

---

# Einführung in die Informatik A

---

## 2.1 Aussagenlogik

- **Syntax:**

Aussagezeichen:  $AZ = \{A_1, A_2, \dots\}$

Zeichenvorrat:  $Z = AZ \cup \{(), \neg, \wedge, \vee\}$

Menge aller aussagenlogischen Formeln (AL) mit Eigenschaft:

jedes  $A_i$  ist in AL und  $F, G$  in AL  $\Rightarrow (\neg F), (F \wedge G), (F \vee G)$  in AL

- **Semantik:**

- Belegung:  $\alpha : AZ \rightarrow \{0,1\}$  und  $\alpha^* : AL \rightarrow \{0,1\}$

- $F$  gültig in  $\alpha$  ( $\alpha \models F$ )  $\Leftrightarrow \alpha^*(F) = 1$

- $F$  allgemeingültig  $\Leftrightarrow \alpha^*(F) = 1$  für alle Belegungen (Wahrheitstafel nur „1“-er)

- $F$  erfüllbar  $\Leftrightarrow$  mind. eine Belegung  $\alpha$  existiert mit  $\alpha \models F$   
(Wahrheitstafel mind einen „1“-er)

- $\alpha$  Modell von  $M$   $\Leftrightarrow \alpha \models F \forall F \in M$

- $M$  widerspruchsvoll  $\Leftrightarrow$  es gibt kein Modell für  $M$

- $F$  Folgerung von  $M$  ( $M \models F$ )  $\Leftrightarrow M \cup \{\neg F\}$  widerspruchsvoll  
(Prüfe  $M \wedge \neg F$  auf Erfüllbarkeit)

- Äquivalenzen

- Idempotenz:  $(F \wedge F) \equiv (F \vee F) \equiv F$

- Kommutativität:  $(F \wedge G) \equiv (G \wedge F)$  (analog  $\vee$ )

- Assoziativität:  $(F \wedge G) \wedge H \equiv F \wedge (G \wedge H)$  (analog  $\vee$ )

- Distributivität:  $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$  (analog  $+, *$ )

- Absorption:  $F \wedge (F \vee G) \equiv F$  (analog Vertauschung von  $\wedge$  und  $\vee$ )

- deMorgan:  $\neg(F \wedge G) \equiv \neg F \vee \neg G$  (analog Vertauschung von  $\wedge$  und  $\vee$ )

- *konjunktive Normalform (KNF):*  $\bigwedge \bigvee L_{ij}$  mit  $L_{ij}$  Literal also nur  $A$  bzw.  $\neg A$

- Algorithmus 1: Negationszeichen nach innen,  $\vee$  soweit wie möglich nach innen

- Algorithmus 2: Wahrheitstafel:  $\bigvee$  „1“-er:  $(L_1 \wedge L_2 \wedge \dots)$  ( $L_i$ :  $A_i$  für 1,  $\neg A_i$  für 0)

- *disjunktive Normalform (DNF):*  $\bigvee \bigwedge L_{ij}$  (s.o.)

- Algorithmus 1: wie KNF, nur  $\vee$  und  $\wedge$  vertauschen

- Algorithmus 2: wie KNF, aber 0 und 1 sowie  $\wedge$  und  $\vee$  vertauschen

- *Erfüllbarkeitstest* (Klauseln):

- i.  $F$  in KNF darstellen:  $F = (L_{11} \vee L_{12} \vee \dots) \wedge \dots \wedge (L_{k1} \vee L_{k2} \vee \dots)$

- ii.  $F$  als Klauselmengemenge:  $F = \{\{L_{11}, L_{12}, \dots\}, \dots, \{L_{k1}, L_{k2}, \dots\}\} = \{K_1, K_2, \dots\}$

- iii. Resolvent bilden:  $R = K_1 \setminus \{L\} \cup K_2 \setminus \{\neg L\}$  (übersichtlichere Darstellung als Baum!)

