Laboration 4 - Sortering

Syftet med laborationen är att du ska implementera ett par sorteringsmetoder och använda interfacen *Comparable* och *Comparator*.

Filer som används i laborationen är:

- paketet f4: Sorting.java
  Utility.java
- paketet laboration4: RealNbr.java
  Person.java
  IdAscending.java
  SortPersons.java

Grundläggande uppgifter

1. Skriv metoden **public static double[] randomArray(int n, double min, double max)** vilken skapar en double-array med kapaciteten **n** och fyller den med slumpmässiga double-värden i intervallet **min** – **max** (dock ej **max**).
   Se även filen Utility.java i föreläsningsmaterialet (Sortering (Jonas)).

2. Skriv en metod, **insertionsortDesc**, som sorterar en double-array avtagande. Gör sorteringen med hjälp av **insertionsort**.

3. Skriv en metod, **bubblesortDesc**, som sorterar en double-array avtagande. Gör sorteringen med hjälp av **bubblesort**.

4. Skriv alternativa metoder för dina lösningar i uppgift 2-3. Dessa lösningar ska räkna antalet jämförelser och swapar när sorteringen genomförs. Sedan ska resultatet skrivas ut. (se även exempelfiler på föreläsningsunderlaget (Sortering (Jonas)))

5. Skriv en metod som sorterar en double-array avtagande. Gör sorteringen med hjälp av **mergesort**.

6. Testa hur lång tid det tar att sortera en double-array med 10000 element med dina lösningar på uppgift 2, 3 och 5.
   Tidstestningen gör du så här (sort refererar till klass med metoden insertionsort):

```
long first, stop;
first = System.currentTimeMillis();
sort.insertionsort( array );
stop = System.currentTimeMillis();
System.out.println( "Insertionsort: " + ( stop - first ) + " ms " );
```

7. Skriv metoden
   public static void swap(Object[] array, int i, int j)
   vilken skiftar plats på elementen i position i och j i arrayen array.

8. På websidan ***http://www.cs.uwaterloo.ca/~bwbecker/sortingDemo/index.html*** kan man jämföra ett antal sorteringsalgoritmer avseende antalet jämförelser (Compare) och antalet byten av element (Moves). Ju färre av vardera desto bättre sorteringsalgoritm.
   Starta sortering (klicka på Start-knappen) för följande sorteringsalgoritmer:
   Bubble Sort, Selection Sort, Insertion Sort, Merge Sort (översta) och Quick Sort (andra).
   Sorteringen kommer att pågå ett antal minuter så fortsätt med nästa uppgift och titta till

sorteringssidan ibland. När sorteringarna är färdiga så skriv upp antalet jämförelser och antalet byten av element för vardera algoritm.

9. Skriv en metod som sorterar en Object-array växande. Använd **bubblesort** i din lösning.
För att sorteringen ska fungera så måste objekten i arrayen implementera interfacet Comparable. Klassen RealNbr implementerar Comparable. Jämförelsen mellan objekten i position j-1 och j blir så här:

```
Comparable com = ( Comparable )array[ j-1 ];
if( comp.compareTo( array[ j ] ) > 0)
    // swap elements
```

Metoden **randomNbr** i klassen RealNbr returnerar en array med n st RealNbr-objekt. Dessa objekt har slumpmässiga värden i intervallet min - max. Skapa en RealNbr-array och testa din sorteringsmetod med hjälp av RealNbr-arrayen.

10. I föreläsningsunderlaget saknas metoden **partition** för att **quicksort** ska fungera.
Komplettera med metoden *partition* så att *quicksort*-metoden fungerar för int-arrayer.

```
public class QuickSort {
    public static void sort(int[] arr) {
        sort(arr,0,arr.length-1);
    }

    private static void sort(int[] arr, int left, int right) {
        int pivotIndex;
        if(left<right) {
            pivotIndex = partition(arr,left,right,(left+right)/2);
            sort(arr,left,pivotIndex-1);
            sort(arr,pivotIndex+1,right);
        }
    }

    private static int partition(int[] arr, int left, int right,
                                 int pivotIndex) {
        int pivotValue = arr[pivotIndex];
        // komplettera
    }
}
```

partition-metoden ska dela upp innehållet i positionerna *start-end* i arrayen *arr* så att alla element som är mindre än *pivotValue* är till vänster om *pivotValue* och övriga element till höger om *pivotValue*.

