Übungsblatt 4 (Block 1)

Prof. Dr. Olaf Hellwich und Mitarbeiter

Handsimulation und Implementierung von Sortierverfahren

Erinnerung: Bitte denken Sie an den abschließenden Block-Test.

Verfügbar ab:	**14.05.18**
Abgabe bis:	**21.-25.05.18**

Aufgabe 1: MySort

5 Punkte

Laden Sie zur Bearbeitung der Aufgaben das komprimierte Verzeichnis `Vorgaben.zip` von der Webseite der Veranstaltung herunter. Das Verzeichnis enthält die Dateien

filme1.dat	- Textdatei, die Daten von Filmen enthält (unsortiert).
filme2.dat	- Textdatei, die Daten von Filmen enthält (aufsteigend sortiert).
filme3.dat	- Textdatei, die Daten von Filmen enthält (absteigend sortiert).
Parser.java	- Java-Klasse zum Einlesen von Filmdaten.
Film.java	- Java-Klasse, die Filme repräsentiert.
TestMySort.java	- Java-Klasse zum Testen der Implementierung von Aufgabe 1.
TestQuickSort.java	- Java-Klasse zum Testen der Implementierung von Aufgabe 2.

Das Interface `Comparable` besitzt eine Methode `int compareTo(Comparable other)`. Der zurückgelieferte `int`-Wert `value` hat folgende Bedeutung:[1]

1. `value > 0` : dieses Objekt ist größer als das Objekt `other`
2. `value = 0` : dieses Objekt ist gleich dem Objekt `other`
3. `value < 0` : dieses Objekt ist kleiner als das Objekt `other`.

Die Klasse `Film` implementiert das Interface `Comparable`. Der Rückgabewert der Methode `int compareTo(Comparable other)` ist durch die lexikographische Ordnung der Filmtitel bestimmt.

- Ist das übergebene Objekt kein Film-Objekt (`instanceof`), so soll die Rückgabe 0 sein.
- Für die Klasse `String` existiert bereits eine `compareTo`-Implementierung, die anhand der lexikographischen Ordnung entscheidet.

Aufgabe:

1. Implementieren Sie den Rumpf der Methode `int compareTo(Comparable other)` der Klasse `Film`.
2. Erstellen, implementieren und kommentieren Sie eine Klasse `MySort`. Die Klasse soll die statische Methode

```
public static void mySort(Comparable[] f)
```

[1] In dem ersten Tutoriumsblatt haben wir das Prinzip des Interface *Comparable* schon einmal behandelt. Sollten hier also Verständnisprobleme auftreten, ist es ratsam, sich die Tutoriumsaufzeichnungen noch einmal anzuschauen.

besitzen, die das im Folgenden beschriebene Sortierverfahren realisiert.

- Das Array wird in einen bereits sortierten Teil (vorne, am Anfang leer) und einen noch nicht sortierten Teil (hinten, am Anfang das komplette Array) unterteilt.
- Durchlaufe nun den noch nicht sortierten Teil von vorne bis hinten und merke die Position des kleinsten Elements.
- Tausche das gefundene kleinste Element mit dem ersten Element des unsortierten Bereichs.
- Der sortierte Bereich ist nun ein Element größer als vorher, der unsortierte Bereich ein Element kleiner.
- Wiederhole, bis der noch nicht sortierte Teil leer ist.
- Das Sortierverfahren soll zusätzlich die Anzahl der ausgeführten Aufrufe der Methode `compareTo` und die Anzahl der vorgenommenen Vertauschungen auf der Konsole ausgeben.

3. Beantworten Sie die folgenden Fragen:
   a) Unter welchem Namen ist dieser Sortieralgorithmus gemeinhin bekannt?
   b) Ist dieser Sortieralgorithmus stabil? Warum/warum nicht?
4. Testen Sie Ihr Programm mit Hilfe der Klasse `TestMySort.java`. Gehen Sie dabei wie folgt vor:
   a) Sortieren Sie die Filme (jeweils für jede der drei Dateien filme1.dat filme2.dat und filme3.dat) anhand ihrer Titel aufsteigend.
   b) Geben Sie nach jeder Sortierung alle Filme mit Hilfe der Methode `toString()` der Klasse `Film` auf der Konsole aus.
   c) Beantworten Sie folgende Frage: Bei welchem vorliegenden Array (unsortiert, aufsteigend sortiert, absteigend sortiert) ist MySort am schnellsten und bei welchem am langsamsten? Begründen Sie Ihre Antwort.