- iv. Falls  $\emptyset \in R \Rightarrow$  „ $F$  nicht erfüllbar“, sonst „ $F$  erfüllbar“

(Hinw.: formal  $\text{Res}(F) = F \cup \{R\}$ )

## 2.2 Prädikatenlogik:

- **Syntax**

	Menge	Symbole $\in$
Variablen:	$V = \{x, y, z, \dots\}$	$\{x, y, z, \dots\}$
Funktionszeichen:	$FZ = \{+, -, *, \dots\}$	$\{f, g, h, \dots\}$
Prädikatenzeichen:	$PZ = \{<, =, >, \dots\}$	$\{P, Q, R, \dots\}$
Junktoren:	$\{\neg, \wedge, \vee\}$	
Quantoren:	$\{\forall, \exists\}$	

- bindend  $\Leftrightarrow$  x direkt hinter  $\exists$  oder  $\forall$
- gebunden  $\Leftrightarrow$  x kommt in folgendem Ausdruck nochmal vor
- frei  $\Leftrightarrow$  sonst

- **Semantik**

- $\sigma = (U_\sigma, I_\sigma)$  mit  $U_\sigma$ : Individuenbereich und  $I_\sigma$ : Interpretation
- $\sigma^* = T_B \cup PL_B \ni U_\sigma \cup \{0,1\}$  mit  $T_B$ : Menge aller Terme und  $PL_B$ : Menge aller prädikatenlogischen Formeln mit Basis B
- weitere Äquivalenzen:
  - $\neg \forall x F \equiv \exists x \neg F$  bzw.  $\neg \exists x F \equiv \forall x \neg F$
  - $\forall x (F \wedge G) \equiv (\forall x F) \wedge (\forall x G)$  (Umkehrung mit  $\forall$  gilt nicht !!!)
  - $\exists x (F \vee G) \equiv (\exists x F) \vee (\exists x G)$  (Umkehrung mit  $\vee$  gilt nicht !!!)
- Unentscheidbarkeit:  $\neg \exists$  Algorithmus für Allgemeingültigkeit einer Formel
- Semi-Entscheidbarkeit: terminiert mit ja, falls F allgemeingültig bzw. terminiert nicht, sonst
- Umformung:
  - Umbenennung gebundener Variablen
  - Bereinigung (alle Quantoren verschiedene Variablen)
  - Pränex Normalform (Quantoren rausziehen)
- Satz:  $\forall x F \Rightarrow F[x/t] \Rightarrow \exists x F$

## 3.1 Spezifikation

- Problem  $\Rightarrow$  Algorithmus  $\Rightarrow$  Programm  $\Rightarrow$  Lösung
- Datentyp:  $D = \langle M, f_1, \dots, f_n \rangle$  mit  $M$ : Trägermenge und  $f$ : Funktionen bzw. Operationen
- funktionale Spezifikation:
  - Deklarationen (Konstanten, Typen, Funktionen, Prädikate)
  - Eingabe
  - Vorbedingungen
  - Ausgabe
  - Nachbedingungen

## 3.2 Algorithmus

- Finitheit der Beschreibung
- Effektivität (
- Determiniertheit
- Terminiertheit
- weitere Eigenschaften:
  - Korrektheit
  - Komplexität (Effizienz)
  - Robustheit
  - Genauigkeit

