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

Abgabetermin: SA in der KW 17

☑ **Gr. 1, E. Pitzer** **Name** Stefan Kert **Aufwand in h** 5

☐ **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 Lösungsideen

1.1 Das Problem von Richard Hamming

Zur Bildung der Hammingfolge wird eine eigene Klasse implementiert. Es sollte drei öffentliche Methoden geben, welche die Hammingfolge auf unterschiedliche Art und Weise erstellen:

- *ArrayList<long> calculateSmootNumbersBy5UntilBoundary(int boundary)*
    - Diese Methode sollte alle 5 – glatten Zahlen bis zu einem bestimmten Grenzwert berechnen. D.h. wenn z.B. 8 übergeben wird werden die Zahlen: 1, 2, 3, 4, 5, 6 und 8 zurückgegeben, wo bei der Parameter boundary dem Grenzwert entspricht.

- *ArrayList<long> calculateFixedAmountOfSmoothNumbresBy5(int amount)*
    - Mit dieser Methode sollte eine bestimmte Menge an Hammingfolge generiert werden. Der Parameter *amount* gibt an wie viele generiert werden sollten. Wenn z.B. 5 übergeben wurde wird folgendes Ergebnis zurückgegeben: 1, 2, 3, 4, 5
- long calculateSmoothNumberBy5(int n)
    - Mit dieser Methode sollte eine Hammingfolge an einer bestimmten Stelle berechnet werden. Wenn z.B. 10 übergeben wurde sollte 12 zurückgegeben werden.

All diese Methoden sollten dabei die gleiche Methode aufrufen und sollten im Prinzip nur Wrapper für diese Methode sein. Zur eigentlichen Berechnung der Hammingfolge wird die Methode *calculateSmoothNumbersBy5* eingesetzt werden. Dieser Methode sollte eine *IHammingBreakCondition* übergeben werden welche angibt ob der aktuelle Schleifendurchlauf unterbrochen werden sollte.
Zu beachten ist dabei weiters, dass für den Datentyp int vermutlich zu klein sein wird und deshalb auf long zurückgegriffen werden sollte. Der Algorithmus sollte dabei eine Komplexität O(n) aufweisen, welche den Anforderung der Übung genügen sollte.

1.2 Schlacht der Sortieralgorithmen

1.2.1 Implementierung

Bei der Implementierung der beiden Sortieralgorithmen sollte es jeweils eine Klasse mit einer öffentlichen Methode *sort(int[] arr)* geben welches das übergebene Array sortiert. Die Implementierungen des Quicksorts und des Heapsorts sollten dabei so gewählt werden wie in den letzten Semestern beschrieben und werden daher nicht näher erläutert. Bei der Implementierung sollte jedoch darauf geachtet werden, dass nach jedem Vergleich und nach jedem Tausch eine Countervariable erhöht wird um die beiden Sortieralgorithmen zu vergleichen. Die Zeit welche der jeweilige Sortieralgorithmus benötigt sollte dabei so gemessen werden, dass beim Aufruf der Methode die Nanozeit gespeichert wird und nach der Methode noch einmal und die Differenz zwischen diesen beiden Zahlen wird danach ausgegeben. Weiters sollten zum leichteren Zählen bzw. zur Vermeidung von Codeverdoppelungen eine Basisklasse für gemeinsam verwendete Funktionen verwendet werden.

1.2.2 Effizienzvergleich

Algorithmus	Anzahl Elemente	Benötigte Zeit	SwapCounter	CompareCounter
Quicksort	1.000	0.218ms	2.585	21.826
	10.000	2.58ms	33.518	279.538
	100.000	29.34ms	412043	3.409.916
	1.000.000	326.55ms	48.93.943	40.170.665
	10.000.000	4016.53ms	56.364.426	467.965.366
Heapsort	1.000	0.26ms	10.079	40.009
	10.000	3.21ms	134.201	533.771
	100.000	38.23ms	1.674.959	6.669.535
	1.000.000	553.73ms	20.048.241	79.890.081
	10.000.000	8204.30ms	233.835.629	932.315.508

