Master Sciences, Technologies, Santé
Mention Mathématiques, spécialité Enseignement des mathématiques

Algorithmique et graphes, thèmes du second degré

Feuille TD n°2 – Exercices d'algorithmique
… éléments de correction …

Exercice 1. Lecture et affichage d'une liste

Écrire un algorithme permettant de construire une liste d'entiers naturels strictement positifs à partir d'une suite entrée au clavier et terminée par 0, puis de l'afficher une fois construite.

Réponse. *On construit la liste à l'aide de la structure* `tantque` *(pour détecter la fin de saisie), et on affiche son contenu en la parcourant à l'aide d'une boucle* `pour`*.*

L'algorithme est le suivant :

```
Algorithme lectureEtAffichageListeEntiers
# cet algorithme permet de construire une liste d'entiers naturels
# strictement positifs à partir d'une suite entrée au clavier et
# terminée par 0, puis de l'afficher une fois construite
variables   liste : liste d'entiers naturels
            n, i : entiers naturels
début
        # initialisation
   liste ← [ ]
        # boucle de lecture des valeurs et de création de la liste
   Entrer ( n )
   tantque ( n ≠ 0 )
             # on rajoute l'entier n en fin de liste
        liste ← liste + [ n ]
             # on lit la valeur suivante
        Entrer ( n )
   fin_tantque
        # boucle de parcours pour affichage
   pour i de 0 à NombreEléments ( liste ) - 1
   faire    Afficher ( liste [ i ] )
   fin_pour
fin
```

Exercice 2. Retournement d'une liste

Écrire un algorithme permettant de retourner une liste (son premier élément deviendra dernier, son deuxième avant-dernier, etc.) et d'afficher la liste ainsi retournée.

Réponse. *On retourne la liste en échangeant successivement les premier et dernier éléments, les second et avant-dernier éléments, etc. Attention, il faut arrêter cette série d'échange à la moitié de la longueur de la liste, sinon on la retourne deux fois !...*

L'algorithme, sous forme d'action, est le suivant :

```
Action retournementListeEntiers ( ES liste : liste d'entiers )
# cette action permet de retourner une liste d'entiers
variables   i, nbEléments : entiers naturels
```

```
début
        # initialisation
   nbEléments ← NombreEléments ( liste )
        # retournement de la liste
   pour i de 0 à ( nbEléments div 2 ) - 1
   faire          # on échange les éléments n° i et nbEléments - 1 - i
            liste ← liste [ 0 : i ] + liste [ nbEléments - i - 1]
                      + liste [ i + 1 : nbEléments - i – 1 ] + liste [ i ]
                      + liste [ nbEléments – i : nbEléments ]
   fin_pour
fin
```

On notera l'utilisation d'un paramètre d'entrée-sortie (ES), à la fois en entrée et en sortie : il s'agit d'un paramètre dont la valeur va être modifiée par l'action.

Exercice 3. Nombre d'occurrences d'un élément

Écrire un algorithme permettant de compter le nombre d'occurrences (d'apparitions) d'un élément donné dans une liste.

Réponse. *Il suffit de parcourir la liste en comptant le nombre d'apparitions de l'entier recherché.*

L'algorithme, sous forme de fonction, est le suivant :

```
Fonction nombreOccurrencesDansListe
        ( liste : liste d'entiers, élément : entier ) : entier
# cette fonction permet de déterminer le nombre d'occurrences de
# élément dans liste
variables   liste : liste d'entiers naturels
            n, i, élément, nbOccurrences : entiers naturels
début
        # initialisations
   nbEléments ← NombreEléments ( liste )
   nbOccurrences ← 0
        # parcours de la liste et comptage
   pour i de 0 à nbEléments – 1
   faire    si ( élément = liste [ i ] )
            alors    nbOccurrences ← nbOccurrences + 1
            fin_si
   fin_pour
        # retour résultat
   Retourner ( nbOccurrences )
fin
```

Exercice 4. La liste est-elle triée ?

Écrire un algorithme permettant de déterminer si la liste obtenue est ou non triée par ordre croissant (au sens large).