## 4.1 Verifikation

- *partielle Korrektheit*:  $\{P\} A \{Q\}$ :  
Algorithmus A mit Vorbedingung P und Nachbedingung Q heißt partiell korrekt  $\Leftrightarrow \forall$  Eingaben, die P erfüllen und für die A terminiert, ist anschließend Q erfüllt.
- Hoare Kalkül:
  - Axiome:
    - Zuweisungsaxiom:  $\{P[x/t]\} x := t; \{P\}$
  - Inferenzregeln:
    - Sequenzregel: 
$$\frac{\{P\}a_1\{Q\}, \{Q\}a_2\{R\}}{\{P\}a_1;a_2\{R\}}$$
    - Konsequenzregeln:
      - stärkere Vorbedingung: 
$$\frac{R \Rightarrow P \{P\}a\{Q\}}{\{R\}a\{Q\}}$$
      - schwächere Nachbedingung: 
$$\frac{\{P\}a\{Q\}, Q \Rightarrow R}{\{P\}a\{R\}}$$
    - Alternativregel:
      - IF: 
$$\frac{\{P \wedge p\}a\{Q\}, (P \wedge \neg p) \Rightarrow Q}{\{P\}IF p THEN a END\{Q\}}$$
      - IF ... ELSE: 
$$\frac{\{P \wedge p\}a_1\{Q\}, \{P \wedge \neg p\}a_2\{Q\}}{\{P\}IF p THEN a_1 ELSE a_2 END\{Q\}}$$
    - Iterationsregel (WHILE): 
$$\frac{\{P \wedge p\}a\{P\}}{\{P\}WHILE p DO a END\{P \wedge \neg p\}}$$
- Hauptproblem: Schleifeninvarianten finden  
Vorgehen:
  1. finde P, so daß
    - a)  $(P \wedge \neg \pi) \Rightarrow Q_0$  (Nachbedingung)
    - b)  $\{P \wedge \pi\} a \{P\}$  (rückwärts vorgehen)
  2. Zeige, daß Schleifeninvariante P auch vor Ausführung der Schleife gilt (rückwärts)
  $\{P_0\} a_0 \{P\}$  ( $\sim$  schwächste Vorbedingung)

## 4.2 Test

- Def. Testen: Programm mit Absicht ausführen, Fehler zu finden  
Testen erfolgreich falls Fehler aufgefunden, d.h. Testen ist ein destruktiver Prozess
- Code Inspektion: ...
- Black-Box-Test:  
Testen ohne Berücksichtigung der internen Programmstruktur
  - Äquivalenzanalyse (Äquivalenzklassen mit jeweiligem Repräsentanten betrachten)
  - Grenzwertanalyse (Grenzwerte der Äquivalenzklassen betrachten)
  - Ursachen-Wirkungs-Analyse (Bez. zwischen Eingabe- und Ausgabevariablen, Wirkungsgraph)
- White-Box-Test  
mit Berücksichtigung der internen Programmstruktur
  - Anweisungsüberdeckung (jede Anweisung einmal ausführen)
  - Entscheidungsabdeckung (jeden THEN, ELSE, CASE, WHILE Fall; funktioniert gut, wenn je nur eine Bedingung)
  - Bedingungs- und Entscheidungsabdeckung (Kombination der ersten beiden)
  - Mehrfach-Bedingungsabdeckung (jede Kombination der ersten drei jeweils durchlaufen; ist insbesondere bei Mehrfachbedingungen notwendig)

## 5. Analyse von Algorithmen

### 5.1 Ausführungskosten

- Speicherplatz  
i.A.  $\neq$  Anzahl der Variablen des Programms
- Rechenzeit
  - Zeitkomplexität bezieht sich (meist) auf schlechtesten Fall
  - hier: nur Betrachtung uniformer Kosten ( $\leftrightarrow$  logarithmische Kosten)
  - spezielle Maschine mit Eigenschaften:
    - Operationen auf Datenobjekten
    - Eingaben annehmen
    - Ausgabe liefern
    - Datenobjekte speichern
    - Algorithmus ausführen
  - spezielle abstrakte Maschine
    - Zielmaschine  $M_k$  mit
      - wesentlichen Eigenschaften konkreter Rechner
      - von technischen Details (so weit, wie möglich) abstrahiert
  - Random Access Machine (RAM)
    - Eingabe und Ausgabenband (unidirektional)
    - Zentrale Recheneinheit
      - Befehlszähler (i)
      - Akkumulator (Zielregister für/von Berechnungen mit Adresse 0)
      - Arithmetic Logical Unit (ALU): Operationsausführung
    - Befehlssatz:
      - LOAD, STORE
      - ADD, SUB, MUL, DIV
      - GOTO, IF... THEN...
      - READ, WRITE

