Efficiëntie van sorteeralgoritmes

Practicum 2 - HERKANSING

Bart Endhoven 500761169 & Peter Sabel (Zabel) 500759473

2017-2018

Efficiëntie van sorteeralgoritmes

Practicum 2 - HERKANSING

AUTEUR

Bart Endhoven 500761169 & Peter Sabel (Zabel) 500759473

Datum

25 januari 2018

VAK

Datastructure

Versie

3.0

© 2018 Copyright Hogeschool Amsterdam

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand, of openbaar gemaakt, in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door print-outs, kopieën, of op welke manier dan ook, zonder voorafgaande schriftelijke toestemming van de Hogeschool Amsterdam.

Inhoudsopgave

[Inleiding. 4](#_Toc504683000)

[1. Opdracht 1. 5](#_Toc504683001)

[1.1 Studentclass 5](#_Toc504683002)

[2. Opdracht 1.1 7](#_Toc504683003)

[2.1 Main. 7](#_Toc504683004)

[3. Opdracht 1.2 8](#_Toc504683005)

[3.1 Methode 1: de quicksort. 8](#_Toc504683006)

[3.2 Methode 2: Injection sort. 9](#_Toc504683007)

[3.3 Methode 3: bubble sort. 9](#_Toc504683008)

[4. Opdracht 1.3 11](#_Toc504683009)

[5. Meet de efficiëntie. 15](#_Toc504683010)

[Samenvatting. 16](#_Toc504683011)

[Bronnen 17](#_Toc504683012)

Inleiding.

In opdracht 2 gaan we aan de slag met arrays. Het sorteren van deze arrays en het tellen van het aantal dezelfde elementen. Ook itereren wij door de lijst met klassen heen zodat alle 10.000 leerlingen worden verdeeld in de 5 soorten klassen. Hierna gaan wij aan de slag met verschillende manieren van sorteren.

# Opdracht 1.

Er werd gevraagd een class student te implementeren met de variabelen studentnummer en een toets cijfer. Een student begint met een studentnummer vanaf 50080001 en de toets cijfers tussen de 1.0 en de 10.0.

## Studentclass

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

/\*\*

**\***

**\*** **@author** **Bart & Peter**

 \*/

**public** **class** Student **implements** Comparable<Student> {

**private** **final** int BEGIN\_NUMMER = 50080001;

**private** int studentnummer;

**private** double toetsCijfer;

**public** Student(int studentnummer, double toetsCijfer) {

**this**.studentnummer = BEGIN\_NUMMER + studentnummer;

**this**.toetsCijfer = toetsCijfer;

}

**public** int getStudentnummer() {

**return** studentnummer;

}

**public** void setStudentnummer(int studentnummer) {

**this**.studentnummer = studentnummer;

}

**public** double getToetsCijfer() {

**return** toetsCijfer;

}

**public** void setToetsCijfer(double toetsCijfer) {

**this**.toetsCijfer = toetsCijfer;

}

@Override

**public** **String** toString() {

**return** "Studentnummer: " + studentnummer + "\t"

+ "Cijfer: " + toetsCijfer;

}

@Override

**public** int compareTo(Student s) {

**if** (**this**.getToetsCijfer() == s.getToetsCijfer()) {

**return** **this**.getStudentnummer() - s.getStudentnummer();

} **else** **if** (**this**.getToetsCijfer() < s.getToetsCijfer()) {

**return** -1;

} **else** {

**return** 1;

}

}

**public** **static** Student[] sorterenStudenten(Student[] studentenlijst) {

Student temp;

**for** (int i = 1; i < studentenlijst.length; i++) {

**for** (int j = i; j > 0; j--) {

**if** (studentenlijst[j] != **null**) {

**if** (studentenlijst[j].compareTo(studentenlijst[j - 1]) < 0) {

temp = studentenlijst[j];

studentenlijst[j] = studentenlijst[j - 1];

studentenlijst[j - 1] = temp;

}

}

}

}

**return** studentenlijst;

}

}

# Opdracht 1.1

## Main.

**public** **static** void main(String[] args) {

**final** int AANTAL\_STUDENTEN = 16000;

Student[] studenten = **new** Student[AANTAL\_STUDENTEN];

**for** (int i = 0; i < AANTAL\_STUDENTEN; i++) {

**Random** rand = **new** **Random**();

double afronden = (rand.nextInt(91) + 10) / 10.0;

studenten[i] = **new** Student(i, afronden);

