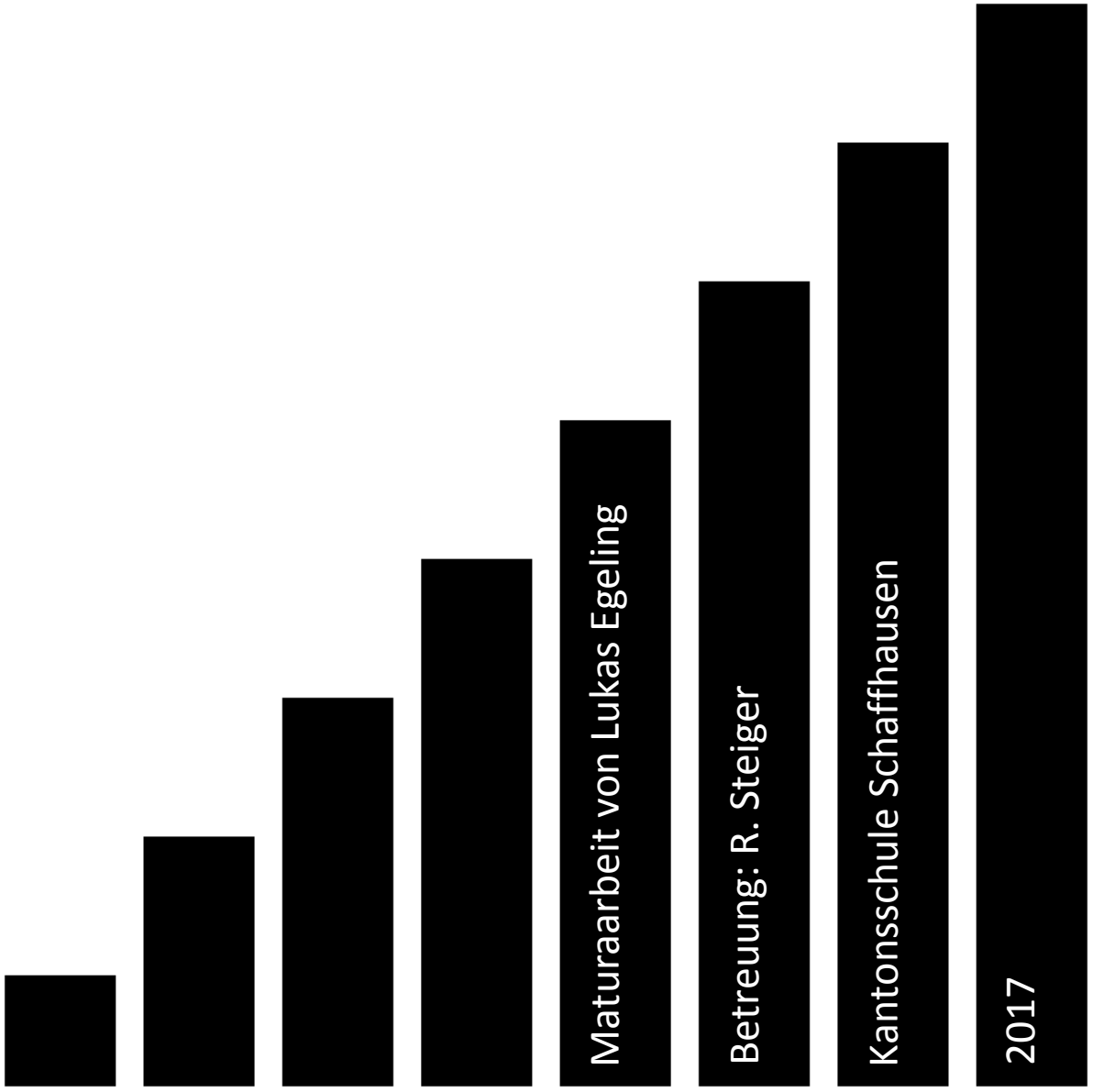
Die Welt der Sortieralgorithmen



Inhalt

1 Einführung

1.1 Was sind Sortieralgorithmen?

In der Informatik tauchen öfter Situationen auf, in denen Listen von Zahlen sortiert werden müssen: das Erstellen einer Rangliste, Auflisten von Dateien nach ihrer Grösse oder die Auswertung von Resultaten eines wissenschaftlichen Versuchs. In all diesen Fällen müssen die Werte mit dem geringstmöglichen Aufwand geordnet werden, besonders dann, wenn tausende oder sogar Millionen von Zahlen zu sortieren sind. Dies macht ein Sortieralgorithmus, welcher eine Liste von beliebigen Zahlen annimmt und sie in aufsteigender Reihenfolge wieder ausgibt. Es gibt nicht nur einen einzigen Sortieralgorithmus, sondern eine ganze Menge, wovon einige effizienter sind und andere weniger. Manche wiederum sind nicht ganz ernst gemeint und zum Sortieren von Listen mit mehr als ein paar Elementen völlig untauglich. Ein solcher Algorithmus ist der Bogo Sort, welcher die Liste zufällig durcheinandermischt, bis sie sortiert ist. Letztere werden daher nicht in dieser Arbeit behandelt. Die Inspiration zum Thema Sortieralgorithmen ist ein YouTube-Video, das «15 Sorting Algorithms in 6 Minutes» heisst. Dieses stellt den Sortiervorgang von 15 Sortieralgorithmen mit Bild und Ton dar.

1.2 Fachbegriffe

Für die Beschreibung der Eigenschaften von Sortieralgorithmen gibt es eine Menge von Fachbegriffen. Diese werden nun erklärt und im Rest der Arbeit benutzt. Die meisten dieser Begriffe sind standardisiert, werden also von allen Spezialisten verstanden. Die Unordnung einer Liste wird hier aber unabhängig definiert und eingesetzt.

Jedes Element einer Liste hat zwei wichtige Bestandteile, nämlich den Index, der die Stelle innerhalb der Liste angibt, und den Schlüssel, der den Wert anzeigt. Der Zeiger gibt an, welches Element ein Algorithmus gerade betrachtet. Die Anzahl Elemente in der Liste wird n genannt. Den meisten Sortieralgorithmen liegen zwei grundlegende Operationen zugrunde. Die eine ist der Vergleich. Dabei werden zwei Schlüssel verglichen und je nachdem, ob der erste grösser als, kleiner als, oder gleich gross wie der zweite ist, kann der Algorithmus unterschiedlich verfahren. Die andere Operation ist der Tausch zweier Schlüssel. So wird die Liste in die sortierte Permutation gebracht. Da Sortieralgorithmen zum Grossteil diese zwei Vorgänge benutzen, kann anhand der Anzahl Vergleiche und Tausche bei der Sortierung verschieden grosser Listen ungefähr abgeschätzt werden, wie effizient ein Algorithmus ist.

Der wichtigste Faktor zur Beschreibung der Effizienz eines Sortieralgorithmus ist die Komplexität, wovon es zwei Arten gibt. Beide werden in der Form $O * f(n)$ dargestellt. Dabei ist Oein Proportionalitätsfaktor und $f(n)$ eine Funktion bezüglich der Anzahl Elemente, die in der Liste enthalten sind. Die Zeitkomplexität beschreibt den nötigen Aufwand zum Sortieren einer Liste. Ein Beispiel einer möglichen Zeitkomplexität ist $O * n^2$. Das bedeutet, dass der Zeitaufwand eines Sortieralgorithmus quadratisch mit der Grösse der Liste wächst. Auf diese Weise kann der durchschnittliche, maximale sowie minimale Aufwand dargestellt werden. $O * n$ ist der heilige Gral der Sortieralgorithmen, kann aber im Durchschnitt nicht erreicht werden, höchstens beim Best-Case Szenario. Es gibt auch die Platzkomplexität, welche den

benötigten Speicherplatz ausserhalb der zu sortierenden Liste darstellt. Einfache Algorithmen benötigen meist nur eine bestimmte Anzahl von Hilfsvariablen, haben also eine Platzkomplexität von $O * 1$. Bei rekursiven Verfahren ist sie meistens $O * \log n$, da jede Schicht der Rekursion ihre eigenen Variablen hat. Da sie die Liste meistens in zwei Teile teilt, erreicht die Rekursion häufig eine maximale Tiefe von $\log_2 n$. Manche Algorithmen arbeiten mit einer versteckten Liste, welche nur temporär benutzt wird, während Teile der ursprünglichen Liste verändert werden. Da die versteckte Liste alle Elemente enthalten kann, haben solche Algorithmen meistens eine Platzkomplexität von $O * n$.

Ein weiteres Kriterium, wonach Sortieralgorithmen klassifiziert werden können, ist die Stabilität. Stabiles Sortieren bedeutet, dass gleich grosse Schlüssel ihre relative Reihenfolge niemals ändern. Kann dies doch geschehen, handelt es sich um einen instabilen Algorithmus.

Ein neuer Begriff, der nun unabhängig von anderen Quellen definiert wird, ist die Unordnung einer Liste, welche alle Zahlen 1 bis n je einmal enthält. Als absolute Unordnung U_{abs} wird die Summe der Differenzen zwischen den Indizes von Elementen und ihren Schlüsseln definiert. Algebraisch ausgedrückt ist das $U_{abs} = \sum_{i=1}^{n} |i - S_i|$, wobei S_i der i-te Schlüssel ist (da Computer Indizes von 0 aus aufzählen statt von 1, müssen diese in einem Programm um 1 verringert werden). Die höchstmögliche absolute Unordnung U_{max} hat eine Liste in der absteigenden Permutation. Sie entspricht $\left\lfloor \frac{n^2}{2} \right\rfloor$. Der Beweis dafür folgt im nächsten Abschnitt. Für die relative Unordnung gilt $U_{rel} = \frac{U_{abs}}{U_{max}}$. Daher ist sie immer zwischen 0 und 1. Ein Algorithmus wird unordnungsstet genannt, wenn sich die Unordnung während des Sortiervorgangs niemals vergrössert. Wird ein unordnungssteter Sortieralgorithmus mit einer bereits sortierten Liste konfrontiert, ändert sich beim Sortieren die relative Unordnung deshalb nie.

Beweis dass $U_{max} = \left\lfloor \frac{n^2}{2} \right\rfloor$: Bei einer verkehrten Liste mit einer geraden Anzahl Elemente haben das erste und $\frac{n}{2} + 1$-te Element zusammen eine absolute Unordnung von n, genauso das zweite und $\frac{n}{2} + 2$-te, usw. bis zum $\frac{n}{2}$-ten und n-ten. Weil es $\frac{n}{2}$ solcher Paare gibt, ist die maximale Unordnung $n * \frac{n}{2}$, vereinfacht $\frac{n^2}{2}$. Hat die verkehrte Liste aber eine ungerade Anzahl Elemente, haben das erste und $\frac{n+1}{2} + 1$-te Element zusammen eine absolute Unordnung von $n + 1$, genau wie das zweite und $\frac{n+1}{2} + 2$-te, usw. bis zum $\frac{n+1}{2} - 1$-ten und n-ten. Es gibt $\frac{n-1}{2}$ solcher Paarungen, deshalb ist hier die maximale Unordnung $(n + 1) * \frac{n-1}{2}$, was sich zu $\frac{n^2}{2} - \frac{1}{2}$ vereinfachen lässt. Weil n ungerade ist, ergibt dieser ganze Ausdruck eine ganze Zahl. Daher können $\frac{n^2}{2}$ und $\frac{n^2}{2} - \frac{1}{2}$ zu $\left\lfloor \frac{n^2}{2} \right\rfloor$ vereinigt werden. Damit ist der Beweis erbracht.

2 Programm für Sortieralgorithmen

Um selber Sortieralgorithmen testen zu können, wurde eigens für diese Arbeit ein Programm erstellt, mit welchem die Algorithmen geprüft werden können. Dieses wird für alle nachfolgenden Analysen verwendet. Es wurde in der Programmiersprache Python geschrieben, da diese im Informatikunterricht benutzt wurde und nicht allzu schwierig ist. Nun wird das Programm genauer betrachtet.

2.1 Aufbau des Programms

Das Gesamtpaket mit allem, was dazu gehört, befindet sich in einem Ordner mit dem Namen «Sortieralgorithmen». Darin enthalten sind ein Python-Programm, das dazugehörige Titelbild, ein weiterer Ordner mit den Sortieralgorithmen sowie zwei Textdateien, eine für die Testresultate, die das Programm ausgibt, und eine, welche die aktuellen Einstellungen speichert. Weil die Algorithmen als separate Python-Programme in einem Ordner enthalten sind, statt direkt im Hauptprogramm einprogrammiert, können neue Sortierverfahren leicht zu den bereits existierenden hinzugefügt und getestet werden, ohne andere Dateien zu bearbeiten. Allerdings müssen alle Algorithmen so angepasst werden, dass sie wichtige Daten wie die Anzahl Vergleiche und Tausche oder die benötigte Zeit festhalten und ausgeben können. Die Textdatei für den Output des Programms erlaubt die Auswertung und Speicherung von Resultaten. Das Gesamtpaket kann mit der folgenden Internetadresse als Zip-Datei heruntergeladen werden: http://s000.tinyupload.com/?file_id=29208404108172331393.

2.2 Benutzeroberfläche

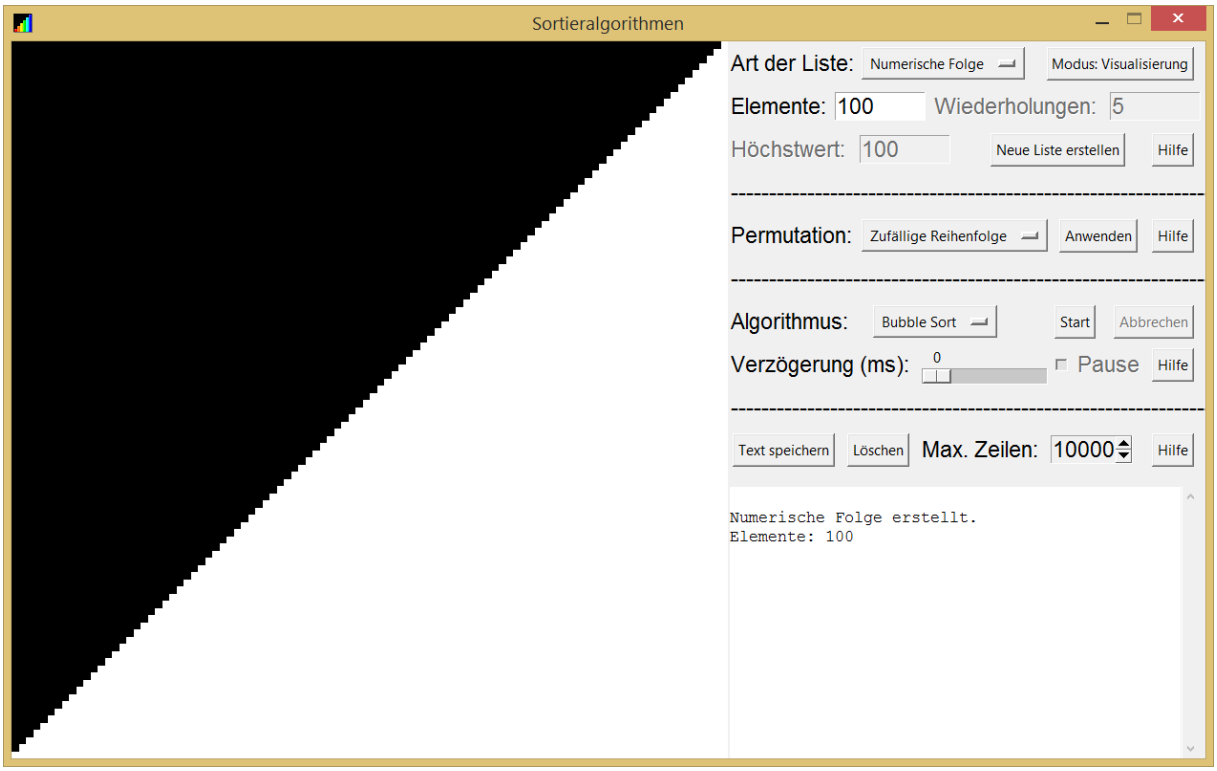


Abbildung 2.2: Die Benutzeroberfläche

Die Benutzeroberfläche des Programms besteht aus zwei wichtigen Bestandteilen: dem Anzeigebereich und dem Einstellungsbereich. Der Anzeigebereich nimmt die linken 60% vom Programmfenster ein. Dieser zeigt je nach Modus verschiedene Daten an. In Abbildung 2.2 stellt dieser im Visualisierungsmodus die Liste bildlich dar. Den Rest des Fensters nimmt der Einstellungsbereich ein, der seinerseits aus vier kleineren Bereichen besteht, welche durch gestrichelte Linien unterteilt sind. Jeder dieser Unterbereiche hat einen Hilfebutton, welcher

beim Anklicken die jeweilige Anleitung einblendet. Ein erneutes Anklicken desselben Buttons blendet sie wieder aus.