$\Rightarrow$  Rechenzeit = Anzahl ausgeführter RAM Befehle
- O-Notation
  - $O(f) := \{ g \mid \exists c > 0: g(n) \leq c f(n) \}$  (verkürzt!) d.h. g wächst nicht wesentlich stärker als f
  - $\Omega(f)$ : Gegenteil von O
  - $\theta(f)$ : erfüllt  $O(f)$  und  $\Omega(f)$
  - $O(1)$ : konstante Komplexität
  - $O(\log n)$ : logarithmische Komplexität
  - $O(n)$ : lineare Komplexität
  - $O(n \log n)$ : leicht überlineare Komplexität
  - $O(n^2)$ : quadratische Komplexität
  - $O(2^n)$ : exponentielle Komplexität
  - $P = \cup O(n^k)$ : polynomielle Komplexität
  - NP: nichtdeterministisch polynomiell lösbare Probleme (bei gegebener Lösung)  
Satz: es ist nicht bekannt, ob  $P=NP$  oder  $P \neq NP$  gilt

## 5.2 Sortierprobleme

- Vorr.:
  - Sortieren nur über Größenvergleiche
  - Zeitkomplexität = #Vergleiche + #Zugriffe auf Array-Komponenten
  - Universum  $U$ : nur Ordnungsrelation „ $\geq$ “ bekannt
- **Sortieren durch Auswahl (SA)**
  - Bestimme für  $k=n$  bis  $k=2$  das Maximum  $s_m$  von  $(s_1, \dots, s_k)$  und vertausche  $s_k$  und  $s_m$ .
  - $O(n^2)$
  - Satz: jeder Algorithmus (mit binären Vergleichen) braucht mindestens  $n-1$  Schritte zur Bestimmung des Maximums aus  $(s_1, \dots, s_n) \rightarrow$  Tennisspieler
- **Sortieren durch Einfügen (SE)**
  - Beginne mit leerer Liste
  - Füge Listenelemente der Reihe nach an der richtigen Position ein  $\Rightarrow O(n^2) + O(n^2) = O(n^2)$
  - Verbesserung: SE mit binärer Suche (SEB)
    - $T^{\min}(n) \in O(n \log n)$
- **Sortieren durch Vertauschen: Bubble Sort (BS)**
  - „leichteste“ Elemente wandern nach „oben“ durch  $\Rightarrow O(n^2)$
  - BS ist Kombination von SE und SA
  - Verbesserung: Aufhören wenn nicht mehr vertauscht werden kann  $\Rightarrow T^{\min} \in O(n)$
- **Sortieren durch Teilen: Quicksort (QS)**
  - Prinzip des "Divide-and-Conquer": Problem in Teilprobleme zerlegen und diese lösen, anschl. Lösungen kombinieren
  - Vorgehen:
    - i.  $(s_l, \dots, s_r)$  Partition,  $s_l = x$
    - ii. von links suchen nach erstem  $s_i > x$
    - iii. von rechts suchen nach erstem  $s_j \leq x$
    - iv. Falls  $i < j \Rightarrow$  vertausche  $s_i$  und  $s_j$  und wiederhole (ii) und (iii)
    - v. Vertausche  $s_j$  und  $x$
  - $T^{\max} \in O(n^2)$ ,  $T^{\min} \in O(n \log n) \sim$  im Mittel gutes Verhalten
  - Verbesserung: Wähle das mittlere Element von  $(s_l, \dots, s_r)$  als  $x$   
 $\Rightarrow$  Anzahl der Vergleiche im schlechtesten Fall halbiert, im Mittel um 15% reduziert
- **Sortieren durch kluges Abspeichern: Heapsort (HS)**
  - Baum: linker Sohn:  $\text{links}(i) = 2i$ , rechter Sohn:  $\text{rechts}(i) = 2i + 1$
  - BuildHeap: Vertausche Knoten solange, bis in jedem Knoten der Wert größer als oder gleich den Werten in den Söhnen
  - FOR  $i := 1$  TO  $n-1$   
   Vertausche  $s_1$  und  $s_{n-1}$   
   reheap (  $s, 1, n-i-1$  )
  - $T_{\text{BuildHeap}} \in O(n)$ ,  $\sum T_{\text{reheap}} \in O(n)$ ,  $T_{\text{Vertauschung}} \in O(n) \Rightarrow O(n)$
  - Verbesserung: Bottom-Up-Heapsort: ???  
 $\Rightarrow$  ab  $n=16000$  schneller als Clever-Quicksort
  - Satz: jedes allgemeine Sortierverfahren benötigt im Mittel und im schlechtesten Fall mind.  $\Omega(n \log n)$  Vergleiche  
 $\Rightarrow$  Heapsort ist optimal
- **Sortieren durch Dosenwerfen: Binsort**
  - Vorr.: Elemente  $k \in O(1)$  (z.B. ASCII Zeichensatz)
  - Vorgehen:
    - i. Dosen  $D_1, D_2, \dots, D_k$  aufstellen
    - ii. FOR  $i:=1$  TO  $n$  DO wirf  $s_i$  in Dose  $D_{s_i}$
    - iii. FOR  $j:=1$  TO  $k$  DO enleere Dose  $D_j$
  - $T_{\text{Binsort}}(n) \in O(n)$   
 $\hat{=}$  sehr effizient unter dieser Voraussetzung
- **Erkenntnis:**
  - Dive-andConquer oft effizient
  - Algo optimal? Aussage nur bei Kenntnis unterer Schranken, welche von Vor. abhängen

