Master Sciences, Technologies, Santé

Mention Mathématiques, spécialité Enseignement des mathématiques

Algorithmique et graphes, thèmes du second degré

Séance de travaux pratiques n° 1

Quelques éléments de correction…

Les corrections pour les algorithmes de base sont proposées en langage algorithmique, plus concis que le langage AlgoBox… Rien ne vous empêche naturellement de mettre en œuvre certains de ces algorithmes sous AlgoBox si vous ne l'avez déjà fait…

Algorithmique de base

Exercice 1. Décomposition d'un montant en euros

Écrire un algorithme permettant de décomposer un montant entré au clavier en billets de 20, 10, 5 euros et pièces de 2, 1 euros, de façon à minimiser le nombre de billets et de pièces.

Réponse. *Rappelons que l'opérateur* `div` *(*`floor ( .. / .. )` *en Algobox) permet d'obtenir le quotient et l'opérateur* `mod` *(% en Algobox) le reste de la division entière. L'idée consiste ici à déterminer dans un premier temps le nombre de billets de 20 euros nécessaires (qui correspond au quotient de la division du montant par 20) puis, pour la somme restante (à calculer…), le nombre de billets de 10 euros nécessaires et ainsi de suite.*

Il est recommandé de ne pas modifier la variable `montant` *(donnée de départ), d'où l'intérêt d'utiliser une variable de travail* `reste`*. Le but premier de cet exercice est de proposer aux élèves un algorithme ne nécessitant pas l'utilisation de structures de contrôle. L'algorithme est le suivant :*

```
Algorithme décompositionMontant
# Cet algorithme décompose un montant entré au clavier en billets
# de 20, 10, 5 euros et pièces de 2, 1 euros.
variables   montant, reste : entiers naturels
            billets20, billets10, billets5 : entiers naturels
            pièces2, pièces1 : entiers naturels
début
       # lecture donnée
   Entrer ( montant )
       # calculs
   billets20 ← montant div 20
   reste ← montant mod 20      # ou reste ← montant – (20 * billets20)
   billets10 ← reste div 10
   reste ← reste mod 10
   billets5 ← reste div 5
   reste ← reste mod 5
   pièces2 ← reste div 2
   reste ← reste mod 2
   pièces1 ← reste
       # affichage résultat
   Afficher ( billets20, billets10, billets5, pièces2, pièces1 )
fin
```

Remarquons que (par hasard ?) les montants 20, 10, 5, 2, 1 sont tels que chacun est la moitié (entière) du précédent… On aurait ainsi pu utiliser une boucle `pour` *:*

```
   reste ← montant
   billet ← 20
```

```
   pour i de 1 à 5
   faire
        nb ← reste div billet
        Afficher ( nb )
        billet ← billet div 2
   fin_pour
```

Remarquons également que l'on aurait pu s'arrêter dès que la valeur de `reste` *est 0. Pour la première version, on aurait alors une séquence de* `si-alors-sinon` *imbriqués ; pour la deuxième version, la boucle* `pour` *serait remplacée par une boucle* `tantque`*.*

De la même façon, il est possible, à l'aide d'une structure `si-alors` *de n'afficher les nombres de billets ou de pièces que lorsqu'ils sont non nuls…*

Exercice 2. Calcul de la n^{ième} valeur d'une suite

Écrire un algorithme permettant de calculer la n^{ième} valeur d'une suite de la forme $u_n = au_{n-1} + b$, $u_0 = c$ (a, b et c sont des entiers naturels entrés au clavier).