Der oberste Teilbereich der Einstellungen betrifft das Erstellen einer neuen Liste. Der Button ganz oben rechts ändert beim Anklicken den Modus, wobei spezifische Einstellungen aktiviert oder deaktiviert werden. Im Visualisierungsmodus kann man zwischen drei verschiedenen Listenarten auswählen: Eine numerische Folge, eine Folge mit wiederholten Werten und eine Folge zufällig verteilter Zahlen. Die numerische Folge enthält jede natürliche Zahl von 1 bis zum Wert von «Elemente» je einmal. In der Liste mit wiederholten Werten kommt jede dieser Zahlen so oft vor, wie in «Wiederholungen» eingegeben wurde. Das dritte, mit «Höchstwert» angeschriebene Eingabefeld wird für die Liste von zufälligen Zahlen benutzt. Es gibt den Höchstwert für die natürlichen Zahlen an, von denen so viele wie bei «Elemente» angegeben in der Liste verteilt werden. Im Zeitmessungs- und Unordnungsmodus ist aber nur die numerische Folge verfügbar. Im Zeitmessungsmodus werden die Beschriftungen der drei Eingabefelder durch die Parameter der Zeitmessung ersetzt. Diese werden in 2.4 genauer erklärt. Erst mit einem Mausklick auf den «Neue Liste erstellen»-Button treten die eben erwähnten Parameter und der ausgewählte Modus in Kraft. Ausserdem wird die Liste gemäss den Einstellungen generiert und im Visualisierungsmodus im Anzeigebereich dargestellt. Die Liste wird immer schon in der sortierten Permutation generiert.

Im Visualisierungs- und Unordnungsmodus ist der zweitoberste Teilbereich der Einstellungen verfügbar, in welchem eine von vier Permutationen der Liste ausgewählt werden kann. Zur Auswahl stehen eine zufällige Permutation, die verkehrte Reihenfolge, eine zufällige fast sortierte Permutation sowie die ganz sortierte. Das häufigste Szenario in den Anwendungen von Sortieralgorithmen ist eine zufällige Reihenfolge. So kann die allgemeine Effizienz eines Algorithmus ermittelt werden. Mit den übrigen Permutationen kann getestet werden, wie gut er mit speziellen Situationen umgehen kann. Der mit «Anwenden» beschriftete Button bringt die Liste in die gewünschte Permutation und zeichnet sie im Visualisierungsmodus neu.

Im nächsten Teilbereich der Einstellungen befinden Sich die Einstellungen zum eigentlichen Sortiervorgang. Falls der Ordner, der die Algorithmen enthält, leer ist, werden diese Einstellungen durch einen Warnhinweis ersetzt. Sofern Sortieralgorithmen verfügbar sind, kann einer davon ausgewählt werden. Ein Mausklick auf den «Start»-Button beginnt die Sortierung. Diese kann mit dem entsprechenden Button pausiert oder abgebrochen werden. Mit dem Schieber unten links kann eingestellt werden, wie viel Zeit in Millisekunden nach jedem Sortierschritt vergehen soll. Bei 0 Millisekunden wird der Algorithmus so schnell wie möglich ausgeführt. Dieser Schieber ist im Zeitmessungsmodus nicht verfügbar.

Zuunterst im Einstellungsbereich befinden sich das Textausgabefeld und die dazugehörigen Einstellungen. Dort werden Testresultate und sonstige Ereignisse als Text ausgegeben. Der darin enthaltene Text kann durch Anklicken des «Text speichern»-Buttons zum Inhalt der dafür gedachten Textdatei hinzugefügt werden. Der «Löschen»-Button hingegen leert das Textausgabefeld. Neben der «Max. Zeilen»-Beschriftung kann dessen Kapazität in Schritten von 1000 Zeilen vergrössert oder verkleinert werden. Falls der Text diese um eine gewisse Anzahl Zeilen überschreitet, werden die ältesten entsprechend gelöscht.

2.3 Visualisierungsmodus

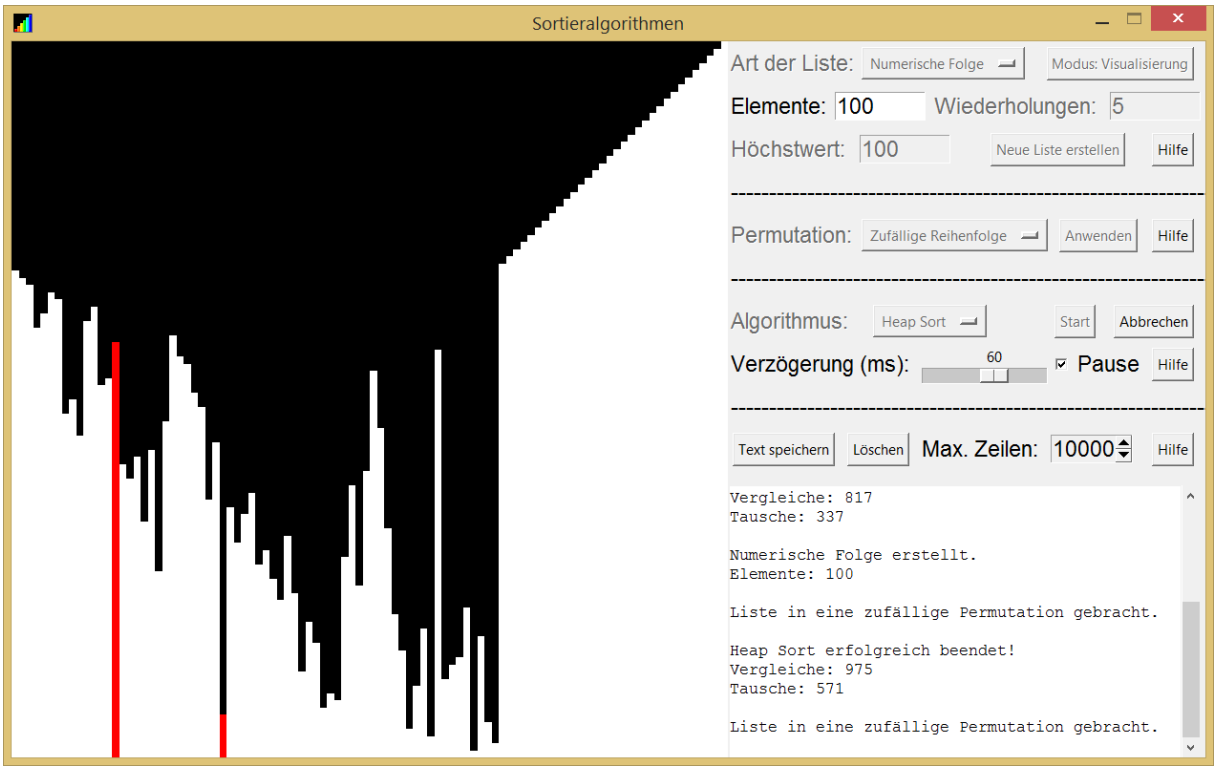


Abbildung 2.3: Visualisierung des Heap Sort

Der Visualisierungsmodus stellt die Liste im Anzeigebereich als Balkendiagramm dar. Die Länge der Balken entspricht der Grösse der Schlüssel: kurze Balken stehen für kleine Schlüssel, lange für grosse. Nach dem Erstellen einer neuen Liste in diesem Modus oder der Veränderung der Permutation erscheint das entsprechende Diagramm im Anzeigebereich. Wenn keine Sortierung aktiv ist, kann die visualisierte Liste manuell bearbeitet werden. Wenn der Nutzer zwei verschiedene Balken nacheinander anklickt, werden diese vertauscht. Wenn man aber einen Balken mit der Maus durch den Anzeigebereich zieht, wird dieser nach dem Loslassen beim Mauszeiger eingeschoben. Alle Änderungen in der Anzeige werden auch an der Liste selber vorgenommen.

Wenn ein Sortiervorgang aktiv ist, lässt sich die Liste nicht mehr manuell bearbeiten. Dafür werden die Balken zu den Elementen, die gerade vom Sortieralgorithmus behandelt werden, farbig markiert. Im Visualisierungsmodus kann die Geschwindigkeit der Sortierung angepasst werden. Sie lässt sich aber auch ganz anhalten. Dies dient dazu, dass der Benutzer des Programms genauer nachvollziehen kann, wie der gewählte Sortieralgorithmus funktioniert. Passiert ein Error während des Prozesses, wird er sofort abgebrochen. Der Grund dafür ist fast immer ein schlecht programmierter Algorithmus. Ist der Sortiervorgang beendet, prüft das Hauptprogramm zunächst, ob sich die Liste tatsächlich in der sortierten Permutation befindet. Wenn nicht, wird der Sortierprozess als fehlerhaft bezeichnet. Danach werden die Resultate in der Textausgabe gemeldet. Nebst der Information, ob die Sortierung erfolgreich oder fehlerhaft war, ein Error vorkam oder der Benutzer den Abbruch-Button anklickte, wird wenn möglich auch die Anzahl von Vergleichen und Tauschen ausgegeben.

2.4 Zeitmessungsmodus

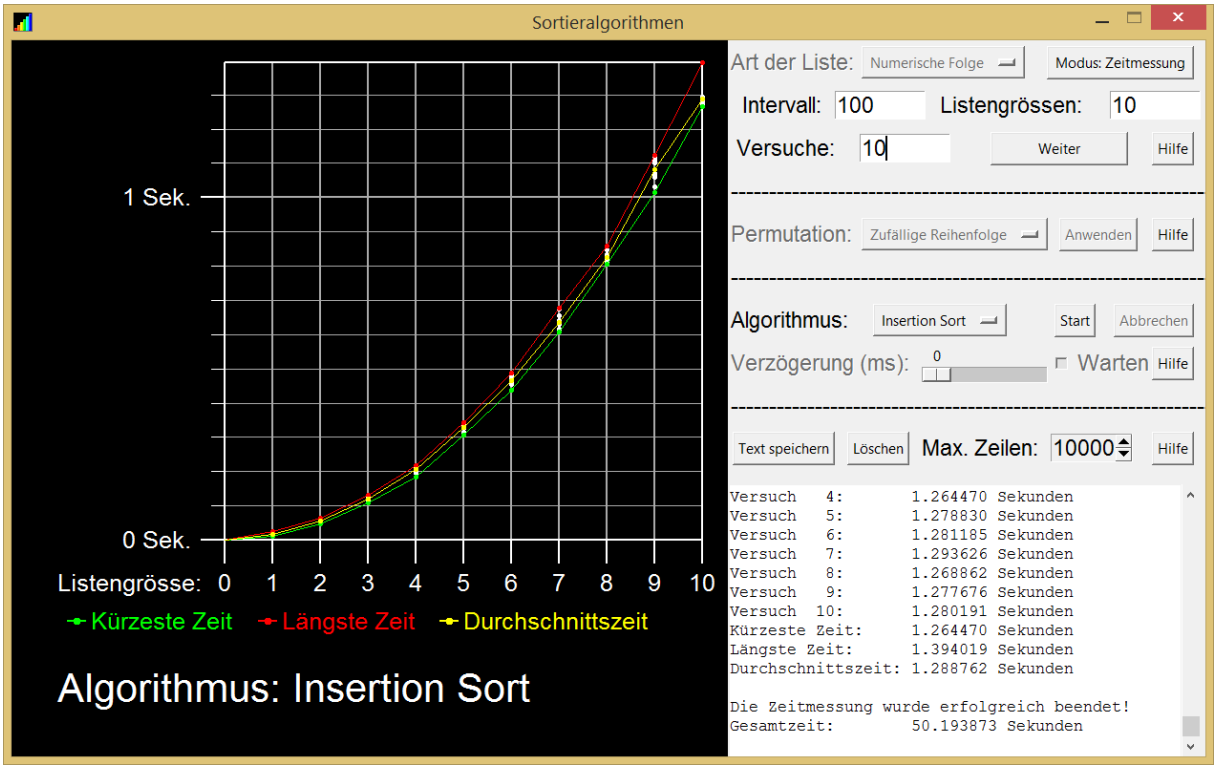


Abbildung 2.4: Resultate der Zeitmessung des Insertion Sort

Der Zeitmessungsmodus dient dazu, einen Sortieralgorithmus mehrmals hintereinander an verschieden grossen zufälligen Listen auszuführen, um dessen Zeitkomplexität zu ermitteln. Die Textfelder zuoberst in den Einstellungen wechseln in diesem Modus ihre Beschriftung. Wie in Abbildung 2.4 zu sehen sind sie nun mit «Intervall», «Listengrössen» und «Versuche» angeschrieben. Der Wert in «Intervall» gibt an, wie viele Elemente die Liste ursprünglich enthält und um wie viel grösser sie bei jeder Erhöhung wird. «Listengrössen» bestimmt die Anzahl verschiedener Listengrössen, die benutzt werden, wobei jede um «Intervall» grösser ist als die letzte. Das Textfeld «Versuche» schliesslich zeigt an, wie viele Sortierungen an Listen derselben Grösse durchgeführt werden. Damit soll der Einfluss von besonders langen oder kurzen Zeiten auf das Gesamtergebnis verringert werden. Ein Beispiel: 500 in «Intervall», 10 in «Listengrössen» und 20 in «Versuche» bedeutet, dass 20 Listen mit je 500 Elementen sortiert werden, 20 mit 1000, 20 mit 1500, usw., bis 20 Listen mit 5000 Elementen.

