Hogeschool van Amsterdam

Sorting & Searching

Efficiëntie van geavanceerde sorteeralgoritmes

Practicum 1

Author:
Robert Bakker

Studentnummer:
500689284

Klas:
IVSE4

Author:
Mark van der Steenhoven

Studentnummer:
500693745

Klas:
IVSE4

Blok 2, 2016 - 2017

Inhoudsopgave

a Resultaten van studenten sorteren met een advanced sort

Voor deze opdracht wordt gebruik gemaakt van de volgende Student-klasse:

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

    private String group;
    private int studentNumber;
    private float grade;

    public Student(String group, int studentNumber, float grade) {
        this.group = group;
        this.studentNumber = studentNumber;
        this.grade = grade;
    }

    // Getters and Setters

    @Override
    public int compareTo(Student that) {
        if (this.grade < that.getGrade()) return 1;
        if (this.grade > that.getGrade()) return -1;
        if(this.studentNumber > that.getStudentNumber()) return -1;
        if(this.studentNumber < that.getStudentNumber()) return 1;

        return 0;
    }
}
```

Waarbij het met name gaat om de implementatie van de Comparable-interface. Als twee Student-objecten met elkaar worden vergeleken, wordt in eerste instantie gekeken naar het cijfer (grade). Mochten de cijfers niet groter of kleiner zijn dan elkaar oftwel gelijk zijn, wordt er teruggevallen op een vergelijking op het studentnummer.

a.1 Quicksort implemenatie

Hieronder de Quicksort implementatie voor de experimenten in paragraaf a.2 met zoveel mogelijk uitleg in het codecommentaar:

```
// De quicksort accepteert een lijst van objecten met een Comparable
// interface (bijv. de Student-klasse), het beginpunt vanaf links,
// en het beginpunt vanaf rechts
private void quicksort(Comparable[] list, int low, int high) {

    // Neem het middelpunt van de array als spil (draaipunt)
    Comparable pivot = list[low + (high - low) / 2];

    int i = low; // linkerkant
    int j = high; // rechterkant

    while (i <= j) {
        // Wanneer object vanaf links kleiner is dan de spil
        // Verschuif naar de volgende in de linkerlijst
        while (list[i].compareTo(pivot) < 0) i++;

        // Wanneer object vanaf rechts groter is dan de spil
        // Verschuif naar de volgende in de rechterlijst
        while (list[j].compareTo(pivot) > 0) j--;

