Recueil des algorithmes fondamentaux

Christian Percebois

Pôle Algorithmique et Programmation
IUT Paul Sabatier, Dpt Informatique

Octobre 2009

TABLE DES MATIERES

Algorithmes fondamentaux

Savoir programmer exige du développeur créativité et méthodologie. Pour cela, il convient de s'appuyer sur des algorithmes maîtrisés et éprouvés depuis des décennies, appelés *algorithmes fondamentaux*, comme la lecture et le rangement de données, le parcours séquentiel, la recherche d'un élément dans une table, le tri d'un tableau, l'insertion d'un élément… Il s'agit donc pour l'essentiel d'acquérir quelques principes de construction, indépendamment des contraintes d'un langage cible.

1. Faire ses gammes

Comme un pianiste, le développeur doit faire ses gammes s'il envisage de faire carrière. Elles lui permettront de s'approprier un savoir-faire méthodologique et d'aborder des partitions de plus en plus complexes. Les *conventions d'écriture des algorithmes* devront être scrupuleusement respectées durant cet apprentissage car elles contribuent à l'acquisition de réflexes partagés par la communauté. Alors seulement, le développeur débutant sera à même de résoudre des variations algorithmiques, respectant les constructions basiques.

Ces gammes seront complétées par la lecture régulière de programmes écrits par des développeurs expérimentés, de façon à partager leur langue commune. Une grande partie de cette culture peut s'acquérir par l'apprentissage *d'idiomes de programmation,* définis comme des constructions réellement utilisées par les développeurs.

2. Les algorithmes fondamentaux

Les algorithmes fondamentaux de ce recueil sont décrits de façon générique. Pour cela on suppose spécifiées les entités ci-dessous :

```
-- lit une donnée d
procédure lire (sortie d <T>) ;

-- écrit un résultat r
procédure écrire (entrée r <T>) ;

-- traite e
procédure traiter (entrée e <T>) ;

type TabElts[T] : tableau [1 à N] de <T> ;

type MatElts[T] : tableau [1 à N, 1 à M] de <T> ;
```

Dans la pratique, il conviendra donc d'instancier ces paramètres génériques (sous-programme, type, …) par rapport à la spécificité du problème à résoudre.

2.1. La lecture de *n* valeurs

2.1.1. Algorithme

```
-- lire et traiter séquentiellement une suite de n valeurs
lire le nombre de valeurs à lire et à traiter ;
tantque il reste une valeur à lire et à traiter faire
        lire une valeur ;
        traiter la valeur lue ;
fin tantque ;
```

2.1.2. Procédure

```
-- lit et traite séquentiellement une suite de n valeurs
-- format des données à lire :
--      n      valeur1      valeur2      ...      valeurn
procédure lireValeurs

glossaire
        n <Entier> ;          -- nombre de valeurs à lire
        valeur <T> ;          -- valeur courante lue
        cpt <Entier> ;        -- nombre de valeurs lues

début
        -- lire le nombre de valeurs à lire et à traiter
        lire (n) ;
        -- lire et traiter les valeurs
        cpt <- 0 ;
        tantque cpt < n faire
                -- lire une valeur
                lire (valeur) ;
                -- traiter la valeur lue
                traiter (valeur) ;
                cpt <- cpt + 1 ;
        fin tantque ;
fin
```

2.1.3. Application : le rangement des *n* valeurs lues dans un tableau

```
-- lit une suite de n valeurs et les range dans le tableau tab
-- format des données à lire :
--      n      valeur1      valeur2      ...      valeurn
-- nécessite 0 ≤ n ≤ N
-- entraîne ∀i ∈ [1..n], tab[i] = valeuri et nb = n
procédure lireTableauValeurs
        (sortie tab <TabElts[T]>, sortie nb <Entier>)

glossaire
        i <Entier> ;          -- indice de parcours de  tab

début
        -- lire le nombre de valeurs à lire
        lire (nb) ;
        -- lire et ranger dans le tableau tab les valeurs de la suite
        i <- 0 ;
        tantque i < nb faire
                -- lire et ranger la valeur lue  dans le tableau tab
                i <- i + 1 ;
                lire (tab[i]) ;
        fin tantque ;
fin
```

