

<input type="checkbox"/> Gr. 1, E. Pitzer	Name <u>Wolfgang 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 $2^x \cdot 3^y \cdot 5^z$ 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.

Übung3:

1. Lösungs-Idee:

Beispiel1:

Es soll ein effizienter Algorithmus entwickelt werden, welcher eine Folge von Hammingzahlen berechnet. Wie in der Angabe gegeben, kann eine Hammingzahl berechnet werden, indem man eine bestehende Hammingzahl nimmt, (1) und diese mit 2, 3, und 5 Multipliziert.

Ersterer Ansatz wäre also einfach eine Schleife bis zur Schranke, welche in eine Liste die neu erzeugten Werte einfügt, falls sie noch nicht enthalten sind.

Als erstes Element wird also die 1 eingefügt. Danach in der Schleife, erstes Element also 1 genommen, mal 2,3,5 gerechnet, und die Ergebnisse, welche rauskommen, eingefügt, falls sie noch nicht in der Liste enthalten sind.

Leider ist hierbei das Problem, dass man immer gegen die ganze Liste prüfen muss, ob ein Wert schon enthalten ist. Daher ist dieser nicht wirklich effizient.

Die Grundidee können wir jedoch behalten. Wir ändern jedoch die Implementierung, indem man nur das kleinste Element vom Ergebnis einfügt, welches jedoch größer ist als das Maximum in der Liste enthaltene. Vom eingefügten Element wird nun der Counter erhöht, um das nächste eingefügte Element in der Liste zu verwenden. Die beiden anderen Counter werden nicht erhöht, die Multiplikation bleibt also die gleiche als vorher. Danach wieder die Prüfung. Sind alle berechneten Elemente kleiner dem max. der Liste, so ist mit Sicherheit ein Element gleich groß als das zuletzt eingefügte Element. In diesem Fall wird der Counter von diesem erhöht.

Dieser Algorithmus kann zwar noch verbessert werden, indem man zum Beispiel die Multiplikation nicht öfters durchführt, für gleiche Zahlen, funktioniert aber schon ausreichend genug um die geforderte Zahl deutlich unter einer Sekunde zu berechnen.

Beispiel2:

Hierbei ist nicht allzu viel zu erwähnen. Die beiden Algorithmen Heap und Quicksort sind uns ohnehin bekannte Algorithmen. Ich habe eine Klasse SortAlgorithms erstellt, welche die Funktionen für HeapSort und Quicksort sowie Hilfs Methoden enthält. Eine weitere Klasse SortStatistics wird verwendet, um ein Logging über die Sortierung, Vertauschungen, Vergleiche, Zeiten zu speichern.

Die Klasse SortAlgorithms enthält als member Variable das zu sortierende Array, und eine vom Typ SortStatistic. Wird nun QuickSort oder HeapSort aufgerufen, wird diese richtig initialisiert, und kann danach aufgerufen werden.

Eine Funktion generateRandomValues(int n) initialisiert das Array der Klasse mittels n Random erstellten Zahlen.

Für die Zufallszahlen Erstellung wird Math.random() verwendet.

