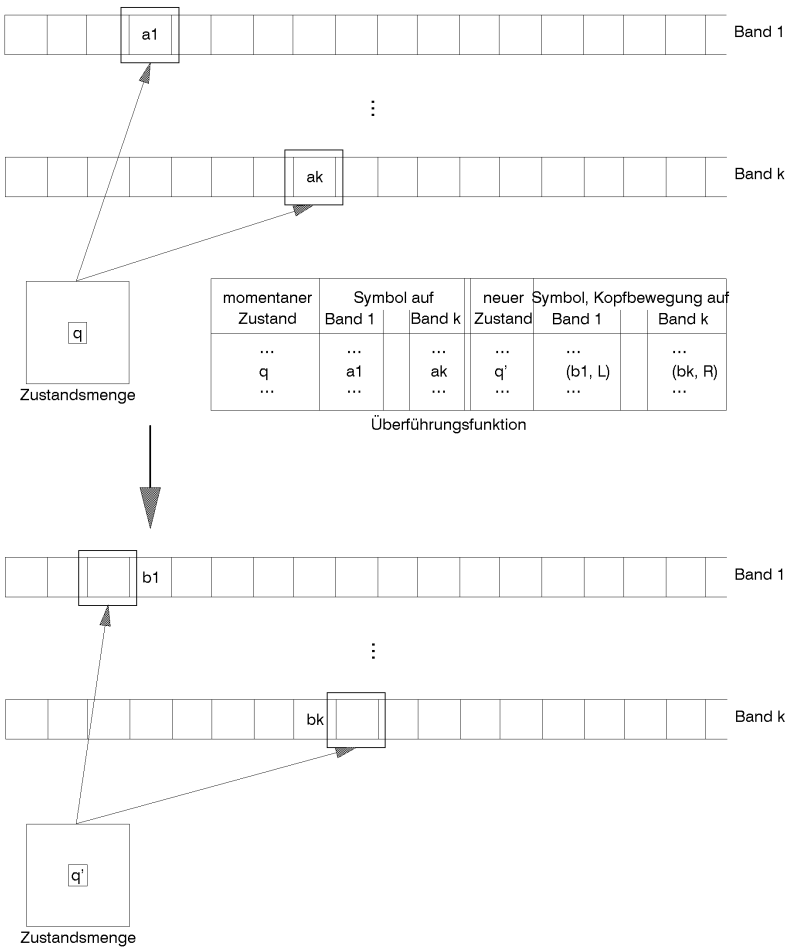


Ist $d(q,a_1,...,a_k)$ nicht definiert, so **hält** die *k*-DTM **im Zustand** q **an** (**stoppt im Zustand** q). Sobald also der akzeptierende Zustand q_{accept} erreicht wird, hält die Turingmaschine an, da $d(q,a_1,...,a_k)$ für $q = q_{accept}$ nicht definiert ist. Sie kann aber auch eventuell vorher in einem anderen Zustand anhalten (nämlich dann, wenn $d(q,a_1,...,a_k)$ nicht definiert ist), oder sie **kann beliebig lange weiterlaufen (sie hält nicht an)**. Diese Situation tritt beispielsweise dann ein, wenn die Turingmaschine in einen Zustand $q \neq q_{accept}$ kommt und $d(q,a_1,...,a_k) = (q,(a_1,S),...,(a_k,S))$ ist. Eine andere Möglichkeit eines endlosen Weiterlaufens ergibt sich dann, wenn die Turingmaschine in einen Zustand $q \neq q_{accept}$ kommt, alle Köpfe über Zellen stehen, die das Leerzeichen b enthalten, rechts dieser Zellen auf allen Bändern nur noch Leerzeichen stehen und $d(q,b,...,b) = (q,(b,R),...,(b,R))$ ist.

Die Werte $d(q,a_1,...,a_k) = (q',(b_1,d_1),...,(b_k,d_k))$ der Überführungsfunktion werden häufig in Form einer endlichen Tabelle angegeben:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf			neuer Zustand	neues Symbol, Kopfbewegung auf		
	Band 1	...	Band *k*		Band 1	...	Band *k*
q	a_1	...	a_k	q'	b_1, d_1	...	b_k, d_k

Eine **Konfiguration** *K* einer *k*-DTM *TM* beschreibt den gegenwärtigen Gesamtzustand von *TM*, d.h. den gegenwärtigen Zustand zusammen mit den Bandinhalten und den Positionen der Schreib/Leseköpfe:
$K = (q,(a_1,i_1),...,(a_k,i_k))$ mit $q \in Q$, $a_j \in \Sigma^*$, $i_j \geq 1$ für j = 1, ..., *k*.

Diese Konfiguration *K* wird folgendermaßen interpretiert:

TM ist im Zustand *q*, das *j*-te Band (für j = 1, ..., *k*) enthält linksbündig die endliche Zeichenkette a_j (jeder Buchstabe von a_j belegt eine Zelle), gefolgt von Zellen, die das Leerzeichen enthalten (leere Zellen), der Schreib/Lesekopf des *j*-ten Bands steht über der i_j-ten Zelle; für $i_j \in [1:|a_j|]$ ist dieses der i_j-teBuchstaben von a_j, für $i_j \geq |a_j|+1$ steht der Schreib/Lesekopf hinter a_j über einer Zelle, die das Leerzeichen enthält.

Ist a_j der i_j-te Buchstabe von a_j bzw. $a_j = b$ für $i_j \geq |a_j|+1$, und ist
$d(q,a_1,...,a_k) = (q',(b_1,d_1),...,(b_k,d_k))$,
dann geht die *TM* in die **Folgekonfiguration** K' über, die durch
$K' = (q',(b_1,i_1'),...,(b_k,i_k'))$
definiert wird. Dabei entsteht b_j aus a_j durch Ersetzen von a_j durch b_j. Die Positionen i_j' der Schreib/Leseköpfe der Folgekonfiguration K' lauten

$$i_j' = \begin{cases} i_j + 1 & \text{für } d_j = R \\ i_j & \text{für } d_j = S \\ i_j - 1 & \text{für } d_j = L \text{ und } i_j \geq 2. \end{cases}$$

Man schreibt in diesem Fall
$K \Rightarrow K'$.
Die Bezeichnung $K \Rightarrow^* K'$ besagt, daß entweder keine Konfigurationsänderung stattgefunden hat (es ist dann $K = K'$) oder daß es eine Konfiguration K_1 gibt mit $K \Rightarrow K_1$ und $K_1 \Rightarrow^* K'$ (auch geschrieben als $K \Rightarrow K_1 \Rightarrow^* K'$).

Man schreibt $K \Rightarrow^m K'$ mit $m \in \mathbf{N}$, wenn K' aus K durch m Konfigurationsänderungen hervorgegangen ist, d.h. wenn es m Konfigurationen K_1, ..., K_m gibt mit

$K \Rightarrow K_1 \Rightarrow ... \Rightarrow K_m = K'$.

Für $m = 0$ ist dabei $K = K'$.

Eine Konfiguration K_0, die den Anfangszustand q_0 und auf dem 1. Band ein Wort $w \in I^*$ enthält, wobei sich auf allen anderen Bändern nur Leerzeichen befinden und die Köpfe über den am weitesten links stehenden Zellen stehen, d.h. eine Konfiguration der Form

$K_0 = (q_0, (w,1), (e,1), ..., (e,1))$,

heißt **Anfangskonfiguration mit Eingabewort** *w*. Da $w \in I^*$ ist und das Leerzeichen nicht zu *I* gehört, kann *TM* (bei entsprechender Definition der Überführungsfunktion) das Ende von *w*, nämlich das erste Leerzeichen im Anschluß an *w*, erkennen. Im folgenden wird gelegentlich für eine Eingabe $w \in I^*$ auch $w \in \Sigma^*$ geschrieben und dabei implizit angenommen, daß *w* nur Buchstaben aus *I* enthält.

Eine Konfiguration K_{accept}, die den akzeptierenden Zustand q_{accept} enthält, d.h. eine Konfiguration der Form

$K_{accept} = (q_{accept}, (a_1, i_1), ..., (a_k, i_k))$,

heißt **akzeptierende Konfiguration** (**Endkonfiguration)**.

Ein Wort *w* über dem Eingabealphabet wird von *TM* **akzeptiert**, wenn gilt:

$(q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^* K_{accept}$

mit einer Endkonfiguration K_{accept}.

Die von einer *k*-DTM *TM* **akzeptierte Sprache** ist die Menge

$$L(TM) = \{ w \mid w \in I^* \text{ und } w \text{ wird von } TM \text{ akzeptiert} \}$$
$$= \{ w \mid w \in I^* \text{ und } (q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^* (q_{accept}, (a_1, i_1), ..., (a_k, i_k)) \}.$$

Zu beachten ist, daß *TM* eventuell auch dann bereits eine Endkonfiguration erreicht, wenn das Eingabewort *w* noch gar nicht komplett gelesen ist. Auch in diesem Fall gehört *w* zu *L*(*TM*).

Für $w \notin L(TM)$ hält *TM* entweder nicht im Zustand q_{accept}, oder *TM* läuft unendlich lange weiter, d.h. die Überführungsfolge $K_0 \Rightarrow K'$ läßt sich unendlich lang fortsetzen, ohne daß eine Konfiguration erreicht wird, die den Endzustand enthält.

Eine 2-DTM Turingmaschine zur Akzeptanz von

$L(TM) = \{ w \mid w \in \{0,1\}^+$ und w ist die Binärdarstellung einer geraden Zahl $\} \cup \{e\}$

Die 2-DTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ wird gegeben durch
$Q = \{q_0, q_1, q_2\},\ \Sigma = \{0, 1, b\},\ I = \{0, 1\},\ q_{accept} = q_1$ mit der Überführungsfunktion d, die durch folgende Tabelle definiert ist:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf		neuer Zustand	neues Symbol, Kopfbewegung auf	
	Band 1	Band 2		Band 1	Band 2
q_0	b	b	q_1	b, S	1, S
	0	b	q_0	0, R	b, S
	1	b	q_2	1, R	b, S
q_2	b	b	q_2	b, R	b, S
	0	b	q_0	0, R	b, S
	1	b	q_2	1, R	b, S

TM stoppt bei Eingabe von *w* nicht, falls *w* die Binärdarstellung einer ungeraden Zahl ist.

Eine 2-DTM Turingmaschine zur Akzeptanz der Palindrome über { 0, 1}

Die folgende Turingmaschine 2-DTM *TM* akzeptiert genau die Palindrome über {0, 1}:
$L(TM) = \{ w \mid w \in \{0,1\}^+$ und $w = a_1 ... a_{n-1} a_n = a_n a_{n-1} ... a_1 \} \cup \{e\}$.

$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $Q = \{q_0, ..., q_5\},\ \Sigma = \{0, 1, b, \#\},\ I = \{0, 1\}$, $q_{accept} = q_5$.

TM arbeitet wie folgt:

1. Die 1. Zelle des 2. Bandes wird mit # markiert. Dann wird das Eingabewort auf das 2. Band kopiert; der Kopf des 1. Bandes steht jetzt unmittelbar rechts des Eingabeworts.
2. Der Kopf des 2. Bandes wird bis zum Zeichen # zurückgesetzt.
3. Der Kopf des 1. Bandes wird jeweils um 1 Zelle nach links und der Kopf des 2. Bandes um 1 Zelle nach rechts verschoben. Wenn die von den Köpfen jeweils gelesenen Symbole sämtlich übereinstimmen, ist das Eingabenwort ein Palindrom, und die Maschine geht in den Zustand $q_f = q_5$ und stoppt. Sonst stoppt die Maschine in einem von q_f verschiedenen Zustand.

Die Überführungsfunktion wird durch folgende Tabelle gegeben:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf		neuer Zustand	neues Symbol, Kopfbewegung auf	
	Band 1	Band 2		Band 1	Band 2
q_0	0	b	q_1	0, S	#, R
	1	b	q_1	1, S	#, R
	b	b	q_{accept}	b, S	b, S
q_1	0	b	q_1	0, R	0, R
	1	b	q_1	1, R	1, R
	b	b	q_2	b, S	b, L
q_2	b	0	q_2	b, S	0, L
	b	1	q_2	b, S	1, L
	b	#	q_3	b, L	#, R
q_3	0	0	q_4	0, S	0, R
	1	1	q_4	1, S	1, R
q_4	0	0	q_3	0, L	0, S
	0	1	q_3	0, L	1, S
	1	0	q_3	1, L	0, S
	1	1	q_3	1, L	1, S
	0	b	q_{accept}	0, S	b, S
	1	b	q_{accept}	1, S	b, S

Beobachtet man die Arbeitsweise einer Turingmaschine *TM* bei einem Eingabewort *w*, so liegt nach endlich vielen Überführungen eine der folgenden Situationen vor:

1. Fall: *TM* ist bereits in einem Zustand $q \neq q_{accept}$ stehengeblieben (d.h. $d(q,...)$ ist nicht definiert). Dann ist $w \notin L(TM)$.

2. Fall: *TM* ist bereits im Zustand q_{accept} stehengeblieben. Dann ist $w \in L(TM)$.

3. Fall: *TM* ist noch nicht stehengeblieben, d.h. *TM* befindet sich in einem Zustand $q \neq q_{accept}$, für den $d(q,...)$ definiert ist. Dann ist noch nicht entschieden, ob $w \in L(TM)$ oder $w \notin L(TM)$ gilt. *TM* ist eventuell noch nicht lange genug beobachtet worden. Es ist nicht „vorhersagbar", wie lange *TM* beobachtet werden muß, um eine Entscheidung zu treffen (es ist **algorithmisch unentscheidbar**).

Eine Turingmaschine *TM* kann als **Berechnungsvorschrift** definiert werden: Das 1. Band wird als **Eingabeband** und ein Band, etwa das *k*-te Band, als **Ausgabeband** ausgezeichnet. Die Turingmaschine *TM* **berechnet eine partielle Funktion** $f_{TM} : I^* \to \Sigma^*$, wenn gilt:

Startet *TM* im Anfangszustand mit *w*, d.h. startet *TM* mit *w* auf dem Eingabeband im Zustand q_0 (alle anderen Bänder sind leer), und stoppt *TM* nach endlich vielen Schritten im akzeptierenden Zustand q_{accept}, dann wird die Bandinschrift *y* des Ausgabebands von *TM* als Funktionswert $y = f_{TM}(w)$ interpretiert. Falls *TM* überhaupt nicht oder nicht im Endzustand stehenbleibt, dann ist $f_{TM}(w)$ nicht definiert. Daher ist f_{TM} eine partielle Funktion.

Eine 2-DTM zur Berechnung der (totalen) Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & n+1 \end{cases}$

Die folgende 2-DTM berechnet die (totale) Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & n+1 \end{cases}$:

Hierbei wird ein $n \in \mathbf{N}$ als Zeichenkette in seiner Binärdarstellung auf das Eingabeband geschrieben; die höchstwertige Stelle steht dabei ganz links. Der Wert 0 wird als Zeichenkette 0 eingegeben, alle anderen Werte $n \in \mathbf{N}$ ohne führende Nullen. Entsprechend steht abschließend *n* + 1 in seiner Binärdarstellung ohne führende Nullen auf dem Ausgabeband (2. Band).

$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $Q = \{q_0, ..., q_{19}\}$, $\Sigma = \{0, 1, b, \#\}$, $I = \{0, 1\}$, $q_{accept} = q_{19}$.

TM arbeitet wie folgt:

1. Auf das 2. Band wird zunächst das Zeichen # geschrieben, das das linke Ende des 2. Bandes markieren soll. Dann wird auf das 2. Band eine 0 geschrieben und das Eingabewort *w* auf das 2. Band kopiert (auf dem 2. Band steht jetzt #0*w*, d.h. das Eingabewort *w* ist durch eine führende 0 ergänzt worden).
2. Auf dem 2. Band werden von rechts alle 1‘en in 0‘en invertiert, bis die erste 0 erreicht ist; diese wird durch 1 ersetzt (Addition $n := n + 1$).
3. Der Kopf des 2. Bandes wird auf die Anfangsmarkierung # zurückgesetzt und geprüft, ob rechts dieses Zeichens eine 0 steht (das bedeutet, daß die anfangs an *w* angefügte führende 0 wieder entfernt werden muß).
4. In diesem Fall wird der Inhalt des 2. Bandes komplett um zwei Position nach links verschoben. Dadurch werden die führende 0 und das Zeichen # auf dem 2. Band entfernt. Es ist zu beachten, daß am rechten Ende des 2. Bandes 2 Leerzeichen gesetzt werden.
5. Andernfalls wird der Inhalt des 2. Bandes um 1 Position nach links verschoben und dadurch das Zeichen # entfernt.

