ADP Aufgabe 2: Naives und Komplexes Sortieren

Team 1
Hugo Protsch, Justin Hoffmann

9. Dezember 2020

Inhaltsverzeichnis

1 Formales

Aufgabenaufteilung
Der Entwurf, der Code und die Analyse für Insertion Sort wurde zusammen entwickelt.
Der Entwurf, der Code und die Analyse für Quick Sort wurde von Hugo Protsch entwickelt.
Der Entwurf, der Code und die Analyse für Heap Sort wurde von Justin Hoffmann entwickelt.

Quellenangaben
`https://en.wikipedia.org/wiki/Log%E2%80%93log_plot`

Bearbeitungszeitraum
Der gesamte Arbeitsaufwand für den Entwurf belief sich auf 59 Stunden und teilt sich wie folgt auf:

- Entwurf: ca. 18 Stunden
- Implementation: ca. 16 Stunden
- Laufzeitmessung und Analyse: ca. 25 Stunden

Aktueller Stand
Der Entwurf wurde überarbeitet und in Erlang implementiert. Die Laufzeitmessung und Analyse wurde durchgeführt und dokumentiert.

Änderungen des Entwurfes

- Insertion Sort: Formulierung im Graph geändert
- Quicksort: Pivot-Element Wahl hinzugefügt
- Quicksort: Verzweigung zu Insertion Sort hinzugefügt
- Das Dokument wurde leicht umstrukturiert (Sektion-Hierarchy)
- Heapsort: Der Max-Heap wird nun Top-Down konstruiert
- Heapsort: Bei heapify Vergleiche zwischen den Kindern hinzugefügt

2 Entwurf

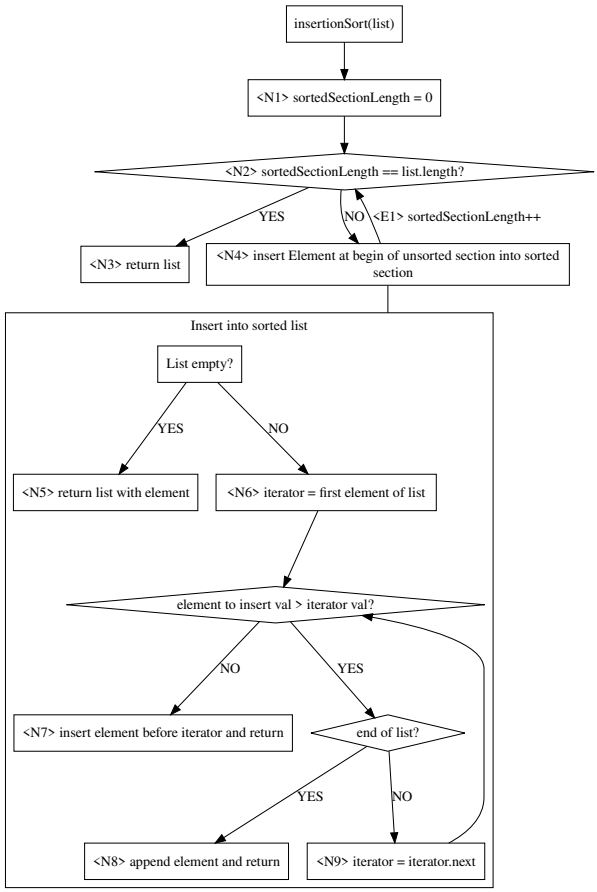
2.1 Insertion Sort

Algorithmus Siehe Abbildung 1. Bei Insertion Sort wird eine Liste durch das Einfügen von Elementen aus einem unsortierten Bereich (zunächst die komplette Liste) in einen sortieren Bereich (zunächst leer, <N1>) sortiert.

Bei dem Einfügen eines Elements E in den sortierten Bereich muss dabei jeweils der Bereich bis zu dem Element durchlaufen werden, hinter das das Element E eingefügt werden muss (Siehe »Insert into sorted list«Subgraph).

Somit wird der sortierte Bereich mit jeder Iteration um 1 erhöht <E1>. Sobald der sortierte Bereich alle Elemente enthält, wird die Liste zurückgegeben. Bei der Implementation des Algorithmus auf einfach verkettete Listen kann der sortierte und unsortierte Bereich getrennt werden.

Abbildung 1: Insertion Sort



Laufzeit Die erwartete Laufzeit beträgt $O(n^2)$, da bei jeder Iteration das einzufügende Element mit allen noch nicht sortierten Elementen verglichen wird. Im Fall einer bereits sortierten Liste tritt der Best-Case ein. Hierbei müssen nur n Vergleiche durchgeführt werden. Somit beträgt die Laufzeit $\Theta(n)$.

Bei der Laufzeitmessung überprüfen wir, ob der gemessene Zusammenhang mit dem erwarteten übereinstimmt. Dafür verwenden wir für den Best-Case zufällige Listen, im Worst-Case bereits sortierte Listen.

