3 Modellierung und Transformationen

In diesem Abschnitt überlegen wir zunächst allgemein, wie man die Semantik von randomisierten Algorithmen beschreiben kann. Hierzu benutzen wir eine randomisierte Version der Registermaschine. Diese Modellierung erlaubt es, den Wahrscheinlichkeitsraum präzise zu spezifizieren, auf dem unsere Aussagen über Fehlerwahrscheinlichkeiten und Erwartungswerte für Laufzeiten u. ä. beruhen. Wir stellen weiter allgemeine Techniken bereit, um die Fehlerwahrscheinlichkeit zu verringern und im Fall von fehlerbehafteten randomisierten Algorithmen worst-case-Laufzeitschranken (anstelle von Erwartungswerten) herzustellen. Zuletzt diskutieren wir den Zusammenhang zwischen fehlerfreien und „selbst-verifizierenden“ randomisierten Algorithmen.

3.1 Modell, Grundbegriffe

Wir gehen vom Modell der *Registermaschine* (RAM) aus, wie es in Vorlesungen zu „Berechenbarkeit und Komplexitätstheorie“ besprochen wird. (Zur Auffrischung: Anhang A.) Eine solche Registermaschine wird durch ein Programm $(B_i)_{0\le i<l}$ spezifiziert, wobei die Befehlszeilen B_i aus dem Befehlsvorrat des RAM-Modells stammen (Lade- und Speicherbefehle, arithmetische Befehle, Sprungbefehle). Wenn wir eine Eingabe in passender Form in einige Register schreiben und M mit Befehlszeile 0 starten, ergibt sich in eindeutiger Weise eine Berechnung. Falls diese hält, kann das Resultat aus den Registerinhalten abgelesen werden.

Wir wollen dies noch etwas formaler fassen. Die Menge $\mathcal{K}$ der Konfigurationen (die gar nicht von M abhängt) hat als Elemente Paare (z,α), wobei $z \in \mathbb{N}$ (Befehlszählerstand) und $\alpha\colon \mathbb{N} \to \mathbb{N}$ eine Funktion ist, die zu i den Registerinhalt $\alpha(i) = \langle R_i\rangle$ angibt. Wir verlangen, dass $\alpha(i) = 0$ für „fast alle i“, d. h. dass $\alpha(i) \neq 0$ nur für endlich viele i gilt.

Eine *Startkonfiguration* $k = (z,\alpha)$ ist eine Konfiguration mit $z = 0$ und $\alpha(i) = 0$ für alle $i > 0$ außer ungeraden $i < 2 \cdot \alpha(0)$. Wenn $k = (z,\alpha) \in \mathcal{K}$ eine Konfiguration ist, dann ist k entweder eine *Haltekonfiguration* (nämlich wenn $z \ge l$ gilt, also kein nächster Schritt definiert ist) oder es kann Befehlszeile B_z ausgeführt werden, was zu einer neuen Konfiguration k' führt. Wir schreiben in diesem Fall: $k \vdash_M k'$, und sagen,

dass k' die eindeutig bestimmte Nachfolgekonfiguration von k ist. Wenn man in einer Startkonfiguration k_0 beginnt, ergibt sich so eine Konfigurationenfolge

$$k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots ,$$

die in einer Haltekonfiguration enden oder unendlich lange weiterlaufen kann.

Wir wollen die Ein-/Ausgabekonventionen im Vergleich zum Anhang, wo nur n-Tupel von Zahlen als Inputs zugelassen sind, ein wenig verändern: Ein *Algorithmus* $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ besteht aus einer Registermaschine $M = M_{\mathcal{A}}$ (also einem Programm) und zusätzlich noch einer Inputmenge $I = I_{\mathcal{A}}$ und einer Outputmenge $Z = Z_{\mathcal{A}}$, sowie einer Inputfunktion $\text{IN} = \text{IN}_{\mathcal{A}}$ mit $\text{IN}\colon I \to \mathcal{K}$, die einen Input $x \in I$ in eine Startkonfiguration $\text{IN}(x) \in \mathcal{K}$ transformiert, und einer Outputfunktion $\text{OUT} = \text{OUT}_{\mathcal{A}}$ mit $\text{OUT}\colon \{k \in \mathcal{K} \mid k \text{ Haltekonfiguration}\} \to Z$, die aus einer Haltekonfiguration k ein Ergebnis $\text{OUT}(k) \in Z$ extrahiert. (Die Funktionen IN und OUT sollten „sehr einfache“ Kodierungs- und Dekodierungsfunktionen sein und keine Berechnung „verstecken“.) Um $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ auf x auszuführen, startet man M auf $k_0 = \text{IN}(x)$, und betrachtet die davon erzeugte Konfigurationenfolge $k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots$. Falls diese mit einer Haltekonfiguration k_t endet, ist

$$\mathcal{A}(x) := \text{OUT}(k_t) \in Z$$

das Resultat.

Die Churchsche These (siehe „Automaten, Sprachen und Komplexität (ASK)“) besagt unter anderem, dass jeder Algorithmus in ein RAM-Programm transformiert werden kann; in diesem Sinn ist es also gerechtfertigt, RAM-Programme zusammen mit Ein-/Ausgabekonventionen mit Algorithmen gleichzusetzen. Unter gewissen Einschränkungen an die Bitlänge der in den Registern zu speichernden Zahlen kann man auch sagen, dass die Rechenzeit eines RAM-Algorithmus dem Standard-Rechenzeitbegriff aus der Algorithmik ziemlich genau entspricht.

Wir wollen nun das RAM-Modell um den Aspekt „Randomisierung“ erweitern. Hierzu führen wir einen weiteren Befehl ein:

RANDOM

Dieser Befehl soll (intuitiv gesehen) folgenden Effekt haben: Wenn in R_0 eine Zahl $r = \langle R_0 \rangle \geq 2$ steht, so wird eine Zahl s aus der Menge $\{0, 1, \ldots, r-1\}$ zufällig (gemäß

der uniformen Verteilung) gewählt und nach R_0 geschrieben. Falls in R_0 die Zahl 0 oder 1 steht, wird 0 nach R_0 geschrieben. Der Befehlszähler wird um 1 erhöht.

Registermaschinen, die diesen neuen Befehl zur Verfügung haben, heißen „***randomisierte Registermaschinen* (*RRAM*)**“. In Anlehnung an die Churchsche These wollen wir eine RRAM M zusammen mit Ein-/Ausgabekonventionen einen **(*sequentiellen*) *randomisierten Algorithmus*** nennen.

Beispiel 3.1.1
Das folgende Programm in Pseudocode hat keinen Input und kann nur die Ausgabe „1“ erzeugen:

$R_0 \leftarrow 2$
RANDOM
while $R_0 = 0$ **do**
    $R_0 \leftarrow 2$; RANDOM
return R_0

Als RRAM-Programm:

Zeile	Befehl	Kommentar
0	$R_0 \leftarrow 2$	
1	RANDOM	0 oder 1 in R_0
2	if $R_0 > 0$ goto 6	Ende mit $\langle R_0 \rangle = 1$
3	$R_0 \leftarrow 2$	
4	RANDOM	0 oder 1 in R_0
5	goto 2	
6	$R_1 \leftarrow R_0$	Resultatformat
7	$R_0 \leftarrow 1$	herstellen

Beispiel 3.1.2
Das folgende Programm in Pseudocode berechnet wenigstens eine eventuell interessante Ausgabe. Der Input ist $s \geq 0$, was die Anzahl der Seiten eines fairen „Würfels“ sein soll. Die Seiten heißen $0, 1, \ldots, s-1$. Der Algorithmus soll wiederholt würfeln, bis eine 0 erscheint, und die Anzahl der ausgeführten Würfe ausgeben. Bei Eingabe 0 wird 0 ausgegeben.

Input: $s \in \mathbb{N}$.
if $s = 0$ **then return** 0
count $\leftarrow 1$
$R_0 \leftarrow s$
RANDOM
while $R_0 > 0$ **do**
    count $\leftarrow$ count $+ 1$
    $R_0 \leftarrow s$
    RANDOM
return count.

Die Ausgabe ist $\mathcal{A}(s)$, eine Zufallsvariable. Der Zufall „steckt“ dabei nicht in s (das ist durch die Eingabe festgelegt), sondern in den Zufallsexperimenten, die der Algorithmus ausführt.

Wenn wir diesen Algorithmus als RRAM-Programm darstellen wollen, sorgen wir dafür, dass die IN-Funktion aus einer Eingabe $x = s$ das richtige Eingabeformat herstellt (Befehlszähler auf 0, 1 nach R_0, s nach R_1, sonstige Register auf 0) und dass die OUT-Funktion die Ausgabe, die am Ende in R_1 bzw. $\alpha(1)$ steht, ausgibt.

Zeile	Befehl	Kommentar
0	if $R_1 = 0$ goto 12	$s = 0$ liefert Ausgabe 0
1	$R_4 \leftarrow 1$	Konstante 1
2	$R_2 \leftarrow 1$	Schrittzähler
3	$R_0 \leftarrow R_1$	s nach R_0 kopieren
4	RANDOM	würfeln
5	if $R_0 = 0$ goto 10	Ende wenn Ergebnis 0
6	$R_2 \leftarrow R_2 + R_4$	Zähler erhöhen
7	$R_0 \leftarrow R_1$	s nach R_0 kopieren
8	RANDOM	würfeln
9	goto 5	Schleifenanfang
10	$R_1 \leftarrow R_2$	Zählerstand nach R_1
11	$R_0 \leftarrow 1$	

Bis auf nicht so wesentliche technische Einzelheiten (irrationale Wahrscheinlichkeiten oder rationale Wahrscheinlichkeiten p, die nur mit sehr großem Zähler und sehr großem Nenner darstellbar sind) lässt sich mit dieser Modellierung jedes Zufallsexperiment nachbauen, das man im Bereich der randomisierten Algorithmen braucht.

(Übung: Wie realisiert man auf einer RRAM den Wurf einer Münze mit Wahrscheinlichkeit p für „Kopf“ und $1-p$ für „Zahl“, wobei $0 \leq p \leq 1$ rational ist?)

Wenn $k = (z, \alpha)$ eine Konfiguration einer RRAM M ist, wo B_z der RANDOM-Befehl ist, und $\alpha(0) \geq 2$, so hat k genau $\alpha(0)$ viele verschiedene legale Nachfolgekonfigurationen $k' = (z+1, \alpha')$, für jeden neuen Wert $0, 1, \ldots, \alpha(0)-1$ als $\alpha'(0)$ eine. Wir schreiben $k \vdash_M k'$ für jede dieser Konfigurationen und definieren eine Übergangswahrscheinlichkeit:

$$p_{k,k'} := \frac{1}{\alpha(0)}, \text{ für jedes solche } k'.$$

(Wenn der RANDOM-Befehl auszuführen ist und $\alpha(0) \in \{0, 1\}$, wird wie bei gewöhnlichen Rechenschritten eine eindeutige Nachfolgekonfiguration festgelegt.) Falls k nur eine Nachfolgekonfiguration k' besitzt, setzen wir $p_{k,k'} := 1$, auch in dem Fall, wo die Ausführung eines gewöhnlichen RAM-Befehls von k zu k' führt.

Nun können wir jeder (Teil-)Berechnung eine Wahrscheinlichkeit zuordnen: Wenn $\varphi = (k_0, k_1, \ldots, k_s)$ eine Konfigurationenfolge mit $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_s$ ist, so setzen wir

$$p_\varphi := p_{k_0 k_1} \cdot p_{k_1 k_2} \cdot p_{k_2 k_3} \cdots p_{k_{s-1} k_s}.$$

Diese Festlegung (Multiplikation der Wahrscheinlichkeiten) modelliert die Unabhängigkeit der von den RANDOM-Befehlen ausgelösten Zufallsexperimente. Wenn in φ genau u RANDOM-Befehle vorkommen, wobei in R_0 die Inhalte $r_1, \ldots, r_u \geq 2$ stehen, dann ist $p_\varphi = \frac{1}{r_1} \cdots \frac{1}{r_u}$.

