Datastructures PO2

Luca Camphuisen en Dennis Mulder

21 oktober 2017

Studentnummers
500756672 en 500751613

Klas
iS204

Samenvatting

Dit verslag gaat over de opdracht 02.

In deze opdracht moesten we diverse lijsten studenten sorteren met bubblesort en bucket sort. Vervolgens moesten we de efficientie van deze algoritmes bepalen doormiddel van de Big O notatie.

Inhoudsopgave

Hoofdstuk 1

Genereren van studenten

Voor het generen van studenten en cijfers hebben we een configurabele builder gemaakt. Deze kun je instellen op het aantal studenten, cijfer generatie methode en meer.

```
package nl.hva.dmci.ict.se.datastructures;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Random;
import java.util.function.Supplier;

/**
 * @author Luca Camphuisen & Dennis Mulder
 * @since 29-9-17
 */
public class StudentBuilder {

    private long startId = 50080001;
    private int amount = 10_000;
    private Supplier<Double> gradeGenerator;

    private StudentBuilder() {
        gradeGenerator = randomGradeSupplier();
    }

    /**
     * Deze {@link Supplier} implementatie genereert een random double met een
     * scale van 1. Cijfers zijn tussen de 1.0 en 10.0
     *
     * @return
     */
    private Supplier<Double> randomGradeSupplier() {
        return () -> {
            //genereer een double van 1 tot en met 10
            double grade = (double)ThreadLocalRandom.current().nextInt(10, 101)/10;
            //verander precision naar 1 dus dan is de formatting ##.# ipv ##.#####
            DecimalFormat doubleFormat = new DecimalFormat("##.#");
            grade = Double.parseDouble(doubleFormat.format(grade));
```

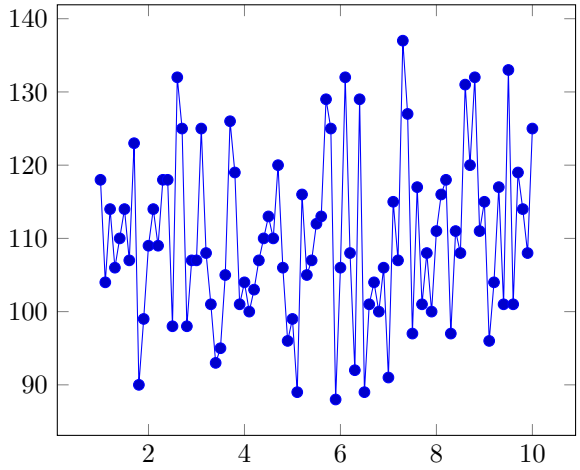
```
            return grade;
        };
    }

    public static StudentBuilder builder() {
        return new StudentBuilder();
    }

    /**
     * Zet het aantal studenten dat gemaakt moet worden.
     *
     * @param amount
     * @return
     */
    public StudentBuilder amount(int amount) {
        this.amount = amount;
        return this;
    }

    /**
     * Bij welk getal moeten we beginnen voor studenten id's
     *
     * @param startId
     * @return
     */
    public StudentBuilder startingStudentId(long startId) {
        this.startId = startId;
        return this;
    }

    /**
     * Supplier die een random getal terug geeft.
     *
     * @param grade
     * @return
     */
    public StudentBuilder gradeGenerator(Supplier<Double> grade) {
        this.gradeGenerator = grade;
        return this;
    }

    /**
     * Genereert alle studenten zoals geconfigureerd.
     *
     * @return
     */
    public Student[] build() {
        Student[] students = new Student[amount];
        String[] klassen = KlasGenerator.maakKlassen(amount);
        for (long i = startId; i < startId + amount; i++) {
            students[(int) (i - startId)] = buildSingle(i, klassen[(int) (i - startId)]);
        }
        return students;
```

```
    }

    /**
     * Maakt een enkele student aan met de gegeven informatie.
     *
     * @param id studenten id
     * @param klas de klas van de student.
     * @return een student object met een random cijfer en de gegeven
     * parameters.
     */
    private Student buildSingle(long id, String klas) {
        return new Student(id, gradeGenerator.get(), klas);
    }

}
```

