Gliederung der Grundvorlesung

1. Grundlagen der Programmierung

~~1.1 Algorithmen und Sprachen~~
~~1.2 Aussagen über Algorithmen~~
~~1.3 Daten und ihre Strukturierung~~
~~1.4 / 1.5 Grundbegriffe der Programmierung und die Sprache Ada 95~~
1.6 Komplexität von Algorithmen und Programmen
1.7 Semantik von Programmen

Gliederung des Kapitels

1.6 Komplexität von Algorithmen und Programmen

1.6.1 Rechenmodell "Turingmaschine"
1.6.2 Churchsche These
1.6.3 Komplexitätsklassen
1.6.4 Beispiele
1.6.5 Andere Rechenmodelle

1.6.1 Rechenmodell Turingmaschine

Turingmaschinen wurden bereits im November vorgestellt (siehe handschriftliche Folien im Netz unter 21.11.2002).

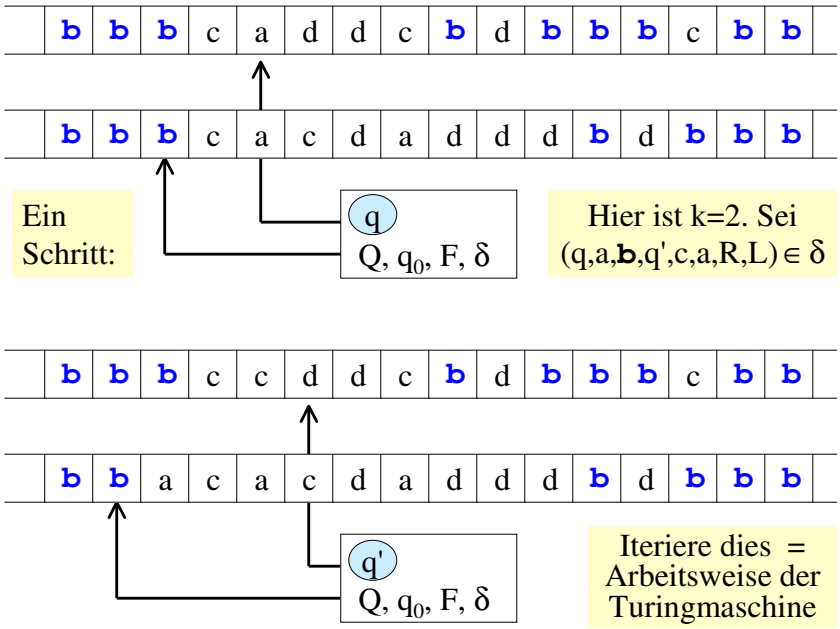
1.6.1.1 Definition (nach A. M. Turing, 1912-1954, engl. Mathem.)
$M = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$ heißt (nichtdeterministische) k-Band-Turingmaschine $\Leftrightarrow$

(1) Q ist eine nicht-leere endliche Menge ("Zustandsmenge"),
(2) Σ ist eine endliche Menge ("Eingabealphabet"),
(3) Γ ist eine endliche Menge ("Bandalphabet") mit , $\Sigma \subset \Gamma$,
(4) $q_0 \in Q$ ist der Anfangszustand,
(5) $F \subset Q$ ist die Menge der (akzeptierenden) Endzustände,
(6) $\mathbf{b} \in \Gamma \setminus \Sigma$ ist das Blanksymbol,
(7) $k \in \mathbb{N}$ ist die Anzahl der Bänder,
(8) $\delta \subseteq Q \setminus F \times \Gamma^k \times Q \times \Gamma^k \times \{L, 0, R, H\}^k$ ist die Überführungsrelation.

1.6.1.2 Definition
Obiges $M = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$ heißt deterministische k-Band-Turingmaschine, wenn zusätzlich gilt:

δ ist eine (partielle) Abbildung bzgl. der ersten beiden Komponenten, d.h.: Zu jedem $q \in Q$ und $a \in \Gamma^k$ existiert höchstens ein $(q, a, q', a', B) \in \delta$.
Man schreibt dann δ als (partielle) Abbildung
$\delta: Q \setminus F \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, 0, R, H\}^k$
und nennt δ die Überführungs- oder Übergangsfunktion von M.

Bei einer deterministischen Maschine kann es zu jedem Zeitpunkt höchstens eine Fortsetzungsmöglichkeit geben.
Eine nichtdeterministische Maschine verhält sich wie eine Grammatik: Der nächste Schritt ist nicht eindeutig bestimmt, vielmehr kann die Maschine eventuell eine von mehreren Fortsetzungsmöglichkeiten willkürlich auswählen.



1. Anfangssituation: "Ein Wort $w \in \Sigma^*$ wird eingegeben" bedeutet:

Alle Felder aller Bänder sind anfangs mit dem Blanksymbol beschriftet. Dann wird das Wort $w \in \Sigma^*$ auf das erste Band geschrieben, der Lese-Schreibkopf wird auf dem ersten Zeichen von w positioniert und die Turingmaschine befindet sich im Zustand q_0. Alle anderen Lese-Schreibköpfe befinden sich irgendwo auf den anderen Bändern.

2. Berechnung:

Es wird solange ein Schritt ausgeführt, bis man auf "H" in der Übergangsrelation trifft oder bis man auf eine Situation trifft, zu der es keine Folgesituation in δ gibt. Befindet sich M zu diesem Zeitpunkt in einem Endzustand, so ist die Berechnung erfolgreich abgeschlossen, anderenfalls erfolglos.