Réponse. *Il suffit d'utiliser une boucle* `pour` *calculant le i-ième terme en fonction du terme précédent. Il suffit pour cela d'une variable* `un`*, et il est inutile de stocker les valeurs intermédiaires* u_1*,* u_2*, etc.*

L'algorithme est le suivant :

```
Algorithme niemeNombreSuite
# cet algorithme permet de calculer la n-ième valeur d'une suite de la
# forme un = aun-1 + b, u0 = c
variables   a, b, c, n, un, i : entiers naturels
début
        # lecture des données
   Entrer ( a, b, c, n )
        # initialisation
   un ← c
        # boucle de calcul
   pour i de 1 à n
   faire    un ← a * un + b
   fin_pour
        # affichage résultat
   Afficher ( un )
fin
```

Exercice 3. Nombres parfaits

Un nombre est parfait s'il est égal à la somme de ses diviseurs stricts (différents de lui-même). Ainsi par exemple, l'entier 6 est parfait car 6 = 1 + 2 + 3. Écrire un algorithme permettant de déterminer si un entier naturel est un nombre parfait.

Réponse. *Il suffit de calculer la somme des diviseurs propres de l'entier* `n` *(il est donc nécessaire de déterminer les diviseurs de* `n` *compris entre* `1` *et* `n div 2`*…). Les premiers nombres parfaits sont : 6, 28, 496, 8 128, 33 550 336, 8 589 869 056 (le 7 octobre 2008, on ne connaissait que 46 nombres parfaits).*

L'algorithme est le suivant :

```
Algorithme nombreParfait
# cet algorithme permet de déterminer si un nombre est parfait
variables   n, diviseur, somme : entiers naturels
début
        # lecture des données
   Entrer ( n )
        # cas où n est nul
   si ( n = 0 )
   alors Afficher ( "le nombre 0 n'est pas parfait" )
```

```
      # cas général
   sinon
         # initialisation de la somme des diviseurs, 1 divise n
      somme ← 1

         # boucle de parcours
      pour diviseur de 2 à n div 2
      faire    si ( n mod diviseur = 0 )
               alors    somme ← somme + diviseur
               fin_si
      fin_pour

         # affichage du résultat
      si ( n = somme )
      alors    Afficher ( "le nombre ", n, " est parfait" )
      sinon    Afficher ( "le nombre ", n, " n'est pas parfait" )
               Afficher ( "(la somme vaut ", somme, ")" )
      fin_si

   fin_si

fin
```

Manipulation de listes

Exercice 4. La liste est-elle monotone ?

Écrire un algorithme permettant de déterminer si une liste est ou non triée par ordre croissant ou décroissant au sens large. On commencera naturellement par saisir une liste entrée au clavier (on demandera au préalable le nombre d'éléments de cette liste), mais sans vérifier cette propriété au fur et à mesure de la saisie…

Réponse. *Une fois la liste construite on doit dans un premier temps la parcourir pour rechercher les deux premiers éléments distincts (lignes 25 à 48). On utilise un booléen*[1] `trouve` *pour mémoriser le fait que deux tels éléments existent, et un booléen* `croissant` *pour mémoriser l'ordre de ces éléments. Si tous les éléments sont égaux (*`trouve` *vaut 0), la liste est constante. Sinon, on parcourt la fin de liste pour vérifier si la monotonicité est ou non préservée, en s'arrêtant éventuellement si une rupture de monotonie est détectée (lignes 56 à 72).*

L'algorithme est le suivant :

```
liste_monotone  -  09.03.2011
******************************************
Cet algorithme détermine si une liste lue est monotone ou pas
(croissante ou décroissante au sens large)
******************************************
1   VARIABLES
2     L EST_DU_TYPE LISTE
3     nbElements EST_DU_TYPE NOMBRE
4     i EST_DU_TYPE NOMBRE
5     trouve EST_DU_TYPE NOMBRE
6     croissant EST_DU_TYPE NOMBRE
7     monotone EST_DU_TYPE NOMBRE

8   DEBUT_ALGORITHME

9     //Lecture du nombre d'éléments de la liste
10    AFFICHER "Nombre d'éléments ? "
11    LIRE nbElements

12    SI (nbElements == 0) ALORS
13      DEBUT_SI
14      //Cas de la liste vide...
15      AFFICHER "La liste est vide"
16      FIN_SI

17      SINON
18        DEBUT_SINON

19        //Lecture de la liste
```