Die Überführungsfunktion wird durch folgende Tabelle gegeben:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf		neuer Zustand	neues Symbol, Kopfbewegung auf		Bemerkung
	Band 1	Band 2		Band 1	Band 2	
q_0	0	*b*	q_1	0, *S*	#, *R*	linkes Ende für *n*+1
	1	*b*	q_1	1, *S*	#, *R*	markieren
q_1	0	*b*	q_2	0, *S*	0, *R*	führende 0 erzeugen
	1	*b*	q_2	1, *S*	0, *R*	
q_2	0	*b*	q_2	0, *R*	0, *R*	Inhalt des Eingabe-
	1	*b*	q_2	1, *R*	1, *R*	bandes auf 2. Band
	b	*b*	q_3	*b*, *S*	*b*, *L*	kopieren
q_3	*b*	0	q_4	*b*, *S*	1, *S*	1 addieren:
	b	1	q_3	*b*, *S*	0, *L*	anhängende 1‘en invertieren, erste 0 von rechts invertieren
q_4	*b*	0	q_4	*b*, *S*	0, *L*	auf # zurückgehen
	b	1	q_4	*b*, *S*	1, *L*	
	b	#	q_5	*b*, *S*	#, *R*	

../..

q_5	b	0	q_6	b, S	$0, R$	muß führende 0
	b	1	q_{15}	b, S	$1, S$	entfernt werden?
q_6	b	0	q_7	b, S	$0, L$	gelesenes Zeichen
	b	1	q_8	b, S	$1, L$	im Zustand merken
	b	b	q_{13}	b, S	b, L	
q_7	b	0	q_9	b, S	$0, L$	2. Kopf um 1 Positi-
	b	1	q_9	b, S	$1, L$	on nach links setzen
q_8	b	0	q_{10}	b, S	$0, L$	2. Kopf um 1 Positi-
	b	1	q_{10}	b, S	$1, L$	on nach links setzen
q_9	b	0	q_{11}	b, S	$0, R$	0 schreiben
	b	1	q_{11}	b, S	$0, R$	
	b	#	q_{11}	b, S	$0, R$	
q_{10}	b	0	q_{11}	b, S	$1, R$	1 schreiben
	b	1	q_{11}	b, S	$1, R$	
	b	#	q_{11}	b, S	$1, R$	
q_{11}	b	0	q_{12}	b, S	$0, R$	2. Kopf um 1 Positi-
	b	1	q_{12}	b, S	$1, R$	on nach rechts set- zen
q_{12}	b	0	q_6	b, S	$0, R$	2. Kopf 1 weitere
	b	1	q_6	b, S	$1, R$	Position nach rechts setzen
q_{13}	b	0	q_{14}	b, S	b, L	letztes Zeichen
	b	1	q_{14}	b, S	b, L	löschen und nach links gehen
q_{14}	b	0	$q_{19} = q_f$	b, S	b, S	letztes Zeichen
	b	1	$q_{19} = q_f$	b, S	b, S	löschen und STOP
q_{15}	b	0	q_{16}	b, S	$0, L$	gelesenes Zeichen
	b	1	q_{17}	b, S	$1, L$	im Zustand merken
	b	b	q_{14}	b, S	b, L	
q_{16}	b	0	q_{18}	b, S	$0, R$	0 schreiben
	b	1	q_{18}	b, S	$0, R$	
	b	#	q_{18}	b, S	$0, R$	

../..

q_{17}	b	0	q_{18}	b, S	1, R	1 schreiben
	b	1	q_{18}	b, S	1, R	
	b	#	q_{18}	b, S	1, R	
q_{18}	b	0	q_{15}	b, S	0, R	2. Kopf um 1 Posi-
	b	1	q_{15}	b, S	1, R	tion nach rechts set- zen

Die Konzepte der Berechnung partieller Funktionen und das Akzeptieren von Sprachen mittels Turingmaschinen sind äquivalent:

Satz 2.1-1:

Berechnet die Turingmaschine *TM* die partielle Funktion $f_{TM} : I^* \to \Sigma^*$, dann kann man eine Turingmaschine TM' konstruieren mit $L(TM') = \{ w\# y \mid y = f_{TM}(w) \}$.

Akzeptiert die Turingmaschine *TM* die Sprache $L(TM)$, dann läßt sich *TM* so modifizieren, daß sie die partielle Funktion $f_{TM} : I^* \to \Sigma^*$ berechnet, die definiert ist durch $f_{TM}(w) = y$ genau dann, wenn gilt:

$$(q_0,(w,1),(e,1),...,(e,1),(e,1)) \Rightarrow^* (q_{accept},(a_1,i_1),(a_2,i_2),...,(a_{k-1},i_{k-1})(y,|y|+1))$$

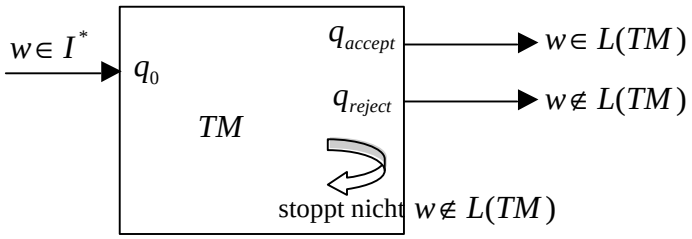
Die Aussage läßt die Interpretation zu, daß die Verifikation eines Funktionsergebnisses genauso komplex ist wie die Berechnung des Funktionsergebnisses selbst.

Die Überführungsfunktion d einer Turingmaschine *TM* werde folgendermaßen modifiziert: Die Zustandsmenge *Q* wird um einen neuen Zustand q_{reject} erweitert. Für alle bisherigen Zustände $q \in Q$ mit $q \neq q_{accept}$, für die $d(q,...)$ nicht definiert ist, werden in d die Zeilen

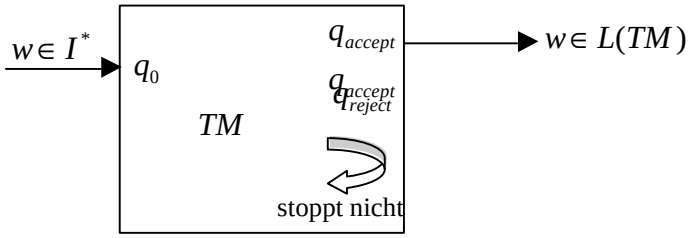
$d(q, a_1, ..., a_k) = (q_{reject}, (a_1, S), ..., (a_k, S))$ mit $a_i \in \Sigma$ für $i = 1, ..., k$

aufgenommen. Für q_{reject} ist d nicht definiert. Kommt die so modifizierte Turingmaschine bei Eingabe eines Wortes $w \in I^*$ in einen Zustand $q \in Q$ mit $q \neq q_{accept}$, für die $d(q,...)$ bisher nicht definiert war (es ist dann $w \notin L(TM)$), so macht die modifizierte Turingmaschine noch eine Überführung, ohne die Bandinhalte zu ändern, und stoppt im Zustand q_{reject}, ohne das Wort *w* zu akzeptieren. Man kann daher annehmen, daß eine Turingmaschine *TM* so definierte Zustände q_{accept} und q_{reject} enthält. Der Zustand q_{accept} heißt **akzeptierender Zustand**, der Zustand q_{reject} heißt **nicht-akzeptierender (verwerfender) Zustand**. Startet *TM* bei Ein-

gabe eines Wortes $w \in I^*$ im Anfangszustand q_0, so zeigt sie folgendes Verhalten: Entweder kommt *TM* in den Zustand q_{accept}, dann ist $w \in L(TM)$ und **die Eingabe** *w* **wird akzeptiert**; oder *TM* kommt in den Zustand q_{reject}, dann ist $w \notin L(TM)$ und **die Eingabe** *w* **wird verworfen**; oder *TM* läuft unendlich lange weiter, dann ist ebenfalls $w \notin L(TM)$. Man kann *TM* daher als Blackbox darstellen, die eine Eingangsschnittstelle besitzt, über die im Anfangszustand q_0 ein Wort $w \in I^*$ eingegeben wird und der Start der Berechnung gemäß der Überführungsfunktion d erfolgt. Sie besitzt zwei „aktivierbare" Ausgangsschnittstellen: Die erste Ausgangsschnittstelle wird von *TM* dann aktiviert, wenn *TM* bei der Berechnung den Zustand q_{accept} erreicht und stoppt; es ist dann $w \in L(TM)$. Die zweite Ausgangsschnittstelle wird von *TM* aktiviert, wenn *TM* bei der Berechnung den Zustand q_{reject} erreicht und stoppt; es ist dann $w \notin L(TM)$. Wird keine Ausgangsschnittstelle aktiviert, weil *TM* nicht anhält, dann ist ebenfalls $w \notin L(TM)$. Eine graphische Repräsentation von *TM* sieht wie folgt aus.



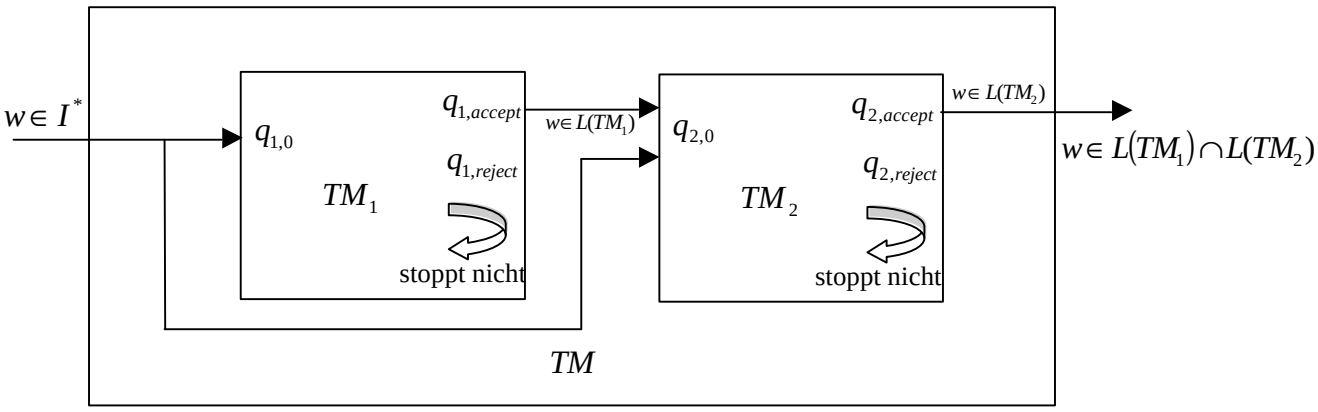
Da die Turingmaschine *TM* dazu verwendet wird, um festzustellen, ob ein Eingabewort $w \in I^*$ von ihr akzeptiert wird, kann man *TM* vereinfacht auch folgendermaßen darstellen:



Turingmaschinen können zu neuen Turingmaschinen zusammengesetzt werden: Es seien zwei Turingmaschinen $TM_1 = (Q_1, \Sigma_1, I, d_1, b, q_{1,0}, q_{1,accept})$ und $TM_2 = (Q_2, \Sigma_2, I, d_2, b, q_{2,0}, q_{2,accept})$ mit disjunkten Zustandsmengen ($Q_1 \cap Q_2 = \varnothing$) und demselben Eingabealphabet $I \subseteq \Sigma_1$ und $I \subseteq \Sigma_2$ und jeweils getrennten Bändern gegeben. Die Anzahl der Bänder von TM_1 sei k_1, die

von TM_2 sei k_2. Jeweils das 1. Band ist das Eingabeband. Beispiele für die Möglichkeiten der **Zusammensetzung** sind:

- Man kann eine neue Turingmaschine *TM* durch **Hintereinanderschaltung** von TM_1 und TM_2 konstruieren, die die Bänder von TM_1 und TM_2 umfaßt, d.h. $k_1 + k_2$ viele Bänder besitzt, und folgendermaßen arbeitet: Eine Eingabe $w \in I^*$ für die zusammengesetzte Turingmaschine *TM* wird auf das 1. Band von TM_1 gegeben und auf das 1. Band von TM_2 kopiert. Jetzt wird zunächst das Verhalten von TM_1 auf *w* simuliert. Falls TM_1 das Wort *w* akzeptiert, d.h. in den Zustand wird $q_{1,accept}$ gelangt, wird das Verhalten von TM_2 auf *w* simuliert. *TM* akzeptiert das Wort *w*, falls TM_2 das Wort *w* akzeptiert. Es gilt: $L(TM) = L(TM_1) \cap L(TM_2)$. Graphisch läßt sich *TM* wie folgt darstellen.



Die formale Notation von *TM* mit Hilfe der Überführungsfunktion ist etwas mühsam:

Es ist $TM = (Q, \Sigma, I, d, b, q_{1,0}, q_{2,accept})$. Dabei ist
$Q = Q_1 \cup Q_2 \cup \{q_{copy}, q_{skip}\}$ die Zustandsmenge von *TM* mit zwei neuen Zuständen q_{copy} und q_{skip}, die nicht in $Q_1 \cup Q_2$ enthalten sind,
$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{\#\}$ das Arbeitsalphabet von *TM* mit einem neuen Symbol #, das nicht in $\Sigma_1 \cup \Sigma_2$ enthalten ist,
$d : (Q \setminus \{q_{2,accept}\}) \times \Sigma^{k_1+k_2} \to Q \times (\Sigma \times \{L, R, S\})^{k_1+k_2}$ die Überführungsfunktion, die sich aus den Überführungsfunktionen d_1 und d_2 wie folgt ergibt:

Die Bänder von *TM* werden von 1 bis $k_1 + k_2$ numeriert; die ersten k_1 Bänder sind die ursprünglichen Bänder von TM_1. Insbesondere ist das Eingabeband von *TM* das ursprüngliche Eingabeband von TM_1. Das Band mit der Nummer $k_1 + 1$ ist das ursprüngliche Eingabeband von TM_2. Ein Eingabewort $w \in I^*$ wird auf das erste Band von *TM* gegeben.

Es sei $n = |w|$. In die erste Zelle des Bands mit der Nummer $k_1 + 1$ wird das Zeichen # geschrieben und *w* auf dieses Band kopiert. Das Zeichen # dient dazu, das linke Ende des Bands mit der Nummer $k_1 + 1$ zu erkennen. Nach diesem Kopiervorgang stehen die Köpfe des ersten und des $(k_1 + 1)$-ten Bands jeweils über der (*n*+1)-ten Zelle, alle übrigen Köpfe wurden nicht bewegt. Das Ende des Kopiervorgangs wurde dadurch erkannt, daß auf dem ersten Band das Leerzeichen gelesen wurde. Nun werden die Köpfe des ersten und des $(k_1 + 1)$-ten Bands beide zurück auf das erste Zeichen von *w* gesetzt.

Für den Kopier- und den Rücksetzvorgang der Köpfe werden folgende Zeilen in die Überführungsfunktion d von *TM* aufgenommen:

$$d\left(q_{1,0}, \underbrace{a, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left(q_{copy}, \underbrace{(a, S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(\#, R), (b, S), ..., (b, S)}_{k_2}\right) \quad \text{für alle}$$

$a \in \Sigma_1$ (*b* ist das Leerzeichen),

$$d\left(q_{copy}, \underbrace{a, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left(q_{copy}, \underbrace{(a, R), (b, S), ..., (b, S)}_{k_1}, \underbrace{(a, R), (b, S), ..., (b, S)}_{k_2}\right) \quad \text{für alle}$$

$a \in I$ (man beachte, daß *a* = Leerzeichen hierbei nicht vorkommt),

$$d\left(q_{copy}, \underbrace{b, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left(q_{skip}, \underbrace{(b, S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(b, L), (b, S), ..., (b, S)}_{k_2}\right),$$

$$d\left(q_{skip}, \underbrace{a', b, ..., b}_{k_1}, \underbrace{a, b, ..., b}_{k_2}\right) = \left(q_{skip}, \underbrace{(a', L), (b, S), ..., (b, S)}_{k_1}, \underbrace{(a, L), (b, S), ..., (b, S)}_{k_2}\right) \text{ für alle}$$

$a' \in I \cup \{b\}$ und $a \in I$,

$$d\left(q_{skip}, \underbrace{a', b, ..., b}_{k_1}, \underbrace{\#, b, ..., b}_{k_2}\right) = \left(q_{1,0}, \underbrace{(a', S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(\#, R), (b, S), ..., (b, S)}_{k_2}\right) \text{ für alle}$$

$a' \in I \cup \{b\}$.

Nun wird das Verhalten von TM_1 simuliert, wobei die Inhalte der Bänder mit den Nummern $k_1 + 1$ bis $k_1 + k_2$ (entsprechend den ursprünglichen Bändern von TM_2) und die Positionen der Köpfe auf diesen Bändern nicht verändert werden. Die dafür erforderlichen Zeilen in der Überführungsfunktion d lauten:

$$d\left(q_1, \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{a, b, ..., b}_{k_2}\right) = \left(q_1', \underbrace{(b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(a, S), (b, S), ..., (b, S)}_{k_2}\right) \text{für}$$

jeden Eintrag $d_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion d_1 von TM_1 und $a \in \Sigma_1$.

