Informatique – Correction du TP n° 1 – Exercices 1 à 6 et 8

Exercice 1 – Soit la suite définie par $u_0 = 0$, et pour tout $n \in \mathbb{N}$, $u_{n+1} = \sqrt{3u_n + 4}$. On montre facilement que cette suite converge vers 4 en croissant.

1. Écrire un programme demandant à l'utilisateur un entier n en renvoyant tous les termes de la suite jusqu'à u_n.

```
program tp1exo1a;
uses crt;

var u:real;
    n,i:integer;

begin
  clrscr;
  writeln('Entrez une valeur positive de n: ');
  readln(n);
  u:=0;                 {Initialisation}
  for i:=1 to n do      {Le premier terme calculé est u_1, le dernier u_n}
    u:=sqrt(3*u+4);     {On passe d'un terme au suivant}
  writeln('u_',n,'=',u);
end.
```

Le principe de calcul d'une suite récurrence d'ordre 1 est toujours le même. Il faut voir l'utilisation de la boucle `for` comme une récurrence (=une répétition d'une certaine opération) : l'initialisation se fait avant (à ne pas oublier !), et l'hérédité consiste en l'instruction effectuée dans la boucle (passage d'un terme au suivant).On utilise donc une variable qui prendra successivement toutes les valeurs de la suite les unes après les autres. On actualise la valeur de la variable à chaque passage dans la boucle à l'aide de la relation de récurrence.

Pensez à faire coïncider la valeur du compteur de la boucle à l'indice du terme calculé dans cette boucle. Par exemple, ici, le terme calculé lors du premier passage dans la boucle est u_1. Ainsi, je commence la boucle à 1.

Remarquez que si $n = 0$, la boucle va de 1 à 0. Dans ce cas, aucun passage dans la boucle n'est effectué. La valeur renvoyée au final est donc la valeur donnée lors de l'initialisation.

Enfin, on ne peut pas exclure que l'utilisateur entre une valeur incohérente (un entier négatif en l'occurrence). Pour être tout à fait parfait, il faudrait faire un test de cohérence portant sur la variable n.

2. Écrire un programme renvoyant le plus petit entier n pour lequel $u_n > 3,99999999$.

```
program tp1exo1b;
uses crt;

var u:real;
    n:integer;

begin
  clrscr;
  u:=0;
  n:=0;
```

```
  repeat
    u:=sqrt(3*u+4);
    n:=n+1;
  until
    u>3.99999999;
  writeln('Le plus petit entier n pour lequel u_n>3.99999999 est ',n);
end.
```

Dans cet exercice, on ne sait pas à l'avance combien de fois il va falloir passer dans la boucle. Par conséquent, on ne connaît pas la borne supérieure du compteur. On ne peut donc pas utiliser de boucle `for`, qui présuppose la connaissance préalable de cette valeur. En revanche, l'arrêt est déterminé par une condition. On peut dans ce cas utiliser au choix une boucle `repeat...until` (on exprime une condition d'arrêt) ou `while...do` (on exprime une condition de continuation). Remarquez que dans la boucle `repeat`, le test est fait après, donc on passe au moins une fois dans la boucle, contrairement à la boucle `while`.

Remarquez également que dans une boucle `repeat`, il n'y a pas d'ambiguïté pour savoir où se terminent les instructions qui doivent être répétées (ce sont les instructions entre `repeat` et `until`). Par conséquent, il n'est pas nécessaire de regrouper ces instructions dans une instruction composée `begin ... end`.

Enfin, le fait de ne plus utiliser la boucle `for` nous fait perdre le compteur, donc l'indice de la suite. Il faut donc le recréer manuellement. D'où la variable n, représentant l'indice de la suite. On fait évoluer n parallèlement à u : Dès qu'on modifie u, on modifie aussi l'indice en conséquence. Autrement dit, lorsqu'on initialise u, on initialise aussi n ; lorsqu'on calcule le terme u suivant, on calcule aussi l'indice suivant (donc on ajoute 1 à l'indice).

