison('a','cb') ;
>
> > point d'arrêt :
> > on affiche 'cb'+'a' soit 'cba'
>
> st:='ab' ; <— cette valeur n'a plus d'importance,elle sera perdue

st:='ab'+'c'='abc' ; <— cette valeur n'a plus d'importance,elle sera perdue

Chapitre I – Les boucles

Chapitre I-1

Transformer une boucle en une procédure récursive

Soit la procédure suivante :

```
procedure compter;
var i : integer;
begin
for i:=1 to 10 do
    memo1.lines.add(inttostr(i));
end;
```

Cette procédure peut être traduite en une procédure récursive, qui admet un paramètre; l'instruction qui l'appellera sera "compter(1);":

```
procedure compter(i : integer );
begin
if i<11 then
   begin
   memo1.lines.add(inttostr(i));
   compter(i+1);
   end;
end;
```

Transformer deux boucles imbriquées en une procédure récursive

Supposons qu'on ait maintenant deux boucles imbriquées. Nous allons traduire progressivement cette procédure itérative en une procédure récursive avec deux paramètres :

```
procedure affiche;
var a,b : integer;
begin
for a:=0 to 3 do
for b:=0 to 9 do
    memo1.lines.add(inttostr(a*10+b));
end;
```

Pour supprimer les deux boucles, on commence par supprimer la première en suivant l'exemple ci-dessus;on obtient la procédure suivante que l'on appelle avec "affiche(0);" :

```
procedure affiche(a : integer);
var b : integer;
begin
if a<4 then
   begin
   for b:=0 to 9 do
       memo1.lines.add(inttostr(a*10+b));
   affiche(a+1);
   end;
end;
```

Il ne nous reste plus qu'à supprimer la deuxième boucle; en sachant que lorsque "b=10" dans la procédure initiale, le programme revient à la boucle sur "a" et remet "b" à zéro, alors on a 2 appels récursifs :

- le premier dans le cas où "b" est inférieur à 10 et alors on appelle la procédure avec "a" inchangé et "b" incrémenté de 1

- le deuxième où "b=10" et alors on appelle la procédure avec "a" incrémenté de 1 et "b" initialisé à 0.

L'appel sera "affiche(0,0);".

```
procedure affiche(a,b : integer);
begin
if a<4 then
   if b<10 then
      begin
      memo1.lines.add(inttostr(a*10+b));
      affiche(a,b+1);
      end
   else
      affiche(a+1,0);
end;
```

Calculer la factorielle d'un entier

L'exemple ci-dessous est utilisé pour calculer la factorielle d'un nombre.

La factorielle d'un nombre "n" se note "n!" et représente le nombre de permutations de "n" objets.

On rappelle que la factorielle d'un nombre "n" est égale au produit de tous les nombres de 1 jusqu'à "n". Ainsi, "5!"="1*2*3*4*5"

La fonction itérative est donnée ci-dessus :

```
function fact(n : integer) : integer;
var i,j : integer;
begin
j:=1;
for i:=1 to n do
    j:=j*i;
fact:=j;
end;
```

En reprenant l'exemple de la transformation d'une boucle simple en une procédure récursive, on obtient pour "factorielle" la fonction récursive suivante :

```
function fact(n : integer) : integer;
begin
if n=1 then fact:=1
   else fact:=n*fact(n-1)
end;
```

On constate, et c'est vrai en général, que les procédures récursives sont plus courtes que celles itératives.

Chapitre I-2

Remplir un carré en diagonale

Supposons qu'on a un carré de 10x10 cases, et qu'on veut noircir toutes ses cases; le remplissage ne s'effectuera pas en ligne ou en colonne mais en diagonale, de gauche à droite et du bas vers le haut.

Pour cela, si à un instant t on se trouve à la case de coordonnées (x,y), alors la prochaine case à noircir sera la case de coordonnées (x+1,y-1).

Il faut prendre soin de tester les effets de bord (ne pas sortir du carré à force de décrémenter x ou d'incrémenter y).

On en déduit donc que le point d'arrêt de la procédure est atteint quand les coordonnées x ou y se retrouvent en dehors du carré.

On sait ensuite que lorsqu'on aura atteint la première ligne (y=1), alors il faut redémarrer le remplissage à la première colonne.

```
procedure trace(x,y,x_droite,y_bas,total_posees : integer);
begin
tab[x,y]:='$';
afficher_tableau;
if (x>=dim) and (y>=dim) then exit;  // point d'arrêt
if y=1 then
   begin
   if x>=dim then inc(x_droite);  // effet de bord
   x:=x_droite;
   y:=y_bas+1;
   if y>dim then dec(y);  // effet de bord
   y_bas:=y;
   if total_posees<dim*dim then  // carré rempli ?
      trace(x,y,x_droite,y_bas,total_posees+1);
   end
else
if x>=dim then
   begin
   if y_bas=dim then inc(x_droite);  // effet de bord
   x:=x_droite;y:=y_bas;
   if total_posees<dim*dim then  // carré rempli ?
      trace(x,y,x_droite,y_bas,total_posees+1);
   end
else
   if total_posees<dim*dim then  // carré rempli ?
      trace(x+1,y-1,x_droite,y_bas,total_posees+1);
end;
```

On appelle cette procédure avec l'instruction suivante: "trace(1,1,1,1,1);".

Chapitre II – Chaînes de caractères

Chapitre II-1

L'inversion d'une chaîne de caractères

Pour continuer à expliquer la récursivité dans ce livre, nous allons commencer par un exemple très simple:

- l'inversion d'une chaîne de caractères.

Soit une chaîne de caractères; supposons qu'on veuille faire aussi bien la fonction que la procédure qui nous renvoient l'inverse de cette chaîne; la fonction admet donc un paramètre qui est la chaîne initiale (transmise par valeur), et la procédure un paramètre qui est une variable de type "string" (transmis par adresse, c'est à dire en utilisant le mot "var" devant, qui indique que toutes les modifications du paramètre à un niveau quelconque de la procédure récursive se répercuteront sur le paramètre initial).

De manière itérative, cette fonction et cette procédure se font très facilement :on parcourt la chaîne d'un bout à l'autre et on accumule les caractères un par un dans une chaîne auxiliaire en prenant soin de mettre chaque nouveau caractère en tête de la chaîne auxiliaire. Voici cette première solution :

```
function inverse(st : string) : string;
var aux : string;
    i   : integer;
begin
aux:='';
for i:=1 to length(st) do
    aux:=st[i]+aux;
inverse:=aux;
end;

procedure inverse(var st : string);
var aux : string;
    i   : integer;
begin
aux:='';
for i:=1 to length(st) do
    aux:=st[i]+aux;
st:=aux;
end;
```

Voici une autre solution, qui consiste à lire les caractères de la chaîne passée en paramètre en sens inverse et de les accumuler dans une variable auxiliaire:

```
function inverse(st : string) : string;
var aux : string;
    i   : integer;
begin
aux:='';
for i:=length(st) downto 1 do
    aux:=aux+st[i];
inverse:=aux;
end;
```

```
procedure inverse(var st : string);
var aux : string;
    i   : integer;
begin
aux:='';
for i:=length(st) downto 1 do
    aux:=aux+st[i];
st:=aux;
end;
```

Transformons le principe de cette fonction et de cette procédure, de façon à les transformer en une fonction récursive et une procédure récursive; en appliquant le principe décrit à la fonction et la procédure ci-dessus, on voit qu'il faut toujours mettre en tête le dernier caractère; on en déduit donc l'algorithme:

- soit une chaîne 'abc'

- pour obtenir son inverse on met le dernier caractère en tête:

"c" et on le colle à l'inverse du reste qui est "ab"

- pour obtenir l'inverse de "ab" on met le dernier caractère en tête:

"b" et on le colle à l'inverse du reste qui est "a"

- pour obtenir l'inverse de "a" on met le dernier caractère en tête:

"a" et on le colle à l'inverse du reste qui est ""

- on a donc "cba"

```
function inverse(st : string) : string;
var dernier_caractere : char;
    reste             : string;
    inverse_du_reste  : string;
begin
if st='' then
   inverse:=''     {--- ceci est le point d'appui ---}
else
   begin
   dernier_caractere:=st[length(st)];
   reste:=copy(st,1,length(st)-1);
   inverse_du_reste:=inverse(reste);
   inverse:=dernier_caractere+inverse_du_reste;
   end;
end;

procedure inverse(var st : string);
var dernier_caractere : char;
    reste             : string;
begin
if st='' then st:='' //--- ceci est le point d'appui, inutile ici
else                 //--- on aurait pu traduire cela par "if st<>'' then"
   begin
   dernier_caractere:=st[length(st)];
   reste:=copy(st,1,length(st)-1);
   inverse(reste);  //--- on inverse le reste
   st:=dernier_caractere+reste;
   end;
end;
```

Cette fonction et cette procédure sont convertibles en une fonction et une procédure sans variables internes, qui leur sont identiques du point de vue du résultat:

```
function inverse(st : string) : string;
begin
if st='' then inverse:=''
   else inverse:=st[length(st)]+inverse(copy(st,1,length(st)-1));
```

```
end;

procedure inverse(var st : string);
var c : char;
begin
if st<>'' then
   begin
   c:=st[length(st)];
   delete(st,length(st),1);
   inverse(st);
   st:=c+st;
   end;
end;
```

Pour obtenir l'inverse d'une chaîne avec cette procédure on fait tout simplement l'appel "inverse(s);", mais à condition que la chaîne soit déclarée comme une variable de type string, à cause du mot "var"; donc l'appel à cette procédure avec "inverse('abc');" ne fonctionnera pas.

Chapitre II-2

Evaluation d'un nombre écrit en chiffres romains

Nous allons réaliser maintenant un programme permettant d'évaluer un nombre romain, mais auparavant on va introduire les chiffres romains:

M = 1000

D = 500

C = 100

L = 50

X = 10

V = 5

I = 1

On constate que les nombres s'arrêtaient au milliers; cela montre le progrès des mathématiques depuis le temps des romains.

L'écriture des nombres romains

1 s'écrit I.

2 s'écrit II.

3 s'écrit III.

4 s'écrit IV.

5 s'écrit V.

6 s'écrit VI.

7 s'écrit VII.

8 s'écrit VIII.

9 s'écrit IX.

10 s'écrit X.

Tous les nombres finissant par 1,2 ou 3 se terminent dans l'écriture romaine par I,II ou III. Idem pour 6,7 ou 8.

Ceci est valable aussi bien pour les unités, comme on vient de le voir que pour les dizaines et les centaines; ainsi 30 s'écrit XXX, 300 s'écrit CCC.

Tous les nombres finissant par 4, se terminent dans l'écriture romaine par IV.

Tous les nombres finissant par 9, se terminent dans l'écriture romaine par IX. Ainsi 49 se termine par IX, et il s'écrit XLIX.

Identiquement, les nombres finissant par 90 se terminent dans l'écriture domaine par XC.

15 s'écrit XV.

47 s'écrit XLVII.

97 s'écrit XCVII.

14 s'écrit XIV.

149 s'écrit CXLIX (et non CIL, comme on pourrait le penser)

On constate ici la décomposition 100+40+9 = C + XL + IX

1490 s'écrit MCDXC = 1000 + 400 + 90 = M + CD + XC

1900 s'écrit MCM = 1000+900 = M + CM.

1990 s'écrit MCMXC (et non MXM, comme on pourrait le penser)

1000+900+90 = M + CM + XC

1999 = MCMXCIX

Observations

Soit deux chiffres romains. Si le premier chiffre a une valeur inférieure au deuxième, alors on le soustrait de la valeur de tout le reste, sinon on l'additionne à la valeur de tout le reste.

```
En effet, étudions cela sur un exemple :
MCMXCIX.
|On est sur le premier M.
|Son successeur est C, il est plus petit, donc notre résultat final sera la
|valeur de M (1000) plus la valeur du reste (CMXCIX).
|  La valeur du reste est la suivante :
|  C est plus petit que M (une soustraction aura lieu) donc la valeur de
|  CMXCIX est égale à la valeur de MXCIX moins la valeur de C
|  |   La valeur de MXCIX est la suivante :
|  |   |M est plus grande que X donc on a 1000+valeur(XCIX).
|  |   |   La valeur de XCIX est égale à la valeur de CIX moins la valeur de X
|  |   |   |car le premier X est plus petit que son successeur.
|  |   |   |   La valeur de CIX est égale à 100 + valeur(IX) car C est plus
|  |   |   |   |grand que I.
|  |   |   |   |    La valeur de IX est égale à la valeur de X moins la valeur
|  |   |   |   |    de I, soit 10-1 = 9.
|  |   |   |   CIX = C + 9 = 109
|  |   |   XCIX = CIX - X = 109 - 10 = 99
|  |   MXCIX = M + XCIX = 1000 + 99 = 1099
|  CMXCIX = MXCIX - C = 1099 - 100 = 999
MCMXCIX = 1000 + 999 = 1999
```

On voit la forme de l'algorithme général:

Soit un nombre romain. Si le premier chiffre de ce nombre a une valeur inférieure au deuxième, alors on le soustrait de la valeur de tout le reste, sinon on l'additionne à la valeur de tout le reste.

Si le nombre romain a un seul chiffre, alors on prend simplement la correspondance (M=1000, D = 500,...).

Appelons "Eval" la fonction d'évaluation d'un nombre romain.

Son ***point d'appui*** est "Eval(r)= valeur(r)" si r est un caractère romain.

Le ***cas général*** est le suivant :

- si la valeur du premier chiffre romain est plus grande que la valeur du

  deuxième, alors "Eval(r)=valeur(premier chiffre)+Eval(reste)"

- si la valeur du premier chiffre romain est plus petite que la valeur du

  deuxième, alors "Eval(r)=Eval(reste)-valeur(premier chiffre)"

On constate qu'avec l'algorithme ci-dessus, le nombre MIM (qui pourtant n'existe pas chez les romains) est évalué à 1999. Le programme décrit ci-dessus n'effectue pas un test d'exactitude de la chaîne passée en paramètre pour être évaluée, il se contente seulement de l'évaluer.

```
function valeur(c : char) : integer;
begin
case c of
   'M' : valeur:=1000;
   'D' : valeur:= 500;
   'C' : valeur:= 100;
   'L' : valeur:=  50;
   'X' : valeur:=  10;
   'V' : valeur:=   5;
   'I' : valeur:=   1;
   end;
end;

function eval(s : string) : integer;
var n1 : integer;
begin
if length(s)=1 then
   eval:=valeur(s[1])
else
   begin
   n1:=valeur(s[1]);
   if n1<valeur(s[2]) then n1:=-n1;
   eval:=n1+eval(copy(s,2,length(s)-1));
   end;
end;
```

Chapitre II-3

Palindromes

On appelle "palindrome" un mot ou une phrase qui se lit de la même façon à l'endroit comme à l'envers, sans tenir compte des espaces.

exemple : le mot "ABCBA" est un palindrome.

La phrase "ESOPE RESTE ET SE REPOSE" (sans tenir compte des espaces on obtient le mot "ESOPERESTEETSEREPOSE") se lit de façon identique de la gauche vers la droite ou de la droite vers la gauche.

L'algorithme qui teste si un mot est un palindrome est extrêmement simple:

- un mot de longueur 0 (le mot vide, ou en programmation, la variable string '') est un palindrome

- un mot de longueur 1 (donc une lettre) est un palindrome

- un mot est un palindrome si sa première lettre est identique à sa dernière et si son intérieur (le sous-mot commençant à la deuxième position et finissant à l'avant dernière lettre: dans le cas de "ABCBA" le sous-mot est "BCB") est un palindrome.

On voit donc à cette dernière ligne l'apparition de la récursivité : "un mot est un palindrome si... et si son sous-mot est un palindrome".

Voici le programme qui teste si un mot est un palindrome:

```
function palindrome(s : string) : boolean;
begin
if length(s)<2 then
   palindrome:=true
else
   if (s[1]=s[length(s)]) then
      palindrome:=palindrome(copy(s,2,length(s)-2))
   else
      palindrome:=false;
end;
```

Chapitre II-4

Evaluation d'une chaîne de caractères formée d'une somme de nombres

Soit une chaîne de caractères du type "s:='5+3+4+7';";faisons le programme qui évalue cette chaîne de caractères.

Voyons l'algorithme de cette procédure :

- d'abord on sait que la valeur d'une chaîne vide est 0

- ensuite on sait évaluer un nombre seul avec la fonction "strToInt".

- on sait extraire une sous-chaîne avec la fonction "copy" et nous allons extraire des caractères jusqu'au premier signe "+"

- l'évaluation de la chaîne initiale est égale à la valeur du nombre extrait plus la valeur de la chaîne restante (appel récursif).

Voici le programme correspondant :

```
function eval(st : string) : integer;
var a,i : integer;
begin
if length(st)=0 then
   eval:=0
else
   begin
   i:=1;
   repeat inc(i) until (i>length(st)) or (st[i]='+');
   a:=strToInt(copy(st,1,i-1));
   delete(st,1,i-1);
   eval:=a+eval(st)
   end;
end;
```

On appelle cette fonction de la façon suivante :

b:=eval('+51+7+67+31');

Vers une mini-calculatrice

Soit une chaîne de caractères du type "s:='5+3*4/2-5*3+4*7/2';";faisons le programme qui évalue cette chaîne de caractères.

La première chose à faire est la séparation des termes; on doit en effet extraire tous les termes qui sont séparés par des "+" et des "-".

Ensuite nous devons évaluer chacun des termes extraits, pour arriver à la fin à une somme (ou à une différence) de termes.

Pour commencer, on suppose qu'on doit évaluer une somme (ou une différence) de termes. Pour cela, nous reprenons le programme

fait précédemment, et nous l'enrichissons, de façon à ce qu'il puisse évaluer aussi les différences.

```
function eval(st : string) : integer;
var a,i : integer;
begin
if length(st)=0 then
   eval:=0
else
   begin
   i:=1;
   repeat inc(i) until (i>length(st)) or (st[i] in ['+','-']);
   a:=strToInt(copy(st,1,i-1));
   delete(st,1,i-1);
   eval:=a+eval(st)
   end;
end;
```

Maintenant, on doit créer la procédure qui évalue une expression formée uniquement de produits et quotients; ici, on ne peut pas appliquer la même méthode : pour les sommes, on évaluait le premier terme auquel on ajoutait le résultat du reste.

Mais avec les produits et les quotients, c'est différent : en effet, si on veut évaluer le résultat de "12/4*6/2" on ne peut pas dire qu'il est égal à la valeur du premier terme "12" divisé par la valeur du reste "4*6/2". Le résultat est en réalité "((12/4)*6)/2" soit 9.

Il faut donc lire non pas de gauche à droite, mais de droite à gauche.

Ainsi le résultat de "12/4*6/2" est égal au résultat de "12/4*6" divisé par 2.

Le résultat de "12/4*6" est égal au résultat de "12/4" multiplié par 6

Le résultat de "12/4" est égal au résultat de "12" divisé par 4.

On peut évaluer cette expression, soit 3.

On a, en remontant dans les appels récursifs, 3*6=18.

On a, en remontant encore une fois dans les appels récursifs, 18/2=9.

L'algorithme est le suivant :

- on lit une expression de la droite vers la gauche.

- si on est sur un signe "*" alors :

  - on extrait le dernier nombre de la chaîne

  - on évalue la chaîne restante

  - le résultat est égal à l'évaluation de la chaîne restante (***appel récursif***) multipliée par le dernier nombre

- si on est sur un signe "/" alors :

  - on extrait le dernier nombre de la chaîne

  - on évalue la chaîne restante

- le résultat est égal à l'évaluation de la chaîne restante (***appel récursif***) divisée par le dernier nombre

- si on est au début de la chaîne (aucune multiplication, ni division) alors :

- le résultat est égal à l'évaluation du nombre.

Voici le programme qui évalue une expression mathématique contenant des nombres entiers et les 4 opérations mathématiques. Ce programme fonctionne avec des nombres entiers. Ainsi, quand vous faites une division, si elle ne tombe pas juste, le résultat est arrondi, grâce à la fonction "div". Mais la transformation en réels est très facile.

```
function eval0(st : string) : integer;
var a,i : integer;
    signe : char;
begin
i:=length(st);
repeat dec(i) until (i=0) or (st[i] in ['*','/']);
a:=strToInt(copy(st,i+1,length(st)-i));
if i>0 then signe:=st[i] else signe:=#0;
delete(st,i,length(st)-1+i);
if signe='*' then eval0:=eval0(st)*a else
if signe='/' then eval0:=eval0(st) div a else
   eval0:=a;
end;

function eval(st : string) : integer;
var a,i : integer;
begin
if length(st)=0 then
   eval:=0
else
   begin
   i:=1;
   repeat inc(i) until (i>length(st)) or (st[i] in ['+','-']);
   a:=eval0(copy(st,1,i-1));
   delete(st,1,i-1);
   eval:=a+eval(st)
   end;
end;
```

Chapitre II-5

Anagrammes

Supposons qu'on veuille faire un programme qui affiche les combinaisons d'une chaîne de caractères; par exemple les anagrammes des lettres de "abc" sont "abc, acb, bac, bca, cab, cba".

Pour réaliser ce programme nous allons commencer par des petites chaînes, et nous allons d'abord le faire de manière itérative.

Pour "a", toutes les combinaisons sont : "a".

Pour "ab", toutes les combinaisons sont : "ab,ba".

On remarque que ces combinaisons ne sont en fait qu'une rotation de la chaîne initiale vers la gauche, et ceci 2 fois de suite.

On peut donc en déduire l'algorithme général, qui ne sera constitué que de rotations successives vers la gauche.

```
Pour "abc", toutes les combinaisons sont :
   première rotation  : "bca" suivi d'une rotation de "ca" donc "bac"
                              suivi de 2 rotations de "ca" donc "bca"
   deuxième rotation  : "cab" suivi d'une rotation de "ab" donc "cba"
                              suivi de 2 rotations de "ca" donc "cab"
   troisième rotation : "abc" suivi d'une rotation de "bc" donc "acb"
                              suivi de 2 rotations de "bc" donc "abc"
```

Nous allons faire la procédure qui affiche les combinaisons des lettres d'une chaîne de 3 caractères.

```
procedure combinaison(st : string);
var i1,i2,i3 : integer;
    tete1,tete2,reste1,reste2 : string;
begin
for i1:=1 to 3 do
    begin
    tete1 :=st[1];
    reste1:=copy(st,2,length(st)-1);
    st:=reste1;
    for i2:=1 to 2 do
        begin
        tete2 :=st[1];
        reste2:=copy(st,2,length(st)-1);
        st:=reste2;
        for i3:=1 to 1 do
            memo1.lines.add(tete1+tete2+st);  { on affiche les têtes successives }
        st:=reste2+tete2;             { ici on fait la rotation }
        end;
    st:=reste1+tete1;                 { ici on fait la rotation }
    end;
end;
```

Nous allons transformer maintenant cette procédure itérative en une procédure récursive. On obtient la procédure suivante, qui est appelée avec l'instruction "combinaison('abc','');" :

```
procedure combinaison1(st,tete : string);
var i : integer;
    tete_local,reste_local : string;
begin
if length(st)=1 then memo1.lines.add(tete+st)
   else
   for i:=1 to length(st) do
       begin
```

```
        tete_local :=st[1];
        reste_local:=copy(st,2,length(st)-1);
        combinaison1(reste_local,tete+tete_local);
        st:=reste_local+tete_local;
        end;
end;
```

Maintenant on peut supprimer les variables locales qui sont "tete_local" et "reste_local" et on obtient:

```
procedure combinaison2(st,tete : string);
var i : integer;
begin
if length(st)=1 then memo1.lines.add(tete+st)
   else
   for i:=1 to length(st) do
       begin
       combinaison1(copy(st,2,length(st)-1),tete+st[1]);
       st:=copy(st,2,length(st)-1)+st[1];
       end;
end;
```

Une autre solution à ce problème, et qui appartient à mon ancien professeur d'informatique de lycée, est la suivante :

on prend un mot et on permute sa première lettre avec chacune des autres lettres. Ensuite pour chacun des nouveaux mots obtenus, on ne touche pas à la première lettre et à partir de la deuxième lettre on refait les permutations entre la deuxième lettre et toutes les autres lettres jusqu'à la fin; ensuite pour chacun des nouveaux mots obtenus, on ne touche pas à la deuxième lettre et à partir de la troisième lettre on refait les permutations entre la troisième lettre et toutes les autres lettres jusqu'à la fin; on voit donc la récursivité:

```
procedure combinaisons;
var  ch : string;
     l  : byte;

     procedure EchangeCar (var ch : string; i, j : byte);
     var car : char;
     begin
     if i <> j then
        begin
        car  :=ch[i];
        ch[i]:=ch[j];
        ch[j]:=car;
        end
     end;

     procedure anagram (ch : string; i : byte);
     var j : byte;
     begin
     if i = l then memo1.lines.add(ch)
        else
        for j := i to l do
            begin
            EchangeCar (ch,i,j);
            Anagram (ch,i+1);
            EchangeCar (ch,i,j);
            end;
     end;

begin
ch:='abc';l:=length(ch);
anagram(ch,1);
end;
```

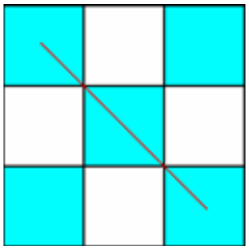
Chapitre III – Les problèmes d'échecs

Chapitre III-1

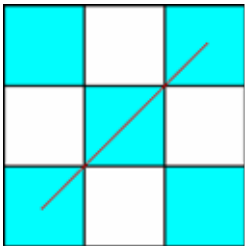
Le parcours du fou

Nous avons un échiquier et un fou situé dans un coin noir. Il faut faire arriver ce dernier dans le coin opposé, en le faisant passer une seule fois sur une case, dans le même sens de déplacement. Cela signifie que le fou ne peut pas passer deux fois par la même case en ayant le même sens de déplacement.

exemple : un fou a deux sens de déplacement. On décide que le premier sens, nommé A, et le deuxième sens, nommé B, sont indiqués sur les dessins suivants :

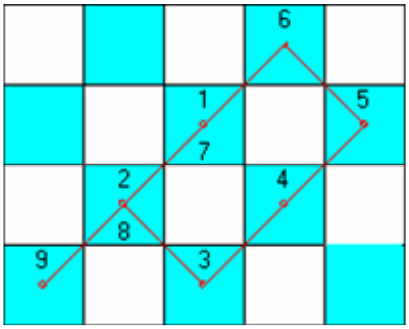


sens A

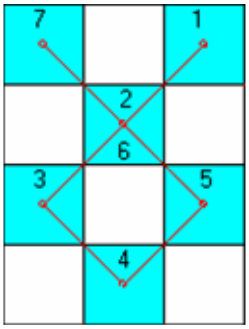


sens B

Ainsi, le parcours correspondant à l'image ci-dessous n'est pas possible, car le fou parcourt les cases 1,2,3,4,5,6 sans problème, mais ensuite il va de 7 (numéroté aussi 1) en 8 (numéroté aussi 2) dans le sens B (défini ci-dessus). Or, la case 1 avait déjà été visitée dans le sens B, pour aller dans la case 2. De même, le fou ne peut aller de la case 9 à la case 6 en passant par les cases 2 et 1, car ces deux cases ont déjà été visitées dans le sens B.



En revanche, le parcours correspondant au dessin suivant est possible, car le fou parcourt les cases 1,2,3 dans le sens B. Ensuite, une fois arrivé en 5 il passe à nouveau dans la case 2 (numérotée aussi 6), mais cette fois-ci dans le sens A. Une fois en 6, il ne pourra aller qu'en 7 car les cases 1 et 3 ont déjà été parcourues dans le sens B, et la case 5 a déjà été parcourue dans le sens A quand on est allé de 5 vers 6.



Nous allons créer l'algorithme de ce programme. On a un échiquier dont chaque case est associée à deux sens. Pour savoir si toutes les cases ont été parcourues, on ne peut pas compter le nombre de sauts, car on ne sait pas combien il y en a. En effet, le fou peut passer 2 fois sur une même case, et ceci pour n cases. Nous devons donc associer aussi aux cases une valeur booléenne, dont le rôle sera d'indiquer qu'elle a été visitée ou non. Si toutes les cases noires ont été visitées, alors on obtient un total de 32. Pendant le parcours, on doit mémoriser les numéros des cases parcourues, et, comme une case peut être visitée au maximum 2 fois, on a prévu un tableau de mémorisation dont la taille est égale au double du nombre de cases noires. Il est vrai que le fou ne peut aller dans les coins qu'une fois, donc il faudrait enlever 2 au résultat, mais le fait d'ajouter 2 éléments au tableau n'est pas un lourd handicap.

*exemple : pour un échiquier de 8*8 cases, il y a (8*8)/2 = 32 cases noires. Le nombre de coups théorique est de 32*2 (passage 2 fois dans chaque case) moins 2 (les coins), soit 62.*

Structures des données

```
const lim1 = 8;
      lim2 = 8;
type case_ec    = record
                  sensA,
                  sensB,
                  visite : boolean;
                  end;
     echiquier  = array[1..lim1,1..lim2] of case_ec;
var ech    : echiquier;  //--- echiquier des coups joués
    coups : array[1..lim1*lim2] of integer;  //--- liste des coups mémorisés
```

On écrit ensuite la fonction qui vérifie que toutes les cases ont été visitées:

```
function tout_a_ete_visite : boolean;
var i,j,total : integer;
begin
total:=0;
for i:=1 to lim1 do
for j:=1 to lim2 do
    if ech[i,j].visite then inc(total);
tout_a_ete_visite:=(total=(lim1*lim2 div 2));
end;
```

La procédure qui simule le parcours du fou est bien entendu récursive. Elle a trois paramètres :

- *x1* et *y1* sont les coordonnées du fou au coup précédent

- *coup* : numéro du coup, utilisé pour mémoriser les déplacements du fou

Nous avons décidé d'arrêter l'algorithme après le premier parcours trouvé, mais rien ne vous empêche de supprimer le test sur la variable *fini*, pour laisser le programme chercher toutes les solutions.

Voyons le coeur du programme : on a quatre directions possibles, représentées par les tableaux *dx* et *dy*. A partir de chaque case, on essaye d'aller dans chacune des quatre directions, ce qui nous amène à l'utilisation d'une boucle *for i:=1 to 4 do*. On calcule ensuite les coordonnées de la prochaine case, en utilisant les tableaux *dx* et *dy* : *new_x:=x1+dx[i]; new_y:=y1+dy[i];* et on s'assure que cette case est bien sur l'échiquier. Suivant la valeur de i, on se trouve ou bien dans le sens A, ou bien dans le sens B. Supposons que nous sommes dans le sens A. Comme on va se déplacer dans une autre case, il faut sauvegarder les paramètres de la case en cours, marquer son sens A, marquer le sens A de la case où on va sauter (de cette façon, si on y repasse au bout de n sauts, on ne pourra pas utiliser le sens A), faire un appel récursif avec les coordonnées de la nouvelle case, et quand on sort de la récursion, restaurer les valeurs telles qu'elles étaient avant le saut. Il ne faut évidemment pas oublier de positionner à *true* le booléen *visite* de chaque case qui a été visitée au moins une fois.

Le même ensemble d'instructions s'applique pour le sens B; nous obtenons le programme suivant:

```
procedure chemin(x1,y1,coup : integer);
const dx : array[1..4] of integer = (1,1,-1,-1);  //--- 1 et 3 : sens 1
      dy : array[1..4] of integer = (1,-1,-1,1);  //--- 2 et 4 : sens 2
var i,new_x,new_y : integer;
    old_s11,old_s12,old_s21,old_s22,visit : boolean;
begin
coups[coup]:=y1*10+x1;
if tout_a_ete_visite then
   begin
   max_coups:=coup;
   fini:=true;
   end
else
   begin
   for i:=1 to 4 do
       begin
       new_x:=x1+dx[i];
       new_y:=y1+dy[i];
       if not fini then
          if (new_x>=1) and (new_x<=lim1) and
             (new_y>=1) and (new_y<=lim2) then
          if (i mod 2)=1 then
             begin
             if not ech[new_x,new_y].sensA then
                begin
                //--- sauvegarde anciennes valeurs
                old_s11:=ech[new_x,new_y].sensA;
                old_s12:=ech[x1,y1].sensA;  //--- sauvegarde sensA
                visit:=ech[new_x,new_y].visite;
                //--- modifications
                ech[new_x,new_y].sensA:=true;
                ech[new_x,new_y].visite:=true;
                ech[x1,y1].sensA:=true;    //--- modif sensA
                //--- appel récursif
                chemin(new_x,new_y,coup+1);
                //--- restauration anciennes valeurs
                ech[x1,y1].sensA:=old_s12;  //--- restaure sensA
                ech[new_x,new_y].sensA:=old_s11;
                ech[new_x,new_y].visite:=visit;
                end;
             end
          else
             begin
             if not ech[new_x,new_y].sensB then
                begin
                //--- sauvegarde anciennes valeurs
```

```
                old_s21:=ech[new_x,new_y].sensB;
                old_s22:=ech[x1,y1].sensB;
                visit:=ech[new_x,new_y].visite;
                //--- modifications
                ech[new_x,new_y].sensB:=true;
                ech[new_x,new_y].visite:=true;
                ech[x1,y1].sensB:=true;
                //--- appel récursif
                chemin(new_x,new_y,coup+1);
                //--- restauration anciennes valeurs
                ech[x1,y1].sensB:=old_s22;
                ech[new_x,new_y].sensB:=old_s21;
                ech[new_x,new_y].visite:=visit;
                end;
            end;
        end;
    end;
end;
```

Chapitre III-2

Le problème du cavalier

Un autre problème bien connu est le parcours du cavalier : comment faut-il faire jouer un cavalier sur un echiquier de façon à ce qu'il passe par toutes les cases de l'échiquier ?

On distingue 2 types de parcours pour le cavalier :

1) les parcours constituant un cycle fermé (c'est à dire que le cavalier, après avoir parcouru toutes les cases de l'échiquier se retrouve à la même position qu'au départ)

2) les parcours constituant un cycle ouvert (le cavalier part d'une case quelconque et après avoir parcouru les 63 cases restantes il se retrouve sur une autre position que celle de départ.

Le problème du cavalier, par rapport à celui du fou ou des reines, est qu'il demande un temps considérable pour être résolu, cela à cause du grand nombre de combinaisons possibles.

Nous allons voir deux manières de résoudre ce problème.

Première solution

On considère l'échiquier comme un tableau en deux dimensions, représenté par la variable suivante : t : array[1..8,1..8] of byte.

Pour représenter les 8 directions, on s'aide de deux tableaux "t_x" et "t_y".

Quand le cavalier a sauté sur une case, alors la case est initialisée à 1, sinon à 0.

Un cavalier peut sauter dans 8 directions, mais le saut n'est pas toujours possible, notamment si le cavalier se trouve sur les bords de l'échiquier.

Pour chaque position du cavalier, on essaie les 8 directions; pour chaque direction possible, on met la case à 1 (simulation du saut), on sauvegarde la position dans un tableau et on fait un appel récursif; ensuite, une fois revenu, il faut annuler le coup, donc on remet la case à 0.

Voici le programme :

```
const t_x : array[1..8] of integer = (1,2,2,1,-1,-2,-2,-1);
      t_y : array[1..8] of integer = (-2,-1,1,2,2,1,-1,-2);

var t     : array[1..8,1..8] of byte; //--- l'échiquier
    p     : array[1..65] of integer;  //--- mémorise les sauts consécutifs
    s     : string;                   //--- variable auxiliaire d'affichage

procedure remplir;
begin
fillchar(t,sizeof(t),0);
end;

function on_peut_jouer(y,x,nr : byte) : boolean;
```

```
begin
on_peut_jouer:=false;
inc(x,t_x[nr]);inc(y,t_y[nr]);
if (x in [1..8]) and (y in [1..8]) then  //--- si le cavalier est dans l'échiquier
   on_peut_jouer:=t[y,x]=0;  //--- et si la case n'est pas occupée
end;

procedure récursive(y,x,cpt : byte);
var a : byte;
begin
if cpt=64 then
   begin
   s:='';
   for a:=1 to 64 do s:=s+inttostr(p[a])+';';
   inc(nb);
   form1.edit1.text:='solution numéro : '+inttostr(nb)+' : '+s;
   end
else
   for a:=1 to 8 do
       if on_peut_jouer(y,x,a) then
          begin
          t[y+t_y[a],x+t_x[a]]:=1;
          p[cpt]:=x*10+y;
          récursive(y+t_y[a],x+t_x[a],cpt+1);
          form1.edit2.text:=inttostr(y+t_y[a]);
          t[y+t_y[a],x+t_x[a]]:=0;
          end;
end;

procedure debut;
var ligne,colonne : byte;
begin
remplir;
ligne:=1;
colonne:=1;
t[ligne,colonne]:=1;
récursive(ligne,colonne,1);
t[ligne,colonne]:=0;
end;


procedure TForm1.Button1Click(Sender: TObject);
begin
debut;
end;
```

Le problème de cette solution est qu'elle est extrêmement lente : le cavalier ne parcourt que 250.000 cases par minute.

Deuxième solution

Voici maintenant une deuxième solution beaucoup plus efficace :

on ne considère plus le parcours de l'échiquier comme un ensemble de cases devant être initialisées à 1 quand le cheval est passé dessus (et remise à 0 quand le cheval revient d'une descente récursive) mais comme un ensemble de liens; on numérote les cases de 1 à 64. Ainsi quand le cavalier saute de la case 1 à la case 11, on initialise le lien "1->11" avec la valeur 1, ensuite on initialise tous les liens arrivant en 11 avec la valeur 1 (ce sont "5->11","17->11","21->11","26->11","28->11"); ainsi, quand l'ordinateur arrivera sur l'une de ces cases, il ne prendra aucun de ces chemins. Le fait de couper à chaque mouvement les sauts possibles pour une case donnée fait que le nombre de possibilités est réduit (malgré cela, il est encore très grand) et le programme trouve des solutions beaucoup plus rapidement. A titre indicatif, la première solution est trouvée en quatre minutes sur un K6-350, au bout de 17 millions de sauts du cavalier. Dans cette version du programme, le cavalier parcourt 4,4 millions de cases par minute. Le gain de temps est énorme. Nous avons

laissé tourner le PC pendant deux jours (ensuite nous l'avons arrêté); il a exploré 12 milliards de coups et a trouvé environ 9000 solutions.

Voici une représentation de l'échiquier utilisé dans cette version:

```
01 02 03 04 05 06 07 08
09 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64
```

Si on saute de 1 en 11 alors les liens "05->11, 17->11, 21->11,26->11, 28->11" sont coupés; il faut les mémoriser avant le saut, pour les restaurer après, pendant la remontée des niveaux récursifs

Les autres structures de données nécessaires sont le tableau "nb_sauts", qui indique le nombre de sauts possibles pour chaque case de l'échiquier:

```
nb_sauts : array[1..64] of integer =
   (2,3,4,4,4,4,3,2,
    3,4,6,6,6,6,4,3,
    4,6,8,8,8,8,6,4,
    4,6,8,8,8,8,6,4,
    4,6,8,8,8,8,6,4,
    4,6,8,8,8,8,6,4,
    3,4,6,6,6,6,4,3,
    2,3,4,4,4,4,3,2);

pos_saut : array[1..64] of integer =  // pour savoir à quelle position on doit
   (  1,  3,  6, 10, 14, 18, 22, 25,  // chercher la case suivante lors d'un
     27, 30, 34, 40, 46, 52, 58, 62,  // saut;
     65, 69, 75, 83, 91, 99,107,113,  // on recherche la case suivante dans le
    117,121,127,135,143,151,159,165,  // tableau "liens"
    169,173,179,187,195,203,211,217,
    221,225,231,239,247,255,263,269,
    273,276,280,286,292,298,304,308,
    311,313,316,320,324,328,332,335);

liens  : array[1..336] of integer =
  (11,18,    //----- liens "1->11", "1->18"
   12,17,19,   //----- liens "2->12", "2->17", "2->19"
   9,13,18,20,
   10,14,19,21,
  ...
```

On a un total de 336 sauts pour 64 cases, soit une moyenne de 5,25 sauts par case.

Et comme on a 64 cases à parcourir, alors le nombre total de possibilités est de 5,25 élevé à la puissance 64, soit $1{,}23*10^{46}$.

On utilise aussi des tableaux auxiliaires, comme "occupe" qui est un tableau de 336 booléens indiquant si un lien a déjà été visité ou pas. Pour chaque lien occupé (1->11) on occupe les liens allant vers 11. Mais une fois remonté, on les restaure; on a donc besoin de tableaux mémorisant les liens qu'on va occuper à chaque saut. Cette mémorisation s'effectue de la façon suivante:

```
por:=0;
for j:=1 to 336 do  // on occupe les liens
    if liens[j]=new_x then
       begin
       inc(por);
       po[por]:=j;
       oc[por]:=occupe[j];
       occupe[j]:=true;
       end;
```

et la restauration des liens après le remontée récursive s'effectue de la manière suivante:

```
for j:=1 to por do occupe[po[j]]:=oc[j];


procedure TForm1.Button4Click(Sender: TObject);
var num_saut,posit : integer;

    procedure remplir1;
    begin
    fillchar(occupe,sizeof(occupe),#0);
    end;

   procedure sauter(x,numero_saut : integer);
   var po : array[1..8] of integer;
       oc : array[1..8] of boolean;
       i,ii,j,k,m,por,new_x,c : integer;
       s1 : string;
   begin
   inc(num_saut);
   if numero_saut>63 then
      begin
      memo1.lines.clear;
      memo1.lines.add('saut='+inttostr(num_saut));
      memo1.lines.add(soluce);
      memo1.refresh;
      exit;
      end;
   for i:=1 to nb_sauts[x] do
       if not occupe[pos_saut[x]+i-1] then
          begin
          c:=pos_saut[x]+i-1;
          occupe[c]:=true;  // si x=1, "occupe[1]:=true" indique : lien "1->11"
occupé
          new_x:=liens[c];  // new_x:=11,18
          //--------- on occupe les liens
          por:=0;
          for j:=1 to 336 do
              if liens[j]=new_x then
                 begin
                 inc(por);
                 po[por]:=j;
                 oc[por]:=occupe[j];
                 occupe[j]:=true;
                 end;
          //--------- appel récursif
          s1:=inttostr(new_x)+';';
          soluce:=soluce+s1;
          sauter(new_x,numero_saut+1);
          delete(soluce,length(soluce)-length(s1)+1,length(s1));
          //--------- restauration des liens occupées
          for j:=1 to por do occupe[po[j]]:=oc[j];
          occupe[c]:=false;
          end;
   end;
begin  // chercher solution
remplir1;
soluce:='';num_saut:=1;
occupe[2]:=true;  // on interdit le saut "1->18" car symétrie avec "1->11"
posit:=1;
sauter(posit,1);
end;
```

Chapitre III-3

Le problème des huit reines

Un problème dont tout le monde a entendu parler et qui peut se résoudre facilement par l'ordinateur est le suivant: comment placer 8 reines sur un échiquier sans que chacune "mange" au moins une des 7 autres ?