2.2 Quick Sort

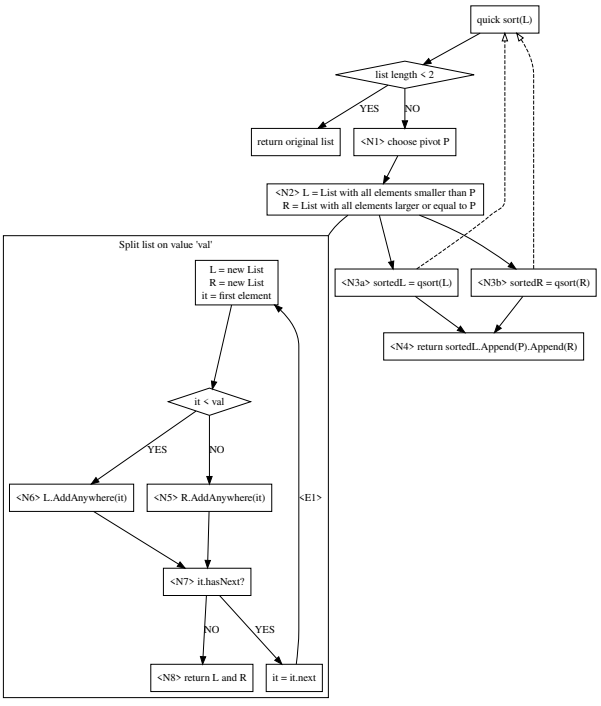
Algorithmus Siehe Abbildung 2. Bei Quick Sort wird zunächst willkürlich ein Pivot-Element aus der Liste ausgewählt.

Anschließend wird die Liste in zwei Teile aufgeteilt: Die Liste L erhält alle Elemente, die kleiner als das Pivot-Element sind, die Liste R alle Elemente, die größer als das Pivot-Element sind. Dafür wird die Liste elementweise durchlaufen und jedes Element mit dem Pivot-Element verglichen. Die Reihenfolge der Elemente in den Listen spielt keine Rolle. Das Pivot-Element selber kommt nicht in den Listen vor.

Da alle Elemente, die kleiner als das Pivot-Element sind und alle Elemente, die größer als das Pivot-Element sind nun getrennt vorliegen, ist die Position des Pivots eindeutig als 'zwischen den Listen L und R' bestimmt. Somit kann der Algorithmus rekursiv auf die jeweiligen Listen erneut angewandt werden.

Das Pivot-Element wird an die Liste R vorangestellt. Die Liste L wird der Liste R, inklusive Pivot, vorangestellt. Die Liste ist nun sortiert.

Abbildung 2: Quicksort



Für die Wahl des Pivot-Elementes werden insgesamt 5 Methoden implementiert:

- left – wählt das erste Elemente der Liste
- middle – wählt das mittlere Element der Liste
- right – wählt das letzte Element der Liste
- median – wählt den Median der drei obigen Werte
- random – wählt ein zufällig ausgewähltes Element der Liste

Des Weiteren soll bei der Implementation des Algorithmus ab einem Schwellenwert auf Insertion Sort umgeschaltet werden. Dafür wird zu Beginn die Länge der Liste abgefragt, falls diese unter dem übergebenem Schwellenwert liegt, wird restliche Liste stattdessen mithilfe von Insertion Sort sortiert.

Laufzeit Die Laufzeit beträgt im Best-Case $\Omega(n \cdot \log(n))$, da bei gut gewähltem Pivot-Elementen die Anzahl an Vergleichen mit jedem Rekursionsschritt halbiert wird, wobei die Teillisten trotzdem durchlaufen werden müssen. Wenn das Pivot-Element bei jeder Iteration schlecht gewählt ist (kleinstes oder größtes Element in der List), beträgt die Laufzeit $O(n^2)$, da jeweils eine Partition die Größe 1 hat und die Anzahl an Vergleichen somit nicht reduziert wird.

Bei der Laufzeitmessung testen wir, ob die Komplexität des implementierten Algorithmus mit der erwarteten übereinstimmt. Um den Worst-Case zu überprüfen, nutzen wir bereits sortierte Listen und wählen das letzte bzw. erste Element als Pivot-Element. Für den Best-Case nutzten wir zufällig generierte Listen.

Wir ermitteln des Weiteren die Elementanzahl, ab der Quicksort schneller als Insertion Sort ist und wie sich die unterschiedlich Methoden, das Pivot-Element auszuwählen, auf die Laufzeit auswirken.

Außerdem wird die Implementation mit Erlang List Comprehension der Eigenen gegenübergestellt. Wir erwarten bei der List Comprehension eine leicht langsamere Laufzeit als bei der eigenen Implementierung, da bei der List Comprehension die Liste zwei Mal durchlaufen wird: Das erste Mal um alle Elemente, die größer – das zweite Mal um alle Elemente, die kleiner als das Pivot-Element sind, zu filtern. Bei der eigenen Implementation wird die Liste lediglich einmal durchlaufen. Hierbei ist es interessant zu sehen,

inwiefern sich die Anzahl an Durchläufen durch die Liste in der Laufzeit wieder spiegelt.

2.3 Heap Sort

Algorithmus

Allgemeiner Aufbau Betrachte Abbildung 3. Im Allgemeinen besteht Heap Sort im Wesentlichen aus zwei Teilen: Dem Bauen eines Max-Heaps aus der zu sortierenden Liste und dem eigentlichen Sortieren mithilfe dieses Heaps. Der sogenannte Max-Heap ist eine binäre Baum-Datenstruktur, in der jeder Kindwert kleiner dem Elternwert ist, die entscheidende Eigenschaft für den Heap-Sort.

Strukturell haben wir einen "Divide and ConquerAnsatz gewählt und den Algorithmus in mehrere Subroutinen unterteilt.

