1 Hammingfolge

1.1 Lösungsidee

Folgend ist die Lösungsidee für die Aufgabenstellung Hammingfolge berechnen angeführt.
Da es sich hierbei lediglich um einen einzigen Algorithmus handelt soll dieser als Klassenmethode implementiert werden. Das diese Klasse lediglich diese Klasenmethode enthalten soll, soll in dieser Klasse ein Privater Konstruktor implementiert werden um zu verhindern, dass diese Klasse instanziert werden kann.

Da eine Hammingfolge wie folgt definiert ist:
$1 \in H$
$x \in H \Rightarrow 2 * x \in H \wedge 3 * x \in H \wedge 5 * x \in H$
wissen wir dass folgende Elemente Aufgrund dessen das $1 \in H$ gilt in der Folge vorhanden sind.
$1 \in H \wedge 2 \in H \wedge 3 \in H \wedge 5 \in H$
daher können wir einen Algorithmus definieren der sich wie folgt verhalten soll:

1. Erstelle eine Liste und initialisiere diese Liste mit dem Element 1
2. Berechne die nachfolgenden Hammingzahlen $(2 * list.get(i) \wedge 3 * list.get(i) \wedge 5 * list.get(i))$ für das Element am Index i
   - Ist die Zahl vorhanden: Dann füge sie nicht der Liste hinzu
   - Ist die Zahl nicht vorhanden: Dann füge sie der Liste hinzu
3. Wiederhole Schritt 2 solange folgendes gilt: $list.size(i) < (n + 4)$

Nun stellt sich die Frage warum folgende Schleifenbedingung gilt $list.size(i) < (n + 4)$
Dies ist erforderlich da ansonsten nicht die Folge von Hammingzahlen bis zur Schranke n (=Anzahl) berechnet würde. Es würden zwar gültige Hammingzahlen in der Liste vorhanden sein, jedoch würden bei einer berechneten Hammingfolge die letzten berechneten Zahlen nicht die letzten Hammingzahlen der berechneten Folge sein.

Im folgenden Beispiel wird Problematik genau erläutert:
Sei $n = 10$ daraus folgt $H_{10} = \{1, 2, 3, 4, 5, 6, 8, 9, 10, 12\}$
Mit dem oben beschriebenen Algorithmus würde nun folgende passieren, wobei die Aufzählungen dem Index $i = (x - 1)$ entsprechen

1. $1 * 2 = 2 \wedge 1 * 3 = 3 \wedge 1 * 5 = 5$
   $H = \{1, 2, 3, 5\}$
2. $2 * 2 = 4 \wedge 2 * 3 = 6 \wedge 2 * 5 = 10$
   $H = \{1, 2, 3, 5, 4, 6, 10\}$
3. $3 * 2 = 6 \wedge 3 * 3 = 9 \wedge 3 * 5 = 15$
   $H = \{1, 2, 3, 5, 4, 6, 10, 9, 15\}$
4. $5 * 2 = 10 \wedge 5 * 3 = 15 \wedge 5 * 5 = 25$
   $H = \{1, 2, 3, 5, 4, 6, 10, 9, 15, 25\}$

Nun stop der Algorithmus, da wir bereits die definierte Schranke erreicht haben.
Wenn wir die Liste sortieren dann erhalten wir folgende Folge $H = \{1, 2, 3, 4, 5, 6, 9, 10, 15, 25\}$. Diese Liste entspricht aber nicht der zu erwartenden Liste.
Warum ist das so ?
Das Problem liegt darin, dass wir zwar gültige Hammingzahlen berechnen aber diese unsortiert in der

Liste vorkommen und wir daher Hammingzahlen berechnen, die in der zu erwartenden Folge nicht vorkommen dürfen.
Daher müssen wir die Anzahl erhöhen um sicherzustellen das in der Liste alle zu erwartenden Hammingzahlen vorhanden sind.
Die Zahlen die zu viel vorhanden sind können einfach am Ende des Algorithmus wieder entfernt werden wobei hier von $i = (list.size() - 1) \rightarrow i >= count$ iteriert wird und die Elemente, die zu viel sind wieder entfernt werden.