}

studenten = Student.sorterenStudenten(studenten);

**for** (Student student : studenten) {

**System**.out.println(student);

}

Map<**Double**, **Integer**> map = **new** TreeMap<>();

**for** (Student temp : studenten) {

**Integer** count = map.get(temp.getToetsCijfer());

map.put(temp.getToetsCijfer(), (count == **null**) ? 1 : count + 1);

}

**System**.out.println(map);

}

Zoals hierboven te zien worden er 16000 studenten aangemaakt met een random gegenereerd cijfer. Om te checken of de cijfers tussen 1.0 en 10.0 eerlijk verdeeld zijn heb ik een Treemap gebruikt om gemakkelijk te tellen hoeveel dezelfde elementen er in zitten.

De bovenstaande code is 10 keer uitgevoerd. En dit zijn de gemiddeldes van het aantal gebruikte cijfers:

1.0=170, 1.1=182, 1.2=163, 1.3=182, 1.4=170, 1.5=195, 1.6=147, 1.7=158, 1.8=179, 1.9=175, 2.0=175, 2.1=174, 2.2=193, 2.3=166, 2.4=156, 2.5=182, 2.6=180, 2.7=171, 2.8=184, 2.9=166, 3.0=177, 3.1=168, 3.2=167, 3.3=177, 3.4=153, 3.5=173, 3.6=188, 3.7=141, 3.8=190, 3.9=170, 4.0=178, 4.1=152, 4.2=182, 4.3=168, 4.4=190, 4.5=173, 4.6=193, 4.7=173, 4.8=190, 4.9=187, 5.0=177, 5.1=181, 5.2=181, 5.3=203, 5.4=174, 5.5=179, 5.6=176, 5.7=187, 5.8=168, 5.9=192, 6.0=177, 6.1=178, 6.2=164, 6.3=172, 6.4=162, 6.5=180, 6.6=192, 6.7=178, 6.8=161, 6.9=180, 7.0=169, 7.1=163, 7.2=149, 7.3=185, 7.4=172, 7.5=195, 7.6=197, 7.7=187, 7.8=174, 7.9=176, 8.0=175, 8.1=191, 8.2=166, 8.3=175, 8.4=166, 8.5=177, 8.6=146, 8.7=173, 8.8=196, 8.9=201, 9.0=182, 9.1=186, 9.2=154, 9.3=168, 9.4=174, 9.5=164, 9.6=190, 9.7=190, 9.8=158, 9.9=194, 10.0=187

Zoals hier te zien valt is er geen cijfer dat ver onder en ver boven de rest uit steekt. De cijfers worden tussen de 140 en 210 keer gebruikt.

# Opdracht 1.2

In deze opdracht werd er gevraagd om de array vol met studenten en cijfers te sorteren op de cijfers. Als studenten hetzelfde cijfer hebben worden ze gesorteerd op studentnummer.

Aangezien dit de herkansing is, hebben wij verschillende manieren geïmplementeerd om de studenten te sorteren.

## Methode 1: de quicksort.

**public** **static** Student[] sorterenStudenten(Student[] studentenlijst) {

Student temp;

**for** (int i = 1; i < studentenlijst.length; i++) {

**for** (int j = i; j > 0; j--) {

**if** (studentenlijst[j] != **null**) {

**if** (studentenlijst[j].compareTo(studentenlijst[j - 1]) < 0) {

temp = studentenlijst[j];

studentenlijst[j] = studentenlijst[j - 1];

studentenlijst[j - 1] = temp;

}

}

}

}

**return** studentenlijst;

}

@Override

**public** int compareTo(Student s) {

**if** (**this**.getToetsCijfer() == s.getToetsCijfer()) {

**return** **this**.getStudentnummer() - s.getStudentnummer();

} **else** **if** (**this**.getToetsCijfer() < s.getToetsCijfer()) {

**return** -1;

} **else** {

**return** 1;

}

}

**public** int compare(Student student) {

**return** **this**.getStudentnummer() - student.getStudentnummer();

}

## Methode 2: Injection sort.

**public** **static** Student[] insertionSort(Student[] toSort) {

int N = toSort.length;

//voor elke waarde i in de array...

**for** (int i = 1; i < N; i++) {

//... kijk of er een waarde j voor de waarde i is die lger gesorteerd word...

