1. Quelques algorithmes fondamentaux sur les vecteurs (ou tableaux)

Dans la suite, on note les variables de façon différente selon qu'elles sont passées à la fonction par adresse ou par valeur. Lors du codage dans un langage de programmation X, il convient d'adapter ces conventions à la syntaxe du langage utilisé. En particulier, si le langage utilisé est le C++, le passage par adresse peut prendre deux formes : *pointeur* ou *référence* (à noter, pas de passage par référence en C).

Remarques :

1. dans les langages utilisés, les indices varient à partir de la valeur 0, cette convention est utilisée dans la suite,
2. le type des éléments des vecteurs peut être quelconque ; dans la suite, on désigne par T le type de ces éléments. Il est important de noter que les notations supposent que tous les opérateurs sont utilisables avec tout type T, ce qui n'est pas toujours vrai (c'est faux en particulier pour les chaînes de caractères au sens C, de même que pour String en Java) : comparer l'exemple utilisant le type string et le type (char *) pour effectuer le même traitement.
3. dans certains cas, en C/C++, on peut choisir, pour des raisons d'efficacité du code des passages de paramètres différents de ceux utilisés ici. A préciser lors des TP,
4. l'utilisation d'un "for" est bien adapté au traitement des vecteurs, l'utilisation d'un "tant que" est toutefois préférée au niveau algorithmique à cause des variantes trop nombreuses de représentation d'un schéma "pour".

1.1. *Partie I : vecteurs non triés*

1.1.1. Parcours

Il est possible de parcourir les éléments d'un vecteur du premier jusqu'au dernier. Ce traitement peut permettre l'affichage du contenu, l'initialisation, ou la recherche d'un élément maximal. Seul le traitement au sein de la boucle de parcours change. Ecrire une fonction Parcours_Vecteur permettant par exemple d'afficher le contenu d'un vecteur. Voici le prototype de la fonction :

```
algo vide Parcours_Vecteur (↔ T v[],→ entier n)
```

Vous disposez d'une fonction `affiche (Elem)` permettant d'afficher le contenu d'un élément Elem de type *T*.

```
algo vide Affiche (→ T Elem)
```

Remarque : Le passage par adresse peut se faire avec le symbole « ↔ » ou en écrivant adr. Le passage par valeur s'écrira « → » ou val. Le prototype de la méthode précédente sera donc :

```
algo vide Parcours_Vecteur (adr T v[], val entier n)
```

Solution :

```
algo vide Parcours_Vecteur (adr T v[], val entier n)
var
      val entier i;

début

      i ← 0;
      tantque (i < n)
            // un traitement quelconque sur l'élément courant
            // par exemple
            // affiche(v[i]) ;
            i ← i + 1;
      fintq;

fin; // Parcours_Vecteur
```

1.1.2. Accès en position k : trivial

L'accès a un élément *k* est trivial avec un vecteur v. il s'agit de v[k]

1.1.3. Accès associatif (ou par valeur)

On recherche dans un vecteur v de n éléments la position (indice) d'une valeur val. La fonction retourne la position de val dans v si cette valeur est présente, -1 sinon. Ecrire cette fonction.
Le prototype de la fonction est :

```
algo entier Acces_Associatif_Vecteur (adr T v[], val entier n, val T val)
```

Solution

```
algo entier Acces_Associatif_Vecteur (adr T v[], val entier n, val T val)
variable
      val entier i;

début
      i ← 0;
      tantque (i < n)
            si (v[i] == val)
                  retourner i;
            finsi;
            i ← i + 1;
      fintq;

      retourner -1;

fin; // Acces_Associatif_Vecteur
```

1.1.4. Insertion en position k dans un vecteur

On cherche à insérer dans un vecteur v de n éléments une valeur valins en position k. La fonction retourne vrai si l'insertion a pu être réalisée. Le nombre d'éléments de v est modifié en fin d'action : la vérification que le vecteur peut comporter 1 élément supplémentaire est laissée à la fonction appelante (donc au programmeur si utilisation d'un vecteur statique en C/C++).
Voici le prototype de la fonction à réaliser :

```
algo logique Insertion_Position_Vecteur (adr T v[], adr entier n, val T valins, val entier k)
```

Solution :

```
algo logique Insertion_Position_Vecteur (adr T v[], adr entier n, val T valins, val entier k)
var
      val entier i;

début

      si (( k < 0) ou ((k > n))
            retourner faux;
      finsi;
      si (k == n)
            v[n] = valins;
            n ← n + 1;
            retourner vrai;
      finsi;
      i ← n;
      tantque (i ≥ k)
            v[i] ← v[i-1]
            i ← i - 1;
      fintq;
      v[k] ← valins;
      n ← n + 1;
```

```
        retourner vrai;

fin;  // Insertion_Position_Vecteur
```

1.1.5. Insertion associative dans un vecteur

On cherche à insérer dans un vecteur v de n éléments une valeur valins après un élément de valeur apres.

