

Algorithmen und Datenstrukturen

Jonas Milkovits

Last Edited: 30. April 2020

Inhaltsverzeichnis

1	Einleitung	1
1.1	Probleme in der Informatik	1
1.2	Definitionen für Algorithmen	1
2	Sortieren	2
2.1	Einführung ins Sortieren	2
2.2	Analyse von Algorithmen - Teil 1	3
2.3	Analyse von Algorithmen - Teil 2	3
2.4	Analyse von Algorithmen - Teil 3	4
2.5	Insertion Sort	7
2.6	Bubble Sort	8
3	Pseudocode in der Vorlesung AuD	10

1 Einleitung

1.1 Probleme in der Informatik

- Problem im Sinne der Informatik
 - Enthält eine Beschreibung der Eingabe
 - Enthält eine Beschreibung der Ausgabe
 - Gibt **keinen** Übergang von Eingabe und Ausgabe an
 - z.B.: Finde den kürzesten Weg zwischen zwei Orten
- Probleminstanzen
 - Probleminstanz ist eine konkrete Eingabenbelegung, für die entsprechende Ausgabe gewünscht ist
 - z.B.: Was ist der kürzeste Weg vom Audimax in die Mensa?

1.2 Definitionen für Algorithmen

- Begriff des Algorithmus
 - Endliche Folge von Rechenschritten, der eine Ausgabe in eine Eingabe verwandelt
- Anforderungen an Algorithmen
 - Spezifizierung der Eingabe und Ausgabe
 - Anzahl und Typen aller Elemente ist definiert
 - Eindeutigkeit
 - Jeder Einzelschritt ist klar definiert und ausführbar
 - Die Reihenfolge der Einzelschritte ist festgelegt
 - Eindlichkeit
 - Notation hat eine endliche Länge
- Eigenschaften von Algorithmen
 - Determiniertheit
 - Für gleiche Eingabe stets die gleiche Ausgabe (andere mögliche Zwischenzustände)
 - Determinismus
 - Für gleiche Eingabe stets identische Ausführung und Ausgabe
 - Terminierung
 - Algorithmus läuft für jede Eingabe nur endlich lange
 - Korrektheit
 - Algorithmus berechnet stets die spezifizierte Ausgabe (falls dieser terminiert)
 - Effizienz
 - Sparsamkeit im Ressourcenverbrauch (Zeit, Speicher, Energie,...)

2 Sortieren

2.1 Einführung ins Sortieren

- **Das Sortierproblem**

- Ausgangspunkt: Folge von Datensätzen D_1, D_2, \dots, D_n
- Zu sortierende Elemente heißen auch Schlüssel(werte)
- Ziel: Datensätze so anzuordnen, dass die Schlüsselwerte sukzessive ansteigen/absteigen
- Bedingung: Schlüsselwerte müssen vergleichbar sein
- Durchführung:
 - Eingabe: Sequenz von Schlüsselwerten $\langle a_1, a_2, \dots, a_n \rangle$
 - Eingabe ist eine **Instanz** des Sortierproblems
 - Ausgabe: Permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ derselben Folge mit Eigenschaft $a'_1 \leq \dots \leq a'_n$
- Algorithmus **korrekt**, wenn dieser das Problem für alle Instanzen löst

- **Exkurs: Totale Ordnung**

- Sei M eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf M
- Das Paar (M, \leq) heißt genau dann totale Relation auf der Menge M , wenn Folgendes erfüllt ist:
 - Reflexivität: $\forall x \in M : x \leq x$
 - Transitivität: $\forall x, y, z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z$
 - Antisymmetrie: $\forall x, y \in M : x \leq y \wedge y \leq x \Rightarrow x = y$
 - Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$
- z.B.: \leq Ordnung auf natürlichen Zahlen bildet eine totale Ordnung ($1 \leq 2 \leq 3 \dots$)
- z.B.: Lexikographische Ordnung \leq_{lex} ist eine totale Ordnung ($A \leq B \leq C \dots$)

- **Vergleichskriterien von Sortieralgorithmen**

- Berechnungsaufwand $O(n)$
- Effizient: Best Case vs Average Case vs Worst Case
- Speicherbedarf:
 - in-place (in situ): Zusätzlicher Speicher von der Eingabegröße unabhängig
 - out-of-place: Speichermehrbedarf von Eingabegröße abhängig
- Stabilität: Stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht
- Anwendung als Auswahlfaktor:
 - Hauptoperationen beim Sortieren: Vergleiche und Vertausche
 - Diese Operationen können sehr teuer oder sehr günstig sein, je nach Aufwand
 - Anpassung des Verfahrens abhängig von dem Aufwand dieser Operationen