[1] Le type booléen n'existe pas en AlgoBox, on utilise donc un entier prenant les valeurs 0 (faux) ou 1 (vrai).

```
        AFFICHER "Entrez les éléments de la liste..."
        POUR i ALLANT_DE 0 A nbElements-1
          DEBUT_POUR
          LIRE L[i]
          FIN_POUR
        //On cherche les deux premiers éléments distincts
        trouve PREND_LA_VALEUR 0
        i PREND_LA_VALEUR 0
        TANT_QUE (trouve==0 ET i <= (nbElements-2)) FAIRE
          DEBUT_TANT_QUE
          SI (L[i]==L[i+1]) ALORS
            DEBUT_SI
            //égalité, on avance...
            i PREND_LA_VALEUR i+1
            FIN_SI
            SINON
              DEBUT_SINON
              // on a trouvé... croissant ou pas ?
              SI (L[i] < L[i+1]) ALORS
                DEBUT_SI
                croissant PREND_LA_VALEUR 1
                FIN_SI
                SINON
                  DEBUT_SINON
                  croissant PREND_LA_VALEUR 0
                  FIN_SINON
              trouve PREND_LA_VALEUR 1
              FIN_SINON
          FIN_TANT_QUE
        // si trouve vaut 0, la liste est constante...
        SI (trouve == 0) ALORS
          DEBUT_SI
          AFFICHER "La liste est constante"
          FIN_SI
          SINON
            DEBUT_SINON
                 //on parcourt la fin de liste pour vérifier que la monotonicité est préservée
            monotone PREND_LA_VALEUR 1
            //la position i a été testée, on avance d'un rang
            i PREND_LA_VALEUR i + 1
            TANT_QUE ((monotone == 1) ET (i < nbElements-1)) FAIRE
              DEBUT_TANT_QUE
              // si rupture de monotonie, on arrête...
               SI (((croissant==1) ET (L[i]>L[i+1])) OU ((croissant==0) ET (L[i]<L[i+1]))) ALORS
                DEBUT_SI
                monotone PREND_LA_VALEUR 0
                FIN_SI
                SINON
                  DEBUT_SINON
                  // sinon on avance...
                  i PREND_LA_VALEUR i + 1
                  FIN_SINON
              FIN_TANT_QUE
            //affichage du résultat
            SI (monotone == 0) ALORS
              DEBUT_SI
              AFFICHER "La liste n'est pas monotone"
              FIN_SI
              SINON
                DEBUT_SINON
                AFFICHER "La liste est monotone "
                SI (croissant == 1) ALORS
                  DEBUT_SI
                  AFFICHER "croissante"
                  FIN_SI
                  SINON
                    DEBUT_SINON
```

```
                    AFFICHER "décroissante"
                    FIN_SINON
                FIN_SINON
            FIN_SINON
        FIN_SINON

FIN_ALGORITHME
```

Remarquons ici la nature des conditions de continuation de nos boucles `tantque` *: celles-ci ne font jamais référence à un élément de la liste du type* `L[i]`*… En effet, selon les langages de programmation utilisés, un tel test peut conduire à une erreur d'exécution lorsque la variable* `i` *« sort » de l'intervalle de définition de la liste… Au niveau algorithmique, il est donc indispensable de procéder comme nous l'avons fait pour que notre algorithme soit correct quel que soit le langage de programmation utilisé par la suite…*

Exercice 5. Tri par insertion

Écrire un algorithme permettant de saisir une liste au clavier (on demandera au préalable le nombre d'éléments de cette liste), en la triant par insertion au fur et à mesure, et de l'afficher une fois la saisie terminée. Ainsi, chaque nouvel élément devra être inséré en bonne position dans la liste en cours de construction.