Ce problème se résout par des essais successifs; on applique la récursivité. Nous allons présenter ici plusieurs solutions, à commencer d'abord par une solution itérative, que nous transformerons ensuite en une solution récursive.

Première solution, itérative :

On dispose de l'échiquier, qui est vide au début. On sauvegarde l'échiquier dans un tableau auxiliaire. On fait 8 boucles imbriquées qui représentent les 8 lignes; chaque boucle va de 0 à 7 ce qui représente une boucle de la première colonne à la huitième colonne. On utilise aussi une chaîne de caractères qui va représenter les positions des reines sur l'échiquier.

Pour chaque position définie par les coordonnées "ligne,colonne" :

- on vérifie si la case est vide

- si oui alors :

  - on occupe toutes les cases de l'échiquier dans les cinq directions décrites ci-dessous à partir de la position actuelle

  - on mémorise la colonne dans une chaîne de caractères, qui représente en réalité la position da la reine se trouvant sur la ligne en cours

  - on sauvegarde l'échiquier car il sera modifié par les coups suivants

  - on passe à la ligne suivante, à la première colonne

  - si on est à la dernière ligne alors on affiche la chaîne

- sinon :

  - on essaye la colonne suivante

- si on est à la huitième colonne on sort de la boucle et on restaure

  l'échiquier

Voici les cinq directions dont il s'agissait ci-dessus :

```
+--------+  X represente une reine.
¦        ¦
¦444X0000¦  Etant donné qu'on pose les 8 reines en descendant sur
¦  321   ¦  l'échiquier, cela n'a pas de sens de completer les 3
¦ 3 2 1  ¦  directions vers le haut, car on vient d'en haut et l'échiquier
¦3  2  1 ¦  a déjà été complété par un coup précédent.
```

```
+--------+

const max = 8;
      dx : array[0..4] of integer = (1,1,0,-1,-1);
      dy : array[0..4] of integer = (0,1,1, 1, 0);

type tab = array[0..max-1,0..max-1] of byte;

procedure occuper_toutes_les_directions(var tt : tab;xx,yy,valeur : byte);
var i,colonne,ligne : integer;
begin
tt[xx,yy]:=valeur;
for i:=0 to 4 do
    begin
    colonne:=xx;ligne:=yy;
    inc(colonne,dx[i]);inc(ligne,dy[i]);
    while (ligne in [0..7]) and (colonne in [0..7]) do
       begin
       tt[ligne,colonne]:=valeur;
       inc(colonne,dx[i]);inc(ligne,dy[i]);
       end;
    end;
end;

procedure jouer;
var x0,x1,x2,x3,x4,x5,x6,x7  : byte;
    st : string[8];
    tab_sauve : array[0..7] of tab;
begin
st:='       ';  // pour mémoriser les positions
fillchar(t[0],sizeof(t),#0);  // initialisation plateau
move(t,tab_sauve[0],64);
form1.memo1.lines.clear;
for x0:=0 to max-1 do
    if t[0,x0]=0 then
       begin
       occuper_toutes_les_directions(t,x0,0,2);
       st[1]:=chr(48+x0);
       move(t,tab_sauve[1],64);
       for x1:=0 to max-1 do
           if t[1,x1]=0 then
              begin
              occuper_toutes_les_directions(t,x1,1,2);
              st[2]:=chr(48+x1);
              move(t,tab_sauve[2],64);
              for x2:=0 to max-1 do
                  if t[2,x2]=0 then
                     begin
                     occuper_toutes_les_directions(t,x2,2,2);
                     st[3]:=chr(48+x2);
                     move(t,tab_sauve[3],64);
                     for x3:=0 to max-1 do
                         if t[3,x3]=0 then
                            begin
                            occuper_toutes_les_directions(t,x3,3,2);
                            st[4]:=chr(48+x3);
                            move(t,tab_sauve[4],64);
                            for x4:=0 to max-1 do
                                if t[4,x4]=0 then
                                   begin
                                   occuper_toutes_les_directions(t,x4,4,2);
                                   st[5]:=chr(48+x4);
                                   move(t,tab_sauve[5],64);
                                   for x5:=0 to max-1 do
                                       if t[5,x5]=0 then
                                          begin
                                          occuper_toutes_les_directions(t,x5,5,2);
                                          st[6]:=chr(48+x5);
                                          move(t,tab_sauve[6],64);
                                          for x6:=0 to max-1 do
                                              if t[6,x6]=0 then
                                                 begin
                                  occuper_toutes_les_directions(t,x6,6,2);
                                                 st[7]:=chr(48+x6);
                                                 for x7:=0 to max-1 do
                                                     if t[7,x7]=0 then
                                     form1.memo1.lines.add(st+chr(48+x7));
                                                 move(tab_sauve[6],t,64);
```

```
                                                    end;
                                                move(tab_sauve[5],t,64);
                                                end;
                                            move(tab_sauve[4],t,64);
                                            end;
                                        move(tab_sauve[3],t,64);
                                        end;
                                    move(tab_sauve[2],t,64);
                                    end;
                                move(tab_sauve[1],t,64);
                                end;
                            move(tab_sauve[0],t,64);
                            end;
end;
```

Deuxième solution, récursive :

Nous allons maintenant transformer toutes ces boucles imbriquées en une procédure récursive, comme on l'a déjà vu.

En tenant compte du fait que la récursivité a la propriété d'empiler les variables locales à une procédure, nous allons déclarer un échiquier local dans lequel on va faire les sauvegardes de l'échiquier, ainsi que la position du pion en cours, c'est à dire la chaîne "st".

Etant donné que la procédure comporte huit boucles, nous allons rajouter un paramètre à la procédure "jouer", qui représente le numéro de la ligne, et un autre paramètre pour la chaîne de caractères; en même temps il ne faut pas oublier que d'une procédure à l'autre (en réalité d'un niveau "n" pendant la récursivité, au niveau "n+1") on doit transmettre l'échiquier.

```
procedure jouer1(ligne : integer;var t : tab;st : string);  // récursive 1
var colonne     : byte;
    tab_sauve   : tab;
begin
move(t,tab_sauve,64);
for colonne:=0 to max-1 do
    if t[ligne,colonne]=0 then
       begin
       occuper_toutes_les_directions(t,colonne,ligne,2);
       if ligne<7 then
          jouer1(ligne+1,st+chr(49+colonne))
       else
          form1.memo1.lines.add(st+chr(49+colonne));
       move(tab_sauve,t,64);
       end;
end;
```

Troisième solution, récursive :

Nous allons voir maintenant comment on peut améliorer ce programme du point de vue de la rapidité; pour commencer, nous n'allons plus remplir l'échiquier dans les 5 directions à chaque coup, mais nous allons vérifier si une case se trouve dans la ligne de mire d'une reine précédente; pour cela on fait une fonction "occupé", qui ne vérifie que les trois directions manquantes, celles qui vont vers le haut.

```
const max = 8;
      dx2 : array[0..2] of integer = (-1,0,1);
      dy2 : array[0..2] of integer = (-1,-1,-1);

function occupe2(x,y : integer;t : tab) : boolean;
var i,colonne,ligne : integer;
    sortir : boolean;
begin
i:=-1;sortir:=false;
repeat
    inc(i);colonne:=x;ligne:=y;
    dec(colonne,dx2[i]);dec(ligne,dy2[i]);
    repeat
      inc(colonne,dx2[i]);inc(ligne,dy2[i]);
```

```
      if (colonne in [0..max-1]) and (ligne>=0) then
         sortir:=(t[ligne,colonne]<>0);
    until (colonne=-1) or (colonne=max) or (ligne=-1) or sortir;
until (i=2) or sortir;  // i=0,1,2 : les trois directions
occupe2:=sortir;
end;

procedure jouer2(ligne : byte;var t : tab;st : string);
var x  : byte;
begin
for x:=0 to max-1 do
    if not occupe2(x,ligne) then
       begin
       t[ligne,x]:=1;   // on pose une reine
       if ligne<max-1 then
          jouer(ligne+1,t,st+chr(49+x))
       else
          form1.memo1.lines.add(st+chr(49+x));
       t[ligne,x]:=0;   // on enlève la reine
       end;
end;
```

Quatrième solution, récursive :

Il y a d'autres solutions à ce problème. Par exemple, l'échiquier est représenté par une chaîne de 8 caractères. Chacun des 8 caractères représente une ligne de l'échiquier, et la colonne est représentée par le numéro inscrit dans la chaîne.

Supposons qu'on a la situation suivante :

- une reine en ligne 1, colonne 1

- une reine en ligne 2, colonne 3

- une reine en ligne 3, colonne 5

La chaîne est donc égale à : st='13500000'.

Supposons qu'on veuille placer une reine sur la quatrième ligne; pour cela il faut que la quatrième ligne de l'échiquier soit vide, c'est à dire qu'il faut que le quatrième élément de la chaîne soit '0'.

Ensuite il faut parcourir les huit colonnes de cette ligne et dès qu'il y a une case qui ne soit pas en danger, y placer une reine. Pour savoir si la case n'est pas en danger il faut vérifier si la colonne n'a pas déjà été occupée.

Ainsi si on veut placer une reine en quatrième ligne et première colonne, on ne peut pas car la première colonne (notée par "1" dans la chaîne "st") a déjà été utilisée par la première reine.

Par contre la deuxième colonne n'a pas encore été utilisée, car la chaîne "st" ne contient pas le chiffre "2"; pour savoir si on peut placer une reine, il ne nous reste plus qu'à vérifier les diagonales en partant de la position actuelle qui est ligne 4, colonne 2; il ne faut pas qu'on ait une reine sur les positions suivantes:

```
ligne 3, colonne 1  ---> première diagonale, gauche,haut
ligne 3, colonne 3  +
ligne 2, colonne 4  ¦--> deuxième diagonale, droite, haut
ligne 1, colonne 5  +
```

On remarque que pour faire les tests sur la diagonale on décrémente de 1 la ligne et on décrémente (ou incrémente) de 1 la colonne, et ceci jusqu'à

sortir de l'échiquier (en fait arriver aux bords de "st") ou rencontrer une reine. La fonction qui vérifie les diagonales s'appelle "test" et elle est récursive. La procédure qui place les reines sur l'échiquier est elle aussi récursive. On a le programme :

```
function test(ligne,colonne,delta : integer;var st : string) : boolean;
begin
if (ligne in [0,max+1]) or (colonne in [0,max+1]) then
   test:=true
else
   test:=(st[ligne]<>chr(48+colonne)) and test(ligne-1,colonne+delta,delta,st)
end;

procedure jouer3(lg : integer;st : string);
var col : integer;
begin
if lg=max+1 then
   form1.memo1.lines.add(st)
else
   for col:=1 to max do
       if (pos(chr(col+48),st)=0) and test(lg,col,-1,st) and test(lg,col,1,st) then
          begin
          st[lg]:=chr(48+col);
          jouer3(lg+1,st);
          st[lg]:='0';
          end;
end;
```

Et voici les appels à chacune de ces quatre procédures :

```
procedure TForm1.Button1Click(Sender: TObject);
var t : tab;
begin  // cherche solution
memo1.Clear;
//---- solution itérative à 8 boucles
// jouer;

//---- solution récursive 1
//fillchar(t,sizeof(t),#0);
//jouer1(0,t,'');

//---- solution récursive 2
//fillchar(t,sizeof(t),#0);
//jouer2(0,t,'');

//---- solution récursive 3
jouer3(1,'        ');  // chaîne de 8 espaces
end;
```

Chapitre IV - Divers

Chapitre IV-1

Le triangle de Pascal

Le triangle de Pascal est le tableau des coefficients qui sont utilisés pour le développement de certaines expressions comme $(a+b)^2$ ou $(a+b)^n$.

Cela s'appelle la "formule du binôme de Newton". Les coefficients s'appellent les "coefficients binomiaux" ou "coefficients du binôme".

Ce triangle est le suivant :

```
0 : 1           (a+b)^0 = 1
1 : 1 1         (a+b)^1 = 1*a + 1*b
2 : 1 2 1       (a+b)^2 = 1*a^2 + 2*a*b + 1*b^2
3 : 1 3 3 1     (a+b)^3 = 1*a^3 + 3*a^2*b + 3*a*b^2 + 1*b^3
4 : 1 4 6 4 1   (a+b)^4 = 1*a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + 1*b^4
```

On obtient chaque coefficient en additionnant le nombre qui lui est situé au-dessus ainsi que celui qui lui est situé au-dessus à gauche.

Exemples :

- on obtient le 6 en faisant 3+3

- on obtient le 4 qui est après le 6 en faisant 3+1.

Le numéro qui est en tête de chaque ligne de ce triangle est la puissance à laquelle "a+b" est élevé;ainsi pour la puissance 4,"$(a+b)^4$" admet les coefficients 1, 4, 6, 4, 1 qu'on voit dans l'expression développée.

Etudions l'algorithme nécessaire à l'affichage de ce triangle :

- on remarque que la diagonale est toujours à 1 ---> point d'appui

- on remarque que la première colonne est toujours à 1 ---> point d'appui

- pour tout autre élément qui se trouve à la ligne y et colonne x, on

  affiche la somme de - l'élément de coordonnées y-1,x

  - l'élément de coordonnées y-1,x-1

```
function comb(x,y:integer):integer;
begin
if (x=1) or //--- point d'appui : la première colonne
   (y=x)    //--- point d'appui : la diagonale
   then comb:=1
else
   comb:=comb(x,y-1)+comb(x-1,y-1);  //--- appel récursif
end;
```

On voit ici la définition récursive, car tout élément du tableau est défini par deux autres éléments qui sont inconnus, chacun étant défini par deux autres et ainsi de suite, excepté la diagonale et la première colonne; c'est grâce à ces deux indices qu'on va déterminer tout le tableau.

Voici le programme qui affiche le triangle de Pascal.

```
procedure tform1.button1Click(sender : tobject);
var  ligne,colonne:integer;

     function comb(x,y:integer):integer;
     begin
     if (x=1) or (y=x) then
        comb:=1
     else
        comb:=comb(x,y-1)+comb(x-1,y-1);
     end;
begin
for ligne:=1 to 15 do
    for colonne:=1 to ligne do
        canvas.textOut(colonne*30,ligne*30,inttostr(comb(colonne,ligne)));
end;
```

Et maintenant, un peu de maths :

Formule du triangle de Pascal

$$C_{n-1}^{p-1} + C_{n-1}^{p} = C_n^p$$

Démonstration combinatoire

Soit E un ensemble à n éléments et a un élément de E.

Notons :

A l'ensemble des parties de E à p éléments ($p \leq n$),

B l'ensemble des parties à p éléments de E contenant a,

C l'ensemble des parties à p éléments de E ne contenant pas a.

Le cardinal de A est C_n^p.

Comme B est l'ensemble des parties à p-1 éléments de $E - \{a\}$ auxquelles on a ajouté a, le cardinal de B est égal à celui de l'ensemble des parties à p-1 éléments de $E - \{a\}$. D'où Card(A)= C_{n-1}^{p-1}.

Comme C est l'ensemble des parties à p éléments de $E - \{a\}$, Card(C)= C_{n-1}^{p}.

A étant la réunion disjointe de B et C, on a Card(A)=Card(B)+Card(C), d'où le résultat.

Démonstration directe

$$C_{n-1}^{p-1} + C_{n-1}^{p} = \frac{(n-1)!}{(p-1)!(n-p)!} + \frac{(n-1)!}{p!(n-p-1)!} = \frac{(n-1)!}{(p-1)!(n-p)(n-p-1)!} + \frac{(n-1)!}{p(p-1)!(n-p-1)!}$$

$$= \frac{p(n-1)!+(n-p)(n-1)!}{p(p-1)!(n-p)(n-p-1)!} = \frac{n!}{p!(n-p)!}$$

Formule du binôme de Newton

$$(a+b)^n = \sum_{p=0}^{n} C_n^p a^p b^{n-p}$$

Démonstration combinatoire

Le coefficient du terme $a^p b^{n-p}$ est égal au nombre de façons de choisir simultanément p paires de parenthèses contenant a parmi les n, soit C_n^p, d'où le résultat (c'est aussi le nombre de façons de choisir simultanément n-p paires de parenthèses contenant b parmi les n, soit C_n^{n-p}. Or $C_n^p = C_n^{n-p}$ …).

Démonstration par récurrence

La formule se vérifie aisément pour n=0.

Supposons qu'elle est vraie pour n fixé et montrons qu'alors $(a+b)^{n+1} = \sum_{p=0}^{n+1} C_{n+1}^p a^p b^{n+1-p}$

$$(a+b)^{n+1} = (a+b)(a+b)^n = \sum_{p=0}^{n} C_n^p a^{p+1} b^{n-p} + \sum_{p=0}^{n} C_n^p a^p b^{n+1-p}$$

$$= \sum_{p=1}^{n+1} C_n^{p-1} a^p b^{n-p+1} + \sum_{p=0}^{n} C_n^p a^p b^{n+1-p}$$

$$= a^{n+1} + \sum_{p=1}^{n} C_n^{p-1} a^p b^{n-p+1} + b^{n+1} + \sum_{p=1}^{n} C_n^p a^p b^{n+1-p}$$

$$= a^{n+1} + \sum_{p=1}^{n} \left(C_n^{p-1} + C_n^p\right) a^p b^{n+1-p} + b^{n+1}$$

$= a^{n+1} + \sum_{p=1}^{n} C_{n+1}^p a^p b^{n+1-p} + b^{n+1}$, d'après la formule de Pascal, d'où le résultat.

La formule étant vraie pour n=0 et l'étant pour n+1 sous l'hypothèse qu'elle l'est pour n fixé, est donc vraie pour tout entier naturel n en vertu de l'axiome de récurrence.

Une application amusante

Calculons 1001^6.

$$1001^6 = \sum_{p=0}^{6} C_6^p 1000^p = 1 + C_6^1 \times 1000 + C_6^2 \times 1000000 + C_6^3 \times 1000000000 + C_6^4 \times 1000000000000$$
$$+ C_6^5 \times 1000000000000000 + C_6^6 \times 1000000000000000000$$

=
1+6x1000+15x1000000+20x1000000000+15x1000000000000+6x10000
00000000000+1000000000000000000

Nous laissons au lecteur le soin de faire l'addition !

Un résultat remarquable

$$\sum_{p=0}^{n} C_n^p = 2^n$$

Démonstration 1

Le résultat est immédiat en faisant a=b=1 dans la formule du binôme !

Démonstration 2 (combinatoire)

La somme de tous les C_n^p pour n fixé (la somme de tous les coefficients binomiaux d'une ligne du triangle de Pascal) est égale au nombre de façons de choisir simultanément entre 0 et n éléments d'un ensemble à n éléments, c'est à dire exactement au nombre de parties de cet ensemble, soit 2^n.

On montre directement que le nombre de parties d'un ensemble E à n éléments est 2^n en remarquant qu'on peut associer à chaque partie P de E l'application f de E sur {0,1} ainsi définie :

f(x)=0 si x n'est pas un élément de P

f(x)=1 si x est un élément de P.

Comme Card(E)=n et Card({0,1})=2, on a le résultat, puisque le nombre d'applications d'un ensemble de cardinal p dans un ensemble de cardinal n est n^p.)

Chapitre IV-2

Les tours de Hanoï

Un problème bien connu qui se résout très facilement par un algorithme récursif (et difficilement par un algorithme itératif, comme nous le verrons plus loin) est "les tours de Hanoi". La légende dit que dans un temple de Hanoï, à l'origine du monde, 64 disques de diamètre croissant étaient empilés sur un taquet. Deux autres taquets sont disponibles, qui sont utilisés pour déplacer les disques, la condition étant qu'un disque d'un certain diamètre ne peut pas être placé au dessus d'un disque de diamètre inférieur. Donc, les disques sont toujours empilés dans un certain ordre: les plus grands sont toujours en bas du taquet. La légende dit aussi que des moines sont en train de déplacer les 64 disques vers l'un des deux autres, au rythme de un par seconde et quand ils auront terminé, ce sera la fin du monde.

En faisant un petit calcul rapide, on a 2^{64}-1 secondes, ce qui correspond à 585 milliards d'années et sachant que l'âge de l'univers est de seulement 15 milliards d'années, on a encore de la marge : ouf ! On l'a échappé belle! :)))

Voyons maintenant la programmation de cet algorithme.

On décide que les disques ont un numéro correspondant à leur diamètre, et que les diamètres vont de 1 à 64.

Nous allons d'abord étudier deux cas concrets de déplacement: comme d'habitude, d'abord on observe, ensuite on programme l'algorithme.

Soit deux disques à déplacer du taquet 1 vers le 3.

Le taquet final est 3, celui initial est 1, donc celui intermédiaire est 2.

Opérations de déplacement :

- disque 1 de taquet 1 vers taquet 2

- disque 2 de taquet 1 vers taquet 3 ---> milieu

- disque 1 de taquet 2 vers taquet 3

Soit trois disques à déplacer du taquet 1 vers le 3.

Le taquet final est 3, celui initial est 1, donc celui intermédiaire est 2.

Maintenant on doit se retrouver avec les deux premiers disques sur le taquet 2, de façon à pouvoir déplacer le disque 3 vers la taquet 3 et ensuite les deux premiers du taquet 2 vers le taquet 3.

Opérations de déplacement :

- disque 1 de taquet 1 vers taquet 3

- disque 2 de taquet 1 vers taquet 2

- disque 1 de taquet 3 vers taquet 2

- disque 3 de taquet 1 vers taquet 3 ---> milieu

- disque 1 de taquet 2 vers taquet 1

- disque 2 de taquet 2 vers taquet 3

- disque 1 de taquet 1 vers taquet 3

En regardant bien les déplacements effectués, on constate les choses suivantes :

- si on a "n" disques à déplacer :

  - on déplace "n-1" disques vers le taquet intermédiaire

  - on déplace le disque "n" du taquet initial vers le taquet final ---> milieu

  - on déplace les "n-1" disques du taquet intermédiaire vers le taquet final

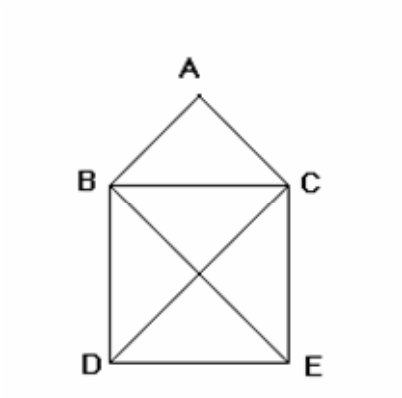
On peut d'ores et déjà en déduire l'algorithme :

```
procedure TForm1.Button1Click(Sender: TObject);
   procedure hanoi(nb_disques,depart,arrivee : integer);
   var intermediaire : integer;
   begin
   if nb_disques=1 then
      memo1.lines.add('On déplace un disque de '+
            inttostr(depart)+' vers '+inttostr(arrivee))
   else
      begin
      intermediaire:=6-depart-arrivee;
      hanoi(nb_disques-1,depart,intermediaire);
      memo1.lines.add('On déplace un disque de '+
             inttostr(depart)+' vers '+inttostr(arrivee));
      hanoi(nb_disques-1,intermediaire,arrivee);
      end;
   end;
begin  // deplacer 4 disques
hanoi(4,1,2);
end;
```

Chapitre IV-3

La maison

Soit la figure suivante :



On demande de faire le programme qui trace cette maison en un seul coup, c'est à dire sans "lever le crayon".

Structure des données

Nous devons d'abord représenter la maison en mémoire; on voit que la maison a 5 sommets : A, B, C, D, E. Pour représenter cela, nous allons utiliser un tableau de 2 dimensions qui contiendra 5 lignes et 5 colonnes.

Ensuite, dans ce tableau, on va représenter tous les chemins qu'on peut tracer par un 0 et tous les chemins qu'on ne peut pas tracer (ou qui ont déjà été tracés) par un 1.

On ne peut pas aller de A à A, donc dans le tableau, aux coordonnées (0,0) nous avons un 1; de même pour les autres sommets. Donc la diagonale du tableau (représentant les chemins AA,BB,CC,DD,EE) est mise à 1; en regardant la maison on voit qu'on ne peut pas aller de A à D (donc de D à A non plus) et de A à E (donc de E à A non plus); pour indiquer qu'on ne peut pas (ou plus) aller de A à D, on remplit le tableau en première ligne et quatrième colonne avec 1; pour indiquer qu'on ne peut pas aller de D à A on remplit le tableau en quatrième ligne (car D est le quatrième élément) et première colonne (car A est le premier élément) avec un 1. Idem pour AE.

Analyse de l'algorithme

Faisons une petite analyse du programme:

- on compte les segments à dessiner: il y en a 8 : AB,AC,BC,BD,BE,CD,CE,DE.

- nous allons faire tous les essais possibles sur cette maison; on va donc commencer à la première ligne du tableau et parcourir les colonnes jusqu'à trouver un 0, ce qui indique qu'on peut aller vers

ce sommet (par exemple nous sommes à la première ligne, donc au sommet A, et on parcourt les colonnes de cette ligne; on trouve un 0 à la deuxième colonne, donc on peut aller à B; on occupe immédiatement cette case ainsi que celle qui va de B vers A, et on démarre maintenant à la ligne correspondant à B, c'est à dire à la deuxième ligne; la première colonne vide est celle qui correspond à C, étant donné que BA a été rempli ci-dessus; on remplit les cases correspondant aux chemins BC et CB et on démarre à la ligne correspondant à C, c'est à dire la troisième ligne, et ainsi de suite.

- tout ceci se fait par des appels récursifs; la procédure récursive s'appelle "cherche" et elle a 3 paramètres :

- y: c'est le sommet d'où on part, c'est à dire la ligne en cours dans le tableau

- niveau: c'est la variable qui indique combien de segments on a tracé; si on a tracé le huitième segment, alors un appel récursif a lieu avec niveau=9; ceci est le point d'arrêt et on affiche les positions accumulées dans le paramètre "st"

- st : chaîne de caractères qui mémorise le chemin parcouru

Le programme

Voici le programme correspondant :

```
procedure TForm1.Button1Click(Sender: TObject);
type tab = array[0..4,0..4] of integer;

const t1 : tab =
          ((1,0,0,1,1),
           (0,1,0,0,0),
           (0,0,1,0,0),
           (1,0,0,1,0),
           (1,0,0,0,1));

var t : tab;

   procedure cherche(colonne,niveau : integer;st : string);
   var ligne : integer;
   begin
   if niveau=9 then
      memo1.lines.add(st)
   else
      for ligne:=0 to 4 do
          if t[ligne,colonne]=0 then
             begin
             st:=st+chr(65+colonne)+chr(65+ligne)+' ';
             t[ligne,colonne]:=1;t[colonne,ligne]:=1;
             cherche(ligne,niveau+1,st);
             t[ligne,colonne]:=0;t[colonne,ligne]:=0;
             delete(st,length(st)-2,3);
             end;
   end;

   procedure recherche;
   var colonne : integer;
   begin
   memo1.lines.clear;
   move(t1,t,sizeof(t));
   for colonne:=0 to 4 do
       cherche(colonne,1,'');
   end;

begin
```

```
recherche;
end;
```

Chapitre IV-4

Le labyrinthe

Soit un labyrinthe quelconque qui a une entrée et une sortie, ainsi qu'au moins un chemin menant de l'entrée vers la sortie. On veut faire le programme qui trouve l'un de ces chemins. Le principe est simple: nous allons explorer une direction et si elle ne donne aucun résultat, on reprendra une autre direction dans le labyrinthe (une autre branche de l'arbre récursif).

Nous allons étudier quelques détails d'un parcours dans un labyrinthe:

- dans un labyrinthe nous avons quatre directions: nord, est, sud, ouest

- on avance dans le labyrinthe, il faut laisser une trace du passage dans le labyrinthe, car si on a un mur isolé dans le labyrinthe, il ne faut pas tourner à l'infini autour de ce mur

- si à un certain moment on est bloqué, alors on efface le chemin parcouru jusqu'au dernier "carrefour" rencontré dans le labyrinthe et on essaie un autre chemin partant de ce carrefour

- si on est à l'arrivée alors on initialise une variable "trouvé" à true, pour indiquer qu'il ne faut plus effacer la trace laissée derrière, car il faut quand même visualiser le chemin trouvé.

Voyons maintenant l'algorithme :

- l'algorithme sera une procédure qui admet deux paramètres: "ligne" et "colonne"; ils représentent la position en cours

- le point d'appui est atteint quand "ligne" et "colonne" sont égales aux coordonnées du point d'arrivée (la sortie du labyrinthe)

- si on n'est pas au point d'arrivée alors:

1) - on laisse une trace dans le labyrinthe à la position en cours

2) - on affiche le labyrinthe avec la trace

3) - on essaie les quatre directions

- si on peut aller dans une de ces quatre directions (il n'y a pas de mur devant) alors on recommence l'algorithme avec la nouvelle position, celle qui va dans la direction choisie

- si on a essayé toutes les directions alors si "trouvé" est "false" (on n'a pas trouvé la sortie), on efface le chemin.

Remarque: cet algorithme ne trouve pas le meilleur chemin; il trouve simplement un chemin; de plus la longueur du chemin et le temps mis pour le trouver dépend du labyrinthe, de l'emplacement

de l'entrée et de la sortie, et du sens qu'on a choisi pour l'exploration des directions d'un carrefour; ici on explore les directions dans l'ordre suivant: nord, ouest, sud, est.

Voici le programme correspondant :

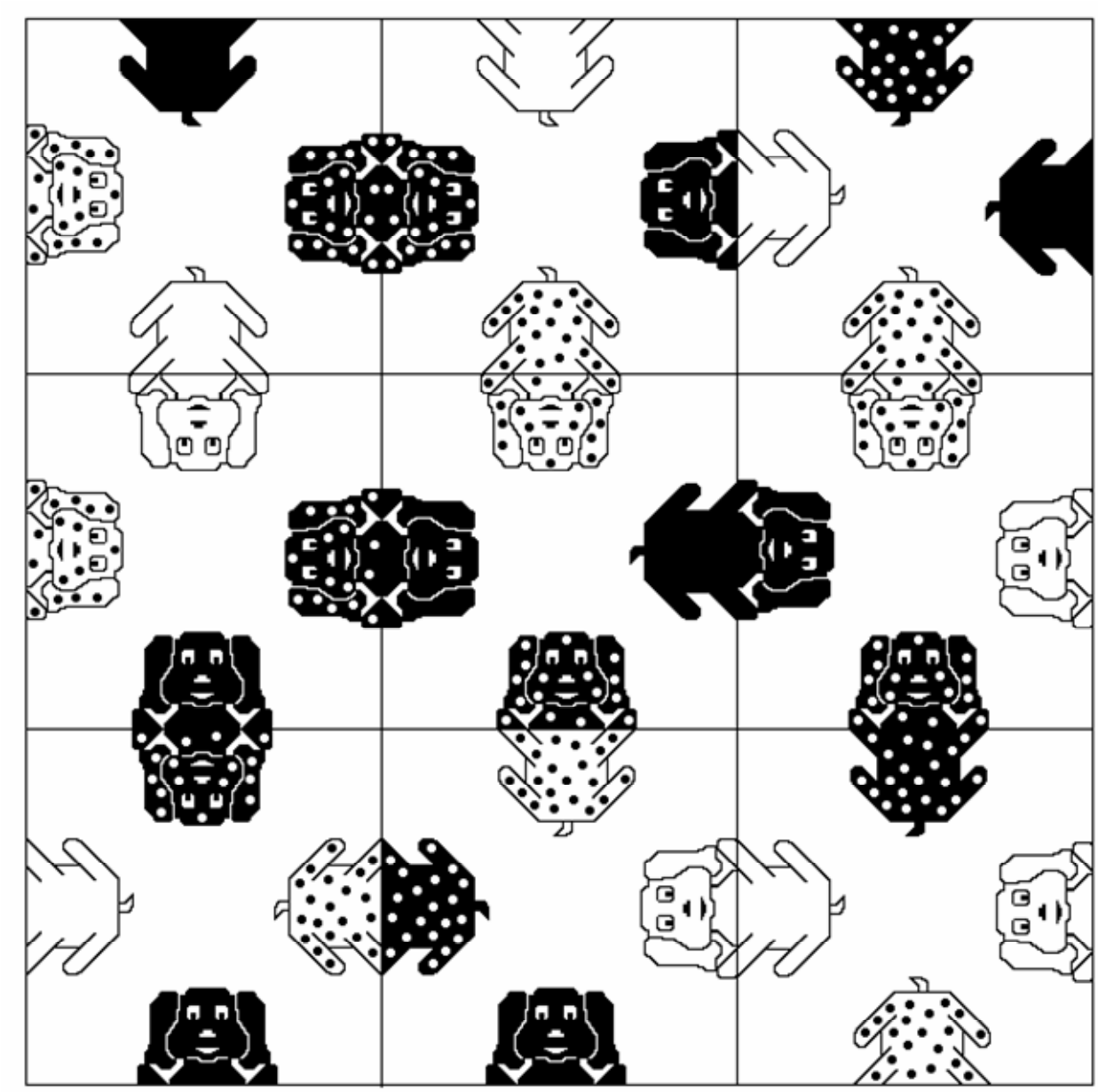
```
type lab = array[1..15] of string[40];

const l: lab = ('$$$$$$ $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$',
                '$ $$ $         $$$    $$ $$ $$$$ $ $ $$$',
                '$  $ $$$$$$$$$ $$$$$$        $   $     $',
                '$ $$      $$$$        $$ $$$$$ $$$ $$$$$',
                '$ $  $$ $ $    $$$ $$$$$       $   $$$ $',
                '$ $$ $  $$$ $$$$$$$$$  $ $$ $$$$ $$$ $ $',
                '$    $ $$         $ $ $$$$$ $      $   $',
                '$$$$ $ $$ $$$$ $$$$   $$  $$$ $$$$$$ $$$',
                '$$   $    $ $$ $$   $$$$ $$           $$',
                '$$ $$$$$$$$ $$$$$ $$$$ $ $$ $$$$$ $$$$$$',
                '$$ $                    $$           $',
                '$$ $$$ $$ $$$$$ $$$ $$$ $$$$$ $$ $ $$ $',
                '$$ $$$ $$ $$    $$$  $$ $$ $$$ $$$$$$$$',
                '$$     $   $$$$     $$               $$',
                '$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ $$$$$');

       dx : array[0..3] of integer = (0,-1,0,1);
       dy : array[0..3] of integer = (-1,0,1,0);

var  t : lab;
     trouve : boolean;

procedure tform1.aff_labyrinthe;
var ligne,colonne : integer;
begin
image1.picture.bitmap.canvas.font.size:=8;
for ligne:=1 to 15 do
for colonne:=1 to 40 do
    begin
    case t[ligne][colonne] of
      '$' : with image1.picture.bitmap.canvas do
                begin
                brush.Style:=bsSolid;
                brush.color:=clBlack;
                pen.color:=clBlack;
                rectangle((colonne-1)*15,(ligne-1)*15,colonne*15,ligne*15);
                end;
      '*' : with image1.picture.bitmap.canvas do
                begin
                brush.Style:=bsClear;
                font.color:=clBlue;
                textOut((colonne-1)*15+5,(ligne-1)*15+2,'*');
                end;
      ' ' : with image1.picture.bitmap.canvas do
                begin
                brush.Style:=bsSolid;
                brush.color:=clWhite;
                pen.color:=clWhite;
                rectangle((colonne-1)*15,(ligne-1)*15,colonne*15,ligne*15);
                end;
      end;
    end;
end;

procedure tform1.avancer(ligne,colonne : integer);
var i  : integer;
begin
if not trouve then
   if (ligne=1) and (colonne=7) then
      begin
      trouve:=true;
      edit1.text:='trouvé';
      end
   else
      begin
      t[ligne][colonne]:='*';
      aff_labyrinthe;image1.refresh;
```

```
      for i:=0 to 3 do
          if t[ligne+dy[i]][colonne+dx[i]]=' ' then
             avancer(ligne+dy[i],colonne+dx[i]);
      if not trouve then
         begin
         t[ligne][colonne]:=' ';
         aff_labyrinthe;image1.refresh;
         end;
      end;
   end;

procedure TForm1.FormCreate(Sender: TObject);
var tb : tbitmap;
begin
move(l,t,sizeof(l));
trouve:=false;
tb:=tbitmap.create;
tb.width:=15*40;
tb.height:=15*15;  //--- nb d'elems de t; high(t)
image1.picture.bitmap:=tb;
image1.autosize:=true;
aff_labyrinthe;
end;



procedure TForm1.Button1Click(Sender: TObject);
begin
edit1.text:='recherche en cours...';edit1.refresh;
avancer(15,34);
aff_labyrinthe;
end;
```

Chapitre IV-5

Les petits chiens

Un problème amusant et pouvant être facilement résolu par l'ordinateur est le problème des petits chiens: on dispose de neuf cartes, chacune contenant quatre têtes ou corps des chiens; le but du jeu est de placer les cartes de manière à avoir réunis la tête et le corps des chiens de même couleur; il y a plusieurs possibilités et comme pour le problème des 8 reines ou du cavalier, nous allons utiliser la récursivité. Les cartes doivent former un carré.



Il est évident qu'il faut former des combinaisons des 9 cartes; pour cela on utilise le programme des anagrammes; on se sert de la deuxième solution, car elle est plus rapide.

```
procedure combinaisons;
```

```
var  ch : string;
     l  : byte;

     procedure EchangeCar (var ch : string; i, j : byte);
     var car : char;
     begin
     if i <> j then
        begin
        car  :=ch[i];
        ch[i]:=ch[j];
        ch[j]:=car;
        end
     end;

     procedure anagram (ch : string; i : byte);
     var j : byte;
     begin
     if i = l then memo1.lines.add(ch)
        else
        for j := i to l do
            begin
            EchangeCar (ch,i,j);
            Anagram (ch,i+1);
            EchangeCar (ch,i,j);
            end;
     end;

begin
ch:='abc';l:=length(ch);
anagram(ch,1);
end;
```

Cette procédure est insuffisante pour résoudre le problème, car les cartes peuvent tourner sur elles-mêmes, quatre fois chacune. Donc pour chaque combinaison obtenue de 9 cartes, il faut faire tourner les cartes et vérifier que la position obtenue crée 12 chiens, sinon il faut essayer une autre position.

Pour vérifier que les chiens sont en position, on fait les tests suivants:

- on pose la première carte

- on pose la deuxième; l'ouest de la deuxième doit correspondre à l'est de la première

- on pose la troisième; l'ouest de la troisième doit correspondre à l'est de la deuxième

- on pose la quatrième; le nord de la quatrième doit correspondre au sud de la première

- on pose la cinquième; le nord de la cinquième doit correspondre au sud de la deuxième et l'ouest de la cinquième doit correspondre à l'est de la quatrième

- on pose la sixième; le nord de la sixième doit correspondre au sud de la troisième et l'ouest de la sixième doit correspondre à l'est de la cinquième

- on pose la septième; le nord de la septième doit correspondre au sud de la quatrième

- on pose la huitième; le nord de la huitième doit correspondre au sud de la cinquième et l'ouest de la huitième doit correspondre à l'est de la septième

- on pose la neuvième; le nord de la neuvième doit correspondre au sud de la sixième et l'ouest de la neuvième doit correspondre à l'est de la huitième

On fait donc une fonction qui vérifie ce qu'il faut en fonction du numéro de la carte passée en paramètre; mais pour vérifier qu'une tête correspond à un corps, on numérote les têtes avec des valeurs positives et les corps correspondants avec des valeurs négatives, de sorte que la somme de la tête et du corps du même chien fasse zéro. En faisant les sommes on peut donc vérifier si une carte est bien placée.

Ensuite pour chaque combinaison obtenue avec les anagrammes, on fait les rotations; la procédure qui s'occupe des rotations fait les étapes suivantes :

- elle a un paramètre qui est le numéro de la carte à tourner

- elle fait une boucle 4 fois pour tourner la carte

- si la carte est bien placée (vérification avec la fonction ci-dessus) alors elle s'appelle elle-même avec un paramètre correspondant au numéro de la carte suivante;

- si le paramètre est 10, alors c'est qu'elle a déjà tourné 9 cartes et qu'elles sont toutes bien placées; c'est donc la solution, et on affiche les 9 cartes.

Le nombre de possibilités (façon d'arranger les cartes) est de $(4^9)*9!$ possibilités. C'est à dire environ 95 milliards, ce qui en fait un bon paquet pour un jeu aussi simple à première vue!

Voici le programme correspondant :

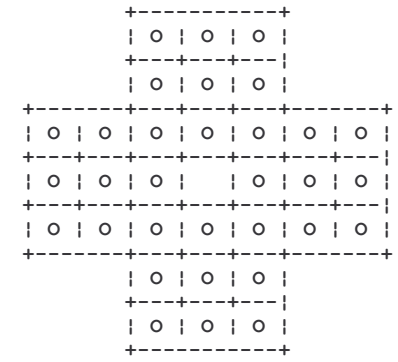
```
type carte = record
             nord,est,sud,ouest : integer;
             end;

const  TETE1 = 1; CORPS1 = -1;  {--- rouge ---}
       TETE2 = 2; CORPS2 = -2;  {--- vert ---}
       TETE3 = 3; CORPS3 = -3;  {--- bleu ---}
       TETE4 = 4; CORPS4 = -4;  {--- jaune ---}

var t        : array[1..9] of carte;
    longueur : byte;
    st       : string;
    ch       : char;
    car      : carte;
    nb       : longint;
    num      : integer;


//*************************************************************************

function carte_ok(nb : integer) : boolean;
begin
   case nb of
   1 : carte_ok:=true;
   2 : carte_ok:=t[1].est+t[2].ouest=0;
   3 : carte_ok:=t[2].est+t[3].ouest=0;
   4 : carte_ok:=t[1].sud+t[4].nord=0;
   5 : carte_ok:=(t[2].sud+t[5].nord=0) and (t[4].est+t[5].ouest=0);
   6 : carte_ok:=(t[3].sud+t[6].nord=0) and (t[5].est+t[6].ouest=0);
   7 : carte_ok:=(t[4].sud+t[7].nord=0);
   8 : carte_ok:=(t[5].sud+t[8].nord=0) and (t[7].est+t[8].ouest=0);
   9 : carte_ok:=(t[6].sud+t[9].nord=0) and (t[8].est+t[9].ouest=0);
```

```
   end;
end;

procedure init_carte(var c : carte;nord,est,sud,ouest : integer);
begin
c.nord:=nord;c.est:=est;c.sud:=sud;c.ouest:=ouest;
end;

procedure initialisation;
begin
num:=0;
st:='123456789';
init_carte(t[1],CORPS1,TETE4,CORPS2,TETE3 );
init_carte(t[2],TETE2,TETE4,TETE1,TETE3 );
init_carte(t[3],CORPS3,TETE4,CORPS2,TETE1 );
init_carte(t[4],CORPS1,TETE4,TETE1,TETE3 );
init_carte(t[5],CORPS3,TETE2,TETE1,CORPS4);
init_carte(t[6],TETE2,TETE4,TETE1,TETE3 );
init_carte(t[7],CORPS1,CORPS3,CORPS2,CORPS4);
init_carte(t[8],TETE4,CORPS3,TETE1,CORPS2);
init_carte(t[9],TETE2,CORPS3,CORPS2,CORPS4);
end;

procedure rotation_droite(var c : carte);
var b : integer;
begin
b:=c.nord;
c.nord:=c.ouest;c.ouest:=c.sud;c.sud:=c.est;c.est:=b;
end;

procedure rotations(numero : integer);
var i : integer;
begin
if numero=10 then
   begin
   Dessine(Form1.Image1.Canvas,0,0,Form1.Image1.Width div 3);
   Screen.Cursor:=crDefault;
   inc(num);
   Form1.Label1.Caption:='Solution '+IntToStr(num);
   showmessage('Solution '+IntToStr(num)+' - Cartes : '+st);
   Screen.Cursor:=crHourGlass;
   end
else
   for i:=1 to 4 do
       begin
       rotation_droite(t[numero]);
       if carte_ok(numero) then rotations(numero+1);
       end;
end;

procedure anagram(i : byte);
var j : byte;
begin
if i = 9 then rotations(1)
else
   for j := i to 9 do
       begin
       car:=t[i];t[i]:=t[j];t[j]:=car;
       ch:=st[i];st[i]:=st[j];st[j]:=ch;
       anagram (i+1);
       car:=t[i];t[i]:=t[j];t[j]:=car;
       ch:=st[i];st[i]:=st[j];st[j]:=ch;
       end
end;
```

Chapitre IV-6

Le solitaire

Le jeu du solitaire est un jeu (de réflexion) qui se joue seul; le but du jeu est de ne rester qu'avec un seul pion sur l'échiquier; chaque pion peut sauter par dessus un autre pion à condition que la case d'arrivée soit vide; dans ce cas le pion par dessus lequel on a sauté disparaît. Voici l'échiquier du jeu :



Cet échiquier peut se représenter en mémoire par un tableau de 2 dimensions de 7 sur 7; il faut déjà choisir un échiquier de 9 sur 9, pour prévoir une case supplémentaire sur les bords, car si on saute vers l'extérieur, on peut tomber sur n'importe quelle valeur qui se trouve en mémoire; par contre si on a réservé une case, on tombe sur cette case qui aura une valeur indiquant qu'on ne peut pas y sauter. Cette technique sera aussi utilisée dans le jeu d'othello.