2.2. La lecture d'une suite de valeurs

2.2.1. Algorithme

```
-- lire et traiter séquentiellement une suite de valeurs
lire la valeur du marqueur de fin ;
lire la première valeur ;
tantque valeur lue /= marqueur de fin faire
        traiter la valeur lue ;
        lire la valeur suivante ;
fin tantque ;
```

2.2.2. Procédure

```
-- lit et traite séquentiellement une suite de valeurs
-- terminée par un marqueur
-- format des données à lire :
--      valeur0     valeur1     valeur2   ...   valeurn   valeur0
 procédure lireValeurs

glossaire
        marqueur <T> ;      -- marqueur de fin (valeur0)
        valeur <T> ;        -- valeur courante lue

début
        -- lire la valeur du marqueur de fin
        lire (marqueur) ;
        -- lire la première valeur
        lire (valeur) ;
        -- lire et traiter les valeurs
        tantque valeur /= marqueur faire
                -- traiter la valeur lue
                traiter (valeur) ;
                -- lire la valeur suivante
                lire (valeur) ;
        fin tantque ;
fin
```

2.2.3. Application : le rangement des valeurs lues dans un tableau

```
-- lit une suite de valeurs terminée par un marqueur
-- range les n valeurs lues dans le tableau tab
-- format des données à lire :
--      valeur0     valeur1     valeur2   ...   valeurn   valeur0
-- nécessite 0 ≤ n ≤ N
-- entraîne ∀i∈[1..n],  tab[i] = valeuri et nb = n
procédure lireTableauValeurs
        (sortie tab <TabElts[T]>, sortie nb <Entier>)

glossaire
        marqueur <T> ;      -- marqueur de fin (valeur0)
        i <Entier> ;        -- indice du tableau tab
        valeur <T> ;        -- valeur courante lue

début
        -- lire la valeur du marqueur de fin
        lire (marqueur) ;
        -- lire la première valeur
        lire (valeur) ;
        -- lire et ranger les valeurs dans le tableau tab
        i <- 1 ;
        tantque valeur /= marqueur faire
                -- ranger la valeur lue dans le tableau tab
                tab[i] <- valeur ;
                i <- i + 1 ;
                -- lire la valeur suivante
                lire (valeur) ;
        fin tantque ;
        nb <- i - 1 ;
fin
```

2.3. Le parcours séquentiel d'un tableau

2.3.1. Algorithme

```
-- traiter séquentiellement les éléments d'un tableau
se positionner sur le premier élément du tableau ;
tantque il reste un élément à examiner faire
        traiter l'élément courant du tableau ;
        passer à l'élément suivant du tableau ;
fin tantque ;
```

2.3.2. Procédure

```
-- traite séquentiellement les n éléments du tableau tab
-- nécessite 1 ≤ n ≤ N
procédure traiterEléments
        (entrée tab <TabElts[T]>, entrée n <Entier>)

glossaire
        i <Entier> ;  -- indice de parcours du tableau tab

début
        -- se positionner sur le premier élément du tableau tab
        i <- 1 ;
        -- traiter séquentiellement les éléments du tableau tab
        tantque  i <= n faire
                -- traiter l'élément courant du tableau tab
                traiter (tab[i]) ;
                -- passer à l'élément suivant du tableau tab
                i <- i + 1 ;
        fin tantque ;
fin
```

2.3.3. Variante : le traitement séquentiel d'une tranche d'un tableau

```
-- traite séquentiellement les éléments
-- de la tranche tab[iDébut..iFin] du tableau tab
-- nécessite 1 ≤ iDébut ≤ iFin ≤ N
procédure traiterTrancheEléments
        (entrée tab <TabElts[T]>,
        entrée iDébut <Entier>, entrée iFin <Entier>)

glossaire
        i <Entier> ;          -- indice de parcours
                              -- de la tranche tab[iDébut..iFin]
début
        -- se positionner sur le premier élément de la tranche
        i <- iDébut ;
        -- traiter séquentiellement les éléments de la tranche
        -- tab[iDébut..iFin]
        tantque  i <= iFin faire
                -- traiter l'élément courant de la tranche
                traiter (tab[i]) ;
                -- passer à l'élément suivant de la tranche
                i <- i + 1 ;
        fin tantque ;
fin
```

