## Gliederung des Kapitels

### 1.6 Komplexität von Algorithmen und Programmen


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




## 1.6.4 Beispiele

Wir haben bereits die Komplexität einzelner Algorithmen betrachtet.

Am ausführlichsten geschah dies für die Intervallschachtelung (schnelle Suche in einem sortierten Feld), siehe 1.3.2.3.d. Dort hatten wir auf Grund der Analyse im Mittel (average case) den Algorithmus SEARCH1 zur Prozedur SEARCH2 abgewandelt, die im Durchschnitt um 50% schneller arbeitet als das erste Verfahren.

Schauen Sie sich jenes Beispiel nochmals genau an. Eventuell rechnen Sie auch noch einmal das Beispiel zu Beginn von Abschnitt 1.2.3 durch.



1.6.4.1 Größter gemeinsamer Teiler (schon wieder)

Zum ggT (englisch: *gcd* greatest common divisor) siehe bisher:
1.1.2.3: Eigenschaften und Programm euklid1,
1.2.4.1: Beschreibung durch Gleichungen,
1.4.2.1: Darstellung als Funktion (in Ada),
1.4.2.6: Rekursive Darstellung (auch in Ada).

Frage: Wie lange rechnet der Euklidische Algorithmus, bis er anhält?

Beispielrechnung mit a = 144, b = 89:

$$ggT(144,89) = ggT(89,55) = ggT(55,34) = ggT(34,21)$$
$$= ggT(21,13) = ggT(13,8) = ggT(8,5) = ggT(5,3)$$
$$= ggT(3,2) = ggT(2,1) = ggT(1,1) = ggT(1,0) = 1.$$

Dieses Beispiel benötigt 11 Schleifendurchläufe.



*Algorithmus:* -- es ist hier ggT(a,0) = a, auch für a=0.

```
var A, B, R: natural;
begin  read (A); read (B);
       while B≠0 do R:=A mod B; A:=B; B:=R od;
       write (A)
end
```

Werteverlauf für die Variablen A und B bei Eingabe von a und b:

| A | B |
|---|---|
| a | b |
| b | a mod b |
| a mod b | b mod (a mod b) |
| b mod (a mod b) | (a mod b) mod (b mod (a mod b)) |

Kann man etwas über die Größe dieser Werte sagen?



Hilfssatz:

b mod (a mod b) < b/2, für alle a ≥ b > 0, a mod b ≠ 0, d.h.,

nach spätestens zwei Schleifendurchläufen hat sich der Wert von B mehr als halbiert.

Beweis: Es ist immer 0 ≤ a mod b < b.

Fall 1: 0 ≤ a mod b ≤ b/2.
Dann gilt: b mod (a mod b) < a mod b ≤ b/2,
da für y > 0 stets x mod y < y ist.

Fall 2: b/2 < a mod b < b.
Dann gilt: b mod (a mod b) = b - (a mod b) < b - b/2 = b/2,
da für x/2 < y < x stets x mod y = x - y ist.

Damit ist der Hilfssatz bewiesen.



Folgerung:

Der Euklidische Algorithmus durchläuft höchstens 2·log(b) mal seine Schleife.

Die uniforme Zeitkomplexität des Euklidischen Algorithmus

```
begin  read (A); read (B);
       while B≠0 do R:=A mod B; A:=B; B:=R od;
       write (A)
end
```

beträgt somit höchstens 8·log(b) + 4 Zeiteinheiten.
Da log(a) + log(b) = n die Länge der Eingabe ist und log(b) somit in der Größenordnung von n liegt, so folgt:
Der Euklidische Algorithmus besitzt eine lineare uniforme Zeitkomplexität (d.h., er liegt in DTime (n)). Diese günstige Komplexitätsklasse liegt aber an der Uniformität.

Schwäche der uniformen Zeitkomplexität:

Sie misst, wie viele elementare Anweisungen ausgeführt und wie viele Ausdrücke ausgewertet werden, aber sie gibt keine Auskunft darüber, wie aufwändig die Durchführung einer elementaren Anweisung oder die Ausrechnung eines Ausdrucks ist.

Wie lange dauert denn die Wertzuweisung R := A mod B ? Der ungünstigste Fall liegt vor, wenn A k Stellen und B ungefähr k/2 Stellen lang ist. Im Binärsystem führt man dann k/2 Subtraktionen der Länge k/2 aus, d.h., die Berechnung des Restes A mod B erfordert zeichenweise $O(k^2)$ Schritte.
Da k beim Euklidischen Algorithmus in der Größenordnung von n liegt (k = log(a)), würde der obige Algorithmus also in Wahrheit $O(n^3)$ Schritte erfordern.



Genau dieses Ergebnis würde die Komplexitätsuntersuchung mit Hilfe von Turingmaschinen ergeben, d.h., der Euklidische Algorithmus liegt in DTime$^{TM}(n^3)$.