Zu einem gegebenen randomisierten Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ betrachten wir für jeden Input x die Menge der Berechnungen, die möglicherweise zu interessanten Ergebnissen führen. Dies sind natürlich nur terminierende Berechnungen:

$$\Omega_{\mathcal{A},x} := \{\varphi \mid \varphi = (k_0, k_1, \ldots, k_t), \text{IN}(x) = k_0 \vdash_M \cdots \vdash_M k_t,\ k_t \text{ Haltekonfiguration}\}$$

Eine andere (anschauliche) Sicht auf diese Menge ist durch den Begriff des *Berechnungsbaums* $\text{CT}_{\mathcal{A},x}$ gegeben. Dies ist ein (meist unendlicher) Baum, der folgendermaßen induktiv aufgebaut wird: Die Wurzel ist mit der Konfiguration $k_0 = \text{IN}(x)$ beschriftet. Knoten, die mit einer Haltekonfiguration beschriftet sind, haben keine Nachfolger, sind also Blätter. Wenn Knoten v mit Konfiguration $k = (z, \alpha)$ beschriftet ist und in Befehlszeile z der Befehl „RANDOM“ ist, und wenn $r = \alpha(0) \geq 2$ ist, dann hat v genau r Nachfolgeknoten $v_0, \ldots, v_{r-1}$, die mit den r Nachfolgekonfigurationen von k beschriftet sind. Die Kante von v nach v_i wird mit „$\frac{1}{r}$“ markiert, für $0 \leq i < r$. In allen anderen Fällen hat v einen Nachfolgeknoten v', der mit der

eindeutig bestimmten Nachfolgekonfiguration k' von k beschriftet ist. Die Kante von v nach v' wird mit „1“ markiert.

Man sieht sofort, dass die Wege in $\mathrm{CT}_{\mathcal{A},x}$ genau die Berechnungen von M auf x sind, wobei es möglicherweise auch unendliche Wege, also nichtterminierende Berechnungen gibt. Der Menge $\Omega_{\mathcal{A},x}$ entsprechen Wege, die an Blättern enden. Die Wahrscheinlichkeit, in einem Knoten v „vorbeizukommen“, ist genau das Produkt der Kantenbeschriftungen auf dem Weg von der Wurzel nach v.

Wir würden gerne die Menge $\Omega_{\mathcal{A},x}$ als Trägermenge eines Wahrscheinlichkeitsraumes benutzen, um die Berechnungen von $\mathcal{A}$ auf x zu modellieren. Geht das? Wir beobachten:

Lemma 3.1.3

$$\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi \leq 1.$$

Man beachte, dass es sich um eine Summe von eventuell unendlich vielen positiven Summanden handelt. Da $\mathcal{K}$ abzählbar unendlich ist, ist auch $\Omega_{\mathcal{A},x}$ höchstens abzählbar unendlich. Der Wert solcher Summen ist immer wohldefiniert.

Beweis des Lemmas:[1] Wir sagen, dass $\varphi = (k_0, \ldots, k_s)$ eine partielle Berechnung von $\mathcal{A}$ auf x ist, wenn $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_s$ und $k_0 = \mathrm{IN}(x)$ ist. (Wir verlangen nicht, dass k_t Haltekonfiguration ist.)
Für $t \geq 0$ definiere

$$S_t := \sum\{p_\varphi \mid \varphi = (k_0, \ldots, k_s),\ \varphi \text{ ist partielle Berechnung auf } x \text{ mit } s = t \text{ oder } (s < t \text{ und } \varphi \in \Omega_{\mathcal{A},x})\ \}.$$

Man zeigt durch Induktion über t, dass $S_t = 1$ für alle t gilt. Für $t = 0$ ist dies klar; der Induktionsschritt von $t-1$ auf t benutzt, dass sich die Summe nicht ändert, wenn man einen Rechenschritt ausführt und manche partiellen Berechnungen durch einen RANDOM-Befehl in Schritt t in mehrere partielle Berechnungen aufgespaltet werden.
Damit gilt für die Teilsumme

$$S'_t = \sum\{p_\varphi \mid \varphi = (k_0, \ldots, k_s), s \leq t \text{ und } \varphi \in \Omega_{\mathcal{A},x}\ \},$$

[1] Für Interessierte (in der Vorlesung weggelassen).

die die bis Schritt t beendeten Berechnungen erfasst, erst recht $S'_t \leq 1$. Daher ist $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi = \sup\{S'_t \mid t \geq 0\} \leq 1$, und das Lemma ist bewiesen. □

Könnte es sein, dass $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi < 1$ ist? Dann gäbe es ein $\delta > 0$ derart, dass $S'_t \leq 1-\delta$ ist für alle $t \geq 0$. Daraus folgt, dass für jedes t die Werte p_φ für die bis zum Schritt t nicht beendeten partiellen Berechnungen φ eine Summe von mindestens δ ergeben; informal kann man das so interpretieren, dass die zufallsgesteuerte Berechnung von $\mathcal{A}$ auf x mit Wahrscheinlichkeit mindestens δ nie anhält. Mit Berechnungen, die mit positiver Wahrscheinlichkeit nicht anhalten, wollen wir uns in dieser Vorlesung nicht beschäftigen. Wir schließen daher randomisierte Algorithmen $\mathcal{A}$, die $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi < 1$ für ein $x \in I$ erfüllen, von der weiteren Betrachtung aus.

Bemerkung 3.1.4
Wenn man es doch einmal mit einem solchen Algorithmus zu tun bekommt, kann man ihn wie folgt modifizieren. Aus x berechnet man eine Zeitschranke $b(x) \in \mathbb{N}$ derart, dass die Eigenschaften des Algorithmus nicht entscheidend verändert werden, wenn man ihn nach $b(x)$ Schritten abbricht. Dann ändert man das RRAM-Programm so, dass die Schritte gezählt und laufend mit der Schranke $b(x)$ verglichen werden. Wird diese Zeitschranke überschritten, wird die Berechnung abgebrochen und das Ergebnis „nicht fertig“ ausgegeben. Diese Änderung verlangsamt den Algorithmus um einen konstanten Faktor, erzwingt aber, dass er immer hält.

Festlegung.[2]
Wir betrachten nur randomisierte Algorithmen $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$, die $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi = 1$ erfüllen, für alle $x \in I$.

Wir haben schon bemerkt, dass $\Omega_{\mathcal{A},x}$ abzählbar ist. Daher bildet nach der „Festlegung“ das Paar

$$(\Omega_{\mathcal{A},x}, (p_\varphi)_{\varphi\in\Omega_{\mathcal{A},x}})$$

einen *Wahrscheinlichkeitsraum*. Dieser Wahrscheinlichkeitsraum ist die Basis für die Untersuchung des Verhaltens von $\mathcal{A}$ auf x. Wenn wir uns explizit hierauf beziehen wollen, schreiben wir

$$\mathbf{Pr}_{\mathcal{A},x}(\ldots), \mathbf{E}_{\mathcal{A},x}(\ldots), \mathbf{Var}_{\mathcal{A},x}(\ldots), \text{ usw.}$$

Man beachte, dass jede Eingabe $x \in I$ ihren eigenen Wahrscheinlichkeitsraum erzeugt. Wir lassen die Indizes weg, wenn aus dem Zusammenhang klar ist, welcher Wahrscheinlichkeitsraum gemeint ist.

[2] Man beachte dabei, dass die Menge der Algorithmen mit dieser Eigenschaft *unentscheidbar* ist. Weder sie selbst noch ihr Komplement ist semientscheidbar (rekursiv aufzählbar).

Definition 3.1.5
Für einen Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$, $x \in I$ und $\varphi \in \Omega_{\mathcal{A},x}$ definieren wir:

$$t_{\mathcal{A},x}(\varphi) := t, \text{ wenn } \varphi = (k_0, \ldots, k_t).$$

Damit ist $t_{\mathcal{A},x} \colon \Omega_{\mathcal{A},x} \to \mathbb{N}$ eine *Zufallsvariable*, für die wir auch $t_{\mathcal{A}}(x)$ schreiben. Man sollte sich von dieser Notation nicht zu dem Eindruck verleiten lassen, dass es sich bei $t_{\mathcal{A}}(x)$ um eine Zahl handelt.

Analog definieren wir den Speicherplatz:

Definition 3.1.6
Für einen Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$, $x \in I$ und $\varphi \in \Omega_{\mathcal{A},x}$ definieren wir:

$$s_{\mathcal{A},x}(\varphi) := \max\{j \mid \text{ in } \varphi = (k_0, \ldots, k_t) \text{ wird } R_j \text{ gelesen oder beschrieben}\}.$$

Wir schreiben auch $s_{\mathcal{A}}(x)$ für diese Zufallsvariable.

Wir können zu diesen Zufallsvariablen Erwartungswerte bilden:

Definition 3.1.7
Die **erwartete Rechenzeit** von $\mathcal{A}$ auf x:

$$\overline{t}_{\mathcal{A}}(x) := \mathbf{E}(t_{\mathcal{A},x}) = \sum_{\varphi \in \Omega_{\mathcal{A},x}} p_\varphi \cdot t_{\mathcal{A},x}(\varphi).$$

Der **erwartete Speicherplatz** von $\mathcal{A}$ auf x:

$$\overline{s}_{\mathcal{A}}(x) := \mathbf{E}(s_{\mathcal{A},x}).$$

Man beachte: Nach Fakt 2.2.9 gilt: $\mathbf{E}(t_{\mathcal{A},x}) = \sum_{i \geq 1} \mathbf{Pr}(t_{\mathcal{A},x} \geq i)$. (Analog für $\mathbf{E}(s_{\mathcal{A},x})$.)

Im Kontrast dazu stehen ***worst-case-Rechenzeit*** und ***worst-case-Speicherplatz***:

Definition 3.1.8
Die **Rechenzeit** von $\mathcal{A}$ auf x im schlechtesten Fall:

$$\widehat{t}_{\mathcal{A}}(x) := \sup\{t_{\mathcal{A},x}(\varphi) \mid \varphi \in \Omega_{\mathcal{A},x}\}.$$

(Dies ist ein Wert in $\mathbb{N} \cup \{\infty\}$.)
Der **Speicherplatz** von $\mathcal{A}$ auf x im schlechtesten Fall:

$$\widehat{s}_{\mathcal{A}}(x) := \sup\{s_{\mathcal{A},x}(\varphi) \mid \varphi \in \Omega_{\mathcal{A},x}\}.$$

Wie bei gewöhnlichen Algorithmen betrachtet man Größenklassen von Inputs. Hierzu nimmt man an, dass eine Abbildung size: $I \to \mathbb{N}$ gegeben ist, die jedem Input x seine ***Größe*** $\text{size}(x) = |x|$ zuordnet. Die Größenklassen sind die Mengen $I_n = \{x \in I \mid |x| = n\}$. Wir maximieren Zeitschranken über solche Größenklassen:

Definition 3.1.9
Für $n \in \mathbb{N}$ definiere:

$$\overline{T}_{\mathcal{A}}(n) := \sup\{\overline{t}_{\mathcal{A}}(x) \mid |x| = n\}.$$

Diese Größe gibt die „erwartete Rechenzeit im schlechtesten Fall über alle Inputs der Größe n“ an; auf englisch: „worst case expected time“.

Definition 3.1.10
Für $n \in \mathbb{N}$ definiere:

$$\widehat{T}_{\mathcal{A}}(n) := \sup\{\widehat{t}_{\mathcal{A}}(x) \mid |x| = n\} = \sup\{t_{\mathcal{A},x}(\varphi) \mid |x| = n, \varphi \in \Omega_{\mathcal{A},x}\}.$$

Diese Größe gibt die „Rechenzeit im schlechtesten Fall über alle Inputs der Größe n“ an.

Natürlich könnte man zu $\overline{T}_{\mathcal{A}}(n)$ und $\widehat{T}_{\mathcal{A}}(n)$ analoge Größen auch für den Speicherplatz definieren.

Zuletzt wollen wir noch definieren, was wir als Resultat eines randomisierten Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ auf einem Input x ansehen wollen.

Definition 3.1.11
Sei $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ ein randomisierter Algorithmus. Das **Resultat** von $\mathcal{A}$ auf $x \in I$ ist die folgende Zufallsgröße:

$$\text{res}_{\mathcal{A},x}(\varphi) := \text{OUT}(k_t), \text{ für } \varphi = (k_0, \ldots, k_t) \in \Omega_{\mathcal{A},x}.$$

Statt $\text{res}_{\mathcal{A},x}$ schreiben wir meistens $\boldsymbol{\mathcal{A}(x)}$.

