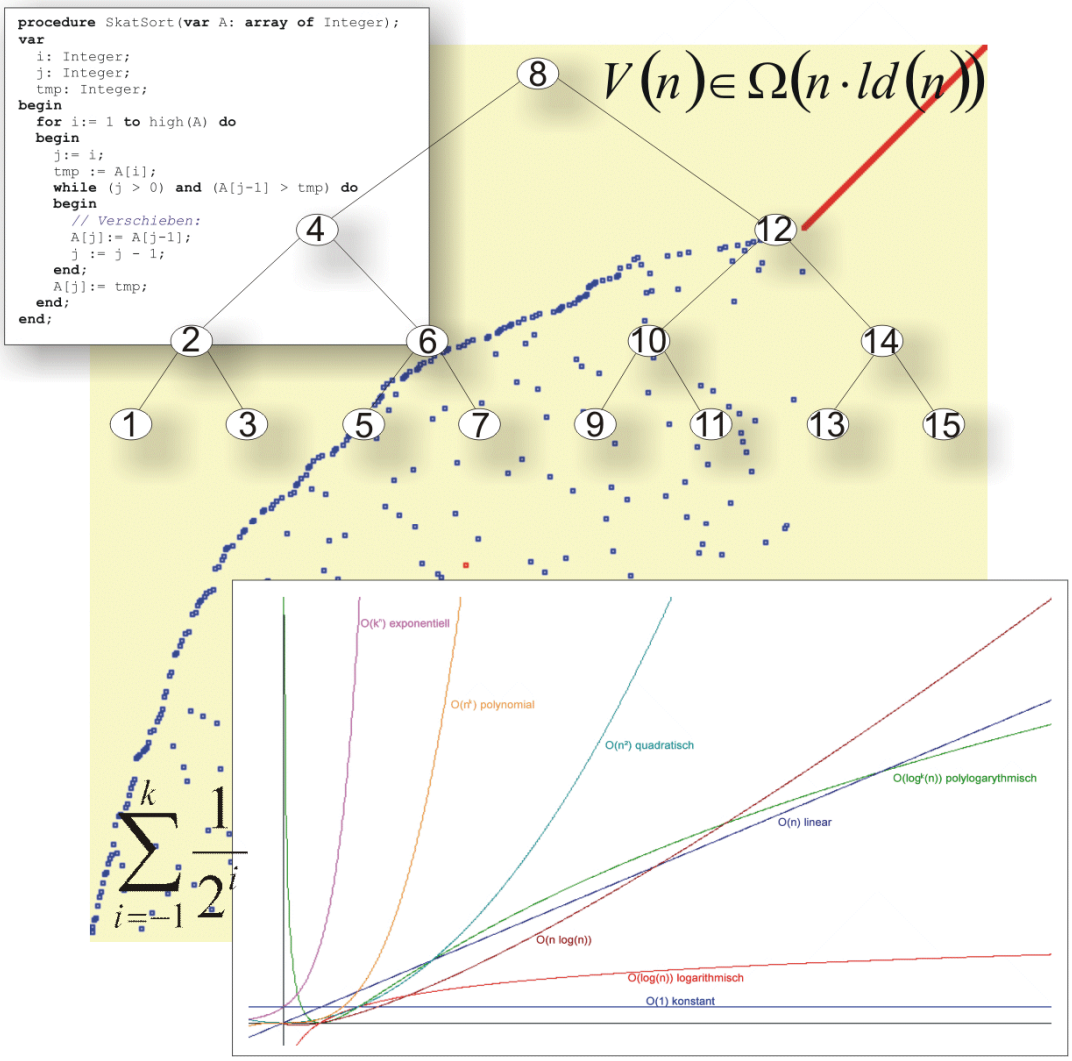
# Sortieralgorithmen unter mathematischen Gesichtspunkten

## Erweiterte Facharbeit

von Christian Rehn

23.01.06 bis 02.05.06



# Inhalt

# Einführung

Heute, im Zeitalter moderner Massenmedien, in dem fast jede erdenkliche Information nur einen Mausklick entfernt ist, in einer Zeit, in der Informationen als Ware gehandelt werden und die Technik mit atemberaubender Geschwindigkeit voranschreitet, scheint die Mathematik als Wissenschaft in den Hintergrund zu treten. Alles scheint mithilfe des Computers vorstell-, mach- und automatisierbar zu sein. Zeit scheint durch immer schneller werdende Computersysteme, Netzwerke und Supercomputer zweitrangig geworden zu sein. Ist dieser Eindruck aber gerechtfertigt? Hat Technik die Mathematik zu den Alten Griechen verbannt? Ist Zeit wirklich ein nahezu unbegrenzter Rohstoff?

Schauen wir uns nämlich einmal die heutigen Informationstechnologien genauer an, so entdecken wir, dass ohne Mathematik kaum etwas mehr funktionieren würde. Die ganze Technik, auf die sich unsere Wirtschaft, unsere Zivilisation, ja unser Alltagsleben stützt, funktioniert nur auf Basis der Mathematik. Abgesehen von den finanziellen Kreisläufen, die unsere Wirtschaft in Gang halten, werden die Wege, auf denen Informationen verbreitet und verarbeitet werden immer wichtiger. Der Computer hat die Mathematik nicht ersetzt oder verdrängt; er hat sie direkt nutzbar gemacht. Das, was unsere modernen Informationstechnologien ausmacht, ist die konkrete Anwendung der Mathematik. Und nicht nur, dass Computer auf der Basis der Mathematik funktionieren, sie bringen auch neue Aufgabenfelder, in denen die Mathematik eine tragende Rolle spielt. Ein Beispiel hierfür ist die Algorithmik. Selbst, wenn die Technik noch so schnell voranschreitet, auch, wenn sich die Computersysteme noch so schnell entwickeln und ihre Aufgaben in immer kürzerer Zeit erledigen, so nutzt der beste Computer nichts, wenn die verwendete Software, konkret: der verwendete Algorithmus, ungeeignet ist. Die Geschwindigkeit eines Computers ist nur ein konstanter Faktor, d.h. hat sich die Rechenleistung eines Computersystems verdoppelt, so halbiert sich bestenfalls die Zeit, in der eine bestimmte Arbeit erledigt wird. Durch die Wahl eines geeigneteren Algorithmus lässt sich in manchen Fällen jedoch erreichen, dass Aufgaben, die sonst Jahre gedauert hätten, in Sekundenbruchteilen erledigt sind.

Ein Anwendungsgebiet, in dem dies besonders deutlich wird, sind die Sortieralgorithmen. Tagtäglich kommen wir mit Sortiervorgängen in Berührung, ohne, dass wir dies merken: In Suchmaschinen, Bürosoftware und Datenbanken wird praktisch ständig sortiert. Bei der Internetrecherche, beim Verwalten von Terminen und Adressen, beim Telefonieren, ja selbst beim Behördengang begegnen uns sortierte Listen, sowie Sortierprozesse.

Dass die Aufgabe des Sortierens großer Datenmengen nicht zu unterschätzen ist, zeigen Untersuchungen, die besagen, dass Sortiervorgänge mehr als ein Viertel der kommerziell genutzten Rechenzeit in Anspruch nehmen[i]. Ungeachtet dessen, ob dieser Zahlenwert nun gerechtfertigt ist, kann man wohl mit gutem Recht behaupten, dass das Sortieren eine zentrale Aufgabe in der heutigen Zeit ist.

Um nun zu untersuchen, was ein Sortierverfahren von einem anderen unterscheidet, was es vielleicht sogar schneller macht, habe ich mich den Sortieralgorithmen von der mathematischen Seite aus genähert.

# 1. Die Landau-Symbole[I]

Oft ist es sehr schwierig oder sogar unmöglich eine exakte Funktion für das Laufzeitverhalten von Algorithmen zu bestimmen. Deshalb wird die Funktion durch Asymptotische Analyse angenähert, d.h. eine Funktion gesucht, die ein ähnliches Wachstumsverhalten zeigt. Zwei Funktionen f und g haben genau dann ein ähnliches Wachstumsverhalten, wenn gilt:

$$\lim_{n\to\infty}\left(\frac{f(x)}{g(x)}\right)=c \text{ mit } c\in R$$

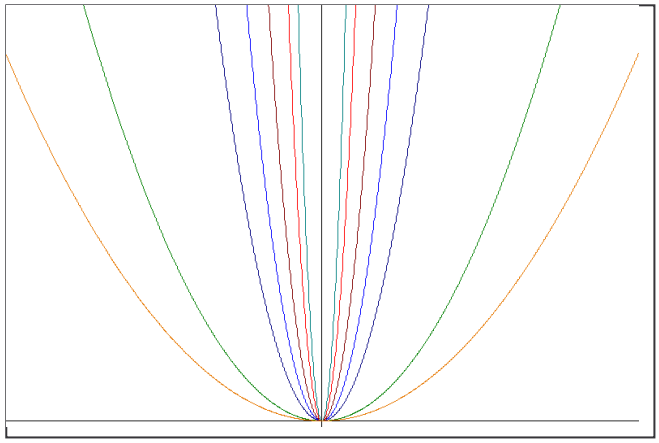


**Abb. 1:** 7 ähnlich wachsende Funktionen; alle O(n²)

Man sagt $f(x)$ ist von der Ordnung $g(x)$, wenn $f(x)$ und $g(x)$ ein ähnliches Wachstumsverhalten haben. Dabei ist $g(x)$ meist eine einfachere Funktion, sodass man Funktionen grob in solche Ordnungen einteilen kann.

Um das asymptotische Verhalten von Funktionen zu beschreiben, benutzt man die Landau-Symbole, auch O-Notation genannt. Diese bilden eine Möglichkeit, Funktionen und in unserem Fall Sortieralgorithmen grob zu klassifizieren, indem sie deren Ordnung beschreiben.

Folgende Landau-Symbole werden wir hauptsächlich verwenden:

***Asymptotisch obere Schranke***[ii]**:**
$f(x)\in O(g(x))$
Für die Funktionen $f(x)$ und $g(x)$ gilt: $f(x)\le c_0 g(x)$
mit $x>x_0$, $x_0\in R$ und $c_0\in R$

***Asymptotisch untere Schranke***
$f(x)\in \Omega(g(x))$
Für alle Funktionen $f(x)$ und $g(x)$ gilt: $f(x)\ge c_0 g(x)$
mit $x>x_0$, $x_0\in R$ und $c_0\in R$

***Asymptotisch scharfe Schranke***
$f(x)\in \Theta(g(x))$
Für die Funktionen $f(x)$ und $g(x)$ gilt:
$f(x)\in O(g(x))$ und $f(x)\in \Omega(g(x))$
Also: $c_0 g(x)\le f(x)\le c_1 g(x)$ mit $x>x_0$, $x_0\in R$, $c_0\in R$ und $c_1\in R$

Der deutsche Mathematiker **Paul Bachmann**(1837-1920) verwendete 1892 erstmals ein großes O(damals eigentlich ein großes Omikron) um eine Ordnung auszudrücken.

**Edmund Landau**(1877-1938) – ebenfalls ein deutscher Mathematiker – machte später die O-Notation bekannt. Nach ihm wird sie vor allem im deutschen Sprachraum auch Landau-Notation genannt.

**Info 1:** Paul Bachmann und Edmund Landau

Des weiteren gibt es noch folgende Landau-Symbole:

***Asymptotisch vernachlässigbar***[iii]
$f(x)\in o(g(x))$

[I] Info1: vgl. o.V.: Wikipedia – Paul Bachman, http://de.wikipedia.de/wiki/Paul_Bachmann, Stand 20.04.06, o.V.: Wikipedia – Edmund Landau, http://de.wikipedia.de/wiki/Edmund_Landau, Stand 20.04.06

Für die Funktionen $f(x)$ und $g(x)$ gilt: $\lim_{x\to\infty}\left(\frac{f(x)}{g(x)}\right)=0$

***Asymptotisch dominant***

$f(x)\in\omega(g(x))$

Für die Funktionen $f(x)$ und $g(x)$ gilt: $\lim_{x\to\infty}\left(\frac{f(x)}{g(x)}\right)\to\infty$

***Notation***

Bei der Schreibweise ist zu beachten, dass es 2 Möglichkeiten gibt, auszudrücken, dass eine Funktion $f(x)$ der Ordnung $g(x)$ ist. Sowohl das Gleichheitszeichen, als auch das Elementzeichen sind möglich. Beispiele:

$f(x)=O(1)$

$f(x)\in O(x^2)$

Obwohl sich die Notation mit dem Gleichheitszeichen teilweise eingebürgert hat, ist sie doch irreführend, da sie eigentlich impliziert, dass es sich um eine Gleichung oder wenigstens um eine Äquivalenz handelt. Dass dem aber nicht so ist, zeigt schon die Überlegung, dass $O(g(x))$ als die Menge aller Funktionen, deren Werte unter Vernachlässigung von konstanten Faktoren ab einem bestimmten $x_0$ kleiner oder gleich den Werten der Funktion $g(x)$ sind, gesehen werden kann. Entsprechendes gilt auch für andere Landau-Symbole. Demnach ist die eigentlich korrekte Schreibweise das Elementsymbol, da es sich um Mengen von Funktionen handelt. Ich werde also diese Schreibweise benutzen, trotzdem sei gesagt, dass auch andere Schreibweisen möglich sind oder zumindest häufig verwendet werden.[iv]

Des weiteren müsste eigentlich jeweils angegeben werden, welcher Grenzwert die Grundlage bildet. Eine Funktion $f(x)$ kann für $x\to\infty$ der Ordnung $O(x^2)$ sein. Für $x\to 0$ muss dies aber nicht zutreffen. Meist ergibt sich das allerdings aus dem Kontext, sodass derartige Missverständnisse eher selten sind. In unseren Fall betrachten wir nur Grenzwerte für $x\to\infty$.

***Grenzen der Landau Notation[v]***

Wenn man die Landau-Notation benutzt, sollte man sich auch über deren Grenzen bewusst sein. Diese Notationsweise gibt zwar eine grobe Klassifikation vor, jedoch kann ein gegebener Wert auch täuschen. Es gibt mehrere Gründe, warum man mit der Interpretation von Werten unter Benutzung der Landau-Symbole vorsichtig sein sollte:

1. Es handelt sich meist um eine obere oder untere Schranke. Inwieweit diese wirklich erreicht wird oder erreicht werden kann, ist nicht gesagt. Es steht nur fest, dass es niemals größere bzw. kleinere Werte geben kann.
2. In der Praxis kann es sein, dass die Eingabedaten, die den günstigsten oder ungünstigsten Fall hervorrufen praktisch nicht auftreten(vgl. QuickSort Kapitel 3.2.).
3. Es wird jeweils der Grenzwert für $x\to\infty$ betrachtet, was bedeutet, dass unter Umständen nur für sehr große Eingabemengen ein Algorithmus höherer Ordnung schlechter ist, als einer niedrigerer Ordnung.
4. Konstante Faktoren werden nicht berücksichtigt. Das kann dazu führen, dass zwei Algorithmen gleicher Ordnung von sehr unterschiedlicher Effizienz sein können. Ein konstanter Faktor 10 kann bedeuten, dass ein Algorithmus 3 Sekunden und ein anderer für die selbe Arbeit 30 Sekunden benötigt.

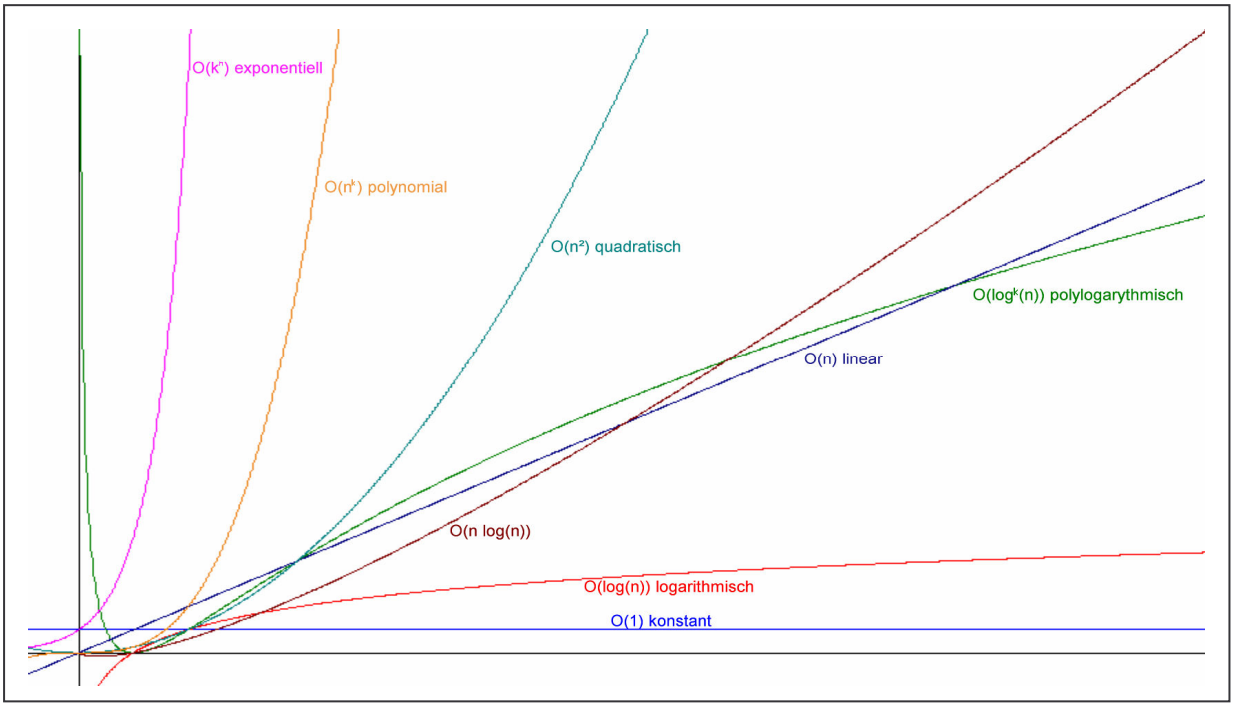



**Abb. 2:** Funktionen unterschiedlicher Ordnung

***Vorteile der Landau-Notation***

Auch, wenn die Landau-Notation ihre Grenzen hat und dadurch Informationen verloren gehen, hat sie einige Vorteile, weswegen sie auch häufig verwendet wird um Algorithmen zu klassifizieren. Wenn man sich also der Grenzen der Landau-Notation bewusst ist, so kann man die Vorteile der Landau-Symbole nutzen:

1. Eine einfache Klassifizierung des Laufzeitverhaltens von Algorithmen wird möglich.
2. Diese ist unabhängig von verwendeter Hardware und Betriebssystem.
3. Auch die Implementierung des einzelnen Algorithmus spielt keine Rolle. Es wird nur das Verfahren an sich charakterisiert, nicht die Umsetzung in einer Programmiersprache.
4. Die einzige Größe, auf die die Abhängigkeit reduziert wird, die so genannte Problemgröße, ist die Menge der Eingabedaten.
5. Oft ist es nur mit großem Aufwand oder gar unmöglich eine genaue Funktion für die Laufzeit eines Algorithmus anzugeben. Somit ist die Landau-Notation manchmal die einzige Möglichkeit darüber etwas auszusagen.