2.2 Analyse von Algorithmen - Teil 1

- **Schleifeninvariante (SIV)**
 - Sonderform der Invariante
 - Am Anfang/Ende jedes Schleifendurchlaufs und vor/nach jedem Schleifendurchlauf gültig
 - Wird zur Feststellung der Korrektheit von Algorithmen verwendet
 - Eigenschaften:
 - Initialisierung: Invariante ist vor jeder Iteration wahr
 - Fortsetzung: Wenn SIV vor der Schleife wahr ist, dann auch bis Beginn der nächsten Iteration
 - Terminierung: SIV liefert bei Schleifenabbruch, helfende Eigenschaft für Korrektheit
 - Beispiel für Umsetzung: **Insertion Sort - SIV**
- **Laufzeitanalyse**
 - Aufstellung der Kosten und Durchführungsanzahl für jede Zeile des Quelltextes
 - Beachte: Bei Schleifen wird auch der Aufruf gezählt, der den Abbruch einleitet
 - Beispiel für Umsetzung: **Insertion Sort - Laufzeit**
 - Zusätzliche Überprüfung des **Best Case**, **Worst Case** und **Average Case**
- **Effizienz von Algorithmen**
 - Effizienzfaktoren
 - Rechenzeit (Anzahl der Einzelschritte)
 - Kommunikationsaufwand
 - Speicherplatzbedarf
 - Zugriffe auf Speicher
 - Laufzeit hängt von versch. Faktoren ab
 - Länge der Eingabe
 - Implementierung der Basisoperationen
 - Takt der CPU

2.3 Analyse von Algorithmen - Teil 2

- **Komplexität**
 - Abstrakte Rechenzeit $T(n)$ ist abhängig von den Eingabedaten
 - Übliche Betrachtungsweise der Rechenzeit ist asymptotische Betrachtung
- **Asymptotik**
 - Annäherung an einer sich ins Unendliche verlaufende Kurve
 - z.B.: $f(x) = \frac{1}{x} + x$ | Asymptote: $g(x) = x$ | ($\frac{1}{x}$ läuft gegen Null)
- **Asymptotische Komplexität**
 - Abschätzung des zeitlichen Aufwands eines Algorithmus in Abhängigkeit einer Eingabe
 - Beispiel für Umsetzung: **Insertion Sort - Laufzeit Θ**
- **Asymptotische Notation**
 - Betrachtung der Laufzeit $T(n)$ für sehr große Eingaben $n \in \mathbb{N}$
 - Komplexität ist unabhängig von konstanten Faktoren und Summanden
 - Nicht berücksichtigt: Rechnergeschwindigkeit / Initialisierungsauswände
 - Komplexitätsmessung via Funktionsklasse ausreichend
 - Verhalten des Algorithmus für große Problemgrößen

- Veränderung der Laufzeit bei Verdopplung der Problemgröße
- **Gründe für die Nutzung der theoretischen Betrachtung statt der Messung der Laufzeit**
 - Vergleichbarkeit
 - Laufzeit abhängig von konkreter Implementierung und System
 - Theoretische Betrachtung ist frei von Abhängigkeiten und Seiteneffekten
 - Theoretische Betrachtung lässt direkte Vergleichbarkeit zu
 - Aufwand
 - Wieviele Testreihen?
 - In welcher Umgebung?
 - Messen führt in der Ausführung zu hohem, praktischen Aufwand
 - Komplexitätsfunktion
 - Wachstumsverhalten ausreichend
 - Praktische Evaluation mit Zeiten nur für Auswahl von Systemen mögliche
 - Theoretischer Vergleich (Funktionsklassen) hat ähnlichen Erkenntnisgewinn

2.4 Analyse von Algorithmen - Teil 3

• Θ -Notation

- Θ -Notation beschränkt eine Funktion asymptotisch von oben und unten
- Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$ (\mathbb{N} : Eingabelänge, \mathbb{R} : Zeit)

$$\Theta(g) = \{f : \exists c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

\uparrow \uparrow
 Funktion f Positive Konstanten Für alle n größer gleich n_0 $f(n)$ wird von $c_1 g(n)$ und $c_2 g(n)$ für hinreichend große n eingeschlossen