Wir könnten ein Set verwenden, was uns das Problem mit den Duplikaten ersparen würde, hätten dann aber das Problem, dass wir während der Iteration über das Set keine Elemente hinzufügen können so wie bei der Liste. Ebenso könnte es der Performance schaden das Set bei jedem add zu sortieren (TreeSet).
Die Verwendung von list.contains(obj) mag vielleicht nicht gerade perfekt sein jedoch sollte es kein Problem sein 10.000 Elemente der Hammingfolge unter 1 Sekunde zu ermitteln.

Die verwendete ArrayList muss auf jeden Fall mit einer Kapazität $capicity = (n + 4)$ initialisiert werden, da wir ansonsten das Problem haben das die ArrayList ein Array.copyOf(...) durchführen würde und dynamisch um den Faktor $1.5 * oldCapicity$ wachsen würde, was einerseits ein mehrmaliges kopieren des Arrays und andererseits schlussendlich ein Array produzieren würde, welches weitaus größer ist als benötigt.
Da wir aber ohnehin wissen wie viele Elemente berechnet werden müssen kann dies über ein korrektes setzen der Kapazität vermieden werden.

1.2 Source-Code

Folgend ist der Implementierte Source und Test-Source angeführt.

1.3 Hamming.java

../src/main/java/at/fhooe/swe4/lab3/hamming/Hamming.java

```
1  package at.fhooe.swe4.lab3.hamming;
2
3  import java.math.BigInteger;
4  import java.util.ArrayList;
5  import java.util.Collections;
6  import java.util.List;
7
8  /**
9   * Provides class methods which are used to handle hamming numbers.
10  *
11  * @author Thomas Herzog
12  *
13  */
14 public class Hamming {
15
16   /**
17    * Not meant to be instantiated
18    */
19   private Hamming() {
20     super();
21   }
22
23   /**
24    * Calculates the hamming numbers to the given count.
25    *
26    * @param count
```

```
27     *            the count of hamming numbers to calculate
28     * @return the sorted list containing the hamming numbers
29     * @throws IllegalArgumentException
30     *             if count <= 0
31     */
32    public static List<BigInteger> calulcateHammingNumbers(final int count) {
33      // At least one is in the hamming list
34      if (count <= 1) {
35        throw new IllegalArgumentException("The count must be at least one !!!");
36      }
37      final List<BigInteger> list = new ArrayList<BigInteger>((count + 4));
38      list.add(BigInteger.ONE);
39      // The allowed factors
40      final BigInteger second = BigInteger.valueOf(2);
41      final BigInteger three = BigInteger.valueOf(3);
42      final BigInteger five = BigInteger.valueOf(5);
43
44      // As long as all of the intended numbers have been calculated
45      // Calculate for 4 more elements because otherwise some numbers would be
46      // missing
47      for (int i = 0; (list.size() < (count + 4)); i++) {
48        BigInteger secondMult = list.get(i).multiply(second);
49        BigInteger threeMult = list.get(i).multiply(three);
50        BigInteger fiveMult = list.get(i).multiply(five);
51        // Avoid duplicates of 2 * x
52        if (!list.contains(secondMult)) {
53          list.add(secondMult);
54        }
55        // Avoid duplicates of 3 * x
56        if (!list.contains(threeMult)) {
57          list.add(threeMult);
58        }
59        // Avoid duplicates of 5 * x
60        if (!list.contains(fiveMult)) {
61          list.add(fiveMult);
62        }
63      }
64      Collections.sort(list);
65      // Remove the elements which are to much
66      for (int i = (list.size() - 1); i >= count; i--) {
67        list.remove(list.get(i));
68      }
69      return list;
70    }
71
72  }
```

