

Suchen und Sortieren

In diesem Kapitel behandeln wir Algorithmen zum Suchen und Sortieren

Inhalt

1. Grundlagen	2
2. Sortieren	6
1.1. Vertauschen	13
1.2. Selektion	16
1.3. Einfügen	19
1.4. Quicksort	22
3. Suchen	28
1.5. Sequentielle Suche	28
1.6. Binäre Suche	29

1. Grundlagen

Um **Algorithmen** zu **analysieren**, braucht man Kriterien, mit denen man eine Bewertung vornehmen kann. Hier sollen nicht Lesbarkeit oder Güte der Strukturierung verwendet werden; hier ist **Laufzeit** und **Speicherbedarf** von Interesse.

Als **Grundlage** wird eine **Anweisung** genommen. Dabei werden eine Zuweisung, ein Vergleich und ein Funktionsaufruf jeweils als 1 gewertet. Dann werden die Einzelschritte gezählt und man hat ein Maß für den Algorithmus.

Damit ist die Bewertung **unabhängig** von der **Ausführungsumgebung** und der Güte des Compilers.

Zum **Zählen** der Anweisungen verwenden wir **Annäherungen**, um von Details abstrahieren zu können, uns interessieren nur **Größenordnungen**. Betrachten wir dazu 3 C++ Fragmente:

<pre>x = x + 1;</pre> <p>(a)</p>	<pre>for (int i=1; i<=n; i++) { x = x + 1; }</pre> <p>(b)</p>	<pre>for (int i=1; i<=n; i++) { for (int j=1; j<=n; j++) { x = x + 1; } }</pre> <p>(c)</p>
----------------------------------	--	--

Die Laufzeit dieser Fragmente ist:

- (a) die Zuweisung wird als **1** gezählt
- (b) die Zuweisung wird n mal ausgeführt, also ist die Rechenzeit **n**
- (c) durch die verschachtelten for Schleifen ergibt sich als Rechenzeit $n*n=n^2$

Da wir nur an **Größenordnungen** interessiert sind, gilt es bei der Analyse die **Anweisungen** zu betrachten, die den **größten Beitrag** zur Rechenzeit beitragen und **alles andere wegzulassen**. Deshalb wird bei der Rechenzeit oben das Inkrementieren von i und j nicht gezählt.

Die Ermittlung der Laufzeit für (c) zeigt, dass man von der exakten Anzahl abweicht:

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

Um dies zum Ausdruck zu bringen, verwendet man die **O-Notation**, (c) hat die Laufzeit **$O(n^2)$** , wobei das O oft weggelassen wird.

Man kann das aber mathematisch exakt definieren (-> Analysis)

Definition (O Notation)

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge c > 0, \text{ so dass } \forall n \geq n_0 : g(n) \leq c * f(n)\}$$

Natürlich sprachig:

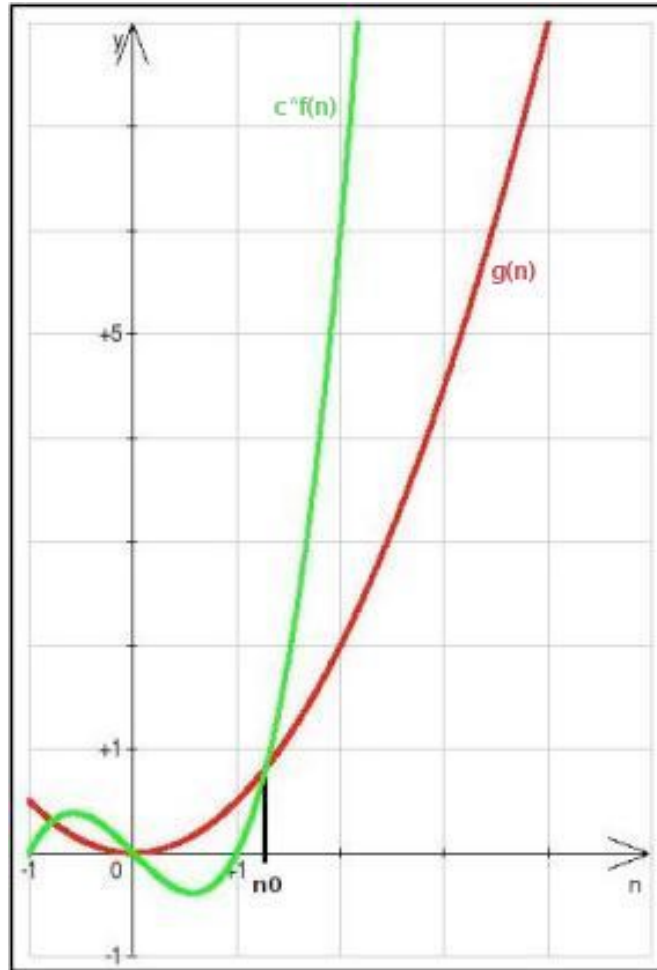
*$O(f(n))$ ist die Menge aller der Funktionen $g(n)$, für die es zwei Konstanten n_0 und c gibt, so dass für alle $n \geq n_0$ gilt: ' $g(n) \leq c * f(n)$ '*

Interpretation:

Die O-Notation eine **obere Schranke** zur Laufzeit eines Algorithmus, die selbst im ungünstigsten Fall nicht überschritten wird.

