SWE 4x	**Übung zu Softwareentwicklung mit modernen Plattformen 4**	**SS 2015, Übung 3**

Abgabetermin: SA in der KW 17

☐ **Gr. 1, E. Pitzer** **Name** _____ **Aufwand in h** _____

☐ **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.

a) Implementierung, Dokumentation und ausführliches Testen des HeapSort-Algorithmus auf Integer Felder.

b) Implementierung, Dokumentation und ausführliches Testen des QuickSort-Algorithmus auf Integer Felder.

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

1.2 Sourcecode

Hamming.java

```java
package at.lumetsnet.swo.ue3;

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

/***
 * Hamming number generator class
 * @author romanlum
 *
 */
public class Hamming {

	/**
	 * Constant 2 in BigInteger representation
	 */
	private static final BigInteger TWO = BigInteger.valueOf(2);
	/**
	 * Constant 3 in BigInteger representation
	 */
	private static final BigInteger THREE = BigInteger.valueOf(3);
	/**
	 * Constant 5 in BigInteger representation
	 */
	private static final BigInteger FIVE = BigInteger.valueOf(5);

	/***
	 * Calculates the hamming numbers till the given upper barrier
	 * @param upper
	 * @return
	 */
	public static List<BigInteger> calculate(BigInteger upper) {
		List<BigInteger> result = new ArrayList<BigInteger>();
		//add first hamming number
		result.add(BigInteger.ONE);

		BigInteger value2 = TWO;
		BigInteger value3 = THREE;
		BigInteger value5 = FIVE;
		int index2 , index3, index5;
		index2 = index3 = index5 = 0;

		BigInteger currentMin;
		while(true) {
			//add the next hamming number to list
```

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

HammingTest.java

```
1  package at.lumetsnet.swo.ue3;
2  import java.math.BigInteger;
3  import java.util.List;
4  import java.util.concurrent.TimeUnit;
5
6  public class HammingTest {
7
8          public static void main(String[] args) {
9
10                 long time = System.nanoTime();
11                 System.out.println("Testcase I: Hamming number 1 - 10");
12                 List<BigInteger> result = Hamming.calculate(BigInteger.valueOf(10));
13                 System.out.println("Time "+TimeUnit.MILLISECONDS.convert(System.nanoTime()-time,T
14                 result.forEach((x) -> System.out.print(x + ","));
15                 System.out.println();
16                 System.out.println();
17
18                 System.out.println("Testcae III: 10 000 Hamming number");
19                 calculateAndPrintHamming(BigInteger.valueOf(288325195312500001L));
```

```
20
21                  System.out.println("Testcae III: 1 000 000 Hamming number");
22                  calculateAndPrintHamming(BigInteger.valueOf( 51931278044839L).multiply(BigInteger
23
24          }
25
26          private static void calculateAndPrintHamming(BigInteger upperBoundary) {
27                  long time = System.nanoTime();
28                  List<BigInteger> result = Hamming.calculate(upperBoundary);
29                  System.out.println("Time "+TimeUnit.MILLISECONDS.convert(System.nanoTime()-time,T
30                  System.out.println("Count: "+ result.size());
31                  System.out.println("Last entry: " + result.get(result.size()-1));
32                  System.out.println();
33
34          }
35  }
```

1.3 Testfälle