Übungen 9

Patrick Bucher

25.04.2017

1 Quicksort – theoretisch durchgespielt

a)

Erster Durchgang:

12 10 52_1 9 77 23 18 52_2 11 25 8 5 *17*

12 10 5 9 8 11 *17* 52_2 23 25 77 52_1 18

Zweiter Durchgang:

12 10 5 9 8 *11* | 17 | 52_2 23 25 77 52_1 *18*

8 10 5 9 *11* 12 | 17 | *18* 23 25 77 52_1 52_2

Dritter Durchgang:

9 10 5 *9* | 11 12 17 18 | 23 25 77 52_1 *52_2*

8 5 *9* 10 | 11 12 17 18 | 23 25 52_1 *52_2* 77

Vierter Durchgang:

8 *5* | 9 10 11 12 17 18 | 23 25 *52_1* | 52_2 77

5 8 | 9 10 11 12 17 18 | 23 25 *52_1* | 52_2 77

Fünfter Durchgang

5 8 9 10 11 12 17 18 | 23 *25* | 52_1 52_2 77

5 8 9 10 11 12 17 18 23 *25* 52_1 52_2 77

b)

Die Zahl 52_1 war nach dem ersten Durchgang rechts von 52_2. 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 Verfierfachung der Laufzeit; eine Verzehnfachung der Elemente erhöht die Laufzeit ca. um Faktor 90).