2. Code:

```
package at.fhhagenberg.swe.uebung03;

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

public class HammingNumbers {

    /**
     * Helper Method to return lowest value which is greater than
     valueHolder
     * if valueHolder is bigger than one,two,three return null
     * @param one
     * @param two
     * @param three
     * @param valueHolder
     * @return
     */
    private static BigInteger determinMin(BigInteger one, BigInteger two,
    BigInteger three, BigInteger valueHolder){
        if(one.compareTo(two)<=0 && one.compareTo(three) <= 0 &&
one.compareTo(valueHolder)>0){
            return one;
        }
        if(two.compareTo(one) <= 0 && two.compareTo(three) <=0 &&
two.compareTo(valueHolder)>0){
            return two;
        }
        if(three.compareTo(one) <= 0 && three.compareTo(two) <= 0 &&
three.compareTo(valueHolder) > 0){
            return three;
        }
        return null;
    }

    /**
     * calculates amount of hammingnumbers.
     * @param amount
     * @return
     */
    public static List<BigInteger> calculateHammingNumbers(int amount){
        if(amount < 1){
            throw new IllegalArgumentException("Amount must be equal
or greather 1");
        }
        long startTime = System.currentTimeMillis();
        List<BigInteger> result = new ArrayList<BigInteger>();

        BigInteger second = new BigInteger("2");
        BigInteger third = new BigInteger("3");
        BigInteger five = new BigInteger("5");
        BigInteger valueHolder = BigInteger.ONE;

        int count=0;
        int multCount1 = 0;
        int multCount2 = 0;
        int multCount3 = 0;

        // add first entry
```

```
        result.add(valueHolder);

        while(count < amount-1){
            BigInteger multValue2 =
result.get(multCount1).multiply(second);
            BigInteger multValue3 =
result.get(multCount2).multiply(third);
            BigInteger multValue5 =
result.get(multCount3).multiply(five);
            BigInteger value = determinMin(multValue2, multValue3,
multValue5, valueHolder);

            if(value != null){
                if(value.equals(multValue2)){
                    multCount1++;
                }else if(value.equals(multValue3)){
                    multCount2 ++;
                }else if(value.equals(multValue5)){
                    multCount3 ++;
                }

                result.add(value);
                valueHolder = value;
                count ++;
            }else{
                if(valueHolder.equals(multValue2)){
                    multCount1++;
                }
                if(valueHolder.equals(multValue3)){
                    multCount2++;
                }
                if(valueHolder.equals(multValue5)){
                    multCount3++;
                }
            }
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Duration in Millis: "+ (endTime -
startTime));
        return result;
    }
}
```

```
package at.fhhagenberg.swe.uebung03;

import static org.junit.Assert.*;

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

import org.junit.Test;

public class HammingNumbersTest {

    @Test
    public void calculate10Numbers() {
        List<BigInteger> numbers =
HammingNumbers.calculateHammingNumbers(10);
        for(BigInteger number : numbers){
            System.out.print(number + " | ");
        }
        assertEquals(new BigInteger("12"), numbers.get(numbers.size()-
1));
        assertEquals(10,numbers.size());
    }

    @Test
    public void calculate10000Numbers(){
        List<BigInteger> numbers =
HammingNumbers.calculateHammingNumbers(10000);
        BigInteger test = new BigInteger("288325195312500000");
        assertEquals(test,numbers.get(numbers.size()-1));
        assertEquals(10000,numbers.size());
    }

    @Test
    public void illegalParameter(){
        try{
            List<BigInteger> numbers =
HammingNumbers.calculateHammingNumbers(0);
            fail("Called with wrong Parameter should Throw Illegal
Argument Exception");
        }catch(IllegalArgumentException ex){
            assert(true);
        }
    }
}
```

```
package at.fhhagenberg.swe.uebung03;
import java.util.Date;

public class SortStatistics {

    private int swaps;
    private int recCalls;
    private int compares;
    private String sortAlgorithm;
    private long startTime;
    private long endTime;

    public SortStatistics(String name) {
        this.sortAlgorithm = name;