Bauen des Max-Heaps Der Bau-Algorithmus des Max-Heaps beginnt mit dem Inkrementieren der Heap-Size, da ein neues Element eingesetzt wird. Jedes nachfolgende Element wird unten eingefügt. Der Baum wird von links nach rechts und top-down aufgebaut. Jeder Funktionsdurchlauf berechnet den Einsetz-Pfad im Heap. Wir vergleichen den derzeitigen Knoten mit unserem temporären Laufwert. Ist dieser Laufwert größer als der derzeitige Knoten, werden beide Werte vertauscht, sprich der Laufwert wird in den derzeitigen Knoten eingesetzt und der (nun ehemalige) derzeitige Knoten wird zum temporären Laufwert. So ßickert"jeder Wert aus der Liste and die richtige Position im Max-Heap, bis der korrekte Pfad abgelaufen worden ist.

In Abbildung 5 spiegelt sich dieser Absteig-Vorgang in der unteren Schleife wieder.

Heapify In Heapify (siehe Abbildung 4) machen wir uns die Tatsache zunutze, dass eigentlich ein Max-Heap vorliegt und sich nur ein einziges Element an der falschen Stelle befindet. Dieses Element wird, falls nötig, zu einer passenden Position nach unten "durchsickern".

Zunächst wird überprüft, ob das beide Kinder der derzeitigen Postion im Baum (beim ersten Durchlauf root) leer sind, wir also einen Knoten ohne Kinder vorliegen haben. Ist dies der Fall sind wir am Ende des Baumes und geben den Heap zurueck. Ist dies nicht der Fall, kann es sein, dass nur das rechte Kind leer ist, was ebenfalls ueberprueft wird. Dies bedeutet, dass das linke Kind das letzte Element im Heap ist. Ein letzter Vergleich mit

dem linken Kind (und ein letzter Tausche falls noetig) und der Baum wird zurueckgegeben. Das eigentliche "Durchsickern"des Lauf-Elements passiert nach diesen beiden Abfragen. Bei jeder Position wird zunaechst ueberprueft, ob das Lauf-Element groesser als beide Kinder ist. Ist dies der Fall ist das Element an die richtige Stelle durchgesickert. Ist dies jedoch nicht der Fall, tauscht das Element mit dem jeweils groesseren Kind die Position und wandert so nach unten.

So wird der Heap für die weitere Sortierung "heapified".

Sortierung Auf das Umwandeln der Liste in einen Max-Heap wurde bereits eingegangen. Der erstellte Max-Heap wird dann zum Sortieren benutzt. Wie oben bereits erwähnt ist die bestimmende Eigenschaft eines Max-Heaps der ausschlaggebende Aspekt dieses Sortier-Algorithmus. Wir fügen also das Wurzelelement des Heaps in die Output-Liste ein. Dieses Element gilt nun als sortiert (alle Elemente müssen vorne angefügt werden). Nun wird das letzte Element des Heaps mit der Wurzel getauscht und die ehemalige Wurzel entfernt. Die Struktur des Max-Heaps wird, wie oben beschrieben, durch Heapify wieder hergestellt. Dieser Vorgang wiederholt sich nachfolgend mit dem jeweils reduzierten Max-Heap, bis keine Elemente mehr im Heap vorhanden sind. Abschließend kann eine vollständige sortierte Liste zurückgegeben werden.

Laufzeit Die Laufzeit beträgt im Worst-Case $O(n \cdot log\ n)$. Dieser tritt ein, wenn eine bereits weitgehend vorsortierte Liste vorliegt, da die Liste beim Heap-Aufbau gewissermaßen invertiert wird und somit viele Vergleiche pro Element erforderlich sind. Wünschenswert wäre demnach eine invertiert-sortierte Liste, wodurch die Anzahl der Vergleiche auf 1 pro Element minimiert ist. Es gilt mit den Laufzeittests herauszufinden, ob unsere Komplexitäts–Annahmen realistisch sind. Um Heap-Sort hinreichend zu testen, binden wir den Algorithmus erweiternd in den Quick-Sort-Test mit ein und stellen beide gegenüber. Diese drei Fälle werden untersucht und verglichen:

- Vorsortierte Liste
- Invertiert sortierte Liste
- Zufällig generierte Liste

Wir erwarten, dass Heapsort nur dann schneller ist als Quicksort, wenn die fuer Quicksort unguenstigsten Bedingungen bestehen.