Anders ausgedrückt: $g(n)$ wächst höchstens so schnell wie $f(n)$.

Veranschaulichung:



$g(n)$ wächst höchstens so schnell wie $f(n)$.

Für $O(1)$, $O(n)$, $O(n^2)$, $O(2^n)$ sagt man jeweils **konstante**, **lineare**, **quadratische** und **exponentielle** Laufzeit.

2. Sortieren

Im Bereich Informatik ist das **Sortieren** sehr früh ausführlich behandelt worden, denn es ist in fast allen Anwendungsbereichen von Bedeutung (Datenbanken, Compiler, Betriebssysteme, ...).

Hier werden Sortierverfahren vorgestellt und dabei gezeigt, dass die **Laufzeit** von Programmen im Wesentlichen von der **Wahl** eines guten **Algorithmus** abhängt.

Sortieren ist ein Prozess, der eine Gruppe von ähnlichen Informationen in auf- oder absteigender Folge anordnet.

Definition (Sortieren)

Sei $R = (R_1, R_2, \dots, R_n)$ eine Folge von Rekords.

Jedem Rekord R_i sei ein Schlüssel K_i zugeordnet.

Sei weiterhin $<$ eine Ordnungsrelation auf der Schlüsselmenge K so dass gilt:

$$\forall x, y \in K : x = y \text{ oder } x < y \text{ oder } y < x.$$

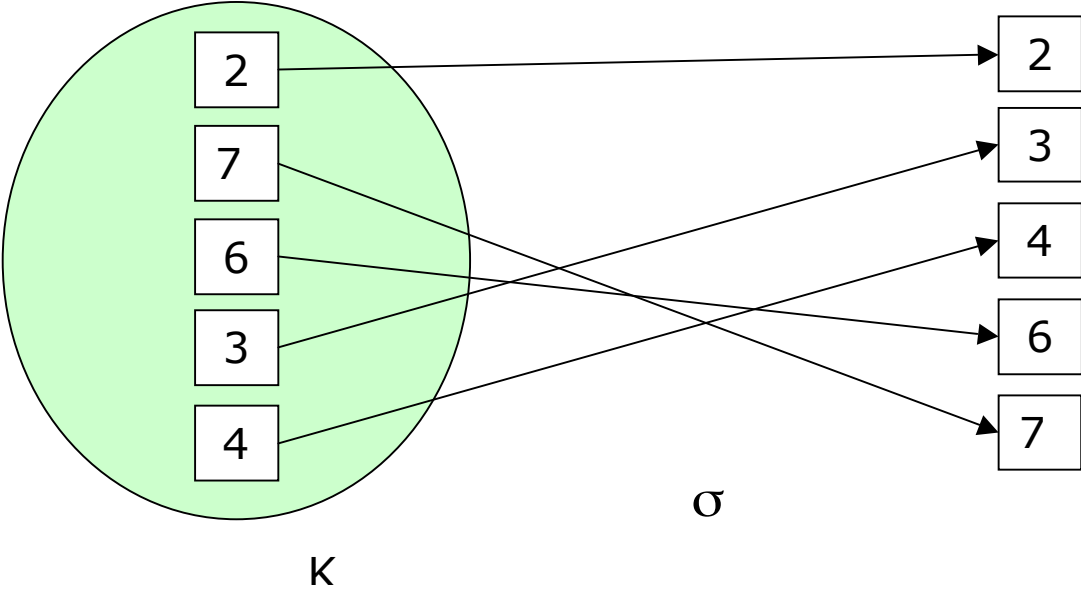
Die Relation $<$ sei transitiv, d.h. $\forall x, y, z \in K : x < y$ und $y < z$ impliziert $x < z$.

Eine Sortierung von R ist eine Permutation σ für die gilt:

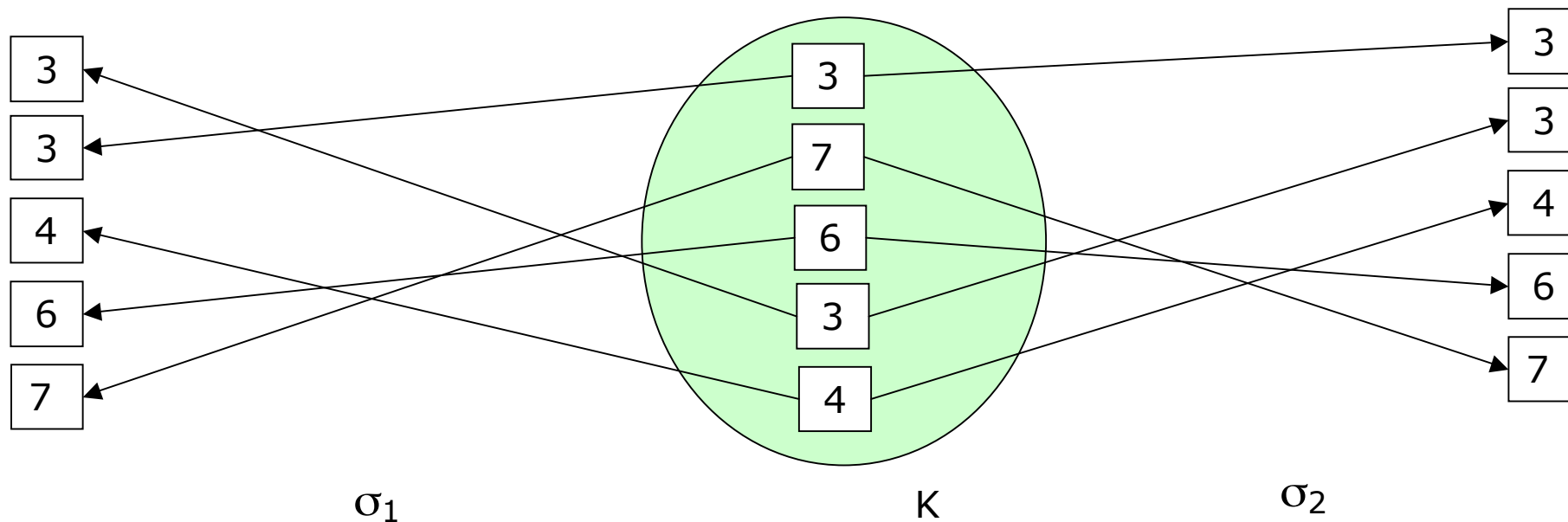
$$K_{\sigma(i)} \leq K_{\sigma(i+1)} \text{ für } 1 \leq i \leq n-1.$$

Die Sortierfolge ist dann $(R_{\sigma(1)}, R_{\sigma(2)}, \dots, R_{\sigma(n)})$.

Beispiel:



Wenn die Menge R mehrere Elemente enthält, die den gleichen Schlüssel haben, so ist die Sortierung (Permutation) nicht eindeutig bestimmbar.



Man kann eine Permutation auszeichnen, die die Eigenschaft hat, dass sie eine Sortierreihenfolge erzeugt, die bei Schlüsseln die **Ursprungsreihenfolge** dieser Rekords **erhält**.

Definition (Stabile Sortierung):

Sei $R = (R_1, R_2, \dots, R_n)$ eine Folge von Rekords.

Jedem Rekord R_i sei ein Schlüssel K_i zugeordnet.

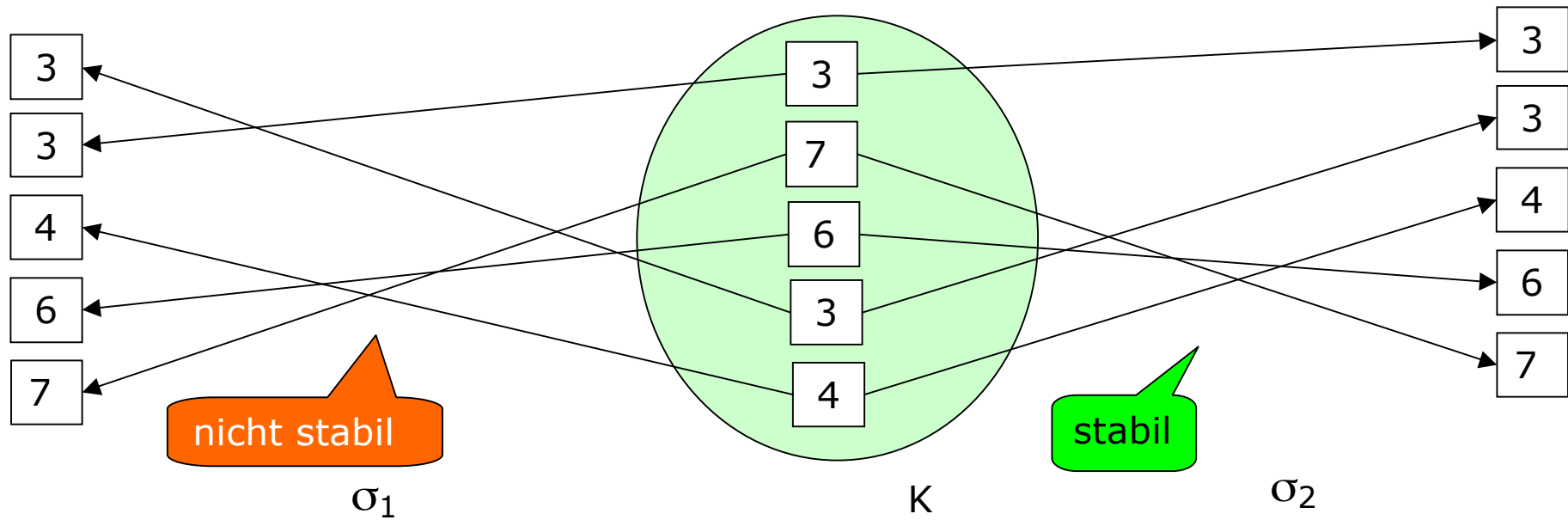
Sei weiterhin σ eine Sortierung von R für die gilt:

(i) $K_{\sigma(i)} \leq K_{\sigma(i+1)}$ für $1 \leq i \leq n-1$.

