Licence 1 | Université de Nantes
X2I0010 Algorithmique et programmation | 2016/2017

Travaux dirigés et pratiques n° 3
Tableaux

Partie TD (3 séances)

Exercice 3.1 (Recherche d'un élément dans un tableau d'entiers)
Ecrire un sous-algorithme qui détermine si un entier est présent dans un tableau d'entiers.

1. Définir le type du tableau.
2. Écrire le sous-algorithme en faisant l'hypothèse que le tableau est entièrement rempli.
3. Écrire le sous-algorithme en faisant l'hypothèse que seules les nb premières cases du tableau sont remplies.

▽ **Correction**

Exercice pouvant être soumis pour l'évaluation ; demander un sous-algo

1. En cours, j'ai conseillé de préfixer tous les types enregistrements par T_tab_ et de fixer la taille du tableau avec une constante (notée en capitales).

```
type  T_tab_entiers = tableau de N entier
```

2. Le résultat du sous-algorithme est un booléen =>

```
fonction chercher (d T_tab_entiers table, d entier elt) : booleen
```

Cas 1 : tableau entièrement rempli
Analyse :
booléen $trouve$ initialisé à $faux$
entier i initialisé à 1 pour parcourir le tableau
Répétitive
Traitement
- comparer $table[i]$ à elt et mise à jour de $reussi$
- incrémenter i

Arrêt quand la fin du tableau est atteinte ($i > N$) ou quand elt est trouvé ($reussi = vrai$)
=> continuation (($i \leq N$) et $non(trouve)$)
Le traitement peut ne jamais être effectué => tant que

```
fonction chercher (d T_tab_entiers table, d entier elt) : booleen
lexique
  booleen trouve
  entier i
debut
  i ← 1
  trouve ← faux
  tant que ((i ≤ N) et non(trouve)) faire
    trouve ← table[i] = elt
    i ← i + 1
  fin tant que
  retourner (trouve)
fin
```

3.

```
fonction d d d entier booleen

fonction d d entiers d entier booleen
lexique
   booleen
   entier
debut

   tant que et non faire

   fin tant que
   retourner
fin
```

Exercice 3.2 (Recherche du nombre d'occurrences d'un élément dans un tableau de chaînes de caractères)

Ecrire un sous-algorithme qui détermine le nombre d'occurrences d'un auteur dans un tableau mémorisant des auteurs de roman : "Camus", "Zola", etc..

1. Définir le type du tableau.
2. Écrire le sous-algorithme en faisant l'hypothèse que le tableau est entièrement rempli.
3. Écrire le sous-algorithme en faisant l'hypothèse que seules les nb premières cases du tableau sont remplies.

▽ **Correction**

Exercice pouvant être soumis pour l'évaluation ; demander un sous-algo

```
type  T_tab_auteurs = tableau de N chaine
```

1. Le résultat du sous-algorithme est un entier =>

   ```
   fonction nb_occurrences(d T_tab_auteurs auteurs, d chaine nom) : entier
   ```

   Cas 1 : tableau entièrement rempli

   Analyse :

   entier occ initialisé à 0

entier i initialisé à 1 pour parcourir le tableau
Répétitive
Traitement
- comparer $auteurs[i]$ à nom et mise à jour de occ
- incrémenter i

Arrêt quand la fin du tableau est atteinte ($i > N$)
Le traitement doit être effectué N fois => répétitive pour

```
fonction nb_occurrences (d T_tab_auteurs auteurs, d chaine nom) : entier
lexique
  entier occ
  entier i
debut
  occ ← 0
  pour i de 1 a N faire
    si (auteurs[i] = nom) alors
      occ ← occ + 1
    fin si
  fin pour
  retourner(occ)
fin
```

2. Il faut ajouter un paramètre pour connaître le nombre de cases utilisées =>

```
fonction nb_occurrences(d T_tab_auteurs auteurs, d entiers nb, d chaine nom) : entier
```

Analyse :
entier occ initialisé à 0
entier i initialisé à 1 pour parcourir le tableau
Répétitive
Traitement
- comparer $auteurs[i]$ à nom et mise à jour de occ
- incrémenter i

