Sortieren mit zeitlicher Begrenzung

von Sven Marquardt 240747

23.01.2017

1) Aufgabenstellung

Es soll ein Sortieralgorithmus entwickelt werden, der auch dann noch brauchbare Ergebnisse liefert, wenn die Zeit zum vollständigen Sortieren nicht ausreicht. Die Ausführungszeit wird dabei dadurch gemessen, in dem die Anzahl der Vergleichsoperationen begrenzt wird. Bei jedem Durchlauf dürfen also die Werte des übergebenen Arrays nur 300 mal verglichen werden. Die Werte zum testen müssen über die mitgelieferte Klasse RandomArrayGenerator generiert werden, um so einen gleichen Ausgangspunkt für alle zu garantieren. Es sollen Arrays der folgenden Größen erstellt und getestet werden: 100,120,140,160. Es muss garantiert werden, dass zu keinem Zeitpunkt mehr als die vorgegebenen Vergleichsoperationen durchgeführt werden können. Der Grad der erzielten Ordnung wird durch den Kendall'schen Rangkorrelationskoeffizienten τ gemessen. Hierbei muss in 90% der Fälle mindestens ein Grad der Ordnung 0.4 erreicht werden. Dies gilt für jede Größe des Arrays wie oben beschrieben.

2) Erklärung des Programms

2.1 Module

In dem Mavenpackage Randomarraygenerator befindet sich nur, die vorgefertigte Klasse zum erstellen eines Testarrays. Im Modul sort-algorithm-implementations befinden sich alle genutzten Sortieralgorithmen. Im Modul sort-algorithm-tautest befindet sich das Programm um die Aufgabe zu erfüllen. Es Akkreditiert die nötigen Daten und wertet diese aus.

2.2 Verhindern der Überschreitung der Vergleichsoperatoren

Um zu verhindern, dass die Anzahl der Vergleiche die grenze von 300 überschreitet, wurden eigenen Methoden geschrieben zum vergleichen der Werte. Jedes mal wenn ein Vergleich stattfindet wird in diesen ein Zähler inkrementiert. Wenn der Zähler kleiner als die grenze der zulässigen Vergleichsoperatoren ist, wird der momentane Stand der Sortierung in einem separatem Array gespeichert. Ist der Zähler über der grenze, wird der momentane Stand nicht mehr gespeichert. Dies

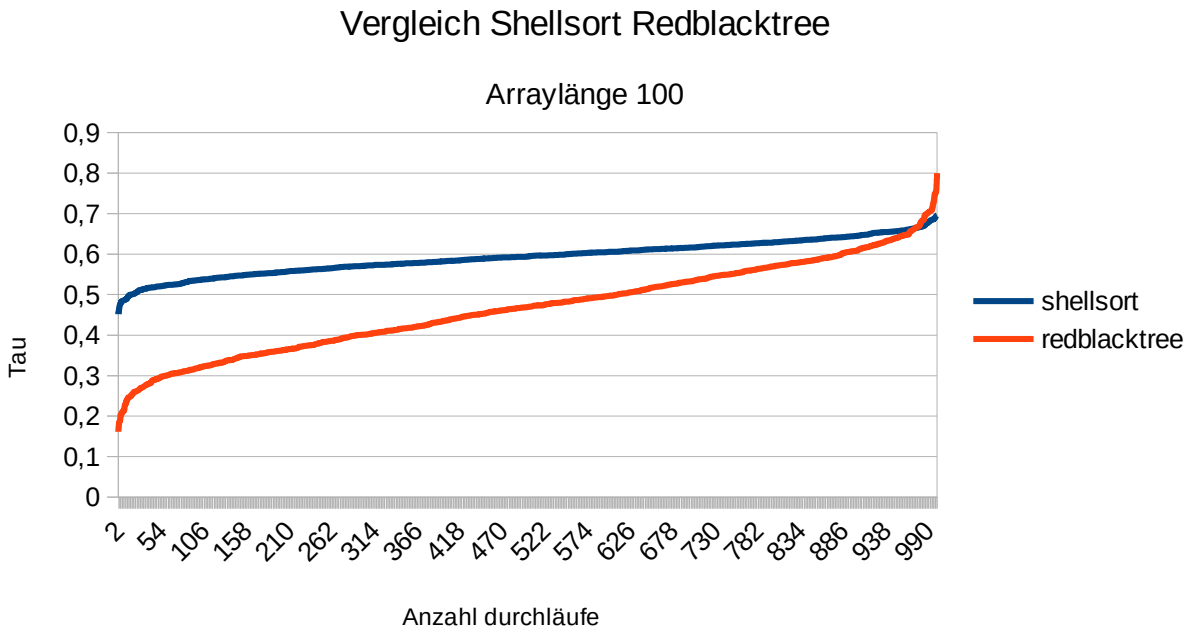
ist zu finden in der Klasse AbstractSort in der Methode inCounter. Der Zähler wurde als AtomicInteger implementiert um parallele Ausführung zu zulassen.