Om aan te tonen dat er geen uitschieters in de cijfers zitten hebben we de volgende code geschreven:

```
    public static void gradeFrequencyGraph(Student[] students) {
        System.out.println("Cijfer freqenties:");
        Arrays.stream(students)
                .map(Student::getGrade)
                .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
                .entrySet().stream()
                .sorted(Map.Entry.comparingByKey())
                .forEach(grade -> System.out.printf("%2.1f,%d%n",
                 grade.getKey(), grade.getValue()));
        System.out.println();
    }
```

De output is een CSV geformatteerd resultaat met het cijfer en de frequentie dat deze voorkomt. Als wij een grafiek maken hiervan(met 10000 studenten) ziet deze er als volgt uit:



Op de X-as is het cijfer te zien en op de Y-as de freqentie waarmee deze voorkomt.

Wij kunnen hiermee bewijzen dat dit echt random gegenereerd is, omdat er verder geen uitschieters zijn en alles netjes verdeeld is.

Hoofdstuk 2

BubbleSort

Om een array van studenten te sorteren hebben we ten eerste een sorteer interface gemaakt. Hiermee kunnen we gemakkelijk meerdere algoritmes implementeren:

```
package nl.hva.dmci.ict.se.datastructures.sort;

import java.util.Comparator;

/**
 * Inteface waarmee je een sorteer algoritme kunt implementeren.
 * @author Luca Camphuisen & Dennis Mulder
 * @since 29-9-17
 */
public interface Sorter<T> extends Comparator<T> {

    /**
     * Sorteer een array.
     * @param unsorted ongesorteerde array.
     * @return de gesorteerde array.
     */
    T[] sort(T[] unsorted);
}
```

Vervolgens hebben we een BubbleSort implementatie gemaakt, specifiek voor de Student klasse.

```
package nl.hva.dmci.ict.se.datastructures.sort;


import nl.hva.dmci.ict.se.datastructures.Student;

/**
 * @author Luca Camphuisen & Dennis Mulder
 * @since 29-9-17
 */
public class StudentBubbleSorter implements Sorter<Student> {

    @Override
    public int compare(Student student, Student t1) {
```

```
        return student.compareTo(t1);
    }

    @Override
    public Student[] sort(Student[] array) {
        for(int i = 0; i < array.length; i++) {
            for(int secondary = 0; secondary < array.length - 1; secondary++) {
                if (compare(array[secondary], array[secondary + 1]) > 0) {
                    array = swap(array, secondary, secondary + 1);
                }
            }
        }
        return array;
    }

    /**
     * Swap value in array
     * @param array
     * @param index
     * @param secondIndex
     * @return
     */
    private Student[] swap(Student[] array, int index, int secondIndex) {
        Student s = array[index];
        array[index] = array[secondIndex];
        array[secondIndex] = s;
        return array;
    }
}
```

Vervolgens hebben wij deze methode getest op een lijst van 50 studenten doormiddel van de volgende methode. Deze zullen we later ook gebruiken voor het meten van de efficientie.

```
    private static void bubbleSortPOW(int amount, boolean print) {
        System.out.println("BubbleSort Proof(" + amount +" elementen):");
        Student[] students = StudentBuilder.builder()
                .amount(amount)
                .startingStudentId(50080001)
                .build();
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        Student[] studentsSorted = sort(students, new StudentBubbleSorter());
        stopWatch.stop();
        if(print) {
            boolean isStijgend = new RijControle().isStijgend(
               Arrays.asList(studentsSorted));
            Arrays.stream(studentsSorted).forEach(System.out::println);
            System.out.println("Is stijgend: " + isStijgend);
        }
        System.out.println(stopWatch.toString());
    }
```

Deze methodes zijn afhankelijk van de compareTo implementatie in de Student klasse.

```
    @Override
    public int compareTo(Student student) {
        //the comparison of grade, which we need to check first.
        int gradeCompare = Double.compare(grade, student.getGrade());
        //If the grades are not equal then return the calculated compare value.
        if(gradeCompare != 0) {
            return gradeCompare;
        }
        //If the grades are equal when then want to compare student id's.
        //Finally we just return the result of that comparison.
        return Long.compare(studentId, student.getStudentId());
    }
```