- $\Theta(g)$ enthält alle f , die genauso schnell wachsen wie g
- Schreibweise: $f \in \Theta(g)$ (korrekt), manchmal auch $f = \Theta(g)$
- $g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$
- $f(n) = \Omega(g(n))$ gilt, wenn $f(n) = O(g(n))$ und $f(n) = \Omega(g(n))$ erfüllt sind

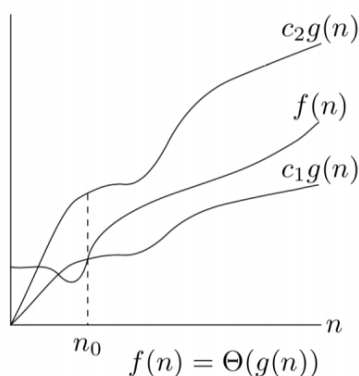


Abbildung 1: Veranschaulichung

- z.B.: $f(n) = \frac{1}{2}n^2 - 3n \mid f(n) \in \Theta(n^2)$
- Aus $\Theta(n^2)$ folgt, dass $g(n) = n^2$
- Vorgehen:
 - Finden eines n_0 und c_1, c_2 , sodass
 - $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ erfüllt ist
 - Konkret: $c_1 * n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 * n^2$
 - Division durch n^2 : $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$
 - Ab $n = 7$ positives Ergebnis: $0,0714 \mid n_0 = 7$
 - Deswegen setzen wir $c_1 = \frac{1}{14}$
 - Für $n \rightarrow \infty$: $0,5 \mid c_2 = 0,5$
 - Natürlich auch andere Konstanten möglich

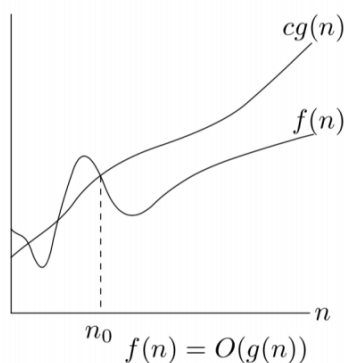
• O -Notation

- O -Notation beschränkt eine Funktion asymptotisch von oben

$$O(g) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

\uparrow \uparrow \uparrow
 Funktion f Positive Konstanten Für alle n größer gleich n_0
 $f(n)$ wird von $cg(n)$
für hinreichend große n
beschränkt

- $O(g)$ enthält alle f , die höchstens so schnell wie g wachsen
- Schreibweise: $f = O(g)$
- $f(n) = \Theta(g) \rightarrow f(n) = O(g) \mid \Theta(g(n)) \subseteq O(g(n))$
- Ist f in der Menge $\Theta(g)$, dann auch in der Menge $O(g)$



- z.B.: $f(n) = n + 2 \mid f(n) = O(n)$?
- Ja $f(n)$ ist Teil von $O(n)$ für z.B. $c = 2$ und $n_0 = 2$

Abbildung 2: Veranschaulichung

• O -Notation Rechenregeln

- Konstanten:
 - $f(n) = a$ mit $a \in \mathbb{R}$ konstante Funktion $\rightarrow f(n) = O(1)$
 - z.B. $3 \in O(1)$
- Skalare Multiplikation:
 - $f = O(g)$ und $a \in \mathbb{R} \rightarrow a * f = O(g)$
- Addition:
 - $f_1 = O(g_1)$ und $f_2 = O(g_2) \rightarrow f_1 + f_2 = O(\max\{g_1, g_2\})$
- Multiplikation:
 - $f_1 = O(g_1)$ und $f_2 = O(g_2) \rightarrow f_1 * f_2 = O(g_1 * g_2)$

• Ω -Notation

- Ω -Notation beschränkt eine Funktion asymptotisch von unten

$$\Omega(g) = \{f : \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

\uparrow \uparrow \uparrow
 Funktion f Positive Konstanten Für alle n größer gleich n_0
 $f(n)$ wird von $cg(n)$
für hinreichend große n
unten beschränkt

- Ω -Notation enthält alle f , die mindestens so schnell wie g wachsen

- Schreibweise: $f = \Omega(g)$

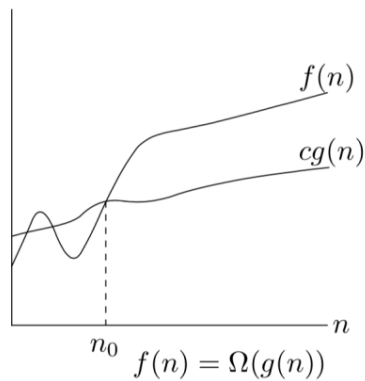


Abbildung 3: Veranschaulichung

• Komplexitätsklassen

- n ist hier die Länge der Eingabe

Klasse	Bezeichnung	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	Matrixmultiplikation
$\Theta(n^k)$	Polynomiell	
$\Theta(2^n)$	Exponentiell	Travelling-Salesman*
$\Theta(n!)$	Faktoriell	Permutationen