Nach jeder erfolgreichen Sortierung wird die verstrichene Zeit in der Textausgabe eingetragen. Nach einem Error, einer fehlerhaft sortierten Liste oder einem manuellen Abbruch wird der Zeitmessungsvorgang frühzeitig beendet und eine Warnung im Anzeigebereich eingeblendet. Wird der Prozess aber ohne Probleme beendet, erscheint stattdessen eine Grafik wie diejenige in Abbildung 2.4, die alle Zeiten für jede Listengrösse darstellt. Die besten, schlechtesten und durchschnittlichen Zeiten jeder Listengrösse werden grün, rot respektive gelb eingezeichnet. Um sehr ähnliche Zeiten zu unterscheiden, kann die Grafik durch Ziehen mit der rechten Maustaste ein- und ausgezoomt werden. Ziehen mit der linken Taste hingegen bewegt den sichtbaren Ausschnitt.

2.5 Unordnungsmodus

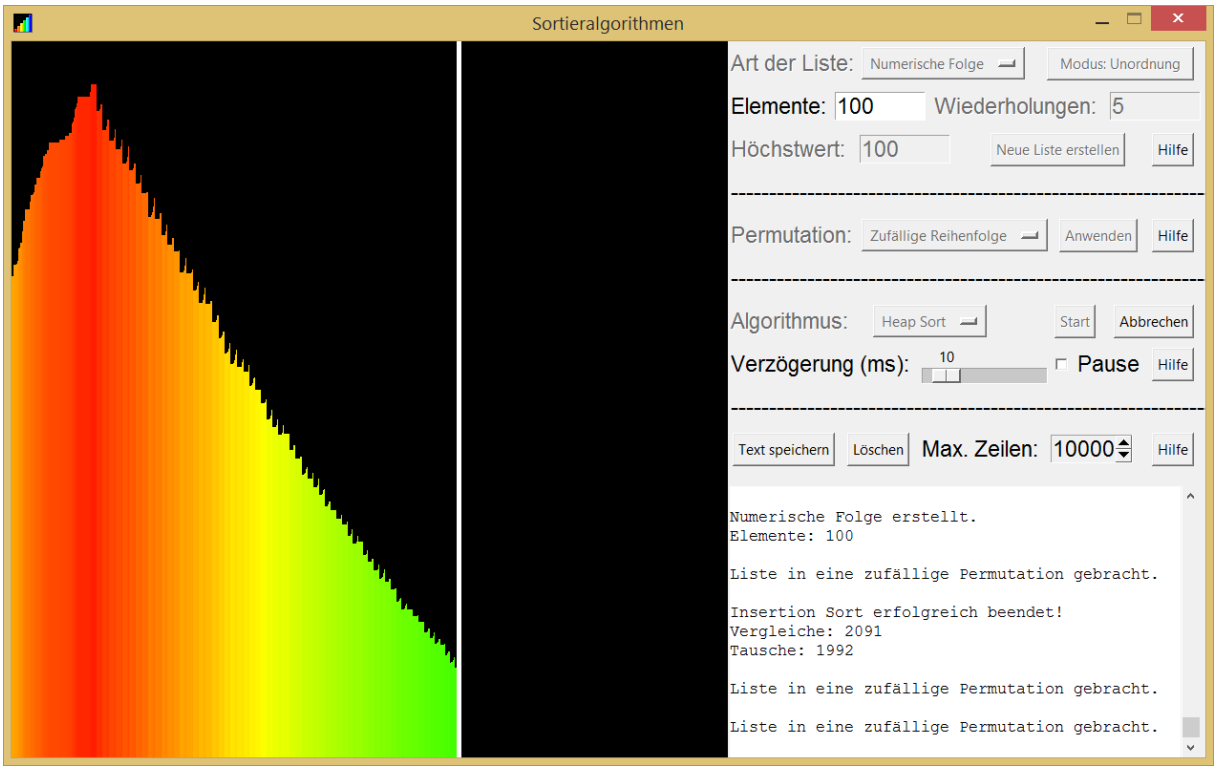


Abbildung 2.5: Unordnungsverlauf des Heap Sort

Der Unordnungsmodus funktioniert ähnlich wie der Visualisierungsmodus. Die meisten dort verfügbaren Parameter und Einstellungen können auch hier gebraucht werden. Eine Ausnahme ist die Art der Liste. Damit die relative Unordnung berechnet werden kann, ist nur die Folge natürlicher Zahlen brauchbar. Die Liste lässt sich auch nicht manuell bearbeiten, dafür können immer noch alle vier Permutationen in den Einstellungen gewählt werden.

Während des Sortiervorgangs wird nach jedem Sortierschritt die neue relative Unordnung berechnet. Danach wird abhängig von dieser ein neuer Balken im Graph hinzugefügt. Beim Wert 1 nimmt er die volle Höhe des Anzeigebereichs ein und ist ganz rot. Bei tieferen Werten hingegen ist er kürzer und grüner. Ein weisser Balken zeigt den Aktuellen Stand. Wird der rechte Rand des Anzeigebereichs erreicht, beginnt die Aufzeichnung wieder von links und überschreibt den bisherigen Unordnungsverlauf. Abbildung 2.5 zeigt ein mögliches Beispiel. Der Unordnungsmodus dient vor allem dazu, zu erkennen, ob die Unordnung für einen bestimmten Algorithmus nur kleiner wird oder ob sie auch gelegentlich wächst.

2.6 Implementierung der Sortieralgorithmen

Alle Sortieralgorithmen ausser der eigene, die später analysiert werden, sind bereits gebräuchlich. Wie sie funktionieren, wurde aus dem Buch *The Art Of Computer Programming Vol. 3: Sorting And Searching* von Donald Knuth entnommen. Nur von diesen Beschreibungen ausgehend wurden die Algorithmen, die später analysiert werden, programmiert. Allerdings wurden sie so angepasst, dass sie vom Hauptprogramm aus schrittweise ausgeführt werden können. Dabei wird am Anfang einer Sortierung eine Instanz der Klasse Sortierung

vom gewählten Sortieralgorithmus erstellt. So können Daten wie die Anzahl Vergleiche und Tausche festgehalten und auf Befehl je ein Sortierschritt durchgeführt werden. Das Ende des Sortiervorgangs gibt der Sortieralgorithmus selber an.

3 Geläufige Sortieralgorithmen

Von der enormen Vielfalt von Sortieralgorithmen werden nun einige herausgepickt und genauer betrachtet. Diese werden mithilfe des eigenen Programms geprüft. So kommen auch deren Vor- und Nachteile zum Vorschein. Bei der folgenden Analyse werden zuerst einige sehr einfache, später etwas kompliziertere Algorithmen behandelt. Jeder Sortieralgorithmus wird nach dem folgenden Prozedere analysiert: Zuerst wird die durchschnittliche Zeitkomplexität mithilfe des Zeitmessungsmodus des Programms ermittelt. Danach werden auch bestimmte Permutationen der Liste geprüft, darunter die sortierte, die verkehrte und eine fast sortierte. So sollen die minimale und maximale Zeitkomplexität gefunden werden. Die Platzkomplexität wird danach ebenfalls behandelt. Anschliessend kommt die Stabilität an die Reihe. Schliesslich wird der Verlauf der Unordnung während des Sortiervorgangs betrachtet, bei einer normalen, aber auch bei speziellen Permutationen der Liste. Dafür wird wieder das eigene Programm verwendet. Mit allen Daten lassen sich die Vor- und Nachteile jedes Sortieralgorithmus aufzeigen.

3.1 Bubble Sort

Wir fangen an mit einem einfachen Sortieralgorithmus, dem Bubble Sort. Dieser funktioniert, indem er jedes Element vom ersten bis zum zweitletzten mit dem jeweils nächsten vergleicht und die Schlüssel tauscht, wenn sie in der falschen Reihenfolge sind. Dies wird solange wiederholt, bis die Liste fertig sortiert ist. Eine verbesserte Variante des Algorithmus sorgt dafür, dass die Stelle, an der bei einem Listendurchgang zum letzten Mal getauscht wurde, der Schlusspunkt für den nächsten Durchgang ist, da sich alle Elemente von dort an bereits an ihrer endgültigen Position befinden müssen. Diese verbesserte Version wird hier verwendet.

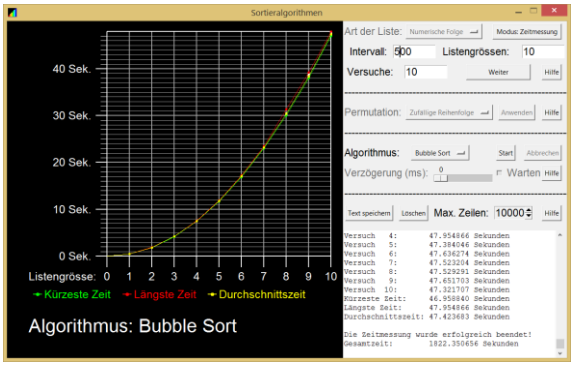


Abbildung 3.1a: Zeitmessung des Bubble Sort

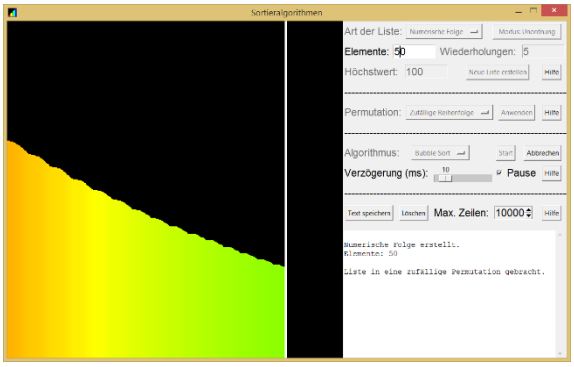


Abbildung 3.1b: Unordnungsverlauf des Bubble Sort (unvollständig)

Das Resultat der Zeitmessung des Bubble Sort ist in Abbildung 3.1a zu sehen. Die Grafik zeigt den Zeitaufwand der Sortierung im Verhältnis zur Grösse der Liste. Der Algorithmus wurde je zehn Mal an zufälligen Listen von 500, 1000, 1500, usw. bis 5000 Elementen getestet. Es lässt sich leicht erkennen, dass die Kurve Teil einer Parabel, also einer quadratischen Funktion ist. Daher ist die durchschnittliche Zeitkomplexität des Bubble Sort $O * n^2$. Das kann auch logisch erklärt werden: Es werden ungefähr n Durchgänge benötigt, um die Liste zu sortieren. Diese betreffen zunächst die ganze Liste, werden aber immer kleiner. Im Durchschnitt wird also die Hälfte der Elemente geprüft. Der Aufwand jedes Durchgangs ist deshalb proportional n. Daher steigt der Aufwand des Bubble Sort im Normalfall quadratisch mit der Grösse der Liste.

Der Sortieralgorithmus wird an drei verschiedenen Permutationen der Liste getestet, darunter die bereits sortierte, die verkehrte Reihenfolge und eine fast sortierte. Im Falle der bereits sortierten Liste geht sie der Bubble Sort nur einmal durch und hört gleich auf, da nirgends getauscht wurde. Weil nur ein vollständiger Durchgang gemacht wurde, entspricht der Zeitaufwand $O * n$. Das ist das Best-Case-Szenario des Bubble Sort. Ganz anders sieht es aus, wenn sich die Liste in der verkehrten Permutation befindet. Da muss dieser Sortieralgorithmus jedes Element einzeln an den richtigen Platz bringen. Damit hat der Bubble Sort bei dieser ungünstigsten Situation einen Zeitaufwand von $O * n^2$, was nur unwesentlich schlechter als der Durchschnitt ist. Ist die Liste aber fast sortiert, kann alles Mögliche geschehen. Befinden sich nämlich nur ganz am Anfang Elemente am falschen Ort, hat der Algorithmus wenig zusätzlichen Aufwand, diese zu ordnen. Eine solche Permutation der Liste ist nahe am Optimum. Ein einziger kleiner Schlüssel am Ende reicht aber schon aus, um den schlimmsten Fall

hervorzurufen, denn dieses Element mit dem kleinen Schlüssel wird pro Durchgang bloss um eine Stelle nach vorne verschoben. Bis es seinen Platz am Anfang der Liste eingenommen hat, können sehr viele Durchgänge geschehen sein. In diesem Fall wäre der Zeitaufwand gleich dem der ungünstigsten Situation, nämlich $O * n^2$.

Die Platzkomplexität des Bubble Sort ist $O * 1$, da nur ein paar Hilfsvariablen nötig sind. Deren Anzahl wird weder von der Grösse noch von der Permutation der Liste beeinflusst.

Der Bubble Sort sortiert stabil, Elemente mit gleichem Schlüssel wechseln ihre Reihenfolge untereinander also nicht. Das lässt sich leicht einsehen: Der Algorithmus tauscht Elemente nur dann, wenn das mit dem kleineren Index einen grösseren Schlüssel hat. Da «gleich gross» nicht unter «grösser als» verstanden wird, werden gleich grosse Schlüssel niemals vertauscht. Und da Elemente, die verglichen oder getauscht werden, immer zwei Aufeinanderfolgende sind, ist es ausgeschlossen, dass eines ein anderes mit demselben Schlüssel überspringen kann. Daher ist der Bubble Sort stabil.

Abbildung 3.1b zeigt den Verlauf der relativen Unordnung während des Sortiervorgangs, welcher in diesem Fall noch nicht beendet war. Die Höhe des Graphs zeigt den Stand der relativen Unordnung an. Beim Bubble Sort fällt auf, dass diese jeweils zuerst langsam sinkt und dann immer schneller, bis sie wieder fast konstant ist. Ein solcher Zyklus entspricht einem Durchgang der Liste. Am Anfang kommt es öfter vor, dass nicht getauscht wird. Ausserdem ist es wahrscheinlicher, dass ein Element, das weiter nach hinten gehört, nach vorne geschoben wird. Daher ändert sich die relative Unordnung zunächst langsam, aber weil sich der grössere Schlüssel immer in die richtige Richtung bewegt, wird sie nie grösser. Der Bubble Sort ist also unordnungsstet. Weil am Schluss des Durchgangs fast immer getauscht wird und sich Elemente seltener von ihrem richtigen Platz wegbewegen, wird die relative Unordnung dann rasch kleiner. Da jeder Durchgang um mindestens ein Element kürzer ist als der letzte, werden die Zyklen im Unordnungsverlauf immer kleiner, bis sie nicht mehr zu erkennen sind.

Im Falle der bereits sortierten Liste ändert sich während des einzigen Durchgangs die Unordnung nicht, da nicht getauscht wird. Ist die Liste aber in der verkehrten Reihenfolge, ist die relative Unordnung am Anfang 1. Sie ändert sich in der ersten Hälfte jedes Durchgangs nicht, da dort noch grosse Schlüssel sind, die jedoch nach vorne geschoben werden. In der jeweils anderen Hälfte werden keine Elemente mehr in die falsche Richtung bewegt, daher verkleinert sich dann die relative Unordnung. Bei einer sehr ungünstigen, fast sortierten Permutation, wie sie vorher erwähnt wurde, ist die Unordnung von Beginn an klein. Doch weil fast so viel Zeit benötigt werden kann wie bei der verkehrten Reihenfolge, wird sie sehr langsam kleiner, bis sie 0 erreicht.