Abbildung 3: Heap Sort

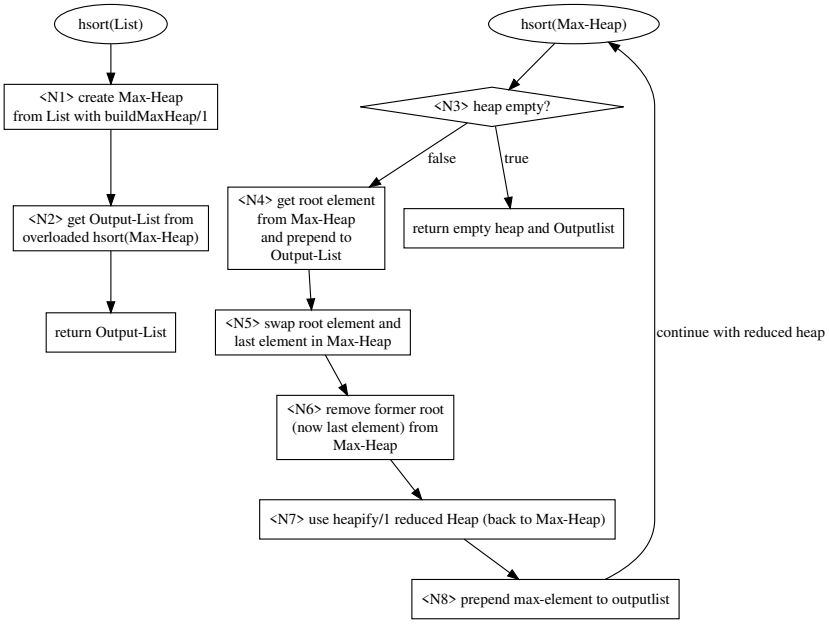


Abbildung 4: Heapify

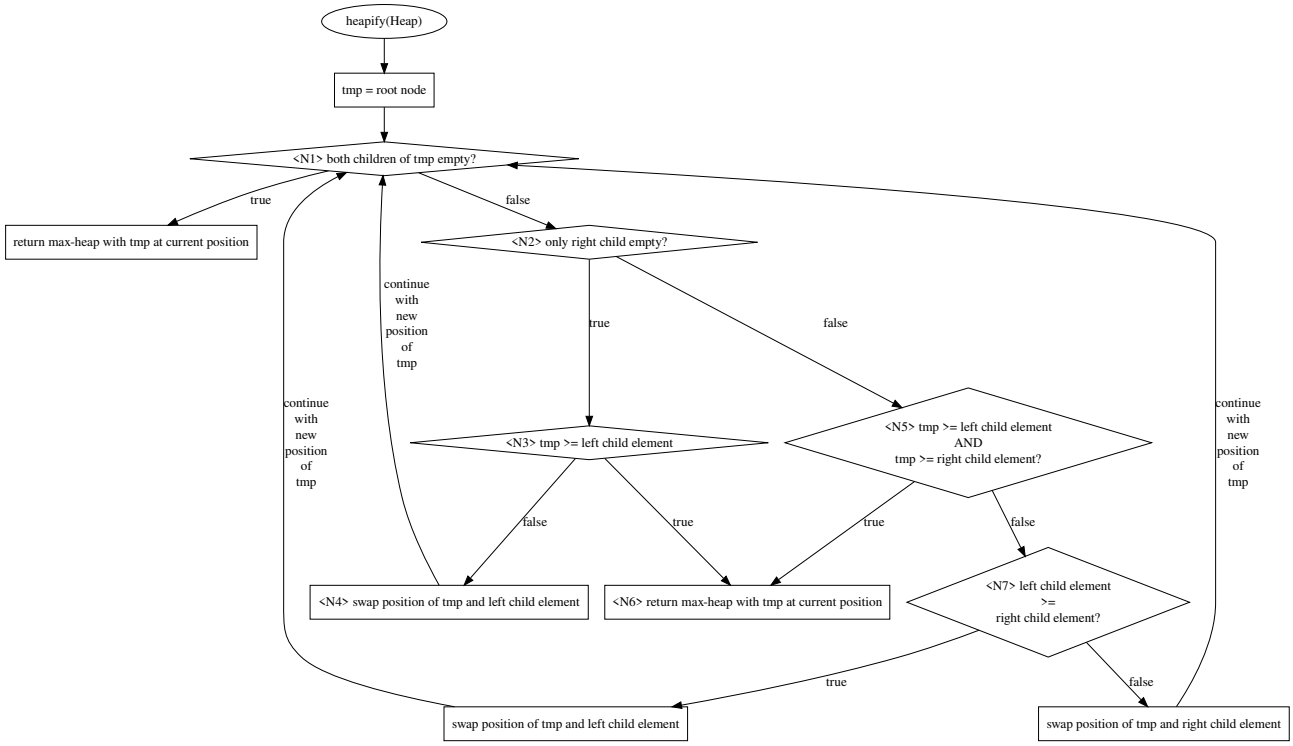
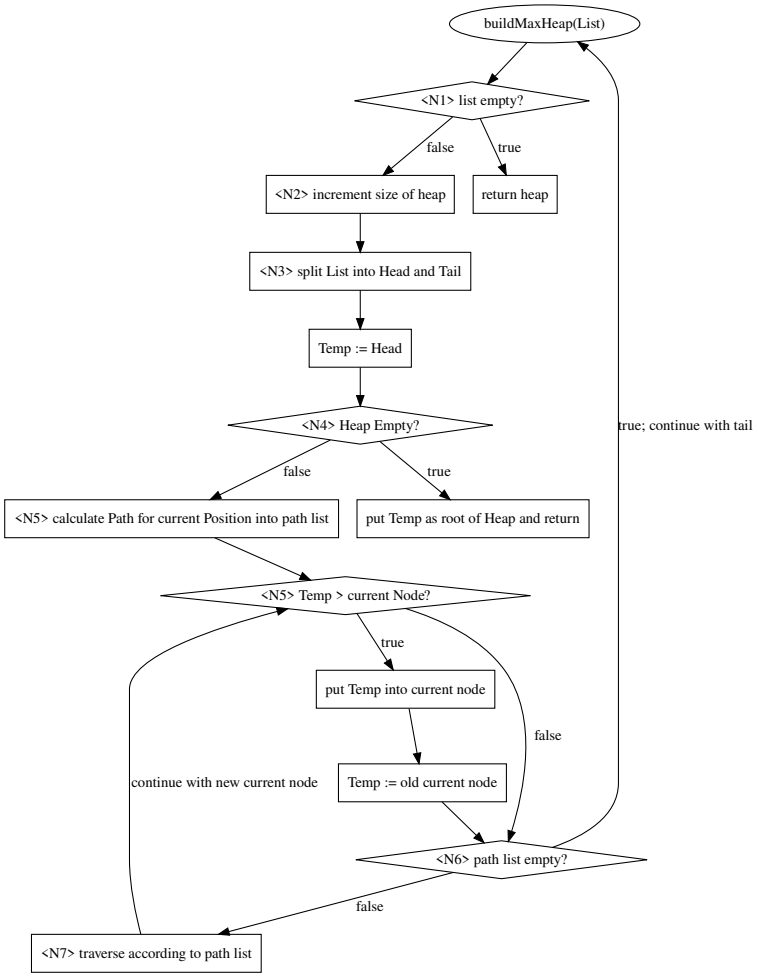