Bei $\text{res}_{\mathcal{A},x} = \mathcal{A}(x)$ handelt es sich also um eine *Zufallsgröße* mit Werten in Z.

3.2 Typen randomisierter Algorithmen

Definition 3.2.1
$\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ sei ein randomisierter Algorithmus, und $f\colon I \to Z$ sei eine Funktion. Die Zahl

$$\text{err}_{\mathcal{A},f}(x) := \mathbf{Pr}(\mathcal{A}(x) \neq f(x))$$

heißt die **Fehlerwahrscheinlichkeit von $\mathcal{A}$ bzgl. f auf Eingabe x**. Noch genauer, wenn man den Wahrscheinlichkeitsraum explizit macht:

$$\text{err}_{\mathcal{A},f}(x) = \sum_{\substack{\varphi \in \Omega_{\mathcal{A},x} \\ \text{res}_{\mathcal{A},x}(\varphi) \neq f(x)}} p_\varphi.$$

Klar: Mit Wahrscheinlichkeit $1 - \text{err}_{\mathcal{A},f}(x)$ berechnet $\mathcal{A}$ auf Input x den Wert $f(x)$.

Definition 3.2.2
$\mathcal{A}$ und f seien wie eben. Wir sagen, dass $\mathcal{A}$ die Funktion f „mit unbeschränktem Fehler“[3] berechnet, falls für jedes $x \in I$ gilt: $\text{err}_{\mathcal{A},f}(x) < \frac{1}{2}$.

Bemerkung 3.2.3
Wenn es eine Funktion f gibt, die von $\mathcal{A}$ mit unbeschränktem Fehler berechnet wird, dann ist f eindeutig bestimmt. (*Beweis*: Sei $x \in I$. Dann gibt es maximal ein $z \in Z$ mit $\mathbf{Pr}(\mathcal{A}(x) \neq z) < \frac{1}{2}$. Dies sieht man wie folgt: Seien $z, z' \in Z$ verschieden. Dann sind $\{\mathcal{A}(x) = z\}$ und $\{\mathcal{A}(x) = z'\}$ zwei disjunkte Ereignisse, die also

[3] Korrekter müsste man „mit unter $\frac{1}{2}$ nicht beschränkter Fehler*wahrscheinlichkeit*“ oder noch besser „... *Irrtumswahrscheinlichkeit*“ sagen, aber das ist zu umständlich und daher nicht üblich.

nicht beide Wahrscheinlichkeit $> \frac{1}{2}$ haben können. Also können die Ungleichungen $\mathbf{Pr}(\mathcal{A}(x) \neq z) < \frac{1}{2}$ und $\mathbf{Pr}(\mathcal{A}(x) \neq z') < \frac{1}{2}$ nicht gleichzeitig gelten.)
Die Fehlerwahrscheinlichkeit $\frac{1}{2}$ ist hier die Grenze: Der Algorithmus, der auf Input x einfach zufällig einen der Werte 0 und 1 wählt und ausgibt, berechnet *jede* Funktion $f\colon I \to \{0,1\}$ „mit Fehlerwahrscheinlichkeit $\leq \frac{1}{2}$“. Solche Algorithmen können natürlich zur Berechnung von Funktionen nicht geeignet sein.

Um den Überblick zu behalten, werden wir oft Fehlerschranken angeben, die für alle Inputs einer Größe gelten. Uns interessiert also die Situation, wo es für jedes n eine Schranke $\varepsilon_n < \frac{1}{2}$ für die Werte $\mathrm{err}_{\mathcal{A},f}(x)$ für alle x mit $|x| = n$ gibt.

Definition 3.2.4
$\mathcal{A}$ und f seien wie eben.
Sei $(\varepsilon_n)_{n\geq 0}$ eine Folge von Zahlen mit $0 \leq \varepsilon_n < \frac{1}{2}$. Wir sagen, dass $\mathcal{A}$ die Funktion f **mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet**, falls für jedes $x \in I$ mit $|x| = n$ gilt: $\mathrm{err}_{\mathcal{A},f}(x) \leq \varepsilon_n$.
$\mathcal{A}$ berechnet die Funktion f **mit beschränktem Fehler**, falls ein $\varepsilon < \frac{1}{2}$ existiert, so dass $\mathrm{err}_{\mathcal{A},f}(x) \leq \varepsilon$ für alle $x \in I$. In diesem Fall heißt $\mathcal{A}$ ein **Monte-Carlo-Algorithmus**.

Beispiel: Es sei $\mathcal{A}$ der Algorithmus MinCut aus Kapitel 1, in der Variante mit $n^2/2$ Wiederholungen, also Fehlerwahrscheinlichkeit e^{-1}, so modifiziert, dass er nicht den ermittelten Schnitt C, sondern nur die Kardinalität $|C|$ als Resultat ausgibt. Dieser Algorithmus berechnet die Funktion f, die einem Graphen G die Größe eines minimalen Schnitts in G zuordnet, mit Fehlerschranke e^{-1}. Wenn man $n^2(\ln n)/2$ Wiederholungen ansetzt, erhält man einen Algorithmus mit Fehlerschranke $(1/n)_{n\geq 1}$. (Hierbei ist die Eingabegröße n, die Anzahl der Knoten.)

Wir betrachten nun noch den äußerst häufigen und wichtigen Spezialfall von *Entscheidungsproblemen*, also Funktionen mit nur zwei Werten 0 („nein“) und 1 („ja“), mit einem Algorithmus, der nur bei einem Funktionswert einen Fehler machen kann.

Definition 3.2.5
Sei $\mathcal{A}$ ein Algorithmus für Inputs aus einer Menge I, derart dass $\mathrm{res}_{\mathcal{A},x}$ nur Werte in $\{0,1\}$ annimmt, und sei $f\colon I \to \{0,1\}$ eine Funktion. Wir sagen, dass $\mathcal{A}$ die Funktion f „mit einseitigem Fehler“ berechnet, falls für jedes $x \in I$ gilt:

$$\begin{aligned} f(x) = 0 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) = 1; \\ f(x) = 1 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) < 1. \end{aligned}$$

Das heißt also Folgendes:

- Wenn $\mathcal{A}$ auf Input x als Resultat 1 ausgibt (auf mindestens einem Berechnungsweg), dann ist garantiert $f(x) = 1$.
- Wenn $\mathcal{A}$ auf Input x als Resultat 0 ausgibt, ist $f(x) = 0$ und $f(x) = 1$ möglich.

Der Zusammenhang zwischen Algorithmus und Funktion hat Ähnlichkeit mit der Situation bei nichtdeterministischen Maschinenmodellen. Man sieht sofort, dass ein solcher Algorithmus die Funktion f eindeutig bestimmt. Wieder ist die Situation besonders interessant, wo die Fehlerwahrscheinlichkeit durch eine Schranke kleiner als 1 beschränkt ist, für alle Inputs (einer Größe) gleichmäßig.

Definition 3.2.6
$\mathcal{A}$ und f seien wie eben, mit Werten 0 und 1. Sei weiter $(\varepsilon_n)_{n\geq 0}$ eine Folge von Zahlen mit $0 \leq \varepsilon_n < 1$. Wir sagen, dass $\mathcal{A}$ die Funktion f mit einseitigem Fehler und Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet, wenn für jedes $x \in I$ mit $|x| = n$ gilt:

$$\begin{aligned} f(x) = 0 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) = 1; \\ f(x) = 1 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) \leq \varepsilon_n. \end{aligned}$$

Gängige, etwas ungenaue Abkürzung: „$\mathcal{A}$ berechnet f mit einseitigem Fehler ε_n."
$\mathcal{A}$ berechnet die Funktion f **mit beschränktem einseitigen Fehler**, falls ein $\varepsilon < 1$ existiert, so dass $\mathcal{A}$ die Funktion f mit einseitigem Fehler und Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet, mit $\varepsilon_n = \varepsilon$ für alle $n \geq 0$. In diesem Fall heißt $\mathcal{A}$ ein **Monte-Carlo-Algorithmus mit einseitigem Fehler**.

Beispiel: Die Algorithmen in Abschnitt 1.2 (Vergleich von Polynomprodukten) und 1.3 (Verifikation von Matrixprodukten) sind Monte-Carlo-Algorithmen mit einseitigem Fehler.

Schließlich sind natürlich Algorithmen interessant, die stets (mit Wahrscheinlichkeit 1) das richtige Resultat liefern.

Definition 3.2.7
$\mathcal{A}$ sei ein Algorithmus für Inputs aus einer Menge I, $f\colon I \to Z$ sei eine Funktion. Wir sagen, dass $\mathcal{A}$ die Funktion f „ohne Fehler“ berechnet, falls für jedes $x \in I$ gilt:

$$\mathrm{err}_{\mathcal{A},f}(x) = 0 \text{ , d. h. } \mathbf{Pr}(\mathcal{A}(x) \neq f(x)) = 0.$$

In diesem Fall heißt $\mathcal{A}$ ein **Las-Vegas-Algorithmus**.

Beispielsweise ist Randomisiertes Quicksort (Abschnitt 1.4) ein Las-Vegas-Algorithmus. Man verwendet Las-Vegas-Algorithmen, wenn sie einfacher (zu programmieren) sind als deterministische Varianten, wenn die erwartete Laufzeit besser ist als die worst-case-Laufzeit, und um worst-case-Inputs auszuschließen.

3.3 Wahrscheinlichkeitsverbesserung

Was sollen wir mit einem Algorithmus $\mathcal{A}$ anfangen, der eine 0-1-wertige Funktion f mit einseitigem Fehler 0,99 berechnet? Wenn dieser Algorithmus auf Eingabe x die Ausgabe 1 liefert, wissen wir, dass $f(x) = 1$ ist – gut. Aber wenn die Ausgabe 0 ist, wissen wir ja eigentlich gar nichts.

Genauso kann man fragen, was ein Algorithmus $\mathcal{A}$ nützt, der eine Funktion f mit Fehlerschranke 0,49 berechnet. Wenn auf Input x die Ausgabe y geliefert wird, ist es sehr gut möglich, dass $f(x) \neq y$ ist. Was sollen wir mit einem solchen Resultat anfangen?

Der Ansatz der „Wahrscheinlichkeitsverbesserung“ (engl.: *probability amplification*) führt hier weiter. Man benutzt den Algorithmus $\mathcal{A}$ mit der schlechten Fehlerschranke, um einen neuen Algorithmus $\mathcal{A}'$ mit kleinerer Fehlerwahrscheinlichkeit zu erhalten.

3.3.1 Einseitiger Fehler

Wir beginnen mit MC-Algorithmen mit einseitigem Fehler.

Sei $\mathcal{A}$ ein Algorithmus, der f mit einseitigem Fehler mit Schranke $(\varepsilon_n)_{n\geq 0}$ berechnet.

Wir betrachten den folgenden neuen Algorithmus $\mathcal{A}'$ (in Pseudocode-Notation), der im Wesentlichen $\mathcal{A}$ auf Eingabe x mit $|x| = n$ c-mal wiederholt (wobei $c = c_n$ eine zu wählende Zahl ist) und dabei beobachtet, ob einmal das Resultat 1 auftritt.

Algorithmus 3.3.1 ***Wiederholung einseitig***

INPUT: x mit $|x| = n$.
METHODE:
1 Bestimme Wiederholungszahl $c = c_n$
2 **for** i := 1 **to** c **do**
3 r := $\mathcal{A}(x)$;
4 **if** r = 1 **then return** 1;
5 **return** 0. // Habe c-mal Resultat 0 erhalten.

Algorithmus $\mathcal{A}'$ ruft also c-mal Algorithmus $\mathcal{A}$ auf demselben Input x auf, für $c = c_n$. Wichtig ist, dass bei jedem Aufruf neue Zufallsexperimente durchgeführt werden, so dass die Resultate $r_1, \ldots, r_c$ (stochastisch) unabhängig sind.