Dennoch hat die uniforme Komplexität gewisse Vorteile. In den meisten praktischen Fällen wird die mod-Funktion durch einen Coprozessor für natürliche Zahlen, die in einem "normalen" Bereich liegen, oder mit Hilfe einer Software-simulation berechnet. In solchen Fällen scheint die Modulo-Bildung in konstanter Zeit abzulaufen, so dass sich der Euklidische Algorithmus in der Praxis meist wie ein Linear-zeitalgorithmus verhält.

■



### 1.6.4.2 Multiplikation natürlicher Zahlen

Wie lange dauert die ziffernweise Multiplikation, die wir in der Schule kennen gelernt haben? Dort wird für die Multiplikation a·b ein Schema aufgeschrieben, das an ein Parallelogramm erinnert; dessen Fläche ist log(a)·log(b). Wenn beide Zahlen a und b ungefähr n/2 Stellen besitzen, so dauert die Multiplikation also $O(n^2)$ Schritte.

Als Programm nimmt man gewisse Modifikationen vor, um statt des Parallelogramms immer nur eine Zeile bzw. eine Variable mitzuführen. Eine einfache Technik beruht auf den Formeln: Sei Z (anfangs 0) das angestrebte Ergebnis, so gilt

$Z + A \cdot B = Z + (2A) \cdot (B/2)$, sofern B gerade ist,
$Z + A \cdot B = (Z + A) + A \cdot (B-1)$, sofern B ungerade ist.

Dies ergibt folgendes Verfahren:



*Algorithmus:*

```
var A, B, Z: Natural;
begin  read(A); read(B);
       Z:=0;
       while B > 0 do
          if (B mod 2 = 0) then B := B div 2; A := A+A
          else B := B-1; Z := Z+A fi od;
       write(Z)
end
```

Wenn B>0 ist, so beginnt die Binärdarstellung von B mit einer 1 und somit ist B im letzten Schleifendurchlauf 1; dort wird also spätestens der Wert von Z verändert.

*Hinweis:* Die Schleifeninvariante lautet {Z+A·B = a·b}, wenn a und b die eingelesenen Anfangswerte sind. D.h.: Der obige Algorithmus berechnet korrekt das Produkt.



*Wie lange dauert dieser Algorithmus für die Eingaben a, b?*
Die uniforme Zeitkomplexität liefert:

```
var A, B, Z: Natural;
begin  read(A); read(B);      2 Schritte
       Z:=0;                  1 Schritt
       while B > 0 do         1 Schritt                k+1 mal
          if (B mod 2 = 0) then    1 Schritt
               B := B div 2; A := A+A      2 Schritte
          else                                          k mal
               B := B-1; Z := Z+A  fi  od;  2 Schritte
       write(Z)               1 Schritt
end
```

Insgesamt: 4k + 5 Schritte, mit k = log(b) und wegen k ≈ n/2 liegt dies in DTime(n).



Schauen wir uns nun aber die elementaren Anweisungen im Detail an. Wir nehmen an, alle Zahlen seien binär dargestellt. Die Eingabelänge sei n = log(a) + log(b). Damit ist auch die Ausgabe durch n Stellen beschränkt.

Die Anweisungen read(A), read(B) und write(Z) dauern zeichenweise so lange, wie die Zahlen lang sind, also O(n). Z:=0 dauert einen Schritt. Ebenso kann man "B>0" in einem Schritt entscheiden. Der Ausdruck (B mod 2 = 0) kostet einen Schritt, da man nur das letzte Zeichen von B auf Null testen muss. B := B div 2 und A := A+A kosten ebenfalls jeweils einen Schritt, da nur die letzte Null gestrichen bzw. eine Null angehängt werden muss. Auch B := B-1 kostet nur einen Schritt, da B ja ungerade ist und daher nur die letzte 1 in eine 0 umgewandelt werden muss. Z := Z+A kann dagegen bis zu n Schritte dauern. Es ist also der else-Zweig, der die Laufzeit im Wesentlichen bestimmt.

In den else-Zweig gelangt man immer, wenn B ungerade ist. Dies ist genau der Fall, wenn in der Binärdarstellung von B am Ende eine 1 steht. Da im then-Zweig die letzte Ziffer von B jeweils gestrichen wird, so wird der else-Zweig also genau so oft durchlaufen, wie Einsen in der Binärdarstellung von b auftreten. Der then-Zweig dagegen wird genau so oft durchlaufen, wie b lang ist, also log(b) mal.

Es sei also eins(b)

$1 \le \text{eins}(b) \le \log(b) \in O(n)$

die Anzahl der Einsen in der Binärdarstellung der Eingabezahl b, dann lautet die zeichenweise Zeitkomplexität des obigen Algorithmus zur Multiplikation

$\text{eins}(b)\cdot n + 3\cdot n + 3\cdot\text{eins}(b) + 5 \in O(n^2)$ Schritte.