- Ausführungsdauer, falls eine Operation n genau $1\mu s$ dauert

Eingabe- größe n	$\log_{10} n$	n	n^2	n^3	2^n
10	$1\mu s$	$10\mu s$	$100\mu s$	1ms	$\sim 1ms$
100	$2\mu s$	$100\mu s$	10ms	1s	$\sim 4 \times 10^{16}y$
1000	$3\mu s$	1ms	1s	16min 40s	?
10000	$4\mu s$	10ms	1min 40s	$\sim 11,5d$?
10000	$5\mu s$	100ms	2h 46min 40s	$\sim 31,7y$?

- Asymptotische Notationen in Gleichungen

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
- $\Theta(n)$ fungiert hier als Platzhalter für eine beliebige Funktion $f(n)$ aus $\Theta(n)$
- z.B.: $f(n) = 3n + 1$

• o-Notation

- -Notation stellt eine echte obere Schranke dar
- Ausschlaggebend ist, dass es für alle $c \in \mathbb{R}_{>0}$ gelten muss
- Außerdem $<$ statt \leq
- z.B.: $2n = o(n^2)$ und $2n^2 \neq o(n^2)$

$$o(g) = \{f : \underbrace{\forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)}\}$$

Gilt für **alle** Konstanten $c > 0$.

In O-Notation gilt es für eine Konstante $c > 0$

- **ω -Notation**

- ω -Notation stellt eine echte untere Schranke dar
- Ausschlaggebend ist, dass es für alle $c \in \mathbb{R}_{>0}$ gelten muss
- Außerdem $>$ statt \geq
- z.B.: $\frac{n^2}{2} = \omega(n)$ und $\frac{n^2}{2} \neq \omega(n^2)$

$$\omega(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

2.5 Insertion Sort

- **Idee**

- Halte die linke Teilfolge sortiert
- Füge nächsten Schlüsselwert hinzu, indem es an die korrekte Position eingefügt wird
- Wiederhole den Vorgang bis Teilfolge aus der gesamten Liste besteht

- **Code**

```

1 FOR j = 1 TO A.length - 1
2   key = A[j]
3   // Füge A[j] in die sortierte Sequenz A[0...j-1] ein
4   i = j - 1
5   WHILE i >= 0 and A[i] > key
6     A[i + 1] = A[i]
7     i = i - 1
8   A[i + 1] = key

```

- **Schleifeninvariante von Insertion Sort**

- Zu Beginn jeder Iteration der **for**-Schleife besteht die Teilfolge $A[0 \dots j-1]$ aus den Elementen der ursprünglichen Teilfolge $A[0 \dots j-1]$ enthaltenen Elementen, allerdings in sortierter Reihenfolge.

- **Korrektheit von Insertion Sort**

- Initialisierung:
 - Beginn mit $j=1$, also Teilfeld $A[0 \dots j-1]$ besteht nur aus einem Element $A[0]$. Dies ist auch das ursprüngliche Element und Teilfeld ist sortiert.
- Fortsetzung:
 - Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Ausführungsblock der **for**-Schleife sorgt dafür, dass $A[j-1]$, $A[j-2]$,... je um Stelle nach rechts geschoben werden bis $A[j]$ korrekt eingefügt wurde. Teilfeld $A[0 \dots j]$ besteht aus ursprünglichen Elementen und ist sortiert. Inkrementieren von j erhält die Invariante.
- Terminierung:
 - Abbruchbedingung der **for**-Schleife, wenn $j > A.length - 1$. Jede Iteration erhöht j . Dann bei Abbruch ist $j = n$ und einsetzen in Invariante liefert das Teilfeld $A[0 \dots n-1]$ welches aus den ursprünglichen Elementen besteht und sortiert ist. Teilfeld ist gesamtes Feld.
- Algorithmus **Insertion Sort** arbeitet damit korrekt.