2.3.4. Application 1 : la somme des 10 premiers éléments d'un tableau

```
-- calcule la somme des 10 premiers éléments du tableau tab
-- nécessite N ≥ 10
-- entraîne résultat = ∑(i=1..10) tab[i]
fonction somme10Elts (entrée tab <TabElts[Entier]>)
retourne <Entier>

glossaire
        i <Entier> ;                -- indice du tableau tab
        somme <Entier> ;            -- somme des 10 éléments

début
        somme <- 0 ;
        -- se positionner sur le premier élément du tableau tab
        i <- 1 ;
        -- cumuler les 10 premiers éléments du tableau tab
        tantque  i <= 10 faire
                -- ajouter l'élément courant du tableau tab
                -- à la somme des éléments
                somme <- somme + tab[i] ;
                -- passer à l'élément suivant du tableau tab
                i <- i + 1 ;
        fin tantque ;
        retourner (somme) ;
fin
```

2.3.5. Application 2 : l'écriture des éléments d'une tranche de tableau

```
-- écrit les caractères de la tranche tab[iDébut..iFin]
-- nécessite 1 ≤ iDébut ≤ iFin ≤ N
procédure écrireTrancheEléments
        (entrée tab <TabElts[Caractère]>,
        entrée iDébut <Entier>, entrée iFin <Entier>)

glossaire
        i <Entier> ;  -- indice de parcours du tableau tab
                      -- de la tranche tab[iDébut..iFin]

début
        -- se positionner sur le premier caractère de la tranche
        i <- iDébut ;
        -- écrire séquentiellement les caractères de la tranche
        -- tab[iDébut..iFin]
        tantque i <= iFin faire
                -- écrire le caractère courant de la tranche
                écrire (tab[i]) ;
                -- passer au caractère suivant de la tranche
                i <- i + 1 ;
        fin tantque ;
fin
```

2.4. La recherche séquentielle

2.4.1. Algorithme

```
-- rechercher la première occurrence d'un élément
-- dans un tableau
se positionner sur le premier élément du tableau ;
tantque      il reste un élément à examiner
             et élément courant ≠ élément recherché
faire
        passer à l'élément suivant du tableau ;
fin tantque ;
```

2.4.2. Procédure

```
-- recherche la première occurrence de l'élément x
-- dans le tableau tab de n éléments
-- si l'élément existe, trouvé = VRAI et rang
-- désigne la place de l'élément dans le tableau tab,
-- sinon trouvé = FAUX et rang est indéfini
-- nécessite 1 ≤ n ≤ N
-- entraîne
--      (∃i∈[1..n], tab[i] = x et ∀j∈[1..i-1], tab[j] ≠ x
--              ⇒ trouvé = VRAI et rang = i)
--      et (∀i∈[1..n], tab[i] ≠ x ⇒ trouvé = FAUX)
procédure rechercherOccurrence
        (entrée tab <TabElts[T]>,
        entrée n <Entier>, entrée x <T>,
        sortie trouvé <Booléen>, sortie rang <Entier>)

glossaire
        i <Entier> ;  -- indice de parcours du tableau tab
début
        -- se positionner sur le premier élément du tableau tab
        i <- 1 ;
        -- parcourir et comparer les éléments du tableau tab
        tantque  i <= n et tab[i] /= x faire
                -- passer à l'élément suivant du tableau tab
                i <- i + 1 ;
        fin tantque ;
        -- fournir le résultat de la recherche
        si i <= n alors
                trouvé <- VRAI ;
                rang <- i ;
        sinon
                trouvé <- FAUX ;
        fin si ;
fin
```

2.4.3. Variante

```
-- recherche la première occurrence de l'élément x
-- dans le tableau tab de n éléments
-- si l'élément existe, trouvé = VRAI et rang
-- désigne la place de l'élément dans le tableau tab,
-- sinon trouvé = FAUX et rang est indéfini
-- nécessite 1 ≤ n ≤ N
-- entraîne
--      (∃i∈[1..n], tab[i] = x et ∀j∈[1..i-1], tab[j] ≠ x
--              ⇒ trouvé = VRAI et rang = i)
--      et (∀i∈[1..n], tab[i] ≠ x ⇒ trouvé = FAUX)
```

```
procédure rechercherOccurrence
        (entrée tab <TabElts[T]>,
        entrée n <Entier>, entrée x <T>,
        sortie trouvé <Booléen>, sortie rang <Entier>)

glossaire
        i <Entier> ;                    -- indice du tableau tab

début
        trouvé <- FAUX ;
        -- se positionner sur le premier élément du tableau tab
        i <- 1 ;
        -- parcourir et comparer les éléments du tableau tab
        tantque  i <= n et non trouvé faire
                si tab[i] = x alors
                        trouvé <- VRAI ;
                sinon
                        --  passer à l'élément suivant du tableau tab
                        i <- i + 1 ;
                fin si ;
        fin tantque ;
        -- fournir le résultat de la recherche
        si trouvé alors
                rang <- i ;
        fin si ;
fin
```