***Anwendung:***

Unter anderem werden die Landau-Symbole, wie schon erwähnt, genutzt um die Laufzeit von Algorithmen zu klassifizieren. Somit lässt sich eine überschaubare Menge an Geschwindigkeitsklassen definieren. Da die Eingebemenge jeweils eine natürliche Zahl darstellt, d.h. da es sich genau genommen um eine Folge und keine Funktion handelt, verwendet man statt dem allgemeinen x, als Funktionsvariable ein n um dies anzudeuten. Die wichtigsten Ordnungen sind hier noch einmal zusammengefasst:

| **Klase** | **Name** | **Bemerkungen** |
|---|---|---|
| $O(1)$ | konstant; | Unabhängig von der Eingebemenge |
| $O(\log(n))$ | logarithmisch | Die Basis ist hier nebensächlich, da sie nur einen konstanten Faktor ausmacht. Tritt z.B. bei binärer Suche auf. |
| $O(\log^k(n))$ | polylogarithmisch | Auch hier ist die Basis unerheblich.; k>1 |
| $O(n)$ | linear | |
| $O(n\log(n))$ | „n log n“ | Auch hier ist die Basis unerheblich |

| | | |
|---|---|---|
| $O(n^2)$ | quadratisch | |
| $O(n^k)$ | polynomial | k>1 |
| $O(k^n)$ | exponentiell | k>1 |
| $O(n!)$ | „n Fakultät“ | |
| $O(n^n)$ | superexponentiell | |

***Beispiel:***

Bestimmung der Ordnung von $f(n)=\frac{1}{3}n\left(n+\frac{1}{2}n^2-3\right)^2$.

1. Vereinfachen:

$$\begin{aligned}
f(n) &= \frac{1}{3}n\left(n+\frac{1}{2}n^2-3\right)^2 \\
&= \frac{1}{3}n\left(n+\frac{1}{2}n^2-3\right)\left(n+\frac{1}{2}n^2-3\right) \\
&= \frac{1}{3}n\left(n^2+\frac{1}{2}n^3-3n+\frac{1}{2}n^3+\frac{1}{4}n^4-\frac{3}{2}n^2-3n-\frac{3}{2}n^2+9\right) \\
&= \frac{1}{3}n\left(\frac{1}{4}n^4+n^3-2n^2-9n+9\right) \\
&= \frac{1}{12}n^5+\frac{1}{3}n^4-\frac{2}{3}n^3-3n^2+3n
\end{aligned}$$

2. Bestimmung des asymptotisch dominanten Terms:

$$\lim_{x\to\infty}\left(\frac{\frac{1}{3}n^4-\frac{2}{3}n^3-3n^2+3n}{\frac{1}{12}n^5}\right)=0$$

$\frac{1}{3}n^4-\frac{2}{3}n^3-3n^2+3n \in o\left(\frac{1}{12}n^5\right)$[I] bzw. $\frac{1}{12}n^5 \in \omega\left(\frac{1}{3}n^4-\frac{2}{3}n^3-3n^2+3n\right)$

3. Bestimmung der oberen Schranke und damit der „Ordnung“:

$f(n) \in O(n^5)$ und damit polynomial(5. Grades).

[I] Der konstante Faktor 1/12 ist hier überflüssig und dient nur der Verdeutlichung

# 2. Sortieren

## 2.1. Sortieren – wie macht man das? [vi]

***Problemstellung:*** Gegeben ist eine Folge beliebiger Objekte $a_1, a_2, ..., a_n$ mit $n > 0$, deren Elemente sich durch ein eindeutiges Kriterium unterscheiden. Sortieren ist das Umordnen dieser Folge in eine Folge $a_{j_1}, a_{j_2}, ..., a_{j_n}$, bei der gilt:
$a_{ij} \leq a_{ij+1}$ (*aufsteigende Sortierung*) bzw.: $a_{ij} \geq a_{ij+1}$ (*absteigende Sortierung*).[vii]

Bevor man an die Auswahl eines geeigneten Verfahrens zur Lösung dieses Problems geht, stellt sich zuerst einmal die Frage, wie man überhaupt etwas sortiert. Welche Ideen stecken hinter den einzelnen Sortieralgorithmen?
Im Grunde genommen gibt es nur drei Arten interner Sortieralgorithmen, die auf drei zentralen Ideen basieren: Sortieren durch Auswählen, durch Einfügen und durch Austauschen. Die meisten Verfahren basieren auf einer dieser Ideen und bieten nur unterschiedliche Ansätze diese Grundideen umzusetzen. Trotzdem werden wir noch sehen, dass diese Ansätze den jeweiligen Algorithmen ihre charakteristischen Eigenschaften verleihen.[viii]

***Sortieren durch Auswählen***
Das nächste passende Element wird gezielt aus der Menge der noch zu sortierenden Elemente ausgewählt(d.h. gesucht) und an die gewünschte Position gesetzt.
Beispiele:

- Min-Sort/Selection-Sort
- HeapSort
- ...

***Sortieren durch Einfügen***
Das nächste Element wird an der passenden Stelle im bereits sortierten teil der Liste eingefügt.
Beispiele:

- Straight insertion/Skat-Sort
- Binary insertion
- ShellSort
- ...

***Sortieren durch Austauschen:***
Zwei Elemente werden vertauscht, wenn ihre relative Position zueinander der gewünschten Ordnungsrelation widerspricht.
Beispiele:

- Bubble-Sort
- Shaker-Sort
- QuickSort
- ...

## 2.2. Merkmale von Sortieralgorithmen

Will man etwas untersuchen, ganz gleich was es ist, so benötigt man bestimmte Kriterien, nach denen man unterscheiden, bewerten und klassifizieren kann. Um Algorithmen zu klassifizieren, benötigen wir also auch bestimmte Kenngrößen, nach denen sich die zu untersuchenden Algorithmen unterscheiden und die diese charakterisieren. Da es das Ziel eines Algorithmus' ist, ein Problem mit möglichst geringem Aufwand zu lösen, liegt es nahe, den Aufwand als Kriterium zu nehmen. Die einfachste Form der Aufwandsermittlung wäre wohl zweifellos seine Laufzeit, d.h. die Zeit, die ein Algorithmus unter bestimmten Voraussetzungen benötigt um ein Problem zu lösen.

Hierbei ergeben sich allerdings schwerwiegende Probleme, denn die Laufzeit eines Algorithmus hängt stark von der verwendeten Hardware, dem Betriebssystem und sonstigen Faktoren ab. Somit ist ein Vergleich von Algorithmen allein auf Basis der Laufzeit nur eingeschränkt möglich, da sich die Laufzeit nur experimentell herausfinden lässt, also mathematisch nicht zu fassen ist und somit ein Vergleich schwierig wird, da die Bedingungen bei den Messungen immer exakt gleich sein müssten, was schwer zu gewährleisten ist.

Man muss also für jede Art von Algorithmus separat Kriterien festlegen, die den Aufwand der Algorithmen charakterisiert. Bei Sortieralgorithmen wäre das die Anzahl der Vergleiche, sowie die Anzahl der Zuweisungen. Je nach den gegebenen Bedingungen kann es nämlich sein, dass das Vergleichen oder das Zuweisen „teuer", d.h. aufwändig, ist. In manchen Fällen muss bei der Untersuchung der Laufzeit zwischen bast-case, worst-case und average-case, also dem besten, schlechtesten und dem durchschnittlichen Fall unterschieden werden. Die meisten Algorithmen verhalten sich nämlich je nach Eingabedaten anders, sodass ein Verfahren bei bestimmten Eingabedaten schneller oder langsamer ist, als bei anderen.

Des weiteren spielen noch andere Eigenschaften, wie z.B. die Stabilität, bei der Auswahl des geeigneten Algorithmus eine Rolle. So kann es sein, dass in einem Fall der eine und im anderen ein anderer Algorithmus vorzuziehen ist.

Folgende Eigenschaften charakterisieren also einen Sortieralgorithmus:

- Aufwand(Laufzeit) in Abhängigkeit der Anzahl zu sortierender Elemente
  - Anzahl Vergleiche
  - Anzahl Zuweisungen
- Stabilität
- Simplizität(d.h. Aufwand der Implementierung)
- Iterativität/Rekursivität
- Speicherverbrauch(in-place/out-of-place, Rekursion)

Bei der Untersuchung von Sortieralgorithmen werde ich mich auf einfache, iterative, vergleichsbasierte in-place-Verfahren konzentrieren, aber auch QuickSort(als einen schnellen) und BogoSort(als einen unbenutzbaren Algorithmus) vorstellen.

# 2.3. Geschwindigkeit von Sortieralgorithmen[ix]

## *2.3.1. Minimale Anzahl Vergleiche[x]*

Wenn man Sortieralgorithmen untersucht, so fragt man sich, bis zu welchen Grad man Sortierverfahren, die ihre Informationen lediglich aus dem Vergleich von Elementen ziehen, verbessern kann. Denn, dass es eine solche Grenze geben muss, zeigt allein der Gedanke, dass man nicht gänzlich ohne Aufwand sortieren kann.

**Behauptung:**
Um eine Menge von n Objekten zu sortieren, benötigt ein Sortierverfahren, das seine Informationen ausschließlich aus Vergleichsoperationen zwischen den Objekten erhält, im schlechtesten Fall[I] mindestens $\Omega(n \cdot ld(n))$ Vergleiche.

**Beweis:**
Ein allgemeiner Sortiervorgang lässt sich durch einen binären Entscheidungsbaum darstellen, bei dem die Blätter den Permutationen der Objektmenge entsprechen. Die Anzahl der nötigen Vergleiche entspricht dann der Baumhöhe. Die Anzahl der Permutationen beträgt n!. Die maximale Anzahl der Blätter(Baum ist vollständig) eines binären Baumes der Höhe k beträgt $2^k$. Somit gilt:

$$n! \leq 2^{V(n)}$$

$$ld(n!) \leq V(n)$$

Mit dem 1. Logarithmengesetz folgt:

$$V(n) \geq ld(n!) = \sum_{i=1}^{n} ld(i)$$

Durch Weglassen der ersten n/2 Summanden[II] (nötig für nächsten Schritt) ergibt sich:

$$\sum_{i=1}^{n} ld(i) \geq \sum_{i=\frac{n}{2}}^{n} ld(i)$$

Mit Abschätzung nach unten:

$$\sum_{i=\frac{n}{2}}^{n} ld(i) \geq \frac{n}{2} ld\left(\frac{n}{2}\right)$$

$$= \frac{n}{2}(ld(n)-1) \in \Omega(n \cdot ld(n))$$

Somit gilt:

$$V(n) \in \Omega(n \cdot ld(n))$$

**Logarithmus dualis**
In Analysis und Physik wird der natürliche Logarithmus(zur Basis e(Eulersche Zahl ~2,72)) häufig verwendet und mit $\ln$ abgekürzt.
In der Informatik tritt der Logarithmus zu Basis 2 sehr häufig auf, weshalb man ihn $ld$ (manchmal auch $\lg$) abkürzt.

**Info 2:** Logarithmus dualis

**Summen und Reihen**
$\sum_{i=1}^{n} i$ ist eine abkürzende Schreibweise für $1+2+...+(n-1)+n$.
*(siehe Anhang C)*

**Info 3:** Summen und Reihen

[I] Im besten Fall ist die Menge bereits sortiert und dies muss nur noch getestet werden, was $n-1 \in \Omega(n)$ Vergleiche erfordert.
[II] Dies ist möglich, da dadurch die Ordnung der Funktion nicht geändert wird

## 2.3.2. Minimale Anzahl Zuweisungen

Auch die minimalen Anzahl an Zuweisungen kann man bestimmen. Hierbei beschränken wir uns auf die Zuweisungen, die direkt die Daten betreffen, d.h. Zuweisungen an ein Element des Datenfeldes oder Zuweisungen an eine lokale Variable mit einem Wert aus dem Array. Der Hintergrund ist der, dass Zuweisungen, die nicht mit den eigentlichen Daten zusammenhängen(z.B. Zählvariablen) leicht vom Compiler optimiert werden können, indem diese z.B. in einem Prozessorregister abgelegt werden. Der Einfluss auf die Laufzeit des Algorithmus ist in so einem Fall im Vergleich zu der Anzahl Vergleiche und Zuweisungen, die das Datenfeld betreffen vernachlässigbar klein.

***Best-case:***
Die minimale Anzahl Zuweisungen im best-case beträgt 0. Dies kommt genau dann vor, wenn die zu sortierende Menge bereits sortiert ist. Dann sind nämlich keine Zuweisungen mehr nötig, was aber nicht heißt, dass manche Sortieralgorithmen trotzdem Zuweisungen vornehmen.

***Worst-case:***
Im schlechtesten Fall ist keines der zu sortierenden Elemente am richtigen Platz. In diesem Fall muss jedes Element mindestens einmal an die richtige Stelle gesetzt werden. Somit beträgt die minimale Anzahl Zuweisungen $n \in \Omega(n)$. Dies gilt aber nur für Externe Sortierverfahren. Bei internen ist jede Zuweisung ein Tausch zweier Elemente, der nur über einen temporären Hilfsspeicher geschehen kann:

```
// a und b vertauschen
t := a;
a := b;
b := t;
```

Man benötigt also für jeden Tausch 3 Zuweisungen. Hat man n-1 Elemente an die richtige Stelle gesetzt, so ergibt sich die Position des letzen automatisch, das bedeutet die minimale Anzahl Zuweisungen im worst-case beträgt $3(n-1) \in \Omega(n)$.

***Average-case:***
Je nach gegebener Situation kann der Durchschnittsfall unterschiedlich aussehen. In manchen Fällen wird oft die Sortierung umgekehrt(auf- und absteigende Sortierung) oder es werden neue Elemente ans Ende gesetzt, was jeweils bedeutet, dass sich sehr wenige(oder gar keine) Elemente an der richtigen Stelle befinden. Somit träte der worst-case relativ häufig auf. Es lässt sich allerdings auch allgemein zeigen, dass auch hier die Anzahl der Zuweisungen $Z(n) \in \Omega(n)$ beträgt:
Betrage der Anteil der sich an der richtigen Stelle befindenden Elemente k%, so beträgt die Anzahl der sich an falscher Stelle befindenden Elemente und damit der nötigen Vertauschungen:

$$n\left(\frac{100 - k}{100}\right)$$

Somit beträgt die Anzahl der nötigen Zuweisungen bei einem in-place-Verfahren:

$$Z(n) = 3n\left(\frac{100-k}{100}\right)$$
$$= n\left(\frac{300-3k}{100}\right)$$

Da $\left(\frac{300-3k}{100}\right)$ ein konstanter Faktor ist, gilt auch hier:
$Z(n) \in \Omega(n)$.

## 2.4. Optimieren von Sortieralgorithmen

Neben den verschiedenen Sortieralgorithmen gibt es auch unterschiedliche Versionen ein und des selben Sortieralgorithmus. Diese abgewandelten oder optimierten Algorithmen basieren meist auf dem selben Prinzip und ändern nur kleine Teile, was aber unter Umständen bedeutende Verbesserungen bewirken kann. Je nach Algorithmus und Einsatzgebiet lassen sich somit Algorithmen unterschiedlich optimieren.