Arrêt quand la partie non utilisée du tableau est atteinte ($i > nb$)
Le traitement doit être efffectué nb fois => pour

```
fonction nb_occurrences(d T_tab_auteurs auteurs, d entiers nb, d chaine nom) : entier
lexique
  entier occ
  entier i
debut
  occ ← 0
  pour i de 1 a nb faire
    si (auteurs[i] = nom) alors
      occ ← occ + 1
    fin si
  fin pour
  retourner(occ)
fin
```

Exercice 3.3 (Insertion)

Ecrire un sous-algorithme qui insère un élément elt dans la case pos d'un tableau d'entiers dont les nb premières cases sont occupées. Les cases occupées du tableau devront rester contigües et en début de tableau. Si la position d'insertion n'est pas valide ou si le tableau est plein, il n'y a pas d'insertion.

▽ **Correction**

Exercice pouvant être soumis pour l'évaluation ;

Analyse

Il faut déterminer les emplacements valides où peut être inséré elt : $1 \leq pos \leq nb + 1$.
Spécifier une fonction tableau_plein qui rend $vrai$ si toutes les cases du tableau sont occupées.

```
fonction tableau_plein(d T_tab_entiers table, d entier nb) : booleen
```

Analyse : Le sous-algorithme $insere$ modifie potentiellement le tableau et le nombre de cases occupées => deux paramètres en modification ; ne modifie pas elt => paramètre en donnée.

```
procedure inserer(m T_tab_entiers table, m entier nb, d entier elt)
```

$\boxed{(1 \leq pos) \text{ et } (pos \leq nb{+}1) \text{ et non (tableau plein)}}$

V

insertion de elt dans le tableau

insertion de elt dans le tableau :
- décaler les éléments de pos à nb d'une case vers la droite (copier en commençant par la fin). Répétitive dont le nombre de répétitions est connu => pour. Nécessité d'une variable i (entier) pour parcourir la portion de tableau concernée (de nb à pos).
- inserer elt
- incrémenter nb

```
procedure inserer(m T_tab_entiers table, m entier nb, d entier elt)
lexique
  entier i
debut
  si ((1 ≤ pos) et (pos ≤ nb+1) et non (tableau_plein(table, nb)) alors
    pour i de nb a pos par pas de −1 faire
       table[i+1] ← table[i]
    fin pour
    table[pos] ← elt
    nb  ← nb + 1
  fin si
fin

fonction tableau_plein(d T_tab_entiers table, d entier nb) : booleen
debut
     retourner(nb = N)
fin
```

Exercice 3.4 (Manipulations d'un tableau)

Il s'agit d'écrire différents sous-algorithmes s'appliquant à un tableau de taille N dont les nb premières cases sont occupées. Pour chacune des fonctionnalités suivantes, effectuer l'analyse du problème, définir la signature du sous-algorithme, ses éventuelles préconditions, puis écrire son algorithme :

1. Le sous-algorithme `tableau_vide` rend vrai si le tableau est vide.
2. Le sous-algorithme `affiche_tableau` affiche les éléments contenus dans le tableau.
3. Le sous-algorithme `ote_fin_tableau` ôte l'élément situé dans la dernière case occupée du tableau.
4. Le sous-algorithme `saisie_tableau` permet à l'utilisateur de saisir des valeurs qui seront stockées dans le tableau.

5. Le sous-algorithme `rechercher_tableau` recherche l'élément elt dans le tableau. Il rend son indice s'il est trouvé, un nombre négatif dans le cas contraire.
6. Le sous-algorithme `ote_tableau` ôte l'élément situé dans la case d'indice ind tableau. Si nécessaire, les autres éléments contenus dans le tableau sont décalés.
7. Le sous-algorithme `minimum_tableau` détermine la valeur minimum contenue dans le tableau.
8. Le sous-algorithme `moyenne_tableau` calcule la moyenne des élements contenus dans le tableau.