Etudions en quelques lignes ce programme :

- on va avoir un échiquier de 9 sur 9, mais en une dimension, donc un tableau linéaire de 81 cases; ceci est fait par souci de rapidité

- pour chaque pion (ou presque) on a quatre directions possibles pour sauter

- on peut sauter si la case jointe est occupée par un pion ET si la case suivante est vide

- une fois qu'on a joué un pion, on fait un appel récursif et on commence la recherche du prochain pion à jouer à partir de la première case

- nous allons créer plusieurs procédures :

  - l'initialisation de l'échiquier (procédure init_tab;)

  - l'affichage de l'échiquier (procédure aff_tab;)

  - une fonction pour déterminer quel est le pion qui doit jouer

- la procédure représentant le coeur du programme, la récursivité

Nous allons étudier en détail deux procédures :

- la fonction ***pion_qui_doit_jouer*** : elle a deux paramètres :

- *dernier_pion*, qui indique quel est le dernier pion qui a sauté

- *last_dir*, qui indique quelle est la dernière direction dans la quelle le dernier pion a sauté; si le dernier pion peut sauter dans plusieurs directions et il ne les a pas toutes essayées, alors il les essaie jusqu'au bout, sinon un autre pion est choisi

- pour déterminer quel pion doit jouer, un balayage de l'échiquier a lieu et ce jusqu'à le trouver ou jusqu'à la fin de l'échiquier, à savoir la case numéro 68 dans le tableau, qui correspond au dernier pion de l'échiquier pouvant sauter.

- la procédure "**joue**",qui représente le programme :

- elle n'a pas de paramètres car il faut essayer d'éviter de perdre du temps avec l'empilement et le désempilement de variables pendant l'exécution, vu le nombre élevé de possibilités

- le point d'arrêt est atteint quand on a fait 32 coups, dans ce cas on initialise la variable "trouve" à true et ensuite on ne cherche plus d'autre solution, mais on affiche les sauts et on sort.

- on simule chaque coup par des remplissages du tableau avec des PION et avec des "VIDE", on rappelle la procédure, et ensuite on annule le coup pour essayer une autre possibilité.

Le programme a été testé sur un K6-II 350 et il a calculé 7,4 millions de coups avant de trouver la première solution. Nous avons choisi de l'arrêter délibérément après la première solution, mais vous pouvez le laisser continuer à rechercher d'autres solutions en vous inspirant du programme concernant le saut du cavalier, où les coups sont sauvegardés dans des fichiers distincts.

Voici enfin le programme :

```
const INCONNU = 5;
      PION    = 3;
      VIDE    = 0;

      d  : array[0..3] of integer = (-9, 1, 9, -1);

type tab      = array[0..80] of byte;
     tab_n    = array[1..32] of tab;

var  t           : tab;
     plateaux    : tab_n;
     numero_coup : integer;
     nb_coups    : longint;
     trouve      : boolean;

procedure init_tab;
var x,y : integer;
begin
for x:=0 to 8 do
for y:=0 to 8 do
    t[x*9+y]:=INCONNU;
```

```
for y:=1 to 7 do
for x:=3 to 5 do
    begin
    t[y*9+x]:=PION;
    t[x*9+y]:=PION;
    end;
t[4*9+4]:=VIDE;
end;

function pion_qui_doit_jouer(dernier_pion : integer;
    var last_dir : integer) : integer;
var i,x,y,k1,k2 : integer;
begin
x:=dernier_pion mod 10 -1;
y:=dernier_pion div 10;
while (y*10+x<78) do
   begin
   inc(x);
   if (x=8) then begin x:=0; inc(y);end;
   for i:=last_dir to 3 do
       begin
       k1:=y*9+x;k2:=d[i];
       if (t[k1]=PION) then
       if (t[k1+k2]=PION) then
       if (t[k1+2*k2]=VIDE) then
       begin
          last_dir:=i;
          pion_qui_doit_jouer:=y*10+x;
          exit;
          end;
       end;
   last_dir:=0;
   end;
pion_qui_doit_jouer:=0;
end;

procedure joue(numero_coup : integer;var trouve : boolean);
var i,x,y,k1,k2,pion_qui_joue,dernier_pion_qui_a_joue,last_dir,k : integer;
begin
if (numero_coup=32) then
   begin
   move(t,plateaux[numero_coup],sizeof(tab));
   trouve:=true;
   if Form1.CheckBox1.Checked then Printer.BeginDoc;   //Si on veut imprimer
   for k:=1 to 32 do afficher_coup(k);
   if Form1.CheckBox1.Checked then Printer.EndDoc;   //Si on veut imprimer
   exit;
   end;
last_dir:=0;
dernier_pion_qui_a_joue:=0;
inc(nb_coups);
if nb_coups mod 100000=0 then
   begin form1.edit1.text:=inttostr(nb_coups);form1.edit1.refresh;end;
repeat
   pion_qui_joue:=pion_qui_doit_jouer(dernier_pion_qui_a_joue,last_dir);
   dernier_pion_qui_a_joue:=pion_qui_joue;
   if (dernier_pion_qui_a_joue>0) then
      begin
      move(t,plateaux[numero_coup],sizeof(tab));
      i:=last_dir;
      inc(last_dir);
      y:=pion_qui_joue div 10;
      x:=pion_qui_joue mod 10;
      k1:=y*9+x;k2:=d[i];
      t[k1+2*k2]:=PION;
      t[k1+k2]:=VIDE;
      t[k1]:=VIDE;
      joue(numero_coup+1,trouve);
      move(plateaux[numero_coup],t,sizeof(tab));
// l'instruction suivante disparaîtra quand on décidera de chercher toutes les
solutions
      if trouve then exit;
      end;
until (pion_qui_joue=0);
end;
```

Chapitre IV-7

Le compte est bon

Un problème bien connu par tout le monde est le jeu télévisé "Le compte est bon". Rappelons le principe du jeu pour ceux qui ne le connaissent pas: on tire 6 nombres au hasard parmi les suivants "1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100" et un autre nombre entre 101 et 999. Il faut trouver le dernier nombre en faisant des opérations mathématiques sur les 6 premiers nombres en ne les utilisant qu'une fois. On reste dans le domaine des entiers naturels (donc positifs). On dispose de quatre opérations : " +, -, *, / ".

Supposons qu'on a seulement deux nombres; les opérations qu'on peut faire sur ces deux nombres sont les suivantes :

"n1 + n2" "n2 + n1"

"n1 - n2" "n2 - n1"

"n1 * n2" "n2 * n1"

"n1 / n2" si n1 est un multiple de n2 "n2 / n1" si n2 est un multiple de n1

On peut tout de suite diviser par 2 le nombre des opérations possibles. On se ramène pour cela au cas où n1 >= n2 :

"n1 + n2" est équivalent à "n2 + n1"

"n2 - n1" n'a pas de sens car cela donnerait un nombre négatif, donc il suffit de considérer "n1-n2"

"n1 * n2" est équivalent à "n2 * n1"

"n1 / n2" n'a de sens que si "n1 >= n2" et si n1 est un multiple de n2

Supposons maintenant qu'on a 3 nombres, par exemple 3, 9, 8. D'après ce qui précède, il faut que les nombres soient triés par ordre décroissant. Pour cela, nous allons les mettre dans un tableau, et on aura : 9, 8, 3. Nous allons travailler tout le long du programme avec seulement des paires de 2 nombres. Ici nous avons trois paires possibles :"9,8", "9,3", "8,3". Pour chacune de ces paires, on fait les quatre opérations et on obtient de nouveaux nombres qu'il faudra combiner avec celui qui reste. Pour mieux comprendre, nous allons détailler les opérations pour ce cas. Il ne faut pas oublier que les nombres sont rangés dans un tableau et que nous allons utiliser ce tableau :

```
Etude de la première paire :
9 + 8 = 17                                                    ¦ 1
    On a un nouveau tableau trié qui contient 2 nombres : 17, 3    ¦
    17 + 3 = 20                                               ¦ 2
    17 - 3 = 14                                               ¦ 3
    17 * 3 = 51                                               ¦ 4
    Division impossible.                                      ¦
9 - 8 = 1                                                     ¦ 5
```

```
    On a un nouveau tableau trié qui contient 2 nombres : 3, 1     ¦
    3 + 1 = 4                                                      ¦ 6
    3 - 1 = 2                                                      ¦ 7
    3 * 1 = 3                                                      ¦ 8
    3 / 1 = 3                                                      ¦ 9
9 * 8 = 72                                                         ¦ 10
    On a un nouveau tableau trié qui contient 2 nombres : 72, 3    ¦
    72 + 3 = 75                                                    ¦ 11
    72 - 3 = 69                                                    ¦ 12
    72 * 3 = 216                                                   ¦ 13
    72 / 3 = 24                                                    ¦ 14
Division impossible.                                               ¦
                                                                   ¦
Etude de la deuxième paire :                                       ¦
9 + 3 = 12                                                         ¦ 15
    On a un nouveau tableau trié qui contient 2 nombres : 12, 8    ¦
    12 + 8 = 20                                                    ¦ 16
    12 - 8 = 4                                                     ¦ 17
    12 * 8 = 96                                                    ¦ 18
    Division impossible.                                           ¦
9 - 3 = 6                                                          ¦ 19
    On a un nouveau tableau trié qui contient 2 nombres : 8, 6     ¦
    8 + 6 = 14                                                     ¦ 20
    8 - 6 = 2                                                      ¦ 21
    8 * 6 = 48                                                     ¦ 22
    Division impossible.                                           ¦
9 * 3 = 27                                                         ¦ 23
    On a un nouveau tableau trié qui contient 2 nombres : 27, 8    ¦
    27 + 8 = 35                                                    ¦ 24
    27 - 8 = 19                                                    ¦ 25
    27 * 8 = 216                                                   ¦ 26
    Division impossible.                                           ¦
9 / 3 = 3                                                          ¦ 27
    On a un nouveau tableau trié qui contient 2 nombres : 8, 3     ¦
    8 + 3 = 11                                                     ¦ 28
    8 - 3 = 5                                                      ¦ 29
    8 * 3 = 24                                                     ¦ 30
    Division impossible.                                           ¦
                                                                   ¦
Etude de la troisième paire :                                      ¦
8 + 3 = 11                                                         ¦ 31
    On a un nouveau tableau trié qui contient 2 nombres : 11, 9    ¦
    11 + 9 = 20                                                    ¦ 32
    11 - 9 = 2                                                     ¦ 33
    11 * 9 = 99                                                    ¦ 34
    Division impossible.                                           ¦
8 - 3 = 5                                                          ¦ 35
    On a un nouveau tableau trié qui contient 2 nombres : 9, 5     ¦
    9 + 5 = 14                                                     ¦ 36
    9 - 5 = 4                                                      ¦ 37
    9 * 5 = 45                                                     ¦ 38
    Division impossible.                                           ¦
8 * 3 = 24                                                         ¦ 39
    On a un nouveau tableau trié qui contient 2 nombres : 24, 9    ¦
    24 + 9 = 33                                                    ¦ 40
    24 - 9 = 15                                                    ¦ 41
    24 * 9 = 216                                                   ¦ 42
    Division impossible.                                           ¦
Division impossible.                                               ¦
```

On peut donc en déduire l'algorithme général du programme :

1 - on fait un algorithme qui sélectionne une paire

2 - on fait une opération sur cette paire

3 - si on a trouvé le nombre voulu, on quitte l'algorithme

4 - on sauvegarde le tableau courant dans un tableau auxiliaire

5 - avec le nombre obtenu après l'opération on obtient un tableau auxiliaire plus petit d'un élément

6 - on trie le tableau auxiliaire

7 - on fait un appel récursif pour recommencer le raisonnement sur ce tableau

8 - si on n'a pas fini les quatre opérations, on boucle sur l'étape 2

9 - si on n'a pas fini toutes les paires, on boucle sur l'étape 1

Voyons maintenant quel est le temps d'exécution.

Si toutes les divisions étaient possibles, alors on aurait 60 opérations pour ces 3 nombres, ou trois paires, ce qui n'est pas énorme.

Pour 4 nombres n1, n2, n3, n4 on aurait 6 paires : n1 n2, n1 n3, n1 n4, n2 n3, n2 n4, n3 n4 et, pour chacune de ces paires, on obtiendrait trois sous-paires : par exemple "n1 n2" forme un nombre qui sera combiné avec n3 et avec n4 (c'est le cas ci-dessus, donc 60 combinaisons). Il faut envisager les quatre signes pouvant apparaître entre n1 et n2 donc, pour les trois nombres formés par "n1 n2", "n3", "n4", on a en tout 4*60=240 combinaisons possibles; comme on a six paires à 240 possibilités chacune, on a en tout 1440 possibilités; la croissance est énorme mais en réalité elle est bien inférieure à cause des divisions impossibles et aussi à cause de certaines soustractions, comme nous le verrons par la suite; donc il y a beaucoup de branches de l'arbre récursif qui ne sont pas exploitées.

En faisant le même raisonnement pour 5 nombres on aurait 10 paires à 4 possibilités chacune et à 1440 sous-possibilités, soit 57600 possibilités. Et enfin pour 6 nombres on a 15 paires à 4 possibilités chacune et à 57600 sous-possibilités chacune donc en tout 3.456.000.

En réalité, grâce aux divisions impossibles et grâce aux soustractions qui donnent un reste nul (si on a deux fois 25, alors on a 25 - 25 = 0, inutile de continuer à explorer dans cette direction) on n'explore que 60% des possibilités, c'est à dire environ 2 millions de possibilités.

Pour réduire le temps de réflexion, il faut faire le programme d'une manière différente, car ici on a beaucoup d'opérations qui se répètent comme :

- les opérations 2, 16 et 32, car a+b+c=a+c+b=b+c+a

- les opérations 13, 26 et 42 car a*b*c=a*c*b=b*c*a

Il faut aussi tenir compte de la façon de programmer : étant donné le nombre élevé de combinaisons il faut avoir le moins de procédures possibles, et avec le plus petit nombre de paramètres possible, car pendant la récursivité, le fait d'empiler les variables locales et les paramètres demande beaucoup de temps.

Dans le programme, nous avons dans le tableau "nombres", à l'indice 0, le nombre à trouver (ici 963) et ensuite tous les nombres tirés au hasard. C'est ce tableau qu'il vous faudra modifier pour trouver une solution à un autre problème.

Voici le programme :

```
type tab           = array[0..6] of longint;

const signe        : string[4]  = '+-*/';
```

```
      nombres      : tab        = (963,25,5,4,3,3,1);

var trouve,echange : boolean;
    aa             : longint;
    ii,jj          : integer;

procedure operations(var t : tab;max : integer);
var i,j1,j2 : integer;
    a       : longint;
    t1      : tab;
begin
for i:=1 to 4 do
    for j1:=1 to max-1 do
        for j2:=j1+1 to max do
            begin
            case i of
               1 : a:=t[j1]+t[j2];
               2 : a:=t[j1]-t[j2];
               3 : a:=t[j1]*t[j2];
               4 : begin
                   a:=t[j1] div t[j2];
                   if t[j2]*a<>t[j1] then a:=0;
                   end;
               end;
            if a>0 then
               begin
               if a=t[0] then
                  begin
                  //gotoxy(1,8-max);write(t[j1],signe[i],t[j2],'=',a);

Form1.ListBox1.Items.Add(inttostr(t[j1])+signe[i]+inttostr(t[j2])+'='+inttostr(a));
                  trouve:=true;exit;
                  end;
               move(t,t1,28);
               t1[j1]:=a;t1[j2]:=0;
               repeat
                 echange:=false;
                 for ii:=1 to max-1 do
                     if t1[ii]<t1[ii+1] then
                        begin
                        aa:=t1[ii];t1[ii]:=t1[ii+1];t1[ii+1]:=aa;
                        echange:=true;
                        end;
               until not echange;
               operations(t1,max-1);
               if trouve then
                  begin
                  //gotoxy(1,8-max);write(t[j1],signe[i],t[j2],'=',a);

Form1.ListBox1.Items.Add(inttostr(t[j1])+signe[i]+inttostr(t[j2])+'='+inttostr(a));
                  exit;
                  end;
               end;
            end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
     Screen.Cursor:=crHourGlass;
     trouve:=false;
     ListBox1.Clear;
     Application.ProcessMessages;
     operations(nombres,6);
     Screen.Cursor:=crDefault;
end;
```

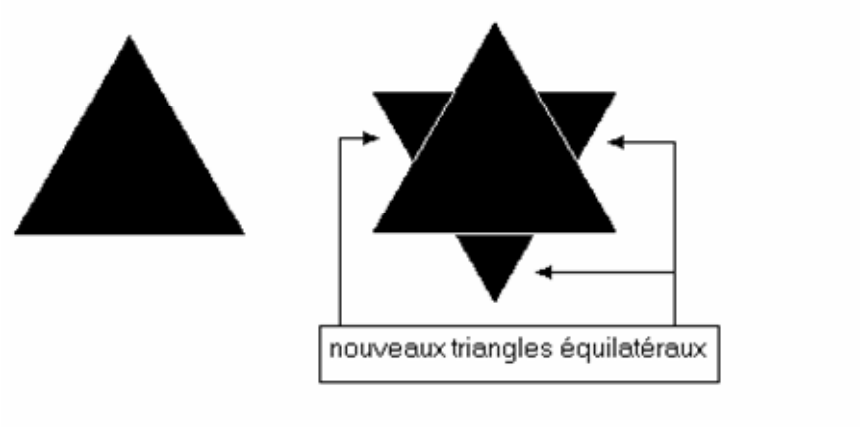
Chapitre IV-8

Génération récursive de courbes fractales

Les courbes fractales sont des courbes possédant la propriété d'autosimilarité : chacune des parties de la courbe est semblable à n'importe quelle autre. Qu'on s'approche aussi près que l'on veut de la courbe, et on verra toujours la même chose ! Par exemple, le pourtour d'une côte, les arbres et les nuages ont des structures fractales qui peuvent être modélisées mathématiquement.

Nous allons prendre pour exemple la très célèbre courbe en flocon de neige de Von Koch (1904).

Cette courbe est obtenue en partant d'un triangle équilatéral. Sur chaque côté, on construit à l'extérieur du triangle, sur le tiers central, un nouveau triangle équilatéral. En poursuivant à l'infini, on obtient la courbe de Von Koch. Cette courbe n'est pas rectifiable (sa longueur est infinie), elle est continue (elle ne comporte aucun trou) et elle n'est différentiable presque nulle part (il est impossible de construire une tangente en presque chacun de ses points - en mathématiques, on dit qu'un ensemble E possède une propriété P *presque partout* si on a P pour tous les points de E sauf pour un sous-ensemble dénombrable de points de E -).



Algorithme du programme « Flocon de Von Koch »

Variables globales :

Ecrx, Ecry : Largeur et hauteur de l'écran

Le tracé est réalisé par deux procédures, la principale étant la procédure Koch.

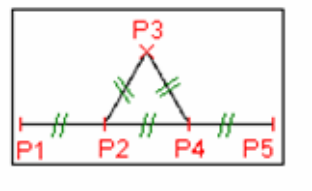
Définition de la procédure récursive « segment »

Ses paramètres sont x1,y1,x5,y5, c'est-à-dire les cordonnées des extrémités P1 et P5 du "segment" à tracer.

Variables locales :

l : longueur du segment P1P5

x2,x3,x4,y2,y3,y4 : abscisses et ordonnées des points P2, P3, P4.



Calculer la longueur de P1P5 et la stocker dans l

Si la longueur est inférieure à 2 pixels alors tracer le segment P1P5 (en informatique, rien n'est infini !)

Sinon :

Calculer les coordonnées des points P2 et P4, respectivement barycentres de {(P1,2)(P5,1)} et de {(P1,1)(P5,2)}.

On utilise la formule des barycentres pour couper un segment en trois. Un exemple concret de barycentres est le suivant : on a une poutre avec un poids de 2 kilos à un bout, et un poids de 1 kilo à l'autre bout. La formule du barycentre nous donne le point exact de la poutre où celle-ci est en équilibre.

Dans le programme, comme on l'a dit au début, on dessine sur chaque côté un nouveau triangle équilatéral; mais ce nouveau triangle sera incliné; les triangles qui seront construits sur ses côtés seront aussi inclinés, mais auront un autre angle d'inclinaison, donc nous sommes obligés d'utiliser les formules de rotation dans le plan.

Calculer les coordonnées du point P3, image de P4 par la rotation d'angle 60° et de centre P2

*(on montre que x3=x2/2-cos30*y2+x4/2+cos30*y4 et que y3=cos30*x2+y2/2-cos30*x4+y4/2)*

Appeler la procédure segment avec les paramètres (x1,y1,x2,y2),

Appeler la procédure segment avec les paramètres (x2,y2,x3,y3),

Appeler la procédure segment avec les paramètres (x3,y3,x4,y4),

Appeler la procédure segment avec les paramètres (x4,y4,x5,y5).

Définition de la procédure Koch

Sans paramètres

Variables locales :

x0, y0 : coordonnées du point de départ du tracé taille : longueur des côtés du triangle initial

Sélectionner une couleur (blanc)

Définir la taille (on prend la moitié de la largeur de l'écran)

Définir x0 pour que le flocon soit centré horizontalement (largeur de l'écran/4)

Définir y0 pour que le flocon soit centré verticalement

Appeler la procédure segment avec les paramètres :

x0, y0, x0+taille, y0

x0+taille, y0, x0+taille/2, y0+cos30*taille

x0+taille/2, y0+cos30*taille, x0, y0

Génération itérative de courbes fractales

Nous allons donner un exemple bien connu de dessin fractal, celui de la courbe de Mandelbrot. Ce dessin est réalisé de façon itérative avec une boucle, à cause de la simplicité de la fonction utilisée; par contre, le dessin du flocon de Von Koch (1904) est récursif.

La courbe de Mandelbrot utilise les nombres complexes, donc le dessin s'effectue dans le plan complexe.

Le programme affiche un dessin qui est l'ensemble des points du plan complexe d'affixe z tels que la suite (|z(n)|) pour n entier naturel, soit convergente, avec : $z(0)=z0$ et $z(n+1)=[z(n)]^2+z0$.

Vous pouvez initialiser le paramètre "seuil de divergence" à 10, et changer le progressivement de 10 en 10 par exemple. Cela vous permettra de suivre l'évolution de la courbe et des couleurs ("déplacement" progressif).

Vous pouvez aussi changer la variable "profondeur de calcul". Une initialisation à 10 suivi d'une augmentation progressive de 10 en 10 vous permettra de voir la courbe se dessiner à l'écran d'une façon de plus en plus précise.

```
procedure Dessine;
var large,haut,iimag,ireal,prof:integer; dreal,dimag,lreal,limag,re,im,xx,yy:real;
begin
  large:=Form1.Image1.Width;
  haut:=Form1.Image1.Height;
  Form1.Image1.Canvas.Brush.Style:=bsSolid;
  Form1.Image1.Canvas.Brush.Color:=clWhite;
  Form1.Image1.Canvas.Rectangle(0,0,large,haut);
  dreal:=(re_max-re_min)/large;
  dimag:=(im_max-im_min)/haut;
  FOR iimag:= 0 TO haut-1 do
    FOR ireal:= 0 TO large -1 do begin
      lreal:=re_min+ireal*dreal;
      limag:=im_min+iimag*dimag;
      re:=lreal;
      im:=limag;
      prof:=0;
      REPEAT
        xx:=re*re;
        yy:=im*im;
        im:=2*re*im-limag;
        re:=xx-yy-lreal;
        inc(prof);
      UNTIL (prof=calprof) OR (xx+yy>infini);
```

```
      IF prof<calprof then begin
        IF prof>desprof then Form1.Image1.Canvas.Pixels[ireal,iimag]:=color(prof mod
19)
        ELSE IF ((prof MOD 2)=0) then
Form1.Image1.Canvas.Pixels[ireal,iimag]:=color(prof mod 19);
      end;
    end; //next ireal
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
     infini:=StrToFloat(Edit5.Text);  // le nombre que l'on considère comme "infini"
(seuil de divergence)
     calprof:=StrToInt(Edit6.Text);   // profondeur de calcul (nombre d'itérations
pour tester la convergence)
     desprof:=StrToInt(Edit7.Text);   // profondeur de dessin (nb de couleurs
utilisées)
     re_min:=StrToFloat(Edit1.Text);  // x min
     re_max:=StrToFloat(Edit2.Text);  // x max
     Im_min:=StrToFloat(Edit3.Text);  // y min
     Im_max:=StrToFloat(Edit4.Text);  // y max
     if infini<=0 then infini:=10;
     if calprof<=0 then calprof:=20;
     if desprof<=0 then desprof:=10;
     if re_min>re_max then
        begin
        x:=re_min;
        re_min:=re_max;
        re_max:=x;
        end;
     if Im_min>Im_max then
        begin
        x:=Im_min;
        Im_min:=Im_max;
        Im_max:=x;
        end;
     Screen.Cursor:=crHourGlass;
     Dessine;
     Screen.Cursor:=crDefault;
end;
```

Chapitre IV-9

Quick sort

Il s'agit d'une méthode de tri récursive dont l'efficacité est une fonction croissante du désordre dans le tableau à trier, c'est à dire que plus le tableau est désordonné, plus cette méthode de tri est efficace.

Algorithme du programme QuickSort

Variables globales :

t : le tableau à trier.

Pour l'exemple, nous prendrons t=(3,5,2,4,6,1,4,10), tableau à 8 éléments.

Paramètres :

debut : l'indice du premier élément du tableau à trier,

fin : l'indice du dernier élément du tableau à trier.

Variables locales :

pivot : l'indice de l'élément qui sert de pivot, comme nous allons l'expliquer plus loin,

gauche : l'indice de l'élément en cours de comparaison avec le pivot, situé avant le pivot,

droite : l'indice de l'élément en cours de comparaison avec le pivot, situé après le pivot,

temp : variable tampon juste pour faire l'échange de deux éléments du tableau.

Principe

On appelle la procédure QuickSort en lui passant en paramètres les indices du premier et du dernier élément du tableau à trier : *QuickSort(1,8)*.

On choisit un élément du tableau au hasard. Ce peut être n'importe lequel. Ici, nous choisissons le premier élément du tableau : 3. L'indice de cet élément est appelé le *pivot* (ici, c'est 1).

On initialise *gauche* à *début* (ici, 1) et *droite* à *fin* (ici, 8).

La première partie du travail consiste à ranger le tableau de telle sorte que tous les éléments inférieurs à l'élément d'indice *pivot* se trouvent placés à la gauche de celui-ci (et donc tous les éléments supérieurs à sa droite).

Ceci se fait par comparaisons et échanges successifs :

```
repeat
          if t[gauche]>=t[droite] then
             begin  //échanger t[droite] et t[gauche]
                    temp:=t[gauche];
                    t[gauche]:=t[droite];
                    t[droite]:=temp;
                    pivot:=gauche+droite-pivot; //nouvelle position du pivot
             end;
          if pivot=gauche then dec(droite) else inc(gauche);
     until gauche=droite;
```

Nous comparons d'abord le premier élément du tableau (3) avec le dernier (10).

Comme 3<10, nous ne faisons rien (donc on a encore *pivot=gauche*) et nous comparons 3 avec l'avant-dernier élément (*dec(droite)*)du tableau (4).

Comme 3<4, nous ne faisons rien (donc on a encore *pivot=gauche*) et nous comparons 3 avec l'élément d'indice 6 (*dec(droite)*)du tableau (1).

Comme 3>1, nous échangeons les deux éléments afin que 1 se retrouve à gauche de 3 :

```
3  5      2      4      6      1      4      10

1  5      2      4      6      3      4      10
```

Comme *pivot* désigne maintenant un nouvel emplacement (3 est maintenant en sixième position), nous notons celui-ci : *pivot:=gauche+droite-pivot;*. Ceci est un raccourci élégant pour *if pivot=gauche then pivot:=droite else pivot:=gauche;*. En effet, *pivot* est alors égal à *droite* ou à *gauche* car *pivot* était égal avant à *gauche* ou à *droite*.

pivot étant maintenant égal à *droite*, nous augmentons *gauche* d'une unité (*inc(gauche)*) car 1 est désormais placé à gauche de 3 et nous n'avons plus à nous en occuper. A ce stade, *gauche=2, pivot=droite=6*. Comme *gauche* est différent de *droite*, nous continuons.

Comme 5>3, nous échangeons les deux éléments afin que 5 se retrouve à droite de 3 :

```
1  5      2      4      6      3      4      10

1  3      2      4      6      5      4      10
```

Comme *pivot* désigne maintenant un nouvel emplacement (3 est maintenant en deuxième position), nous notons celui-ci : *pivot:=gauche+droite-pivot=gauche=2;*.

pivot étant maintenant égal à *gauche*, nous diminuons *droite* d'une unité (*dec(droite)*) car 5 est désormais placé à droite de 3 et nous n'avons plus à nous en occuper. A ce stade, *gauche=pivot=2, droite=5*. Comme *gauche* est différent de *droite*, nous continuons.

Comme 3<6, nous ne faisons rien (donc on a encore *pivot=gauche*) et nous comparons 3 avec l'élément d'indice 4 (*dec(droite)*)du tableau (4).

Comme 3<4, nous ne faisons rien (donc on a encore *pivot=gauche*) et nous comparons 3 avec l'élément d'indice 3 (*dec(droite)*)du tableau (2).

Comme 3>2, nous échangeons les deux éléments afin que 2 se retrouve à gauche de 3 :

```
1  3       2       4       6       5       4       10
```

```
1  2       3       4       6       5       4       10
```

Comme *pivot* désigne maintenant un nouvel emplacement (3 est maintenant en troisième position), nous notons celui-ci : *pivot:=gauche+droite-pivot=droite=3;*.

pivot étant maintenant égal à *droite*, nous augmentons *gauche* d'une unité (*inc(gauche)*) car 2 est désormais placé à gauche de 3 et nous n'avons plus à nous en occuper. A ce stade, *gauche=3, pivot=droite=3*. Comme *gauche* est égal à *droite*, nous sortons de la boucle repeat/until et nous avons terminé la première phase du travail.

Le tableau est maintenant divisé en deux parties par l'élément d'indice pivot (3) :

```
1  2
```

et :

```
4  6       5       4       10
```

Les éléments de la partie gauche sont tous inférieurs à 3 et ceux de la partie droite lui sont tous supérieurs. C'est le but que nous nous étions fixé.

Il ne reste plus qu'à laisser opérer la magie de la récursivité, c'est à dire à appeler à nouveau (récursivement) notre procédure *QuickSort* pour chacun des deux sous-tableaux !

Comme *debut<gauche-1* (1<3-1), c'est ce que nous nous empressons de faire avec *QuickSort(1,2)*, et avec *QuickSort(4,8)*, puisque *fin>droite+1* (8>3+1) !

L'appel à *QuickSort* sur la partie gauche n'amènera pas de modification puisque le sous-tableau gauche est déjà trié.

L'appel à *QuickSort* sur la partie droite conduira à deux échanges pour placer le 6 avant le 10, qui suffiront à terminer le tri. En cours de chemin, il y aura deux sous-appels (inutiles !) à *QuickSort* qui n'amèneront donc pas de modifications.

Code de la procédure (récursive) QuickSort

```
procedure QuickSort(debut,fin:integer);
var pivot,gauche,droite,temp:integer;
begin
     pivot:=debut;
     gauche:=debut;
     droite:=fin;
     repeat
```

```
           if t[gauche]>=t[droite] then
              begin  //échanger t[droite] et t[gauche]
                     temp:=t[gauche];
                     t[gauche]:=t[droite];
                     t[droite]:=temp;
                     pivot:=gauche+droite-pivot; //nouvelle position du pivot
                     //pivot est alors égal à droite ou à gauche car pivot était égal
                     //avant à gauche ou à droite.
                     //c'est un raccourci élégant pour "if pivot=gauche then
pivot:=droite else pivot:=gauche;"
              end;
           if pivot=gauche then dec(droite) else inc(gauche);
     until gauche=droite;
     if debut<gauche-1 then QuickSort(debut,gauche-1); //appel récursif sur la partie
gauche
     if fin>droite+1 then QuickSort(droite+1,fin); //appel récursif sur la partie
droite
end;
```

Chapitre IV-10

Décompositions en sommes de carrés

Un petit problème intéressant concernant les nombres est la décomposition du carré d'un nombre en une somme de carrés de nombres différents tous plus petits que le premier.

exemple : soit le nombre 11. $11^2=121$. Il faut décomposer le nombre 121 en une somme de carrés de nombres distincts plus petits que 11. Plusieurs solutions se présentent à chaque fois, et plus le nombre initial est grand, plus il y a de solutions. Voici deux décompositions pour le carré de 11 (une autre solution vous sera donnée par l'algorithme) :

$121=100+16+4+1=10^2+4^2+2^2+1^2$

$121=81+36+4=9^2+6^2+2^2$

Nous allons d'abord étudier sur le cas concret présenté ci-dessus l'approche à effectuer.

Notre nombre est 11. On doit donc prendre un par un tous les nombres inférieurs à 11, soustraire leur carré de 121, extraire la racine carrée (éventuellement majorée si le résultat n'est pas entier) du résultat et recommencer.

exemple:

On doit trouver 121 avec la somme des carrés de nombres plus petits que 11.

Les nombres plus petits que 11 sont 10,9,8,7,6,5,4,3,2,1. On doit tous les essayer, donc une boucle s'impose.

*a0) On prend 10 et on l'élève au carré : 10*10=100.*

b0) On soustrait : 121-100=21.

c0) On extrait la racine carrée de 21 et on obtient 4,6; on majore le résultat par 5.

appel récursif :

On doit maintenant trouver le résultat du b0) avec la somme des carrés de nombres plus petits que 5.

*a1) On prend 4 et on l'élève au carré: 4*4=16.*

b1) On soustrait : 21-16=5.

c1) On extrait la racine carrée de 5 et on obtient 2,2; on majore le résultat pour obtenir 3.

appel récursif :

On doit maintenant trouver le résultat du b1) avec la somme des carrés de nombres plus petits que 3.

*a2) On prend 2 et on l'élève au carré: 2*2=4.*

b2) On soustrait : 5-4=1.

c2) On extrait la racine carrée de 1 et on obtient 1; on ne majore pas le résultat car il est entier.

appel récursif :

On doit maintenant trouver le résultat du b2) avec la somme des carrés de nombres plus petits ou égaux à 1.

Ici on a "ou égaux à" car il n'y a pas eu de majoration.

*a3) On prend 1 et on l'élève au carré: 1*1=1*

b3) On soustrait: 1-1=0

Et c'est fini, donc on affiche la solution, et on remonte dans la récursion au niveau 2.

*a2) On prend 1 et on l'élève au carré: 1*1=1.*

b2) On soustrait: 5-1=4.

c2) On extrait la racine carrée de 4 et on obtient 2; on ne majore pas le résultat car il est entier.

Mais on voit que le nombre 2 obtenu est plus grand que le nombre 1 du a2). Donc on s'arrête.

*a1) On prend 3 et on l'élève au carré: 3*3=9.*

b1) On soustrait 21-9=12.

c1) On extrait la racine carrée de 12 et on obtient 3,5; on majore le résultat pour obtenir 4.

appel récursif :

On doit maintenant trouver le résultat du b1) avec la somme des carrés de nombres plus petits que 4.

*a2) On prend 3 et on l'élève au carré: 3*3=9.*

b2) On soustrait : 12-9=3.

c2) On extrait la racine carrée de 3 et on obtient 1,7; on majore le résultat par 2.

appel récursif :

On doit maintenant trouver le résultat du b) avec la somme des carrés de nombres plus petits ou égaux à 1.

... et ainsi de suite.

Nous devons donc créer une fonction qui extrait la racine carrée d'un nombre et qui la majore lorsque le résultat n'est pas entier.

```
function racine_majoree(carre : integer) : integer;
var a : integer;
begin
a:=round(sqrt(carre));
if a*a<carre then inc(a);
racine_majoree:=a;
end;
```

Nous devons utiliser ensuite un tableau d'entiers dans lequel nous allons accumuler les valeurs décroissantes dont la somme des carrés donne le nombre cherché. Nous allons ensuite les concaténer tous dans une chaîne, de façon à pouvoir les afficher.

```
t : array[0..50] of integer;
s : string;
```

On a enfin la procédure "cherche", qui a plusieurs paramètres, comme nous l'avons vu dans l'exemple commenté ci-dessus :

- *a_trouver* : le nombre à trouver qui est au début un carré, ensuite une différence de carrés. Ce nombre diminue au fur et à mesure qu'on descend dans la récursion.

- *nb* : le nombre à partir duquel on va chercher les carrés. Ainsi, si le nombre initial est 11, ce paramètre de la procédure sera 10. Il est calculé dans cette procédure avant chaque appel récursif.

- *niveau* : une variable qui sert d'indice pour le tableau dans lequel on accumule les nombres qui constitueront la solution finale

- *exacte* : une variable booléenne qui indique si la précédente racine carrée extraite était entière ou si elle avait été majorée. Si elle avait été majorée, alors on doit décrémenter *nb*.

Pendant la recherche et l'accumulation des nombres dans le tableau, on doit aussi vérifier que chaque nombre ajouté est plus petit que son prédécesseur, sinon on obtient n'importe quel nombre comme une somme de nombres 1 (qui est lui-même le carré de 1, donc le résultat est bien une somme de carrés !).

L'algorithme est le suivant :

```
procedure cherche(a_trouver,nb,niveau : integer; exacte : boolean);
var bool : boolean;
    rac,diff,carre,i,j : integer;
begin
if not exacte then dec(nb);
for i:=nb downto 1 do
    begin
    carre:=i*i;
    diff:=a_trouver-carre;
    if (diff=0) and (i<t[niveau-1]) then
       begin
       s:='';
       for j:=1 to niveau-1 do s:=s+inttostr(t[j]*t[j])+'+';
       s:=s+inttostr(carre)+'='+inttostr(nb1*nb1);
       memo1.lines.add(s);
       end
    else
       begin
       rac:=racine_majoree(diff);
       t[niveau]:=i;
```

```
            bool:=(rac*rac=diff);
            if (i<t[niveau-1]) then //--- on s'assure que les nombres sont distincts
                cherche(diff,rac,niveau+1,bool);
            end;
        end;
end;
```

Chapitre IV-11

Remplissage de volumes

On dispose d'un seau d'eau vide et de bouteilles de tailles différentes qui sont remplies d'eau. Quelle est la combinaison de bouteilles dont l'eau, une fois versée dans le seau, le remplit au maximum? Le problème peut être transposé à des chansons qui doivent remplir au mieux une cassette audio. Dans l'exemple suivant on a pris arbitrairement un seau dont le volume est de 1885 et 5 bouteilles dont les volumes sont dans le tableau t.

Une solution à ce problème est donnée en utilisant le programme qui faisait des anagrammes; il suffit de chercher la combinaison de bouteilles qui remplit au mieux un volume donné.

