Sortieralgorithmen

Rafael Ostertag

2006-11-12

Zusammenfassung

In diesem Dokument werden die einfachen Sortieralgorithmen wie Bubble Sort, Selection Sort, Insertion Sort und Shell Sort auf einer sehr abstrakten Ebene kurz vorgestellt. Für das Verständis sind keinerlei Programmierkenntnisse nötig. Kompliziertere Sortieralgorithmen wie Quick Sort oder Merge Sort werden nicht behandelt. Es wird ebenso wenig gezeigt, wie die erwähnten Algorithmen in einer Programmiersprache implementiert werden können. Dafür sei auf [1] und [4] verwiesen.

Inhaltsverzeichnis

1 Was ist sortieren? 2
1.1 Stabiles und unstabiles Sortieren 4

2 Wozu wird das Sortieren gebraucht? 4

3 Der sortierende Computer 5

4 Sortieralgorithmen 6
4.1 Bubble Sort 7
4.2 Selection Sort 8
4.3 Insertion Sort 10
4.4 Shell Sort 12

5 Fazit 15

A Code-Beispiele in C++ 16
A.1 Bubble Sort 16
A.2 Selection Sort 16
A.3 Insertion Sort 16
A.4 Shell Sort 17

Abbildungsverzeichnis

1 Die Aufteilung von Speicher in gleich grosse Bereiche 6
2 Bubble Sort illustriert 7
3 Performance von Bubble Sort 8
4 Selection Sort illustriert 9
5 Performance von Selection Sort 9
6 Insertion Sort Vorbereitung illustriert 10
7 Insertion Sort illustriert 11
8 Performance von Insertion Sort 11
9 Performance von Shell Sort 13
10 Shell Sort illustriert 14
11 Performance im Vergleich 15

Tabellenverzeichnis

1 Sortieren von Spielkarten nach einem Schlüssel 3
2 Sortieren von zufällig generierten Zahlen 3
3 Stabiles und unstabiles Sortieren 4

1 Was ist sortieren?

Ein loser Stapel Blätter wird den Seitenzahlen nach sortiert und das Kartenspiel entsprechend dem Wert der Karten. Die Beispiele zeigen, dass der Vorgang des Sortieren eigentlich trivial und intuitiv ist. Doch nur schon ein Kartenspiel mit 52 Karten ist mühsam zu sortieren und eine Liste mit über hundert Namen möchte man am liebsten von jemand anderem sortiert haben. Man stelle sich auch vor, wie anstrengend es wäre, in einer Liste aus hundert Namen den gesuchten zu finden, wenn die Liste nicht sortiert ist. So ist es naheliegend, dem Computer das Sortieren zu lehren.

Sortieren heisst in einem Satz erklärt, eine Menge Dinge in eine geordnete Reihenfolge zu bringen. Donald E. Knuth [1] erklärt den Vorgang des Sortierens jedoch so (frei übersetzt):

> [...]
> Es sind N Elemente gegeben
>
> $$R_1, R_1, \ldots, R_N$$
>
> welche sortiert werden sollen. Wir nennen sie Datensätze. Die Menge aller Datensätze wird Datei genannt. Jeder Datensatz R_j hat einen Schlüssel K_j, welcher den Sortiervorgang bestimmt. [...]
>
> Eine Ordnungsbeziehung “$<$” wird für die Schlüssel festgelegt, so dass die folgenden Bedingungen für jeden Schlüsselwert a, b, c erfüllt sind:
>
> 1. Exakt eine der Möglichkeiten $a < b$, $a = b$, $b < a$ ist wahr. ...
> 2. Wenn $a < b$ und $b < c$, dann ist $a < c$.
>
> [...]

Diese kompliziert anmutende Erklärung ist aber im Grunde einfach und naheliegend. Wenn etwas sortiert werden soll, müssen drei Bedingungen erfüllt sein:

1. Die Dinge, welche sortiert werden sollen, nachfolgend *Elemente* genannt, müssen in einer Beziehung zueinander stehen.
2. Es muss ein Kriterium geben, welches erlaubt, Elemente einem Wert entsprechend zu sortieren, der *Schlüssel*.
3. Es muss möglich sein zu bestimmen, welcher von zwei Schlüsseln den grösseren Wert hat.

Birnen, Äpfel und Zwetschgen stehen als Früchte zueinander in Beziehung. Ebenso stehen Bube, Dame und König zueinander in Beziehung, nämlich als Spielkarten. Beides erfüllt also die erste Bedingung. Die zweite Bedingung zeigt aber, dass Birnen, Äpfel und Zwetschgen nicht sortiert werden können, den es wird schwierig ein Kriterium für die Sortierreihenfolge festzulegen.[1] Bube, Dame und König erfüllen jedoch die zweite Bedingung, denn sie haben je einen Wert, der als Sortierkriterium, also Schlüssel, brauchbar ist. Bube hat den Wert zehn, Dame den Wert elf und der König den Wert zwölf. Auch die dritte Bedingung wird erfüllt, denn aus der Arithmetik ist bekannt, dass $10 < 11$ und $11 < 12$.

Hier noch zwei Beispiele, um den Sachverhalt des Sortierens zu illustrieren.

Spielkarten sortieren Es sollen die nachfolgenden Spielkarten sortiert werden. Dabei lassen wir die Farben der Karten (Herz, Karo, Pik und Kreuz) ausser acht, und konzentrieren uns nur auf die Werte (Tabelle 1).

Unsortiert		Sortiert	
Element	Schlüssel	Element	Schlüssel
König	12	Bube	10
Ass	13	Dame	11
Bube	10	König	12
Dame	11	Ass	13

Tabelle 1: Sortieren von Spielkarten nach einem Schlüssel

Zufallszahlen sortieren Eine Reihe von zufällig generierten Zahlen soll sortiert werden (Tabelle 2).