../src/test/java/at/fhooe/swe4/lab3/test/hamming/HammingTest.java

```
1  package at.fhooe.swe4.lab3.test.hamming;
2
3  import java.math.BigInteger;
4  import java.util.List;
5
6  import org.junit.Test;
7  import org.junit.runner.RunWith;
8  import org.junit.runners.JUnit4;
9
10 import at.fhooe.swe4.lab3.hamming.Hamming;
11
12 /**
13  * This is the test for the calculating of the hamming numbers.
14  *
15  * @author Thomas Herzog
```

```
16  *
17  */
18 @RunWith(JUnit4.class)
19 public class HammingTest {
20
21   @Test(expected = IllegalArgumentException.class)
22   public void test_invalid_count_negativ() {
23     Hamming.calulcateHammingNumbers(-1);
24   }
25
26   @Test(expected = IllegalArgumentException.class)
27   public void test_invalid_count_zeor() {
28     Hamming.calulcateHammingNumbers(0);
29   }
30
31   @Test
32   public void test_caluclation() {
33     int count = 1;
34     final int factor = 10;
35     final int repeation = 1;
36     for (int i = 0; i < repeation; i++) {
37       count *= factor;
38       final long startMillis = System.currentTimeMillis();
39       final List<BigInteger> result = Hamming.calulcateHammingNumbers(count);
40       final long diffMillis = System.currentTimeMillis() - startMillis;
41       System.out.println("
    ------------------------------------------------------------------------");
42       System.out.println(new StringBuilder("Spend time in millis: '").append(diffMillis
    ).append("' for '").append(count)
43           .append("' hamming numbers (including sorting)").toString());
44       System.out.println("
    ------------------------------------------------------------------------");
45       for (int j = 0; j < result.size(); j++) {
46         System.out.println(new StringBuilder().append(j + 1).append(": ").append(result
    .get(j)).toString());
47       }
48       if (result.size() > 10) {
49         System.out.println("...");
50         System.out.println(new StringBuilder().append(result.size()).append(": ").
    append(result.get(result.size() - 1)).toString());
51       }
52       System.out.println("
    ------------------------------------------------------------------------");
53     }
54   }
55 }
```

1.4 Tests

Folgend sind die Tests der Aufgabenstellung Hammingfolge angeführt.

2 Hammingfolge

2.1 Lösungsidee

2.2 Source Code

../src/main/java/at/fhooe/swe4/lab3/hamming/Hamming.java

```java
package at.fhooe.swe4.lab3.hamming;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Provides class methods which are used to handle hamming numbers.
 *
 * @author Thomas Herzog
 *
 */
public class Hamming {

  /**
   * Not meant to be instantiated
   */
  private Hamming() {
    super();
  }

  /**
   * Calculates the hamming numbers to the given count.
   *
   * @param count
   *            the count of hamming numbers to calculate
   * @return the sorted list containing the hamming numbers
   * @throws IllegalArgumentException
   *             if count <= 0
   */
  public static List<BigInteger> calulcateHammingNumbers(final int count) {
    // At least one is in the hamming list
    if (count <= 1) {
      throw new IllegalArgumentException("The count must be at least one !!!");
    }
    final List<BigInteger> list = new ArrayList<BigInteger>((count + 4));
    list.add(BigInteger.ONE);
    // The allowed factors
    final BigInteger second = BigInteger.valueOf(2);
    final BigInteger three = BigInteger.valueOf(3);
    final BigInteger five = BigInteger.valueOf(5);

    // As long as all of the intended numbers have been calculated
    // Calculate for 4 more elements because otherwise some numbers would be
    // missing
    for (int i = 0; (list.size() < (count + 4)); i++) {
      BigInteger secondMult = list.get(i).multiply(second);
      BigInteger threeMult = list.get(i).multiply(three);
      BigInteger fiveMult = list.get(i).multiply(five);
      // Avoid duplicates of 2 * x
      if (!list.contains(secondMult)) {
        list.add(secondMult);
      }
      // Avoid duplicates of 3 * x
      if (!list.contains(threeMult)) {
        list.add(threeMult);
      }
      // Avoid duplicates of 5 * x
      if (!list.contains(fiveMult)) {
        list.add(fiveMult);
      }
    }
```

```
64      Collections.sort(list);
65      // Remove the elements which are to much
66      for (int i = (list.size() - 1); i >= count; i--) {
67        list.remove(list.get(i));
68      }
69      return list;
70    }
71
72  }
```