Abbildung 5: Build Max-Heap



3 Laufzeitmessung

3.1 Insertion Sort

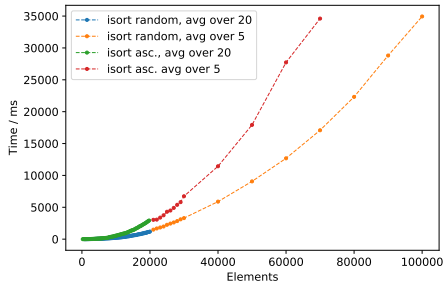
3.1.1 Komplexität im Average und Worst Case

In Abbildung 6a ist die Laufzeit mit zufälligen Zahlen und aufsteigenden Zahlen zu sehen. Der Plot sieht auf den ersten Blick quadratisch aus. Diese Vermutungen bestätigt sich beim Betrachten von Abbildung 12b, in der die Messung mit zufälligen Zahlen als Log-Log-Plot dargestellt ist. Da die Steigung der Kurve ähnlich der Steigung der quadratischen Funktion ist, lässt sich daraus schließen, dass eine quadratische Komplexität vorliegt.

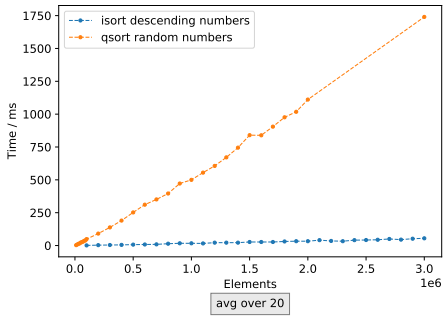
Dies bestätigt unsere Vermutungen aus dem Entwurf.

Abbildung 6: Insertion Sort

(a) average case, worst case



(b) best case



3.1.2 Komplexität im Best-Case

Der Best-Case ist in Abbildung 6b dargestellt. Dieser tritt auf, wenn die Liste absteigend sortiert ist. Dies lässt sich an der Erlang Implementation erklären, da bei absteigenden Zahlen mit der *insertToList()* Methode jeweils kleinere Zahlen in den sortieren Abschnitt eingefügt werden. Diese Operation hat eine Komplexität von $\Theta(1)$, da die kleinere Zahl der Liste vorangestellt wird. Somit ergibt sich bei absteigenden Zahlen eine Gesamtkomplexität von $\Theta(n)$, da dieser diese Methode für alle Elemente der Liste ein Mal ausgeführt wird.

Bei aufsteigenden Zahlen muss die *insertToList()* Methode hingegen jeweils bis zum Ende des sortieren Abschnitts laufen, da die Elemente jeweils größer werden, wodurch eine Gesamtkomplexität von $\Theta(n^2)$ entsteht.

Zusammenfassen haben wir also festgestellt, dass sich die Laufzeit unserer Implementation von Insertion Sort durch $O(n^2)$ und $\Omega(n)$ beschreiben lässt, was unseren Erwartungen entspricht.

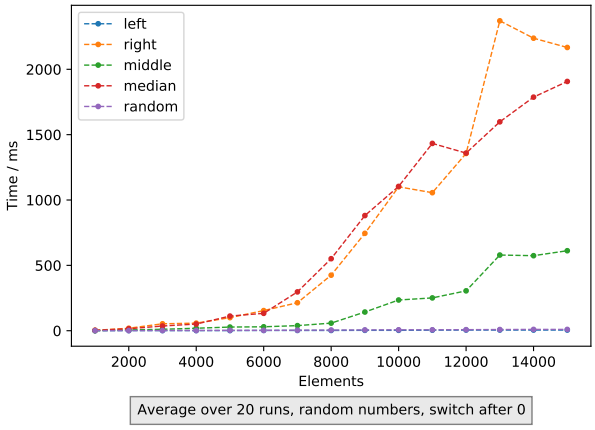
3.2 Quick Sort

Zur Analyse der finalen Implementation, überspringe bis Abschnitt 3.2.2.

3.2.1 Pivot Methoden - Anfängliche Probleme

Erste Implementation Bei der ersten Implementation vom Quicksort Algorithmus ist bei der Verwendung der Pivot-Methoden *right* und *median* eine deutlich längere Laufzeit aufgefallen (siehe Abbildung 7). Gleichzeit ist die Laufzeit von der *middle* Pivot Methode besser, obwohl die Erwartung dieser schlechter ist: Diese muss eineinhalb mal – einmal zum Finden der Länge l komplett, danach bis zum $l/2$ Element – durchlaufen werden. Die Pivot Methode *right* muss nur einmal bis zum Ende der Liste laufen.