2.5. La recherche dichotomique

2.5.1. Algorithme

```
-- rechercher par dichotomie de l'élément x
-- dans un tableau tab
tantque     il reste une tranche du tableau tab à examiner
            et que x ≠ élément milieu de cette tranche
faire
        si élément x < élément milieu de la tranche alors
                choisir la moitié gauche de la tranche ;
        sinon
                choisir la moitié droite de la tranche ;
        fin si ;
fin tantque ;
```

2.5.2. Procédure itérative

```
-- recherche par dichotomie de l'élément x
-- dans le tableau tab de n éléments
-- si l'élément existe, trouvé = VRAI
-- et rang désigne sa position dans le tableau tab,
-- sinon trouvé = FAUX et rang n'est pas significatif
-- nécessite 1 ≤ n ≤ N et ∀i ∈ [1..n -1], tab[i] ≤ tab[i +1]
-- entraîne
        (∃i ∈ [1..n], tab[i] = x ⇒ trouvé = VRAI et rang = i)
--
        et (∀i ∈ [1..n], tab[i] ≠ x ⇒ trouvé = FAUX)
procédure rechercherParDichotomie
        (entrée tab <TabElts[T]>, entrée n <Entier>,
        entrée x <T>,
        sortie trouvé <Booléen>, sortie rang <Entier>)
```

```
glossaire
        iDébut <Entier> ;          -- indice de début d'une
                                   -- tranche du tableau tab
        iFin <Entier> ;            -- indice de fin d'une tranche
                                   -- du tableau tab
        iMilieu <Entier> ;         -- indice milieu de la tranche
                                   -- tab[iDébut..iFin]

début
        -- rechercher par dichotomie x dans le tableau tab
        iDébut <- 1 ;
        iFin <- n ;
        iMilieu <- (iDébut + iFin) div 2 ;
        tantque iDébut <= iFin et tab[iMilieu] /= x faire
                si x < tab[iMilieu] alors
                        -- choisir la tranche tab[iDébut..iMilieu - 1]
                        iFin <- iMilieu - 1 ;
                sinon
                        -- choisir la tranche tab[iMilieu + 1..iFin]
                        iDébut <- iMilieu + 1 ;
                fin si ;
                -- calculer l'indice milieu
                -- de la tranche tab[iDébut..iFin]
                iMilieu <- (iDébut + iFin) div 2 ;
        fin tantque ;
        -- fournir le résultat de la recherche
        si iDébut <= iFin alors
                trouvé <- VRAI ;
                rang <- iMilieu ;
        sinon
                trouvé <- FAUX ;
        fin si ;
fin
```

2.6. Le tri par maxima successifs

2.6.1. Algorithme

```
-- trier un tableau par la méthode des maxima successifs
tantque il reste un élément du tableau à classer faire
        ranger le plus grand élément à la fin du tableau ;
        réduire le tableau d'un élément ;
fin tantque ;
```