Réponse. *Ondoit parcourir la liste en comparant chaque élément à son suivant dans la liste. Pour cela, le booléen* `estTriée`*, initialisé à* `vrai`*, basculera à* `faux` *lorsque deux éléments consécutifs de la liste ne sont pas rangés dans l'ordre. La structure* `tantque` *permet d'arrêter le parcours dès que le booléen passe à* `faux` *ou que l'on atteint l'avant-dernier élément de la liste.*

L'algorithme, sous forme de fonction, est le suivant :

```
Fonction listeTriée ( liste : liste d'entiers ) : booléen
# cet algorithme permet de déterminer la liste
# est ou non triée par ordre croissant au sens large
variables   n, i, nbEléments : entiers naturels
            estTriée : booléen
```

```
début
        # initialisations
   estTriée ← vrai
   i ← 0
   nbEléments ← NombreEléments ( liste )
        # parcours et test de la liste : est-elle triée ?
   tantque ( ( i ≤ nbEléments – 2 ) et estTriée )
   faire     si ( L [i] > L [i+1] )
             alors estTriée ← faux
             sinon i ← i + 1
             fin_si
   fin_tantque
        # retour du résultat
   Retourner ( estTriée )
fin
```

Exercice 5. 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 (une telle liste est dite monotone, croissante ou décroissante respectivement).

Réponse. *Une fois la liste construite, on doit la parcourir, dans un premier temps, pour déterminer si elle est croissante ou décroissante (on utilisera un booléen* `estCroissante` *pour mémoriser cela) en cherchant les deux premiers éléments consécutifs distincts (si tous les éléments sont égaux, la liste est monotone). Ensuite, on procède comme dans l'exercice précédent, en adaptant le test des éléments consécutifs, et en continuant le parcours à l'endroit où l'on a détecté la première différence….*

L'algorithme est le suivant :

```
Algorithme listeMonotone
# cet algorithme permet de déterminer si une liste lue au clavier
# est monotone (croissante ou décroissante au sens large)
variables   liste : liste d'entiers naturels
            n, i, nbEléments : entiers naturels
            trouvé, estCroissante, estMonotone : booléen
début
        # initialisation liste
   liste ← [ ]
        # boucle de lecture des valeurs et de création de la liste
   Entrer ( n )
   tantque ( n ≠ 0 )
             # on rajoute l'entier n en fin de liste
        liste ← liste + [ n ]
             # on lit la valeur suivante
        Entrer ( n )
   fin_tantque
        # croissante ou décroissante ?
             # initialisations
   trouvé ← faux
   estCroissante ← vrai
   i ← 0
   nbEléments ← NombreEléments ( liste )
             # parcours cherche deux éléments consécutifs distincts
   tantque ( ( i ≤ nbEléments – 2 ) et ( non trouvé ) )
   faire     si ( L [i] < L [i+1] )
             alors     trouvé ← vrai
             sinon     si ( L [i] > L [i+1] )
                       alors     estCroissante ← faux
                                 trouvé ← vrai
```

```
                    sinon i ← i + 1
                    fin_si
          fin_si
  fin_tantque

       # initialisations parcours suivant
  listeMonotone ← vrai

       # parcours et test de la liste : est-elle monotone ?
  tantque ( ( i ≤ nbEléments – 2 ) et estMonotone )
  faire    si ( ( estCroissante et ( L [i] > L [i+1] ) )
                ou ( ( non estCroissante ) et ( L [i] < L [i+1] ) ) )
           alors estMonotone ← faux
           sinon i ← i + 1
           fin_si
  fin_tantque

       # affichage résultat
  si ( estMonotone )
  alors    Afficher ( "La liste est monotone" )
           si ( estCroissante )
           alors    Afficher ( "croissante." )
           sinon    Afficher ( "décroissante." )
           fin_si
  sinon    Afficher ( "La liste n'est pas monotone" )
  fin_si
fin
```

Remarque. *Notons que si tous les éléments de la liste sont égaux (ou si celle-ci contient 0 ou 1 élément), le booléen* `estCroissante` *est à vrai après le premier parcours ; le second parcours n'est pas effectué, et le message* `"La liste est monotone croissante."` *est affiché.*

Exercice 6. Tri par insertion

Écrire un algorithme permettant de construire une liste *triée par ordre croissant* d'entiers naturels strictement positifs à partir d'une suite entrée au clavier et terminée par 0. Ainsi, chaque nouvel élément devra être inséré en bonne position dans la liste en cours de construction.