(ii) Wenn $i < j$ und $K_i = K_j$ dann ist $R_{\sigma(i)} < R_{\sigma(j)}$

Diese Permutation nennen wir stabil.

Beispiel:



Im Folgenden werden nicht Rekords, sondern **lediglich** die **Schlüssel** betrachtet.

Man kann drei grundlegende Arten des Sortierens von Arrays unterscheiden (Beispiel Kartenspiel)

- **Vertauschen**

Die Karten eines Kartenspiels werden auf den Tisch gelegt.

Es werden solange Paare von Karten vertauscht, bis die richtige Reihenfolge hergestellt ist.

- **Selektion**

Die Karten eines Kartenspiels werden auf den Tisch gelegt.

Man nimmt die Karte mit dem kleinsten Wert auf die Hand. Dann nimmt man die nächste Karte mit dem kleinsten Wert vom Tisch und steckt sie hinter die letzte Karte auf der Hand.

Dies wird solange wiederholt, bis keine Karte mehr auf dem Tisch liegt.

- **Einfügen**

Man hält die Karten in der Hand und nimmt eine nach der anderen. Jedes Mal legt man eine Karte auf einen Stapel auf dem Tisch, wobei die Karte an die richtige Position gesteckt wird.

Sortierverfahren sind nach folgenden Kriterien zu **bewerten**:

- Wie schnell kann der Algorithmus in **durchschnittlichen** Fällen sortieren?
- Wie schnell ist er im **besten** und im **schlechtesten** Fall?
- Ist das Verfahren **stabil**?
- Arbeitet das Verfahren **natürlich** oder **unnatürlich** (natürlich: wenn für eine bereits sortierte Liste am wenigsten Zeit verbraucht wird).

1.1. Vertauschen

Ein Vertreter der Vertauschverfahren ist Bubblesort (vgl. Kapitel Programmstrukturierung).

Die Idee ist, wiederholt benachbarte Elemente zu vertauschen (falls erforderlich) bis alle Elemente sortiert sind; verglichen wird immer „linker Nachbar“ > „aktuelles Element“.

Ausgangssituation	12	3	2	6
1. Lauf: 1. Schritt	12	3	2	6
2. Schritt	12	2	3	6
3. Schritt	2	12	3	6
2. Lauf: 1. Schritt	2	12	3	6
2. Schritt	2	12	3	6
3. Schritt	2	3	12	6
3. Lauf: 1. Schritt	2	3	6	12

```

$ cat bubbleSort.cpp
#include <iostream.h>
#include <string>
void bubbleSort(string &daten) {
    int zaehler;

    zaehler = daten.length();
    for (int lauf=1; lauf <zaehler; lauf++) {           // Läufe
        for (int element=zaehler-1; element>=lauf; element--) { // ein Lauf
            if (daten[element-1] > daten[element]) {      // vertauschen
                char tmp = daten[element-1];
                daten[element-1] = daten[element];
                daten[element] = tmp;
            }
        }
    }
}

```

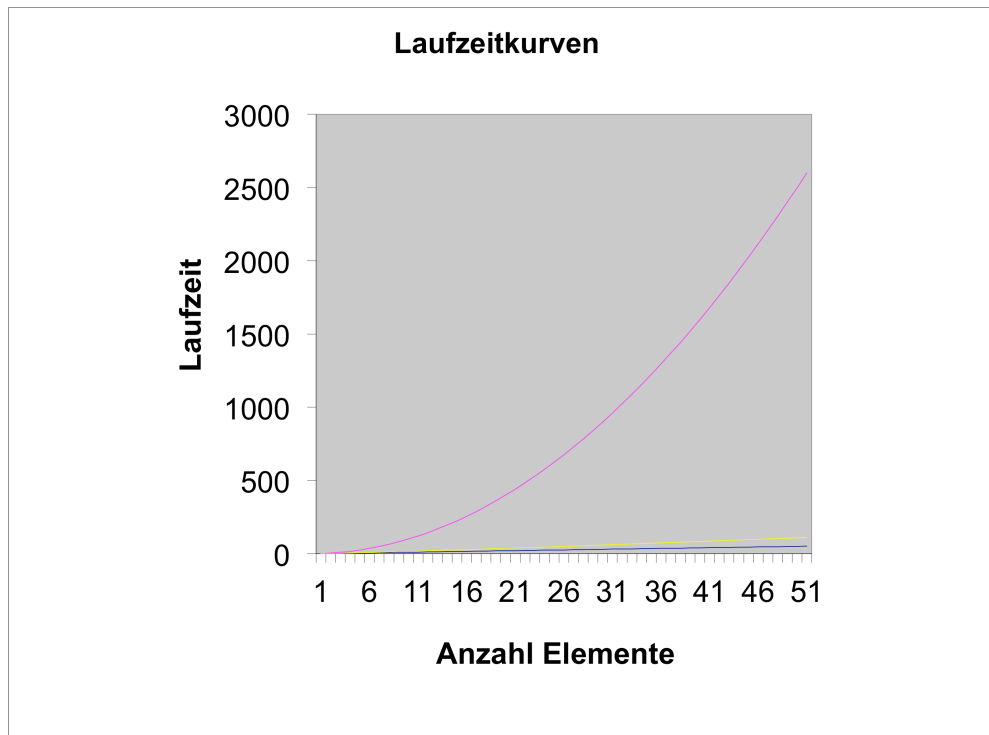
Wie betrachten hier (und in den folgenden Verfahren auch) nur die **Anzahl der Vergleiche** als **Maß** für die „**Schnelligkeit**“. Eine vollständige Analyse müsste zusätzlich die Anzahl der Vertauschungen betrachten.

