Folge 13 - Quicksort

13.1 Grundprinzip des Quicksort

Schritt 1

Gegeben ist ein unsortierter Array von ganzen Zahlen. Ein Element des Arrays wird nun besonders behandelt, es wird nämlich zum so genannten **Median** befördert.

Im Grunde ist es egal, welches der Arrayelemente zum Median wird, in unterschiedlichen Büchern findet man verschiedene Verfahren. Häufig wird zum Beispiel das letzte Element des Arrays genommen, in diesem Skript nehmen wir das mittlere Element des Arrays bzw. das Element links von der Mitte (bei geradzahliger Arraylänge).

43	20	10	12	60	50	17	80	93	54	67	18

Das Element 50 wäre hier ein geeigneter Median, das Element liegt ziemlich in der Mitte des Arrays, und sein Wert liegt ungefähr in der Mitte des Zahlenbereichs. Wie man einen geeigneten Median findet, ist zwar eine Wissenschaft für sich, aber im Augenblick nicht unser Problem; hier geht es nur um das Grundprinzip des Quicksort.

Durch den gewählten Median wird der Array in zwei **Partitionen** zerlegt, in eine linke und in eine rechte:

43	20	10	12	60	50	17	80	93	54	67	18
linke Partition					Med.	rechte Partition					

Schritt 2

Alle Elemente der linken Partition, die *größer* sind als der Median, werden in die rechte Partition verfrachtet. Und alle Elemente der rechten Partition, die *kleiner* sind als der Median, werden in die linke Partition verfrachtet:

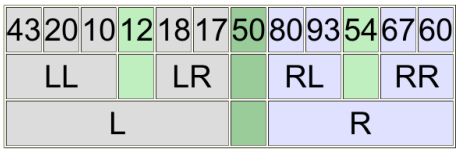
43	20	10	12	18	17	50	80	93	54	67	60

Die 60 ist zum Beispiel nach rechts geschafft worden, die 18 und die 17 nach links. Durch diese Verschiebungsaktionen ist der Median um eine Position nach rechts gewandert.

Der *sortierte* Teil des Arrays besteht jetzt aus der Zahl 50. Diese Zahl ist bereits an der *richtigen Position*. Alle Zahlen links der 50 sind kleiner, alle Zahlen rechts der 50 sind größer.

Schritt 3

Nun müssen die beiden Partitionen auf *genau die gleiche Weise* verändert werden, wie in den Schritten 1 und 2 besprochen. So entstehen aus der linken Partition zwei neue Partitionen (LL = links-links und LR = links-rechts) und aus der rechten Partition ebenfalls (RL und RR):



Das Grundprinzip des Quicksort müsste jetzt eigentlich klar sein: "Teile und herrsche". Das heißt hier: Teile den Array in zwei (im optimalen Fall gleich große) Hälften, ordne die Zahlen so an, dass alle Elemente links vom Median kleiner und alle rechts vom Median größer sind als der Median, und verfahre dann mit den beiden Hälften analog, solange, bis es nicht mehr weitergeht, weil der Array sortiert ist.

Der Teufel steckt im Detail

Ganz so einfach, wie eben beschrieben, ist der Quicksort aber nicht. Was geschieht beispielsweise, wenn in der linken Partition 3 Zahlen größer sind als der Median, in der rechten Partition aber nur 2 Zahlen kleiner? Zweimal kann getauscht werden, aber was passiert mit der dritten größeren Zahl?

Oder wie sieht es aus, wenn in einer Partion der Median so gewählt wurde, dass alle Zahlen der Partition größer sind als der Median?

Schauen wir uns dazu mal einen klassischen Quicksort-Quelltext an und analysieren wir ihn Schritt für Schritt.