Abbildung 7: Vergleich der Pivot Methoden – erste Implementation



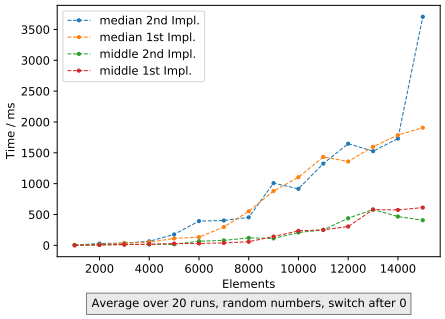
Unsere erste Vermutung war, dass unsere Methoden, die in einem Durchlauf die Länge und gleichzeitig dass Pivot-Element ermitteln dafür verantwortlich sein könnten. In der Methoden zum Finden des letzten Elementes, die auch von der median Pivot Methode verwendet wird, benutzten wir Pattern Matching, um zwischen dem Letzten und vorletztem Element zu unterscheiden. Dabei muss allerdings jeweils vorausgeschaut werden, was die schlechte Performance erklären könnte.

Somit die Vermutung, dass man die Performance verbessern kann, indem man zunächst die Länge der Liste ermittelt und anschließend diese zum Finden des n-ten Elements ein zweites Mal durchläuft.

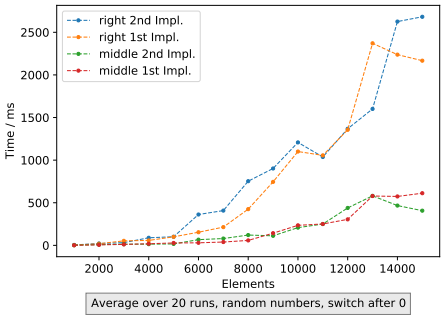
Zweite Implementation In der zweiten Implementation wurden die Ermittlung des Pivots und das Finden der Länge der Liste getrennt. Das letzte Element wurde nun ermittelt, indem zunächst die Länge l der Liste berechnet, anschließend mithilfe dieser zum $l - 1$ -ten Element gelaufen wird.

Abbildung 8: Zweite Implementation vs. erste Implementation

(a) median



(b) right

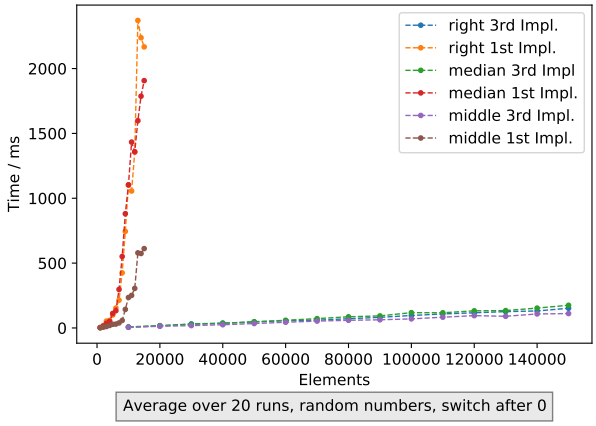


In Abbildung 8a und 8b sind jeweils die Ergebnisse der Laufzeitmessung der zweiten Implementation von *median* und *right* zu sehen. Damit systematische Unterschiede zwischen den beiden Kompilationen ausgeschlossen sind, ist zusätzlich die gleiche Implementation, von *middle* dargestellt. Bei beiden überarbeiteten Implementationen ist keine Besserung der Laufzeit zu erkennen, was darauf schließen lässt, dass das in der ersten Implementation verwendete Pattern Matching nicht der Grund der schlechten Performance war.

Weitere Mögliche Ursachen Eine weitere mögliche Ursache für die Ergebnisse ist der Fakt, das bei der Methode zum Finden des n-ten Elementes und des Restes *listGetNthAndRest()* die Erlang List Concatenation (++) verwendet wird, und somit bei jedem Rekursionsschritt ein zusätzlicher Aufwand von n, wobei n die Anzahl der Elemente in der Subliste darstellt, hinzukommt. Dies würde auch erklären, warum *middle* eine bessere Laufzeit aufweist, da dabei nur ein zusätzlicher Aufwand von $n/2$ halbe dazukommt.

Dritte Implementation In der dritten Implementation wurde die List Concatenation ersetzt.

Abbildung 9: Dritte Implementation vs. erste Implementation



Wie in Abbildung 9 zu sehen, verbessert dies die Laufzeit dramatisch. Die Laufzeit der Pivot Methode *middle* hat sich auch verbessert, was darauf zurückzuführen ist, dass diese ebenfalls die *listGetNthAndRest ()* Methode verwendet.

3.2.2 Vergleich der Pivot Methoden

In Abbildung 10 ist die Laufzeit der Pivot Methoden der dritten Implementation mit zufälligen Zahlen dargestellt. Der Worst-Case ist des Weiteren in Abschnitt 3.2.3, insbesondere in Abbildung 11b dargestellt.

Die Pivot Methoden *left* und *random* weisen die beste Laufzeit auf, die Methoden median und right die schlechteste. Die Laufzeit von *middle* liegt zwischen den anderen.

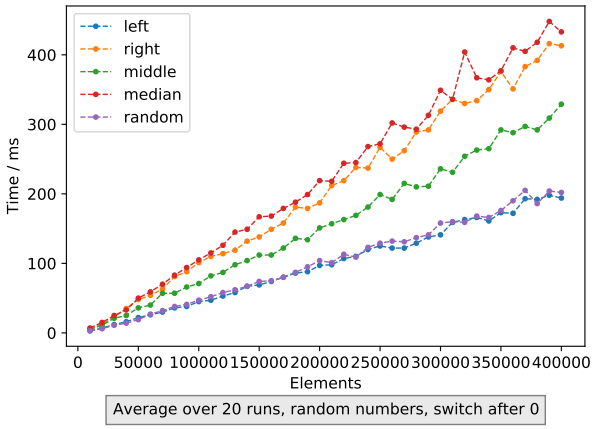
Right und Median Die schlechte Laufzeit von *right* und *median* lässt sich dadurch erklären, dass die Liste zum Finden des Pivots insgesamt zweimal komplett durchlaufen wird: Zunächst zum Finden der Länge, anschließend zum Finden des letzten Elementes.