Die verschiedenen Arten der Optimierung lassen sich folgendermaßen unterteilen:

- Reduzierung unnötiger Vergleiche/Zuweisungen
- Benutzung anderer Teilalgorithmen
- Anpassen an häufiger vorkommende Spezialfälle
- Anpassen an gegebene Datenstruktur

Ein Beispiel für eine Optimierung durch Reduzierung unnötiger Vergleiche wäre BubbleSort(siehe Kapitel 3.1.2.). Dort kann die Schleife abgebrochen werden, sobald keine Vergleiche mehr nötig sind. Diese Art der Optimierung ist nur bei manchen Algorithmen möglich und bewirkt erfahrungsgemäß nur eine geringe Verbesserung.

Bei InsertionSort z.B. kann man durch Verwendung des binären Suchalgorithmus statt des sequentiellen, dessen Laufzeit verbessern. Da sich durch eine solche Anpassung der Algorithmus nicht unwesentlich ändert, verwendet man für die einzelnen Versionen manchmal auch unterschiedliche Namen. InsertionSort nennt man, bei Verwendung der sequenziellen suche auch „straight insertion“ oder „SkatSort“ und mit binärer Suche „binary insertion“.

Wird eine Liste z.B. häufig umgekehrt sortiert(aufsteigend - absteigend), so kann man manche Sortieralgorithmen so an die gegebenen Umstände anpassen, dass diese häufigen Spezialfälle besser erledigt werden.

Eine weitere Art der Optimierung ist die Anpassung an die gegebene Datenstruktur. Felder(Arrays) erlauben z.B. einen schnellen Zugriff auf jedes Element, sind aber langsam, wenn es darum geht Elemente einzufügen oder das Feld zu vergrößern[I]; verkettete Listen hingegen verhalten sich anders. Hier ist der Zugriff auf ein bestimmtes Element langsam, jedoch lassen sich leicht Elemente löschen, einfügen oder ganze Abschnitte verschieben[II]. Anpassungen die den Eigenschaften der verwendeten Datenstruktur Rechnung tragen betreffen oft nur die Implementierung und nicht den eigentlichen Algorithmus, jedoch können auch diese sehr wirksam sein.

Bei manchen Algorithmen lassen sich diese Optimierungen gut zeigen und auch bestimmen, inwieweit es sich hier um eine Verbesserung handelt. Bei der Untersuchung der Optimierungsmöglichkeiten werde ich mich hauptsächlich auf die ersten beiden Möglichkeiten konzentrieren, da diese direkt den Algorithmus beeinflussen.

---

[I] Das ist dadurch begründet, dass diese Datenstruktur darauf beruht, dass alle Elemente eines Feldes in der gegebenen Reihenfolge nebeneinander im Speicher liegen. Möchte man also auf ein bestimmtes Element zugreifen, kann man dessen Position im Speicher leicht ausrechnen, das Verschieben(oder Löschen) eines Elementes bewirkt aber immer eine Verschiebung der nachfolgenden, was aufwändig ist.

[II] Der Grund hierfür ist die andere Datenorganisation. Jedes Element erhält einen Verweis(Zeiger) auf das nächste. Will man ein Element löschen oder verschieben, so muss man nur diese Verweise ändern; der Zugriff auf ein Element aber ist weitaus aufwendiger, da die Verweise aller vorangehenden Elemente aufgelesen werden müssen.

# 3. Laufzeituntersuchung von Sortieralgorithmen

# 3.1. Elementare Sortierverfahren

## 3.1.1. MinSort/SelectionSort[xi]

SelectionSort

| **Typ** | elementar | | |
|---|---|---|---|
| **Eigenschaften** | iterativ, in-place, stabil, sortierter Bereich | | |
| **Simplizität:** | sehr einfach | | |
| | | | |
| **Laufzeit:** | **best-case** | **average-case** | **worst-case** |
| **Vergleiche** | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| **Zuweisungen** | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |

**Idee:**
SelectionSort ist wohl das einfachste Sortierverfahren. Die Idee ist denkbar einfach: „Suche das kleinste Element und setze es an den Anfang. Suche dann das nächst kleinere und setze es dahinter. Fahre so fort, bis das das letzte Element an seinem Platz ist.“ Da dieses Verfahren so offensichtlich ist, wird es auch sehr häufig benutzt und zwar von uns Menschen. Intuitiv benutzen wir normalerweise dieses Verfahren. Aber, wie effizient sortiert der Mensch im Allgemeinen?

**Name:**
Der Name SelectioSort(manchmal auch SelectSort) lässt sich leicht erklären, denn er beschreibt genau das, was der Algorithmus tut: Das nächste Element auswählen(eng: (to) select *auswählen*; eng: selection *Auswahl*). Manchmal wird dieser Algorithmus auch MinSort genannt, was darauf zurückzuführen ist, dass – beim Aufsteigenden Sortieren – immer das kleinste Element(d.h. das Minimum) ausgewählt wird.

```
procedure SelectionSort(var A: array of Integer);
var
  i: Integer;
  k: Integer;
  min: Integer;
  tmp: Integer;
begin
  for i := 0 to high(A)-1 do
  begin
    // kleinstes Element der (Rest)Menge bestimmten
    min := i;
    for k := i+1 to high(A) do
    begin
      if A[k] < A[min] then
      begin
        min := k;
      end;
    end;
    // Austauschen:
    tmp := A[i];
    A[i] := A[min];
    A[min] := tmp;
  end;
end;
```

### *Vergleiche:*

**Behauptung:**
SelectionSort benötigt zum Sortieren $V(n) \in \Theta(n^2)$ Vergleiche.

**Beweis[xii]:**
Es gibt in diesem Algorithmus genau eine Stelle an der ein Vergleich ausgeführt wird. Dieser Befindet sich in der inneren Schleife. Somit ist die Anzahl der Vergleiche gleich der Anzahl der inneren Schleifendurchgänge. Beim ersten Durchlauf der äußeren Schleife(Finden des kleinsten Elementes) sind dies n-1, beim zweiten(Finden des zweitkleinsten Elementes), n-2 usw.:

$$V(n) = (n-1) + (n-2) + ... + 2 + 1$$

$$= \sum_{i=1}^{n-1} i$$

Mit der Gauß'schen Summenformel:

$$\frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

$$\frac{n^2}{2} - \frac{n}{2} \in \Theta(n^2)$$

Somit gilt:

$$V(n) \in \Theta(n^2)$$

## *Zuweisungen:*

**Behauptung:**
SelectionSort benötigt zum Sortieren $Z(n) \in \Theta(n)$ Zuweisungen.

**Beweis:**
Eine Vertauschung erfolgt bei jedem Durchlauf der äußeren Schleife. Da diese n-1 Mal durchlaufen wird(`for i := 0 to high(A)-1 do`), beträgt die Anzahl der Vertauschungen:

$$n-1$$

Und damit die Anzahl der Zuweisungen:

$$Z(n) = 3(n-1)$$

$$= 3n - 3$$

$$3n - 3 \in \Theta(n)$$

Somit gilt:

$$Z(n) \in \Theta(n)$$

**Optimierungen:**
SelectionSort nimmt in der Original-Variante unnötige Zuweisungen vor, d.h., wenn ein Element bereits an der richtigen Stelle ist, wird es mit sich selbst vertauscht. Überprüft man nun diesen Fall, so erreicht SelectionSort auch im best- bzw. „average"-case, die untere Schranke. Der Beweis hierfür deckt sich mit dem in Kapitel 2.3.2.

```
// wenn nötig: Austauschen:
if I <> min then
begin
  tmp := A[i];
  A[i] := A[min];
  A[min] := tmp;
end;
```

Erkauft wird dieser Vorteil mit einem weiteren Vergleich pro äußerem Schleifendurchlauf. Da dieser aber das Datenfeld nicht betrifft, ist der Geschwindigkeitsverlust vernachlässigbar klein.

**Ergebnis:**
SelectionSort ist in Bezug auf die Vergleiche eher langsam. Es werden keine besonderen „Tricks“ angewandt, d.h. die Informationen, die die beim ersten Durchgang gesammelt wurden, dienen ausschließlich dazu das nächste Element zu bestimmen und werden sonst nicht weiter verwendet.
Betrachtet man jedoch die gemachten Zuweisungen, so erkennt man, dass SelectionSort hier sehr sparsam ist und an die untere Schranke(siehe Kapitel 2.3.2.) herankommt. Nur werden auch im besten Fall 3n Zuweisungen getätigt, selbst, wenn diese unnötig sind. Durch die erwähnte Optimierung wird auch in diesem Fall die untere Schranke erreicht.
Das zeigt auch, dass SelectionSort vollkommen unabhängig von der ursprünglichen Anordnung der Eingabedaten arbeitet. Alleine die Eingabemenge ist entscheidend. Man kann also eine scharfe Schranke( $\Theta$ ) angeben.
Der Vorteil von SelectionSort ist, wie schon erwähnt, dass die Anzahl der Zuweisungen minimiert wird. Wenn also eine Zuweisung im Vergleich zu einem Vergleich „teuer“ ist(wenn z.B. die Datensätze relativ groß sind), so ist SelectionSort von Vorteil. Außerdem ist SelectionSort stabil, was – insbesondere bei komplexen Algorithmen – nicht immer der Fall ist.

## 3.1.2. BubbleSort[xiii]

BubbleSort

| **Typ** | elementar | | |
|---|---|---|---|
| **Eigenschaften** | iterativ, in-place, stabil, sortierter Bereich | | |
| **Simplizität:** | sehr einfach | | |
| | | | |
| **Laufzeit:** | **best-case** | **average-case** | **worst-case** |
| **Vergleiche** | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| **Zuweisungen** | 0 | $\Theta(n^2)$ | $O(n^2)$ |

**Idee:**
Durchlaufe das zu sortierende Feld und vertausche dabei, wenn nötig, benachbarte Elemente. Nach $n-1$ Durchläufen ist das Datenfeld sortiert, da ein Element, nach a Vertauschungen(a = Abstand eines Elementes von seiner Position im sortierten Feld) sich an der richtigen Stelle befindet. Da größte Distanz in einem Datenfeld $a=(n-1)$ gleich der Anzahl der Schleifendurchgänge ist, befindet sich selbst im schlimmsten Fall(Array ist genau umgekehrt sortiert) jedes Element auf seinem Platz.

**Name:**
Der Name BubbleSort, rührt von der Betrachtung her, dass man sich ein Element als Luftblase(eng. bubble *Blase*) vorstellen kann. Diese Luftblasen steigen solange auf, bis sie von einer größeren Blase aufgehalten werden, die dann ihrerseits aufsteigt.

```
procedure BubbleSort(var A: array of Integer);
var
  i: Integer;
  k: Integer;
  tmp: Integer;
begin
  for i:=0 to high(A)-1 do
  begin
    for k:=0 to high(A)-1 do
    begin
      if A[k] > A[k+1] then
      begin
        // Austauschen:
        tmp := A[k];
        A[k] := A[k+1];
        A[k+1] := tmp;
      end;
    end;
  end;
end;
```

### *Vergleiche*

**Behauptung:**
BubbleSort hat in Bezug auf die Anzahl der Vergleiche eine Laufzeit von $V(n) \in \Theta(n^2)$.

**Beweis:**
Es wird genau ein Vergleich ausgeführt und zwar in der inneren Schleife. Die äußere Schleife wird n-1 mal ausgeführt, die innere ebenfalls. Die Anzahl der Verbleiche beträgt also:

$V(n) = (n-1)(n-1) = (n-1)^2 = n^2 - 2n + 1$
$n^2 - 2n + 1 \in \Theta(n^2)$
Somit gilt:
$V(n) \in \Theta(n^2)$

## *Zuweisungen:*

**Behauptung:**
BubbleSort benötigt im besten Fall 0 und im schlechtesten, sowie im durchschnittlichen $Z(n) \in O(n^2)$ bzw. $Z(n) \in \Theta(n^2)$ Vergleiche

***Best-case:***
Im besten Fall ist das Array bereits sortiert. Da BubbleSort vor jedem Tausch prüft, ob dieser nötig ist, wird, wenn kein Tausch nötig ist, auch keine Zuweisung ausgeführt.

***Worst-case:***
Im schlechtesten Fall ist das Array umgekehrt sortiert. Dann muss zuerst das größte Element, welches sich ganz vorne befindet nach hinten gebracht werden. Dies erfordert $n-1$ Vertauschungen(innere Schleife). Bein zweiten Durchlauf wird das zweitgrößte Element, das sich nun vorne befindet an die vorletzte Stelle gebracht. Das zweitgrößte Element wird dann zwar noch mit dem größten(letzten) verglichen, jedoch nicht mehr ausgetauscht. Die Positionierung des zweitgrößten Elementes benötigt also $n-2$ Vertauschungen, usw.

Anzahl der Vertauschungen:
$$(n-1)+(n-2)+\ldots+2+1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$
Anzahl der Zuweisungen:
$$Z(n) = 3\left(\frac{n^2}{2} - \frac{n}{2}\right) = \frac{3}{2}n^2 - \frac{3}{2}n \in O(n^2)$$

***Average-case:***
Die maximale Anzahl an Vertauschungen beträgt $(n-1)+(n-2)+\ldots+2+1$. Angenommen k% dieser Vertauschungen werden nicht durchgeführt, dann reduziert sich diese Anzahl auf:
$$\frac{100-k}{100}\left((n-1)+(n-2)+\ldots+2+1\right) = \frac{100-k}{100}\sum_{i=1}^{n-1} i$$
$$= \frac{100-k}{100} \cdot \frac{(n-1)n}{2}$$
$$= \frac{100-k}{100} \cdot \frac{(n^2-n)}{2}$$

Die Anzahl der nötigen Zuweisungen beträgt dann:
$$Z(n) = 3\left(\frac{100-k}{100} \cdot \frac{(n^2-n)}{2}\right) = \frac{300-3k}{200} \cdot (n^2-n) \in \Theta(n^2)$$

**Optimierungen:**

BubbleSortführt in der Originalvariante viele unnötige Vergleiche durch. Nehmen wir einmal an, wir haben ein bereits sortiertes Array. BubbleSort würde trotzdem $(n-1)^2$ Vergleiche vornehmen, auch, wenn bereits beim ersten Durchlauf klar wäre, dass das Feld bereits sortiert ist. Auf dieser Überlegung fußt die erste Optimierungsmöglichkeit: Prüft man nämlich, ob bei einem Durchgang keine Vertauschungen vorgenommen wurden, was genau dann passiert, wenn das Array sortiert ist, kann man die Schleife vorzeitig unterbrechen, da bei allen weiteren Schleifendurchläufen nichts mehr vertauscht würde.
Eine weitere Optimierungsmöglichkeit zeigt sich, wenn man sich genau ansieht, was bei jedem Durchlauf geschieht. Wie schon bei der Untersuchung des worst-case bei den Zuweisungen angedeutet, wird bei jedem Durchlauf der äußeren Schleife das nächste Element an die richtige Stelle gebracht(dass dabei noch weitere Vertauschungen stattfinden ist für unsere Momentane Überlegung nebensächlich). Ähnlich wie bei SelectionSort wird also zuerst das größte Element bestimmt und ans Ende der Liste gesetzt. Danach das zweitgrößte an seine Stelle. Auch, wenn klar ist, dass beim zweiten Durchgang das zweitgrößte Element gesucht wird, wird es dennoch mit dem größten verglichen. Durch eine weitere Optimierung(Anpassung der Durchläufe der inneren Schleife) lassen sich auch hier unnötige Vergleiche einsparen. Somit sähe unser verbesserter BubbleSort-Algorithmus so aus:

```
procedure BubbleSort(var A: array of Integer);
var
  i: Integer;
  k: Integer;
  tmp: Integer;
  done: Boolean;
begin
  for i:=0 to high(A)-1 do
  begin
    done := True;
    for k:=0 to high(A) - 1 - i do // unnötige Vergleiche auslassen
    begin
      if A[k] > A[k+1] then
      begin
        // Austauschen:
        tmp := A[k];
        A[k] := A[k+1];
        A[k+1] := tmp;
        done := False; // Vertauschung wurde vorgenommen
      end;
    end;
    if done then // keine Vertauschung → fertig
      exit;
  end;
end;
```

Was aber hat unsere Verbesserung bewirkt? Ist BubbleSort durch unsere Optimierung merklich schneller geworden oder war dies nur eine marginale Verbesserung? Bei der Untersuchung des Laufzeitverhaltens des optimierten Algorithmus, stellen wir zuerst fest, dass die Anzahl der Vergleiche nun nicht mehr unabhängig von der Eingabe ist. Somit müssen wir auch hier best-, worst- und average-case unterscheiden:

## *Vergleiche*

**Best-case:**
Dieser Fall tritt nun ein, wenn das Array bereits sortiert ist. Die äußere Schleife wird dann nur einmal durchlaufen, festgestellt, dass nichts vertauscht wurde und der Sortiervorgang schließlich beendet. Wir haben also $n-1$ Vergleiche in der inneren Schleife und durch unsere Optimierung einen weiteren in der äußeren, den wir aber außen vor lassen, weil er das Datenfeld nicht betrifft:
$V(n) = (n-1) \in \Omega(n)$

**Worst-case:**
Wenn das Array genau umgekehrt sortiert ist, tritt der entgegen gesetzte Fall ein. Der Sortiervorgang wird nicht vorher abgebrochen, jedoch greift unsere zweite Optimierung und unnötige Vergleiche werden ausgelassen:

$$V(n) = (n-1) + (n-2) + ... + 2 + 1$$

$$= \sum_{i=1}^{n-1} i$$

$$= \frac{(n-1)n}{2} - 1 = \frac{n^2}{2} - \frac{n}{2}$$

$$\frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$

Somit bleibt es bei $V(n) \in O(n^2)$

**Average-case:**
Ein eindeutiger Durchschnittsfall ist aufgrund der auf Kapitel 2.3.2. schon dargelegten Umstände nicht eindeutig auszumachen. Trotzdem lässt sich wieder allgemein zeigen, dass sich hier an der Ordnung nichts ändert:
Sei k die Anzahl der Durchläufe der äußeren Schleife die nicht mehr ausgeführt werden( $0 \le k \le n$ ), da der Sortiervorgang vorher abgebrochen wird. Die Anzahl der Vergleiche beträgt dann:

$$V(n) = \sum_{i=(k+1)}^{n-1} i$$

$$= \sum_{i=1}^{n-1} i - \sum_{i=1}^{k} i$$

$$= \frac{(n-1)n}{2} - \frac{k(k+1)}{2}$$

$$= \frac{n^2 - n - (k^2 + k)}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2} - \frac{k^2}{2} - \frac{k}{2} \in O(n^2)$$

Somit gilt immer noch: $V(n) \in O(n^2)$

Die Ordnung der Laufzeit des Algorithmus hat sich also nicht verändert. Bleibt aber immer noch ein konstanter Faktor, um den sich der Algorithmus verbessert haben könnte. Um diesen zu ermitteln, vergleichen wir direkt die beiden Funktionen, die die Anzahl der Vergleiche angeben:

$$n^2 - 2n + 1 = r\left(\frac{n^2}{2} - \frac{n}{2} - \frac{k^2}{2} - \frac{k}{2}\right)$$

r ist nun der Faktor um den sich die Laufzeit in Bezug auf die Vergleiche verbessert hat.

$$r = \frac{n^2 - 2n + 1}{\left(\frac{n^2}{2} - \frac{n}{2} - \frac{k^2}{2} - \frac{k}{2}\right)}$$
$$= \frac{2n^2 - 4n + 2}{n^2 - n - k^2 - k}$$

Für $n \to \infty$ ergibt sich:

$$\lim_{n\to\infty}\left(\frac{2n^2 - 4n + 2}{n^2 - n - k^2 - k}\right) = 2$$

Für große n halbiert sich also die fast Anzahl der Vergleiche. Wie aber sieht es mit verhältnismäßig kleinen n aus? Ab wann müssen wir damit rechnen, dass der Original-BubbleSort schneller ist?