▽ **Correction**

Exercice pouvant être soumis pour l'évaluation ; demander un ou deux sous-algorithmes

Difficultés inégales

1. `tableau_vide`

```
// aucune precondition
fonction tableau_vide(d T_tab_entiers table, d entier nb) : booleen
debut
    retourner(nb = 0)
fin
```

2. `affiche_tableau`

algorithme donné en cours

```
// pre = { 0 ≤ nb ≤ N}
procedure affiche_tableau(d T_tab_entiers table, d entier nb)
lexique
   entier i
debut
   pour i de 1 a nb faire
      ecrire (table[i])
   fin pour
fin
```

3. `ote_fin_tableau`

```
// pre = {nb> 0} Il y a au moins un element
procedure ote_fin_tableau(m T_tab_entiers table, m entier nb)
debut
   nb ← nb − 1
fin
```

4. `saisie_tableau`

saisie_tableau : il faut décider de la manière dont la saisie s'arrête.

```
procedure saisie_tableau(m T_tab_entiers table, m entier nb)
lexique
   entier i
   char rep
debut
   i ← 1
   ecrire ("Voulez-vous saisir un entier (o/n) ?")
   lire(rep)
   tant que ((rep = 'o') et (nb ≤ N)) faire
      ecrire ("Valeur ?")
      lire(table[i])
      i ← i + 1
      ecrire ("Voulez-vous saisir un entier (o/n) ?")
      lire(rep)
   fin tant que
   nb ← i − 1
fin
```

5. rechercher_tableau

```
// pre = { 0 ≤ nb <= N}
fonction rechercher_tableau(d T_tab_entiers table, d entiers nb) : entier
lexique
  booleen trouve
  entier i
debut
  i ← 1
  trouve ← faux
  tant que ((i ≤ nb) et non(trouve)) faire
    trouve ← table[i] = elt
    i ← i + 1
  fin tant que
  i ← i − 1
  si (table[i] = elt) alors
    retourner(i)
  sinon
    retourner(−1)
  fin si
fin
```

6. ote_tableau Il faut au moins un elt

```
// pre = { 1 ≤ nb ≤ ind <= N}
procedure ote_tableau(m T_tab_entiers table, m entier nb, d entier ind)
lexique
   entier i
   char rep
debut
   pour i de ind a nb − 1 faire
      table[i] ← table[i+1]
   fin pour
   nb ← nb − 1
fin
```

7.

```
//{precondition : table n'est pas vide}
fonction minimum(d T_tab_entiers table, d entier nb) : entier
lexique
   entier i, min
debut
   min ← table[1]
   pour i de 2 a nb faire
      si (table[i] < min) alors
         min ← table[i]
      fin si
   fin pour
   retourner min
fin
```

8.

```
//{precondition : table n'est pas vide}
fonction moyenne(d T_tab_entiers table, d entier nb) : reel
lexique
   entier i, somme
debut
   somme ← 0
   pour i de 1 a nb faire
      somme ← somme + table[i]
   fin pour
   retourner somme / nb
fin
```

Exercice 3.5 (Nouveau type pour les tableaux)

Proposer un nouveau type pour les tableaux afon de mémoriser à la fois le contenu du tableau et le nombre de cases occupées. Modifier quelques algorithmes précédemment réalisés sur les tableaux (exercices 3.1 à 3.4) afin de les adapter à ce nouveau type.