        // Als er een index van de linkerlijst is gevonden, met een waarde
        // die groter is dan de spil, en een index in de rechterlijst met
        // een waarde die kleiner is dan de spil, moeten de 2 waarden
        // worden omgedraaid
        if (i <= j) {
            Comparable temp = list[i];
            list[i] = list[j];
            list[j] = temp;
            i++;
            j--;
        }
    }
    // Hetzelfde voor de rest van de linkerlijst
    if (low < j) {
        quicksort(list, low, j);
    }
    // en voor de rechterlijst
    if (high > i) {
        quicksort(list, i, high);
    }
}
```

a.2 Resultaten experiment

Het experiment bestond uit het meten van de tijd die de Quicksort implementatie in de vorige paragraaf nodig heeft om lijsten met verschillende aantallen studenten te sorteren.

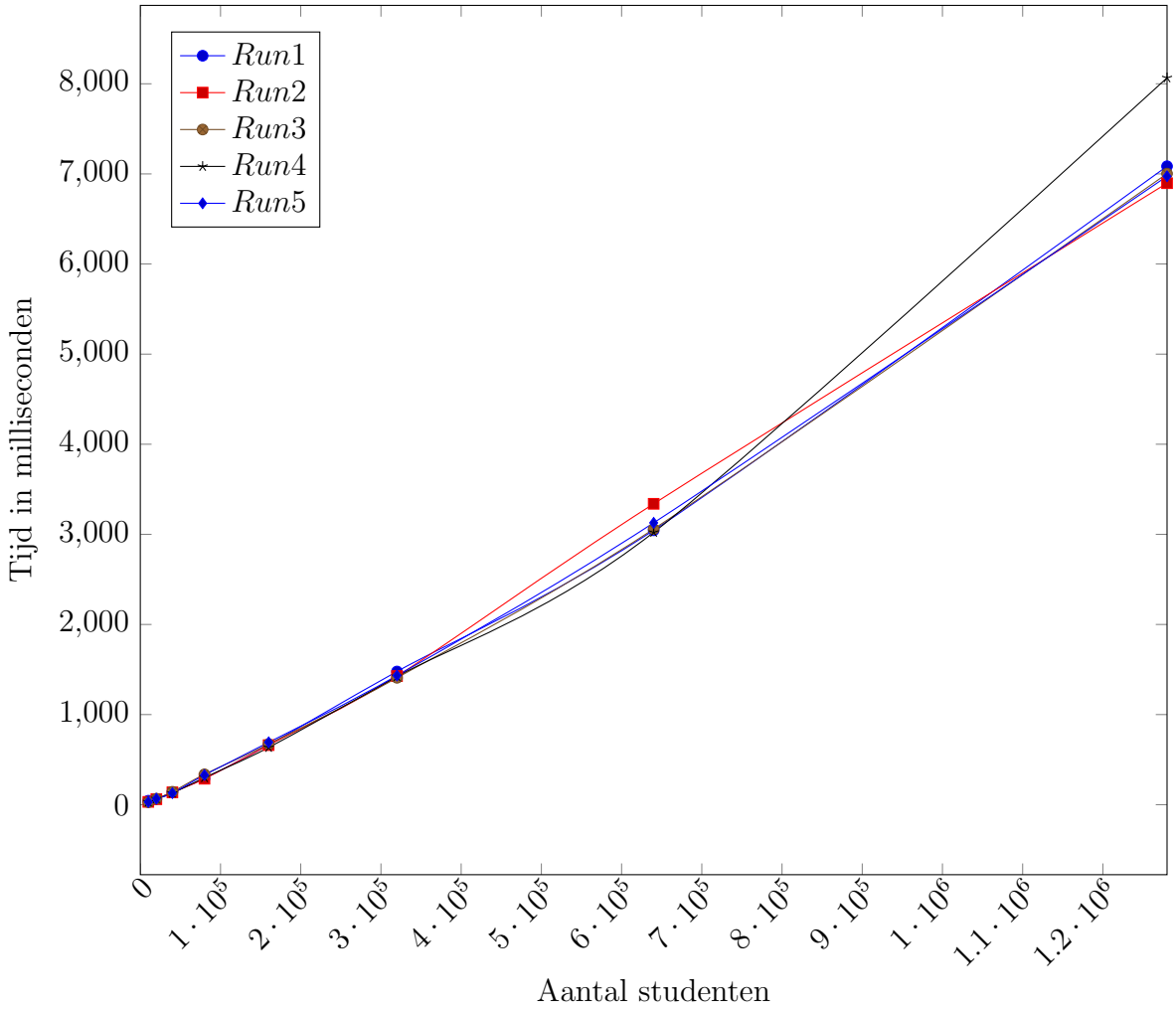
Om meetresultaten accurater te maken werd Java met de VM optie -Djava.compiler=NONE uitgevoerd om optimalisaties uit te schakelen. Die optie voorkomt dat Java het programma met extra optimalisaties gaat uitvoeren, wat anderszijds invloed zou hebben op de meetresultaten.

```
int[] numberOfStudents = {10000, 20000, 40000, 80000,
160000, 320000, 640000, 1280000};
// N_EXPERIMENTS = 5
long[][] quicksortResults = new long[numberOfStudents.length][N_EXPERIMENTS];