Anhand der erhaltenen Daten sind nun die Vor- und Nachteile des Bubble Sort zu erkennen. Vorteilhaft ist aber fast nur die Einfachheit des Algorithmus. Mit einem durchschnittlichen Zeitaufwand von $O * n^2$ eignet er sich nicht für die Sortierung grösserer Listen. Nur bei der bereits sortierten Permutation wird wenig Zeit benötigt, doch schon der Tausch der ersten und letzten Elemente bringt den Aufwand zum Maximum. Daher ist der Bubble Sort für einen grossen Sortierauftrag nicht zu empfehlen.

3.2 Insertion Sort und Gnome Sort

Der Insertion Sort beruht auf dem Prinzip des Einschubs. Er schiebt ein Element solange Richtung Listenanfang, bis der Schlüssel davor kleiner ist als der Eigene oder es ganz vorne ist. So geht der Algorithmus mit allen Elementen vom zweiten bis zum letzten vor. So entsteht eine sortierte Teilliste, die sich bei jedem Durchgang um ein Element vergrössert, bis alles fertig sortiert ist. Es gibt eine vereinfachte Variante des Algorithmus mit dem Namen Gnome Sort. Immer wenn ein Tausch erfolgt, bewegt sich der Zeiger eine Stelle nach vorne, sofern er nicht schon ganz am Anfang ist, sonst eine nach hinten. Wenn der Zeiger den Schluss der Liste erreicht, ist der Sortiervorgang beendet.

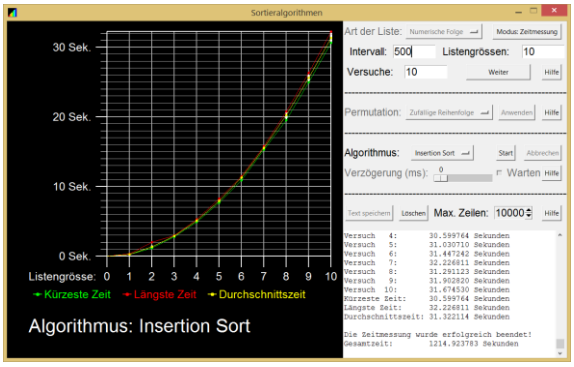


Abbildung 3.2a: Zeitmessung des Insertion Sort

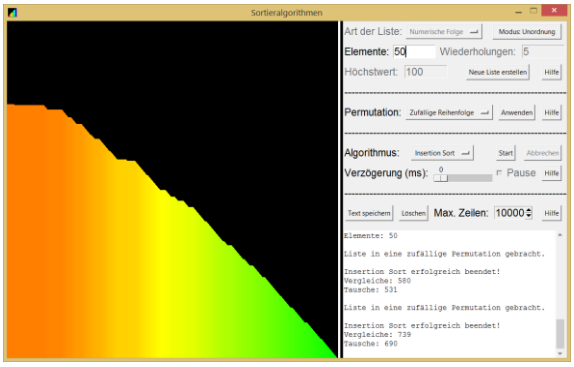


Abbildung 3.2b: Unordnungsverlauf des Insertion Sort

Abbildung 3.2a stellt das Resultat der Zeitmessung des Insertion Sort dar. Benutzt wurden sonst dieselben Parameter wie für den Bubble Sort, nämlich je zehn Versuche der Listengrössen 500, 1000, 1500, usw. bis 5000. Auch bei diesem Algorithmus ist der Graph eine quadratische Funktion, der durchschnittliche Zeitaufwand also $O * n^2$. Das kommt daher zustande, dass jedes Element einzeln in die sortierte Teilliste eingeschoben wird. Jeder Einschub wiederum benötigt im Durchschnitt immer mehr Schritte, da dieser sortierte Teil ständig wächst, bis er alle Elemente beinhaltet. Im Falle des Gnome Sort wird der Aufwand ungefähr verdoppelt, da der Zeiger nach jedem Einschub nur schrittweise nach hinten wandert, bis er das nächste Element erreicht, das einzuschieben ist. Im Gegensatz dazu wird der Zeiger beim Insertion Sort sofort dorthin bewegt. Auffällig ist für den normalen Insertion Sort auch, dass nach jedem Vergleich ausser dem Letzten eines Einschubs auch ein Tausch folgt.

Wie auch beim Bubble Sort ist für den Insertion Sort die sortierte Permutation die Günstigste. Da keines der Elemente nach vorne geschoben werden muss, ist für jedes davon lediglich ein einziger Vergleich nötig. Somit ist der Zeitaufwand in diesem günstigen Fall $O * n$. Dasselbe gilt für den Gnome Sort, da sich der Zeiger ohne Unterbruch bis zum Ende der Liste bewegt. Befinden sich die Elemente aber in der verkehrten Reihenfolge, muss jedes bis an den Anfang der sortieren Teilliste geschoben werden, da jeder neue Schlüssel immer der bisher Kleinste ist. Dieser schlimmste Fall bringt einen Zeitaufwand von $O * n^2$ mit sich. Der erste auffällige Unterschied zum Bubble Sort in Bezug auf den Zeitaufwand erscheint bei einer fast sortierten Liste. Der Bubble Sort kann schon auf kleine Änderungen sehr stark reagieren. Für den Insertion Sort hingegen bleibt der Aufwand bei $O * n$. Kleine Schlüssel, die am falschen Ort sind, werden mit nur einem zusätzlichen Einschub eingeordnet. Grössere

Werte hingegen werden jedes Mal um nur eine Stelle nach hinten geschoben, da jeweils kleinere Schlüssel davorgeschoben werden. Allerdings benötigen solche Einschübe nur einzelne Tausche, daher entsteht nur wenig zusätzlicher Aufwand. Fast sortierte Listen sind für den Insertion Sort also fast so günstig wie vollständig Geordnete.

Wie der Bubble Sort braucht dieser Sortieralgorithmus nur Platz für ein paar wenige Hilfsvariablen, unabhängig von der Gestalt der Liste. Seine Platzkomplexität ist also $O * 1$.

Auch der Insertion Sort und der Gnome Sort sind stabil. Ähnlich wie beim Bubble Sort ist es unmöglich, dass ein Element ein anderes mit demselben Schlüssel überspringt, weil nur aufeinanderfolgende Elemente verglichen und getauscht werden. Ausserdem hört ein Einschub dann auf, wenn der Schlüssel vor dem, der eingeschoben wird, kleiner oder gleich gross ist. Deshalb bleiben gleich grosse Werte in der ursprünglichen Reihenfolge zueinander.

In Abbildung 3.2b ist der Unordnungsverlauf des Insertion Sort zu sehen. Er hat eine ähnliche zyklische Struktur wie der des Bubble Sort, allerdings wird sie für diesen Algorithmus eine Zeit lang gleichmässig kleiner und bleibt bis zum Ende des Durchgangs gleich. Weil sich die sortierte Teilliste am Anfang der gesamten Liste befindet, sind die Elemente der ersteren entweder an der richtigen Stelle oder zu weit vorne. Daher bewegt sich das Element, das eingeschoben wird, zunächst in die richtige Richtung, passiert seinen richtigen Platz aber irgendwann und entfernt sich davon wieder. Elemente, die beim Einschub nach hinten verschoben werden, nähern sich jedes Mal ihrer endgültigen Position. Deshalb verkleinert sich am Anfang eines Einschubs die relative Unordnung und bleibt ab einem bestimmten Punkt bis zum Schluss gleich. Gegen Schluss des Sortiervorgangs nimmt der sortierte Teil fast die ganze Liste ein und Elemente können nicht mehr weit an ihrer richtigen Position vorbeigeschoben werden, daher wird die relative Unordnung immer stetiger kleiner. Der Gnome Sort hat einen fast identischen Unordnungsverlauf. Anders sind nur die längeren Phasen der gleichbleibenden Unordnung, die entstehen, wenn der Zeiger zum nächsten einzuschiebenden Element wandert. Weil die relative Unordnung nie grösser wird, besteht Unordnungsstetigkeit.

Da der Insertion Sort unordnungsstet ist, passiert bei einer bereits sortierten Liste nichts. Ist sie aber in der verkehrten Reihenfolge, ist die relative Unordnung zu Beginn 1. Und da bleibt sie auch sehr lange, denn die Elemente in der ersten Hälfte der Liste bewegen sich immer weg von ihrer richtigen Position, da sie weiter hinten hingehören. Nachher wird die Unordnung in jedem weiteren Einschub kleiner, zuerst nur ein wenig, dann immer mehr. Wenn zuletzt die kleinsten Schlüssel nach vorne geschoben werden, wird sie fast gleichmässig kleiner, wie auch bei der zufälligen Permutation. Im Falle der fast sortierten Liste ist die relative Unordnung schon am Anfang klein, wird aber mit nur wenigen Phasen der Verkleinerung zu 0.

Ein grosser Vorteil des Insertion Sort und des Gnome Sort gegenüber dem Bubble Sort ist, dass sie fast sortierte Listen schnell und effizient berichtigen können. Ausserdem sind beide Varianten sehr einfach zu implementieren, besonders der Gnome Sort, welcher innerhalb weniger Programmzeilen geschrieben werden kann. Allerdings besteht derselbe Nachteil, den auch der Bubble Sort hat, nämlich dass wegen der Zeitkomplexität von $O * n^2$ für das Sortieren grosser Listen sehr viel Zeit nötig sein kann.

3.3 Selection Sort

Wie es der englische Name ahnen lässt, wählt der Selection Sort jeweils ein bestimmtes Element aus. Er speichert zunächst den Index des ersten Elements in einer Hilfsvariablen und geht dann die ganze Liste durch. Trifft der Algorithmus auf ein Element mit einem kleineren Schlüssel als dasjenige, auf welches die Hilfsvariable zeigt, wird der aktuelle Index des Zeigers stattdessen gespeichert. Nach dem Durchgang wird das erste Element mit demjenigen getauscht, das von der Hilfsvariable angegeben wird. Ein neuer Durchgang vom zweiten Element aus findet den nächstgrösseren Schlüssel, ein weiterer den Dritten, usw. Dabei entsteht am Anfang der Liste eine wachsende sortierte Teilliste, die am Schluss alle Elemente beinhaltet. Varianten des Algorithmus suchen statt des Kleinsten den jeweils grössten Schlüssel oder sogar beide.

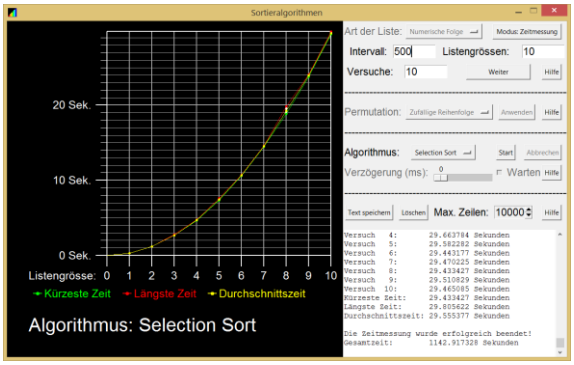


Abbildung 3.3a: Zeitmessung des Selection Sort

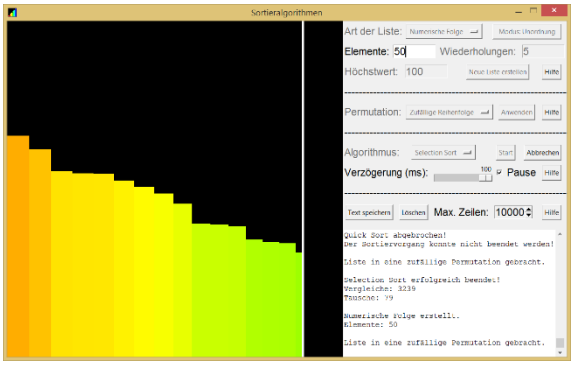


Abbildung 3.3b: Unordnungsverlauf des Selection Sort (unvollständig)

Wie schon die bisherigen einfachen Algorithmen produziert der Selection Sort die Kurve einer quadratischen Funktion, in Abbildung 3.3a zu sehen. Die durchschnittliche Zeitkomplexität ist also erneut $O * n^2$. Die Anzahl der Durchgänge ist wie beim Bubble Sort abhängig von der Grösse der Liste, weil für jedes Element einer stattfindet. In den ersten Durchgängen werden noch fast alle Schlüssel geprüft, später aber bleiben immer weniger davon übrig und die letzten Durchgänge betreffen nur noch ein paar Elemente. Dieses Verhalten wurde auch schon im Bubble Sort festgestellt. Allerdings kann der Selection Sort keine Durchgänge überspringen. Der Aufwand steigt daher auch bei diesem Sortieralgorithmus quadratisch zur Grösse der Liste. Ein auffälliger Unterschied zum tauschfreudigen Insertion Sort ist, dass hier sehr wenige Tausche stattfinden, nämlich nur einer pro Element. Der Insertion Sort hingegen tauscht nur am Ende eines Einschubs nicht. Gegeben, dass beide Algorithmen gleich grosse Listen sortieren müssen, führt der Selection Sort genauso viele Tausche aus wie der Insertion Sort Vergleiche ohne Tausche.

Eine spezielle Eigenschaft des Selection Sort ist, dass der Zeitaufwand überhaupt nicht von der Permutation der Liste beeinflusst wird. Daher sind sowohl die maximale als auch die minimale Zeitkomplexität beide $O * n^2$. Selbst wenn die Liste schon sortiert ist, geht der Algorithmus immer noch für jede Position alle Elemente durch, um das jeweils Nächstgrössere zu finden. Dasselbe trifft für die verkehrte Permutation zu.

Wie die anderen bisher behandelten Algorithmen benötigt der Selection Sort nebst der Liste nur Platz für die üblichen Hilfsvariablen. Deshalb ist seine Platzkomplexität $O * 1$.

Der Selection Sort sortiert im Gegensatz zum Bubble Sort und dem Insertion Sort instabil. Es kann nämlich vorkommen, dass unter Elementen mit demselben Schlüssel eines davon den Anfang der unsortierten Teilliste bildet. Wenn der kleinste Schlüssel hinter einem anderen derartigen Element liegt, wird das vorher Erwähnte am Schluss des Durchgangs dorthin geschoben. So kann es geschehen, dass die relative Reihenfolge von identischen Schlüsseln beim Tausch geändert wird. Deshalb ist dieser Sortieralgorithmus instabil.

Auffällig für den Selection Sort ist sein Unordnungsverlauf, welcher unvollendet in Abbildung 3.3b gezeigt wird. Da nur wenige Tausche stattfinden, bleibt die relative Unordnung während jedes Durchgangs konstant und sinkt danach ruckartig im Gegensatz zu der stetigen Verkleinerung bei den anderen einfachen Algorithmen. Das kommt daher, dass der kleinste Schlüssel der unsortierten Teilliste sofort zu seiner endgültigen Position gelangt. Das Element, das vorher dort war, bewegt sich zunächst immer auf seinen richtigen Platz zu, kann ihn aber auch überspringen. Deshalb sind manche Stufen in der relativen Unordnung grösser als andere. Lag der kleinste Schlüssel schon an der richtigen Stelle, bleibt diese Stufe ganz aus. Weil das eine Element direkt zu seinem endgültigen Platz gelangt, kann die relative Unordnung niemals steigen, damit ist dieser Algorithmus unordnungsstet.