3. Ausgabe:

Am Ende wird das Wort ausgegeben, das auf dem ersten Band vom linkesten Nicht-Blanksymbol bis zum rechtesten Nicht-Blanksymbol steht. Dies ist ein Wort $v \in \Gamma^*$. Im nichtdeterministischen Fall wird einem Eingabewort also eine Resultats-Menge

$Res_M(w) = \{v \mid$ es gibt eine endliche Berechnung, die für die Eingabe w in einem Endzustand endet und hierbei v als Ausgabe besitzt$\}$

als Ergebnis zugeordnet (eventuell ist diese Menge leer). Im deterministischen Fall besitzt $Res_M(w)$ höchstens ein Element, d.h. Res_M ist dann eine (partielle) Abbildung $Res_M: \Sigma^* \to \Gamma^*$.

1.6.1.3 Definition:

Diese Menge bzw. diese Abbildung Res_M ist die Bedeutung ("Semantik") der Turingmaschine M.

1.6.1.4: Beispiele finden Sie auf den handschriftlichen Folien vom 21.11.02: Quersumme einer binär dargestellten Zahl und Palindrome. Hierfür wurde auch eine Aufwandsabschätzung angegeben (7n+2 bzw. bis zu $(n+1)^2$). [Palindrom = Wort, das vorwärts und rückwärts gelesen gleich ist; Pal sei die Menge der Palindrome über einem gegebenen Alphabet.]

Wir zeigen nun: Hat man für die Berechnung mindestens zwei Bänder zur Verfügung, so benötigt man nur O(n) Schritte.

Idee: Es sei $\Sigma = \{0,1\}$. Kopiere die Eingabe, also den Inhalt des ersten Bandes auf das zweite Band und vergleiche dann schrittweise das Wort auf dem ersten Band rückwärts mit dem Wort auf dem zweiten Band. Dies liefert folgende deterministische 2-Band-Turingmaschine M_{Pal}.

$M_{Pal} = (Q, \Sigma, \Gamma, \delta, q_0, F, \mathbf{b}, k)$ mit $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\mathbf{b}\}$, $k = 2$ und δ sei durch folgende Tabelle gegeben (für alle $x, y \in \Sigma$, für alle $c, d \in \Gamma$ mit $c \neq d$):

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	0	**b**	q_0	0	0	R	R
q_0	1	**b**	q_0	1	1	R	R
q_0	**b**	**b**	q_1	**b**	**b**	L	L
q_1	c	y	q_1	c	y	0	L
q_1	c	**b**	q_2	c	**b**	0	R
q_2	x	x	q_2	**b**	**b**	L	R
q_2	**b**	**b**	q_4	1	d	H	H
q_2	c	d	q_3	**b**	**b**	L	0
q_3	x	d	q_3	**b**	d	L	0
q_3	**b**	d	q_4	0	d	H	H

Erläuterung: Im Zustand q_0 kopiert M_{Pal} das Eingabewort w auf das zweite Band. Im Zustand q_1 wird der Lese-Schreibkopf des zweiten Bandes an den Anfang des kopierten Wortes gesetzt. Im Zustand q_2 werden die Wörter verglichen. Lag ein Palindrom vor, so gelangt man direkt in den Zustand q_4, sonst zunächst in q_3.

Diese Turingmaschine M_{Pal} berechnet folgende Resultatsfunktion $Res_{M_{Pal}}$

$Res_{M_{Pal}}: \{0,1\}^* \to \{0,1,\mathbf{b}\}^*$ mit

$$Res_{M_{Pal}}(w) = \begin{cases} 1, & w \text{ ist ein Palindrom} \\ 0, & w \text{ ist kein Palindrom} \end{cases}$$

Da alle Ergebniswerte aus $\{0,1\}$ sind, kann man diese Abbildung auch als Abbildung von $\{0,1\}^*$ in $\{0,1\}^*$ auffassen.

Hinweis: $Res_{M_{Pal}}$ ist die charakteristische Funktion χ_{Pal} der Menge Pal der Palindrome. [Denn für jedes $L \subseteq \Sigma^*$ heißt die Abbildung $\chi_L : \Sigma^* \to \{0,1\}$ mit $\chi_L(w) =$ if $w \in L$ then 1 else 0 fi die charakteristische Funktion von L, siehe Anmerkg. 1.2.2.4]

Wieviel Zeit (d.h. wie viele einzelne Schritte) benötigt diese Maschine, bis sie anhält? Wir messen dies stets in Abhängigkeit von der Länge n der Eingabe w (n = |w|).

M_{Pal} ist so gebaut, dass sie in jedem Fall 3n+3 Schritte durchführt: n+1 Schritte, um zum Ende der Eingabe zu laufen, n+1 Schritte, um bis zum Anfang des kopierten Wortes zurückzulaufen, und n+1 Schritte, um das Eingabewort von rechts nach links zu löschen, zugleich mit dem kopierten Wort abzugleichen und am Ende eine 0 oder 1 zu drucken.

Also lässt sich Pal mit 2 Bändern in O(n) Schritten entscheiden. (Wir haben sogar $\Theta(n)$ bewiesen.) ■

1.6.1.5 Definition:

(1) Für zwei Alphabete Σ und Δ heißt eine Abbildung $f: \Sigma^* \to \Delta^*$ Turing-berechenbar (oder partiell-berechenbar oder oft kurz berechenbar), wenn es eine deterministische Turingmaschine M mit $f = Res_M$ gibt (vgl. 1.1.3.5).

(2) Ist f zusätzlich eine totale Funktion (also auf ganz Σ^* definiert), so heißt f total-berechenbar oder total-rekursiv.

(3) Eine Menge $L \subseteq \Sigma^*$ heißt entscheidbar, wenn ihre charakteristische Funktion χ_L total-berechenbar ist (vgl. 1.2.2.3).

(4) Eine Menge $L \subseteq \Sigma^*$ heißt Haltebereich einer Turingmaschine, wenn es eine deterministische Turingmaschine gibt, die genau für alle Wörter aus L in einem Endzustand anhält.

(5) Eine Menge $L \subseteq \Sigma^*$ heißt aufzählbar, wenn sie Haltebereich einer (deterministischen) Turingmaschine ist.