2.6.2. Procédure

```
-- échange le contenu de x et y
-- entraîne x'=y et y'=x
procédure échanger (màj x <T>, màj y <T>)

glossaire
        aux <T> ;      -- pour réaliser l'échange

début
        aux <- x ;
        x <- y ;
        y <- aux ;
fin
```

```
-- fournit l'indice du maximum de la tranche tab[1..n]
-- nécessite 1 ≤ n ≤ N
-- entraîne ∀i ∈ [1..n], tab[résultat] ≥ tab[i]
fonction indiceMax
        (entrée tab <TabElts[T]>, entrée n <Entier>)
retourne <Entier>
```

```
glossaire
        i <Entier> ;          -- indice du tableau tab
        iMax <Entier> ;       --  indice du maximum

début
        iMax <- 1 ;
        -- se positionner sur le deuxième élément du tableau tab
        i <- 2 ;
        -- parcourir et comparer les éléments du tableau tab
        tantque i <= n faire
                -- comparer l'élément courant du tableau
                -- tab et l'élément maximum
                si tab[i] > tab[iMax] alors
                        iMax <- i ;
                finsi ;
                -- passer à l'élément suivant du tableau tab
                i <- i + 1 ;
        fin tantque ;
        retourner (iMax) ;
fin


-- tri du tableau tab de n éléments par la méthode
-- des maxima successifs
-- nécessite 1 ≤ n ≤ N
-- entraîne ∀i ∈ [1..n -1], tab[i]'≤ tab[i +1]'
procédure trierParMaxSuccessifs
        (màj tab <TabElts[T]>, entrée n <Entier>)

glossaire
        iFin <Entier> ;           -- indice du dernier élément
                                  -- d'une tranche non triée
        iMax <Entier> ;           -- indice du maximum

début
        iFin <- n ;
        tantque iFin > 1 faire
                -- ranger le plus grand élément
                --  à la fin de la tranche tab[1..iFin]
                iMax <- indiceMax (tab, iFin)
                échanger (tab[iMax], tab[iFin]);
                -- réduire la tranche tab[1..iFin] d'un élément
                iFin <- iFin - 1 ;
        fin tantque ;
fin
```

2.7. Le tri à bulles

2.7.1. Algorithme

```
-- trier un tableau par la méthode des bulles
tantque il reste un élément du tableau à classer faire
        ramener (par permutations successives) le plus
        grand élément à fin du tableau ;
        réduire le tableau d'un élément ;
fin tantque ;
```

```
-- ramener (par permutations successives) le plus grand
-- élément à la fin d'un tableau
se positionner sur le premier élément du tableau ;
tantque     tous les éléments du tableau
            n'ont pas été examinés
faire
        -- comparer l'élément courant du tableau
        --  avec l'élément suivant
        si élément courant > élément suivant alors
                échanger l'élément courant et l'élément suivant ;
        fin si ;
        passer à l'élément suivant du tableau ;
fin tantque ;
```

2.7.2. Procédure

```
-- ramène la plus grande valeur de la tranche
-- tab[iDébut..iFin] à la position tab[iFin]
-- échange = VRAI si un échange est intervenu lors de la
-- remontée de la plus grande valeur,
-- échange = FAUX sinon
-- nécessite 1 ≤ iDébut ≤ iFin ≤ N
-- entraîne ∀i ∈ [iDébut..iFin -1], tab[iFin]'≥ tab[i]'
procédure ramenerValeurMax
        (màj tab <TabElts[T]>,
        entrée iDébut <Entier>, entrée iFin <Entier>,
        sortie échange <Booléen>)

glossaire
        i <Entier> ;  --  indice de parcours du tableau tab
début
        -- se positionner sur le premier élément du tableau tab
        i <- iDébut ;
        -- parcourir et comparer les éléments du tableau tab
        échange <- FAUX ;
        tantque i < iFin faire
                -- comparer l'élément courant à l'élément suivant
                si tab[i] > tab[i + 1] alors
                        échanger (tab[i], tab[i + 1]) ;
                        échange <- VRAI ;
                fin si ;
                -- passer à l'élément suivant du tableau tab
                i <- i + 1 ;
        fin tantque ;
fin


-- tri du tableau tab de n éléments par la méthode des bulles
-- nécessite 1 ≤ n ≤ N
-- entraîne ∀i ∈ [1..n -1], tab[i]'≤ tab[i +1]'
procédure trierParBulles
        (màj tab <TabElts[T]>, entrée n <Entier>)

glossaire
        nb <Entier> ;           -- tranche tab[1..nb] examinée
        terminé <Booléen> ;     -- la fin du tri
        échange <Booléen> ;     -- indicateur d'échange

début
        nb <- n ;
        terminé <- FAUX ;
```

```
    tantque non terminé faire
        -- ramener la plus grande valeur à fin
        -- du tableau dtab
        ramenerValeurMax (tab, 1, nb, échange) ;
        terminé <- non échange ;
        -- réduire le tableau tab d'un élément
        nb <- nb - 1 ;
    fin tantque ;
fin
```