Falls bei der Simulation der akzeptierende Zustand $q_{1,accept}$ von TM_1 erreicht wird, d.h. $w \in L(TM_1)$, wird die Simulation des Verhaltens von TM_2 auf *w* gestartet. Auf den ersten k_1 ändert sich dabei nichts mehr. Folgende Zeilen werden in die Überführungsfunktion d aufgenommen:

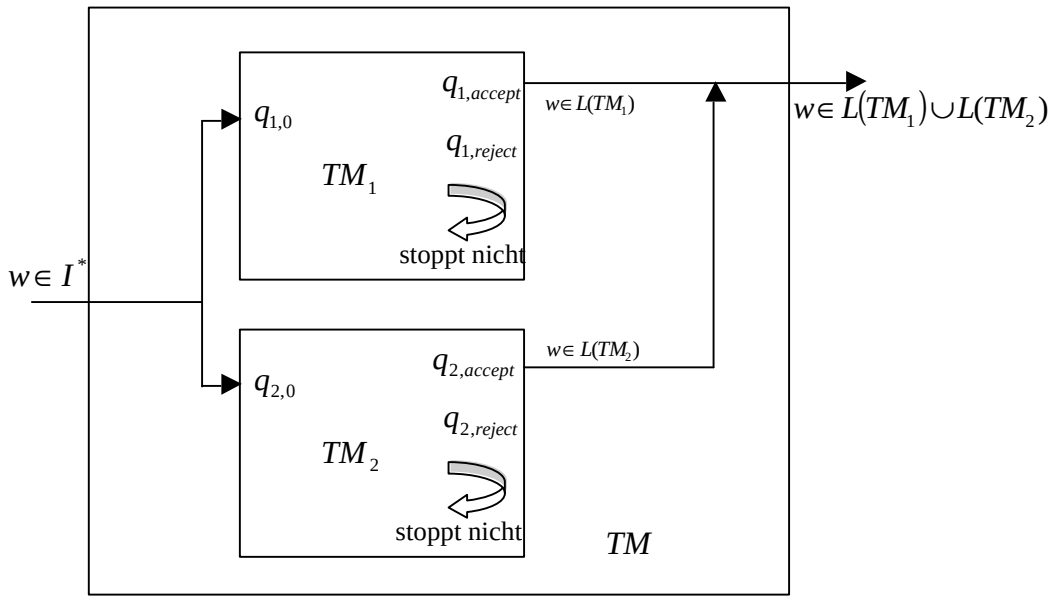
$$d\left(q_{1,accept}, \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{a, b, ..., b}_{k_2}\right) = \left(q_{2,0}, \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(a, S), (b, S), ..., (b, S)}_{k_2}\right)$$

mit $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und $a \in \Sigma_1$ und

$$d\left(q_2, \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2}\right) = \left(q_2', \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(h_1, d_1), (h_2, d_2), ..., (h_{k_2}, d_{k_2})}_{k_2}\right)$$

für jeden Eintrag $d_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, d_1), (h_2, d_2), ..., (h_{k_2}, d_{k_2}))$ in der Überführungsfunktion d_2 von TM_2 und $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$.

- Durch **Parallelschaltung** kann man aus TM_1 und TM_2 eine neue Turingmaschine *TM* bilden: ein Wort *w* wird sowohl auf das 1. Band von TM_1 als auch auf das 1. Band von TM_2 gegeben und das Verhalten beider Turingmaschinen simultan simuliert. Sobald eine der beiden Turingmaschinen *w* akzeptiert, wird *w* von *TM* akzeptiert. Es gilt: $L(TM) = L(TM_1) \cup L(TM_2)$.



Auch hier ist die formale Notation von *TM* mit Hilfe der Überführungsfunktion etwas mühsam:

Es ist $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept} \right)$. Dabei ist
$Q = (Q_1 \times Q_2) \cup \left\{ \left(q_{copy}, q_{2,0}\right), \left(q_{skip}, q_{2,0}\right), \left(q_{acc}, q_{acc}\right) \right\}$ die Zustandsmenge von *TM* mit zwei neuen Zuständen q_{copy} und q_{skip}, die nicht in Q_1 enthalten sind, und einem neuen Zustand q_{acc}, der nicht in $Q_1 \cup Q_2$ enthalten ist (die Zustände bestehen jetzt aus Paaren von Zuständen der Turingmaschinen TM_1 und TM_2 und drei neuen Zuständen),
$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{\#\}$ das Arbeitsalphabet von *TM* mit einem neuen Symbol #, das nicht in Σ_2 enthalten ist,
$d : (Q \setminus \left\{ \left(q_{acc}, q_{acc}\right) \right\}) \times \Sigma^{k_1+k_2} \to Q \times \left(\Sigma \times \{L, R, S\}\right)^{k_1+k_2}$ die Überführungsfunktion, die sich aus den Überführungsfunktionen d_1 und d_2 ergibt (siehe unten),
$q_0 = \left(q_{1,0}, q_{2,0} \right)$ der Anfangszustand und
$q_{accept} = \left(q_{acc}, q_{acc} \right)$ der akzeptierende Zustand von *TM*.

Die Bänder von *TM* werden wieder von 1 bis $k_1 + k_2$ numeriert; die ersten k_1 Bänder sind die ursprünglichen Bänder von TM_1. Insbesondere ist das Eingabeband von *TM* das ursprüngliche Eingabeband von TM_1. Das Band mit der Nummer $k_1 + 1$ ist das ursprüngliche Eingabeband von TM_2. Ein Eingabewort $w \in I^*$ wird auf das erste Band von *TM* gegeben. Wieder wird in die erste Zelle des Bands mit der Nummer $k_1 + 1$ das Zeichen # geschrieben und *w* auf dieses Band kopiert. Anschließend werden der Kopf des ersten Bands auf den Bandanfang und der Kopf des $(k_1 + 1)$-ten Bands auf die zweite Zelle (rechte Nachbarzelle der Zelle, die das Zeichen # enthält) zurückgesetzt. Der Vorgang verläuft ähnlich dem Vorgang, der bei der Hintereinanderschaltung von TM_1 und TM_2 beschrieben wurde. Die erforderlichen Einträge in der Überführungsfunktion lauten jetzt:

$$d\left(\left(q_{1,0}, q_{2,0}\right), \underbrace{a, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2} \right) = \left(\left(q_{copy}, q_{2,0}\right), \underbrace{(a, S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(\#, R), (b, S), ..., (b, S)}_{k_2} \right)$$

für alle $a \in \Sigma_1$ (*b* ist das Leerzeichen),

$$d\left(\left(q_{copy}, q_{2,0}\right), \underbrace{a, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2} \right) = \left(\left(q_{copy}, q_{2,0}\right), \underbrace{(a, R), (b, S), ..., (b, S)}_{k_1}, \underbrace{(a, R), (b, S), ..., (b, S)}_{k_2} \right)$$

für alle $a \in I$ (man beachte, daß a = Leerzeichen hierbei nicht vorkommt),

$$d\left(\left(q_{copy}, q_{2,0}\right), \underbrace{b, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left(\left(q_{skip}, q_{2,0}\right), \underbrace{(b, S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(b, L), (b, S), ..., (b, S)}_{k_2}\right)$$

$$d\left(\left(q_{skip}, q_{2,0}\right), \underbrace{a', b, ..., b}_{k_1}, \underbrace{a, b, ..., b}_{k_2}\right) = \left(\left(q_{skip}, q_{2,0}\right), \underbrace{(a', L), (b, S), ..., (b, S)}_{k_1}, \underbrace{(a, L), (b, S), ..., (b, S)}_{k_2}\right)$$

für alle $a' \in I \cup \{b\}$ und $a \in I$,

$$d\left(\left(q_{skip}, q_{2,0}\right), \underbrace{a', b, ..., b}_{k_1}, \underbrace{\#, b, ..., b}_{k_2}\right) = \left(\left(q_{1,0}, q_{2,0}\right), \underbrace{(a', S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(\#, R), (b, S), ..., (b, S)}_{k_2}\right)$$

für alle $a' \in I \cup \{b\}$.

Nun wird parallel das Verhalten von TM_1 und von TM_2 simuliert, wobei für die Simulation von TM_1 nur auf die Inhalte der Bänder mit den Nummern 1 bis k_1 (entsprechend den ursprünglichen Bändern von TM_1) und für die Simulation von TM_2 nur auf die Inhalte der Bänder mit den Nummern k_1+1 bis k_1+k_2 (entsprechend den ursprünglichen Bändern von TM_2) zugegriffen wird. Die dafür erforderlichen Zeilen in der Überführungsfunktion d lauten:

$$d\left(\left(q_1, q_2\right), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2}\right)$$

$$= \left(\left(q_1', q_2'\right), \underbrace{(b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(h_1, e_1), (h_2, e_2), ..., (h_{k_2}, h_{k_2})}_{k_2}\right),$$

falls $d_1\left(q_1, a_1, a_2, ..., a_{k_1}\right) = \left(q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1})\right)$ in der Überführungsfunktion d_1 von TM_1 und $d_2\left(q_2, g_1, g_2, ..., g_{k_2}\right) = \left(q_2', (h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2})\right)$ in der Überführungsfunktion d_2 von TM_2 ist.

Falls bei der Simulation ein Zustand der Form $\left(q_{1,reject}, q_2\right)$ mit $q_2 \in Q_2$ bzw. der Form $\left(q_1, q_{2,reject}\right)$ mit $q_1 \in Q_1$ erreicht wird, muß sichergestellt werden, daß die Simulation von TM_2 bzw. von TM_1 weiterläuft. Daher werden auch folgende Einträge in die Überführungsfunktion d aufgenommen:

$$d\left(\left(q_{1,reject}, q_2\right), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2}\right)$$

$$= \left(\left(q_{1,reject}, q_2'\right), \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2})}_{k_2}\right)$$

mit $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und falls $d_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2}))$ in der Überführungsfunktion d_2 von TM_2 ist und

$$d\left((q_1, q_{2,reject}), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2}\right)$$

$$= \left((q_1', q_{2,reject}), \underbrace{(b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(g_1, S), (g_2, S), ..., (g_{k_2}, S)}_{k_2}\right)$$

für jeden Eintrag $d_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion d_1 von TM_1 und $g_1 \in \Sigma_2$, ..., $g_{k_2} \in \Sigma_2$.

Schließlich soll *TM* die Eingabe *w* akzeptieren, wenn TM_1 oder TM_2 die Eingabe akzeptiert. Folgende Einträge werden daher in die Überführungsfunktion von *TM* aufgenommen:

$$d\left((q_{1,accept}, q_2), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2}\right)$$

$$= \left((q_{acc}, q_{acc}), \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(g_1, S), (g_2, S), ..., (g_{k_2}, S)}_{k_2}\right) \text{ und}$$

$$d\left((q_1, q_{2,accept}), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2}\right)$$

$$= \left((q_{acc}, q_{acc}), \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(g_1, S), (g_2, S), ..., (g_{k_2}, S)}_{k_2}\right)$$

für alle $q_1 \in Q_1$, $q_2 \in Q_2$, $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und $g_1 \in \Sigma_2$, ..., $g_{k_2} \in \Sigma_2$.

- Dadurch, daß man die Turingmaschine TM_2 als Unterprogramm in die Berechnung der Turingmaschine TM_1 einsetzt, erhält man eine neue Turingmaschine *TM*, die prinzipiell folgendermaßen abläuft: Eine Eingabe $w \in I^*$ wird in TM_1 eingegeben. Ein Band von TM_1 wird als „Parameterübergabeband“ für TM_2 ausgezeichnet. Sobald TM_1 in einen als Parameterübergabezustand ausgezeichneten Zustand $q_?$ gelangt, wird der Inhalt des Parameterübergabebands auf das erste Band von TM_2 kopiert. Wird diese Eingabe von TM_2 akzeptiert, wobei bei Akzeptanz der Inhalt des k_2-ten Bands *y* lautet, wird das Parameterübergabeband gelöscht, *y* darauf kopiert, das erste Band von TM_2 gelöscht und die Be-

rechnung von TM_1 fortgesetzt. Die Ausformulierung der Details dieser Konstruktion werden dem Leser als Übung überlassen.

Obige Überlegungen und Ausführungen in Kapitel 3.1 zeigen:

Satz 2.1-2:

Die Klasse der von Turingmaschinen akzeptierten Mengen ist gegenüber Vereinigungsbildung und Schnittbildung abgeschlossen.

Die Klasse der von Turingmaschinen akzeptierten Mengen ist gegenüber Komplementbildung nicht abgeschlossen: Wenn $L \subseteq \Sigma^*$ von einer Turingmaschine *TM* akzeptiert wird, d.h. $L = L(TM)$, muß das Komplement $\Sigma^* \setminus L$ von L nicht auch notwendigerweise von einer Turingmaschine akzeptiert werden.

Für eine *k*-DTM $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept} \right)$ und eine Eingabe $w \in L(TM)$ gelte
$(q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^m K_{accept}$
mit einer Endkonfiguration $K_{accept} = (q_{accept}, (a_1, i_1), ..., (a_k, i_k))$. Dann wird durch $t_{TM}(w) = m$ eine partielle Funktion $t_{TM} : \Sigma^* \rightarrow \mathbf{N}$ definiert, die angibt, **wieviele Überführungen *TM* macht, um *w* zu akzeptieren**. Es handelt sich hierbei um eine partielle Funktion, da t_{TM} für Wörter $w \notin L(TM)$ nicht definiert ist.

Die **Zeitkomplexität** von *TM* (**im schlechtesten Fall, worst case**) wird definiert durch die partielle Funktion $T_{TM} : \mathbf{N} \rightarrow \mathbf{N}$ mit
$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \leq n \}$.

Bemerkung: In der Literatur findet man auch die Definition

$$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| = n \}$$

Entsprechend kann man die **Platzkomplexität** von *TM* (**im schlechtesten Fall, worst case**) $S_{TM}(n)$ als den maximalen Abstand vom linken Ende eines Bandes definieren, den ein Schreib/Lesekopf bei der Akzeptanz eines Worts $w \in L(TM)$ mit $|w| \leq n$ erreicht.

Beispiele:

Die oben angegebene Turingmaschine zur Akzeptanz der Palindrome über $\{0,1\}$ hat eine Zeitkomplexität der Größe $T_{TM}(n) = 4n + 3$ und eine Platzkomplexität $S_{TM}(n) = n + 2$.

Die oben angegebene Turingmaschine zur Berechnung der Funktion $f : \begin{cases} \mathbf{N} & \rightarrow & \mathbf{N} \\ n & \rightarrow & n+1 \end{cases}$ hat folgende Zeitkomplexität: Der Wert n wird in Binärdarstellung auf das Eingabeband gegeben. Die Eingabe $w = bin(n)$ belegt daher für $n \geq 1$ $|w| = |bin(n)| = \lfloor \log_2(n) \rfloor + 1$ viele Zeichen bzw. ein Zeichen für $n = 0$. Dann führt die Turingmaschine $O(|w|) = O(\log(n))$ viele Schritte aus. Die Zeitkomplexität ist linear in der Länge der Eingabe.

Bei der Betrachtung der Zeitkomplexität werden in diesem Beispiel also die **Bitoperationen** gezählt, die benötigt werden, um $f(n)$ zu berechnen, wenn n in Binärdarstellung gegeben ist. Die Turingmaschine dieses Beispiels läßt sich leicht zu einer Turingmaschine modifizieren, die die Funktion $SUM : \begin{cases} \mathbf{N} \times \mathbf{N} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & n+m \end{cases}$ berechnet. Hierbei werden die Zahlen n und m in Binärdarstellung, getrennt durch das Zeichen #, auf das Eingabeband geschrieben, d.h. die Eingabe hat die Form $w = bin(n)\#bin(m)$. Dann werden die Zahlen n und m stellengerecht addiert. Auch für die arithmetischen Funktionen $DIF : \begin{cases} \mathbf{N} \times \mathbf{N} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & \begin{cases} n-m & \text{für } n \geq m \\ 0 & \text{für } n < m \end{cases} \end{cases}$,

$MULT : \begin{cases} \mathbf{N} \times \mathbf{N} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & n \cdot m \end{cases}$ und $DIV : \begin{cases} \mathbf{N} \times \mathbf{N}_{>0} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & \lfloor n/m \rfloor \end{cases}$ lassen sich entsprechende Turingmaschinen angeben. Dabei wird beispielsweise die Berechnung von *MULT* auf sukzessive Anwendung der Berechnung für *SUM* zurückgeführt. In jedem Fall erfolgt die Eingabe in der Form $w = bin(n)\#bin(m)$, und die Zeitkomplexitäten geben an, wieviele Bitoperationen zur Berechnung der jeweiligen Funktionen erforderlich sind. Die Ergebnisse sind in folgendem Satz zusammengefaßt.

Satz 2.1-3:

Es seien *n* und *m* natürliche Zahlen. Mit $size(n)$ werde die Länge der Binärdarstellung der Zahl *n* (ohne führende binäre Nullen) bezeichnet. Dabei sei $size(0)=1$. Es gelte $|n| \geq |m|$, $k = size(n) \geq size(m)$, $k \in O(\log(n))$. Dann gibt es Turingmaschinen, die die Funktionen $SUM(n,m)$, $DIF(n,m)$, $MULT(n,m)$ und $DIV(n,m)$ berechnen und folgende Zeitkomplexitäten besitzen:

(i) Die Addition und Differenzenbildung der Zahlen *n* und *m* (Berechnung von $SUM(n,m)$ und $DIF(n,m)$) erfordert jeweils $O(k)$, also $O(\log(n))$ viele Bitoperationen.

(ii) Die Multiplikation der Zahlen *n* und *m* (Berechnung von $MULT(n,m)$) kann in $O(k^2)$, also in $O((\log(n))^2)$ vielen Bitoperationen durchgeführt werden.

(iii) Ist $size(n) \geq 2 \cdot size(m)$, dann kann die Berechnung des ganzzahligen Quotienten $\lfloor n/m \rfloor$ (Berechnung von $DIV(n,m)$) in $O(k^2)$, also in $O((\log(n))^2)$ vielen Bitoperationen durchgeführt werden.