1.6.1.6: Beispiel "wiederholungsfreie Wörter"

Es sei Σ ein Alphabet. Ein Wort $w \in \Sigma^*$ heißt wiederholungsfrei, wenn es keine Wörter x, y und z gibt mit $w=xyyz$ (mit $y \neq \varepsilon$). Kein echtes Teilwort kommt also in w zweimal hintereinander vor. Insbesondere darf kein Buchstabe auf sich selbst folgen.

Ist Σ ein- oder zweielementig, so gibt es nur endlich viele wiederholungsfreie Wörter (bitte ausprobieren!).

Ist Σ dagegen mindestens dreielementig, so gibt es nur unendlich viele wiederholungsfreie Wörter. Betrachte etwa $\Sigma = \{a, b, c\}$, so kann man solche beliebig langen Wörter konstruieren:

a b a c b a b c a b a c b c a b c a c b a b c ...

Es sei $WF = \{w \in \Sigma^* \mid w \text{ ist wiederholungsfrei}\}$.
Aufgabe: Entscheide zu $w \in \Sigma^*$, ob w in WF liegt oder nicht, d.h.: Berechne die charakteristische Funktion X_{WF} zu WF.

Lösungsansatz, deterministisch:

Sei Σ mindestens dreielementig. Sei $n=|w|$ und $w = w_1 w_2 \ldots w_n$ mit $w_i \in \Sigma$ für i =1, 2, ..n. Falls $n \leq 1$, dann ist $w \in$ WF.

Anderenfalls prüfe für jede Position i ($1 \leq i \leq n-1$) und für jede Länge k ($1 \leq k \leq (n-i+1)$ div 2, bzw. $i+2k-1 \leq n$), ob $w_i w_{i+1} .. w_{i+k-1} = w_{i+k} w_{i+k+1} \ldots w_{i+2k-1}$ gilt.
Trifft dies für kein i und k zu, so ist w wiederholungsfrei.

Wir formulieren diese Überprüfung in Ada, wobei wir das Sprachelement exit verwenden. Dieses hat die Syntax:
exit_statement ::= exit [*loop*-name] [when condition];

Bedeutung: Verlasse die umgebende Schleife, sofern die Bedingung hinter when zutrifft.

Mit exit kann man Schleifen verlassen. Gibt man nichts an, so wird die innerste Schleife, die die exit-Anweisung umfasst, verlassen. Will man mehrere Schleifen gleichzeitig verlassen, wie in unserem Fall, so muss die Schleife, die zu verlassen ist, einen Namen erhalten, den man in der exit-Anweisung angibt.

```
Schleife_i:
  for i in 1..n-1 loop
    for k in 1..(n+i-1)/2 loop
      wh := true;
      for j := i to i+k-1 loop wh := wh and (W(j)=W(j+k)); end loop;
      exit Schleife_i when wh;
    end loop;
  end loop;
if wh then "das Eingabewort besitzt eine Wiederholung"
else "das Eingabewort ist wiederholungsfrei" end if;
```


Zeitaufwand dieses Algorithmus?

Der am längsten dauernde Fall liegt vor, wenn das Wort wiederholungsfrei ist. Dann werden alle Kombinationen durchprobiert.
Hierbei können i und k in der Größenordnung von n/2 liegen, d.h., die beiden äußeren Schleifen benötigen bereits $O(n^2)$ Schritte.
Die innerste Schleife erfordert nochmals bis zu n/2 Schritte. Insgesamt erhalten wir somit einen Zeitaufwand von $O(n^3)$.

Ein zusätzlicher *Speicheraufwand* (außer konstant viel Platz für die Hilfsvariablen i, j, k, wh und Zwischenergebnisse wie (n+i-1)/2) entsteht nicht, da der Algorithmus nur auf dem Wort W arbeitet.

Einspruch: Genau genommen stimmt dies aber nicht. Es fällt doch zusätzlicher Speicherplatz in Abhängigkeit von der Länge der Eingabe an.

Denn es müssen die natürlichen Zahlenwerte von i, j, k und n dargestellt werden.

Um die Zahl n darzustellen, braucht man log(n) Ziffern in der Binärdarstellung und in jeder anderen Darstellung ebenfalls O(log(n)) Ziffern. Für wh braucht man dagegen nur eine Ziffer.

Also erfordert der Algorithmus insgesamt zusätzlichen Speicherplatz in der Größenordnung 4 log(n), also O(log(n)). In der Praxis ist dies eine geringe Größe.

Wie realisiert man diesen Algorithmus nun mit einer deterministischen Turingmaschine?

Das Eingabewort wird auf dem ersten Band abgelegt. Die Variablen i, j, k und n werden auf 4 Bändern gespeichert. Die Vergleiche und das Zählen ist höchstens in der Größenordnung der Länge der Darstellungen, also O(log(n)).

Da ein Direktzugriff W(j) auf einem Turingband nicht möglich ist, muss man die Abfrage W(j)=W(j+k) mit k Schritten durchführen, wobei man ggf. noch eine Hilfsvariable verwendet, um j oder k nicht löschen zu müssen.

Eine 6-Band-Turingmaschine müsste das Problem also in $O(n^4)$ Schritten lösen können. Die konkrete Turingmaschine wird relativ groß und daher nicht ausformuliert.

Lösungsansatz, nichtdeterministisch:

Sei Σ mindestens dreielementig. Sei $n = |w|$ und $w = w_1 w_2 ... w_n$ mit $w_i \in \Sigma$ für i =1, 2, ..n. Falls $n \leq 1$, dann ist $w \in$ WF.

Anderenfalls erzeuge man nichtdeterministisch eine Position i ($1 \leq i \leq n-1$) und eine Länge k ($1 \leq k \leq (n-i+1)$ div 2). Für diese prüft man, ob
$w_i w_{i+1} .. w_{i+k-1} = w_{i+k} w_{i+k+1} ... w_{i+2k-1}$ gilt.
Trifft dies zu, so ist w nicht wiederholungsfrei.
Anderenfalls geht man nicht in einen Endzustand.

