Programmation Delphi : Algorithmes obligatoires

I^re B, 2016–2017

Version 4.3 du 11 septembre 2015

Table des matières

1 Mathématiques élémentaires

1.1 Fonction « puissance à exposant naturel »

1.1.1 Version itérative

La fonction suivante calcule b^e pour un exposant e naturel. Le cas particulier $b = e = 0$ n'est pas traité correctement.

```
function puissance(base:extended; expo:integer):extended;
var i:integer;
    p:extended;
begin
  p:=1;
  for i:=1 to expo do
    p:=p*base;
  puissance:=p
end;
```

1.1.2 Version récursive

La fonction suivante calcule b^e pour un exposant e naturel. Le cas particulier $b = e = 0$ n'est pas traité correctement.

```
function puissance(base:extended; expo:integer):extended;
begin
  if expo=0 then
    puissance:=1
  else
    puissance:=base*puissance(base,expo-1)
end;
```

1.2 Fonction « puissance rapide à exposant naturel »

```
function puissRapid(base:extended; expo:integer):extended;
begin
  if expo=0 then
    puissRapid:=1
  else if expo mod 2 = 0 then
    puissRapid:=puissRapid(base*base,expo div 2)
  else
    puissRapid:=base*puissRapid(base,expo-1)
end;
```

1.3 Fonction « factorielle »

Les fonctions suivantes calculent $n!$ pour n naturel.

1.3.1 Version itérative

```
function factorielle(n:integer):integer;
var i:integer;
    fact:integer;
begin
  fact:=1;
  for i:=2 to n do
    fact:=fact*i;
  factorielle:=fact
end;
```

1.3.2 Version récursive

```
function factorielle(n:integer):integer;
begin
  if n<2 then
    factorielle:=1
  else
    factorielle:=n*factorielle(n-1)
end;
```

1.4 Fonction « pgcd »

Les fonctions suivantes déterminent le pgcd de deux nombres naturels non nuls.

1.4.1 Algorithme d'Euclide par soustraction

C'est sous cette forme, transcrite en langage de programmation moderne, qu'Euclide a présenté l'algorithme dans le livre 7 des *Éléments* (vers 300 av. J.-C.).

```
function euclideDiff(a,b:integer):integer;
begin
  while a<>b do
    if a>b then
      a:=a-b
    else
      b:=b-a;
  euclideDiff:=a
end;
```

1.4.2 Algorithme d'Euclide par division

```
function euclideDivi(a,b:integer):integer;
var c:integer;
begin
  while b>0 do begin
    c:=a mod b;
    a:=b;
    b:=c
  end;
  euclideDivi:=a
end;
```

1.5 Nombre premier ?

```
function primTest(n:integer):boolean;
var i,lim:integer;
    prim:boolean;
begin
  if n<2 then
    prim:=false
  else if n=2 then
    prim:=true
  else if n mod 2 = 0 then
    prim:=false
  else begin
    i:=3;
    prim:=true;
    lim:=round(sqrt(n));
    while (i<=lim) and prim do
      if n mod i = 0 then prim:=false
      else i:=i+2
  end;
  primTest:=prim
end;
```

2 Fréquence, minimum et maximum

2.1 Fréquence d'un élément dans une liste

```
function frequence(liste:TListBox; cle:string):integer;
var i,freq:integer;
begin
  freq:=0;
  for i:=0 to liste.Items.Count-1 do
    if liste.Items[i]=cle then
      freq:=freq+1;
  frequence:=freq
end;
```

2.2 Minimum d'une liste d'entiers non vide

```
function minimum(liste:TListBox):integer;
var i,mini:integer;
begin
  mini:=StrToInt(liste.Items[0]);
  for i:=1 to liste.Items.Count-1 do
    if StrToInt(liste.Items[i])<mini then
      mini:=StrToInt(liste.Items[i]);
  minimum:=mini
end;
```

2.3 Maximum d'une liste d'entiers non vide

```
function maximum(liste:TListBox):integer;
var i,maxi:integer;
begin
  maxi:=StrToInt(liste.Items[0]);
  for i:=1 to liste.Items.Count-1 do
    if StrToInt(liste.Items[i])>maxi then
      maxi:=StrToInt(liste.Items[i]);
  maximum:=maxi
end;
```