Meist ist man nicht am exakten Wert der Anzahl der Konfigurationsänderungen interessiert, sondern nur an der **Größenordnung der Zeitkomplexität (**in Abhängigkeit von der Größe der Eingabe). Bei der Analyse wird man meist eine obere Schranke $g(n)$ für $T_{TM}(n)$ herleiten, d.h. man wird eine Aussage der Form $T_{TM}(n) \leq g(n)$ begründen. In diesem Fall ist $T_{TM}(n) \in O(g(n))$. Eine zusätzliche Aussage $T_{TM}(n) \leq h(n) \leq g(n)$ mit einer Funktion $h(n)$ führt auf die verbesserte Abschätzung $T_{TM}(n) \in O(h(n))$. Man ist also bei der worst case-Analyse an einer möglichst kleinen oberen Schranke für $T_{TM}(n)$ interessiert.

Es gibt eine Reihe von **Varianten von Turingmaschinen**:

- Die Turingmaschine besitzt Bänder, die nach links und rechts unendlich lang sind.
- Die Turingmaschine besitzt ein einziges nach links und rechts oder nur zu einer Seite unendlich langes Band.
- Die Turingmaschine besitzt mehrdimensionale Bänder bzw. ein mehrdimensionales Band.
- Das Eingabealphabet der Turingmaschine besteht aus zwei Zeichen, etwa $I = \{0, 1\}$.
- Das Arbeitsalphabet der Turingmaschine besteht aus zwei Zeichen, etwa $\Sigma = \{0, 1\}$.
- Die Turingmaschine hat endlich viele Endzustände.

Es zeigt sich, daß alle Definitionen **algorithmisch äquivalent** sind, d.h. eine Turingmaschinenvariante kann eine andere Turingmaschinenvariante simulieren. Allerdings ändert sich da-

bei u.U. die Zeitkomplexität, in der von der jeweiligen Turingmaschinenvariante Sprachen erkannt werden. Beispielsweise kann eine *k*-DTM *TM* mit Zeitkomplexität $T_{TM}(n)$ durch eine 1-DTM mit Zeitkomplexität $O\left(T_{TM}^2(n)\right)$ simuliert werden.

2.2 Random Access Maschinen

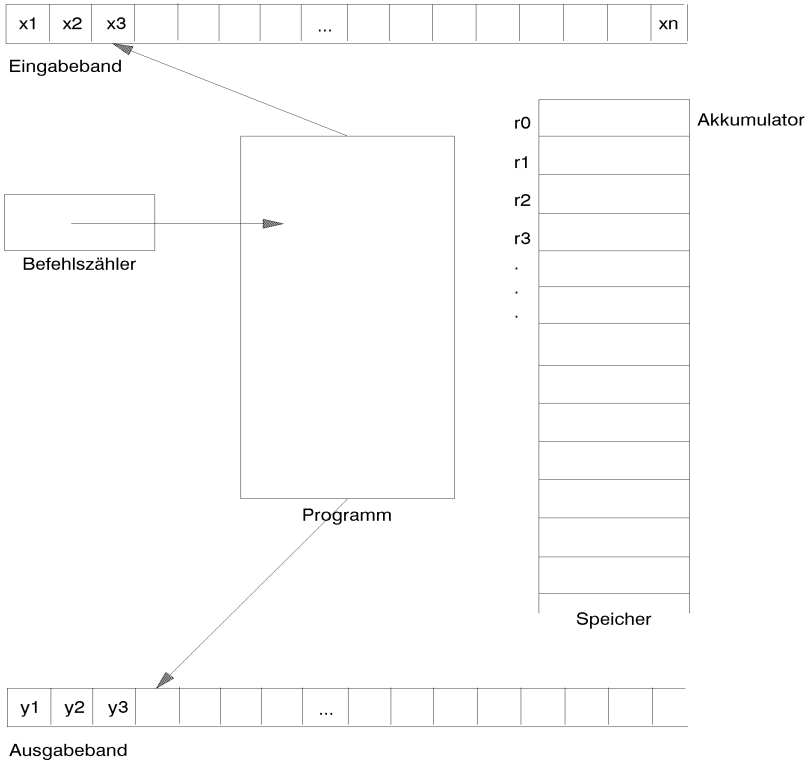
Eine **Random Access Maschine (RAM)** modelliert einen Algorithmus in Form eines sehr einfachen Computers, der ein Programm ausführt, das sich nicht selbst modifizieren kann und fest in einem Programmspeicher geladen ist. Das Konzept der RAM ist zunächst als Modell für die Berechnung partieller Funktionen $f : \mathbf{Z}^n \to \mathbf{Z}^m$ gedacht, die *n*-stellige Zahlenfolgen (ganzer Zahlen) auf *m*-stellige Zahlenfolgen abbilden. Eine Turingmaschine dagegen ist zur Akzeptanz von Sprachen $L \subseteq \Sigma^*$ bzw. zur Berechnung von partieller Funktionen $f : \Sigma^* \to \Sigma'^*$ konzipiert, die Zeichenketten (Wörtern) über einem endlichen Alphabet auf Zeichenketten über eventuell einem anderen Alphabet abbilden. Es wird sich jedoch zeigen, daß beide Modelle äquivalent sind. Beide Modelle sind in der Lage, numerische Funktionen zu berechnen und als Sprachakzeptoren eingesetzt zu werden. Während die „Programmierung" einer Turingmaschine in der Definition der Überführungsfunktion besteht, eine Vorgehensweise, die verglichen mit der gängigen Programmierung eines Computers zumindest gewöhnungsbedürftig ist, entspricht die Programmierung einer RAM eher der Programmierung eines Computers auf Maschinensprachebene.

Eine RAM hat folgende Bestandteile:

- **Eingabeband**: eine Folge von Zellen, die als Eingabe *n* ganze (positive oder negative) Zahlen x_1, ..., x_n enthalten und von einem Lesekopf nur gelesen werden können. Nachdem eine Zahl x_i gelesen wurde, rückt der Lesekopf auf die benachbarte Zelle, die die ganze Zahl x_{i+1} enthält.
- **Ausgabeband**: eine Folge von Zellen, über die ein Schreibkopf von links nach rechts wandert, der nacheinander ganze Zahlen y_j schreibt. Ist eine Zahl geschrieben, kann sie nicht mehr verändert werden; der Schreibkopf rückt auf das rechts benachbarte Feld.
- **Speicher**: eine unendliche Folge r_0, r_1, r_2, ..., von **Registern**, die jeweils in der Lage sind, eine beliebig große ganze Zahl aufzunehmen. Das Register r_0 wird als **Akkumulator** bezeichnet. In ihm finden alle arithmetische Operationen statt.
- **Programm**: eine Folge aufsteigend numerierter Anweisungen, die fest in der RAM „verdrahtet" sind. Zur besseren Lesbarkeit eines RAM-Programms können einzelne Anweisungen auch mit **symbolischen Marken** versehen werden. Jede RAM stellt ein eigenes Programm dar, das natürlich mit unterschiedlichen Eingaben konfrontiert werden kann.

Das Programm kann sich nicht modifizieren. Ein Programm ist aus einfachen Befehlen aufgebaut (siehe unten). Die einzelnen Anweisungen werden nacheinander ausgeführt.

- **Befehlszähler**: enthält die Nummer der nächsten auszuführenden Anweisung.



Um die Bedeutung des **Befehlsvorrats** einer RAM festzulegen, wird die **Speicherabbildungsfunktion** $c : \mathbf{N} \to \mathbf{Z}$ verwendet. $c(i)$ gibt zu jedem Zeitpunkt des Programmlaufs den Inhalt des Registers r_i an. Zu Beginn des Programmlaufs wird jedes Register mit dem Wert 0 initialisiert, d.h. es ist $c(i) = 0$ für jedes $i \in \mathbf{N}$.

Jeder **Anweisung (Befehl)** ist aus einem **Operationscode** und einem **Operanden** aufgebaut. Ein Operand kann eine der folgenden Formen aufweisen:

1. i
2. $c(i)$, für $i \geq 0$; bei $i < 0$ stoppt die RAM
3. $c(c(i))$, für $i \geq 0$ und $c(i) \geq 0$; bei $i < 0$ oder $c(i) < 0$ stoppt die RAM.

Die RAM-Anweisungen und ihre Bedeutung sind folgender Tabelle zu entnehmen. Dabei steht i für eine natürliche Zahl und m für eine Anweisungsnummer bzw. für eine Anweisungsmarke im Programm der RAM.

RAM-Anweisung	Bedeutung
LOAD i	$c(0) := i$
LOAD $c(i)$ für $i \geq 0$	$c(0) := c(i)$
LOAD $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(c(i))$
STORE $c(i)$ für $i \geq 0$	$c(i) := c(0)$
STORE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) := c(0)$
ADD i	$c(0) := c(0) + i$
ADD $c(i)$ für $i \geq 0$	$c(0) := c(0) + c(i)$
ADD $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) + c(c(i))$
SUB i	$c(0) := c(0) - i$
SUB $c(i)$ für $i \geq 0$	$c(0) := c(0) - c(i)$
SUB $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) - c(c(i))$
MULT i	$c(0) := c(0) * i$
MULT $c(i)$ für $i \geq 0$	$c(0) := c(0) * c(i)$
MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * c(c(i))$
DIV i für $i \neq 0$	$c(0) := \lfloor c(0) / i \rfloor$
DIV $c(i)$ für $i \geq 0$	$c(0) := \lfloor c(0) / c(i) \rfloor$, falls $c(i) \neq 0$ ist, sonst stoppt die RAM
DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / c(c(i)) \rfloor$, falls $c(c(i)) \neq 0$ ist, sonst stoppt die RAM
READ $c(i)$ für $i \geq 0$	$c(i) :=$ momentaner Eingabewert, und der Lesekopf rückt eine Zelle nach rechts
READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) :=$ momentaner Eingabewert, und der Lesekopf rückt eine Zelle nach rechts

../..

WRITE *i*	*i* wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
WRITE $c(i)$ für $i \geq 0$	$c(i)$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i))$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
JUMP *m*	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke *m* trägt (nächste auszuführende Anweisung)
JGTZ *m*	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke *m* trägt (nächste auszuführende Anweisung), falls $c(0) > 0$ ist, sonst wird der Inhalt des Befehlszählers um 1 erhöht
JZERO *m*	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke *m* trägt (nächste auszuführende Anweisung), falls $c(0) = 0$ ist, sonst wird der Inhalt des Befehlszählers um 1 erhöht
HALT	Die RAM stoppt

Falls eine Anweisung nicht definiert ist, z.B. STORE 3 oder STORE $c(c(3))$ mit $c(3) = -10$, dann stoppt die RAM, desgleichen bei einer Division durch 0.

Die RAM *RAM* definiert bei Beschriftung der ersten n Zellen des Eingabebandes mit x_1, ..., x_n eine partielle Funktion $f_{RAM} : \mathbf{Z}^n \to \mathbf{Z}^m$ (die Funktion ist partiell, da *RAM* nicht bei jeder Eingabe stoppen muß): falls *RAM* bei Eingabe von x_1, ..., x_n stoppt, nachdem y_1, ..., y_m in die *m* ersten Zellen des Ausgabebands geschrieben wurde, dann ist $f_{RAM}(x_1, ..., x_n) = (y_1, ..., y_m)$.

Eine RAM zur Berechnung der Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & \begin{cases} 1 & \text{für} \quad n = 0 \\ n^n & \text{für} \quad n \geq 1 \end{cases} \end{cases}$

Ein entsprechendes RAM-Programm lautet:

RAM-Anweisungsfolge		Bemerkung zur Bedeutung
Zeilennummer	RAM-Befehl	
1	READ *c*(1)	$r_1 := n$, Eingabe
2 3 4 5	LOAD *c*(1) JGTZ pos WRITE 0 JUMP endif	**IF** $r_1 \leq 0$ **THEN** *Write*(0)
pos 7	LOAD *c*(1) STORE *c*(2)	$r_2 := r_1$
8 9 10	LOAD *c*(1) SUB 1 STORE *c*(3)	$r_3 := r_1 - 1$
while 12 13	LOAD *c*(3) JGTZ continue JUMP endwhile	**WHILE** $r_3 > 0$ **DO**
continue 15 16	LOAD *c*(2) MULT *c*(1) STORE *c*(2)	$r_2 := r_2 * r_1$
17 18 19	LOAD *c*(3) SUB 1 STORE *c*(3)	$r_3 := r_3 - 1$
20	JUMP while	
endwhile	WRITE *c*(2)	*Write* (r_2)
endif	HALT	

Eine RAM kann auch als **Akzeptor einer Sprache** interpretiert werden. Die Elemente des endlichen Alphabets Σ werden dazu mit den Zahlen 1, 2, ..., $|\Sigma|$ identifiziert. Das RAM-Programm *RAM* akzeptiert $L \subseteq \Sigma^*$ auf folgende Weise. In die ersten *n* Zellen des Eingabebandes werden Zahlen geschrieben, die den Buchstaben x_1, ..., x_n eines Wortes $w \in \Sigma^*$, $w = x_1 \ldots x_n$, entsprechenden. In die $(n+1)$-te Zelle kommt als Endmarkierung der Wert 0. *RAM* akzeptiert *w,* falls alle den Buchstaben von *w* entsprechenden Zahlen und die Endmarkierung 0 gelesen wurden, von *RAM* eine 1 auf das Ausgabeband geschrieben wurde und *RAM* stoppt, dann ist $w \in L$. Falls $w \notin L$ ist, dann stoppt *RAM* mit einer Ausgabe ungleich 1, oder *RAM* stoppt nicht. Die auf diese Weise akzeptierten Wörter aus Σ^* bilden die Menge $L(RAM)$.

Ein RAM-Programm *P* zur Akzeptanz von

$$L(P) = \{ w \mid w \in \{1,2\}^* \text{ und die Anzahl der 1'en ist gleich der Anzahl der 2'en} \}$$

P liest jedes Eingabesymbol nach Register r_1 (hier wird angenommen, daß nur die Zahlen 0, 1 und 2 auf dem Eingabeband stehen) und berechnet in Register r_2 die Differenz *d* der Anzahlen der bisher gelesenen 1‘en und 2‘en. Wenn beim Lesen der Endmarkierung 0 diese Differenz gleich 0 ist, wird eine 1 auf das Ausgabeband geschrieben, und *P* stoppt.

RAM-Anweisungsfolge		Bemerkung zur Bedeutung
Zeilennummer	RAM-Befehl	
	LOAD 0 STORE *c*(2)	$d := 0$
	READ *c*(1)	*Read* (*x*)
while	LOAD *c*(1) JZERO endwhile	**WHILE** $x \neq 0$ **DO**
	LOAD *c*(1) SUB 1 JZERO one	**IF** $x \neq 1$
	LOAD *c*(2) SUB 1 STORE *c*(2)	**THEN** $d := d - 1$
	JUMP endif	
one	LOAD *c*(2) ADD 1 STORE *c*(2)	**ELSE** $d := d + 1$
endif	READ *c*(1) JUMP while	*Read* (*x*)
endwhile	LOAD *c*(2) JZERO output HALT	**IF** $d = 0$ **THEN** *Write* (1)
output	WRITE 1 HALT	

Auch beim RAM-Modell interessieren **Zeit- und Raumkomplexität einer Berechnung**.

Mit $\mathbf{Z}^n$ werde das *n*-fache kartesische Produkt von **Z** bezeichnet, d.h. $\mathbf{Z}^n = \underbrace{\mathbf{Z} \times ... \times \mathbf{Z}}_{n\text{-mal}}$. Die Menge $\mathbf{Z}^*$ aller endlichen Folgen ganzer Zahlen sei definiert durch $\mathbf{Z}^* = \bigcup_{n \geq 0} \mathbf{Z}^n$.

Ein RAM-Programm *RAM* lese die Eingabe $x_1, ..., x_n$ und durchlaufe bis zum Erreichen der HALT-Anweisung *m* viele Anweisungen (einschließlich der HALT-Anweisung). Dann wird durch $t_{RAM}(x_1, ..., x_n) = m$ eine partielle Funktion $t_{RAM} : \mathbf{Z}^* \to \mathbf{N}$ definiert. Falls *RAM* bei Eingabe von $x_1, ..., x_n$ nicht anhält, dann ist $t_{RAM}(x_1, ..., x_n)$ nicht definiert.

In diesem Modell werden zwei Kostenkriterien unterschieden:

Beim **uniformen Kostenkriterium** benötigt die Ausführung jeder Anweisung eine Zeiteinheit, und jedes Register belegt eine Platzeinheit. Die **Zeitkomplexität** von *RAM* (**im schlechtesten Fall, worst case**) wird **unter dem uniformen Kostenkriterium** definiert durch die partielle Funktion $T_{RAM} : \mathbf{N} \to \mathbf{N}$ mit
$T_{RAM}(n) = \max\{ t_{RAM}(w) \mid w \text{ besteht aus bis zu } n \text{ vielen natürlichen Zahlen} \}$.

Entsprechend wird die **Platzkomplexität** von *RAM* (**im schlechtesten Fall, worst case**) $S_{RAM}(n)$ **unter dem uniformen Kostenkriterium** als die während der Rechnung benötigte maximale Anzahl an Registern definiert, wenn bis zu *n* viele Zahlen eingelesen werden.