../src/main/java/at/fhooe/swe4/lab3/sort/api/Sorter.java

```
1   package at.fhooe.swe4.lab3.sort.api;
2
3   import java.util.List;
4
5   import at.fhooe.swe4.lab3.sort.api.Heap.HeapType;
6   import at.fhooe.swe4.lab3.stat.StatisticsProvider;
7
8   /**
9    * This interface specifies the sorter functionalities.
10   *
11   * @author Thomas Herzog
12   *
13   * @param <V>
14   *            the values type of the collections or array elements
15   */
16  public interface Sorter<V extends Comparable<V>> {
17    /**
18     * This enumeration specifies the sort order for a heap sort instance.
19     *
20     * @author Thomas Herzog
21     *
22     */
23    public static enum SortType {
24      /**
25       * Will result in an ascending ordered result
26       */
27      DESCENDING,
28      /**
29       * Will result in an descending ordered result
30       */
31      ASCENDING;
32
33      /**
34       * Compares the two comparable instances.
35       * <ul>
36       * <li>
37       * {@link SortType#DESCENDING} performs an x < 0 comparision</li>
38       * <li>{@link SortType#ASCENDING} performs an x > 0 comparision</li>
39       * </ul>
40       *
41       * @param left
42       *            the instance which invokes the comparesTo method
43       * @param right
44       *            the parameter for lefts compareTomethod invocation
45       * @return the proper result for the specified heap type
46       */
47      public <T extends Comparable<T>> boolean compare(T left, T right) {
48        switch (this) {
49        case DESCENDING:
50          return left.compareTo(right) > 0;
51        case ASCENDING:
52          return left.compareTo(right) <= 0;
```

```
53          default:
54            throw new IllegalStateException("This enum is not handled here but should. enum
    =" + this.name());
55          }
56       }
57    }
58
59    /**
60     * Sorts the given array.
61     *
62     * @param array
63     *            the array to be sorted
64     * @param sorterType
65     *            the type of the sorting
66     * @return the sorted array
67     * @see SortType
68     * @throws IllegalArgumentException
69     *             if the array is null, or the {@link SortType} is null
70     */
71    public V[] sort(V[] array, SortType sorterType);
72
73    /**
74     * Sorts the given list5.
75     *
76     * @param list
77     *            the list to be sorted
78     * @param sorterType
79     *            the type of the sorting
80     * @return the sorted array
81     * @see SortType
82     * @throws IllegalArgumentException
83     *             if the list is null, or the {@link SortType} is null
84     */
85    public List<V> sort(List<V> list, SortType sorterType);
86
87    /**
88     * Gets the statistics of the current instance
89     *
90     * @return the current statistics
91     */
92    public StatisticsProvider getStatisitcs();
93 }
```

