Sortierverfahren*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Sortierverfahren, Insertionsort, Bubblesort, Mergesort, Heaps, Heapsort, Prioritätsschlangen, Bucketsort, Radixsort, Topologisches Sortieren, Komplexität des Sortierproblems

Empfohlene Vorkenntnisse: $\mathcal{O}$-Notation, Rekursion, Iteration

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

Eines der bestuntersuchten Problem in der Informatik ist das *Sortieren* von Mengen von Objekten. Sortieren muss man nicht nur Anwendungsdaten, z.B. Telefonbücher. Viele komplexere Algorithmen enthalten auch Sortierschritte als Teil des eigentlichen Verfahrens, z.B. die Algorithmen zur Berechnung von Spannbäumen. Allerdings wird kaum ein Informatiker heute noch einen Sortieralgorithmus selbst programmieren. Jede brauchbare Programmiersprache hat heute Bibliotheken mit effizient implementierten Sortieralgorithmen. Trotzdem lernt jeder Informatiker Sortieralgorithmen kennen, quasi als „kleines Einmaleins“ der Informatik. Die verschiedenen Sortierverfahren eignen sich aber auch zur Illustration von unterschiedlichen algorithmischen Vorgehensweisen. Des weiteren bilden die Ergebnisse zum Sortierproblem Ausgangspunkt für weitere Resultate in der Komplexitätstheorie. Aus all diesen Gründen lohnt es sich, die Sortierproblematik genauer anzuschauen.

1.1 Einteilungen

Wir haben also eine Menge von Objekten, die wir in eine Reihenfolge bringen möchten. Um das zu ermöglichen, muss es für die Objekte eine *Ordnung* geben. Eine ganz grundlegende Unterscheidung ergibt sich dabei zunächst, ob es für die Objekte eine *Totalordnung* oder nur eine *Partielle Ordnung* gibt.

Bei einer **Totalordnung** kann man für jedes Paar von Objekten entscheiden, in welche Reihenfolge sie gehören, wenn sie nicht sowieso schon gleich sind. Beispiele sind die ganzen Zahlen mit ihrer natürlichen Ordnung, oder Zeichenketten, die lexikographisch geordnet werden.

Bei einer **partiellen Ordnung** oder **Halbordnung** sind einige Objekte vergleichbar, andere aber nicht. Ein Beispiel wäre die Ahnenbeziehung. Eine Person A käme vor einer Person B, wenn A Nachkomme von B ist. Also Kind kommt vor Vater und Mutter, Vater vor Opa, Opa vor Uropa usw. Aber Onkel und Tante wären nicht vergleichbar mit dieser Beziehung (in normalen Familien). Die Sortierverfahren dafür nennt man *Topologisches Sortieren*

Für Objekte mit Totalordnung unterscheidet man noch folgende zwei Fälle:

Vergleichsbasiertes Sortieren: Der Algorithmus nutzt lediglich aus, dass man für je zwei Objekte die Reihenfolge bestimmten kann.

Nicht vergleichsbasiertes Sortieren: Man kann für jedes Objekt direkt entscheiden, an welche Position es kommt, indem man es z.B. in eine von einer Reihe vorsortierter Kisten tut.

1.2 Eigenschaften

Für die praktische Anwendung von Sortieralgorithmen sind verschiedene Eigenschaften wichtig:

Die Zeitkomplexität: diese ist um so wichtiger, je mehr Objekte zu sortieren sind. Die Zeitkomplexität wird dabei in $\mathcal{O}(f(n))$ angegeben, wobei n die Anzahl der zu sortierenden Objekte ist, und $f(n)$ das Wachstum der Funktion.
Man unterscheidet dabei:
- **Best Case**: wie schnell sortiert der Algorithmus im besten Fall, und wie sieht dieser Fall aus?
- **Average Case**: wie schnell sortiert der Algorithmus so im Durchschnitt, und wie muss dieser Durchschnittsfall aussehen?
- **Worst Case**: wie schnell sortiert der Algorithmus im schlimmsten Fall, und wie sieht dieser Fall aus?

Die Speicherstruktur: Zunächst einmal ist es wichtig, ob der Algorithmus auf Arrays oder auf verketten Liste, oder auf beiden arbeiten kann. Ein weiterer Aspekt ist, ob der Algorithmus zusätzlichen Speicher benötigt, oder mit dem Ausgangsspeicher für die Objekte auskommt. Dann bezeichnet man ihn als *in place*-Algorithmus.

Die Stabilität: Ein *stabiler* Algorithmus behält die Reihenfolge von Objekten, die gleich sind, bei, und sortiert die nicht nochmal um. Das hat sehr praktische Auswirkungen. Angenommen, man sortiert eine Notenliste zunächst nach Namen. Sagen wir, Maier hat eine 1.0, und Schulze hat auch eine 1.0. Danach sortieren wir die Liste noch nach Noten. Ein stabiler Algorithmus behält die Reihenfolge Maier, Schulze bei. Ein nicht-stabiler Algorithmus kann die Reihenfolge bei der Notensortierung u.U. umdrehen.

Online/Offline-Algorithmus: Ein Sortierverfahren ist ein *Online-Algorithmus*, falls die zu sortierenden Objekte auch im Laufe des Sortiervorgangs noch ergänzt werden können. Falls alle Daten zu Beginn des Algorithmus bekannt sein müssen, ist es ein *Offline-Algorithmus*.

Parallelisierung: betrifft das Problem, den Algorithmus zu parallelisieren. Da fast jeder moderne Computer mehrere Prozessoren hat, ist die Möglichkeit zur Parallisierung des Algorithmus ein immer wichtiger werdendes Kriterium. Dies wurde allerdings bisher noch nicht für alle Algorithmen untersucht.

Die meisten der vorgestellten Algorithmen haben eine Zeitkomplexität zur Sortierung von n Objekten von $\mathcal{O}(n^2)$ oder $\mathcal{O}(n \log(n))$. Wie signifikant dieser Unterschied ist, zeigt die folgende Tabelle.

n	$n \log_2(n)$	n^2	Unterschied
10	$33,2$	100	3
20	$86,4$	400	4.6
100	$664,4$	10000	15
1.000	$9965,8$	1.000.000	100
1.000.000	$1.993.156,8$	$1 Billion$	501.716

In den folgenden Kapiteln werden zunächst einige der prominentesten vergleichsbasierten Sortieralgorithmen eingeführt, dann die beiden nicht vergleichsbasierten Sortieralgorithmen Bucketsort und

Radixsort. Das letzte Kapitel widmet sich den topologischen Sortieralgorithmen. In der Literatur findet man aber eine ganze Reihe weiterer Sortieralgorithmen, die hier nicht behandelt werden.

Des weiteren wird eines der fundamentalen Ergebnisse der theoretischen Informatik behandelt: es kann keinen vergleichsbasierten Sortieralgorithmus geben, dessen Zeitkomplexität besser als $\mathcal{O}(n \log(n))$ ist.