▽ **Correction**

Exercice pouvant être soumis pour l'évaluation ; demander un ou deux sous-algorithmes

```
type
   T_tab_entiers = tableau de N entier

   T_tableau = enregistrement
                  T_tab_entiers tab
                  entier nb
              fin enregistrement

//----------------------------------------
fonction chercher (d  T_tableau table , d entier elt) : booleen
lexique
  booleen trouve
  entier i
debut
  i ← 1
  trouve ← faux
  tant que ((i ≤ table.nb) et non(trouve)) faire
    trouve ← table.tab[i] = elt
    i ← i + 1
  fin tant que
  retourner(trouve)
fin
//----------------------------------------
fonction nb_occurrences(d  T_tableau table , d entier elt) : entier
lexique
  entier occ
  entier i
debut
  occ ← 0
  pour i de 1 a  table.nb faire
    si (table.tab[i] = elt) alors
      occ ← occ + 1
    fin si
  fin pour
  retourner(occ)
fin
//----------------------------------------
procedure inserer(m  T_tableau table , d entier elt)
lexique
  entier i
debut
  si ((1 ≤ pos) et (pos ≤ nb) et non (tableau_plein(table)) alors
    pour i de table.nb − 1 a pos par pas de −1 faire
       table.tab[i+1] ← table.tab[i]
    fin pour
    table.tab[pos] ← elt
    table.nb  ← table.nb + 1
  fin si
fin
//----------------------------------------
fonction tableau_plein(d  T_tableau table) : booleen
debut
```

```
    retourner(table.nb = N)
fin
//----------------------------------------
//{precondition : table n'est pas vide}
fonction minimum(d T_tableau table) : entier
lexique
    entier i, min
debut
    min ← table.tab[1]
    pour i de 2 a table.nb − 1 faire
       si (table.tab[i] < min) alors
          min ← table.tab[i]
       fin si
    fin pour
    retourner min
fin
//----------------------------------------
//{precondition : table n'est pas vide}
fonction moyenne(d T_tableau table) : reel
lexique
    entier i, somme
debut
    somme ← 0
    pour i de 1 a table.nb faire
       somme ← somme + table.tab[i]
    fin pour
    retourner somme / table.nb
fin
//----------------------------------------
// etc.
```

Exercice 3.6 (Décalage circulaire)

Écrire une procédure qui prend en paramètre un entier *decal* positif et un tableau d'entiers, et réalise un décalage circulaire de *decal* cases à droite des valeurs mémorisées dans le tableau d'entiers. En donner aussi une version récursive fondée sur un décalage d'une seule case.
Exemples : décalage d'une case de [1, 2, 3, 4] → [4, 1, 2, 3] ; décalage de 2 cases de [7, 8, 9] → [8, 9, 7] ; décalage de 0 case de [4, 5] → [4, 5].

▽ Correction

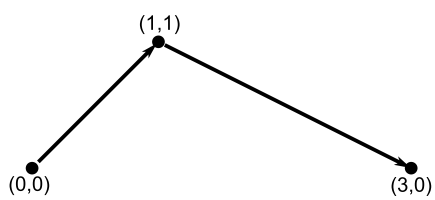
Exercice pouvant être soumis pour l'évaluation ;

```
type
    T_tab_int = tableau de N entier
//------------------------------------------------------------
procedure decalage(entier decal, T_tab_int table)
lexique
    tableau de (decal mod N + 1) entiers tmp // +1 pour eviter un tableau de longueur nulle
    entiers cpt
debut
    // normalisation de decal
    decal ← decal mod N
    // recopier les decal derniers elements de table dans tmp pour sauvegarde
    pour cpt de 1 a decal faire
      tmp[cpt] ← table[N−decal+cpt]
    fin pour
    // decaler les elements de tab de n cases
```

```
    pour cpt de 0 a N-decal-1 faire
        table[N-cpt] ← table[N-decal-cpt]
    fin pour
    // recopier les n elements sauvegardes en debut de tab
    pour cpt de 1 a decal faire
        table[cpt] ← tmp[cpt]
    fin pour
fin
//------------------------------------------------------------
procedure decalage_rec(entier decal, m T_tab_int table)
lexique
    entiers cpt, tmp
debut
    // base : si decal=0, rien a faire
    si decal ≠ 0 alors // recurrence
        // sauvegarde du dernier element
        tmp ← table[N]
        // decalage d'une case a droite
        pour cpt de 1 a N-1 faire
            table[N-cpt+1] ← tab[N-cpt]
        fin pour
        // recopie de l'element sauvegarde
        table[1] ← tmp
        // appel recursif
        decalage_rec(n-1, table)
    fin si
fin
```

Exercice 3.7 (Polylignes)