Exempel (pivotValue svart, blå element = de som är mindre än pivotValue, röda element = övriga, ordningen på blå element / röda element beror på din algoritm

```
                                  arr =  {25,11,19,12,14,15,18,15,17};
partition(arr, 0, arr.length-1, 4); =>  {11,12,14,25,17,15,18,15,19}

                                  arr =  {25,11,19,12,14,15,18,15,17};
partition(arr, 2, 5, 3); =>             {25,11,12,15,14,19,18,15,17}

                                  arr =  {25,11,19,12,14,15,18,15,17};
partition(arr, 0, 4, 2); =>             {11,14,12,19,25,15,18,15,17}
```

Om du vill kan du följa nedanstående algoritm:

Tilldela pivotValue värdet i arr[pivotIndex]
Tilldela variablen storeIndex värdet i left
Skifta (swap) elementen i positionerna pivotIndex och right
För varje element i positionerna left till right-1
  Om elementet i vald position < pivotValue så
    Skifta valt element med elementet i storeIndex
    Öka storeIndex med 1
Skifta elementen i positionerna storeIndex och right
Returnera storeIndex

Fördjupande

11. Lös uppgift 9 på nytt men använd **mergesort** för att sortera arrayen. Det är i merge-metoden som jämförelsen mellan element görs.

12. Objekt som implementerar **Comparable** kan endast sorteras på ett sätt. Med hjälp av en hjälpklass, en implementering av interfacet **Comparator**, går det att sortera en array med objekt på andra sätt. Det gäller bara att skriva Comparator-implementeringen så att sorteringen sker på önskvärt sätt.
    Filerna SortPersons.java, Person.java och IdAscending.java ger exempel på hur en Comparator-implementering kan användas tillsammans med Arrays.sort. Det är Person-objekt som sorteras med hjälp av klassen IdAscending.
    a) Testkör SortPersons.java
    b) Ändra sorteringsanropet från Arrays.sort(…); till f4.Sorting.bubblesort(…);
    Kör SortPersons med debuggern efter ändringen. Nu kan du se hur jämförelser mellan Person-objekt sker i compare-metoden i klassen IdAscending.

13. Skriv en egen Comparator-implementering, **LastnameAsc**, som sorterar Person-objekten växande avseende efternamn. Du kan utgå från att alla efternamn börjar på 'A' – 'Z' (String-klassens compareTo-metod duger för jämförelse).

14. Skriv även en klass, **FirstnameAsc**, som på samma sätt sorterar Person-objekt växande men med avseende på förnamnen.

    Hur blir sorteringen om du gör anropen:
    ```
    Arrays.sort( persons, new FirstnameAsc() );
    Arrays.sort( persons, new LastnameAsc() );
    ```

    Hur blir sorteringen om du ändrar ordningen på anropen?
    ```
    Arrays.sort( persons, new LastnameAsc() );
    Arrays.sort( persons, new FirstnameAsc() );
    ```

15. Skriv metoden
    public static void insertionsort(Object[] array, Comparator comp)
    vilken ska ordna objekten med hjälp av en klass som implementerar Comparator.
    Metoden måste alltså anropa metoden **comp.compare(Object, Object)** för att få en jämförelse.
    Testa din metod med hjälp av SortPersons.java, Person.java och någon av Comparator-implementeringarna från uppgift 12 eller uppgift 13.

Extra

16. Skriv en **Comparator**-implementering som sorterar Person-objekt (se uppgift 11) avtagande med avseende på deras vikt. Om flera personer har samma vikt ska de sorteras så att den kortaste personen står först (växande med avseende på längd).

17. Skriv metoderna **mergesort(Object[] array, Comparator comp)** resp **quicksort(Object[] array, Comparator comp)**. Ersätt anropen i uppgift 13 med två anrop till mergesort respektive quicksort, t.ex.:

    ```
    mergesort( personer, new FirstnameAsc () );
    mergesort( personer, new LastnameAsc() );
    ```

    Vad blev resultatet för **mergesort** respektive **quicksort**?