../src/main/java/at/fhooe/swe4/lab3/sort/api/Heap.java

```
1  package at.fhooe.swe4.lab3.sort.api;
2
3  import java.util.Collection;
4  import java.util.List;
5
6  import at.fhooe.swe4.lab3.stat.StatisticsProvider;
7
8  /**
9   * This interface specifies the heap functionalities.
10  *
11  * @author Thomas Herzog
12  *
13  * @param <V>
14  *            the value type of the elements in the heap
15  */
16 public interface Heap<V extends Comparable<V>> {
17
18    /**
19     * This enumeration specifies the supported heap types
```

```
20      *
21      * @author Thomas Herog
22      *
23      */
24     public static enum HeapType {
25       /**
26        * WIll result an ascending ordered heap
27        */
28       MAX_HEAP, /**
29        * WIll result an descending ordered heap
30        */
31       MIN_HEAP;
32
33       /**
34        * Compares the two comparable instances.
35        * <ul>
36        * <li>
37        * {@link HeapType#MIN_HEAP} performs an x < 0 comparision</li>
38        * <li>{@link HeapType#MIN_HEAP} performs an x > 0 comparision</li>
39        * </ul>
40        *
41        * @param left
42        *            the instance which invokes the comparesTo method
43        * @param right
44        *            the parameter for lefts compareTomethod invocation
45        * @return the proper result for the specified heap type
46        */
47       public <T extends Comparable<T>> boolean compare(T left, T right) {
48         switch (this) {
49         case MAX_HEAP:
50           return left.compareTo(right) < 0;
51         case MIN_HEAP:
52           return left.compareTo(right) > 0;
53         default:
54           throw new IllegalStateException("This enum is not handled here but should. enum
       =" + this.name());
55         }
56       }
57     }
58
59     /**
60      * Initializes this heap with the given array of elements.
61      *
62      * @param originalArrayValues
63      *            the values to build an heap structure from
64      * @param sortType
65      *            the type of how the elements should be
66      */
67     public void init(V[] originalArrayValues, HeapType sortType);
68
69     /**
70      * Initializes this heap with the given collection which provides the
71      * elements.
72      *
73      * @param originalArrayValues
74      *            the values to build an heap structure from
75      * @param sortType
76      *            the type of how the elements should be
77      */
78     public void init(Collection<V> originalIterableValues, HeapType sortType);
79
80     /**
81      * Puts an element on the heap and keeps heap type specified order.
```

```
82      *
83      * @param value
84      *            the element to be put on the heap
85      */
86     public void enqueue(V value);
87
88     /**
89      * Gets the top element of the heap
90      *
91      * @return the top element
92      */
93     public V dequeue();
94
95     /**
96      * Converts the heap to a flat list which represents the backed tree
97      * structure.
98      *
99      * @return the list representing the heap. Will be a new instance
100     */
101    public List<V> toList();
102
103    /**
104     * Converts the heap to an flat array which represents the bakced trees
105     * structure
106     *
107     * @return the array representing the heap
108     */
109    public V[] toArray();
110
111    /**
112     * Answers the question if the heap has another element
113     *
114     * @return true if there is still an element left on the heap
115     */
116    public boolean hasNext();
117
118    /**
119     * Returns the current size of the heap.
120     *
121     * @return the heap element size
122     */
123    public int size();
124
125    /**
126     * Gets the statistics of the current instance
127     *
128     * @return the current statistics
129     */
130    public StatisticsProvider getStatisitcs();
131 }
```