Middle Bei *middle* wird die Liste nur eineinhalb Mal durchlaufen, da beim zweiten Durchlauf nur bis zur Mitte gelaufen wird. Dies erklärt die bessere Laufzeit im Vergleich zu den obigen Methoden.

Left und Random Die gute Laufzeit von *left* lässt sich dadurch erklären, dass diese bei zufälligen Zahlen den geringsten Aufwand benötigt, da die Liste zum Finden des Pivots nicht durchlaufen werden muss.

Die fast identische Laufzeit von *random* ist jedoch überraschend, da dabei schließlich im Mittel zusätzlich bis zum $\frac{n}{2}$ Element gelaufen wird. Somit scheint *random* eine gute Alternative zu *left* zu sein, da dadurch der Worst-Case von *left* (aufsteigend sortierten Listen) umgangen wird.

Abbildung 10: Vergleiche der Pivot Methoden – Dritte Implementation



3.2.3 Switch Number

Bei der Implementation des Quicksort-Algorithmus wird ein Parameter *switchnumber* übergeben, welches den Schwellenwert angibt, ab wann von Quicksort auf Insertion Sort umgeschaltet werden soll.

Average Case In Abbildung 11a sind jeweils Laufzeitmessungen mit variierender *switchnumber* zu sehen. Es wird deutlich, dass im Average-Case die optimale *switchnumber* bei ca. 20–50 Elementen liegt.

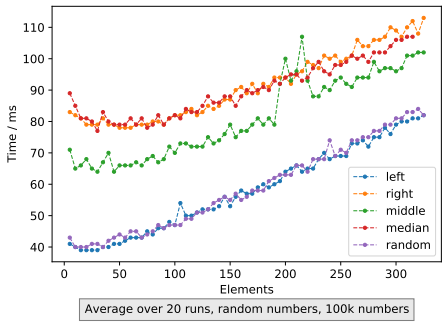
Interessant ist dabei auch, dass die optimale *switchnumber* mit den effizienteren Pivot Methoden niedriger ist, als mit den ineffizienteren. Das lässt sich damit erklären, dass bei letzteren der Overhead pro Iteration größer ist. Da Insertion Sort einen geringen Aufwand pro Iteration aufweist, ist dieser trotz schlechterer Komplexität länger im Vorteil.

Worst Case In Abbildung 11b ist die *switchnumber* im Worst-Case von Quicksort untersucht worden. An den Daten zu absteigenden Zahlen befindet sich Insertion Sort im Best-Case und hat somit eine Laufzeit von $\Omega(n)$, dies erklärt, warum der Verlauf mit zunehmender *switchnumber* scheinbar gegen 0ms läuft. (Zu Details zum Best-Case von Insertion Sort, siehe Abbildung 6b in Abschnitt 3.1)

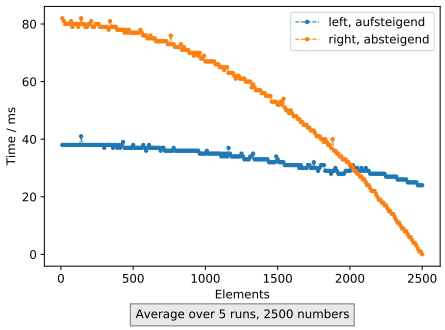
An den Daten zu aufsteigenden Zahlen befindet sich Insertion Sort im Average-Case und hat eine Komplexität von $O(n^2)$. Da Quicksort im Worst-Case ebenfalls eine Komplexität von $O(n^2)$ aufweist, wird hierdurch verdeutlicht, dass der Aufwand pro Iteration, also der Faktor vor dem n, bei Insertion Sort geringer ist als bei Quicksort.

Abbildung 11: Vergleich der Switch Number

(a) Average Case



(b) Worst Case



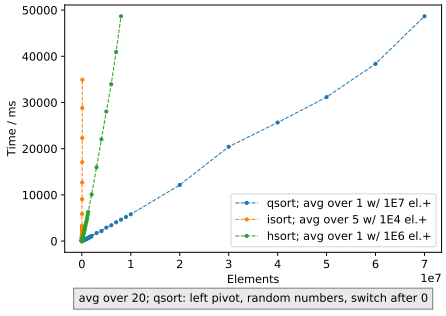
3.2.4 Komplexität im Best Case

An dem linearen Plot (Abbildung 12a) ist kaum zu erkennen, ob eine lineare oder über-logarithmische Komplexität vorliegt. Wir haben uns erhofft, dies mithilfe eines Log-Log-Plots (Abbildung 12b) genauer zu untersuchen. Auf solchem ist anhand der Steigung des Graphen der Exponent zu erkennen. Somit müsste die Steigung unserer Messkurve zwischen den beiden Funktionen $f(x) = x$ und $f(x) = x^2$ liegen, falls diese über-logarithmisch ist.

