

<input type="checkbox"/> Gr. 1, E. Pitzer	Name <u>Roman Lumetsberger</u>	Aufwand in h <u>8</u>
<input checked="" type="checkbox"/> Gr. 2, F. Gruber-Leitner	Punkte _____	Kurzzeichen Tutor / Übungsleiter _____ / _____

Das Problem von Richard H.**(9 Punkte)**

Implementieren Sie einen effizienten Algorithmus in Java um die „5-glatten“ Zahlen bis zu einer Schranke n zu finden. Das sind alle Zahlen, deren Primfaktoren kleiner gleich fünf sind. Anders gesagt, alle Zahlen, die sich als $2^x * 3^y * 5^z$ darstellen lassen. Eine dritte Möglichkeit ist die Definition als sogenannte Hammingfolge H :

- $1 \in H$
- $h \in H \Rightarrow 2 \cdot h \in H \wedge 3 \cdot h \in H \wedge 5 \cdot h \in H$
- keine weiteren Zahlen sind Elemente von H

Die ersten 10 Hammingzahlen sind somit 1, 2, 3, 4, 5, 6, 8, 9, 10 und 12.

Die Implementierung sollte dabei effizient genug sein um z.B. die 10000-ste Hammingzahl (288325195312500000) in deutlich unter einer Sekunde zu berechnen.

Schlacht der Sortieralgorithmen (in Java)**(6 + 6 + 3 Punkte)**

Nachdem wir uns in der Übung wieder mit der Heap-Datenstruktur beschäftigt haben, kommen sicher Erinnerungen an die ersten beiden Semester wieder, wo wir uns mit Sortieralgorithmen beschäftigt haben. Insbesondere mit dem Heapsort- sowie dem Quicksort-Algorithmus. Implementieren Sie beide Algorithmen in Java auf einfache Integer Felder und vergleichen Sie sowohl die Anzahl der Elementvergleiche als auch die Anzahl der Vertauschungsoperationen.

- Implementierung, Dokumentation und ausführliches Testen des HeapSort-Algorithmus auf Integer Felder.
- Implementierung, Dokumentation und ausführliches Testen des QuickSort-Algorithmus auf Integer Felder.
- Vergleichen Sie die beiden Implementierungen mit Hilfe von `System.nanoTime()` sowie durch Instrumentieren der Algorithmen um die Anzahl der Elementvergleiche und Vertauschungsoperationen (swaps) mit zu zählen. Erstellen Sie eine kleine Statistik für Felder bis zu einer Größe von mindestens 50000 Elementen z.B. alle Zweierpotenzen und führen Sie eine ausreichende Anzahl von Wiederholungen durch um eine statistisch Signifikante Aussage machen zu können.

1 Das Problem von Richard H.

1.1 Lösungsidee

Die Hammingfolge lässt sich mit Hilfe der Definition laut Angabe umsetzen.

Dabei wird 1 als erste Hammingzahl in eine Liste eingefügt. Dann können die weiteren Zahlen durch multiplizieren mit 2, 3 und 5 hinzugefügt werden.

Durch Mitführen von Index-Variablen wird sichergestellt, dass jede Zahl in der Liste genau einmal mit 2, 3 und 5 multipliziert wird. Sobald die angegebene obere Schranke erreicht ist, kann die Schleife beendet werden.