Musterlösung:

```
1  public class Film implements Comparable {
2
3     // Attribute
4     public String titel;
5     public double preis; // in EUR
6     public int laenge; // in min
7     public String beschreibung;
8     public String erscheinungsdatum; // ISO-8601 (JJJJ-MM-TT)
9
10    // Konstruktor
11    public Film(String titel, double preis, int laenge, String
          beschreibung, String erscheinungsdatum) {
12       this.titel = titel;
13       this.preis = preis;
14       this.laenge = laenge;
15       this.beschreibung = beschreibung;
16       this.erscheinungsdatum = erscheinungsdatum;
17    }
18
19    // Methoden
20    public String toString() {
21       return "Film={" +
22             "titel=" + this.titel + "," +
23             "preis=" + this.preis + "," +
```

```
24              "laenge=" + this.laenge + "," +
25              "beschreibung=" + this.beschreibung + "," +
26              "erscheinungsdatum=" + this.erscheinungsdatum +
27              "}";
28      }
29
30      public int compareTo(Comparable other) {
31          if (!(other instanceof Film)) return 0;
32
33          Film o = (Film)other; // Nur Filme haben einen titel, Comparable-
                Instanzen nicht unbedingt -> casten
34          int rc;
35          if (this.titel.compareTo(o.titel) > 0) rc = 1;
36          else if (this.titel.compareTo(o.titel) < 0) rc = -1;
37          else rc = 0;
38          return rc;
39      }
40
41  }
```

```
1   /**
2     * Implementiert einen Selection-Sort-Algorithmus.
3     * Ist instabil... die Reihenfolge von Elementen mit gleichem
          Schluessel bleibt nicht erhalten.
4     */
5   public class MySort {
6
7       /**
8        * @param f : Array von Objekten, die Comparable implementieren
9        *
10       * Diese Methode sortiert das Array f mit der Selection-Sort
11       * Methode und gibt die Anzahl der Vertauschungen und Aufrufe
12       * der Methode compareTo() aus.
13       */
14
15      public static void mySort(Comparable[] f) {
16          int swapCounter = 0;
17          int compareCounter=0;
18
19          for (int unsortedStart = 0; unsortedStart < f.length;
                unsortedStart++) {
20
21              // finde das kleinste Element um unsortierten Teil des Arrays
22              int lowestPos = unsortedStart;
23              for (int j = unsortedStart; j < f.length; j++) {
24                  if (f[j].compareTo(f[lowestPos]) < 0) {
25                      lowestPos = j;
26                      compareCounter++;
27                  }
28              }
29
30              // tausche es ans Ende des sorterten Teils
31              Comparable tmp = f[unsortedStart];
32              f[unsortedStart] = f[lowestPos];
33              f[lowestPos] = tmp;
34              swapCounter++;
35
```

```
36        }
37
38        System.out.println("Aufrufe von compareTo(): " + compareCounter);
39        System.out.println("Anzahl Vertauschungen: " + swapCounter);
40    }
41
42 }
```

```
1  public class TestMySort {
2
3     public static void main(String[] args) {
4
5        // Einlesen der Filme
6        String file1 = "filme1.dat";
7        String file2 = "filme2.dat";
8        String file3 = "filme3.dat";
9        Film[] f1 = Parser.readFilme(file1);
10       Film[] f2 = Parser.readFilme(file2);
11       Film[] f3 = Parser.readFilme(file3);
12
13       // MySort
14       System.out.println("MySort, filme1.dat");
15       System.out.println("---------------");
16       MySort.mySort(f1);
17       for (int i=0; i<f1.length;i++){
18          System.out.println(f1[i].toString());
19       }
20       System.out.println();
21
22       System.out.println("MySort, filme2.dat");
23       System.out.println("---------------");
24       MySort.mySort(f2);
25       for (int i=0; i<f2.length;i++){
26          System.out.println(f2[i].toString());
27       }
28       System.out.println();
29
30       System.out.println("MySort, filme3.dat");
31       System.out.println("---------------");
32       MySort.mySort(f3);
33       for (int i=0; i<f3.length;i++){
34          System.out.println(f3[i].toString());
35       }
36       System.out.println();
37
38    }
39 }
```