2 Vergleichsbasierte Sortieralgorithmen

Alle Algorithmen in diesem Abschnitt gehen davon aus, dass man für je zwei Objekte die Reihenfolge bestimmen kann. In fast allen Programmiersprachen nimmt man dazu eine Funktion:

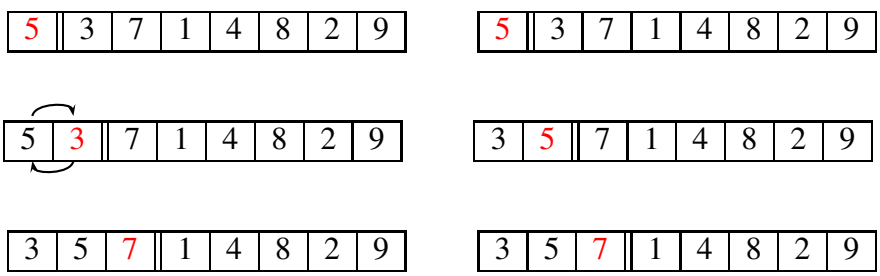
$$\mathit{Reihenfolge}(\mathit{Objekt\ 1}, \mathit{Objekt\ 2}) = \begin{cases} -1 & \text{falls } \mathit{Objekt\ 1} \text{ vor } \mathit{Objekt\ 2} \text{ kommt,} \\ 0 & \text{falls } \mathit{Objekt\ 1} \text{ und } \mathit{Objekt\ 2} \text{ gleich sind,} \\ +1 & \text{sonst} \end{cases}$$

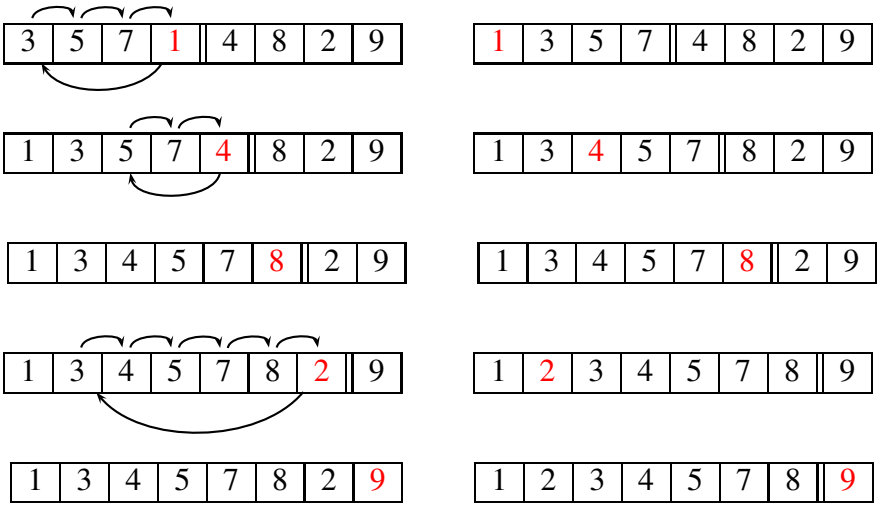
Der Fall, dass die beiden Objekte gleich sind, kommt z.B. bei der Sortierung einer Klausurliste nach Noten vor. Alle Teilnehmer mit gleicher Note werden als gleich betrachtet. Dem Algorithmus ist es dann frei gestellt, in welche Reihenfolge er sie bringt. Ein stabiler Algorithmus würde allerdings die Ursprungsreihenfolge beibehalten.

2.1 Insertionsort

Jeder, der einmal ein Kartenspiel gespielt hat, hat entweder dieses Verfahren schon selbst angewandt, oder wenigstens gesehen, wie ein anderer es angewandt hat. Um den Stapel Karten, den sie unsortiert auf der Hand haben, zu sortieren, gehen Spieler von links nach rechts so vor: Die linkeste Karte bleibt zunächst, wo sie ist. Die zweite Karte bleibt entweder da, wo sie ist, oder sie muss mit der ersten Karte vertauscht werden. Ab der dritten Karte, sagen wir, Karte an Position n, wird der bisher sortierte Teil durchsucht, um die richtige Position j für die nächste Karte zu finden. Alle Karten ab Position j bis Position n werden um eine Stelle nach hinten verschoben, so dass Position j für die Karte von Position n frei wird. Genau so funktioniert *Insertionsort*.

Folgendes Beispiel mit *einem Array* von Zahlen illustriert das Verfahren. Das linke Array zeigt in rot den aktuell einzusortierenden Wert. Mit den Pfeilchen werden die nötigen Verschiebungen angezeigt. Das rechte Array zeigt das Ergebnis der Verschiebungen.





2.1.1 Eigenschaften von Insertionsort

Speicherstruktur:
Das Verfahren arbeitet ausschließlich auf dem Eingabearray. Es benötigt dazu keinen extra Speicher und ist daher *in place*.

Das Array hat auch den Vorteil, dass man die richtige Position für das Einfügen des aktuellen Objekts durch *Binäre Suche*[1] finden kann. Allerdings muss man Elemente im Array verschieben können.

Stabilität:
Das Verfahren verändert nur dann Reihenfolgen wenn es notwendig ist. Ansonsten bleiben die Element in der vorgefundenen Reihenfolge. Insertionsort ist also *stabil*.

Zeitkomplexität für das Sortieren von n Elementen:
Best Case: Am wenigsten Aufwand ist, wenn die Elemente schon in der richtigen Reihenfolge sortiert sind. Dann muss nichts verschoben werden, und ein Durchgang durch das Array genügt. Der Aufwand ist dann $\mathcal{O}(n)$.

Worst Case: Am meisten Aufwand gibt es, wenn die Elemente in der falschen Reihenfolge sortiert sind. Dann kommt das aktuelle Element immer an den Anfang, und der Rest muss verschoben werden. Es sind dann $\Sigma_{i=2}^{n-1} n$ Verschiebungen notwendig, mit $\mathcal{O}(n^2)$ Aufwand.

Average Case: Wenn das jeweilig nächste Element ungefähr in die Mitte des sortierten Teils verschoben werden muss, hat man den $\mathcal{O}(\log(n))$ Aufwand für die Suche der richtigen Position, und immer noch $\mathcal{O}(n^2)$ Aufwand für das Verschieben. D.h. der Gesamtaufwand ist immer noch $\mathcal{O}(n^2)$.

[1] Binäre Suche arbeite so, wie man in einem alphabetische sortierten Lexikon einen Eintrag suchen würde: man halbiert die Seiten, und testet, ob der Eintrag in der linken oder rechten Hälfte ist. Abhängig davon halbiert man diese Hälfte wieder und testet wieder, in welcher Hälfte der Eintrag ist. Man halbiert so lange, bis es nichts mehr zu halbieren gibt. Ein Array mit n Einträgen kann man $\log_2(n)$ mal halbieren. Daher ist die Zeitkomplexität von binärer Suche $\mathcal{O}(\log(n))$.