Unsortiert	Sortiert
39	6
95	36
56	39
77	40
53	53
6	56
36	77
40	95

Tabelle 2: Sortieren von zufällig generierten Zahlen

[1] Eine Möglichkeit wäre natürlich, sie der Süssigkeit entsprechend zu sortieren. Jedoch ist dies eine subjektive Wahrnehmung und soll hier nicht betrachtet werden.

1.1 Stabiles und unstabiles Sortieren

Am Rande soll noch kurz erklärt werden, was *stabiles* und *unstabiles* Sortieren bedeutet.

Die Unterscheidung zwischen stabilem und unstabilem Sortieren kommt dann zum Tragen, wenn die zu sortierenden Elemente Duplikate, also den selben Schlüssel mehrfach, enthalten. Als Beispiel dient uns eine Liste von Namen (Tabelle 3).

Bei einer stabilen Sortierung behalten die Elemente ihre relative Position. Bei einer unstabilen Sortierung geht die raltive Position hingegen verloren. Siehe dazu auch [2] und [5].

Liste		Unstabil		Stabil	
Element	Schlüssel	Element	Schlüssel	Element	Schlüssel
Adams	1	Adams	1	Adams	1
Black	2	Smith	1	Smith	1
Brown	4	Washington	2	Black	2
Jackson	2	Jackson	2	Jackson	2
Jones	4	Black	2	Washington	2
Smith	1	White	3	White	3
Thompson	4	Wilson	3	Wilson	3
Washington	2	Thompson	4	Brown	4
White	3	Brown	4	Jones	4
Wilson	3	Jones	4	Thompson	4

Tabelle 3: Stabiles und unstabiles Sortieren

Die Frage, wie der Computer sortieren kann, wird im Abschnitt 3 auf der nächsten Seite erläutert.

2 Wozu wird das Sortieren gebraucht?

Nachdem wir uns im letzten Abschnitt eine ungefähre Ahnung davon verschaft haben, was sortieren ist, wollen wir hier der Frage nachgehen, wozu das Sortieren gebraucht werden kann.

Telefonbücher wären zum Beispiel kaum zu gebrauchen, wenn sie nicht alphabetisch sortiert wären. Ebenso würden die Suchergebnisse von Google nicht viel hergeben, wenn sie nicht nach Relevanz sortiert würden. Statistische Funktionen wie Minimum und Maximum wären ineffizient, würden die Daten nicht in irgendeiner Weise sortiert, um den grössten und kleinsten Wert zu erhalten und Computer Algebra Systeme wie Maple, Mathematica oder graphische Taschenrechner stünden vor einem grossen Problem, wenn die Terme nicht nach Potenzen sortiert werden könnten. Weniger offensichtliche Beispiele sind riesige Datenbanken wie Wikipedia, die um vieles langsamer wären, würden die Daten nicht in einer Form abgelegt, die eine Sortierung aufweist, um die Suche nach Artikeln zu beschleunigen.

Überdies besteht die einfachste Möglichkeit, aus einer Liste von Einträgen doppelte zu entfernen darin, die Liste zuerst zu sortieren und danach der Reihe nach alle durchzugehen und den aktuellen Eintrag mit dem nächsten zu vergleichen. Sind zwei aufeinanderfolgende Einträge identisch, wird einer davon entfernt.

3 Der sortierende Computer

Um der Frage auf den Grund zu gehen, wie man dem Computer das Sortieren lehrt, muss man sich vor Augen halten, dass der Computer nur mit Zahlen umgehen kann. Dabei werden die ganzen Zahlen bevorzugt behandelt, denn sie lassen sich ohne grosse Umstände binär kodieren.[2] Selbstverständlich kann der Computer auch mit Dezimalbrüchen umgehen, jedoch ist ein grösserer Aufwand erforderlich, um mit diesen Zahlen zu rechnen und sie in vernünftiger Form darzustellen.

Das ist wichtig zu wissen, denn wenn der Computer nur mit Zahlen umgehen kann, wie ist es denn möglich, eine Liste mit Namen zu sortieren? Dazu muss man wissen, dass eben alle Zeichen und Buchstaben systemintern als Zahlenwerte dargestellt werden. Zum Beispiel hat der Buchstabe 'A' den Wert 65 und der Buchstabe 'B' den Wert 66 usw.[3] Somit ist es möglich herauszufinden, welcher von zwei Buchstaben der "grössere" ist.

Wir haben nun gesehen, dass der Computer alles nur als Zahlen behandelt und wenden uns nun der Organisation dieser Zahlen zu. Denn es muss irgendeine Möglichkeit geben, diese Zahlen oder Werte zu speichern und sie wieder abzufragen.

Eine gängige Art, wenn auch nicht die einzige, etwas zu speichern, besteht darin, einen zusammenhängenden Bereich (Speicher) zu reservieren und diesen dann in Unterbereiche aufzuteilen. Diese Unterbereiche haben eine feste Grösse, und können daher nur Werte bis zu einer bestimmten Grösse aufnehmen, zum Beispiel von 0–255 (siehe Abbildung 1 auf der nächsten Seite).[4]

Diese Methode erlaubt nun, über den Index auf die Elemente im Speicher zuzugreifen. Dies ist äussert nützlich, denn beim Sortieren müssen die Elemente gelesen, vertauscht, sowie geschrieben werden. Unter Vertauschen versteht man eine Operation oder Funktion, die mathematisch ausgedrückt folgendes macht:

$$a = 1; \quad b = 2$$

[2]Bekanntlich kennt der Computer nur die Zustände Strom *aus* und Strom *ein*, also die Werte 0 und 1. Wenn man nun mehrere solche Zustände hintereinander hängt, lassen sich auch grössere Zahlen als 0 und 1 darstellen. Mathematisch ausgedrückt sieht das so aus: $a_1 \cdot 2^0 + a_2 \cdot 2^1 + \cdots + a_n \cdot 2^{(n-1)}$, wobei a_n entweder 0 oder 1 ist und $n \in \mathbb{N}_0$.