Dies ist auch die Turingmaschinenkomplexität, d.h., unser Multiplikationsalgorithmus liegt in $\text{DTime}^{TM}(n^2)$.



Nochmals der Hinweis:
Berechnet man die Komplexität $t_A(w)$ bzw. $s_A(w)$ so, dass jede elementare Anweisung und jede Abfrage genau einen Schritt dauern, bzw. dass jede Zahl genau einen Speicherplatz belegt, so spricht man von der **uniformen Komplexität**.
Berechnet man die Komplexitäten so, dass jede Operation so viel Zeit kostet, wie die zeichenweise Ausführung erfordert, bzw. dass Werte so viel Platz belegen, wie sie Zeichen haben, so spricht man von der **logarithmischen Komplexität**.

In der Praxis berechnet man zunächst die uniforme Komplexität: Sie gibt meist eine hinreichend genaue Orientierung über den zu erwartenden Aufwand. Erst für eine genaue Abschätzung bzw. bei der Programmierung betrachtet man die logarithmische Komplexität, die den tatsächlichen Aufwand genauer wiedergibt.



*Gibt es einen schnelleren Algorithmus für die Multiplikation?*
Oder kann man beweisen, dass es kein schnelleres Verfahren geben kann?
Wenn man zwei natürliche Zahlen (ungleich Null), die jeweils k bzw. m Ziffern lang sind, miteinander multipliziert, so entsteht eine Zahl, die (k+m-1) oder (k+m) Ziffern besitzt.
Da also das Ergebnis der Multiplikation nur so lang sein kann, wie die beiden Multiplikanten zusammen, könnte es eventuell einen Algorithmus geben, der die Multiplikation proportional zu n durchführt.
Dies würde bedeuten, dass man die Multiplikation bis auf einen konstanten Faktor genau so schnell ausführen könnte wie die Addition.
Das glaubt eigentlich niemand. Dennoch ist es bisher nicht gelungen zu beweisen, dass die Multiplikation deutlich mehr Zeit als die Addition erfordert.



Nun wollen wir ein Verfahren vorstellen, welches schneller als mit der Zeitkomplexität $O(n^2)$ multipliziert.

Gegeben seien zwei k-stellige natürliche Zahlen x und y (mit $k \approx n/2$); die Länge k sei eine gerade Zahl. Setze p=k/2. Dann lassen sich x und y schreiben in der Form:

$x = A\cdot 2^p + B$ und $y = C\cdot 2^p + D$,

wobei A, B, C, D mindestens 0 und echt kleiner als $2^p$ sind. Dann gilt

$x \cdot y = (A\cdot 2^p + B)\cdot(C\cdot 2^p + D) = A\cdot B\cdot 2^{2p} + (A\cdot D + B\cdot C)\cdot 2^p + B\cdot D.$

Man kann also die Multiplikation zweier k-stelliger Zahlen durchführen, indem man sie auf 4 Multiplikationen von halber Länge k/2 und drei Additionen zurückführt. Dies ergibt erneut eine quadratische Zeitkomplexität. Kommt man vielleicht mit weniger als vier Multiplikationen halber Länge aus?



Es gilt: $(A\cdot D + B\cdot C) = (A-B)\cdot(D-C) + A\cdot C + B\cdot D$,
wie man durch Ausrechnen leicht nachprüft.

Somit erhalten wir aus obiger Gleichung:

$x \cdot y = A\cdot C\cdot 2^{2p} + (A\cdot D + B\cdot C)\cdot 2^p + B\cdot D$
$= A\cdot C\cdot 2^{2p} + ((A-B)\cdot(D-C) + A\cdot C + B\cdot D)\cdot 2^p + B\cdot D$

wobei A, B, C, D mindestens 0 und echt kleiner als $2^p$ sind.

Nun haben wir es geschafft: Auf der rechten Seite stehen nur noch drei verschiedene Multiplikationen, nämlich:

$A\cdot C$, $B\cdot D$ und $(A-B)\cdot(D-C)$

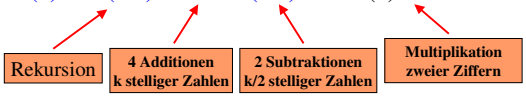
*Beachte:* Die Multiplikation mit $2^p$ bzw. $2^{2p}$ ist keine echte Multiplikation, sondern nur ein Anhängen von p bzw. 2p Nullen.



Somit haben wir die Multiplikation zweier k-stelliger Zahlen auf drei Multiplikationen von k/2-stelligen Zahlen zurückgeführt. Der Preis, den wir dafür bezahlen müssen, sind zwei zusätzliche Subtraktionen zweier k/2-stelliger Zahlen und eine zusätzlich Additionen zweier k-stelliger Zahlen. Da aber die Zeit für die Addition und die Subtraktion proportional zur Länge der Zahlen ist, müssten wir dennoch schneller fertig werden. Wir prüfen dies nach.