Laufzeit durchschnittlich	Äußere Schleife n-1 mal, innere Schleife n/2 mal: $(n-1) * n / 2 = \frac{1}{2}(n^2 - n)$	Zahl der Vergleiche stets die gleiche, da zwei feste for-schleifen
Laufzeit schlechtesten Fall		
Laufzeit im besten Fall		

stabil	?	
natürlich	+?	

Dieses **quadratische** Laufzeit-Verhalten ist sehr schlecht. Bei einer Größe von mehreren Tausend Elementen ist auch ein schneller Rechner zu langsam.

Beispiel: Laufzeitkurven für $y=x^2$, $y=x^{1,2}$ und $y=x$

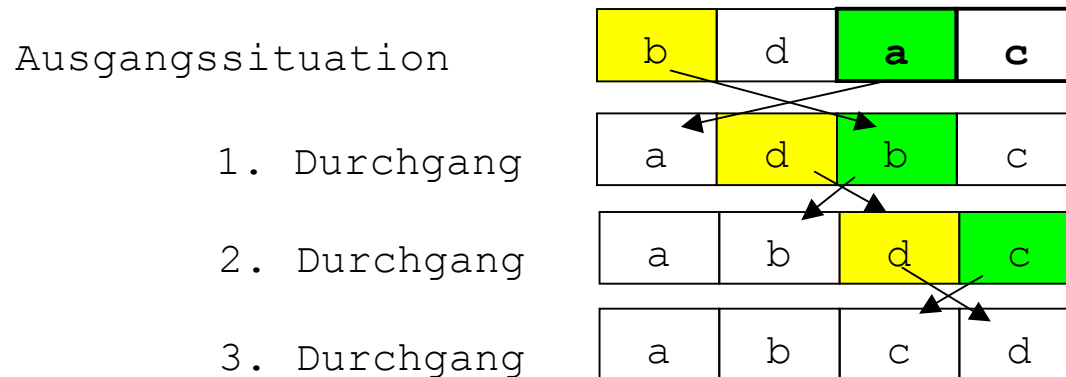


Wie lange braucht ein Rechner, der für eine Vergleichsoperation etwa 10^{-5} Sekunden braucht, für eine Array mit 10.000.000 Telefonnummern und quadratischer Laufzeit?

1.2. Selektion

Ein Selektionssort wählt das Element mit dem **niedrigsten Wert** aus und vertauscht es mit dem **ersten** Element. Von den verbleibenden $n-1$ Elementen wird dann das kleinste genommen und mit dem zweiten Element vertauscht. Dies wird solange wiederholt, bis zu den letzten zwei Elementen.

Beispiel (aufsteigend Sortieren):



Damit ist das Programm einfach zu programmieren:

```

$ cat selectionSort.cpp
#include <iostream.h>
#include <string>
void selectionSort(string &daten) {
    int zaehler;

    zaehler = daten.length();
    for (int a=0; a <zaehler-1; a++) {
        int c=a; // index kleistes Element
        char min = daten[a]; // kleinstes Element
        for (int b=a+1; b<zaehler; b++) { // suche kleinstes Element
            if (daten[b] < min) {
                c = b;
                min = daten[b];
            }
        }
        daten[c] = daten[a]; // vertausche aktuelles Element mit kleinstem Element
        daten[a] = min;
    }
}

```

Laufzeit durchschnittlich	Äußere Schleife n-1 mal, innere Schleife n/2 mal:	Zahl der Vergleiche stets die gleiche, da zwei feste for-schleifen
Laufzeit schlechtesten Fall		
Laufzeit im besten Fall		