[3]Das gilt nur für Computer die den ASCII-Zeichensatz verwenden. Es ist durchaus möglich, dass eine andere Art von Zeichensatz verwendet wird.

[4]Als Analogie kann man sich einen quadratischen Setzkasten vorstellen, der in viele gleichgrosse, quadratische Fächer aufgeteilt ist.

Speicher

Elemente	6	20	13	4	17	2	9	14	19	3	7
Index	1	2	3	4	5	6	7	8	9	10	12

Abbildung 1: Die Aufteilung von Speicher in gleich grosse Bereiche

es wird eine neue (temporäre) Variable z eingeführt

$$z = a$$

und die Werte vertauscht

$$a = b; \quad b = z$$

Somit hat nach dem Vertauschen a den Wert 2 und b den Wert 1.

Ich möchte hierauf nicht weiter eingehen, denn das würde uns schon sehr nahe an die eigentliche Programmierung heranführen. Interessierte seien auf den Abschnitt A auf Seite 16 verwiesen, der einige Code-Beispiele in C++ enthält.

Nachdem kurz die Datenorganisation angerissen wurde, fehlt nur noch der eigentliche Algorithmus, der das Sortieren übernimmt. Seit der Zeit in der die elektronische Datenverarbeitung Einzug gehalten hat, wurden viele Sortieralgorithmen entwickelt, zum Teil für allgemeine Anwendungen und zum Teil für sehr spezifische Anwendungen. Dies erlaubt dem Entwickler einer Software, gezielt einen Algorithmus auszuwählen, um dem Problem der Sortierung am besten gerecht zu werden.

In diesem Dokument werden aber nur die einfachsten Algorithmen behandelt. Als weiterführende Literatur sei auf [4] und [1] verwiesen.

Bevor wir die Sortieralgorithmen besprechen, möchte ich noch kurz die wichtigsten Punkte dieses Abschnitts aufzeigen:

- Der Computer kann nur mit Zahlen umgehen
- Die Elemente oder Daten müssen in einer geeigneten Form vorliegen
- Es muss möglich sein, Elemente zu vertauschen
- Die Daten müssen mit Hilfe eines Sortieralgorithmus sortiert werden

4 Sortieralgorithmen

Dieser Abschnitt befasst sich mit den einfachen Sortieralgorithmen *Bubble Sort*, *Insertion Sort*, *Selection Sort* und *Shell Sort*. Sie werden auch elementare Sortieralgorithmen genannt und zeichnen sich durch ihre Einfachheit aus, einmal abgesehen von Shell Sort, welcher sich eines Tricks bedient.

4.1 Bubble Sort

Bubble Sort ist der einfachste Sortieralgorithmus, allerdings auch der langsamste. Die Abbildung 2 illustriert den Algorithmus.

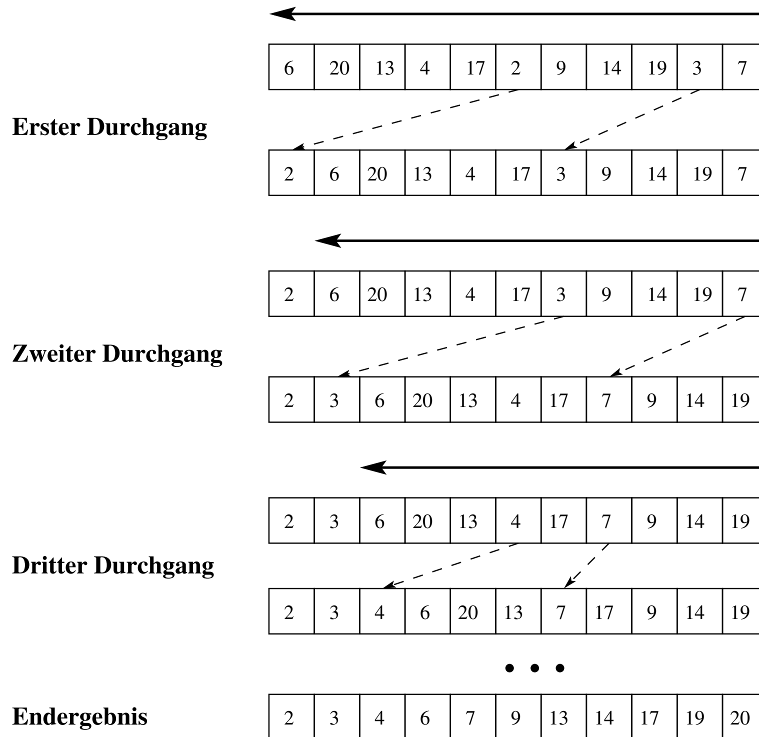

Abbildung 2: Bubble Sort illustriert

Die Elemente werden von rechts nach links durchlaufen. Dabei werden immer die zwei benachbarten Elemente miteinander verglichen. Ist das rechte kleiner als das linke, so werden sie vertauscht. Sollte aber das linke Element bereits kleiner sein als das rechte, so bleibt das rechte stehen, und das linke wandert nun weiter. In der Abbildung passiert dies im ersten Durchgang mit dem Wert 3, der zu Beginn "wanderte", dann aber stehen bleibt, und die 2 weiter wandert.