Exercice 2 –
Calculer $\sum_{n=0}^{1000} u_n$ où $u_0 = 1$ et $\forall n \geqslant 0$, $u_{n+1} = \dfrac{1}{u_n + 1}$.

```
program tp1exo2;
uses crt;

var n:integer;
    S,u:real;

begin
  clrscr;
  u:=1;             {Initialisation de u}
  S:=1;             {Initialisation de S: on y met déjà u_0}
  for n:=1 to 1000 do
    begin
      u:=1/(u+1);   {On calcule le terme suivant de la suite}
      S:=S+u;       {On l'ajoute à la somme}
    end;
  writeln('La somme demandée vaut: ',S);
end.
```

On calcule à la fois une suite récurrente et une somme. Remarquez qu'on ne peut pas calculer d'abord tous les termes de la suite et ensuite les sommer (donc faire deux boucles séparées), car à l'issue de la première boucle (calcul de u), les valeurs de la suite étant écrasées au fur et à mesure, on n'a retenu que la valeur du dernier terme, ce qui est insuffisant pour faire la somme ! Un moyen de s'en sortir (maladroit !) serait de stocker toutes les valeurs dans un tableau, mais cela mange beaucoup de place en mémoire. Une deuxième solution consiste à calculer la somme en même temps que le terme général : à chaque nouveau terme calculé, on l'ajoute tout de

suite aux précédents. Ainsi, on utilise deux variables u pour le terme général, S pour la somme partielle, et on fait évoluer ces deux quantités simultanément.
Remarquez la nécessité de regrouper les instructions répétées dans une instruction composée pour une boucle `for`, lorsqu'il y en a plusieurs.

Exercice 3 –
Soit $F_0 = 0$, $F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$ (suite de Fibonacci). Écrire une fonction prenant en paramètre une valeur de n et calculant F_n.

```
function Fibonacci(n:integer):longint;


begin
  a:=0;       {Initialisation de a et b, deux termes consécutifs de la suite}
  b:=1;
  for i:=2 to n do
    begin
      c:=a+b; {Calcul du terme suivant, stocké provisoirement dans c}
      a:=b;   {actualisation de a et b: on décale tous les indices d'un cran}
      b:=c;
    end;
  if n<0 then writeln('Erreur!')
         else if n=0 then F:=0;
                     else F:=b;
end;
```

Le calcul de suites définies par une récurrence d'ordre 2 (ou plus) est un grand classique. Ici, nous devons avoir constamment en mémoire deux termes consécutifs afin de calculer le suivant. Il nous faut donc deux variables a et b pour stocker ces valeurs. Elles sont initialisées avant la boucle. Dans la boucle, on décale les indices de 1. Autrement dit, on remplace la valeur de a par la suivante, à savoir b, et la valeur de b par la suivante, à savoir $a + b$. Si on commence par actualiser la valeur de a, on perd l'ancienne valeur de a, nécessaire pour le calcul de $a + b$. Si on commence par actualiser b, on perd l'ancienne valeur de b, nécessaire pour actualiser a. D'où la nécessité de stocker provisoirement le terme suivant $a + b$ dans une variable auxiliaire c, ce qui nous permet d'écraser l'ancienne valeur de a qui ne nous est plus utile, et ensuite d'actualiser b.
Encore une fois, faite coïncider le compteur à l'indice du terme calculé dans la boucle. Le terme calculé lors du premier passage est F_2, d'où un départ de boucle à 2.
La structure conditionnelle finale sert à afficher la bonne valeur dans le cas où l'utilisateur entre la valeur 0. Remarquez que si l'utilisateur rentre la valeur 1, aucun passage n'est effectué dans la boucle et b a bien la valeur de F_1.
Remarquez également que le type choisi pour le résultat est `longint`. En effet, les nombres de Fibonacci croissent très vite et le type `integer` est trop limité. Même le type `longint` ne permet pas de calculer F_n pour beaucoup de valeurs de n.
Enfin, remarquez qu'à la fin d'une fonction, il y a un point-virgule et non un point (ce n'est pas la fin du programme).