Analyse: 1. Zeitbedarf. Zur Organisation der Wiederholungen ist in jedem Schleifendurchlauf Aufwand $O(1)$ nötig. Ein Durchlauf durch $\mathcal{A}$ kostet erwartete Zeit $\bar{t}_{\mathcal{A}}(x)$. Wegen der Linearität des Erwartungswertes ist $\bar{t}_{\mathcal{A}'}(x) = O(c_n) + c_n \cdot \bar{t}_{\mathcal{A}}(x)$.
2. Fehlerwahrscheinlichkeit. Wenn $f(x) = 0$ ist, dann liefern (mit Wahrscheinlichkeit 1) alle Aufrufe von $\mathcal{A}$ das Resultat 0, also gibt auch $\mathcal{A}'$ den Wert 0 aus. Wenn $f(x) = 1$ ist, sehen wir, nach dem Aufbau des Algorithmus:

$$\mathbf{Pr}(\mathcal{A}'(x){=}0) = \mathbf{Pr}(r_1{=}0 \wedge \ldots \wedge r_{c_n}{=}0) = \prod_{1 \le i \le c_n} \mathbf{Pr}(r_i = 0) = \mathbf{Pr}(\mathcal{A}(x) = 0)^{c_n} \le \varepsilon_n^{c_n}.$$

Dabei haben wir die Unabhängigkeit der Ergebnisse $r_1, \ldots, r_{c_n}$ benutzt, sowie die Tatsache, dass die Fehlerwahrscheinlichkeit in jedem Aufruf dieselbe ist.

Wenn $\varepsilon_n \le \frac{1}{2}$, dann gilt also $\mathbf{Pr}(\mathcal{A}'(x) = 0) \le 2^{-c_n}$, eine Zahl, die man klein machen kann, indem man c_n genügend groß wählt.

Wir sehen uns den Fall $\frac{1}{2} < \varepsilon_n < 1$ noch etwas genauer an.[4] Setze $\delta_n := 1 - \varepsilon_n$. Dann gilt $0 < \delta_n < \frac{1}{2}$. Wir haben

$$\mathbf{Pr}(\mathcal{A}'(x) = 0) \le (1 - \delta_n)^{c_n} \overset{\text{Prop. A.1.2}}{\le} e^{-c_n \delta_n}.$$

Wenn wir uns eine Fehlerschranke $\varepsilon'_n < \varepsilon_n$ wünschen, können wir vorab bestimmen,

[4] Man stelle sich zur Illustration die Schranke $\varepsilon_n = 1 - \frac{2}{n^2}$ vor, wie sie beim MinCut-Algortihmus in 1.1 aufgetreten ist.

welche Wiederholungszahl $c_n = c(\varepsilon_n, \varepsilon'_n)$ ausreichend ist:

$$\mathbf{Pr}(\mathcal{A}'(x) = 0) \leq \varepsilon'_n \Leftarrow e^{-c_n\delta_n} \leq \varepsilon'_n \Leftarrow e^{c_n\delta_n} \geq \frac{1}{\varepsilon'_n} \Leftarrow c_n\delta_n \geq \ln(1/\varepsilon'_n) \Leftarrow c_n \geq \frac{\ln(1/\varepsilon'_n)}{1-\varepsilon_n}.$$

Wir erhalten:

Satz 3.3.2
Wenn $\mathcal{A}$ die Funktion $f\colon I \to \{0,1\}$ mit einseitigem Fehler $(\varepsilon_n)_{n\geq 0}$ berechnet und $(\varepsilon'_n)_{n\geq 0}$ eine Folge von Wunsch-Fehlerschranken ist, so berechnet jede Version des Algorithmus 3.3.1, die $c_n \geq (\ln 2)\log(1/\varepsilon'_n)/(1-\varepsilon_n)$ Wiederholungen benutzt, die Funktion f mit einseitigem Fehler $(\varepsilon'_n)_{n\geq 0}$. Der erwartete Zeitbedarf ist $\bar{t}_{\mathcal{A}'}(x) = c_n \cdot (\bar{t}_{\mathcal{A}}(x) + O(1))$.

Es ist noch interessant zu überlegen, wie sich verschiedene Werte ε_n und ε'_n auf die Wiederholungszahl $c(\varepsilon_n, \varepsilon'_n) \approx \log(1/\varepsilon'_n)/(1-\varepsilon_n)$ auswirken. Der Einfluss von ε_n auf die Wiederholungszahl ist im Wesentlichen proportional zu $\frac{1}{1-\varepsilon_n}$. Beispielsweise könnte $\varepsilon_n = 1 - n^{-k}$ sein. In diesem Fall wäre der Beitrag des Nenners zur Wiederholungszahl ein Faktor von n^k. Der Einfluss von ε'_n ist im Vergleich viel geringer. Um die Fehlerschranke zu halbieren, muss man den Zähler $\log(1/\varepsilon'_n)$ nur um 1 erhöhen. Zum Beispiel führt die ziemlich kleine Wunsch-Fehlerschranke $\varepsilon'_n = 2^{-20} \approx 10^{-6}$ zu einem Zähler $20\ln 2$ in $c(\varepsilon_n, \varepsilon'_n)$. Soll die Fehlerschranke $\varepsilon'_n = 1/n^2$ sein, ist ein Zähler von $(2\ln 2)\log n$ ausreichend. Mit einem Zähler von $(\ln 2)n^\ell$ erreicht man sogar eine Fehlerschranke von $\varepsilon'_n = 2^{-n^\ell}$.

Beispielsweise können wir mit einem Aufwand, der nur einem zusätzlichen Faktor $n = |V|$ in der Laufzeit entspricht, die Fehlerwahrscheinlichkeit beim MinCut-Algorithmus auf e^{-n} drücken.

3.3.2 Allgemeine Algorithmen mit beschränktem Fehler

Hier betrachten wir Algorithmen, die Funktionen $f\colon I \to Z$ berechnen, mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$, wobei $\varepsilon_n < \frac{1}{2}$ gilt.

Die Grundidee zur Verringerung der Fehlerwahrscheinlichkeit ist dieselbe wie vorher: *Wiederholung*. Wir lassen also den gegebenen Algorithmus $\mathcal{A}$ auf Eingabe x mit $|x| = n$ c-mal ablaufen, für eine (noch zu bestimmende) Wiederholungszahl $c = c_n$. Es entstehen Ergebnisse $r_1, \ldots, r_c$ aus Z. Beispielsweise könnte $Z = \mathbb{N}$, $c = 15$ und

$$(r_1, \ldots, r_c) = (2, 1, 3, 1, 1, 2, 1, 1, 4, 2, 1, 1, 1, 3, 1)$$

sein. Es liegt nahe, hier das Ergebnis 1 auszugeben, da es ziemlich oft vorkommt. Allgemeiner gesagt: Wenn es einen Wert r gibt, der in $(r_1, \ldots, r_c)$ mehr als $\frac{1}{2}c$-mal vorkommt, sollte man diesen Wert ausgeben. Er ist dann eindeutig bestimmt. Wenn es keinen solchen Wert gibt (z.B. bei der Ergebnisfolge $(r_1, \ldots, r_c) = (2, 1, 3, 1, 2, 2, 1, 3, 4, 2, 1, 2, 1, 3, 1)$), dann ist es eigentlich egal, welchen Wert wir ausgeben.

Wir betrachten also einen neuen Algorithmus $\mathcal{A}'$ (Algorithmus 3.3.3), der im Wesentlichen $\mathcal{A}$ c-mal wiederholt und dabei beobachtet, ob ein Resultat r in der (absoluten) Mehrheit der Fälle auftritt.

Algorithmus 3.3.3 ***Mehrheitsverfahren einfach***
INPUT: $x \in I$ mit $|x| = n$.
METHODE:
1 Bestimme $c = c_n$;
2 **for** `i` $:= 1$ **to** c **do**
3 `r[i]` $:= \mathcal{A}(x)$;
4 **if** in `r[1..`c`]` kommt Wert r häufiger als $\frac{1}{2}c$-mal vor
5 **then return** r
6 **else return** einen beliebigen Wert aus Z.

Analyse: 1. Zeitbedarf. Wir nehmen der Einfachheit halber an, dass die Resultate von $\mathcal{A}$ Objekte sind, die in konstanter Zeit zu speichern und zu vergleichen sind. Die erwartete Zeit für die Wiederholungen ist wie vorher $c_n \cdot (\bar{t}_{\mathcal{A}}(x) + O(1))$. Wir werden weiter unten noch sehen, dass Aufwand $O(c_n)$ genügt, um ein Ergebnis zu identifizieren, das häufiger als $\frac{1}{2}c_n$-mal vorkommt (falls es existiert); daher ist der zusätzliche Aufwand für die Auswahl der Ausgabe ebenfalls $O(c_n)$.

2. Fehlerwahrscheinlichkeit. Sei $x \in I$ ein fester Input mit $n = |x|$. Sei $\varepsilon := \varepsilon(x) := \mathrm{err}_{\mathcal{A},f}(x)$ die Fehlerwahrscheinlichkeit von $\mathcal{A}$ auf x. Sei $c = c_n$. Die Zufallswerte $r_1, \ldots, r_c$ sollen die Resultate der c Aufrufe von $\mathcal{A}$ im Verlauf des Ablaufs von $\mathcal{A}'$ auf x sein. Nach der Formulierung des Algorithmus $\mathcal{A}'$ gilt:

$$\mathcal{A}'(x) \neq f(x) \;\Rightarrow\; \text{mindestens } c/2 \text{ der } r_i \text{ sind } \neq f(x).$$

Wir müssen also eine obere Schranke für die Wahrscheinlichkeit finden, dass dies passiert. Wir beobachten: Wenn $r_i \neq f(x)$ für mindestens $c/2$ der $i \in \{1, \ldots, c\}$, dann gibt es eine Teilmenge $D \subseteq \{1, \ldots, c\}$ mit Kardinalität $|D| \geq c/2$, so dass

$r_i \neq f(x)$ für alle $i \in D$ und $r_i = f(x)$ für alle $i \notin D$ gilt. Wir definieren Ereignisse

$$A_D := \{\forall i \in D : r_i \neq f(x) \wedge \forall i \notin D : r_i = f(x)\}.$$

Dann gilt, wie eben beobachtet:

$$\{\mathcal{A}'(x) \neq f(x)\} \subseteq \bigcup_{\substack{D \subseteq \{1,\ldots,c\} \\ |D| \geq c/2}} A_D.$$

Für jedes D gilt wegen der Unabhängigkeit der c Wiederholungen:

$$\mathbf{Pr}(A_D) = \varepsilon(x)^{|D|}(1-\varepsilon(x))^{c-|D|} = \varepsilon^{|D|}(1-\varepsilon)^{c-|D|}.$$

Mit Monotonie (Fakt 2.1.6(d)) und der Vereinigungsschranke (Fakt 2.1.6(c)) ergibt sich

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sum_{\substack{D \subseteq \{1,\ldots,c\} \\ |D| \geq c/2}} \mathbf{Pr}(A_D) = \sum_{\substack{D \subseteq \{1,\ldots,c\} \\ |D| \geq c/2}} \varepsilon^{|D|}(1-\varepsilon)^{c-|D|}.$$

Für jedes j gibt es genau $\binom{c}{j}$ viele Teilmengen $D \subseteq \{1,\ldots,c\}$ mit Kardinalität j. Wir erhalten:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sum_{c/2 \leq j \leq c} \binom{c}{j} \varepsilon^j (1-\varepsilon)^{c-j}.$$

Weil $\varepsilon < \frac{1}{2}$ und $j \geq c/2$, gilt $\varepsilon^{j-c/2} \leq (1-\varepsilon)^{j-c/2}$, und wir können folgern:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sum_{c/2 \leq j \leq c} \binom{c}{j} (\varepsilon(1-\varepsilon))^{c/2} \leq \sum_{0 \leq j \leq c} \binom{c}{j} (\varepsilon(1-\varepsilon))^{c/2}.$$

Mit $\sum_{0 \leq j \leq c} \binom{c}{j} = 2^c$ erhalten wir schließlich:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq (4\varepsilon(1-\varepsilon))^{c/2}.$$

Wir beobachten, dass die Funktion $t \mapsto t(1-t)$ im Intervall $[0, \frac{1}{2}]$ strikt monoton wächst, und daher $4\varepsilon(1-\varepsilon) < 4 \cdot \frac{1}{2} \cdot \frac{1}{2} = 1$ gilt. Damit sieht unser Ergebnis schon ganz gut aus: Wir erhalten eine Schranke für die Fehlerwahrscheinlichkeit von $\mathcal{A}'$, die in der Wiederholungszahl c exponentiell fällt (mit Basis $\sqrt{4\varepsilon(1-\varepsilon)} < 1$). Weil $\varepsilon_n < \frac{1}{2}$ eine obere Schranke für $\varepsilon = \varepsilon(x)$ ist, gilt wegen der Monotonie von $t \mapsto t(1-t)$ auch $\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq (4\varepsilon_n(1-\varepsilon_n))^{c_n/2}$.