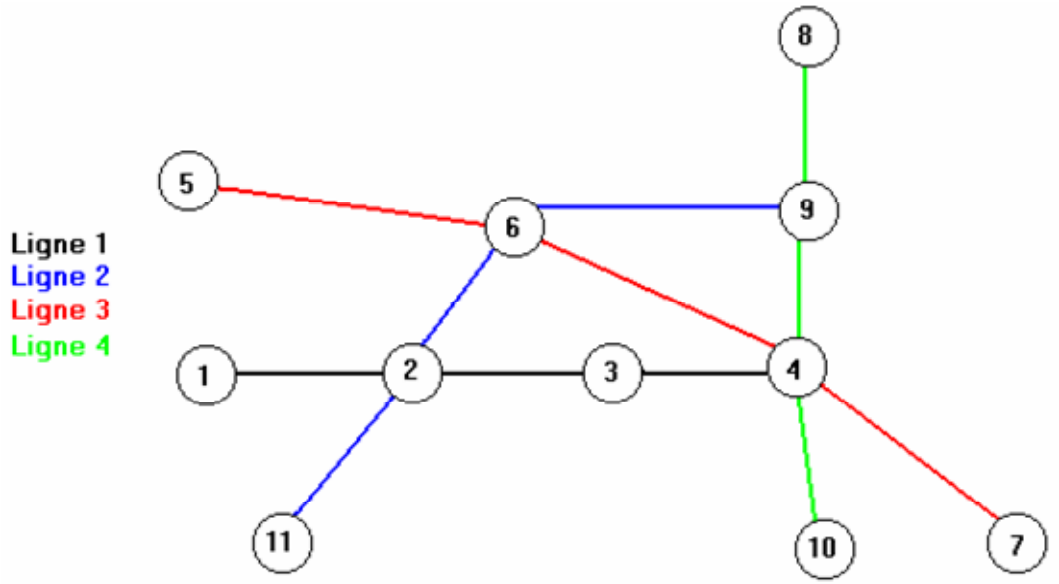
```
const t   : array[1..5] of integer = (250,370,451,749,634 );
      max : longint                = 1885;

var temoin     : string;
    difference : longint;
    nb_bouteilles : integer;

procedure anagrammes(st : string;nb1,nb2 : byte;result : integer);
var i : byte;
begin
for i:=nb1 to nb2 do
    if difference=0 then exit else
    if nb2<nb_bouteilles then anagrammes(st+chr(64+i),i+1,nb2+1,result+t[i])
              else
              if (max-result-t[i]<difference) and (max-result-t[i]>=0) then
                 begin
                 temoin:=st+chr(64+i);
                 difference:=max-result-t[i];
                 form1.memo1.lines.add('meilleure combinaison pour l''instant : '+
                      temoin+';reste à remplir:  '+inttostr(difference)+' litres');
                 end;
end;

procedure recherche;
var i : integer;
    s : string;
begin
nb_bouteilles:=high(t);
form1.memo1.lines.add('Vous disposez de '+inttostr(nb_bouteilles)+' bouteilles.');
form1.memo1.lines.add('Ces bouteilles sont numérotées avec des lettres.');
form1.memo1.lines.add('Leurs volumes respectifs sont :');
for i:=1 to nb_bouteilles do
    form1.memo1.lines.add(chr(64+i)+'='+inttostr(t[i])+' litres');
form1.memo1.lines.add('On doit remplir un volume de '+inttostr(max)+' litres.');
for i:=1 to nb_bouteilles do anagrammes('',1,i,0);
form1.memo1.lines.add('');
s:='bouteilles = '+temoin+'; partie remplie = '+inttostr(max-difference)+' litres';
form1.memo1.lines.add(s);
s:='reste à remplir = '+inttostr(difference)+' litres; maximum remplissable =
'+inttostr(max)+' litres';
form1.memo1.lines.add(s);
s:='pourcentage de remplissage : '+inttostr(100-round(difference*100/max))+' %';
form1.memo1.lines.add(s);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
difference:=max;temoin:='';
recherche;
end;
```

Chapitre IV-12

Le métro

Le programme suivant cherche le chemin le plus court pour aller d'une station de métro à une autre. Il fonctionne avec le métro parisien, mais il peut être bien évidemment être utilisé avec n'importe quel autre réseau.

Dans un premier temps, nous allons appliquer cet algorithme à un réseau très simple, avec quelques stations. C'est le programme qui suit. Le programme réel, avec toutes les stations du métro et les optimisations, se trouve sur CD.



Nous allons d'abord définir le type "arc", représentant le chemin entre deux stations; il contient le numéro de la station de départ, le numéro de la station d'arrivée et un indicateur qui nous dit si un arc a déjà été emprunté (afin d'éviter les boucles):

```
arc = record
      depart,arrivee : integer;
      parcouru : boolean;
      end;
```

Nous allons ensuite entrer les arcs correspondants:

```
procedure init_arcs;
begin
t[ 1].depart:= 1;t[ 1].arrivee:= 2;t[ 1].parcouru:=false;
t[ 2].depart:= 2;t[ 2].arrivee:= 1;t[ 2].parcouru:=false;
t[ 3].depart:= 2;t[ 3].arrivee:= 3;t[ 3].parcouru:=false;
t[ 4].depart:= 2;t[ 4].arrivee:= 6;t[ 4].parcouru:=false;
t[ 5].depart:= 2;t[ 5].arrivee:=11;t[ 5].parcouru:=false;
t[ 6].depart:= 3;t[ 6].arrivee:= 2;t[ 6].parcouru:=false;
t[ 7].depart:= 3;t[ 7].arrivee:= 4;t[ 7].parcouru:=false;
t[ 8].depart:= 4;t[ 8].arrivee:= 3;t[ 8].parcouru:=false;
t[ 9].depart:= 6;t[ 9].arrivee:= 4;t[ 9].parcouru:=false;
t[10].depart:= 4;t[10].arrivee:= 7;t[10].parcouru:=false;
t[11].depart:= 4;t[11].arrivee:= 9;t[11].parcouru:=false;
t[12].depart:= 4;t[12].arrivee:=10;t[12].parcouru:=false;
t[13].depart:= 5;t[13].arrivee:= 6;t[13].parcouru:=false;
t[14].depart:= 6;t[14].arrivee:= 2;t[14].parcouru:=false;
t[15].depart:= 6;t[15].arrivee:= 4;t[15].parcouru:=false;
t[16].depart:= 6;t[16].arrivee:= 5;t[16].parcouru:=false;
```

```
t[17].depart:= 6;t[17].arrivee:= 9;t[17].parcouru:=false;
t[18].depart:= 7;t[18].arrivee:= 4;t[18].parcouru:=false;
t[19].depart:= 8;t[19].arrivee:= 9;t[19].parcouru:=false;
t[20].depart:= 9;t[20].arrivee:= 4;t[20].parcouru:=false;
t[21].depart:= 9;t[21].arrivee:= 6;t[21].parcouru:=false;
t[22].depart:= 9;t[22].arrivee:= 8;t[22].parcouru:=false;
t[23].depart:=10;t[23].arrivee:= 4;t[23].parcouru:=false;
t[24].depart:=11;t[24].arrivee:= 2;t[24].parcouru:=false;
min_stations:=10000;  // initialisé à plus l'infini
end;
```

On doit faire ensuite une procédure qui pour un station donnée cherche toutes les stations qui sont à une distance de 1 de celle-ci. Ainsi, dans le dessin précédent, les stations situées à une distance de 1 de la station numéro 4 sont les suivantes: 3,6,9,7 et 10.

```
procedure trouve_partantes(depart : integer);
var i : integer;
begin
nb_partantes:=0;
for i:=1 to 24 do
    begin
    if t[i].depart=depart then
       begin
       inc(nb_partantes);
       partantes[nb_partantes]:=i;  // valeur de "i" utilisée dans "trouver"
       end;
    end;
end;
```

Ensuite, pendant le parcours du graphe (afin de chercher le chemin le plus court) si on a emprunté un arc, on doit le marquer; de cette façon, on empêche le programme de boucler. Inversement, si on a pris un chemin et qu'il s'est révélé infructueux, on doit faire demi-tour (en réalité on monte d'un niveau dans la pile de la procédure récursive) et marquer l'arc comme n'étant pas emprunté. Ce qui nous donne :

```
procedure marquer_arc(depart,arrivee : integer;marque : boolean);
var i : integer;
begin
for i:=1 to 23 do
    begin
    if (t[i].depart=depart) and (t[i].arrivee=arrivee) then
       t[i].parcouru:=marque;
    if (t[i].depart=arrivee) and (t[i].arrivee=depart) then
       t[i].parcouru:=marque;
    end;
end;
```

Il nous reste maintenant à faire la fonction de recherche du plus court chemin entre deux stations. Cette fonction a les paramètres suivants :

- départ et arrivée, indiquant les numéros de stations

- nb_stations: variable indiquant (à un niveau n de la récursivité) le nombre de stations parcourues.

- liste: une chaîne de caractères, dans laquelle on mémorise les numéros des stations.

La programmation de cette fonction se fait de la manière suivante :

- pour chaque chemin trouvé, on mémorise le nombre de stations dans une variable, appelée "min_stations". De cette façon, on est sûr que la longueur des chemins trouvés va aller en diminuant.

- si on n'a pas trouvé le chemin (c'est à dire "départ<>arrivée") alors pour la station en cours (c'est à dire "départ") on mémorise toutes les stations partantes, et on les essaye une par une. Grâce à la récursivité, pour

chacune de ces stations partantes on mémorise leur propres stations partantes et on les essaie (descente récursive et exploration de tous les chemins).

Il faut donc marquer l'arc emprunté, essayer la première station et si elle a échoué alors effacer l'arc emprunté et prendre la deuxième station; et ainsi de suite.

```
function trouver(depart,arrivee,nb_stations : integer;liste : string) : boolean;
var i : integer;
    partantes1    : tp;
    nb_partantes1 : integer;
begin
if depart=arrivee then
   begin
   trouver:=true;
   if nb_stations<=min_stations then
      begin
      min_stations:=nb_stations;
      form1.memo1.lines.add('Nouveau plus petit chemin :');
      form1.memo1.lines.add(inttostr(nb_stations)+' stations.');
      form1.memo1.lines.add(liste);
      form1.memo1.lines.add('============================');
      end;
   end
else
   begin
   trouve_partantes(depart);
   move(partantes,partantes1,sizeof(tp));
   nb_partantes1:=nb_partantes;
   for i:=1 to nb_partantes1 do
       if t[partantes1[i]].parcouru=false then
          begin
          marquer_arc(depart,t[partantes1[i]].arrivee,true);
          trouver:=trouver(t[partantes1[i]].arrivee,arrivee,
                      nb_stations+1,liste+','+inttostr(t[partantes1[i]].arrivee));
          marquer_arc(t[partantes1[i]].arrivee,depart,false);
          end;
   end;
end;
```

On n'a plus qu'à appeler la fonction ci-dessus :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
init_arcs;
trouver(5,  // station départ
        7,  // station arrivée
        0,  // stations parcourues
        '5');  // chemin parcouru
end;
```

Passons maintenant dans le cas réel, c'est à dire environ 300 stations de métro.

Là, prendre tous les chemins peut s'avérer extrêmement long. C'est pour cette raison qu'on introduit un nombre maximal de changements autorisés.

L'algorithme prend les chemins dans l'ordre de leur introduction en mémoire. Si on lui laisse un nombre maximal de 5 changements par exemple, alors le nombre de chemins possibles sera beaucoup plus grand que si on lui impose seulement 2 changements. C'est donc à vous de choisir et de faire des essais, pour déterminer le trajet et le temps de trajet qui vous convient le mieux.

Le programme fonctionnel des stations de métro se trouve sur le CD-ROM du livre.

On peut ensuite modifier l'algorithme de recherche en ne lui donnant que les stations qui se trouvent à un croisement, de cette façon la recherche sera encore plus rapide. Actuellement le programme met de 1 seconde à quelques minutes pour trouver un chemin optimal, mais l’affichage des solutions est progressif.

Chapitre IV-13

Ecrire un nombre en toutes lettres

Introduction

De nos jours, de nombreux programmes doivent convertir des sommes en lettres. Nous vous présentons ici un programme qui traduit un nombre entier (une suite de chiffres) quelconque en son expression littérale correspondante en français (exemple : 314 = « trois cent quatorze »). Le programme traite les nombres décimaux en traitant de façon distincte la partie entière et celle décimale, puis concatène ensuite les deux. Il peut être utilisé par exemple dans les logiciels qui impriment des chèques.

Nous allons découvrir quels sont les problèmes liés à la langue française qui peuvent apparaître dans ce programme.

Façon d'écrire un nombre en toutes lettres en français

Problèmes d'écriture

Nous allons étudier un peu les différents cas qui peuvent se présenter.

Unités :

Le chiffre des unités ne pose aucun problème, chaque chiffre a un correspondant littéral :

0= « zéro » , 1 = « un », 2= « deux » et ainsi de suite.

Dizaines :

Par contre, dès qu'on arrive aux dizaines, les choses se compliquent car la façon d'écrire et de prononcer les nombres varie selon le chiffre utilisé. Ainsi on dit « onze » au lieu de « dix et un » (par contre on dit « vingt et un ») , on dit « douze » au lieu de « dix-deux » (par contre on dit « vingt-deux ») et ainsi de suite jusqu'à seize. Mais à partir de là on dit bien « dix-sept » (de même que « vingt-sept »), « dix-huit », etc.

Le cas du mot « **et** » (conjonction de coordination)

Noter le fait qu'on rajoute le mot « et » pour le chiffre « 1 » (vingt ET un, trente ET un, quarante ET un, cinquante ET un, soixante ET un, soixante ET onze) mais pas pour les autres dizaines.

D'autres particularités concernent le soixante-dix, quatre-vingt, quatre-vingt-dix. Ces cas sont spécifiques au français utilisé en France, qui est différent de celui utilisé en Suisse, où il y a une façon bien plus simple de dire ces nombres : septante, huitante, nonante (et c'est très logique !) , ou en Belgique qui utilise « septante, octante, nonante »

On constate qu'on a trois cas où on utilise « onze » (11,71,91) et 7 cas où on utilise « un » (1,21,31,41,51,61,81).

En plus de cela, il y a les cas des mots « **vingt** » et « **quatre-vingt** »:

VINGT

1. L'adjectif numéral prend la marque du pluriel s'il est multiplié par un nombre et s'il n'est pas suivi d'un autre adjectif de nombre.

Exemple : Quatre-vingts feuilles, quatre-vingt-huit francs.

Attention aux mots million, milliard qui ne sont pas des adjectifs numéraux, mais des noms.

Exemple : Quatre-vingts millions.

Précédé de cent ou de mille, l'adjectif numéral est toujours invariable.

Exemple : Cent vingt personnes.

2. Dans les adjectifs numéraux composés, le trait d'union s'emploie seulement entre les éléments qui sont l'un et l'autre inférieurs à cent, sauf si ces éléments sont joints par la conjonction « et ».

Exemple : Vingt et un. Vingt-cinq.

QUATRE-VINGT(s)

adj. et n. m. · Adjectif numéral cardinal. Quatre fois vingt. Il a quatre-vingts ans, elle a quatre-vingt-deux ans. Note.- L'adjectif numéral cardinal s'écrit avec un « s » s'il est multiplié par un nombre et s'il n'est pas suivi d'un autre adjectif numéral. Adjectif numéral ordinal invariable.

Exemple : Quatre-vingtième. La page quatre-vingt. En mil neuf cent quatre-vingt.
· Nom masculin invariable.

Exemple : Des quatre-vingt en lettres lumineuses.

Notes

1. Attention aux nombres composés des mots million, milliard qui ne sont pas des adjectifs numéraux, mais des noms et qui permettent donc la marque du pluriel à vingt si l'adjectif numéral est multiplié par un nombre.

Exemple : Quatre-vingts millions de francs.

2. Après l'adjectif numéral, la conjonction « et » ne s'emploie pas devant « un » , contrairement à « trente et un », « quarante et un »...

Exemple : Quatre-vingt-un citrons, quatre-vingt-une tomates.

3. Les adjectifs numéraux composés de quatre-vingt s'écrivent avec des traits d'union.

Exemples : Quatre-vingt-deux, quatre-vingt-trois, quatre-vingt-dix, quatre-vingt-onze,

quatre-vingt-dix-sept.

Centaines :

Pour les centaines c'est plus simple, car le seul cas particulier qui existe est celui de « cent » (on ne dit pas « **un** cent » mais « cent » tout court), tous les autres multiples de 100 ayant le nom du chiffre devant : « **deux** cent » , « **trois** cent »…

Le problème du « s » (utilisé dans certains cas pour le pluriel de « cent ») est traité ici.

L'adjectif « cent » :

il prend un « s » quand il est multiplié par un autre nombre et qu'il termine l'adjectif numéral.

exemple : J'ai lu sept cents pages.

il est invariable quand il n'est pas multiplié par un autre nombre

Exemple : Il a lu cent pages.

il est invariable quand il est suivi d'un autre nombre.

Exemple : Elle a écrit trois cent vingt-sept pages.

devant millier, million, milliard , il s'accorde quand il n'est pas suivi d'un nom de nombre.

Exemple : Quatre cents millions de francs.

Le reste

Au delà de mille, million, milliard les nombres s'écrivent de la même façon, et ils sont lus par groupes de 3 chiffres.

On voit donc qu'il est important de connaître le nombre de chiffres des nombres pour bien les prononcer. Paradoxalement, il est clair que si la lecture d'un nombre se fait bien (tout comme un mot) de gauche à droite, sa prononciation dépend fortement des groupements de trois chiffres que l'on peut opérer à partir de la droite cette fois. D'où l'intérêt d'un caractère séparateur de milliers (caractères « espace » ou « . ») pour faciliter cette lecture dans la vie courante (exemple : 1.325.240), comme on le voit dans les pays anglo-saxons.

Les structures utilisées dans le programme

Nous allons créer un tableau comportant les nombres de un à dix-neuf, qui seront utilisés trois fois :

pour tout nombre compris dans l'intervalle 1..19

pour tout nombre compris dans l'intervalle 61..79

pour tout nombre compris dans l'intervalle 81..99

A noter que parmi ces trois cas, seuls les nombres 61 et 71 se voient attribuer le mot « ET » entre le chiffre des dizaines et celui des unités.

On crée ensuite un tableau des dizaines qui contient les nombres vingt, trente, quarante, cinquante, soixante, soixante, quatre-vingt, quatre-vingt.

Remarque : 'soixante' et 'quatre-vingt' apparaissent deux fois, une fois en propre et l'autre pour former les cas spéciaux des intervalles 70..79 et 90..99.

On crée ensuite un tableau qui contient les unités restantes : mille, million, milliard,…

Nous allons supposer que le nombre saisi au clavier peut aussi contenir deux décimales.

Le principe récursif fait qu'on sait qu'un nombre à évaluer est égal à l'évaluation de sa partie principale (groupe de trois chiffres maximum) plus l'évaluation du reste

Exemple :

```
13.236.526,65 = évaluation de '13' + évaluation de '236.526,65'
   236.526,65 = évaluation de '236' + évaluation de '526,65'
   526,65 = évaluation de '526' + 'virgule' + évaluation de '65'
               65 = évaluation de '60' + évaluation de '5' =
                                  'soixante' + 'cinq' = 'soixante cinq' ;
          526 = évaluation de '500' + évaluation de '26'
           26 = évaluation de '20' + évaluation de '6' = 'vingt' + 'six' = 'vingt
six'
          évaluation de '500' = 'cinq' + 'cent' = 'cinq cent' ;
          et ainsi de suite…
```

ensuite, on n'a plus qu'à concaténer les résultats des différentes évaluations pour obtenir le résultat final.

La seule difficulté consiste à déterminer la partie principale d'un nombre. Nous avons en effet l'habitude de lire les nombres de gauche à droite en isolant les groupes de trois chiffres (millions, milliers) puis à l'intérieur les centaines, dizaines et unités. La seule difficulté pratique du langage consiste à déterminer le rang maximal du nombre, c'est-à-dire si le premier groupe à partir de la gauche est celui des milliards, millions, milliers , … mais pour cela, les yeux comptent les chiffres à partir de la droite !

Cette petite contradiction levée, l'algorithme que l'esprit suit intuitivement consiste à évaluer le groupe de trois chiffres le plus à gauche, puis à répéter l'opération en ne considérant que les chiffres restants à droite : cette méthode d'évaluation / séparation est caractéristique d'un processus récursif ! Seul point négatif, les résultats intermédiaires sont moins significatifs (voire même inexistants) que dans la méthode itérative et rendent difficile la gestion des exceptions du pluriel, etc …

Techniquement, l'algorithme va essayer de déterminer le groupe maximal à chaque étape et va analyser son coefficient, le traduire, puis relancer la fonction pour les chiffres restants : cela garantit qu'un groupe de trois chiffres tels que les millions ne seront évalués qu'une fois, car au prochain appel récursif un groupe de milliers (au plus) sera analysé , … , jusqu'à ce que l'on arrive finalement aux unités finales (ou des décimales) du nombre, terme de la récursivité.

Les spécificités de l'algorithme

Avant d'appeler la procédure récursive, on s'assurera tout d'abord que le nombre ne contient pas une virgule, auquel cas on évaluera d'abord sa partie entière, puis ensuite celle décimale (arrondie à deux décimales) , que l'on collera à la suite de la première et du mot « virgule » .

L'étape de base consiste à récupérer la longueur du nombre, afin de prendre la partie principale et l'évaluer. On fera ensuite un appel récursif et on collera cette évaluation à l'évaluation du reste.

Nous avons créé un tableau contenant l'expression littérale des nombres de zéro à dix-neuf. Un nombre constitué d'un seul chiffre constitue le ***point terminal*** de la récursivité, et on renvoie tout simplement son équivalent en français.

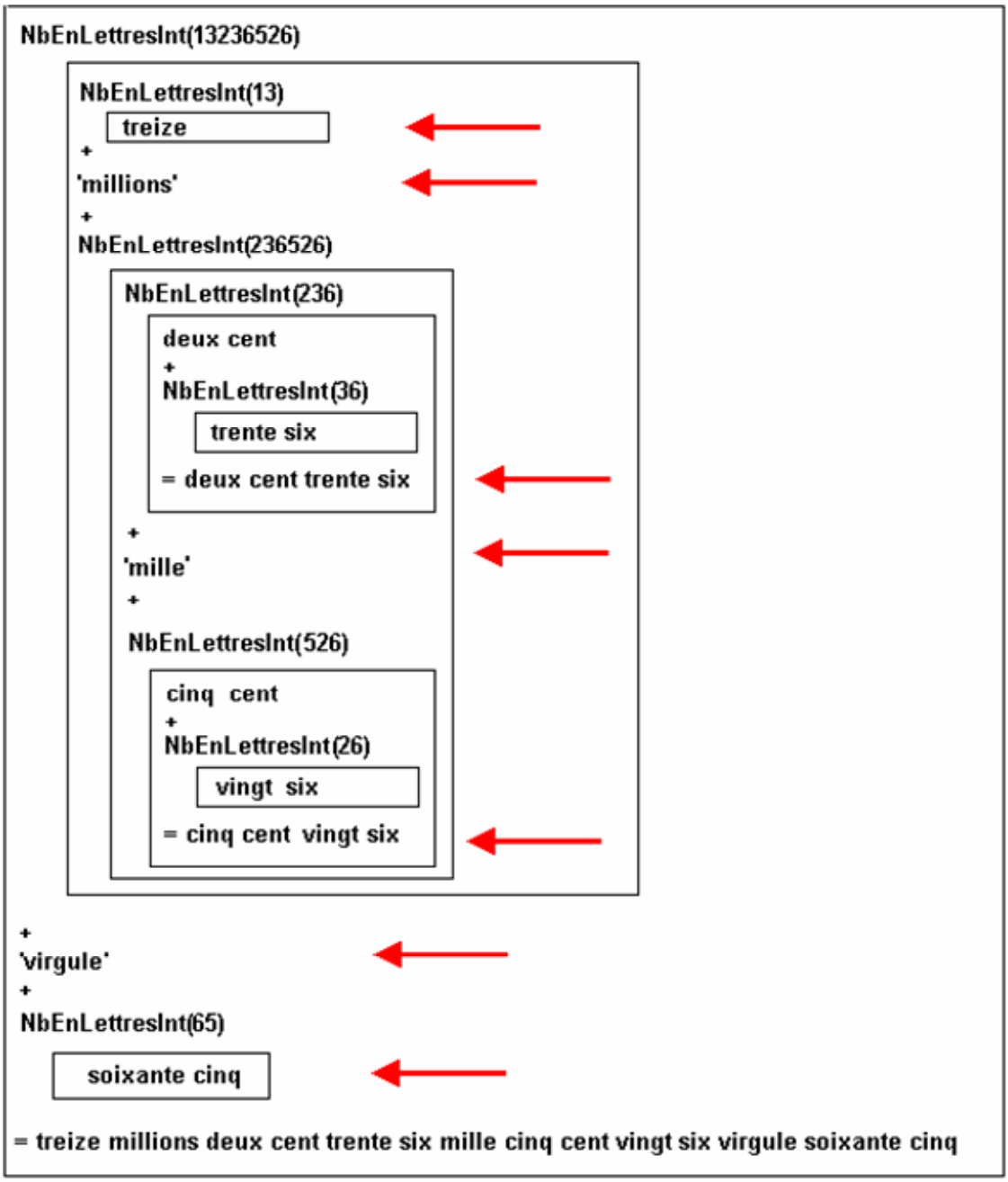
Si le nombre reçu est compris dans les intervalles 10..99 , alors on considère que l'évaluation du nombre est égale au chiffre des dizaines plus l'évaluation (***appel récursif***) des unités. On distinguera ici les cas où on doit insérer un « ET » et les cas des dizaines « soixante-dix » et « quatre-vingt-dix ».

Si le nombre reçu est compris dans les intervalles 70..79 ou 90..99 alors l'évaluation est égale au chiffre des dizaines (lu dans le tableau) décrémenté de un plus l'évaluation (***appel récursif***) du chiffre des unités incrémenté de 10. On distinguera ici les cas où on doit insérer un « ET » .

Si le nombre reçu est compris dans l'intervalle 100..999 alors l'évaluation est égale au chiffre des centaines (exception faite du nombre 100) plus la chaîne « cent » plus une évaluation (***appel récursif***) du reste. On prendra en compte le cas du « s » final qui vient s'ajouter au « cent » dans certains cas.

Si le nombre est supérieur à 999 alors on détecte d'abord sa puissance de mille (mille, million, milliard…) qui constitue ce qu'on a appelé ci-dessus la partie principale, on l'évalue par un ***appel récursif*** (en lui collant ensuite la puissance de mille correspondante) et on lui concatène le reste qui est évalué par un ***appel récursif***.

Voici un exemple des appels récursifs pour le nombre donné plus haut.


NbEnLettresInt(13236526)
NbEnLettresInt(13)
treize
+
'millions'
+
NbEnLettresInt(236526)
NbEnLettresInt(236)
deux cent
+
NbEnLettresInt(36)
trente six
= deux cent trente six
+
'mille'
+
NbEnLettresInt(526)
cinq cent
+
NbEnLettresInt(26)
vingt six
= cinq cent vingt six
+
'virgule'
+
NbEnLettresInt(65)
soixante cinq
= treize millions deux cent trente six mille cinq cent vingt six virgule soixante cinq

Chapitre V – Programmation de contraintes

Chapitre V-1

Le problème d'Einstein

De son vivant, Einstein a posé un petit problème de logique, destiné à tout le monde. Le problème n'est pas compliqué en soi, mais il demande un peu de réflexion et une très bonne organisation.

On a cinq maisons alignées de couleurs différentes.

Dans chaque maison vit une personne de nationalité différente.

Chaque personne boit une boisson différente.

Chaque personne fume un type de cigarette différent.

Chaque personne élève un animal différent.

Il faut trouver qui élève les poissons.

Indices :

1) L'anglais vit dans la maison rouge

2) Le suédois élève des chiens

3) Le danois boit du thé.

4) La maison verte est juste à gauche de la maison blanche.

5) Le propriétaire de la maison verte boit du café.

6) Le fumeur de Pall Mall élève des oiseaux.

7) Le propriétaire de la maison jaune fume des Dunhills.

8) L'homme qui vit dans la maison du centre boit du lait.

9) Le norvégien vit dans la première maison.

10) L'homme qui fume des Blends vit à côté de celui qui élève des chats.

11) L'homme qui élève des chevaux vit à côté du fumeur de Dunhills.

12) L'homme qui fume des Blue Masters boit de la bière.

13) L'allemand fume des Prince.

14) Le norvégien vit à côté de la maison bleue.

15) L'homme qui fume des Blends a un voisin qui boit de l'eau.

Nous allons par commencer par définir les types de données utilisés :

```
type_homme     = set of (anglais,suedois,danois,norvegien,allemand);
type_boisson   = set of (the,cafe,eau,lait,biere);
type_cigarette = set of (pall_mall,dunhills,blends,blue_masters,prince);
type_animal    = set of (chiens,oiseaux,chats,chevaux,poissons);
type_couleur   = set of (rouge,verte,blanche,jaune,bleue);
maison         = record
                 couleur   : type_couleur;
                 homme     : type_homme;
                 boisson   : type_boisson;
                 cigarette : type_cigarette;
                 animal    : type_animal;
                 end;
```

Pour résoudre ce problème, nous allons utiliser le programme récursif de "permutations" ou "anagrammes", que nous avons déjà étudié précédemment. Il nous donne toutes les façons possibles de placer les 5 maisons. Les variables qu'on utilisera sont donc un tableau de 5 maisons et une liste contenant les anagrammes.

```
maisons      : array[1..5] of maison;
permutations : tStringList;
```

Nous allons maintenant traduire en langage informatique les contraintes du problème; on utilise un tableau de 15 booléens, car on a 15 contraintes:

1) L'anglais vit dans la maison rouge

```
for i:=1 to 5 do
    if (maisons[i].homme=[anglais]) and (maisons[i].couleur=[rouge ]) then
ok[1]:=true;
```

2) Le suédois élève des chiens

```
for i:=1 to 5 do
    if (maisons[i].homme=[suedois]) and (maisons[i].animal =[chiens]) then
       ok[2]:=true;
```

3) Le danois boit du thé.

```
for i:=1 to 5 do
    if (maisons[i].homme=[danois ]) and (maisons[i].boisson=[the   ]) then
       ok[3]:=true;
```

4) La maison verte est juste à gauche de la maison blanche.

Pour une maison donnée "i", le "juste à gauche" se traduit par la maison "i-1".

```
for i:=1 to 5 do
    if (i>1) and (maisons[i].couleur=[blanche]) and (maisons[i-1].couleur=[verte])
       then ok[4]:=true;
```

5) Le propriétaire de la maison verte boit du café.

```
for i:=1 to 5 do
    if (maisons[i].couleur=[verte]) and (maisons[i].boisson=[cafe]) then
       ok[5]:=true;
```

6) Le fumeur de Pall Mall élève des oiseaux.

```
for i:=1 to 5 do
    if (maisons[i].cigarette=[pall_mall]) and (maisons[i].animal=[oiseaux]) then
       ok[6]:=true;
```

7) Le propriétaire de la maison jaune fume des Dunhills.

```
for i:=1 to 5 do
    if (maisons[i].cigarette=[dunhills ]) and (maisons[i].couleur=[jaune ]) then
```

```
            ok[7]:=true;
```

8) L'homme qui vit dans la maison du centre boit du lait.

Comme on a 5 maisons, celle du centre est la numéro 3.

```
      for i:=1 to 5 do
          if (maisons[3].boisson=[lait]) then ok[8]:=true;
```

9) Le norvégien vit dans la première maison.

```
          for i:=1 to 5 do
              if (maisons[1].homme=[norvegien]) then ok[9]:=true;
```

10) L'homme qui fume des Blends vit à côté de celui qui élève des chats.

Pour une maison donnée "i", "à côté" signifie "i-1" ou "i+1".

Il faut donc faire attention aux bords :

```
          for i:=1 to 5 do
              if (maisons[i].animal=[chats]) then
                 begin
                 if (i<5) and (maisons[i+1].cigarette=[blends]) then ok[10]:=true;
                 if (i>1) and (maisons[i-1].cigarette=[blends]) then ok[10]:=true;
                 end;
```

11) L'homme qui élève des chevaux vit à côté du fumeur de Dunhills.

```
          for i:=1 to 5 do
              if (maisons[i].cigarette=[dunhills]) then
                 begin
                 if (i<5) and (maisons[i+1].animal=[chevaux]) then ok[11]:=true;
                 if (i>1) and (maisons[i-1].animal=[chevaux]) then ok[11]:=true;
                 end;
```

12) L'homme qui fume des Blue Masters boit de la bière.

```
      for i:=1 to 5 do
          if (maisons[i].cigarette=[blue_masters]) and (maisons[i].boisson=[biere]) then
             ok[12]:=true;
```

13) L'allemand fume des Prince.

```
      for i:=1 to 5 do
          if (maisons[i].homme=[allemand]) and (maisons[i].cigarette=[prince]) then
             ok[13]:=true;
```

14) Le norvégien vit à côté de la maison bleue.

```
          for i:=1 to 5 do
              if (maisons[i].couleur=[bleue]) then
                 begin
                 if (i<5) and (maisons[i+1].homme=[norvegien]) then ok[14]:=true;
                 if (i>1) and (maisons[i-1].homme=[norvegien]) then ok[14]:=true;
                 end;
```

15) L'homme qui fume des Blends a un voisin qui boit de l'eau.

```
          for i:=1 to 5 do
              if (maisons[i].cigarette=[blends]) then
                 begin
                 if (i<5) and (maisons[i+1].boisson=[eau]) then ok[15]:=true;
                 if (i>1) and (maisons[i-1].boisson=[eau]) then ok[15]:=true;
                 end;
```

Et maintenant voici la procédure de recherche de la solution; avant d'arriver ici, nous avons stocké dans une liste "permutations" les 120 façons différentes de placer les 5 maisons.

Le principe est le suivant :

on choisit une par une les façons de placer les maisons (120 possibilités).

pour chaque façon trouvée :

on choisit une par une les façons de placer les hommes (120 possibilités)

pour chaque façon trouvée :

on choisit une par une les façons de placer les boissons (120 possibilités)

pour chaque façon trouvée :

on choisit une par une les façons de placer les cigarettes (120 possibilités)

pour chaque façon trouvée :

on choisit une par une les façons de placer les animaux (120 possibilités)

on teste si toutes les contraintes sont vérifiées.

Le problème de cet algorithme, tel qu'il est décrit ci-dessus, est que le temps requis est énorme.

En effet, il revient à tester 120^5 possibilités, soit environ 25 milliards. Il faut donc le modifier et rajouter les tests après chaque façon de placer les maisons, hommes,...

Nous obtenons donc le programme suivant :

on choisit une par une les façons de placer les maisons (120 possibilités).

pour chaque façon trouvée :

si conditions(4) alors

on choisit une par une les façons de placer les hommes (120 possibilités)

pour chaque façon trouvée :

si condition(1) alors

si condition(9) alors

si condition(14) alors

on choisit une par une les façons de placer les boissons (120 possibilités)

pour chaque façon trouvée :

si condition(3) alors

si condition(5) alors

si condition(8) alors

on choisit une par une les façons de placer les cigarettes (120 possibilités)

pour chaque façon trouvée :

si conditions(7) then

si conditions(12) then

si conditions(13) then

si conditions(15) then

on choisit une par une les façons de placer les animaux (120 possibilités)

si conditions(2) alors

si conditions(6) alors

si conditions(10) alors

si conditions(11) alors

affiche_solution;

La solution est donnée instantanément : c'est l'allemand.

Chapitre V-2

Les quatre musiciens

Quatre musiciens sont à gauche d'un pont; il fait nuit et il y a une seule torche; deux personnes au maximum peuvent traverser à la fois, et celui ou ceux qui traversent doivent avoir la torche pour éclairer le pont, sinon ils tombent dans l'eau. Le premier met 10 minutes pour traverser le pont, le deuxième 5 minutes, le troisième 2 minutes et le quatrième une minute.

Si le premier traverse en même temps que le deuxième, le temps de leur traversée sera de 10 minutes.

Ils doivent traverser le pont en 17 minutes.

Introduction

Nous allons réaliser le programme qui trouve les solutions de ce problème.

On commence par présenter les données :

1) - les musiciens qui sont à gauche du pont

2) - les musiciens qui sont à droite du pont

3) - le temps de traversée pour chacun

4) - le temps total de traversée

5) - l'endroit où se trouve la torche.

Et voici la modélisation de ces données :

1) - gauche : array[1..4] of boolean;

Si le premier musicien est à gauche du pont alors gauche[1]=true, sinon false. En même temps que gauche[1]=true, on a droite[1]=false, car le musicien ne peut pas se trouver en deux endroits différents au même moment

2) - droite : array[1..4] of boolean;

Si le premier musicien est à droite du pont alors droite[1]=true, sinon false.

3) const temps : array[1..4] of integer = (10,5,2,1); // minutes nécessaires

4) var minutes : integer;

5) var torche_a_gauche : boolean;

Analyse du problème

Les musiciens traversent le pont au maximum deux à la fois.

S'ils sont deux, les possibilités sont :

1,2; 1,3; 1,4; 2,3; 2,4; 3,4

Cela se traduit par deux boucles imbriquées :

```
for i:=1 to 4 do
for j:=i+1 to 4 do
```

Si à un instant donné on a la torche à gauche et qu'on veut traverser le pont, alors on considère que la torche va passer à droite, donc on écrit tout simplement :

```
torche_a_gauche:=false;
```

La traversée d'un musicien de gauche à droite se traduit par :

```
gauche[i]:=false;
droite[i]:=true;   //--- "i" étant le numéro du musicien
```

Ce programme étant récursif (c'est à dire qu'il fait des essais de traversée successifs de la gauche vers la droite, de la droite vers la gauche et ainsi de suite), il faut mémoriser la position en cours avant chaque traversée (de façon à la restaurer si la traversée faite débouche plus tard sur un échec (temps total supérieur à 17 minutes)).

On note le sens de traversée par un entier égal à "1" ou "-1", ce qui facilite l'appel récursif avec un appel "chercher(-sens)"; au début sens=1, ensuite dans le parcours de l'arbre, suivant le niveau de profondeur, il sera égal à 1 ou -1.

On va ajouter un autre paramètre à la procédure "chercher" qui représente le chemin parcouru à un instant donné, au bout de n traversées.

Si, par exemple, on a deux musiciens ("i" et "j") qui traversent le pont, alors on les mets dans une chaîne de caractères "s1" :

```
s1:=inttostr(i)+','+inttostr(j);
```

On appelle ensuite récursivement la procédure "chercher", avec le chemin déjà parcouru noté "s" auquel on ajoute le sens de la traversée (indiqué par une flèche "<-" ou "->";

```
chercher(-sens,s+' <-'+s1+';');
```

On doit ensuite prendre en compte le dernier déplacement effectué par un ou deux musiciens. En effet, rien n'empêche le programme récursif de faire le mouvement suivant:

- déplacement vers la droite du musicien 1. La torche se trouve donc à droite.

- déplacement vers la gauche du musicien 1. La torche est maintenant à gauche.

- déplacement vers la droite du musicien 1. La torche est maintenant à droite.

Et ainsi de suite, ce qui nous entraîne dans une boucle infinie (en fait ici elle n'est pas infinie à cause du temps qui s'additionne pendant les traversées; mais cela nous fait perdre énormément de temps pendant la récursivité). On doit donc sauvegarder le dernier mouvement effectué et le comparer à celui que nous allons effectuer. Supposons que nous déplaçons le musicien "1" vers la droite; on mémorise dans une chaîne "1". Si ensuite on veut déplacer "1" vers la gauche, on compare avec la chaîne mémorisée, et on voit qu'elles sont égales, donc ce mouvement ne sera pas effectué.

Voici l'entête de la procédure "chercher" :

```
procedure chercher(sens : integer; s,last : string);
```

où "s" est le chemin parcouru et "last" le(s) musicien(s) ayant effectué la dernière traversée; pour deux musiciens on a :

```
s1:=inttostr(i)+','+inttostr(j);
if s1<>last then
   chercher(-sens,s+' <-'+s1+';',s1);
```

Une traversée complète d'un musicien "i" sera donc codée par :

```
droite[i]:=true; gauche[i]:=false;  // traversée
torche_a_gauche:=false;             // on déplace la torche à droite
inc(minutes,temps[i]);              // addition nombre de minutes
s1:=inttostr(i);
if s1<>last then                     // si mouvement différent du mouvement
précédent
   chercher(-sens,s+s1+'->;',s1);  // alors appel récursif
droite[i]:=false;gauche[i]:=true;  // restauration positions initiales
torche_a_gauche:=true;              // on remet la torche à gauche
dec(minutes,temps[i]);              // restauration nombre de minutes
```

La procédure récursive sera donc constituée de deux boucles imbriquées, et de nombreux tests pour déterminer quels sont les musiciens qui se déplacent, d'affectations de positions et de restauration de positions. Il est plus intéressant de commencer par déplacer les musiciens deux par deux au lieu de un par un (c'est à dire tout seuls).

Le point d'arrêt de la procédure est le suivant :

```
if minutes=17 then
   if droite[1] and droite[2] and droite[3] and droite[4] then
      begin
      fini:=true;
      listbox1.items.add('trouvé !');
      listbox1.items.add(s);
      listbox1.refresh;
      exit;
      end
   else
      exit;
```

En exécutant cette procédure on constate que le programme affiche des solutions doubles. Cela est dû à l'opération suivante :

une fois qu'on a fait "3,4 ->" alors on teste :

"<- 1,2"

"<- 1,3" 3 part tout seul à gauche car droite[1]=false; (*)

"<- 1,4" 4 part tout seul à gauche car droite[1]=false;

"<- 2,3" 3 part tout seul à gauche car droite[1]=false; (*)

L'ordinateur prend donc deux fois le musicien "3" (pendant le parcours récursif) et le déplace à gauche, et comme ce mouvement mène à la solution, on a des doublons dans l'affichage.

Il faut donc mémoriser les solutions trouvées et comparer chaque nouvelle solution à celles déjà trouvées.

Chapitre VI – Récursivité croisée

Chapitre VI-1

Suites à récurrence double

Nous sommes parfois amenés à travailler sur des suites qui dépendent d'autres suites. Ainsi, dans l'exemple suivant, la suite *u* dépend de la suite *v*, qui elle dépend de *u*. Nous pouvons facilement réaliser le programme permettant de calculer les n-ièmes termes de ces deux suites, mais d'un point de vue algorithmique, cela nous amène à programmer une récursivité croisée; en effet, la fonction "u" appelle la fonction "v", cette dernière appelant "u".

Voici la définition des deux suites, suivie du programme;

u(0)=1, v(0)=2 et, pour tout entier naturel n :

u(n+1)=3u(n)+2v(n)

v(n+1)=2u(n)+3v(n)

Remarques :

1) la suite u-v est constante égale à 1.

2) Comme la fonction "v" est définie après "u", cela nous amène à utiliser le mot "forward".