2.3 Struktur des Programms

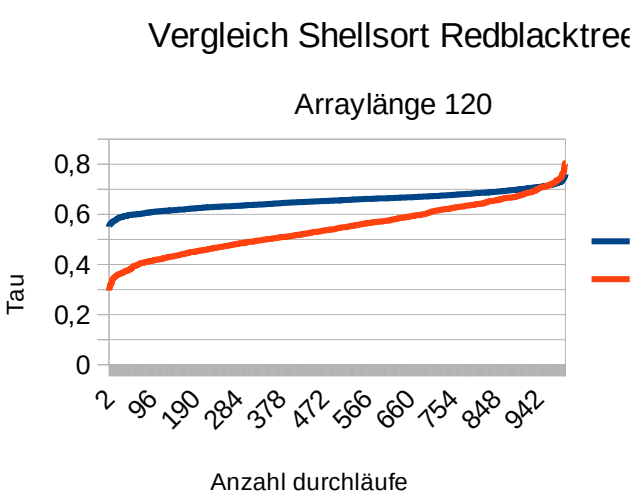
Alle Sortieralgorithmen erben von der Klasse AbstractSort, die dafür sorgt das die Anzahl der Vergleichsoperatoren eingehalten werden. Die AbstractSort implementiert zwei Interfaces. Einmal SimpleSort, welche Methoden vorschreibt die eine normale Sortierklasse haben sollte um zu sortieren und das sortierte Array lesen zu können, und RangedSort, die das Array liefert welches die Werte beinhaltet bevor die Anzahl der Vergleichsoperatoren überschritten wird. Die Methoden in der SimpleSort, müssen von der jeweiligen Kind klasse von AbstractSort selbst implementiert werden.