Wie groß muss man $c_n = c(\varepsilon_n, \varepsilon'_n)$ wählen, um eine vorgegebene Fehlerwahrscheinlichkeit ε'_n zu erreichen?

Wenn $\varepsilon_n \leq \frac{2-\sqrt{3}}{4} \approx 0{,}067$, dann gilt $\varepsilon_n(1-\varepsilon_n) \leq \frac{1}{16}$, also $\sqrt{4\varepsilon_n(1-\varepsilon_n)} \leq \frac{1}{2}$, und

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sqrt{4\varepsilon_n(1-\varepsilon_n)}^{c_n} \leq 2^{-c_n}.$$

Damit genügt in diesem Fall eine Wiederholungszahl von $c_n \geq \log(1/\varepsilon'_n)$. Wir kümmern uns von hier an nur noch um den Fall $\frac{2-\sqrt{3}}{4} < \varepsilon_n < \frac{1}{2}$. Wir setzen $\delta_n := \frac{1}{2} - \varepsilon_n$. Dann gilt $4\varepsilon_n(1-\varepsilon_n) = 4(\frac{1}{2}-\delta_n)(\frac{1}{2}+\delta_n) = 1 - 4\delta_n^2$. Damit, wie vorher:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq (1-4\delta_n^2)^{c_n/2} \overset{\text{Prop. A.1.2}}{\leq} e^{-2\delta_n^2 \cdot c_n}.$$

Damit $\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \varepsilon'_n$ ist, genügt es,

$$e^{-2\delta_n^2 \cdot c_n} \leq \varepsilon'_n$$

zu haben, oder

$$2\delta_n^2 \cdot c_n \geq \ln(1/\varepsilon'_n) = (\ln 2)\log(1/\varepsilon'_n).$$

Hierfür hinreichend:

$$c_n \geq \frac{(\ln 2)\log(1/\varepsilon'_n)}{2\delta_n^2} = \frac{(2\ln 2)\log(1/\varepsilon'_n)}{(1-2\varepsilon_n)^2}. \tag{3.3.1}$$

Wir erhalten:

Satz 3.3.4
Wenn $\mathcal{A}$ die Funktion f mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet, wobei $0 \leq \varepsilon_n < \frac{1}{2}$ ist, und $(\varepsilon'_n)_{n\geq 0}$ mit $\varepsilon'_n < \frac{1}{2}$ gegeben ist, dann berechnet jede Version von Algorithmus 3.3.3, die $c_n \geq (2\ln 2)\cdot\log(1/\varepsilon'_n)/(1-2\varepsilon_n)^2$ Wiederholungen benutzt, die Funktion f mit Fehlerschranke $(\varepsilon'_n)_{n\geq 0}$. Der erwartete Zeitbedarf ist $\bar{t}_{\mathcal{A}'}(x) = O(c_n \cdot \bar{t}_{\mathcal{A}}(x))$, für x mit $|x| = n$.

Ebenso wie im Fall des einseitigen Fehlers kann man auch hier überlegen, welcher Aufwand nötig ist, um mit schlechten Fehlerwahrscheinlichkeiten ε_n (nahe an $\frac{1}{2}$) zurechtzukommen. Wenn $\varepsilon_n = \frac{1}{2} - \delta_n$, sind $c_n \geq (2\ln 2)\cdot 2/(2\delta_n)^2 = (\ln 2)/\delta_n^2$ Wiederholungen ausreichend, um auf Fehlerwahrscheinlichkeit $\varepsilon'_n = \frac{1}{4}$ zu kommen. Für $\varepsilon_n = 0{,}49$ sind dies z. B. etwa $(\ln 2)/\frac{1}{100^2} \approx 6932$ Wiederholungen. Bei Fehlerwahrscheinlichkeiten $\varepsilon_n \leq \frac{1}{2} - n^{-k}$ kommt man unter polynomiellem Aufwand ($\Theta(n^{2k})$ Wiederholungen)

Algorithmus 3.3.5 *Mehrheitsverfahren*

INPUT: x
METHODE:
```
1   count := 0;
2   for i := 1 to c do
3     r := A(x);
4     if count = 0
5       then count := 1; R := r
6       else
7         if r = R
8           then count := count + 1
9           else count := count − 1
10  return R.
```

zu einer Fehlerwahrscheinlichkeit $\frac{1}{4}$. Wenn wir von einem Algorithmus $\mathcal{A}$ mit Fehlerwahrscheinlichkeit $\varepsilon_n = \varepsilon = \frac{1}{4}$ ausgehen, ist es weit weniger teuer, zu recht kleinen Fehlerwahrscheinlichkeiten zu kommen: $1{,}39 \cdot 20/(1/2)^2 \approx 111$ Wiederholungen genügen, um Fehlerwahrscheinlichkeit 2^{-20} zu erreichen, und 221 Wiederholungen für eine Fehlerwahrscheinlichkeit von $2^{-40} < 10^{-12}$.

Ermittlung des Mehrheitswertes in Linearzeit. Wenn man Algorithmus 3.3.3 naiv implementiert, muss man die Resultate $r_1, \ldots, r_c$ aufbewahren und dann ein Ergebnis ermitteln, das häufiger als $\frac{c}{2}$-mal vorkommt – falls ein solches existiert. Auf den ersten Blick vermutet man, dass hier ein komplizierterer Zähl- oder Sortieraufwand nötig ist. Mit einem kleinen Trick lässt sich dies vermeiden: man muss nur ein Resultat speichern, und $c - 1$ Identitätsvergleiche zwischen Resultaten durchführen. Wir benötigen ein Register `R`, das Werte aus Z speichern kann, und einen Zähler `count`. Das Verfahren ist als Algorithmus 3.3.5 angegeben.

Solange `count` einen positiven Wert enthält, beharrt der Algorithmus auf dem gegenwärtigen Inhalt des Registers `R`. Falls dieses Resultat erneut eintrifft, gilt es als noch besser bestätigt, und `count` wird erhöht. Falls ein Ergebnis eintrifft, das vom Inhalt von `R` verschieden ist, gilt der gespeicherte Wert als weniger stark bestätigt, und `count` wird heruntergezählt. Eine Änderung von `R` erfolgt nur, wenn `count` durch

solche abweichenden Werte wieder auf 0 gefallen ist. Dann beginnt man mit einem neuen Wert von vorn.

Beispiel: Wenn die Resultate der Aufrufe von $\mathcal{A}$ auf x die Werte

$$(22, 22, 11, 33, 11, 11, 22, 33, 44, 11, 11, 22, 11, 11, 11, 33, 11)$$

sind, dann sind die Inhalte von `R` und `count` nach den Runden $i = 0, 1, \ldots, 17$:

–	22	22	22	22	11	11	11	11	44	44	11	11	11	11	11	11	11
0	1	2	1	0	1	2	1	0	1	0	1	0	1	2	3	2	3

Die Ausgabe ist 11, wie gewünscht. Würde man nach $c' = 10$ Runden abbrechen, wäre die Ausgabe 44, die keineswegs besonders häufig ist. Allerdings gibt es unter den Resultaten $r_1, \ldots, r_{10}$ auch keines, das häufiger als 5-mal auftritt.

Proposition 3.3.6
Wenn in $r_1, \ldots, r_c$ ein Wert r^* mehr als $\frac{1}{2}c$-mal vorkommt, dann gibt Algorithmus 3.3.5 diesen Wert r^* aus. (Damit gilt die Analyse aus Satz 3.3.4 ebenfalls für Algorithmus 3.3.5.)

Beweis: Es sei c_i der Stand von `count` ($0 \leq i \leq c$) und s_i der Inhalt von `R` ($1 \leq i \leq c$) nach dem i-ten Schleifendurchlauf. Wir definieren einen ganzzahligen „Pegelstand“ p_i, $i = 0, \ldots, c$: $p_0 = 0$ und

$$p_i := \begin{cases} c_i, & \text{falls } s_i = r^*, \\ -c_i, & \text{falls } s_i \neq r^*, \end{cases}$$

für $1 \leq i \leq c$. (Wenn in `R` der richtige Wert steht, zählen wir den Inhalt von `count` positiv, sonst negativ.)

Wir beobachten nun (vgl. Abb. 3.3.1):

- In jedem Schleifendurchlauf erhöht oder erniedrigt sich p_i um 1.
  (Dies folgt daraus, dass Algorithmus 3.3.5 in Zeilen 5, 8 und 9 den Wert c_i um 1 erhöht oder erniedrigt.)
- Wenn $r_i = r^*$, dann gilt $p_i = p_{i-1} + 1$.
  (1. Fall: $p_{i-1} > 0$. Dann ist $s_{i-1} = r^*$, also wird in Zeile 8 `count` um 1 erhöht.

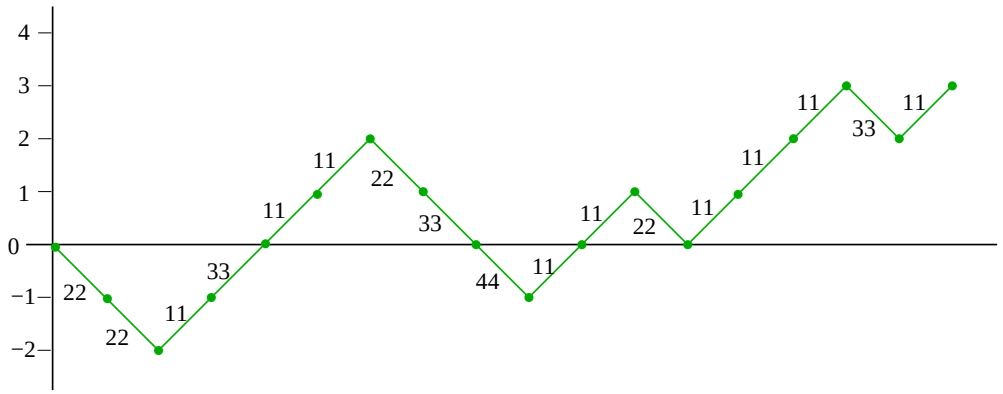

Abbildung 3.3.1: Die Pegelstände p_i, $i = 0, \ldots, 17$, zu den Ergebnissen $r_1, \ldots, r_{17}$ im Beispiel.

2. Fall: $p_{i-1} = 0$. Dann ist $c_i = 1$ und $s_i = r^*$, nach Zeile 5, also $p_i = 1$.
3. Fall: $p_{i-1} < 0$. Dann ist $s_{i-1} \neq r^*$, aber $r_i = r^*$, also ist aktuelles Ergebnis und Inhalt von `R` verschieden. Auch nach Runde i gilt $s_i \neq r^*$. Nach Zeile 9 gilt $c_i = c_{i-1} - 1$, also $p_i = -(c_{i-1} - 1) = p_{i-1} + 1$.

(Man beachte, dass ein negativer Pegelstand auch steigen kann, wenn ein falsches Resultat kommt. Der Zähler/Pegel zählt also keineswegs exakt die Anzahl der korrekten und falschen Ergebnisse oder deren Differenz.) Wir starten mit $p_0 = 0$; in mehr als $\frac{c}{2}$ Runden wächst p_i um 1; p_i fällt um 1 in weniger als $\frac{c}{2}$ Runden. Also ist $p_c > 0$, und das bedeutet, dass $s_c = r^*$ ist und der Wert r^* ausgegeben wird. □

3.3.3 Diskussion: Was bedeuten Fehlerwahrscheinlichkeiten?

Es sei $f\colon I \to Z$ eine Funktion und $\mathcal{A}$ sei ein randomisierter Algorithmus, der f mit einer Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet. Es sei n fest. Wie sollen wir diese Situation interpretieren?