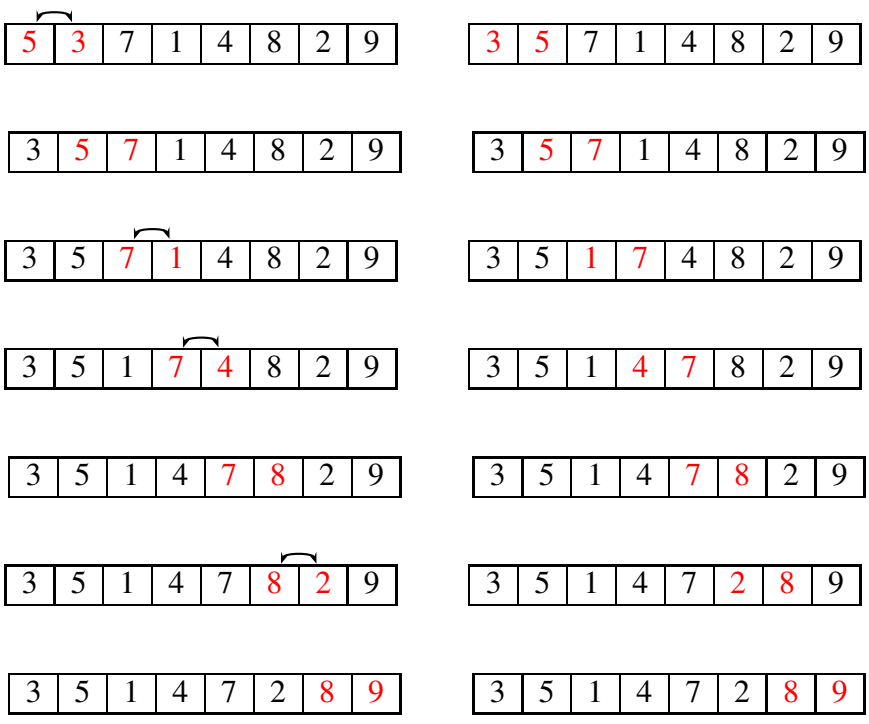
Online-Algorithmus: Die Idee des Algorithmus hängt nicht davon ab, dass die einzusortierende Elemente im Array selbst sind. Sie können auch nacheinander von außen kommen. Daher ist das Verfahren als *Online-Algorithmus* verwendbar.

2.2 Bubblesort

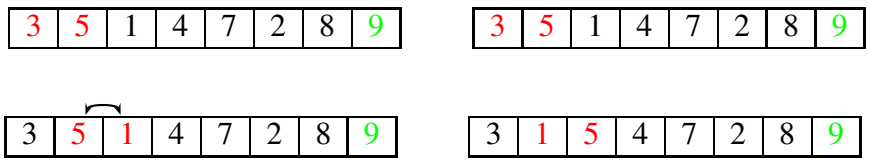
Dieser Algorithmus sortiert ebenfalls ein *Array* von Objekten. Dabei läuft er *mehrfach* von links nach rechts durch das Array und vergleicht immer 2 Objekte paarweise. Falls die beiden Objekte in der falschen Reihenfolge stehen, werden sie vertauscht. Im 1. Durchgang wird dabei das größte Objekt ganz nach rechts geschoben, so dass es im 2. Durchgang nicht mehr betrachtet werden muss. Im 2. Durchgang wird das zweitgrößte Objekt nach rechts geschoben usw. Eine erste Optimierung besteht darin, sich zu merken, ob während eines Durchgangs überhaupt eine Vertauschung notwendig war. Ist das nicht der Fall, ist kein weiterer Durchgang notwendig.

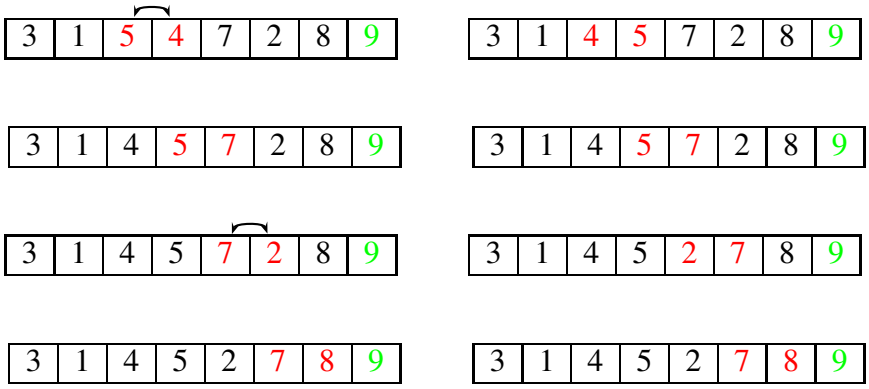
Folgendes Beispiel illustriert die Vorgehensweise. Man sieht wie die rot markierten Paare durch das Array wandern, und, falls nötig, vertauscht werden. Die ab dem 2. Durchgang grün markierten Zahlen ganz rechts brauchen dann nicht mehr getestet werden.