Der Selection Sort macht bei einer schon sortierten Liste keine Änderungen an der Unordnung, da jedes Element bereits an der richtigen Stelle ist und nicht herumgeschoben wird. Ist die Liste aber in der verkehrten Permutation, beginnt die relative Unordnung wie gewohnt bei 1. Weil der kleinste Schlüssel ganz am Schluss liegt, werden nach dem ersten Durchgang die äussersten Elemente getauscht, was in einer grossen Stufe der Unordnung resultiert. Da die getauschten Elemente jedes Mal näher zueinander sind und die Durchgänge kürzer werden, ist jede solche Stufe etwas weniger breit und weniger tief als die Letzte. Wenn die relative Unordnung 0 erreicht, ist der Sortiervorgang aber noch nicht beendet, denn erst die Hälfte aller Durchgänge hat stattgefunden. Tatsächlich wurden die grossen Schlüssel beim Tausch mit den jeweils Kleinsten bereits an ihre endgültige Position gebracht, da sich die Liste am Anfang in der verkehrten Permutation befand. Weil der Selection Sort noch die andere Hälfte der Elemente durchgehen muss und wegen der Zeitkomplexität von $O * n^2$ bleibt nur noch ein Viertel des Gesamtaufwandes übrig. Daher ist die Liste schon nach drei Vierteln der Zeit vollständig sortiert, auch wenn der Algorithmus noch weiterarbeitet. Bei einer fast sortierten Liste fängt die relative Unordnung tief an. Ist die Permutation durch wenige Tausche von Elementen der sortierten Liste entstanden, gelangt diese in wenigen grösseren Sprüngen zu 0. Wurde aber ein Element aus der Liste entfernt und woanders eingefügt, geschieht die Verkleinerung der relativen Unordnung in vielen kleinen Schritten.

Weil der Selection Sort unabhängig von der Permutation der Liste eine Zeitkomplexität von $O * n^2$ aufweist, ist er sehr ineffizient. Einzig wenn ein Rechner, der den Algorithmus ausführt, einen sehr schnellen Prozessor hat und nur begrenzten und langsamen Speicher, ist der Selection Sort dank der wenigen Tausche einigermassen brauchbar, doch ein solches Szenario ist ziemlich unwahrscheinlich. Ausserdem ist das ein weiterer sehr einfacher Algorithmus. Die folgenden Sortieralgorithmen sind zwar komplexer, aber wesentlich effizienter.

3.4 Quick Sort

Der Name ist hier Programm. Quick Sort ist einer der schnellsten bekannten Sortieralgorithmen, entwickelt 1962 vom britischen Informatiker C. A. R. Hoare. Der Algorithmus sucht zuerst ein bestimmtes Element aus, genannt Pivotelement. Dann teilt er die Liste in zwei Teile, sodass der Erste nur Schlüssel enthält, die kleiner oder gleich dem des Pivotelements sind, der Zweite nur Grössere oder Gleiche. Danach wiederholt der Quick Sort diesen Vorgang rekursiv an den beiden Teillisten. Bleiben weniger als zwei Elemente übrig zum Aufteilen, bricht die Rekursion ab. Es gibt viele Implementierungen dieses Algorithmus, doch diejenige von R. Sedgewick wird hier benutzt. Das erste Element der Liste wird jeweils zum Pivotelement.

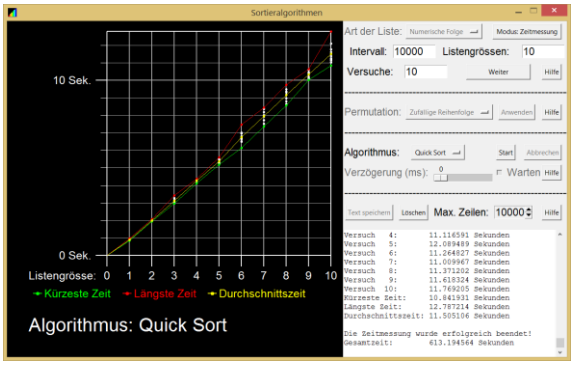


Abbildung 3.4a: Zeitmessung des Quick Sort

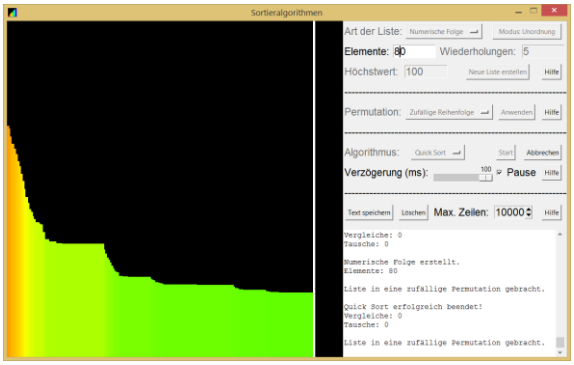


Abbildung 3.4b: Unordnungsverlauf des Quick Sort (unvollständig)

Abbildung 3.4a zeigt das Resultat der Zeitmessung des Quick Sort. Für diesen Algorithmus und alle Folgenden ist die Grundgrösse der Liste nicht 500 Elemente, sondern 10000. Die durchschnittliche Zeitkomplexität des Quick Sort ist $O * n \log n$, was wesentlich effizienter als die bisherigen Algorithmen ist. Die erste Teilung allein benötigt einen Zeitaufwand von $O * n$, da alle Elemente eingeteilt werden. Alle Teilungen der nächsten Rekursionstiefe brauchen zusammen nochmals so viel Aufwand. Nach dem Muster geht es weiter. Die Rekursion bricht meist in einer Tiefe von ungefähr $\log n$ ab. Da die Liste nicht unbedingt in zwei gleich grosse Teile aufgeteilt wird, ist etwas zusätzlicher Aufwand zu erwarten, aber in der Regel nicht genug, um die Zeitkomplexität erheblich zu vergrössern.

Eine günstige Permutation der Liste für den Quick Sort ist eine, bei der das Pivotelement möglichst häufig einen mittelgrossen Schlüssel hat. So entstehen immer ähnlich grosse Teillisten und die ideale Zeitkomplexität von $O * n \log n$ tritt ein. Weil die Methode von R. Sedgewick jeweils das erste Element zum Pivotelement macht, ist die sortierte Permutation der Liste jedoch sehr ungünstig für den Insertion Sort. Das Pivotelement bildet nach vollendeter Teilung die Begrenzung der Teillisten, die rekursiv sortiert werden und liegt dabei schon am endgültigen Platz. Weil es sich aber im Fall der sortierten Liste ganz vorne befindet, enthält die erste Teilliste 0 Elemente und die zweite eines weniger als vorher. Deshalb wird eine Rekursionstiefe proportional zu n erreicht. Dadurch entsteht der schlimmste mögliche Zeitaufwand von $O * n^2$. Dieser wird aber auch bei einer verkehrten Permutation erreicht, da auch dann das Pivotelement nach der Teilung immer an einer der äussersten Stellen der Liste lan-

det. Nach der ersten Teilung wird es ganz nach hinten bewegt und mit dem kleinsten Schlüssel getauscht. Da dieser nun an der ersten Stelle ist, wird er zum Pivotelement, aber da er schon an seiner endgültigen Position ist, beginnt die nächste Rekursionstiefe beim nächsten Element. Beide Teillisten enthalten so abwechselnd 0 Elemente. Eine fast sortierte Liste benötigt fast ebenso viel Aufwand wie die beiden vorherigen Fälle, weil das Pivotelement meistens sehr klein ist und daher die Teillisten völlig unterschiedliche Grössen haben. Es gibt allerdings adaptive Implementierungen des Quick Sort, die bessere Pivotelemente auswählen. Eine davon prüft drei Elemente und wählt das mit dem mittleren Schlüssel als Pivotelement. So wird die Liste meistens in ähnlich grosse Teillisten aufgeteilt und der Sortiervorgang benötigt weniger Zeit.

Eine Besonderheit des Quick Sort ist, dass die Platzkomplexität abhängig von der Permutation der Liste ist. Jede Stufe der Rekursion hat ihren eigenen konstanten Bestand an Hilfsvariablen, der einen Platz von $O * 1$ braucht. Im besten Fall erreicht die Rekursionstiefe ein Maximum von $\log_2 n$ und damit insgesamt eine Platzkomplexität von $O * \log_2 n$. Diese wird bei den vorher erwähnten Permutationen auf $O * n$ erhöht.

Kommen mehrere gleiche Schlüssel vor, kann es sehr wohl vorkommen, dass diese ihre Reihenfolge ändern. Die Elemente, die getauscht werden, können grössere Distanzen zurücklegen. Dabei kann ein Element ein anderes mit demselben Schlüssel überspringen. Deshalb ist der Quick Sort instabil.

In Abbildung 3.4b ist der Grossteil vom Unordnungsverlauf des Quick Sort zu sehen. Es ist leicht zu erkennen, dass einige grössere Verkleinerungen der relativen Unordnung geschehen, gefolgt von immer kleineren. Die allererste Teilung, nämlich die der gesamten Liste, hat meistens die stärkste Einwirkung. Am grössten kann diese sein, wenn die beiden Teillisten ähnlich gross sind, weil mehr Elemente überhaupt getauscht werden müssen. Wenn mit der Vertiefung der Rekursion die Teillisten kleiner werden, ändert sich die relative Unordnung immer weniger. Auch der Quick Sort ist unordnungsstet, weil durch einen Tausch das grössere Element immer nach hinten geschoben wird und das kleinere nach vorne.

Bei einer sortierten Liste passiert wie gewohnt nichts mit der relativen Unordnung, da keine Tausche geschehen. Beim Sortieren einer verkehrten Liste aber erzeugt der Quick Sort einen ähnlichen Unordnungsverlauf wie der Selection Sort, weil jeweils die äussersten Elemente, dann die zweitäussersten, usw. getauscht werden. Liegt aber eine fast sortierte Permutation vor, ist die relative Unordnung wie zu erwarten von Anfang an tief. In wenigen kurzen Schüben springt diese auf 0, wenn während der Teilung die wenigen fehlplatzierten Elemente an ihren richtigen Platz gebracht werden.

Der Quick Sort wird zwar allgemein als der schnellste aller bekannten Sortieralgorithmen angesehen, doch die Implementierung im eigenen Programm ist langsamer als der Heap Sort. Ausserdem hat der Quick Sort eine maximale Zeitkomplexität von $O * n^2$, die bei ungünstigen Permutationen eintrifft. In solchen ungünstigen Fällen kann die Platzkomplexität bis zu $O * n$ steigen, was ebenfalls nachteilig ist. Daher begrenzt sich der Nutzungsbereich auf zufällige Listen. Ein weiteres Problem des Quick Sort ist, dass er relativ kompliziert zu implementieren ist.

3.5 Merge Sort

Der Merge Sort sortiert, indem er solange je zwei aufeinanderfolgende sortierte Teillisten zu einer zusammenführt, bis die gesamte Liste in der richtigen Permutation ist. Es gibt extrem viele Varianten dieses Algorithmus, weil mit oder ohne Rekursion gearbeitet werden kann und manche Ausführungen bereits gegebene sortierte Teillisten ausnutzen, wohingegen dies andere nicht tun. Benutzt wird hier aber eine häufige Variante, welche mit Rekursion arbeitet, aber nicht adaptiv funktioniert. Die Liste wird zuerst rekursiv in je zwei gleich grosse Teile geteilt. Dieser Prozess wiederholt sich solange für beide Hälften, bis die Teillisten lediglich einzelne Elemente sind. An diesem Punkt erreicht die Rekursion ihre grösste Tiefe. Nachher werden die sortierten Teillisten wieder zusammengeführt, zuerst die Kleinsten, dann immer Grössere. Dabei werden zwei Gruppen sortierter Elemente zu einer verarbeitet, indem jeweils das kleinere der Anfangselemente weggenommen und zu einer versteckten Liste hinzugefügt wird. Nach diesem Prozess wird der Inhalt der Schattenliste, welcher sortiert ist, in die Hauptliste kopiert.

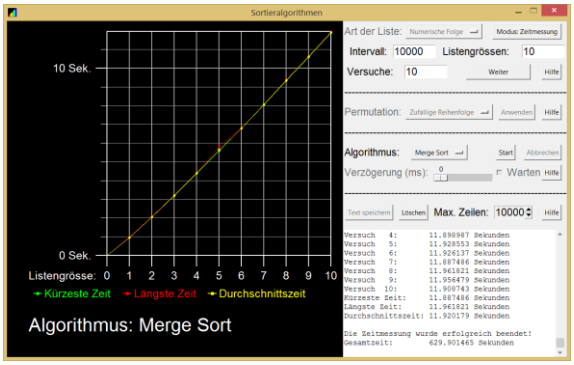


Abbildung 3.5a: Zeitmessung des Merge Sort

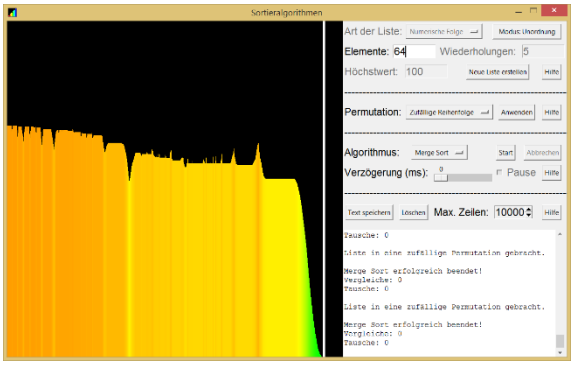


Abbildung 3.5b: Unordnungsverlauf des Merge Sort

In Abbildung 3.5a ist das Resultat der Zeitmessung des Merge Sort zu sehen. Ähnlich wie beim Quick Sort ist die Zeitkomplexität $O * n \log_2 n$. Die Rekursion erreicht im Durchschnitt eine Tiefe von $\log_2 n$. In jeder Rekursionsstufe ausser der letzten enthalten die zusammenzuführenden Teillisten zusammen alle Elemente der Gesamtliste genau einmal. So entsteht ein Aufwand von $O * n \log_2 n$. Eine Besonderheit des Merge Sort ist, dass keine Tausche innerhalb der ursprünglichen Liste stattfinden. Die Elemente werden nämlich von dem Inhalt der Schattenliste überschrieben, um eine sortierte Teilliste zu erzeugen.