Réponse. *Chaque élément lu doit être inséré en bonne position dans la liste en cours de construction. Pour cela, il faut chercher le premier élément de la liste plus grand ou égal à l'élément à insérer.*

L'algorithme est le suivant :

```
Algorithme triParInsertion

# cet algorithme permet de construire une liste triée par ordre croissant
# à partir d'une suite entrée au clavier et terminée par 0, en utilisant
# le principe du tri par insertion

variables   liste : liste d'entiers naturels
            n, i, nbEléments : entiers naturels
            trouvé : booléen

début

       # initialisation liste
  liste ← [ ]
  nbEléments ← 0

       # boucle de lecture des valeurs et d'insertion dans la liste
  Entrer ( n )
  tantque ( n ≠ 0 )

            # on cherche la position du nouvel élément dans la liste
                 # initialisations
       i ← 0
       trouvé ← faux
       nbEléments ← NombreEléments ( liste )

                 # on cherche la position d'insertion
       tantque ( (non trouvé) et (i < nbEléments ) )
       faire     si ( L [i] ≥ n )
                 alors trouvé ← vrai
```

```
                sinon i ← i + 1
                # on insère n en position i, en distinguant 3 cas
        selon que
                    # insertion en tête de liste
        i = 0 :  liste ← [ n ] + liste
                    # insertion en fin de liste
        i = nbEléments  :  liste ← liste + [ n ]
                    # insertion en milieu de liste
        sinon :
             liste ← liste [ 0 : i-1 ] + [ n ] + liste [ i : nbEléments-1
]
        fin_selon
             # on lit la valeur suivante
        Entrer ( n )
   fin_tantque
        # on affiche la liste triée
   pour i de 0 à nbEléments - 1
   faire    Afficher ( liste [ i ] )
   fin_pour
fin
```

Remarques. *Que se passe-t-il lorsque la liste entrée par l'utilisateur est vide ? La variable nbEléments vaut dans ce cas 0 (d'où l'intérêt de son initialisation !), et le corps de la boucle d'affichage n'est pas exécuté car 0 > -1.*

Notons également que lorsque l'élément à insérer est plus grand que tous les éléments déjà contenus dans la liste, la position d'insertion `i` *vaut* `nbEléments` *à la sortie de la boucle de recherche.*

Exercice 7. 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…

Réponse. *L'idée est la suivante : on parcourt en parallèle les éléments des deux listes d'entrée et, à chaque tour, on rajoute en fin de la liste résultat le plus petit des deux éléments courants, puis on progresse dans la liste dont l'élément a été retenu. À la fin de cette séquence de comparaisons (ou au début si l'une des deux listes est vide), il reste des éléments non traités dans l'une des deux listes. Il suffit alors de recopier cette fin de liste dans la liste résultat…*

L'algorithme est le suivant :

```
Algorithme fusionListes
# cet algorithme permet de fusionner deux listes triées
variables   liste1, liste2, listeFusion : listes d'entiers naturels
            n, i1, i2, nbEléments1, nbEléments2, nbMin : entiers naturels
début
        # initialisation liste1
   liste1 ← [ ]
        # boucle de lecture liste1
   Entrer ( n )
   tantque ( n ≠ 0 )
             # on rajoute l'entier n en fin de liste
        liste1 ← liste1 + [ n ]
             # on lit la valeur suivante
        Entrer ( n )
   fin_tantque
        # initialisation liste2
   liste2 ← [ ]
        # boucle de lecture liste2
   Entrer ( n )
```

```
   tantque ( n ≠ 0 )
        liste2 ← liste2 + [ n ]
        Entrer ( n )
   fin_tantque

        # initialisations pour fusion
   nbEléments1 ← nombreEléments ( liste1 )
   nbEléments2 ← nombreEléments ( liste2 )
   listeFusion ← [ ]
   i1 ← 0
   i2 ← 0
   si ( nbEléments1 < nbEléments2 )
   alors     nbMin ← nbEléments1
   sinon     nbMin ← nbEléments2
   fin_si

        # boucle principale de fusion
   tantque ( i1 < nbEléments1 ) et ( i2 < nbEléments2 )
   faire
        si ( liste1 [ i1 ] < liste2 [ i2 ] )
        alors     listeFusion ← listeFusion + [ liste1 [ i1 ] ]
                  i1 ← i1 + 1
        sinon     listeFusion ← listeFusion + [ liste2 [ i2 ] ]
                  i2 ← i2 + 1
        fin_si
   fin_tantque

        # on recopie la fin de la liste non épuisée
   si ( i1 < nbEléments1)
   alors          # on recopie la fin de liste1
        listeFusion ← listeFusion + liste1 [ i1, nbEléments1 – 1 ]
   sinon          # on recopie la fin de liste2
        listeFusion ← listeFusion + liste2 [ i2, nbEléments2 – 1 ]

        # on affiche la liste fusionnée
   pour i de 0 à nbEléments1 + nbEléments2 - 1
   faire     Afficher ( listeFusion [ i ] )
   fin_pour
fin
```

Exercice 8. 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. *L'idée est en fait très simple : on doit conserver le premier élément, ainsi que tous les éléments différents de leur prédécesseur…*

On peut proposer deux versions de cet algorithme. Dans la première, on construit une liste résultat distincte de la liste d'entrée. Dans la deuxième, on modifie la liste elle-même.