Une ligne polygonale du plan est définie par la liste ordonnée de ses sommets. Un sommet est un point représenté par un enregistrement contenant ses coordonnées réelles x et y.
Exemple :



1. Définir les différents types pour représenter une ligne polygonale.
2. Écrire une procédure permettant de saisir au clavier une ligne polygonale.
3. Écrire une fonction qui calcule la longueur d'une ligne polygonale donnée.
4. Écrire une procédure qui supprime d'une ligne polygonale le sommet numéro num passé en paramètre.
5. Écrire une fonction qui renvoie vrai si la ligne polygonale est fermée et faux sinon.

▽ **Correction**

1.

```
types
    t_point = enregistrement
            reel abscisse, ordonnee
        fin enregistrement
    t_polyligne = enregistrement :
```

```
                tableau de N t_points sommets
                entier nb_sommets
             fin enregistrement
```

2.

```
procedure saisir_point(m t_point p)
debut
   ecrire("veuillez entrer les coordonnees du point (deux reels) : ")
   lire(p.abscisse, p.ordonnee)
fin
```

Pour saisir une ligne, saisir successivement les sommets. Arrêt de la saisie sur ordre de l'utilisateur.

```
procedure saisir_ligne(m t_polyligne ligne)
lexique
   caractere rep
   entier i
debut
   i ← 1
   repeter
         saisie_point(ligne.sommets[i]);
         i ← i+1; // un point de plus
         ecrire "continuer (o/n) ? "
         lire rep
   jusqu'a (rep = 'n') ou (ligne.nb_sommets > N)
   ligne.nb_sommets ← i−1
   si (ligne.nb_sommets = N) alors
      ecrire "Plus de place pour d'autres sommets"
   fin si
fin
```

3. La fonction dist calcule la distance euclidienne entre 2 points

```
fonction longueur(d t_polyligne ligne) : reel
lexique
   t_point pointcourant, pointprec
   entier i
   reel lg
debut
      pointprec.abscisse ← ligne.sommets[1].abscisse
      pointprec.ordonnee ← ligne.sommets[1].ordonnee
      long ← 0
      pour i de 2 a ligne.nb_sommets faire
         pointcourant.abscisse ← ligne.sommets[i].abscisse
         pointcourant.ordonnee ← ligne.sommets[i].odonnee
         lg ← lg + dist(pointcourant,pointprec)
         pointprec.abscisse ← pointcourant.abscisse
         pointprec.ordonnee ← pointcourant.ordonnee
      fin pour
      retourner lg
fin
```

4. La liste des sommets étant ordonnee, pour supprimer un sommet du tableau il faut effectuer des decalages. (les sommets doivent occupent des places consécutives dans le tableau).Décrementer ensuite le nombre de sommets de la ligne.

```
procedure supprimer_sommet(m t_polyligne ligne, entier num)
lexique
   entier i
debut
   si (1 ≤ num ≤ ligne.nb_sommets) alors
      pour i de num a ligne.nb_sommets−1 faire
```

```
        ligne.sommets[i].abscisse ← ligne.sommets[i+1].abscisse
        ligne.sommets[i].ordonnee ← ligne.sommets[i+1].ordonnee
      fin pour
      ligne.nb_sommets ← ligne.nb_sommets − 1
   finsi
fin
```

5. Précondition : le nombre de sommets est non nul.

```
fonction ligne_fermee(d t_polyligne ligne) : booleen
debut
   retourner (ligne.sommets[1].abscisse = ligne.sommets[ligne.nb_sommet].abscisse)
             et (ligne.sommets[1].ordonnee = ligne.sommets[ligne.nb_sommet].ordonnee)
fin
```

Exercice 3.8 (Jeu de Nim)

Deux joueurs s'opposent dans un jeu de Nim : sur une rangée de 15 allumettes, ils peuvent retirer, à tour de rôle, une, deux ou trois allumettes. Le gagnant est celui qui retire la dernière allumette. Modéliser ce jeu au moyen d'un tableau de booléens à 15 cases, où chaque joueur piochera à un bout du tableau : le joueur 1 en début de tableau, le joueur 2 en fin de tableau. Déterminer la condition de victoire, et écrire l'algorithme permettant de jouer.

