INFORMATIQUE

Exercice 1 On peut calculer la racine carrée d'un nombre avec l'algorithme de Newton. Soit *x* la racine carrée d'un nombre *n*. On choisit une valeur comme approximation de départ pour *x* (1 par exemple), puis on applique la récurrence :

$$x_{k+1} = 1/2 *(x_k + n/x_k)$$

L'algorithme s'arrête après un nombre donné d'itérations (ou encore, quand l'approximation *x* atteigne une certaine précision). Le programme suivant doit s'arrêter après 10 itérations :

```
import java.util.Scanner;
/**
 * Algorithme de Newton pour calculer la racine quarre
 */
public class Newton {
    private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.println("Donner un nombre : ");
        double n = scanner.nextDouble();
        double x = 1;
        for (int i = 1; i < 10; i++) {
            //calculer la  valeur suivante
            x = (1/2) * (x + n / x);
        }
        System.out.println("La racine de " + n + " est " + x);
    }
}
```

Q1) Essayez l'algorithme avec le nombre 2.0 . La valeur calculée est NaN (Not a Number), … Combien de fois la boucle est exécutée ? Cherchez les erreurs avec le débogage d'Eclipse.

Exercice 2 Ici, on va étudier et tester un algorithme pour trier un tableau. Le tri par bulle est un algorithme simple qui fonctionne de la façon suivante. On parcoure le tableau par une bulle imaginaire de deux éléments, et on change les deux éléments s'ils ne sont pas en bon ordre (puis on déplace la bulle pour les éléments suivants). Voici une illustration pas à pas avec le tableau [3 2 5 1 4] :

[***3 2*** 5 1 4] → [2 ***3 5*** 1 4] → [2 3 ***5 1*** 4] → [2 3 1 ***5 4***] → [2 3 1 4 5] (pas fini!)

[***2 3*** 1 4 5] → [2 ***3 1*** 4 5] → [2 1 ***3 4*** 5] → [2 1 3 ***4 5***] (ici pas de changement, mais ...)

[***2 1*** 3 4 5] → [1 ***2 3*** 4 5] → (pas de changement dans les prochaines bulles)

Voici un premier programme :

```
import java.util.Scanner;
/**
 * Tri par bule
 */
public class Tri1 {
    private static Scanner scanner = new Scanner(System.in);
```

```
    private static void Bule(double[] a) {
    boolean sorted = false;
    double temp;
    while(!sorted) {
        for (int i = 0; i < a.long; i++) {
            if (a[i] > a[i+1]) {
                temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
            }
        }
     }
    }
    public static void main(String[] args) {
        System.out.println("Trier un tableau 1) donnez le nombre d'elements : ");
        int n = scanner.nextInt();
      double tab[] = new double[n];
        for (int i = 0; i < n; i++) {
        System.out.println("Donnez l'element suivant : ");
        tab[i] = scanner.nextDouble();
      }
        Bule(tab);
        System.out.println("Le tableau trie est ");
        for (int i = 0; i < n; i++) {
             System.out.println(tab[i]);
        }
    }
}
```

Q2) Lancez le programme. Cherchez les erreurs avec le débogage d'Eclipse.

Exercice 3 Ici, on va étudier et tester un algorithme plus performant pour trier un tableau, qui utilise l'idée de diviser pour régner : 1) Diviser en découpant le problème en sous-problèmes (en deux par exemple) 2) Régner : résoudre les sous-problèmes (par exemple récursivement) 3) calculer une solution au problème initial à partir des solutions des sous-problèmes.

Voici l'algorithme à étudier qui coupe un tableau à trier en deux parties et qui les trie récursivement avant d'unifier les résultats :

```
public static void fSort(int[] tab, int g, int d) { // g,d indice gauche, droite
    if (d <= g) return;
    int m = (g+d)/2;
    fSort(tab, g, m);
    fSort(tab, m+1, d);
    fusion(tab, g, m, d);
}
```

Pour unifier les sous-tableaux :

```
public static void fusion(int[] tab, int g, int m, int d) {
    // longueurs des soustableaux
    int longg = m  g + 1;
    int longd = d  m;

    // copies des soustableaux  eja tries
    int gtab[] = new int [longg];
    int dtab[] = new int [longd];
    for (int i = 0; i < longg; i++)
```

```
            gtab[i] = tab[g+i];
        for (int i = 0; i < longd; i++)
            dtab[i] = tab[m+i+1];

        // indices dans les soustableaux
        int gind = 0;
        int dind = 0;

        // construire le tableau d'origine depuis les copies gtab and dtab
        for (int i = g; i < d + 1; i++) {
            // s'il y a des éléments à copier dans les deux tableaux, copier le min
            if (gind < longg && dind < longd) {
                if (gtab[gind] < dtab[dind]) {
                    tab[i] = gtab[gind];
                    gind++;
                }
                else {
                    tab[i] = dtab[dind];
                    dind++;
                }
            }
            // si dtab est déjà copié, copie du rest de gtab
            else if (gind < longg) {
                tab[i] = gtab[gind];
                gind++;
            }
            // si gtab est déjà copié, copie du rest de dtab
            else if (dind < longd) {
                tab[i] = dtab[dind];
                dind++;
            }
        }
    }
```

Q3) Complétez (main) et lancez la classe. A l'aide du débogueur d'Eclipse, observez l'évolution des variables et des tableaux . Tracez cette évolution sur papier.

Pour comprendre le fonctionnement de `fusion`, voici le méta-code d'une fusion :

entrée : deux tableaux triés A et B
sortie : un tableau trié qui contient exactement les éléments des tableaux A et B
fonction fusion(A[1, …, a], B[1, …, b])
    si A est le tableau vide
        renvoyer B
    si B est le tableau vide
        renvoyer A
    si A[1] ≤ B[1]
        renvoyer A[1] :: fusion(A[2, …, a], B)
    sinon
        renvoyer B[1] :: fusion(A, B[2, …, b])