Réponse. *On lit le premier élément, puis les éléments suivants un à un. Pour chaque élément lu, on cherche sa position d'insertion (lignes 28 à 43), on décale les éléments à droite de cette position y compris cette position (lignes 44 à 49), et on insère l'élément dans la position ainsi libérée (ligne 51).*

L'algorithme est le suivant :

```
liste_tri_insertion  -  13.02.2012
******************************************
Cet algorithme lit une liste élément par élément et la trie au fur
et à mesure (méthode de tri par insertion)
******************************************
1   VARIABLES
2     L EST_DU_TYPE LISTE
3     nbElements EST_DU_TYPE NOMBRE
4     i EST_DU_TYPE NOMBRE
5     elem EST_DU_TYPE NOMBRE
6     j EST_DU_TYPE NOMBRE
7     trouve EST_DU_TYPE NOMBRE
8   DEBUT_ALGORITHME
9     //Lecture du nombre d'éléments de la liste
10    AFFICHER "Nombre d'éléments ? "
11    LIRE nbElements
12    SI (nbElements <= 0) ALORS
13      DEBUT_SI
14      // Anomalie...
15      AFFICHER "Saisie incorrecte - fin de l'algorithme"
16      FIN_SI
17      SINON
18        DEBUT_SINON
19        // Lecture du premier élément
20        AFFICHER "Entrez les éléments de la liste..."
21        LIRE L[1]
22        // Lecture des éléments suivants
23        // i représente le nombre d'éléments déjà dans la liste
24        POUR i ALLANT_DE 2 A nbElements
25          DEBUT_POUR
26          LIRE elem
27          // on décale les éléments supérieurs à elem
28          // en partant de la droite...
29          trouve PREND_LA_VALEUR 0
30          j PREND_LA_VALEUR i-1
31          TANT_QUE (trouve==0 ET j>=1) FAIRE
32            DEBUT_TANT_QUE
33            SI (L[j]>elem) ALORS
34              DEBUT_SI
35              L[j+1] PREND_LA_VALEUR L[j]
36              j PREND_LA_VALEUR j-1
37              FIN_SI
38              SINON
```

```
                DEBUT_SINON
                trouve PREND_LA_VALEUR 1
                FIN_SINON
            FIN_TANT_QUE
          // on range elem à sa place
          L[j+1] PREND_LA_VALEUR elem
          FIN_POUR
        // affichage de la liste triée
        AFFICHER "Liste triée :"
        POUR i ALLANT_DE 1 A nbElements
          DEBUT_POUR
          AFFICHER L[i]
          AFFICHER " "
          FIN_POUR
        FIN_SINON
  FIN_ALGORITHME
```

Primitives graphiques

Exercice 6. Dessin de fonction

Écrire un algorithme permettant de dessiner la courbe de la fonction $f(x) = x^2/10$, définie sur l'intervalle [-10,10]. (L'utilisateur choisira une valeur de pas et la fonction sera dessinée point par point.)

Réponse. *La fonction (appelée* `F1`*) est définie dans l'onglet* `Utiliser une fonction numérique` *(ce qui permet de changer facilement de fonction…) et une boucle* `tantque` *permet de dessiner simplement la courbe point par point…*