Hierzu setzen wir k=0, da wir unabhängig von der ursprünglichen Verteilung der Elemente im Datenfeld argumentieren wollen. Das können wir, da k, also die Konstante, die definiert, wie viele Schleifendurchgänge nicht mehr durchgeführt werden, die Anzahl der Vergleiche reduziert. Wir gehen also davon aus, dass der Sortiervorgang nicht vorher abgebrochen wird. In diesen Fällen hätte der verbesserte Algorithmus sowieso einen Vorteil.

$$n^2 - n = 2n^2 - 4n + 2$$
$$0 = n^2 - 3n + 2$$
$$n_{1,2} = \frac{3}{2} \pm \sqrt{\left(\frac{3}{2}\right)^2 - 2}$$
$$n_1 = 2$$
$$n_2 = 1$$

Da es wenig Sinn macht nur ein Element zu sortieren, ist $n_1$ hier der interessante Wert. Da wir auch schon bewiesen haben, dass der Faktor für $n \to \infty$ 2 beträgt, also größer 1 ist, kommen wir zu dem Schluss, dass sich der verbesserte Algorithmus bei einer Eingabemenge von 3 oder mehr lohnt. Auch sollte der Unterschied bei 2 Elementen so gering sein, dass dies nicht ins Gewicht fällt. Haben wir also die Wahl entweder den Original-BubbleSort-Algorithmus oder den verbesserten zu nehmen, so ist der verbesserte immer vorzuziehen. Der einzige Nachteil ist seine geringere Simplizität. Die zusätzlichen Zuweisungen betreffen nicht das Datenfeld und sind somit vernachlässigbar.

**Ergebnis:**

BubbleSort gehört mit seiner quadratischen Laufzeit eher zu den langsamen Algorithmen. Er ist jedoch sehr einfach zu implementieren und bei überschaubaren Datenmengen(ca. 50 Elemente) noch annehmbar schnell.
In optimierter Form hat dieser Algorithmus allerdings Vorteile, wenn die Liste bereits sortiert ist. Dies wird sofort erkannt und der Sortiervorgang abgebrochen.

## 3.1.3. InsertionSort(SkatSort, BinaryInsertion)[xiv]

InsertionSort

| **Typ** | elementar | | |
|---|---|---|---|
| **Eigenschaften** | iterativ, in-place, stabil, sortierter Bereich | | |
| **Simplizität:** | sehr einfach | | |
| | | | |
| **Laufzeit:** | **best-case** | **average-case** | **worst-case** |
| **Vergleiche** | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ |
| **Zuweisungen** | 0 | $\Theta(n^2)$ | $O(n^2)$ |

**Idee:**
Einen ganz anderen Ansatz verfolgt InsertionSort. Statt das nächste Element in der sortierten Liste zu bestimmen, wird der nächste Wert in die Liste an der richtigen Stelle eingefügt. Konkret heißt dass, dass zuerst das erste Element als sortierter Bereich angesehen wird(eine Menge von einer Zahl ist immer sortiert). Danach wird jedes Element in der unsortierten Menge in den sortierten Bereich eingefügt.

**Name:**
Der Name InsertionSort(oder InsertSort) leitet sich vom englischen Wort für ‚einfügen' *(to) insert* ab. Des weiteren nennt man den originalen InsertionSort-Algorithmus, der mit sequentieller Suche arbeitet, manchmal auch SkatSort, da man – unbewusst – beim Kartenspiel diesen Algorithmus oft verwendet. Auch der Name „StraightInsertion"(etwa: direktes Einfügen) bezeichnet InsertionSort mit sequentieller Suche.

```
procedure SkatSort(var A: array of Integer);
var
  i: Integer;
  j: Integer;
  tmp: Integer;
begin
  for i:= 1 to high(A) do
  begin
    j:= i;
    tmp := A[i];
    while (j > 0) and (A[j-1] > tmp) do
    begin
      // Verschieben:
      A[j]:= A[j-1];
      j := j - 1;
    end;
    A[j]:= tmp;
  end;
end;
```

### *Vergleiche*

**Behauptung:**
SkatSort benötigt im worst-case $V(n) \in O(n^2)$ Vergleiche, im best-case $V(n) \in \Omega(n)$ und im average-case $V(n) \in \Theta(n^2)$.

**Beweis:**

Der entscheidende Vergleich ist der zweite im Kopf der while-Schleife. Die Anzahl der Vergleiche hängt demnach wieder von der Vorsortierung des Arrays ab:

***Best-case:***
Im besten Fall ist das Array bereits sortiert. Dann ergibt der Vergleich immer false, eine Verschiebung ist nicht nötig und demnach auch kein weiterer Vergleich.

$$V(n) = n - 1 \in \Omega(n)$$

***Worst-case:***
Im schlechtesten Fall ist das Datenfeld genau umgekehrt sortiert. Dann wird die while-Schleife solange ausgeführt bis $j < 1$ ist. Dann wird das nächste Element ganz vorne platziert.

$$V(n) = 1 + 2 + ... + (n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \in O(n^2)$$

$$V(n) \in O(n^2)$$

***Average-case:***
Im Durchschnitt(d.h. unter der Annahme, wir hätten eine Liste mit zufälliger Permutation) wird wohl das einzufügende Element in der Mitte des sortierten Bereichs seinen Platz haben. Demnach wird der Vergleich `A[j-1] > tmp` an dieser Stelle false liefern, sodass das Element dort eingefügt wird:

$$V(n) = \sum_{i=1}^{n} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{n} i = \frac{n(n+1)}{4} = \frac{n^2}{4} + \frac{n}{4} \in \Theta(n^2)$$

**Inversionsfolge**
Mithilfe der so genannten Inversionsfolge kann man die genaue Anzahl der benötigten Schritte bestimmen.

**Definition:** Sei $a_0, a_1, ..., a_{n-1}$ die Folge der Elemente der unsortierten Menge. Eine Inversion besteht dann, wenn ein Element $a_i$ kleiner ist als ein Element $a_j$ mit j<i. Eine Inversion ist also eine „Vertauschung“, eine falsche Position im Datenfeld.

Man kann für jedes Element im Array die Anzahl der Inversionen bestimmen. Diese ergibt sich aus der Anzahl der größeren Element links des Elementes.

**Definition:** Ist die Folge $v_0, v_1, ..., v_{n-1}$ die Folge der Inversionen der Elemente der Folge $a_0, a_1, ..., a_{n-1}$, so nennt man sie Inversionsfolge.

**Beispiel:** Gegeben ist die Folge $a = 3,5,1,2,8$, dann ist die zugehörige Inversionsfolge $v = 0,0,2,2,0$

Eine sortierte Folge hat die Inversionsfolge $v_0, v_1, ..., v_{n-1}$ mit $v_i = 0$.

Die Summe der Inversionen $\sum_{i=0}^{n-1} v_i$ entspricht der Anzahl der nötigen Schritte für den InsertionSort-Algorithmus, da InsertionSort genau $v_i$ Schritte benötigt um das Element $a_i$ einzufügen.

**Info 4**: Inversionsfolge[xv]

## *Zuweisungen*

**Behauptung:**
SkatSort benötigt zum Sortieren mindestens $Z(n) \in \Omega(n)$, höchstens $Z(n) \in O(n^2)$ und durchschnittlich $Z(n) \in \Theta(n^2)$ Zuweisungen.

**Beweis:**

***Best-case:***
Zuweisungen gibt es sowohl in der while-Schleife, als auch in der äußeren for-schleife. Ist das Array bereits sortiert, so wird nur letztere ausgeführt:
$Z(n) = n - 1 \in \Omega(n)$

***Worst-case***
Im schlechtesten Fall ist das Array genau umgekehrt sortiert. In diesem Fall muss das nächste Element immer wieder ganz nach vorne gebracht werden, was bedeutet, dass die while-Schleife solange ausgeführt wird, bis $j > 0$ ist. Es wird also immer wieder der gesamte sortierte Bereich nach rechts verschoben um ganz am Anfang Platz zu schaffen.

$$Z(n) = (n-1) + \sum_{i=1}^{n-1} i = (n-1) + \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} + n - 1 = \frac{n^2}{2} + \frac{n}{2} - 1 \in O(n^2)$$

***Average-case***
Wie schon bei den Vergleichen erwähnt wird im Durchschnitt wohl etwa die Hälfte des sortierten Bereiches jeweils verschoben werden müssen um dem einzufügenden Element Platz zu machen:

$$Z(n) = (n-1) + \sum_{i=1}^{n-1} \frac{i}{2} = (n-1) + \frac{1}{2}\sum_{i=1}^{n-1} i = (n-1) + \frac{(n-1)n}{4} = \frac{n^2}{4} - \frac{n}{4} + n - 1 = \frac{n^2}{4} + \frac{3}{4}n - 1 \in \Theta(n^2)$$

**Optimierungen:**
Wenn man die Suche nach der Position des einzufügenden Elementes im sortieren Bereich effizienter gestaltet, lässt sich InsertionSort weiter verbessern. Wir optimieren also InsertionSort, indem wir statt der linearen, eine binäre Suche verwenden:

```
procedure BinaryInsertion(var A: array of Integer);
var
  i: Integer;
  k: Integer;
  left: Integer;
  right: Integer;
  current: Integer;
  tmp: Integer;
begin
  for i:= 1 to high(A) do
  begin
    // Position bestimmen mittels binärer Suche:
    left := 0;
    right := i-1;
    tmp := A[i];
    while left <= right do
    begin
      current := (left + right) div 2;
      if A[current] > tmp then
      begin
        right := current -1;
      end
      else
      begin
        left := current +1;
      end;
    end;
    // Verschieben:
    for k := i downto left do
    begin
      A[k] := A[k-1];
    end;
    // Einfügen:
    A[left] := tmp;
  end;
end;
```

**Suchalgorithmen**
Es gibt im allgemeinen zwei wichtige Suchmethoden. 1. die *lineare* oder auch *sequentielle Suche*(Diese vergleicht jedes Element einzeln mit dem zu suchenden) und 2. die binäre Suche(Diese funktioniert ähnlich, wie das suchen in einem Wörterbuch).
*(Siehe auch Anhang E)*

**Info 5:** Suchalgorithmen

An der Anzahl der Zuweisungen ändert sich nichts. Wohl aber an der Anzahl der Vergleiche. Aber in wie weit hat sich diese verbessert und gibt es vielleicht auch Fälle, in denen sich die Anzahl der nötigen Vergleiche erhöht hat?

Die Anzahl der Vergleiche entspricht nun bei jeder neuen Einfüge-Operation der Laufzeit der binären Suche(siehe Anhang E):
$V(n) = (n-1)\log(n) \in \Theta(n\log(n))$
$V(n) \in \Theta(n\log(n))$. Im best-case hat sich das Laufzeitverhalten also von linear auf $n\log(n)$ verschlechtert, im average- und worst-case aber von quadratisch auf $n\log(n)$ verbessert.

**Ergebnis:**
InsertionSort gehört mit seiner quadratischen Laufzeit prinzipiell zu den langsameren Algorithmen. Wenn die zu sortierenden Menge aber schon fast sortiert ist, d.h. die Anzahl der Inversionen relativ klein ist, läuft InsertionSort annähernd linear. Die Optimierung der Suche verbessert die Anzahl der Vergleiche im worst- und average-case, jedoch lassen sich die Zuweisungen nicht optimieren. Außerdem wird durch die Optimierung der Vorteil bei fast sortierten Listen aufgegeben.

## 3.1.4. Vergleich dreier elementarer Sortierverfahren

Da wir nun die drei wichtigsten elementaren Sortieralgorithmen untersucht haben, bietet sich ein Vergleich dieser an. Wo liegen die Unterschiede, wo die Gemeinsamkeiten?

Alle drei Algorithmen sind iterativ, in-place, stabil und erzeugen einen sortierten Bereich. Außerdem sind alle einfach zu implementieren.

Auch das Laufzeitverhalten ist relativ ähnlich:

| **SelectionSort** | | | **BubbleSort** | | | **InsertionSort** | | |
|---|---|---|---|---|---|---|---|---|
| ***Vergleiche:*** | | | | | | | | |
| **Best-case** | **Avera-ge-case** | **Worst-case** | **Best-case** | **Avera-ge-case** | **Worst-case** | **Best-case** | **Avera-ge-case** | **Worst-case** |
| $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ |
| ***Zuweisungen:*** | | | | | | | | |
| **Best-case** | **Avera-ge-case** | **Worst-case** | **Best-case** | **Avera-ge-case** | **Worst-case** | **Best-case** | **Avera-ge-case** | **Worst-case** |
| $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ | 0 | $\Theta(n^2)$ | $O(n^2)$ | 0 | $\Theta(n^2)$ | $O(n^2)$ |

Schauen wir uns zuerst einmal die Anzahl der Vergleiche im worst-case an: In jedem der drei Fälle beträgt die Ordnung $O(n^2)$. Wie sieht es aber mit konstanten Faktoren aus? Bei allen drei Algorithmen konnten wir genaue Werte für diesen Fall errechnen:

| | **SelectionSort** | **BubbleSort** | **InsertionSort** |
|---|---|---|---|
| **Vergleiche im worst-case** | $V(n) = \frac{n^2}{2} - \frac{n}{2}$ | $V(n) = n^2 - 2n + 1$ | $V(n) = \frac{n^2}{2} + \frac{n}{2}$ |
| **Zuweisungen im worst-case** | $Z(n) = 3n - 3$ | $Z(n) = \frac{3}{2}n^2 - \frac{3}{2}n$ | $Z(n) = \frac{n^2}{2} + \frac{n}{2}$ |

Hier sehen wir, dass SelectionSort, sowie InsertionSort etwa halb so viele Vergleiche benötigen, wie BubbleSort. Dabei ist SelectionSort noch einen Tick besser. In der optimierten Version ist BubbleSort hier jedoch wieder ähnlich schnell, wie die beiden anderen(ebenfalls $V(n) = \frac{n^2}{2} - \frac{n}{2}$).

Im Best-case hingegen ist InsertionSort mit seiner linearen Laufzeit klar vorne.

Bei den Zuweisungen sieht es anders aus: Hier ist SelectionSort linear und BubbleSort, sowie InsertionSort quadratisch, wobei hier BubbleSort fast drei mal so viele Zuweisungen vornimmt. Im best-case nehmen diese beiden jedoch auch keine Zuweisungen mehr vor. In der optimierten Variante verhält sich SelectionSort genauso.

Als vorzeitiges Zwischenergebnis können wir also festhalten, dass in Anbetracht der Vergleiche InsertionSort von Vorteil ist. Sind jedoch Zuweisungen „teuer“, so hat SelectionSort einen Vorteil. BubbleSort ist in der Original-Variante eher unbrauchbar. In der optimierten Version aber reicht er in Bezug auf die Vergleiche an die beiden anderen heran ohne jedoch besondere Vorteile zu bieten.