1.2 Sourcecode

Hamming.java

```
1 package at.lumetsnet.swo.ue3.hamming;
2
3 import java.math.BigInteger;
4 import java.util.ArrayList;
5 import java.util.List;
6
7 /**
8  * Hamming number generator class
9  *
10  * @author romanlum
11  *
12  */
13 public class Hamming {
14
15     /**
16      * Constant 2 in BigInteger representation
17      */
18     private static final BigInteger TWO = BigInteger.valueOf(2);
19     /**
20      * Constant 3 in BigInteger representation
21      */
22     private static final BigInteger THREE = BigInteger.valueOf(3);
23     /**
24      * Constant 5 in BigInteger representation
25      */
26     private static final BigInteger FIVE = BigInteger.valueOf(5);
27
28     /**
29      * Calculates the hamming numbers till the given upper barrier
30      *
31      * @param upper
32      * @return
33      */
34     public static List<BigInteger> calculate(BigInteger upper) {
35         List<BigInteger> result = new ArrayList<BigInteger>();
36         // add first hamming number
37         result.add(BigInteger.ONE);
38
39         BigInteger value2 = TWO;
40         BigInteger value3 = THREE;
41         BigInteger value5 = FIVE;
42         // variables which point to the current
43         // index to multiply with
44         int index2, index3, index5;
45         index2 = index3 = index5 = 0;
```

```
46
47     BigInteger currentMin;
48     while (true) {
49         // add the next hamming number to list
50         currentMin = (value3.min(value5)).min(value2);
51
52         // stop if we have reached upper limit
53         if (currentMin.compareTo(upper) == 1) {
54             break;
55         }
56
57         result.add(currentMin);
58
59         // check all values against the current min
60         // and increase the indexes if needed
61         if (currentMin.compareTo(value2) == 0) {
62             index2++;
63             value2 = TWO.multiply(result.get(index2));
64         }
65         if (currentMin.compareTo(value3) == 0) {
66             index3++;
67             value3 = THREE.multiply(result.get(index3));
68         }
69         if (currentMin.compareTo(value5) == 0) {
70             index5++;
71             value5 = FIVE.multiply(result.get(index5));
72         }
73     }
74
75     return result;
76 }
77 }
```

HammingTest.java

```
1 package at.lumetsnet.swo.ue3.test;
2 import java.math.BigInteger;
3 import java.util.List;
4 import java.util.concurrent.TimeUnit;
5
6 import at.lumetsnet.swo.ue3.hamming.Hamming;
7
8 /**
9  * Testclass for Hamming class
10  * @author romanlum
11  *
12  */
13 public class HammingTest {
14
```

```

14 public static void runTests() {
15
16     System.out.println("Testcase I: Hamming number 1 - 10");
17     long time = System.nanoTime();
18     List<BigInteger> result = Hamming.calculate(BigInteger.valueOf(10));
19     System.out.println("Time "+TimeUnit.MILLISECONDS.convert(System.nanoTime()
20         -time,TimeUnit.NANOSECONDS) + "ms");
21
22     result.forEach((x) -> System.out.print(x + ","));
23     System.out.println();
24     System.out.println();
25
26     System.out.println("Testcae III: 10 000 Hamming number");
27     calculateAndPrintHamming(BigInteger.valueOf(288325195312500001L));
28
29     System.out.println("Testcae III: 1 000 000 Hamming number");
30     calculateAndPrintHamming(BigInteger.valueOf( 51931278044839L)
31         .multiply(BigInteger.valueOf(10).pow(70)));
32
33 }
34
35
36 private static void calculateAndPrintHamming(BigInteger upperBoundary) {
37     long time = System.nanoTime();
38     List<BigInteger> result = Hamming.calculate(upperBoundary);
39     System.out.println("Time "+TimeUnit.MILLISECONDS.convert(System.nanoTime()
40         -time,TimeUnit.NANOSECONDS) + "ms");
41     System.out.println("Count: "+ result.size());
42     System.out.println("Last entry: " + result.get(result.size()-1));
43     System.out.println();
44
45 }
46 }

```

1.3 Testfälle

[illegible]

2 Schlacht der Sortieralgorithmen

2.1 Lösungsidee

Heapsort

- Für den Heapsort wird zu Beginn das Eingabefeld in einen Heap übergeführt. Damit befindet sich das größte Element an erster Stelle.
- Dieses wird dann mit dem letzten Element vertauscht und die Größe um 1 verringert.
- Danach wird wieder sichergestellt, dass es sich um einen Heap handelt.
- Damit ist das nächst größere Element an erster Stelle, welches wieder vertauscht wird.
- Dies wird dann solange wiederholt bis die Heapgröße gleich 1 ist.

Dann hat man ein aufsteigend sortiertes Feld.

Quicksort

Beim Quicksort wird die Menge in Teillisten getrennt und in sich sortiert. Die Trennung erfolgt durch ein Pivot Element.

- In der linken Liste sind jene Elemente, die kleiner als das Pivot Element sind.
- In der rechten Liste sind jene Elemente, die größer als das Pivot Element sind.

Statistik

Um die Statistik erstellen zu können, wird eine eigene Klasse *InstrumentationData* erstellt, die die benötigten Daten aufnehmen kann.