Diesen Nichtdeterminismus können wir nicht mehr in einer gängigen Programmiersprache ausdrücken, wohl aber mit einer nichtdeterministischen Turingmaschine mit geeignet vielen Bändern. Hier reichen ebenfalls 6 Bänder sicher aus.

Die nichtdeterministische Turingmaschine ermittelt zunächst die Länge n der Eingabe w und stellt dann in O(log(n)) Schritten irgendwelche Werte für die Variablen i, j, k her.
Dann wird in $O(n^2)$ Schritten die j-Schleife simuliert. Falls anschließend wh true ist, geht die Turingmaschine in einen Endzustand, anderenfalls hält sie an, ohne in einen Endzustand zu gelangen.
Somit kann eine nichtdeterministische 6-Band-Turingmaschine das Problem, ob w nicht in WF liegt, in $O(n^2)$ Schritten lösen. Man sagt, nichtdeterministisch ist das Komplement von WF in quadratischer Zeit lösbar, während mit unserem Algorithmus oben der deterministische Fall $O(n^4)$ Schritte braucht.
Der zusätzliche Aufwand für den Speicherplatz ist in beiden Fällen gleich. ■

1.6.2 Churchsche These

Warum befassen wir uns mit Turingmaschinen?

Bisher haben wir Algorithmen in einer Programmiersprache als Programme formuliert. Diese Programme müssen einer Maschine (einem Computer, einem Rechner, einer Datenverarbeitungsanlage) übergeben werden, die sie ausführt. Mit Hilfe eines Compilers lassen sich die Programme in die Maschinensprache eines Computers übertragen und vom Computer durchrechnen. Die Behauptung lautet nun:

1.6.2.1: Jeder Algorithmus lässt sich durch eine geeignete deterministische Turingmaschine darstellen.

Dies wollen wir plausibel machen.

Beispiel 1.6.2.2: Betrachte folgenden Algorithmus, der hier als Block in Ada notiert ist:

```
z, s: natural;
begin   get (z); s := 3;
        while z > 1 loop
                s := s + z;
                z := z - 2;
        end loop;
        put (s);
end;
```

Dieses Programmstück lässt sich durch eine deterministische Turingmaschine M realisieren. Diese verwendet die Binärdarstellung für natürliche Zahlen und benutzt nur zwei Bänder: das erste für z und das zweite für s.

$M = (Q, \Sigma, \Gamma, \delta_M, q_0, F, \mathbf{b}, k)$ mit $Q = \{q_0, q_1, q_2, ..., q_{15}\}$, $F = \{q_{15}\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\mathbf{b}\}$, $k = 2$ und δ_M sei durch folgende Tabelle gegeben (für alle $x, y \in \Sigma$, für alle $c, d \in \Gamma$):

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_0	c	**b**	q_1	c	1	0	R
q_1	c	**b**	q_2	c	1	0	0
q_2	x	y	q_2	x	y	R	0
q_2	**b**	y	q_3	**b**	y	L	0
q_3	**b**	y	q_{14}	**b**	y	0	0
q_3	0	y	q_4	0	y	L	0
q_3	1	y	q_4	1	y	L	0

Erläuterung: Anfangs steht z ohnehin auf dem ersten Band (=get(z)). Mit den Zuständen q_0 und q_1 schreibt die Turingmaschine die Zahl 3 (binär 11) auf das zweite Band. Der zweite Lese-Schreibkopf steht auf dem letzten Zeichen von s. Mit q_2 wird das Ende von z aufgesucht. Mit q_3 beginnt dann die while-Schleife. Es ist $z>1$ zu prüfen. $z \leq 1$ kann durch das leere Wort oder durch eine alleinstehende 0 oder 1 erkannt werden. Dies wird in den Zuständen q_3 und q_4 geprüft. Ist $z \leq 1$, so wird nach Zustand q_{14} verzweigt, anderenfalls nach q_6.

Additionstabelle ohne Übertrag

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_4	**b**	y	q_5	**b**	y	R	0
q_4	x	y	q_6	x	y	R	0
q_5	x	y	q_{14}	**b**	y	0	0
q_6	**b**	**b**	q_8	**b**	**b**	R	R
q_6	**b**	0	q_6	**b**	0	L	L
q_6	**b**	1	q_6	**b**	1	L	L
q_6	0	**b**	q_6	0	0	L	L
q_6	0	0	q_6	0	0	L	L
q_6	0	1	q_6	0	1	L	L
q_6	1	**b**	q_6	1	1	L	L
q_6	1	0	q_6	1	1	L	L
q_6	1	1	q_7	1	0	L	L

Erläuterung: Der Zustand q_5 dient dazu, das erste Band auf jeden Fall zu leeren, da am Ende s auf Band 1 kopiert werden muss (das Ergebnis muss am Ende auf dem ersten Band stehen, siehe 1.6.1.2, 3. Ausgabe).

Ist $z > 1$, so wird der erste Lese-Schreibkopf auf das letzte Zeichen von z gestellt. q_6 bedeutet den Eintritt in den Rumpf der Schleife. Es ist $s := s+z$ zu berechnen. Dies geschieht wie bekannt stellenweise von rechts nach links, wobei das Blanksymbol wie die 0 behandelt wird und M sich den Übertrag im Zustand merkt (q_6 = kein Übertrag, q_7 = es liegt ein Übertrag vor). Die Addition ist beendet, wenn auf beiden Bändern das Blanksymbol gelesen wird (danach: Zustand q_8).