```
function v(n:integer):int64; forward;

function u(n:integer):int64;
begin
     if n=0 then result:=1  //condition de sortie de la récursion
     else result:=3*u(n-1)+2*v(n-1);
end;

function v;
begin
     if n=0 then result:=2  //condition de sortie de la récursion
     else result:=2*u(n-1)+3*v(n-1);
end;

procedure TForm1.Button1Click(Sender: TObject);
var i:integer;
begin
     Screen.Cursor:=crHourGlass;
     Memo1.Clear;
     for i:=0 to SpinEdit1.Value-1 do
         begin
         Memo1.Lines.Add('u('+IntToStr(i)+') = '+IntToStr(u(i)));
         Memo1.Lines.Add('v('+IntToStr(i)+') = '+IntToStr(v(i)));
         Memo1.Lines.Add(' ');
         end;
     Screen.Cursor:=crDefault;
end;
```

Chapitre VI-2

Tri bulle boustrophédon

Nous allons voir dans ce chapitre l'utilisation du tri bulle boustrophédon pour trier un tableau d'entiers. Mais auparavant nous allons expliquer le tri à bulle (ou tri bulle) simple. Soit un tableau d'entiers non trié. Prenons par exemple :

9, 2, 15, 7, 3, 1, 6, 13, 17, 4

Le tri bulle consiste à parcourir le tableau de gauche à droite et chaque fois qu'on est sur un nombre, si son voisin droit a une valeur inférieure, alors on permute ces deux nombres. Une fois qu'on est au bout, si une permutation a eu lieu alors on recommence depuis le début.

Voici l'exemple: On est sur le premier nombre. Son voisin est plus petit que lui donc on les permute pour obtenir:

2, 9, 15, 7, 3, 1, 6, 13, 17, 4

Ensuite on avance pour tomber sur le deuxième nombre. Pas de permutation car 9<15. On avance. On est sur le troisième nombre. Comme 7<15 alors on les permute et on obtient :

2, 9, 7, 15, 3, 1, 6, 13, 17, 4

On passe ensuite au quatrième nombre qui est ici 3 et qui est plus petit que son voisin droit, donc on les permute:

2, 9, 7, 15, 1, 3, 6, 13, 17, 4

On continue, et on trouve une permutation à la dernière position :

2, 9, 7, 15, 1, 3, 6, 13, 4, 17

On est arrivé à la fin, mais des permutations ont eu lieu, donc on recommence depuis le début; voici les étapes successives:

```
position 1 : 2, 9, 7, 15, 1, 3, 6, 13, 4, 17  --> pas de changement
position 2 : 2, 7, 9, 15, 1,3, 6, 13, 4, 17
position 3 : 2, 7, 9, 15, 1,3, 6, 13, 4, 17  --> pas de changement
position 4 : 2, 7, 9, 1, 15, 3, 6, 13, 4, 17
position 5 : 2, 7, 9, 1, 3, 15, 6, 13, 4, 17
position 6 : 2, 7, 9, 1, 3, 6, 15, 13, 4, 17
position 7 : 2, 7, 9, 1, 3, 6, 13, 15, 4, 17
position 8 : 2, 7, 9, 1, 3, 6, 13, 4, 15, 17
position 9 : 2, 7, 9, 1, 3, 6, 13, 4, 15, 17  --> pas de changement
```

Mais on a eu des permutations, donc on recommence

```
position 1 : 2, 7, 9, 1, 3, 6, 13, 4, 15, 17  --> pas de changement
position 2 : 2, 7, 9, 1, 3, 6, 13, 4, 15, 17  --> pas de changement
position 3 : 2, 7, 1, 9, 3, 6, 13, 4, 15, 17
position 4 : 2, 7, 1, 3, 9, 6, 13, 4, 15, 17
position 5 : 2, 7, 1, 3, 6, 9, 13, 4, 15, 17
position 6 : 2, 7, 1, 3, 6, 9, 13, 4, 15, 17   --> pas de changement
position 7 : 2, 7, 1, 3, 6, 9, 4, 13, 15, 17
position 8 : 2, 7, 1, 3, 6, 9, 4, 13, 15, 17   --> pas de changement
position 9 : 2, 7, 1, 3, 6, 9, 4, 13, 15, 17   --> pas de changement
```

A cette étape nous avons eu 4 permutations, donc on recommence:

```
position 1 : 2, 7, 1, 3, 6, 9, 4, 13, 15, 17   --> pas de changement
position 2 : 2, 1, 7, 3, 6, 9, 4, 13, 15, 17
position 3 : 2, 1, 3, 7, 6, 9, 4, 13, 15, 17
position 4 : 2, 1, 3, 6, 7, 9, 4, 13, 15, 17
position 5 : 2, 1, 3, 6, 7, 9, 4, 13, 15, 17   --> pas de changement
position 6 : 2, 1, 3, 6, 7, 4, 9, 13, 15, 17
position 7 : 2, 1, 3, 6, 7, 4, 9, 13, 15, 17   --> pas de changement
position 8 : 2, 1, 3, 6, 7, 4, 9, 13, 15, 17   --> pas de changement
position 9 : 2, 1, 3, 6, 7, 4, 9, 13, 15, 17   --> pas de changement
```

A cette étape nous avons eu 4 permutations, donc on recommence:

```
position 1 : 1, 2, 3, 6, 7, 4, 9, 13, 15, 17
position 2 : 1, 2, 3, 6, 7, 4, 9, 13, 15, 17   --> pas de changement
position 3 : 1, 2, 3, 6, 7, 4, 9, 13, 15, 17   --> pas de changement
position 4 : 1, 2, 3, 6, 7, 4, 9, 13, 15, 17   --> pas de changement
position 5 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17
position 6 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
position 7 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
position 8 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
position 9 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
```

A cette étape nous avons eu 2 permutations, donc on recommence:

```
position 1 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
position 1 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
position 1 : 1, 2, 3, 6, 4, 7, 9, 13, 15, 17   --> pas de changement
position 1 : 1, 2, 3, 4, 6, 7, 9, 13, 15, 17
position 5 : 1, 2, 3, 4, 6, 7, 9, 13, 15, 17   --> pas de changement
position 6 : 1, 2, 3, 4, 6, 7, 9, 13, 15, 17   --> pas de changement
position 7 : 1, 2, 3, 4, 6, 7, 9, 13, 15, 17   --> pas de changement
position 8 : 1, 2, 3, 4, 6, 7, 9, 13, 15, 17   --> pas de changement
position 9 : 1, 2, 3, 4, 6, 7, 9, 13, 15, 17   --> pas de changement
```

A cette étape nous avons eu une permutation, donc on recommence, et quand on arrive à la fin aucune permutation n'a été effectuée, donc le tableau est trié et le programme s'arrête. Le tri a été réalisé en 7 boucles.

On constate que la première amélioration possible est de limiter le nombre d'avancées du curseur. En effet, chaque boucle sur les éléments du tableau a pour effet de ramener le plus grand nombre rencontré à la fin du tableau (ou à sa position correcte). Sachant donc qu'après le premier parcours le nombre 17 se trouve à la fin, il ne sera plus nécessaire à l'étape suivante d'aller jusqu'à 17, il suffira de s'arrêter à son prédécesseur. Ainsi à chaque étape le nombre de paires à comparer diminue.

Mais on peut encore améliorer cet algorithme : c'est le tri bulle boustrophedon.

Le parcours du tableau se fait alternativement de gauche à droite et de droite à gauche.

Quand on va de gauche à droite (cas ci-dessus), le plus grand nombre se trouve à l'extrémité droite du tableau (et la nouvelle extrémité devient don prédécesseur).

Quand on va de droite à gauche, le plus petit nombre se trouve à l'extrémité gauche du tableau et la nouvelle extrémité gauche devient son successeur. Quand les deux extrémités se rejoignent, alors le tableau est trié et le programme s'arrête. On peut aussi affirmer que si pendant un parcours (de gauche à droite ou inversement) aucune permutation n'a lieu, alors le tableau est trié.

Pour réaliser le tri bulle boustrophedon, nous allons réaliser deux procédures qui s'appellent l'une l'autre, d'où le nom de récursivité croisée.

Dans ce programme, nous allons utiliser une procédure de permutation de deux nombres. Voici un moyen d'échanger la valeur de deux nombres sans utiliser de variable auxiliaire :

soit deux variables a et b; a=3, b=5. On veut avoir à la fin a=5 et b=3.

On fait donc les opérations suivantes :

a := a + b = 8

b := a - b = 8 - 5 = 3

a := a - b = 8 - 3 = 5

Et comme on peut le constater, les deux valeurs sont échangées.

Nous allons voir les structures de données utilisées : le type "tab" qui est un tableau d'entiers, un tableau de type "tab" qui est un tableau de constantes (et que nous allons copier dans la variable "t2" pour pouvoir effectuer le tri dessus).

Comme on a une récursivité croisée, et comme nous n'avons pas utilisé la programmation orientée objet, il faut déclarer l'une des deux procédures en "forward".

```
type tab = array[1..10] of integer;
const t1 : tab = (9, 2, 15, 7, 3, 1, 6, 13, 17, 4);
var t2 : tab;

procedure arriere(var t : tab;limg,limd : integer);forward;

procedure affiche(t : tab);
var i : integer;
    s : string;
begin
s:='';
for i:=1 to 10 do
    s:=s+inttostr(t[i])+'; ';
form1.memo1.lines.add(s);
end;

procedure echange(var a,b : integer);
begin
a := a + b;
b := a - b;
a := a - b;
end;

procedure avant(var t : tab;limg,limd : integer);
var permute : boolean;
    i : integer;
begin
permute:=false;
for i:=limg to limd-1 do
    if t[i]>t[i+1] then
       begin
       echange(t[i],t[i+1]);
       permute:=true;
       end;
if permute then
   begin
   affiche(t);
   arriere(t,limg,limd-1);
   end;
end;
```

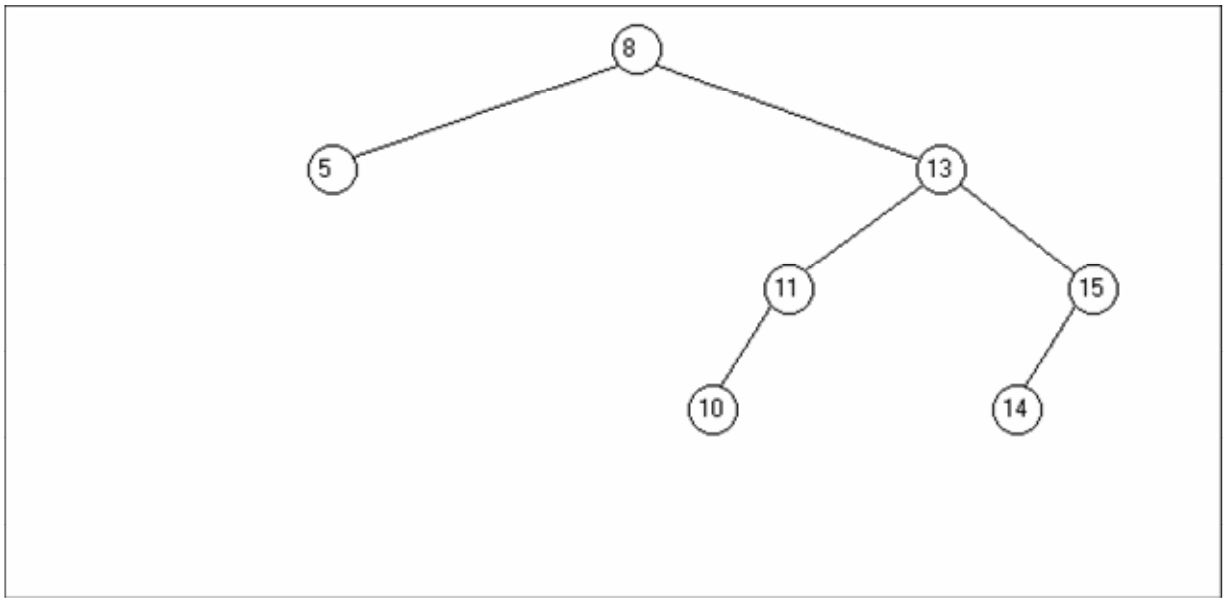
```
procedure arriere(var t : tab;limg,limd : integer);
var permute : boolean;
    i : integer;
begin
permute:=false;
for i:=limd downto limg+1 do
    if t[i]<t[i-1] then
       begin
       echange(t[i],t[i-1]);
       permute:=true;
       end;
if permute then
   begin
   affiche(t);
   avant(t,limg+1,limd);
   end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
move(t1,t2,sizeof(t1));
avant(t2,1,10);
end;
```

Chapitre VII – Les arbres

Chapitre VII-1

Les arbres binaires

Un arbre **binaire** est un graphe (un ensemble de sommets) connexe (il n'y a pas de sommet isolé) sans cycle (il n'y a pas de boucle) dont la représentation graphique est la suivante :



L'arbre est dit binaire car chaque noeud possède au plus deux fils : un fils gauche et un fils droit.

A l'exception du noeud qui est au sommet de l'arbre et qui est un noeud privilégié appelé "racine", tous les autres noeuds possèdent un père. Un noeud qui n'a pas de fils est appelé une feuille.

Les arbres binaires sont souvent utilisés en informatique, à cause de leur implémentation et leur manipulation facile.

Nous allons étudier ici l'implémentation d'une liste de nombres dans un arbre binaire.

Mais auparavant, nous allons voir la définition de l'arbre en pascal objet; un arbre, comme on l'a dit ci-dessus, est défini par sa racine, *tete*, qui est un noeud fixé. Comme on doit parfois se déplacer dans l'arbre sans utiliser obligatoirement une procédure récursive (en raison de sa structure qui s'y prête fortement), on a aussi une variable auxiliaire (nommée *courant*, de type *noeud*).

Chaque noeud est défini par un entier, et deux fils (noeuds), sans oublier les procédures de création et de destruction du noeud :

```
noeud = class
private
```

```
   entier : integer;
   gauche,droite : noeud;
   constructor create(nb : integer);
   destructor  destroy;
end;
```

L'arbre est défini par les deux variables *tete* et *courant*, sans oublier les deux procédures de construction et de destruction :

```
arbre = class
private
   tete,courant : noeud;
public
   constructor create(nb : integer);
   destructor  destroy;
end;
```

Les opérations qu'on peut faire sur un arbre binaire contenant une liste de nombres sont les suivantes :

1) le parcours et l'affichage

2) l'ajout d'un noeud dans l'arbre

3) le compte du nombre de noeuds d'un arbre

4) le compte du nombre de niveaux

5) l'équilibrage d'un arbre

6) la comparaison de deux arbres

1) Affichage d'un arbre

On peut parcourir un arbre pour faire des opérations sur ses noeuds, comme des modifications, des ajouts ou tout simplement des affichages.

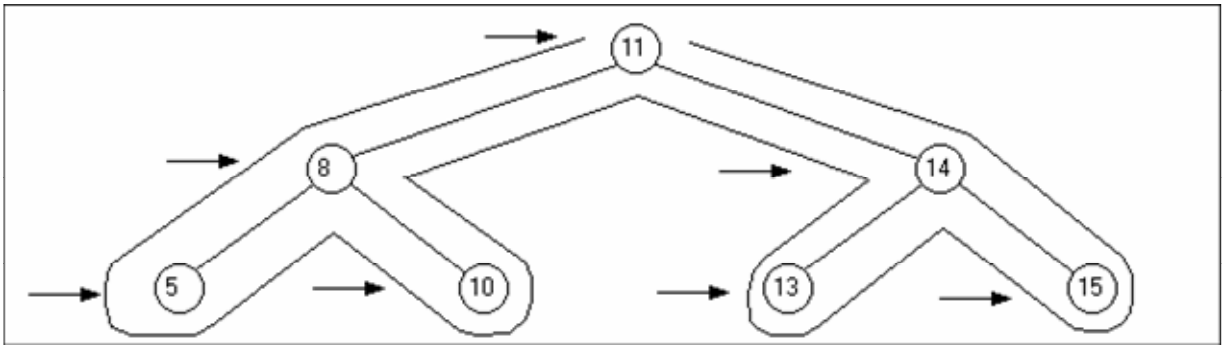
Plusieurs parcours sont possibles pour afficher un arbre :

- les parcours en profondeur

- les parcours en largeur

Les parcours en profondeur

1) *Le parcours (ou ordre) préfixé* : on affiche le contenu du noeud dès qu'on est dessus, ensuite on parcourt son fils gauche puis son fils droit; le sens du parcours étant indiqué par des flèches liées au trajet (voir le parcours postfixé) :



La procédure permettant de faire un parcours préfixé est la suivante :

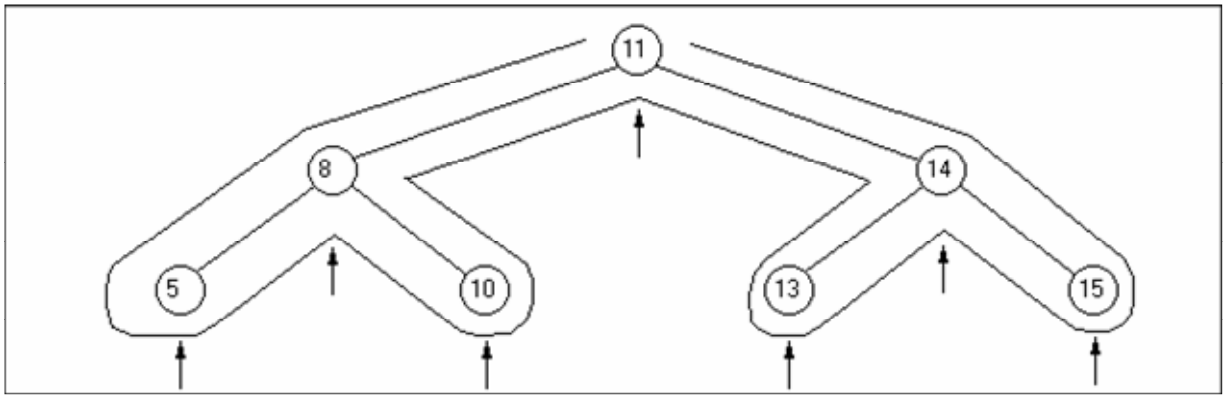
```
procedure aff(n : noeud);
begin
if n=nil then exit;  // test de sortie de la procédure récursive
edit5.text:=edit5.text+intToStr(n.entier)+';';   // chaîne qui sera affichée
aff(n.gauche);   // parcours fils gauche
aff(n.droite);   // parcours fils droit
end;
```

On l'appelle avec l'instruction suivante : *aff(arbre_nb.tete);*

Ainsi, un parcours préfixé pour cet arbre va nous donner comme résultat :

11;8;5;10;14;13;15

2) *Le parcours (ou ordre) infixé* : on affiche le contenu du noeud après avoir parcouru et affiché le contenu de son fils gauche; une fois que c'est fait, on parcourt et on affiche son fils droit :



La procédure permettant de faire un parcours infixé est la suivante :

```
procedure aff(n : noeud);
begin
if n=nil then exit;   // test de sortie
aff(n.gauche);        // parcours fils gauche
edit6.text:=edit6.text+intToStr(n.entier)+';';  // affichage noeud courant
aff(n.droite);       // parcours fils droit
end;
```

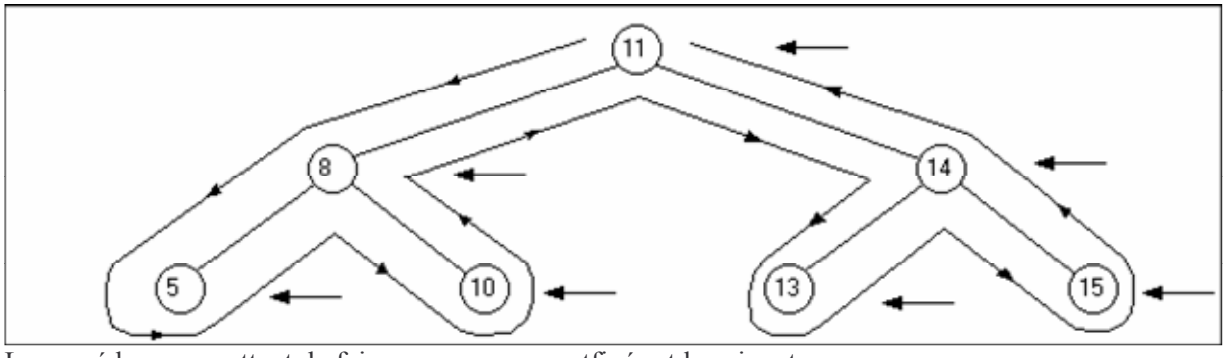
On l'appelle avec l'instruction suivante : *aff(arbre_nb.tete);*

Un parcours infixé pour cet arbre va nous donner comme résultat :

5;8;10;11;13;14;15

On constate que c'est aussi un tri par ordre croissant

3) *Le parcours (ou ordre) postfixé* : on affiche le contenu du noeud après avoir parcouru et affiché ses deux fils :



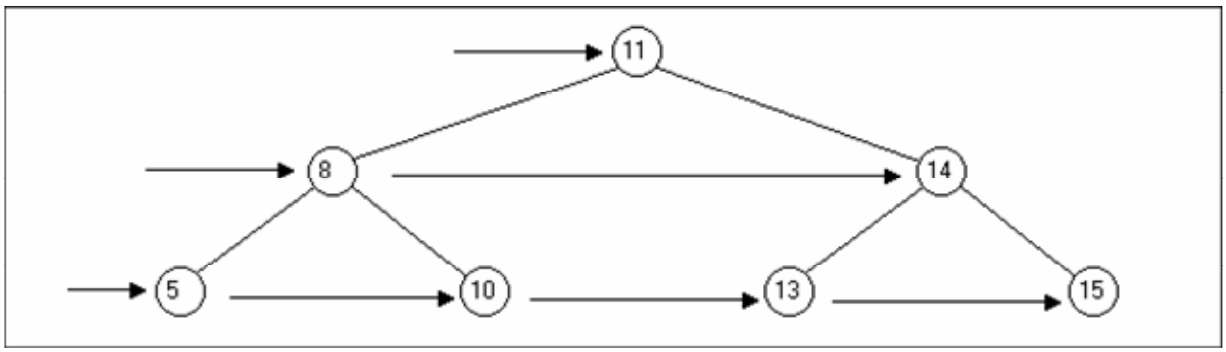
La procédure permettant de faire un parcours postfixé est la suivante :

```
procedure aff(n : noeud);
begin
if n=nil then exit;
aff(n.gauche);
aff(n.droite);
edit7.text:=edit7.text+intToStr(n.entier)+';';
end;
```

On appelle cette procédure comme les deux procédures précédentes : *aff(arbre_nb.tete);*

Le parcours en largeur

Le parcours en largeur est défini par le dessin suivant :



Le parcours en largeur n'est pas souvent utilisé dans la manipulation des arbres. Néanmoins, nous allons expliquer la manière dont on le réalise. Le temps requis pour faire ce parcours n'est plus linéaire. Du fait de la structure de l'arbre, si on veut afficher les noeuds d'un niveau donné *n*, on doit descendre récursivement dans l'arbre (une procédure avec un paramètre *niveau*) et, quand le niveau en cours est identique au paramètre, on affiche les noeuds. On quitte ensuite la procédure. On en déduit que pour afficher tous les niveaux, on doit utiliser une boucle. Voici la procédure correspondante :

```
procedure TForm1.Button13Click(Sender: TObject);
var i,j : integer;
   procedure aff(n : noeud;niveau_courant,niveau : integer);
   begin
   if n=nil then exit;
   if niveau=niveau_courant then
      edit9.text:=edit9.text+intToStr(n.entier)+';'
   else
      begin
      aff(n.gauche,niveau_courant+1,niveau);
      aff(n.droite,niveau_courant+1,niveau);
      end;
   end;
begin     // affichage largeur
edit9.text:='';
if arbre_nb=nil then exit;
j:=arbre_nb.compter_niveaux(arbre_nb.tete);
for i:=1 to j do
    aff(arbre_nb.tete,1,i);
end;
```

2) Ajout d'un noeud dans un arbre

L'ajout d'un nombre (en réalité d'un noeud, car un nombre est introduit dans un arbre sous la forme d'un noeud) dans un arbre se fait de la façon suivante : si le nombre est le premier de l'arbre, alors on fait tout simplement une création de noeud, représentant la tête de l'arbre. Sinon, comme, on a déjà des nombres dans l'arbre, on compare le nombre à ajouter avec le noeud en cours : si le nombre est plus grand que le contenu du noeud, alors on fait un appel récursif pour l'introduire dans

l'arbre dérivé du fils droit (si le fils droit est *nil*, alors on le crée et son contenu sera le nombre). Mais, si le nombre est plus petit que le contenu du noeud, alors on fait un appel récursif pour l'introduire dans l'arbre dérivé du fils gauche (si le fils gauche est *nil*, alors on le crée et son contenu sera le nombre qu'on veut introduire).

```
procedure noeud.ajouter(nb : integer);
begin
if nb>=entier then
   if droite<>nil then
      droite.ajouter(nb)
   else
      droite:=noeud.create(nb)
else
   if gauche<>nil then
      gauche.ajouter(nb)
   else
      gauche:=noeud.create(nb);
end;
```

3) Comptage des noeuds d'un arbre

Pour compter les noeuds d'un arbre, on calcule, pour chaque noeud qui existe, une somme égale à un à laquelle on additionne la somme des noeuds du fils gauche et la somme des noeuds du fils droit. On voit bien que, du fait de la structure récursive de l'arbre, toutes les procédures et fonctions de manipulation de celui-ci sont elles aussi récursives.

```
function arbre.compter_noeuds(posit : noeud) : integer;
begin
if posit=nil then compter_noeuds:=0
else
   compter_noeuds:=1+compter_noeuds(posit.gauche)+compter_noeuds(posit.droite);
end;
```

4) Comptage des niveaux d'un arbre

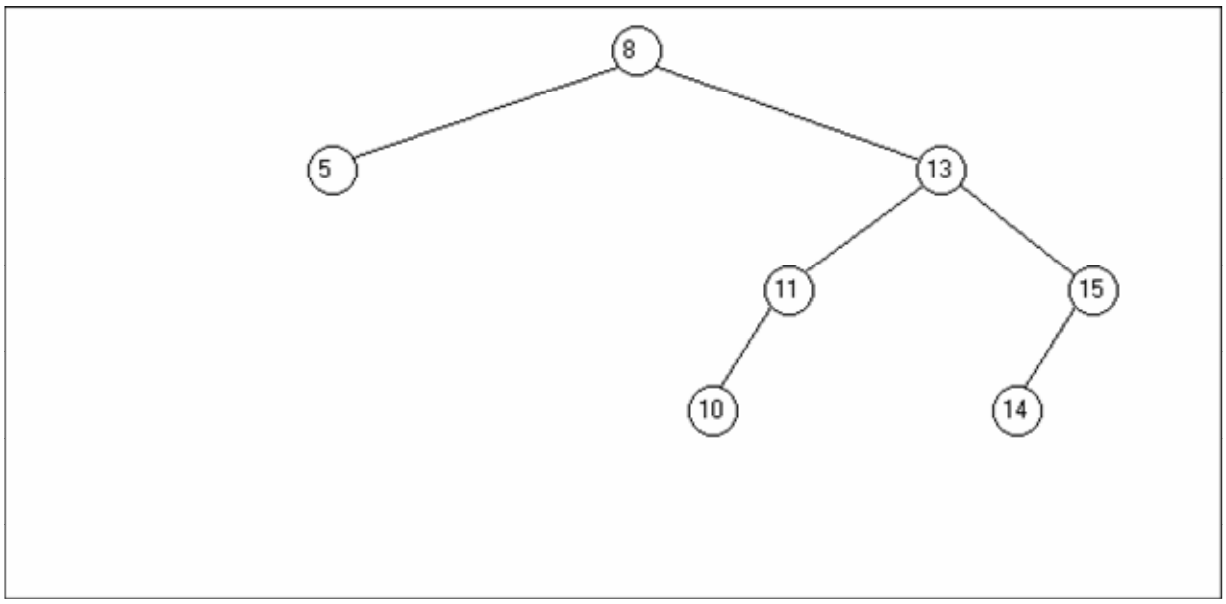
Pour compter les niveaux d'un arbre, on doit d'abord compter les niveaux du fils gauche, ensuite les niveaux du fils droit, comparer les deux et garder le plus grand, puis enfin ajouter 1, car il faut aussi prendre en compte le niveau du noeud sur lequel on se trouve. En effet, supposons qu'on a un arbre avec seulement la tête. Le fils droit est *nil*, donc il a 0 niveaux; idem pour le fils gauche. Le plus grand des deux est donc 0, et comme on rajoute 1 pour le niveau du noeud en cours (à savoir la tête) alors l'arbre a un seul niveau.

```
function arbre.compter_niveaux(posit : noeud) : integer;
var ng,nd : integer;
begin
if posit=nil then compter_niveaux:=0
else
   begin
   ng:=1+compter_niveaux(posit.gauche);
   nd:=1+compter_niveaux(posit.droite);
   if ng>nd then compter_niveaux:=ng else compter_niveaux:=nd;
   end;
end;
```

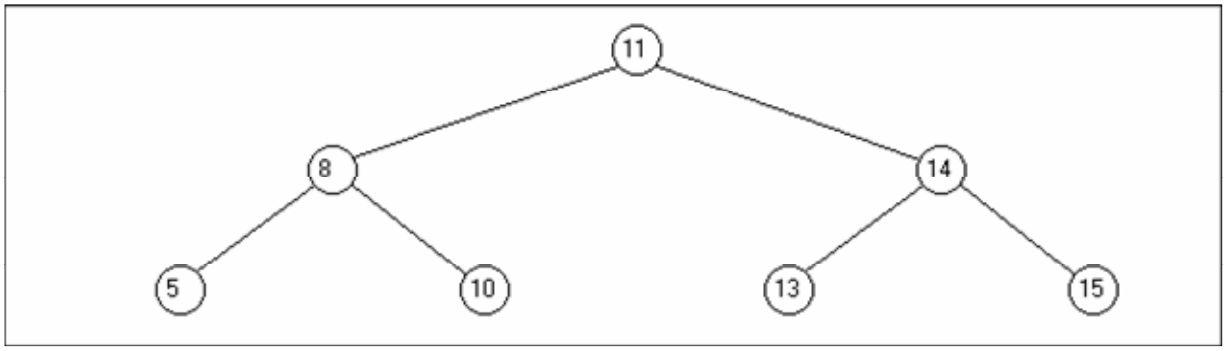
5) Equilibrage d'un arbre

Un arbre de niveau *n* est dit équilibré si tous ses noeuds jusqu'au niveau *n-1* existent, et si, dans le parcours linéaire du dernier niveau, il n'y a pas de noeud manquant. C'est ainsi que l'arbre du dessin *a)* n'est pas équilibré, alors que celui du *b)* l'est :

a)



b)



Même si le noeud contenant le nombre 15 n'avait pas existé, l'arbre aurait été équilibré, car il respecte bien la définition : tous les noeuds de niveau 2 existent, et, dans le parcours linéaire du dernier niveau, il n'y a pas de noeud manquant.

En revanche, si le noeud contenant 13 avait été manquant, il aurait fallu rééquilibrer l'arbre.

Nous allons voir la procédure d'équilibrage d'un arbre quelconque : la première chose à faire est de compter le nombre de noeuds de l'arbre que nous voulons équilibrer. Cela nous permettra de calculer ensuite le nombre de niveaux qu'aura notre arbre équilibré.

exemples:

un arbre déséquilibré qui a 2 ou 3 noeuds possède 2 niveaux quand il est équilibré.

un arbre qui a de 4 à 7 noeuds possède 3 niveaux quand il est équilibré

un arbre qui a de 8 à 15 noeuds possède 4 niveaux quand il est équilibré

Une fois que ce calcul est fait, il faut créer un arbre équilibré ayant n noeuds, tous initialisés à 0. Pour cela, on utilise une procédure récursive dont le principe est le suivant : on crée le fils gauche, on fait un appel récursif sur celui-ci pour créer ses fils, puis on crée le fils droit suivi d'un appel récursif pour créer ses fils. Nous devons donc transmettre en

paramètre le noeud en cours (car on va créer son fils gauche et droit), le niveau en cours (qui doit être égal à une constante établie précédemment) et enfin le nombre de noeuds créés, qui augmente progressivement avec chaque création de noeuds fils.

```
procedure creer_arbre(n : noeud;niv : integer;var nb_n : integer);
begin
if nb_n=nb_noeuds then exit;  // tous les noeuds nécessaires ont été créés
if niv<nb_niv then
   begin
   n.gauche:=noeud.create(0);
   inc(nb_n);
   creer_arbre(n.gauche,niv+1,nb_n);
   n.droite:=noeud.create(0);
   inc(nb_n);
   creer_arbre(n.droite,niv+1,nb_n);
   end;
end;
```

Maintenant que cet arbre est créé, il faut le remplir avec les valeurs. Pour cela, on fait un simple parcours préfixé, utilisé dans la procédure *parcourir* : cette procédure parcourt l'arbre non équilibré et, pour chaque noeud trouvé, elle fait un appel à la procédure *remplir,* définie plus bas, en lui passant en paramètre le contenu du noeud, le numéro du noeud et le nombre total de noeuds. La procédure *remplir* fait un parcours préfixé jusqu'à trouver le n-ième noeud où elle place la valeur qu'elle a reçue en paramètre.

```
procedure parcourir(n : noeud;var nb : integer);
begin
if n=nil then exit;
parcourir(n.gauche,nb);
inc(nb);remplir(arbre_eq.tete,n.entier,nb,nb_noeuds);
parcourir(n.droite,nb);
end;

procedure remplir(n : noeud;valeur,posit,nb_n : integer);
var n1 : integer;
begin
if nb_n=1 then
   n.entier:=valeur
else
   begin
   n1:=1;
   while nb_n>n1 do n1:=n1*2;
   if posit=(n1 div 2) then
      n.entier:=valeur
   else
   if posit<(n1 div 2) then
      remplir(n.gauche,valeur,posit,arbre_nb.compter_noeuds(n.gauche))
   else
      remplir(n.droite,valeur,posit-(n1 div 2),arbre_nb.compter_noeuds(n.droite));
   end;
end;
```

6) Comparaison de deux arbres

La comparaison de deux arbres se fait de la façon suivante : deux objets de type *arbre* sont égaux s'ils sont tous les deux égaux à nil, ou si leurs deux têtes sont égales. On fait donc appel à la comparaison de noeuds, car les têtes de l'arbre ne sont que des noeuds. Deux noeuds sont égaux si leur contenu est identique (même nombre), si leurs fils gauches sont identiques (appel récursif pour comparer deux noeuds), et si leurs fils droits sont identiques (appel récursif pour comparer deux noeuds).

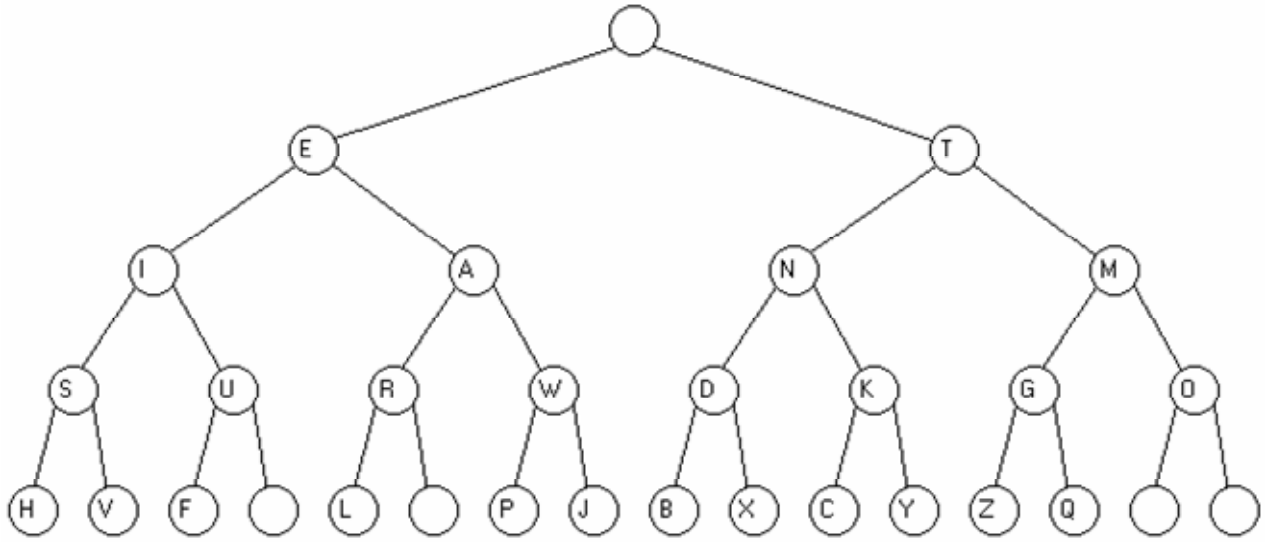
```
function noeud.egal_a(noeud2 : noeud) : boolean;
begin
if (self=nil) and (noeud2=nil) then
   egal_a:=true
else
if ((self<>nil) and (noeud2=nil)) or
```

```
   ((self=nil) and (noeud2<>nil)) or
   (entier<>noeud2.entier) then
   egal_a:=false
else
   egal_a:=gauche.egal_a(noeud2.gauche) and droite.egal_a(noeud2.droite);
end;

function arbre.egal_a(arbre2 : arbre) : boolean;
begin
if (self=nil) and (arbre2=nil) then egal_a:=true
else
if (self=nil) and (arbre2<>nil) then egal_a:=false
else
if (self<>nil) and (arbre2=nil) then egal_a:=false
else
    egal_a:=tete.egal_a(arbre2.tete);


end;
```

Chapitre VII-2

Le code Morse

En utilisant les techniques des arbres binaires, on peut facilement réaliser un programme qui représente graphiquement le code Morse par un arbre binaire. Dans cet arbre on considère seulement les lettres de A à Z, car leur représentation tient sur 5 niveaux. On rappelle que le code Morse est constitué de points et de tirets : ainsi le "S" est représente par 3 points "..." et le "O" par trois tirets "---". On considère que les points sont représentés par un fils gauche d'un noeud, et un tiret par un fils droit d'un noeud. Voici la représentation graphique d'un arbre contenant l'alphabet:



Nous devons réaliser le programme qui dessine cet arbre. On utilise une chaîne de caractères pour stocker les lettres. Il faut savoir que l'alphabet Morse a plus de caractères que ce que nous avons choisi, qui font qu'un arbre binaire descend à sept niveaux. C'est pourquoi nous nous sommes limités seulement à 5 niveaux, et nous avons rempli les caractères manquants par des espaces.

Pour réaliser cet arbre, nous utilisons le parcours préfixe que nous avons vu précédemment et nous utilisons une formule mathématique afin d'introduire à chaque noeud de l'arbre le caractère correspondant.

La chaîne de caractères est donc, en fonction du parcours préfixe, la suivante :

const st : string = ' EISHVUF ARL WPJTNDBXKCYMGZQO ';

Les structures de données utilisées sont les suivantes:

```
noeud = class
private
   carac : char;
   gauche,droite : noeud;
   constructor create(c : char);
   destructor  destroy;
   procedure   creer(nb,niv : integer);
   procedure   afficher(x,y,l : integer);
end;
```

```
  arbre = class
  private
     tete,courant : noeud;
  public
     constructor create(c : char);
     destructor  destroy;
     procedure   afficher(x,y,l : integer);
  end;


constructor noeud.create(c : char);
begin
carac:=c;
gauche:=nil;droite:=nil;
end;

destructor noeud.destroy;
begin
if gauche<>nil then gauche.destroy;
if droite<>nil then droite.destroy;
end;

procedure noeud.afficher(x,y,l : integer);
begin
with form1.image1.picture.bitmap.canvas do
   begin
   ellipse(x,y,x+25,y+25);
   textOut(x+5,y+5,carac);
   if gauche<>nil then
      begin
      moveto(x+3,y+20);lineto(x-(l div 2)+10,y+70);
      gauche.afficher(x-(l div 2),y+60,l div 2);
      end;
   if droite<>nil then
      begin
      moveto(x+22,y+20);lineto(x+(l div 2)+10,y+70);
      droite.afficher(x+(l div 2),y+60,l div 2);
      end;
   end;
end;

procedure noeud.creer(nb,niv : integer);
var i : integer;
begin
if niv<5 then
   begin
   gauche:=noeud.create(st[nb+1]);
   gauche.creer(nb+1,niv+1);
   i:=nb+(round(exp((5-niv)*ln(2))));
   droite:=noeud.create(st[i]);
   droite.creer(i,niv+1);
   end;
end;

//--------------------------------------
constructor arbre.create(c : char);
begin
tete:=noeud.create(c);
courant:=tete;
end;

destructor arbre.destroy;
begin
courant:=nil;
tete.destroy;
end;

procedure arbre.afficher(x,y,l : integer);
begin
tete.afficher(x,y,l);
end;