Weiters müssen die Algorithmen erweitert werden, damit sie die Daten zur Verfügung stellen. Durch die neue Basisklasse *Sorter* wird die Instrumentierung abstrahiert.

Die Statistik wird in den Testfällen generiert.

2.2 Sourcecode

HeapSort.java

```
1 package at.lumetsnet.swo.ue3.sort;
2
3 /**
4  * Heap sort implementation
5  *
6  * @author romanlum
7  *
8  */
9 public class HeapSort extends Sorter {
10
11     private int heapSize;
12
13     protected void doSort(int[] arr) {
14         if (arr == null || arr.length == 0) {
15             // nothing todo
16             return;
17         }
18         buildHeap(arr);
19         for (int i = arr.length - 1; i >= 0; i--) {
20             swap(arr, 0, i);
21             heapSize--;
22             heapify(arr, 0);
23         }
24     }
25
26     private void buildHeap(int[] arr) {
27         heapSize = arr.length - 1;
28         for (int i = heapSize / 2; i >= 0; i--) {
29             heapify(arr, i);
30         }
31     }
32
33     private void heapify(int[] arr, int index) {
34         int left = 2 * index + 1;
35         int right = 2 * index + 2;
36         int largest = index;
37         if (left <= heapSize && arr[left] > arr[index]) {
38             largest = left;
39             instrumentationData.addComparison();
40         }
41         if (right <= heapSize && arr[right] > arr[largest]) {
42             largest = right;
43             instrumentationData.addComparison();
44         }
45     }
```

```
46     if (largest != index) {
47         swap(arr, index, largest);
48         heapify(arr, largest);
49     }
50 }
51 }
```

QuickSort.java

```
1 package at.lumetsnet.swo.ue3.sort;
2
3 /**
4  * Quicksort implementation
5  *
6  * @author romanlum
7  *
8  */
9 public class QuickSort extends Sorter {
10
11     private int sortDataSize;
12     private int[] data;
13
14     /**
15      * Sorts the data
16      *
17      * @param values
18      */
19     protected void doSort(int[] values) {
20
21         // check for empty or null array
22         if (values == null || values.length == 0) {
23             return;
24         }
25
26         this.data = values;
27         sortDataSize = values.length;
28         quicksort(0, sortDataSize - 1);
29     }
30
31     private void quicksort(int low, int high) {
32         int i = low, j = high;
33
34         // get pivot element in the middle
35         int pivot = data[low + (high - low) / 2];
36
37         // Divide into two lists
38         while (i <= j) {
39
40             while (data[i] < pivot) {
```



```
41         i++;
42         instrumentationData.addComparison();
43     }
44
45     while (data[j] > pivot) {
46         instrumentationData.addComparison();
47         j--;
48     }
49
50     if (i <= j) {
51         swap(data, i, j);
52         i++;
53         j--;
54     }
55 }
56
57 // recursion
58 if (low < j)
59     quicksort(low, j);
60 if (i < high)
61     quicksort(i, high);
62 }
63
64 }
```

InstrumentationData.java

```
1 package at.lumetsnet.swo.ue3.sort;
2
3 /**
4  * Encapsulates the instrumentation data for sorting
5  * @author romanlum
6  *
7  */
8 public class InstrumentationData {
9     private int comparisonCount;
10    private int swapCount;
11    private long sortTime;
12
13    public void addSwap() {
14        swapCount++;
15    }
16
17    public void addComparison() {
18        comparisonCount++;
19    }
20
21
22    public void clear() {
```

```
23     comparisonCount = 0;
24     swapCount = 0;
25     sortTime = 0;
26 }
27
28 public String toString() {
29     StringBuilder builder=new StringBuilder();
30     builder.append("Sort time: ");
31     builder.append(sortTime);
32     builder.append(" ns \n");
33     builder.append("Comparison count: ");
34     builder.append(comparisonCount);
35     builder.append("\n");
36     builder.append("Swap count: ");
37     builder.append(swapCount);
38     builder.append("\n");
39
40     return builder.toString();
41 }
42
43 public int getComparisonCount() {
44     return comparisonCount;
45 }
46
47 public void setComparisonCount(int comparisonCount) {
48     this.comparisonCount = comparisonCount;
49 }
50
51 public int getSwapCount() {
52     return swapCount;
53 }
54
55 public void setSwapCount(int swapCount) {
56     this.swapCount = swapCount;
57 }
58
59 public long getSortTime() {
60     return sortTime;
61 }
62
63 public void setSortTime(long sortTime) {
64     this.sortTime = sortTime;
65 }
66
67
68 }
```