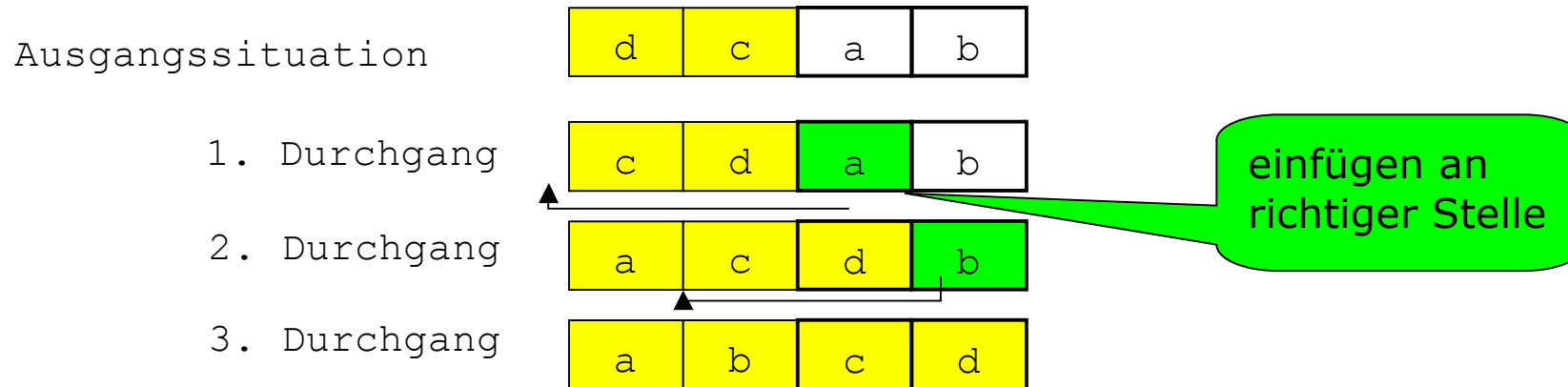
	$(n-1) * n / 2 = \frac{1}{2}(n^2 - n)$	
stabil	?	
natürlich	-?	

Also immer noch ein Algorithmus mit **quadratischer** Laufzeit!

1.3. Einfügen

Der letzte einfache Sortieralgorithmus ist Einfügesort. Zuerst werden die beiden ersten Elemente des Array sortiert. Dann setzt das Verfahren das dritte Element in Bezug auf die ersten beiden Elemente sortierte Position. Danach wird das vierte Element an die richtige Stelle in der Liste der ersten drei eingefügt. Diese Prozedur wird solange wiederholt, bis alle Elemente sortiert sind.

Beispiel (aufsteigend Sortieren):



Damit ist das Programm einfach zu programmieren:

```

$ cat insertionSort.cpp
#include <iostream.h>
#include <string>
void insertionSort(string &daten) {
    int zaehler;

    zaehler = daten.length();
    for (int a=1; a <zaehler; a++) {           // Läufe
        char tmp=daten[a];
        int b = a-1;
        while (b>=0 && tmp<daten[b]) {       // einfügen an richtiger Stelle
            daten[b+1] = daten[b];
            b--;
        }
        daten[b+1] = tmp;
    }
}

```

Anders als bei den vorherigen Verfahren muss hier unterschieden werden, ob die Liste bereits sortiert ist oder nicht; im Programm sind **keine** zwei feste for-Schleifen.

Laufzeit im besten Fall	Äußere Schleife n-1 mal, while bricht sofort beim 1. mal ab: n	Die Liste ist bereits sortiert
Laufzeit schlechtesten Fall	Äußere Schleife n-1 mal, while bricht erst am Ende ab, also beim 1. Durchlauf sind n Vergleiche notwendig, beim 2.	Die Liste ist umgekehrt sortiert

	Durchlauf n-1 , ... beim letzten Durchlauf 1: $n + (n-1) + (n-2) + \dots + 1 = \sum_{i=1}^n i = \frac{n * (n+1)}{2} = \frac{1}{2}(n^2 + n)$	
Laufzeit durchschnittlich	Äußere Schleife n-1 mal, while bricht in der „Mitte“ ab: $\sum_{i=1}^n i / 2 = \frac{n * (n+1)}{4} = \frac{1}{4}(n^2 + n)$	
stabil	+	
natürlich	+	

Also immer noch ein Algorithmus mit **quadratischer** Laufzeit!

1.4. Quicksort

Die bisher betrachteten Sortierverfahren hatten alle den Nachteil, dass sie quadratische Laufzeit hatten. Dies ist für praktische Anwendungen nicht akzeptable, wenn größere Datenmengen zu bearbeiten sind.

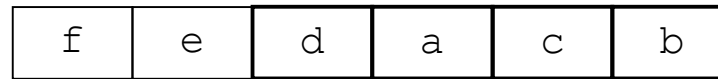
Der zur Zeit „schnellste“ Algorithmus wurde von C.A.R. Hoare erfunden und **Quicksort** genannt.

Er basiert auf der Vertauschmethode (wie Bubblesort). Die **Grundidee** ist das **Einteilen in Klassen**:

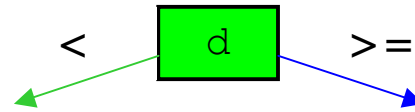
Man wählt zunächst einen Vergleichswert aus und teilt dann das Array in zwei Klassen, von denen eine die Elemente enthält, die größer oder gleich als der Vergleichswert sind, die andere Klasse enthält die Elemente, die kleiner sind. Dann wird das Verfahren auf jede der beiden Klassen angewendet bis das gesamte Feld sortiert ist.