▽ Correction

L'analyse doit faire apparaitre le notion de tour de jeu, établir comment le jeu s'arrête, etc.

```
constante N = 15
type   T_tab_bool = tableau de N booleen

   // affiche le jeu
   procedure affiche_allumettes(T_tab_bool jeu)
   lexique
      entier cpt
   debut
      pour cpt de 1 a N faire
         si (jeu[cpt] = faux) alors
            ecrire(' ') // un espace
         sinon
            ecrire('|') // une allumette
         fin si
      fin pour
   fin

   lexique
      tab_bool allumettes
      entier tour, choix, cpt, j1, j2

   debut
      // remplissage du tableau
      pour cpt de 1 a N faire
         allumettes[cpt] ← vrai
      fin pour
      // initialisation du jeu
      tour ← 0
      victoire ← faux
      j1 ← 1
      j2 ← N

      // boucle de jeu
```

```
      tant que (j1≤j2) faire
         // debut d'un nouveau tour
         tour ← tour+1
         si (tour mod 2 = 1) alors // joueur 1
            ecrire("A vous de jouer, joueur 1")
         sinon // joueur 2
            ecrire("A vous de jouer, joueur 2")
         fin si
         ecrire("allumettes : ")
         affiche_allumettes(allumettes)
         repeter // controle de saisie
            ecrire("combien en prenez-vous (1 a ", min(3,j2−j1+1), ") : ")
            lire choix
         jusqu'a (1≤ choix ≤ 3) et (choix ≤ j2−j1+1)
         //retrait des allumettes
         si(tour mod 2 = 1) alors // joueur 1
            pour cpt de j1 a j1+choix−1 faire
               allumettes[cpt] ← faux
            fin pour
            j1 ← j1+choix
         sinon // joueur 2
            pour cpt de j2−choix+1 a j2 faire
               allumettes[cpt] ← faux
            fin pour
            j2 ← j2−choix
         fin si
      fin tant que
      si tour mod 2 = 1 alors // joueur 1 gagne
         ecrire "bravo joueur 1, vous avez gagne !"
      sinon
         ecrire "bravo joueur 2, vous avez gagne !"
      fin si
   fin
```

Exercice 3.9 (Dominance de Pareto)

Soient v_1 et v_2 deux vecteurs de $\mathbb{R}^n$. On dit que v_1 domine v_2 au sens de Pareto si $\forall i \in \{1, \ldots, n\}, v_1[i] \geq v_2[i]$ et $\exists j \in \{1, \ldots, n\}, v_1[j] > v_2[j]$. écrire une fonction prenant en paramètre deux tableaux de réels de même taille et contenant le même nombre d'éléments (ils représentent les vecteurs v_1 et v_2) et renvoyant VRAI si et seulement si v_1 domine v_2.

▽ **Correction**

```
type
    T_tab_reels = tableau de N reel

fonction dominance(d T_tab_reels v1, d T_tab_reels v2) : booleen
lexique
    entier dimension, i
    booleen domine, egal
 debut
    dimension ← N
    domine ← vrai
    egal ← vrai
    i ← 1
    tant que domine et (i ≤ dimension) faire
      si v1[i] < v2[i] alors
```

```
      domine ← faux
    sinon
      si v1[i] > v2[i] alors
        egal ← faux
      fin si
    fin si
    i ← i + 1
  fin tant que
  retourner domine et non egal
fin
```

Exercice 3.10 (Crible d'Ératosthene)

Un nombre est dit premier s'il admet exactement deux diviseurs distincts, lui-même et l'unité. Au 3^{e} siècle avant notre ère, le philosophe et mathématicien grec *Ératosthene* a proposé un procédé simple pour déterminer tous les nombres premiers inférieurs à un entier n donné : dans la liste des entiers de 2 à n, rayer peu à peu tous ceux qui admettent des diviseurs ; le premier entier, 2, n'est pas rayé ; donc ses multiples sont rayés ; le prochain entier non rayé est 3, et ses multiples sont rayés ; Le prochain entier non rayé est 5, donc tous les multiples de 5 sont rayés... et ainsi de suite jusqu'à avoir épuisé tous les entiers inférieurs à n. À la fin, ceux qui ne sont pas rayés sont premiers. Écrire un algorithme qui réalise ce procédé sur un tableau de booléens où la valeur $faux$ signifie que l'entier indiçant la case est rayé. Les nombres premiers trouvés seront affichés.

