Team: 03, Sebastian Diedrich – Murat Korkmaz

Aufgabenaufteilung:

- Aufgaben, für die Teammitglied 1 verantwortlich ist:

    (1) Skizze

    Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:

    (1) QuickSort (alle Varianten -> method_pivot)

- Aufgaben, für die Teammitglied 2 verantwortlich ist:

    (1) Skizze

    Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:

    (1) sortNum

    (2) InsertionSort

Quellenangaben: Vorlesung am 04.11.15

Bearbeitungszeitraum: 10.11. (4h), 13.11. (7h), 15.11. (2h), 19.11 (5h), 20.11. (1h)

Aktueller Stand: Skizze *Version 3*, *Implementation Version 2* und Messung fertig

Veränderungen Implementation Version 2:

- Auswahl-Funktion MedianOf3 wurde korrigiert und getestet

Veränderungen Version 3:

- Pseudocode der Insertion-Sort Methode wurde um Start- und Endposition erweitert

- Pseudocode der Quick-Sort Methode wurde so erweitert, dass er alle 4 Auswahlmöglichkeiten des Pivotelementes beinhaltet

- Bei der textuellen Beschreibung des Quick-Sort Algorithmus, muss nicht mehr auf einen konkreten Fall (z.B. Pivotelement ganz links) eingegangen werden

Veränderungen Version 2:

- Beschreibung des Insertion-Sort Algorithmus (angelehnt an das VL-Skript)
- Beschreibung des Quick-Sort Algorithmus (angelehnt an das VL-Skript)
- Ausführlichere Beschreibung der Messungen und der Dokumentation

Skizze: (ab Seite 2)

Messungen Team03:

Aufbau: siehe Skizze

Resultate: Zahlen.dat (von der Hompage von Prof. Dr. Klauck) – 20003 Elemente

Die Laufzeit wurde in Nanosekunden (ns) gemessen und in Millisekunden (ms) umgerechnet:

Resultat 1:

- Zahlen.dat (von der Hompage von Prof. Dr. Klauck) – 20003 Elemente

Quicksort OHNE Insortionsort	**Links**	**Rechts**	**Random**	**Median**
Team 03	5352	5297	5281	5491
	4940	4864	4950	4930
	4933	4872	4947	5113
	4957	4882	4880	4945
	4967	4883	4919	4943
Durchschnitt	**5029,8**	**4959,6**	**4995,4**	**5084,4**
Quicksort mit Insortionsort (ab < 12 Elemente)	**Links**	**Rechts**	**Random**	**Median**
	6302	6209	6305	6352
	4932	4864	4892	4943
	6308	6205	6311	6318
	4971	4879	4972	4977
	6292	6221	6152	6314
Durchschnitt	**5761**	**5675,6**	**5726,4**	**5780,8**

Quicksort mit Insortionsort (ab 50 Elemente)	**Links**	**Rechts**	**Random**	**Median**
	6381	6357	6355	6355
	6314	6237	6285	6402
	6301	6240	6366	6335
	4944	4863	4955	5007
	4960	4892	5024	4998
Durchschnitt	**5780**	**5717,8**	**5797**	**5819,4**

Quicksort mit Insortionsort (ab 100 Elemente)	**Links**	**Rechts**	**Random**	**Median**
	14963	14977	14969	15044
	14999	14973	14976	15066
	15317	15279	15459	15360
	16311	16258	16315	16347
	14933	14900	14968	14994
Durchschnitt	**15304,6**	**15277,4**	**15337,4**	**15362,2**

Auswertung 1:

Man sieht bei den Messungen, dass der Quicksort Algorithmus OHNE Insertionsort bei unserer Implementierung am schnellsten ist.

Wird Insertionsort mitverwendet, steigt die Laufzeit.

Resultat 2:

- Worse-case Szenario mit 10.000 Elementen

	Links	Rechts	Random	median
Team 09	137137	136707	1122	136918
Team 03	931	536	569	865

Auswertung 2:

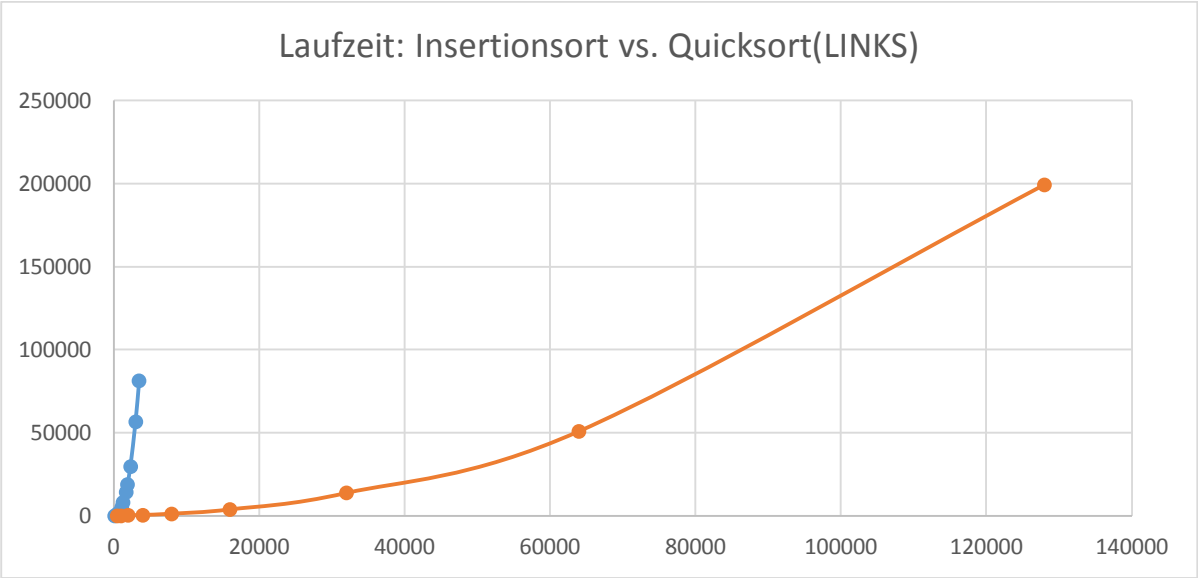
Es wurde einmal das ADTArray des Teams 09 und unser ADTArray mit unserer Implementation des Quicksort-Algorithmus verwendet.

Es zeigt sich, dass es zu einer massiven Laufzeit Erhöhung kam, als wir das ADTArray mit einem fremden ADTArray austauschten.

Resultat 3:

Quicksort	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**
Laufzeit	61	100	216	489	1230	3802	13708	50866	199286
Lesezugriffe	10080	33516	82372	188914	432059	990200	2369269	6209014	17910310
Schreibzugriffe	1679870	1685064	1696414	1720498	1770888	1873606	2084838	2506220	3354848

Insertion-Sort	**100**	**500**	**700**	**1100**	**1300**	**1700**	**1900**	**2300**	**3000**	**3500**
Laufzeit (ms)	84	954	2046	5028	8140	14213	18970	29631	56589	81443
Lesezugriffe	10098	250498	490698	1211098	1691298	2891698	3611898	5292298	9002998	12253498
Schreibzugriffe	5049	125249	245349	605549	845649	1445849	1805949	2646149	4501499	6126749



Auswertung 3:

Es zeigt sich beim Insortionsort Algorithmus, dass es zu einer exponentiellen Steigung komm, sodass nach 3500 Elementen die Messung beendet wurde.

Im Vergleich dazu steigt der Quicksort Algorithmus deutlich langsamer an.

Bei beiden Algorithmen wurde der worse-case Fall betrachtet.

Skizze Aufgabe 2:

Aufgabe: 2.1

Ziel: Zahlengenerator implementieren

Angaben zur Implementation:

- Das Trennungssymbol zwischen den einzelnen Zahlen soll ein Leerzeichen sein. Es sollen nur positive Zahlen erzeugt werden. Zahlen dürfen mehrfach vorkommen.
- Auch der beste und schlimmste Fall (Zahlen sind sortiert vs. Zahlen sind umgekehrt sortiert) soll mit einer beliebigen Anzahl von Zahlen generierbar sein. Dafür soll eine Methode mit gleichem Namen, aber mit zwei Parametern implementiert werden.

Vorgaben für die Implementation:

- Semantische Vorgabe:
    a. anzahlZahlen -> Datei
    Datei enthält die gewünschte Anzahl von zufällig erzeugten positiven Zahlen.
    b. Größe der Zahlenwerte
    Zahlenwerte sollen im Bereich von 0 bis 1.000 liegen.
- Syntaxtische Vorgabe:
    a. Name der Klasse: SortNum
    b. Name der Methode: sortNum
    c. Aufruf der Methode:
    Sortnum.sortNum(int anzahlZahlen) -> zufällige Zahlenfolge
    Sortnum.sortNum(int anzahlZahlen, boolean bool):
        - true: komplett sortiert
        - false: komplett umgekehrt sortiert
    d. Endung der Datei: .dat

Aufgabe: 2.2

Quellen:

- http://users.informatik.haw-hamburg.de/~klauck/AlguDat/TIB3-AD-skript.pdf (S. 57 ff)
- Vorlesung vom 04.11.2015 bei Prof. Dr. Klauck

Ziel: Sortieren einer Zahlenreihe basierend auf dem Sortieralgorithmus: **Insertion-Sort**

Allgemeiner Ablauf:

Pseudo-Code aus dem Vorlesungsskript von Prof. Dr. Klauck (S.58):

```
public void insertionSort(int startPos, int endPos){
        for ( int i = startPos; i <= endPos ; i++ )
        {
                int j = i;
                Datensatz<T> t = a[i];
                int k = t.key;
                while(a[j-1].key > k)
                {
                        a[j] = a[j-1];
                        j = j-1;
                }
                a[j] = t;
        }
}
```

Anpassung des Pseudo-Codes für das folgende Beispiel:

- Key-Abfrage nicht nötig, da Elemente Zahlen sind
- Erweiterung der while-Bedingung (j>1), um eine sichere Terminierung zu gewährleisten

Erläuterung an einem Beispiel:

(1) Gegeben sei eine Zahlenreihe *Z* = [5,3,6,1,2,7,4]. Begonnen wird mit dem Index *i* = 2 (in unserem Beispiel die Zahl 3).

(2) Gespeichert wird *i* in *j* und Z[i] in *k*. Dieses ist nötig, da ggf. eine höher-wertige Zahl an diese Position verschoben wird. Im weiteren Verlauf wird nun beginnend bei der Zahl mit dem *Index j-1, k* mit dieser verglichen. Wobei j nicht kleiner als 2 sein darf, da j-1 minimal den Index der ersten Zahl der Zahlenreihe darstellen kann.

(3) Sollte *j* nicht kleiner als 2 sein, gehe zu (4), sonst zu (6)

(4) Sollte Z[j-1] größer als *k* sein, gehe zu (5), sonst zu (6)

(5) Es wird die Zahl mit dem *Index j-1* an die Position *j* kopiert und *j* wird um 1 verkleinert:
Z = [5,**5**,6,1,2,7,4]

Gehe zu (3)

(6) k wird nach Z[j] gespeichert:
Z = [**3**,5,6,1,2,7,4]

(7) Der Index i wird um 1 erhöht und ab (2) wiederholt bis einschließlich i = n (n = Anzahl der Zahlen).

Hinweise zur Implementierung:

Um eine doppelte Abfrage in der while-Schleife zu verhindern, kann an der Position Z[0] ein Dummy (mit kleinstmöglichem Wert) gespeichert werden, welches als Stopper-Element dient und dazu führt, dass die while-Schleife sicher terminiert.

Test

Um die Richtigkeit und die Zuverlässigkeit des Algorithmus zu gewährleisten, sollen umfangreiche JUnit-Tests implementiert werden.

Dateiname des Jar-Files: insertionJUt.jar

Mindestanforderung:

1. Grenzfälle
    a. Zahlenreihe mit n = 2 Zahlen (sortiert, unsortiert und beide Zahlen gleich)
    b. Zahlenreihe mit n = 1.000 Zahlen (umgekehrt sortiert)
        i. Beispiel: (1.000, 999, … , 5, 4, 3, 2, 1)
    c. Zahlenreihe mit n = 1.000 Zahlen (sortiert)
        i. Beispiel: (1, 2, …, 999, 1.000)
2. Belastung
    a. Zahlenreihe mit n > 10.000
    b. Zahlenreihe mit n > 100.000
    c. Zahlenreihe mit n > 200.000

Vorgaben für die Implementation:

- Semantische Vorgabe:
    a. Insertionsort: array x startPos x endPos -> array
    startPos: dabei handelt es sich um eine positive Zahl, die nicht größer sein darf als die Endposition (endPos)
- Syntaxtische Vorgabe:
    a. Name der Klasse: Insertionsort
    b. Name der Methode: insertionsort
    c. Aufruf der Methode:
    Insertionsort.insertionsort(ADTArray array, int startPos, int endPos)

Aufgabe: 2.3

Quellen:

- http://users.informatik.haw-hamburg.de/~klauck/AlguDat/TIB3-AD-skript.pdf (S. 63 ff)
- Vorlesung vom 04.11.2015 bei Prof. Dr. Klauck

Ziel: Sortieren einer Zahlenreihe basierend auf dem Sortieralgorithmus: **Quick-Sort**

Allgemeiner Ablauf:

Pseudo-Code:

```
void quicksort(int iLinks, int iRechts, Enum pivotAuswahl)
{
	int pivot,i,j;
	if ( iRechts > iLinks )
	{
		i = iLinks;
		j = iRechts;

		int pivotIndex = method_pivot(pivotAuswahl, i , j);

		pivot = Z[pivotIndex];

		while(1)
		{
			while(Z[i] < pivot && i < iRechts)
			{
				i++;
			}
			while(Z[j] >= pivot && j > iLinks)
			{
				j--;
			}
			if ( j <= i ) break;

			//Index-Pivot merken, wenn j <= I noch nicht erfüllt
			if(indexPivot == i) indexPivot = j;

			swap(i,j);//vertauschen
		}
		swap(i,indexPivot);//Pivotelement in die Mitte tauschen
		quicksort(iLinks,i-1,pivotAuswahl);
		quicksort(i+1,iRechts,pivotAuswahl);
	}
}


int method_pivot(Enum pivotAuswahl, int iLinks, int iRechts){

	if(pivotAuswahl = links) return iLinks;

	if(pivotAuswahl = rechts) return iRechts;
```

```
    if(pivotAuswahl = random) return random(iLinks .. iRechts)

    if(pivotAuswahl = medianOf3) return medianOf3(iLinks, iRechts, (int) iRechts/2)

}
```

Definitionen:

Pivot-Element:

Das Pivot-Element dient als Referenzelement. Vor der Teilung der Zahlenreihe (Rekursiver Aufruf), befinden sich links vom Pivot-Element nur Elemente die kleiner sind und rechts davon nur Elemente die größer oder gleich sind.

Die Auswahlbestimmung, wird als Parameter beim Aufruf der Methode „quicksort“ mit übergeben. Je nach Auswahl befindet sich das Pivot-Element links, rechts, wird mittels Zufallszahl bestimmt oder wird durch die Bestimmung des Medians von drei Zahlen ermittelt.

Start und End Index:

iLinks = Index der Zahl ganz links in der (Teil)Zahlenreihe

iRechts = Index der Zahl ganz rechts in der (Teil)Zahlenreihe

Iteratoren:

i = von links nach rechts

j = von rechts nach links

Algorithmus:

Zunächst wird das Pivot-Element bestimmt. Wie dieses geschieht, ist nicht fest vorgegeben.

Der Pseudocode wurde generisch beschrieben und deckt alle 4 beschriebenen Auswahlmethoden ab.

Das aktuelle Pivot-Element wird so lange mit allen anderen Zahlen der (Teil)Zahlenreihe verglichen, bis eine Zahl gefunden wird, die kleiner ist als das Pivot-Element (beginnend bei *iRechts*) und eine Zahl, die größer oder gleich ist, als das aktuelle Pivot-Element (beginnend bei *iLinks*). Dabei bewegen sich *i* und *j* aufeinander zu. Die gefundenen Zahlen werden dann getauscht.

Dieses wird so lange fortgesetzt, bis *i* und *j* „zusammenstoßen“ (bzw: j <= i). Das aktuelle Pivot-Element wird dann mit der Zahl an der Position *i* getauscht.

Nun wird die (Teil)Zahlenreihe in zwei Teile aufgeteilt, wobei das Pivot-Element die „Grenze“ der beiden Teil-Zahlenreihen darstellt. Da das Pivot-Element sich bereits an seiner endgültigen Position befindet, wird es weder der linken noch der rechten Teil-Zahlenreihe zugeordnet.

Der Algorithmus wird nun auf jede Teil-Zahlenreihe angewandt, bis jede Teil-Zahlenreihe nur noch aus einer Zahl besteht. Nun ist die Zahlenreihe sortiert und der Algorithmus endet, da die if-Bedingung (iRechts > iLinks) nicht mehr erfüllt wird.

Hinweis zur Implementierung:

Bei der Erhöhung des Iterators *i* muss beachtet werden, dass *i* nicht größer als iRechts sein darf, da wir uns sonst außerhalb der (Teil)Zahlenreihe befinden.

Bei der Erniedriegung des Iterators j muss beachtet werden, dass j nicht kleiner als iLinks sein darf, da wir uns sonst außerhalb der (Teil)Zahlenreihe befinden.

- Semantische Vorgabe:
    a. Quicksort: array x method_pivot -> array
    method_pivot: bestimmt die Pivot-Element-Auswahl
    Folgende Auswahlmöglichkeiten sollen implementiert werden:
        i. Immer ganz links
        ii. Immer ganz rechts
        iii. Medianof3
            1. Median von folgenden Zahlen:
                a. Zahl an erster Position
                b. Zahl an letzter Position
                c. Zahl an mittlerer Position
        iv. Random
- Syntaxtische Vorgabe:
    a. Name der Klasse: Quicksort
    b. Name der Methode: quicksort
    c. Aufruf der Methode: Quicksort.quicksort(ADTArray array, Enum pivotAuswahl)
- Sollte die (Teil)Zahlenreihe weniger als 12 Elemente beinhalten, werden die (Teil)Zahlenreihen mittels Insertionsort sortiert. Deshalb wurde diese Methode um die Parameter, startPos und endPos erweitert.