Weil diese Variante des Merge Sort die Liste unabhängig von der Permutation teilt und zusammenführt, gibt es keinen besonders günstigen oder ungünstigen Fall, im Gegenteil, der Aufwand ist für gleich grosse Listen praktisch konstant. Anders wäre es bei einer adaptiven Implementierung für diesen Sortieralgorithmus. Ist die Liste bereits sortiert, würde sie nicht mehr aufgeteilt werden, weil sie als Ganzes schon eine sortierte Teilliste bildet. In diesem Fall wäre der Zeitaufwand $O * n$. Bei einer verkehrten Liste hätte diese Variante des Algorithmus kaum mehr Aufwand als die Nichtadaptive. Und eine fast sortierte Liste schliesslich würde wenig zusätzlichen Aufwand verursachen verglichen mit einer Sortierten, weil sie in ein paar wenige sortierte Teillisten aufgeteilt werden kann.

Eine Gemeinsamkeit der geläufigen Versionen des Merge Sort ist die Platzkomplexität. Weil eine versteckte Liste nebst der Normalen benötigt wird, und diese an einem Zeitpunkt alle Elemente beinhaltet, ist die Platzkomplexität $O * n$.

Im Gegensatz zum Quick Sort sortiert der Merge Sort stabil. Wenn beim Zusammenführen zweier Teillisten identische Schlüssel vorkommen, werden zuerst diejenigen des vorderen Teils in die Schattenliste übernommen, bevor dies mit den anderen Exemplaren geschieht. Beim Kopieren wird deren Reihenfolge nicht geändert.

Der Merge Sort hat einen ungewöhnlichen Unordnungsverlauf, was in Abbildung 3.5b zu sehen ist. Weil nur die reguläre Liste für die Berechnung der relativen Unordnung berücksichtigt wird, ändert sie sich nur während des Kopierens von der Schattenliste. Meistens verkleinert sich die relative Unordnung dabei vorübergehend In der ersten Hälfte des Sortiervorgangs, in der zweiten erhöht sie sich. Beim Kopieren werden Schlüssel zunächst mit grosser Wahrscheinlichkeit durch grössere überschrieben, später aber immer häufiger durch kleinere. Geschieht dies am Anfang der gesamten Liste, sinkt die relative Unordnung meistens zunächst, weil die kleineren Elemente dorthin gehören. Wenn die grössten Elemente einer Teilliste kopiert werden, sind sie eher fehl am Platz, daher steigt die Unordnung wieder. Genau umgekehrt ist es am Schluss der Gesamtliste. Im Gegensatz zu den bisherigen Sortieralgorithmen besteht beim Merge Sort keine Unordnungsstetigkeit, weil die relative Unordnung steigen kann.

Obwohl dieser Algorithmus nicht unordnungsstet ist, geschieht nichts mit der relativen Unordnung, wenn die Liste vorher schon sortiert war. Während des Kopiervorgangs werden dabei die Elemente durch sich selbst überschrieben, weil deren Reihenfolge bereits stimmte. Befand sich die Liste aber in der verkehrten Permutation, ist die relative Unordnung zunächst 1. Bis vor der letzten Zusammenführung von Teillisten ist diese nach jedem Kopiervorgang wieder 1, weil für jedes Element, das sich seinem richtigen Platz nähert, ein anderes sich ebenso weit davon entfernt. Das liegt daran, dass bis dahin jedes Element in der falschen Hälfte der Gesamtliste ist. Erst die letzte Zusammenführung bringt alles an seine endgültige Stelle. War die Liste in einer fast sortierten Permutation, sieht der Unordnungsverlauf ähnlich aus wie bei einer zufälligen, bloss ausgehend von einer geringeren Unordnung und häufiger Auslassung von deren Änderungen, weil keine Elemente ihren Platz gewechselt haben. Eine besondere Eigenschaft des Merge Sort ist, dass während des Kopierens die relative Unordnung den Wert 1 überschreiten kann, da ein Element ein anderes überschreiben kann, bevor es seinerseits am alten Platz ersetzt wird. Allerdings ist dies nur temporär, da später wieder alle vorherigen Elemente da sind und die relative Unordnung wieder 1 ist.

Was den Merge Sort auszeichnet, ist, dass er immer eine Zeitkomplexität von $O * n \log n$ hat und stabil ist. Ein grosser Nachteil ist aber die hohe Platzkomplexität von $O * n$. Bei sehr grossen Listen reicht der Arbeitsspeicher eventuell nicht aus. Ausserdem ist der Merge Sort nicht besonders leicht zu implementieren. Insgesamt ist dieser Sortieralgorithmus am besten für mittelgrosse Sortierungen geeignet, die Stabilität voraussetzen. Fehlt der nötige Speicherplatz, ist der Insertion Sort eine gute Alternative für kleinere Listen.

3.6 Heap Sort

Der Heap Sort ist ein kurioser Sortieralgorithmus, welcher die Elemente der Liste in eine gewisse Datenstruktur einsetzt, nämlich einen Heap (was wörtlich Haufen bedeutet). Zuoberst in dieser Struktur liegt der grösste Schlüssel. Jedem Element im Heap sind zwei weitere untergeordnet, welche kleinere Schlüssel haben. Ausgenommen sind die untersten Elemente. Der Heap Sort arbeitet in zwei Phasen: Während der ersten Phase des Heap Sort wird die Liste so anpasst, dass sie die Eigenschaften eines Heaps erfüllt. Dabei stellt das erste Element die oberste Schicht dar, Elemente 2 und 3 die Zweite, Elemente 4 bis 7 die Dritte, usw. In der zweiten Phase wird der erste Schlüssel, der ja der Grösste ist, mit dem Letzten getauscht, und wird aus dem Heap entfernt. Die übrige Liste wird anschliessend erneut zu einem Heap gemacht. Dieser Prozess wird solange wiederholt, bis der Heap leer und damit der Sortiervorgang beendet ist.

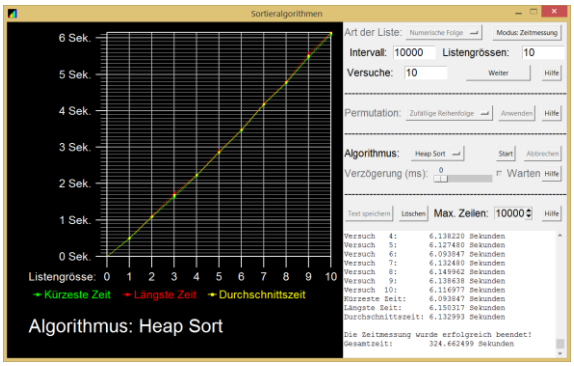


Abbildung 3.6a: Zeitmessung des Heap Sort

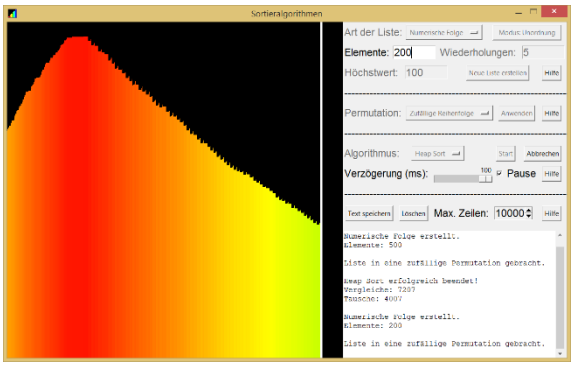


Abbildung 3.6b: Unordnungsverlauf des Heap Sort (unvollständig)

Wie der Quick Sort und der Merge Sort hat der Heap Sort eine durchschnittliche Zeitkomplexität von $O * n \log_2 n$, was in Abbildung 3.6a abgelesen werden kann. Um das zu erklären, muss zuerst das Verhältnis zwischen der Liste und dem Heap genauer betrachtet werden. Das erste Element der Liste bildet die erste Schicht des Heaps. Wenn i der Index eines beliebigen Elements ist, haben die Untergeordneten die Indizes $2 * i$ und $2 * i + 1$ und das Übergeordnete $\lfloor \frac{i}{2} \rfloor$. Die erste Phase der Sortierung macht aus der Liste einen Heap. Dies funktioniert, indem jeder Schlüssel, der Untergeordnete besitzt, solange mit dem jeweils grösseren davon getauscht wird, bis dieser kleiner als der aktuelle ist oder die unterste Schicht des Heaps erreicht wurde. Dies wird zuerst mit dem letzten solchen Schlüssel gemacht, dann mit dem zweitletzten, usw. bis zum allerersten. Weil jedes Element höchstens $\log_2 n$ Schichten hinuntergeschoben werden kann und es n Elemente gibt, hat diese Phase durchschnittlich einen Zeitaufwand von $O * n \log n$. Dasselbe gilt für die zweite Phase. In dieser wird immer das erste Element hinter den Heap geschoben und aus ihm entfernt. Dann wird die in der ersten Phase benutzte Prozedur bloss am neuen ersten Element der Liste ausgeführt. Dies wiederholt sich, bis alle Elemente aus dem Heap entfernt wurden und sich am richtigen Platz befinden. Weil beide Phasen eine Zeitkomplexität von $O * n \log n$ haben, ist das auch diejenige des ganzen Algorithmus.

Im besten Fall erfüllt die Liste bereits die Eigenschaft eines Heaps. Dies trifft z.B. für die verkehrte Permutation zu. So wird der Aufwand der ersten Phase auf $O * n$ gesenkt, weil keines

der Elemente nach unten bewegt werden kann. Der einzige Aufwand stammt vom Vergleichen. Die Komplexität der zweiten Phase kann jedoch nicht unter $O * n \log n$ gebracht werden. Das jeweils erste Element muss immer noch bis zu $\log n$ Schichten hinuntergeschoben werden. Die Zeitkomplexität bleibt nach wie vor $O * n \log n$. Es gibt für den Heap Sort aber auch keine besonders ungünstigen Permutationen wegen der Tiefe des Heaps, die ja logarithmisch mit der Grösse der Liste steigt. Daher hat der Algorithmus einen ähnlichen Aufwand für die sortierte oder eine fast sortiere Permutation.

Weil diese Ausführung des Heap Sort gleich die Liste selber als Heap benutzt, statt die Elemente in einer externen Struktur einzuordnen, benötigen nur die wenigen Hilfsvariablen zusätzlichen Speicherplatz. Deshalb ist die Platzkomplexität $O * 1$.

Der Heap Sort ist nicht stabil. Es ist nämlich möglich, dass identische Schlüssel in völlig verschiedenen Stellen des Heaps sind. Manchmal wird einer der hinteren zuerst nach oben bewegt, wenn dieser einem sehr grossen Schlüssel untergeordnet ist. Somit kann sich die relative Reihenfolge von Elementen mit demselben Schlüssel ändern.

Abbildung 3.6b stellt den unvollendeten Unordnungsverlauf des Heap Sort dar. Es fällt sofort auf, dass die relative Unordnung zunächst stark zunimmt. Dies geschieht während der ersten Phase der Sortierung, weil grosse Schlüssel eher zum Anfang der Liste bewegt werden und kleine eher nach hinten. Tatsächlich liegt nach einem Tausch der grössere Schlüssel immer weiter vorne, deshalb steigt die relative Unordnung jedes Mal. In der zweiten Phase aber sinkt diese, wenn das vorderste Element hinter den Heap gebracht wird, da es dort gleich am richtigen Platz ist. Wenn die übrige Liste wieder zu einem gültigen Heap gemacht wird, wächst die Unordnung wieder wie in der ersten Phase. Dieser Zyklus wiederholt sich für jedes Element, bis der Sortiervorgang fertig ist. Wegen den häufigen Zunahmen der Unordnung ist der Heap Sort nicht unordnungsstet.

Ein auffällig anderer Unordnungsverlauf erscheint dann, wenn die Liste in einer günstigen Permutation ist, die bereits die Eigenschaften eines Heaps hat. In diesem Fall bleibt die anfängliche Steigung der Unordnung während der ersten Phase aus, weil nur Vergleiche und keine Tausche gemacht werden. Ein Sonderfall ist die verkehrte Liste. Da beginnt die relative Unordnung bei 1 und bleibt bis zum Anfang der zweiten Phase bei diesem Wert. In dieser sieht der übrige Unordnungsverlauf für jede Permutation ähnlich aus. Eine Besonderheit des Heap Sort ist, dass er als einziger unter den hier behandelten Algorithmen eine schon sortierte Liste während des Sortierens verändert. Die relative Unordnung ist zunächst 0, doch diese wird von der ersten Phase genauso erhöht wie wenn die Liste in einer zufälligen Permutation wäre.

Der Heap Sort ist ein sehr effizienter Sortieralgorithmus, welcher wenig zusätzlichen Speicherplatz braucht. Tatsächlich benötigte er im eigenen Programm sogar am wenigsten Zeit für grosse Listen. Dieser Titel des schnellsten Algorithmus geht normalerweise an den Quick Sort. Dass der Heap Sort in diesem Fall schneller war, liegt bloss an der Implementierung im eigenen Programm. Womöglich der einzige Nachteil ist, dass er schon sortierte Listen unnötig bearbeitet. Trotzdem handelt es sich um einen Algorithmus, der für alle Listengrössen geeignet ist.

3.7 Shell Sort

Das grosse Problem des Insertion Sort ist, dass Elemente, die weit weg von ihrer endgültigen Position sind, viele Schritte dorthin benötigen. Shell Sort, benannt nach seinem Entwickler Donald Shell, behebt dieses Problem. Er setzt zunächst einen gewissen Abstand h fest. Dann beginnt er, alle Elemente ab dem $h + 1$ten der Reihe nach gemäss dem Prinzip des Insertion Sort einzuordnen. Allerdings erfolgen Vergleiche und Tausche zwischen Elementen mit dem Abstand h. Ein Einschub endet also dann, wenn das Element, das sich h Stellen weiter vorne befindet, einen kleineren Schlüssel hat oder gar nicht existiert. Wurden alle Einschübe durchgeführt, wird der Abstand h verkleinert und der Prozess wiederholt. Wenn h am Schluss 1 ist, verhält sich der Shell Sort wie ein regulärer Insertion Sort. Allerdings wurden die Elemente von der vorherigen Prozedur schon nahe an ihre endgültige Position gebracht und es bleibt wenig zu tun übrig. Die Wahl der Abstände h steht weitgehend offen, doch der Einfachheit halber wird hier die Folge von Hibbard benutzt, also Zahlen der Art $2^k - 1$. Konkret sind dies Abstände von 1, 3, 7, 15 usw. Elementen. Der grösste Abstand, der benutzt wird, ist jeweils der höchste, der kleiner als n ist. Obwohl seit Jahrzehnten bekannt, ist der Shell Sort immer noch nicht vollständig verstanden. Es ist nicht bekannt, welche Folge von Abständen am besten ist. Daher wird die genaue Zeitkomplexität für diesen Algorithmus nicht wie die anderen aufgrund logischer Zusammenhänge erklärt.