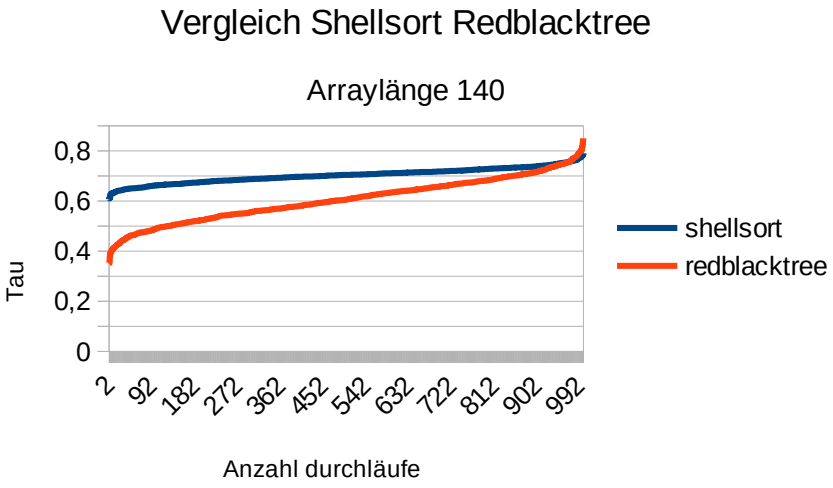
2.4 OwnSort Erklärung

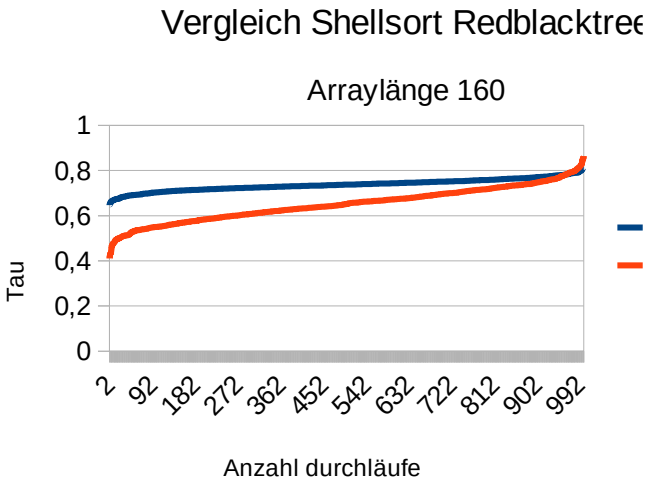
OwnSort ist das selbst entwickelte Sortierverfahren. Es beinhaltet einen Redblacktree[i] und einen Shellsort[ii] Sortieralgorithmus. Der Redblacktree wurde leicht abgewandelt und mit Rotation[iii] versehen sowie mit einer inorder Traversierung[iv] für die sortierte Ausgabe. Für Arrays >= 160 wird der Redblacktree verwendet. Für Arrays, die kleiner sind der Shellsortalgorithmus. Shellsort wird für kleine Arrays verwendet, weil der Redblacktree bei diesen keine guten Werte liefert. Der Redblacktree wird effizienter mit der Größe des Arrays. Zur Veranschaulichung einmal das Ergebnis beim sortieren eines Arrays mit der Länge 100.

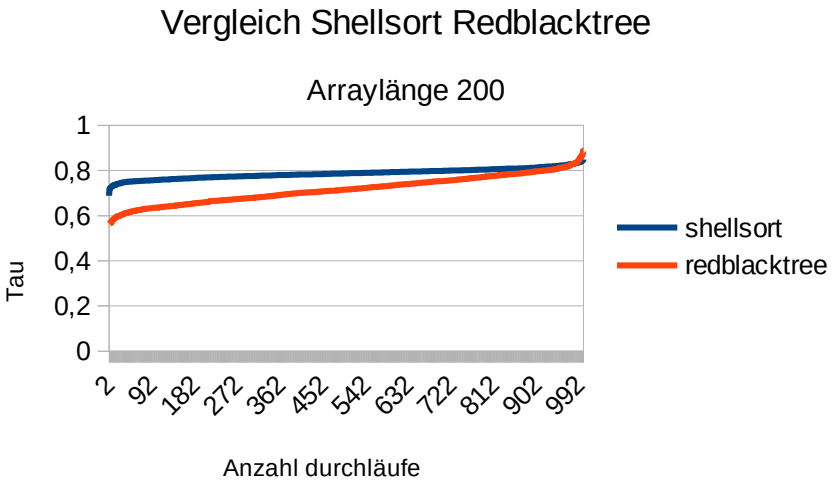


Hier kann man sehen, das Shellsort bei 100% der Durchläufe einen Tauwert von mindestens 0,4 erreicht. Der Redblacktree hingegen nur bei 70,8%. In den darauffolgenden Arraygrößen steigt aber der Grad der Ordnung beim Redblacktree.