Anhand dieser Tabelle können wir einen Vergleich zwischen den beiden Algorithmen anstellen. Es wurden für den Vergleich Arrays mit zufällig erzeugten Werten übergeben. Um ein besseres Ergebnis bzw. einen genaueren Vergleich zu erhalten, wurde jeweils das gleiche Array übergeben. Damit eine

statistisch wertvolle Zeitmessung erfolgen kann, wurde die Zeitmessung 1000-mal durchgeführt und danach die durchschnittliche Zeit verwendet. Man kann erkennen, dass der Quicksort nicht nur um einiges schneller ist, als der Heapsort, sondern, dass er auch 1. Weniger Vertauschungsoperationen durchführt und zweitens weniger vergleiche benötigt. Vor allem bei mehr Elementen steigt die Laufzeit des Heapsorts sehr stark an, wo beim Quicksort die Laufzeit die Laufzeit sich zwar nicht ganz linear zu der Anzahl der Elemente verhält, jedoch um einiges besseres Verhalten aufweist wie der Heapsort.

2 Quelltext

2.1 Main

```
package at.skert.swe.ue3;
import at.skert.swe.ue3.hamming.tests.HammingTests;
import at.skert.swe.ue3.sorting.tests.SortTests;

public class Main {

	public static void main(String[] args) {
		// Execute Hamming Tests.
		HammingTests hammingTests = new HammingTests();
		hammingTests.executeAll();


		//Executing SortTests
		SortTests sortTests = new SortTests();
		sortTests.executeAll();
	}

}
```

2.2 Hamming

```
package at.skert.swe.ue3.hamming;

import java.util.ArrayList;
import at.skert.swe.ue3.utility.Utilities;


public class Hamming {
	final static long two = 2;
	final static long three = 3;
	final static long five = 5;
	private ArrayList<Long> hammingNumbers;

	/**
	 * Defines BreakCondition for calculating Smooth Numbers
	 */
	interface IHammingBreakCondition {
		boolean CheckIfCurrentValueShouldBeAdded(long currentValue);
	}

	public Hamming(){
		hammingNumbers = new ArrayList<Long>();
	}

	/**
	 * Calculates the 5 smooth numbers until fixed boundary
	 */
	public ArrayList<Long> calculateSmoothNumbersBy5UntilBoundary(int boundary){
		return calculateSmoothNumbersBy5(currentValue -> currentValue <= boundary);
	}


	/**
	 * Calculates the fixed amount of 5 smooth numbers
	 */
	public ArrayList<Long> calculateFixedAmountOfSmoothNumbersBy5(int amount){
		return calculateSmoothNumbersBy5(nextValue -> hammingNumbers.size() < amount);
	}
```

```
        /**
         * Calculates the n-th 5 smooth number
         */
        public long calculateSmoothNumberBy5(int n){
                ArrayList<Long> list = calculateFixedAmountOfSmoothNumbersBy5(n);
                return list.get(list.size() - 1);
        }

        /**
         * Calculates all 5 Smooth Numbers until the given condition returns false for the current
value
         */
        private ArrayList<Long> calculateSmoothNumbersBy5(IHammingBreakCondition breakCondition) {
                hammingNumbers.clear();
                hammingNumbers.add((long) 1); //Always add 1 because 1 is the first hamming Number
                int i = 0, j = 0, k = 0;
                long x2 = two, x3 = three, x5 = five;
                int index = 1;
                while(true){
                        long currentValue = Utilities.getMinimum(x2, x3, x5);
                        if(!breakCondition.CheckIfCurrentValueShouldBeAdded(currentValue)){
                                break;
                        }
                        hammingNumbers.add(currentValue);
                        //Recalculate values and increase index if needed
                        if (hammingNumbers.get(index) == x2)
                                x2 = recalculateValue(two, ++i);
                        if (hammingNumbers.get(index) == x3)
                                x3 = recalculateValue(three, ++j);
                        if (hammingNumbers.get(index) == x5)
                                x5 = recalculateValue(five, ++k);
                        index++;
                }
                return hammingNumbers;
        }

        /**
         * Returns the next Hamming Number for value in list.
         */
        private long recalculateValue(long fixedValue, int listIndex){
                return fixedValue * hammingNumbers.get(listIndex);
        }
}
```