Wenn $t(k)$ die Anzahl der durchzuführenden Operationen für die Multiplikation zweier k-stelliger Zahlen nach diesem Verfahren ist, dann erhalten wir also folgende Gleichung:

$t(k) = 3\cdot t(k/2) + 4\cdot k + 2\cdot(k/2)$ mit $t(1)=1$



Wie lautet die Lösung dieser Gleichung?

Probieren wir es aus. Ersetzen von t(k/2), t(k/4) usw. entsprechend der Rekursionsformel t(k) = 3·t(k/2) + 5·k ergibt:

$$
\begin{aligned}
t(k) &= 3\cdot t(k/2) + 5\cdot k \\
&= 3\cdot(3\cdot t(k/4) + 5\cdot k/2) + 5\cdot k \\
&= 3\cdot 3\cdot t(k/4) + 3\cdot 5\cdot k/2 + 5\cdot k \\
&= 3\cdot 3\cdot t(k/4) + 5\cdot k\cdot(1 + 3/2) \\
&= 3\cdot 3\cdot(3\cdot t(k/8) + 5\cdot k/4) + 5\cdot k\cdot(1 + 3/2) \\
&= 3\cdot 3\cdot 3\cdot t(k/8) + 5\cdot k\cdot(1 + 3/2 + 9/4) \\
&= 3\cdot 3\cdot 3\cdot(3\cdot t(k/16) + 5\cdot k/8) + 5\cdot k\cdot(1 + 3/2 + 9/4) \\
&= 3\cdot 3\cdot 3\cdot 3\cdot t(k/16) + 5\cdot k\cdot(1 + 3/2 + 9/4 + 27/8) \\
&= \ldots
\end{aligned}
$$



Die allgemeine Form nach i Schritten lautet daher:

$$
\begin{aligned}
t(k) &= 3^i\cdot t(k/2^i) + 5\cdot k\cdot(1 + 3/2 + 9/4 + \ldots + 3^{i-1}/2^{i-1}) \\
&= 3^i\cdot t(k/2^i) + 10\cdot k\cdot((3/2)^i - 1)
\end{aligned}
$$

Beachte die geometrische Reihe

$$1 + a + a^2 + a^3 + \ldots + a^m = \frac{a^{m+1} - 1}{a - 1}$$

In unserem Fall ist a = 3/2.



Diese Ersetzungen kann man vornehmen, bis $k = 2^i$ geworden ist, also bis $i = \log(k)$. Dann ist $t(k/2^{\log(k)}) = t(1) = 1$. Wir setzen dies ein und erhalten:

$$
\begin{aligned}
t(k) &= 3^{\log(k)}\cdot t(k/2^{\log(k)}) + 10\cdot k\cdot((3/2)^{\log(k)} - 1) \\
&= k^{\log(3)} + 10\cdot k\cdot(k^{\log(1,5)} - 1) \\
&= 11\cdot k^{\log(3)} - 10\cdot k \;\approx\; 11\cdot k^{1,585} - 10\cdot k \in O(k^{1,585})
\end{aligned}
$$

*Beachte* hierbei die Formeln:
log ist hier der Logarithmus zur Basis 2,

$a^{\log(b)} = b^{\log(a)}$ und

$k\cdot k^{\log(1,5)} = k^{1+\log(1,5)} = k^{\log(2)+\log(1,5)} = k^{\log(2\cdot 1,5)} = k^{\log(3)}$



Also gilt:

Die Multiplikation zweier k-stelliger Zahlen lässt sich zeichenweise in proportional zu $k^{1,585}$ Schritten durchführen.

(Die Schulmethode benötigt $k^2$ Schritte, siehe früher.)

Geht es noch schneller? Ja. Der beste derzeit bekannte Algorithmus, der Algorithmus nach Schönhage und Strassen (1971), der sich allerdings nur für riesige Zahlen eignet, benötigt
$O(k\cdot\log(k)\cdot\log(\log(k)))$ Schritte.

Dies ist schon relativ dicht an der Größenordnung O(k), so dass man vielleicht eines Tages doch einen Algorithmus finden wird, der die Multiplikation in O(k) Schritten - und damit ungefähr so schnell wie eine Addition - durchführt?



Anmerkung: Ab wann lohnt sich dieses rekursive Verfahren? Hierfür müssen wir wissen, wie aufwändig eine normale Multiplikation genau ist, also einschließlich der Konstanten. Wenn k = log(b) = log(a) = n/2 ist, so benötigt der zuerst angegebene Algorithmus von Folie 62 höchstens $k\cdot n + 3\cdot n + 3\cdot k + 5 = 2\cdot k^2 + 9\cdot k + 5$ Schritte. (Bitte im Detail selbst nachrechnen: Alles geht in einem Schritt, nur bei "Z:=Z+A" können bis zu n-stellige Zahlen auftreten.)