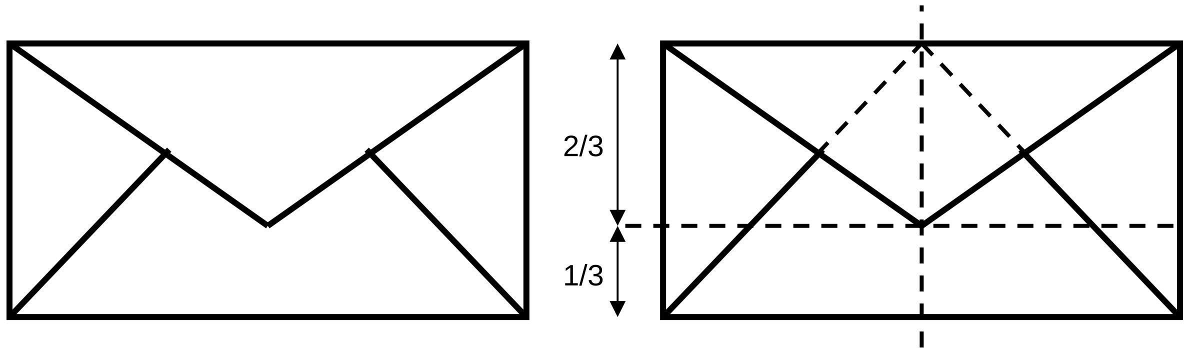
L'algorithme est le suivant :

```
dessin_fonction  -  12.03.2011
******************************************
Cet algorithme permet de dessiner la courbe d'une fonction
sur l'intervalle [-10, 10], le pas de dessin étant choisi par
l'utilisateur.
******************************************
  VARIABLES
    pas EST_DU_TYPE NOMBRE
    i EST_DU_TYPE NOMBRE
    x EST_DU_TYPE NOMBRE
  DEBUT_ALGORITHME
    // initialisations
    AFFICHER "Valeur du pas ?"
    LIRE pas
    x PREND_LA_VALEUR -10
    // boucle de dessin
    TANT_QUE (x <= 10) FAIRE
      DEBUT_TANT_QUE
      TRACER_POINT (x,F1(x))
      x PREND_LA_VALEUR x + pas
      FIN_TANT_QUE
  FIN_ALGORITHME
```

```
Fonction numérique utilisée :
F1(x)=(x*x)/10
```

Exercice 7. Dessin d'une enveloppe

Écrire un algorithme permettant de dessiner une enveloppe selon le profil suivant (la hauteur et la largeur seront données par l'utilisateur) :

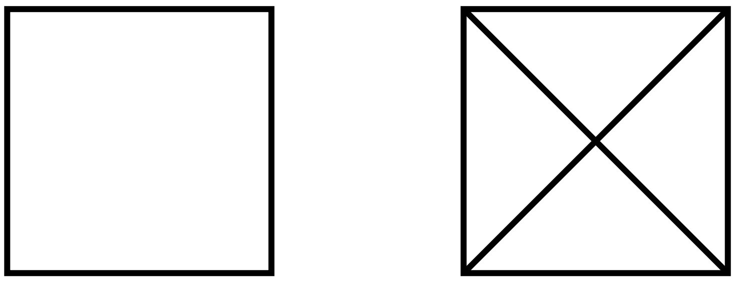


Réponse. *Un petit exercice de géométrie (calcul des coordonnées des extrémités de segments), laissé au lecteur…*

Exercice 8. Dessin de polygones

Écrire un algorithme permettant de dessiner un polygone régulier à N côtés, de rayon donné, en dessinant, ou pas, les « rayons », selon le modèle suivant :

Polygone à quatre côtés



sans les rayons avec les rayons

Réponse. *En fonction du nombre* `n` *de côtés, on calcule l'*`angle` *entre deux rayons successifs (ligne 20). Un peu de trigonométrie permet alors de déterminer les coordonnées des « coins » successifs du polygone (ligne 29 et 30)…*

L'algorithme est le suivant :

```
polygone  -  12.03.2011
********************************************
Cet algorithme permet de dessiner un polygone régulier
à n côtés, avec ou sans "rayons" (rayon de longueur 8)
********************************************
1   VARIABLES
2     N EST_DU_TYPE NOMBRE
3     angle EST_DU_TYPE NOMBRE
4     angleParcours EST_DU_TYPE NOMBRE
5     I EST_DU_TYPE NOMBRE
6     pointX EST_DU_TYPE NOMBRE
7     pointY EST_DU_TYPE NOMBRE
8     suivantX EST_DU_TYPE NOMBRE
9     rayon EST_DU_TYPE NOMBRE
10    suivantY EST_DU_TYPE NOMBRE
11    dessineRayons EST_DU_TYPE CHAINE

12  DEBUT_ALGORITHME
13    // lecture des données
14    AFFICHER "Nombre de côtés du polygone ?"
15    LIRE N
16    AFFICHER "On dessine les rayons (O/N) ?"
17    LIRE dessineRayons

18    // Initialisations
19    rayon PREND_LA_VALEUR 8
20    angle PREND_LA_VALEUR (2.0 * Math.PI)/N
```

```
  angleParcours PREND_LA_VALEUR angle/2.0
  suivantX PREND_LA_VALEUR rayon * cos(angleParcours)
  suivantY PREND_LA_VALEUR - (rayon * sin(angleParcours))
  // Dessin du polygone
  POUR I ALLANT_DE 1 A N
    DEBUT_POUR
    pointX PREND_LA_VALEUR suivantX
    pointY PREND_LA_VALEUR suivantY
    suivantX PREND_LA_VALEUR rayon * cos(angleParcours)
    suivantY PREND_LA_VALEUR rayon * sin(angleParcours)
    TRACER_SEGMENT (pointX,pointY)->(suivantX,suivantY)
    SI (dessineRayons=="O") ALORS
      DEBUT_SI
      TRACER_SEGMENT (0,0)->(pointX,pointY)
      FIN_SI
    angleParcours PREND_LA_VALEUR angleParcours + angle
    FIN_POUR
FIN_ALGORITHME
```