//--------------------------------------


procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
morse:=nil;
image1.picture.bitmap:=tbitmap.create;
with image1.picture.bitmap do
   begin
   width:=639;height:=500;
   end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
morse:=arbre.create(' ');
morse.tete.creer(1,1);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
morse.afficher(image1.width div 2 -10, 10, image1.width div 2);
end;
```

Nous allons expliquer maintenant la création de l'arbre.

Le sommet contient un espace.

A gauche du sommet on a la lettre "E" et à droite la lettre "T", c'est à dire les positions 2 et 17 de notre chaîne de caractères "st". 2=1+1 et 17=1+16.

Au niveau suivant, à gauche du "E" on a "I" et à droite "A", c'est à dire les positions 3 et 10 de notre chaîne de caractères "st". Mais 3=2+1 et 10=2+8, où 2 est la position de "E" dans "st".

Au niveau suivant, à gauche de "I" on a "S" et à droite "U", c'est à dire les positions 4 et 7 de notre chaîne de caractères "st". Mais 4=3+1 et 7=3+4.

On constate donc que le fils gauche d'un noeud à la valeur du père incrémenté de 1 (on parle des caractères qui sont dans la chaîne "st" : "E"+1="I"; "I"+1="S") et que le fils droit à la valeur du père incrémenté de $2^{5\text{-niveau}}$. On lit cela "2 élevé à la puissance (5-niveau)". On rappelle qu'ici le sommet de l'arbre est au niveau 1, donc la lettre "E" est au niveau 2; par conséquent, le fils droit de "E" sera incrémenté de 8, car $2^3=8$. De même, le fils droit du sommet sera incrémenté de 16, car $2^{5-1} = 2^4 = 16$.

Chapitre VII-3

Les arbres binaires

Évaluation d'une expression arithmétique

Nous convenons d'appeler "expression arithmétique" toute chaîne de caractères construite à partir de la liste des symboles 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,).

Les expressions considérées ne mettront en jeu que des entiers relatifs.

Certaines expressions sont incorrectes comme "2+" (que nous pouvons toutefois interpréter comme "2+0") ou "3x(6+4". Nous supposerons dans ce qui suit que les expressions sont correctes.

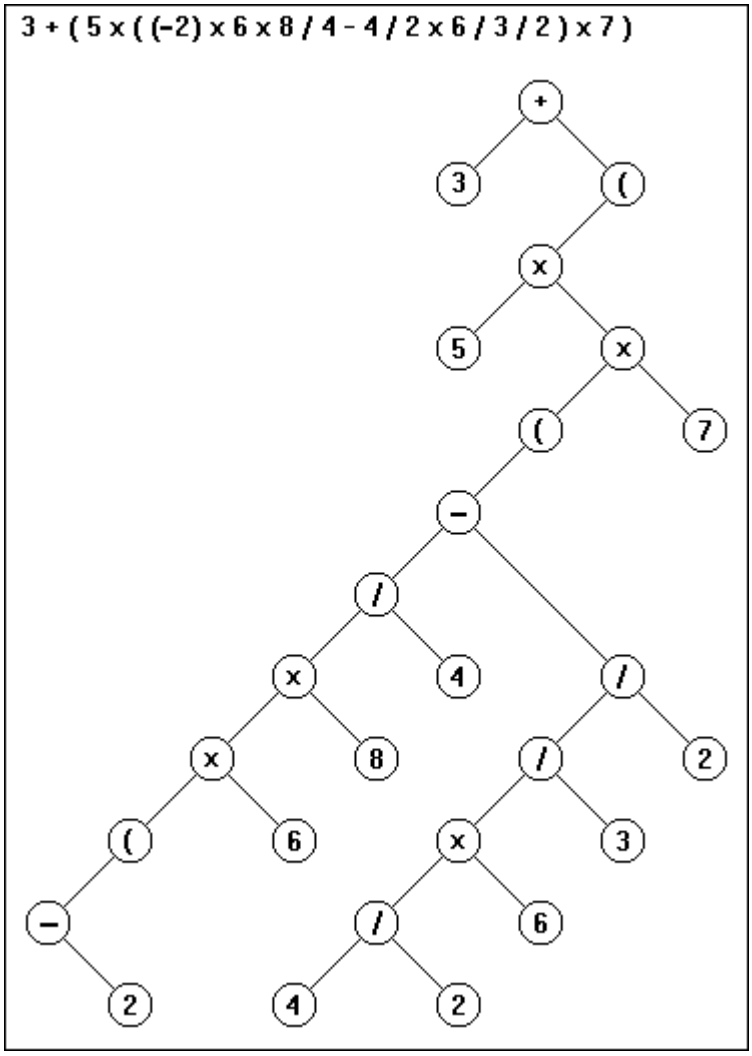
Une expression arithmétique peut être représentée par un arbre binaire.

La structure d'arbre binaire

L'arbre est dit binaire car chaque noeud "opérateur" possède deux fils : un fils gauche et un fils droit. Un noeud "opérateur" est un noeud qui contient un des symboles "+, -, * ou / ". Un noeud fils est soit un noeud opérateur, soit un noeud opérande - un entier - (c'est alors une feuille de l'arbre car il n'a pas de fils). La tête de l'arbre est toujours un noeud opérateur sauf dans le cas trivial où l'expression est réduite à un entier.

exemple : 3+(5*((-2)*6*8/4-4/2*6/3/2)*7)

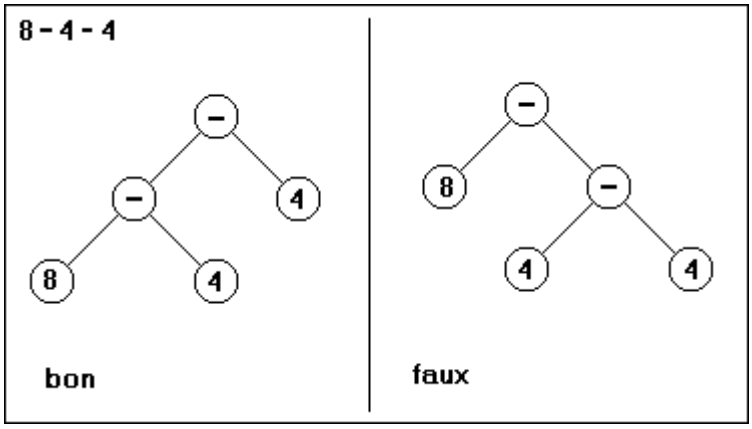
Voici le dessin de l'arbre binaire correspondant à cette expression :



On constate que pour le nombre " - 2 ", tout en bas de l'arbre, à gauche, on n'a pas de fils droit. Dans ce cas on dit que le signe "-" (ce signe "-" bien précis) est un opérateur unaire, puisqu'il n'a qu'un fils. Cette notion d'opérateur unaire est applicable aussi pour les parenthèses, parce qu'elles n'ont qu'un fils au lieu de deux, comme devrait avoir tout noeud d'un arbre binaire. Il faut noter que par rapport au signe "-", la parenthèse ouvrante peut aussi bien avoir un fils gauche avec son contenu, qu'un fils droit. Ici, dans ce dessin, nous avons choisi délibérément le fils gauche, pour la distinguer du signe "-", qui, lui, a obligatoirement le fils à droite.

Ensuite, pour construire l'arbre, il faut faire attention au sens de lecture de l'expression.

ex. 3 : 8-4-4



On voit dans cet exemple que l'expression doit être lue de droite à gauche. Lue de gauche à droite, l'expression conduirait à l'arbre de droite dont le résultat est manifestement faux car égal à 8.

Voici la déclaration sous Delphi (pascal objet) de la structure utilisée :

```
Noeud = class
    private
       op    :char; //le genre du noeud : +, -, *, /, ou "c" s'il s'agit d'une
feuille
       valeur  : integer; //dans le cas d'une feuille, sa valeur entière
       FilsGauche,FilsDroit : Noeud;
    public
       constructor create;
       destructor  destroy;
    end; { Noeud }

 Arbre = class
    private
       tete,courant:noeud;
    public
       constructor create;
       destructor  destroy;override;
       procedure   Remplit(s:string);
       function    evalue(n:noeud):integer;
    end; {arbre}
```

Vous noterez que la procédure suivante, permettant de détruire un noeud, est récursive :

```
destructor noeud.destroy;
begin
if FilsGauche<>nil then FilsGauche.destroy;
if FilsDroit<>nil then FilsDroit.destroy;
inherited destroy;
end;
```

En effet, FilsGauche.Destroy provoque un appel à noeud.destroy puisque FilsGauche est un noeud ! D'où le test "if FilsGauche<>nil...".

Comment évaluer un arbre binaire

Pour évaluer le résultat d'un arbre, on applique tout simplement la formule suivante :

- si le noeud à évaluer contient un entier, le résultat est égal à cet entier.

- sinon, le résultat est égal à :

"evalue(FilsGauche) **opération** evalue(FilsDroit) ", où "opération" désigne l'un des quatre opérateurs usuels.

La procédure permettant d'évaluer le résultat de l'expression est la procédure récursive suivante :

```
function arbre.evalue(n:noeud):integer;    //récursive, évidemment
begin
if n.op='c' then // c'est une feuille "chiffre"
   evalue:=n.valeur
else
   begin       // noeud opérateur
   case n.op of
       '+': evalue:=evalue(n.FilsGauche)+evalue(n.FilsDroit);
       '-': evalue:=evalue(n.FilsGauche)-evalue(n.FilsDroit);
       '*': evalue:=evalue(n.FilsGauche)*evalue(n.FilsDroit);
       '/': evalue:=evalue(n.FilsGauche) div evalue(n.FilsDroit);
       end;
   end;
end;
```

Un exemple d'appel de cette procédure est : UnArbre.Evalue(UnArbre.tete).

Comment construire l'arbre binaire

Le travail le plus délicat ici n'est pas l'évaluation de l'arbre, mais la façon de le remplir à partir de la chaîne de caractères qui représente l'expression à évaluer.

C'est ce qu'on appelle la construction d'un arbre d'analyse par descente récursive.

Traitons pour commencer le cas où l'expression considérée ne comporte que des additions et des soustractions (et aucune parenthèse).

L'appel initial de la procédure "Remplit" reçoit l'expression entière comme paramètre.

Le principe de la procédure est que eval(a±b±c)=eval(a±b) ± c.

Pour cette expression, « a±b » est appelé "gauche" et "c" est appelé "droite". Nous nous ramenons toujours ainsi à l'évaluation de deux sous-expressions dont l'une est triviale (celle de droite, qui constitue une feuille de l'arbre).

Nous commençons par initialiser gauche à vide car gauche ne sera pas nécessairement affecté par la première partie de la procédure : en effet si on veut construire l'arbre de "-2" la partie gauche n'est pas initialisée.

Ensuite nous cherchons le premier signe "+" ou "-" en lisant l'expression de droite à gauche. Si nous n'en trouvons pas, c'est que l'expression est réduite à un entier et nous avons fini (condition de sortie de la récursion).

Sinon, nous coupons l'expression en deux (gauche et droite, de part et d'autre de l'opérateur trouvé), nous affectons au noeud courant le symbole opérateur trouvé et nous affectons à "droite" la valeur de l'entier isolé à droite, ce qui nous permet de remplir le fils droit du noeud opérateur de l'arbre dont nous étions partis.

Nous créons ensuite le fils gauche à partir de "gauche".

S'il est réduit à un entier, c'est fini (condition de sortie de la récursion).

S'il contient encore au moins un opérateur, il nous suffit de rappeler récursivement la procédure Remplit en lui passant "gauche" comme paramètre. Nous prenons au préalable la précaution de fixer le noeud courant au fils gauche actuel pour que le remplissage de l'arbre se poursuive au bon endroit.

Vous noterez que cette méthode permet d'évaluer certaines expressions "irrégulières" comme : +2+3, *vide*, 89, 5+97-, 78++5 ou "+".

C'est pour cette raison que nous introduirons la notion d'automate, après la procédure de création de l'arbre binaire, pour vérifier si une expression mathématique est correcte d'un point de vue syntaxique.

Mais voici d'abord une première version de la création d'un arbre :

```
procedure arbre.Remplit(s:string);    //récursive !!!
var p,pp,pm :integer;
    gauche,droite:string;
begin
     //pour une expression du type a ± b ± c
     gauche:='';
     p:=length(s);
     while (p>0) and (s[p]<>'+') and (s[p]<>'-') do dec(p);
     if p>0 then  //il y a bien un "+" ou un  "-" donc on a un arbre binaire
        begin
        gauche:=copy(s,1,p-1);
        droite:=copy(s,p+1,length(s));
        courant.op:=s[p];    //la tête, lors du premier appel
        courant.FilsDroit:=noeud.create;
        courant.FilsDroit.op:='c'; // c pour "chiffre"
        if droite<>'' then
           courant.FilsDroit.valeur:=StrToInt(droite)
        else
           courant.FilsDroit.valeur:=0;
        end
     else    //il n'y a pas/plus de "+" ni de "-", donc il reste un entier
        begin
        courant.op:='c'; // c pour "chiffre"
        if s<>'' then
           courant.valeur:=StrToInt(s)
        else
           courant.valeur:=0;
        exit;
        end;
     //créons le fils gauche
     courant.FilsGauche:=noeud.create;
     if gauche<>'' then
        begin
        p:=length(s);
        while (p>0) and (s[p]<>'+') and (s[p]<>'-') do dec(p);
        if p>0 then
           begin
           courant:=courant.FilsGauche;
           Remplit(gauche); //il faut continuer --->>> APPEL RÉCURSIF
           end
        else    //c'est fini
           begin
           courant.FilsGauche.op:='c'; // c pour "chiffre"
           courant.FilsGauche.valeur:=StrToInt(gauche);
           exit;
           end;
        end
     else //on met un zéro
          begin
          courant.FilsGauche.op:='c'; // c pour "chiffre"
          courant.FilsGauche.valeur:=0;
          end;
end;
```

Nous allons maintenant modifier progressivement cette procédure pour arriver à la création d'un arbre binaire d'une expression contenant des parenthèses.

Tout d'abord nous allons créer une fonction "nb_elems" qui va recevoir comme paramètre une expression saisie au clavier, et qui va nous dire si cette expression contient un ou plusieurs éléments; si elle contient plusieurs éléments alors on va renvoyer comme réponse le nombre 2; on utilise cette valeur car un arbre binaire a deux fils au maximum; ainsi, comme nous l'avons vu précédemment, pour l'expression "2+3 - 5" on a l'arbre constitué des branches "2+3" et "5".

Cette fonction renvoie aussi la partie gauche et la partie droite d'une expression, d'où l'utilisation du mot "var g,d : string" dans les paramètres de la fonction.

Et elle renvoie aussi le signe qui est au sommet de l'arbre binaire (dans le cas des branches "2+3" et "5", le signe du sommet sera " - ").

Nous devons transmettre aussi les séparateurs : " + " et " - " ensemble, ou bien " * " et " / " ensemble, car ce sont eux qui vont nous aider à délimiter la partie gauche de la partie droite.

Sachant que par la suite nous allons avoir des parenthèses, nous allons les prendre en compte dès maintenant en utilisant un compteur "nb" qui sera incrémenté chaque fois qu'on rencontrera une fermante (ne pas oublier que le sens de la lecture se fait de la droite vers la gauche, donc la première parenthèse susceptible d'être rencontrée sera fermante) et décrémenté chaque fois qu'on rencontrera une ouvrante.

```
function nb_elems(s : string;var g,d : string;s1,s2 : char;var sig : char) : integer;
var trouve : boolean;
    p,nb   :integer;
begin        // on extrait la partie gauche et droite, séparées par s1 ou s2
p:=length(s)+1;nb:=0; // en faisant attention aux parenthèses, s'il y en a
repeat
   dec(p);
   if s[p]='(' then dec(nb);
   if s[p]=')' then inc(nb);
   trouve:=(nb=0) and ((s[p]=s1) or (s[p]=s2));
until (p=1) or (trouve);
if p>1 then  // deux ou plusieurs elements; mais dans un arbre binaire il y en a 2
   begin
   d:=copy(s,p+1,length(s));
   g:=copy(s,1,p-1);
   sig:=s[p];
   nb_elems:=2;
   end
else    // un seul élément
   begin
   d:=s;
   g:='';
   sig:=#0;
   nb_elems:=1;
   end;
end;
```

Notre procédure initiale devient donc la procédure "traiter_sans_parentheses" qui va recevoir comme paramètres un noeud, l'expression saisie et les deux signes séparateurs (" +, - " ou bien " *, / ").

Supposons que notre expression contient une expression avec les quatre signes; au début les séparateurs sont les signes " + " et " - ".

Nous allons distinguer les deux cas, celui où on a deux éléments et celui où on n'en a qu'un :

- si on a deux fils, alors :

- on crée le fils droit, on fait un appel récursif avec la nouvelle expression constitué de la partie droite et les deux nouveaux séparateurs " * " et " / ".

- on met le signe qui séparait la partie droite de la gauche dans le nœud

- on crée le fils gauche et on fait un appel récursif avec la nouvelle expression constituée de la partie gauche

- si on a un seul fils, alors :

- ou bien les séparateurs initiaux étaient "+" et " - ", auquel cas on fait un appel récursif avec les séparateurs "*" et "/".

- ou bien c'est un entier seul et on le met dans le noeud.

Voici donc cette procédure programmée, qu'on peut appeler avec l'instruction suivante :

"traiter_sans_parentheses(noeud1,expression,'+','-',gauche,droite,signe)"

```
procedure traiter_sans_parentheses(courant : noeud;s : string;signe1,signe2 : char);
var g2,d2 : string;
    sig2 : char;
begin
if nb_elems(s,g2,d2,signe1,signe2,sig2)=2 then  // deux termes ou plus
   begin
   courant.FilsDroit:=noeud.create;
   traiter_sans_parentheses(courant.FilsDroit,d2,'*','/');
   courant.op:=sig2;
   courant.FilsGauche:=noeud.create;
   traiter_sans_parentheses(courant.FilsDroit,g2,signe1,signe2);
   end
else
   if nb_elems(s,g2,d2,'*','/',sig2)=2 then
      traiter_sans_parentheses(courant,s,'*','/')
   else
      begin
      courant.op:='c'; // c pour "chiffre"
      courant.valeur:=strToInt(s);
      end;
end;
```

Les deux procédures ci-dessus vont se trouver dans notre nouvelle procédure "remplit", qui va traiter une expression contenant des parenthèses. Vous remarquerez que le corps de cette procédure est presque identique à celui de la procédure "traiter_sans_parentheses"; en effet on doit distinguer et traiter les deux cas possibles : un terme ou deux termes.

Si on a deux termes (par rapport aux signes en cours "+ -" ou " x / ", alors on les sépare et on fait un appel récursif.

Si on n'a qu'un seul terme, on doit distinguer les trois cas suivants :

- ou bien on a un seul terme par rapport à "+ -", auquel cas on fait un appel récursif avec " x / ".

- ou bien on a une seule parenthèse, auquel cas on fait un appel récursif avec l'intérieur de la parenthèse

- ou bien on a un seul nombre

```
procedure arbre.Remplit(courant:noeud; s:string; signe1,signe2:char);
var g1,d1  : string;
    sig1   : char;

    function nb_elems(s : string;var g,d : string;s1,s2 : char;
                 var sig : char) : integer;
    begin.... end;

    procedure traiter_sans_parentheses(courant : noeud;s : string;
                 signe1,signe2 : char);
    begin.... end;

begin
if nb_elems(s,g1,d1,signe1,signe2,sig1)=2 then  // deux termes ou plus
   begin
   courant.FilsDroit:=noeud.create;
   remplit(courant.FilsDroit,d1,'*','/');
   courant.op:=sig1;
   courant.FilsGauche:=noeud.create;
   remplit(courant.FilsGauche,g1,signe1,signe2);
   end
else  // un seul terme
   if nb_elems(s,g1,d1,'*','/',sig1)=2 then  // un terme pour l'addition mais
      remplit(courant,s,'*','/')    // plusieurs pour la multiplication : 2*3*(-5)
   else
      if d1[1]='(' then   // ou bien une seule parenthèse, ex : (2+3)
         begin
         courant.op:='(';
         courant.FilsDroit:=noeud.create;
         d1:=copy(d1,2,length(d1)-2);  // on supprime les parenthèses
         Remplit(courant.FilsDroit,d1,'+','-')
         end
      else              // ou bien un seul nombre
         traiter_sans_parentheses(courant,d1,'+','-');
end;
```

La fonction d'évaluation de l'arbre binaire devient donc :

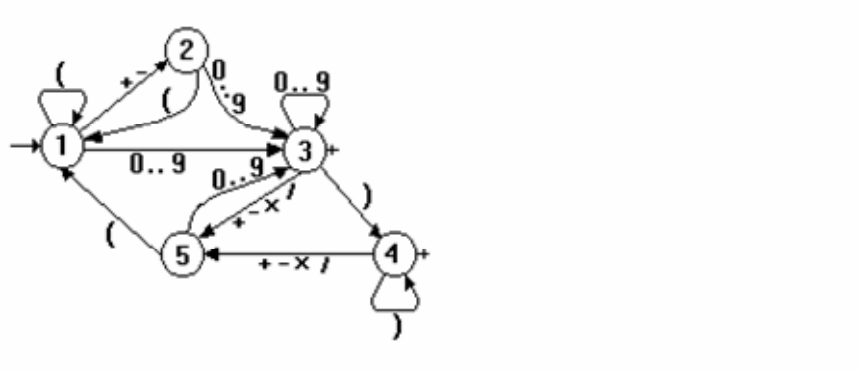
```
function arbre.evalue(n:noeud):integer;    //récursive, évidemment
begin
     if n=nil then evalue:=0 else
     if n.op='c' then // c'est une feuille "chiffre"
        evalue:=n.valeur
     else begin       // noeud opérateur
          case n.op of
               '(': evalue:=evalue(n.FilsDroit);
               '+': evalue:=evalue(n.FilsGauche)+evalue(n.FilsDroit);
               '-': evalue:=evalue(n.FilsGauche)-evalue(n.FilsDroit);
               '*': evalue:=evalue(n.FilsGauche)*evalue(n.FilsDroit);
               '/': evalue:=evalue(n.FilsGauche) div evalue(n.FilsDroit);
          end;
     end;
end;
```

Nous allons voir maintenant comment on construit un automate permettant de vérifier qu'une expression saisie au clavier est correcte d'un point de vue syntaxique.

Automate de validation d'une expression arithmétique

Avant de soumettre une expression à notre calculette, nous devons en vérifier la validité (parenthèses correctement placées et pas d'erreurs syntaxiques).

Pour cela, nous allons utiliser un automate. En voici le schéma :



On va de l'état "1" à l'état "2" par un signe "+" ou "-".

Ce schéma comporte des cellules circulaires appelées *états* et des flèches appelées arcs ou ***transitions***. Au dessus de chaque flèche se trouve une étiquette indiquant quels sont les caractères qui provoquent la transition.

L'état 1 est appelé "***état initial***" et il est identifiable par une flèche entrante.

Les états 3 et 4 sont appelés "***états terminaux***" et sont identifiables par une croix (ou un "+") placée à côté.

L'automate s'utilise en lui passant un par un, dans le sens de lecture naturel (de gauche à droite), les caractères qui constituent la chaîne à valider.

Certains états présentent des transitions sous forme de boucles, indiquant qu'on peut rester dans le même état en examinant deux caractères consécutifs : on va de l'état "1" à l'état "1" par "(" : cela indique qu'on peut avoir plusieurs parenthèses ouvrantes d'affilée : "((".

Appliquons cet automate pour voir si l'expression « 1+2 » est correcte.

Le premier caractère de l'expression est « 1 ». Nous soumettons ce caractère à l'automate qui se trouve alors dans l'état **1**. Le chiffre "1" étant un entier entre 0 et 9, nous suivons la transition correspondante qui amène l'automate dans l'état **3**.

Le caractère suivant est « + ». L'automate passe dans l'état **5**.

Le dernier caractère est « 2 ». L'automate passe dans l'état **3**.

C'est un état terminal, l'expression envisagée était donc correcte.

Si l'expression avait été « 1+ », nous serions resté dans l'état **5**, qui n'est pas terminal, et l'expression aurait donc été jugée incorrecte.

Si l'expression avait été « 1+/ », nous n'aurions pu aller nulle part depuis l'état **5** (il n'y a aucune transition possible depuis l'état **5** avec le caractère « / ») et l'expression aurait également été jugée incorrecte.

Une expression est donc incorrecte dans deux cas :

elle laisse finalement l'automate dans un état non terminal

elle comporte un caractère qui, dans un état donné, ne provoque aucune transition.

Pour représenter l'automate, nous utilisons un tableau à deux dimensions :

```
const etats = 5;  //  nombre d'états de l'automate
      caracs = 16;   // 16 caractères autorisés pour la saisie
      st = '0123456789+-*/()';  // les 16 caractères
      tab : array[1..etats,1..caracs] of integer =
          //0 1 2 3 4 5 6 7 8 9 + - * / ( )
          ((3,3,3,3,3,3,3,3,3,3,2,2,0,0,1,0),  {etat 1}
           (3,3,3,3,3,3,3,3,3,3,0,0,0,0,1,0),  {etat 2}
           (3,3,3,3,3,3,3,3,3,3,5,5,5,5,0,4),  {etat 3}
           (0,0,0,0,0,0,0,0,0,0,5,5,5,5,0,4),  {etat 4}
           (3,3,3,3,3,3,3,3,3,3,0,0,0,0,1,0)); {etat 5}
```

Chaque case du tableau indique quel état est amené par la transition provoquée par le caractère en cours en partant de l'état courant :

etat :=tableau[etat,caractere];

Si, par exemple, l'automate se trouve dans l'état 2 (ligne 2 du tableau) et qu'on lit une parenthèse ouvrante, alors l'automate passe dans l'état 1 (donc, pour le prochain caractère on lira dans la ligne 1 du tableau). Si l'automate se trouve dans l'état 1 et qu'on lit un « / », alors l'automate passe dans l'état 0 (il y a une erreur) et on s'arrête.

Les zéros indiquent qu'un caractère donné ne provoque aucune transition à partir d'un état donné (impasse).

Il ne nous reste plus qu'à soumettre à notre automate l'expression à valider, caractère par caractère.

Label1.Caption:=Automate(Edit1.Text);

Si, à un moment quelconque de l'analyse de notre expression mathématique, etat=0, alors nous sommes dans une impasse, et l'expression est donc incorrecte.

Si, à la fin, l'automate ne se trouve pas dans l'état 3 ou l'état 4 (les seuls états terminaux), l'expression est incorrecte.

Nous avons ajouté une variable « equilibre », chargée de mesurer la différence entre le nombre de parenthèses ouvrantes et le nombre de parenthèses fermantes.

En effet, on démontre mathématiquement qu'un automate du type de celui que nous utilisons n'est pas capable de reconnaître les fautes de parenthésage. Il faut utiliser un automate "à pile" (muni d'un compteur). La variable "equilibre" joue ce rôle de compteur.

Au début equilibre=0 (il y a équilibre, puisqu'il n'y a aucune parenthèse).

Pour chaque parenthèse ouvrante trouvée, nous ajoutons 1 à la variable "equilibre".

Pour chaque parenthèse fermante trouvée, nous retranchons 1 à la variable "equilibre".

Ainsi, trois cas peuvent se présenter :

1. En cours de validation, "equilibre" est strictement négatif : cela signifie que nous avons rencontré plus de fermantes que d'ouvrantes. Inutile d'aller plus loin : l'expression est incorrecte.

2. En fin de validation, "equilibre" est nul : tout s'est bien passé au niveau des parenthèses, il y a autant d'ouvrantes que de fermantes, et elles ne sont pas placées n'importe comment. En effet, comme "equilibre" est évalué dans l'ordre de lecture, une séquence comme "())(" aurait déjà été rejetée - à cause d'un équilibre négatif - au cours de la validation. Si donc "equilibre" est nul, et que l'expression est incorrecte, le problème ne vient pas d'une faute de parenthésage.

3. En fin de validation, "equilibre" est strictement positif : cela signifie que nous avons rencontré plus d'ouvrantes que de fermantes : l'expression est incorrecte.

Voici la fonction "automate" programmée :

```
function Automate(s:string):string;
const etats = 5;
      caracs = 16;   // 16 caractères autorisés pour la saisie
      st = '0123456789+-*/()';  // les 16 caractères
      tab : array[1..etats,1..caracs] of integer =
          //0 1 2 3 4 5 6 7 8 9 + - * / ( )
          ((3,3,3,3,3,3,3,3,3,3,2,2,0,0,1,0),  {etat 1}
           (3,3,3,3,3,3,3,3,3,3,0,0,0,0,1,0),  {etat 2}
           (3,3,3,3,3,3,3,3,3,3,5,5,5,5,0,4),  {etat 3}
           (0,0,0,0,0,0,0,0,0,0,5,5,5,5,0,4),  {etat 4}
           (3,3,3,3,3,3,3,3,3,3,0,0,0,0,1,0)); {etat 5}

var p,n:integer; car:char; etat,equilibre:integer;
begin
if s='' then
   begin
   result:='Chaîne vide';
   exit;
   end;
equilibre:=0; //comptage des parenthèses
p:=0; //position courante dans la chaîne s
etat:=1;
repeat
    inc(p);
    car:=s[p];  //caractère en cours
    if car='(' then inc(equilibre);
    if car=')' then dec(equilibre);
    if equilibre>=0 then
       begin
       n:=pos(car,st);
       // la position du caractère en cours dans la chaîne de caractères autorisés
       if n>0 then etat:=tab[etat,n]; //c'est un caractère autorisé
       end;
until (p=length(s)) or (equilibre<0) or (n=0) or (etat=0);
if equilibre<0 then
   result:='Il y a une parenthèse fermante en trop à la position '+inttostr(p)
else
   if (equilibre>0) then
      result:='Il y a une parenthèse ouvrante en trop'
   else
      if n=0 then
         result:='Caractère non autorisé à la position '+inttostr(p)
      else
         if etat=0 then
            result:='Expression incorrecte (erreur à la position '+inttostr(p)+')'
         else
            if (etat<>3) and (etat<>4) then
               result:='Expression incorrecte (etat final non terminal)'
            else
               result:='Expression correcte';
end;
```

Chapitre VII-4

Les arbres n-aires : codage d'un dictionnaire

Nous allons étudier dans ce Chapitre la façon de coder un dictionnaire dans un arbre n-aire (chaque noeud de l'arbre possède n branches).

Structure des données

Pour commencer, nous allons étudier la structure d'un noeud. On suppose que le dictionnaire est en majuscules, c'est pourquoi nous n'avons que 26 lettres possibles pour un noeud. Bien évidemment, cette structure peut être élargie si on veut aussi prendre en compte les minuscules et les chiffres ou autres signes. Nous voulons aussi afficher l'arbre n-aire à l'écran (sur un exemple comportant seulement quelques mots), ce qui implique que chaque noeud aura une position à l'écran, matérialisée par deux coordonnées x et y. Chaque noeud contient une lettre, donc une variable de type char, et il doit contenir aussi un indicateur boolean utilisé pour indiquer si un mot est entier : par exemple, supposons qu'on code le mot "CARTE" dans un arbre; l'arbre aura donc 5 niveaux. La dernière lettre du mot (le "E") aura son indicateur boolean positionné à true. Par contre, la lettre "A" n'aura pas son indicateur positionné, car le mot "CA" n'est pas un mot du dictionnaire. La lettre "R" aura son indicateur positionné, car le mot "CAR" fait partie du dictionnaire de la langue française.

Un noeud possède donc:

- 26 fils, les lettres possibles de 'A' à 'Z'

- les coordonnées (x,y) qui seront utilisées pour l'affichage graphique de l'arbre

- une lettre

- un indicateur boolean

```
Noeud=
   class
      private
         Fils    : array['A'..'Z'] of Noeud;  //liste de fils
         x,y : integer;  // pour dessin arbre
         Lettre  : char;       // lettre contenue dans ce noeud de l'arbre
         Entier  : boolean;     //flag indiquant si le mot est entier
      public
         constructor create;
         destructor destroy;
   end; { Noeud }
```

Etudions maintenant la structure d'un arbre. Comme nous l'avons déjà vu, un arbre est constitué par sa tête, et par une autre variable nous permettant de parcourir l'arbre. Les opérations relatives à un arbre sont les suivantes :

- ajout d'un mot

- chargement d'un fichier dans l'arbre en utilisant l'opération d'ajout

- compter le nombre de noeuds terminaux se trouvant en-dessous d'un noeud quelconque

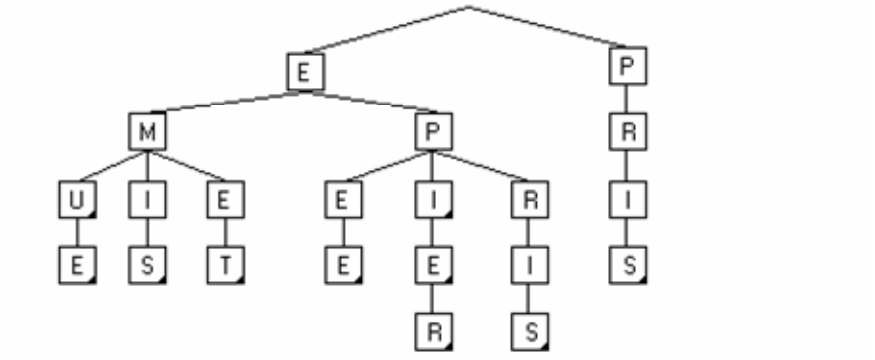
- l'affichage graphique et l'affichage texte (création d'une liste de mots du dictionnaire)

- recherche d'un mot

Dans notre exemple, nous avons choisi une liste de 13 mots de 5 lettres chacun. A l'aide de la fonction qui compte le nombre de noeuds terminaux, on peut donc dire combien de mots comporte l'arbre.

```
//arbre d'ordre 26
Arbre =
  class
     private
        tete,courant : noeud;
     public
        constructor create;
        destructor  destroy;override;
        procedure   rajouter_mot(LeMot : string);
        procedure   charger(NomFic : string);
        procedure   Affiche(S,prefixe:string);
        procedure   dessine(en_cours : noeud;limg,old_x,old_y : integer);
        function    Trouve(LeMot: string) : boolean;
        function    compter_terminaux(en_cours : noeud) : integer;
  end;
```

Nous allons voir, une par une, toutes les procédures décrites ci-dessus. Mais auparavant, voici un exemple concret d'arbre n-aire qui contient des mots de longueur 3 (EMU,EPI), de longueur 4 (EMUE, EMIS, EMET, EPIE, PRIS) et de longueur 5 (EPIER, EPRIS), dont l'indicateur est positionné à true, et qui est représenté sur ce schéma par un petit triangle noir dans le coin des cases en bas à droite.



La procédure d'ajout d'un mot

On part du sommet de l'arbre. La procédure est récursive, et comme toute procédure récursive, elle a un point d'appui (ou d'arrêt). On va ajouter le mot dans l'arbre lettre par lettre, en descendant dans l'arbre d'un niveau à chaque lettre. Le mot est donc "mangé" progressivement du début vers la fin.

1) Le point d'arrêt est atteint quand on n'a plus de lettres à ajouter dans l'arbre. Dans ce cas, on positionne le noeud courant à true, pour indiquer que c'est un mot complet et on quitte la procédure.

2) On a encore des lettres à ajouter dans l'arbre. Prenons le cas du mot "EPI". On est au sommet de l'arbre. Si la lettre "E" existe déjà dans l'arbre en tant que noeud fils du sommet, alors :

- on "mange" le "E"; le mot à rajouter devient "PI" mais il faut le mettre sous le "E"

- on descend sur le fils "E"

- on rajoute (récursivement) le mot "PI"

Si la lettre "E" n'existe pas, alors :

- on crée le noeud fils "E"

- on exécute les étapes ci-dessus.

Voici le programme correspondant :

```
procedure arbre.rajouter_mot(LeMot : string);
var lettre : char;
begin
if LeMot='' then
   begin
   courant.entier:=true;
   exit;
   end;
lettre:=LeMot[1];
if courant.fils[lettre]<>nil then  // si la lettre existe déjà
   courant:=courant.fils[lettre]   // alors on se positionne sur la lettre suivante
else  // sinon il faut créer cette lettre dans l'arbre
   begin
   courant.fils[lettre]:=noeud.create;
   courant:=courant.fils[lettre];
   courant.lettre:=lettre;  // la lettre est maintenant dans l'arbre
   end;
delete(LeMot,1,1);  // on efface la lettre du mot puisqu'elle est déjà dans l'arbre
rajouter_mot(LeMot); // et on rajoute le reste
end;
```

La procédure de chargement d'un fichier

Le fichier étant un simple fichier texte, il suffit de le lire ligne par ligne et d'ajouter au fur et à mesure les mots lus dans l'arbre.

```
procedure arbre.charger(NomFic : string);
var   s : string;
      chemin : string; // le chemin de l'appli, AVEC l'antislash final
      f : textfile;
begin
chemin:=ExtractFilePath(Application.ExeName);
assignFile(f,chemin+nomFic);
reset(f);
repeat
    readln(f,s);
    courant:=tete;  // on se positionne en tête de l'arbre
    rajouter_mot(s);  // et on rajoute le mot
until eof(f);
closeFile(f);
end;
```

Comptage des noeuds terminaux

Pour dessiner un arbre, nous avons besoin de cette fonction. En effet, si un père a deux noeuds terminaux, sa position à l'écran sera au milieu de ses deux fils (il s'agit de la position sur l'axe des abscisses, les "x"; en tant que père il sera positionné au dessus de ses fils sur l'axe des ordonnées,

les "y"). Cela est valable aussi pour un père qui n'a pas directement de noeuds terminaux, mais dont les fils ou sous-fils en ont. Il faut donc parcourir récursivement le sous-arbre dont le père est le noeud courant pour arriver jusqu'aux noeuds terminaux.

Une solution serait la suivante :

- la somme des noeuds terminaux ayant pour ancêtre un noeud donné est :

    - égale à un si le noeud en question n'a pas de fils (point d'arrêt)

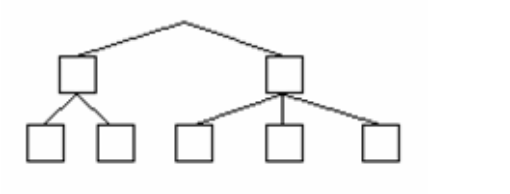
    - égale à la somme des noeuds terminaux de ses fils sinon

Mais, dans ce cas, on peut supprimer le point d'arrêt et faire fusionner les deux conditions. En effet, on n'a qu'à parcourir tous les fils du noeud en cours et, s'ils ne sont pas égaux à nil, incrémenter alors une variable locale avec leurs propres noeuds terminaux. Si, à la fin, le total obtenu est égal à zéro, alors aucun des fils du noeud en cours n'existe (point d'arrêt de la récursivité) donc on initialise la variable locale à 1 et on sort.

```
function arbre.compter_terminaux(en_cours : noeud) : integer;
var i     : char;
    total : integer;
begin
total:=0;
for i:='A' to 'Z' do
    if en_cours.fils[i]<>nil then
       inc(total,compter_terminaux(en_cours.fils[i]));
if total=0 then inc(total);
compter_terminaux:=total;
end;
```

Affichage graphique d'un arbre n-aire

La procédure d'affichage d'un arbre nécessite, pour commencer, une image et ensuite quelques petits calculs mathématiques, ainsi que le nombre total de noeuds terminaux qui se trouvent en-dessous du noeud courant. La première chose à faire est de compter le nombre de noeuds terminaux dérivant du noeud en cours, afin de décider où il sera placé graphiquement, ceci nous amenant donc au calcul de ses coordonnées. Ensuite pour chaque noeud dessiné, on doit tracer une ligne le reliant à son père, ce qui nous amène à l'utilisation de 2 paramètres (old_x,old_y) de type integer. On suppose que chaque noeud occupe 50 pixels en largeur et 80 en hauteur. Ainsi, si un noeud a 4 fils, sa position sera à la moitié de 4*50, soit " (4*50) div 2". S'il n'a que 2 fils, il sera à " (2*50) div 2". Mais comment faire pour que les noeuds d'un même niveau se décalent vers la droite ? En effet dans le dessin suivant, au niveau 1, nous avons deux fils, le premier ayant 2 fils terminaux et le deuxième trois fils terminaux.



Si on applique la formule ci-dessus, alors le premier noeud du niveau 1 est situé à "x:=(2*50)/2" (soit x=50) et le deuxième est situé à "x:=(3*50)/2" (soit x=75). On constate alors que ces deux noeuds se

chevauchent, puisque le premier est situé à x=50 et il occupe 50 pixels (jusqu'à cent; or le deuxième est situé à x=75). Donc, à chaque dessin d'une partie de l'arbre, il faut voir quelle est la largeur qu'il requiert et la passer en paramètre à ses frères, de façon à ce que ceux-ci commencent leur dessin avec un décalage vers la gauche. Ainsi dans l'exemple ci-dessus, le premier noeud nécessite 100 pixels (car il a deux fils terminaux, qui seront dessinés à partir de la position 0), et le deuxième noeud nécessite 150 pixels (car il a trois noeuds terminaux, qui seront dessinés à partir de la position 100). Ceci nous amène à un nouveau paramètre, appelé "limg" qui indique la position à partir de laquelle on doit dessiner le sous arbre du noeud en cours. On effectue ensuite les opérations suivantes :

- on calcule la position du noeud en cours (x et y)

- on relie ce noeud à son père

- on parcourt tous ses fils et pour chaque fils non vide on dessine le sous-arbre correspondant en faisant un appel récursif.

```
procedure arbre.dessine(en_cours : noeud;limg,old_x,old_y : integer);
var x,y,nb : integer;
    i      : char;
begin
//nb:=compter_terminaux(courant); // effet joli
nb:=compter_terminaux(en_cours);
x:=limg+(50*nb) div 2;
y:=old_y+80;
if en_cours<>tete then
   with form1.image1.picture.bitmap.canvas do
      begin
      textout(x,y,en_cours.lettre);
      en_cours.x:=x;en_cours.y:=y;
      moveto(x,y-5);lineto(old_x,old_y);
      end;
for i:='A' to 'Z' do
    if en_cours.fils[i]<>nil then
       begin
       nb:=compter_terminaux(en_cours.fils[i]);
       dessine(en_cours.fils[i],limg,x,y+20);
       limg:=limg+nb*50;
       end;
end;
```

Affichage de texte (création d'une liste de mots du dictionnaire)

Pour réaliser un affichage de tous les mots de l'arbre, on n'a qu'à parcourir ses noeuds et quand on arrive à un noeud dont l'indicateur boolean est positionné, on ajoute le mot à une liste. Ensuite, on parcourt tous les fils du noeud en cours en faisant un appel récursif. Comme les mots se constituent au fur et à mesure qu'on descend dans l'arbre, on utilise un paramètre à cet effet.

```
procedure arbre.Affiche(S: string);
var i   : char;
    aux : noeud;
begin
if courant.Entier then Form1.Memo1.Lines.Add(s);
for i := 'A' to 'Z' do
    if courant.fils[i]<>nil then
       begin
       aux:=courant;
       courant:=courant.fils[i];
       Affiche(S+i);
       courant:=aux;
       end;
end;
```

Recherche d'un mot dans l'arbre

La recherche se fait par une descente récursive dans l'arbre; à chaque descente, on regarde si le noeud en cours a un fils qui est identique à la première lettre du mot. Si c'est le cas, alors on "mange" le début du mot (la première lettre) et on cherche à partir de la position courante le mot restant. Si, à un moment, pendant l'exploration de l'arbre, on se retrouve avec un mot vide, alors le mot appartient à l'arbre. Attention : cette condition n'est valable que pour les arbres dont tous les mots ont la même longueur. Sinon, il faut rajouter une condition supplémentaire, et cette tâche vous revient.