**for** (int j = i; j > 0 && toSort[j].compareTo(toSort[j - 1]) < 0; j--) {

//... en switch ze van plaats.

exchange(toSort, j, j - 1);

}

}

**return** toSort;

}

**private** **static** void exchange(Student[] toSort, int first, int second) {

Student studentToBeExchanged = toSort[first];

toSort[first] = toSort[second];

toSort[second] = studentToBeExchanged;

}

## Methode 3: bubble sort.

@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;

}

Al deze methodes hier boven genereren en sorteren een lijst met studenten, gesorteerd op het student nummer. In de volgende fase gaan we een klas toevoegen aan de student en daarop sorteren.

# Opdracht 1.3

Voor deze opdracht is er een Klas element toegevoegd aan de studentklas. Een student krijgt dus direct bij het genereren van de lijst een klas toegewezen. Deze klas wordt opgemaakt in de klasgenerator die al voor ons was aangemaakt.

**private** **final** int BEGIN\_NUMMER = 50080001;

**private** int studentnummer;

**private** double toetsCijfer;

**private** String klas;

**public** String getKlas() {

**return** klas;

}

**public** void setKlas(String klas) {

**this**.klas = klas;

}

**public** Student(int studentnummer, double toetsCijfer, String klas) {

**this**.studentnummer = BEGIN\_NUMMER + studentnummer;

**this**.toetsCijfer = toetsCijfer;

**this**.klas = klas;

}

In de main wordt dus direct een klas meegegeven.

**public** **static** void main(String[] args) {

**final** int AANTAL\_STUDENTEN = 50;

Student[] studenten = **new** Student[AANTAL\_STUDENTEN];

**String**[] klassen = KlasGenerator.maakKlassen(AANTAL\_STUDENTEN);

**for** (int i = 0; i < AANTAL\_STUDENTEN; i++) {

**Random** rand = **new** **Random**();

double afronden = (rand.nextInt(91) + 10) / 10.0;

studenten[i] = **new** Student(i, afronden, klassen[i]);

}

Nu hebben we een lijst met studenten die ook een klas hebben. We gaan nu via een bucketsort de klassen ook sorteren.

**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**

     \*/

**private** Comparator<Student> studentIdComparator() {

//Compare de studentnummers.

**return** (student1, student2)

-> **Long**.compare(student1.getStudentnummer(), student2.getStudentnummer());

}

**public** **String** getKlas() {

**return** klas;

}

**public** SortedLinkedList<Student> getStudents() {

**return** students;

}

}

**public** **class** Bucket **implements** Comparable<Bucket> {

**private** **String** klas;

**private** Queue<Student> studenten;

**public** Bucket(**String** klas) {

**this**.klas = klas;

**this**.studenten = **new** Queue<>();

}

**public** **String** getKlas() {

**return** klas;

}

**public** void setKlas(**String** klas) {

**this**.klas = klas;

}

**public** Queue<Student> getStudenten() {

**return** studenten;

}

**public** void setStudenten(Queue<Student> studenten) {

**this**.studenten = studenten;

}

**public** void voegStudentInBucket(Student student) {

**this**.studenten.add(student);

}

@Override

**public** int compareTo(Bucket other) {

**if** (**this**.getKlas().compareTo(other.getKlas()) < 0) {

**return** -1;

} **else** **if** (**this**.getKlas().equals(other.getKlas())) {

**return** 0;

} **else** {

**return** 1;

}

}

@Override

**public** **String** toString() {

**String** returnString = "";

returnString += "Klas: " + klas;

**for** (Student student : studenten) {

returnString += "\nStudent: " + student.getStudentnummer()

+ " Cijfer " + student.getToetsCijfer();

}

**return** returnString + "\n";

}

}

**public** **class** StudentBucketSorter **implements** Sorter<Student> {

@Override

**public** Student[] sort(Student[] unsorted) {

//Generate een sorted linkedlist .

SortedLinkedList<KlasBucket> klassen = **new** SortedLinkedList(klasBucketComparator());

**for** (Student student : unsorted) {

KlasBucket bucket = getBucket(klassen, student.getKlas());

bucket.addStudent(student);

}

//loop door de linkedlist heen om er wee een array van te maken.