Der Algorithmus ist deshalb langsam, da er für N-Elemente im schlechtesten Fall $\frac{N^2}{2}$ Vertauschungen vornimmt. Dasselbe gilt für die Vergleiche, also ob das rechte Element kleiner als das linke Element ist (siehe [6]). Gesamthaft kann man also sagen, dass er für N-Elemente N^2 Operationen ausführt. Der Graph in Abbildung 3 auf der nächsten Seite zeigt, dass mit zunehmender Anzahl Elemente die Zahl der Vergleiche, beziehungsweise Vertauschungen, exponentiell zunimmt.

Bubble Sort verdankt seinen Namen übrigens der Tatsache, dass die Elemente wie (Luft-)Blasen (engl. bubble) in Wasser aufsteigen.

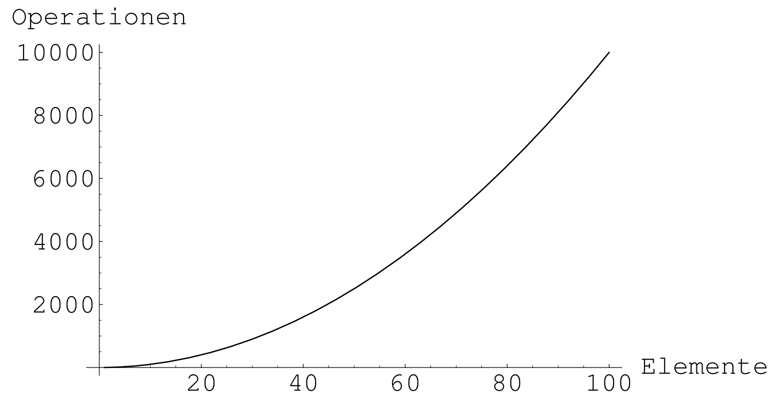

Abbildung 3: Performance von Bubble Sort

4.2 Selection Sort

Selection Sort ist ebenfalls ein äusserst einfacher Algorithmus. Selection Sort beginnt am Anfang der Element-Liste und sucht von links nach rechts das kleinste Elemente. Dann tauscht er das gefundene Element mit dem ersten Element aus. Als nächstes beginnt er beim zweiten Element von links und sucht nun das Element welches kleiner als das zweite von links ist. Wird es gefunden, so tauscht er es gegen das zweite Element aus. Dann verfährt er mit dem dritten bis zum N-ten Element in der Liste nach der selben Methode, bis die gesamte Liste sortiert ist. Als Illustration dieses Vorgangs soll Abbildung 4 auf der nächsten Seite dienen.

Trotz diesem einfachen Ansatz, oder gerade deswegen, ist Selection Sort von der Geschwindigkeit her gesehen nicht überwältigend. Obschon er etwa doppelt so schnell wie Bubble Sort ist, benötigt er für N-Elemente etwa $\frac{N^2}{2}$ Vergleiche und N Vertauschungen (siehe [6]). Das heisst, dass um N-Elemente zu sortieren $\frac{N^2+2N}{2}$ Operationen benötigt. Dieser Sachverhalt wird graphisch in Abbildung 5 auf der nächsten Seite dargestellt.