Additionstabelle mit Übertrag

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_7	**b**	**b**	q_8	**b**	1	R	R
q_7	**b**	0	q_6	**b**	1	L	L
q_7	**b**	1	q_7	**b**	0	L	L
q_7	0	**b**	q_6	0	1	L	L
q_7	0	0	q_6	0	1	L	L
q_7	0	1	q_7	0	0	L	L
q_7	1	**b**	q_7	1	0	L	L
q_7	1	0	q_7	1	0	L	L
q_7	1	1	q_7	1	1	L	L
q_8	**b**	**b**	q_9	**b**	**b**	L	L
q_8	sonst		q_8	sonst		R	R
q_9	c	d	q_{10}	c	d	L	0

Erläuterung: Im Zustand q_8 werden beide Lese-Schreibköpfe wieder auf das letzte Zeichen der beiden Zahlen z und s gesetzt. Da beide Köpfe gleichmäßig bei der Addition nach links bewegt wurden, kann man beide Köpfe gleichzeitig nach rechts laufen lassen, bis auf beiden Bändern das Blanksymbol gelesen wird. "sonst" bedeutet in der Tabelle: alle sonstigen Kombinationen ungleich **b b**.

Ab Zustand q_9 wird nun die Wertzuweisung z := z - 2 ausgeführt. Binär bedeutet dies: Ziehe eine 1 ab der vorletzten Stelle ab. Den zweiten Lese-Schreibkopf ignoriert man daher und setzt den ersten eine Position nach links (Zustand q_{10}).

q	a_1	a_2	q'	a'_1	a'_2	B_1	B_2
q_{10}	**b**	d	q_{10}	**b**	d	H	H
q_{10}	0	d	q_{10}	1	d	L	0
q_{10}	1	d	q_{11}	0	d	L	0
q_{11}	x	d	q_{11}	x	d	L	0
q_{11}	**b**	d	q_{12}	**b**	d	R	0
q_{12}	0	d	q_{12}	**b**	d	R	0
q_{12}	**b**	d	q_{14}	**b**	d	0	0
q_{12}	1	d	q_{13}	1	d	R	0
q_{13}	x	d	q_{13}	x	d	R	0
q_{13}	**b**	d	q_3	**b**	d	L	0
q_{14}	**b**	x	q_{14}	x	x	L	L
q_{14}	**b**	**b**	q_{15}	**b**	**b**	R	R

Erläuterung: Im Zustand q_{10} wird nun 1 subtrahiert, indem man nach links gehend jede 0 in eine 1 umwandelt, bis man auf ein Blanksymbol oder auf eine 1 trifft. Trifft man auf 1, so ersetze man sie durch 0 und beende die Subtraktion. Trifft man auf das Blanksymbol, dann muss ein Fehler vorgelegen haben, da z > 1 zu Beginn des Schleifenrumpfs war (wir fügen eine Halt-Zeile hierfür in δ_M ein, die unter normalen Bedingungen nicht erreicht werden kann).
Nun können aber führende Nullen entstanden sein, die die Abfrage "z>1" verfälschen würden. Also müssen erst die führenden Nullen beseitigt werden, indem man nach links zum Anfang der Zahl z geht und dann nach rechts laufend führende Nullen löscht.

Restliche Erläuterung: Damit ist das Ende des Schleifenrumpfs erreicht, und sobald man im Zustand q_{13} das rechte Ende von z gefunden hat, muss man die Schleife neu starten im Zustand q_3.
Trifft die Abfrage "z>1" nicht zu, so wird die Ausgabe vorbereitet. Hierzu muss die Zahl s auf das erste Band kopiert werden. Beachte, dass in diesem Fall das erste Band bereits (Zustände q_4 und q_5) gelöscht wurde. Man muss also im Zustand q_{14} nur von links nach rechts jedes Zeichen vom zweiten auf das erste Band schreiben, bis das Blanksymbol gelesen wird.
Damit endet der Algorithmus.

Überzeugen Sie sich durch penibles Nachvollziehen, dass diese Turingmaschine genau den vorgegebenen Algorithmus realisiert! Zugleich erkennen Sie, dass man mit Turingmaschinen while-Schleifen und beliebige Wertzuweisungen nachvollziehen kann. Auch die Alternative und andere Anweisungen lassen sich in die "Sprache der Turingmaschinen" übersetzen.

■

Die hier vorgestellte Übersetzungsmethode lässt sich automatisieren, so dass man einen Compiler angeben kann, der jedes Ada-Programm in eine gleichwertige Turingmaschine umwandelt. (Siehe Veranstaltungen zum Compilerbau.)

Die Laufzeit der Turingmaschine ist gleich der Laufzeit des Ada-Programms, aber mit den Erschwernissen, dass alle Operationen zeichenweise durchzuführen sind und dass kein Datenzugriff über Pointer oder Indizes erfolgt, sondern dass die Daten hintereinander auf den Bändern abgelegt sind und daher das Band sequentiell bis zu der Stelle, an der die gesuchten Daten stehen, durchlaufen werden muss.

Wir fassen diese Aussagen zusammen in drei Behauptungen:

Behauptung 1.6.2.3:

Jeder Algorithmus lässt sich durch eine Turingmaschine realisieren

oder anders ausgedrückt:

Die formale Definition des intuitiven Begriffs "Algorithmus" ist die Turingmaschine.

Dies ist die berühmte Churchsche These aus dem Jahre 1936 (nach Alonzo Church, 1903-1995).

Diese Behauptung lässt sich nicht beweisen, da der Begriff "Algorithmus" nur umgangssprachlich festgelegt ist. Man hat den Begriff "Algorithmus" aber auf viele verschiedene Arten definiert und alle erwiesen sich als gleichwertig zur Turingmaschine. Daher ist man von der Gültigkeit der Churchschen These heute überzeugt.

Was wir aber beweisen können, ist die

Behauptung 1.6.2.4:

(1) Zu jedem Ada-Programm π gibt es eine Turingmaschine T_π, deren Resultatsfunktion gleich der von dem Ada-Programm realisierten Abbildung ist (vgl. Abschnitt 1.1.3).

