Informatique – TP n° 1 – Correction des exercices 7,9,10

Exercice 7 –
Tout d'abord, pour tout cet exercice, je suppose qu'on a défini dans l'entête du programme un type tableau, de la sorte :

```
const Nmax=1000;
type tableau=array[1..Nmax] of integer;
```

Vous remarquerez que dans les fonctions et les procedures de calcul, on ne fait aucun affichage, ni aucune demande de valeur (pas d'interface avec l'utilisateur) : on travaille uniquement avec les paramètres (valeurs fournies, valeurs récupérées en paramètres passés en variables). L'interface avec l'utilisateur se fait, si besoin, dans le corps de programme.

1. On parcourt le tableau en commençant par le début, et en gardant à chaque fois en mémoire la valeur du maximum provisoire, c'est-à-dire de la plus grande valeur rencontrée jusqu'à présent. À chaque nouvelle étape, on compare le nouvel élément considéré au maximum provisoire : si le nouvel élément est plus grand que le maximum provisoire, c'est lui le nouveau maximum provisoire, sinon, le maximum provisoire n'est pas modifié. On note n la taille réellement utilisée du tableau, l'entier `Nmax` étant un majorant de cette taille, supposé suffisamment grand pour couvrir nos besoins.

```
function maximum(T:tableau; n:integer):real;
                     {n'oubliez pas le type de sortie!}

var i,M:integer;     {i: compteur; M:maximum provisoire}
                     {ils sont pris en variables locales, car n'ont
                     pas de pertinence hors de la fonction}
begin
  M:=T[1];           {initialisation du maximum provisoire}
  for i:=2 to n do
    if T[i] > M then M:=T[i];
                     {actualisation du maximum local si on trouve une
                     valeur plus grande}
  maximum:=M;
                     {n'oubliez pas de définir la valeur de sortie de
                     la fonction!}
end;                 {pas de point, c'est la fin de la fonction, pas
                     du programme}
```

2. On compare le deuxième élément au premier, et on les place respectivement. On compare ensuite le 3e au deux premiers, et on le place où il faut, en l'insérant au bon endroit. On continue ainsi jusqu'au dernier élément.

```
procedure tri(var T:tableau; n:interger);
                     {le tableau ressort trié, donc modifié:
                     n'oubliez pas de le passer en variable}
var i,j,k : integer;   {compteurs}
    x : real;          {variable tampon (ie sauvegarde provisoire)}

begin
  for i:=2 to n do
```

```
    begin
      j:=0;
      repeat
        j:=j+1;
      until T[j] >= T[i];
                    {recherche parmi les éléments déjà triés du
                    premier élément plus grand que T[i]. Ainsi, T[i]
                    sera à insérer juste avant cet élément.
                    Remarquez que si tous les éléments sont plus
                    petits que T[i], la boucle s'arrête lorsque j=i,
                    du fait de l'inégalité large: insérer T[i] avant
                    T[i] revient alors à ne rien faire, T[i]
                    étant déjà en place}
      x:=T[i];      {sauvegarde de la valeur de T[i]}
      for k:=i-1 downto j do
        T[k+1]:=T[k]; {décalage des termes du tableau, de T[j] à T[i-1]}
      T[j]:=x;      {mise en place de l'ex-valeur de T[i]}
    end;
end;
```

3. Une fois le tableau trié, il suffit de chercher la valeur du milieu si le nombre de termes est impair, ou de prendre la moyenne des deux termes du milieu si le nombre de termes est pair. Comme on utilise la procedure tri, il faut placer cette fonction après la procedure tri dans le programme.

```
function mediane(T:tableau; n:integer): real;


begin
  tri(T,n);                  {on commence par trier le tableau}
  if n mod 2=1 then
    mediane:= T[n div 2+1]   {pas de point-virgule avant else!}
  else
    mediane:= (T[n div 2] + T[n div 2 +1])/2;
end;
```

   Vous remarquerez qu'à l'issue de la fonction, le tableau T reprend la valeur initiale (le tri est oublié) puisque T est passé en valeur dans cette fonction.

Exercice 9 – Le crible d'Eratosthène consiste à commencer à barrer, dans un tableau contenant tous les entiers jusqu'à 32000, tous les multiples stricts de 2, puisqu'ils ne sont pas premiers. On cherche ensuite le premier nombre non barré (3 en l'occurrence), qui est forcément premier (non divisé par un nombre non égal à 1 plus petit) et on recommence l'opération avec les multiples stricts de 3, etc. Dans la procédure que nous allons écrire, pour barrer un nombre nous remplacerons sa valeur par 0.

- La première étape va donc consister à remplir un tableau avec les entiers successifs de 2 à 32000
- On dispose d'une variable p correspondant à la recherche des entiers premiers successifs. Initialement, $p = 2$.
- On barre tous les multiples stricts de p inférieurs à 32000
- On recherche l'entier premier suivant, donc le nombre non barré strictement supérieur à p ; ce sera, s'il existe la nouvelle valeur de p
- On recommence avec cette nouvelle valeur de p (boucle)
- A la fin, on parcourt l'ensemble du tableau, en affichant tous les nombres non mis à 0, qui sont les nombres premiers recherchés. Ici, on fait un affichage dans la procedure, puisque c'est ce qui est demandé !