        swaps = 0;
        recCalls = 0;
        compares = 0;
    }
    public String getSortAlgorithm() {
        return sortAlgorithm;
    }
    public int getCompares() {
        return compares;
    }
    public int getRecCalls() {
        return recCalls;
    }
    public int getSwaps() {
        return swaps;
    }

    public void incSwaps(){
        this.swaps++;
    }
    public void incRecCalls(){
        this.recCalls++;
    }
    public void incCompares(){
        this.compares++;
    }
    public long getEndTime() {
        return endTime;
    }
    public void setEndTime(long endTime) {
        this.endTime = endTime;
    }
    public long getStartTime() {
        return startTime;
    }
    public void setStartTime(long startTime) {
        this.startTime = startTime;
    }
    public long duration(){
        return this.endTime - this.startTime;
    }
    @Override
    public String toString() {
        return "Statistics for: " + this.sortAlgorithm + "\nSwaps: " +
this.swaps + "\nCompares: " + this.compares + "\nRecCalls: " + this.recCalls +
        "\nDuration: " + this.duration();
    }
}
```

```
package at.fhhagenberg.swe.uebung03;

public class SortAlgorithms {
    // Member variables for Statistics of Sorting and value array to sort
    private SortStatistics statistic;
    private int[] values;

    /**
     * Constructor
     */
    public SortAlgorithms() {
        super();
    }

    /**
     * Constructor with given array
     * @param values
     */
    public SortAlgorithms(int[] values){
        this.values = values;
    }

    /**
     * reinitialize values array with amount of random values
     * @param amount
     */
    public void generateRandomValues(int amount){
        values = new int[amount];
        for(int i=0; i<values.length; i++){
            values[i] = ((Double) (Math.random()*100)).intValue();
        }
    }

    /**
     * Method to swap values of an int array
     * @param values
     * @param i
     * @param j
     */
    private void swap(int i, int j) {
        int temp = values[i];
        values[i] = values[j];
        values[j] = temp;
    }

    /**
     * Private helper Method heap sort
     * @param values
     * @param i
     * @param n
     */
    private void sink(int i, int n) {
        while(i <= (n / 2) - 1) {
            statistic.incCompares();
            // calculate Index of left child
            int kindIndex = ((i+1) * 2) - 1;

            //determin if left child exists
            if(kindIndex + 1 <= n -1) {
                statistic.incCompares();
                if(values[kindIndex] < values[kindIndex+1]) {
                    statistic.incCompares();
                    kindIndex++;
                }
            }
        }
    }
}
```

```

        // check if element has to sink
        if(values[i] < values[kindIndex]) {
            statistic.incCompares();
            statistic.incSwaps();
            swap(i, kindIndex);
            // repeat with new position
            i = kindIndex;
        } else break;
    }
}
/**
 * generate MaxHeap for Heapsort
 * @param values
 */
private void generateMaxHeap() {
    // start from middle backward
    for(int i = (values.length / 2) - 1; i >= 0 ; i--) {
        sink(i, values.length);
    }
}
/**
 * HeapSort
 * @param values
 */
public void heapSort() {
    if(this.values == null){
        return;
    }
    this.statistic = new SortStatistics("HeapSort");
    this.statistic.setStartTime(System.nanoTime());
    generateMaxHeap();
    // sorting
    for(int i = values.length - 1; i > 0; i--) {
        statistic.incSwaps();
        swap(i, 0);
        sink(0, i);
    }
    this.statistic.setEndTime(System.nanoTime());
}

public void quickSort() {
    if(this.values == null){
        return;
    }
    this.statistic = new SortStatistics("QuickSort");
    this.statistic.setStartTime(System.nanoTime());
    doQuickSort(0, values.length - 1);
    this.statistic.setEndTime(System.nanoTime());
}

private void doQuickSort(int low, int high) {
    int i = low, j = high;
    // Get the pivot element from the middle of the list
    int pivot = values[low + (high - low) / 2];
    // Divide into two lists
    while (i <= j) {
        statistic.incCompares();
        // If the current value from the left list is smaller
        then the pivot
        // element then get the next element from the left list
        while (values[i] < pivot) {
            statistic.incCompares();