2.4 IntSortBase

```
package at.skert.swe.ue3.sorting;

import at.skert.swe.ue3.utility.Utilities;

public abstract class IntSortBase {
    private long _swapCounter = 0;
    private long _compareCounter = 0;

    protected boolean isSortingNeeded(int[] arr){
                if (arr == null || arr.length < 2) {
                        return false;
                }
                return true;
    }

        protected boolean less(int firstValue, int secondValue) {
                _compareCounter++;
                return firstValue < secondValue;
        }
        protected boolean lessOrEqual(int firstValue, int secondValue) {
                _compareCounter++;
                return firstValue <= secondValue;
        }
        protected void swap(int[] arr, int i, int j){
                _swapCounter++;
                Utilities.swapArrayElements(arr, i, j);
        }

        public long getCompareCounter() {
                return _compareCounter;
        }

        public long getSwapCounter() {
                return _swapCounter;
        }
}
```

2.5 Quicksort

```
package at.skert.swe.ue3.sorting;

public class QuickSort extends IntSortBase {
	public void sort(int[] values) {
		if (!isSortingNeeded(values))
			return;
		quicksort(values, 0, values.length - 1);
	}

	private void quicksort(int[] values, int low, int high) {
		int i = low, j = high;
		int pivot = values[low + (high - low) / 2]; // Get centered element.

		//arrange elements in array
		while (lessOrEqual(i, j)) {
			while (less(values[i], pivot)) {
				i++;
			}
			while (less(pivot, values[j])) {
				j--;
			}
			if (lessOrEqual(i, j)) {
				swap(values, i, j);
				i++;
				j--;
			}
		}
		//recursive call for the left side
		if (less(low, j)){
			quicksort(values, low, j);
		}
		//recursive call for the right side
		if (less(i, high)){
			quicksort(values, i, high);
		}
	}
}
```

2.6 Heapsort

```
package at.skert.swe.ue3.sorting;

public class HeapSort extends IntSortBase{
    private int total;

    public void sort(int[] values)
    {
                if (!isSortingNeeded(values))
                        return;

        total = values.length - 1;

        // Create Binary Tree in Array
        for (int i = total / 2; i >= 0; i--)
            heapify(values, i);

        // Sink elements in the tree
        for (int i = total; i > 0; i--) {
            swap(values, 0, i);
        total--;
            heapify(values, 0);
        }
    }


    private void heapify(int[] values, int i)
    {
        int left = i * 2;
        int right = left + 1;
        int grt = i;

        if (lessOrEqual(left, total) && less(values[grt], values[left])){
        grt = left;
        }
        if (lessOrEqual(right, total) && less(values[grt], values[right])){
        grt = right;
        }
        if (grt != i) {
        swap(values, i, grt);
            heapify(values, grt);
        }
    }
}
```

3 Tests

3.1 Tests

3.1.1 10.000ste Hamming Zahl berechnen

Source:

```
public void testGenerate10000ThHammingNumber(){
        Hamming h = new Hamming();

        long startTime = System.nanoTime();
        long number = h.calculateSmoothNumberBy5(10000);
        long endTime = System.nanoTime();
        double timeInSeconds = Utilities.getSecondsForNanoSeconds(endTime - startTime);

        System.out.println("The 10000. Hamming Number is " + number);
        System.out.println("It took " + timeInSeconds + "s to generate number.");
}
```

Result:

```
The 10000. Hamming Number is 288325195312500000
It took 0.059429038s to generate number.
```

3.1.2 Die ersten 10 Hamming Zahlen berechnen

Source:

```
public void testGenerateFirst10HammingNumbers(){
        Hamming h = new Hamming();

        long startTime = System.nanoTime();
        ArrayList<Long> numbers = h.calculateFixedAmountOfSmoothNumbersBy5(10);
        long endTime = System.nanoTime();
        double timeInMilliSeconds = Utilities.getMilliSecondsForNanoSeconds(endTime - startTime);

        System.out.println(Arrays.toString(numbers.toArray()));
        System.out.println("It took " + timeInMilliSeconds + "ms to generate number.");
}
```