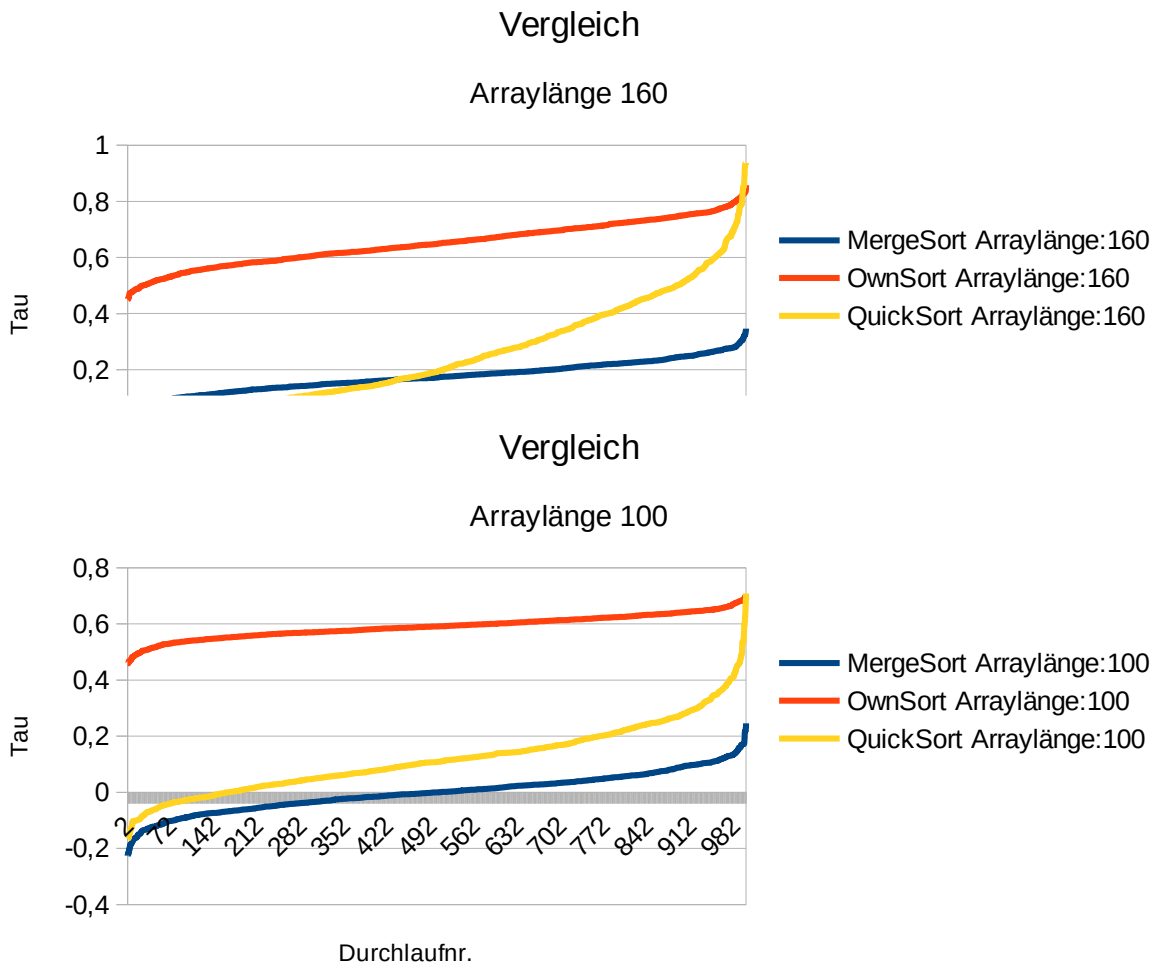
Deswegen ist dies ein kombinierter Sortieralgorithmus aus beiden Algorithmen, wobei der richtige vom Programm gewählt wird.