Test

Um die Richtigkeit und die Zuverlässigkeit des Algorithmus zu gewährleisten, sollen umfangreiche JUnit-Tests implementiert werden.

Dateiname des Jar-Files:

- quickJUt.jar

Mindestanforderung:

3. Grenzfälle
    a. Zahlenreihe mit n = 2 Zahlen (sortiert, unsortiert und beide Zahlen gleich)
    b. Zahlenreihe mit n = 1.000 Zahlen (umgekehrt sortiert)
        i. Beispiel: (1.000, 999, … , 5, 4, 3, 2, 1)
    c. Zahlenreihe mit n = 1.000 Zahlen (sortiert)
        i. Beispiel: (1, 2, …, 999, 1.000)
4. Belastung
    a. Zahlenreihe mit n > 10.000
    b. Zahlenreihe mit n > 100.000
    c. Zahlenreihe mit n > 200.000

Aufgabe 2.4

Versuchsaufbau:

Es wird eine Kopie der Algorithmus-Implementation erstellt und um folgende Komponenten erweitert:

- Laufzeit:
    a. Zu Beginn des Algorithmus wird die aktuelle Zeit festgehalten. Am Ende des Algorithmus wird die aktuelle Zeit festgehalten und der Betrag der Differenz der beiden ausgegeben. Dieser Betrag entspricht der Laufzeit. (zu beachten ist dabei, dass diese Laufzeit NUR mit Laufzeiten verglichen werden darf, die ebenfalls auf dem gleichen Ausführungssystem gemessen worden sind)
    b. Bei der Laufzeitmessung von Quicksort muss beachtet werden, dass die Laufzeit von Insertionsort abgezogen wird. (Sortierung von Zahlenreihen mit weniger als 12 Zahlen)
- Zugriffe (Lesen und Schreiben)
    a. Lesen

        Ein Lesezugriff ist definiert, durch das Auslesen eines Wertes aus der Zahlenreihe.
    b. Schreiben

        Ein Schreibzugriff ist definiert, durch das mutieren der Zahlenreihe oder durch die Belegung einer Variablen mit einem Wert.

Messungen:

Vorgaben für die Messungen: Anzahl der Zahlen der Zahlenfolgen:

1) 500
2) 1.000
3) 2.000
4) 4.000
5) 8.000
6) 16.000
7) 32.000
8) 64.000
9) 128.000

1. Mittels sortNum wird eine zufällig generierte Zahlenfolge nach obigen Vorgaben erstellt.
    (1) Messung der Laufzeit mittels Insertionsort
    (2) Messung der Laufzeit mittels Quicksort (links)
    (3) Messung der Laufzeit mittels Quicksort (rechts)
    (4) Messung der Laufzeit mittels Quicksort (MedianOf3)
    (5) Messung der Laufzeit mittels Quicksort (random)
2. Mittels sortNum wird eine zufällig generierte Zahlenfolge nach obigen Vorgaben erstellt.
    (1) Messung der Schreib- und Lesezugriffe mittels Insertionsort
    (2) Messung der Schreib- und Lesezugriffe mittels Quicksort (links)
    (3) Messung der Schreib- und Lesezugriffe mittels Quicksort (rechts)
    (4) Messung der Schreib- und Lesezugriffe mittels Quicksort (MedianOf3)
    (5) Messung der Schreib- und Lesezugriffe mittels Quicksort (random)

Damit ergeben sich 2x5x9 Messungen.

Resultate:

Die Ergebnisse der obigen Messungen werden in eine Excel-Tabelle eingetragen:

Algorithmus X									
	500	1000	2000	4000	8000	16000	32000	64000	128000
Laufzeit (ms)									
Lesezugriffe									
Schreibzugriffe									

Aus den Ergebnissen werden Excel-Graphiken erzeugt, um die Steigung der einzelnen interpolierten Kurven vergleichen zu können.

Die daraus resultierenden Schlussfolgerungen werden zusammen mit den Graphiken in einem PDF dokumentiert.

Mögliche Fragen wären: Welcher Algorithmus ist am schnellsten (gemessen an der Laufzeit). Welcher Algorithmus benötigt am wenigsten Schreib- bzw. Lesezugriffe!