On suppose qu'on a défini une constant `Nmax` et un type tableau dans l'entête du programme de la manière suivante

```
const Nmax=32000;
type tableau = array[2..Nmax] of integer;
```

La procedure s'écrit alors :

```
procedure eratosthene; {pas de paramètre: le tableau est défini à
                       l'intérieur de la procédure}
var T: tableau;
    p,k:integer;

begin
  for k:=2 to Nmax do
    T[k]:=k;
  p:=2;
    repeat
      for k:=2 to Nmax div p do  {dernier multiple de p avant Nmax}
        T[p*k]:= 0;
      repeat                     {recherche de la valeur de p suivante}
        p:=p+1;
      until
        (T[p] <>0) or (p=Nmax);
    until
      p=Nmax;                    {on arrête le procédé lorsque p a
                                 atteint la valeur maximale}
   for k:=2 to Nmax do
     if T[k] <>0 then writeln(T[k]);
                                 {affichage des nombres premiers}
end;
```

Exercice 10 – Pour pouvoir passer des polynômes en paramètre, il faut d'abord avoir défini un nouveau type, dans l'entête du programme :

```
const Nmax=50;
type polynome=array[0..Nmax] of real;
```

1. L'alogrithme de Hörner est à connaître. Il est explicitement au programme. Il consiste à exploiter la factorisation suivante :

$$a_0 + a_1X + a_2X^2 + \cdots + a_nX^n = a_0 + X(a_1 + X(a_2 + \cdots + X(a_{n-1} + Xa_n)))$$

   Ainsi, partant d'une variable contenant initialement a_n, on la multiplie par a (valeur en laquelle on veut évaluer) puis on ajoute a_{n-1}. On mutliplie le résultat par a et on ajoute a_{n-2}, etc. À chaque étape, on mutliplie par a et on ajoute le coefficient précédent, jusqu'à parvenir au coefficient a_0. Cela donne donc la fonction suivante (on l'écrit sous forme de fonction, puisque le but est de calculer une valeur numérique)

```
function horner(P: polynome; d:integer;a:real):real
                  {d est le degré de P, a le réel en lequel on évalue}
var i:integer;
    Pa: real;     {variable de calcul: on y stocke les résultats partiels}

begin
  Pa:=P[d];       {initialisation avec le coefficient dominant}
```

```
  for i:=d-1 downto 0 do
    Pa:=Pa*a+P[i]; {on répète l'opération consistant à multiplier par
                    $a$ et ajouter le coefficient précédent}
  horner:=Pa;
end;
```

2. Le monôme $a_k X^k$ se dérive en $k a_k X^{k-1}$ Ainsi, pour tout $k \in [\![1, d]\!]$, d étant le degré de P, le coefficient de degré $k-1$ de P' est ka_k. Ainsi, il faut décaler les coefficients de P vers la gauche, en les multipliant au passage par k.

   On commence ce décalage par la gauche, puisque c'est là qu'on dispose d'une « case vide » permettant d'amorcer le décalage (le coefficient constant de P ne nous est d'aucune utilité, on peut l'écraser). Ainsi, on obtient la procedure suivante (ici, on est obligé d'écrire une procédure, le résultat du calcul n'étant pas de type numérique) :

```
procedure derive(var P:polynome; var d:integer);
var k:integer;
begin
  for k:=1 to d do
    P[k-1]:= k*P[k];
  P[d]:=0;
  d:=d-1;              {modification du degré}
end;
```

   Vous remarquerez qu'ici, le paramètre P joue à la fois le rôle d'entrée (c'est par ce paramètre qu'on fournit les données du calcul) et de sortie (c'est aussi par ce paramètre qu'on récupère le résultat du calcul) Il faut donc le passer en paramètre variable, pour que les modifications apportées soient prises en compte.

3. On rappelle qu'un polynôme P admet a pour racine d'ordre de multiplicité r si et seulement si $P(a) = P'(a) = \cdots = P^{(r-1)}(a) = 0$, et $P^{(r)}(a) \neq 0$. L'algorithme consiste donc à dériver successivement P jusqu'à obtenir une valeur non nulle pour l'évaluation de cette dérivée. Pour cela nous utilisons les deux fonctions ou procédures définies précédemment.

   Remarquons que la première chose à faire est un test pour savoir si $P(a) = 0$, et on dérive seulement si ce test est positif. Ainsi, il est préférable de choisir une structure répétitive qui fait le test avant l'instruction, donc la boucle `while ... do`.

   Cela donne l'algorithme suivant, écrit sous forme d'une fonction, puisque ce que l'on recherche est une valeur numérique.

```
function multiplicite(P:polynome;d:degre;a:real):integer;
var m:integer;
begin
  m:=0;           {initialisation de la multiplicité}
  while horner(P,d,a)=0 do
    begin
      derive(P,d);{cette instruction remplace P par sa dérivée}
      m:=m+1;     {la multiplicité est au moins un de plus}
    end;
  multiplicite:=m;
end;
```

   Remarquez que si $P = 0$, on boucle indéfiniment. Il faudrait donc faire un test initial pour savoir si $P = 0$, et traiter ce cas séparément. Pour simplifier, on a supposé ci-dessus que l'utilisateur entre un polynôme non nul.

   Remarquez aussi qu'au cours de cet algorithme, la valeur de P est modifiée, du fait des dérivations successives. Néanmoins, à l'issue du calcul, P reprend sa valeur initiale, puisque p est passé par valeur et non pas par variable.