Dann läuft die Frage, ab wann sich die rekursive Methode lohnt, auf die Frage hinaus, ab welchem k gilt: $11\cdot k^{1,585} - 10\cdot k < 2\cdot k^2 + 9\cdot k + 5$ ? Dies trifft schon für relativ kleine Werte von k zu, also etwa ab k = 15. Die Methode kann sich also lohnen, allerdings haben wir die Verwaltung der Zwischenergebnisse bisher nicht berücksichtigt. Eine ganz exakte Analyse müsste *alle* auftretenden Operationen im übersetzten Programm (einschl. der FIFO-Verwaltung) einbeziehen.

■



1.6.4.3 Durchlauf von Graphen

Vergleiche hierzu 1.3.3.4. Dort wurden Graphen definiert und Adjazenzlisten vorgestellt.
1.1.5.6, 1.1.5.7: Dort wurden Bäume eingeführt. Dies sind spezielle zyklenfreie Graphen mit genau einer Wurzel. Zur Darstellung von Bäumen siehe auch 1.3.3.3.

Wir wollen mit einer Prozedur GD ("Graphdurchlauf") einen Graphen durchlaufen und dabei jeden Knoten und jede Kante besuchen. Zu den Knoten werden wir häufiger gelangen, jede Kante wird aber nur genau einmal durchlaufen.

Hierfür benötigen wir einen Booleschen Wert "besucht" in jedem Knoten, der uns sagt, ob wir diesen Knoten bereits besucht haben oder nicht. Üblicherweise definiert man den Durchlaufalgorithmus rekursiv:

Initialisierung: Setze alle besucht-Werte auf false. Durchlaufe dann alle Knoten entlang der Knotenliste. Bei jedem Knoten u mache man Folgendes (Prozeduraufruf GD):

Falls u als "besucht" markiert ist, so mache nichts.

Sonst: Markiere den Knoten u als "besucht".
Bearbeite den Knoten u.
Für alle von u ausgehenden Kanten e:
Folge der Kante e zu ihrem Endknoten v
und rufe die Prozedur GD rekursiv mit v auf.

Auf diese Weise wird jeder Knoten so oft besucht, wie Kanten zu ihm führen, und einmal mehr, weil zusätzlich die Knotenliste durchlaufen wird.

Um den Durchlauf zu programmieren, müssen wir Knoten und Kanten deklarieren (wie in 1.3.3.4, in Ada).



```
type Knoten;  type Kante;
type NextKnoten is access Knoten;
type NextKante is access Kante;
type Knoten is record
        Id: Knotenname;
        besucht: Boolean; zahl1, zahl2: integer;
        Inhalt: <weitere Komponenten>;
        NKn: NextKnoten;
        EIK: NextKante;
end record;
type Kante is record
        W: <Typ des Gewichts der Kanten>;
        EKn: NextKnoten;
        NKa: NextKante;
end record;
```

Der Graph mit n Knoten und m Kanten liege als Adjazenzliste vor. Auf die Knotenliste verweist hierbei eine Variable mit dem Namen "Anfang".

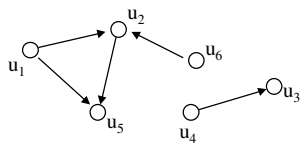


*Algorithmus Graphdurchlauf:*

```
...
Anfang, p: Nextknoten;
procedure GD (u: in out NextKnoten) is
v: NextKnoten; e: NextKante;
begin if not u.besucht then
      u.besucht := true; < "bearbeite den Knoten u" >;
      e := u.EIK;
      while e /= null loop  < "bearbeite die Kante e" >;
            GD(e.EKn); e:=e.NKa; end loop;
end GD;
begin  ... < "baue den Graphen auf" >; ... ;
      p := Anfang;
      while p /= null loop p.besucht:=false; p := p.NKn; end loop;
      p := Anfang;
      while p /= null loop GD(p); p := p.NKn; end loop;  ...
end;
```



Machen Sie sich den Algorithmus an folgendem einfachen Beispiel klar. Die Reihenfolge in der Knotenliste des Graphen sei $u_1$, $u_2$, $u_3$, $u_4$, $u_5$, $u_6$ und $(u_1, u_2)$ sei die erste von $u_1$ ausgehende Kante.

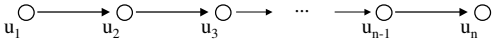



Dann werden die Knoten in der Reihenfolge $u_1, u_2, u_5, u_3, u_4, u_6$ bearbeitet (entsprechend < "bearbeite den Knoten u" > in der Prozedur GD).



*Aufwand:* Die Prozedur GD wird n mal in der Schleife
while p /= null loop GD(p); p := p.NKn; end loop;
und m-mal in der Prozedur selbst aufgerufen. Jeder Knoten und jede Kante werden genau einmal bearbeitet. Die uniforme *Zeitkomplexität* beträgt daher O(n+m).