../src/main/java/at/fhooe/swe4/lab3/sort/heap/impl/HeapArrayListImpl.java

```
1  package at.fhooe.swe4.lab3.sort.heap.impl;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Iterator;
6  import java.util.List;
7
8  import at.fhooe.swe4.lab3.sort.api.Heap;
9  import at.fhooe.swe4.lab3.stat.CodeStatistics;
10 import at.fhooe.swe4.lab3.stat.StatisticsProvider;
11 import at.fhooe.swe4.lab3.stat.DefaultStatisticsProviderImpl;
```

```
12
13 /**
14  * This is the ArrayList implementation of the heap.
15  *
16  * @author Thomas Herzog
17  *
18  * @param <V>
19  *            the value type of the heap managed elements
20  */
21 public class HeapArrayListImpl<V extends Comparable<V>> implements Heap<V> {
22
23   public HeapType heapType;
24   public List<V> container = new ArrayList<V>();
25
26   public StatisticsProvider statProvider = new DefaultStatisticsProviderImpl();
27
28   /**
29    * Empty constructor
30    */
31   public HeapArrayListImpl() {
32     super();
33   }
34
35   /**
36    * Initializes the heap with the given array
37    *
38    * @param array
39    *            the array providing the elements for the heap
40    * @param heapType
41    *            the type of the heap
42    * @see HeapType
43    */
44   public HeapArrayListImpl(final V[] array, final HeapType heapType) {
45     super();
46     init(array, heapType);
47   }
48
49   /**
50    * Initializes the heap with the given collection
51    *
52    * @param list
53    *            the collection providing the elements for the heap
54    * @param heapType
55    *            the type of the heap
56    * @see HeapType
57    */
58   public HeapArrayListImpl(final Collection<V> list, final HeapType heapType) {
59     super();
60     init(list, heapType);
61   }
62
63   @Override
64   public void init(final V[] originalArrayValues, final HeapType heapType) {
65     this.heapType = heapType;
66     int size = ((originalArrayValues == null) || (originalArrayValues.length == 0)) ? 0
       : originalArrayValues.length;
67     statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(
       " elements[").append(size).append("]").toString());
68     if (size > 0) {
69       container = new ArrayList<V>(size);
70       final CodeStatistics stat = statProvider.getCtx().newStatistic("init(array)");
71       for (V value : originalArrayValues) {
72         enqueue(value);
```

```
73        }
74      } else {
75        container = new ArrayList<V>(0);
76      }
77    }
78
79    @Override
80    public void init(final Collection<V> originalIterableValues, final HeapType heapType)
        {
81      this.heapType = heapType;
82      final int size = (originalIterableValues == null) ? 0 : originalIterableValues.size
        ();
83      statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(
        " elements[").append(size).append("]").toString());
84      if (size > 0) {
85        container = new ArrayList<V>(size);
86        final Iterator<V> it = originalIterableValues.iterator();
87        while (it.hasNext()) {
88          enqueue(it.next());
89        }
90      } else {
91        container = new ArrayList<V>(0);
92      }
93    }
94
95    @Override
96    public void enqueue(final V value) {
97      container.add(value);
98      upHeap(container);
99    }
100
101   @Override
102   public V dequeue() {
103     final V value = container.get(0);
104     container.set(0, container.get(container.size() - 1));
105     downHeap(container);
106     container.remove(container.size() - 1);
107     return value;
108   }
109
110   @Override
111   public boolean hasNext() {
112     return container.size() > 0;
113   }
114
115   @Override
116   public int size() {
117     return container.size();
118   }
119
120   @Override
121   public List<V> toList() {
122     return new ArrayList<V>(container);
123   }
124
125   @Override
126   @SuppressWarnings("unchecked")
127   public V[] toArray() {
128     return (V[]) container.toArray();
129   }
130
131   @Override
132   public StatisticsProvider getStatisitcs() {
```

```
133     return statProvider.endContext();
134   }
135
136   // Private heap methods
137   /**
138    * Performs an up heap on the given heap represented by the given list
139    *
140    * @param container
141    *            the list representing the heap
142    */
143   private void upHeap(final List<V> container) {
144     final CodeStatistics stat = statProvider.getCtx().byKey("upHeap()", Boolean.TRUE);
145
146     int i = container.size() - 1;
147     V tmp = container.get(i);
148     while ((i != 0) && (heapType.compare(container.get(parent(i)), tmp))) {
149       stat.incIf().incSwap();
150       container.set(i, container.get(parent(i)));
151       i = parent(i);
152     }
153     container.set(i, tmp);
154   }
155
156   /**
157    * Performs an down heap on the given heap represented by the given list
158    *
159    * @param container
160    *            the list representing the heap
161    */
162   private void downHeap(final List<V> container) {
163     final CodeStatistics stat = statProvider.getCtx().byKey("downHeap()", Boolean.TRUE)
       ;
164     int idx = 0;
165     int largeIdx;
166     V tmp = container.get(0);
167     while (idx < (container.size() / 2)) {
168       int leftIdx = left(idx);
169       int rightIdx = right(idx);
170       stat.incIf();
171       if ((rightIdx < container.size()) && (heapType.compare(container.get(leftIdx),
      container.get(rightIdx)))) {
172         largeIdx = rightIdx;
173       } else {
174         largeIdx = leftIdx;
175       }
176       stat.incIf();
177       if (!heapType.compare(tmp, container.get(largeIdx))) {
178         break;
179       }
180       stat.incSwap();
181       container.set(idx, container.get(largeIdx));
182       idx = largeIdx;
183     }
184     container.set(idx, tmp);
185   }
186
187   // Private helper
188   /**
189    * Gets the parent index of the element on index i
190    *
191    * @param i
192    *            the index to get its parent index
193    * @return the parent index
```

```
194     */
195   private static int parent(final int i) {
196     return (i - 1) / 2;
197   }
198
199   /**
200    * Gets the left neighbor index of the element on index i
201    *
202    * @param i
203    *            the index to get its left neighbor index
204    * @return the left neighbor index
205    */
206   private static int left(final int i) {
207     return (i * 2) + 1;
208   }
209
210   /**
211    * Gets the right neighbor index of the element on index i
212    *
213    * @param i
214    *            the index to get its right neighbor index
215    * @return the right neighbor index
216    */
217   private static int right(final int i) {
218     return (i * 2) + 2;
219   }
220
221   @Override
222   public String toString() {
223     final int new_line_count = 10;
224     final StringBuilder sb = new StringBuilder();
225     sb.append(this.getClass().getName()).append("[size=").append(container.size()).
      append("]\n");
226     sb.append("idx[0 - ").append(new_line_count).append("]: ");
227     for (int i = 0; i < container.size(); i++) {
228       sb.append(container.get(i));
229       if ((i + 1) < container.size()) {
230         sb.append(", ");
231       }
232       if ((i > 0) && (i % new_line_count == 0)) {
233         final int idxEnd = ((i + new_line_count) < container.size()) ? (i +
      new_line_count) : (container.size() - 1);
234         sb.append(System.getProperty("line.separator"));
235         sb.append("idx[").append(i + 1).append(" - ").append(idxEnd).append("]: ");
236       }
237     }
238     return sb.toString();
239   }
240 }
```