Sorter.java

```
1 package at.lumetsnet.swo.ue3.sort;
2
3 /**
4  * Base class used for abstracting sorting and instrumentation
5  * @author romanlum
6  */
7 public abstract class Sorter {
8
9     protected InstrumentationData instrumentationData;
10
11     /**
12      * Sorts the data
13      * @param data
14      */
15     public void sort(int[] data) {
16         instrumentationData = new InstrumentationData();
17         long begin = System.nanoTime();
18         doSort(data);
19         long end = System.nanoTime();
20         instrumentationData.setSortTime(end-begin);
21     }
22
23     /**
24      * Sort method which needs to be implemented
25      * @param data
26      */
27     protected abstract void doSort(int[] data);
28
29     /**
30      * Swaps the two field elements
31      * refreshes instrumentation data
32      * @param arr
33      * @param x
34      * @param y
35      */
36     protected void swap(int[] arr, int x, int y) {
37         int temp = arr[x];
38         arr[x] = arr[y];
39         arr[y] = temp;
40         instrumentationData.addSwap();
41     }
42
43     /**
44      * Gets the instrumentation data for the current sort operation
45      * @return
46      */
47     public InstrumentationData getInstrumentationData() {
48         return instrumentationData;
49     }
50 }
```

50
51 }

SortTest.java

```
1 package at.lumetsnet.swo.ue3.test;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6 import java.util.Random;
7 import java.util.concurrent.TimeUnit;
8
9 import at.lumetsnet.swo.ue3.sort.HeapSort;
10 import at.lumetsnet.swo.ue3.sort.InstrumentationData;
11 import at.lumetsnet.swo.ue3.sort.QuickSort;
12 import at.lumetsnet.swo.ue3.sort.Sorter;
13
14 /**
15  * Testclass for Heapsort and Quicksort
16  * @author romanlum
17  *
18  */
19 public class SortTest {
20
21     public static void runTests() {
22         sortTest(new HeapSort());
23         sortTest(new QuickSort());
24         sortComparison();
25     }
26
27     private static void sortTest(Sorter sorter) {
28         String testCaseSuffix = sorter.getClass().getSimpleName();
29         System.out.println("Testcase "+testCaseSuffix+" I: Empty array/null");
30         int[] data = new int[0];
31         sorter.sort(data);
32         sorter.sort(null);
33         System.out.println("No error.");
34         System.out.println();
35
36         System.out.println("Testcase "+testCaseSuffix+" II: Small data");
37         data = new int[]{9,8,3,5,6,2,1,100};
38         sorter.sort(data);
39         sorter.sort(null);
40         Arrays.stream(data).forEach((x)->System.out.print(x + ","));
41         System.out.println();
42         System.out.println();
43     }
44 }
```

```
45     System.out.println("Testcase "+testCaseSuffix+" III: Same data test");
46     data = new int[]{9,9,9,9,6,2,6,100,2,2};
47     sorter.sort(data);
48     sorter.sort(null);
49     Arrays.stream(data).forEach((x)->System.out.print(x + ","));
50     System.out.println();
51     System.out.println();
52
53     System.out.println("Testcase "+testCaseSuffix+" IV: Instrumentation data");
54     data = new int[]{9,9,9,9,6,2,6,100,2,2};
55     sorter.sort(data);
56     System.out.println(sorter.getInstrumentationData());
57
58 }
59
60 /**
61  * Compares heapsort and quicksort by using array lengths
62  * from 2 to 70000.
63  * For every size the sort is done 100 times and the average
64  * is used.
65  */
66 private static void sortComparison() {
67     System.out.println("Testcase: Sort comparison");
68     int currentPow = 1;
69     int currentSize = 2;
70     HeapSort heapSort = new HeapSort();
71     QuickSort quickSort = new QuickSort();
72
73     while(currentSize < 70000) {
74         ArrayList<InstrumentationData> hData = new ArrayList<InstrumentationData>(100);
75         ArrayList<InstrumentationData> qData = new ArrayList<InstrumentationData>(100);
76
77         //do the sort 100 times
78         for(int i = 0; i < 100; i++) {
79
80             int[] data = getData(currentSize);
81             //Copy the array to have equal test data for both sorts
82             heapSort.sort(Arrays.copyOf(data,data.length));
83             quickSort.sort(data);
84             hData.add(heapSort.getInstrumentationData());
85             qData.add(quickSort.getInstrumentationData());
86         }
87
88
89         System.out.printf("Size %1d\n", currentSize);
90         System.out.printf("  HeapSort:\t%4d micro sec\t%5d comps\t%5d swaps\n",
91             getTimeAverage(hData),
92             getComparisonAverage(hData),
93             getSwapAverage(hData));
```