3 Algorithmes de tri

Les algorithmes de cette section réalisent un tri *lexicographique*. La procédure `echange` réalise l'échange de deux éléments d'une liste.

```
1  procedure echange(var liste:TListBox; i,j:integer);
2  var aux:string;
3  begin
4    aux:=liste.Items[i];
5    liste.Items[i]:=liste.Items[j];
6    liste.Items[j]:=aux
7  end;
```

3.1 Tri par sélection

3.1.1 Version itérative

```
1  procedure triSelectionI(var liste:TListBox);
2  var i,j,min:integer;
3  begin
4    for i:=0 to liste.Items.Count-2 do begin
5      min:=i;
6      for j:=i+1 to liste.Items.Count-1 do
7        if liste.Items[j]<liste.Items[min] then
8          min:=j;
9      echange(liste,i,min)
10   end
11 end;
```

L'algorithme devient légèrement plus rapide lorsqu'on effectue l'échange uniquement pour des indices différents. Pour cela, on remplace la ligne 9 par
`if i<>min then echange(liste,i,min)`

3.1.2 Version récursive

```
1  procedure triSelectionR(var liste:TListBox; debut:integer);
2  var j,min:integer;
3  begin
4    min:=debut;
5    for j:=debut+1 to liste.Items.Count-1 do
6      if liste.Items[j]<liste.Items[min] then
7        min:=j;
8    echange(liste,debut,min);
9    if debut<liste.Items.Count-2 then
10     triSelectionR(liste,debut+1)
11 end;
```

Appel de la procédure : `triSelectionR(liste, 0);`

3.2 Tri par insertion

3.2.1 Version itérative

```
procedure triInsertionI(var liste:TListBox);
var i,j:integer;
    candidat:string;
    termine:boolean;
begin
  for i:=1 to liste.Items.Count-1 do begin
    candidat:=liste.Items[i];
    j:=i;
    termine:=false;
    while (not termine) and (j>0) do
      if liste.Items[j-1] > candidat then begin
        liste.Items[j]:=liste.Items[j-1];
        j:=j-1
      end
      else termine:=true;
    if j<i then
      liste.Items[j]:=candidat
  end
end;
```

3.2.2 Version récursive

```
procedure triInsertionR(var liste:TListBox; fin:integer);
var j:integer;
    candidat:string;
    termine:boolean;
begin
  if fin>0 then begin
    triInsertionR(liste,fin-1);
    candidat:=liste.Items[fin];
    j:=fin;
    termine:=false;
    while (not termine) and (j>0) do
      if liste.Items[j-1] > candidat then begin
        liste.Items[j]:=liste.Items[j-1];
        j:=j-1
      end
      else termine:=true;
    if j<fin then
      liste.Items[j]:=candidat
  end
end;
```

Appel de la procédure : `triInsertionR(liste, liste.Items.Count-1);`

Dans l'algorithme précédent, l'appel récursif a lieu **au début** (ligne 7) du bloc d'instructions, avant les manipulations de la liste. Cela diffère de l'algorithme 3.1.2, où l'appel récursif a lieu **à la fin** du bloc d'instructions. On peut évidemment aussi implémenter le tri par insertion avec un appel récursif à la fin du bloc, comme l'illustre le code alternatif suivant :

```
1  procedure triInsertionR(var liste:TListBox; indiceCandidat:integer);
2  var j:integer;
3      candidat:string;
4      termine:boolean;
5  begin
6    if indiceCandidat<liste.Items.Count then begin
7      candidat:=liste.Items[indiceCandidat];
8      j:=indiceCandidat;
9      termine:=false;
10     while (not termine) and (j>0) do
11       if liste.Items[j-1] > candidat then begin
12         liste.Items[j]:=liste.Items[j-1];
13         j:=j-1
14       end
15       else termine:=true;
16     if j<indiceCandidat then
17       liste.Items[j]:=candidat;
18     triInsertionR(liste,indiceCandidat+1)
19   end
20 end;
```

Appel de la procédure : `triInsertionR(liste, 1);`
Comme la première tentative d'insertion consiste à insérer éventuellement le **deuxième** élément de la liste devant le premier, l'algorithme reçoit comme deuxième argument la valeur 1, c'est-à-dire l'indice du **deuxième** élément. (Grâce au test $j > 0$ de la ligne 10, l'algorithme fonctionnerait même si l'on passe comme deuxième argument la valeur 0, mais il effectuerait alors quelques instructions superflues.)