Ein Register bzw. eine Zelle des Eingabe- und Ausgabebandes kann im RAM-Modell eine beliebig große Zahl aufnehmen. Eine Berechnung mit *n* „großen Zahlen“ ist unter dem uniformen Kostenkriterium genauso komplex wie eine Berechnung mit *n* „kleinen Zahlen“. Diese Sichtweise entspricht häufig nicht den praktischen Gegebenheiten. Dies gilt in der Praxis besonders dann, wenn Speicherzellen eine beschränkte Größe aufweisen, so daß für die Handhabung großer Zahlen pro Zahl mehrere Speicherzellen erforderlich sind. Es erscheint daher sinnvoll, die Größenordnung der bei einer Berechnung beteiligten Zahlen bzw. Operanden mit zu berücksichtigen. Dieses führt auf das **logarithmische Kostenkriterium**, das die Größen der bei einer Berechnung beteiligten Zahlen (gemessen in der Anzahl der Stellen zur Darstellung einer Zahl in einem geeigneten Stellenwertsystem) berücksichtigt.

Mit $l(i)$ werde die **Länge** einer ganzen Zahl i bezeichnet:

$$l(i) = \begin{cases} \lfloor \log|i| \rfloor + 1 & \text{für } i \neq 0 \\ 1 & \text{für } i = 0 \end{cases}$$

Die **Kosten eines Operanden** einer RAM-Anweisung faßt folgende Tabelle zusammen.

Operand	Kosten
i	$l(i)$
$c(i)$	$l(i) + l(c(i))$
$c(c(i))$	$l(i) + l(c(i)) + l(c(c(i)))$

Beispielsweise erfordert die Ausführung einer Anweisung ADD $c(c(i))$ folgende Einzelschritte:

1. „Decodieren“ des Operanden i: Kosten $l(i)$
2. „Lesen“ von $c(i)$: Kosten $l(c(i))$
3. „Lesen“ von $c(c(i))$: Kosten $l(c(c(i)))$
4. Ausführung von ADD $c(c(i))$: Kosten $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$.

Die folgende Tabelle zeigt die Kosten unter dem logarithmischen Kostenkriterium der Ausführung für jede RAM-Anweisung.

RAM-Anweisung	Bedeutung	Kosten der Ausführung
LOAD i	$c(0) := i$	$l(i)$
LOAD $c(i)$ für $i \geq 0$	$c(0) := c(i)$	$l(i) + l(c(i))$
LOAD $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(c(i))$	$l(i) + l(c(i)) + l(c(c(i)))$
STORE $c(i)$ für $i \geq 0$	$c(i) := c(0)$	$l(c(0)) + l(i)$
STORE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) := c(0)$	$l(c(0)) + l(i) + l(c(i))$
ADD i	$c(0) := c(0) + i$	$l(c(0)) + l(i)$
ADD $c(i)$ für $i \geq 0$	$c(0) := c(0) + c(i)$	$l(c(0)) + l(i) + l(c(i))$
ADD $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) + c(c(i))$	$l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$

../..

SUB i SUB $c(i)$ für $i \geq 0$ SUB $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) - i$ $c(0) := c(0) - c(i)$ $c(0) := c(0) - c(c(i))$	$l(c(0)) + l(i)$ $l(c(0)) + l(i) + l(c(i))$ $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
MULT i MULT $c(i)$ für $i \geq 0$ MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * i$ $c(0) := c(0) * c(i)$ $c(0) := c(0) * c(c(i))$	$l(c(0)) + l(i)$ $l(c(0)) + l(i) + l(c(i))$ $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
DIV i für $i \neq 0$ DIV $c(i)$ für $i \geq 0$ DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / i \rfloor$ $c(0) := \lfloor c(0) / c(i) \rfloor$ $c(0) := \lfloor c(0) / c(c(i)) \rfloor$	$l(c(0)) + l(i)$ $l(c(0)) + l(i) + l(c(i))$ $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
READ $c(i)$ für $i \geq 0$ READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(i) :=$ Eingabewert $c(c(i)) :=$ Eingabewert	$l(\text{Eingabewert}) + l(i)$ $l(\text{Eingabewert}) + l(c(i))$
WRITE i WRITE $c(i)$ für $i \geq 0$ WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	Ausgabe von i Ausgabe von $c(i)$ Ausgabe von $c(c(i))$	$l(i)$ $l(c(i))$ $l(c(c(i)))$
JUMP m	Befehlszähler := m	1
JGTZ m	Befehlszähler := m, falls $c(0) > 0$ ist	$l(c(0))$
JZERO m	Befehlszähler := m, falls $c(0) = 0$ ist	$l(c(0))$
HALT	Die RAM stoppt	1

Die **Zeitkosten eines RAM-Programms** bei gegebenen Eingabewerten **unter dem logarithmischen Kostenkriterium** ist gleich der Summe der Kosten der abzuarbeitenden Anweisungen. Entsprechend sind die **Platzkosten eines RAM-Programms** bei gegebenen Eingabewerten **unter dem logarithmischen Kostenkriterium** gleich dem Produkt der während der Rechnung benötigte maximalen Anzahl an Registern und der Länge der längsten während der Abarbeitung abgespeicherten Zahl.

Kosten des RAM-Programms zur Berechnung der Funktion

$$f: \begin{cases} \mathbf{N} & \rightarrow & \mathbf{N} \\ n & \rightarrow & \begin{cases} 1 & \text{für} \quad n = 0 \\ n^n & \text{für} \quad n \geq 1 \end{cases} \end{cases}$$

	uniformes Kostenkriterium	Logarithmisches Kostenkriterium
Zeitkomplexität	$O(n)$	$O(n^2 \cdot \log(n))$
Platzkomplexität	$O(1)$	$O(\log(n))$

Kosten des RAM-Programm *P* zur Akzeptanz von

$L(P) = \{ w \mid w \in \{1,2\}^* \text{ und die Anzahl der 1'en ist gleich der Anzahl der 2'en} \}$

	uniformes Kostenkriterium	Logarithmisches Kostenkriterium
Zeitkomplexität	$O(n)$	$O(n \cdot \log(n))$
Platzkomplexität	$O(1)$	$O(\log(n))$

Eine RAM *RAM* kann eine deterministische Turingmaschine mit *k* Bändern (*k*-DTM) *TM* simulieren. Dazu wird jede Zelle von *TM* durch ein Register von *RAM* nachgebildet. Genauer: der Inhalt der *i*-ten Zelle des *j*-ten Bandes von *TM* kann in Register r_{ik+j+c} gespeichert werden. Hierbei wird angenommen, daß die Symbole des Alphabets Σ von *TM* für *RAM* mit den Zahlen 1, ..., $|\Sigma|$ identifiziert werden, so daß man davon sprechen kann, daß „ein Symbol aus Σ in einem Register oder einer Zelle des Eingabe- oder Ausgabebandes von *RAM* gespeichert werden kann“. Der Wert $c \geq k$ ist eine Konstante, durch die die Register r_1, ..., r_c als zusätzliche Arbeitsregister für *RAM* reserviert werden. Unter diesen Arbeitsregistern werden *k* Register, etwa r_1, ..., r_k, dazu verwendet, die jeweilige Kopfposition von *TM* während der Simulation festzuhalten (r_j enthält die Kopfposition des *j*-ten Bandes). Um den Inhalt einer Zelle von *TM* in der Simulation zu lesen, wird indirekte Adressierung eingesetzt. Beispielsweise kann der Inhalt der simulierten Zelle, über der sich gerade der Schreib/Lesekopf des *j*-ten Bandes befindet, in den Akkumulator durch die RAM-Anweisung LOAD *c*(*c*(*j*)) geladen werden.

Satz 2.2-1:

Hat die *k*-DTM *TM* die Zeitkomplexität $T_{TM}(n) \geq n$, dann gibt es eine RAM *RAM*, die die Eingabe von *TM* in die Register liest, die das erste Band nachbilden, und anschließend die $T_{TM}(n)$ Schritte von *TM* simuliert. Dieser Vorgang verläuft unter uniformem Kostenkriterium in $O(T_{TM}(n))$ vielen Schritten, unter logarithmischem Kostenkriterium in $O(T_{TM}(n) \cdot \log(T_{TM}(n)))$ vielen Schritten.

Umgekehrt gilt:

Es sei *L* eine Sprache, die von einem RAM-Programm der Zeitkomplexität $T_{RAM}(n)$ unter dem logarithmischen Kostenkriterium akzeptiert wird. Falls das RAM-Programm keine Multiplikationen oder Divisionen verwendet, dann wird *L* von einer *k*-DTM (für ein geeignetes *k*) mit Zeitkomplexität $O(T_{RAM}{}^2(n))$ akzeptiert.

Falls das RAM-Programm Multiplikationen oder Divisionen einsetzt, werden diese Operationen jeweils durch Unterprogramme simuliert, in denen als arithmetische Operationen nur Additionen und Subtraktionen verwendet werden. Es läßt sich zeigen, daß diese Unterprogramme so entworfen werden können, daß die logarithmischen Kosten zur Ausführung des jeweiligen Unterprogramms höchstens das Quadrat der logarithmischen Kosten der Anweisung betragen, die es nachbildet.

Die folgende Übersicht zeigt (in Ergänzung mit den Aussagen aus Kapitel 2.1) die Anzahl der Schritte (Zeitkomplexität), die Maschinenvariante *A* benötigt, um die Ausführung einer Rechnung zu simulieren, der auf Maschinenvariante *B* bei einem akzeptierten Eingabewort *w* der Länge *n* eine Schrittzahl (Anzahl der Überführungen bis zum Akzeptieren) von $T(n)$ benötigt. Dabei wird für das RAM-Modell das logarithmische Kostenkriterium angenommen.

	simulierende Maschine *A*		
Simulierte Maschine *B*	1-DTM	*k*-DTM	RAM
1-DTM	-	$O(T(n))$	$O(T(n)\log(T(n)))$
k-DTM	$O(T^2(n))$	-	$O(T(n)\log(T(n)))$
RAM	$O(T^3(n))$	$O(T^2(n))$	-

2.3 Programmiersprachen

In den meisten Anwendungen werden Algorithmen mit Hilfe der gängigen Programmiersprachen formuliert (siehe Kapitel 1.3). Es zeigt sich (siehe angegebene Literatur), daß man sehr einfache Programmiersprachen definieren kann, so daß man mit in diesen Sprachen formulierten Algorithmen Turingmaschinen simulieren kann. Umgekehrt kann man mit Turingmaschinen das Verhalten dieser Algorithmen nachbilden. Da eine Turingmaschine einen unendlich großen Speicher besitzt, muß man für Algorithmen in diesen Programmiersprachen unendlich viele Variablen zulassen bzw. voraussetzen, daß die Algorithmen auf Rechnern ablaufen, die einen unendlich großen Speicher besitzen. Dieses Prinzip wurde ja auch im RAM-Modell verwirklicht.

Eine Programmiersprache, deren Programme die Turingberechenbarkeit nachzubilden in der Lage sind, benötigt die Definition von:

- abzählbar vielen Variablen, die Werte aus **N** annehmen können (ganzzahlige und rationale Werte werden durch natürlichzahlige Werte nachgebildet, vgl. die Festpunktdarstellung von Zahlen, reelle Werte durch natürlichzahlige Werte approximiert, vgl. die Gleitpunktdarstellung von Zahlen)
- elementaren Anweisungen wie Wertzuweisungen an Variablen, einfachen arithmetischen Operationen (Addition, Subtraktion, Multiplikation, ganzzahlige Division) zwischen Variablen und Konstanten
- zusammengesetzten Anweisungen (Sequenz *anweisung*$_1$; *anweisung*$_2$;), Blockbildung (**BEGIN** *anweisung*$_1$; ...; *anweisung*$_n$ **END)**, bedingte Anweisungen (**IF** *bedingung* **THEN** *anweisung*$_1$ **ELSE** *anweisung*$_2$; hierbei ist *bedingung* ein Boolescher Ausdruck, der ein logisches Prädikat mit Variablen darstellt), Wiederholungsanweisungen (**WHILE** *bedingung* **DO** *anweisung*;)
- einfachen Ein/Ausgabeanweisungen (**READ** (*x*), **WRITE** (*x*)).

Auf eine formale Beschreibung dieses Ansatzes soll hier verzichtet werden.

Auch bei der Formulierung eines Algorithmus mit Hilfe einer Programmiersprache kann man nach der Zeit- und Raumkomplexität fragen. Dabei muß man wieder zwischen uniformen und logarithmischen Kostenkriterium unterscheiden.

Ein Programm *PROG* habe die Eingabe $x = [x_1, ..., x_n]$, die sich aus *n* Parametern (Formalparameter bei der Spezifikation des Programms, Aktualparameter bei Aufruf des Programms) zusammensetzt. Beispielsweise kann *x* ein Graph sein, der *n* Knoten besitzt. *PROG* berechne eine Ausgabe $y = [y_1, ..., y_m]$, die sich aus *m* Teilen zusammensetzt. Beispielsweise kann *y*

das Ergebnis der Berechnung einer Funktion f sein, d.h. $[y_1, ..., y_m] = f(x_1, ..., x_n)$. Oder *PROG* soll entscheiden, ob die Eingabe x eine spezifizierte Eigenschaft besitzt; dann ist $y \in \{\mathsf{TRUE}, \mathsf{FALSE}\}$. In der Regel wird die Zeit- und Raumkomplexität von *PROG* in Abhängigkeit der Eingabe $x = [x_1, ..., x_n]$ gemessen.

Bei der Berechnung der **uniformen Zeitkomplexität** von *PROG* bei Eingabe von x wird die Anzahl der durchlaufenen Anweisungen gezählt. Bei der Berechnung der **uniformen Raumkomplexität** von *PROG* bei Eingabe von x wird die Anzahl der benötigten Variablen gezählt.

Der Ansatz bietet sich immer dann an, wenn die so ermittelten Werte der Zeit- und Raumkomplexität nur von der Anzahl n der Komponenten der Eingabe $x = [x_1, ..., x_n]$ abhängen. Diese Situation liegt beispielsweise vor, wenn *PROG* die Aufgabe hat, die Komponenten der Eingabe (nach einem „einfachen“ Kriterium) zu sortieren.

Das folgende Beispiel zeigt jedoch, daß der Ansatz nicht immer adäquat ist. Die Pascal-Prozedur `zweier_potenz` berechnet bei Eingabe einer Zahl $n > 0$ den Wert $c = 2^{2^n} - 1$.

```
PROCEDURE zweier_potenz (    n : INTEGER;
                         VAR c : INTEGER);

VAR idx : INTEGER;
    p   : INTEGER;

BEGIN {zweier-potenz }
  idx := n;                 { Anweisung 1 }
  p   := 2;                 { Anweisung 2 }
  WHILE idx > 0 DO          { Anweisung 3 }
    BEGIN
      p   := p*p;           { Anweisung 4 }
      idx := idx - 1;       { Anweisung 5 }
    END;
  c := p - 1;               { Anweisung 6 }
END   { zweier-potenz };
```

Werden nur die Anzahl der ausgeführten Anweisungen gezählt, so ergibt sich bei Eingabe der Zahl $n > 0$ eine Zeitkomplexität der Ordnung $O(n)$ und eine Raumkomplexität der Ordnung $O(1)$. Das Ergebnis $c = 2^{2^n} - 1$ belegt jedoch 2^n viele binäre Einsen und benötigt zu seiner Erzeugung mindestens 2^n viele (elementare) Schritte. Vergrößert man die Eingabe n um eine Konstante k, d.h. betrachtet man die Eingabe $n + k$, so bleibt die Laufzeit in der Ordnung $O(n)$, während das Ergebnis um den *Faktor* 2^k größer wird.

Es bietet sich daher eine etwas sorgfältigere Definition der Zeit- und Raumkomplexität an. Diese wird hier aufgezeigt, wenn die Ein- und Ausgabe eines Programms aus numerischen Daten bzw. aus Daten besteht, die als numerische Daten dargestellt werden können (beispielsweise die Datentypen **BOOLEAN** oder **SET OF** ... in Pascal).

Wenn die Zeit- und Raumkomplexität nicht nur von der Anzahl *n* der Komponenten der Eingabe $x = [x_1, ..., x_n]$ abhängt, sondern wie im Beispiel der Prozedur `zweier_potenz` von den Zahlenwerten $x_1, ..., x_n$ selbst, wird folgender Ansatz gewählt.

Für eine ganze Zahl *z* sei die **Größe** $size(z) = |bin(z)|$ die Anzahl signifikanter Bits, die benötigt werden, um *z* im Binärsystem darzustellen. Für negative Werte von *z* wird die Komplementdarstellung gewählt. Dabei sei $size(0) = 1$. Es gilt also $size(z) = \begin{cases} \lfloor \log_2(|z|) \rfloor + 1 & \text{für } z \neq 0 \\ 1 & \text{für } z = 0 \end{cases}$ und $size(z) \in O(\log(|z|))$. Für $x = [x_1, ..., x_n]$ sei $size(x) = n \cdot \max\{ size(x_1), ..., size(x_n) \}$.

Der Wert $size(x)$ repräsentiert damit die Anzahl der Bits, um die Zahlenwerte darzustellen, die in *x* vorkommen.