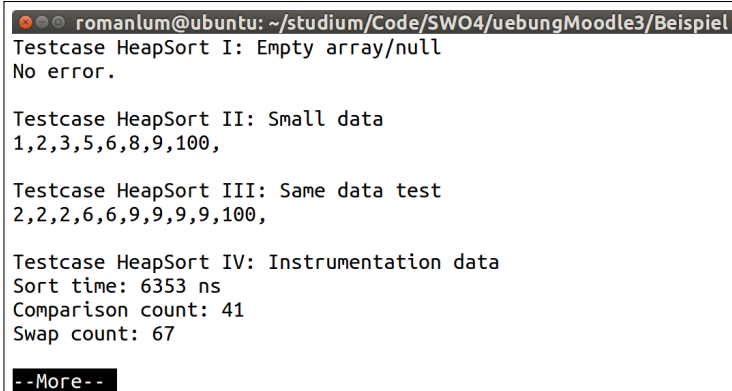
```
94     System.out.printf(" QuickSort:\t%4d micro sec\t%5d comps\t%5d swaps\n",
95         getTimeAverage(qData),
96         getComparisonAverage(qData),
97         getSwapAverage(qData));
98     currentPow++;
99     currentSize = (int) Math.pow(2, currentPow);
100 }
101 }
102
103 /**
104  * Gets the average comparison count of the given data
105  * @param data
106  * @return
107  */
108 private static int getComparisonAverage(List<InstrumentationData> data) {
109     return (int) data.stream().mapToDouble(c -> c.getComparisonCount()).average().getAsDouble();
110 }
111 /**
112  * Gets the average swap count of the given data
113  * @param data
114  * @return
115  */
116 private static int getSwapAverage(List<InstrumentationData> data) {
117     return (int) data.stream().mapToDouble(c -> c.getSwapCount()).average().getAsDouble();
118 }
119 /**
120  * Gets the average time of the given data
121  * @param data
122  * @return
123  */
124 private static long getTimeAverage(List<InstrumentationData> data) {
125     long nsec = (long) data.stream().mapToDouble(c -> c.getSwapCount())
126         .average().getAsDouble();
127     return TimeUnit.MICROSECONDS.convert(nsec, TimeUnit.NANOSECONDS);
128 }
129
130 /**
131  * Gets random test data
132  * @param size
133  * @return
134  */
135 private static int[] getData(int size) {
136     Random rnd = new Random();
137     rnd.setSeed(System.nanoTime());
138     int[] data = new int[size];
139     for (int i = 0; i < size; i++) {
140         data[i] = rnd.nextInt();
141     }
142     return data;
143 }
```

```
143     }  
144 }
```

Main.java

```
1 package at.lumetsnet.swo.ue3.test;  
2  
3 import at.lumetsnet.swo.ue3.test.HammingTest;  
4 import at.lumetsnet.swo.ue3.test.SortTest;  
5  
6 public class Main {  
7  
8     public static void main(String[] args) {  
9         HammingTest.runTests();  
10        SortTest.runTests();  
11    }  
12 }
```