*Platz:* Die rekursiven Aufrufe kosten zusätzlichen Platz im lokalen Speicher. Der ungünstigste Fall liegt vor, wenn der Graph eine Kette von Knoten ist:




In diesem Fall werden n-1 rekursive Aufrufe ineinander geschachtelt, bevor die erste Prozedur wieder verlassen wird. Da hierbei die Variablen v und e neu angelegt werden, erhalten wir einen *Zusatz-Platzbedarf* der Größe 2n.



***Einschub:*** Definition ungerichtete Wege und Pfade (vgl. 1.1.5.7)

Gegeben sei ein ungerichteter Graph G = (V, E) mit
$E \subseteq \{ \{x, y\} \mid x, y \in V, x \neq y \} \cup \{ \{x\} \mid x \in V \}$.
Eine Folge von Knoten $(u_1, u_2, u_3, ..., u_r)$ mit $r \geq 1$ heißt Weg im Graphen G, wenn $\{u_i, u_{i+1}\} \in E$ für i = 1, 2, ..., r-1 gilt.

r-1 heißt die Länge des Weges.

Man sagt, der Weg $(u_1, u_2, ..., u_r)$ führt vom Knoten $u_1$ zum Knoten $u_r$ oder $u_r$ ist von $u_1$ aus (über diesen Weg) erreichbar.

Zwei Knoten u und v heißen verbunden, wenn es einen Weg von u nach v gibt.

Betrachtet man dagegen die Kanten, die diesen Weg bilden, so spricht man von "Pfaden" im Graphen, d.h., wenn $(u_1, u_2, u_3, ..., u_r)$ ein Weg ist, so ist
$(\{u_1,u_2\}, \{u_2,u_3\}, \{u_3,u_4\}, ..., \{u_{r-1},u_r\})$
der zugehörige Pfad im Graphen.

***Einschub:*** Definition Zusammenhang

Ein Weg $(u_1, u_2, u_3, ..., u_r)$ heißt doppelpunktfrei oder einfach, wenn $u_i \neq u_j$ für alle $i \neq j$ gilt.

Ein Weg heißt geschlossen, wenn $u_r = u_1$ gilt.

Ein Weg $(u_1, u_2, u_3, ..., u_r)$ heißt Kreis oder Zyklus, wenn $r \geq 4$ ist, der Weg geschlossen ist und $(u_1, u_2, u_3, ..., u_{r-1})$ doppelpunktfrei ist. Ein Graph heißt zyklenfrei oder azyklisch, wenn er keine Zyklen besitzt.

Ein Graph heißt zusammenhängend, wenn jeder Knoten mit jedem Knoten verbunden ist.

Die Menge
$Z(u) = \{ v \in V \mid u \text{ und } v \text{ sind verbunden } \}$
heißt die Zusammenhangskomponente des Knotens u.
(Es ist klar, dass für alle $v \in Z(u)$ gilt: $u \in Z(v)$.)



***Einschub:*** Definition gerichtete Wege und Pfade (vgl. 1.1.5.7)

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit $E \subseteq V \times V$. Eine Folge von Knoten $(u_1, u_2, u_3, ..., u_r)$ mit $r \geq 1$ heißt (gerichteter) Weg im Graphen G, wenn $(u_i, u_{i+1}) \in E$ für $i = 1, 2, ..., r-1$ gilt. r-1 heißt die Länge des Weges.

Man sagt, der Weg $(u_1, u_2, ..., u_r)$ führt vom Knoten $u_1$ zum Knoten $u_r$.

Betrachtet man die Kanten, die diesen Weg bilden, so spricht man von "Pfaden" im Graphen, d.h., wenn $(u_1, u_2, u_3, ..., u_r)$ ein Weg ist, so ist

$$((u_1,u_2), (u_2,u_3), (u_3,u_4), ..., (u_{r-1},u_r))$$

der zugehörige Pfad im Graphen.

Zwei Knoten u und v heißen verbunden, wenn es einen Weg von u nach v und einen Weg von v nach u gibt.



***Einschub:*** Definition Zusammenhang für gerichtete Graphen

Ein Weg $(u_1, u_2, u_3, ..., u_r)$ heißt doppelpunktfrei oder einfach, wenn $u_i \neq u_j$ für alle $i \neq j$ gilt.

Ein Weg heißt geschlossen, wenn $u_r = u_1$ gilt.

Ein Weg $(u_1, u_2, u_3, ..., u_r)$ heißt Kreis oder Zyklus, wenn $r \geq 4$ ist, der Weg geschlossen ist und $(u_1, u_2, u_3, ..., u_{r-1})$ doppelpunktfrei ist. Ein Graph heißt zyklenfrei oder azyklisch, wenn er keine Zyklen besitzt.

Ein gerichteter Graph heißt stark zusammenhängend, wenn jeder Knoten mit jedem Knoten verbunden ist.