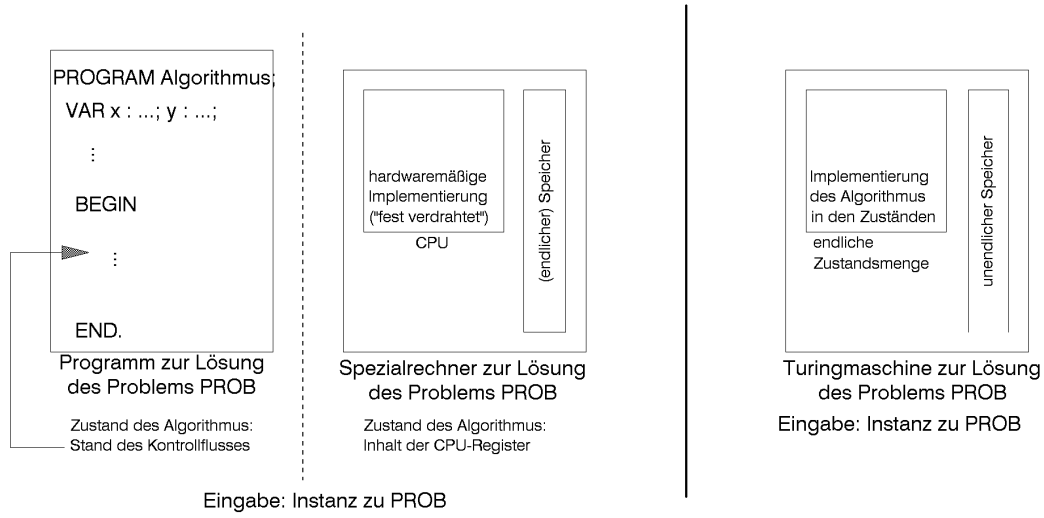
Die **logarithmische Zeitkomplexität** eines Programms *PROG* bei Eingabe von *x* ist die Anzahl der durchlaufenen Anweisungen, gemessen in Abhängigkeit von der so definierten Größe $size(x)$. Bei der Berechnung der **logarithmischen Raumkomplexität** von *PROG* bei Eingabe von *x* wird die Anzahl der Bits gezählt, um alle von *PROG* benötigten Variablen abzuspeichern; dieser Wert wird in Abhängigkeit von $size(x)$ genommen.

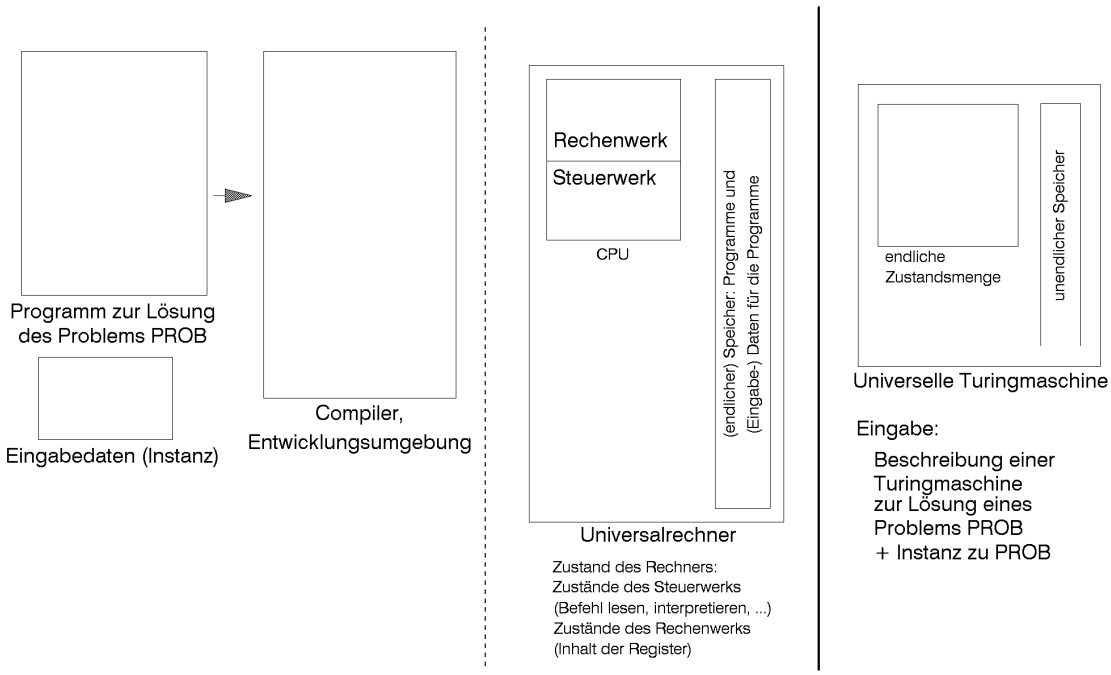
In obiger Pascal-Prozedur `zweier_potenz` gilt für die Eingabe *n* die Beziehung $k = size(n) = c \cdot \log_2(n)$; die Prozedur hat eine Laufzeit der Ordnung $O(n) = O\left(2^{\log_2(n)}\right) = O\left(2^{1/c} \cdot 2^{c \cdot \log_2(n)}\right) = O\left(2^k\right)$, also exponentielle Laufzeit, gemessen in der Größe der Eingabe. Seine Raumkomplexität wird durch die Anzahl der Bits bestimmt, um die Variable `p` zu speichern. Diese ist von der Ordnung $O\left(2^n\right) = O\left(2^{2^k}\right)$. Insgesamt gibt das logarithmische Kostenkriterium die Realität exakt wieder.

2.4 Universelle Turingmaschinen

Jede Turingmaschine und jeder in einer Programmiersprache formulierte Algorithmus ist zur Lösung eines spezifischen Problems entworfen. Dabei ist natürlich die Eingabe wechselnder Werte der Problemparameter möglich. Entsprechend könnte man in dieser Situation den Algorithmus hardwaremäßig implementieren und hätte dadurch einen „Spezialrechner“, der in der Lage ist, das spezifische Problem, für das er entworfen ist, zu lösen. Ein Computer wird

jedoch üblicherweise anders eingesetzt: ein Algorithmus wird in Form eines in einer Programmiersprache formulierten Quelltextes einem Compiler (etwa innerhalb einer Entwicklungsumgebung) vorgelegt. Das Programm wird im Computer in Maschinensprache übersetzt, so daß er in der Lage ist, das Programm „zu interpretieren“. Anschließend wird es von diesem Computer mit entsprechenden eingegebenen Problemparametern ausgeführt. Der Computer ist also in der Lage, nicht nur einen speziellen Algorithmus zur Lösung eines spezifischen Problems, sondern jede Art von Algorithmen auszuführen, solange sie syntaktisch korrekt formuliert sind.





Diese Art der Universalität ist auch im Turingmaschinen-Modell möglich. Im folgenden wird dazu eine **universelle Turingmaschine** *UTM* definiert. Diese erhält als Eingabe die Beschreibung einer Turingmaschine *TM* (ein Problemlösungsalgorithmus) und einen Eingabedatensatz *w* für *TM*. Die universelle Turingmaschine verhält sich dann genauso, wie sich *TM* bei Eingabe von *w* verhalten würde.

Im folgenden wird zunächst gezeigt, wie man die Beschreibung einer Turingmaschine aus ihrer formalen Definition erhält.

Eine deterministische *k*-Band-Turingmaschine $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ kann als endliches Wort über {0, 1} kodiert werden. Dazu wird angegeben, wie eine mögliche Kodierung von *TM* aussehen kann (die hier vorgestellte Kodierung ist natürlich nur eine von vielen möglichen):

Es sei # ein neues Zeichen, das in $Q \cup \Sigma$ nicht vorkommt. Das Zeichen # dient als syntaktischen Begrenzungssymbol innerhalb der Kodierung.

Die Zustandsmenge von *TM* sei $Q = \{q_0, .., q_{|Q|-1}\}$, ihr Arbeitsalphabet $\Sigma = \{a_0, .., a_{|\Sigma|-1}\}$. Man kann annehmen, daß der Startzustand q_0 und der akzeptierende Zustand $q_{accept} = q_{|Q|-1}$ ist. Das Blankzeichen *b* kann mit a_0 identifiziert werden. Die Anzahl der Zustände, die Anzahl der Elemente in Σ und die Anzahl der Bänder werden in einem Wort $w_0 \in \{0, 1, \#\}^*$ festgehalten:
$w_0 = \#bin(|Q|)\#bin(|\Sigma|)\#bin(k)\#\#$.

Hierbei bezeichnet wieder $bin(n)$ die Binärdarstellung von *n*.

Die Überführungsfunktion d habe *d* viele Zeilen. Die *t*- te Zeile laute:
$d(q_i, a_{i_1}, ..., a_{i_k}) = (q_j, (a_{j_1}, d_1), ..., (a_{j_k}, d_k))$.
Es sind $a_{i_l} \in \Sigma$, $a_{j_m} \in \Sigma$ und $d_i \in \{L, R, S\}$. Diese Zeile wird zunächst als Zeichenkette
$w_t = (bin(i)\#bin(i_1)\#...\#bin(i_k))(bin(j)\#(bin(j_1)\#d_1)\#...\#(bin(j_k)\#d_k))\#$
kodiert. Diese Zeichenkette ist ein Wort über dem Alphabet $\{0, 1, (,), \#, L, R, S\}$. Zu beachten ist, daß w_t sich öffnende und schließende Klammern enthält; der Ausdruck $bin(i)$ enthält keine Klammern, sondern ist eine Zeichenkette über $\{0, 1\}$.

Die gesamte Turingmaschine *TM* läßt sich durch Konkatination w_{TM} der Wörter w_0, w_1, ..., w_d kodieren: $w_{TM} = w_0 w_1 ... w_d$. Es ist $w_{TM} \in \{0, 1, (,), \#, L, R, S\}^*$. Die 8 Buchstaben dieses Alphabets werden buchstabenweise gemäß der Tabelle

Zeichen	Umsetzung	Zeichen	Umsetzung
0	000	#	100
1	001	*L*	101
(	010	*R*	110
)	011	*S*	111

umgesetzt. Das Resultat der Umsetzung von w_{TM} in eine Zeichenkette über {0, 1} wird mit $code(TM)$ bezeichnet.

Beispiel: Kodierung einer Turingmaschine

Gegeben sei die 1-DTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, b\}$, $I = \{0, 1\}$, $q_{accept} = q_2$ und d gemäß der folgenden Tabelle:

Überführungsfunktion $d(q_i, a_{i_1}, ..., a_{i_k}) = (q_j, (a_{j_1}, d_1), ..., (a_{j_k}, d_k))$	Zwischenkodierung $w_t = (bin(i)\#bin(i_1)\#...\#bin(i_k))$ $(bin(j)\#(bin(j_1)\#d_1)\#...\#(bin(j_k)\#d_k))$; entsprechend dem Wort über {0, 1}
$d(q_0,1) = (q_1, (0,R))$	$w_1 = (0\#1)(1\#(0\#R))\#$ 0100001000010110100011000100001001100110111100
$d(q_1,0) = (q_0, (1,R))$	$w_2 = (1\#0)(0\#(1\#R))\#$ 0100011000000110100001000100011001100110111100
$d(q_1,1) = (q_2, (0,R))$	$w_3 = (1\#1)(10\#(0\#R))\#$ 0100011000010110100010001000100001001100110111100
$d(q_1,b) = (q_1, (1,L))$	$w_4 = (0\#11)(1\#(1\#L))\#$ 0100011000010010110100011000100011001010110111100

Das Wort w_0 lautet: $w_0 = \#11\#11\#1\#\#$. Dieses Wort wird übersetzt in eine 0-1-Kodierung und ergibt 1000010011000010011001100100.

Die Turingmaschine *TM* ist also kodiert durch $code(TM)$ =
100001001100001001100110010001000010000101101000110001000010011001101110
001000110000001101000010001000110011001101110001000110000101101000100010
001000010011001101110001000110000100101101000110001000110010101101110 0.

Die Turingmaschine TM_{min} mit der kürzesten Kodierung ist die k-DTM $TM_{min} = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $k = 0$, $Q = \{q_0\}$, $\Sigma = \{b\}$ und $d = \varnothing$. Das dieser Turingmaschine entsprechende Wort w_0 lautet: $w_0 = \#1\#1\#0\#\#$, übersetzt in eine 0-1-Kodierung

10000110000110000010010 0. Da $d = \varnothing$ ist, stimmt $code(TM_{min})$ mit dieser 0-1-Folge bereits überein.

Zu jeder Turingmaschine *TM* gibt es also ein Wort $w \in \{0,1\}^*$, $w = code(TM)$, das *TM* kodiert. Offensichtlich ist $\{code(TM) \mid TM \text{ ist eine Turingmaschine}\} \subseteq \{0,1\}^*$.
Für unterschiedliche Turingmaschinen TM_1 und TM_2 gilt dabei $code(TM_1) \neq code(TM_2)$, d.h. die so definierte Abbildung *code* ist injektiv.

Andererseits kann man einem Wort $w \in \{0,1\}^*$ „ansehen“, ob es eine Turingmaschine kodiert. Nicht jedes Wort $w \in \{0,1\}^*$ kodiert eine Turingmaschine, und natürlich kann es vorkommen, daß die durch ein Wort $w \in \{0,1\}^*$ kodierte Turingmaschine wenig Sinnvolles leistet. Die oben informell beschriebenen Regeln der Kodierung einer Turingmaschine *TM* durch ein Wort $code(TM)$ erlauben ein (deterministisches) algorithmisches Verfahren der syntaktischen Analyse, das feststellt, ob ein Wort $w \in \{0,1\}^*$ die Kodierung einer Turingmaschine darstellt. Dieses Verfahren, auf dessen detaillierte Darstellung hier verzichtet werden soll, werde mit *VERIFIZIERE_TM* bezeichnet.

Für ein Wort $w \in \{0,1\}^*$ ist

$$VERIFIZIERE\_TM(w) = \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Turingmaschine darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Turingmaschine darstellt} \end{cases}$$

Falls ein Wort $w \in \{0,1\}^*$ eine Turingmaschine *TM* kodiert, d.h. $w = code(TM)$, dann bezeichnet man mit $TM = \overset{-1}{code}(w)$ **die durch *w* kodierte Turingmaschine** (diese Notation ist zulässig, da *code* injektiv ist). Wir schreiben dafür kürzer
$TM = K_w$.

Mit Hilfe der Kodierungen von Turingmaschinen kann man eine **lineare Ordnung auf der Menge der Turingmaschinen** definieren. Es sei $w \in \{0,1\}^*$. Für $i \in \mathbf{N}_{>0}$ heißt ***w* Kodierung der *i*-ten Turingmaschine**, falls gilt:

(i) $VERIFIZIERE\_TM(w) = \mathsf{TRUE}$

(ii) die Menge $\left\{ x \;\middle|\; \begin{array}{l} x \in \{0,1\}^* \text{ und } x \text{ liegt in der lexikographischen Ordnung vor } w \\ \text{und } VERIFIZIERE\_TM(x) = \mathsf{TRUE} \end{array} \right\}$ enthält genau $i-1$ viele Elemente.

Falls $w = code(TM)$ die Kodierung der *i*-ten Turingmaschine ist, dann heißt $TM = K_w$ die ***i*-te Turingmaschine**.

Der folgende Algorithmus ermittelt bei Eingabe einer Zahl $i \in \mathbf{N}_{>0}$ die Kodierung der *i*-ten Turingmaschine.

Eingabe: Eine natürliche Zahl $i \in \mathbf{N}_{>0}$

Verfahren: Aufruf der Funktion `Generiere_TM (`*i*`)`

Ausgabe: $w \in \{0,1\}^*$, *w* ist die Kodierung der *i*-ten Turingmaschine.

Die Funktion `Generiere_TM` wird in Pseudocode beschrieben.

```
FUNCTION Generiere_TM (i : INTEGER) : STRING;

VAR x : STRING;
    y : STRING;
    k : INTEGER;

BEGIN { Generiere_TM }
  x := '0';  { x∈{0,1}* }
  k := 0;

  WHILE k < i DO
     BEGIN
       IF VERIFIZIERE_TM(x)
       THEN BEGIN
              k := k + 1;
              y := x;
            END;
       x := Nachfolger von x in der lexikographischen Reihenfolge von {0,1}* ;
     END;

 Generiere_TM := y;
END   { Generiere_TM }
```

Der folgende Algorithmus ermittelt bei Eingabe von $w \in \{0,1\}^*$ die Nummer *i* in der oben definierten Ordnung auf der Menge der Turingmaschinen, falls *w* überhaupt eine Turingmaschine kodiert; ansonsten gibt er den Wert 0 aus.

Eingabe: $w \in \{0,1\}^*$

Verfahren: Aufruf der Funktion `Ordnung_TM (`*w*`)`

Ausgabe: $i > 0$, falls *w* die Kodierung der *i*-ten Turingmaschine ist,
$i = 0$, sonst.

Die Funktion `Ordnung_TM` wird in Pseudocode beschrieben.

```
FUNCTION Ordnung_TM (w : STRING) : INTEGER;

VAR x : STRING;
    k : INTEGER;

BEGIN { Ordnung_TM }
  IF NOT VERIFIZIERE_TM(w)
  THEN BEGIN
         Ordnung_TM := 0;
         Exit;
       END;

  x := '0';  { x∈{0,1}* }
  k := 1;

  WHILE NOT (x = w) DO
     BEGIN
       IF VERIFIZIERE_TM(x)
       THEN k := k + 1;
       x := Nachfolger von x in der lexikographischen Reihenfolge von {0,1}*;
     END;

 Ordnung_TM := k;
END   { Ordnung_TM }
```

Ein Wort $w \in \{0,1\}^*$ kann damit sowohl als die Kodierung der *i*-ten Turingmaschine als auch als ein Eingabewort für eine gegebene Turingmaschine oder auch als Binärzahl interpretiert werden. Die Nummer *i* kann aus *w* algorithmisch deterministisch ermittelt werden (falls *w* überhaupt eine Turingmaschine kodiert); ebenso kann die Kodierung der *i*-ten Turingmaschine erzeugt werden.