Beispiel:

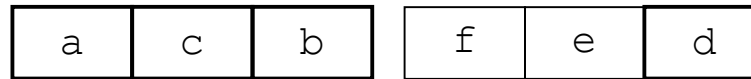
Ausgangssituation



wähle Vergleichswert



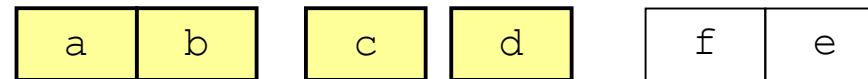
bilde Klassen



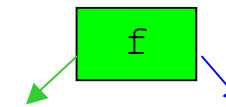
wähle Vergleichswert



bilde Klassen



wähle Vergleichswert



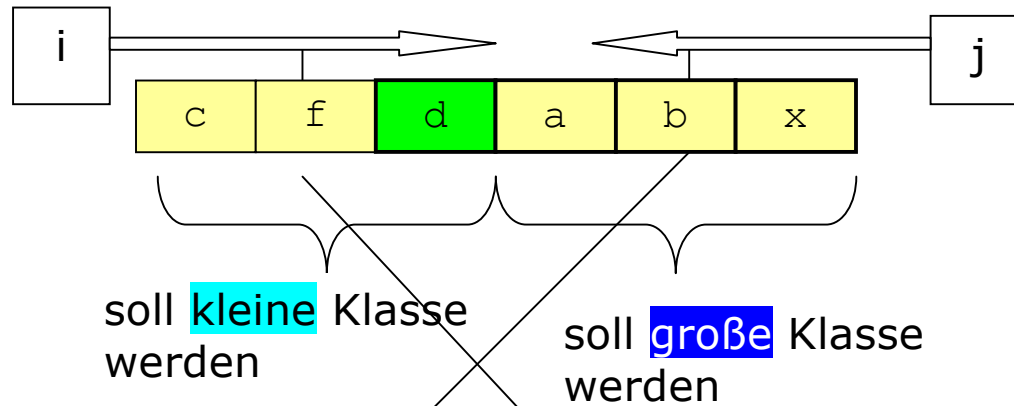
bilde Klassen



Aus dem o.a. Beispiel sieht man, dass man zuerst ein Vergleichswert wählt, dann die Einteilung in Klassen vornimmt und dann das gleiche Verfahren auf die Klassen anwendet, also ein **rekursiv** definiertes Verfahren.

Die **Wahl** des **Vergleichselementes** ist beliebig: man kann zufällig einen Wert innerhalb der Klasse auswählen, das linke, das rechte Element nehmen. In der Praxis hat es sich bewährt, einfach das **mittlere** Element zu nehmen.

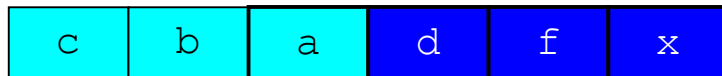
Im Programm werden zwei Indexe (i,j) verwendet, um die Klassen zu bestimmen, wobei i für die Ermittlung der „kleinen Klasse“ und j zur Ermittlung der „großen Klasse“ zuständig ist:



Dabei wird immer das Element an Position i mit dem an Position j vertauscht



bis die Klassen gebildet sind:



Programm:

```

$ cat quickSort.cpp
void qs( string &daten, int links, int rechts) {
    int i,j;
    char x,y;

    i=links; j=rechts;
    x = daten[(links+rechts)/2]; // mittleres Element

    do {
        while (daten[i]<x && i<rechts) i++; // such Element für "große Klasse"
        while (x<daten[j] && j>links) j--; // such Element für "kleine Klasse"
        if (i<=j) { // vertausche Elemente
            y = daten[i];
            daten[i] = daten[j];
            daten[j] = y;
            i++; j--;
        }
    } while (i<=j);

    if (links<j) qs(daten, links, j); // Aufruf für "kleine Klasse"
    if (i<rechts) qs(daten, i, rechts); // Aufruf für "große Klasse"
}

```

Die Arbeitsweise kann man sich verdeutlichen, in dem man die Zwischenschritte ausgibt:

```

$ QuickSortMitAusgabe
Zeichenkette: 8274360567
QS: | 8274360567 | [3]
    | 8274360567 | i:0 j:6
    | 0274368567 | i:2 j:4
    | 0234768567 | i:3 j:2
QS: | 023 | 4768567 [2]
    | 023 | 4768567 i:1 j:1
QS: 023 | 4768567 | [8]
    023 | 4768567 | i:6 j:9
    023 | 4767568 | i:9 j:8
QS: 023 | 476756 | 8 [6]
    023 | 476756 | 8 i:4 j:8
    023 | 466757 | 8 i:5 j:7
    023 | 465767 | 8 i:6 j:5
QS: 023 | 465 | 7678 [6]
    023 | 465 | 7678 i:4 j:5
QS: 023 | 45 | 67678 [4]
    023 | 45 | 67678 i:3 j:3
QS: 023456 | 767 | 8 [6]
    023456 | 767 | 8 i:6 j:7
QS: 0234566 | 77 | 8 [7]
    0234566 | 77 | 8 i:7 j:8
sortiert:      0234566778