for (int i = 0; i < numberOfStudents.length; i++) {
    for (int j = 0; j < N_EXPERIMENTS; j++) {
        ResultList list = StudentListGenerator.generate(numberOfStudents[i]);
        list.shuffle(); // StdRandom.shuffle()
        long startTime = System.currentTimeMillis();
        list.quicksort(); // De Quicksort implementatie
        long endTime = System.currentTimeMillis();
        quicksortResults[i][j] = (endTime - startTime);
    }
}
```

Het experiment is meerdere keren uitgevoerd en daarvan is het gemiddelde genomen. Dit zijn de resultaten:

Aantal studenten	Run 1	Run 2	Run 3	Run 4	Run 5	Gemiddelde
10000	44	31	27	30	30	32
20000	64	61	66	66	66	64
40000	139	137	143	127	128	134
80000	291	288	337	301	329	309
160000	665	661	673	635	692	665
320000	1478	1427	1408	1422	1433	1433
640000	3045	3339	3059	3022	3128	3118
1280000	7085	6897	7003	8066	6974	7205


Run1
Run2
Run3
Run4
Run5
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
Tijd in milliseconden
0
1 · 10^5
2 · 10^5
3 · 10^5
4 · 10^5
5 · 10^5
6 · 10^5
7 · 10^5
8 · 10^5
9 · 10^5
1 · 10^6
1.1 · 10^6
1.2 · 10^6
Aantal studenten

a.3 Efficiëntie

t is de gemiddelde tijd per aantal studenten n. Hier berekenen de gemiddelde factor waarmee de tijd groeit als het aantal studenten verdubbelt:

t	t+1 / t
32	-
64	64/32 = 2
134	134/64 = 2.093
309	309/134 = 2.306
665	665/309 = 2.152
1433	1433/665 = 2.154
3118	3118/1433 = 2.176
7085	7085/3118 = 2.272

Bij verdubbeling van het aantal studenten wordt de tijd gemiddeld:

(2+2.093+2.306+2.152+2.154+2.176+2.172) / 7 = 2.15042857143

zo groot.

(2 log 2.15042857143) / (2 log 2) = 1.10462421156...

$2^{1.10462421156} = 2.15042857143$

Als in de tabel te zien, wanneer n (aantal studenten) verdubbeld, verdubbeld ongeveer de tijd ook mee.

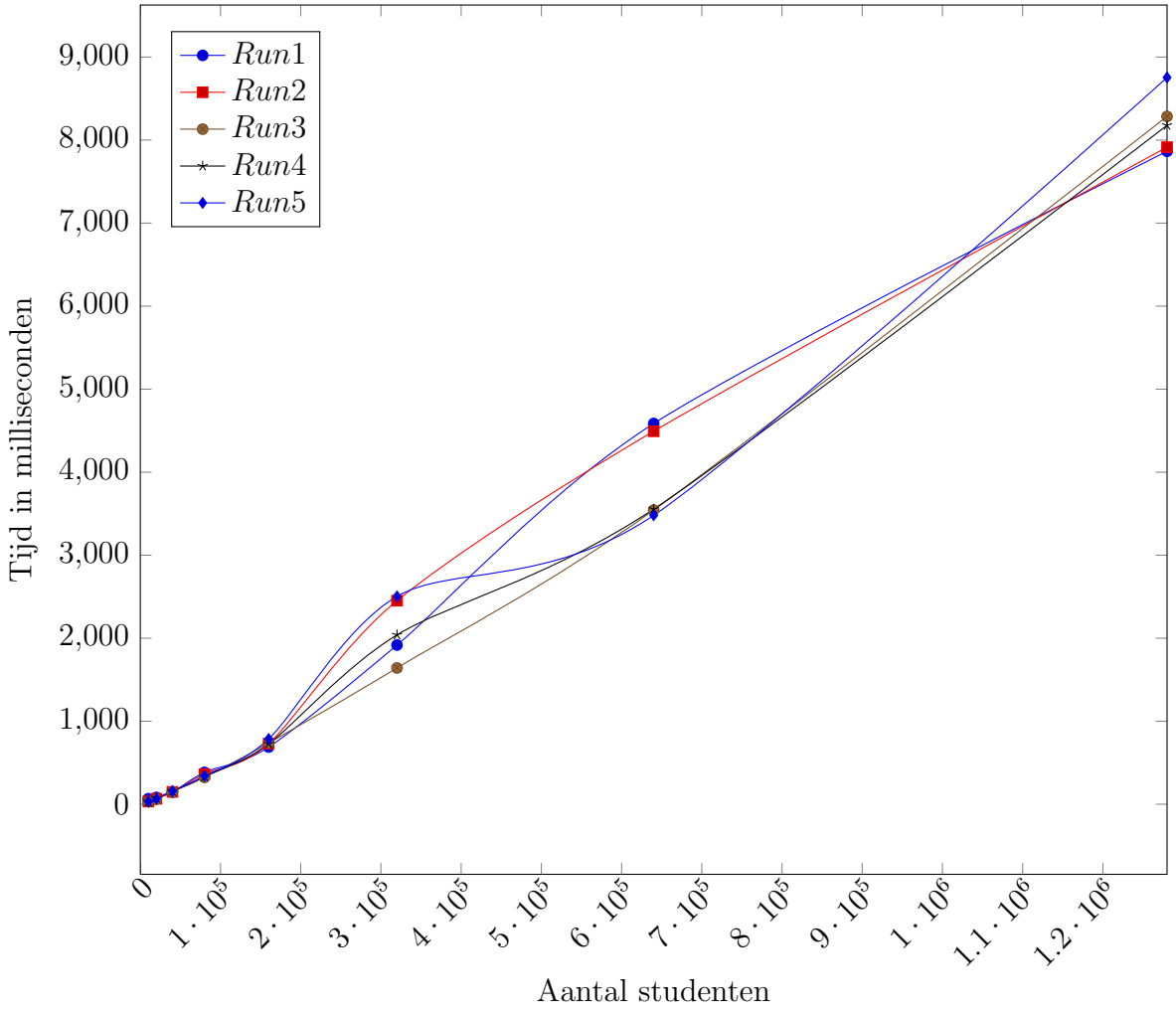
De Big-O is ongeveer $O(n^{1.10462421156})$

b Verbetering toevoegen aan algoritme

b.1 Experiment

Hetzelfde als het vorige experiment, maar dan met het Median-of-3 algoritme vanuit het boek.

Aantal studenten	Run 1	Run 2	Run 3	Run 4	Run 5	Gemiddelde
10000	67	33	30	29	34	38
20000	82	67	61	75	68	70
40000	146	149	150	158	158	152
80000	384	363	324	333	341	349
160000	689	730	738	727	787	734
320000	1918	2452	1642	2040	2504	2111
640000	4586	4493	3543	3555	3480	3931
1280000	7865	7915	8285	8178	8754	8199



b.2 Efficiëntie

t is de gemiddelde tijd per aantal studenten n. Hier berekenen de gemiddelde factor waarmee de tijd groeit als het aantal studenten verdubbelt:

t	t+1 / t
38	-
70	70/38 = 1.842
152	152/70 = 2.171
349	349/152 = 2.296
734	734/349 = 2.103
2111	2111/734 = 2.876
3931	3931/2111 = 1.862
8199	8199/3931 = 2.085

Bij verdubbeling van het aantal studenten wordt de tijd gemiddeld:

(1.842+2.171+2.296+2.103+2.876+1.862+2.085)/7 = 2.17642857143

zo groot.

(2 log 2.17642857143) / (2 log 2) = 1.12196267286...

$2^{1.12196267286} = 2.17642857143$

Als in de tabel te zien, wanneer n (aantal studenten) verdubbeld, verdubbeld ongeveer de tijd ook mee.

De Big-O is ongeveer $O(n^{1.12196267286})$

Verwaarloosbaar verschil met het vorige experiment.

c Resultaten in een Binary Search Tree en implementatie van rank()

c.1 Duplicaten

Om duplicaten(studenten met hetzelfde cijfer) in de BST te kunnen plaatsen, krijgt de Node binnen de BST in plaats van 1 waarde, een lijst van waardes:

