TD et TP de Java Numéro 3
Les Tableaux 1D et les Tableaux 2D

Dr. KENMOGNE Edith Belise

Questions de Cours

1) On rappelle qu'en java, un tableau est un objet, que l'opérateur d'instanciation (de création) d'un tableau est *new*, et que si *tab* est un tableau, *tab.length* désigne sa taille. Schématiser l'évolution des variables du bout de code suivant. Nous utilisons les classes Point et Cercle du TD/TP n°1.

```
int[] tabint1 ; //Déclaration du tableau
tabint1= new int[5] ; //Création du tableau
String[] args ; //Déclaration du tableau
int[] tabInt2 = {3, 5, 7, 9}; //Déclaration, création et initialisation du tableau
Point[] tabPoint={new Point(2,4), new Point(4,5)} ; //Déclaration, création et initialisation.
Cercle[] tabCercle1={new Cercle(2,4,14),new Cercle(4,5,18)} ;//Déclaration, création, initialisation
Cercle[] tabCercle2= new Cercle[3] ; //Déclaration et création
Point p ; //Déclaration de l'objet
p= new Point(4,5) ; //Création de l'objet
tabCercle2[0]=new Cercle(p, 12) ;
tabCercle2[1]=new Cercle(p, 14) ;
```

2) Schématiser l'évolution des variables du bout de code suivant. Qu'est ce que désigne *notes3.length* ? Qu'est ce que désigne *notes3[i].length* pour i appartenant à *{0, 1, 2}* ?

```
int[][] notes1;
int[][] notes2;
notes1 = new int[3][3]; //Création. Chaque ligne est de taille 3.
int[][] notes3={{3, 4, 5, 6}, {7, 8, 9}, {10, 11}} ; //Déclaration, création, initialisation
notes2 = new int[3][]; //Création. Chaque ligne a une taille qui lui est propre.
note2[0]=new int[4] ;
note2[1]=new int[3] ;
note2[2]=new int[2] ;
```

3) On rappelle qu'on n'a pas le droit d'envoyer un message à une reference nulle, ni a une variable qui ne pointe pas sur un objet. En déduire pourquoi la deuxième instruction du bout de code suivant est incorrecte.

```
Eleve[] tabEleve = new Eleve[5] ; // Déclaration, création du tableau
tabEleve[0].setNom(''EBELE'') ;
```

Exercice 1

Écrire les méthodes de classe de la classe Calcul décrites ci-dessous.

1) Inversion d'un vecteur (tableau 1D) : *void inverse(int[] tab)* qui inverse *tab*. Par exemple le tableau [2, 20, 18, 60, 1] devient [1, 60, 18, 20, 2].

2) Fonctions matricielles : Ecrire les fonctions suivantes :

2.a) *void sum1(float[][] c, float[][] a, float[][] b) et float[][] sum2(float[][] a, float[][]) pour* la somme de deux matrices carrées a et b.

2.b) Même question pour le produit de deux matrices.

2.c) *boolean equals(float[][] a, float[][] b)* retourne *true* si les matrices carrées *a* et *b* sont égales et *false* sinon.

2.d) Une méthode qui retourne A^p où A est une matrice carrée d'ordre n et p un entier positif ou nul. On rappelle que $A^0=I$, où I est la matrice identité. On commencera d'abord par décomposer le problème de départ en sous problèmes.

3) Résolution d'un système triangulaire : Une méthode qui prend en entrée une matrice triangulaire inférieure *A d'*ordre n, un vecteur b de taille *n*, résout le système triangulaire $AX=b$, et retourne *X*. On suppose que tous les coefficients *A[i,i]* sont distincts de zéro.

4) Trie par insertion dans un vecteur (tableau 1D) :

4.a) *void insertion(float[] tab, int index, float val)* qui insère *val* dans *tab* de manière à ce que *tab[0...index+1]* soit trié par ordre croissant, sachant que *tab[0...index]* est trié par ordre croissant.

4.b) Sur la base de *insertion(float[] tab, int index, float val)* écrire une fonction *trie_insertion(float[] tab)* qui trie le tableau *tab*.

5) Recherche Dichotomique(RD) dans un vecteur (tableau 1D) : Écrire une méthode qui prend en argument un vecteur de flottants trié par ordre croissant et la valeur recherchée et dit si cette valeur est dans le vecteur. Le principe de la recherche dichotomique est décrit comme suit :

Si la taille du vecteur est égale à zéro **alors** arrêter la recherche et retourner *false*.

Sinon si la valeur recherchée est égale à la valeur située au milieu du vecteur **alors** arrêter la recherche et retourner *true*.

Sinon si la valeur recherchée est inférieure à la valeur située au milieu du vecteur **alors** poursuivre la recherche dans la partie gauche du vecteur en ré appliquant le principe de RD.

Sinon poursuivre la recherche dans la partie droite du vecteur en ré appliquant le principe de RD.

6) Sous vecteurs parfaits

6.a) Écrire une méthode *boolean sousVecteurParfait(int[] svp, int[] v)* qui retourne *true* si *sv* est une sous-vecteur parfait de *v et false sinon.* Par exemple, *[1,2], [4, 5, 6, 7]* , [6, 7] sont des sous-vecteurs parfaits de *v=[1, 2, 3, 4, 5, 6, 7, 8, 9]*. *[3, 5, 7]* n'est pas un sous-vecteur parfait de *v* puisque ses éléments n'apparaissent pas dans v de manière successive.

6.b) Ecrire une fonction *int sousVecteurParfait(int[] svp, int[] v) qui retourne la première position à partir de laquelle svp est un sous-vecteur parfait de v.* Si cette position n'existe pas la méthode doit retourner -1.

7) Fusion de deux vecteurs triés par ordre croissant

void fusion(int[] c, int[] a, int[] b) fusionne les deux vecteurs *a* et *b* (triés par ordre croissant) dans *c*. On suppose que c est suffisamment grand pour recevoir la fusion. On peut effectuer la fusion de la manière suivante : On considère un index *ia* de *a* initialisé à 0, un index *ib* de *b* initialisé à *0* et un index *ic* de *c* initialisé à *0*. A Chaque étape, la plus petite valeur indexée par *ia* et *ib* est ajoutée à *c* à la position *ic* puis *ic* et l'index ayant produit la plus petite valeur sont incrémentés de *1*. Dès qu'un des deux vecteurs *a* et *b* est épuisé *(ia==a.length ou ib==b.length)* on recopie le reste des éléments du vecteur non-épuisé (à partir de son index) dans *c* (à partir de l'index de *ic* de c).