## 6. Entwurfsmethoden für Algorithmen

- Ziele
  - Korrektheit
  - Effizienz
  - niedrige Entwurfskosten
  - niedrige Wartungs- und Updatekosten
- **Entwurfsprinzipien** (bei jedem Entwurf einsetzbare Entwurfsmethoden)  $\hat{=}$  [Prog I]
  - i. Zerlegung (Modularisierung)
  - ii. Daten- und Programmstrukturierung
  - iii. schrittweise Verfeinerung
- **Entwurfstechniken** (problemspezifisch anzuwendende Verfahren und Lösungswege)
  - Divide-and-Conquer  
Zeitaufwand für Problem der Größe  $n$ :  $T(n) = k T(n/c) + d(n)$  mit  $d(n)$ : Aufwand für Aufteilen/Zusammenführen
  - Transformation: Transformiere  $P$  in  $Q \hat{=}$  Löse  $Q \hat{=}$  Rücktransformation
  - Systematische Suche  
 $P$  ist NP-vollständig  $\Leftrightarrow P$  ist  $NP \wedge Q \in NP$  läßt sich (mit polynomiellm Aufwand) in  $P$  transformieren und Lösung von  $P$  läßt sich in Lösung von  $Q$  transformieren  
 $\Rightarrow$  Ist ein einziges NP-vollständiges Problem in polynomieller Zeit lösbar, dann auch alle anderen Probleme aus NP
    - Backtracking: Konstruiere sukzessive immer größere Teilmenge, falls nicht fortsetzbar zurück zur letzten Teillösung
    - Branch-and-Bound: Baumdiagramm mit Abbruchkriterium (bound-Funktion) um Ast so früh, wie möglich, abzuschneiden
    - Greedy Algorithmus: wähle jeweils das lokale Optimum (z.B. kürzeste Strecke zu nächstem Ort)
    - Dynamisches Programmieren:  $\hat{=}$  [OR]
  - Randomisieren (z.B. Monte Carlo Methode zur Integration)
  - Parallelisieren