• Laufzeitanalyse von Insertion Sort

```

INSERTION-SORT (A)
1 FOR j = 1 TO A.length - 1
2   key = A[j]
3   // Füge A[j] in die
   //sortierte Sequenz A[0..j-1]
4   i = j - 1
5   WHILE i ≥ 0 and A[i] > key
6     A[i+1] = A[i]
7     i = i - 1
8   A[i+1] = key

```

Laufzeit:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n-1)$$

Zeile	Kosten	Anzahl
1	c_1	n
2	c_2	$n-1$
3	0	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=1}^{n-1} t_j$
6	c_6	$\sum_{j=1}^{n-1} (t_j - 1)$
7	c_7	$\sum_{j=1}^{n-1} (t_j - 1)$
8	c_8	$n-1$

- Festlegung der Laufzeit für jede Zeile
- Jede Zeile besitzt gewissen Kosten c_i
- Jede Zeile wird x mal durchgeführt
- $\text{Laufzeit} = \text{Anzahl} * \text{Kosten}$ jeder Zeile
- Schleifen: Abbruchüberprüfung zählt auch
- t_j : Anzahl der Abfragen der **While**-Schleife

- Warum n in Zeile 1?
 - Die Überprüfung der Fortführungsbedingung beinhaltet auch die letzte Überprüfung
 - Quasi die Überprüfung, durch die die Schleife abbricht
- Warum $\sum_{j=1}^{n-1}$ in Zeile 5?
 - Aufsummierung aller einzelnen t_j über die Anzahl der Schleifendurchläufe
 - Diese ist allerdings $n-1$ und nicht n , da die Abbruchüberprüfung dort auch enthalten ist
- Warum $t_j - 1$ in Zeile 6?
 - Selbes Argument wie oben, bei t_j ist die Abbruchüberprüfung enthalten
 - Deswegen wird die **while**-Schleife nur $t_j - 1$ -mal ausgeführt
- **Best Case**
 - zu sortierendes Feld ist bereits sortiert
 - t_j wird dadurch zu 1, da die **While**-Schleife immer nur einmal prüft (Abbruch)
 - Die zwei Zeilen innerhalb der **While**-Schleife werden nie ausgeführt
 - Durch Umformen ergibt sich, dass die Laufzeit eine lineare Funktion in n ist
- **Worst Case**
 - zu sortierendes Feld ist umgekehrt sortiert
 - t_j wird dadurch zu $j+1$, da die **While**-Schleife immer die gesamte Länge prüft
 - Durch Umformen ergibt sich, dass die Laufzeit eine quadratische Funktion in n ist (n^2)
- **Average Case**
 - im Mittel gut gemischt
 - t_j wird dadurch zu $j/2$
 - Die Laufzeit bleibt aber eine quadratische Funktion in n (n^2)

• Asymptotische Laufzeitbetrachtung Θ

- $T(n)$ lässt sich als quadratische Funktion $an^2 + bn + c$ betrachten
- Terme niedriger Ordnung sind für große n irrelevant
- Deswegen Vereinfachung zu n^2 und damit $\Theta(n^2)$

2.6 Bubble Sort

• Idee

- Vergleiche Paare von benachbarten Schlüsselwerten
- Tausche das Paar, falls rechter Schlüsselwert kleiner als linker

• Code

```

1 FOR i = 0 TO A.length - 2
2   FOR j = A.length - 1 DOWNT0 i + 1
3     IF A[j] < A[j-1]
4       SWAP(A[j], A[j-1])

```

- **Analyse von Bubble Sort**
 - Anzahl der Vergleiche:
 - Es werden stets alle Elemente der Teilfolge miteinander verglichen
 - Unabhängig von der Vorsortierung sind **Worst** und **Best Case** identisch
 - Anzahl der Vertauschungen:
 - **Best Case**: 0 Vertauschungen
 - **Worst Case**: $\frac{n^2-n}{2}$ Vertauschungen
 - Komplexität:
 - **Best Case**: $\Theta(n)$
 - **Average Case**: $\Theta(n^2)$
 - **Worst Case**: $\Theta(n^2)$

3 Pseudocode in der Vorlesung AuD

- Datentypen
 - String
 - Aufbau:
`"Die Summe ist"`
 - Konkatenation:
`"Die Summe ist" summe`
 - Array
 - A: Bezeichnung eines Arrays A
 - A[i] Zugriff auf (i+1)-tes Element des Arrays
- Methoden
 - Rückgabe:
`return summe`
- Schleifen
 - While-Schleife

```
WHILE summe <= n // Falls j = 1 to A.length -1: to ist dasselbe wie <=
    summe = summe + 1
ENDWHILE
```
- Variablen
 - Initialisierung
`summe := 0`