Pour ceux qui progresseraient (trop ?) rapidement…

Exercice 9. Fusion de deux listes triées

Écrire un algorithme permettant, à partir de deux listes triées, de construire « l'union » triée de ces deux listes. À partir des listes [3, 6, 9] et [1, 6, 8, 12, 15], on obtiendra la liste [1, 3, 6, 6, 8, 9, 12, 15]. On supposera que l'utilisateur entre correctement les deux listes triées (ou bien réutiliser l'algorithme de tri par insertion réalisé précédemment pour créer les deux listes)…

Réponse. *Cet algorithme procède en deux phases. Lors de la première phase, on progresse en parallèle dans* L1 *et* L2 *en recopiant dans* L3 *le plus petit des deux éléments (lignes 30 à 49). Cette phase se termine dès que l'une des deux listes est épuisée. La deuxième phase (lignes 50 à 68) consiste à recopier dans* L3 *celle des deux listes qui n'a pas été épuisée lors de la phase 1.*

L'algorithme est le suivant :

```
fusion_listes  -  12.03.2011
******************************************
Cette algorithme réalise la fusion de deux liste L1 et L2 triées
dans une nouvelle liste L3
******************************************
1   VARIABLES
2     L1 EST_DU_TYPE LISTE
3     L2 EST_DU_TYPE LISTE
4     L3 EST_DU_TYPE LISTE
5     nbElements1 EST_DU_TYPE NOMBRE
6     nbElements2 EST_DU_TYPE NOMBRE
7     i EST_DU_TYPE NOMBRE
8     i1 EST_DU_TYPE NOMBRE
9     i2 EST_DU_TYPE NOMBRE
10    i3 EST_DU_TYPE NOMBRE
11    nbElements3 EST_DU_TYPE NOMBRE
12  DEBUT_ALGORITHME
13    // lecture des deux listes
14    AFFICHER "Nombre d'éléments de la liste 1 :"
15    LIRE nbElements1
16    POUR i ALLANT_DE 0 A nbElements1 - 1
17      DEBUT_POUR
18      LIRE L1[i]
19      FIN_POUR
20    AFFICHER "Nombre d'éléments de la liste 2 :"
21    LIRE nbElements2
22    POUR i ALLANT_DE 0 A nbElements2 - 1
23      DEBUT_POUR
24      LIRE L2[i]
25      FIN_POUR
26    // initialisations
```

```
    i1 PREND_LA_VALEUR 0
    i2 PREND_LA_VALEUR 0
    i3 PREND_LA_VALEUR 0

    // boucle de fusion
    TANT_QUE ((i1 < nbElements1) ET (i2 < nbElements2)) FAIRE
      DEBUT_TANT_QUE

      SI (L1[i1] < L2[i2]) ALORS
        DEBUT_SI
        // on insère L1[i1] dans L3
        L3[i3] PREND_LA_VALEUR L1[i1]
        i3 PREND_LA_VALEUR i3 + 1
        // on avance dans L1
        i1 PREND_LA_VALEUR i1 + 1
        FIN_SI

        SINON
          DEBUT_SINON
          // on insère L2[i2] dans L3
          L3[i3] PREND_LA_VALEUR L2[i2]
          i3 PREND_LA_VALEUR i3 + 1
          // on avance dans L2
          i2 PREND_LA_VALEUR i2 + 1
          FIN_SINON
      FIN_TANT_QUE

    // on recopie dans L3 la liste non terminée
    SI (i1 < nbElements1) ALORS
      DEBUT_SI
      // on recopie la fin de L1
      POUR i ALLANT_DE i1 A nbElements1 - 1
        DEBUT_POUR
        L3[i3] PREND_LA_VALEUR L1[i]
        i3 PREND_LA_VALEUR i3 + 1
        FIN_POUR
      FIN_SI

      SINON
        DEBUT_SINON
        // on recopie la fin de L2
        POUR i ALLANT_DE i2 A nbElements2 - 1
          DEBUT_POUR
          L3[i3] PREND_LA_VALEUR L2[i]
          i3 PREND_LA_VALEUR i3 + 1
          FIN_POUR
        FIN_SINON

    // mise à jour du nombre d'éléments de L3
    nbElements3 PREND_LA_VALEUR i3

    // affichage de la liste L3
    POUR i ALLANT_DE 0 A nbElements3 - 1
      DEBUT_POUR
      AFFICHER L3[i]
      AFFICHER " "
      FIN_POUR
  FIN_ALGORITHME
```