Bij een lijst van 50 studenten gaf het de volgende output(uitgevoerd op een Raspberry Pi):

```
BubbleSort Proof(50 elementen):
50080014;IB20001;1.0
50080020;IG20001;1.1
50080030;IB20001;1.1
50080047;IT20001;1.2
50080026;IB20001;1.5
50080040;IS20001;1.5
50080007;IN20001;2.1
50080003;IB20001;2.2
50080049;IG20001;2.2
50080006;IT20001;2.3
50080019;IS20001;2.4
50080050;IS20001;2.4
50080045;IT20001;2.7
50080016;IB20001;2.9
50080004;IS20001;3.0
50080021;IB20001;3.0
50080029;IS20001;3.0
50080018;IT20001;3.1
50080009;IN20001;3.2
50080041;IN20001;3.3
50080036;IT20001;3.7
50080035;IS20001;3.9
50080042;IS20001;4.0
50080043;IG20001;4.0
50080022;IG20001;4.3
50080044;IG20001;5.1
50080038;IN20001;5.2
50080001;IG20001;5.3
50080005;IN20001;5.7
50080017;IN20001;5.8
50080027;IT20001;5.9
50080037;IG20001;6.0
50080010;IS20001;6.1
50080031;IT20001;6.1
50080048;IB20001;6.4
50080023;IN20001;6.5
50080032;IB20001;6.5
```

```
50080039;IS20001;7.1
50080034;IT20001;7.2
50080025;IB20001;7.4
50080011;IB20001;7.7
50080002;IN20001;8.3
50080015;IN20001;8.3
50080012;IT20001;8.5
50080033;IG20001;8.6
50080008;IS20001;8.7
50080046;IT20001;8.7
50080024;IN20001;9.0
50080013;IG20001;9.5
50080028;IG20001;10.0
Is stijgend: true
Tijd: 6ms
```

Hoofdstuk 3

BucketSort de klassen

Voor bucketsort hebben wij ook de Sorter interface gebruikt. Deze klasse maakt ook gebruik van onder meer een eigen SortedLinkedList en een KlasBucket. Hier is de code van de StudentBucketSorter class:

```
package nl.hva.dmci.ict.se.datastructures.sort;

import nl.hva.dmci.ict.se.datastructures.Student;

import java.util.Comparator;
import java.util.Iterator;
import nl.hva.dmci.ict.se.datastructures.sort.bucket.KlasBucket;
import nl.hva.dmci.ict.se.datastructures.sort.bucket.SortedLinkedList;

/**
 * Bucket sorting algorithm implementation which creates a bucket for every class(klas)
 * @author Luca Camphuisen & Dennis Mulder
 * @since 12-10-17
 */
public class StudentBucketSorter implements Sorter<Student> {

    @Override
    public Student[] sort(Student[] unsorted) {
        //Generate a sorted linkedlist and scatter all students in their respective
            bucket.
        SortedLinkedList<KlasBucket> klassen = new
            SortedLinkedList(klasBucketComparator());
        for(Student student : unsorted) {
            KlasBucket bucket = getBucket(klassen, student.getKlas());
            bucket.addStudent(student);
        }

        //Inserts the sorted results back into the array causing it to be a sorted array.
        Iterator<KlasBucket> klasIterator = klassen.iterator();
        int counter = 0;
        while(klasIterator.hasNext()) {
            KlasBucket bucket = klasIterator.next();
            Iterator<Student> studentIterator = bucket.getStudents().iterator();
```

```
            while(studentIterator.hasNext()) {
                Student student = studentIterator.next();
                unsorted[counter] = student;
                counter++;
            }

        }

        return unsorted;
    }

    /**
     * Gets the bucket linked to the given klas or creates it if it does not exist.
     * @param buckets the {@link SortedLinkedList containing our buckets.
     * @param klas the klas to lookup.
     * @return an existing or a new {@link KlasBucket}
     */
    private KlasBucket getBucket(SortedLinkedList<KlasBucket> buckets, String klas) {
        Iterator<KlasBucket> klasBucketIterator = buckets.iterator();
        while(klasBucketIterator.hasNext()) {
            KlasBucket bucket = klasBucketIterator.next();
            if(bucket.getKlas().equals(klas)) {
                return bucket;
            }
        }
        KlasBucket bucket = new KlasBucket(klas);
        buckets.add(bucket);
        return bucket;
    }

    /**
     * Shouldn't be used for comparing as the bucketsort is implemented in the sort
         method.
     * @param student
     * @param t1
     * @return
     */
    @Override
    public int compare(Student student, Student t1) {
        return student.compareTo(t1);
    }

    /**
     * @return {@link Comparator} implementatie voor een {@link KlasBucket}
     */
    private Comparator<KlasBucket> klasBucketComparator() {
        return (bucket1, bucket2) -> bucket1.getKlas().compareTo(bucket2.getKlas());
    }
}
```