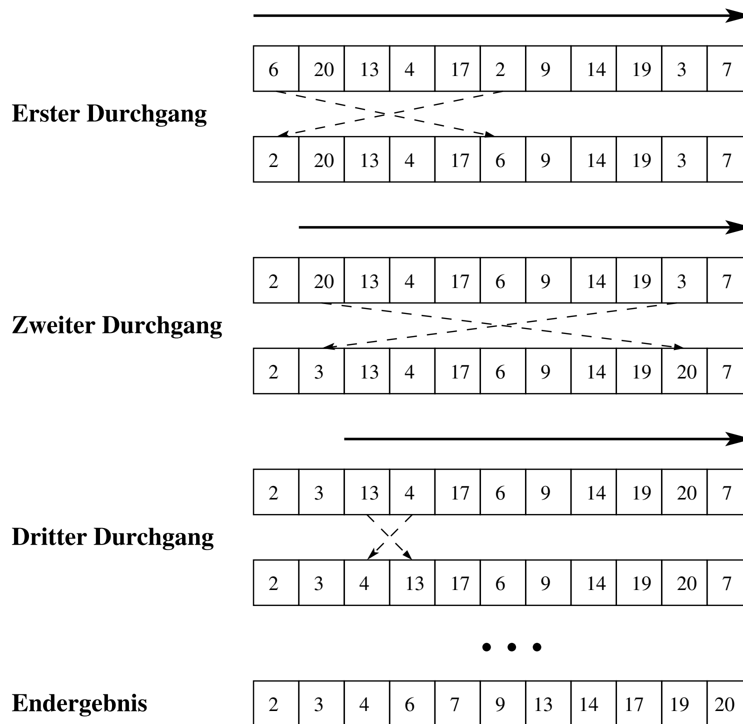

Abbildung 4: Selection Sort illustriert

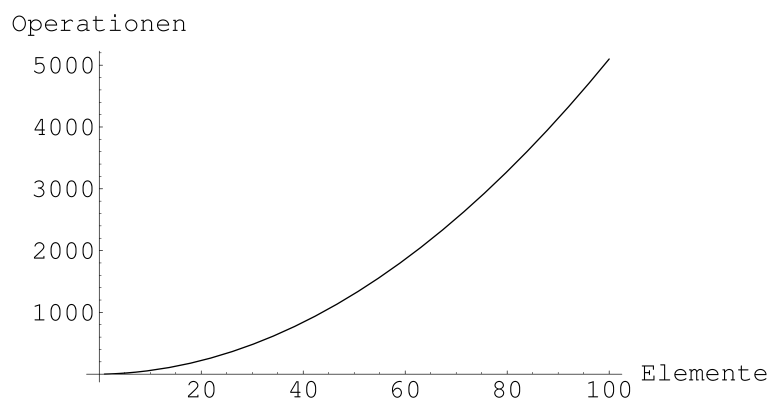

Abbildung 5: Performance von Selection Sort

4.3 Insertion Sort

Die Algorithmen, die bis jetzt betrachtet wurden, haben die zu sortierenden Daten in keiner Weise vorbereitet. Die Daten wurden in der Art, wie sie angetroffen wurden, sortiert. Der hier vorgestellte Insertion Sort Algorithmus beschreitet diesbezüglich einen etwas anderen Ansatz.

Bevor das eigentliche Sortieren beginnt, wird das kleinste Element an den Anfang der Element-Liste gebracht. Dort wird es als *sentinel key*, auf deutsch etwa "Hinweisschlüssel", gebraucht. Dieser sentinel key ist von grosser Bedeutung, denn da er den kleinsten Wert aller Elemente hat, ist es eine Art Stoppschild, welches ein Abbruchkriterium für den Algorithmus darstellt. Leider ist es ziemlich kompliziert, diesen Sachverhalt verständlich darzulegen, ohne in die Tiefen der Programmierung hinabzusteigen. Deshalb halten wir einfach fest, dass dieser sentinel key notwendig ist und ein Abbruchkriterium darstellt.

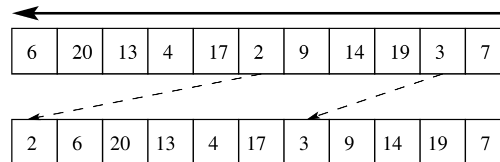

Abbildung 6: Insertion Sort Vorbereitung illustriert

Die Methode, um das kleinste Element an den Anfang der Element-Liste zu bringen, also nach links, ist dem Bubble Sort ähnlich. Man beginnt am Ende der Liste, und vergleicht jeweils das rechte Element mit dem angrenzend linken. Abbildung 6 veranschaulicht diese Methode.

Nachdem der sentinel key gesetzt wurde, beginnt der Algorithmus beim dritten Element von links und wandert nach rechts. Diese dritte Element speichert er temporär an einem anderen Ort und vergleicht alle vorhergehenden Elemente, also nach links schauend, mit diesem temporär abgespeicherten Element. Sobald er ein Element findet, welches kleiner als das temporäre ist, verschiebt er nach dem kleineren Element alle nachfolgenden um eins nach rechts und setzt das temporäre Element an die jetzt frei gewordene Stelle. Danach verfährt der Algorithmus mit dem vierten Element bis zum N-ten Element von rechts in selber Weise. Siehe Abbildung 7 auf der nächsten Seite zur Veranschaulichung.

Bezüglich der Performance[5] des Insertion Sort lässt sich sagen, dass er für N-Elemente $\frac{N^2}{4}$ Vergleiche macht und $\frac{N^2}{4}$ "Verschiebungen" durchführt. Gesamthaft führt der Algorithmus also für N-Elemente $\frac{N^2}{2}$ Operationen durch, was etwas besser ist, als der Durchsatz von Selection Sort, im allgemeinen aber immer noch ziemlich langsam (siehe Abbildung 8 auf der nächsten Seite).

[5]Das Wort "Performance" wird in der Informatik in etwa als "Durchsatz" verstanden.