1. Fehlerrate tolerierbar: Wenn wir $\mathcal{A}$ häufig (auf verschiedenen Eingaben) ausführen, dann erwarten wir, dass im Durchschnitt über diese vielen Aufrufe in nicht viel mehr als etwa $100\varepsilon_n$ Prozent der Fälle ein falsches Ergebnis ausgegeben wird. (Nach dem „Gesetz der großen Zahlen“ wird diese Vorstellung um so sicherer eintreffen, je mehr Wiederholungen stattfinden.) Man kann sich dann überlegen, welcher Prozentsatz

von falschen Resultaten toleriert werden kann, und man kann gegebenenfalls die Fehlerwahrscheinlichkeit durch Wiederholung auf demselben Input (Algorithmus 3.3.5) auf einen akzeptablen Wert senken.

2. Keine Fehler tolerierbar: Es gibt Algorithmen, bei denen das Auftreten eines Fehlers prinzipiell nicht akzeptabel ist. Manche Algorithmen werden auch nur ein einziges Mal, wenn überhaupt, ausgeführt, und es ist entscheidend, dass sie in dieser Situation korrekt funktionieren. Beispielsweise sollte ein Algorithmus, der bei einem seltenen Störfall in einem Kernkraftwerk benutzt wird, sicher funktionieren und nicht eine spürbare Versagenswahrscheinlichkeit haben. Ebenso sollten Algorithmen, die (auch wiederholt) an kritischer Stelle zur Steuerung von Flugzeugen oder bei Operationsrobotern eingesetzt werden, keine Versagenswahrscheinlichkeit einer Größe aufweisen, die erwarten lässt, dass ein solcher Fehler mitunter auftritt. Hier muss man also so vorgehen, dass man durch Wiederholung auf demselben Input (Algorithmus 3.3.5) die Fehlerwahrscheinlichkeit ε_n auf einen extrem kleinen Wert senkt. Zum Beispiel kann man ein Monte-Carlo-Verfahren mit einseitigem Fehler, das eigentlich Fehlerwahrscheinlichkeit $\varepsilon \leq \frac{1}{4}$ hat, durch nur 40-fache Wiederholung auf eine Fehlerwahrscheinlichkeit von $\varepsilon' = 2^{-80} \approx 10^{-24}$ bringen. Diese Fehlerwahrscheinlichkeit ist viel kleiner als die Wahrscheinlichkeit eines Hardwareversagens oder eines Fehlers in der Berechnung, der durch andere Einflüsse verursacht wird; sie kann im Zusammenhang mit dem Einsatz realer Rechner und realer technischer Geräte, die vielfältige andere Fehlerquellen haben, auf jeden Fall vernachlässigt werden.

Allerdings ist hier eine **Warnung** angebracht: Bei Verwendung randomisierter Algorithmen in kritischen Bereichen ist immer zu bedenken, dass die verwendeten Zufallszahlen normalerweise der Annahme der idealen Zufälligkeit nicht entsprechen, sondern von „Pseudo-Zufallszahlen-Generatoren“ bereitgestellt werden. Die Annahme, dass sich solche Zahlen wie Zufallszahlen verhalten, muss also ebenfalls immer kritisch hinterfragt werden. In dieser Vorlesung vernachlässigen wir diesen Aspekt.

3.4 Laufzeitkappung

Ziel dieses Abschnittes ist klarzumachen, dass man sich bei der Betrachtung von fehlerbehafteten Algorithmen (insbesondere Monte-Carlo-Algorithmen) im Wesentlichen auf worst-case-Laufzeitschranken beschränken kann – sie sind nicht viel schlechter als erwartete Laufzeiten. Einige technische Details sind dabei aber zu beachten.

Gegeben sei also ein randomisierter Algorithmus $\mathcal{A}$ für eine Funktion f. Die Idee

ist, die Berechnung von $\mathcal{A}$ auf einer Eingabe x abzubrechen, sobald eine gewisse Zeitschranke $s(x)$ überschritten wird. Dann wird eine beliebige Ausgabe ausgegeben. Dieses Abbrechen führt zu einer neuen Quelle für falsche Ausgaben, zusätzlich zu der, die schon in $\mathcal{A}$ liegt. Wir müssen darauf achten, dass diese neue Fehlerquelle die Fehlerwahrscheinlichkeit nicht allzu sehr erhöht.

Eine sehr naheliegende Abbruchschranke wäre $c \cdot \bar{t}_{\mathcal{A}}(x)$, für eine Konstante c. Eine kleine technische Schwierigkeit ist, dass diese Zahl normalerweise nicht bekannt und nicht leicht zu berechnen ist.

Definition 3.4.1
Eine Funktion $s\colon I \to \mathbb{N}$ heißt eine *konstruierbare Schätzfunktion* für $\bar{t}_{\mathcal{A}}(x)$, wenn $s(x)$ in Zeit $t_s(x) = O(s(x))$ berechenbar ist und $\bar{t}_{\mathcal{A}}(x) \leq s(x)$ ist für alle $x \in I$.

Die worst-case-Laufzeitschranken hängen direkt von der Schätzfunktion ab; wir sollten uns also bemühen, möglichst genaue Schätzfunktionen zu entwickeln. Dies bedeutet, dass wir $\mathcal{A}$ möglichst genau analysieren sollten. Der neue Algorithmus $\mathcal{A}'$ benutzt eine (noch zu diskutierende) Konstante $c \geq 1$.

Algorithmus 3.4.2 ***Laufzeitkappung allgemein***

INPUT: x

METHODE:
1 berechne $t := s(x)$;
2 Lasse $\mathcal{A}$ auf x für maximal $c \cdot t$ RRAM-Schritte laufen;
3 **falls** Berechnung endet: Ausgabe $\mathcal{A}(x)$;
4 **falls** Berechnung nicht endet: Ausgabe 0.

Analyse: 1. Laufzeit: Die Berechnung von $s(x)$ kostet Zeit $O(t_s(x))$; eine durch einen Schrittzähler kontrollierte Berechnung der RRAM, die spätestens nach $c \cdot s(x)$ Schritten abgebrochen wird, kostet Zeit $O(s(x))$. Als worst-case-Zeitschranke erhalten wir $O(t_s(x) + s(x)) = O(s(x))$, als Schranke für die erwartete Laufzeit: $\bar{t}_{\mathcal{A}'}(x) = O(t_s(x) + \bar{t}_{\mathcal{A}}(x))$.

2. Fehlerwahrscheinlichkeit: Wenn $\mathcal{A}'$ auf x das falsche Resultat liefert, dann hat entweder $\mathcal{A}$ bis zum Ende gerechnet, aber mit dem falschen Ergebnis, oder $\mathcal{A}$ ist nicht fertig geworden. Mit der Vereinigungsschranke und der Markov-Ungleichung erhalten

wir:

$$\begin{aligned}
\mathrm{err}_{\mathcal{A}',f}(x) &= \mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \\
&\leq \mathbf{Pr}(\mathcal{A}(x) \neq f(x)) + \mathbf{Pr}(t_{\mathcal{A}}(x) \geq c \cdot s(x)) \\
&\leq \mathrm{err}_{\mathcal{A},f}(x) + \frac{\mathbf{E}(t_{\mathcal{A}}(x))}{c \cdot s(x)} \\
&\leq \mathrm{err}_{\mathcal{A},f}(x) + \frac{\mathbf{E}(t_{\mathcal{A}}(x))}{c \cdot \mathbf{E}(t_{\mathcal{A}}(x))} \\
&= \mathrm{err}_{\mathcal{A},f}(x) + c^{-1}.
\end{aligned}$$

Im Fall eines Monte-Carlo-Algorithmus $\mathcal{A}$ mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ und einer „Wunsch-Fehlerschranke“ $(\varepsilon'_n)_{n\geq 0}$ mit $\varepsilon_n < \varepsilon'_n < \frac{1}{2}$ sollten wir $c_n = (\varepsilon'_n - \varepsilon_n)^{-1}$ wählen; dann ergibt sich $\mathrm{err}_{\mathcal{A}',f}(x) \leq \varepsilon'_n$. Damit die Laufzeit nicht zu groß wird, sollte zwischen ε_n und $\frac{1}{2}$ kein zu kleiner Abstand sein – gegebenenfalls sollte man also *vor* dem Schritt der Laufzeitkappung eine Wahrscheinlichkeitsverbesserung durchführen.

Im Fall eines Monte-Carlo-Algorithmus $\mathcal{A}$ mit einseitigem Fehler und konstanter Fehlerschranke $\varepsilon < 1$ kann man analog vorgehen. Wichtig ist hier, dass die Ausgabe im Abbruchfall 0 sein muss, so dass für Inputs x mit $f(x) = 0$ das richtige Resultat erscheint, selbst wenn die Berechnung abgebrochen wird.

3.5 Las-Vegas-Algorithmen und selbstverifizierende Algorithmen

Grundsätzlich betrachtet man bei randomisierten Algorithmen, die keine Fehler machen, also Las-Vegas-Algorithmen, die erwartete Rechenzeit. Es gibt auch eine Variante von fehlerfreien Algorithmen, die eine worst-case-Laufzeitschranke haben.

Definition 3.5.1
Ein Algorithmus $\mathcal{A} = (M, I, Z \cup \{\text{„?“}\}, \text{IN}, \text{OUT})$ heißt ein **selbstverifizierender (s.v.) Algorithmus** für $f\colon I \to Z$, wenn $\mathbf{Pr}(\mathcal{A}(x) \in \{f(x), \text{„?“}\}) = 1$ gilt; dabei ist „?“ $\notin Z$.

Ein selbstverifizierender Algorithmus gibt entweder das richtige Resultat aus oder teilt über die Ausgabe „?“ mit, wenn es ihm nicht gelungen ist, das korrekte Ergebnis zu berechnen. (Die Ausgabe „?“ kann man als „Ich weiß nicht“ lesen.)

Ziel dieses Abschnitts ist es zu zeigen, dass s.v. Algorithmen mit Versagenswahrscheinlichkeit < 1 und Las-Vegas-Algorithmen praktisch dasselbe sind. Laufzeitkappung führt uns von Las-Vegas-Algorithmen zu s.v. Algorithmen; Wiederholung macht aus s.v. Algorithmen wieder Las-Vegas-Algorithmen. Sei zunächst $\mathcal{A}$ ein Las-Vegas-Algorithmus mit einer konstruierbaren Schätzfunktion $s(x)$ für die erwartete Laufzeit $\bar{t}_{\mathcal{A}}(x)$. Die Zahl $c \geq 1$ sei eine Konstante. Folgendes ist Algorithmus $\mathcal{A}'$.

Algorithmus 3.5.2 ***Laufzeitkappung Las Vegas***

INPUT: x

METHODE:
1 berechne $t := s(x)$;
2 Lasse $\mathcal{A}$ auf x für maximal $c \cdot t$ RRAM-Schritte laufen;
3 **falls** Berechnung endet: Ausgabe $\mathcal{A}(x)$;
4 **falls** Berechnung nicht endet: Ausgabe „?“.

Analyse: 1. Laufzeit: Wie bei den Monte-Carlo-Algorithmen erhalten wir als worst-case-Zeitschranke $O(t_s(x) + s(x)) = O(s(x))$, als Schranke für die erwartete Laufzeit: $\bar{t}_{\mathcal{A}'}(x) = O(t_s(x) + \bar{t}_{\mathcal{A}}(x))$.

2. Fehlerwahrscheinlichkeit: Wenn $\mathcal{A}'$ auf x das falsche Resultat liefert, dann ist $\mathcal{A}$ nicht fertig geworden. Mit der Markov-Ungleichung erhalten wir:

$$\mathrm{err}_{\mathcal{A}',f}(x) = \mathbf{Pr}(t_{\mathcal{A}}(x) \geq c \cdot s(x)) \leq \frac{\mathbf{E}(t_{\mathcal{A}}(x))}{c \cdot s(x)} \leq c^{-1}.$$

Wenn wir also bei $\mathcal{A}'$ eine Fehlerwahrscheinlichkeit $\varepsilon < 1$ tolerieren wollen, sollten wir $c \geq 1/\varepsilon$ wählen.