## 7. Datenstrukturen, abstrakte Datentypen

- Datentyp  $D = (M, f_1, \dots, f_n)$
- **skalare Datentypen**  $\Leftrightarrow$  alle Werte atomar, Wertebereich total geordnet
  - Indextypen: BOOLEAN, CARDINAL, INTEGER, CHAR, Aufzählungstypen
  - keine Indextypen: REAL, LONGREAL, SHORTREAL
- **strukturierte Datentypen**  $\Leftrightarrow$  Werte sind Zusammenfassungen von Werten anderer Datentypen
  - homogen: alle Komponenten vom gleichen Typ (z.B. ARRAY)
  - inhomogen: Komponenten unterschiedlichen Typs (z.B. RECORD)
  - Anzahl der Komponenten: fest (statisch)  $\leftrightarrow$  variabel (dynamisch)
- **statische Datentypen**  $\Leftrightarrow$  Struktur der Objekte liegt fest, kann während Laufzeit nicht verändert werden
  - ARRAY, RECORD, SET
- **dynamische Datentypen**  $\Leftrightarrow$  Erzeugung/ Vernichtung/ Größenänderung/ Strukturänderung während Laufzeit möglich
  - POINTER: ...
  - SEQUENZ (EOF, WRITE, PUT, OPEN, ...)
  - Bäume (Darstellung durch POINTER oder ARRAY (nur bei Heaps))
    - Suchbäume: ...
- **abstrakte Datentypen (ADT)**  $\Leftrightarrow$  Implementierung bleibt Benutzer verborgen
  - $D = (S, \Sigma, E)$  mit  $S$ : nichtleere Menge von Sorten,  $\Sigma$ : Signatur (Mege von Operatorsymbolen mit Argumentsorten und Ergebnissorte,  $E$ : Gleichungen/Axiome
  - Stack (Stapel, Keller) mit Operationen: empty, newstack, psuh, pop, head (LIFO - Prinzip)  
 $\hat{=}$  Anwendung: wohlgeformten Klammerausdruck (WKA) prüfen
  - Queue (Warteschlange) mit Operationen empty, newqueue, enqueue, dequeue, front (FIFO - Prinzip)  
 $\hat{=}$  Anwendungen: Postschalter, Druckerschlange

**Anhang: Komplexität der Sortierverfahren**

<i>Sortierverfahren</i>	<i>Abk.</i>	<i>Kurzbeschreibung</i>	<i>WORST</i>	<i>AVG</i>	<i>BEST</i>
<b>Sortieren durch Auswahl</b>	SA	vertausche max mit letztem Element	$O(n^2)$	$\theta(n^2)$	$\theta(n^2)$
<b>Sortieren durch Einfügen</b>	SE	Füge jew. Element an richtiger Position in leere Folge ein	$O(n^2)$	$O(n^2)$	$O(n)$
Verbesserung: <i>binäre Suche</i>	SEB	Pos. per BinarySearch bestimmen	"	"	$O(n \log n)$
<b>Bubble Sort</b> (Vertauschen)	BS	leichtestes Element wandert nach oben durch	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$
Verbesserung: <i>Aufhören</i>	---	Stoppen wenn nicht mehr vertauscht	"	"	$O(n)$
<b>Quicksort</b> (Teilen)	QS	in Teilprobl. zerlegen, diese lösen	$\theta(n^2)$	$\theta(n \log n)$	$\Omega(n \log n)$
Verbesserung: mittleres Element wählen	---	mittleres statt erstes Element auswählen	50% +	15% +	
<b>Heapsort</b> (Baum)	HS	größtes Element wandert nach vorne, anschl. ans Ende setzen	$\Omega(n \log n)$	$\Omega(n \log n)$	
<b>Binsort</b> (Dosenwerfen)	Bin	Dose für jedes Element aufstellen, in Dosen werfen, ausschütten	$O(n)$	$O(n)$	$O(n)$