Sehen wir uns nun folgende Spezialfälle an: sortierte, umgekehrt sortierte, „fast sortierte“ und zufällige Mengen.

**Sortierte Mengen:**
Sortierte Mengen könnten theoretisch mit einem Aufwand von $\Omega(n)$ Vergleichen „sortiert“ werden, da in diesem Fall nur eine Überprüfung der Sortierung notwendig ist und keine Datenbewegungen. BubleSort, InsertionSort und der optimierte SelectionSort nehmen auch keine Zuweisungen vor. In Bezug auf die Vergleiche sind hier SelectionSort und der Original-BubbleSort mit ihrer quadratischen Laufzeit im Hintertreffen. InsertionSort und der verbesserte BubbleSort erreichen in diesem Fall die untere Schranke von $\Omega(n)$.

**Umgekehrt sortierte Mengen:**
Ist die Liste genau umgekehrt sortiert, so tritt bei allen drei Algorithmen der worst-case ein. Wie oben schon erwähnt hätte da SelectionSort einen kleinen Vorteil. SelectionSort und InsertionSort lassen sich jedoch so modifizieren, dass umgekehrt sortierte Listen dem best-case entsprechen[I]. Allerdings wird dabei nur best-case und worst-case vertauscht. Sortierte Listen resultieren dann also in einer langen Laufzeit. Deshalb sind solche Modifikationen nur eingeschränkt sinnvoll.

**„fast sortierte“ Mengen:**
Ist eine Liste schon vorsortiert, d.h. sind nur einige Elemente vertauscht oder verschoben, so sortiert InsertionSort diese Liste in annähernd linearer Zeit. BubbleSort und SelectionSort haben mit solchen Listen schon mehr Probleme. Dies verdeutlicht, dass InsertionSort der flexibelste dieser drei Algorithmen ist.

**Zufällige Mengen:**
Eine Zufällige Menge entspricht ungefähr dem, was wir als „Normalfall“ ansehen würden. Dass dies nicht so pauschal gesagt werden kann, habe ich bereits erläutert. Trotzdem ist eine Liste mit Elementen in einer zufälligen Reihenfolge recht häufig und dient deshalb oft als Beispiel für den average-case.

| | **SelectionSort** | **BubbleSort** | **InsertionSort** |
|---|---|---|---|
| **Vergleiche im average-case** | $V(n)=\frac{n^2}{2}-\frac{n}{2}$ | $V(n)=n^2-2n+1$ | $V(n)=\frac{n^2}{4}+\frac{n}{4}$ |
| **Zuweisungen im average-case** | $Z(n)=3n-3$ | $Z(n)=\frac{300-3k}{200}\cdot(n^2-n)$ | $Z(n)=\frac{n^2}{4}+\frac{3}{4}n-1$ |

Auch im average-case sehen wir sehr deutlich, dass InsertionSort die wenigsten Vergleiche benötigt und SelectionSort die wenigsten Zuweisungen. BubbleSort hingegen präsentiert sich als eher langsame Sortiermöglichkeit.

[I] Dazu muss nur die jeweilige Schleife rückwärts zählen.

## 3.2. QuickSort[xvi]

QuickSort

| **Typ** | asymptotisch optimal | | |
|---|---|---|---|
| **Eigenschaften** | halbrekursiv, in-place, instabil, kein sortierter Bereich | | |
| **Simplizität:** | noch überschaubar, aber nicht mehr einfach | | |
| | | | |
| **Laufzeit:** | **best-case** | **average-case** | **worst-case** |
| **Vergleiche** | $\Omega(n\log(n))$ | $\Theta(n\log(n))$ | $O(n^2)$ |
| **Zuweisungen** | 0 | $\Theta(n\log(n))$ | $O(n^2)$ |

**Idee:**
Wähle ein Element(das so genannte Pivotelement) aus. Zerlegt die zu sortierende Menge in zwei Teilmengen, für die gilt: In der ersten Menge befinden sich nur Elemente, die kleiner oder gleich dem zuvor gewählten Element sind und in der anderen nur solche, die größer gleich diesem sind. Verfahre dann mit den beiden Teilmengen genauso, bis sich in den Teilmengen nur noch genau ein Element befindet.

**Name:**
Der Name QuickSort leitet sich aus dem englischen Wort für schnell(*quick*) ab. Die Bezeichnung nimmt also auf die im Durchschnitt sehr gute Laufzeit Bezug.

```
procedure QuickSort(var A: array of Integer);

  procedure QuickSort(iLo, iHi: Integer);
  var
    Lo, Hi, Mid, tmp: Integer;
  begin
    Lo := iLo;
    Hi := iHi;
    Mid := A[(Lo + Hi) div 2];
    repeat
      // Annähern:
      while A[Lo] < Mid do
      begin
        Lo := Lo +1;
      end;
      while A[Hi] > Mid do
      begin
        Hi := Hi -1;
      end;

      if Lo <= Hi then
      begin
        // Austauschen:
        tmp := A[Hi];
        A[Hi] := A[Lo];
        A[Lo] := tmp;
        // Weinterrücken:
        Lo := Lo +1;
        Hi := Hi -1;
      end;
    until Lo > Hi;
    // Teilbereiche sortieren:
    if Hi > iLo then QuickSort(iLo, Hi);
    if Lo < iHi then QuickSort(Lo, iHi);
  end;

begin
  QuickSort(0, high(A));
end;
```

## *Vergleiche*

**Behauptung:**
QuickSort benötigt im besten, sowie in durchschnittlichen Fall $\Omega(n\log(n))$ bzw. $\Theta(n\log(n))$ Vergleiche im schlechtesten Fall, ist QuickSort jedoch quadratisch.

**Beweis:**

**Best-case:**
QuickSort funktioniert nach dem Prinzip „Teile und Herrsche"(eng.: *„devide and conquer"*), d.h. dadurch, dass das komplexe Sortierproblem in kleinere leichtere Teilprobleme zerlegt wird. Ist die Zerlegung optimal wird also immer in der Mitte geteilt, so erreicht QuickSort seine maximale Geschwindigkeit.
In diesem Fall werden die beiden while-Schleifen solange ausgeführt bis sie die Mitte erreicht haben[1].

```
while A[Lo] < Mid do
begin
  Lo := Lo +1;
end;
while A[Hi] > Mid do
begin
  Hi := Hi -1;
end;
```

Danach wird in den beiden Teilfeldern genauso fortgefahren. Die Vergleiche finden in den Schleifenköpfen der erwähnten while-Schleifen statt. Da ein Vergleich doppelt vorgenommen wird(Den Vergleich mit dem mittleren Element führen beide Schleifen durch.), beträgt die Anzahl der Vergleiche für einen Aufruf von QuickSort $m+1$ wobei $m$ der Anzahl der Elemente des Teilfeldes entspricht:

Da QuickSort ein rekursiver Algorithmus ist, sieht unser Beweis etwas anders aus, als gewöhnlich:
$V_m = 2V_{m/2} + (m+1)$
$V_m$ ist die Anzahl der nötigen Vergleiche für eine Eingabemenge(bzw. Teilmenge) mit $m$ Elementen. Ein Durchlauf von QuickSort benötigt $(m+1)$ Vergleiche(siehe oben). Zusätzlich wird aber QuickSort noch zweimal aufgerufen, jedoch mit einer um die Hälfte kleineren Teilmenge.

Um diese rekurrente Beziehung aufzulösen substituieren wir $m$ mit $2^k$:

$$\begin{aligned} V_{2^k} &= 2V_{2^k/2} + (2^k+1) \\ &= 2V_{2^{k-1}} + (2^k+1) \end{aligned}$$

Dann teilen wir durch m bzw. $2^k$:

$$\begin{aligned} \frac{V_{2^k}}{2^k} &= 2\frac{V_{2^{k-1}}}{2^k} + \frac{2^k}{2^k} + \frac{1}{2^k} \\ &= \frac{V_{2^{k-1}}}{2^{k-1}} + 1 + \frac{1}{2^k} \end{aligned}$$

Als nächstes lösen wir Schritt für Schritt die Rekursion auf. Dabei setzen wir den ursprünglichen Term ein:

[1] ggf. werden sie unterbrochen und nach der Austauschoperation aufgrund der repeat-Schleife weitergeführt.

$$\frac{V_{2^k}}{2^k} = \frac{V_{2^{k-1}}}{2^{k-1}} + 1 + \frac{1}{2^k}$$
$$= \frac{2V_{2^{k-2}} + 2^{k-1} + 1}{2^{k-1}} + 1 + \frac{1}{2^k}$$
$$= \frac{2V_{2^{k-2}}}{2^{k-1}} + \frac{2^{k-1}}{2^{k-1}} + \frac{1}{2^{k-1}} + 1 + \frac{1}{2^k}$$
$$= \frac{V_{2^{k-2}}}{2^{k-2}} + 1 + \frac{1}{2^{k-1}} + 1 + \frac{1}{2^k}$$
$$= \frac{V_{2^{k-2}}}{2^{k-2}} + 2 + \frac{1}{2^k} + \frac{1}{2^{k-1}}$$

Der erste Rekursionsschritt ist aufgelöst. Auf diese Weise fahren wir nun fort:

$$\frac{V_{2^k}}{2^k} = \frac{V_{2^{k-1}}}{2^{k-1}} + 1 + \frac{1}{2^k}$$
$$= \frac{V_{2^{k-2}}}{2^{k-2}} + 2 + \frac{1}{2^k} + \frac{1}{2^{k-1}}$$
$$= \frac{V_{2^{k-3}}}{2^{k-3}} + 3 + \frac{1}{2^k} + \frac{1}{2^{k-1}} + \frac{1}{2^{k-2}}$$
$$= \frac{V_{2^{k-4}}}{2^{k-4}} + 4 + \frac{1}{2^k} + \frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \frac{1}{2^{k-3}}$$
$$\ldots$$
$$\ldots$$
$$\ldots$$
$$= \frac{V_2}{2} + (k-1) + \frac{1}{2^k} + \frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \frac{1}{2^{k-3}} + \ldots + \frac{1}{2^0}$$
$$= \frac{V_2}{2} + (k-1) + \sum_{i=0}^{k} \frac{1}{2^i}$$
$$= V_1 + k + \sum_{i=-1}^{k} \frac{1}{2^i}$$

$V_1$ beschriebt die Anzahl Vergleiche, die nötig ist um eine Menge mit einem Element zu sortieren. $V_1$ ist also 0. $\sum_{i=-1}^{k} \frac{1}{2^i}$ ist asymptotisch vernachlässigbar: $\sum_{i=-1}^{k} \frac{1}{2^i} \in o(k)$ Somit gilt für $k \to \infty$:

$$\frac{V_{2^k}}{2^k} = k$$
$$V_{2^k} = 2^k \cdot k$$

Schließlich können wir die Substitution auflösen:

$$V_m = m \cdot ld(m)$$

Nun ist die rekurrente Beziehung vollständig aufgelöst und somit das n nicht mehr missverständlich. Wir können also ebenfalls schreiben:

$$V(n) = n \cdot ld(n) \in \Omega(n \log(n))$$
$$V(n) \in \Omega(n \log(n))$$

**Worst-case:**

Im schlechtesten Fall wird immer das größte Element als Pivotelement benutzt. Somit entsteht eine Teilmenge mit einem Element und eine mit $m-1$ Elementen, wobei $m$ der Anzahl der Elemente der im aktuellen Durchgang verarbeiteten Teilmenge entspricht. Es werden solange Teilmengen gebildet, bis $m=2$ ist. Somit errechnet sich die Anzahl der Vergleiche im worst-case folgendermaßen:

$$V(n) = 2((n+1)+(n-1+1)+(n-2+1)+\ldots+4+3)$$

$$= 2\sum_{i=1}^{n}(i+1)-3$$

$$= 2\sum_{i=1}^{n+1} i-3$$

$$= 2\frac{(n+1)(n+2)}{2}-3$$

$$= (n+1)(n+2)-3$$

$$= n^2+2n+n+2-3$$

$$= n^2+3n-1 \in O(n^2)$$

$$V(n) \in O(n^2)$$

**Average-case:**
Der Beweis, dass QuickSort im Durchschnitt ebenfalls eine Laufzeit von $n\log(n)$ aufweist, gestaltet sich etwas schwieriger. Man möge mir also glauben, dass auch in diesem Fall QuickSort seinem Namen gerecht wird.

**Was macht QuickSort so schnell?**
Das Geheimnis von QuickSort liegt zum Einen darin, dass Elemente aus großer Entfernung miteinander vertauscht werden, zum Anderen ist es auch das *„Devide and Conquer"-Prinzip*, was die Effizienz von Quicksort ausmacht.

**Optimierungen:**
Es gibt auch einige Möglichkeiten QuickSort zu optimieren, jedoch beschränken diese sich entweder darauf, das Pivotelement günstiger zu wählen und damit die Wahrscheinlichkeit eines worst-case zu verringern bzw. die mittlere worst-case-Laufzeit etwas zu verbessern, oder QuickSort wird, wenn die Anzahl der zu sortierenden Elemente in einem Teilbereich eine gewisse Grenze unterschreitet, abgebrochen und mit einem anderen Sortierverfahren (vorzugsweise SkatSort) weitersortiert[I]. Diese Optimierungen verbessern QuickSort jedoch nur wenig. Im ersten Fall wird nur die Wahrscheinlichkeit für den worst-case verringert bzw. die Laufzeit des schlechtesten Falls geringfügig verbessert. Ganz verhindern kann man den quadratischen worst-case jedoch nicht. Auch die Verwendung von SkatSort bei kleineren Sortiermengen kann die Laufzeit nicht wesentlich verbessern, da QuickSort ja schon asymptoisch optimal ist.

**Ergebnis:**
QuickSort ist nicht umsonst so bekannt und so oft genutzt. Durch seine asymptotisch optimale Laufzeit sortiert Quicksort große Mengen von Eingabedaten in kurzer Zeit. Jedoch hat auch QuickSort einige Nachteile: Da QuickSort ein rekursiver Algorithmus ist, wird zusätzlich Speicherplatz benötigt um die Rücksprungadressen und Parameter zu organisieren. Außerdem ist Quicksort instabil, d.h. die Reihenfolge von gleichrangigen Elementen kann durcheinander geraten.

[I] Letzteres hat seinen Grund darin, dass QuickSort bei kleineren Sortiermengen weniger effizient arbeitet, SkatSort hingegen sehr gut(insbesondere da schon eine gewisse Vorsortierung besteht).

## 3.3. BogoSort[xvii]

BogoSort

| **Typ** | nicht verwendbar | | |
|---|---|---|---|
| **Eigenschaften** | iterativ, in-place, instabil, kein sortierter Bereich | | |
| **Simplizität:** | Komplex | | |
| | | | |
| **Laufzeit:** | **best-case** | **average-case** | **worst-case** |
| **Vergleiche** | $\Omega(n)$ | $\Theta(n!)$ | unendlich |
| **Zuweisungen** | 0 | $\Theta(n \cdot n!)$ | unendlich |

**Idee:**
BogoSort funktioniert kurz gesagt nach dem Schema „Nimm einen Kartenstapel, werfe ihn in die Luft, hebe alle Karten auf und prüfe, ob die Karten sortiert sind. Wenn nicht, fange noch einmal von vorne an.“ Dass diese Art zu sortieren nicht besonders effizient ist, liegt auf der Hand, jedoch kann man auch mathematisch zeigen wie ineffektiv BogoSort ist.

**Name:**
BogoSort wird manchmal auch StupidSort(eng*: „DummSortieren*“; wobei dieser Name auch einem anderen Algorithmus bezeichnen kann) oder RamdomSort(eng: *„ZufallsSortrieren“*) genannt. Der Name *Bogo*Sort stammt vom englischen (Slang)Wort bogus, was so viel wie „Fälschung“ oder „Schwindel“ bedeutet. Höchstwahrscheinlich wurde BogoSort zum Zeitvertreib entworfen. BogoSort ist damit aber kein Einzelfall. Auf diese Weise ist z.B. auch ein Sortierverfahren namens SlowSort[xviii] entwickelt. Es gibt sogar Publikationen, die dies zum Thema machen(„Pessimal Algorithms and Simplexity Analysis“ von Andrei Broder und Jorge Stolfi). Aus ähnlichem Antrieb entstehen z.B. auch so genannte „esoterische Programmiersprachen“[I] und Nihilartikel[II]. BogoSort ist also wahrscheinlich ebenso ein Produkt menschlichen Humors.

**Kein Algorithmus**
BogoSort ist genau genommen nur eine Sortiermethode und kein Sortieralgorithmus. Für einen Algorithmus gelten nämlich folgende Voraussetzungen:

- Allgemeingültigkeit: Ein Algorithmus löst ein Problem, nicht nur einen bestimmten Einzelfall.
- Ausführbarkeit: Ein Algorithmus muss ausführbar sein.
- Endlichkeit: Der Algorithmus muss durch eine endliche Anzahl von Schritten festzulegen sein.
- Eindeutigkeit: Jeder Schritt muss eindeutig sein. Es darf keinen Interpretationsspielraum geben. Auch darf es keine Fälle geben, für die der nächste Schritt undefiniert ist.
- Terminiertheit: Ein Algorithmus muss in jedem Fall ein determiniertes Ende haben.