13.2 Der Quicksort im Detail

In der folgenden Abbildung sehen Sie den Quicksort-Algorithmus von Robert SEDGEWICK aus dem Buch "Algorithmen in C":

```
1   quicksort(int a[], int l, int r)
2   {
3      int v, i, j , t;
4
5      if (r > l)
6      {
7         v = a[r];
8         i = l-1;
9         j = r;
10        for (;;)
11        {
12           while (a[++i] < v) ;
13           while (a[--j] > v) ;
14           if (i >= j) break;
15           t = a[i]; a[i] = a[j]; a[j] = t;
16        }
17        t = a[i]; a[i] = a[r]; a[r] = t;
18        quicksort(a, l, i-1);
19        quicksort(a, i+1, r);
20     }
21  }
```

***13-1** Der Quicksort-Algorithmus nach SEDGEWICK*

In einem unsortierten Array ist es völlig egal, welches Element als Median gewählt wird; die Wahrscheinlichkeit, dass ein Element das kleinste, das größte oder ein mittleres ist, ist bei allen Elementen eines unsortierten Arrays völlig gleich. Daher wählt der SEDGEWICK-Algorithmus einfach das rechte Element als Median (Zeile 7; `v` ist hier der Median, `a[R]` das rechte Element des übergebenen Arrays r).

Wir wollen den Algorithmus einfach mal für einen Array aus 16 Zahlen durchspielen:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	70	12	30	40	50	80	20	13	35	27	16	56	82	33	47
	L															R

Der Quicksort wird dann mit `quicksort(0,15)` aufgerufen, die Variablen `l` und `r` haben die Werte `l = 0` und `r = 15`, und es gilt `r > l`.

Nun wird die Variable `v` auf `a[R]` gesetzt, somit gilt `v = 47`. Dies ist der Median des ersten Durchgangs. Die Laufvariablen `i` und `j` werden auf `l-1 = -1` bzw. `r = 15` gesetzt. Also: `i = -1, j = 15`.

Nun geht es in Zeile 10 mit der endlosen for-Schleife los. In der ersten while-Schleife in Zeile 12

```
while (a[++i] < v)
```

werden - von links kommend - die Zahlen analysiert. Es werden diejenigen Zahlen gesucht, die nicht in die linke Partition passen, also die Zahlen, die nicht kleiner sind (sondern gleich oder größer) als der Median.

Die while-Schleife ist so angelegt, dass zunächst die Laufvariable inkrementiert wird, und erst danach wird das neue Arrayelement mit dem Median verglichen. Ist das Arrayelement kleiner als der Median, ist alles in Ordnung und die while-Schleife wird erneut durchlaufen. Das nächste Arrayelement wird mit dem Median verglichen. Sobald ein Arrayelement gefunden wird, das die Schleifenbedingung nicht erfüllt, wird die Schleife abgebrochen. Es wurde jetzt ein Element gefunden, das in die rechte Partition gehört, und `i` ist der Index dieses Arrayelements.

In unserem Beispiel würde die while-Schleife bereits bei `i = 1` stoppen, denn die 70 ist eindeutig nicht kleiner als die 47. Merken wir uns also die 70 an Position 1:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	70	12	30	40	50	80	20	13	35	27	16	56	82	33	47
	L	I														R

Die erste while-Schleife wird beendet, und weiter geht es mit der zweiten while-Schleife. Diese startet nicht links im Array, sondern rechts. In Zeile 9 hatte ja `j` den Wert des Parameters `r` angenommen.

```
while (a[--j] > v)
```

Die Laufvariable `j` wird solange dekrementiert, bis ein Array-Element gefunden wird, das *nicht größer* als der Median ist. Die ist bereits beim zweiten Element von rechts der Fall, bei der 33. Wir merken uns also die 33 an Position 14:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	70	12	30	40	50	80	20	13	35	27	16	56	82	33	47
	L	I													J	R

Beide while-Schleifen wurden abgearbeitet, jetzt kommt die if-Bedingung

```
if (i >= j) break;
```

`i` hat den Wert 1, `j` den Wert 14, also ist die Bedingung nicht erfüllt, und die for-Schleife wird nicht beendet.