Exercice 10. Suppression des doublons

Écrire un algorithme permettant de supprimer les doublons (éléments déjà présents) dans une liste triée donnée. À partir de la liste [3, 3, 6, 9, 9, 9, 9, 11], on obtiendra la liste [3, 6, 9, 11].

Réponse. *On va utiliser deux indices,* `i` *et* `isansdoublons`*. L'indice* `i` *parcourt la liste initiale, l'indice* `isansdoublons` *correspond à la liste résultat. L'élément* `L[isansdoublons-1]` *correspond alors au dernier élément conservé. Si* `L[i]` *est un nouvel élément, on le recopie dans* `L[isansdoublons]`*, sinon on avance dans la liste.*

L'algorithme est le suivant :

```
suppression_doublons  -  12.03.2011
******************************************
Cet algorithme supprime les doublons dans une liste triée
******************************************
```

```
VARIABLES
  L EST_DU_TYPE LISTE
  nbElements EST_DU_TYPE NOMBRE
  i EST_DU_TYPE NOMBRE
  isansdoublons EST_DU_TYPE NOMBRE
  nbElementsSansDoublons EST_DU_TYPE NOMBRE

DEBUT_ALGORITHME
  // lecture de la liste
  AFFICHER "Nombre d'éléments ?"
  LIRE nbElements
  POUR i ALLANT_DE 0 A nbElements - 1
    DEBUT_POUR
    LIRE L[i]
    FIN_POUR

  // initialisation (le premier élément n'est pas un doublon)
  isansdoublons PREND_LA_VALEUR 1

  // parcours des éléments à partir du second
  POUR i ALLANT_DE 1 A nbElements - 1
    DEBUT_POUR
    // si L[i] n'est pas un doublon, on le garde
    SI (L[i] != L[isansdoublons - 1]) ALORS
      DEBUT_SI
      // on recopie L[i]
      L[isansdoublons] PREND_LA_VALEUR i
      isansdoublons PREND_LA_VALEUR isansdoublons + 1
      FIN_SI
    FIN_POUR

  // nombre d'éléments dans liste résultat
  nbElementsSansDoublons PREND_LA_VALEUR isansdoublons

  // affichage liste résultat
  POUR i ALLANT_DE 0 A nbElementsSansDoublons - 1
    DEBUT_POUR
    AFFICHER L[i]
    AFFICHER " "
    FIN_POUR
FIN_ALGORITHME
```