Lösningar

Uppgift 1

```
public static double[] randomArray(int n, double min, double max) {
    double[] arr = new double[n];
    for (int i = 0; i < arr.length; i++) {
        arr[i] = Math.random() * (max - min) + min;
    }
    return arr;
}

private static void swap(double[] array, int elem1, int elem2) {
    double temp = array[elem1];
    array[elem1] = array[elem2];
    array[elem2] = temp;
}
```

Uppgift 2

```
public static void insertionSortDesc(double[] array) {
    for (int i = 1; i < array.length; i++) {
        for (int j = i; (j > 0) && (array[j - 1] > array[j]); j--) {
            swap(array, j, j - 1);
        }
    }
}
```

Uppgift 3

```
public static void bubblesortDesc(double[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        for (int j = array.length - 1; j > i; j--) {
            if (array[j] < array[j - 1]) {
                swap(array, j, j - 1);
            }
        }
    }
}
```

Uppgift 4

```
public static void insertionSortDescC(double[] array) {
    int swap = 0, comp = 0;
    for (int i = 1; i < array.length; i++) {
        comp++;
        for (int j = i; (j > 0) && (array[j - 1] > array[j]); j--) {
            swap++;
            swap(array, j, j - 1);
            comp++;
        }
    }
    System.out.println("Jämförelser: " + comp + "  Swappar: " + swap);
}

public static void bubblesortDescC(double[] array) {
    int swap = 0, comp = 0;
    for (int i = 0; i < array.length - 1; i++) {
        for (int j = array.length - 1; j > i; j--) {
            comp++;
            if (array[j] < array[j - 1]) {
                swap++;
                swap(array, j, j - 1);
            }
        }
    }
    System.out.println("Jämförelser: " + comp + "  Swappar: " + swap);
}
```

Uppgift 5

```
    public static void mergeSortDesc(double[] array) {
        double[] temp = new double[array.length];
        mergesortDesc(array, 0, array.length, temp);
        temp = null;
    }

    private static void mergesortDesc(double[] array, int first, int n, double[] temp) {
        int n1, n2;
        if (n > 1) {
            n1 = n / 2;
            n2 = n - n1;
            mergesortDesc(array, first, n1, temp);
            mergesortDesc(array, first + n1, n2, temp);
            mergeDesc(array, first, n1, n2, temp);
        }
    }

    private static void mergeDesc(double[] array, int first, int n1, int n2, double[]
temp) {
        int counter = 0, cursor1 = 0, cursor2 = n1, end = n1 + n2;
        while ((cursor1 < n1) && (cursor2 < end)) {
            if (array[first + cursor1] > array[first + cursor2]) {
                temp[counter] = array[first + cursor1];
                cursor1++;
            } else {
                temp[counter] = array[first + cursor2];
                cursor2++;
            }
            counter++;
        }
        while (cursor1 < n1) {
            temp[counter] = array[first + cursor1];
            cursor1++;
            counter++;
        }
        while (cursor2 < end) {
            temp[counter] = array[first + cursor2];
            cursor2++;
            counter++;
        }
        for (int i = 0; i < n1 + n2; i++) {
            array[first + i] = temp[i];
        }
    }
```

Uppgift 7

```
    public static void swap(Object[] array, int i, int j) {
        Object temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

Uppgift 9

```
    public static void bubblesort(Object[] array) {
        Comparable comp;
        for (int i = 0; i < array.length - 1; i++) {
            for (int j = array.length - 1; j > i; j--) {
                comp = (Comparable) array[j - 1];
                if (comp.compareTo(array[j]) > 0) {
                    swap(array, j, j - 1);
                }
            }
        }
    }
```

Uppgift 10