Deze klasse gebruikt een SortedLinkedList en een KlasBucket. Een KlasBucket wrapt een SortedLinkedList van het type Student en bevat de klasnaam als een soort van bucket identifier.

```java
package nl.hva.dmci.ict.se.datastructures.sort.bucket;

import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Een LinkedList die de elementen die worden toegevoegd automatisch sorteert.
 *
 * Deze klasse implementeert ook een {@link Iterable<T>}. Hierdoor kunnen we met
 * de Java API's door de lijst heen lopen.
 *
 * @author Luca Camphuisen & Dennis Mulder
 */
public class SortedLinkedList<ELEMENT> implements Iterable<ELEMENT> {

    private Node<ELEMENT> head = null;
    private Node<ELEMENT> tail = null;
    private int size = 0;
    private Comparator<ELEMENT> sorter;

    /**
     * Construct a sorted linkedlist. This uses the given comparator to
     * implement sorting.
     *
     * @param sorter a {@link Comparator} implementation used to sort the list
     * with.
     */
    public SortedLinkedList(Comparator<ELEMENT> sorter) {
        this.sorter = sorter;
    }

    /**
     * Method used to simplify comparisons on {@code Node#getValue}
     * The Node their values will be compared by the set {@code sorter}
     * @param node the first node to compare the value of.
     * @param node1 the second node to compare the value of.
     * @return the comparison result of node's value and node1's value.
     */
    private int compare(Node<ELEMENT> node, Node<ELEMENT> node1) {
        return sorter.compare(node.getValue(), node1.getValue());
    }

    /**
     * Method used to easily insert a {@code Node} after one another.
     * @param nodeToInsertAfter the {@code Node} which will be inserted after.
     * @param newNode the {@code Node} to insert.
     */
    private void insertAfter(Node nodeToInsertAfter, Node newNode) {
        Node<ELEMENT> iterationNode;
        //Some logic to swap and insert inbetween nodes.
        //We have to correctly set all next and previous attributes inside of the Nodes
            involved.
```

```
    if (nodeToInsertAfter == null) {
        //Set the head.
        head.setPrevious(newNode);
        iterationNode = this.head;
        head = newNode;
        head.setNext(iterationNode);
        if (size <= 1) {
            tail = iterationNode;
            tail.setNext(null);
        }
    } else if (nodeToInsertAfter.getNext() == null) {
        //When we're inserted at the last position in de list also set the tail.
        newNode.setPrevious(nodeToInsertAfter);
        nodeToInsertAfter.setNext(newNode);
        tail = newNode;
    } else {
        //Just place it after the insertion node.
        Node<ELEMENT> previous = nodeToInsertAfter;
        Node<ELEMENT> next = nodeToInsertAfter.getNext();
        previous.setNext(newNode);
        newNode.setPrevious(previous);
        newNode.setNext(next);
        next.setPrevious(newNode);
    }
    this.size++;
}

/**
 * Find the Node in the current list prior to the given new Node.
 * The prior node is found through comparing all the current Node values.
 * These are compared with the new Node value and thus we can find our closest match
     to the left.
 * The closest match to the left is chosen because we will be inserting our new Node
     after the prior Node.
 * @param newNode the {@code Node} to find the closest(leftside) Node with.
 * @return a Node which is the closest to the {@code newNode} from the left
     side(negative).
 */
private Node<ELEMENT> findPriorNode(Node<ELEMENT> newNode) {
    Node<ELEMENT> curNode = head;
    if (compare(newNode, curNode) < 0) {
        return null;
    }
    while (curNode.getNext() != null) {
        if (compare(newNode, curNode) == 0) {
            return curNode;
        } else if (compare(newNode, curNode) > 0 && compare(newNode,
             curNode.getNext()) < 0) {