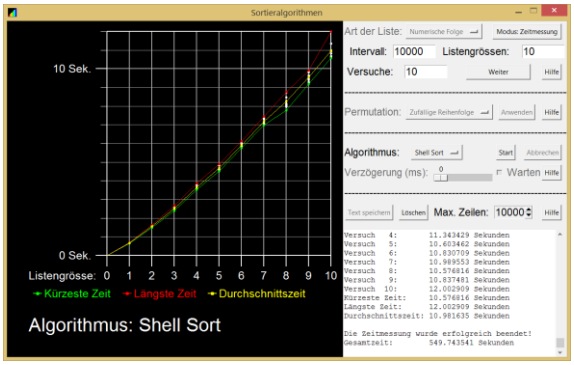


Abbildung 3.7a: Zeitmessung des Shell Sort

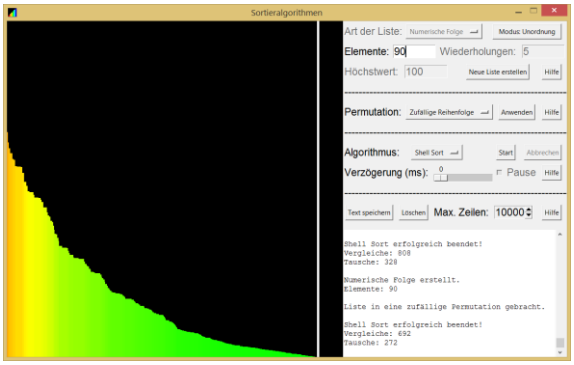


Abbildung 3.7b: Unordnungsverlauf des Shell Sort

Die Zeitkomplexität des Shell Sort ist besonders schwierig zu berechnen, weil sich verschiedene Reihen von Abständen unterschiedlich verhalten. Bei der Reihe von $2^k - 1$ werden $\log_2 n$ verschiedene Werte von h benutzt. Es finden ebenso viele untergeordnete Sortiervorgänge statt. Die Zeitkomplexität jedes solchen Vorgangs ist etwas über $O * n$, weil dank der Vorarbeit die einzelnen Einschübe meist sehr kurz sind. Allerdings können diese abhängig von der Grösse der Liste einige weitere Tausche benötigen. Laut Knuth wurde empirisch ermittelt, dass die durchschnittliche Zeitkomplexität beim Shell Sort mit Hibbards Folge von Abständen ungefähr $O * n^{5/4}$ ist. Dies stimmt ziemlich gut mit dem Graph von Abbildung 3.7a überein, welcher ein wenig stärker gekrümmt ist als diejenigen von den Algorithmen mit der Zeitkomplexität $O * n \log n$.

Der Idealfall für den Shell Sort ist eine schon sortierte Liste. Da in diesem Fall keine Tausche nötig sind, enden die Einschübe nach bloss einem Vergleich. Da in jedem Durchgang $n - h$ Elemente eingeschoben werden und der Einfluss von h immer kleiner wird, ist die Zeitkomplexität eines einzelnen Durchgangs im Durchschnitt ungefähr $O * n$. Weil die Anzahl der

verschiedenen Abstände h logarithmisch mit der Grösse der Liste wächst, ist der Zeitaufwand des Shell Sort bei einer sortierten Liste $O * n \log n$. Ist die Liste aber in der verkehrten Permutation, ist zwar ein grösserer Zeitaufwand als bei einer zufälligen Liste zu erwarten, doch beim Testen mit dem eigenen Programm war jedes Mal weniger Aufwand für die verkehrte Permutation nötig. Diese scheinbar widersprüchliche Beobachtung lässt sich so erklären, dass die frühen Durchgänge wegen der Permutation einen grösseren Einfluss als sonst haben. Da können Elemente noch mit wenig Aufwand weit geschoben werden. So bleibt für die letzten Durchgänge, die potenziell den meisten Aufwand verursachen, weniger zu tun übrig. Die tatsächliche maximale Zeitkomplexität mit Hibbards Abständen wurde auch empirisch ermittelt und liegt bei ungefähr $O * n^{1.5}$.

Wie auch der gewöhnliche Insertion Sort benötigt der Shell Sort nur einige Hilfsvariablen, deshalb ist seine Platzkomplexität ebenfalls $O * 1$.

Da der Shell Sort Elemente mit grossem Abstand tauschen kann, ist es möglich, dass gleiche Schlüssel ihre relative Reihenfolge ändern können. Liegt nämlich h Stellen vor einem solchen Schlüssel einer, der grösser ist und zwischendrin ein gleich grosser, kann letzterer übersprungen werden. Daher ist der Shell Sort instabil.

Abbildung 4.7b zeigt einen vollständigen Unordnungsverlauf des Shell Sort. Die relative Unordnung sinkt zunächst schnell, später aber langsamer. Weil am Anfang des Sortiervorgangs ein einziger Tausch Elemente sehr weit bewegt, können diese viel näher an ihre richtige Stelle geschoben werden, aber auch daran vorbei. Deshalb sind zunächst grosse und kleine Stufen in der Unordnung zu sehen. Später aber ist der Abstand h kleiner und damit auch die maximale Grösse dieser Stufen. Wenn h den Wert 1 erreicht und damit der normale Insertion Sort eintritt, sinkt die relative Unordnung von da an langsam und stetig auf 0, da die Anzahl der Tausche jedes Einschubs nicht mehr gross variiert. Weil der Shell Sort nach demselben Prinzip wie der Insertion Sort funktioniert und nach jedem Tausch der grössere Schlüssel weiter hinten ist, ist auch er unordnungsstet.

Wie bei allen anderen hier besprochenen Sortieralgorithmen, vom Heap Sort abgesehen, ändert sich die relative Unordnung während des Sortierens nie, wenn die Ausgangsliste schon sortiert ist, zumal der Shell Sort unordnungsstet ist. Anders sieht es bei einer verkehrten Liste aus. Die relative Unordnung ist da ursprünglich 1. Sonst sieht der Unordnungsverlauf nicht wesentlich anders aus als bei einer zufälligen Liste. Allerdings sind die einzelnen Phasen, in denen sich die relative Unordnung verkleinert, viel regelmässiger. Der Unordnungsverlauf einer fast sortierten Liste sieht auch nicht aussergewöhnlich aus, er beginnt lediglich bei einer tiefen relativen Unordnung.

Insgesamt ist der Shell Sort eine viel effizientere Variante des Insertion Sort. Allerdings fehlen diesem einige gute Eigenschaften. Ein Beispiel ist die Stabilität. Der Shell Sort ist im Gegensatz zum Insertion Sort instabil. Ausserdem braucht er im Falle einer ganz oder fast sortierten Liste unnötigen Zeitaufwand, da die Durchgänge mit höherem Abstand keinen oder nur einen geringen Einfluss auf die Liste haben. Der reguläre Insertion Sort hingegen braucht sehr wenig Aufwand, um die wenigen nötigen Tausche durchzuführen. Dadurch ist der Shell Sort in keinem Szenario die beste Wahl.

4 Eigener Sortieralgorithmus

Es gibt zwar schon zig Sortieralgorithmen, doch zu diesen wird noch ein weiterer hinzugefügt, der selber entwickelt wurde. Es gibt zwar keine Garantie, dass niemand vorher auf diese Idee gekommen ist, doch es finden sich keine Veröffentlichungen eines solchen Algorithmus. Da er bisher keinen vereinbarten Namen hat, wird er in dieser Arbeit Lukas Sort genannt. In mancher Hinsicht ist er ähnlich wie der Merge Sort, weil er rekursiv arbeitet und die Liste in gleiche Teile teilt, doch statt zuerst aufzuteilen und dann wieder zusammenzuführen, macht er zuerst die nötigen Vergleiche und Tausche, bevor er die Liste aufteilt. Im Folgenden wird zuerst die Funktionsweise des Lukas Sort erklärt, danach wird er nach demselben Schema analysiert wie die anderen Algorithmen.

4.1 Prinzip

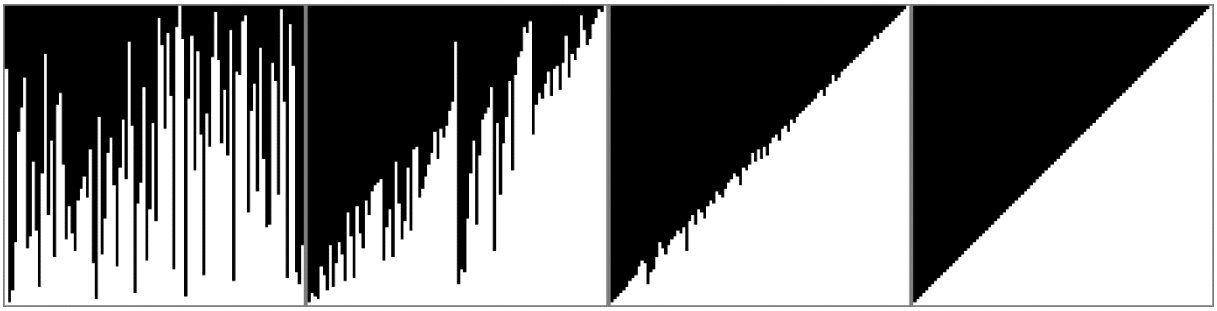


Abbildung 4.1: Die Liste nach 0, 1, 3 und 6 Durchgängen des Lukas Sort

Der Lukas Sort funktioniert, indem er einen relativ einfachen Vorgang mehrmals hintereinander ausführt, welcher die Liste jedes Mal näher zur sortierten Permutation bringt, bis sie fertig sortiert ist. Jeder derartige Prozess beginnt damit, dass das vorderste Element mit dem hintersten verglichen wird. Hat das vordere einen grösseren Schlüssel, werden beide getauscht. Dies geschieht auch mit den zweitäussersten Elementen, den drittäussersten, usw. bis die Mitte erreicht wird. Danach wird die Liste in zwei gleich grosse Teillisten aufgeteilt. Falls eine ungerade Anzahl Elemente vorliegt, wird das mittlere in beiden Hälften benutzt. Dies stellt sicher, dass dieses, wenn nötig, auch bewegt werden kann. Beide Teillisten werden dann genauso behandelt wie die Gesamtliste, d.h. ihre Elemente werden ebenfalls von aussen nach innen verglichen und getauscht. Diese Teillisten werden danach wieder aufgeteilt. Dies wiederholt sich solange, bis die Teillisten nur noch aus zwei Elementen bestehen. Ist der Prozess beendet, wird geprüft ob die Liste sortiert ist. Dabei werden alle Paare aufeinanderfolgender Elemente verglichen. Wird eines gefunden, in welchem der vordere Schlüssel grösser als der hintere ist, wird die Prüfung der Liste abgebrochen und die vorherige Prozedur nochmals durchgeführt. Abbildung 4.1 zeigt, wie eine zufällige Liste ursprünglich, nach einer, nach drei und nach sechs Iterationen aussieht.

Die genaue Funktionsweise des Lukas Sort wird im folgenden Pseudocode nochmals gezeigt. Die Begriffe in den Klammern bei «Definiere Iteration» und «Definiere Lukas Sort» sind Funktionsargumente, also der Input, den Funktionen annehmen. Der erste Block ist der Pseudocode für die einzelnen Iterationen. Der zweite lässt diese nacheinander ausführen, bis die Liste sortiert ist.

```
Definiere Iteration (Liste, Untergrenze, Obergrenze)
    Zeiger1 = Untergrenze
    Zeiger2 = Obergrenze
    Solange Zeiger1 <= Zeiger2
        Falls Zeiger1-tes Element von Liste > Zeiger2-tes Element von Liste
            Tausche Zeiger1-tes Element und Zeiger2-tes Element von Liste
        Vergrössere Zeiger1 um 1
        Verkleinere Zeiger2 um 1
    Falls Untergrenze +1 < Obergrenze
        Führe Iteration (Liste, Untergrenze, Zeiger2) aus
        Führe Iteration (Liste, Zeiger1, Obergrenze) aus

Definiere Lukas Sort (Liste)
    Sortiert = nein
    Solange Sortiert = nein
        Führe Iteration (Liste, 1, Anzahl Elemente in Liste) aus
        Sortiert = ja
        Für jedes n zwischen 1 und Anzahl Elemente in Liste – 1
            Falls n-tes Element von Liste > n+1-tes Element von Liste
                Sortiert = nein
                Für-Schleife abbrechen
```

4.2 Analyse

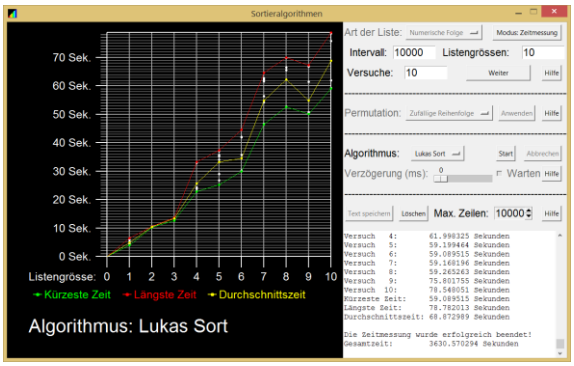


Abbildung 4.2a: Zeitmessung des Lukas Sort

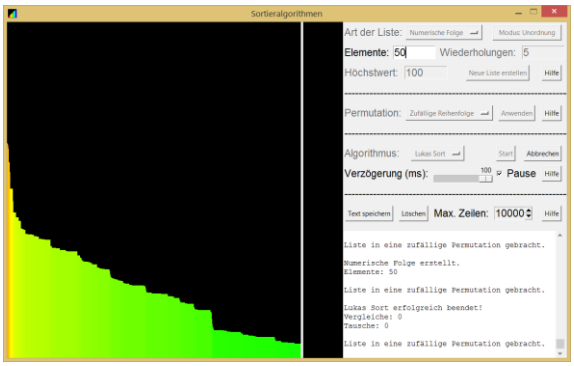


Abbildung 4.2b: Unordnungsverlauf des Lukas Sort (unvollständig)

Wie in Abbildung 4.2a zu sehen, steigt der Zeitaufwand in einem merkwürdigen Verhältnis zur Grösse der Liste. Das hat damit zu tun, dass der sich wiederholende Sortiervorgang nicht teilweise durchgeführt werden kann, sondern immer beendet wird. Er geschieht bei zufälligen Listen meistens ungefähr $\log_2 n$ Mal. Die Sprünge sollten daher ungefähr bei Listengrössen liegen, die Zweierpotenzen sind. In diesem Fall sind die plötzlichen Vergrösserungen des Zeitaufwands zwischen 30000 und 40000 sowie zwischen 60000 und 70000 Elementen gut zu sehen. Das entspricht der Theorie, da sich die Zweierpotenzen 32768 und 65536 dazwischen befinden. Die Iterationen haben ihrerseits eine Zeitkomplexität von je $O * n \log n$, da jedes Element in ungefähr $\log_2 n$ Vergleichen involviert ist. Daher hat der Lukas Sort insgesamt eine Zeitkomplexität von $O * n (\log n)^2$.

Da der Lukas Sort nur nach jedem Durchgang prüft, ob die Liste sortiert ist, wird mindestens einer ausgeführt. Bei einer schon sortierten Liste ist deshalb die minimale Zeitkomplexität wegen der einzigen Iteration $O * n \log n$. Allerdings kann der Algorithmus auch leicht umprogrammiert werden, sodass er auch am Anfang die Liste prüft, wobei der Zeitaufwand für eine sortierte Liste auf $O * n$ gesenkt wird. Wenn die Liste in der verkehrten Permutation ist, braucht der Lukas Sort nur eine Iteration, um diese zu sortieren, weil gleich am Anfang alle Paare sich gegenüberliegender Elemente verglichen und in diesem Fall getauscht werden. Fast sortierte Listen können ebenfalls in einem einzigen Durchgang geordnet werden, wenn sich keine Elemente mehr als ein paar Stellen von ihrem richtigen Platz befinden. Allerdings kann schon ein einziges Paar vertauschter Elemente die nötige Anzahl Iterationen der einer zufälligen Liste gleichstellen, weil Elemente, die vorher an ihrer richtigen Stelle waren, sehr weit davon entfernt werden können.