2.8. Le tri par insertion

2.8.1. Algorithme

```
-- trier par insertion un tableau
se positionner sur le deuxième élément du tableau ;
tantque il reste un élément dans le tableau à classer faire
    insérer l'élément courant dans le tableau ;
    passer à l'élément suivant du tableau ;
fin tantque ;

-- insérer l'élément courant dans le tableau
rechercher (en décalant vers la droite) l'emplacement
    de la valeur dans le tableau ;
ranger la valeur dans le tableau à l'emplacement libre ;
```

2.8.2. Procédure

```
-- insère une valeur v dans le tableau tab
-- dont les éléments tab[1..k] sont triés par ordre croissant
-- nécessite 1≤ k < N et ∀i∈ [1..k -1], tab[i] ≤ tab[i +1]
-- entraîne ∀i∈ [1..k], tab'[i] ≤ tab'[i +1]
procédure insérerValeur
    (màj tab <TabElts[T]>,
    entrée k <Entier>, entrée v <T>)

glossaire
    i <Entier> ;  -- indice de parcours du tableau tab

début
    -- rechercher (en  décalant vers la droite) l'emplacement
    -- que doit occuper la valeur v dans la suite
    i <- k ;
    tantque i >= 1 et tab[i] > v faire
        tab[i + 1] <- tab[i] ;
        i <- i - 1 ;
    fin tantque ;
    -- ranger la valeur v dans la suite à l'emplacement libre
    tab[i + 1] <- v ;
fin

-- tri par insertion du tableau tab de n éléments
-- nécessite 1 ≤ n ≤ N
-- entraîne ∀i∈ [1..n -1], tab[i]' ≤ tab[i +1]'
procédure trierParInsertion
    (màj tab <TabElts[T]>, entrée n <Entier>)

glossaire
    i <Entier> ;        -- indice du tableau tab
    k <Entier> ;        -- indice de la tranche tab[1..i-1]
    v <T> ;             -- valeur courante à insérer dans tab

début
    -- se positionner sur le deuxième élément du tableau tab
    i <- 2 ;
```

```
    -- trier les éléments du tableau tab
    tantque i <= n faire
        -- insérer l'élément courant dans le tableau tab
        insérerValeur (tab, i - 1, tab[i]) ;
        -- passer à l'élément suivant du tableau tab
        i <- i + 1 ;
    fin tantque ;
fin
```

2.8.3. Application : la lecture et le tri des valeurs lues

```
-- lit et tri séquentiellement une suite de valeurs
-- terminée par un marqueur
-- format des données à lire :
--      valeur0   valeur1   valeur2   ...   valeurn   valeur0
-- entraîne ∀i ∈ [1..n-1], tab'[i] ≤ tab'[i+1]
procédure trierParInsertion
    (sortie tab <TabElts[T]>, sortie n <Entier>)

glossaire
    marqueur <T> ;      -- valeur du marqueur (valeur0)
    v <T> ;             -- valeur lue

début
    n <- 0 ;
    -- lire la valeur du marqueur de fin
    lire (marqueur) ;
    -- lire la première valeur v
    lire (v) ;
    -- insérer les valeurs dans le tableau tab
    tantque v /= marqueur faire
        -- insérer la valeur v dans la suite
        insérerValeur (tab, n, v) ;
        n <- n + 1 ;
        -- lire la valeur v suivante
        lire (v) ;
    fin tantque ;
fin
```

2.9. Le tri rapide (*quicksort*)

2.9.1. Algorithme

-- trier un tableau (*quicksort*)
partager le tableau à trier en deux tranches (telles que tous les éléments de la première tranche soient inférieurs à tous les éléments de la seconde) ;
recommencer le tri sur les deux tranches ainsi obtenues jusqu'à obtenir des tranches réduites à un élément ;