            return curNode;
        }
        curNode = curNode.getNext();
    }
    return curNode;
```

```
}

/**
 * Insert a new element and put it sorted into the current linked list.
 * @param value the new element to insert.
 */
public void add(ELEMENT value) {
    Node<ELEMENT> newNode = new Node(value);
    Node<ELEMENT> nodeToInsertAfter;
    if (head == null) {
        head = newNode;
        head.setNext(tail);
        size++;
    } else {
        nodeToInsertAfter = findPriorNode(newNode);
        if (nodeToInsertAfter == null || compare(nodeToInsertAfter, newNode) != 0) {
            insertAfter(nodeToInsertAfter, newNode);
        }
    }
}

/**
 * @return Simpele iterator implementatie. Deze is gekopieerd uit opdracht
 * 1C(Deque).
 */
@Override
public Iterator<ELEMENT> iterator() {
    return new Iterator<ELEMENT>() {
        //huidige {@code Node} waar we zijn.
        private Node<ELEMENT> next = head;

        /**
         * Check of er nog {@code Node} objecten volgen.
         *
         * @return
         */
        @Override
        public boolean hasNext() {
            return next != null;
        }

        /**
         * Verkrijgt de volgende {@code Node} value. Vervolgens wordt de
         * volgende {@code Node} al klaar gezet.
         *
         * @return
         */
        @Override
        public ELEMENT next() {
            if (next == null) {
                throw new NoSuchElementException("No next element found.");
            }
            ELEMENT val = next.getValue();
```

```
            next = next.getNext();
            return val;
        }

    };
}

/**
 * Represents a single element inside of the sorted linkedlist.
 *
 * @param <T>
 */
private class Node<T> {

    /**
     * The next element/node inside of the list.
     */
    private Node<T> next;
    private Node<T> previous;
    /**
     * Value of the node which is basically the added element.
     */
    private T value;

    /**
     * Constructs a new Node.
     *
     * @param next the next {@code Node} in the list.
     * @param previous the {@code Node} before this one.
     * @param value the actual element that was added.
     */
    public Node(Node<T> next, Node<T> previous, T value) {
        this(previous, value);
        this.next = next;
    }

    /**
     * Constructs a new Node.
     *
     * @param previous the {@code Node} before this one.
     * @param value the actual element that was added.
     */
    public Node(Node<T> previous, T value) {
        this(value);
        this.previous = previous;
    }

    /**
     * Constructs a new Node without setting a next {@code Node}
     *
     * @param value the actual element that was added.
     */
    public Node(T value) {
```

```
            this.value = value;
        }

        /**
         * Gets the next {@code Node} in the list.
         *
         * @return next {@code Node} but null if this is currently the last.
         */
        public Node<T> getNext() {
            return next;
        }

        /**
         * Set the next {@code Node}
         *
         * @param next the {@code Node} which should be after the current Node.
         */
        public void setNext(Node<T> next) {
            this.next = next;
        }

        /**
         * @return {@code Node} value which is the element that was added.
         */
        public T getValue() {
            return value;
        }

        /**
         * Sets the {@code Node} value.
         *
         * @param value the element to replace the current one with.
         */
        public void setValue(T value) {
            this.value = value;
        }

        public Node<T> getPrevious() {
            return previous;
        }

        public void setPrevious(Node<T> previous) {
            this.previous = previous;
        }

    }

}
```

