

Übungen AD 2

Patrick Bucher

26.04.2017

1 Quicksort – theoretisch durchgespielt

a)

Erster Durchgang:

12 10 52₁ 9 77 23 18 52₂ 11 25 8 5 17

12 10 5 9 8 11 17 52₂ 23 25 77 52₁ 18

Zweiter Durchgang:

12 10 5 9 8 11 | 17 | 52₂ 23 25 77 52₁ 18

8 10 5 9 11 12 | 17 | 18 23 25 77 52₁ 52₂

Dritter Durchgang:

9 10 5 9 | 11 12 17 18 | 23 25 77 52₁ 52₂

8 5 9 10 | 11 12 17 18 | 23 25 52₁ 52₂ 77

Vierter Durchgang:

8 5 | 9 10 11 12 17 18 | 23 25 52₁ | 52₂ 77

5 8 | 9 10 11 12 17 18 | 23 25 52₁ | 52₂ 77

Fünfter Durchgang

5 8 9 10 11 12 17 18 | 23 25 | 52₁ 52₂ 77

5 8 9 10 11 12 17 18 23 25 52₁ 52₂ 77

b)

Die Zahl 52₁ war nach dem ersten Durchgang rechts von 52₂. Dass die Reihenfolge im zweiten Durchgang noch einmal (und zwar endgültig) änderte, ist reiner Zufall. Quicksort arbeitet *instabil*.

c)

Beim ersten Durchgang kämen 12 (Index 0), 18 (Index 6) und 17 (Index 12) in Frage. Dadurch würde erneut 17 als Trennelement fungieren.

Beim zweiten Durchgang würde es links wiederum genau gleich ablaufen, rechts würde aber mit 25 ein anderes Element verwendet werden. Das könnte die Sortierung etwas beschleunigen und evtl. einen fünften Durchgang ersparen.

2 Quicksort – klassisch programmiert

a)

```
public static void quickSort(Character[] data, int left, int right) {
    int up = left;
    int down = right - 1;
    char t = data[right];
    do {
        while (data[up] < t) {
            up++;
        }
        while (data[down] >= t && down > up) {
            down--;
        }
        if (up >= down) {
            break;
        }
        swap(data, up, down);
    } while (true);
    swap(data, up, right);
    if (left < up - 1) {
        quickSort(data, left, up - 1);
    }
    if (right > up + 1) {
        quickSort(data, up + 1, right);
    }
}
```

Testfall:

```
@Test
public void testQuickSort() {
    final int n = 200_000;
    Character data[] = SortingUtils.generateRandomCharArray(n, 'A', 'Z');
```

```

Sort.quickSort(data, 0, data.length - 1);
boolean sorted = SortingUtils.isSorted(Arrays.asList(data), true);
Assert.assertTrue(sorted);
}

```

b)

```

public static void quickSort(Character[] data) {
    quickSort(data, 0, data.length - 1);
}

```

c)

```

public static Character[] randomChars(int size, int min, int max) {
    Random random = new Random(System.currentTimeMillis());
    Character array[] = new Character[size];
    for (int i = 0; i < size; i++) {
        array[i] = (char) (random.nextInt(max - min + 1) + min);
    }
    return array;
}

```

d)

| Elemente (n) | Messung (ms) |
|--------------|--------------|
| 1000 | 2 |
| 5000 | 7 |
| 10'000 | 12 |
| 50'000 | 35 |
| 100'000 | 133 |
| 500'000 | 2993 |
| 1'000'000 | 11'888 |

Beispiel: Um welchen Faktor müsste eine Sortierung mit 1'000'000 Elementen länger dauern als eine Sortierung mit 500'000 bzw. 100'000 Elementen?

$$(1'000'000 * \log 1'000'000) / (500'000 * \log 500'000) = 2.1$$

$$(1'000'000 * \log 1'000'000) / (100'000 * \log 100'000) = 12$$

Realität:

```
11'888 / 2993 = 3.97
11'888 / 133 = 89.4
```

Das Laufzeitverhalten scheint eher $O(n^2)$ zu entsprechen (eine Verdoppelung der Elemente führt zu einer Vervierfachung der Laufzeit; eine Verzehnfachung der Elemente erhöht die Laufzeit ca. um Faktor 90).

3 Quick-Insertion-Sort

a)

Zur eigentlichen Methode, die zusätzlich einen Parameter *m* hat (Schwellenwert, unter dem der Insertion-Sort verwendet werden soll), implementiere ich wiederum eine Hilfsmethode.

```
public static void quickInsertionSort(Character[] data, int m) {
    quickInsertionSort(data, 0, data.length - 1, m);
}

static void quickInsertionSort(Character[] data, int l, int r, int m) {
    // bestehender Algorithmus
    // ...
    // veränderte Rekursionsanweisung
    if (left < up - 1) {
        int from = left;
        int to = up - 1;
        if (to - from > m) {
            quickInsertionSort(data, from, to, m);
        } else {
            SimpleSorting.insertionSort(data, from, to);
        }
    }
    if (right > up + 1) {
        int from = up + 1;
        int to = right;
        if (to - from > m) {
            quickInsertionSort(data, from, to, m);
        } else {
            SimpleSorting.insertionSort(data, from, to);
        }
    }
}
```

b)

Ich sortiere eine Million Zeichen mit verschiedenen m -Werten. Dabei erhalte ich folgende Laufzeiten:

| m | Zeit (ms) |
|------|-----------|
| 5 | 9420 |
| 10 | 9278 |
| 15 | 9259 |
| 20 | 9361 |
| 25 | 9269 |
| 30 | 9258 |
| 40 | 9267 |
| 50 | 9263 |
| 75 | 9267 |
| 100 | 9289 |
| 125 | 9265 |
| 150 | 9260 |
| 200 | 9300 |
| 250 | 9259 |
| 500 | 9255 |
| 1000 | 9410 |

Bei einer Datenmenge von einer Million Zeichen scheinen sinnvolle m -Werte von 10 bis 500 zu liegen.

c)

Mit $m = 25$ kann ich beim Quick-Insertion-Sort (QIS) gegenüber dem Quick-Sort (QS) keinen Laufzeitvorteil feststellen:

| n | QS (ms) | QIS (ms) |
|---------|---------|----------|
| 1000 | 2 | 2 |
| 5000 | 6 | 6 |
| 10000 | 4 | 15 |
| 50000 | 35 | 33 |
| 100000 | 134 | 126 |
| 500000 | 3080 | 3043 |
| 1000000 | 11866 | 12101 |
| 2000000 | 47465 | 48360 |