1. Durchlauf

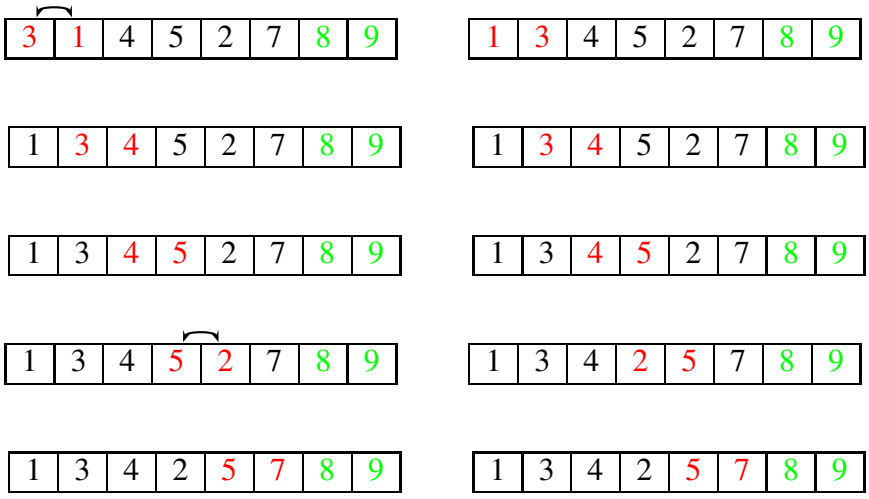


2. Durchlauf:

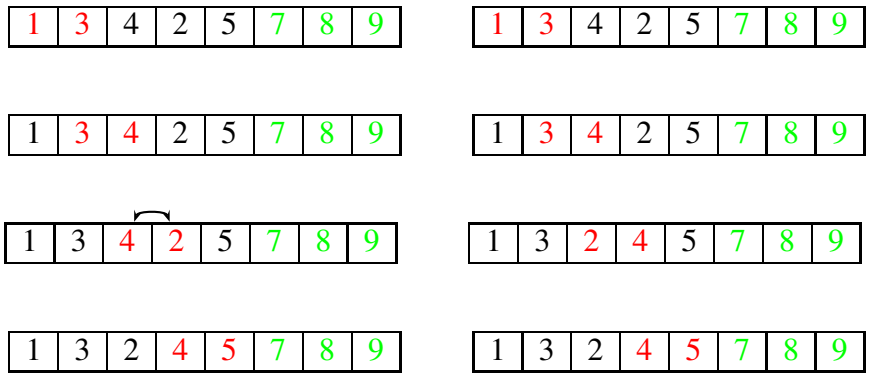




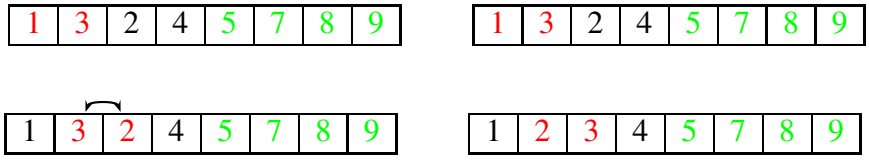
3. Durchlauf



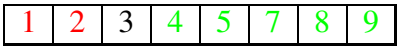
4. Durchlauf

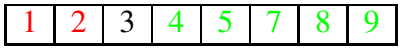


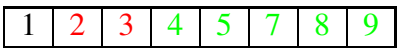
5. Durchlauf

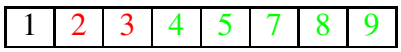


6. Durchlauf









Es wurde keine Vertauschung durchgeführt. Daher ist kein weiterer Durchgang notwendig.

2.2.1 Eigenschaften von Bubblesort

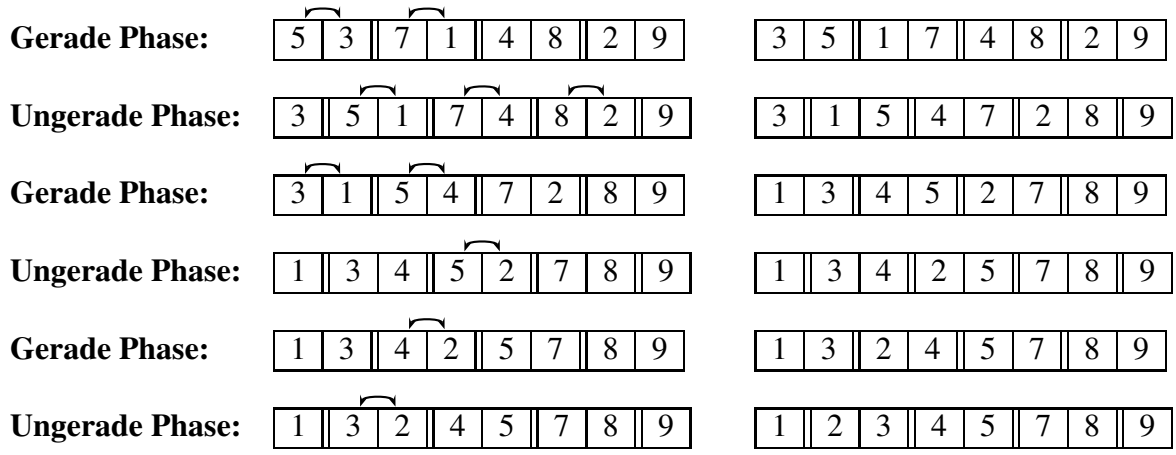
Speicherstruktur:
Das Verfahren arbeitet ebenfalls ausschließlich auf dem Eingabearray. Es benötigt dazu keinen extra Speicher und ist daher *in place*.

Stabilität: Bubblesort vertauscht nur wenn es unbedingt nötig ist. Es ist daher ebenfalls ein stabiles Verfahren.

Zeitkomplexität für das Sortieren von n Elementen:
- Best Case: Wenn das Array schon sortiert ist, dann ist beim ersten Durchgang keine Vertauschung notwendig, und das Verfahren stoppt. Der Aufwand für einen Durchgang ist $\mathcal{O}(n)$.
- Average Case: Man kann die erwartete Anzahl von Vergleichen bei einer zufällig gewählten Permutation der Zahlen $1, \ldots, n$ berechnen. Der Aufwand dafür ist $\mathcal{O}(n^2)$.
- Worst Case: Falls die Liste falsch herum sortiert ist, braucht müssen $n \cdot (n-1)/2$ Vertauschungen durchgeführt werden, so dass der Aufwand ebenfalls $\mathcal{O}(n^2)$ ist.

Online-Algorithmus: Da nach dem k-ten Durchlauf die k größten Elemente nicht mehr angefasst werden, kann auch ein neu angekommenes Element nicht mehr dazwischen einsortiert werden. Man könnte das Element ganz links einfügen, und den Algorithmus nochmal ganz von vorne laufen lassen.

Parallelisierung: Bubblesort kann in der Tat parallelisiert werden, wobei man bis zu $n/2$ Prozessoren einsetzen kann. Man teilt das Verfahren in zwei Phasen auf, die sich abwechseln: die *gerade Phase* und die *ungerade Phase*. In der geraden Phase vergleichen sich *parallel* die Element an den Positionen 0 und 1, 2 und 3 usw. In der ungeraden Phase vergleichen sich die Element an den Positionen 1 und 2, 3 und 4 usw. Die Phasen wechseln sich so lange ab, bis keine Vertauschung mehr stattfindet. Das gleiche Beispiel wie oben illustriert die Vorgehensweise.



In der nächsten geraden Phase gibt es keine Vertauschung mehr, und das Verfahren endet. Im schlimmsten Fall, wenn das kleinste Element von ganz rechts nach ganz links verschoben werden muss, braucht man n Durchgänge.

2.3 Mergesort

Dies ist ein Verfahren sortiert ein Array oder eine Liste *rekursiv*. Es funktioniert folgendermaßen:

- Das Array wird in zwei möglichst gleich große Teilarrays zerlegt.
- Die Teilarrays werden mit Mergesort rekursiv sortiert.
- Die beiden sortierten Teile werden im Reißverschlussverfahren zu einem sortierten Array zusammengefasst (engl. *merging*).

Die Basis der Rekursion ist der Fall, dass das nur ein oder zwei Elemente zu sortieren sind.

Als Beispiel sortieren wir (5,3,10,7,1,9,2,8).