KlasBucket:

```
package nl.hva.dmci.ict.se.datastructures.sort.bucket;
```

```java
import java.util.Comparator;
import nl.hva.dmci.ict.se.datastructures.Student;

/**
 * Wrapper of {@link SortedLinkedList} which contains a bucket identifier(klas
     attribute).
 * @author Luca Camphuisen & Dennis Mulder
 */
public class KlasBucket {

    private String klas;
    private SortedLinkedList<Student> students;

    public KlasBucket(String klas) {
        this.klas = klas;
        this.students = new SortedLinkedList(studentIdComparator());
    }

    public void addStudent(Student student) {
        students.add(student);
    }

    /**
     * @return {@link Comparator} implementatie voor het sorteren op studentnr.
     */
    private Comparator<Student> studentIdComparator() {
        //Compare de studentnummers.
        return (student1, student2) ->
                Long.compare(student1.getStudentId(), student2.getStudentId());
    }

    public String getKlas() {
        return klas;
    }

    public SortedLinkedList<Student> getStudents() {
        return students;
    }

}
```

Om deze bucketsort te testen hebben wij weer een test methode geschreven. Deze zal ook worden gebruikt in de efficientie metingen(wederom getest op de Raspberry Pi met de -Xint JVM optie):

```
    private static void bucketSortPOW(int amount, boolean print) {

        System.out.println("BucketSort Proof(" + amount +" elementen):");
        Student[] students = StudentBuilder.builder()
                .amount(amount)
                .startingStudentId(50080001)
                .build();
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        Student[] studentsSorted = sort(students, new StudentBucketSorter());
        stopWatch.stop();
        if(print) {
            Arrays.stream(studentsSorted).forEach(System.out::println);
        }
        System.out.println(stopWatch.toString());
    }
```

Na het runnen van deze methode met 50 studenten is dit de output:

```
BucketSort Proof(50 elementen):
50080004;IB20001;1.9
50080017;IB20001;1.9
50080001;IB20001;3.1
50080047;IB20001;4.4
50080045;IB20001;6.1
50080039;IB20001;6.5
50080020;IB20001;6.7
50080006;IB20001;7.2
50080036;IB20001;7.6
50080023;IB20001;9.0
50080008;IG20001;1.2
50080029;IG20001;2.1
50080030;IG20001;3.5
50080035;IG20001;3.9
50080005;IG20001;4.5
50080002;IG20001;5.9
50080031;IG20001;6.1
50080041;IG20001;7.4
50080032;IG20001;8.1
50080049;IG20001;8.6
50080028;IN20001;1.1
50080048;IN20001;1.5
50080025;IN20001;2.1
50080026;IN20001;2.3
50080050;IN20001;4.0
50080037;IN20001;5.5
50080016;IN20001;5.8
50080027;IN20001;6.1
50080038;IN20001;7.4
```

```
50080015;IN20001;9.4
50080040;IS20001;3.2
50080012;IS20001;3.4
50080007;IS20001;6.3
50080014;IS20001;7.1
50080022;IS20001;8.1
50080043;IS20001;8.7
50080003;IS20001;9.0
50080046;IS20001;9.0
50080011;IS20001;9.2
50080033;IS20001;9.8
50080024;IT20001;2.7
50080010;IT20001;3.8
50080034;IT20001;4.7
50080009;IT20001;5.2
50080018;IT20001;5.4
50080044;IT20001;5.9
50080042;IT20001;6.1
50080013;IT20001;6.5
50080019;IT20001;8.5
50080021;IT20001;8.8
Tijd: 12ms
```

Hoofdstuk 4

Efficientie meten

Om de efficientie te meten van de sorteermethodes gebruiken wij de volgende code:

```
    private static void efficiencyMeasurement() {
        System.out.println("Efficientie meten:");

        bubbleSortPOW(500, false);
        bubbleSortPOW(1000, false);
        bubbleSortPOW(2000, false);
        bubbleSortPOW(4000, false);
        bubbleSortPOW(8000, false);
        bubbleSortPOW(16000, false);

        bucketSortPOW(500, false);
        bucketSortPOW(1000, false);
        bucketSortPOW(2000, false);
        bucketSortPOW(4000, false);
        bucketSortPOW(8000, false);
        bucketSortPOW(16000, false);
    }
```

Vervolgens voerde wij dit programma uit op de Raspberry Pi doormiddel van het command:

```
java -Xint -jar sort-students.jar > results.txt
```