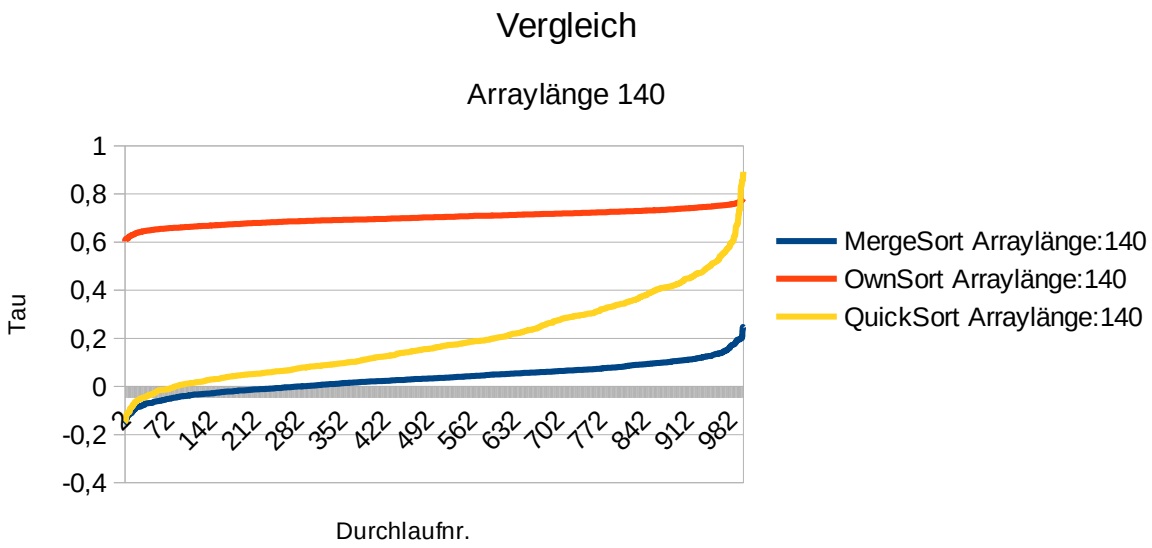
3) Ergebnis

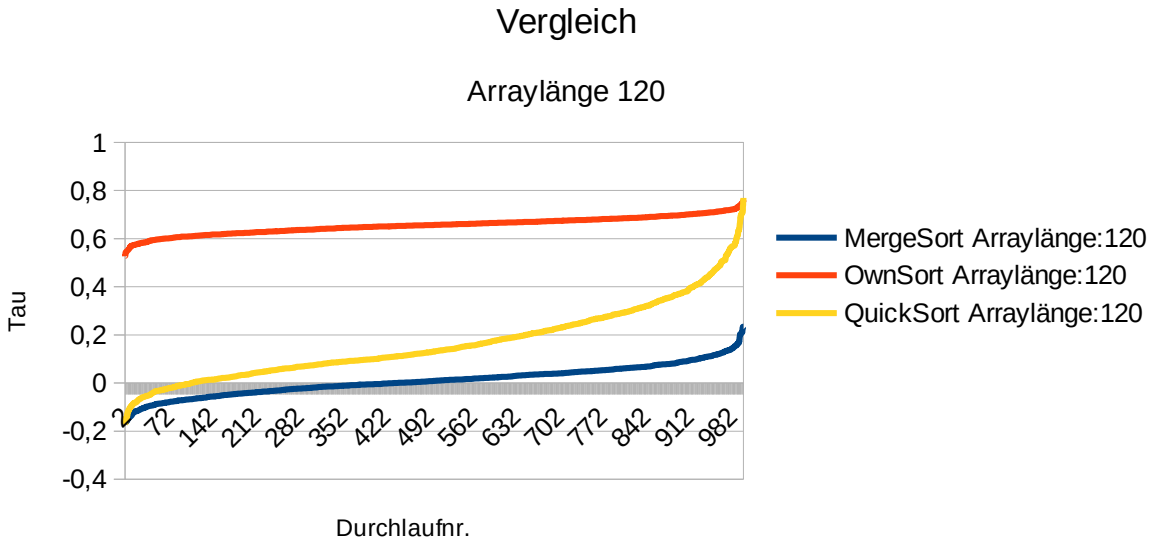
Das Programm. um die Ergebnisse für die Aufgabe anzuzeigen, auszuführen lebt in der TestSortAlgklasse. Zuerst wird der Durchschnittswert pro Sortieralgorithmus ausgewertet. Diese betragen

```
Ergebnis QuickSort:-0.0313131313131313
Ergebnis MergeSort:-0.08944427685123785
Ergebnis OwnSort:0.5392463912756256
Ergebnis QuickSort:0.0012044817927170939
Ergebnis MergeSort:-0.07077051342595457
Ergebnis OwnSort:0.6088661685916616
Ergebnis QuickSort:0.016053442959917782
Ergebnis MergeSort:-0.042211614561900895
Ergebnis OwnSort:0.6624595311692241
Ergebnis QuickSort:0.02273584905660378
Ergebnis MergeSort:0.1090912030413676
Ergebnis OwnSort:0.5489150943396227
Ergebnis QuickSort:0.0627537688442211
Ergebnis MergeSort:0.288212547277262
Ergebnis OwnSort:0.6362914572864322
```

bis zur Percentile 10,0 bei 1000 Durchläufen. Somit erfüllt der Algorithmus die Eigenschaft mindestens 90% der zeit einen Ordnungsgrad von 0.4 oder größer zu haben. Zum besseren Vergleich noch mal die Ergebnisse der einzelnen Algorithmen als Graphen gegenüber gestellt.







Zu sehen ist jeweils der Algorithmus farblich gekennzeichnet und dessen Ergebnis bei 1000 Durchläufen aufsteigend sortiert. Klar zu erkennen ist, das Merge und Quicksort nicht 90% der Durchläufe über einen Tauwert von 0.4 erreichen. Zum überprüfen der Werte, werden diese nach dem Ausführen des Programms im Verzeichnis des Ausfuhrorts jeweils eine csv Datei erstellt pro Algorithmus.

i https://web.archive.org/web/20171225061735/https://de.wikipedia.org/wiki/Rot-Schwarz-Baum
ii https://web.archive.org/web/20180106172716/https://en.wikipedia.org/wiki/Shellsort
iii https://web.archive.org/web/20171224123545/https://de.wikipedia.org/wiki/Bin%C3%A4rbaum#Rotation
iv https://web.archive.org/web/20171217111735/https://en.wikipedia.org/wiki/Tree_traversal#In-order_2