Der Lukas Sort arbeitet rekursiv und jede Rekursionsstufe hat ihre eigenen Variablen. Weil die Rekursionstiefe an einem Punkt etwa $\log_2 n$ erreicht, ist die Platzkomplexität dieses Algorithmus $O * \log n$.

Weil der Lukas Sort weit entfernte Elemente bewegen kann, ohne diejenigen, die zwischendrin liegen, zu beachten, kann ein Element ein anderes überspringen, welches denselben Schlüssel besitzt. Der Lukas Sort ist daher instabil.

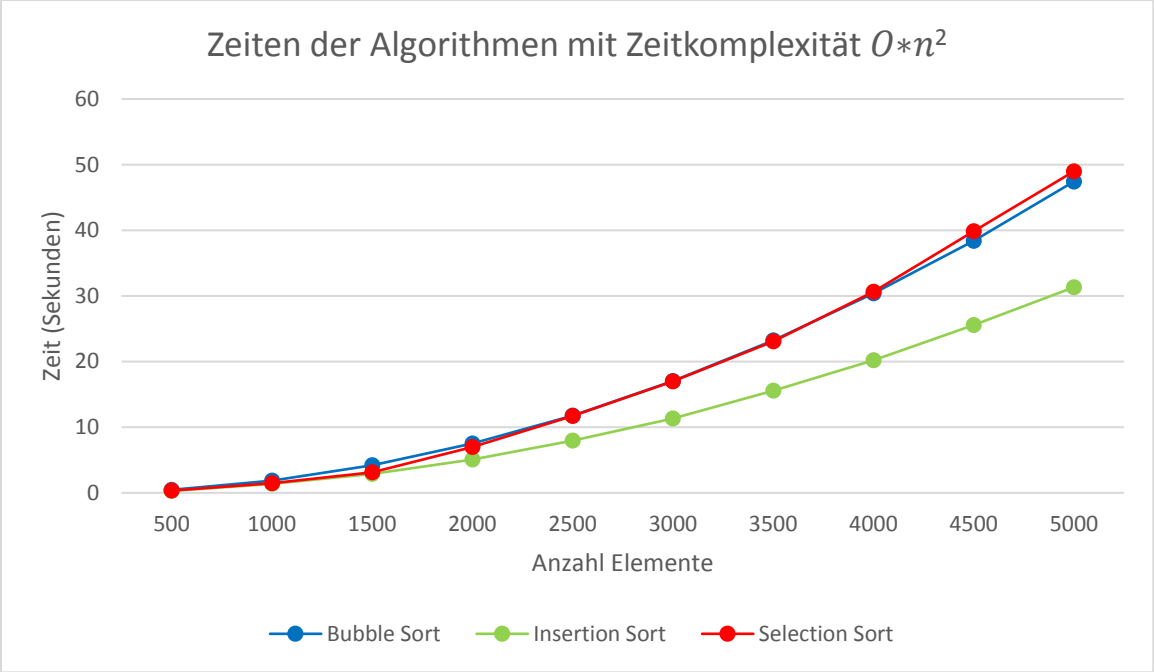
Abbildung 4.2b zeigt einen unvollständigen Unordnungsverlauf des Lukas Sort. Es ist leicht zu erkennen, dass die relative Unordnung zuerst schnell sinkt, dann immer langsamer. In einigen Stufen verkleinert sie sich später aber noch recht schnell. Die grosse Verkleinerung am Anfang entsteht, wenn der Lukas Sort die gesamte Liste behandelt. Da zuerst die äussersten Elemente verglichen und getauscht werden, können sie da am weitesten bewegt werden. Daher sinkt da die relative Unordnung rasch. Wenn nur noch kleinere Abschnitte der Liste bearbeitet werden, kann sie nur noch langsam kleiner werden. Jedes Mal, wenn eine neue Iteration des Lukas Sort begonnen wird und wieder die ganze Liste betrachtet wird, wird die relative Unordnung wieder schneller kleiner. In den letzten Durchgängen, wenn die Liste fast fertig sortiert ist, gibt es keine grossen Sprünge mehr. Die relative Unordnung sinkt schlussendlich langsam auf 0. Weil der Lukas Sort Elemente nur tauscht, wenn das vordere einen grösseren Schlüssel hat, kann die Unordnung nie steigen. Daher ist der Algorithmus unordnungsstet.

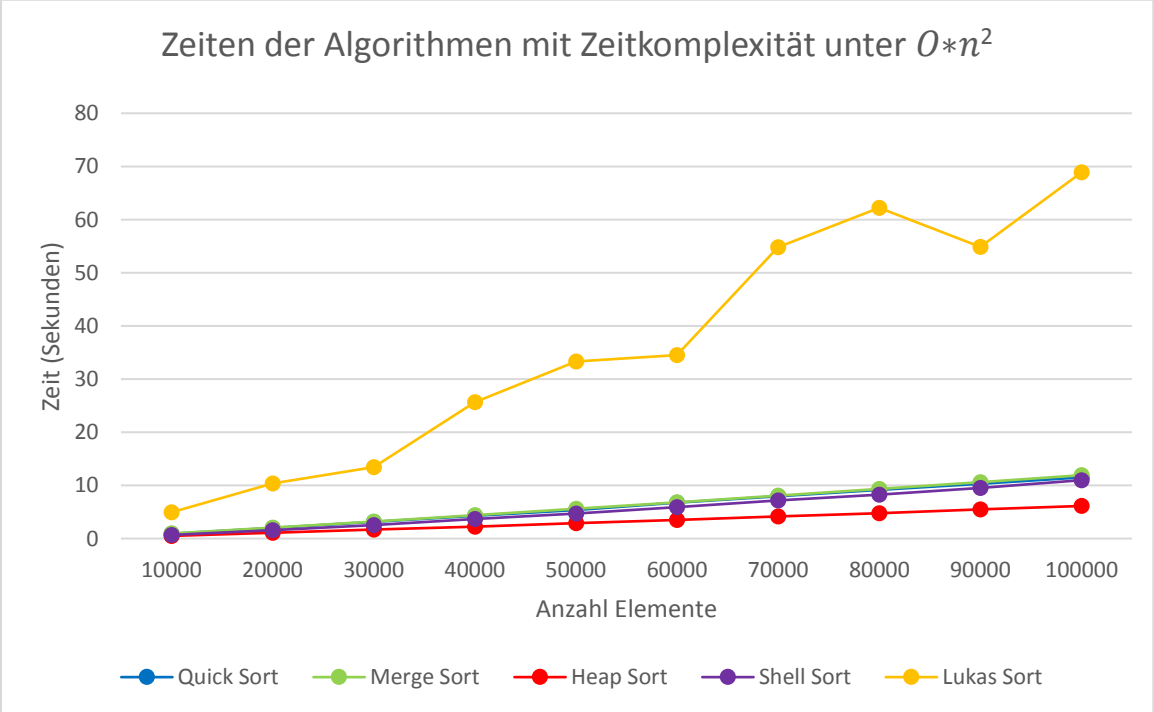
Im Falle einer bereits sortierten Liste beginnt die relative Unordnung bei 0. Dies bleibt auch so, weil der Lukas Sort unordnungsstet ist. Ist die Liste aber in der verkehrten Permutation, ist diese ursprünglich 1. Weil aber gleich die erste Iteration alle Elemente sofort an ihre richtige Stelle bewegt, fällt die relative Unordnung scharf auf 0 ab. Bis zum Schluss des Durchgangs bleibt sie dort. Wenn die Liste anfangs fast sortiert war, fängt die relative Unordnung tief an und braucht je nach Permutation mehr oder weniger Zeit, um in einigen kleinen Sprüngen 0 zu erreichen.

Der Lukas Sort ist zwar generell wesentlich effizienter als der Bubble und der Selection Sort, doch er hat eine höhere durchschnittliche und maximale Zeitkomplexität als der Merge und der Heap Sort. Verkehrte Listen sortiert der Lukas Sort besonders schnell, was ihm in dieser Situation sehr vorteilhaft macht. Ausserdem ist er nicht allzu kompliziert zu programmieren. Insgesamt ist der Lukas Sort ein guter, allgemein brauchbarer Sortieralgorithmus.

5 Fazit

Da nun einige Sortieralgorithmen ausführlich besprochen wurden, können ihre Eigenschaften übersichtlich zusammengefasst werden. Zunächst werden die durchschnittlichen Zeiten für jede Listengrösse in zwei Grafiken festgehalten. Eine davon enthält die Daten für diejenigen Algorithmen, die eine durchschnittliche Zeitkomplexität von $O * n^2$ haben. Diese wurden bei Listen mit 500 bis 5000 Elementen benutzt. Die zweite Grafik zeigt die Zeiten aller anderen Algorithmen, welche an Listen mit 10000 bis 100000 angewendet wurden.





Die folgende Tabelle zeigt weitere Eigenschaften der Sortieralgorithmen, darunter die verschiedenen Arten der Komplexität, die Stabilität und die Unordnungsstetigkeit. Die bestmöglichen Eigenschaften sind grün schattiert, besonders schlechte rot und mittelmässige gelb. Gäbe es einen perfekten Sortieralgorithmus, hätte dieser eine minimale Zeitkomplexität von $O * n$, eine durchschnittliche und maximale Zeitkomplexität von $O * n \log n$, eine Platzkomplexität von $O * 1$ und wäre stabil sowie unordnungsstet. Allerdings wurde noch kein derartiger Algorithmus entdeckt. Die Eigenschaften in der folgenden Tabelle wurden mit denen von https://www.toptal.com/developers/sorting-algorithms verglichen, wobei eine gute Übereinstimmung besteht.

Eigenschaft	Min. Zeitkomplexität	Durchschnittliche Zeitkomplexität	Max. Zeitkomplexität	Platzkomplexität	Stabil	Unordnungsstet
Bubble Sort	$O * n$	$O * n^2$	$O * n^2$	$O * 1$	Ja	Ja
Insertion Sort	$O * n$	$O * n^2$	$O * n^2$	$O * 1$	Ja	Ja
Selection Sort	$O * n^2$	$O * n^2$	$O * n^2$	$O * 1$	Nein	Ja
Quick Sort	$O * n \log n$	$O * n \log n$	$O * n^2$	$O * \log n$ bis $O * n$	Nein	Ja
Merge Sort	$O * n \log n$	$O * n \log n$	$O * n \log n$	$O * n$	Ja	Nein
Heap Sort	$O * n \log n$	$O * n \log n$	$O * n \log n$	$O * 1$	Nein	Nein
Shell Sort	$O * n \log n$	$O * n^{1.25}$	$O * n^{1.5}$	$O * 1$	Nein	Ja
Lukas Sort	$O * n \log n$	$O * n \log n^2$	$O * n \log n^2$	$O * n \log n$	Nein	Ja

Da nun alle Testresultate und Eigenschaften der Sortieralgorithmen in übersichtlicher Form dastehen, kann für verschiedene Situationen jeweils der am besten geeignete ausgewählt werden. Dabei können sich mehrere dieser Situationen überschneiden. Wenn zum Beispiel eine fast sortierte Liste vorliegt, nur begrenzter Speicherplatz verfügbar ist und Stabilität vorausgesetzt wird, ist der Insertion Sort die richtige Wahl.

- Für grosse zufällige Listen sind Quick Sort, Merge Sort, Heap Sort und Shell Sort allesamt gut geeignet, da sie eine geringe durchschnittliche Zeitkomplexität haben. Tatsächlich liegen in der zweiten Grafik von Seite 28 die Kurven vom Quick Sort, Merge Sort und Shell Sort fast aufeinander.
- Bei fast sortierten Listen ist der Insertion Sort die beste Wahl, weil seine Zeitkomplexität da zu $O * n$ tendiert.
- Bubble Sort, Insertion Sort, Selection Sort, Heap Sort und Shell Sort benötigen nebst der Liste nur wenig Speicherplatz für Hilfsvariablen und sind daher am besten, falls wenig Speicher zur Verfügung steht.
- Ist Stabilität vorausgesetzt, kommen Bubble Sort, Insertion Sort und Merge Sort in Frage.
- Wenn Unordnungsstetigkeit nötig ist, können alle besprochenen Sortieralgorithmen ausser Merge Sort und Heap Sort benutzt werden.
- Falls der Rechner Daten extrem schnell lesen, aber nur langsam schreiben kann, ist ein Sortieralgorithmus von Vorteil, der wenige Tausche benötigt. Der Selection Sort ist genau dieser Algorithmus.
- Für den Fall, dass die Liste in der verkehrten Permutation oder nahe dran ist, eignet sich der Lukas Sort gut.

Anhand dieser Punkte wird ersichtlich, dass es keinen einzigen besten Sortieralgorithmus gibt. Eine perfekte Methode wurde noch nicht gefunden, daher ist der am besten geeignete Algorithmus in einer Situation nicht derselbe wie in einer anderen. Der Heap Sort zum Beispiel ist generell effizient und braucht wenig Speicherplatz, ist aber instabil. Der Merge Sort hingegen ist stabil, aber Speicherintensiv. Der Insertion Sort hat zwar eine höhere durchschnittliche Zeitkomplexität, ist aber sowohl Stabil als auch platzsparend und bringt fast sortierte Listen besonders schnell in Ordnung. Gerade weil die besten Eigenschaften auf mehrere Sortieralgorithmen verteilt sind, konnten so viele davon entstehen. Der Lukas Sort ist ein Produkt der Suche nach dem idealen Algorithmus. Obwohl er in den meisten Aspekten mittelmässig ist, eignet er sich doch sehr gut für eine bestimmte Situation, nämlich eine Liste, die in der verkehrten Reihenfolge ist. In der Zwischenzeit wird weiterhin der perfekte Sortieralgorithmus gesucht…

6 Literaturnachweise

The Art of Computer Programming Volume 3: Sorting and Searching (Donald Knuth, ©1998)

Benutzt für: Recherchen über Funktionalität der Algorithmen

https://www.youtube.com/watch?v=kPRA0W1kECg (YouTube-Video: 15 Sorting Algorithms in 6 Minutes)

Benutzt für: Inspiration zum Thema Sortieralgorithmen

https://www.cs.wcupa.edu/rkline/ds/shell-comparison.html

https://en.wikipedia.org/wiki/Shellsort

Benutzt für: Zeitkomplexität des Shell Sort mit Hibbards Folge von Abständen

https://www.toptal.com/developers/sorting-algorithms

Benutzt für: Vergleich mit den eigenen Testresultaten

Taschenbuch der Algorithmen (Dorothea Wagner, Heribert Vollmer und mehr, ©2008)

Benutzt für: Fachbegriffe

7 Redlichkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe. Alle Quellen, die ich benutzt habe, sind korrekt angegeben.

Datum: 02.12.2017 Unterschrift: _____