Satz 2.4-1:

Es gibt eine **universelle Turingmaschine** *UTM* mit Eingaben der Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$, so daß $z \in L(UTM)$ genau dann gilt, wenn $w = code(TM)$ für eine Turingmaschine *TM* ist, d.h. $TM = K_w$, und $u \in L(K_w)$ ist.

Die universelle Turingmaschine *UTM* simuliert das Verhalten von K_w auf *u*: Sie betrachtet *w* als die Turingmaschine K_w und verhält sich dann auf *u* so, wie sich K_w auf *u* verhält.

Beweis:

Zunächst prüft *UTM*, ob die Eingabe $z \in \{0,1,\#\}^*$ genau ein Zeichen # enthält. Ist dieses nicht der Fall, so wird *z* nicht akzeptiert. Hat *z* die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$, so überprüft *UTM*, ob *w* eine Turingmaschine kodiert (siehe Funktion *VERIFIZIERE_TM*(*w*)). Falls dieses nicht zutrifft, wird *z* nicht akzeptiert.

Andernfalls hat *w* die Form $w = 100v_1100v_2100v_3100100v$, wobei sich v_1, v_2 und v_3 ausschließlich aus der Konkatination von Zeichenfolgen 000 und 001 zusammensetzen und $v \in \{0,1\}^*$ ist. Nimmt man von v_1 jedes dritte Zeichen, so erhält man eine Binärzahl, die die Anzahl der Zustände von K_w angibt. Der Zustand mit der höchsten Nummer ist der akzeptierende Zustand, der Zustand mit der Nummer 0 der Anfangszustand. Entsprechend erhält man aus v_2 die Anzahl der Zeichen des Arbeitsalphabets und damit implizit das Arbeitsalphabet Σ und aus v_3 die Anzahl *k* der Bänder von K_w. In *v* ist die Überführungsfunktion von K_w kodiert.

UTM erzeugt jetzt auf einem weiteren Band, das als Konfigurations-Simulationsband bezeichnet werden soll, die Kodierung der Anfangskonfiguration von K_w mit Eingabewort *u*, d.h. die Kodierung von $K_0 = \left(q_0, \underbrace{(u,1),(e,1),...,(e,1)}_{k} \right)$. Hierbei kann ein ähnlicher Umsetzungsmechanismus wie zur Erzeugung der Kodierung einer Turingmaschine verwendet werden. Eine Konfiguration von K_w der Form $K = (q_t, (a_1, i_1), ..., (a_k, i_k))$, wobei q_t der *t*-te Zustand von K_w und $a_j \in \Sigma^*$, $i_j \geq 1$ für *j* = 1, ..., *k* ist, wird zunächst (gedanklich) umgesetzt in $(bin(t)\#(b_1\#bin(i_1))\#...\#(b_k\#bin(i_k)))$. Hierbei erhält man b_j aus a_j (für *j* = 1, ..., *k*), indem man jeden Buchstaben durch den Binärwert seiner Nummer in Σ, gefolgt vom Zeichen # ersetzt. Ist beispielsweise $a_j = aacca$ und sind *a* bzw. *c* die Zeichen in Σ mit den Nummern 1 bzw. 3, so ist $b_j = 1\#1\#11\#11\#1\#$. Die Zeichenfolge $(bin(t)\#(b_1\#bin(i_1))\#...\#(b_k\#bin(i_k)))$ wird mittels anfangs angegebener Tabelle in eine Folge über $\{0,1\}$ umgesetzt.

Ist K_w etwa eine 3-DTM mit $\Sigma = \{b, 0, 1, a\}$, wobei *b* das Blankzeichen (mit Nummer 0) bezeichnet und die Zeichen 0, 1 und *a* die Nummern 1 bis 3 tragen, so lautet die Anfangskonfiguration K_0 von K_w mit dem Eingabewort $u = 11001$:
$K_0 = (q_0, (11001, 1), (e, 1), (e, 1), (e, 1))$ bzw.
$(0\#(10\#10\#1\#1\#10\#\#1)\#(0\#\#1)\#(0\#\#1))$ und in eine 0-1-Folge kodiert:
010000100010001000100001000100001100001100001000100100001011100010000100100
001011100010000100100001011011.

Im Endstück *v* des Wortes $w = 100v_1100v_2100v_3100100v$ ist die Überführungsfunktion von K_w kodiert. *UTM* kann mit Hilfe der dort enthaltenen Informationen das Verhalten von K_w simulieren, indem *UTM* die Einträge auf dem Konfigurations-Simulationsband entsprechend der aus *v* gelesenen Überführungsfunktion schrittweise ändert. Auf dem Anfangsstück 010*m*100 des Konfigurations-Simulationsbands (hierbei besteht *m* ausschließlich aus Teilzeichenketten der Form 000 und 001) steht dabei jeweils die Kodierung des aktuellen Zustands von K_w. Endet die Simulation mit einem Wert *m*, der dem akzeptierenden Zustand von K_w entspricht, akzeptiert *UTM* die Eingabe *z*. ///

Die universelle Turingmaschine *UTM* kann so konstruiert werden, daß sie mit einem Band auskommt, da sich jede Turingmaschinen mit mehreren Bändern durch eine einbändige Turingmaschine simulieren läßt.

Das Konzept einer universellen Turingmaschine wird u.a. dazu verwendet, die Grenzen der Berechenbarkeit aufzuzeigen (Kapitel 3.2).

2.5 Nichtdeterminismus

Der Begriff „deterministisch“ in der Definition einer *k*-DTM drückt aus, daß die Nachfolgekonfiguration einer Konfiguration *K* in einer Berechnung der *k*-DTM eindeutig durch den in *K* vorkommenden Zustand *q*, die von den Schreib/Leseköpfen gerade gelesenen Zeichen a_1, ..., a_k, den (eindeutigen) Wert $d(q, a_1, ..., a_k)$ der Überführungsfunktion und die gegenwärtigen Positionen der Schreib/Leseköpfe bestimmt ist. In diesem Kapitel wird das Modell der nichtdeterministischen Turingmaschine eingeführt. Eine nichtdeterministische Turingmaschine unterscheidet sich von einer deterministischen Turingmaschine in der Definition der Überführungsfunktion d .

Eine **nichtdeterministische *k*-Band-Turingmaschine (*k*-NDTM)** *TM* ist definiert durch
$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$
mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Arbeitsalphabet**
3. $I \subseteq \Sigma$ ist eine endliche nichtleere Menge: das **Eingabealphabet**
4. $b \in \Sigma \setminus I$ ist das **Leerzeichen**
5. $q_0 \in Q$ ist der **Anfangszustand (**oder **Startzustand)**
6. $q_{accept} \in Q$ ist der **akzeptierende Zustand (Endzustand)**
7. $d : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}\left(Q \times (\Sigma \times \{L, R, S\})^k\right)$ ist eine partielle Funktion, die **Überführungsfunktion**; insbesondere ist $d(q, a_1, \ldots, a_k)$ für $q = q_{accept}$ nicht definiert; zu beachten ist, daß d für einige weitere Argumente eventuell nicht definiert ist.

Die Überführungsfunktion ordnet also jedem Argument $(q, a_1, \ldots, a_k)$ nicht einen einzigen (eindeutigen) Wert $(q', (b_1, d_1), \ldots, (b_k, d_k))$ zu, sondern eine *endliche Menge von Werten* der Form $(q', (b_1, d_1), \ldots, (b_k, d_k))$, von denen in einer Berechnung ein Wert genommen werden kann (natürlich kann diese endliche Menge auch aus einem einzigen Element bestehen).

Damit wird auch das Aussehen einer Berechnung einer nichtdeterministischen Turingmaschine *TM* neu definiert. Auch hierbei wird wieder der Begriff der **Konfiguration** verwendet, um den gegenwärtigen Gesamtzustand von *TM* zu beschreiben, d.h. den gegenwärtigen Zustand zusammen mit den Bandinhalten und den Positionen der Schreib/Leseköpfe:
$K = (q, (a_1, i_1), \ldots, (a_k, i_k))$ mit $q \in Q$, $a_j \in \Sigma^*$, $i_j \geq 1$ für j = 1, ..., k.

TM startet wie im deterministischen Fall im Anfangszustand mit einer **Anfangskonfiguration** $K_0 = (q_0, (w,1), (e,1), \ldots, (e,1))$. Ist *TM* in eine Konfiguration $K = (q, (a_1, i_1), \ldots, (a_k, i_k))$ gekommen und ist $d(q, a_1, \ldots, a_k)$ definiert, dann besteht $d(q, a_1, \ldots, a_k)$ aus einer endlichen Menge von Werten der Form $(q', (b_1, d_1), \ldots, (b_k, d_k))$, d.h.
$d(q, a_1, \ldots, a_k) = \{(q_1, (b_{11}, d_{11}), \ldots, (b_{1k}, d_{1k})), \ldots, (q_t, (b_{t1}, d_{t1}), \ldots, (b_{tk}, d_{tk}))\}$.
Als **Folgekonfiguration** von *K* wird eine der *t* möglichen Konfigurationen
$K_1 = (q_1, (b_{11}, i'_{11}), \ldots, (b_{1k}, i'_{1k}))$, ..., $K_t = (q_t, (b_{t1}, i'_{t1}), \ldots, (b_{tk}, i'_{tk}))$
genommen. K_i für i = 1, ..., t entsteht aus *K* dadurch, daß man wie im deterministischen Fall q durch q_i und a_1, ..., a_k durch b_{i1}, ..., b_{ik} ersetzt und die Köpfe so bewegt, wie es $(q_i, (b_{i1}, d_{i1}), \ldots, (b_{ik}, d_{ik}))$ in $d(q, a_1, \ldots, a_k)$ angibt. Welche der *t* möglichen Folgekonfigurationen auf die Konfiguration *K* folgt, wird nicht gesagt (wird nichtdeterministisch festgelegt); es wird „eine geeignete Folgekonfiguration“ genommen. Man schreibt dann

$K \Rightarrow K_i$.

Wie im deterministischen Fall heißt eine Konfiguration K_{accept}, die den akzeptierenden Zustand q_{accept} enthält, d.h. eine Konfiguration der Form
$K_{accept} = (q_{accept}, (a_1, i_1), ..., (a_k, i_k))$,
akzeptierende Konfiguration (**Endkonfiguration)**.

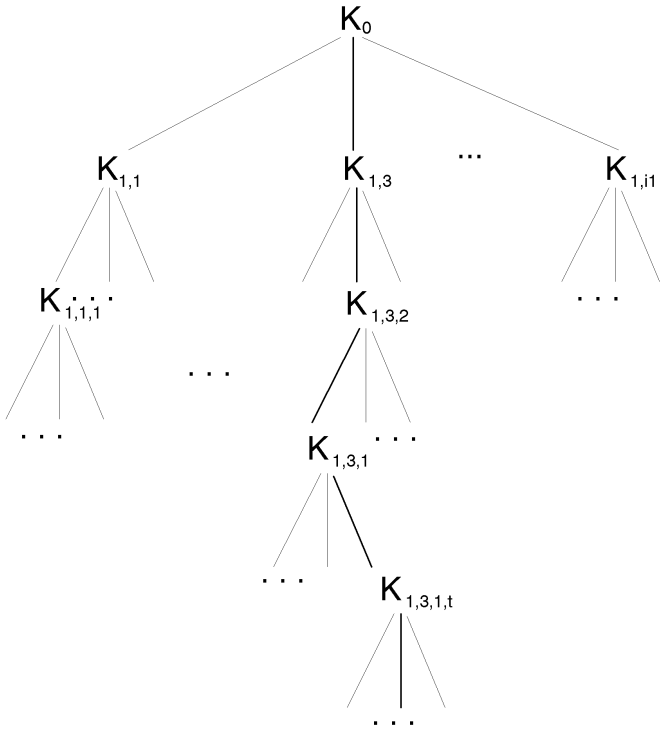
Wie im deterministischen Fall schreibt man $K \Rightarrow^m K'$ mit $m \in \mathbf{N}$, wenn K' aus K durch m Konfigurationsänderungen hervorgegangen ist, d.h. wenn es m Konfigurationen K_1, ..., K_m gibt mit
$K \Rightarrow K_1 \Rightarrow ... \Rightarrow K_m = K'$.
Für $m = 0$ ist dabei $K = K'$.

Die von einer k-NDTM TM **akzeptierte Sprache** ist die Menge
$L(TM) = \{ w \mid w \in I^* \text{ und } w \text{ wird von } TM \text{ akzeptiert} \}$
$= \{ w \mid w \in I^* \text{ und } (q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^* (q_{accept}, (a_1, i_1), ..., (a_k, i_k)) \}$.

Wie im deterministischen Fall kann man die Überführungsfunktion d modifizieren, indem man die Zustandsmenge Q um einen neuen Zustand q_{reject} erweitert und d um entsprechende Zeilen ergänzt (siehe Kapitel 2.1), so daß alle Berechnungen, die in einem Zustand $q \neq q_{accept}$, für die $d(q, ...)$ bisher nicht definiert ist, um eine Überführung in den Zustand q_{reject} fortgesetzt werden können. Für q_{reject} ist d nicht definiert.

Im *deterministischen* Fall kann man sich den Ablauf einer Berechnung als eine Folge
$K_0 \Rightarrow ... \Rightarrow K \Rightarrow K' \Rightarrow ...$
vorstellen, wobei jeder Schritt $K \Rightarrow K'$ eindeutig durch die Überführungsfunktion d feststeht. Im *nichtdeterministischen* Fall hat man in einem Schritt $K \Rightarrow K'$ eventuell mehrere Alternativen, so daß sich eine Berechnung hier als ein *Pfad* durch einen „Berechnungsbaum" darstellt. Jeder Knoten dieses Berechnungsbaums ist mit einer Konfiguration markiert. Die Wurzel ist mit einer Anfangskonfiguration $K_0 = (q_0, (w,1), (e,1), ..., (e,1))$ mit einem Wort $w \in I^*$ markiert. Ist die Konfiguration $K = (q, (a_1, i_1), ..., (a_k, i_k))$ die Markierung eines Knotens und kann hier der Eintrag $d(q, a_1, ..., a_k)$ der Überführungsfunktion angewendet werden, etwa
$d(q, a_1, ..., a_k) = \{ (q_1, (b_{11}, d_{11}), ..., (b_{1k}, d_{1k})), ..., (q_t, (b_{t1}, d_{t1}), ..., (b_{tk}, d_{tk})) \}$, dann enthält dieser Knoten des Berechnungsbaums t direkte Nachfolger, die mit den sich ergebenden Nachfolge-

konfigurationen $K_1 = (q_1, (b_{11}, i'_{11}), ..., (b_{1k}, i'_{1k}))$, ..., $K_t = (q_t, (b_{t1}, i'_{t1}), ..., (b_{tk}, i'_{tk}))$ markiert sind. Führt einer der Pfade von einer Anfangskonfiguration K_0 zu einer Endkonfiguration K_{accept}, so wird das in der Anfangskonfiguration stehende Wort *w* akzeptiert.



Das Partitionenproblem mit ganzzahligen Eingabewerten

Instanz: $I = \{a_1, ..., a_n\}$
I ist eine Menge von *n* natürlichen Zahlen.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
Entscheidung „nein“ sonst.

Offensichtlich lautet bei einer Instanz $I = \{a_1, ..., a_n\}$ mit ungeradem $B = \sum_{i=1}^{n} a_i$ die Entscheidung „nein“. Daher kann *B* als gerade vorausgesetzt werden.

Zunächst wird eine 3-NDTM *TM* angegeben, die dieses Problem löst, wenn die Eingaben in „unärer“ Kodierung eingegeben werden: Eine Eingabeinstanz $I = \{a_1, a_2, ..., a_n\}$ wird dabei als Wort $w = 10^{a_1}10^{a_2} ...10^{a_n}$ kodiert.

Die 3-NDTM $TM = (\{q_0, ..., q_5\}, \{0, 1, b, \$\}, \{0, 1\}, d, b, q_0, q_5)$ arbeitet folgendermaßen:

1. Das Eingabewort wird von links nach rechts gelesen. Jedesmal, wenn eine Folge 10^{a_i} erreicht wird, wird die Folge der Nullen entweder auf das 2. oder das 3. Band kopiert
2. Wenn das Ende des Eingabeworts erreicht ist, wird geprüft, ob auf dem 2. und 3. Band die gleiche Anzahl von Nullen steht; dieses geschieht durch simultanes Vorrücken der Köpfe nach links.

Die Überführungsfunktion d wird durch folgende Tabelle gegeben.