2.3 Testfälle

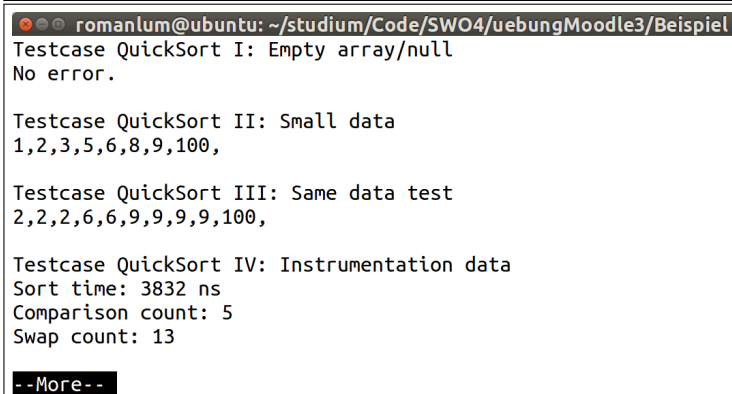


romanlum@ubuntu: ~/studium/Code/SWO4/uebungMoodle3/Beispiel
Testcase HeapSort I: Empty array/null
No error.

Testcase HeapSort II: Small data
1,2,3,5,6,8,9,100,

Testcase HeapSort III: Same data test
2,2,2,6,6,9,9,9,9,100,

Testcase HeapSort IV: Instrumentation data
Sort time: 6353 ns
Comparison count: 41
Swap count: 67
--More--



romanlum@ubuntu: ~/studium/Code/SWO4/uebungMoodle3/Beispiel
Testcase QuickSort I: Empty array/null
No error.

Testcase QuickSort II: Small data
1,2,3,5,6,8,9,100,

Testcase QuickSort III: Same data test
2,2,2,6,6,9,9,9,9,100,

Testcase QuickSort IV: Instrumentation data
Sort time: 3832 ns
Comparison count: 5
Swap count: 13
--More--

```

romanlum@ubuntu: ~/studium/Code/SWO4/uebungMoodle3/Beispiel
Testcase: Sort comparison
Size 2
  HeapSort:      0 micro sec      0 comps      2 swaps
  QuickSort:     0 micro sec      0 comps      1 swaps
Size 4
  HeapSort:      0 micro sec      3 comps      7 swaps
  QuickSort:     0 micro sec      3 comps      2 swaps
Size 8
  HeapSort:      0 micro sec     15 comps     19 swaps
  QuickSort:     0 micro sec     11 comps      7 swaps
Size 16
  HeapSort:      0 micro sec     49 comps     53 swaps
  QuickSort:     0 micro sec     35 comps     19 swaps
Size 32
  HeapSort:      0 micro sec    142 comps    135 swaps
  QuickSort:     0 micro sec     99 comps     45 swaps
Size 64
  HeapSort:      0 micro sec    373 comps    330 swaps
  QuickSort:     0 micro sec    250 comps    106 swaps
Size 128
  HeapSort:      0 micro sec    933 comps    786 swaps
  QuickSort:     0 micro sec    625 comps    242 swaps
Size 256
  HeapSort:      1 micro sec   2243 comps   1823 swaps
  QuickSort:     0 micro sec   1468 comps    546 swaps
Size 512
  HeapSort:      4 micro sec   5252 comps   4156 swaps
  QuickSort:     1 micro sec   3496 comps   1206 swaps
Size 1024
  HeapSort:      9 micro sec  12039 comps   9335 swaps
  QuickSort:     2 micro sec   7943 comps   2650 swaps
Size 2048
  HeapSort:     20 micro sec  27134 comps  20701 swaps
  QuickSort:     5 micro sec  17461 comps   5789 swaps
Size 4096
  HeapSort:     45 micro sec  60404 comps  45509 swaps
  QuickSort:    12 micro sec  38847 comps  12525 swaps
Size 8192
  HeapSort:     99 micro sec 133095 comps 99185 swaps
  QuickSort:    26 micro sec  85361 comps 26910 swaps
--More--

romanlum@ubuntu: ~/studium/Code/SWO4/uebungMoodle3/Beispiel
Size 16384
  HeapSort:    214 micro sec 290774 comps 214767 swaps
  QuickSort:   57 micro sec 185738 comps 57662 swaps
Size 32768
  HeapSort:    462 micro sec 630650 comps 462267 swaps
  QuickSort:   122 micro sec 403475 comps 122781 swaps
Size 65536
  HeapSort:    990 micro sec 1359708 comps 990100 swaps
  QuickSort:   260 micro sec 869542 comps 260737 swaps
romanlum@ubuntu:~/studium/Code/SWO4/uebungMoodle3/Beispiel$

```