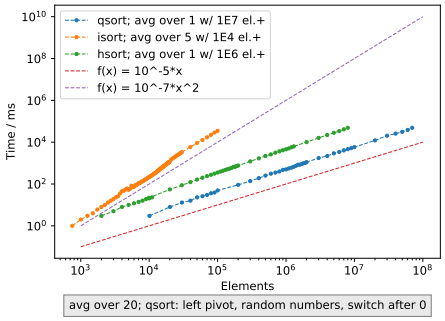
Leider lässt sich auch auf diesem nicht zwischen dem linearen und, nach unserer Erwartung, über-logarithmischen Messkurve unterscheiden.

Abbildung 12: Best Case

(a) Linearer Plot



(b) Log-Log Plot



3.2.5 Komplexität im Worst Case

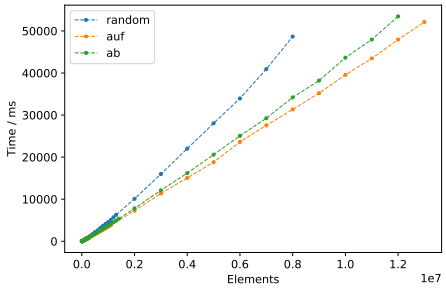
Die Laufzeit zum Worst Case wurde kurz im Abschnitt 3.2.3 besprochen. Aus zeitlichen Gründen werden wir nicht weiter darauf eingehen.

3.3 Heap Sort

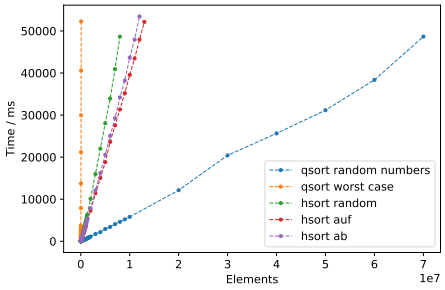
In Abbildung 13a ist die Laufzeit von Heapsort mit zufälligen Zahlen, absteigenden Zahlen und aufsteigenden Zahlen dargestellt. In diesem Test geht es zunächst darum, das Komplexitätsverhalten von Heapsort zu untersuchen und weitergehend den Vergleich zwischen Heap- und Quicksort abzubilden.

Abbildung 13: Heapsort

(a) Heapsort



(b) Heapsort und Quicksort



3.3.1 Komplexität im Avg Case

Betrachten wir zunächst den Average-Case von Heapsort. Hier wurden zufällig generierte Listen verwendet, um eine durchschnittliche Laufzeit-Komplexität zu ermitteln. Anders als in Abbildung 12a, in der es ohne Regression nicht möglich ist, Aussagen über das Komplexitätsverhalten zu treffen, lässt sich in Abbildung 13a mit bloßem Auge eine Krümmung der Kurve erkennen. Dies deutet auf die von uns erwartete Laufzeit von $O(n \cdot \log\ n)$ hin.

3.3.2 Komplexität im Best Case

Außerdem in Abbildung 13a zu sehen ist der Best-Case. Hier sind die Ergebnisse ein wenig interessanter.

Zunächst betrachten wir jene Ergebnisse die unseren Erwartungen entsprechen. Der bei einem Heapsort übliche Best-Case ist eine invertiert-sortierte Liste, also eine Liste von absteigenden Zahlen. Beim Heapsort spielt das Bauen von und Operieren auf einem Max-Heap eine zentrale Rolle. Das größte Element ist die Wurzel des Heaps und die Elemente werden abwärts immer kleiner. Da dieser Struktur eine absteigende Anordnung von Zahlen in einer Liste am nächsten kommt, sollte hier das Bauen des Max-Heaps die effizienteste Laufzeit besitzen.

Unerwartete Ergebnisse erhalten wir jedoch beim Sortieren einer bereits sortierten bzw. aufsteigend-sortierten Liste. Da die Struktur des binären Max-Heaps eher einer absteigenden Liste ähnlich ist und dies einen kleineren Aufwand beim Erstellen des Heaps mit sich bringt, ließ sich ein gegenteilig grosser Aufwand bei einer aufsteigenden Liste vermuten. Dies ist jedoch nicht der Fall gewesen. In Abbildung 13a lässt sich aufsteigend klar eine bessere Laufzeit erkennen als absteigend, ob wohl absteigend intuitiv als effizienter hervortritt.

Ergründen lässt sich dies in der Natur unserer Implementierung und dem Verlauf unseres Projektes. Zunächst erstellten wir im Entwurf des Heapsort-Algorithmus eine Bottom-Up-Herangehensweise der Max-Heap-Konstruktion. In der Implementation haben wir uns jedoch für eine Top-Down-Herangehensweise entschieden. Wir nehmen stark an, dass sich die Ergebnisse bei einer Implementation des ursprünglichen Entwurfs mit unseren Erwartungen gedeckt hätten.

3.3.3 Vergleich mit Quicksort

Für den Vergleich von Quicksort und Heapsort haben wir einerseits bei Quicksort den Worst- und Average-Case und andererseits bei Heapsort die Messungen aus Abbildung 13a benutzt.

Betrachte Abbildung 13b. Bei zufällig generierten Listen ist Quicksort, wie erwartet, deutlich performanter. Bei 10 Mio. Elementen dauert eine Heap-Sortierung schon fast eine Minute, während Quicksort noch bei unter 10 Sekunden liegt.

Stellt man jedoch den schlechten Bedingungen für Quicksort her, liegt (aufgrund von der bereits angesprochenen quadratischen Laufzeitkomplexität von Quicksort im Worst-Case) Heapsort mit der Performanz deutlich vorne. Die im Entwurf vermuteten Erwartungen sind demnach eingetroffen.