Nun kommen wir zur Umkehrung. Gegeben sei also ein s.v. Algorithmus für die Funktion f. Wir nehmen an, die Laufzeit sei $\leq t(x)$ im schlechtesten Fall. Es liegt nahe, im Fall des Ergebnisses „?“ einfach den Algorithmus erneut zu starten. Dies liefert den folgenden Algorithmus $\mathcal{A}'$.

Algorithmus 3.5.3 ***Wiederholung s.v. Algorithmus***

INPUT: x

METHODE:

```
1   repeat
2     r := A(x)
3   until r ≠ „?“;
4   return r.
```

Analyse: 1. Laufzeit (Version A): Wir betrachten einen Input x und untersuchen die Anzahl Y der Schleifendurchläufe auf x. Definiere („Versagenswahrscheinlichkeit“):

$$\varepsilon_x := \mathbf{Pr}(\mathcal{A} \text{ auf } x \text{ liefert „?“}).$$

Dann gilt für $j \geq 1$:

$$\mathbf{Pr}(Y \geq j) = \mathbf{Pr}(\text{die ersten } j-1 \text{ Aufrufe liefern „?“}) = \varepsilon_x^{j-1}.$$

Mit Fakt 2.2.9 erhalten wir:

$$\mathbf{E}(Y) = \sum_{j \geq 1} \mathbf{Pr}(Y \geq j) = \sum_{j \geq 1} \varepsilon_x^{j-1} = \sum_{j \geq 0} \varepsilon_x^{j} = \frac{1}{1-\varepsilon_x}.$$

Da jeder Schleifendurchlauf Zeit $t(x) + O(1)$ kostet, ist die erwartete Laufzeit

$$\bar{t}_{\mathcal{A}'}(x) = O\Big(\frac{t(x)}{1-\varepsilon_x}\Big).$$

Diese Zeitschranke gilt, ohne dass man $t(x)$ oder ε_x kennen muss. Ein spezieller Fall liegt vor, wenn es ein $\varepsilon < 1$ gibt derart, dass $\varepsilon_x \leq \varepsilon$ für alle $x \in I$ gilt. In diesem Fall erhalten wir die Laufzeitschranke $\bar{t}_{\mathcal{A}'}(x) = O(t(x)/(1-\varepsilon))$: die erwartete Laufzeit von $\mathcal{A}'$ ist nur um einen konstanten Faktor schlechter als die (worst-case-)Laufzeit von $\mathcal{A}$.

1. Laufzeit (Version B): Eine noch präzisere (und auf keinen Fall schlechtere) Schranke liefert die folgende elegante Analyse. Wir betrachten zwei *bedingte* erwartete Zeitschranken, je eine für den Fall, dass $\mathcal{A}$ erfolgreich ist und den Fall, dass $\mathcal{A}$ das Ergebnis „?“ liefert:

- $\bar{t}(x) = \mathbf{E}(t_{\mathcal{A}}(x) \mid \mathcal{A}(x) = f(x))$;

- $\overline{s}(x) = \mathbf{E}(t_{\mathcal{A}}(x) \mid \mathcal{A}(x) = \text{„?“})$.

Wir nehmen dabei an, dass der Zusatzaufwand für die Schleifenorganisation in $\mathcal{A}'$ schon in die Laufzeit für einen Durchlauf von $\mathcal{A}$, erfolgreich oder erfolglos, mit einberechnet wurde. Wir berechnen nun $\mathbf{E}(t_{\mathcal{A}'}(x))$ wie folgt: Beim ersten Aufruf von $\mathcal{A}$ gibt es zwei Möglichkeiten: entweder $\mathcal{A}(x) = f(x)$ (Wahrscheinlichkeit $1 - \varepsilon_x$) – dann kostet der Aufruf erwartete Zeit $\overline{t}(x)$; oder $\mathcal{A}(x) = \text{„?“}$ (Wahrscheinlichkeit ε_x) – dann haben wir erwartete Zeit $\overline{s}(x)$ für den ersten Aufruf von $\mathcal{A}$ und zusätzlich die erwarteten Kosten für $\mathcal{A}'$, weil wir ja einfach die Schleife erneut starten. Das liefert:

$$\mathbf{E}(t_{\mathcal{A}'}(x)) = (1 - \varepsilon_x)\overline{t}(x) + \varepsilon_x(\overline{s}(x) + \mathbf{E}(t_{\mathcal{A}'}(x))).$$

Interessant ist, dass die unbekannte erwartete Zeit rechts wieder auftritt, aber das macht nichts, da wir die Gleichung nach $\mathbf{E}(t_{\mathcal{A}'}(x))$ auflösen können. Dies ergibt:

$$\mathbf{E}(t_{\mathcal{A}'}(x)) = \overline{t}(x) + \frac{\varepsilon_x \overline{s}(x)}{1 - \varepsilon_x}.$$

Auch diese Gleichung gilt unabhängig davon, ob wir die erwarteten Laufzeiten bei $\mathcal{A}$ kennen. Sie gibt ein recht natürliches Ergebnis: Wie wir in der ersten Laufzeitanalyse (Version A) berechnet haben, erwarten wir insgesamt $\frac{1}{1-\varepsilon_x}$ Ausführungen von $\mathcal{A}$. Davon ist eine (die letzte) erfolgreich (Ergebnis $\neq \text{„?“}$), die anderen nicht (Ergebnis „?“), und von diesen erwarten wir $\frac{1}{1-\varepsilon_x} - 1 = \frac{\varepsilon_x}{1-\varepsilon_x}$ viele.

2. Fehlerwahrscheinlichkeit: Es sei B_j das Ereignis, dass $\mathcal{A}'$ nach genau j Schleifendurchläufen anhält und das korrekte Ergebnis $f(x)$ liefert. Dann gilt:

$$\mathbf{Pr}(B_j) = \mathbf{Pr}(\text{die ersten } j-1 \text{ Versuche liefern „?“} \wedge \text{der } j\text{-te Versuch liefert } f(x)).$$

Wegen der Unabhängigkeit der Versuche und der Definition von selbstverifizierenden Algorithmen ist $\mathbf{Pr}(B_j) = \varepsilon_x^{j-1}(1-\varepsilon_x)$. Die Ereignisse B_j sind disjunkt, daher können wir addieren:

$$\mathbf{Pr}(\mathcal{A}' \text{ liefert Resultat } f(x)) = \mathbf{Pr}\Big(\bigcup_{j\geq 1} B_j\Big) = \sum_{j\geq 1} \mathbf{Pr}(B_j) = \sum_{j\geq 1} \varepsilon_x^{j-1}(1-\varepsilon_x) = 1.$$

Das heißt, dass $\mathcal{A}'$ ein Las-Vegas-Algorithmus ist.

A Registermaschinen

In diesem Abschnitt stellen wir das Modell der Registermaschine (RAM – **r**andom **a**ccess **m**achine – Maschine mit wahlfreiem Speicherzugriff) vor.

Registermaschinen operieren auf natürlichen Zahlen bzw. endlichen Folgen von Zahlen. Ihre Struktur stellt sie in die Nähe des von-Neumann-Rechenmodells. Sie haben Programme mit Speicher-, Rechen- und Sprungbefehlen und einen Speicher mit Speicherzellen, auf die mit direkter und indirekter Adressierung zugegriffen werden kann. Registermaschinen unterscheiden sich von GOTO-Programmen (Vorlesung „Automaten, Sprachen und Komplexität (ASK)“) hauptsächlich dadurch, dass es potenziell unendlich viele Speicherzellen und indirekte Adressierung gibt.

Eine *Registermaschine* (RAM) hat einerseits einen Speicher, der aus unendlich vielen *Speicherzellen* R_0, R_1, R_2, ... besteht, die man traditionell *Register* nennt. Jedes Register kann eine natürliche Zahl speichern. $\langle R_i \rangle$ bezeichnet die in R_i gespeicherte Zahl. Wir interessieren uns nur für Speicherzustände, in denen alle bis auf endlich viele Register den Inhalt 0 haben. Andererseits hat eine RAM ein Programm, das einem Maschinenprogramm in einer simplen Maschinensprache gleicht. Formal ist das Programm eine Folge $B_0, B_1, \ldots, B_{l-1}$ von l *Befehlen*, die aus einem Befehlsvorrat stammen, der Speicher- oder Kopierbefehle, arithmetische Befehle und bedingte und unbedingte Sprungbefehle enthält. B_k heißt „die k-te Programmzeile“; man stellt sich die Befehle also in l Zeilen angeordnet vor. Um die Arbeitsweise der RAM zu beschreiben, geben wir ihr noch ein zusätzliches Register, den *Befehlszähler* BZ, das ebenfalls natürliche Zahlen speichern kann. Der Inhalt von BZ wird mit $\langle \mathrm{BZ} \rangle$ bezeichnet. In Abb. A.0.2 ist der Aufbau einer RAM schematisch wiedergegeben.

RAMs mit verschiedenen Programmen werden als verschiedene Maschinen angesehen. Hat man eine RAM M (d. h. ein Programm) und stehen in BZ und $R_0, \ldots, R_m$ irgendwelche Zahlen, und in $R_{m+1}, R_{m+2}, \ldots$ Nullen, kann M einen Schritt ausführen:

> Falls $0 \leq \langle \mathrm{BZ} \rangle < l$, wird Befehl $B_{\langle \mathrm{BZ} \rangle}$ ausgeführt (dies verändert $\langle \mathrm{BZ} \rangle$ und eventuell den Inhalt eines Registers). Ist $\langle \mathrm{BZ} \rangle \geq l$, passiert nichts: die RAM hält.

Dieser Vorgang kann natürlich iteriert werden, bis möglicherweise schließlich ein Befehlszählerinhalt $\geq l$ erreicht wird. In der Tabelle auf Seite 30 beschreiben wir den

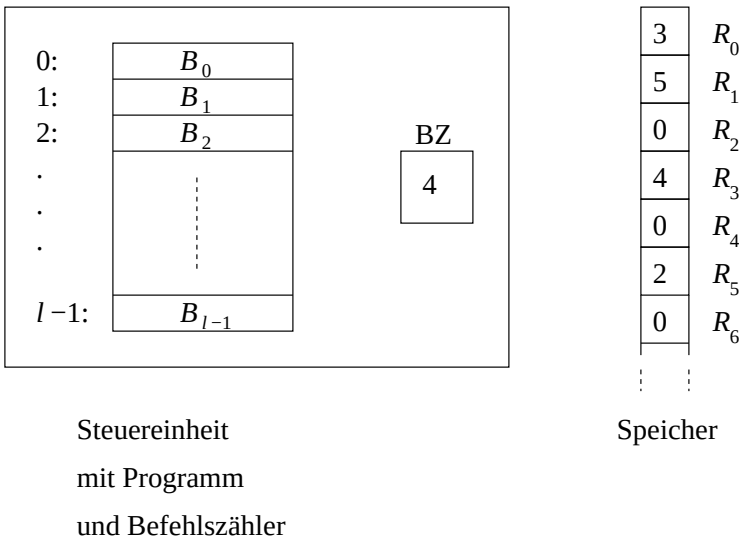

Abbildung A.0.2: Schema einer Registermaschine

Befehlssatz einer RAM. Es handelt sich um sogenannte Dreiadressbefehle: Operanden werden aus zwei Registern geholt, das Resultat einer Operation in einem dritten Register gespeichert. Nicht-Sprungbefehle führen dazu, dass der Befehlszähler um 1 erhöht wird. In der Tabelle steht das Zeichen „:=“ für eine Zuweisung: „$R_i := p$“ für einen Index $i \in \mathbb{N}$ und eine Zahl $p \in \mathbb{N}$ bewirkt, dass nachher das Register R_i die Zahl p enthält. Analog sind Zuweisungen „BZ $:= m$“ zu interpretieren. Bei der Subtraktion werden negative Resultate durch 0 ersetzt; die Division ist die ganzzahlige Division ohne Rest. Division durch 0 führt zum Anhalten.

Mitunter betrachtet man RAMs mit eingeschränktem Befehlsvorrat, z. B. $\{+, -\}$-RAMs, bei denen die $*$- und $\div$-Befehle fehlen. Da man Multiplikation und Division durch Teilprogramme ersetzen kann, die nur Addition und Subtraktion benutzen, bedeutet dies keine prinzipielle Einschränkung; allerdings kann sich die Anzahl der für eine Berechnung nötigen Schritte erhöhen.