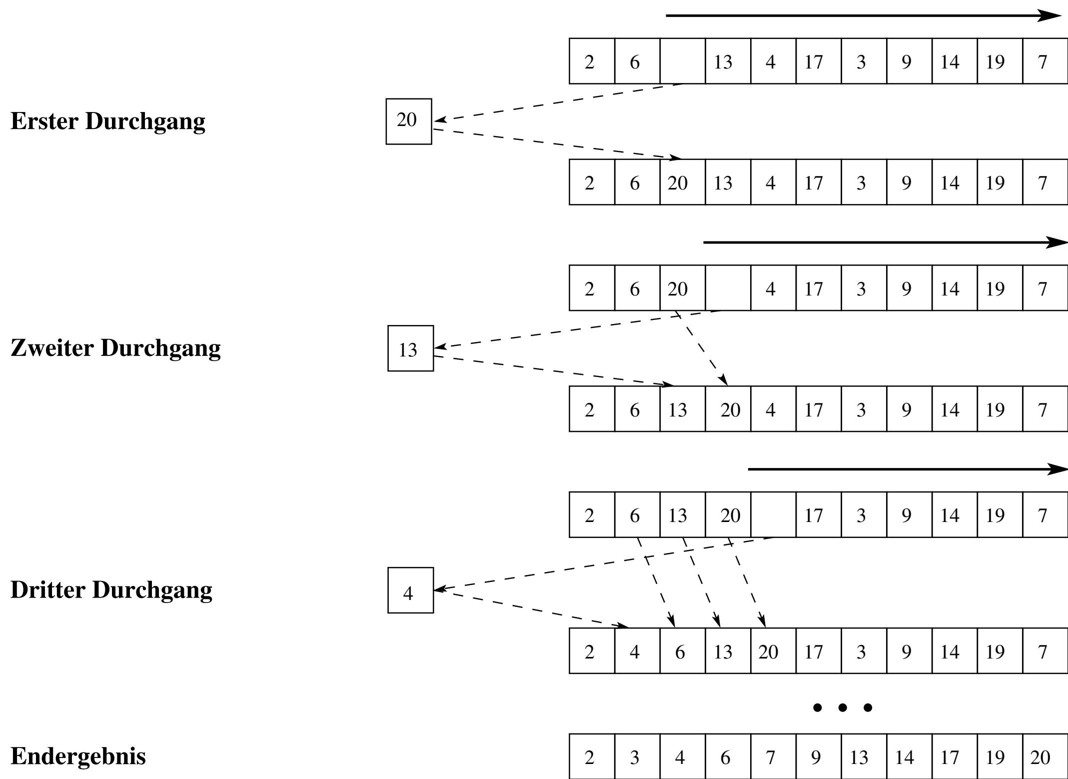

Abbildung 7: Insertion Sort illustriert

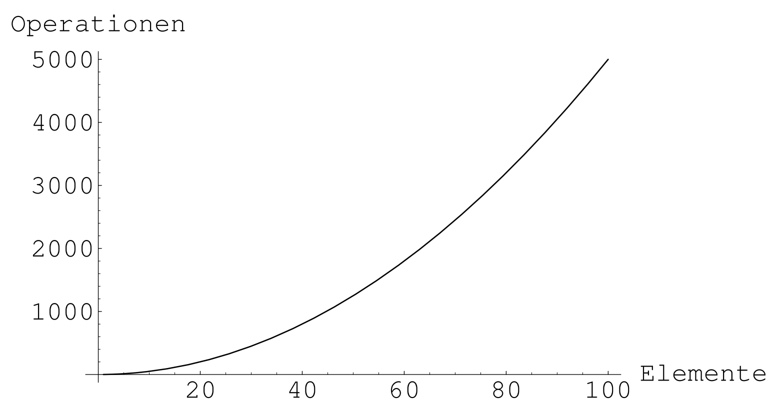

Abbildung 8: Performance von Insertion Sort

4.4 Shell Sort

Shell Sort wurde nach seinem Erfinder Donald Shell benannt, der den Algorithmus 1959 erstmals vorgestellt hatte. Shell Sort basiert auf dem Insertion Sort Algorithmus (siehe Abschnitt 4.3 auf Seite 10), der durch eine einfache Erweiterung schneller und effizienter wird. Gleichzeitig ist er der anspruchvollste Algorithmus, den wir hier besprechen.

Die Idee ist, *nicht* benachbarte Elemente zu vergleichen, sondern Elemente die weiter auseinander liegen. Zu erklären, weshalb und wie das im Detail funktioniert, würde den Rahmen dieses Dokuments sprengen. D. E. Knuth hat das Problem über mehrere Seiten in [1] mathematisch analysiert (siehe [3]). Wir wollen uns hier nur auf das Wesentliche konzentrieren.

Das Herzstück des Algorithmus ist eine Zahlenfolge, welche festlegt, wie weit die Elemente auseinander liegen, die verglichen werden. D. Shell hat die Folge

$$1 \quad 2 \quad 4 \quad 8 \quad 16 \quad \dots$$

verwendet, welche aber nicht sehr effizient gewesen ist. D. E. Knuth hat zehn Jahre später, 1969, die Folge

$$1 \quad 4 \quad 13 \quad 40 \quad 121 \quad \dots$$

entwickelt, die wesentlich bessere Performance erzielt. Bis jetzt ist aber noch nicht klar, ob es andere Zahlenfolgen gibt, die eine noch bessere Performance erzielen [4].

Wie weit die zu vergleichenden Elemente auseinander liegen, und wieviele Durchgänge notwendig sind, ist von der Anzahl der zu sortierenden Elemente abhängig. So wird zum Beispiel eine Liste mit zehn Elementen zuerst mit einer Lücke[6] von vier Elementen sortiert und dann mit einer Lücke von einem Element. Eine Liste mit fünzig Elementen wird zuerst mit einer Lücke von dreizehn, dann von vier und zuletzt von einem Element sortiert. Die oben gennante Zahlenfolge wird also vom Algorithmus rückwärts durchlaufen, vom grösseren zum kleineren Wert.

Wie funktioniert nun aber das Sortieren? Nehmen wir an, eine Liste enthält elf Elemente. Dann wird zuerst das Element E_4 mit dem Element E_0 verglichen.[7] Darauf folgt der Vergleich von E_5 mit E_1, dann E_6 mit E_2 und so weiter und so fort. Wird das Ende der Liste erreicht, beginnt der Algorithmus von vorne, diesmal aber mit einer Lücke von eins, das heisst Element E_1 wird mit E_0 verglichen, E_2 mit E_1 und so weiter.

Das Vergleichen wird vom Insertion Sort Algorithmus vorgenommen, der dann die Elemente bei Bedarf auch gleich vertauscht. Das Durchlaufen der

[6] Als Lücke bezeichne ich den Abstand zwischen den zu vergleichenden Elementen.

[7] Der Einfachheit halber, wird das erste Element mit E_0, das zweite mit E_1 usw. bis E_n bezeichnet. Weshalb bei der Zählung bei Null begonnen wird, ist eine Eigenart der Programmierer.

oben genannten Zahlenfolgen und somit das Bestimmen der Lücke, wird aber vom Shell Sort Algorithmus übernommen, was ganz zu Beginn als die "einfache Erweiterung" bezeichnet wurde. Der Shell Sort könnte ebensogut als eine Verpackung um den Insertion Sort betrachtet werden. Die Abbildung 10 auf der nächsten Seite soll dieses Vorgehen etwas verdeutlichen.