Letztere Bedingung ist bei BogoSort offensichtlich nicht erfüllt. Es ist nicht gewährleistet, dass BogoSort überhaupt irgendwann fertig wird. BogoSort ist also genau genommen kein Algorithmus.

**Info 6**: Kein Algorithmus[xix]

[I] Das Ziel solcher Programmiersprachen ist es möglichst kleine Compiler zu haben.
[II] Lexikonartikel über fiktive Stichwörter.

```
procedure BogoSort(var A: array of Integer);

  function IsSorted: Boolean;
  var
    i: Integer;
  begin
    Result := True;
    for i := Low(A) to High(A) -1 do
    begin
      if A[i] > A[i+1] then
      begin
        Result := False;
        Break;
      end;
    end;
  end:

  procedure RandomizeArray;
  var
    i: Integer;
    tmp: Integer;
    rnd: Integer;
  begin
    for i := Low(A) to High(A) do
    begin
      rnd := random(High(A) +1);
      tmp := A[i];
      A[i] := A[rnd];
      A[rnd] := tmp;
    end;
  end;

begin
  Randomize;
  while not IsSorted do
  begin
    RandomizeArray;
  end;
end;
```

## *Vergleiche*

Alle Vergleiche finden in der Prozedur IsSorted statt.

***Best-case:***

Im besten Fall ist das Array bereits sortiert. In diesem Fall wird genau dies getestet und dann abgebrochen. Das Testen benötigt genau $n-1$ Vergleiche:
$V(n) = n-1 \in \Omega(n)$.

***Worst-case:***

Im schlimmsten Fall wird die richtige Permutation nie gefunden und der Algorithmus iteriert bis in alle Ewigkeit weiter.

***Average-case:***

Die Anzahl der möglichen Permutationen des Arrays beträgt $n!$[xx]. Unter der Annahme, dass es keine gleichen Elemente gibt, ist nur eine dieser Permutationen die gesuchte. Die durchschnittliche Anzahl der nötigen Versuche errechnet sich aus dem Erwartungswert eines entsprechenden Zufallsexperimentes(nichts anderes ist dieses „Sortierverfahren“).

Die Wahrscheinlichkeit die richtige Permutation zu erhalten beträgt:

$$p = \frac{1}{n!}$$

Da es sich um eine Binomialverteilung handelt(IsSorted gibt entweder true oder false zurück), lässt sich der Erwartungswert mittels der Formel $E(X) = np$ berechnen. Um Verwechselungen

auszuschließen nennen wir die Anzahl der nötigen Versuche m. Der Erwartungswert beträgt 1, da eine richtige Permutation genügt um das Datenfeld als sortiert zu erklären:

$$1 = m\frac{1}{n!}$$
$$m = n!$$

Die Anzahl der im Durchschnitt nötigen Versuche beträgt also $n!$. Ein Versuch benötigt aber 1 bis $n-1$ Vergleiche:

Die Wahrscheinlichkeit, dass bei unbekannter Permutation schon die ersten beiden Elemente nicht in der richtigen Reihenfolge sind, beträgt 50%. Die Wahrscheinlichkeit, dass die ersten beiden stimmen, die nächsten beiden aber nicht, beträgt 25%, usw.

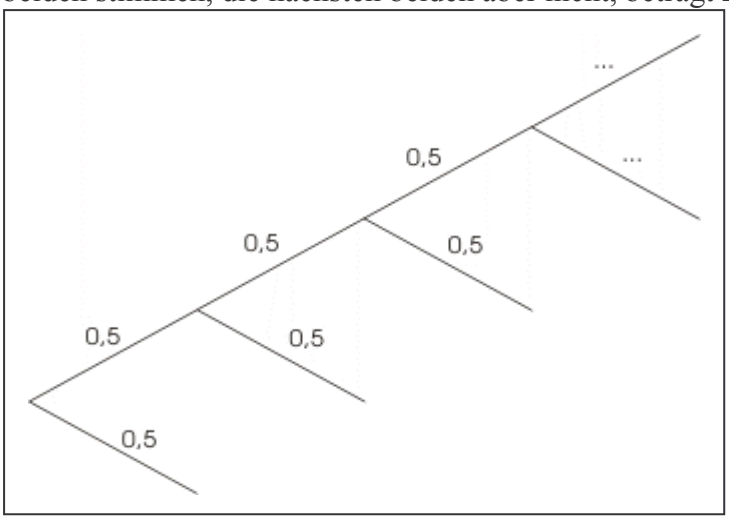



**Abb. 3:** Ein vereinfachter Wahrscheinlichkeitsbaum zu IsSorted

Die Grafik verdeutlicht, dass sich die Wahrscheinlichkeit mit jedem neuen Vergleich halbiert:

$$p_i = 0{,}5$$
$$P(X = k) = \frac{1}{2^k}$$
$$E(X) = \sum_{k=1}^{n-1} \frac{k}{2^k}$$

Für $n \to \infty$:

$$\lim_{n\to\infty}(E(X)) = \lim_{n\to\infty}\left(\sum_{k=1}^{n-1}\frac{k}{2^k}\right) = \sum_{k=1}^{\infty}\frac{k}{2^k}$$

Die Folge $\frac{k}{2^k}$ ist eine Nullfolge deren Reihe $\sum_{k=1}^{\infty}\frac{k}{2^k}$ gegen 2 konvergiert.

| k | $\frac{k}{2^k}$ | $\sum_{k=1}^{\infty}\frac{k}{2^k}$ |
|---|---|---|
| 1 | 0,5 | 0,5 |
| 2 | 0,5 | 1 |
| 3 | 0,375 | 1,375 |
| 4 | 0,25 | 1,625 |
| 5 | 0,15625 | 1,78125 |
| 6 | 0,09375 | 1,875 |
| 7 | 0,0546875 | 1,9296875 |
| 8 | 0,03125 | 1,9609375 |

| | | |
|---|---|---|
| 9 | 0,01757813 | 1,97851563 |
| 10 | 0,00976563 | 1,98828125 |
| 11 | 0,00537109 | 1,99365234 |
| 12 | 0,00292969 | 1,99658203 |
| 13 | 0,00158691 | 1,99816895 |
| 14 | 0,00085449 | 1,99902344 |
| 15 | 0,00045776 | 1,9994812 |
| 16 | 0,00024414 | 1,99972534 |
| 17 | 0,0001297 | 1,99985504 |
| 18 | 6,8665E-05 | 1,99992371 |
| 19 | 3,624E-05 | 1,99995995 |
| 20 | 1,9073E-05 | 1,99997902 |

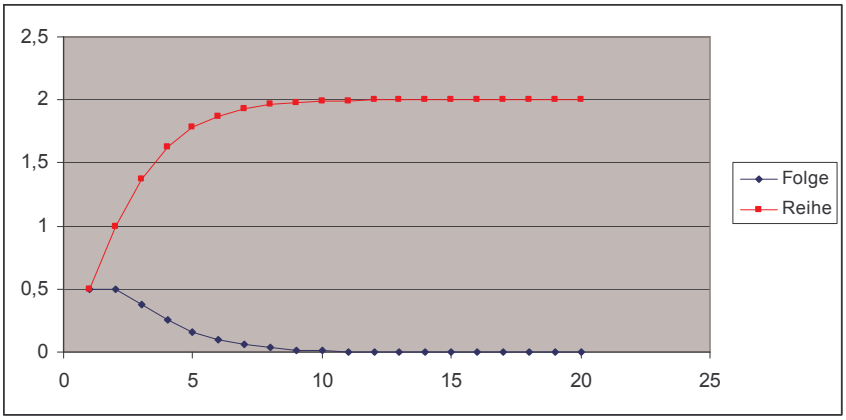



**Abb. 5:** Diagramm: Folge und Reihe

$$\lim_{n\to\infty}(E(X)) = \lim_{n\to\infty}\left(\sum_{k=1}^{n-1}\frac{k}{2^k}\right) = \sum_{k=1}^{\infty}\frac{k}{2^k} = 2$$

Somit beträgt die durchschnittliche Anzahl der nötigen Vergleiche pro Versuch 2. Die Anzahl der Vergleiche, die BogoSort benötigt beträgt also:
$V(n) = 2n!+n \in \Theta(n!)$

$$V(n) \in \Theta(n!)$$

## *Zuweisungen*

***Best-case:***
Im besten Fall ist das Array schon sortiert. Dies wird erkannt und keine weitere Vertauschung mehr vorgenommen.

***Worst-case:***
Wie bei den Vergleichen schon erläutert kann es auch passieren, dass BogoSort niemals fertig sortiert hat. In diesem Fall werden auch immer wieder neue Zuweisungen vorgenommen.

***Average-case:***
BogoSort legt nicht genau fest, wie das Datenfeld durchmischt werden soll. Somit ist die genaue Anzahl der Zuweisungen an die konkrete Implementierung gebunden. In unserem Fall werden immer $n$ Vertauschungen und damit $3n$ Zuweisungen vorgenommen:
$Z(n) = 3n \cdot n! \in \Theta(n \cdot n!)$
$Z(n) \in \Theta(n \cdot n!)$

**Ergebnis:**
BogoSort bemerkt zwar, wenn die Eingabedaten schon sortiert sind, jedoch ist das wohl der einzige „Vorteil“ dieses Sortierverfahrens. Es ist noch nicht einmal sicher gestellt, dass BogoSort überhaupt irgendwann fertig wird. Und selbst im Durchschnittsfall ist BogoSort so langsam, dass selbst kleinste Eingabemengen in längeren Rechenzeiten resultieren. Dabei zeigen die Ergebnisse noch nicht einmal die ganze Langsamkeit von BogoSort, denn das Erzeugen der Zufallszahlen benötigt auch wieder relativ viel Rechenzeit, was BogoSort noch langsamer macht. Man kann BogoSort also mit Recht den Prototyp eines schlechten Algorithmus nennen.

# 4. Fazit

Beschäftigt man sich etwas mit unterschiedlichen Sortierverfahren, so bemerkt man, wie vielfältig die Möglichkeiten sind, so etwas alltägliches wie das Sortieren zu erledigen. Es wird auch klar, wie viel Mathematik uns im Alltag begegnet – ohne, dass wir es merken. Wie wichtig Sortierung ist, zeigt allein schon die Überlegung, dass niemand ein unsortiertes Telefonbuch auch nur anfassen würde. Ohne Sortierung würde in unserer Zeit fast nichts so funktionieren, wie wir es gewohnt sind.

Um nun diese Sortierung möglichst effizient zu gestalten, muss man für jede gegebene Situation den passenden Algorithmus finden. Neben einigen Programmiertechnischen Überlegungen(z.B. die Simplizität), sind es hauptsächlich mathematische Funktionen, die die Eigenschaften eines Algorithmus beschreiben. Somit zeigt sich, dass im Informationszeitalter der Gegenwart die Mathematik wichtiger ist denn je. Ohne Mathematik sähe unsere Welt wohl noch schlimmer aus, als ohne Sortierung. Die Untersuchung von Sortieralgorithmen hat mir das gezeigt.

Zudem bot dies mir die Möglichkeit praktische Anwendungen der Mathematik nachvollziehen und selbst ausprobieren zu können. Die Algorithmik ist ein Gebiet auf dem die Mathematik(nicht nur das Rechnen) direkt zum Einsatz kommt. Insbesondere diejenigen, die sich etwas mit Programmierung beschäftigen, können hier einen direkten Praxisbezug sehen, der der Mathematik ja manchmal abgesprochen wird – zu Unrecht, wie man an diesem Beispiel sieht.

# Anhang

## A Definitionen

***Stabilität:***
Gilt für zwei Objekte $a_k = a_l$ mit $k \neq l$, so sind diese Objekte gleichrangig. Ändert ein Sortierverfahren die relative Reihenfolge gleichrangiger Objekte nicht, so nennt man dieses Sortierverfahren *stabil*, ansonsten *instabil*.[xxi]

***Simplizität***
Für denjenigen, der sich zu entscheiden hat, welchen Algorithmus er für sein Programm wählt und dann auch implementiert, d.h. in Programmcode umsetzt, ist neben anderen Eigenschaften des Algorithmus auch die Simplizität, also dessen „Einfachheit" wichtig. Manchmal lohnt es sich einen langsameren Algorithmus in Kauf zu nehmen(besonders bei relativ kleinen Datenmengen) und ihn dafür schnell und fehlerfrei implementieren zu können.

***Externe und interne Sortierverfahren:***
Benötigt ein Sortierverfahren eine konstante, d.h. von der Anzahl der zu sortierenden Objekte unabhängige, Menge an Speicherplatz, so nennt man es ein *intern* oder *in-place*. Benötigt es hingegen eine von der Anzahl abhängige Menge an Speicherplatz(Dies kommt dadurch zustande, dass statt direkt die Ausgangsfolge zu sortieren eine zweite, sortierte Folge aufgebaut wird.[I]), so heißt es *extern* oder *out-of-place*.[xxii]

***Rekursivität:***
Funktioniert ein Algorithmus dadurch, dass er sich selbst aufruft, so nennt man ihn *rekursiv*[II], benutzt er hingegen nur Wiederholungen(Schleifen), so heißt er *iterativ*[III]. „Zu jedem iterativen Algorithmus kann ein äquivalenter rekursiver Algorithmus angegeben werden."[xxiii] Rekursive Funktionen lassen sich meist relativ leicht implementieren, jedoch müssen sie für jeden Rekursionsschritt eine Rücksprungadresse, samt Parametern speichern, was bei großen Datenmengen dazu führen kann, dass solche Algorithmen viel Speicherplatz benötigen. Als *halbrekursiv*, bzw. h*albiterativ* bezeichnet man Algorithmen, die beide Verfahren verwenden, sich jedoch hauptsächlich auf das eine stützen.

***Sortierter Bereich:***
Manche Sortieralgorithmen erzeugen einen so genannten sortierten Bereich, d.h. einen Teilbereich in der zu sortierenden Menge, der schon sortiert ist.

---

[II] Es wird also mindestens doppelt so viel Speicherplatz benötigt, wie die Objekte selbst belegen
[II] Von lat. recurrere „zurücklaufen"
[III] Von lat. iterare „wiederholen"

# B Der binäre Baum[xxiv]

Bäume als abstrakte Objekte, mit denen man Daten graphisch darstellen kann, kennt man vielleicht von den Stammbäumen und der Stochastik. Aber auch in der Informatik werden Bäume in großem Umfang genutzt. Sowohl in der Darstellung, als auch in der Verarbeitung von Daten werden häufig Baumstrukturen und meist binäre Bäume verwendet.

Im Allgemeinen ist ein Baum eine nichtleere Menge von Knoten(Objekten) und Kanten(Verbindungen zwischen den Knoten). Jeder Knoten ist über Kanten und ggf. andere Knoten mit einem Wurzelknoten verbunden, der per Definition den Ursprung des Baumes bildet. Ist ein Knoten mit einem anderen Knoten verbunden, so bezeichnet man den Knoten, der sich näher an der Wurzel befindet als Vorgänger (oder Elternknoten) und den, der von der Wurzel weiter entfernt ist, als Nachfolger (oder Kindknoten) des jeweils anderen. Knoten, die keine Nachfolger haben heißen Blätter, äußere Knoten oder Endknoten, der Weg von einem Knoten über die einzelnen Kanten zur Wurzel Pfad des Knoten.

Einen Baum, bei dem jeder innere Knoten(Knoten, die keine Blätter sind) genau zwei Nachfolger hat, heißt binärer Baum. Alle Knoten, die den selben Abstand zur Wurzel haben, liegen auf einer Ebene. Die Nummer der Ebene ist gleich dem Abstand zur Wurzel. Demnach liegt die Wurzel in der Ebene 0, die Knoten direkt unter der Wurzel in Ebene 1, usw.. Bei einem *vollständigen* binären Baum befinden sich alle Blätter auf der selben Ebene. Als Höhe eines Baumes bezeichnet man die höchste Ebenennummer.

Ein vollständiger, binärer Baum der Höhe $h$ hat $2^h$ Blätter, da sich mit jeder neuen Ebene die Anzahl der Blätter verdoppelt, und $2^{h+1}-1$ Knoten. Die Summe der Entfernungen jedes Knotens von der Wurzel wird als *Pfadlänge des Baumes* bezeichnet.

Da jeder Knoten eines Baumes als Wurzel betrachtet werden kann und jeder Knoten Wurzel eines Teilbaumes ist, kann man einen Baum auch rekursiv definieren: „Ein Baum ist entweder ein einzelner Knoten oder ein als Wurzel dienender Knoten, der mit einer Menge von Bäumen verbunden ist“[xxv].