Befehlszeile	**Einschränkungen, Beschreibung**	**Wirkung**
Speicherbefehle:		
$R_i \leftarrow R_j$	$i, j \in \mathbb{N}$ ($*$ Register kopieren mit direkter Adressierung $*$)	$R_i := \langle R_j \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_{R_i} \leftarrow R_j$	$i, j \in \mathbb{N}$ ($*$ Register kopieren; Ziel indirekt adressiert $*$)	$R_{\langle R_i \rangle} := \langle R_j \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_{R_j}$	$i, j \in \mathbb{N}$ ($*$ Register kopieren; Quelle indirekt adressiert $*$)	$R_i := \langle R_{\langle R_j \rangle} \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
Arithmetische Befehle:		
$R_i \leftarrow k$	$i, k \in \mathbb{N}$ ($*$ Konstante laden $*$)	$R_i := k$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j + R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, addieren, speichern $*$)	$R_i := \langle R_j \rangle + \langle R_k \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j - R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, subtrahieren, speichern; negatives Resultat durch 0 ersetzen $*$)	$R_i := \max\{0, \langle R_j \rangle - \langle R_k \rangle\}$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j * R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, multiplizieren, speichern $*$)	$R_i := \langle R_j \rangle \cdot \langle R_k \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j \div R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, dividieren, speichern $*$) ($*$ Fehlerhalt $*$)	if $\langle R_k \rangle > 0$ then $\{R_i := \langle R_j \rangle$ div $\langle R_k \rangle$; BZ $:= \langle$BZ$\rangle + 1\}$ else BZ $:= l$
Sprungbefehle:		
goto m	$m \in \mathbb{N}$ ($*$ unbedingter Sprung $*$)	BZ $:= m$
if $(R_i = 0)$ goto m	$i, m \in \mathbb{N}$ ($*$ bedingter Sprung $*$)	BZ $:= \begin{cases} m, \text{ falls } \langle R_i \rangle = 0; \\ \langle \text{BZ} \rangle + 1, \text{ sonst.} \end{cases}$
if $(R_i > 0)$ goto m	$i, m \in \mathbb{N}$ ($*$ bedingter Sprung $*$)	BZ $:= \begin{cases} m, \text{ falls } \langle R_i \rangle > 0; \\ \langle \text{BZ} \rangle + 1, \text{sonst.} \end{cases}$
STOP:		
($*$ bedingte oder unbedingte Sprünge „goto m“ mit Sprungziel $m \geq l$ wirken wie bedingte oder unbedingte STOP-Befehle. Die RAM hält auch, wenn der Befehl in Zeile $l - 1$ ausgeführt wird und es sich nicht um einen Sprungbefehl handelt. $*$)		

Konfigurationen: Der „innere Zustand“ einer Registermaschine M lässt sich durch die Angabe des Befehlszählerstandes und der Registerinhalte vollständig angeben. Wir definieren: Eine *Konfiguration* k ist ein Paar (z, α) aus einer natürlichen Zahl z und einer Funktion $\alpha\colon \mathbb{N} \to \mathbb{N}$, die fast überall den Wert 0 hat, d. h., es existiert n_α mit der Eigenschaft, dass für $i > n_\alpha$ stets $\alpha(i) = 0$ gilt. (Die Menge $\mathcal{K}$ aller Konfigurationen ist abzählbar unendlich. Sie hängt nicht von M ab.)

Schritt, Nachfolgekonfiguration, Haltekonfiguration: Sei eine RAM M (d. h. ein Programm) gegeben. Eine Konfiguration k' heißt *Nachfolgekonfiguration* von k, in Zeichen $k \vdash_M k'$ oder $k \vdash k'$, wenn ein Schritt von M in der oben beschriebenen Weise von k zu k' führt. Jede Konfiguration hat höchstens eine Nachfolgekonfiguration. Es kann sein, dass es keine Nachfolgekonfiguration von k gibt, weil in k eine Division durch 0 auszuführen wäre („Fehlerkonfiguration“). Die andere, „normale“ Möglichkeit ist, dass $k = (z, \alpha)$ für ein $z \geq l$ ist, wo l die Zeilenanzahl von M ist. Solche Konfigurationen heißen *Haltekonfigurationen*.

Startkonfiguration: Eingaben für Registermaschinen sind grundsätzlich n-Tupel $(a_0, \ldots, a_{n-1})$ von natürlichen Zahlen. Einem solchen n-Tupel entspricht die folgende *Startkonfiguration* (andere Konventionen sind möglich): Ist die Eingabe $(a_0, \ldots, a_{n-1}) \in \mathbb{N}^n$, so ist die Startkonfiguration gleich $(0, \alpha)$ mit

$\alpha(0) = n$,
$\alpha(2i + 1) = a_i$, für $0 \leq i \leq n - 1$,
$\alpha(j) = 0$ für alle anderen j.

Damit hat man die Register R_2, R_4, ... zur freien Verfügung. Der Befehlszähler hat anfangs den Inhalt 0. Für die Ausgabe wird die entsprechende (umgekehrte) Konvention benutzt: Als Resultat nach dem Anhalten gilt das Zahlentupel

$$(\alpha(1), \alpha(3), \ldots, \alpha(2\alpha(0) - 1)).$$

Gegeben eine Startkonfiguration $k_0 = (0, \alpha)$ und eine RAM M, gibt es eine eindeutig bestimmte Konfigurationenfolge

$$k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots,$$

die entweder endlich oder unendlich ist. Diese stellt die *Berechnung* von M auf der durch die Startkonfiguration gegebenen Eingabe dar. Diese Berechnung ist die formale Entsprechung der folgenden informalen Beschreibung der Arbeitsweise einer RAM:

① man schreibt $a_0, \ldots, a_{n-1}$ gemäß der Eingabekonvention in die Register von M und setzt BZ auf 0.

② while $0 \leq \langle \mathrm{BZ} \rangle < l$ do
„führe Befehl $B_{\langle \mathrm{BZ} \rangle}$ aus“
(mit der Wirkung wie in der Tabelle angegeben)

③ falls und sobald in ② eine Situation mit $\langle \mathrm{BZ} \rangle \geq l$ erreicht wird, wird aus den Registerinhalten gemäß der Ausgabekonvention das Resultat abgelesen.

Beispiel A.0.1
Wir wollen aus $a_0, \ldots, a_{n-1}$ die Summe $a_0 + \cdots + a_{n-1}$ und das Produkt $a_0 \cdots a_{n-1}$ berechnen. Die Register mit geraden Indizes werden verwendet wie folgt:
– in Register R_2 wird von n nach 0 heruntergezählt,
– in R_4 wird die Summe und in R_6 das Produkt akkumuliert,
– R_8 enthält den Index des nächsten zu verarbeitenden Inputregisters,
– in R_{10} wird der Inhalt dieses Registers zwischengespeichert,
– R_{12} enthält die Konstante 1,
– R_{14} die Konstante 2.

Zeile	Befehl	Kommentar
0	$R_{12} \leftarrow 1$	Konstante
1	$R_{14} \leftarrow 2$	laden
2	$R_2 \leftarrow R_0$	n ins Zählregister
3	$R_4 \leftarrow 0$	Initialisiere Teilsumme
4	$R_6 \leftarrow 1$	und Teilprodukt
5	$R_8 \leftarrow 1$	Indexregister auf R_1 stellen
6	if $(R_2 = 0)$ goto 13	Schleife: Zeilen 6–12
7	$R_{10} \leftarrow R_{R_8}$	Operanden holen
8	$R_4 \leftarrow R_4 + R_{10}$	addieren
9	$R_6 \leftarrow R_6 * R_{10}$	multiplizieren
10	$R_8 \leftarrow R_8 + R_{14}$	Indexregister um 2 erhöhen
11	$R_2 \leftarrow R_2 - R_{12}$	Zähler dekrementieren
12	goto 6	zum Schleifentest
13	$R_1 \leftarrow R_4$	Ausgabeformat
14	$R_3 \leftarrow R_6$	herstellen:
15	$R_0 \leftarrow 2$	2 Ausgabewerte

Beispiel A.0.2
Berechnung von $a_0^{a_1}$. Dabei ist nur das Verhalten auf zweistelligen Eingaben (a_0, a_1) interessant.

(Naive) Idee: a_1-faches Multiplizieren von a_0 mit sich selbst. Realisiert wird dies durch eine Schleife, in der R_3 in Einerschritten von a_1 bis 0 heruntergezählt wird. Das Programm für M sieht folgendermaßen aus:

Zeile	Befehl	Kommentar
0	$R_2 \leftarrow 1$	Konstante 1
1	$R_4 \leftarrow 1$	a_0^0
2	if $(R_3 = 0)$ goto 6	Zeilen 2–5:
3	$R_4 \leftarrow R_4 * R_1$	Schleife
4	$R_3 \leftarrow R_3 - R_2$	
5	goto 2	
6	$R_1 \leftarrow R_4$	Resultatformat
7	$R_0 \leftarrow 1$	herstellen

Auf der Eingabe $(a_0, a_1) = (6, 3)$ läuft das Programm wie folgt ab. Die Zeile zu Schritt t repräsentiert dabei den gesamten Zustand (die „Konfiguration“ der RAM) nach Schritt $t = 0, 1, 2, \ldots$. Schritt 0 entspricht dem Anfangszustand, nach der Initialisierung.

Schritt-Nr.	R_0	R_1	R_2	R_3	R_4	$\langle \mathrm{BZ} \rangle$
0	2	6	0	3	0	0
1	2	6	1	3	0	1
2	2	6	1	3	1	2
3	2	6	1	3	1	3
4	2	6	1	3	6	4
5	2	6	1	2	6	5
6	2	6	1	2	6	2
7	2	6	1	2	6	3
8	2	6	1	2	36	4
9	2	6	1	1	36	5
10	2	6	1	1	36	2
11	2	6	1	1	36	3
12	2	6	1	1	216	4
13	2	6	1	0	216	5
14	2	6	1	0	216	2
15	2	6	1	0	216	6
16	2	216	1	0	216	7
17	1	216	1	0	216	8

Als *Rechenzeit* der RAM M auf Eingabe $(a_0, \ldots, a_{n-1})$ sehen wir die Anzahl der Schritte an, die M auf dieser Eingabe ausführt. Diese Rechenzeit kann auch „unendlich“ sein, wenn die Berechnung nicht anhält. Man sollte bemerken, dass dieses „uniforme Kostenmaß“ nicht ganz fair ist, wenn durch Multiplikationen in wenigen Rechenschritten sehr große Zahlen erzeugt werden. Man beschränkt daher oft die erlaubte Anzahl der Bits der in den Registern gespeicherten Zahlen auf $O(\log(|x|))$, wobei $|x|$ die „(Darstellungs-)Größe“ von Input x ist.

Bemerkung A.0.3
Wir betrachten (idealisierte) Programme in einer Standard-Programmiersprache – der Eindeutigkeit halber nehmen wir etwa Java. Wir können uns alle Zahltypen wie ganze Zahlen oder reelle Zahlen durch Tupel von natürlichen Zahlen repräsentiert denken, Characters ebenfalls durch Zahlen, Zeiger durch Arrayindizes. Übersetzen wir das Programm mittels eines Compilers in Maschinensprache, ergibt sich ein Programm, das relativ leicht in ein RAM-Programm zu transformieren ist. Wir stellen daher fest: Mit Standard-Programmiersprachen berechenbare Funktionen sind auch auf einer RAM berechenbar. – Umgekehrt kann man sich fragen, ob alle RAM-berechenbaren Funktionen von Java-Programmen berechnet werden können. Diese Frage ist dann

zu verneinen, wenn man die Endlichkeit von Rechnern in Betracht zieht, die dazu führt, dass für jedes auf einer festen Maschine ausführbare Java-Programm eine größte darstellbare Zahl und eine Maximalzahl von verwendbaren Registern gegeben ist. Betrachtet man „idealisierte“ Programme und Computer ohne solche Einschränkungen, lässt sich tatsächlich die Äquivalenz beweisen. Wir geben uns damit zufrieden, im RAM-Modell eine Abstraktion zu erkennen, die auf jeden Fall die Fähigkeiten konkreter Rechner und Programmiersprachen umfasst, einschließlich von Rechenzeitbetrachtungen.