▽ **Correction**

```
      T_tab_booleens = tableau de N+1 booleen
//------------------------------------
procedure remplir(m  T_tab_booleens table)
lexique
    Entier i
debut
   pour i de 1 a N faire
      table[i] ← vrai
   fin pour
fin
//------------------------------------
procedure rayer(m T_tab_booleens table, d entier nombre)
lexique
   Entier j
debut
   pour j de 2 a (N div nombre) faire // optim : cribler seulement les multiples de i
      table[j*nombre] ← faux
   fin pour
fin
//------------------------------------
procedure resoudre(m T_tab_booleens table)
lexique
    entier i, j
debut
   pour i de 2 a racine(N) faire  // optim : s'arreter a racine de taille
      si (table[i]) alors
         rayer(table ,i)
      finsi
   fin pour
fin
//------------------------------------
procedure afficher(d T_tab_booleens table)
lexique
```

```
   entier i
debut
   ecrire "les nombres premiers sont : " //affichage des nombres premiers
   pour i de 1 a N faire
      si (tab[i]) alors
         ecrire i
      fin si
   fin pour
fin
//-----------------------------
// algo principal
lexique
   T_tab_booleens crible
debut
   remplir(crible)
   resoudre(crible)
   afficher(crible)
fin
```

<u>Partie TP</u> (2 séances)

Récupérez le programme `binaire_en_decimal.cpp` sur Madoc et ouvrez le avec l'éditeur que vous utilisez habituellement en travaux pratiques.

Ce programme utilise des tableaux. Il commence par définir une **constante globale** N valant 5. L'effet de cette définition est que partout où apparaît N dans le programme, la valeur 5 sera utilisée.

Le programme définit ensuite un **alias de type** au moyen de l'instruction `typedef int T_tab_int[N];`. Cet alias porte le nom `T_tab_int` et correspond au type tableau de N entiers. Notez la position des crochets pour définir la taille d'un tableau. Notez aussi qu'en C++ la taille d'un tableau est définie dès la déclaration du tableau et que les cases sont numérotées à partir de `0`.

Il n'est pas possible d'afficher le contenu d'un tableau en demandant simplement l'affichage de la variable de type tableau : en C++ la valeur d'une variable de type tableau est *l'adresse* du tableau dans la mémoire, notion que nous approfondirons plus tard. Il est donc impératif, au moyen d'une boucle, d'afficher les cases du tableau une à une. C'est ce que fait la procédure `affiche_tableau` qui prend en paramètre un élément de type `T_tab_int` (un tableau de N entiers donc) et réalise l'affichage de ses éléments par indices décroissants, de $N-1$ à 0.

Enfin, le programme principal déclare une variable de type `T_tab_int` mais réalise aussi **l'initialisation à la déclaration** d'un tableau de réels $tabf$. L'initialisation à la déclaration d'un tableau en C++ consiste à déclarer le tableau sans en préciser la taille mais en lui affectant tout de suite, sur la même ligne, un contenu pour chaque case, séparé par des virgules et compris entre accolades ; la taille du tableau se déduit de la quantité de contenu fourni. Après cette opération, $tabf$ est donc un tableau de 5 cases de réels qui sont d'emblée initialisées aux valeurs des premières puissance de 2.

1. Compilez et testez ce programme.
2. Modifiez le programme afin qu'il accepte des nombre d'un octet (8 bits).

1 Transcription

Transcrire les algorithmes vus en TD en programmes C++.