Als Nächstes kommt die Tauschoperation, die Array-Elemente mit den Indices `i` und `j` werden vertauscht:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	**33**	12	30	40	50	80	20	13	35	27	16	56	82	**70**	47
	L	I													J	R

Da wir die for-Schleife nicht verlassen haben, kommt jetzt der nächste Schleifendurchgang. Als erstes wird wieder die erste while-Schleife durchlaufen:

```
while (a[++i] < v)
```

Hier wird jetzt die 12 mit der 47 verglichen, dann die 30 und die 40. `I` hat dann den Wert 4. Schließlich kommt die 50, für die die Bedingung nicht mehr gilt, denn 50 ist nicht kleiner als 47. `I` hat jetzt den Wert 5:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	**50**	80	20	13	35	27	16	56	82	**70**	47
	L					I									J	R

Nun ist wieder die zweite while-Schleife an der Reihe:

```
while (a[--j] > v)
```

Die 82 und die 56 erfüllen die Bedingung, die 16 aber nicht, sie ist kleiner als der Median 47. Also stoppt die zweite while-Schleife, und J hat den Wert 11:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	**50**	80	20	13	35	27	**16**	56	82	70	47
	L					I						J				R

Jetzt kommt wieder die if-Bedingung

```
if (i >= j) break;
```

die aber nicht erfüllt ist. Also wird abermals getauscht:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	**16**	80	20	13	35	27	**50**	56	82	70	47
	L					I						J				R

Und immer noch sind wir in der for-Schleife. Wieder wird die erste while-Schleife durchlaufen, aber bereits die nächste Zahl, die 80 an Position 6, erfüllt die Bedingung nicht. Bei der zweiten while-Schleife erfüllt die 27 an Position 10 die Bedingung ebenfalls nicht, daher werden diese beiden Zahlen vertauscht:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	16	27	20	13	35	80	50	56	82	70	47
	L						I				J					R

Nun kommen wir wieder zu den beiden while-Schleifen. In der ersten Schleife erfüllen die 20, die 13 und die 35 die Bedingung, nicht aber die 80. In der zweiten while-Schleife erfüllt die 35 nicht die Bedingung. Diese beiden Zahlen merken wir uns:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	16	27	20	13	35	80	50	56	82	70	47
	L									J	I					R

Dies ist eine interessante Situation, denn der Wert von `i` ist jetzt größer als der Wert von `j`, es gilt also die Bedingung

```
if (i >= j) break;
```

Die for-Schleife wird daher verlassen. Nun werden die Elemente mit den Indices `i` und `r` vertauscht:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	16	27	20	13	35	47	50	56	82	70	80
	L									J	I					R

Das Pivot-Element 47 befindet sich nun an seiner endgültigen und korrekten Position, denn alle Elemente links der 47 sind kleiner, und alle Elemente rechts der 47 sind größer (oder zumindest nicht kleiner). Der Array besteht jetzt also aus drei Teilen:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	16	33	12	30	40	16	27	20	13	35	47	50	56	82	70	80
	L									J	I					R

Jetzt kommt der rekursive Aufruf

```
quicksort(l, i-1)
```

`l` hat den Wert 0, `i-1` den Wert 9. Das entspricht dem linken Teil des Arrays. Diese Partition wird nun nach dem gleichen Verfahren behandelt wie eben ausführlich besprochen.

Anschließend wird aufgerufen

```
quicksort(i+1,r)
```

`i+1` hat den Wert 11, und `r` den Wert 15. Das entspricht genau dem rechten Teil des Arrays. Auch diese Partition wird nach dem beschriebenen Verfahren behandelt. Aus Platzgründen ersparen wir uns eine weitere Analyse der beiden Partitionen.

Übung 13.1 (3 Punkte)

Implementieren Sie den Quicksort in Java; orientieren Sie sich dabei an dem C-Quelltext von SEDGEWICK.