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf			neuer Zustand	neues Symbol, Kopfbewegung auf		
	Band 1	Band 2	Band 3		Band 1	Band 2	Band 3
q_0	1	b	b	q_1	1, S	\$, R	\$, R
q_1	1	b	b	q_2	1, R	b, S	b, S
				q_3	1, R	b, S	b, S
q_2	0	b	b	q_2	0, R	0, R	b, S
	1	b	b	q_1	1, S	b, S	b, S
	b	b	b	q_4	b, S	b, L	b, L
q_3	0	b	b	q_3	0, R	b, S	0, R
	1	b	b	q_1	1, S	b, S	b, S
	b	b	b	q_4	b, S	b, L	b, L
q_4	b	0	0	q_4	b, S	0, L	0, L
	b	\$	\$	q_5	b, S	\$, S	\$, S

Nichtdeterministische Strategien können den Ablauf von Berechnungen vereinfachen. Als Beispiel werde die Sprache

$$L = \{x\#y \mid x \in \{0,1\}^*, y \in \{0,1\}^*, x \neq e, x \neq y\}$$

betrachtet. Es sei $w \in \{0,1\}^*$. Eine deterministische Turingmaschine würde bei Eingabe von w zunächst an der Position des Zeichens # feststellen, wo y beginnt, und dann x und y buchstabenweise auf einen Unterschied hin vergleichen (falls w kein oder mehr als ein Zeichen # enthält, würde w nicht akzeptiert werden). Eine nichtdeterministische Turingmaschine *NTM* würde ebenfalls (deterministisch) zunächst prüfen, ob w genau ein Zeichen # enthält. Ist dieses nicht der Fall, wird w nicht akzeptiert. Ist $w = x\#y$ und besteht x aus n Buchstaben und y aus m Buchstaben, etwa $x = x_1 ... x_n$ und $y = y_1 ... y_m$, so überprüft *NTM* (deterministisch), ob $n \neq m$ gilt (in diesem Fall wird w akzeptiert). Andernfalls erzeugt NTM auf einem seiner Arbeitsbänder *nichtdeterministisch* den Wert $bin(i)$ mit $i \leq n = m$. Anschließend überprüft *NTM*

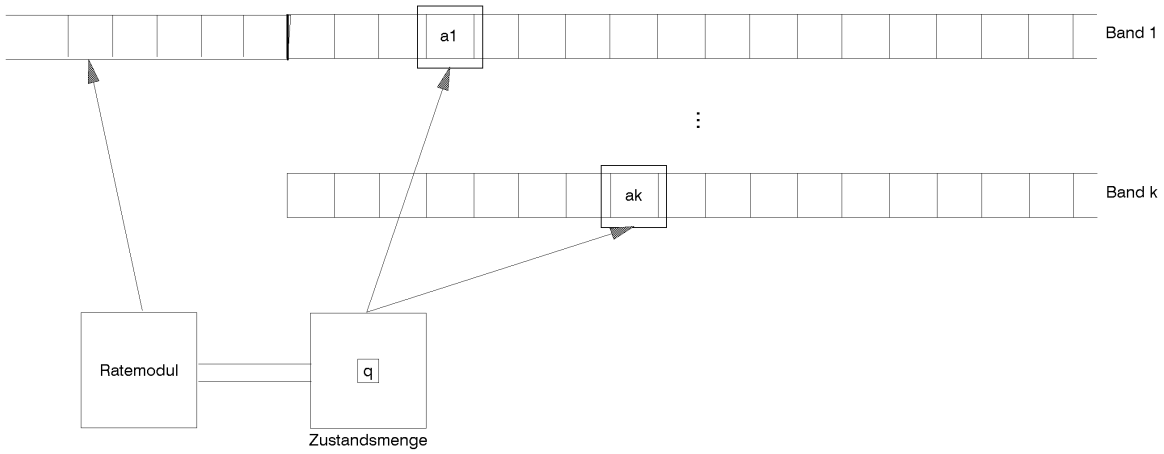
die Buchstaben x_i und y_i. Das Wort *w* wird genau dann akzeptiert, wenn $x_i \neq y_i$ ist. Für die nichtdeterministische Erzeugung von $bin(i)$ muß in die entsprechende Zelle des Arbeitsbands entweder 0 oder 1 geschrieben werden. *NTM* „rät (nichtdeterministisch)“ die Position, an der sich *x* und *y* unterscheiden und verifiziert anschließend die Richtigkeit des Ratevorgangs. Der geratene Wert $bin(i)$ kann auch als Beweis (Zertifikat) dafür angesehen werden, daß $x \neq y$ bzw. $w \in L$ gilt.

Die in diesem Beispiel beschriebene Arbeitsweise ist für eine nichtdeterministische Turingmaschine typisch:

Die nichtdeterministische Turingmaschine *NTM* sei konzipiert , um die Menge $L \subseteq \Sigma^*$ zu akzeptieren. Bei einer Eingabe $x \in \Sigma^*$ **rät** *NTM* **auf nichtdeterministische** Weise einen **Beweis** *B* (ein **Zertifikat**) dafür, daß $x \in L$ gilt. Der Beweis ist eine Zeichenkette über einem endlichen Alphabet Σ_0, d.h. $B \in \Sigma_0^*$. Anschließend **verifiziert** *NTM* **den Beweis**.

Eine nichtdeterministische Berechnung kann auch so organisiert werden, daß zunächst alle „nichtdeterministischen Schritte“ ausgeführt werden und anschließend nur noch deterministische Schritte erfolgen. Dieser Ansatz führt auf ein **Nichtstandardmodell** der NDTM:

Eine nichtdeterministische *k*-Band-Turingmaschine (*k*-NDTM) *TM* ist definiert durch $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept} \right)$. Die Überführungsfunktion ist eine partielle Abbildung $d : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}\left(Q \times \left(\Sigma \times \{L, R, S\}\right)^k\right)$, die sich in zwei Teile d_1 und d_2 ($d = d_1 \cup d_2$) zerlegen läßt. Alle nichtdeterministischen Teile von d sind in d_1 zusammengefaßt, d_2 besteht ausschließlich aus deterministischen Teilen. Das Eingabeband (1. Band) wird nach links abzählbar unendlich erweitert wird; die Zellen werden mit 0, -1, -2, ... numeriert. Zusätzlich verfügt *TM* über ein „Ratemodul“.



Die Arbeitsweise von *TM* verläuft in zwei Phasen. Ein Eingabewort $w = a_1 \ldots a_n,\ w \in I^*$, wird auf das 1. Band in die Zellen mit Nummern 1, ..., *n* eingegeben (allen anderen Zellen enthalten das Leerzeichen, die Schreib/Leseköpfe stehen auf den jeweiligen Bändern über der ersten Zelle, der Schreibkopf des Ratemoduls steht über der Zelle mit Nummer -1). Nun beginnt Phase 1. Das Ratemodul steuert getaktet den Schreibkopf: in jedem Takt wird ein Symbol aus Σ auf das 1. Band geschrieben und der Schreibkopf um eine Zelle nach links bewegt, oder der Ratemodul stoppt und wird „inaktiv“. Dieser Vorgang wird durch den nichtdeterministischen Teil d_1 von d gesteuert. Die so vom Ratemodul auf dem 1. Band erzeugte Zeichenkette wird auch als **Beweis (Zertifikat)** bezeichnet. *TM* befindet sich im Anfangszustand q_0. Eventuell stoppt das Ratemodul nicht, so daß die Turingmaschine bei der entsprechenden Eingabe nicht anhält. Sobald das Ratemodul stoppt und *TM* in den Zustand q_0 geht, beginnt Phase 2. *TM* verhält sich deterministisch, gesteuert durch den deterministischen Teil d_2 von d, wie eine „normale“ *k*-DTM, wobei die Zeichenkette, die das Ratemodul in Phase 1 erzeugt hat, in die Berechnung einbezogen wird.

Phase 2 kann auch als **Verifikationsphase** von TM bezeichnet werden. Es wird nämlich die „Brauchbarkeit“ der in Phase 1 erzeugten („geratenen“) Zeichenkette, der in Phase 1 erzeugte Beweis, für die Berechnung verifiziert.

Die Zeitkomplexität und die Platzkomplexität (im schlechtesten Fall, worst case) einer nichtdeterministischen Turingmaschine wird ähnlich wie im deterministischen Fall definiert:

Für eine *k*-NDTM $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept} \right)$ und eine Eingabe $w \in L(TM)$ gelte
$(q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^m K_{accept}$
mit einer Endkonfiguration K_{accept}. Hierbei sei *m* der *kleinste* Wert, so daß eine Endkonfiguration erreicht wird. Dann wird durch $t_{TM}(w) = m$ eine partielle Funktion $t_{TM} : \Sigma^* \to \mathbf{N}$ definiert, die angibt, wieviele Überführungen *TM in der kürzesten Berechnung* macht, um *w* zu akzeptieren (eventuell ist $t_{TM}(w)$ nicht definiert, nämlich dann, wenn die Eingabe von *w* nicht auf eine Endkonfiguration führt).

Die **Zeitkomplexität** von *TM* (**im schlechtesten Fall, worst case**) wird definiert durch die partielle Funktion $T_{TM} : \mathbf{N} \to \mathbf{N}$ mit
$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \le n \}$.

Entsprechend kann man die **Platzkomplexität** von *TM* (**im schlechtesten Fall, worst case**) $S_{TM}(n)$ als den maximalen Abstand vom linken Ende eines Bandes definieren, den ein Schreib/Lesekopf bei der Akzeptanz eines Worts $w \in L(TM)$ mit $|w| \le n$ *mindestens* erreicht.

Beispielsweise ist die oben angegebene Turingmaschine zur Lösung des Partitionenproblems (2*n* + 2)-zeitbeschränkt und (*n* + 1)-raumbeschränkt.

Nichtdeterministisches Verhalten läßt sich deterministisch simulieren:

Satz 2.5-1:
Falls *L* von einer *k*-NDTM *TM* akzeptiert wird, dann gibt es eine k'-DTM TM' mit $L(TM') = L(TM)$. Man kann TM' so konstruieren, daß gilt:
Falls TM $T(n)$-zeitbeschränkt ist mit $T(n) \ge n$, dann ist TM' $O\left(T(n) \cdot d^{T(n)}\right)$-zeitbeschränkt mit einer Konstanten $d > 0$.

Beweis:

Es sei TM eine *k*-NDTM, die $T(n)$-zeitbeschränkt mit $T(n) \ge n$ ist. Für jedes Wort $w \in L(TM)$ mit $|w| = n$ gibt es also eine Folge von Konfigurationen, die von einer Anfangskonfiguration K_0 mit *w* zu einer akzeptierenden Konfiguration K_{accept} führt und deren Länge $\le T(n)$ ist.
Jede Zeile der Tabelle, mit der die Überführungsfunktion d von *TM* gegeben wird (Überführungstabelle), hat die Form

$d(q,a_1,...,a_k)=\{(q_1,(b_{11},d_{11}),...,(b_{1k},d_{1k})),...,(q_t,(b_{t1},d_{t1}),...,(b_{tk},d_{tk}))\}$

(hier stehen *t* „Teilzeilen“). Wenn in einer Konfigurationenfolge $d(q,a_1,...,a_k)$ angewendet wird, gibt es *t* mögliche Folgekonfigurationen. Der maximale Wert *t* an Teilzeilen in einer Zeile der Überführungstabelle sei *d*. In einer Berechnung von *TM* gibt es für eine Konfiguration maximal *d* mögliche Folgekonfigurationen. Es sei $D=\{0,1,...,d-1\}$. Dann kann man eine von *TM* ausgeführte Berechnung bzw. die dabei durchlaufene Konfigurationsfolge der Länge *l* durch ein Wort $x=d_1...d_l$ über *D* beschreiben: d_i gibt an, daß in der *i*-ten Überführung die (d_i+1)-te Alternative in der entsprechenden Zeile der Überführungstabelle verwendet wird. Eventuell beschreibt nicht jedes Wort über *D* gültige Konfigurationenfolgen von *TM*.

Eine Simulation des Verhaltens von *TM* durch eine deterministische Turingmaschine *TM′* kann folgendermaßen durchgeführt werden. Eine Eingabe $w\in I^*$ wird auf einem Band von *TM′* zwischengespeichert (es wird eine Kopie angelegt). *TM′* erzeugt ein Wort $x\in D^*$ mit $|x|=l\le T(n)$, $x=d_1...d_l$, falls es noch ein nicht bereits erzeugtes Wort dieser Art gibt. Insgesamt können die Wörter in lexikographischer Reihenfolge erzeugt werden. Dann kopiert *TM′* das Eingabewort *w* aus dem Zwischenspeicher auf das Eingabeband von *TM* und simuliert auf deterministische Weise das Verhalten von *TM* für *l* Schritte, wobei in der *i*-ten Überführung die (d_i+1)-te Alternative in der entsprechenden Zeile der Überführungstabelle verwendet wird. Falls die Simulation auf den akzeptierenden Zustand führt, wird das Wort *w* akzeptiert, ansonsten wird das nächste Wort $x\in D^*$ erzeugt und die Simulation erneut ausgeführt.

Um ein Wort $x\in D^*$ mit $|x|=l\le T(n)$ zu erzeugen, benötigt man $O(l)$ viele Schritte (entsprechend der Addition einer 1 auf eine Zahl mit *l* Stellen). Anschließend wird *w* in $O(n)$ vielen Schritten aus dem Zwischenspeicher auf das Eingabeband von *TM* kopiert. Das Verhalten von *TM* wird dann für *l* viele Schritte simuliert. Es gibt d^l viele Möglichkeiten für $x\in D^*$ mit $|x|=l$. Insgesamt ergibt sich ein zeitlicher Aufwand der Größe

$$T'(n)=\sum_{l=0}^{T(n)}(c_1\cdot l+c_2\cdot n+c_3\cdot l)\cdot d^l$$

(der erste Term unter dem Summenzeichen beschreibt den zeitlichen Aufwand, um *x* zu erzeugen, der zweite Term, um *w* aus dem Zwischenspeicher auf das Eingabeband von *TM* zu kopieren, der dritte Term, um *l* Schritte von *TM* zu simulieren). Es gilt

$$T'(n)\le c\left(\sum_{l=0}^{T(n)}\left(l\cdot d^l+n\cdot\sum_{l=0}^{T(n)}d^l\right)\right)=c\left(\frac{d-(T(n)+1)d^{T(n)+1}+T(n)d^{T(n)+2}}{(d-1)^2}+n\frac{d^{T(n)+1}-1}{d-1}\right)$$ mit einer geeigneten Konstanten *c*. Der Ausdruck rechts ist von der Ordnung $O(T(n)\cdot d^{T(n)})$; hierbei wurde $n\le T(n)$ verwendet. ///

Die deterministische Simulation der *k*-NDTM *TM* erfolgt also zum Preis einer exponentiellen Steigerung des Laufzeitverhaltens, denn $O\left(T(n)\cdot d^{T(n)}\right)\subseteq O\left(c^{T(n)}\right)$ mit einer Konstanten $c > 0$.

Bisher ist keine nicht-triviale untere Schranke für die Simulation einer NDTM durch eine DTM bekannt. Insbesondere ist nicht bekannt, ob eine NDTM, deren Laufzeit durch ein Polynom begrenzt ist, durch eine DTM simuliert werden kann, deren Laufzeit ebenfalls ein Polynom ist (eventuell von einem sehr viel höheren Grad).

Anders verhält es sich bei Betrachtung der Platzkomplexität:

Eine Funktion $S:\mathbf{N}\to\mathbf{N}$ heißt **platz-konstruierbar**, wenn es eine *k*-DTM *TM* gibt, die bei Eingabe eines Wortes der Länge *n* ein spezielles Symbol in die $S(n)$-te Zelle eines ihrer Bänder schreibt, ohne jeweils mehr als $S(n)$ viele Zellen auf allen Bändern zu verwenden.

Satz 2.5-2:

Ist *TM* eine *k*-NDTM mit einer platz-konstruierbaren Speicherplatzkomplexität $S(n)$, dann gibt es eine k'-DTM TM' mit einer Speicherplatzkomplexität der Ordnung $O\left(S^2(n)\right)$ und $L(TM')=L(TM)$.

Beweis:

Es sei $TM=\left(Q,\Sigma,I,d,b,q_0,q_{accept}\right)$ eine *k*-NDTM mit einer platz-konstruierbaren Speicherplatzkomplexität $S(n)$. Es wird ein deterministischer Algorithmus, d.h. eine deterministische k'-DTM TM', angegeben, der das Verhalten von *TM* bei Eingabe eines Wortes *w* mit $|w|=n$ simuliert und dessen Speicherplatzbedarf durch einen Wert der Größenordnung $O\left(S^2(n)\right)$ beschränkt ist. Es sei $K=\left(q,(a_1,i_1),...,(a_k,i_k)\right)$ eine Konfiguration in einer Berechnung von *TM* bei Eingabe von *w*. Wegen $|a_j|\le S(n)$ und $1\le i_j\le S(n)$ für $j = 1, ..., k$ ist die Anzahl verschiedener Konfigurationen durch $|Q|\cdot|\Sigma|^{k\cdot S(n)}\cdot\left(S(n)\right)^k\le c^{S(n)}$ mit einer Konstanten $c > 0$ begrenzt. Falls in einer Berechnung von *TM* also $K_1\Rightarrow^* K_2$ gilt, dann kann man annehmen, daß dabei höchstens $c^{S(n)}$ viele Überführungen vorkommen. Es gilt: $K_1\Rightarrow^* K_2$ in höchstens *i* Schritten genau dann, wenn es eine Konfiguration K_3 gibt, so daß $K_1\Rightarrow^* K_3$ in höchstens $\lceil i/2\rceil$ vielen Schritten und $K_3\Rightarrow^* K_2$ in höchstens $\lfloor i/2\rfloor$ vielen Schritten abläuft. Für K_3 kommen nur $c^{S(n)}$ viele Möglichkeiten in Frage. Diese Überlegung führt auf folgenden Algorithmus: