Übung3:

# Lösungs-Idee:

### Beispiel1:

Es soll ein effizienter Algorithmus entwickelt werden, welcher eine Folge von Hammingzahlen berechnet. Wie in der Angabe gegeben, kann eine Hamming zahl berechnet werden, indem man eine bestehende Hamming zahl 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 Ergebnisse 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 eingefügt 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 losging 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.

# 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**;**

**}**

/\*\*

\* reinitalize 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 statisitc

\*/

@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 **);**

**}**

**}**

# 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

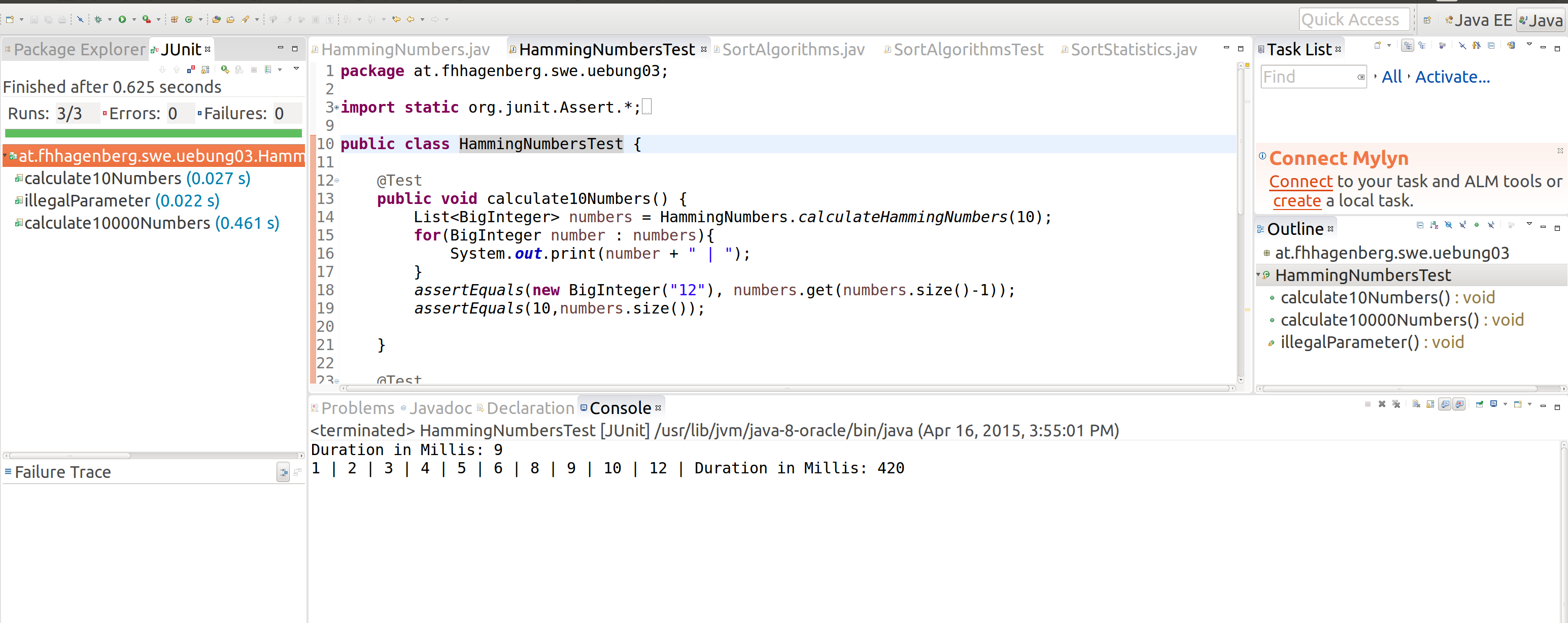
Nach dem erfolgreichen Build können mit folgenden Befehlen die JunitTests durchgeführt werden.

ant HammingNumbersTest

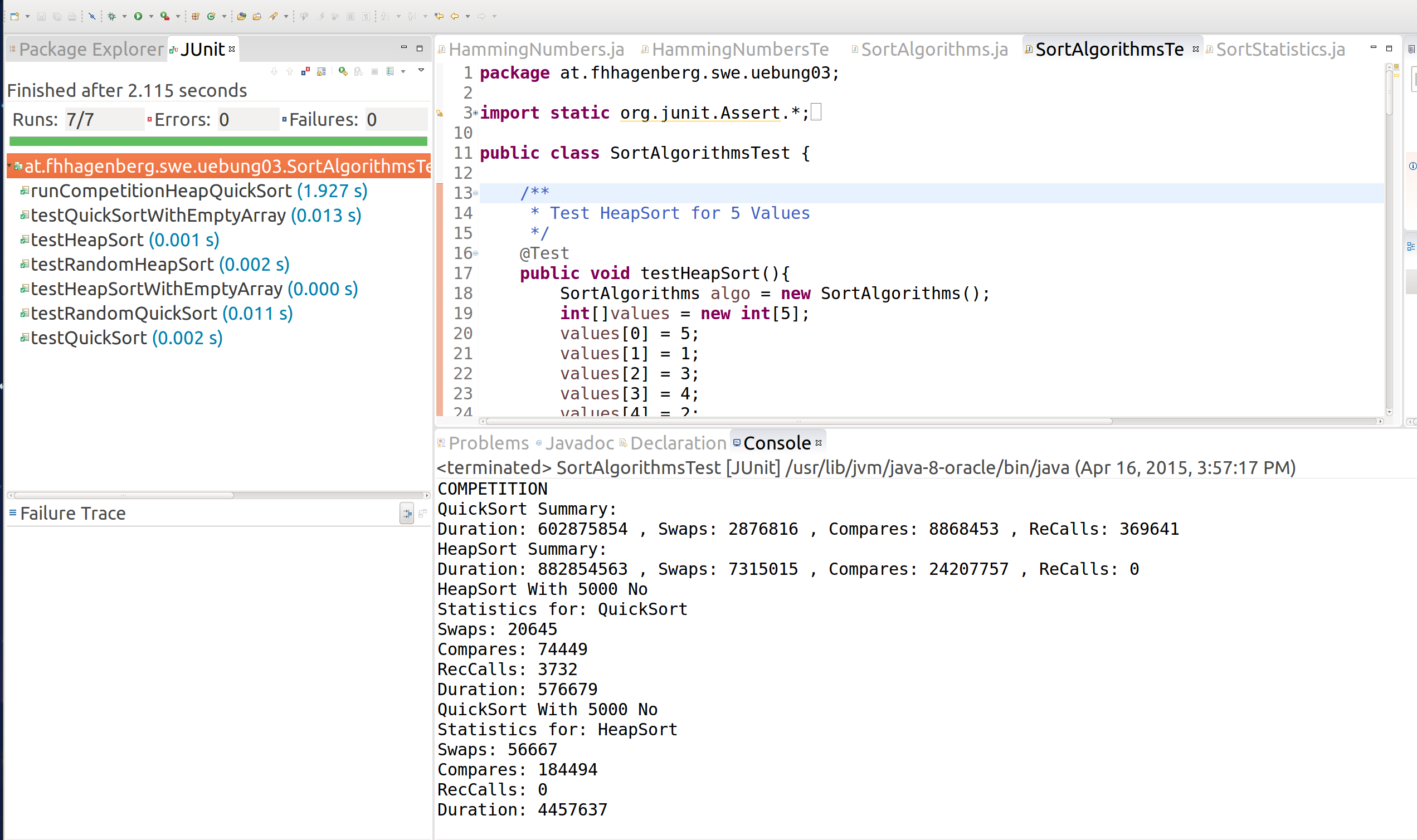
ant SortAlgorithmsTest

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

### 3.1. Hamming-Tests:



### 3.2. Sorting-Tests:



## Per Ant Console