```

```

#include <iostream.h>
#include <string>

void qs( string &daten, int links, int rechts) {
    int i,j;
    char x,y;

    i=links; j=rechts;
    x = daten[(links+rechts)/2]; // mittleres Element
    cout << "QS: "
    <<     daten.substr(0,links) << "|"
    <<     daten.substr(links,rechts-links+1) << "|"
    <<     daten.substr(rechts+1)
    <<     " [" << x << "]" << endl;
    do {
        while (daten[i]<x && i<rechts) i++; // such Element f,r "grofle Klasse"
        while (x<daten[j] && j>links) j--; // such Element f,r "kleine Klasse"
        cout << " "
        <<     daten.substr(0,links) << "|"
        <<     daten.substr(links,rechts-links+1) << "|"
        <<     daten.substr(rechts+1)
        << " i:" << i << " j:" << j << endl;
        if (i<=j) { // vertausche Elemente
            y = daten[i];
            daten[i] = daten[j];
            daten[j] = y;
            i++; j--;
        }
    } while (i<=j);

    if (links<j) qs(daten, links,j); // Aufruf f,r "kleine Klasse"
    if (i<rechts) qs(daten, i, rechts); // Aufruf f,r "grofle Klasse"
}

```

Verdeutlichen Sie sich die Arbeitsweise an der o.a. Ausgabe.

Laufzeit im besten Fall	$O(n)$	Wenn jedes Mal Klassen entstehen, die gleich groß sind
Laufzeit schlechtesten Fall	$O(n^2)$	Der Vergleichswert in jeder Klasse ist der jeweils größte Wert
Laufzeit durchschnittlich	$O(n * \log_2 n)$	
stabil	+	
natürlich	+	

3. Suchen

Suchen bedeutet, einen Schlüssel zu finden, damit der korrespondierende Rekord gelesen werden kann.

1.5. Sequentielle Suche

Die sequentielle Suche startet in einem Array mit der Überprüfung des ersten Elementes; ist der Wert mit dem Suchschlüssel identisch, hat man Erfolg, ansonsten wird das nächste Element verglichen.

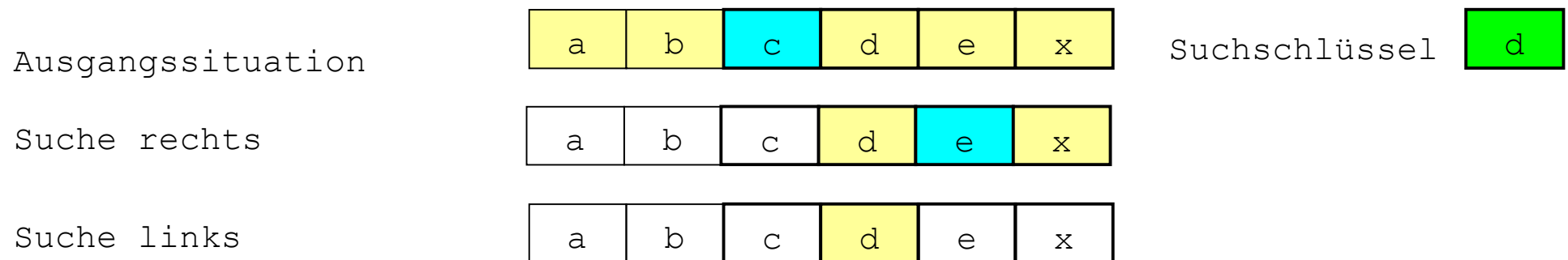
```
$ cat seqSearch.cpp
#include <iostream.h>
#include <string>
int seqSearch(string str, char key) {
    int t;
    int len = str.length();
    for (t=0; t<len; t++)
        if (key==str[t]) return t; // gefunden
    return (-1); // nicht gefunden
}
```

Laufzeit im besten Fall	$1=O(1)$	Das erste Element ist der Schlüssel
Laufzeit schlechtesten Fall	$n=O(n)$	Das letzte Element ist der Schlüssel
Laufzeit durchschnittlich	$1/2n=O(n)$	

1.6. Binäre Suche

Liegen die Daten bereits **sortiert** vor, kann man schneller Suchen. Das Prinzip ist „**Teile und Herrsche**“. Zunächst wird das **mittlere Element** untersucht, ist es der **Suchschlüssel** ist man fertig. Ist der Suchschlüssel kleiner als das mittlere Element, wird der linken Hälfte des Arrays mittels des gleichen Verfahrens bearbeitet, ansonsten der rechte Teil.

Beispiel:



Bei jedem Durchlauf wird der Suchstring halbiert, dadurch vermindert sich die Laufzeit.

```

$ cat binSearch.cpp
...
int binSearch(string str, char key) {
    int low, high, mid;
    low = 0; high = str.length();

    while (low <= high) {
        mid = (low+high)/2;
        if (key < str[mid]) high = mid-1;
        else if (key > str[mid]) low = mid+1;
        else return mid;           // gefunden
    }
    return (-1);                  // nicht gefunden
}

```

Laufzeit im besten Fall	$1=O(1)$	Das mittlere Element ist der Schlüssel
Laufzeit schlechtesten Fall	$O(\log_2 n)$	
Laufzeit durchschnittlich	$O(\log_2 n)$	

Hörsaalübung

Binäre Suche als rekursives Programm.