# 1 Übung 1.1

## 1.1 Solution Idea

This is my idea

## 1.2 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>();
    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() - 4) < count); 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)) {
```

```
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/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

```java
package at.fhooe.swe4.lab3.sort.heap.impl;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

import at.fhooe.swe4.lab3.sort.api.Heap;
import at.fhooe.swe4.lab3.stat.CodeStatistics;
import at.fhooe.swe4.lab3.stat.StatisticsProvider;
import at.fhooe.swe4.lab3.stat.DefaultStatisticsProviderImpl;

/**
 * This is the ArrayList implementation of the heap.
 *
 * @author Thomas Herzog
 *
 * @param <V>
 *            the value type of the heap managed elements
 */
public class HeapArrayListImpl<V extends Comparable<V>> implements Heap<V> {

  public HeapType heapType;
  public List<V> container = new ArrayList<V>();

  public StatisticsProvider statProvider = new DefaultStatisticsProviderImpl();

  /**
   * Empty constructor
   */
  public HeapArrayListImpl() {
    super();
  }

  /**
   * Initializes the heap with the given array
   *
   * @param array
   *            the array providing the elements for the heap
   * @param heapType
   *            the type of the heap
   * @see HeapType
   */
  public HeapArrayListImpl(final V[] array, final HeapType heapType) {
    super();
    init(array, heapType);
  }

  /**
   * Initializes the heap with the given collection
   *
   * @param list
   *            the collection providing the elements for the heap
   * @param heapType
   *            the type of the heap
   * @see HeapType
   */
  public HeapArrayListImpl(final Collection<V> list, final HeapType heapType) {
    super();
    init(list, heapType);
  }

```

```
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

```java
package at.fhooe.swe4.lab3.sort.heap.impl;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import at.fhooe.swe4.lab3.sort.api.Heap;
import at.fhooe.swe4.lab3.sort.api.Heap.HeapType;
import at.fhooe.swe4.lab3.sort.api.Sorter;
import at.fhooe.swe4.lab3.stat.StatisticsProvider;

/**
 * This is the heap sorter implementation of the Sorter interface.
 *
 * @author Thomas Herzog
 *
 * @param <V>
 *            the values type of the to sort array or collection managed
 *            elements
 */
public class HeapSorter<V extends Comparable<V>> implements Sorter<V> {

  private final Heap<V> heap = new HeapArrayListImpl<V>();

  public HeapSorter() {
    super();
  }

  @SuppressWarnings("unchecked")
  @Override
  public V[] sort(final V[] array, final SortType sorterType) {
    if (array == null) {
      throw new IllegalArgumentException("Cannot sort empty array");
    }
    return (array.length == 0) ? array : ((V[]) sort(Arrays.asList(array), sorterType).
    toArray());
  }

  @Override
  public List<V> sort(final List<V> list, final SortType sorterType) {
    if (sorterType == null) {
      throw new IllegalArgumentException("SorterType not defined");
    }
    if (list == null) {
      throw new IllegalArgumentException("Cannot sort null list");
    }
    heap.init(list, convertToHeapType(sorterType));
    final List<V> result = new ArrayList<V>();
    while (heap.hasNext()) {
      result.add(heap.dequeue());
    }
    return result;
  }

  @Override
  public StatisticsProvider getStatisitcs() {
    return heap.getStatisitcs();
  }

  /**
   * Converts the sorter type to the corresponding heap type.
   *
```

```
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 }
```

### 1.3 Tests

| Input | Output | Comment |
|---|---|---|
| a = 100<br>b = 200 | 3 | |