Exercice 4 – On définit la suite de Syracuse par $u_0 \in \mathbb{N}^*$, et

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

On veut vérifier la propriété suivante : il existe un rang N tel que $u_N = 1$ (et à partir de ce rang, la suite boucle : 4, 2, 1, 4, 2, 1, etc.). Écrire un programme demandant à l'utilisateur une valeur initiale u_0, calculant et affichant les différents termes de la suite tant qu'ils ne sont pas égaux à 1, et affichant pour terminer la première valeur de N pour laquelle $u_N = 1$, ainsi que la plus grande valeur obtenue pour u_n.

```
program tp1ex4;

uses crt;

var u,max:longint;
    n:integer;

begin
  clrscr;
  writeln('Entrez une valeur initiale strictement positive: ');
  readln(u);
  n:=0;
  max:=u;
  writeln('u0=',u);
  while u<>1 do
    begin
      if u mod 2=0
      then
        u:=u div 2
      else
        u:=3*u+1;
      n:=n+1;
      if u>max then max:=u;
      writeln('u',n,'=',u);
    end;
  writeln('Il a fallu aller jusqu''au rang ',n);
  writeln('La valeur maximale trouvée est ',max);
end.
```

Remarquez que u doit être de type entier pour pouvoir utiliser `mod` dans le but d'étudier la parité de u. Mais ceci empêche d'utiliser la division usuelle, sous forme d'une barre, car le résultat d'une telle division est de type réel (même si le résultat est un entier : on aura toutes les décimales égales à 0 dans le résultat). Ainsi, on est obligé de se rabattre sur la division euclidienne `div`, donnant le quotient de la division euclidienne. Ici, comme le reste est nul (u est pair), ce quotient donne la même valeur que la division réelle.
Notez également la façon de déterminer le maximum d'un ensemble de valeurs, en parcourant les éléments les uns après les autres, et en comparant chaque nouvel élément rencontré au maximum des précédents (qu'on a stocké dans une variable) : si le nouvel élément est supérieur, c'est lui le nouveau maximum provisoire.

Exercice 5 –
Écrire une fonction prenant en paramètre deux réels strictement positifs a et e, et calculant une valeur approchée de la somme (dont on justifiera la convergence) $\sum_{n=1}^{+\infty} \frac{(-1)^n}{n^a}$ à e près.

Soit $a > 0$. Justifions d'abord la convergence de la série $\sum \frac{(-1)^n}{n^a}$. On utilise pour cela la méthode classique des séries alternées $\sum (-1)^n a_n$ où (a_n) est positive décroissante de limite nulle. Soit pour tout $n \in \mathbb{N}^*$, la somme partielle

$$S_n = \sum_{k=1}^{n} \frac{(-1)^k}{k^a}$$

On a alors :

- $\forall n \in \mathbb{N}^*, \quad S_{2n+2} - S_{2n} = \frac{(-1)^{2n+1}}{(2n+1)^a} + \frac{(-1)^{2n+2}}{(2n+2)^a} = -\frac{1}{(2n+1)^a} + \frac{1}{(2n+2)^a} < 0$, car $\left(\frac{1}{(2n+2)^\alpha}\right)$ est décroissante. Donc $(S_{2n})_{n\in\mathbb{N}^*}$ est décroissante
- $\forall n \in \mathbb{N}^*, \quad S_{2n+1} - S_{2n-1} = \frac{(-1)^{2n}}{(2n)^a} + \frac{(-1)^{2n+1}}{(2n+1)^a} = \frac{1}{(2n)^a} - \frac{1}{(2n+1)^a} > 0$, car $\left(\frac{1}{(2n+2)^\alpha}\right)$ est décroissante. Donc $(S_{2n+1})_{n\in\mathbb{N}^*}$ est croissante.

• $\forall n \in \mathbb{N}^*,\ \ S_{2n+1} - S_{2n} = \frac{(-1)^{2n+1}}{(2n+1)^a}$, donc $\lim_{n\to+\infty} S_{2n+1} - S_{2n} = 0$.

Ainsi, les deux suites $(S_{2n})_{n\in\mathbb{N}^*}$ et $(S_{2n+1})_{n\in\mathbb{N}}$ sont adjacentes, donc convergent vers une limite commune S. Par conséquent, la suite $(S_n)_{n\in\mathbb{N}^*}$ admet une limite, de valeur S. On en déduit que la série est convergente.