(2) Man kann eine berechenbare Funktion c konstruieren, die jedem Ada-Programm π eine solche Turingmaschine $c(\pi) = T_\pi$ zuordnet. (c heißt "Compiler".)

Der Beweis dieser Aussage ergibt sich daraus, dass man einen Compiler für Ada-Programme in eine Maschinensprache angibt und die Maschinensprache durch eine Turingmaschine simuliert, siehe Vorlesungen über Compilerbau. [Hinweis: Es gibt in Ada nichtdeterministische Sprachelemente; in diesem Fall liefert c eine nichtdeterministische Turingmaschine.]

Was wir weiterhin beweisen können, ist die

Behauptung 1.6.2.5:

Es gibt einen Compiler c mit folgender Eigenschaft:
Wenn das Ada-Programm π für die Eingabe w anhält und hierfür S(w) Speicherplätze belegt und T(w) Zeiteinheiten benötigt, so besitzt die Turingmaschine $c(\pi)$ einen Platzbedarf O(S(w)) und benötigt höchstens $O(S(w) \cdot T(w))$ Schritte.

Diese Behauptung folgt aus dem Beweis (den wir hier nicht führen können) zur Behauptung 1.6.2.4 durch genaue Analyse, wie der Speicher und wie die Anweisungen durch die Maschinensprache eines Computers und danach durch eine Turingmaschine simuliert werden.

1.6.2.6: Schlussfolgerung aus diesen Überlegungen:

1. Ada-Programme und Turingmaschinen sind zwei gleichwertige Konzepte, um Algorithmen zu realisieren.

2. Jedes Verfahren, das im intuitiven Sinn ein Algorithmus ist, lässt sich durch eine Turingmaschine oder ein Ada-Programm realisieren.

3. Wenn wir Aussagen über Algorithmen machen wollen, so genügt es, Turingmaschinen zu betrachten (diese sind präzise definiert und daher formalen Untersuchungen zugänglich).

4. Wir können Komplexitätsklassen auf Programmen und auf Turingmaschinen einführen und damit Algorithmen genauer klassifizieren.

1.6.2.7 Aufgabe: Bitte vergleichen Sie selbst im Lichte der vorgestellten Zusammenhänge die Definition 1.6.1.5

mit der Definition 1.1.3.5

$\wp$ = {f | Es gibt ein Programm π mit f = f_π: **D*** → **D***}
ist die Menge der von Programmen berechenbaren Funktionen

mit der Definition 1.1.3.6:

$\Re$ = {f | Es gibt ein Programm π, das immer terminiert, mit f = f_π}
ist die Menge der von Programmen total berechenbaren Funktionen oder die Menge der total rekursiven Funktionen

mit der Anmerkung 1.2.2.3: entscheidbare Menge

mit der bisherigen Definition aufzählbar (siehe Einschub hinter 1.1.3.3) und

mit dem Halteproblem 1.2.2.1 (Dieses besagt nun für Turingmaschinen: Finde einen Algorithmus, der zu jeder TM und zu jedem Wort w entscheidet, ob w im Haltebereich von der TM liegt oder nicht.)

1.6.2.7 Aufgabe (Fortsetzung):

Zu Beginn von Abschnitt 1.2.1 wurden 8 Kriterien a) bis g) aufgelistet, die an einen Algorithmus zu stellen sind. Prüfen Sie nach, dass die Turingmaschine diese Kriterien erfüllt und hier sogar eine gewisse Minimalität besitzt, z.B. "höchstens eine" statt endlich vieler Fortsetzungsmöglichkeiten.

Machen Sie sich weiterhin klar, dass nichtdeterministische Turingmaschinen von deterministischen Turingmaschinen simuliert werden können, im Prinzip genau mit dem Algorithmus in 1.3.3.2, der alle Ableitungen einer Grammatik mit Hilfe einer Warteschlange systematisch durchläuft. Der Zeitaufwand wächst hierbei (höchstens) exponentiell, weshalb man nichtdeterministische Turingmaschinen in der Praxis zu vermeiden sucht.

1.6.3 Komplexitätsklassen

Wir wollen nun Probleme, Algorithmen, Funktionen, Turingmaschinen oder Programme danach klassifizieren, wieviel Ressourcen sie für ihre Lösung oder Ausführung benötigen. Ressourcen sind vor allem Zeit und Platz.

Rückblick: Zu Beginn von 1.2.3 hatten wir die Zeitkomplexität bereits formuliert als "Rechendauer". Dort hieß es:

> Sie ist abhängig von der (Länge der) Eingabe. In der Regel definiert man den Zeitaufwand $t_\pi: \mathbb{N}_0 \to \mathbb{N}_0$ als Funktion der Länge der Eingabe des Programms π:
> $t_\pi(n) =$ maximale Zeit, die das Programm π für irgendeine Eingabe w der Länge n benötigt.

Wie misst man die Zeit? Die Geschwindigkeit auf einem Computer ist ungeeignet, da sich diese mit jeder neuen Technik ändert. Wir werden die Zeit daher in "Schritten" messen. Bei Turingmaschinen ist dies "ganz einfach", da bei diesen formalen Modell genau definiert ist, was ein Schritt ist, nämlich ein einmaliges Anwenden der Übergangsrelation bzw. Übergangsfunktion δ. Bei Programmen zählen wir die elementaren Anweisungen und Ausdrücke, die man bis zum Anhalten des Programms ausführen oder auswerten muss; diese grobe Näherung wird uns zunächst reichen.

Für Komplexitätsuntersuchungen spielen nur Berechnungen, die anhalten, eine Rolle. Daher gehen wir von Algorithmen und Maschinen aus, die für alle Eingaben nach endlich vielen Schritten anhalten. Wir beginnen mit Turingmaschinen.