- Es handelt sich bei MySort um einen Selection-Sort-Suchalgorithmus.
- Im Allgemeinen ist Selection-Sort instabil. Es gibt Möglichkeiten, Selection-Sort auch stabil zu implementieren – dabei geht man jedoch über den eigentlichen Selection-Sort-Algorithmus hinaus. Die Nichtstabilität wird am besten mit einem Gegenbeispiel demonstriert.

  Beispiel:

  Ausgangsarray (Sortiert nach Preis): (Banane, 0.49), (Birne, 0.99), (Apfel, 1.15), (Kiwi,

1.55)

Endarray (nach Sortierung nach **Anfangs**buchstabe): (Apfel, 1.15), (Birne, 0.99), (Banane, 0.49), (Kiwi, 1.55)

Es wurde im ersten Durchlauf Apfel als kleinstes Element mit Banane getauscht. Danach sind keine weiteren Tauschungen nötig, um nach dem Anfangsbuchstaben zu sortieren. Das Zweitordnungskriterium Preis wurde für Birne und Banane verletzt.

- Für welches Array ist MySort/Selection-Sort am schnellsten/langsamsten?:

  MySort/Selection-Sort vergleicht immer alle Elemente des bislang unsortierten Arrayparts paarweise. Von daher ist die Laufzeit für alle Fälle (also unsortiertes Array, aufsteigend sortiertes Array und absteigend sortiertes Array als Eingabe) annähernd gleich.

  (Anders dagegen verhält sich z.B. Insertion-Sort. Am schnellsten ist der Algorithmus für das bereits aufsteigend sortierte Array, da kein Element vertauscht werden muss und die while-Schleife (siehe Lösung) am schnellsten beendet wird. Am langsamsten ist der Algorithmus für das absteigend sortierte Array, da hier jedes Element bis ganz nach links verschoben werden muss (while-Schleife wird am häufigsten durchlaufen).)

Aufgabe 2: Quicksort — 5 Punkte

Für die Bearbeitung dieser Aufgabe verwenden ebenfalls Sie die in Aufgabe 1 genannten Vorgabedateien.

Erstellen Sie die Klasse `SortierenQuicksort`. Sollten Sie Aufgabe 1 noch nicht bearbeitet haben, implementieren Sie zuerst den Rumpf der Methode `int compareTo(Comparable other)` der Klasse `Film`, siehe Aufgabe 1.

Die Klasse `SortierenQuicksort` soll die statische Methode

```
public static void quicksort(Comparable[] f)
```

besitzen, die das Sortierverfahren Quicksort realisiert. Das Sortierverfahren soll zusätzlich die Anzahl der ausgeführten Aufrufe der Methode `compareTo` und die Anzahl der vorgenommenen Vertauschungen auf der Konsole ausgeben. Implementieren Sie den Algorithmus in der **In-Situ**-Variante (siehe Tutorium 4).

Hinweis: Das Zählen der Methodenaufrufe ist bei der Implementierung des Quicksort-Algorithmus vielleicht einfacher, wenn Sie dafür statische Klassenattribute verwenden.