Auf den ersten Blick erscheint die Methode kompliziert und es ist schwer verständlich, weshalb dieses Vorgehen schneller sein soll, als die vorhergehenden Algorithmen. Die Erklärung liegt darin, dass die Liste nach jedem Durchgang besser sortiert ist, und immer weniger Elemente ausgetauscht werden müssen.

Das äussert sich auch in der Geschwindigkeit. Für eine Liste mit N-Elementen braucht es etwa $N^{\frac{3}{2}}$ Operationen, bis sie sortiert ist (siehe Grafik in Abbildung 9).

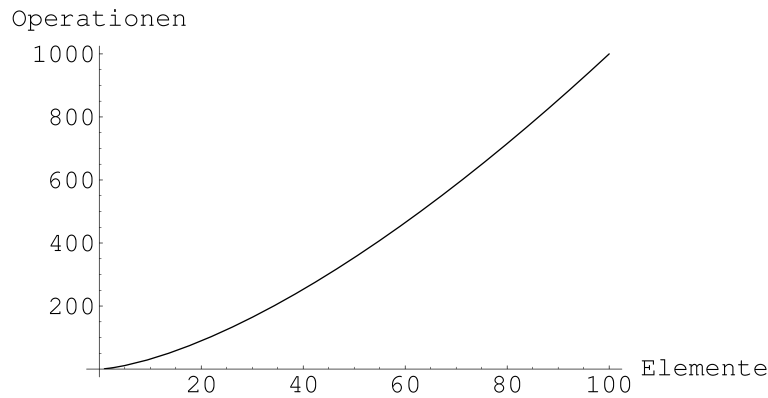

Abbildung 9: Performance von Shell Sort

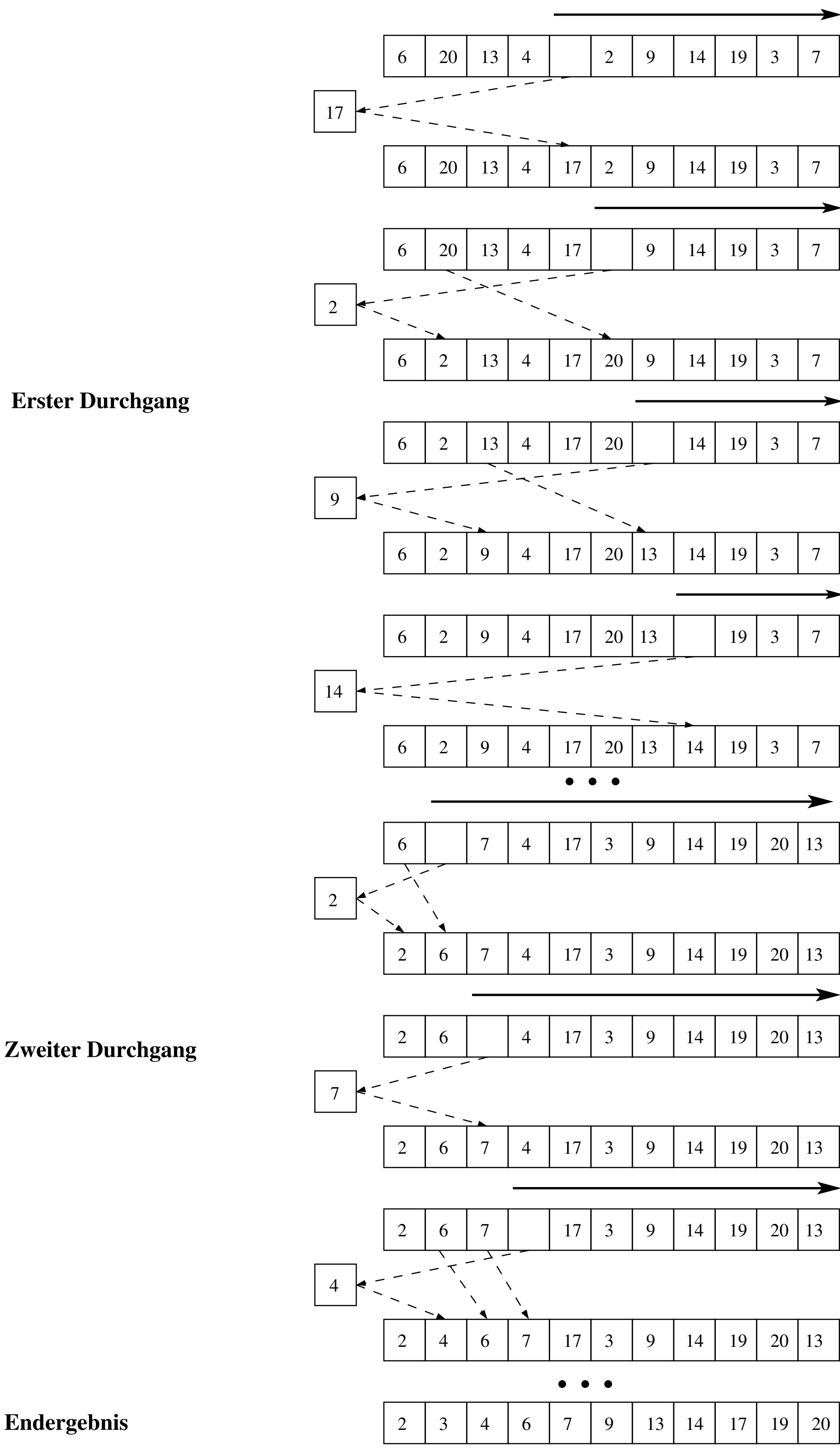

Abbildung 10: Shell Sort illustriert

5 Fazit

Die vorgestellten Algorithmen gehören zu den elementaren Sortieralgorithmen. Sie sind mehr oder weniger einfach verständlich, sind aber nicht geeignet, um riesige Mengen an Daten zu sortieren. Dafür gibt es wesentlich effizientere Algorithmen wie zum Beispiel Merge Sort oder Quick Sort. Trotzdem zeigen die besprochenen Algorithmen, wie Sortieren im Grunde funktioniert.