Übung 13.2 (5 Punkte)

Schreiben Sie eine Konsolen-Anwendung oder ein Applet, welches das Zeitverhalten des Quicksort mit dem des Insertionsort vergleicht. Hier gibt es drei Möglichkeiten:

1. Sie setzen sich mit einer guten Stoppuhr an den Rechner und messen, wie lange die beiden Algorithmen brauchen, um 10.000, 20.000, 40.000, 80.000 und 160.000 Zahlen zu sortieren.
2. Sie recherchieren im Internet, wie man in Java die aktuelle Zeit auslesen kann (in Millisekunden!) und lassen Ihr Java-Programm die benötigten Sortierzeiten selbst messen.
3. Sie messen überhaupt keine Zeiten, sondern lassen von Ihrem Programm die Anzahl der benötigten Rechenoperationen mitzählen und ausgeben (Zahl der Vergleiche + Zahl der Zuweisungen). Hier muss man natürlich aufpassen, denn nicht nur if-Anweisungen, sondern auch while-Schleifen und for-Schleifen führen Vergleiche durch; bei for-Schleifen kommen auch noch Zuweisungen dazu, weil ja die Laufvariable jedes Mal inkrementiert wird.

Übung 13.3 (4 Punkte)

Schreiben Sie ein Applet, das das Zeitverhalten von Bubblesort und Quicksort graphisch darstellt.

13.3 Optimierungen des Quicksort

Es sind schon viele Versuche unternommen worden, den Quicksort, der ja als schnellster Sortieralgorithmus gilt, noch weiter zu optimieren.

Ein erster und wirklich effektiver Ansatz ist es, das **Pivot-Element** etwas sorgfältiger auszuwählen. Im Idealfall hat das Pivot-Element einen mittleren Wert, so dass etwa gleich viele Elemente kleiner bzw. größer als das Pivot-Element sind. Das ergibt dann zwei ungefähr gleich große Partitionen.

Ein praktischer Ansatz ist es, einfach drei beliebige Array-Elemente zu wählen und dann das Element mit dem mittleren Wert als Pivot-Element zu nehmen (median-of-three).

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	**16**	70	12	30	40	50	80	**20**	13	35	27	16	56	82	33	**47**
	1							2								3

In dem obigen Beispiel hat man das linke, das mittlere und das rechte Element genommen. Das Element mit dem Index 7 hat den "mittleren Wert", die 16 ist nämlich kleiner und die 47 größer als dieses Element.

Etwas rechenintensiver wäre es, wenn man in einem ersten Schritt den Mittelwert des *gesamten Arrays* berechnen würde. In unserem Beispiel wäre der Mittelwert gerundet 39. Die 40 an Position 4 wäre daher ein geeignetes Pivot-Element. Testen wir doch einmal, welche Partitionen wir mit der 40 als Pivot-Element erhalten würden: 16, 12, 30, 20, 13, 35, 27, 16 und 33 wären kleiner als die 40, das sind 9 Elemente von 16. Die 70, 40, 50, 80, 56, 82 und 47 wären nicht kleiner als die 40, das sind 7 Elemente. Wir würden in der Tat zwei ungefähr gleich große Partitionen bekommen, die linke mit 9 Elementen, die rechte mit 6 Elementen. Das Pivot-Element gehört ja zu keiner Partition. Vergleicht man das mit dem Ergebnis des SEDGEWICK-Algorithmus, so ist der Unterschied aber nicht wesentlich, hier erhielten wir eine linke Partition mit 10, eine rechte mit 5 Elementen.

Übung 13.4 (2 Punkte)

Implementieren Sie eine entsprechende "Optimierung"!

Ein zweiter Ansatz, den Quicksort zu optimieren, beruht auf der Überlegung, dass der Quicksort eigentlich nur für *große* Zahlenmengen sehr schnell ist. Bei kleinen Arrays oder bei kleinen Partitionen ist der Quicksort dagegen aufwändiger als beispielsweise der Insertionsort. Daher legt man einen Schwellenwert `Min` fest, der die minimale Partitionsgröße angibt, bei der nach dem Quicksort vorgegangen werden soll. Wird im Laufe des Quicksort bei einem rekursiven Aufruf dieser Schwellenwert für eine Partition unterschritten, so wird diese Partition nach dem Insertionsort-Algorithmus sortiert. Einen Idealwert für `Min` scheint es nicht zu geben. Allerdings haben Experimente mit mehreren Millionen von Array-Elementen gezeigt, dass `Min = 25` ein recht guter Schwellenwert ist.

Übung 13.5 (3 Punkte)

Implementieren Sie auch diese Optimierung!