Aufspaltung:	(5,3,10,7)	(1,9,2,8).	Reißverschluss:	(1) Rest (3,5,7,10) (2,8,9)
Aufspaltung:	(5,3) | (10,7)	(1,9) | (2,8).		(1,2) Rest (3,5,7,10) (8,9)
Sortierung:	(3,5) | (7,10)	(1,9) | (2,8)		(1,2,3) Rest (5,7,10) (8,9)
Reißverschluss:	(3) Rest (5) (7,10)	(1) Rest (9) (2,8)		(1,2,3,5) Rest (7,10) (8,9)
	(3,5) Rest () (7,10)	(1,2) Rest (9) (8)		(1,2,3,5,7) Rest (10) (8,9)
	(3,5,7) Rest () (10)	(1,2,8) Rest (9) ()		(1,2,3,5,7,8) Rest (10) (9)
	(3,5,7,10)	(1,2,8,9)		(1,2,3,5,7,8,9) Rest (10) ()
				(1,2,3,5,7,8,9,10)

Das Beispiel verdeutlicht auch, dass für jede Rekursionstiefe der Gesamtaufwand für die Aufspaltung und das Reißverschlussverfahren immer so groß ist wie die Originalliste lang ist.

2.3.1 Eigenschaften von Mergesort

Speicherstruktur Mergesort kann sowohl verkettete Listen als auch Arrays sortieren. Im Fall von Arrays wird für den Merge-Schritt ein gleich großes Hilfsarray verwendet. Dann ist das Verfahren

nicht in place. Mergsort eignet sich auch zum Sortieren sehr großer Listen, die nicht mehr in den Hauptspeicher passe. Dann müssen die Teillisten extern gespeichert werden, und für den Merge-Schritt evtl. stückweise in den Hauptspeicher geladen werden.

Stabilität: Das Verfahren vertauscht auch nur, wenn es nötig ist. Ansonsten wird die Reihenfolge der Elemente beibehalten. Es ist also stabil.

Zeitkomplexität: Das Verfahren macht immer die gleichen Schritte, egal ob die Objekte schon sortiert sind, oder nicht. Daher ist die Komplexität im Worst Case, Average Case und Best Case gleich, nämlich $\mathcal{O}(n \cdot \log(n))$. Die Rekursionstiefe ist ungefähr $\log_2(n)$. In jeder Rekursionstiefe sind n Elemente zu mergen. Das macht $\mathcal{O}(n \cdot \log(n))$.

Online-Algorithmus: Das klappt leider nicht. Wenn während die Rekursion läuft, neue Elemente hinzukommen, können sie nicht berücksichtigt werden.

Parallelisierung: Die jeweiligen Teillisten können unabhängig voneinander, und daher auch parallel, sortiert werden. Nur der Merge-Schritt muss von einem Prozessor durchgeführt werden. Am besten eignet sich dafür eine Master-Slave Architektur für die Parallelisierung.

2.4 Quicksort

Quicksort ist eine Modifikation von Mergesort, bei der der Merge-Schritt vereinfacht wird. Die Idee ist, die Aufteilung in zwei Teilprobleme so zu machen, dass die beiden Teillisten oder Teilarrays nach dem rekursiven Sortieren einfach aneinander gehängt werden können.

Dazu wird zunächst ein Element der zu sortierenden Objekte ausgewählt, das sog. *Pivotelement*. Dann werden die Objekte so vertauscht, dass alle Objekte, die kleiner sind als das Pivotelement links davon kommen, und alle größeren Elemente rechts davon. Jetzt kann man beide Teile unabhängig voneinander rekursiv sortieren. Durch die Vertauschung zu Beginn ist garantiert, dass man die beiden sortierten Teile einfach aneinander hängen kann.

An folgendem Beispiel wird die Idee verdeutlicht:

Meist wählt man einfach das Element in der Mitte als Pivotelement. Das ist die grüne 4.

5	3	7	1	4	8	2	9	6

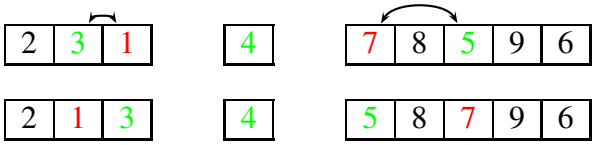
Die Liste muss umgeordnet werden, damit alle Zahlen < 4 links von der 4 stehen, und alle Zahlen > 4 rechts davon. Dazu geht man *synchron* von links und von rechts durch die Liste und sucht das erste Paar, welches falsch steht. Im Beispiel ist das links die 5 und rechts die 2. Diese werden vertauscht.

Links steht noch die 7 falsch. Rechts steht nichts mehr falsch. Daher kann man die 7 mit dem Pivotelement vertauschen.

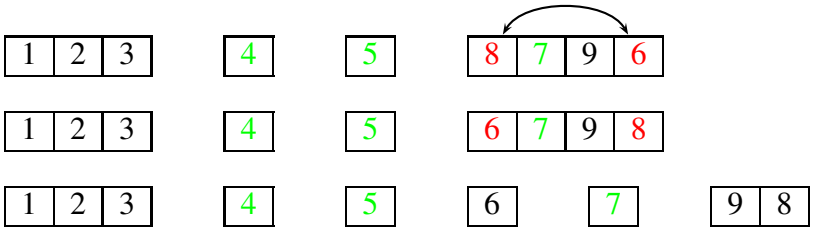
Jetzt ist links vom Pivotelement alles richtig, aber rechts steht die 1 falsch. Diese wird mit dem Pivotelement vertauscht.

Jetzt sind die beiden Teile richtig separiert, und Quicksort kann für sie rekursiv aufgerufen werden.

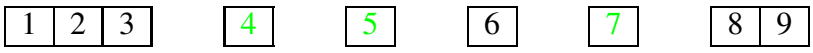
Links wird die 3 als Pivotelement gewählt, und rechts die 5. Daraus ergeben sich zwei Vertauschungen:



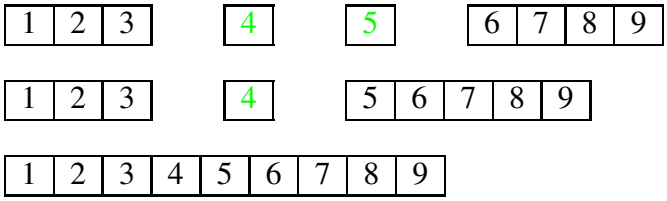
Bei der linken Liste bleibt nur eine Teilliste der Länge 2 zu sortieren. Das macht man direkt. Bei der rechten Liste hat man eine 4-er Liste rekursiv zu sortieren.



Die rechte 2-er Liste kann direkt sortiert werden.



Jetzt ist die Rekursion zuende, und es beginnt der Zusammenbau der Teile:



In diesem Beispiel haben wir das Pivotelement aus der Menge selbst genommen. Das muss aber nicht so sein. Wenn man z.B. Namen alphabetisch sortieren möchte, dann könnte man als Pivotelemente die Buchstaben des Alphabets nehmen, und im Aufteilungsschritt nach den Anfangsbuchstaben der Namen aufteilen. Zu Beginn könnte man z.B. alle Namen mit Anfangsbuchstaben $\leq m$ nach links, und alle mit Anfangsbuchstaben $> m$ nach rechts einordnen. Man kann somit Heuristiken zur Verteilung der Datensätze auf die beiden Teilmengen anwenden.