Première version :

```
Algorithme suppressionDoublonsDansListe1

# cet algorithme permet de construire, à partir d'une liste triée,
# une liste sans doublons

variables   liste1, liste2 : listes d'entiers naturels
            i, précédent : entiers naturels

début

        # initialisation liste1
   liste1 ← [ ]

        # boucle de lecture liste1
   Entrer ( n )
   tantque ( n ≠ 0 )

             # on rajoute l'entier n en fin de liste
        liste1 ← liste1 + [ n ]
```

```
            # on lit la valeur suivante
        Entrer ( n )
    fin_tantque

        # initialisation liste2
    liste2 ← [ ]
    précédent ← 0      # car les éléments de liste1 sont non nuls…
                       # on recopiera donc bien le premier élément

        # parcours de liste1 en ne recopiant que les bons éléments
    pour i de 1 à nombreEléments ( liste1 ) – 1
    faire
        si ( liste1 [ i1 ] <> précédent )
        alors     # on recopie cet élément, et on met à jour précédent
                  liste2 ← liste2 + [ liste1 [ i1 ] ]
                  précédent ← liste1 [ i1 ]
        fin_si
    fin_pour

        # on affiche la liste résultat
    pour i de 0 à nombreEléments ( liste2 )
    faire    Afficher ( liste2 [ i ] )
    fin_pour
fin
```

Pour la deuxième version, il suffit de parcourir la liste lue et de supprimer tous les éléments égaux à leur prédécesseur (on démarrera donc au 2ème élément de la liste…) :

```
Algorithme suppressionDoublonsDansListe2

# cet algorithme permet de supprimer les doublons dans une liste triée

variables   liste : listes d'entiers naturels
            i, précédent : entiers naturels

début

        # initialisation liste
    liste ← [ ]

        # boucle de lecture
    Entrer ( n )
    tantque ( n ≠ 0 )
        liste ← liste + [ n ]

            # on lit la valeur suivante
        Entrer ( n )
    fin_tantque

        # initialisation pour suppression des doublons
    i ← 0
    précédent ← 0      # car les éléments de liste1 sont non nuls…
                       # on recopiera donc bien le premier élément

        # boucle de suppression
    tantque ( i < nbEléments )
    faire    si ( liste [ i ] = précédent )

             alors     # on supprime l'élément, sans faire progresser i !
                  liste ← liste [ 0 : i-1 ]
                            + liste [ i+1, nombreEléments ( liste ) - 1 ]
             sinon     # on conserve l'élément, i progresse
                  i ← i + 1
             fin_si
    fin_tantque

        # on affiche la liste résultat
    pour i de 0 à nombreEléments ( liste )
    faire    Afficher ( liste [ i ] )
    fin_pour
fin
```

Remarque. *Notons que la boucle de suppression des doublons utilise la structure* `tantque` *et non la structure* `pour` *! En effet, la taille de la liste diminuant en cours de route, on changerait alors les*

valeurs de l'intervalle parcouru, ce qui est interdit… Par ailleurs, l'appel à la fonction `nombreEléments` *permet bien de prendre en compte l'évolution de la taille de la liste.*

En fait, on peut également proposer une troisième version, encore plus simple, où la détection des doublons se fait lors de la lecture des valeurs, au moment de créer la liste. On obtient alors :

```
Algorithme suppressionDoublonsDansListe3

# cet algorithme permet de construire une liste triée en évitant de
# conserver les doublons

variables   liste : listes d'entiers naturels
            i, précédent : entiers naturels

début

      # initialisation liste
  liste ← [ ]
  précédent ← 0     # car les éléments de liste1 sont non nuls…
                    # on recopiera donc bien le premier élément

      # boucle de lecture liste avec sélection des éléments
  Entrer ( n )
  tantque ( n ≠ 0 )

           # on rajoute l'entier n en fin de liste uniquement
           # s'il diffère de précédent, et on met à jour précédent
      si ( n <> précédent )
      alors    liste ← liste + [ n ]
               précédent ← n
      fin_si

           # on lit la valeur suivante
      Entrer ( n )
  fin_tantque

      # on affiche la liste résultat
  pour i de 0 à nombreEléments ( liste )
  faire    Afficher ( liste [ i ] )
  fin_pour

fin
```