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;

}

/\*\*

**\***

**\*** **@param** buckets

**\*** **@param** klas

**\*** **@return**

     \*/

**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;

}

/\*\*

**\***

**\*** **@param** student

**\*** **@param** t1

**\*** **@return**

     \*/

@Override

**public** int compare(Student student, Student t1) {

**return** student.compareTo(t1);

}

/\*\*

**\***

**\*** **@return**

     \*/

**private** Comparator<KlasBucket> klasBucketComparator() {

**return** (bucket1, bucket2) -> bucket1.getKlas().compareTo(bucket2.getKlas());

}

}

# Meet de efficiëntie.

Genereer (in een loop) verschillende lijsten van 500, 1.000, 2.000, 4.000, 8.000 en 16.000 studenten met hun resultaten. Sorteer vervolgens deze lijst op beide manieren. Gebruik hiervoor de Raspberry PI.

Presenteer de resultaten van je experimenten in tabellen (en eventueel grafieken). In de presentatie van les 4 zie je vergelijkbare tabellen en grafieken. Bereken op basis van je tests de (tijd)efficiëntie van je implementaties en maak daarbij gebruik van de big-O. Laat zien hoe je de big-O hebt berekend. Vergelijk ook je praktische resultaten met de theoretische big-O van de algoritmes. Als er verschillen zijn, beredeneer dan hoe dat komt.

Omdat wij helaas beide niet beschikten over een Raspberry tijdens het testen hebben we gekozen om dit via NetBeans te doen.

Wij hebben de tijden gemeten door iedere keer één sample size te testen en de runtime over te nemen. Om een eerlijke tijd te meten hebben we iedere sample size drie keer getest en het gemiddelde van deze drie testen genomen. Hiermee voorkomen we dat een test veel sneller uitvalt omdat deze per toeval een al bijna gesorteerde dataset had gegenereerd.

Aangezien de tijdsmetingen niet helemaal naar behoren werden weergeven hebben wij wat extra testjes gedaan met grotere getallen dan dat in de opdracht stond.

Quick sort.

Aantal studenten Metingen Gemiddelde

500 [1.982s, 2.195s, 1.911s] 2.029s

1.000 [2.378s, 1.510s, 2.334s] 2.074s

2.000 [1.940s, 1.424s, 1.383s] 1.582s

4.000 [2.009s, 1.988s, 2.202s] 2.066s

8.000 [3.173s, 2.520s, 2.597s] 2.763s

16.000 [3.181s, 3.699s, 3.760s] 3.547s

32.000 [12.427s, 11.417s, 11.819s] 11.887s

64.000 [29.955s, 30.958s, 29.870s] 30.261s

BIG O van quicksort komt hiermee uit op: O (n log(n))

Bubble sort.

Aantal studenten Metingen Gemiddelde

500 [2.216s, 1.835s, 1.885s] 1.979s

1.000 [2.405s, 1.369s, 1.855s] 1.876s

2.000 [2.917s, 1.486s, 2.536s] 2.313s

4.000 [2.796s, 2.227s, 1.709s] 2.244s

8.000 [3.391s, 3.630s, 3.521s] 3.514s

16.000 [6.148s, 5.915s, 5.778s] 5.947s

32.000 [16.856s, 17.783s, 17.101s] 17,247s

64.000 [65.965s, 65.204s, 65.512s] 65,560s

BIG O van bubble sort valt erg tegen en is uitgekomen op: O(n^2)

Injection sort.

Aantal studenten Metingen Gemiddelde

500 [2.093s, 1.338s, 2.260s] 1.897s

1.000 [1.430s, 1.871s, 1.890s] 1.730s

2.000 [2.421s, 1.951s, 1.447s] 1.940s

4.000 [2.465s, 2.055s, 2.690s] 2.403s

8.000 [2.247s, 1.841s, 2.506s] 2.198s

16.000 [4.068s, 3.852s, 3.688s] 3.869s

32.000 [9.968s, 9.830s, 9.539s] 9,779s

64.000 [18.168s, 18.421s, 18.251s] 18,280s

BIG O van injection sort is een mooie geworden: O(log n)

Samenvatting.

Tijdens dit practicum hebben wij verschillende manieren van sorteren laten zien. We hebben getest welke manier het snelst is en wat de bucketsort functie inhoud.

Het sorteren van arrays kan dus op allerlei manieren. Elke manier is anders. Voor sommige producten is de ene weer sneller dan de andere.

De injection sort werkte bij ons het snelste. Hier hebben wij een Big O van log(n) gekregen.

Bronnen

(mkyoung, 2012)