2.4.1 Eigenschaften von Quicksort

Speicherstruktur: Quicksort arbeitet am besten auf Arrays, kann aber auch für doppelt verkettete Listen eingesetzt werden. Es braucht keinen zusätzlichen Speicher und ist daher *in place*. Durch die Rekursion wird allerdings Speicher auf dem Stack benötigt.

Stabilität: Durch die willkürliche Vertauschung zu Beginn der Rekursion ist der Algorithmus leider nicht stabil. Es können unkontrollierbare Verschiebungen stattfinden.

Zeitkomplexität für das Sortieren von n Elementen.

Worst Case: Die Wahl des Pivotelements ist entscheidend für die Aufteilung in zwei Unterprobleme. Falls es unglücklicherweise so gewählt wird, dass eine Teilmenge immer $n-1$ Elemente hat, und die andere Teilmenge leer ist, dann ist die Rekursionstiefe n, und die Umordnungsoperation zu Beginn einer Rekursion in Rekursionstiefe k muss durch $n-k$ Elemente laufen. Der Aufwand ist damit $n + (n-1) + \ldots + 1$, und das ergibt einen Gesamtaufwand von $\mathcal{O}(n^2)$.

Best Case: In diesem Fall wird das Pivotelement immer so gewählt, dass die Aufteilung die Menge halbiert. Dann gleicht der Aufwand dem von Mergesort. Anstelle des Merge-Schrittes zum Schluss hat man den Aufteilungsschritt zu Beginn, und das hat beides mal einen Aufwand von $\mathcal{O}(n)$. Der Gesamtaufwand ist daher $\mathcal{O}(n \log(n))$.

Average Case: Der Durchschnittsfall liegt wohl auch eher so, dass die Aufteilung in die zwei Teilmengen ungefähr gleich große Teilmengen ergibt. Dann ist der Aufwand ebenfalls $\mathcal{O}(n \log(n))$.

Interessant ist auch das Verhalten von Quicksort, wenn die Mengen schon sortiert sind. Wenn man immer das mittlere Element in der Liste als Pivotelement nimmt, und die Menge richtig herum sortiert ist, dann ist das der Optimalfall, und der Aufwand ist $\mathcal{O}(n \log(n))$. Wenn die Menge falsch herum sortiert ist, dann vertauscht der Aufteilungsschritt zu Beginn die Elemente so, dass sie anschließend richtig herum sortiert sind. Der Aufwand ist dann ebenfalls $\mathcal{O}(n \log(n))$.

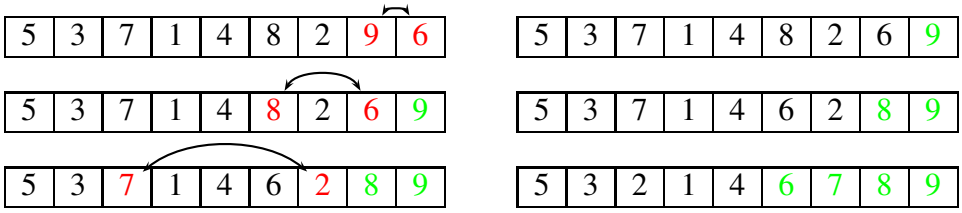
Online/Offline-Algorithmus: Wie bei Mergesort auch, ist es kaum möglich, während die Rekursion läuft, noch weitere Elemente einzubeziehen. Quicksort ist daher ein Offline-Algorithmus.

Parallelisierung: Nach der Aufteilung in die zwei Teilemengen können diese völlig unabhängig voneinander bearbeitet werden. Daher lässt sich Quicksort in einem hierarchischen Master-Slave System sehr gut parallelisieren.

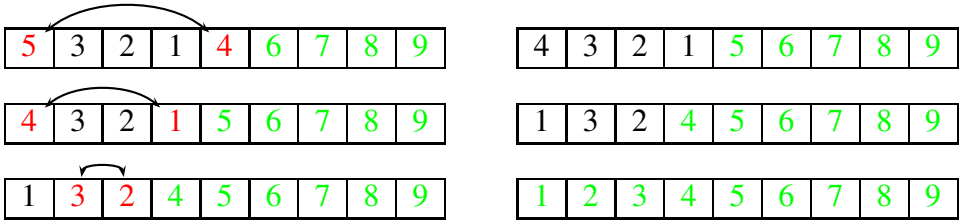
2.5 Selectionsort und Heapsort

Die Grundidee von *Selection-Sort* zum Sortieren in Arrays $a_1, \ldots, a_n$ ist:
- Man sucht das größte Element und vertauscht es mit dem letzten Element im Array.
- Dann sucht man das größte Element im Array $a_1, \ldots, a_{n-1}$ und vertauscht es mit dem vorletzten Element a_{n-1}
- usw.

Auf diese Weise baut sich von hinten nach vorne die sortierte Reihenfolge auf, so wie im nächsten Beispiel.



6 muss nicht vertauscht werden



Die Suche nach dem größten Element in einem Array mit n Elementen braucht zunächst einen Aufwand von $\mathcal{O}(n)$. Das kann aber ganz erheblich beschleunigt werden, indem man einen sog. *Heap* verwendet. Damit kann man mit $\mathcal{O}(\log n)$ Aufwand erreichen, dass das größte Element am Anfang des Arrays steht.

Da Heaps eine baumorientierte Datenstruktur sind, und auch noch weitere Anwendungen, wie z.B. Prioritätsschlangen hat, werden Heaps und Heapsort in einem eigenen Miniskript *Heaps.pdf* eingeführt.

2.5.1 Eigenschaften von Selectionsort und Heapsort

Speicherstruktur: Beide sind effizient nur auf Arrays implementierbar. Allerdings genügt das Ausgangsarray. Heapsort ist daher in-place.

Stabilität: durch das Vertauschen werden die Elemente unkontrollierbar vermischt, beide sind daher *nicht stabil*.

Zeitkomplexität:
Bei Selectionsort braucht die Suche nach dem jeweils größten Element $\mathcal{O}(n)$ Zeit. Insgesamt ergibt sich daher immer ein Zeitbedarf von $\mathcal{O}(n^2)$. Heapsort verbessert die Suche nach dem größten Element

auf $\mathcal{O}(\log(n))$ Zeit. D.h. die **Worst Case**-Zeit ist in $\mathcal{O}(n \log(n))$. Damit gehört Heapsort zu den schnellsten Sortierverfahren. Wenn die Daten schon umgekehrt sortiert sind, dann ist die Suche sogar konstant, so dass im **Best Case** $\mathcal{O}(n)$ ausreicht.

Online/Offline-Algorithmus: Das Hinzufügen neuer Elemente wenn der Algorithmus schon läuft ist kaum möglich. Man müsste das Verfahren dann ganz von vorne wieder anlaufen lassen.

Parallelisierung: Die nächste Vertauschungs-Operation ergibt sich erst, wenn die letzte Vertauschungs-Operation fertig ist. Daher müssen sie sequentiell durchgeführt werden.