Die Ermittlung des Indexes des Pivotelements im [Teil-]Array soll dabei zufällig erfolgen. Zur Erinnerung: Eine natürliche Zufallszahl zwischen 0 und n kann unter Verwendung der Zufallsfunktion der Standardbibliotheken wie folgt erzeugt werden: `int zufallszahl = (int)(Math.random() * n);`

Testen Sie Ihr Programm mit Hilfe der Klasse `TestQuicksort.java`. Gehen Sie dabei wie folgt vor:

1. Sortieren Sie die Filme (jeweils für jede der drei Dateien filme1.dat, filme2.dat und filme3.dat) nach aufsteigender Reihenfolge ihrer Titel (lexikographische Sortierung).
2. Geben Sie nach jeder Sortierung die Filme mit Hilfe der Methode `toString()` der Klasse `Film` auf der Konsole aus.
3. Beantworten Sie folgende Frage: Bei welchem vorliegenden Array (unsortiert, aufsteigend sortiert, absteigend sortiert) ist der Quicksort-Algorithmus am schnellsten bzw. am langsamsten? Nennen Sie die Gründe dafür.

Musterlösung:

```java
public class Film implements Comparable {

   // Attribute
   public String titel;
   public double preis; // in EUR
   public int laenge; // in min
   public String beschreibung;
   public String erscheinungsdatum; // ISO-8601 (JJJJ-MM-TT)

   // Konstruktor
   public Film(String titel, double preis, int laenge, String
       beschreibung, String erscheinungsdatum) {
      this.titel = titel;
      this.preis = preis;
      this.laenge = laenge;
      this.beschreibung = beschreibung;
      this.erscheinungsdatum = erscheinungsdatum;
   }

   // Methoden
   public String toString() {
      return "Film={" +
            "titel=" + this.titel + "," +
            "preis=" + this.preis + "," +
            "laenge=" + this.laenge + "," +
            "beschreibung=" + this.beschreibung + "," +
            "erscheinungsdatum=" + this.erscheinungsdatum +
            "}";
   }

   public int compareTo(Comparable other) {
      if (!(other instanceof Film)) return 0;

      Film o = (Film)other; // Nur Filme haben einen titel, Comparable-
          Instanzen nicht unbedingt -> casten
      int rc;
      if (this.titel.compareTo(o.titel) > 0) rc = 1;
      else if (this.titel.compareTo(o.titel) < 0) rc = -1;
      else rc = 0;
      return rc;
   }

}
```

```java
public class SortierenQuicksort {

   private static int swapCounter;
   private static int compareCounter;

   /**
    * @param f : Array von Objekten, die Comparable implementieren
    *
    * Diese Methode sortiert das Array f mit der Quick-Sort
    * Methode und gibt die Anzahl der Vertauschungen und Aufrufe
    * der Methode compareTo() aus.
    */
```

```
13    public static void quickSort(Comparable[] f) {
14       swapCounter = 0;
15       compareCounter = 0;
16
17       qsort(f, 0, f.length-1);
18
19       System.out.println("Aufrufe von compareTo(): " + compareCounter);
20       System.out.println("Anzahl Vertauschungen: " + swapCounter);
21    }
22
23    /**
24     * @param f : Array von Objekten, die Comparable implementieren
25     *
26     * Diese Methode sortiert das Array f mit der Quick-Sort
27     * Methode und gibt die Anzahl der Vertauschungen und Aufrufe
28     * der Methode compareTo() aus.
29     */
30    private static void qsort(Comparable[] f, int lo, int hi) {
31       if (hi <= lo) return;
32       int j = teilen(f, lo, hi);
33       qsort(f, lo, j-1);
34       qsort(f, j+1, hi);
35    }
36
37    /**
38     *
39     * @param f : Array von Objekten, die Comparable implementieren
40     * @param lo : Erster Index
41     * @param hi : Zweiter Index
42     *
43     * Teilt Arraybereich zwischen lo und hi in zwei Teile.
44     * Der erste Teil enthaelt alle Elemente die kleiner als ein
          Pivotelement
45     * sind, der zweite alle die groesser sind. Die Methode gibt den
          Index des
46     * Pivotelements nach dem Teilen zurueck.
47     */
48    private static int teilen(Comparable[] f, int lo, int hi) {
49       int g = lo;
50       int pivotIndex = getPivotIndex(f, lo, hi);
51       Comparable pivot = f[pivotIndex];
52
53       swap(f, pivotIndex, hi);
54       swapCounter++;
55
56       for (int i = lo; i < hi; i++) {
57          if(f[i].compareTo(pivot) < 0) {
58             swap(f,g,i);
59             g++;
60             swapCounter++;
61          }
62          compareCounter++;
63       }
64
65       swap(f, g, hi);
66       swapCounter++;
67       return g;
```

```
68      }
69
70      /**
71       *
72       * @param f : Array von Objekten, die Comparable implementieren
73       * @param a : Erster Index
74       * @param b : Zweiter Index
75       *
76       * Vertauscht zwei Elemente eines uebergebenen Arrays.
77       */
78      private static void swap(Comparable[] f, int a, int b) {
79          Comparable tmp;
80          tmp = f[a];
81          f[a] = f[b];
82          f[b] = tmp;
83      }
84
85      /**
86       *
87       * @param f : Array von Objekten, die Comparable implementieren
88       * @param a : Erster Index des Intervalls
89       * @param b : Zweiter Index des Intervalls
90       *
91       * Ermittelt den Index des Pivot-Elements.
92       */
93      private static int getPivotIndex(Comparable[] f, int a, int b) {
94          int anzElemente = b - a + 1; // z.B. b=9, a=0, also 10 Elemente
95          int zufallszahl = (int)(Math.random() * anzElemente); //
                Zufallszahl zwischen 0 und anzElemente
96          return a + zufallszahl; // Ergebnis-Index = Startindex +
                Zufallszahl
97      }
98
99  }
```

```
1   public class TestQuicksort {
2
3       public static void main(String[] args) {
4
5           // Einlesen der Filme
6           String file1 = "filme1.dat";
7           String file2 = "filme2.dat";
8           String file3 = "filme3.dat";
9           Film[] f1 = Parser.readFilme(file1);
10          Film[] f2 = Parser.readFilme(file2);
11          Film[] f3 = Parser.readFilme(file3);
12
13          // Quick-Sort
14          System.out.println("Quick-Sort, filme1.dat");
15          System.out.println("---------------");
16          SortierenQuicksort.quickSort(f1);
17
18          for (int i = 0; i < f1.length; i++){
19              System.out.println(f1[i].toString());
20          }
21          System.out.println();
22
```

```
23          System.out.println("Quick-Sort, filme2.dat");
24          System.out.println("---------------");
25          SortierenQuicksort.quickSort(f2);
26
27          for (int i = 0; i < f2.length; i++){
28             System.out.println(f2[i].toString());
29          }
30          System.out.println();
31
32          System.out.println("Quick-Sort, filme3.dat");
33          System.out.println("---------------");
34          SortierenQuicksort.quickSort(f3);
35
36          for (int i= 0; i < f3.length; i++){
37             System.out.println(f3[i].toString());
38          }
39          System.out.println();
40       }
41
42    }
```