```



```

        i++;
    }
    // If the current value from the right list is larger
then the pivot
    // element then get the next element from the right list
    while (values[j] > pivot) {
        statistic.incCompares();
        j--;
    }

    if (i <= j) {
        statistic.incCompares();
        statistic.incSwaps();
        swap(i, j);
        i++;
        j--;
    }
}
// Recursion is here
if (low < j){
    statistic.incCompares();
    statistic.incRecCalls();
    doQuickSort(low, j);
}
if (i < high){
    statistic.incCompares();
    statistic.incRecCalls();
    doQuickSort(i, high);
}

}

////////////////////////////////////////
///// GETTER & SETTER
////////////////////////////////////////
public SortStatistics getStatistic() {
    return statistic;
}
public int[] getValues() {
    return values;
}
public void setValues(int[] values) {
    this.values = values;
}
}

```

```
package at.fhhagenberg.swe.uebung03;

import static org.junit.Assert.*;

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

import org.junit.Test;

public class SortAlgorithmsTest {

    /**
     * Test HeapSort for 5 Values
     */
    @Test
    public void testHeapSort() {
        SortAlgorithms algo = new SortAlgorithms();
        int[] values = new int[5];
        values[0] = 5;
        values[1] = 1;
        values[2] = 3;
        values[3] = 4;
        values[4] = 2;
        algo.setValues(values);
        algo.heapSort();
        int[] expectedValues = new int[5];
        for(int i=0; i<expectedValues.length; i++){
            expectedValues[i] = i+1;
        }
        assert(Arrays.equals(expectedValues, algo.getValues()));
    }

    /**
     * Test QuickSort for 5 values
     */
    @Test
    public void testQuickSort() {
        SortAlgorithms algo = new SortAlgorithms();
        int[] values = new int[5];
        values[0] = 5;
        values[1] = 1;
        values[2] = 3;
        values[3] = 4;
        values[4] = 2;
        algo.setValues(values);
        algo.quickSort();
        int[] expectedValues = new int[5];
        for(int i=0; i<expectedValues.length; i++){
            expectedValues[i] = i+1;
        }
        assert(Arrays.equals(expectedValues, algo.getValues()));
    }

    /**
     * Test QuickSort without Setting an Array
     */
    @Test
    public void testQuickSortWithEmptyArray() {
        SortAlgorithms algo = new SortAlgorithms();
        algo.quickSort();
    }

    /**
     * Test HeapSort wihtout Setting an Array
     */
}
```

```

    */
@Test
public void testHeapSortWithEmptyArray(){
    SortAlgorithms algo = new SortAlgorithms();
    algo.heapSort();
}

/**
 * Run Test with 5000 random Numbers and print statistic
 */
@Test
public void testRandomHeapSort(){
    System.out.println("HeapSort With 5000 No");
    SortAlgorithms algo = new SortAlgorithms();
    algo.generateRandomValues(5000);
    algo.quickSort();
    System.out.println(algo.getStatistic());
}

/**
 * Run Test with 5000 random Numbers and print statistc
 */
@Test
public void testRandomQuickSort(){
    System.out.println("QuickSort With 5000 No");
    SortAlgorithms algo = new SortAlgorithms();
    algo.generateRandomValues(5000);
    algo.heapSort();
    System.out.println(algo.getStatistic());
}

/**
 * Run a competition between Quick and HeapSort
 * Which Sort has less Compares / Swaps
 * The Sort is Processed 100 times with 50000 numbers
 */
@Test
public void runCompetitionHeapQuickSort(){
    System.out.println("COMPETITION");
    SortAlgorithms algorithms = new SortAlgorithms();
    List<SortStatistics> statisticsQuick = new
ArrayList<SortStatistics>();
    List<SortStatistics> statisticsHeap = new
ArrayList<SortStatistics>();
    for(int i=0; i<10; i++){
        algorithms.generateRandomValues(50000);
        algorithms.heapSort();
        statisticsHeap.add(algorithms.getStatistic());
        algorithms.generateRandomValues(50000);
        algorithms.quickSort();
        statisticsQuick.add(algorithms.getStatistic());
    }
    System.out.println("QuickSort Summary:");
    long duration = 0;
    int swaps = 0;
    int compares= 0;
    int recalls = 0;
    for(SortStatistics s : statisticsQuick){
        duration += s.duration();
        swaps += s.getSwaps();
        compares += s.getCompares();
        recalls += s.getRecCalls();
    }
    System.out.println("Duration: "+ duration + " , Swaps: "+ swaps
+ " , Compares: "+ compares + " , ReCalls: "+ recalls );
}

```

```
        duration = 0;
        swaps = 0;
        compares = 0;
        recalls = 0;
        System.out.println("HeapSort Summary:");
        for (SortStatistics s : statisticsHeap) {
            duration += s.duration();
            swaps += s.getSwaps();
            compares += s.getCompares();
            recalls += s.getRecCalls();
        }
        System.out.println("Duration: "+ duration + " , Swaps: "+ swaps
+ " , Compares: "+ compares + " , ReCalls: "+ recalls );
    }
}
```

3. Test-Case:

Die Tests wurden mittels JUnit Tests dargestellt.

Die JUnit Test Classen sind im Block Code ersichtlich. Wenn man sie ausführt, produzieren sie folgende Ergebnisse:

Es befindet sich im Abgegeben Zip ein build.xml. Die Applikation kann mittels Ant gebaut werden.