2.6 Komplexität des vergleichsbasierten Sortierproblems

Es gibt mehrere vergleichsbasierte Sortieralgorithmen, deren Zeitkomplexität zum Sortieren von n Elementen $\mathcal{O}(n \log(n))$ ist. Die Frage ist dann:

Könnte es einen vergleichsbasierten Sortieralgorithmus geben, mit noch besserer Zeitkomplexität?

Z.B. könnte es ja einen solchen Algorithmus mit der Zeitkomplexität $\mathcal{O}(n)$ geben.

Eine der wichtigen Resultate der theoretischen Informatik ist:

Es kann keinen vergleichsbasierten Sortieralgorithmus geben mit noch besserer Zeitkomplexität als $\mathcal{O}(n \log(n))$***!***

Man kann das formal beweisen. Es gibt aber ein einfaches Argument, mit dem dieses Ergebnis einsichtig wird.

Angenommen, wir wollen z.B. eine Namensliste mit n Namen alphabetisch sortieren. Die kürzeste Form des Ergebnisses wäre Aufzählung der Positionen der sortierten Namen, z.B. $(5, 3, 10, 2, \ldots)$, d.h. zuerst kommt der 5. Name, dann der 3. Name, dann der 10. Name usw. Das Ergebnis wäre also *eine Liste von Zahlen*, völlig unabhängig, welches Sortierverfahren gewählt wurde.

Wieviele Bits braucht man, um eine Liste von n nicht-negativen Zahlen binär darzustellen?

Um *eine* beliebige Zahl zwischen 0 und n darzustellen braucht man $\lceil \log_2(n) \rceil$ Bits. Z.B. um eine Zahl zwischen 0 und 7 darzustellen, braucht man $\lceil \log_2(7) \rceil = 3$ Bits. Um n solche Zahlen darzustellen braucht man also $n\lceil \log_2(n) \rceil \in \mathcal{O}(n \log(n))$ Bits. Für jedes dieser Bits muss die Entscheidung, 0 oder 1, getroffen werden. D.h. um die Liste mit n Elementen zu sortieren, und das Ergebnis auszugeben müssen $\mathcal{O}(n \log(n))$ Entscheidungen getroffen werden. Egal welcher Algorithmus das tut, es geht nicht schneller.

3 Nicht-Vergleichsbasierte Sortieralgorithmen

Alle Sortieralgorithmen in dieser Kategorie machen zusätzliche Annahmen, die dem Algorithmus helfen, um evtl. schneller zu werden als die vergleichsbasierten Algorithmen.

3.1 Bucketsort

Wie der Name *bucket* (= Eimer) sagt, sortiert dieser Algorithmus in buckets. Viele Assistenten verwenden diesen Algorithmus nach der Klausurkorrektur, um die Klausuren nach Namen zu sortieren. Das geht folgendermaßen:
- Man legt für jeden Anfangsbuchstaben eine Position auf dem Tisch fest (das Bucket).
- Jede Klausur wird auf die Position gelegt, die dem Anfangsbuchstaben des Namens zugeordnet ist.
- Die Buckets selbst werden nach einem anderen Algorithmus, bei Klausuren z.B. mit Insertionsort sortiert.
- Am Ende werden die Buckets nach dem Alphabet aufgesammelt.

Die Effizienz dieses Algorithmus hängt davon ab, wie viele Buckets man zu Beginn definieren kann. Im Extremfall hat man so viele Buckets wie zu sortierende Elemente. Dann ist die Komplexität zum Sortieren von n Elementen $\mathcal{O}(n)$. Im anderen Extremfall hat man nur ein Bucket. Dann hängt die Komplexität von der Komplexität des Verfahrens ab, mit dem man das Bucket sortiert.

3.2 Radixsort

Diese Methode ist eine Verfeinerung von Bucketsort. Um sie anzuwenden, muss der Schlüssel, nachdem sortiert wird, folgende Bedingungen erfüllen:

- ***Die Schlüssel lassen sich in eine festgelegte Zahl von festgelegten Komponenten aufteilen.***
- ***Das Vergleichskriterium vergleicht die Schlüssel komponentenweise von links nach rechts.***

Zwei Beispiele dafür sind:
- 32-Bit Binärzahlen. Die Komponenten sind 0 und 1, und jeder Schlüssel ist 32 Bit lang,
  Das linkeste Bit ist das höchstwertige Bit.
- Zeichenketten (Worte, Namen) einer festen Maximallänge über einem festgelegten Alphabet.
  Die Komponenten sind die Buchstaben, und jeder Schlüssel ist $\leq$ der Maximallänge.
  Verglichen wird lexikographisch von links nach rechts.

Ganz grob kann man den Algorithmus folgendermaßen schildern:
- Er geht die Komponenten der Schlüssel *von rechts nach links* durch.
- Wenn k die aktuelle Position im Schlüssel ist,
  dann sortiert er die Elemente *stabil* nach dieser Komponente.

Diese sehr abstrakte Beschreibung kann man auf verschiedene Weisen konkretisieren. Wir machen es jetzt konkret für die Sortierung von 4-Bit Zahlen in Binärdarstellung. Dazu sortieren wir die Zahlen:

binär	1010	0101	1100	0111	0001
dezimal	10	5	12	7	1

Man geht jetzt von rechts nach links durch die Bits und sortiert jeweils nur nach diesem Bit, alle mit 0 nach links, und alle mit 1 nach rechts. Dazu kann man ein Hilfsarray benutzen.

Position 3:	1010	0101	1100	0111	0001
0 → links, 1 → rechts	1010	1100	0101	0111	0001

Position 2:	1010	1100	0101	0111	0001
0 → links, 1 → rechts	1100	0101	0001	1010	0111

Position 1:	1100	0101	0001	1010	0111
0 → links, 1 → rechts	0001	1010	1100	0101	0111

Position 0:	0001	1010	1100	0101	0111
0 → links, 1 → rechts	0001	0101	0111	1010	1100
dezimal:	1	5	7	10	12

Ganz wichtig bei den Umsortieroperationen ist die Beibehaltung der bisherigen Reihenfolge innerhalb der Komponenten. Das garantiert auch die Stabilität des Verfahrens als ganzes.

Für diese 4-Bit Zahlen musste man also 4 mal durch die Liste laufen und umsortieren. Allgemeiner, für k-Bit Zahlen müsste man k mal durch eine Liste von n Elemente laufen. Die Komplexität ist daher $\mathcal{O}(k \cdot n)$. Da das k in den typischen Anwendungen von Radixsort konstant ist, ist die Komplexität $\mathcal{O}(n)$, und damit schneller als der schnellste vergleichsbasierte Algorithmus.

Durch geschicktes Verschieben der Elemente im Array kann man sogar auf ein Hilfsarray verzichten, und die Umsortieroperation auch *in place* machen. Allerdings ist es nicht ohne weiteres möglich, während des Verfahrens ein neues Element hinzuzufügen. Daher ist der Algorithmus *offline*.