Die Laufzeit von Quicksort hängt größtenteils von der Wahl des Pivoelements ab. Am schnellsten wird Quisksort, wenn das Pivotelement immer so gewählt wird, dass die Teilarrays möglichst die gleiche Größe haben.

Best-Case: Wenn das Array sortiert ist und das Pivotelement in der Mitte gewählt wird, werden die Teilarrays die gleiche Größe haben. Komplexität $O(nlogn)$

Average-Case: Wenn das Array unsortiert ist (truly random, wie filme1), sind alle drei Schemen gleich, da keine regelmäßige Aufteilung entsteht. Komplexität $O(nlogn)$

Worst-Case: Wenn das Array sortiert ist und als Pivotelement das letzte oder das erste Element gewählt wird, wird immer ein Teilarray leer sein. Diese Aufteilung ist am schlechtesten und sorgt für eine große Anzahl an Vergleichen. Komplexität $O(n^2)$

Wichtig ist, dass es kein optimales Pivotelement gibt und die Geschwindigkeit des Algorithmus stark vom Array abhängt. Wenn beispielsweise in der Mitte des Arrays das kleinste Element ist, wäre die Wahl des mittleren Elements als Pivotelement nicht so effektiv. In der Praxis arbeitet man daher oft mit der Wahl von zufälligen Elementen oder eines Medians.