Vous pouvez remarquer que si $a > 1$, cette étude est inutile, car on obtient directement l'absolue convergence de la série par les résultats de convergence des séries de Riemann. En revanche, si $a \in]0, 1]$, la série n'est pas absolument convergente. Vous avez donc là un exemple de série semi-convergente (convergente mais pas absolument).

L'argument ci-dessus dit de plus que $(S_{2n})_{n\in\mathbb{N}^*}$ est en-dessous de sa limite et $(S_{2n+1})_{n\in\mathbb{N}}$ est au-dessus de sa limite. Ainsi, deux termes consécutifs sont de part et d'autre de la limite S. Donc, pour tout $n \geqslant 2$, $|S_n - S| \leqslant |S_n - S_{n-1}|$. Par conséquent, si on trouve un indice n tel que $|S_n - S_{n-1}| < e$, on aura aussi $|S_n - S| < e$. Nous allons donc calculer la somme partielle jusqu'à avoir cette condition $|S_n - S_{n-1}| < e$, facile à déterminer.

Nous obtenons donc :

```
function tp1ex5(a,e:real):real

var n:integer;
    u,S:real;

begin
  S:=0;          {initialisation de la somme: somme vide}
  n:=0;          {initialisation de l'indice: la somme vide correspond
                  à l'indice n=0}
  repeat
    n:=n+1;      {indice suivant}
    u:= exp(-a*ln(n));  {calcul du terme général}
    S:=S+u;      {somme partielle suivante}
  until
    abs(u)<e;    {u est la différence de deux S consécutifs}
  tp1ex5:=S;
end;
```

Remarquez, dans ce programme comme dans les précédents, que je mets des commentaires pour rendre le programme plus facile à comprendre. Ne vous en privez pas, que ce soit sur l'ordinateur ou sur votre copie. Les commentaires, dans un programme en Pascal, doivent être placés entre accolades.

Exercice 6 – Soit $f(x, y) = x\cos y + y\cos x$. On définit une suite u_n par $u_0 = 0$, $u_1 = 1$, $u_{n+2} = f(u_{n+1}, u_n)$ si n est pair, et $u_{n+2} = f(u_{n+1}, u_{n-1})$ si n est impair. Afficher les N premières valeurs de (u_n). Modifier le programme pour répondre à la même question avec $f(x, y) = x\mathrm{e}^y - y\mathrm{e}^x$.

Globalement, il s'agit d'une relation de récurrence d'ordre 3, puisqu'au pire, on a besoin des 3 termes précédents de la suite pour calculer un terme donné, même si, selon la parité, 2 termes suffisent parfois. Il est bien sûr possible d'être malin et d'exploiter le fait qu'une fois sur deux, les la connaissance des deux termes précédents suffit. Mais ramenons-nous plutôt à une méthode connue et sûre. Il s'agit donc de calculer une récurrence d'ordre 3. Pour cela, nous utilisons trois variables u, v, w qui stockent trois termes consécutifs, à l'aide desquels on peut calculer le terme suivant, stocké momentanément dans une variable tampon t. On effecture ensuite un glissement pour actualiser la donnée du triplet (u, v, w).

Remarquez que l'initialisation se fait sur trois termes, alors qu'on ne dispose dans l'énoncé que des deux premiers termes. On calcule donc séparément le troisième terme.

```
program tp1ex6;

uses crt;
```

```
var n,i:integer;
    u,v,w,t:real;

function f(x,y:real):real;
  begin
    f:=x*cos(y)+y*cos(x);
  end;

begin
  clrscr;
  writeln('Entrez n ');
  readln(n);
  u:=0;
  v:=1;
  w:=f(v,u);
  writeln('u0=',u);
  if n > 0 then writeln('u1=',v);
  if n > 1 then writeln('u2=',w);
  for i:=3 to n do
    begin
      if i mod 2= 0 then t:=f(w,v)
                    else t:=f(w,u);
      u:=v;
      v:=w;
      w:=t;
      writeln('u',i,'=',w);
    end;
end.
```

Pour changer de fonction f, il suffit de faire le changement une fois, dans la structure fonction définie, au lieu d'aller changer l'expression à chaque occurrence.
Les tests avant les instructions d'affichage sont là pour éviter qu'on n'affiche trop de termes si $n = 0$ ou $n = 1$.