Das Entwickeln von effizienten Sortieralgorithmen, sowie die Entwicklung von anderen Algorithmen, ist eine Wissenschaft, die sich lohnt und sehr viele praktische Anwendungen hat. Knuth hat diesem Thema ein ganzes Buch gewidmet. Die Informations Technologie von heute käme ohne Sortieralgorithmen nicht aus. Fast in jedem Programm steckt irgendwo eine Routine, die Daten sortiert, manchmal offensichtlich, manchmal weniger.

Ich möchte das Dokument mit dem Vergleich der Geschwindigkeiten der hier vorgestellten Algorithmen in Abbildung 11 abschliessen, denn die Entwicklung von Sortieralgorithmen zielt darauf ab, immer schneller immer grössere Datenmengen zu bearbeiten.

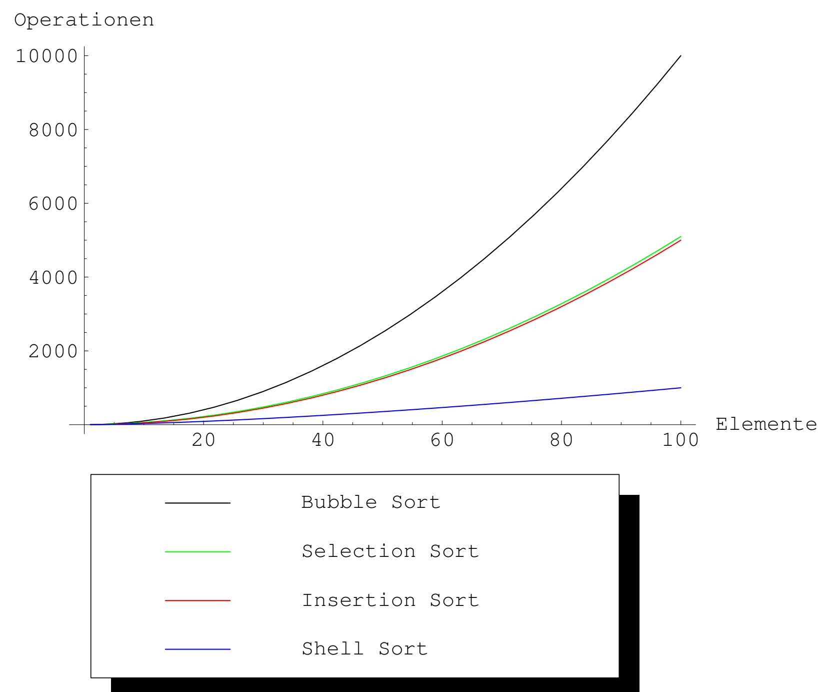

Abbildung 11: Performance im Vergleich

A Code-Beispiele in C++

Hier werden einige C++ Algorithmen aufgelistet, die es einem Programm erlauben Daten zu sortieren. Es wird dabei weder der Anspruch auf Vollständigkeit noch auf Lauffähigkeit der Beispiele erhoben.

A.1 Bubble Sort

```
void Sort::compexch(unsigned long &a, unsigned long &b) {
        if (b < a) {
                std::swap(a,b);
        }
}

void Sort::BubbleSort() {
        for (int i=0; i<entries.size()-1; i++) {
                for (int j = entries.size()-1; j>i; j--) {
                        compexch(entries[j-1],entries[j]);
                }
        }
}
```

A.2 Selection Sort

```
void Sort::SelectionSort() {
        for ( int i = 0; i < entries.size()-1; i++) {
                int min = i;
                for (int j = i+1; j <= entries.size()-1; j++) {
                        if (entries[j] < entries[min]) {
                                min = j;
                        }
                }
                std::swap(entries[i],entries[min]);
        }
}
```

A.3 Insertion Sort

```
void Sort::InsertionSort() {
        int i;
        for (i = entries.size()-1; i > 0; i--) {
                compexch(entries[i-1], entries[i]);
        }

        for (i = 2; i <= entries.size()-1; i++) {
```

```
                int j = i;
                unsigned int v = entries[i];
                while (v < entries[j-1] ) {
                        entries[j] = entries[j-1];
                        j--;
                }
                entries[j] = v;
        }
}
```

A.4 Shell Sort

```
void Sort::ShellSort() {
        int h;
        for (h=1; h<=(entries.size()-1)/9; h=3*h);

        for (; h>0; h/=3) {
                for (int i = h; i<=(entries.size()-1); i++) {
                        int j=i;
                        unsigned int v = entries[i];
                        while (j >= h && v < entries[j-h]) {
                                entries[j] = entries[j-h];
                                j -= h;
                        }
                        entries[j] = v;
                }
        }
}
```

Literatur

[1] D. E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison–Wesley, 2nd edition, 1997.

[2] D. E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching, page 5. Addison–Wesley, 2nd edition, 1997.

[3] D. E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching, page 83 pp. Addison–Wesley, 2nd edition, 1997.

[4] R. Sedgewick. *Algorithms in C++*. Addison–Wesley, 3rd edition, 1998.

[5] R. Sedgewick. *Algorithms in C++*, pages 271–272. Addison–Wesley, 3rd edition, 1998.

[6] R. Sedgewick. *Algorithms in C++*, pages 279–280. Addison–Wesley, 3rd edition, 1998.