```
function arbre.trouve(LeMot : string) : boolean;
var lettre : char;
begin
if LeMot='' then
   trouve:=true
else
   begin
   lettre:=LeMot[1];
   delete(LeMot,1,1);
   if courant.fils[lettre]=nil then
      trouve:=false
   else
      begin
      courant:=courant.fils[lettre];
      trouve:=trouve(LeMot);
      end;
   end;
end;
```

Chapitre VIII - Dérécursification

Chapitre VIII-1

Dérécursification de la procédure de Fibonacci

Nous allons voir, pour commencer, un exemple simple de dérécursification. Nous allons prendre pour exemple la suite de Fibonacci, dont voici la définition :

f (1) =1

f (2) =1

f (n) = f (n-1) + f (n-2), pour tout entier naturel n strictement supérieur à 2.

Sa traduction en langage informatique nous donne :

```
function fibo_rec(n : integer) : integer;
begin
if (n<3) then fibo_rec:=1
else
   fibo_rec:=fibo_rec(n-1)+fibo_rec(n-2);
end;
```

Pour transformer cette procédure récursive en une procédure itérative, nous devons d'abord regarder quelles sont les valeurs successives de cette suite :

1,1,2,3,5,8,13,21,34,55,...

D'après la définition, on voit que chaque terme de la suite est une somme de deux nombres (f(n-1) et f(n-2)).

Nous devons donc avoir deux variables dans notre procédure itérative. Nous allons les appeler n1 et n2. Ces deux nombres verront leur valeur s'incrémenter progressivement.

Essais et observations

1) On veut calculer f(3). n1=1, n2=1, donc f:=n1+n2=2;

2) On veut calculer f(4); n1=1; n2=2; f (4)=1+2=3

3) On veut calculer f(5) : n1=2; n2=3; f(5)=2+3=5

Pour une valeur quelconque x, f(x)=n1+n2; il faut donc faire une boucle dans laquelle nous allons modifier progressivement les valeurs de n1 et n2.

Ainsi pour le cas 1) ci-dessus, on a algorithmiquement :

```
n1=1; n2=1; f:=n1+n2=2
```

Pour le cas 2) ci-dessus on a algorithmiquement:

```
n1=1; n2=1;  //--- valeurs de base
aux:=n2;  //--- sauvegarde de la valeur de n2
n2:=n1+n2=2;   //--- calcul de f(n-1)+f(n-2)
n1:=aux=1;  //--- nouvelle valeur de n1, correspondant à l'ancienne valeur de n2
avant la modif de la ligne du dessus
```

f:=n1+n2=3

Pour le cas 3) ci-dessus on a :

```
n1=1; n2=1;  //--- valeurs de base
aux:=n2;  //--- sauvegarde de la valeur de n2
n2:=n1+n2=2;   //--- calcul de f(3)=f(2)+f(1)
n1:=aux=1;  //--- nouvelle valeur de n1, correspondant à l'ancienne valeur de n2
avant la modif de la ligne du dessus
aux:=n2;  //--- sauvegarde de la valeur de n2
n2:=n1+n2=3;   //--- calcul de f(4)=f(3)+f(2)
n1:=aux=2;  //--- nouvelle valeur de n1, correspondant à l'ancienne valeur de n2
avant la modif de la ligne du dessus
```

f:=n1+n2=5

On remarque que le groupe de 3 lignes peut très bien constituer une boucle répétée deux fois. Donc on peut la généraliser x fois si nécessaire; dans ce cas nous obtenons le programme suivant:

```
function fibo_it(n : integer) : integer;
var i,aux,n1,n2 : integer;
begin
n1:=1;n2:=1;
for i:=3 to 10 do
    begin
    aux:=n1+n2;
    n1:=n2;
    n2:=aux;
    end;
fibo_it:=aux;
end;
```

Comme on peut le constater, la dérécursification est parfois très simple, elle revient à écrire une boucle, à condition d'avoir bien fait attention à l'évolution des variables au cours de l'éxécution. Mais parfois elle est extrêmement difficile à mettre en oeuvre, comme c'est le cas pour certaines fractales ou les tours de Hanoï.

Chapitre VIII-2

La dérécursification des structures de données

Nous savons que la structure d'un arbre binaire est récursive. Nous allons voir comment on peut implémenter cette structure récursive dans une structure statique (c'est à dire un tableau de taille fixe). Voici d'abord la structure dynamique de l'arbre binaire (telle que nous l'avons vue dans le chapitre précédent) :

```
type Noeud=
        class
           private
              Fils    : array['A'..'Z'] of Noeud;  //liste de fils
              x,y : integer;  // pour dessin arbre
              Gauche,Droite : Noeud;
              Lettre  : char;       // lettre contenue dans ce noeud de l'arbre
              Entier  : boolean;     //flag indiquant si le mot est entier
           public
              constructor create;
              destructor destroy;
        end; { Noeud }
```

Nous allons maintenant transformer cette structure dynamique en une structure statique, donc linéaire. Pour cela nous allons utiliser un tableau, dont chaque case a une structure représentée par :

- "lettre" : une variable de type "char"

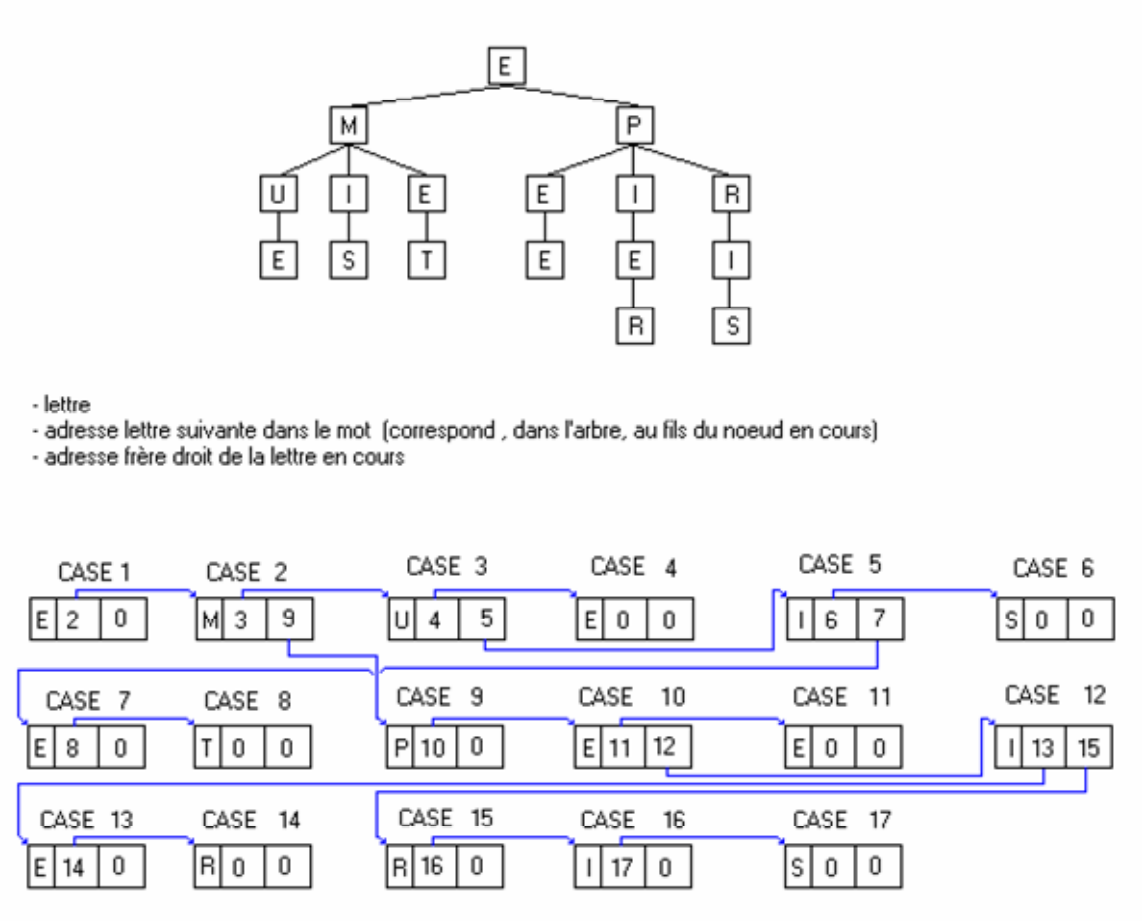
- "fin" : une variable de type booléen, mais représentée ici par un byte, qui indique que le mot est entier

- "droite","bas" : représentent les deux fils gauche et droit de la structure dynamique.

```
type trec = record
         lettre : char;
         fin : byte;
         droite,bas : longint;
         end;

     dictionnaire = record
         t : array[1..5000] of trec;
         derniere_case : longint;
         end;

var  dico : dictionnaire;
```

Nous avons utilisé un tableau de 5.000 cases; mais en plus le dictionnaire contient aussi une variable "derniere_case", de façon à ce qu'on sache où on doit implémenter dans le tableau tous les nouveaux mots.

Nous allons utiliser dans le programme joint sur le CD des mots de la même longueur, mais néanmoins, pour expliquer le principe de l'implémentation dans un tableau, nous utiliserons des mots de longueur différente. Voici une image illustrant parfaitement les types prédéfinis ci-dessus, ainsi que l'implémentation d'un petit arbre n-aire :



Nous allons décrire brièvement une recherche de mot dans ce tableau. Supposons qu'on veuille trouver le mot "EPRIS". Nus sommes au début sur la case 1, donc sur la lettre E. On cherche ses fils. Le premier est à la case 2 et c'est M. Mais nous, on cherche la lettre P, donc on va se positionner sur le frère de M (M,3,9) qui est à la case 9. C'est bien un P, donc on descend maintenant sur son premier fils qui se trouve à la case 10 (car P,10,0). On trouve un E, qui ne nous intéresse pas, donc on va sur le frère de E qui se trouve à la case 12 (car E,11,12). C'est un I, il ne nous intéresse donc pas et on se positionne sur son frère qui se trouve à la case 15 (car I,13,15) et c'est bien un R. Le fils de R se trouve à la case 16 (car R,16,0) et c'est un I; le fils de I est à la case 17 et c'est un S. On a donc bien le mot EPRIS, car derrière le S il n'y a plus de fils, sinon le mot aurait été incomplet. On utilise aussi la variable "fin" pour indiquer que le mot est complet, auquel cas on peut ajouter d'autres lettres derrière ce mot (exemple : EPI et EPIER; les cases I et R ont la variable "fin" positionnée à 1, les autres à 0).

Voici la liste des procédures que nous allons utiliser :

1) - procédure introduis_mot_fin_dico(mot : string;p : longint;bas : boolean);

2) - function positionnement_sur_lettre(lettre : char;courant : longint; var dernier : longint) : longint;

3) - procédure rajoute_mot(mot : string);

4) - function appartient(mot : string) : boolean;

La procédure 1) introduit un mot (ou la fin d'un mot) à la fin du dictionnaire. Elle a plusieurs paramètres :

- le mot à introduire dans le tableau

- la position à partir de laquelle on introduit le mot dans les cases du tableau

- une variable booléenne indiquant si le mot sera introduit en tant que fils de la case en cours (c'est donc la case "droite" qui est initialisée, cela correspondant à la lettre suivante dans le mot) ou bien en tant que frère (c'est donc la case "bas" qui sera initialisée);

exemple 1 : si on est sur la lettre I de EPIER et si on veut introduire la mot EPRIS, alors la lettre R sera introduite à droite de I ***en tant que fils de E.*** *Si par la suite on cherche le mot EPRIS, on tombe premièrement sur le I, et comme il ne nous intéresse pas on prend le fils suivant de son père, donc R, suivi de I et de S.*

exemple 2 : si on a introduit le mot EMIS, et si on est sur la lettre E et si on veut introduire le mot ROND, alors la lettre R sera mise en "bas" du E, ***en tant que frère de E****.*

Il faut savoir que dans tous les cas où l'on introduit un mot ou la fin d'un mot (pour le mot EPRIS on n'a introduit que RIS, car EP avait été introduit avec le mot EPEE), cette insertion se fera automatiquement à la fin du tableau (c'est à dire là où il y a des cases non remplies), donc à partir de la case numéro "derniere_case".

```
procedure introduis_mot_fin_dico(mot : string;p : longint;bas : boolean);
var l,courant : longint;
begin
if bas
   then dico.t[p].bas:=dico.derniere_case
   else dico.t[p].droite:=dico.derniere_case;
l:=length(mot);
for courant:=1 to l do
    with dico.t[dico.derniere_case+courant-1] do
       begin
       lettre:=mot[courant];
       fin:=0;
       droite:=dico.derniere_case+courant;
       bas:=0;
       end;
dico.t[dico.derniere_case+l-1].droite:=0;  // plus rien derrière
dico.t[dico.derniere_case+l-1].fin:=1;  // c'est un mot
inc(dico.derniere_case,l);
end;
```

La fonction 2) sert de positionnement sur une lettre, c'est à dire à trouver la case à partir de laquelle un mot sera introduit dans le tableau. Cette fonction a 3 paramètres :

- la lettre sur laquelle on veut se positionner

- la position à partir de laquelle on commence la recherche dans les cases

- la position de la dernière lettre trouvée (on a EPEE et on veut rajouter EPRIS; la dernière lettre trouvée est P et sa position dans le tableau sera retournée, car on va rajouter un R en tant que fils de P)

```
function positionnement_sur_lettre(lettre : char;courant : longint;
    var dernier : longint ) : longint;
var l : char;
begin
l:=dico.t[courant].lettre;
if (courant>0) and (l=lettre) then dernier:=courant;
while (courant<>0) and (l<>lettre) do
   begin
   courant:=dico.t[courant].bas;
   l:=dico.t[courant].lettre;
   if (courant>0)
       and (l=lettre)
      then dernier:=courant;  // mémorise la dernière position trouvée où l=lettre}
   end;
positionnement_sur_lettre:=courant;
end;
```

La procédure 3) représente le rajout d'un mot dans le tableau. Cette procédure utilise la procédure 1) et la fonction 2). Lors d'un rajout de mot dans notre tableau, nous devons réaliser dans l'ordre les opérations suivantes:

a) - nous positionner sur le noeud dont la lettre est identique à la première lettre du mot. Si le fils existe, alors on doit récupérer son numéro de case dans le tableau et recommencer avec la lettre suivante, sinon on doit récupérer le numéro de la case qui sera pointée par la case en cours.

exemple : dans le tableau décrit au dessin précédent, on a rempli les huit premières cases avec les mots EMUE,EMIS,EMET. La variable "derniere_case" est donc égale à 9. Si on veut rajouter EPEE, on doit récupérer le numéro 2, car P est le frère de M, et ainsi le numéro 9 se retrouve dans la case 2 : M,3,9. La case 9 est pointée par la case 2.

b) - arrivés ici, nous avons un choix à faire: ou bien nous avons trouvé au moins un début de mot qui était déjà dans le tableau (comme EP pour EPEE et EPRIS) et dans ce cas la variable "courant" est positive, ou bien on doit rajouter le mot en entier auquel cas la variable "courant" est nulle.

Si on a trouvé un début de mot, alors on est sur une case quelconque du tableau et on la fait pointer sur la case de numéro "derniere_case" où l'on rajoutera la suite du mot (exemple : on a dans le tableau le mot EPEE et on veut rajouter EPRIS. Le P pointe sur E et E n'a pas de frère, donc sa variable "bas" va pointer sur "derniere_case" où on va introduire le reste du mot, à savoir RIS).

Voici l'algorithme correspondant:

```
procedure rajoute_mot(mot : string);
var i,l,courant,p,precedent,dernier : longint;
    dedans : boolean;
begin
l:=length(mot);courant:=1;i:=1;dernier:=0;
p:=positionnement_sur_lettre(mot[1],1,dernier);
dedans:=true;
while (i<=l) and (p>0) and
      (dedans) and
      (courant>0) and
      (i<dico.derniere_case) do
   begin
   p:=positionnement_sur_lettre(mot[i],courant,dernier);
   if p>0 then courant:=dico.t[p].droite;
```

```
   dedans:=(dernier+1)<dico.derniere_case;
   inc(i);
   end;
if courant>0 then  //donc p=0
   begin
   dec(i);
   if dernier=0 then  // cas spécial
      begin
      if dico.derniere_case=1
         then introduis_mot_fin_dico(mot,1,false)
      else
         begin  // positionnement sur la dernière lettre de la colonne d'un arbre
         p:=courant;
         while p<>0 do
            begin
            p:=dico.t[p].bas;
            if p>0 then courant:=p;
            end;
         introduis_mot_fin_dico(mot,courant,true)
         end;
      end
   else
      if dernier+1=dico.derniere_case then
         if mot[i]=dico.t[dernier].lettre then
            introduis_mot_fin_dico(copy(mot,i,l),dernier,false)
         else
            introduis_mot_fin_dico(copy(mot,i,l),dernier,true)
      else
         begin  // positionnement sur la dernière lettre de la colonne d'un arbre
         p:=courant;
         while p<>0 do
            begin
            p:=dico.t[p].bas;
            if p>0 then courant:=p;
            end;
         introduis_mot_fin_dico(copy(mot,i,l),courant,true);
         end;
   end
else  // donc p<>0
   introduis_mot_fin_dico(copy(mot,i,l),dernier,false)
end;
```

Et maintenant il ne nous reste plus qu'à faire une fonction d'appartenance d'un mot. Elle est simple à réaliser; en effet, pour chacune des lettres du mot, à partir de la première et jusqu'à la dernière on appelle la fonction de positionnement sur une lettre donnée; si cette fonction échoue à un instant t, alors c'est que l'une des lettres du mot n'est pas dans les fils ou frères de la dernière lettre rencontrée; si, par contre, on arrive à la fin du mot, alors on doit tester que ce mot est bien terminé (la variable "fin" positionné à 1).

```
function appartient(mot : string) : boolean;
var i,l,courant,p : longint;
    dernier : longint;
begin
courant:=1;i:=1;p:=1;l:=length(mot);
while (i<=l) and (p>0) do
   begin
   p:=positionnement_sur_lettre(mot[i],courant,dernier);
   if p>0 then courant:=dico.t[p].droite;
   inc(i);
   end;
appartient:=(i>l) and (p>0) and (dico.t[p].fin=1);
end;
```

Chapitre VIII-3

Dérécursification du triangle de Pascal

On veut réaliser de façon itérative le programme qui affiche le triangle de Pascal. On suppose pour cela que nous pouvons afficher seulement 25 lignes, donc on utilise un tableau de 26 entiers. Bien évidemment, on peut modifier sa taille si on veut. Pour afficher les lignes du triangle de Pascal, on entre manuellement le nombre de lignes, et on fait ensuite une boucle:

- qui appelle la procédure de calcul des lignes

- qui les affiche.

On sait que, pour les deux premières lignes, le tableau ne contient que des 1, donc on va l'initialiser manuellement. La procédure de calcul se charge du reste. Le programme sera donc :

```
tab_coef[1]:=1;tab_coef[2]:=1;
for ligne:=2 to nombre_lignes do
    calculer(ligne);
```

On sait que, dans le triangle de Pascal, chaque coefficient est égal à la somme du nombre immédiatement supérieur et du nombre au dessus à gauche.

```
0 : 1           (a+b)^0 = 1
1 : 1 1         (a+b)^1 = 1*a + 1*b
2 : 1 2 1       (a+b)^2 = 1*a^2 + 2*a*b + 1*b^2
3 : 1 3 3 1     (a+b)^3 = 1*a^3 + 3*a^2*b + 3*a*b^2 + 1*b^3
4 : 1 4 6 4 1   (a+b)^4 = 1*a^4 + 4*a^3*b + 6*a^2*b^2 + 4*a*b^3 + 1*b^4
```

Nous n'avons ici qu'un tableau à une dimension que nous allons progressivement modifier. On dispose dès le départ de la ligne 1, et si on veut calculer la ligne 2 on n'a qu'à rajouter un 1 à la fin du tableau (qui correspond à la diagonale) et modifier le contenu du tableau en faisant les sommes qui conviennent.

Il faut faire attention : supposons qu'on a déjà calculé la ligne 2, et qu'on veut calculer la ligne 3. Le deuxième élément de la ligne 3 est égal à la somme du premier élément de la ligne 2 et du deuxième élément de la ligne 2. On le calcule et on le met dans le tableau, pour obtenir : 1 3 1 1. Et quand on calcule le troisième élément, on obtient 1 3 4 1 au lieu de 1 3 3 1.

L'erreur vient du fait qu'on n'a pas utilisé l'ancienne valeur "2" du tableau, mais la nouvelle valeur "3" et "3+1=4". Il faut donc parcourir le tableau en sens inverse. En effet :

- on a 1 2 1.

- on rajoute un 1 à la fin : 1 2 1 1

- on parcourt le tableau en sens inverse à partir de l'avant dernière position (la troisième position) :

```
tab_coef[3]:=tab_coef[2]+tab_coef[3] = 3 donc on a 1 2 3 1
```

- on avance vers le début (en deuxième position):

```
tab_coef[2]:=tab_coef[1]+tab_coef[2] = 3 donc on a 1 3 3 1
```

On remarque que la valeur est la bonne. Nous nous sommes arrêtés à la valeur 2, donc la boucle sera de la forme :

```
for i:=j downto 2 do
```

Voici donc le programme affichant le triangle de Pascal de façon itérative:

```
procedure TForm1.Button1Click(Sender: TObject);
var ligne : integer;
    tab_coef : array[1..26] of integer;

   procedure calculer(j : integer);
   var i : integer;
       s : string;
   begin
   tab_coef[j+1]:=1;
   for i:=j downto 2 do
       tab_coef[i]:=tab_coef[i-1]+tab_coef[i];
   s:='';
   for i:=1 to j+1 do s:=s+inttostr(tab_coef[i])+'  ';
   memo1.lines.add(s);
   end;
begin
if strtoint(edit1.text)>25 then
   begin
   showMessage('La valeur doit être inférieure à 25 !');
   exit;
   end;
fillchar(tab_coef,sizeof(tab_coef),#0);
tab_coef[1]:=1;tab_coef[2]:=1;
for ligne:=2 to strtoint(edit1.text) do
    calculer(ligne);


end;
```

Chapitre VIII-4

Dérécursification des "tours de Hanoï"

Nous allons réaliser un programme itératif pour "dérécursifier" le problème des tours de Hanoï. La difficulté vient du fait que le programme des tours de Hanoï contient un double appel récursif avec des paramètres différents : après avoir effectué le premier appel récursif, tout change (le nombre de disques, les piquets).

Nous avons vu dans les Chapitres précédents que, pour aboutir à une procédure récursive, on exécutait à la main un ensemble d'instructions, on regardait quelles étaient celles qui se répétaient, quelles étaient les variables utilisées, et on regroupait finalement les blocs identiques pour aboutir à une procédure récursive. Pour dérécursifier les tours de Hanoï, on va faire la même chose : on va réaliser à la main les mouvements des disques sur les 3 piquets, et on va essayer de trouver des similitudes ou des répétitions, afin d'en tirer un algorithme itératif.

Rappelons d'abord la définition récursive de la procédure :

```
H (n,depart,arrivee)
   H (n-1,depart,intermédiaire)
   Deplacer(depart,arrivee)
   H (n-1,intermédiaire,arrivée)
```

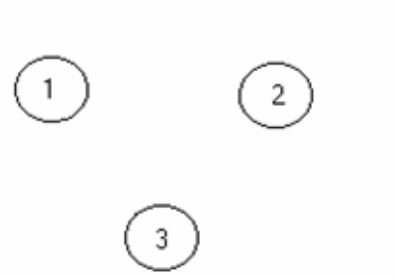
Préliminaire

Nous allons noter Hn le déplacement de *n* disques (Hn est en fait le programme récursif, qui se résume à l'appel de la procédure). On symbolise "départ" par "d", "arrivée" par "a" et "intermédiaire" par "i".

La définition s'écrit alors :

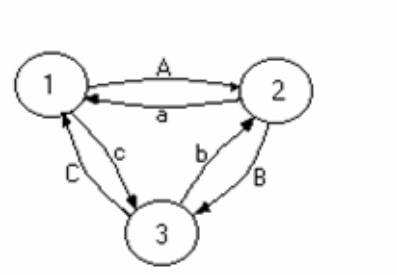
```
Hn (d,a)
   Hn-1 (d,i)
   Deplacer(d,a)
   Hn-1 (i,a)
```

Voici un dessin avec les trois piquets.



On décide que, si on a un nombre pair de disques à déplacer, on les déplace de 1 vers 3, et que, si on a un nombre impair de disques à déplacer, on les déplace de 1 vers 2. Cela est cohérent avec la définition récursive car, par exemple, pour déplacer cinq disques de 1 vers 3, en appliquant la définition, on déplace quatre disques (nombre pair) de 1

vers 2, puis un disque (nombre impair) de 1 vers 3 et enfin le reste de 2 vers 3.



On note :

A le mouvement de 1 vers 2.

a le mouvement de 2 vers 1.

B le mouvement de 2 vers 3.

b le mouvement de 3 vers 2.

C le mouvement de 3 vers 1.

c le mouvement de 1 vers 3.

On décide de dire que :

les mouvements *A* et *a* sont de *type 0*.

les mouvements C et *c* sont de *type 1*.

les mouvements *B* et *b* sont de *type 2*.

Etudions quelques mouvements des disques :

H1 = A (de 1 vers 2)

H2 = A c B (1 vers 2; 1 vers 3; 2 vers 3)

H3 = A c B A C b A

H4 = A c B A C b A c B a C B A c B

On remarque que Hn, pour n supérieur à 2, est constitué d'une suite répétée de trois mouvements de types 0, 1 et 2. En effet, si l'on écrit Hn en majuscules, on obtient ACB ACB ACB..., avec éventuellement un A à la fin, la suite des mouvements ne pouvant se terminer que par A ou B (d'après ce que nous avons dit plus haut concernant la convention que nous avons adoptée suivant la parité de n).

Notons Gn la suite des mouvements de Hn, en faisant abstraction du sens du déplacement (ce qui revient à noter tous les mouvements en majuscules) :

Gn = A C B A C B A C B... (le dernier caractère étant un A pour n impair et un B pour n pair) pour n supérieur à 2, et G1=A. Gn n'est autre que Hn auquel il manque l'information de sens (distingué par les majuscules et les minuscules).

Ecrivons quelques valeurs de Gn :

G1 = A

G2 = A C B

G3 = A C B A C B A

G4 = A C B A C B A C B A C B A C B

Détermination du type du k-ième mouvement

On constate que:

1) pour n pair, Gn est une suite de ACB

2) pour n impair, Gn est une suite de ACB terminée par un A

Notons k l'indice d'un mouvement de Hn. Nous convenons que l'indice du premier mouvement est 0 (par exemple, pour H3, le mouvement d'indice 0 est *A*, le mouvement d'indice 1 est *c*, etc.).

Comme les mouvements se répètent tout les trois mouvements, pour connaître le type du k-ième mouvement de **H**n, il suffit d'utiliser l'opérateur mathématique qui nous donne le reste de la division de k par 3, appelé opérateur *modulo* : *le type du k-ième mouvement de **H**n est donné par k mod 3, reste de la division entière de k par 3*.

Exemple 1

Quel est le quatrième mouvement de **H**4 ?

Il s'agit du mouvement numéro 3, car les mouvements sont numérotés à partir de 0. Comme *3 mod 3 = 0*, le quatrième mouvement de G4 est le même que le mouvement numéro 0, c'est à dire un A. Le quatrième mouvement de H4 est donc un mouvement de type 0. Nous ne savons pas encore en préciser le sens (A ou a).

Exemple 2

Quel est le sixième mouvement de **H**4 ?

Il s'agit du mouvement numéro 5, car les mouvements sont numérotés à partir de 0. Comme 5 *mod 3 = 2*, le sixième mouvement de G4 est le même que le mouvement numéro 2, c'est à dire un B. Le sixième mouvement de H4 est donc un mouvement de type 2. Nous ne savons pas encore en préciser le sens (B ou b).

Détermination du sens du déplacement : "A" ou "a" ?

On va placer les chaînes **H**n et **G**n l'une en dessous de l'autre pour les comparer facilement, puis créer une nouvelle chaîne de caractères, notée **S**n constituée de 0 et de 1, qui présentera un 1 chaque fois qu'une majuscule de **G** remplace une minuscule de **H**, et un 0 sinon**.**

On définit ensuite la chaîne **U**n par :

- pour n pair : **U**n = **S**n 0

- pour n impair **U**n = **S**n 1

H_1 = A (de 1 vers 2)

G_1 = A

S_1 = 0

U_1 = 0 1

H_2 = A c B (1 vers 2; 1 vers 3; 2 vers 3)

G_2 = A C B

S_2 = 0 1 0

U_2 = 0 1 0 0

H_3 = A c B A C b A

G_3 = A C B A C B A

S_3 = 0 1 0 0 0 1 0

U_3 = 0 1 0 0 0 1 0 1

H_4 = A c B A C b A c B a C B A c B

G_4 = A C B A C B A C B A C B A C B

S_4 = 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0

U_4 = 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0

Comparons les différentes valeurs de Un.

Soit f la fonction définie par :

f(0)=01, f(1)=00 et f(ab)=f(a)f(b).

Appliquons cette fonction aux valeurs successives de Un.

$f(U_1) = f(01) = f(0)\ f(1) = 01\ 00 = U_2$

$f(U_2) = f(0100) = f(0)\ f(1)\ f(0)\ f(0) = 01\ 00\ 01\ 01 = U_3$

$f(U_3) = f(01000101) = f(0)\ f(1)\ f(0)\ f(0)\ f(0)\ f(1)\ f(0)\ f(1) = 01\ 00\ 01\ 01\ 01\ 00\ 01\ 00 = U_4$

On constate que $Un = f(Un\text{-}1) = f(f(Un\text{-}2)) = f(f(f(Un\text{-}3))) = f(f(...f(U1)...))$.

Il en résulte qu'un caractère sur deux de Un est toujours un 0. En effet, f(0) = 01 (ici le premier élément est 0, le deuxième étant 1) et f(1) = 00 (ici le premier élément est 0, le deuxième 0 aussi). Et comme Un est identique à Sn, à l'exception du dernier élément, on en déduit qu'un caractère sur 2 de Sn est égal à 0. Comme on compte le nombre de mouvements à partir de 0, alors les mouvements d'indices 0, 2, 4, 6, 8,... de Sn sont tous égaux à 0. Il en résulte que les mouvements correspondants dans Hn s'effectuent dans le sens indiqué par les majuscules.

Exemple 1

Quel est le cinquième mouvement de H4 ?

C'est le mouvement d'indice 4, car on compte à partir de 0. Comme 4 mod 3 = 1, le mouvement est de type 1 ("C" ou "c") puisque le mouvement numéro 1 de H4 est C. Comme dans $\boldsymbol{U_4}$ *le mouvement numéro 4 est un 0 (le mouvement est d'indice pair), alors le cinquième mouvement de H4 est C, et non pas c.*

Exemple 2

Quel est le treizième mouvement de H4 ?

C'est le mouvement d'indice 12, car on compte à partir de 0. Comme 12 mod 3 = 0, le mouvement est de type 0 ("A" ou "a") puisque le mouvement numéro 0 de H4 est A. Comme dans $\boldsymbol{U_4}$ *le mouvement numéro 12 est un 0 (le mouvement est d'indice pair), alors le treizième mouvement de H4 est A, et non pas a.*

Nous savons désormais trouver le sens d'un mouvement d'indice pair (en indexant à partir de zéro). Il nous reste à traiter le cas des mouvements d'indices impairs. Faisons d'abord une comparaison sur les **U**n successifs.

$\mathbf{U}_1$ = 0 1

$\mathbf{U}_2$ = 0 **1** 0 0

$\mathbf{U}_2$ = 0 **1** 0 0

$\mathbf{U}_3$ = 0 **1** 0 0 0 **1** 0 1

$\mathbf{U}_3$ = 0 **1** 0 0 0 **1** 0 1

$\mathbf{U}_4$ = 0 **1** 0 0 0 **1** 0 **1** 0 **1** 0 0 0 **1** 0 0

On a vu que $\mathbf{U}_2 = \mathbf{f}(\mathbf{U}_1)$. L'image du p-ième caractère de $\mathbf{U}_1$ (avec p=0 ou p=1) se trouve dans **U**2 à la position 2p et occupe les caractères d'indices 2p et 2p+1. On sait que $\mathbf{U}_3 = \mathbf{f}(\mathbf{U}_2)$. L'image du caractère d'indice p de $\mathbf{U}_2$ se trouve dans $\mathbf{U}_3$ aux positions 2p et 2p+1.

Exemple

*L'image du caractère d'indice 0 par f de $\mathbf{U}_2$ se trouve dans $\mathbf{U}_3$ aux positions 0 et 1 (2*0=0 et 2*0+1=1). L'image du caractère d'indice 1 par f de $\mathbf{U}_2$ se trouve dans $\mathbf{U}_3$ aux positions 2 et 3 (2*1=2 et 2*1+1=3).*

Ce phénomène est donc propagé par f au sein même des chaînes Un. Il est clair que les **U**n ont un nombre de caractères égal à une puissance de 2 (puisque f fait doubler le nombre de caractères à chaque application).

Exemple

*Considérons le caractère d'indice 0 de $\mathbf{U}_2$ (c'est 0). Par f, nous obtenons 01 qui se trouve aux positions 0 et 1 (2*0=0 et 2*0+1=1). L'image par f du caractère d'indice 1 de $\mathbf{U}_2$ est 00, qui se trouve aux positions 2 et 3 (2*1=2 et 2*1+1=3).*

Ainsi, dans la chaîne **U**n, le caractère qui se trouve à la position 2p est un zéro (comme on l'avait vu précédemment), et celui qui est à la position 2p+1 est un 0 ou un 1(cela dépend uniquement du caractère qui est à la position p) :

- si le caractère d'indice 2p+1 de Un est un 0, alors, à la position 2p on trouve la sous-chaîne 00, ce qui implique que son antécédent par f est un 1, donc à la position p on a un 1.

- si le caractère d'indice 2p+1 de Un est un 1, alors, à la position 2p on trouve la sous-chaîne 01, ce qui implique que son antécédent par f était un 0, donc à la position p on a un 0.

Notons $\mathbf{U}n_{[p]}$ le caractère d'indice p de **U**n. D'après ce qui précède, on a :

$\mathbf{U}n_{[2p]}=0$

$\mathbf{U}n_{[2p+1]} = 1$ si $\mathbf{U}n_{[p]} = 0$

$\mathbf{U}n_{[2p+1]} = 0$ si $\mathbf{U}n_{[p]} = 1$

Ce qui se résume en deux lignes :

$\mathbf{U}n_{[2p]} = 0$

$\mathbf{U}n_{[2p+1]} = 1 - \mathbf{U}n_{[p]}$

Nous pouvons affiner ce dernier résultat : si p est un multiple de 2 (p=2k), alors les indices impairs (2p+1 et 2p+3) s'écrivent 4k+1 et 4k+3. Ainsi, $\mathbf{U}n_{[4k+1]} = 1 - \mathbf{U}n_{[2k]} = 1\text{-}0 = 1$ et $\mathbf{U}n_{[4k+3]} = \mathbf{U}n_{[2p+3]} = \mathbf{U}n_{[2(p+1)+1]} = 1 - \mathbf{U}n_{[p+1]} = 1 - \mathbf{U}n_{[2k+1]} = 1 - (1 - \mathbf{U}n_{[k]}) = \mathbf{U}n_{[k]}$.

Finalement :

$\mathbf{U}n_{[2k]} = 0$: le mouvement d'indice 2k est codé par un 0 dans **S**n, et donc le sens du mouvement correspondant dans Hn est un sens "majuscule".

$\mathbf{U}n_{[4k+1]} = 1$: le mouvement d'indice 4k+1 est codé par un 1 dans **S**n, et donc le sens du mouvement correspondant dans Hn est un sens "minuscule".

$\mathbf{U}n_{[4k+3]} = \mathbf{U}n_{[k]}$: cette relation permet de calculer de proche en proche toutes les valeurs de $\mathbf{U}n_{[4k+3]}$, puisqu'on connaît déjà $\mathbf{U}n_{[k]}$ ($k<4k+3$).

Montrons par récurrence sur k que ces trois relations permettent bien de calculer de proche en proche tous les $\mathbf{U}n_{[k]}$:

$\mathbf{U}n_{[0]}$ est connu (c'est 0).

Supposons que $\mathbf{U}n_{[k]}$ soit connu pour tout entier inférieur ou égal à k.

k+1, comme tout entier naturel, est nécessairement de la forme 4q, ou 4q+1, ou 4q+2, ou 4q+3.

Si k+1=4q ou si k+1=4q+2, k+1 est pair et donc $\mathbf{U}n_{[k+1]}=0$, qui est donc connu.

Si k+1=4q+1, $\mathbf{U}n_{[k+1]}=1$, qui est donc connu.

Si k+1=4q+3, $\mathbf{U}n_{[k+1]}=\mathbf{U}n_{[q]}$. Comme k+1=4q+3, k=4q+2 et donc q<k. $\mathbf{U}n_{[q]}$ est donc déjà connu en vertu de l'hypothèse de récurrence. CQFD.

Exemple

Nous savons maintenant calculer le sens d'un mouvement d'indice impair.

Quel est le quatorzième (celui d'indice 13) mouvement de H4 ?

Comme *13 mod 3 = 1*, le quatorzième mouvement de G4 est le même que le mouvement numéro 1, c'est à dire un C. Le quatorzième mouvement de H4 est donc un mouvement de type 1 (C ou c). Quel est son sens ?

Prenons la chaîne U4 et déterminons son quatorzième caractère, correspondant à l'indice 13 (impair).

On a: 4*3+1=13, donc d'après les relations établies précédemment, $\mathbf{U}4_{[4*3+1]} = 1$. Ainsi, le mouvement numéro 13 de est codé par un 1 dans $\mathbf{U}4$, donc également dans $\mathbf{S}4$: le sens du mouvement correspondant dans H4 est un sens "minuscule". Finalement, le quatorzième mouvement de H4 est *c*.

U4 = 0 **1** 0 0 0 **1** 0 **1** 0 **1** 0 0 0 **1** 0

H4 = A **c** B A C **b** A **c** B **a** C B A **c**

Résumé

Pour déplacer n disques dans le problème des tours de Hanoï, il y a 2^n-1 mouvements à effectuer (une unité de moins que la longueur de la chaîne **U**n).

Pour déterminer le type du mouvement d'indice k (le premier mouvement ayant l'indice 0), on calcule t = k mod 3.

Si t = 0, le mouvement est de type 0 (A ou a).

Si t = 1, le mouvement est de type 1 (C ou c).

Si t = 2, le mouvement est de type 2 (B ou b).

Pour déterminer le sens du mouvement :

Si k=4p (k = 0 mod 4), le sens est "majuscule".

Si k=4p+1 (k = 1 mod 4), le sens est "minuscule".

Si k=4p+2 (k = 2 mod 4), le sens est "majuscule".

Si k=4p+3 (k = 3 mod 4), le sens est le sens du mouvement d'indice p, nécessairement déjà connu.

Conclusion

On constate que la dérécursification des tours de Hanoï est basée sur la recherche d'un algorithme différent de celui récursif. Cela peut arriver souvent, surtout dans les algorithmes qui contiennent plusieurs appels récursifs, ce qui les rend difficilement transformables en algorithmes itératifs. Néanmoins, on sait aujourd'hui que tout algorithme récursif peut être dérécursifié.

Chapitre VIII-4

Dérécursification du problème des "nombres en lettres"

Après avoir créé un programme récursif de traduction en français d'un nombre exprimé sous sa forme numérique (chapitre IV-13), nous allons maintenant voir une façon itérative de réaliser cela, mais en changeant un peu les structures de données de façon à montrer qu'il existe d'autres possibilités de faire la même chose (ceci dans un but éducatif, bien évidemment).

Spécificité de l'algorithme

Nous allons construire la chaîne en prenant chaque chiffre à partir de la droite et en répétant la boucle jusqu'à ce que l'on arrive au premier chiffre de gauche. Cette démarche a pour avantage de générer à chaque fois un résultat temporaire que l'on peut assez facilement utiliser à la boucle suivante pour traiter en particulier les cas spéciaux liés aux accords du pluriel ('s') , aux traits d'union et autres ('un' , 'et' …) . On n'aura plus qu'à faire la concaténation progressive des résultats obtenus. Par contre, cette évaluation s'effectue dans le sens inverse de la lecture naturelle des nombres (c'est à dire de la droite vers la gauche), c'est pourquoi à chaque groupe de trois chiffres on rajoutera l'unité correspondante : mille, million, milliard…

Algorithme

Pour commencer, on va créer des tableaux avec des chaînes de caractères.

On crée un tableau des unités avec les nombres qui ne peuvent pas se décomposer : zéro, un, deux, …, seize : on voit qu'il regroupe les unités et les nombres de dix à seize. Les nombres dix-sept, dix-huit, dix-neuf seront quant à eux logiquement traités en deux étapes (« dix » puis le chiffre restant).

On crée ensuite un tableau des dizaines qui contient les nombres vingt, trente, quarante, cinquante, soixante, quatre-vingt. ("soixante-dix" et "quatre-vingt-dix" n'y figurent pas, étant des cas spéciaux).

On crée ensuite un tableau qui contient les unités restantes : cent, mille, million, milliard, etc.

On suppose que le nombre est entier.

Si le nombre extrait est 0 alors on renvoie « zéro ».

```
275<>0 , donc on continue
```

Début de la boucle :

Considérons les deux exemples suivants : les nombres 275 et 398.

On va les découper en tranches de 3 chiffres, à partir de la fin et répéter ce qui suit jusqu'à ce qu'on soit tout à gauche du nombre initial (donc à son début) : à chaque étape de la boucle on divise le nombre par 1000, donc on va arriver à 0 tôt ou tard, auquel cas on s'arrête.

NB : par souci de clarté, on n'entrera pas ici dans le détail des accords du pluriel, des traits d'union etc…

1ère étape : On récupère d'abord le chiffre des unités, suivi immédiatement de celui des dizaines afin de traiter les cas spéciaux de la langue française.

```
nombre = 275,                          nombre=398
unités=5   , nombre restant=27         unités=8, nombre restant=39
dizaines=7, nombre restant= 2          dizaines=9, nombre restant= 3
```

Soit ***temp*** une variable auxiliaire de type chaîne, qui est vide.

```
Temp := '' ;                           temp := '' ;
```

Si le chiffre des dizaines est « 1,7,9 » alors on décrémente ce chiffre et on incrémente de 10 le chiffre des unités.

```
Dizaines=6, unités=15                  dizaines=8, unités=18
```

Si le chiffre des dizaines est supérieur à 1, alors on initialise ***temp*** avec la valeur correspondante (lue dans le tableau des dizaines)

```
Temp :='soixante' ;                    temp :='quatre vingt' ;
```

Si on a un nombre des dizaines plus petit que 8 et si on a un « un » ou « onze » alors rajouter le mot « et »

```
On n'est pas concerné par ce cas.      On n'est pas concerné par ce cas.
```

Si le nombre des unités est supérieur à seize, alors on rajoute un espace et « dix » à ***temp*** et on décrémente le nombre des unités de 10. On se retrouve donc avec un nombre des dizaines décrémenté de 1, et le nombre des unités compris entre 1 et 16.

```
On n'est pas concerné par ce cas.      Temp :='quatre vingt dix' ;
                                       unités := unités-10 = 18-10 = 8 ;
```

Si le nombre des unités est plus grand que zéro alors on rajoute à ***temp*** la chaîne correspondante aux unités.

```
Temp :='soixante' + 'quinze' =         temp :='quatre vingt dix' + 'huit'  =
'soixante quinze' ;                    'quatre vingt dix huit' ;
```

On mémorise le résultat dans une autre variable.

```
Result := 'soixante quinze' ;          Result :='quatre vingt dix huit' ;
```

2ème étape : on étudie maintenant le 3è chiffre à partir de la droite …

On récupère le chiffre des centaines et on divise le nombre restant par 10.

```
C :=2 ; nombre restant := 0 ;          C :=3 ; nombre restant := 0 ;
```

Si le chiffre des centaines est supérieur à 1, alors on va récupérer l'unité correspondante :

```
Temp :='deux' ;                        Temp :='trois' ;
```

On rajoute 'cent' à cette variable :

```
Temp :='deux cent' ;                   Temp :='trois cent' ;
```

Si le résultat mémorisé est nul, alors on rajoute un 's' sinon on ne fait rien

```
On n'est pas concerné par ce cas.      On n'est pas concerné par ce cas.
```

On colle ces deux résultats :

```
Result :='deux cent'+                  Result :='trois cent' +
'soixante quinze' =                    'quatre vingt dix huit' =
'deux cent soixante quinze' ;          'trois cent quatre vingt dix huit' ;
```

On regarde si on doit rajouter mille, million ou milliard. Ceci se fait à l'aide d'un compteur auxiliaire qui est incrémenté à chaque pas de la boucle.