../src/main/java/at/fhooe/swe4/lab3/sort/heap/impl/HeapSorter.java

```
1  package at.fhooe.swe4.lab3.sort.heap.impl;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.List;
6
7  import at.fhooe.swe4.lab3.sort.api.Heap;
8  import at.fhooe.swe4.lab3.sort.api.Heap.HeapType;
9  import at.fhooe.swe4.lab3.sort.api.Sorter;
10 import at.fhooe.swe4.lab3.stat.StatisticsProvider;
11
12 /**
```

```
13  * This is the heap sorter implementation of the Sorter interface.
14  *
15  * @author Thomas Herzog
16  *
17  * @param <V>
18  *            the values type of the to sort array or collection managed
19  *            elements
20  */
21 public class HeapSorter<V extends Comparable<V>> implements Sorter<V> {
22
23   private final Heap<V> heap = new HeapArrayListImpl<V>();
24
25   public HeapSorter() {
26     super();
27   }
28
29   @SuppressWarnings("unchecked")
30   @Override
31   public V[] sort(final V[] array, final SortType sorterType) {
32     if (array == null) {
33       throw new IllegalArgumentException("Cannot sort empty array");
34     }
35     return (array.length == 0) ? array : ((V[]) sort(Arrays.asList(array), sorterType).
       toArray());
36   }
37
38   @Override
39   public List<V> sort(final List<V> list, final SortType sorterType) {
40     if (sorterType == null) {
41       throw new IllegalArgumentException("SorterType not defined");
42     }
43     if (list == null) {
44       throw new IllegalArgumentException("Cannot sort null list");
45     }
46     heap.init(list, convertToHeapType(sorterType));
47     final List<V> result = new ArrayList<V>();
48     while (heap.hasNext()) {
49       result.add(heap.dequeue());
50     }
51     return result;
52   }
53
54   @Override
55   public StatisticsProvider getStatisitcs() {
56     return heap.getStatisitcs();
57   }
58
59   /**
60    * Converts the sorter type to the corresponding heap type.
61    *
62    * @param sortType
63    *            the sorter type to be converted
64    * @return the corresponding heap type
65    * @throws IllegalArgumentException
66    *             if the sorter type cannot be mapped to a corresponding heap
67    *             type
68    */
69   private HeapType convertToHeapType(final SortType sortType) {
70     switch (sortType) {
71     case ASCENDING:
72       return HeapType.MAX_HEAP;
73     case DESCENDING:
74       return HeapType.MIN_HEAP;
```

```
75      default:
76        throw new IllegalArgumentException("SortType cannot bemapped to corresponding
        HeapType !!!");
77      }
78    }
79  }
```

