

## Avalanchesort besser als Quicksort?

Wer effizient sortieren will, der findet in Lehrbüchern meist zwei Algorithmen beschrieben: Quicksort und Mergesort. Bessere Lehrbücher erwähnen auch Naturell Mergesort. Quicksort braucht zum Sortieren im besten Fall so viele Vergleiche wie Mergesort in der Regel. Naturell Mergesort ist beim Vergleichen oft rund 20% besser als Quicksort im besten Fall. Die Zahl der Vergleiche bei Mergesort ist dabei eine Grenze, mit welcher beim Naturell Mergesort im schlimmsten Fall rechnen muss, weil der Naturell-Mergesort im Gegensatz zu Mergesort oder Quicksort die Vorsortierung ausnutzt. Der Naturell Mergesort bestimmt im ersten Schritt sortierte Teilfolgen (= Runs) und mergt diese. Im Internet habe ich für den Naturell Merge nur iterative Varianten gefunden. Zur Abgrenzung dagegen habe ich meine aufsteigend rekursive Variante Avalanchesort genannt. Die Lawine (=engl.: avalanche) beschreibt schön, wie die sortierte Menge an Daten wie bei einer Lawine klein startet und explosiv anwächst.

So wie eine Lawine nicht weiß, wie viel Schnee hangabwärts noch zu finden ist, so muss der Avalanchesort im Gegensatz zu Quicksort oder Mergesort nicht ‚wissen‘, wie viele Daten er zu sortieren soll. Dies liegt an der aufsteigend rekursiven Steuerung des Saccharosetransporters. Ein unsortierte Datenmenge wird nach folgendem Ablauf sortiert: Der Avalanchesort-Starter beginnt mit dem Rekursionsindex 1 und ruft einen Avalanchesort mit der Ordnung 1 auf. Dieser mergt er zwei Runs (= sortierte Teillisten) der 0. Ordnung zu einem Run 1. Ordnung. Wenn noch Unsortiertes übrig ist, ruft der Starter einen zweiten Avalanchesort mit Ordnung 1 auf. Der Avalanchesort liefert ein zweiten Run 1. Ordnung zurück. Der Starter mergt die beiden Runs 1. Ordnung zu einem Run 2. Ordnung und erhöht gleichzeitig seinen internen Rekursionsindex auf 2. Wenn immer noch Unsortiertes übrig ist, startet er einen Avalanchesort mit der Ordnung 2. Dieser ruft zwei Avalanchesort der Ordnung 1 auf und mergt deren Runs der Ordnung 1 zu einem Run der Ordnung 2. Der Starter mergt das Ergebnis zusammen mit den zuvor schon generierten Run der Ordnung 2 zu einem Run der Ordnung 3. Der Starter erhöht seinen Rekursionsindex solange, bis alles sortiert ist. Die Abbildung zeigt an, wie lawinenartig mit Anstieg des Rekursionsindex die Gier nach Sortierdaten wächst. Zusammenfassen kann man sagen:

**Der Starter zählt den Klassenordnung solange hoch, bis das rekursive Avalanchesort das letzte Datum durch Mergen von zwei Run gleicher Klasse sortieren konnte.**

Auf Github habe ich den Sortieralgorithmus unter "porthd/avalanchesort" als PHP-Code veröffentlicht. Wie beim PHP-Befehl „*usort*“ erlaubt Avalanchesort die Nutzung eigener Vergleichsfunktionen. Im Gegensatz zu „*usort*“ unterstützt Avalanchesort auch mehrfaches Sortieren. Eine stabile Sortierung liegt vor, wenn eine Liste nach zwei Sortierungen bzgl. Vorname bzw. Nachname so aufgebaut ist, dass wie im Adressbuch bei gleichen Nachnamen die Vornamen sortiert vorliegen. Der Test „*testAvalanchesortStable*“ zeigt, dass Avalanchesort stabil sortiert, während „*testUsortUnstable*“ zeigt, dass die PHP-Funktion „*usort*“ beim zweiten Sortieren die erste Sortierung zerschießen kann. Mit diesem Artikel verbinde ich die Hoffnung, dass PHP vielleicht doch noch irgendwann eine stabile Alternative zum instabilen Quicksort in seinen Sortierbefehlen bekommt. Zur Titelfrage: Ich denke, Avalanchesort ist besser als Quicksort, weil es stabil sortiert und vergleichbar effizient bei Vergleichen und Datentransfers ist.