Für die JUnit tests wurden die benötigten Jars mit abgegeben, sie befinden sich im ordner libs

Ant clean build

Nach dem erfolgreichen Build können mit folgenden Befehlen die JunitTests durchgeführt werden.

ant HammingNumbersTest

ant SortAlgorithmsTest

im anschluss befinden sich noch Screenshots von den Test per IDE und per Ant.

3.1. Hamming-Tests:

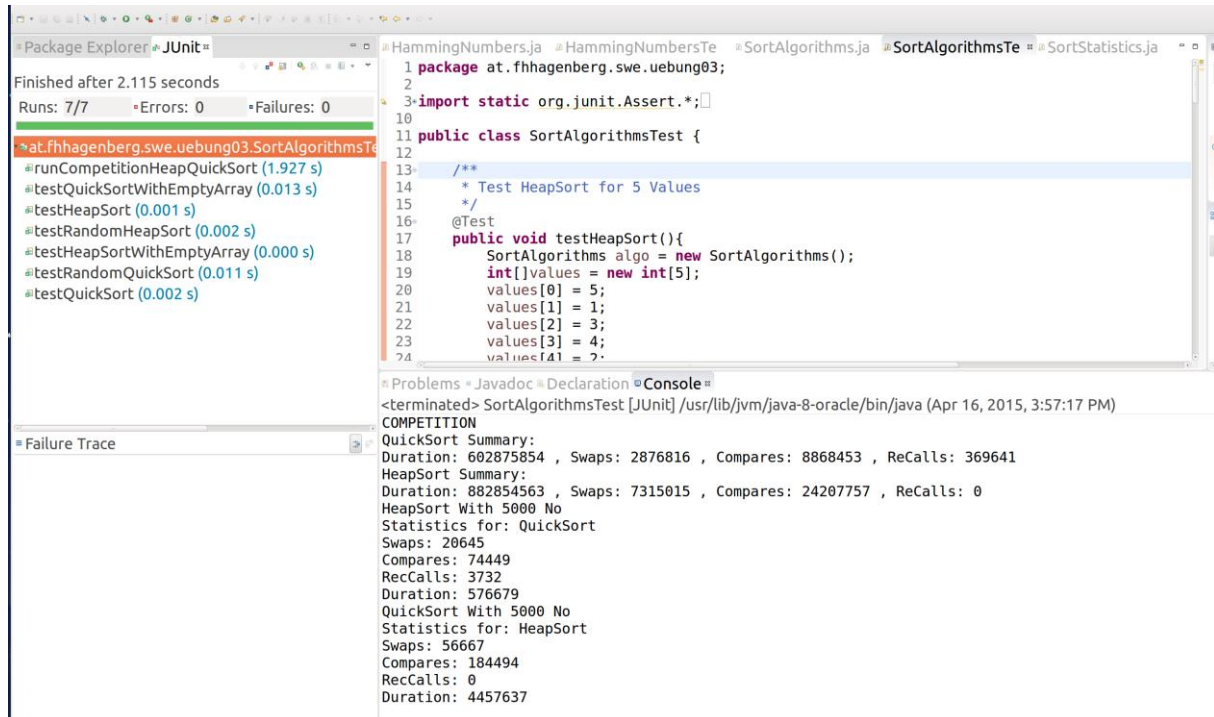
```
1 package at.fhhagenberg.swe.uebung03;
2
3 import static org.junit.Assert.*;
4
5
6
7
8
9
10 public class HammingNumbersTest {
11
12     @Test
13     public void calculate10Numbers() {
14         List<BigInteger> numbers = HammingNumbers.calculateHammingNumbers(10);
15         for(BigInteger number : numbers){
16             System.out.print(number + " | ");
17         }
18         assertEquals(new BigInteger("12"), numbers.get(numbers.size()-1));
19         assertEquals(10, numbers.size());
20     }
21 }
22
23 @Test
```

Finished after 0.625 seconds
Runs: 3/3 Errors: 0 Failures: 0

at.fhhagenberg.swe.uebung03.Hamm
#calculate10Numbers (0.027 s)
#illegalParameter (0.022 s)
#calculate10000Numbers (0.461 s)

<terminated> HammingNumbersTest [JUnit] /usr/lib/jvm/java-8-oracle/bin/java (Apr 16, 2015, 3:55:01 PM)
Duration in Millis: 9
1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | Duration in Millis: 420