Exercice 8 – Écrire une procédure prenant en entrée un tableau contenant dans ses n premières entrées les $n + 1$ coefficients binomiaux $\binom{n}{k}$, $k \in \{0, \ldots, n\}$, (n-ième ligne du triangle de Pascal), et calculant la ligne suivante (utiliser le même tableau). Écrire un programme affichant sous forme de triangle le triangle de Pascal jusqu'à la ligne 20
On définit une constante pour la taille du tableau, et on définit un nouveau type pour pourvoir passer les tableaux en paramètre. Le fait de définir la taille à l'aide d'une constante permet de modifier ensuite cette valeur plus facilement si nécessaire.
On écrit d'abord une procedure permettant d'initialiser avec un tableau rempli de 0.
L'opération permettant de passer d'une ligne du triangle de Pascal à la suivante est la formule de Pascal :

$$\forall n \in \mathbb{N}, \forall k \in \mathbb{N} \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Remarquez que cette formule est aussi vraie pour $k > n$, puisque dans ce cas, par convention, les coefficients binomiaux sont nuls.
Ainsi, la i-ième case du nouveau tableau est égale à la somme des cases i et $i-1$ de l'ancien tableau. L'opération d'actualisation sera donc $T[i] := T[i] + T[i-1]$, à effectuer sur tous les indices i.
Mais, si on considère des valeurs croissantes de i, pour un i donné, on écrase l'ancienne valeur de $T[i]$, dont on a encore besoin pour le calcul de $T[i+1]$, et le résultat obtenu ne sera pas le bon.

En prenant des indices décroissants, en revanche, il n'y a pas de problème. C'est pourquoi on effectue une boucle for avec une décrémentation downto.
Avant l'affichage du tableau entier, on fait une procédure d'affichage d'une ligne donnée. On affiche en ligne les éléments les un après les autres. Le recours aux ln (et notamment la division par $\ln 10$) permet de connaître le nombre de chiffres de l'entier à afficher, et de choisir la longueur de l'espace à insérer en conséquence, afin d'aligner les différentes colonnes du tableau. Je ne vous demandais pas un tel perfectionnisme...
Le corps du programme consiste alors en une initialisation : tableau nul, sauf en 0 (où il y a un 1). Puis, à l'aide d'une boucle, on calcule successivement les lignes du tableau grâce à la procedure lignesuivante, et on les affiche grâce à la procedure affiche.

```
program tp3ex5;

uses crt;

const Nmax=20;
type tableau=array[0..Nmax] of longint;
var T:tableau;
    n,k:integer;

procedure zero(var T:tableau);
  var i:integer;
  begin
    for i:=0 to Nmax do T[i]:=0;
  end;

procedure lignesuivante(var T:tableau);
  var i:integer;
  begin
    for i:=Nmax downto 1 do T[i]:=T[i]+T[i-1];  {formule de Pascal}
  end;

procedure affiche(T:tableau; n:integer);
  var i,j,esp:integer;
  begin
    for i:=0 to n do
      begin
        write(T[i]);                    {write pour ne pas aller à la ligne}
        esp:=6-trunc(ln(T[i])/ln(10)); {calcul du nombre d'espaces à insérer}
        for j:=1 to esp do write(' '); {pour avoir un alignement parfait}
      end;
    writeln;                            {retour à la ligne}
  end;

begin
  clrscr;
  writeln('Jusqu''à quelle ligne écrire le triangle de Pascal?');
  readln(n);
  zero(T);                 {initialisation des lignes}
  T[0]:=1;
  affiche(T,0);            {affichage de la première ligne, d'indice 0}
  for k:=1 to n do
```

```
    begin
      lignesuivante(T);    {calcul successif des différentes lignes}
      affiche(T,k);        {et affichage dans la foulée}
    end;
end.
```