Même hypothèses que pour le paragraphe précédent.
Solution

```
algo logique Insertion_Associative_Vecteur (adr T v[], adr entier n,
val T valins, val T apres)
var
      val entier k;

début

      k ← Acces_Associatif_Vecteur (v, n, apres);
      si (k == -1) alors
            retourner faux;
      sinon
            retourner
                  Insertion_Position_Vecteur (v, n, valins,k+1);
      finsi;

fin;  // Insertion_Associative_Vecteur
```

1.1.6. Suppression en position k dans un vecteur

On cherche à supprimer dans un vecteur v de n éléments l'élément situé en position k. La fonction retourne vrai si la suppression a pu être réalisée. Le nombre d'éléments de v est modifié en fin d'action.
Solution :

```
algo logique Suppression_Position_Vecteur
      (adr T v[], adr entier n, val entier k)
var
      val entier i;

début

      si (( k < 0) ou ((k > n))
            retourner faux;
      finsi;
      si (k = n-1)
            n ← n - 1;
            retourner vrai;
      finsi;
      i ← k;
      tantque (i < n)
            v[i] ← v[i+1]
            i ← i + 1;
      fintq;
      n ← n - 1;

      retourner vrai;

fin;  // Suppression_Position_Vecteur
```

1.1.7. Suppression associative dans un vecteur

On cherche à supprimer dans un vecteur v de n éléments une valeur valsup. Le nombre d'éléments de v est modifié en fin d'action.

Solution :

```
algo logique Suppression_Associative_Vecteur
      (adr T v[], adr entier n, val T valsup)
var
      val entier k;

début

      k ← Acces_Associatif_Vecteur (v, n, valsup);
      si (k = -1) alors
            retourner faux;
      sinon
            retourner Suppresssion_Position_Vecteur (v, n, k);
      finsi;

fin; // Suppression_Associative_Vecteur
```

Remarque :

- Vous retrouverez ces exemples de codage de ces fonctions en C++ avec des vecteurs dont les éléments sont de type différent (entiers, chaînes C, string) : voir g.gesquiere.free.fr
- Des conventions de type JAVADOC sont utilisées (on pourrait générer de la documentation programme C++ par Javadoc).

1.2. *Partie II : vecteurs triés*

L'avantage de disposer d'un vecteur trié est que, outre l'accès séquentiel, il est possible de faire un accès dichotomique ("binary search"), ce qui améliore considérablement les performances pour l'accès à l'information (qui est l'un des traitements les plus utilisés dans les applications informatiques). Nous étudierons les algorithmes permettant de trier des vecteurs au deuxième semestre. Le cours sur la complexité permettra de quantifier le coût de chaque algorithme en terme de nombre de comparaisons et nombre d'affectations (échanges par exemple de valeurs) nécessaires à chaque algorithme.

Pour information :

- **Accès séquentiel : O (n)**
- **Accès dichotomique : O ($\log_2$ (n))**

Dans le cas d'un accès séquentiel, la structure de données dans laquelle s'effectue la recherche comporte au départ n éléments, après le premier accès, il reste (au plus) n-1 éléments à examiner, après l'accès suivant, il reste (au plus) n-2 éléments ; de façon générale, après k accès, il reste (au plus) n-k éléments à examiner.
Dans le cas d'un accès dichotomique, la structure de données (qui ne peut être dans ce cas qu'un vecteur trié) dans laquelle s'effectue la recherche comporte au départ n éléments, après le premier accès, il reste (au plus) n/2 éléments à examiner, après l'accès suivant, il reste (au plus) n/4 éléments ; de façon générale, après k accès, il reste (au plus) $n/2^k$ éléments à examiner.

Soit pour un ensemble comportant 1024 éléments :

- Accès séquentiel : cas favorable : 1 accès ; cas défavorable : 1024 ; **en moyenne : 512**.
- Accès dichotomique : cas favorable : 1 accès ; cas défavorable : **au maximum 10**.

Ecrire une fonction permettant un accès dichotomique à une valeur d'un vecteur trié. Le prototype de la fonction est

```
algo entier Acces_Dichotomique_Vecteur
                        (adr T v[], val entier n, val T cherche)
```

Solution

```
algo entier Acces_Dichotomique_Vecteur
                        (adr T v[], val entier n, val T cherche)
var
      val entier inf,
```

```
                sup,
                mil;

début

      si ((v[0] ≤ cherche) et (cherche ≤ v[n-1]))
      inf ← 0;
      sup ← n-1;
            tantque (inf ≤ sup)
                  mil ← (inf+sup) / 2;
                  si (v[mil] = cherche)
                        retourner mil;
                  sinon si (v[mil] < cherche)
                              inf ← mil + 1;
                        sinon
                              sup ← mil - 1;
                        finsi;
      fintq;
      finsi;

      retourner -1;

fin;     // Acces_Dichotomique_Vecteur
```