3.3 Tri rapide (*Quicksort*)

3.3.1 Version récursive

```
1  procedure triRapide(var liste:TListBox; g,d:integer);
2  var i:integer;
3  begin
4    if g<d then begin
5      i:=division(liste,g,d);
6      triRapide(liste,g,i-1);
7      triRapide(liste,i+1,d)
8    end
9  end;
```

Appel de la procédure, pour trier toute la liste : `triRapide(liste, 0, liste.Items.Count-1);`

3.3.2 Fonction auxiliaire « division »

```
1  function division(var liste:TListBox; g,d:integer):integer;
2  var i,j:integer;
3      pivot:string;
4  begin
5    pivot:=liste.Items[d];
6    j:=d-1;
7    i:=g;
8    while i<=j do
9      if liste.Items[i]<pivot then
10       i:=i+1
11     else if liste.Items[j]>pivot then
12       j:=j-1
13     else begin
14       echange(liste,i,j);
15       i:=i+1;
16       j:=j-1
17     end;
18   echange(liste,i,d);
19   division:=i
20 end;
```

La durée d'exécution du corps de la boucle while n'est pas optimale : lorsque l'indice j doit être diminué successivement plusieurs fois (ligne 12), le test à la ligne 9 est évalué chaque fois avant qu'on ne passe au test significatif de la ligne 11, alors que l'indice i n'a pas changé.

Nous proposons au lecteur intéressé une **version alternative** de la fonction « division » :

- ⋆ On choisit d'abord l'élément du milieu de la liste comme pivot qu'on place à la position g. Cette première étape est facultative, mais elle permet quelquefois d'obtenir un pivot plus équilibré, surtout lorsque la liste à trier est déjà presque triée, ou triée dans l'ordre inverse.
- ⋆ Ensuite on examine tous les éléments de la liste depuis $g+1$ jusqu'à d en veillant à ce que les valeurs strictement inférieures au pivot soient toujours placées avant les valeurs supérieures ou égales au pivot. L'indice j indique constamment la position de la dernière valeur strictement inférieure au pivot dans la partie *déjà traitée* de la liste (ou $j = g$ lorsqu'une telle valeur n'a pas encore été trouvée).
- ⋆ Après la boucle, il suffit d'échanger le pivot avec l'élément à la position j pour placer le pivot au bon endroit.

```
function division(var liste:TListBox; g,d:integer):integer;
var i,j:integer;
    pivot:string;
begin
  echange(liste,g,(g+d) div 2);
  pivot:=liste.Items[g];
  j:=g;
  for i:=g+1 to d do
    if liste.Items[i]<pivot then begin
      j:=j+1;
      if j<>i then echange(liste,j,i)
    end;
  if g<>j then echange(liste,g,j);
  division:=j
end;
```

Pour les listes à taille faible ($d-g<20$), le tri par insertion devient généralement plus rapide que le tri rapide.

4 Algorithme de recherche

Par convention, les algorithmes de recherche suivants renvoient tous l'indice -1, lorsque la clé cherchée n'a pas été trouvée.

4.1 Recherche séquentielle

```
function rechercheSeq(liste:TListBox; cle:string):integer;
var i:integer;
    trouve:boolean;
begin
  i:=0;
  trouve:=false;
  while (not trouve) and (i<liste.Items.Count) do
    if liste.Items[i]=cle then
      trouve:=true
    else
      i:=i+1;
  if trouve then
    rechercheSeq:=i
  else
    rechercheSeq:=-1
end;
```

4.2 Recherche dichotomique

4.2.1 Version itérative

```
function rechDichoI(liste:TListBox; cle:string):integer;
var milieu,g,d:integer;
begin
  g:=0;
  d:=liste.Items.Count-1;
  milieu:=(g+d) div 2;
  while (cle<>liste.Items[milieu]) and (g<=d) do begin
    if cle<liste.Items[milieu] then
      d:=milieu-1
    else
      g:=milieu+1;
    milieu:=(g+d) div 2
  end;
  if cle=liste.Items[milieu] then
    rechDichoI:=milieu
  else
    rechDichoI:=-1
end;
```

4.2.2 Version récursive

```
function rechDichoR(liste:TListBox; cle:string; g,d:integer):integer;
var milieu:integer;
begin
  if g>d then
    rechDichoR:=-1
  else begin
    milieu:=(g+d) div 2;
    if liste.Items[milieu]=cle then
      rechDichoR:=milieu
    else if cle<liste.Items[milieu] then
      rechDichoR:=rechDichoR(liste,cle,g,milieu-1)
    else
      rechDichoR:=rechDichoR(liste,cle,milieu+1,d)
  end
end;
```

Appel de la fonction, pour rechercher dans toute la liste :
... `rechDichoR(liste, cle, 0, liste.Items.Count-1)` ...