Na enkele minuten was het programma klaar met de tests en konden wij results.txt ophalen doormiddel van een FTP client. Hierin stond de volgende informatie:

```
Efficientie meten:
BubbleSort Proof(500 elementen):
Tijd: 566ms
BubbleSort Proof(1000 elementen):
Tijd: 2409ms
BubbleSort Proof(2000 elementen):
Tijd: 9862ms
BubbleSort Proof(4000 elementen):
Tijd: 40290ms
BubbleSort Proof(8000 elementen):
```

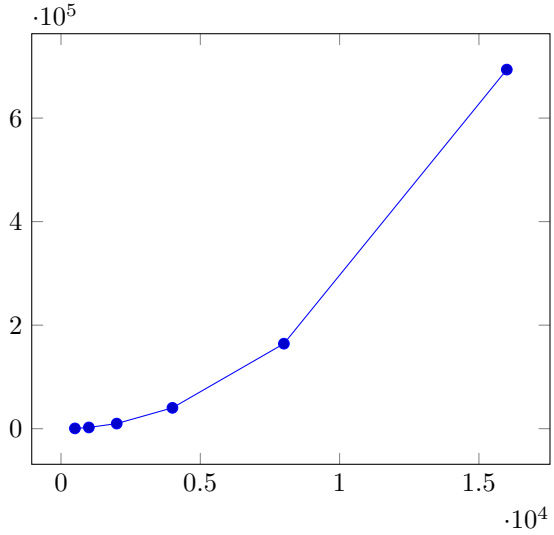
```
Tijd: 164216ms
BubbleSort Proof(16000 elementen):
Tijd: 693651ms
BucketSort Proof(500 elementen):
Tijd: 35ms
BucketSort Proof(1000 elementen):
Tijd: 81ms
BucketSort Proof(2000 elementen):
Tijd: 182ms
BucketSort Proof(4000 elementen):
Tijd: 1501ms
BucketSort Proof(8000 elementen):
Tijd: 3054ms
BucketSort Proof(16000 elementen):
Tijd: 6180ms
```

Als wij de grafiek van de bubblesort plotten kunnen wij zien dat de Big-O ongeveer gelijk is aan

$$o = n^2$$

Verder kunnen wij dit ook bewijzen door direct naar de resultaten te kijken. Bij een verdubbeling van het aantal studenten duurt het ongeveer 4 keer zo lang. Dit hoort typisch bij onze berekende Big-O.

BubbleSort grafiek (x-as = aantal elementen, y-as = tijd in ms)

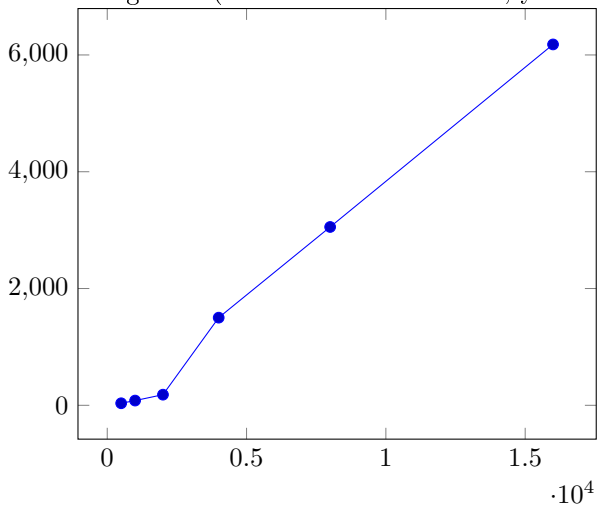


In tegenstelling tot bubblesort lijkt de Big-O van bucketsort meer op:

$$o = n$$

Als wij de grafiek van de bucketsort resultaten plotten kun je ook direct zien dat de lijn bijna lineair loopt.

BucktSort grafiek (x-as = aantal elementen, y-as = tijd in ms)



Onze metingen zijn niet precies gelijk aan de verwachte

$$o = n^2$$

of

$$o = n$$

De grootste reden hiervoor is dat onze code niet een exacte implementatie van bubble/bucketsort is. Dit komt, omdat wij ook een eigen list moesten maken. Hierdoor worden er toch net iets meer instructies op de processor uitgevoerd.