Die Menge
$Z(u) = \{ v \in V \mid u \text{ und } v \text{ sind verbunden } \}$
heißt die starke Zusammenhangskomponente des Knotens u.
(Es ist klar, dass für alle $v \in Z(u)$ gilt: $u \in Z(v)$.)

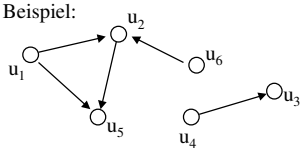


***Einschub:*** Definition schwacher Zusammenhang

Zu einem gerichteten Graphen $G=(V,E)$ sei $G_{ung}=(V,E_{ung})$ mit $E_{ung} = \{\{x,y\} \mid (x,y) \in E \text{ oder } (y,x) \in E\} \cup \{\{x\} \mid (x,x) \in E\}$ die ungerichtete Version (siehe 1.1.5.7).

Die Menge $SwZ(u) = Z(u)$ in $G_{ung}$ heißt die schwache Zusammenhangskomponente des Knotens u im Graphen G.
(Wiederum folgt, dass für alle $v \in SwZ(u)$ gilt: $u \in SwZ(v)$.)

Beispiel:




$(u_6, u_2, u_5)$ ist ein einfacher Weg von $u_6$ nach $u_5$.
$Z(u_1) = \{u_1\}$, $Z(u_3) = \{u_3\}$,
$SwZ(u_1) = \{u_1, u_2, u_5, u_6\}$.
Es gibt keinen Kreis.



***Einschub:*** Definition transitive Hülle

Zu dem gerichteten oder ungerichteten Graphen $G=(V,E)$ heißt der gerichtete bzw. ungerichtete Graph $G_{tH}=(V,E_{tH})$ mit $E_{tH} = \{(x,y) \mid \text{es gibt einen Weg von x nach y}\}$ die transitive Hülle des Graphens G.

Zwei spezielle Graphen:
Ein Graph, in dem je zwei verschiedene Knoten miteinander durch eine Kante verbunden sind, heißt vollständiger Graph $\mathbf{K_n} = (V,E)$ und $E = \{ \{x,y\} \mid x,y \in V, x \neq y \}$ im ungerichteten Fall bzw. $E = \{ (x,y) \mid x,y \in V, x \neq y \}$ im gerichteten Fall.

Der Graph, bei dem n Knoten einen Ring bilden, heißt "Kreis" $\mathbf{C_n} = (V,E)$ mit $V = \{x_1, x_2, ..., x_n\}$ und $E = \{ \{x_1,x_2\}, \{x_2,x_3\}, \{x_3,x_4\}, ..., \{x_{n-1},x_n\}, \{x_n,x_1\} \}$ (ungerichtet) bzw. $E = \{(x_1,x_2), (x_2,x_3), ..., (x_{n-1},x_n), (x_n,x_1)\}$ im gerichteten Fall.



*Einfache Anwendung* des Graphdurchlaufs:

Wenn man *bei ungerichteten Graphen* zu Beginn des Aufrufs GD aus der while-Schleife heraus immer eine neue Liste beginnt (sofern p.besucht false ist), in die man alle in der Prozedur erstmals besuchten Knoten einfügt, so erhält man die Zusammenhangskomponenten.

Denn im ungerichteten Fall gibt es zu jedem Weg von u nach v auch einen Weg von v nach u. Und alle Knoten, die von u aus erreichbar sind, sind dann bereits als "besucht" markiert, wenn man beim Durchlauf durch die Knotenliste auf sie trifft. Es verbleiben also nur noch Knoten aus anderen Zusammenhangskomponenten.

*Hinweise im Detail:* Füge zum Programm hinzu

```
I: integer; Z: array (1..n) of Nextknoten;
```

Vor die letzte while-Schleife füge `I :=0;` hinzu und ersetze die letzte while-Schleife durch:

```
while  p /= null  loop
        if not p.besucht then I:=I+1; Z(I) := null;
                GD(p); p := p.NKn;  end if;  end loop;
```

und in der Prozedur fügen wir unmittelbar nach "then" ein:
`Z(I) := new Nextknoten'(NKn => Z(I); Id => u.Id);`

Am Ende bilden die I Listen Z(1), ..., Z(I) die Zusammenhangskomponenten.

Im gerichteten Fall muss man anders vorgehen. Überlegen Sie, wie.



***Einschub:*** Definition Baum (vgl. 1.1.5.7)

Ein Knoten w heißt Wurzel eines Graphen G, wenn es von w zu jedem Knoten des Graphen einen Weg gibt (im gerichteten Fall muss der Weg natürlich auch gerichtet sein).

Ein ungerichteter Graph G = (V, E) heißt Baum, wenn er azyklisch und zusammenhängend ist (insbesondere ist dann jeder Knoten des Baums auch Wurzel).

Ein gerichteter Graph heißt Baum, wenn er eine Wurzel w besitzt und jeder Knoten ungleich der Wurzel genau einen Vorgänger besitzt (d.h., zu jedem $x \in V$, $x \neq w$ gibt es genau einen Knoten y mit $(y, x) \in E$. Dieser Knoten y heißt dann Vater oder direkter Vorgänger von x).