```
    private static int partition(int[] arr, int left, int right, int pivotIndex) {
        int pivotValue = arr[pivotIndex];
        int storeIndex = left;
        swap(arr,pivotIndex,right);
        for(int i=left; i<right; i++) {
            if(arr[i]<pivotValue) {
                swap(arr,i,storeIndex);
                storeIndex++;
            }
        }
        swap(arr,storeIndex,right);
        return storeIndex;
    }

    private static void swap(int[] arr, int pos1, int pos2) {
        int temp = arr[pos1];
        arr[pos1] = arr[pos2];
        arr[pos2] = temp;
    }
```

Uppgift 11

```
    public static void mergeSort(Object[] array) {
        Object[] temp = new Object[array.length];
        mergesort(array, 0, array.length, temp);
        temp = null;
    }

    private static void mergesort(Object[] array, int first, int n, Object[] temp) {
        int n1, n2;
        if (n > 1) {
            n1 = n / 2;
            n2 = n - n1;
            mergesort(array, first, n1, temp);
            mergesort(array, first + n1, n2, temp);
            merge(array, first, n1, n2, temp);
        }
    }

    private static void merge(Object[] array, int first, int n1, int n2, Object[] temp) {
        int counter = 0, cursor1 = 0, cursor2 = n1, end = n1 + n2;
        while ((cursor1 < n1) && (cursor2 < end)) {
            if (((Comparable) array[first + cursor1]).compareTo(array[first + cursor2]) <
0) {
                temp[counter] = array[first + cursor1];
                cursor1++;
            } else {
                temp[counter] = array[first + cursor2];
                cursor2++;
            }
            counter++;
        }
        while (cursor1 < n1) {
            temp[counter] = array[first + cursor1];
            cursor1++;
            counter++;
        }
        while (cursor2 < end) {
            temp[counter] = array[first + cursor2];
            cursor2++;
            counter++;
        }
        for (int i = 0; i < n1 + n2; i++) {
            array[first + i] = temp[i];
        }
    }
```

Uppgift 12

```
class LastnameAsc implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        return p1.getLastname().compareTo(p2.getLastname());
    }
}
```

Uppgift 13

```
class FirstnameAsc implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        return p1.getFirstname().compareTo(p2.getFirstname());
    }
}
```

Uppgift 14

```
    public static void insertionsort(Object[] array, Comparator comp) {
        for (int i = 1; i < array.length; i++) {
            for (int j = i; (j > 0) && comp.compare(array[j - 1], array[j]) > 0; j--) {
                swap(array, j, j - 1);
            }
        }
    }
```

Uppgift 16

```
class WeightLength implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        double res = p2.getWeight() - p1.getWeight ();
        if (res < 0) {
            return -1;
        } else if (res > 0) {
            return 1;
        } else {
            return p1.getLength() - p2.getLength(); // p1 kortast så negativt, p2 kortast
så positivt
        }
    }
}
```

Uppgift 17

```
    public static void mergeSort(Object[] array, Comparator comp) {
        Object[] temp = new Object[array.length];
        mergesort(array, 0, array.length, temp, comp);
        temp = null;
    }

    private static void mergesort(Object[] array, int first, int n, Object[] temp,
Comparator comp) {
        int n1, n2;
        if (n > 1) {
            n1 = n / 2;
            n2 = n - n1;
            mergesort(array, first, n1, temp, comp);
            mergesort(array, first + n1, n2, temp, comp);
            merge(array, first, n1, n2, temp, comp);
        }
    }

    private static void merge(Object[] array, int first, int n1, int n2, Object[] temp,
Comparator comp) {
        int counter = 0, cursor1 = 0, cursor2 = n1, end = n1 + n2;
        while ((cursor1 < n1) && (cursor2 < end)) {
            if (comp.compare(array[first + cursor1], array[first + cursor2]) < 0) {
                temp[counter] = array[first + cursor1];
                cursor1++;
            } else {
                temp[counter] = array[first + cursor2];
                cursor2++;
            }
            counter++;
        }
        while (cursor1 < n1) {
            temp[counter] = array[first + cursor1];
            cursor1++;
            counter++;
        }
        while (cursor2 < end) {
            temp[counter] = array[first + cursor2];
            cursor2++;
            counter++;
        }
        for (int i = 0; i < n1 + n2; i++) {
            array[first + i] = temp[i];
        }
    }
}
```