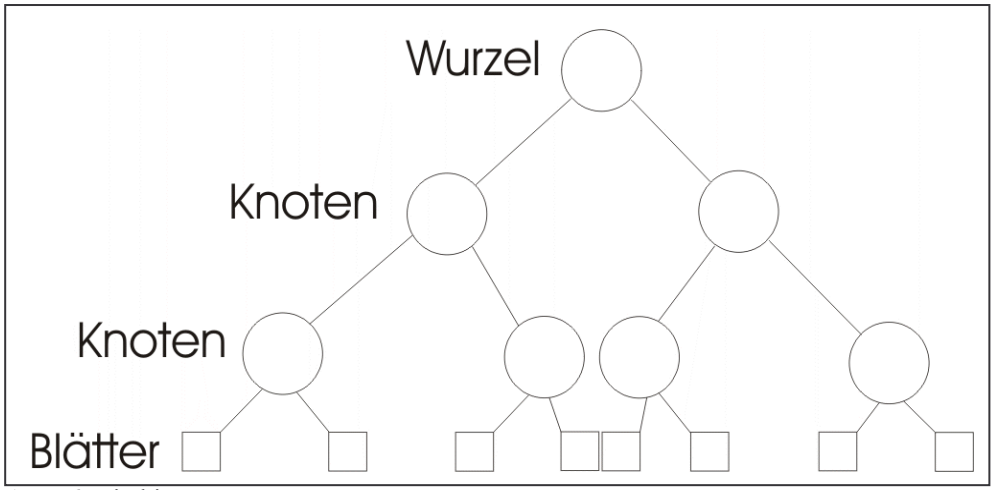


**Abb. 6:** Ein binärer Baum

# C Summen und Reihen[xxvi]

Möchte man Summen mit einer großen Anzahl an Summanden darstellen, so kann man dies auf verschiedene Arten tun. Die einfachste und einleuchtendste ist wohl folgende: $1+2+...+99+100$. So ist es nicht mehr schwer zu erraten, welche Summenden ausgelassen wurden. Auch kann man auf diese Weise Summen mit einer unendlichen Anzahl Summanden darstellen: $1+2+...+(n-1)+n$. $n$ gibt hier die Anzahl der Summanden an.

Eine Andere Art der Darstellung bietet das Summenzeichen(großes Sigma):

$\sum_{i=1}^{n} i$ ist eine abkürzende Schreibweise für $1+2+...+(n-1)+n$.

Das $i$ ist ein Index, der nacheinander verschiedene Werte annimmt. Der Startwert steht unter dem Summenzeichen, der Endwert darüber. Rechts daneben steht, was aufsummiert werden soll.

So summiert $\sum_{i=1}^{n} i$ alle Zahlen, $\sum_{i=1}^{n} i^2$ Quadratzahlen und $\sum_{i=1}^{n} \sqrt{i}$ Quadratwurzeln auf wobei $i$ die Werte von 1 bis $n$ annimmt.

Folgende Schreibweisen ist äquivalent: $\sum_{i=1}^{n} i \equiv \sum_{1 \le i \le n} i \equiv 1+2+...+(n-1)+n$

Bildet man zu einer gegebenen Folge $a_1, a_2, ..., a_n$ die *Partialsummenfolge*, d.h. eine Folge $s_1, s_2, ..., s_n$, für die gilt:

$s_1 = a_1$

$s_2 = a_1 + a_2$

$s_3 = a_1 + a_2 + a_3$

...

$s_n = a_1 + a_2 + ... + a_n = \sum_{i=1}^{n} a_i$

so nennt man diese Partialsummenfolge die *Reihe* der Folge[xxvii]. Hat die Reihe den Endwert $\infty$ so nennt man sie unendliche Reihe: $\sum_{i=1}^{\infty} i$. Wird eine Nullfolge summiert, kann eine unendliche Reihe auch einen endlichen Grenzwert haben. Ist ein endlicher Grenzwert vorhanden, so ist die Reihe konvergent, andernfalls divergent. Diesen endlichen Grenzwert einer Reihe bezeichnet man auch als Summe oder Wert der Reihe.

# D Beweis der Gauß'schen Summenformel[I]

**Behauptung:**

$$1+2+\ldots+(n-1)+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2} \quad \text{für } n \in N$$

**Beweis:**
Durch vollständige Induktion:

**Induktionsanfang:**

$n=1$

$$1=\sum_{i=1}^{1} i=\frac{1(1+1)}{2}$$

Für $n=1$ ist die Behauptung wahr.

**Carl Friedrich Gauß**(1777-1855) war ein deutscher Mathematiker, Astronom und Physiker.

**Info 7:** Carl Friedrich Gauß

**Induktionsvoraussetzung:**

Unter der Annahme, dass für $n \in N$ $\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$ gilt,

ist noch zu zeigen, dass auch $\sum_{i=1}^{n} i+(n+1)=\frac{(n+1)(n+2)}{2}$ gilt.

**Induktionsschritt:**

$$\begin{aligned}\sum_{i=1}^{n} i+(n+1) &= \frac{n(n+1)}{2}+(n+1)\\ &= \frac{n(n+1)+2n+2}{2}\\ &= \frac{n^2+n+2n+2}{2}\\ &= \frac{(n+1)(n+2)}{2}\end{aligned}$$

Somit gilt die Gauß'sche Summenformel für alle $n \in N$.

[I] Zu Carl Friedrich Gauß siehe o.V.: Wikipedia - Carl Friedrich Gauß, http://de.wikipedia.org/wiki/Carl_Friedrich_Gauß, Stand 29.04.06

# E Binäre Suche

Wer etwas in einem Wörterbuch nachschlagen will, der wird wohl kaum ganz vorne anfangen, wenn er das Wort „Zeppelin“ sucht. Auch dauert eine Suche in einem doppelt so dicken Wörterbuch nicht doppelt so lang, denn hat man einmal das Wörterbuch in der Mitte aufgeschlagen, so hat sich das Problem auf die Hälfte reduziert. Das alles ist aufgrund der Sortierung möglich. Wäre das Wörterbuch nicht sortiert, würde die Suche nach einem bestimmten Wort wohl ungleich länger dauern.

**Idee:**
Ganz ähnlich, wie die Suche in einem Wörterbuch funktioniert die binäre Suche oder auch BinarySearch: Bestimme das mittlere Element. Ist dieses das gesuchte, ist es gefunden. Ist das mittlere kleiner als das gesuchte, suche im rechten Teilfeld weiter, ansonsten im linken.

```
function BinarySearch(A: array of Integer; AItem: Integer);
var
  left: Integer;
  right: Integer;
  mid: Integer;
begin
  left := 0;
  right := High(A);
  while left <= right do
  begin
    mid := (left + right) div 2;
    if A[mid] = AItem then
    begin
      Result := mid; // gefunden
      exit;
    end;
    if A[mid] > AItem then
    begin
      right := mid -1;
    end
    else
    begin
      left := mid +1;
    end;
  end;
  Result := -1; // nicht gefunden
end;
```

## *Vergleiche*

**Behauptung:**
BinarySearch benötigt zum auffinden eines Elementes mindestens einen Vergleich und höchstens $V(n) = O(\log(n))$ Vergleiche.

**Beweis:**

**Best-case:**
Im besten Fall ist das gesuchte Element genau in der Mitte. Dann wird genau ein Vergleich ausgeführt(`if A[mid] = AItem then`) und die Suche beendet.

**Worst-case:**
Die binäre Suche wird als Binärbaum interpretiert. Ist dieser nicht vollständig(also $n \neq 2^h - 1$ mit $h \in N$), so nimmt man an, er wäre mit so genannten „Pseudodaten“ gefüllt. Wir gehen also von einem vollständigen Baum aus. Jeder Knoten entspricht einem Element des Arrays.

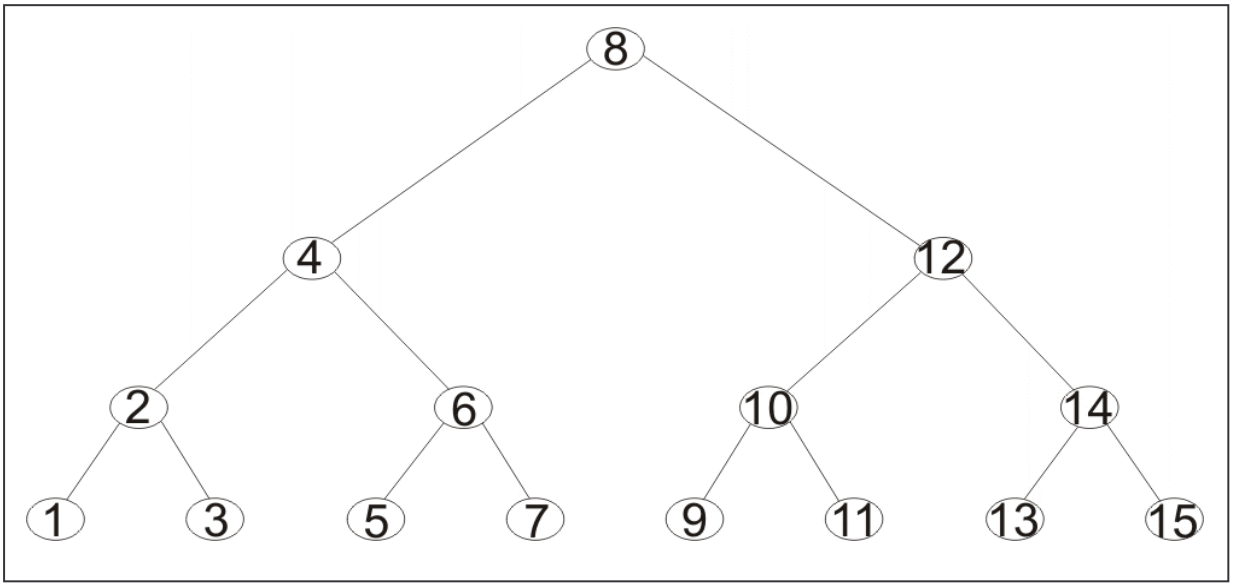



**Abb. 7:** Ein binärer Suchbaum mit einer sortierten Liste der ersten 15 natürlichen Zahlen

Ein vollständiger binärer Baum der Höhe h hat immer $2^{h+1}-1$ Knoten[I]. Daraus ergibt sich die Höhe eines solchen Baumes:

$$k = 2^{h+1} - 1$$
$$k + 1 = 2^{h+1}$$
$$ld(k+1) = h + 1$$
$$h = ld(k+1) - 1$$

Ist der Baum nicht vollständig, ist also $k+1$ keine Zweierpotenz, so ist die nächst kleinere Zweierpotenz entscheidend. Indem wir den Logarithmus von $k+1$ abrunden, erhalten wir den Logarithmus dieses Wertes:

$$h = \lfloor ld(k+1) \rfloor - 1$$

Im schlechtesten Fall muss die gesamte Höhe des Baumes durchlaufen werden, da sich der gesuchte Knoten in der untersten Ebene befindet.
Die Anzahl der nötigen Schleifendurchgänge beträgt dann $h+1 = \lfloor ld(n+1) \rfloor$, da für das Erreichen der Wurzel auch ein Schleifendurchgang notwendig ist.
In der while-Schleife werden zwei Vergleiche vorgenommen. Die Anzahl der Vergleiche beträgt also:

$$V(n) = 2(h+1) = 2(\lfloor ld(n+1) \rfloor) \approx 2 \cdot ld(n) \in O(\log(n))$$
$$V(n) \in O(\log(n))$$

**Gaußklammern:**
Tiefgestellte Klammern $\lfloor\ \rfloor$ geben an, dass der Wert auf die nächste ganze Zahl kleiner oder gleich dieses Wertes abgerundet werden soll. Benannt wurden diese Klammern nach Carl Friedrich Gauß.

Hochgestellte Klammern $\lceil\ \rceil$ bewirken entsprechend eine Aufrundung auf die nächste ganze Zahl größer oder gleich des Wertes.

**Info 8:** Gaußklammern

[I] Siehe Anhang B

# F Glossar

**Absteigende Sortierung:**
Sortierung einer Menge $a_1, a_2, ..., a_n$, für die für jedes $i, j \in N$ mit $i < j$ gilt: $a_i > a_j$

**Algorithmus:**
Eine Folge von Anweisungen zur Lösung eines bestimmten Problems. Siehe Info 6.

**Allgemeingültigkeit:**
Ein →Algorithmus muss algemeingültig sein. Siehe Info 6.

**Array:**
Datenfeld. Eine →Datenstruktur, bei der benachbarte Elemente nebeneinander im Speicher liegen. Siehe Kapitel 2.4.

**Asymptotisch dominant:**
Für $n \to \infty$ bestimmt dieser Teilterm das Ergebnis der Funktion. Konstante Faktoren werden dabei vernachlässigt. Siehe Kapitel 1.

**Asymptotisch obere Schranke:**
Für $n \to \infty$ nähert sich der wert einer Funktion dieser Schranke an, überschreitet diese jedoch nicht. Konstante Faktoren werden dabei vernachlässigt. Siehe Kapitel 1.

**Asymptotisch optimal:**
Ein Algorithmus der in Bezug auf seine Laufzeit die untere Schranke erreicht nennt man asymptotisch optimal. Siehe Kapitel 1.

**Asymptotisch scharfe Schranke:**
Fallen asymptotisch obere und asymptotisch untere Schranke zusammen, so spricht man von einer asymptotisch scharfen Schranke. Siehe Kapitel 1.

**Asymptotisch untere Schranke:**
Für $n \to \infty$ nähert sich der Wert einer Funktion dieser Schranke an, unterschreitet diese jedoch nicht. Konstante Faktoren werden dabei vernachlässigt. Siehe Kapitel 1.

**Asymptotisch vernachlässigbar.**
Für $n \to \infty$ hat dieser Teilterm für das Ergebnis kaum noch Relevanz. Konstante Faktoren werden dabei vernachlässigt. Siehe Kapitel 1.

**Aufsteigende Sortierung:**
Sortierung einer Menge $a_1, a_2, ..., a_n$, für die für jedes $i, j \in N$ mit $i < j$ gilt: $a_i < a_j$

**Aufwand:**
Der Verbrach an Systemressourcen(Rechenzeit, Speicherplatz), die ein →Algorithmus benötigt.

**Ausführbarkeit:**
Ein →Algorithmus muss prinzipiell ausführbar sein. Siehe Info 6.

**Austausch:**
Das Vertauschen zweier Werte in einer Menge.

**Average-case:**
Der durchschnittliche Fall(von eng. average *Durchschnitt*; case *Fall*). Oft werden für die Untersuchung von Algorithmen drei Fälle unterschieden: best-case, worst-case und average-case. Der Abverage-case entspricht dem Fall, der wohl am Häufigsten auftritt bzw. dem der bei einer durchschnittlichen Eingabe resultiert.

**Baum:**
→Datenstruktur zur Organisation und Darstellung. Siehe Anhang B.

**Best-case:**
Der bestmögliche Fall(von eng. best *am besten*; case *Fall*) . Oft werden für die Untersuchung von Algorithmen drei Fälle unterschieden: best-case, worst-case und average-case. Der best-case tritt relativ selten und nur bei bestimmten Eingabedaten ein. Dann jedoch funktioniert der gegebene Algorithmus optimal.

**Binäre Suche:**

Suchverfahren, das dem Suchen in einem Wörterbuch entspricht. Siehe Anhang E.

**Binärer Baum:**
Baum, dessen Knoten entweder zwei oder keine Nachfolger haben. Siehe Anhang B.

**BinaryInsertion:**
Sortieralgorithmus, der eine Variante von InsertionSort darstellt. Zum ermitteln der Einfügeposition wird die Binärsuche verwendet. Siehe Kapitel 3.1.3.

**BinarySearch:**
Der →binäre Suchalgorithmus. Siehe Anhang E.

**Blatt:**
Ein Knoten eines Baumes, der keinen Nachfolger hat. Siehe Anhang B

**BogoSort:**
Der Prototyp eines schlechten Algorithmus. Siehe Kapitel 3.3.

**BubbleSort:**
Ein elementarer, quadratischer Sortieralgorithmus. Siehe Kapitel 3.1.2.

**Compiler:**
Programm zum Übersetzen von Programmquellcode in Maschinensprache.

**Datenfeld:**
Ein →Array.

**Datenstruktur:**
Eine Art Daten in digitaler Form zu organisieren.

**Dekrementieren:**
Eine Variable um eins vermindern.

**Eindeutigkeit:**
Ein Algorithmus darf keinen Interpretationsspielraum haben, muss also eindeutig sein. Siehe Info 6.

**Elementares Sortierverfahren:**
Sortierverfahren, welches einen Grundgedanken umsetzt, sich jedoch um die Simplizität zu bewahren darauf beschränkt.

**Endlichkeit:**
Ein →Algorithmus muss sich mit einer endlichen Anzahl an Anweisungen beschreiben lassen können. Siehe Info 6.

**Esoterische Programmiersprache:**
Eine Programmiersprache, deren Ziel es ist einen möglichst kleinen →Compiler zu haben. Esoterische Programmiersprachen haben keinen besonderen Sinn; sie werden zum Spaß, oder zu Übung geschrieben.