Bei der Programmierung habe ich auch die Datenhaltung vom Algorithmus getrennt, damit man leicht per Interface Injection dem Sortiervorgang verschiedene Datenhaltungsklassen zuweisen kann. Die Testergebnisse zeigen, dass Avalanchesort mit weniger Vergleichen auskommt als

Quicksort. Im Gegensatz dazu braucht Quicksort im unsortierten Fall weniger Datenvertauschungen als Avalanchesort. Diese Effizienz erkaufte es vermutlich mit dem Verlust an Sortierstabilität.

In Bezug auf die Datenhaltung enthält meine Programmierung zwei Varianten: eine für Arrays und eine für Listenarray. Der Listenarray kombiniert die Idee der Array-Indexierung und die Idee der Nachbarschaftsbeziehung von Listen. Wie beim Array kann über einen Index direkt auf das Feld mit seinen Elementdaten zugegriffen werden. Wie bei einer Liste kann aber auch über Prev- und Next-Verweise direkt zum Nachbarn gewechselt werden. Die zweite Variante habe ich programmiert, weil sich in Listen der Aufwand für Speicheränderungen reduziert. Im Quicksort für Arrays muss man die Array-Elemente vertauschen. Im Avalanchesort für Listen braucht man nur die Zeiger auf Elementen vertauschen. Es gibt sogar eine Variante, mit welcher man die sortierte Liste in einen natürlichen Array umordnen kann.

Die Trennung zwischen Datenhaltung und Algorithmus lässt noch einen Unterschied zwischen Avalanchesort und Quicksort offenbar werden. Quicksort braucht „Array“-artige Datenstrukturen, um in der zu sortierenden Liste leicht vor und zurückgehen zu können. Avalanchesort kommt dagegen mit einfachen Listen aus. Avalanchesort wäre vermutlich deshalb auch in der Lage, über mehreren Rechner verteilte Datenlisten effizient zu sortieren, wenn man denn so etwas braucht.

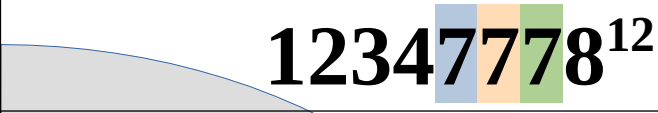
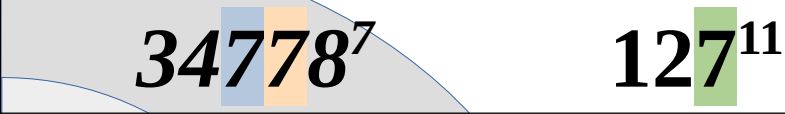
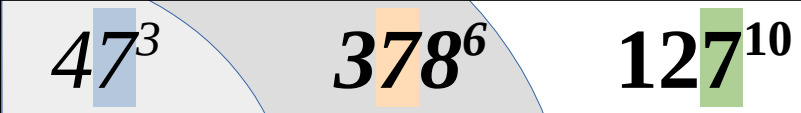

## **Link**

<https://bugs.php.net/bug.php?id=53341&edit=1> (keine stabile Sortierung für PHP)

<https://github.com/porthd/avalanchesort> (PHP-Code für Avalanchesort)

Abbildung:

8 Einträge in 6 Runs (=sortierte Teillisten) mit Indexnummer bei Sortierzwischenständen und mit Kennzeichnung, wie der Sortiererfolg mit Zunahme der Ordnung der Runs von links nach rechts lawinenartig anwächst.

Runordnung - Anzahl Runs	Anzahl Runs 0. Ordnung (max.)	Schnapsschuss der Teillisten in der Datenliste
3 – 1	$2^3=8$	
2 – 2	$2^2=4$	
1 – 3	$2^1=2$	
0 – 6	$2^0=1$	

Zahl der Datenzugriffe und der Vergleiche für 200 Datensätze bei Quicksort, Bubblesort und Avalanchesort mit „testStartSortMethodsGivenRandomFilledArrayThenSortIt“.

Type	Vergleiche			Datentransfers		
	zufällig**	anti-sortiert	sortiert	zufällig**	anti-sortiert	sortiert
Bubblesort	19613	19900	199	22374 (7458)	59454	0 (0)
Quicksort	2307	10713	20098	912 (304)	444	0 (0)
Avalanchesort	1400	962	199	2136 (96)*	2406	0 (0)

\* Die Aufrufe von „cascadeDataListChange“ bei Avalanchesort und „swap“ bei Quicksort bzw. Bubblesort sind in Klammern angegeben. Die Zugriffe bei „cascadeDataListChange“ wurden aus der Run-Länge geschätzt. Jeder „swap“-Aufruf bewirkt drei Datentransfers.

\*\* Werte variieren je nach zufälliger Sortierung.