5 Graphiques

5.1 Affichage de fonctions simples

La procédure `dessFonction` réalise la représentation graphique $\mathcal{C}_f$ de la fonction f, supposée définie, continue et non constante sur l'intervalle d'affichage $[x_1, x_2]$, avec $x_1 < x_2$. L'échelle de l'axe vertical (axe des ordonnées) est ajustée par la procédure en fonction des valeurs maximale et minimale prises par la fonction sur l'intervalle considéré.

Exemple de fonction f :

```
1  function f(x:extended):extended;
2  begin
3    f:=sin(x)+cos(2*x)
4  end;
```

Procédure de dessin, qui accepte comme arguments la surface de dessin (Image) et les nombres x_1 et x_2. Le graphique de la fonction est dessiné en couleur rouge sur le canevas (dont le contenu antérieur n'est pas effacé).

```
1  procedure dessFonction(var im:TImage; x1,x2:extended);
2  var fx,dx,dy,ymin,ymax:extended;
3      i:integer;
4  begin
5    dx:=(x2-x1)/(im.Width-1);
6    ymin:=f(x1);
7    ymax:=ymin;
8    for i:=1 to im.Width-1 do begin
9      fx:=f(x1+i*dx);
10     if fx<ymin then ymin:=fx
11     else if fx>ymax then ymax:=fx
12   end;
13   dy:=(ymax-ymin)/(im.Height-1);
14   im.Canvas.Pen.Color:=clRed;
15   im.Canvas.MoveTo(0,round((ymax-f(x1))/dy));
16   for i:=1 to im.Width-1 do
17     im.Canvas.LineTo(i,round((ymax-f(x1+i*dx))/dy))
18 end;
```

L'algorithme précédent convertit des ordonnées réelles en indices de pixel (lignes 15 et 17). Plus généralement, on pourra utiliser la fonction suivante pour convertir des coordonnées réelles x, y en indices de pixel. On suppose que les intervalles d'affichage horizontal et vertical sont $[\texttt{xmin}, \texttt{xmax}]$ et $[\texttt{ymin}, \texttt{ymax}]$. Pour ne pas devoir passer ces quatre valeurs comme arguments à la fonction, on a recours à des variables globales.

```
type pixel = record
  x,y:integer
end;

function pointToPixel(x,y:extended;im:TImage):pixel;
begin
  result.x:=round((x-xmin)/(xmax-xmin)*(im.Width-1));
  result.y:=round((ymax-y)/(ymax-ymin)*(im.Height-1))
end;
```

5.2 Affichage de polynômes

Un polynôme à coefficients réels et de degré inférieur ou égal à 20 peut être représenté à l'aide du type

```
type poly = record
    c:array[0..20] of extended;
    d:integer
  end;
```

Alternativement, les coefficients peuvent être stockés sous forme de nombres convertis en *strings* dans une *ListBox* ou dans un *StringGrid* à une ligne. On modifie alors le degré du polynôme en redimensionnant le composant choisi.

La méthode de Horner permet d'évaluer efficacement un polynôme p donné en un nombre réel x.

Dans l'algorithme proposé, le polynôme est passé *par variable* pour des raisons d'efficacité : on évite ainsi de copier tout le contenu du polynôme dans la pile, mais on se limite à transmettre un pointeur vers l'endroit où le polynôme original est stocké. En passant le polynôme *par valeur*, on ne modifierait pas les effets produits par la fonction, mais la quantité de mémoire occupée lors de l'appel serait plus élevée.

```
function horner(var p:poly; x:extended):extended;
var i:integer;
    px:extended;
begin
  px:=p.c[p.d];
  for i:=p.d-1 downto 0 do
    px:=px*x+p.c[i];
  horner:=px
end;
```

Pour réaliser la représentation graphique d'un polynôme p donné, il suffit de remplacer dans l'algorithme 5.1 les appels du genre `f(x)` par des appels `horner(p,x)`.