1.6.3.1: Gegeben sei eine k-Band-Turingmaschine
$M = (Q, \Sigma, \Gamma, \delta, q_0, F, \textbf{b}, k)$, die für alle Eingaben anhalten möge, d.h., der Haltebereich von M sei Σ^*. Die Komplexität für die Eingabe w ist die minimale Anzahl der Schritte $t_M(w)$ oder die minimal erforderliche Zahl an Speicherplätzen $s_M(w)$, die M für die Eingabe von w benötigt, um seine Berechnung bis zum Anhalten durchzuführen.
(t kommt von "time complexity" und s von "space complexity".)

$t_M(w)$ = Min { i | Es gibt eine Berechnung, die bei Eingabe von w auf das erste Band von M genau i mal die Übergangsrelation δ verwendet und dann anhält}.

$s_M(w)$ = Min { i | Es gibt eine Berechnung, die bei Eingabe von w auf das erste Band von M anhält und hierbei genau j verschiedene Bandfelder besucht hat}.

In dieser Definition wurde das Minimum gebildet. Der Grund liegt darin, dass die Turingmaschinen nichtdeterministisch sein dürfen und daher mehrere Berechnungen zu lassen.

In der Praxis werden letztlich deterministische Modelle benutzt. Wir werden daher bei der Definition der Komplexitätsklassen zwischen deterministischen und nichtdeterministischen Turingmaschinen unterscheiden.

Man beachte auch, dass t_M und s_M nur im deterministischen Fall in der gleichen Berechnungsfolge ermittelt werden. Im nichtdeterministischen Fall kann eine Berechnungsfolge möglichst günstig für die Zeit und eine andere möglichst günstig für den Platzbedarf sein.

1.6.3.2: Für die Praxis ist die Definition zu speziell, da sie sich auf *jede einzelne* Eingabe bezieht. Man fasst daher alle Eingaben gleicher Länge zusammen und verwendet als Komplexität in der Regel den schlechtesten Fall ("worst case" complexity), manchmal den durchschnittlichen Fall ("average case") oder selten auch den besten Fall ("best case"). Daher definiert man die **Komplexität** wie folgt:

$t_M(n) = \text{Max}\ \{t_M(w) \mid \text{die Länge von w ist n}\}$,
$s_M(n) = \text{Max}\ \{s_M(w) \mid \text{die Länge von w ist n}\}$.

$$t_M^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} t_M(w), \quad s_M^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} s_M(w).$$

$t_M^{best}(n) = \text{Min}\ \{t_M(w) \mid \text{die Länge von w ist n}\}$,
$s_M^{best}(n) = \text{Min}\ \{s_M(w) \mid \text{die Länge von w ist n}\}$.

Im Folgenden benötigen wir noch die semicharakteristische Funktion, da bei nichtdeterministischen Maschinen bereits der Erfolgsfall ausreicht, um einen Menge zu beschreiben (während das Komplement meist nur durch Durchtesten aller Möglichkeiten erkannt werden kann).

Definition 1.6.3.3:

Für jedes $L \subseteq \Sigma^*$ heißt die partielle Abbildung

$$\psi_L : \Sigma^* \to \{1\} \text{ mit}$$

$\psi_L(w) =$ if $w \in L$ then 1 else undefiniert fi

die semi-charakteristische Funktion von L.

1.6.3.4: Hieraus lassen sich nun die **Komplexitätsklassen** für das Berechnungsmodell Turingmaschinen präzise definieren. Es seien T, S: $\mathbb{N}_0 \to \mathbb{N}_0$ zwei gegebene Funktionen, dann setze:

$\text{DTimeSpace}^{TM}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine deterministische Turingmaschine M, die χ_L berechnet, mit $t_M(n) \in O(T(n))$ und $s_M(n) \in O(S(n)).\}$

$\text{NTimeSpace}^{TM}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine nichtdeterministische Turingmaschine M, die ψ_L berechnet, mit $t_M(n) \in O(T(n))$ und $s_M(n) \in O(S(n)).\}$

1.6.3.4 (Fortsetzung)

$\text{DTime}^{TM}(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine deterministische k-Band-Turingmaschine M, die χ_L berechnet, mit $t_M(n) \in O(T(n)).\}$

$\text{NTime}^{TM}(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine nichtdeterministische k-Band-Turingmaschine M, die ψ_L berechnet, mit $t_M(n) \in O(T(n)).\}$

$\text{DSpace}^{TM}(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine deterministische k-Band-Turingmaschine M, die χ_L berechnet, mit $s_M(n) \in O(S(n)).\}$

$\text{NSpace}^{TM}(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt eine nichtdeterministische k-Band-Turingmaschine M, die ψ_L berechnet, mit $s_M(n) \in O(S(n)).\}$

Statt Sprachklassen kann man auf die gleiche Art auch Funktionsklassen definieren. Diese bezeichnet man ebenfalls mit DTime™(T) usw.

Bei nichtdeterministischen Berechnungen kann man sich auch andere Definitionen vorstellen. Wir verlangen hier jedoch nur, dass ψ_L berechnet wird, also für eine Eingabe w nur im Falle, dass $w \in L$ ist, eine erfolgreich anhaltende Berechnung existiert.

Weiterhin kümmern wir uns generell nicht um Konstanten, sondern nur um die Abhängigkeit von der Länge der Eingabe. Dies kann man theoretisch untermauern (siehe spätere Theorie-Vorlesungen), aber auch dadurch begründen, dass durch schnellere Computer jede einmal ermittelte Konstante hinfällig wird.

1.6.3.5. Einfache Folgerungen:

DTime™ (T) $\subseteq$ NTime™ (T) und
DSpace™ (S) $\subseteq$ NSpace™ (S)
da deterministische Turingmaschinen ein Spezialfall der nichtdeterministischen sind.
DTime™ (T) $\subseteq$ DSpace™ (T) und
NTime™ (T) $\subseteq$ NSpace™ (T)
da man, um T(n) Felder zu besuchen, mindestens T(n) Schritte gemacht haben muss.