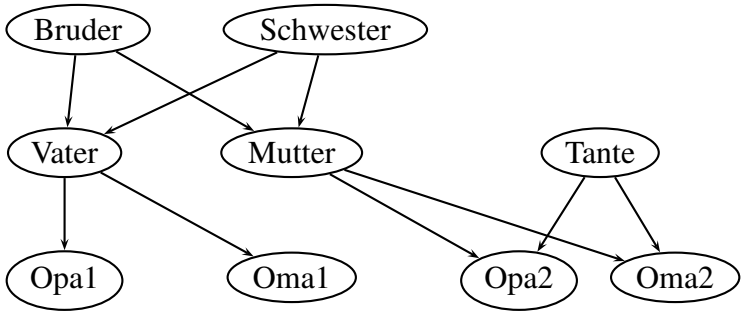
Die Positionen müssen in der Reihenfolge von hinten nach vorne durchgelaufen werden. Da die Sortierungen an Position k von den vorherigen Positionen abhängen, kann man das nicht parallelisieren. Bestenfalls könnte man die Umsortieroperationen parallelisieren.

4 Topologisches Sortieren

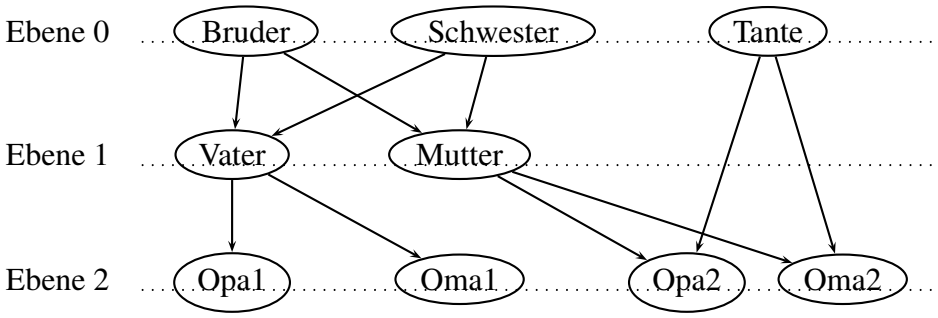
Die bisherigen Sortierverfahren erfordern eine Totalordnung auf den Elementen. Diese garantiert für jede Menge dieser Elemente eine eindeutige Reihenfolge. Es gibt jedoch auch Mengen, auf denen nur eine partielle Ordnung definiert ist. Damit sind mache, aber nicht alle Element miteinander vergleichbar.

Ein Beispiel ist die *ist-Kind-von* Beziehung. *Kind ist-Kind-von Mutter* gilt, aber weder *Bruder ist-Kind-von Schwester* noch *Schwester ist-Kind-von Bruder* gilt. Außerdem hat die *ist-Kind-von* Beziehung keine Zyklen, ansonsten könnte ja jemand sein eigener Nachfolger sein. Die Transitive Hülle von *ist-Kind-von Mutter* ist *ist-Vorfahre-von*. Sie ist eine transitive und irreflexive Ordnung, eine sog. *strenge Halbordnung*. Für solche Ordnungen ist Topologisches Sortieren möglich. Wie das aussehen

könnte illustriert folgendes Beispiel:



Dieses Gebilde ist ein *gerichteter azyklischer Graph* (DAG: Directed Acyclic Graph). Für jeden Knoten kann man darin eine *Ebene* definieren. Knoten K ohne Vorgänger, d.h. es gibt kein Objekte x mit *x ist-Kind-von K* sind auf Ebene 0. Für alle anderen Knoten ist die Ebene die *maximale* Anzahl von Kanten, über die man diesen Knoten erreichen kann. Für unser Familienbeispiel sieht das so aus:



Um eine lineare Reihenfolge herzustellen, kann man jetzt einfach die Knoten der Ebenen aneinander hängen, also erst Ebene 0, dann Ebene 1 usw. Im Beispiel ergäbe sich die Reihenfolge

(Bruder, Schwester, Tante, Vater, Mutter, Opa1, Oma1, Opa2, Oma2).

Innerhalb einer Ebene ist allerdings die Reihenfolge beliebig.

(Schwester, Bruder, Tante, Vater, Mutter, Opa1, Oma2, Opa2, Oma1)

wäre also auch eine mit der *ist-Kind-von* Beziehung kompatible sortierte Reihenfolge.

Beide garantieren die *Kompatibilitätsbedingung*:

Eine sortierte Reihenfolge R ***von Objekten ist kompatibel mit der Halbordnung*** r ***auf den Objekten gdw. falls*** $x\ r\ y$ ***gilt, dann muss*** x ***vor*** y ***in*** R ***stehen.***

Im Beispiel gilt z.B. *Tante ist-Kind-von Opa2*, und daher muss in der Sortierung *Tante* vor *Opa2* stehen.

Ein Topologischer Sortieralgorithmus für einen Halbordnung r:
Die Struktur des Algorithmus ist jetzt evident:

- Berechne für jedes Objekt seine zugeordnete Ebene
- Hänge die Objekte der Ebenen, beginnend mit Ebene 0, dann Ebene 1 usw. aneinander,

Um die Ebenen zu berechnen, startet man mit Ebene 0. Das sind alle Objekte, die keinen Vorgänger in der Relation r haben. Startend mit $k = 0$ berechnet man Ebene $k + 1$, indem man für alle Objekte der Ebene k, für deren Nachfolger in der r-Relation die Ebene $k + 1$ festlegt.

Im Familienbeispiel oben besteht die Ebene 0 aus den kinderlosen Bruder, Schwester und Tante.

Ebene 1 erhält man zunächst für diejenigen Personen y, für die es eine Person x aus Ebene 0 gibt, für die *x ist-Kind-von y* gilt. Das wären zunächst Vater, Mutter, Opa2 und Oma2.

Ebene 2 erhält man dann für diejenigen Personen y, für die es eine Person x aus Ebene 1 gibt, für die *x ist-Kind-von y* gilt. Das sind Opa1, Oma1, Opa2, Oma2.

Wichtig ist jetzt, dass die erste Zuordnung Ebene 1 für Opa2 und Oma2 mit Ebene 2 überschrieben wird.

Hierzu kann man ein Verfahren anwenden, welches ganz ähnlich zum Dijkstra-Algorithmus zur Berechnung von kürzesten Pfaden in DAGs ist. Anstelle des kürzesten Pfades berechnet man den längsten Pfad.

Für n Objekte hängt die Zeitkomplexität von der Anzahl von Relationen (Kanten) zwischen den n Objekten ab, im schlimmsten Fall sind das n^2 Kanten. Das Aneinanderhängen der Ebenen geht in linearer Zeit. Daher ist die Worst Case Zeitkomplexität des Verfahrens $\mathcal{O}(n^2)$, bzw. $\mathcal{O}(n + k)$ wenn k die Anzahl Kanten zwischen den Objekten ist.

Stichwortverzeichnis