2.9.2. Procédures

```
-- partition de la tranche tab[i..j] en fonction du pivot tab[i]
-- nécessite 1 ≤ i ≤ j ≤ N
-- entraîne tab'[k] = tab[i] et ∀l, 1 ≤ l < k ≤ N, tab'[l] ≤ tab[i]
--      et ∀l, N ≥ l > k≥ 1, tab'[l] > tab[i]
procédure placer
    (màj tab <TabElts[T]>,
    entrée i <Entier>, entrée j <Entier>, sortie k <Entier>)

glossaire
    l <Entier> ;        -- indice de parcours du tableau tab

début
    l <- i + 1 ;
    k <- j ;
```

```
        tantque l <= k faire
                -- rechercher le premier élément tab[l] > tab[i]
                tantque l <= j et tab[l] <= tab[i] faire
                        l <- l + 1 ;
                fin tantque ;
                -- rechercher le premier élément tab[k] <= tab[i]
                tantque tab[k] > tab[i] faire
                        k <- k - 1 ;
                fin tantque ;
                -- échanger tab[l] et tab[k]
                si l < k alors
                        échanger (tab[l], tab[k]) ;
                        l <- l + 1 ;
                        k <- k - 1 ;
                fin si ;
        fin tantque ;
        -- échanger le pivot tab[i] et tab[k]
        échanger (tab[i], tab[k]) ;
fin
```

```
-- tri rapide (quicksort) de la tranche de tableau tab[i..j]
-- nécessite 1 ≤ i ≤ j ≤ N
-- entraîne ∀i ∈ [1..n-1], tab[i]' ≤ tab[i+1]'
procédure trierRapide
        (màj tab <TabElts[T]>,
        entrée i <Entier>, entrée j <Entier>)

glossaire
        k <Entier> ;            -- indice de placement de tab[i]
début
        si i < j alors
                -- placer l'élément pivot tab[i] au rang k
                placer (tab, i, j, k) ;
                -- tri rapide sur la tranche à gauche de k
                trierRapide (tab, i, k - 1) ;
                -- tri rapide sur la tranche à droite de k
                trierRapide (tab, k + 1, j) ;
        fin si ;
fin
```

2.10. Le parcours séquentiel d'une matrice ligne à ligne

2.10.1. Algorithme

```
-- traiter séquentiellement les éléments d'une matrice
se positionner sur la première ligne de la matrice ;
tantque il reste une ligne à examiner faire
        se positionner sur la première colonne de la ligne ;
        tantque il reste une colonne à examiner faire
                traiter l'élément courant (ligne, colonne)
                de la matrice ;
                passer à la colonne suivante de la ligne ;
        fin tantque ;
        passer à la ligne suivante de la matrice ;
fin tantque ;
```

2.10.2. Procédure

```
-- traite séquentiellement les n lignes de la matrice mat
-- une ligne est composée de m colonnes
-- nécessite 1 ≤ n ≤ N et 1 ≤ m ≤ M
procédure traiterEléments
        (entrée mat <MatElts[T]>,
        entrée n <Entier>, entrée m <Entier>)

glossaire
        i <Entier> ;  -- indice de parcours des lignes
        j <Entier> ;  -- indice de parcours des colonnes

début
        -- se positionner sur la première ligne de la matrice mat
        i <- 1 ;
        -- traiter séquentiellement les lignes de la matrice mat
        tantque  i <= n faire
                -- traiter séquentiellement les colonnes de la ligne
                j <- 1 ;
                tantque j <= m faire
                        traiter (mat[i, j]) ;
                        j <- j + 1 ;
                fin tantque ;
                -- passer à la ligne suivante de la matrice mat
                i <- i + 1 ;
        fin tantque ;
fin
```

2.10.3. Application : le produit de deux matrices carrées

```
-- calcule le produit matriciel des matrices mat1 et mat2 d'ordre n
-- nécessite 1 ≤ n ≤ N
-- entraîne mat3[i,j] = Σ(k=1..n) mat1[i,k]*mat2[k,j]
procédure calculerProduitMatriciel
        (entrée mat1 <MatElts[T]>, entrée mat2 <MatElts[T]>,
        entrée n <Entier>,
        sortie mat3 <MatElts[T]>)

glossaire
        i <Entier> ;  -- indice de parcours des lignes de mat1
        j <Entier> ;  -- indice de parcours des colonnes de mat1
        k <Entier> ; -- indice de parcours d'une ligne de mat1
                      -- et d'une colonne de mat2
début
        i <- 1 ;
        tantque  i <= n faire
                j <- 1 ;
                tantque j <= n faire
                        mat3[i,j] <- 0 ;
                        k <- 1 ;
                        tantque k <= n faire
                                mat3[i,j] <- mat3[i,j]
                                        + mat1[i,k] * mat2 [k,j] ;
                                k <- k + 1 ;
                        fin tantque ;
                        j <- j + 1 ;
                fin tantque ;
                i <- i + 1 ;
        fin tantque ;
fin
```