Hinweis: Wir vertiefen nicht weiter. Für eine präzise Bestimmung der Komplexität benötigt man formal exakt definierte Modelle, so dass die Turingmaschinen-Komplexitätsklassen eine wichtige Rolle bei der Definition und bei vertieften Untersuchungen spielt. Wir wenden uns nun den Algorithmen zu.

1.6.3.6: Wir betrachten nur Algorithmen und Programme, die für alle Eingaben anhalten. Die Komplexität für die Eingabe w ist die Anzahl der Schritte $t_A(w)$ oder die Zahl an Speicherplätzen $s_A(w)$, die ein Programm oder ein Algorithmus A für die Eingabe von w benötigt, um seine Berechnung bis zum Anhalten durchzuführen.

Genauer: Sei A ein Algorithmus (oder Programm). Eine *Berechnung* für die Eingabe w ist eine Folge von elementaren Anweisungen und Ausdrücken. Hiermit definieren wir:

$t_A(w)$ = Anzahl der elementaren Anweisungen und Ausdrücke, die A in seiner Berechnung bei Eingabe von w durchläuft, bis er anhält.

$s_A(w)$ = Anzahl der Speicherplätze, auf die A während seiner Berechnung bei Eingabe von w zugreift, bis er anhält.

1.6.3.7: Nun geht man wie bei Turingmaschinen weiter vor: Man fasst die Eingaben gleicher Länge zusammen und verwendet als Komplexität in der Regel den schlechtesten Fall ("worst case" complexity), manchmal den durchschnittlichen Fall ("average case") oder selten den besten Fall ("best case"). Daher definiert man die **Zeit- und Platz-Komplexität** für Programme/Algorithmen A wie folgt:

$t_A(n) = \text{Max}\ \{t_A(w) \mid \text{die Länge von w ist n}\}$,
$s_A(n) = \text{Max}\ \{s_A(w) \mid \text{die Länge von w ist n}\}$.

$$t_A^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} t_A(w), \qquad s_A^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} s_A(w).$$

$t_A^{best}(n) = \text{Min}\ \{t_A(w) \mid \text{die Länge von w ist n}\}$,
$s_A^{best}(n) = \text{Min}\ \{s_A(w) \mid \text{die Länge von w ist n}\}$.

1.6.3.8: Hieraus lassen sich nun die **Komplexitätsklassen** für Algorithmen definieren. Es seien T, S: $\mathbb{N}_0 \to \mathbb{N}_0$ zwei gegebene Funktionen, dann setze:

$\text{DTimeSpace}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $t_A(n) \in O(T(n))$ und $s_A(n) \in O(S(n)).\}$

$\text{NTimeSpace}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $t_A(n) \in O(T(n))$ und $s_A(n) \in O(S(n)).\}$

1.6.3.8 (Fortsetzung)

$\text{DTime}(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $t_A(n) \in O(T(n)).\}$

$\text{NTime}(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $t_A(n) \in O(T(n)).\}$

$\text{DSpace}(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $s_A(n) \in O(S(n)).\}$

$\text{NSpace}(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $s_A(n) \in O(S(n)).\}$

Statt Sprachklassen kann man auf die gleiche Art auch Funktionsklassen definieren. Diese bezeichnet man ebenfalls mit DTime (T), NTime (T), DSpace (S) und NSpace (S).

Nichtdeterministische Berechnungen haben wir bei Algorithmen noch nicht kennen gelernt. Sie können sich aber sicher vorstellen, dass solche Berechnungen möglich sind, sofern man nichtdeterministische Sprachelemente in die Programmiersprache einführt. Ein Standardbeispiel sind die "guarded commands", bei denen mehrere Bedingungen zutreffen können, wobei dann eine zutreffende Bedingung willkürlich ausgewählt und ausgeführt wird. (Siehe später.)

Auch hier kümmern wir uns generell nicht um Konstanten, sondern nur um die Abhängigkeit von der Länge der Eingabe.

1.6.3.9. Einfache Folgerungen (Nachweis wie bei 1.6.3.5):

DTime (T) $\subseteq$ NTime (T), DSpace (S) $\subseteq$ NSpace (S).

DTime (T) $\subseteq$ DSpace (T), NTime (T) $\subseteq$ NSpace (T).

Unterschied zu den TM-Komplexitätsklassen:

a) Die Auswertung eines Ausdrucks kostet bei Algorithmen nur einen Schritt.
b) Die Ausführung einer Elementaranweisung kostet nur einen Schritt.
c) Der Zugriff auf eine (indizierte) Variable erfolgt in einem Schritt.

Man nennt die Komplexität bei Algorithmen und Programmen daher das "uniforme Komplexitätsmaß".

Aus der Behauptung 1.6.2.5 und anderen Überlegungen erhält man Zusammenhänge zwischen den verschiedenen Maßen (ohne Beweis):

DTimeSpace (T,S) $\subseteq$ DTimeSpace™ (T·S, S)

DTimeSpace™ (T, S) $\subseteq$ DTimeSpace (T·log(S), S)

Die Komplexitätsklassen sind daher recht ähnlich. Statt mit den üblichen Klassen der Turingmaschinen zu arbeiten, können wir daher die anschaulicheren Klassen für Algorithmen verwenden.

Zwei spezielle Komplexitätsklassen:

DTime (n) = DTime (id) mit id(n) = n für alle n$\in \mathbb{N}_0$.

Für ein Polynom p(n) vom Grad k ist:

DTime (n^k) = DTime (p(n)), klar wegen der O-Klassen.

$$\mathbf{P} = \bigcup_{k\geq 0} \text{DTime } (n^k) \qquad \mathbf{NP} = \bigcup_{k\geq 0} \text{NTime } (n^k)$$

Berühmtestes Problem derzeit: Ist **P** = **NP** oder nicht?

(Man vermutet **P** ≠ **NP**.)