```
On n'est pas concerné par ce cas.      On n'est pas concerné par ce cas.
```

Fin de la boucle

Chapitre IX – Les jeux de réflexion

Chapitre IX-1

Le morpion à trois pions

Pour expliquer la programmation des jeux de réflexion, nous allons d'abord décrire la méthode, puis nous allons l'appliquer.

Nous allons commencer par des jeux très simples, ensuite nous attaquerons des jeux plus difficiles et plus complexes.

Le premier exemple étudié ici est le jeu du morpion qui se joue ici sur un échiquier de 3 sur 3. Le jeu se joue à deux. Chaque joueur pose une pièce sur l'échiquier puis laisse son adversaire poser sa pièce (d'habitude, les pions sont 'X' et 'O'); celui qui réussi à aligner 3 de ses pions en ligne, colonne ou diagonale, a gagné.

Lorsque deux hommes s'affrontent dans un jeu de réflexion, chacun essaie de prévoir le coup de l'adversaire, ou même plusieurs coups à l'avance. Nous allons appliquer la même méthode ici.

La méthode générale pour faire jouer un ordinateur comme un humain est la suivante :

- on décide jusqu'à quelle profondeur on prévoit les coups de l'adversaire

- on simule un coup de l'ordinateur par les étapes suivantes :

    - on mémorise l'échiquier dans un tableau auxiliaire

    - on parcourt l'échiquier jusqu'à trouver une case disponible où on pourrait poser un pion

    - on pose le pion

- on simule un coup du joueur en faisant les mêmes étapes, ensuite on simule un coup de l'ordinateur et ainsi de suite jusqu'à ce qu'on ait atteint le niveau maximum de profondeur fixé au départ

- on évalue la position par une fonction d'évaluation (c'est ce qu'il y a de plus difficile à faire dans un jeu de réflexion)

- en fonction de la note renvoyée par la fonction d'évaluation on décide quel coup on va jouer

Nous allons étudier quelques cas simples pouvant apparaître pendant une partie du jeu de morpion, pour mieux comprendre comment appliquer la stratégie décrite ci-dessus.

On sait que dans un jeu de morpion à 3 pions on a trois possibilités : soit on gagne, soit on fait match nul, soit on perd.

Donc la fonction d'évaluation est très facile à faire pour ce jeu : si on a trois pions alignés on a gagné, si aucun des deux joueurs n'a trois pions alignés alors c'est un match nul, si l'adversaire a trois pions alignés alors on a perdu.

Ici on décide qu'on va descendre jusqu'au niveau 9, c'est à dire qu'on va simuler des coups jusqu'à ce que l'échiquier soit rempli, car cela ne demande pas trop de temps, vu sa taille.

C'est pour cette même raison que l'ordinateur va parcourir tous les coups possibles de l'échiquier et ne choisira que ceux qui le donnent gagnant à coup sûr et si cela est impossible, alors il choisira un coup qui lui rapporte le match nul.

Soit le cas suivant :

```
  A B C                 A B C                 A B C
 +-----+               +-----+               +-----+
1¦X¦ ¦ ¦  L'ordinateur 1¦X¦ ¦O¦  Le joueur    1¦X¦ ¦O¦ et on voit
 +-+-+-¦  joue en C1;   +-+-+-¦  joue en A3;   +-+-+-¦ facilement
2¦ ¦O¦ ¦               2¦ ¦O¦ ¦               2¦ ¦O¦ ¦ qu'il gagne.
 +-+-+-¦               +-+-+-¦               +-+-+-¦
3¦ ¦ ¦X¦               3¦ ¦ ¦X¦               3¦X¦ ¦X¦
 +-----+               +-----+               +-----+
```

Ces deux coups ont été faits par simulations successives en mémoire; on n'a que deux coups car le premier échiquier représente la situation en cours.

Avant de jouer en C1, l'ordinateur a déjà exploré l'arbre qui résulte de B1;on est passé au coup C1 juste pour l'exemple, de même que pour A3. A partir du dernier échiquier, l'ordinateur va encore explorer quelques coups en profondeur, car nous, nous voyons que les X ont gagné, mais lui, il ne sait pas voir ! C'est pourquoi il faut encore simuler des coups jusqu'à ce que les X aient trois pions alignés.

Néanmoins, on sait que le coup A3 est excellent pour le joueur, et que l'ordinateur ne peut pas l'empêcher de gagner; ceci indique que le coup précédent simulé par l'ordinateur (à savoir C1) est mauvais. Il faut donc qu'il essaie un autre coup parmi les restants pour gagner, ou faire match nul dans le pire des cas.

Nous allons commencer par analyser le programme du morpion. On distingue 2 parties :

- une partie graphique (représentation de l'échiquier, possibilité de cliquer avec la souris afin de poser son pion sur l'échiquier...)

- une partie de simulation, où a lieu la mise en oeuvre de la réflexion, la simulation des coups par ordinateur, l'évaluation...

La première partie ne sera pas décrite ici, l'essentiel étant la réflexion.

Voyons la deuxième partie :

- la fonction d'évaluation d'abord, qui est très facile à faire dans ce cas : on représente en mémoire les X par 1 (JOUEUR dans le programme) et les O par -1 (ORDINATEUR dans le programme) ensuite on fait une fonction qui calcule la somme des lignes, des colonnes et des diagonales et si elle trouve à un moment une de ces sommes égale à 3 ou -3 alors c'est que soit le joueur, soit l'ordinateur, a aligné 3 pions. Cette fonction s'appelle note1 dans le programme.

- on a dans ce programme une deuxième fonction d'évaluation, qui fait appel à la première; cette fonction s'appelle note2. Elle est utilisée justement dans des cas spéciaux comme le suivant :

```
  A B C
 +-----+
1¦X¦ ¦X¦    C'est aux X de jouer; ils ont deux possibilités de
 +-+-+-¦    gagner, donc ils sont gagnants quoiqu'il arrive; il est
2¦O¦O¦ ¦    donc inutile de descendre dans l'arbre de simulation des
 +-+-+-¦    coups; la réflexion s'arrête ici, de cette manière on gagne
3¦O¦ ¦X¦    du temps, ce qui est très important dans un jeu de réflexion.
 +-----+
```

- on a dans ce programme une fonction qui s'appelle coup obligatoire; elle est utilisée dans certains cas comme :

```
  A B C
 +-----+
1¦X¦O¦X¦    C'est aux X de jouer; ils ont cinq coups à disposition
 +-+-+-¦    A2, C2, A3, B3, C3.
2¦ ¦O¦ ¦    Mais ici, il est inutile d'explorer tous les coups, car on
 +-+-+-¦    voit qu'il n'y a qu'une case que les X doivent occuper obli-
3¦ ¦ ¦ ¦    gatoirement : B3, pour empêcher les O de gagner. Donc, grâce
 +-----+    à cette fonction on gagne aussi du temps pendant la réflexion.
```

- nous avons enfin la procédure qui représente le coeur du programme, elle s'appelle "jeu"; cette procédure a plusieurs paramètres - nous verrons plus loin pourquoi et quelle est la signification de chacun de ceux-ci -.

Dans cette procédure nous allons simuler des coups, faire des évaluations et comparer la valeur des coups. Commençons par la simulation de coups : sachant que "t" est l'échiquier (tableau à deux dimensions) on simule un coup par l'instruction : "t[ligne,colonne]:=JOUEUR" ou par l'instruction : "t[ligne,colonne]:=ORDINATEUR"; ceci implique qu'il faudrait avoir deux procédures, une qui simule les coups du joueur et l'autre qui simule les coups de l'ordinateur; c'est pour cette raison qu'on a décidé d'initialiser les constantes : "JOUEUR = +1;ORDINATEUR = -1 ".Ainsi, on n'a plus qu'une seule procédure qui admet un paramètre "coef" initialisé une fois à "1" et une fois à "-1"; de cette façon l'instruction décrite ci-dessus "t[ligne,colonne]:=ORDINATEUR" devient l'instruction suivante : "t[ligne,colonne]:=JOUEUR*coef" avec coef=-1; car "JOUEUR*coef=-JOUEUR" et "-JOUEUR=ORDINATEUR".Il fallait donc choisir les valeurs de JOUEUR et ORDINATEUR symétriques par rapport à zéro. Pour annuler un coup, on fait simplement "t[ligne,colonne]:=PERSONNE", sachant que "PERSONNE=0" et qu'au début de la partie, tout l'échiquier est vide, rempli de 0 (donc "PERSONNE").

Etudions un peu la façon de noter et de renvoyer les notes. Supposons qu'on soit au niveau 6, par exemple. Supposons que c'est à l'ordinateur de jouer et qu'il joue un coup qui lui permet de gagner. Alors, il renvoie au

niveau 5 une note égale à MAUVAIS pour indiquer au joueur que son coup était mauvais et qu'il doit en essayer un autre; donc il ne faut pas remonter au niveau 4 pour indiquer à l'ordinateur que le joueur perd et que le coup joué par l'ordinateur au niveau 4 est bon; pour cela, on utilise une variable dans le nom de la procédure "jeu"; cette variable s'appelle "remonte". Etant donné qu'on a une seule procédure pour renvoyer les notes "BON" et "MAUVAIS", on utilise le même système que ci-dessus : on initialise BON à une valeur positive et MAUVAIS à l'opposé de BON, de cette façon on a BON*coef=MAUVAIS avec coef=-1. n aurait pu choisir une autre valeur que 1 (ce choix est complètement arbitraire).

Le dernier paramètre de la procédure est "niveau" qui indique à quel niveau de profondeur on est dans la réflexion, qui est incrémenté à chaque appel récursifs de la procédure.

Voyons un peu maintenant les grandes lignes de cette procédure; nous allons faire cela à partir de situations réelles :

```
  A B C
 +-----+
1¦X¦O¦ ¦    C'est aux X de jouer; ils ont cinq coups à disposition
 +-+-+-¦    C1, C2, A3, B3, C3.
2¦X¦O¦ ¦    Mais ici, il est inutile d'explorer tous les coups, car on
 +-+-+-¦    voit que les X doivent jouer en A3 pour gagner; la première
3¦ ¦ ¦ ¦    chose qu'on fait est donc de voir si les X n'ont pas un coup
 +-----+    obligatoire à jouer pour prendre l'avantage.
```

Si tel n'était pas le cas, alors il faut envisager un coup obligatoire pour bloquer l'adversaire.

Supposons qu'un de ces deux cas apparaît; on doit donc jouer un coup qui est obligatoire (soit pour gagner, soit pour bloquer) :

- s'il y en a un, alors on simule le coup en posant un X et :

  - on vérifie si le JOUEUR a gagné (en faisant un appel à la fonction "note1";si c'est le cas, on annule le coup du joueur en posant un zéro sur l'échiquier et on renvoie au niveau précédent la note "MAUVAIS" pour indiquer à l'ordinateur que son coup était mauvais, étant donné qu'il a permis au joueur de gagner

  - on vérifie si l'ORDINATEUR n'a pas 2 possibilités de gagner, auquel cas on sort de la procédure, car de toutes façons, on ne peut pas le bloquer, et dans ce cas, on renvoie la note "BON" pour indiquer à l'ordinateur que son coup était bon, étant donné qu'il lui a permis de gagner

  - si aucun de ces 2 cas ne se présente, alors on appelle récursivement la procédure "jeu", pour continuer la simulation..

- s'il n'y en a aucun, alors on commence à explorer les cases une par une et simuler les coups :

  - on pose un "X" ou "O" sur l'échiquier

  - on évalue la situation de l'échiquier par un appel à "note1"

- la note a la valeur "BON" : dans ce cas on annule le coup et on renvoie au niveau précédent la valeur "MAUVAIS"

- la note a la valeur "MOYEN" : on fait un appel récursif pour savoir qui va gagner dans les coups suivants et suivant la valeur de la note reçue on sort de la procédure en cours ou on reste (si on reste alors c'est que le coup a engendré soit un match nul, soit une victoire pour l'adversaire. Il faut donc explorer les autres coups, pour essayer de gagner ou de faire match nul

- la note a la valeur "MAUVAIS" : on annule le coup et on continue d'explorer les coups restants.

Si on a exploré tous les coups et qu'ils engendrent tous une défaite, alors il faut renvoyer une note défavorable (ou favorable) au niveau du dessus.

```
const BON        = +1;
      MOYEN      =  0;
      MAUVAIs    = -1;
      ORDINATEUR = -1;
      JOUEUR     = +1;
      PERSONNE   =  0;

var t             : array[0..2,0..2] of integer;
    meilleur_coup : integer;
    le_niveau     : integer;

function note1(j : integer) : integer;
var f : array[0..7] of integer;
    i : integer;
begin
f[0]:=t[0,0]+t[0,1]+t[0,2];f[1]:=t[1,0]+t[1,1]+t[1,2];
f[2]:=t[2,0]+t[2,1]+t[2,2];f[3]:=t[0,0]+t[1,0]+t[2,0];
f[4]:=t[0,1]+t[1,1]+t[2,1];f[5]:=t[0,2]+t[1,2]+t[2,2];
f[6]:=t[0,0]+t[1,1]+t[2,2];f[7]:=t[0,2]+t[1,1]+t[2,0];
note1:=BON   ;j:=3*j;
for i:=0 to 7 do
    if f[i]=j then exit;  //--- on quitte avec la valeur "bon"
note1:=MOYEN ;
end;

function note2(j,nb : integer) : integer;
var x,y,note,nb_coups : integer;
begin
note2:=BON;
nb_coups:=0;
for x:=0 to 2 do
for y:=0 to 2 do
    if t[x,y]=PERSONNE then
       begin
       t[x,y]:=j;
       note:=note1(j);
       if note=BON then inc(nb_coups);
       t[x,y]:=PERSONNE;
       if nb_coups=nb then exit;  //--- on quitte avec la valeur "bon"
       end;
note2:=MOYEN;
end;

function coup_obligatoire(j : integer) : integer;
var x,y,note : integer;
begin
coup_obligatoire:=-1;
for y:=0 to 2 do
for x:=0 to 2 do
    if t[x,y]=PERSONNE then
       begin
       t[x,y]:=j;note:=note1(j);t[x,y]:=PERSONNE;
       if note=BON then
          begin
          coup_obligatoire:=y*10+x;
          exit;
```

```
          end;
       end;
end;

procedure jeu(coef,niveau : integer;var note,remonte : integer);
var x,y,z,nb : integer;
    la_note  : integer;
    monte    : integer;
    nb_poss,nb_notes : integer;
begin
note:=MOYEN;
remonte:=0;
if niveau=9 then
   note:=note1(JOUEUR*coef)
else
   begin
   z:=coup_obligatoire(JOUEUR*coef);nb:=1;
   if z=-1 then
      begin
      z:=coup_obligatoire(ORDINATEUR*coef);
      inc(nb);
      end;
   if z>=0 then
      begin
      y:=z div 10;
      x:=z mod 10;
      t[x,y]:=JOUEUR*coef;
      if note1(JOUEUR*coef)=BON then
         begin t[x,y]:=PERSONNE;note:=MAUVAIS*coef;exit;end;
      if note2(ORDINATEUR*coef,nb)=BON then
         begin t[x,y]:=PERSONNE;note:=BON*coef;exit;end;
      jeu(-coef,niveau+1,note,monte);
      t[x,y]:=PERSONNE;remonte:=0;
      end
   else
      begin
      nb_poss:=0;nb_notes:=0;
      for y:=0 to 2 do
      for x:=0 to 2 do
          if t[x,y]=PERSONNE then
             begin
             t[x,y]:=JOUEUR*coef;inc(nb_poss);
             case note1(JOUEUR*coef) of
               BON   : begin
                       t[x,y]:=PERSONNE;note:=MAUVAIS*coef;
                       exit
                       end;
               MOYEN : begin
                       jeu(-coef,niveau+1,la_note,monte);
                       case coef of
                         +1 : begin
                              if la_note>MOYEN then inc(nb_notes);
                              if (la_note<MOYEN) and (monte<1) then
                                 begin
                                 t[x,y]:=PERSONNE;note:=la_note;
                                 inc(remonte);
                                 exit;
                                 end;
                              end;
                         -1 : begin
                              if la_note<MOYEN then inc(nb_notes);
                              if (la_note>MOYEN) and (monte<1) then
                                 begin
                                 t[x,y]:=PERSONNE;note:=-la_note;
                                 inc(remonte);
                                 exit;
                                 end;
                             end;
                          end;
                       end;
               end;
             t[x,y]:=PERSONNE;
             end;
      if (nb_notes>0) and (nb_notes=nb_poss) then
         note:=coef*BON;
      end;
   end;
end;
```

```
procedure coup_ordinateur;
var x,y,coup : integer;
    la_note  : integer;
    monte   : integer;
begin
coup:=-1;inc(le_niveau);
meilleur_coup:=-1;
for y:=0 to 2 do
for x:=0 to 2 do
    if t[x,y]=PERSONNE then
       begin
       t[x,y]:=ORDINATEUR;
       case note1(ORDINATEUR) of
         BON   : begin
                 t[x,y]:=PERSONNE;
                 meilleur_coup:=y*10+x;
                 exit;
                 end;
         MOYEN : begin
                 jeu(1,le_niveau,la_note,monte);
                 t[x,y]:=PERSONNE;
                 case la_note of
                    MAUVAIS : if meilleur_coup=-1 then meilleur_coup:=y*10+x;
                    MOYEN   : meilleur_coup:=y*10+x;
                    BON     : begin
                              meilleur_coup:=y*10+x;
                              exit;
                              end;
                    end;
                 end;
         end;
       t[x,y]:=PERSONNE;
       end;
end;


procedure tform1.nouveau_jeu;
var x,y : integer;
begin
le_niveau:=0;
with image1.picture.bitmap.canvas do
   begin
   brush.style:=bsSolid;brush.color:=clWhite;
   rectangle(0,0,90,90);
   rectangle(30,0,60,90);
   brush.style:=bsClear;  //---- utilisé pour l'écriture des "x" et "o"
   rectangle(0,30,90,60);
   for x:=0 to 2 do
   for y:=0 to 2 do
       t[x,y]:=0;
   end;
end;

procedure tform1.affiche_jeu;
var x,y : integer;
begin
for x:=0 to 2 do
for y:=0 to 2 do
    if t[x,y]=JOUEUR then image1.picture.bitmap.canvas.textout(x*30+8,y*30+8,'X')
    else
    if t[x,y]=ORDINATEUR then
image1.picture.bitmap.canvas.textout(x*30+8,y*30+8,'O');
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
x:=x div 30;if x>2 then x:=2;
y:=y div 30;if y>2 then y:=2;
edit1.text:='X='+inttostr(1+x)+'; Y='+inttostr(1+y);
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
if le_niveau>8 then
   exit;
```

```
button3.hide;
x:=x div 30;if x>2 then x:=2;
y:=y div 30;if y>2 then y:=2;
if t[x,y]<>0 then
   begin
   showMessage('Case déjà occupée !');
   exit;
   end;
t[x,y]:=JOUEUR;
affiche_jeu;
inc(le_niveau);
if le_niveau<9 then coup_ordinateur;
y:=meilleur_coup div 10;
x:=meilleur_coup mod 10;
t[x,y]:=ORDINATEUR;
affiche_jeu;
if note1(ORDINATEUR)=BON then
   begin
   showMessage('J''ai gagné !');
   le_niveau:=9;
   exit;
   end;
if note1(JOUEUR)=BON then
   begin
   showMessage('Vous avez gagné !');
   le_niveau:=9;
   exit;
   end;
if le_niveau=9 then showMessage('Match nul !');
end;
```

Chapitre IX-2

Le jeu d'Othello

Un jeu assez répandu est le jeu d'Othello. Nous allons nous intéresser ici à la manière de programmer ce jeu. Nous allons décrire et expliquer par des exemples la manière classique de programmer le jeu d'Othello, et nous allons ensuite l'appliquer pour réaliser le programme. Pour faciliter l'explication nous commencerons par un exemple de fin de partie. L'avantage des fins de jeu est le suivant : la fonction d'évaluation d'un coup est très facile à réaliser, car elle consiste à faire le décompte des pions. Celui qui a le plus de pions après le dernier coup a gagné. Par contre, en début ou milieu de partie cette fonction est beaucoup plus difficile à réaliser, car il faut tenir compte du nombre de pions, de la valeur des cases (certaines cases ont une importance plus grande que d'autres, comme les coins), de la position des pions à un instant t... Il faudrait donc avoir l'avis d'un spécialiste de l'Othello.

Description des données et des procédures

Le plateau du jeu d'Othello sera un tableau à une dimension; ce tableau aura 100 cases et sera numéroté de 0 à 99. Ainsi, les cases du plateau seront 11, 12,...,18, 21,...,28,... jusqu'à 81,...,88. Le tableau de jeu s'appelle *work*, mais, pendant la réflexion, comme on va à plusieurs niveaux de profondeur, on est obligé de mémoriser plusieurs plateaux. Pour cela, on utilise le tableau *tab_pense*, qui peut contenir 19 niveaux de jeu (largement plus qu'il n'en faut; de plus, on peut changer quand on veut le nombre de plateaux qu'il contient).

On note les noirs par 0 et les blancs par 2. Les cases vides sont notées généralement avec un 8. Les cases vides qui sont autour d'un pion sont notées avec un 9. Ceci a été fait pour gagner du temps pendant le jeu. En effet, si on est en début de partie, on n'a que quatre pions. Comme la structure du tableau est linéaire, alors on avance linéairement dans le tableau jusqu'à ce qu'on trouve un 9. On sait alors qu'une case remplie avec un pion se trouve à proximité. Si on n'avait pas eu ce 9, alors, pour chacune des cases de l'échiquier on aurait été obligés de tester si on n'est pas en contact avec un pion.

Pour se déplacer dans les 8 directions, on utilise un tableau à 8 éléments appelé *direction*.

Pour essayer le programme, on a pris une situation de fin de partie un peu plus complexe que celle décrite dans les pages suivantes, afin de voir les performances du programme (coups, temps); le plateau représentant cette situation s'appelle *work0*. Sur cet othellier, on représente les pions du joueur par 2 et ceux de l'ordinateur par 0.On voit qu'il nous reste 11 coups à jouer. C'est pour cette raison qu'on a une variable *profondeur*, qui indique jusqu'à quel niveau l'ordinateur doit descendre pendant la réflexion.

Nous allons décrire maintenant les fonctions et les procédures afin de montrer à quoi servent les autres données.

Trouve_pos

Pour pouvoir jouer un coup, il faut d'abord trouver une case où placer son pion. La fonction qui trouve cette case s'appelle *trouve_pos* et elle a deux paramètres : *niveau* et *joueur*. Le paramètre *niveau* indique dans quel niveau (pendant la réflexion) il doit trouver un coup. Si, à ce niveau, on avait déjà joué un coup, il faut chercher un autre coup à partir de la position de ce pion déjà joué. On mémorise donc la dernière place trouvée dans un tableau, appelé dans le programme *position*. L'algorithme de la procédure est le suivant : on cherche un 9 dans l'othellier, puis on regarde les huit directions et si, dans une de ces directions, le pion joint au 9 est un pion ennemi et quelques cases plus loin on a un pion ami, alors c'est qu'on peut jouer à la place occupée par le 9.

Une fois qu'on a trouvé une position, il faut pouvoir placer le pion dans la case et retourner les pions dans toutes les directions possibles; la procédure qui fait cela s'appelle ***joue_coup***.

Etant donné que nous sommes en fin de partie, la meilleure manière d'évaluer un coup est de compter combien de pions il rapporte. Pour cela, on a écrit la fonction ***compte_les_pions***, qui compte les pions pour une couleur indiquée, blanc ou noir (joueur ou ordinateur). Le paramètre est ici inutile car l'ordinateur a les noirs. Il suffirait donc de comparer les pions directement à 0; mais ceci a été laissé au cas ou le lecteur voudrait modifier le programme.

Nous allons voir maintenant la principale procédure du programme, ***arbre***. C'est la procédure qui s'applique à tous les jeux de réflexion, aussi nous allons l'expliquer avec des exemples.

Supposons qu'on va à 3 niveaux de profondeur. A chaque niveau, on joue les coups possibles un par un et, pour chaque coup, on explore l'arbre qui en découle. A chaque niveau final, on évalue la situation de l'échiquier et on retourne la note. Si le niveau 2 était un niveau du joueur, alors la note attribuée à ce niveau sera la plus petite note trouvée, car le but du joueur est que l'ordinateur obtienne la plus petite note possible. En revanche, au niveau 1, celui de l'ordinateur, l'ordinateur choisira le coup qui lui rapporte la plus grande note. Cette façon d'explorer un arbre et de retourner les notes, une fois les plus petites et une fois les plus grandes, s'appelle arbre min-max. Comme ici on explore un arbre en fin de partie, la fonction d'évaluation est faite par le compte des pions de l'ordinateur.

A chaque branche de l'arbre exploré on doit faire plusieurs étapes :

```
- faire les initialisations du niveau correspondant
    - note attribuée au niveau
    - le tableau "position" décrit ci-dessus
    - le nombre de coups
```

Le nombre de coups sera utilisé au cas où aucun coup n'est possible au niveau actuel; dans ce cas, on doit "sauter" le niveau en cours et passer au niveau suivant en ayant pris soin d'incrémenter la profondeur; si, au niveau suivant, il n'y a pas de coup possible, alors c'est qu'on est arrivé à la fin de la partie et on évalue la situation.

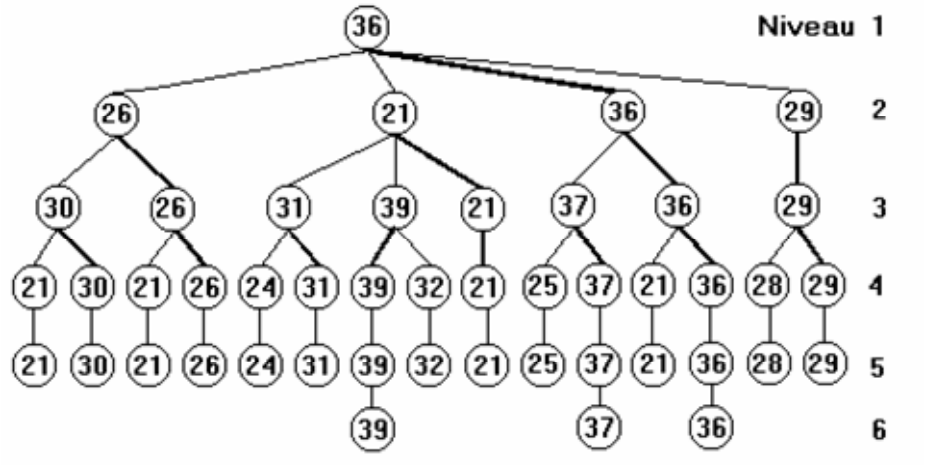
```
repeter
   - chercher un coup
       - si trouvé, alors :
            - jouer ce coup
            - mémoriser le plateau de jeu
            - descendre dans l'arbre si on n'est pas au niveau final, sinon
```

```
                évaluer l'othellier
        - si non trouvé, alors regarder si au niveau précédent on a pu jouer un
             coup, auquel cas on continue à descendre dans l'arbre, sinon les
             deux sont bloqués, donc niveau terminal, donc évaluation.
      - faire l'élagage pour gagner du temps; l'élagage est expliqué par un arbre
             dans les pages suivantes.
jusqu'à ce qu'il n'y ait plus de coups possibles.
```

La procédure ***arbre*** a deux paramètres, *level* (niveau, pour savoir à quel niveau de profondeur on est) et *c* (coefficient pour savoir qui doit jouer : soit le joueur, soit l'ordinateur; on appelle ***joue_coup*** avec le paramètre *1+c*, qui vaut 0 ou 2; on n'a pas choisi des valeurs symétriques par rapport à 0 - comme dans le cas du le morpion -, car ici, on a un tableau de 100 bytes et non pas integer; or il faut gagner du temps pendant la réflexion, car il y a beaucoup de sauvegardes de plateaux).

Une fois le programme terminé, il ne nous restera plus qu'à écrire une fonction d'évaluation en cours de partie et de modifier la procédure ***arbre*** de façon à ce qu'elle n'appelle plus ***compte_les_pions***, mais une autre fonction d'évaluation (malheureusement c'est ce qu'il y a de plus difficile à faire).

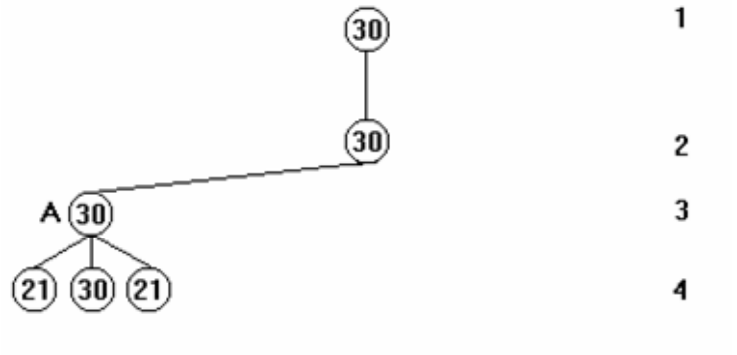
Nous allons étudier maintenant sur un exemple de fin de partie la programmation des jeux de réflexion. Nous allons voir la procédure ***alpha-beta*** (ou *min-max*) et aussi comment on doit faire un élagage (couper les branches d'un arbre), afin de gagner du temps pendant l'exploration d'un arbre (quelque soit le jeu de réflexion qu'on aura programmé). Nous allons illustrer ceci par un exemple concret (voir les images annexes).



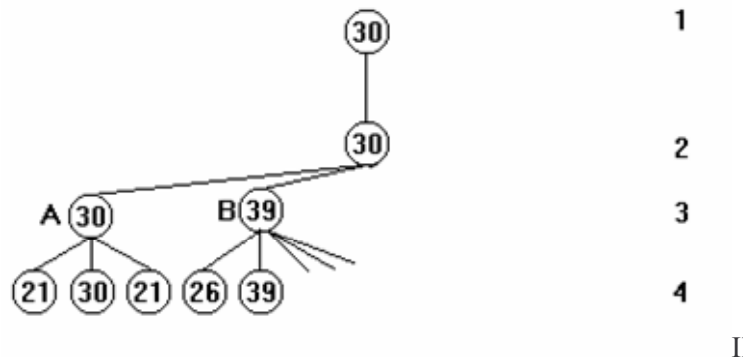
La programmation de la réflexion d'un ordinateur se fait de la manière suivante : l'ordinateur simule un coup pour le joueur, ensuite un coup pour lui, ensuite pour le joueur et ainsi de suite jusqu'à ce qu'il atteigne la profondeur de jeu souhaitée. Une fois cette profondeur atteinte, il évalue le plateau et retourne la note trouvée. Pour les deux images qui illustrent notre étude (ou le dessin ci-dessus), la fonction d'évaluation est faite par le décompte des pions. On obtient successivement les notes 21, 30, 21, 26, 24, 31, 39, 32, 21, 25, 37, 21, 36, 28 et 29. Toutes ces notes seront retournées aux niveaux supérieurs. Mais que va-t-il se passer si un noeud a plusieurs fils ? Quelle note va-t-il choisir ? Pour chaque noeud de l'arbre, si le noeud se trouve à un niveau correspondant au coup du joueur (les blancs jouent) alors c'est la note la plus grande qui est retournée parmi les fils de ce noeud. C'est pourquoi, au niveau 3, on se retrouve avec les notes suivantes: 30, 26, 31, 39, 21, 37, 36, 29. Par contre, pour

les niveaux correspondants aux coups de l'ordinateur, c'est la note la plus petite qui est retournée, ce qui explique qu'au niveau 2 on se retrouve avec les notes 26, 21, 36, 29. Mais au niveau 1, on prend la note la plus grande, d'où le 36 en haut de l'arbre, qui nous provient du troisième coup. L'ordinateur doit donc jouer le troisième coup. Cet algorithme s'appelle *alpha-beta* ou *min-max*, car, une fois sur deux, c'est la note maximale qui est retournée, et, une fois sur deux, la note minimale. Il est normal que parmi tous les coups qui s'offrent à lui, l'ordinateur cherche à minimiser la note de l'humain et à maximiser la sienne. On rappelle que c'est ce simple algorithme qui a battu Kasparov aux échecs (avec une puissance de calcul énorme, de l'ordre de 300 millions de coups à la seconde).

Mais, est-il bien utile d'explorer toutes les branches d'un arbre pendant le jeu ? N'y aurait-il pas un moyen de supprimer certaines branches de l'arbre, pour gagner du temps ? C'est ce que nous allons voir maintenant, avec l'opération d'élagage. Voici un dessin représentant un arbre, dont seulement le premier fils du noeud de niveau 2 a été visité en entier. On suppose que le noeud de niveau 2 a seulement 4 fils :

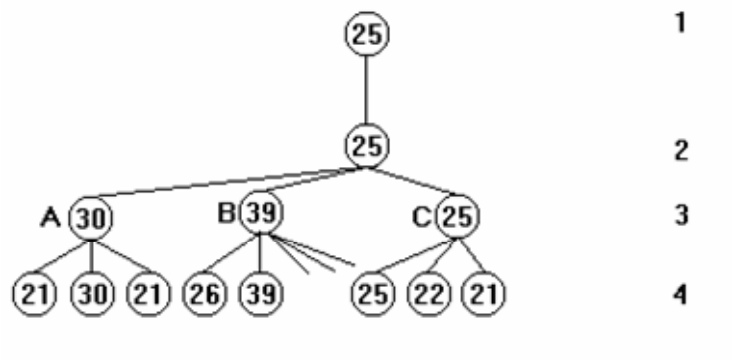


On explore d'abord le noeud A du niveau 3. On décide qu'à ce niveau l'ordinateur cherche à maximiser la note (il aurait très bien pu la minimiser). Le noeud A a trois fils, et le maximum est 30. Cette note est retournée au niveau précédent et comme c'est la première note à être retournée, alors elle devient la note témoin du niveau 2. On rappelle qu'au niveau 2, on cherche à minimiser les notes des fils. L'exploration du noeud A étant finie, l'ordinateur remonte au niveau 2, afin de parcourir les autres fils du niveau 2. Mais la note est retournée au niveau précédent (ceci est toujours valable : on retourne toujours la note au père du noeud en cours).

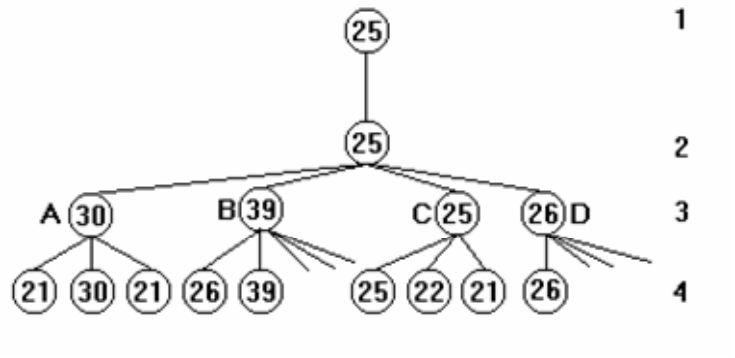


Il passe ensuite au noeud B. Le premier fils de B rapporte 26. Le deuxième fils rapporte 39. Cette note est supérieure à 30 (témoin du niveau précédent). Alors l'exploration du noeud B s'arrête, étant devenue inutile. En effet, le noeud B a une note supérieure à la note témoin de son père. Et comme

son père cherche à minimiser les coups, il choisira le noeud A. Donc à partir de maintenant, quelque soit la note trouvée parmi les fils de B, on sait que son père choisira le noeud A, d'où l'inutilité de l'exploration. Cette opération d'abandon de l'exploration des noeuds fils s'appelle élagage, et elle nous fait gagner beaucoup de temps pendant la réflexion de l'ordinateur. Continuons l'exploration de l'arbre.



Au noeud C, l'exploration des fils continue tant qu'aucune note trouvée n'est supérieure à celle du père (qui jusqu'ici était de 30). Quand le noeud C a été complètement exploré, on constate que sa note est plus petite que celle de son père; elle est donc retournée et devient la nouvelle note témoin. On passe enfin au quatrième coup possible.



Le premier fils de D rapporte une note de 26. Comme cette note est supérieure à celle de son père, l'exploration de D est interrompue (élagage). En effet, l'ordinateur aurait pu trouver des notes supérieures à 26, mais de toutes façons, son père cherche à minimiser, et il aurait donc choisi le noeud C.

On peut donc résumer brièvement l'élagage :

1) on est sur un niveau qui maximise la note de ses fils.

- si la note de ce niveau est supérieure à celle de son père alors on arrête l'exploration et on retourne la note vers son père.

- tant qu'elle est inférieure, on continue l'exploration des fils.

- si on a terminé l'exploration, et si la note est inférieure à celle de son père, alors celle-ci est retournée et servira de témoin de comparaison avec les autres fils

2) on est sur un niveau qui minimise la note de ses fils

- si la note de ce niveau est inférieure à celle de son père alors on arrête l'exploration et on remonte la note vers son père.

- tant qu'elle est supérieure, on continue l'exploration des fils.

- si on a terminé l'exploration, et si la note est inférieure à celle de son père, alors celle-ci est retournée et servira de témoin de comparaison avec les autres fils

Voici le programme de fin de partie (on rappelle que l'ordinateur a les 0, représentés par la couleur noire, et que le joueur a les 2; il reste 11 cases à occuper donc la profondeur de l'arbre à explorer est de 11) :

```
const direction : array[1..8] of integer =
                (-11, -10, -9, -1, 1, 9, 10, 11);
      work0 : array[0..99] of byte =
                (8, 9,9,9,9,9,9,8, 8,
                 8, 9,0,0,0,0,0,9, 9,
                 9, 9,9,0,0,0,0,9,2, 9,
                 9, 2,2,0,0,0,2,0,2, 9,
                 9, 2,2,0,2,2,2,0,2, 9,
                 9, 2,2,2,2,0,0,2,2, 9,
                 9, 2,2,2,2,0,0,2,2, 9,
                 9, 2,9,2,0,0,0,9,9, 9,
                 9, 9,9,2,2,2,2,2,9, 9,
                 9, 9,9,9,9,9,9,9,9, 9);

       COEF_ORDINATEUR      = -1;
       ORDINATEUR           =  0;
       COEF_JOUEUR          =  1;
       JOUEUR               =  2;

type   othellier            = array[0..99] of byte;
       utile                = array[0..18] of integer;
       n_othelliers         = array[0..18] of othellier;

var    work                 : othellier;
       position             : utile;
       temoin               : utile;
       nb_coups             : utile;
       tab_pense            : n_othelliers;
       profondeur           : integer;

function compte_les_pions(couleur : integer) : integer;
var x,total_pions : integer;
begin
x:=10;total_pions:=0;
repeat
  inc(x);
  if work[x]=couleur then inc(total_pions);
until x=88;
compte_les_pions:=total_pions;
end;


function trouve_pos(niveau,joueur : integer):integer;
var a,xx,xv,opposant,delta : integer;
begin
xx:=position[niveau];
trouve_pos:=0;
opposant:=2-joueur;
repeat
   repeat
      inc(xx)
   until (work[xx]=9) or (xx>88);
   position[niveau]:=xx;
   if ((1+xx) mod 10>1) and (xx<89) then
      for a:=1 to 8 do
          begin
          delta:=direction[a];
          if work[xx+delta]=opposant then
             begin
             xv:=xx+delta;
             while work[xv]=opposant do
```

```
               inc(xv,delta);
             if work[xv]=joueur then
               begin
               trouve_pos:=xx;
               exit
               end;
             end;
          end;
until (xx>88);
end;


procedure joue_coup(position_courante,joueur:integer);
var a,xx,opposant,delta : integer;
begin
opposant:=2-joueur;
for a:=1 to 8 do
    begin
    delta:=direction[a];
    xx:=position_courante+delta;
    while work[xx]=opposant do
          inc(xx,delta);
    if work[xx]=joueur then
       repeat
          dec(xx,delta);
          work[xx]:=joueur;
       until xx=position_courante;
    end;
for a:=1 to 8 do
    if work[position_courante+direction[a]]=8 then
       work[position_courante+direction[a]]:=9;
end;


procedure arbre(c,level:integer);
var x,next,last : integer;
begin
nb_coups[level]:=0;
position[level]:=10;
temoin[level]  :=c*25000;
temoin[level+1]:=-temoin[level];
next           :=level+1;
last           :=level-1;
repeat
  move(tab_pense[last],work,100);
  x:=trouve_pos(level,1+c);
  if x>0 then
     begin
     inc(nb_coups[level]);
     joue_coup(x,1+c);
     move(work,tab_pense[level],100);
     if profondeur<>level
        then arbre(-c,next)
        else temoin[next]:=compte_les_pions(ORDINATEUR);
     end
  else
     if nb_coups[level]=0 then
        if nb_coups[last]=0
           then temoin[level]:=compte_les_pions(ORDINATEUR)
           else begin
                move(work,tab_pense[level],100);
                inc(profondeur);
                arbre(-c,next);
                dec(profondeur);
                end;
  case c of
   -1 : begin
        if temoin[level]<temoin[next] then temoin[level]:=temoin[next];
        if temoin[level]>temoin[last] then exit;  //--- élagage
        end;
    1 : begin
        if temoin[level]>temoin[next] then temoin[level]:=temoin[next];
        if temoin[level]<temoin[last] then exit;  //--- élagage
        end;
    end;
until position[level]>88;
end;
```

```
procedure niveau_un;
var x,meilleur_coup,temoin1 : integer;
begin
move(work,tab_pense[0],100);
meilleur_coup:=0;
nb_coups[1]:=0;
temoin[1]:=0;
position[1]:=10;
profondeur:=11;
repeat
  move(tab_pense[0],work,100);
  x:=trouve_pos(1,ORDINATEUR);
  if x>0 then
     begin
     inc(nb_coups[1]);
     if nb_coups[1]=1 then meilleur_coup:=x;
     joue_coup(x,2);
     move(work,tab_pense[1],100);
     arbre(COEF_JOUEUR,2);
     temoin1:=temoin[1];
     if temoin[1]<temoin[2] then temoin[1]:=temoin[2];
     if temoin[1]>temoin1   then meilleur_coup:=position[1];
     end;
until position[1]>88;
form1.memo1.lines.add('Le meilleur coup pour les noirs est
'+inttostr(meilleur_coup));
form1.memo1.lines.add('Il rapporte '+inttostr(temoin[1])+' pions noirs en fin de
partie !');
end;


procedure tform1.affiche_jeu;
var x,y : integer;
begin
for x:=1 to 8 do
for y:=1 to 8 do
    if work[x+10*y]=JOUEUR then
       with image1.picture.bitmap.canvas do
          begin
          brush.color:=clWhite;
          ellipse((x-1)*30+5,(y-1)*30+5,(x-1)*30+25,(y-1)*30+25);
          end
    else
    if work[x+10*y]=ORDINATEUR then
       with image1.picture.bitmap.canvas do
          begin
          brush.color:=clBlack;
          ellipse((x-1)*30+5,(y-1)*30+5,(x-1)*30+25,(y-1)*30+25);
          end;
end;

procedure TForm1.FormShow(Sender: TObject);
var i : integer;
begin
image1.picture.bitmap:=tbitmap.create;
with image1.picture.bitmap do
   begin
   width:=240;
   height:=240;
   canvas.brush.style:=bsSolid;
   for i:=1 to 7 do
       with image1.picture.bitmap.canvas do
          begin
          moveto(i*30,0);lineto(i*30,240);
          moveto(0,i*30);lineto(240,i*30);
          end;
   end;
image1.autosize:=true;
move(work0,work,100);fillchar(temoin,sizeof(temoin),#0);
affiche_jeu;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
niveau_un;
```

```
end;
```