Result:

```
[1,2,3,4,5,6,8,9,10,12]
It took 0.084386ms to generate number.
```

3.1.3 Die Hamming Zahlen bis 12 berechnen

Source:

```
public void testGenerateHammingNumbersUntil12(){
        Hamming h = new Hamming();

        long startTime = System.nanoTime();
        ArrayList<Long> numbers = h.calculateSmoothNumbersBy5UntilBoundary(12);
        long endTime = System.nanoTime();

        double timeInMilliSeconds = Utilities.getMilliSecondsForNanoSeconds(endTime - startTime);

        System.out.println(Arrays.toString(numbers.toArray()));
        System.out.println("It took " + timeInMilliSeconds + "ms to generate number.");
}
```

Result:

```
[1,2,3,4,5,6,8,9,10,12]
It took 1.331939ms to generate number.
```

3.1.4 Heap Sort 10 zufällige Zahlen sortieren

```
public void assertHeapSortArrayIsSorted() {
        int[] randomArray = Utilities.generateRandomLongArray(10);
        HeapSort heap = new HeapSort();

        heap.sort(randomArray);

        if(!Utilities.isArraySorted(randomArray))
              System.err.println("HeapSort failed. Array is not sorted.");
        System.out.println("HeapSort Array is sorted correctly.");
        System.out.println(Arrays.toString(randomArray));
}
```

Result:

```
HeapSort Array is sorted correctly.
[-2056832913, -1678225310, -1494192765, -742157703, -388249626, -376320165, 41554414, 421933713, 932911528, 1730658116]
```

3.1.5 Quick Sort 10 zufällige Zahlen sortieren

```
public void assertQuickSortArrayIsSorted() {
	int[] randomArray = Utilities.generateRandomLongArray(10);
	QuickSort quick = new QuickSort();

	quick.sort(randomArray);

	if(!Utilities.isArraySorted(randomArray))
		System.err.println("QuickSort failed. Array is not sorted.");
	System.out.println("QuickSort Array is sorted correctly.");
	System.out.println(Arrays.toString(randomArray));
}
```

Result:

```
QuickSort Array is sorted correctly.
[-1029462541, -980616243, -215613312, -212346814, 486631031, 1215263303, 1240244186, 1423077018, 1592288984, 2073993654]
```

3.1.6 Sorting Performance Test

```
public void assertCompareHeapAndQuickSortWith1000Elements() {
	final int ARRAY_SIZE = 1000;
	int[] randomHeapArray = Utilities.generateRandomLongArray(ARRAY_SIZE);
	int[] randomQuickArray = new int[ARRAY_SIZE];
	System.arraycopy(randomHeapArray, 0, randomQuickArray, 0, ARRAY_SIZE);
	HeapSort heap = new HeapSort();
	QuickSort quick = new QuickSort();

	long startTime = System.nanoTime();
	heap.sort(randomHeapArray);
	long endTime = System.nanoTime();
	long heapSortTimeNs = endTime - startTime;
	double heapSortTimeMS = Utilities.getMilliSecondsForNanoSeconds(heapSortTimeNs);

	startTime = System.nanoTime();
	quick.sort(randomQuickArray);
	endTime = System.nanoTime();
	long quickSortTimeNs = endTime - startTime;
	double quickSortTimeMS = Utilities.getMilliSecondsForNanoSeconds(quickSortTimeNs);

	System.out.println("Quicksort:");
	System.out.println(ARRAY_SIZE + " Elemente: \t" + quickSortTimeMS + "ms \t " +
quick.getSwapCounter() + " Swaps \t " + quick.getCompareCounter() + " Compares");
		System.out.println("Heapsort:");
		System.out.println(ARRAY_SIZE + " Elemente: \t" + heapSortTimeMS + "ms \t " +
heap.getSwapCounter() + " Swaps \t " + heap.getCompareCounter() + " Compares");

}
```

Result:

```
Quicksort:
1000 Elemente:  3.275111ms      2516 Swaps      23266 Compares
Heapsort:
1000 Elemente:  4.629097ms      10078 Swaps     40000 Compares
```