3.2. Sorting-Tests:



The screenshot shows an IDE with the following components:

- Package Explorer:** Shows the package structure with `at.fhhagenberg.swe.uebung03.SortAlgorithmsTest` selected.
- JUnit Run Results:** Displays the execution of various tests:
 - Finished after 2.115 seconds
 - Runs: 7/7
 - Errors: 0
 - Failures: 0
 - Tests: `#runCompetitionHeapQuickSort (1.927 s)`, `#testQuickSortWithEmptyArray (0.013 s)`, `#testHeapSort (0.001 s)`, `#testRandomHeapSort (0.002 s)`, `#testHeapSortWithEmptyArray (0.000 s)`, `#testRandomQuickSort (0.011 s)`, `#testQuickSort (0.002 s)`
- Source Editor:** Contains the `SortAlgorithmsTest` class:

```
1 package at.fhhagenberg.swe.uebung03;
2
3 import static org.junit.Assert.*;
4
5
6
7
8
9
10
11 public class SortAlgorithmsTest {
12
13     /**
14      * Test HeapSort for 5 Values
15      */
16     @Test
17     public void testHeapSort(){
18         SortAlgorithms algo = new SortAlgorithms();
19         int[] values = new int[5];
20         values[0] = 5;
21         values[1] = 1;
22         values[2] = 3;
23         values[3] = 4;
24         values[4] = 2;
```
- Console:** Shows the output of the tests:

```
<terminated> SortAlgorithmsTest [JUnit] /usr/lib/jvm/java-8-oracle/bin/java (Apr 16, 2015, 3:57:17 PM)
COMPETITION
QuickSort Summary:
Duration: 602875854 , Swaps: 2876816 , Compares: 8868453 , Recalls: 369641
HeapSort Summary:
Duration: 882854563 , Swaps: 7315015 , Compares: 24207757 , Recalls: 0
HeapSort With 5000 No
Statistics for: QuickSort
Swaps: 20645
Compares: 74449
Recalls: 3732
Duration: 576679
QuickSort With 5000 No
Statistics for: HeapSort
Swaps: 56667
Compares: 184494
Recalls: 0
Duration: 4457637
```
- Failure Trace:** Empty.

3.3. Per Ant Console

```
wolfgang@ubuntu:~/Documents/Programming/Java/4.Semester/Uebung_03$ ant HammingNumbersTest
Buildfile: /home/wolfgang/Documents/Programming/Java/4.Semester/Uebung_03/build.xml

HammingNumbersTest:
[junit] Running at.fhhagenberg.swe.uebung03.HammingNumbersTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.35 sec
[junit] Output:
[junit] Duration in Millis: 0
[junit] 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | Duration in Millis: 122
[junit]

BUILD SUCCESSFUL
Total time: 3 seconds
wolfgang@ubuntu:~/Documents/Programming/Java/4.Semester/Uebung_03$ ant SortAlgorithmsTest
Buildfile: /home/wolfgang/Documents/Programming/Java/4.Semester/Uebung_03/build.xml

SortAlgorithmsTest:
[junit] Running at.fhhagenberg.swe.uebung03.SortAlgorithmsTest
[junit] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.789 sec
[junit] Output:
[junit] COMPETITION
[junit] QuickSort Summary:
[junit] Duration: 152759390 , Swaps: 2875115 , Compares: 8927634 , ReCalls: 368791
[junit] HeapSort Summary:
[junit] Duration: 306499597 , Swaps: 7314583 , Compares: 24206628 , ReCalls: 0
[junit] HeapSort With 5000 No
[junit] Statistics for: QuickSort
[junit] Swaps: 20739
[junit] Compares: 71517
[junit] RecCalls: 3645
[junit] Duration: 324911
[junit] QuickSort With 5000 No
[junit] Statistics for: HeapSort
[junit] Swaps: 56769
[junit] Compares: 184708
[junit] RecCalls: 0
[junit] Duration: 745860
[junit]

BUILD SUCCESSFUL
Total time: 4 seconds
wolfgang@ubuntu:~/Documents/Programming/Java/4.Semester/Uebung_03$
```