**Externs Sortierverfahren:**
Ein Sortierverfahren, das zum Sortieren ein weiteres →Array benötigt. Siehe Anhang A.

**Feld:**
Ein →Array.

**For-Schleife:**
Eine →Schleife, die Anweisungen mit einer bestimmten Anzahl an Wiederholungen durchläuft.

**Gauß, Carl Friedrich:**
Bedeutender deutscher Mathematiker. Siehe Info 7.

**Gaußklammern:**
Klammerung, die in einer Abrundung auf die nächst kleinere ganze Zahl resultiert. Siehe Info 8.

**Heap:**
1. Eine →Datenstruktur. 2. Eine Organisationsform des Arbeitsspeichers.

**HeapSort:**
Ein →asymptotisch optimales Sortierverfahren.

**Implementierung:**
Die Umsetzung eines Algorithmus in Programmcode.

**Inkrementieren:**
Eine Variable um eins erhöhen.

**In-place:**
Bezeichnung für ein →internes Sortierverfahren.

**InsertionSort:**
Ein →elementares Sortierverfahren.

**Internes Sortierverfahren:**
Ein Sortierverfahren, das nur im gegebenen →Array abreitet. Siehe Anhang A.

**Inversion:**
Vertauschung von Elementen in einem →Array. Siehe Info 4.

**Inversionsfolge:**
Eine Folge, die für jedes Element im →Array die Anzahl der →Inversionen angibt. Siehe Info 4.

**Iterativität:**
Eigenschaft eines Algorithmus, der →Schleifen verwendet.

**Iterieren:**
In einer →Schleife durchlaufen-

**Kante:**
Verbindung zwischen zwei →Knoten. Siehe Anhang B.

**Knoten:**
Informationsobjekt eines →Baumes. Siehe Anhang B.

**Komplexität:**
→Laufzeit eines →Algorithmus.

**Landau-Symbole:**
Symbole zur Notation der →Ordnung eines →Algorithmus.

**Laufvariablen:**
Variable, die in einer Schleife →inkrementiert(oder →dekrementiert) wird um die Nummer des Schleifendurchgangs anzugeben.

**Laufzeit:**
→Aufwand an Rechenzeit, die ein →Algorithmus benötigt.

**Lineare Suche:**
Auch sequentielle Suche. Ein Suchalgorithmus, der alle Elemente einer Liste durchgeht, bis das gesuchte Element gefunden wurde. Siehe Info 5.

**Linked List:**
Eine →Verkettete Liste.

**MergeSort:**
Ein →asymptotisch optimales Sortierverfahren.

**MinSort:**
→SelectionSort.

**Nihilartikel:**
Ein Artikel in einem Lexikon zu einem fiktiven Stichwort.

**Obere Schranke:**
→asymptotisch obere Schranke.

**O-Notation:**
→Landau-Symbole.

**Optimierung:**
Veränderung eines →Algorithmus mit dem Ziel ihn zu verbessern.

**Ordnung:**
Asymptotische Laufzeit eines Algorithmus. Siehe Kapitel 1.

**Out-of-place**
Bezeichnung für ein →externes Sortierverfahren.

**Pessimal Algorithms and Simplexity Analysis:**
Nicht ganz ernst gemeinte Publikation von Andrei Broder und Jorge Stolfi über schlechte Algorithmen. Siehe Kapitel 3.3.

**Pfad:**
Weg von einem →Knoten über →Kanten und ggf. andere Knoten zur →Wurzel. Siehe Angang B.

**Pivotelement:**
Vergleichselement, das QuickSort benötigt um die zu sortierende Menge in zwei Teilmengen aufzuteilen. Siehe Kapitel 3.2.

**Pointer:**

Zeiger. Eine Zahl, die die Speicheradresse von Daten(oder Code) symbolisiert und so auf die Daten(bzw. den Code) „zeigt“.

**QuckSort:**
Ein →asymptotisch optimaler →rekursiver Sortieralgorithmus. Siehe Kapitel 3.2.

**RandomSort:**
→BogoSort.

**Reihe:**
Folge der Partialsummen einer Folge. Siehe Anhang C.

**Rekurrente Beziehung:**
Eine Funktion, die den Funktionswert für ein anderes x benötigt. Beispiel:
$f(x) = x \cdot f(x-1)$ für $x \in N; x > 1; f(1) = 1$
(Fakultät)

**Rekursivität:**
Eigenschaft eines Algorithmus, die beschreibt, dass der Algorithmus sich selbst aufruft. Siehe Anhang A.

**Repeat-Schleife:**
Eine →Schleife, die nach dem Ausführen derAnweisungen auf eine Abbruchbedingung prüft. Eine repeat-Schleife wird also mindestens einmal durchlaufen.

**Rücksprungadresse:**
Eine Speicheradresse(siehe →Pointer), die angibt, an welche Speicheradresse das Programm nach Ausführung eines rekursiven Aufrufs zurückspringen soll.

**Scharfe Schranke:**
→asymptotisch scharfe Schranke.

**Schleife:**
Ein Programmierkonstrukt, welches bestimmte Anweisungen wiederholt und bei einer definierten Abbruchbedingung abbricht.

**Schleifenkopf:**
Der teil einer →Schleife, in dem sich normalerweise die Abbruchbedingung befindet.

**SelectionSort:**
Ein →elementares Sortierverfahren. Siehe Kapitel 3.1.1.

**Sequentielle Suche:**
→Lineare Suche.

**ShellSort:**
Ein asymptotisch guter Sortieralgorithmus.

**Simplizität:**
Die „Einfachheit“ eines Algorithmus. Eine hohe Simplizität bedeutet geringen Aufwand bei der →Implementierung. Siehe Anhang A.

**SkatSort:**
Ein →elementares Sortierverfahren. →InsertionSort, welches sich der →linearen Suche bedient. Siehe Kapitel 3.1.3.

**Sortierter Bereich:**
Manche Sortieralgorithmen erzeugen im gegebenen →Array eine Teilmenge, die sortiert ist. Siehe Anhang A.

**Stabilität:**
Ein Sortierverfahren, welches gleichrangige Elemente nicht vertauscht, heißt stabil. Siehe Anhang A.

**Stack:**
Organisationsform des Arbeitsspeichers. Auf dem Stack liegen alle Parameter einer Funktion, Rücksprungadressen für rekursive Funktionen, sowie Variablen, die nur in der jeweiligen Funktion gültig sind(lokale Variablen).

**StupidSort:**
→BogoSort.

**Terminiertheit:**
Eigenschaft eines Algorithmus. Ein Algorithmus muss in endlicher Zeit eine Lösung des Problems liefern. Siehe Info 6.

**Umgekehrte Sortierung:**
Soll eine Liste aufsteigend sortiert werden so ist die umgekehrte Sortierung die absteigende und umgekehrt.

**Untere Schranke:**

→asymptotisch untere Schranke.

**Verkettete Liste:**
Eine →Datenstruktur, die Darauf beruht, dass jedes Element einen →Pointer auf das nächste enthält. Siehe Kapitel 2.4.

**Vollständiger Baum:**
Ein →Baum, dessen →Blätter sich alle auf der selben Ebene befinden. Siehe Anhang B.

**While-Schleife:**
Eine →Schleife, bei der die Abbruchbedingung ganz zu Anfang geprüft wird. Es ist also möglich, dass die Anweisungen kein einziges Mal ausgeführt werden.

**Worst-case:**
Der schlechteste Fall(von eng. worst *am schlechtesten*; case *Fall*). Oft werden für die Untersuchung von Algorithmen drei Fälle unterschieden: best-case, worst-case und average-case. Der worst-case entspricht dem Fall, der im langsamsten Laufzeitverhalten resultiert.

**Wurzel:**
Der „Ursprungsknoten“ eines →Baumes. Siehe Anhang B.

**Zeiger:**
Ein →Pointer.

**Zuweisung:**
Setzen des Wertes einer Variablen(insbesondere eines Elementes in einem →Array).

# G Quellen


- Hormann, Kai: Praktische / Angewandte Informatik 1, http://www.in.tu-clausthal.de/~hormann/teaching/PA1WS04/PA1.02.12.2004.pdf, Stand 02.04.06
- Hormann, Kai: Praktische / Angewandte Informatik 1, http://www.in.tu-clausthal.de/~hormann/teaching/PA1WS04/PA1.09.12.2004.pdf, Stand 11.07.05
- Koppehel, Sebastian: Suchen und Sortieren, http://www.bastisoft.de/pascal/suso.html, Stand 26.11.05
- Lang, H.W.: Sequentielle und parallele Sortierverfahren - Insertionsort, http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/insert/insertion.htm, Stand: 18.04.06
- o.V.: Das große Tafelwerk Interaktiv. Formelsammlung für die Sekundarstufen I und II. Berlin [1]2003
- o.V.: Informatik III: Programmierungstechnik. Sortieralgorithmen, http://wwwicg.informatik.uni-rostock.de/Lehre/Informatik3/SS2006/Skript/07_Sortieralgorithmen.pdf, Stand 11.07.05
- o.V.: Laufzeitkomplexität von Algorithmen - die O-Notation, http://www.linux-related.de/index.html?/coding/o-notation.htm, Stand 08.04.06
- o.V.: Wikipedia – Asymptotische Analyse, http://de.wikipedia.de/wiki/AsymptotischeAnalyse, Stand 07.03.06
- o.V.: Wikipedia – Bogosort, http://de.wikipedia.de/wiki/ Bogosort, Stand 07.03.06
- o.V.: Wikipedia – Bubblesort, http://de.wikipedia.de/wiki/ Bubblesort, Stand 07.03.06
- o.V.: Wikipedia – Edmund Landau, http://de.wikipedia.de/wiki/Edmund_Landau, Stand 20.04.06
- o.V.: Wikipedia – In-place, http://de.wikipedia.de/wiki/ In-place, Stand 07.03.06
- o.V.: Wikipedia – Insertionsort, http://de.wikipedia.de/wiki/Insertionsort, Stand 07.03.06
- o.V.: Wikipedia – Komplexität_(Informatik), http://de.wikipedia.org/wiki/Komplexität_(Informatik) , Stand 30.04.06
- o.V.: Wikipedia – Komplexitätstheorie, http://de.wikipedia.de/wiki/Komplexitätstheorie, Stand 11.03.06
- o.V.: Wikipedia – Landau-Symbole, http://de.wikipedia.de/wiki/Landau-Symbole, Stand 07.03.06
- o.V.: Wikipedia – Laufzeit (Informatik), http://de.wikipedia.de/wiki/Laufzeit_(Informatik) , Stand 20.03.06
- o.V.: Wikipedia – Out-of-place, http://de.wikipedia.de/wiki/Out-of-place, Stand 07.03.06
- o.V.: Wikipedia – Paul Bachman, http://de.wikipedia.de/wiki/Paul_Bachmann, Stand 20.04.06
- o.V.: Wikipedia – Quicksort, http://de.wikipedia.de/wiki/Quicksort, Stand 07.03.06
- o.V.: Wikipedia – Selectionsort, http://de.wikipedia.de/wiki/Selectionsort, Stand 07.03.06
- o.V.: Wikipedia – Sortierverfahren, http://de.wikipedia.de/wiki/sortierverfahren
- o.V.: Wikipedia – Stabiles Sortierverfahren, http://de.wikipedia.de/wiki/ Stabiles_Sortierverfahren, Stand 07.03.06
- o.V.: Wikipedia – Summe, http://de.wikipedia.de/wiki/summe, Stand 11.03.06
- o.V.: Wikipedia – Theoretische Informatik, http://de.wikipedia.de/wiki/theoretische_informatik/, Stand 20.03.06
- Sedgewick, Robert: Algorithmen. München [2]2002
- Stegmaier, Karl Ludwig: Suchen, Sortieren, Aufwand. Bad Kreuznach [1]1993. (=PZ-Information Informatik. Heft 24)
- Thielemann, Henning, Klein, aber O, http://www.math.uni-bremen.de/~thielema/Study/Landau/landau.pdf, Stand 07.04.06
- Weigel, Peter: Allgemeine und spezielle Sortieralgorithmen + Suchalgorithmen, http://www.sortieralgorithmen.de, Stand 04.05.03
- Weihe, Karsten: Skript zu Informatik II im Sommersemester 1999, http://www.ub.uni-konstanz.de/kops/volltexte/1999/281/pdf/281_1.pdf, Stand 11.07.05
- Wolf, Daniel R.: Tutorial: Sortier-Algorithmen I+II, http://www.delphipraxis.net/topic344.html, Stand 07.04.06

## H Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Facharbeit selbstständig und ausschließlich unter Benutzung der angegebenen Quellen erstellt habe.

[i] Vgl. Hormann, Kai: Praktische / Angewandte Informatik 1, http://www.in.tu-clausthal.de/~hormann/teaching/PA1WS04/PA1.02.12.2004.pdf, Stand 02.04.06
[ii] Vgl. Sedgewick, a.a.O., S. 99
[iii] Vgl. o.V.: .: Wikipedia – Landau-Symbole, http://de.wikipedia.org/wiki/Landau-Symbole, Stand 07.03.06
[iv] Vgl. Thielemann, Henning: Klein, aber O, http://www.math.uni-bremen.de/~thielema/Study/Landau/landau.pdf, Stand 07.04.06
[v] Vgl. Sedgewick, a.a.O., S.99f.
[vi] Vgl. Stegmaier, Karl Ludwig: Suchen · Sortieren · Aufwand. Bad Kreuznach [1]1993. (=PZ-Information Informatik. Heft 24), S. 21
[vii] Vgl. Stegmaier, a.a.O., S. 21
[viii] Vgl. Stegmaier, a.a.O., S. 21
[ix] Vgl. o.V.: Wikipedia – Sortieralgorithmen, , http://de.wikipedia.de/wiki/Sortierverfahren, Stand 07.03.06
[x] Vgl. Stegmaier, a.a.O., S. 22
[xi] Vgl. o.V.: .: Wikipedia - Selectionsort, http://de.wikipedia.de/wiki/Selectionsort, Stand 07.03.06; Weigel, Peter: Allgemeine und spezielle Sortieralgorithmen + Suchalgorithmen – SelectSort, MinSort, http://www.sortieralgorithmen.de/selectsort/index.html, Stand 04.05.03; Sedgewick, a.a.O., S. 125ff.; Wolf, Daniel R.: Tutorial: Sortier-Algorithmen I+II, http://www.delphipraxis.net/topic344.html, Stand 07.04.06
[xii] Vgl. Stegmaier, a.a.O., S.25
[xiii] Vgl. o.V.: .: Wikipedia - BubbleSort, http://de.wikipedia.de/wiki/Bubblesort, Stand 07.03.06; Weigel, Peter: Allgemeine und spezielle Sortieralgorithmen + Suchalgorithmen – BubbleSort, http://www.sortieralgorithmen.de/bubblesort/index.html, Stand 04.05.03; Sedgewick, a.a.O., S. 129f. ; Wolf, Daniel R.: Tutorial: Sortier-Algorithmen I+II, http://www.delphipraxis.net/topic344.html, Stand 07.04.06
[xiv] Vgl. o.V.: .: Wikipedia - InsertionSort, http://de.wikipedia.de/wiki/Insertionsort, Stand 07.03.06; Weigel, Peter: Allgemeine und spezielle Sortieralgorithmen + Suchalgorithmen – InsertSort, http://www.sortieralgorithmen.de/insertsort/index.html, Stand 04.05.03; Sedgewick, a.a.O., S. 127ff. ; Wolf, Daniel R.: Tutorial: Sortier-Algorithmen I+II, http://www.delphipraxis.net/topic344.html, Stand 07.04.06
[xv] Vgl. Lang, H.W.: Sequentielle und parallele Sortierverfahren - Insertionsort, http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/insert/insertion.htm, Stand: 18.04.06
[xvi] Vgl. o.V.: .: Wikipedia - QuickSort, http://de.wikipedia.de/wiki/Quicksort, Stand 22.04.06; Weigel, Peter: Allgemeine und spezielle Sortieralgorithmen + Suchalgorithmen – QuickSort, http://www.sortieralgorithmen.de/quicksort/index.html, Stand 04.05.03; Hormann, Kai: Praktische / Angewandte Informatik 1, http://www.in.tu-clausthal.de/~hormann/teaching/PA1WS04/PA1.09.12.2004.pdf, Stand 11.07.05, S. 4; Sedgewick, a.a.O., S. 145-161. ; Wolf, Daniel R.: Tutorial: Sortier-Algorithmen I+II, http://www.delphipraxis.net/topic344.html, Stand 07.04.06
[xvii] Vgl. o.V.: .: Wikipedia - BogoSort, http://de.wikipedia.de/wiki/Bogosort, Stand 29.04.06; o.V.: .: Wikipedia - BogoSort, http://en.wikipedia.de/wiki/BogoSort, Stand 29.04.06; Raymond, Eric S.: BogoSort, http://www.catb.org/~esr/jargon/html/B/bogo-sort.html, Stand 29.04.06
[xviii] Vgl. o.V.: Wikipedia – SlowSort, http://de.wikipedia.org/wiki/Slowsort, Stand 18.04.06
[xix] Vgl. o.V.: Das große Tafelwerk Interaktiv. Formelsammlung für die Sekundarstufen I und II. Berlin [1]2003, S. 78
[xx] Vgl. o.V.: Tafelwerk, a.a.O., S.37
[xxi] Vgl. Sedgewick, Robert: Algorithmen. München [2]2002, S. 123
[xxii] Vgl. Sedgewick, a.a.O., S. 122; o.V.: Wikipedia – In-place, http://de.wikipedia.de/wiki/In-place, Stand 07.03.06; o.V.: Wikipedia – Out-of-place, http://de.wikipedia.de/wiki/Out-of-place, Stand 07.03.06
[xxiii] http://www.sortieralgorithmen.de/definitions.htm 12.02.06
[xxiv] Vgl. Sedgewick, a.a.O., S. 57-63
[xxv] Sedgewick, a.a.O., S. 61
[xxvi] Vgl: o.V.: Wikipedia - Summe, http://de.wikipedia.de/wiki/Summe, Stand 20.04.06
[xxvii] Vgl. o.V.: Tafelwerk, a.a.O., S.52