../src/main/java/at/fhooe/swe4/lab3/sort/quick/QuickSorter.java

```
1  package at.fhooe.swe4.lab3.sort.quick;
2
3  import java.util.Arrays;
4  import java.util.Collections;
5  import java.util.List;
6
7  import at.fhooe.swe4.lab3.sort.api.Sorter;
8  import at.fhooe.swe4.lab3.stat.CodeStatistics;
9  import at.fhooe.swe4.lab3.stat.DefaultStatisticsProviderImpl;
10 import at.fhooe.swe4.lab3.stat.StatisticsProvider;
11
12 /**
13  * This is the Sorter implementation for the quicksort algorithm
14  *
15  * @author Thomas Herzog
16  *
17  * @param <V>
18  *            the values type of the to sort elements
19  */
20 public class QuickSorter<V extends Comparable<V>> implements Sorter<V> {
21
22   private final StatisticsProvider statProvider = new DefaultStatisticsProviderImpl();
23
24   public QuickSorter() {
25   }
26
27   @SuppressWarnings("unchecked")
28   @Override
29   public V[] sort(final V[] array, final SortType sorterType) {
30     if (array == null) {
31       throw new IllegalArgumentException("Cannot sort null array");
32     }
33     final List<V> result = sort(Arrays.asList(array), sorterType);
34     return (V[]) result.toArray();
35   }
36
37   @Override
38   public List<V> sort(List<V> list, SortType sorterType) {
39     if (sorterType == null) {
40       throw new IllegalArgumentException("SorterType not defined");
41     }
42     if (list == null) {
43       throw new IllegalArgumentException("Cannot sort null list");
44     }
45     statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(
       " elements[").append(list.size()).append("]").toString());
46     quicksort(list, 0, (list.size() - 1));
47     if (SortType.DESCENDING.equals(sorterType)) {
48       Collections.reverse(list);
49     }
50     return list;
51   }
52
53   /**
54    * Performs a quicksort in ascending order.
```

```
55      *
56      * @param values
57      *            the values to be sorted
58      * @param start
59      *            the start index
60      * @param end
61      *            the end index
62      */
63     private void quicksort(final List<V> values, final int start, final int end) {
64       final CodeStatistics stat = statProvider.getCtx().byKey("quicksort", Boolean.TRUE);
65       int i = start;
66       int k = end;
67
68       if ((end - start) >= 1) {
69         V pivot = values.get(start);
70         while (k > i) {
71           while ((values.get(i).compareTo(pivot) <= 0) && (i <= end) && (k > i)) {
72             stat.incIf();
73             i++;
74           }
75           while ((values.get(k).compareTo(pivot) > 0) && (k >= start) && (k >= i)) {
76             stat.incIf();
77             k--;
78           }
79           if (k > i) {
80             stat.incSwap();
81             swap(values, i, k);
82           }
83         }
84         stat.incSwap();
85         swap(values, start, k);
86         quicksort(values, start, k - 1);
87         quicksort(values, k + 1, end);
88       }
89     }
90
91     /**
92      * Swaps the elements at the indexes
93      *
94      * @param values
95      *            the array list where to swap elements
96      * @param i
97      *            the first index
98      * @param j
99      *            the second index
100     */
101    private void swap(final List<V> values, final int i, final int j) {
102      final V tmp = values.get(i);
103      values.set(i, values.get(j));
104      values.set(j, tmp);
105    }
106
107    @Override
108    public StatisticsProvider getStatisitcs() {
109      return statProvider.endContext();
110    }
111  }
```

Input	Output	Comment
a = 100 b = 200	3	