```
private class Node {

    private Key key;
    private List<Value> val = new LinkedList<Value>(); // Lijst i.p.v. alleen Value
    private Node left, right;
    private int N;

    public Node(Key key, Value val, int N) {
        this.key = key;
        this.val.add(val); // Voeg toe aan lijst
        this.N = N;
    }
}
```

Het probleem met de implementatie uit het boek is dat wanneer, via de put() methode, er 2 Nodes worden toegevoegd met dezelfde Key, de Value wordt overschreven. Dit is opgelost door het aan de List toe te voegen:

```
private Node put(Node x, Key key, Value val) {
    if (x == null) {
        return new Node(key, val, 1);
    }
    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        x.left = put(x.left, key, val);
    } else if (cmp > 0) {
        x.right = put(x.right, key, val);
    } else {
        // Voeg toe aan lijst i.p.v. overschrijven single value
        x.val.add(val);
    }
    x.N = size(x.left) + size(x.right) + 1;
    return x;
}
```

En de get() method geeft nu in plaats van 1 waarde, een lijst terug:

```
private List<Value> get(Node x, Key key) {
    if (x == null) {
        return null;
    }
    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        return get(x.left, key);
    } else if (cmp < 0) {
        return get(x.right, key);
    } else {
        return x.val;
    }
}
```

c.2 Rank()

c.2.1 BST implementatie rank()

```
 private int rank(Key key, Node x) {
        if (x == null) {
            return 0;
        }
        int cmp = key.compareTo(x.key);
        if (cmp < 0) {
            return rank(key, x.left);
        } else if (cmp > 0) {
            return x.val.size() + size(x.left) + rank(key, x.right);
        } else {
            return size(x.left);
        }
    }
```

c.2.2 Voorbeeld data voor rank

De input is een lijst van studenten bestaande uit een cijfer en studentnummer. De input set bestaat uit 16 studenten en de cijfers aan de hand van het voorbeeld in de opdrachtomschrijving. Op deze list voeren wij de rank method uit om een overzicht te krijgen hoeveel studenten lager dan een bepaald cijfer hebben behaald.

```
float[] testGrades = {10f, 9f, 9f, 8f, 8f, 8f,
 7f, 7f, 6f, 6f, 6f, 6f, 6f, 5f, 3f, 2f};

// Maak een lijstje van studenten aan
Student[] studentList =
   StudentListGenerator.generate(testGrades.length).getList();

// Geef de studenten de test waardes als cijfer
for (int i = 0; i < testGrades.length; i++) {
    studentList[i].setGrade(testGrades[i]);
}
BST<Float, Integer> bst = new BST<>();
// Shuffle met als doel de BST gebalanceerder te maken
StdRandom.shuffle(studentList);
for (Student s : studentList) {
    bst.put(s.getGrade(), s.getStudentNumber());
}

// Rank elk cijfer
for (int i = 1; i <= 10; i++) {
    System.out.println("Grade: " + i + ", rank: " + bst.rank((float) i));
}
```

Output

Grade: 1, rank: 0
Grade: 2, rank: 0
Grade: 3, rank: 1
Grade: 4, rank: 2
Grade: 5, rank: 2
Grade: 6, rank: 3
Grade: 7, rank: 8
Grade: 8, rank: 10
Grade: 9, rank: 13
Grade: 10, rank: 15