Ein Graph heißt Wald, wenn alle seine (schwachen) Zusammenhangskomponenten Bäume sind.



***Einschub:*** rekursive Definition Baum

Man kann Bäume auch rekursiv definieren:

1) Die leere Menge ist ein Baum.
2) Wenn w ein Knoten und U eine endliche Menge von Bäumen ist, so ist auch w(U) ein Baum.

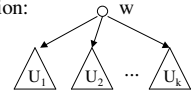
w heißt Wurzel des Baums w(U), die Elemente von U heißen Unterbäume oder Teilbäume von w im Baum w(U).

*Skizze:*

gerichtet oder ungerichtet

Leerer Baum: ∅

Rekursion:




*Hinweis:* Ist die Menge der Bäume $U = \{U_1, ..., U_k\}$ geordnet, so spricht man von einem geordneten Baum.



***Einschub:*** zur Definition Baum

*Hinweis 1:* Ein Baum mit n Knoten besitzt stets n-1 Kanten.

*Hinweis 2:* Bäume kann man stets als gerichtete Graphen auffassen, indem man im ungerichteten Fall willkürlich einen Knoten zur Wurzel w macht und alle Kanten von w weg richtet.

*Bezeichnung:* In Bäumen gibt es stets Knoten ohne Nachfolger. Sie heißen Blätter (im ungerichteten Fall haben sie den Grad 1).

*Bezeichnung:* Knoten, die den gleichen Vater im Baum haben, heißen Brüder.

*Bezeichnung:* Die Länge des längsten Wegs von der Wurzel w zu einem Blatt bezeichnet man als die Tiefe (oder Höhe) des Baums.



### 1.6.4.4 Durchlauf von (binären) Bäumen

Sehr oft benötigt man den Baumdurchlauf, insbesondere den Durchlauf durch binäre Bäume; dies sind gerichtete und geordnete Bäume, bei denen jeder Knoten genau einen linken und einen rechten Nachfolger besitzt (diese Nachfolger können auch leer sein; wichtig ist, dass der linke und der rechte Nachfolger stets unterschieden werden).

Ein binärer Baum, dessen Inhalts-Datentyp geordnet ist (z.B. ganze Zahlen), heißt (binärer) Suchbaum, wenn für alle jeden Knoten u gilt: Alle Inhalte von Knoten im linken Unterbaum von u sind echt kleiner als der Inhalt von u und alle Inhalte im rechten Unterbaum von u sind größer oder gleich dem Inhalt von u.



Aus 1.3.3.3 übernehmen wir die Definition des Baums, hier speziell des binären Baums; als Inhalt wählen wir integer, um in der späteren Anwendung einen Suchbaum benutzen zu können:

```
type BinBaum;
type Ref_BinBaum is access BinBaum;
type BinBaum is record
        inhalt: integer;
        L, R: Ref_BinBaum;
end record;
```

Der Durchlauf durch einen binären Baum erfolgt rekursiv:

```
procedure Inorder (b: Ref_BinBaum) is
begin
        if b /= null then
                Inorder (b.L);
                < bearbeite den Knoten b >;
                Inorder (b.R);
        end if;
end;
```

Man diesen Durchlauf auch einen Inorder-Durchlauf, da der jeweilige Knoten zwischen dem Durchlauf durch seinen linken und seinen rechten Unterbaum bearbeitet wird.



Lautet der Rumpf

```
if b /= null then
        < bearbeite den Knoten b >;
        Inorder (b.L);
        Inorder (b.R);
end if;
```

so spricht man vom Preorder-Durchlauf; lautet er

```
if b /= null then
        Inorder (b.L);
        Inorder (b.R);
        < bearbeite den Knoten b >;
end if;
```

so spricht man von einem Postorder-Durchlauf.



*Aufwand:* Jeder Knoten wird genau dreimal betrachtet. Der Durchlauf erfolgt somit für jeden der drei Durchläufe uniform in 3n = O(n) Schritten.

*Platz:* Auch hier ist die Rekursion zu beachten, die maximal so tief sein kann, wie es Knoten im Baum gibt. D.h., der zusätzliche Platzbedarf beträgt im worst case O(n) Speicherplätze.

Im Mittel wird man jedoch deutlich weniger Plätze benötigen. Der beste Fall liegt vor, wenn der Baum sehr gleichmäßig verzweigt ist, also die Tiefe log(n) ist. In diesem Fall braucht man nur O(log(n)) zusätzlichen Speicherplatz. (Wir werden dies genauer in Kapitel 3 berechnen.)



*Anwendung:* Sortieren von n Zahlen.

*Vorgehen:* Lege n Zahlen $a_1, ..., a_n$ der Reihe nach in einem binären Suchbaum ab. Lies dann den Baum inorder aus.

*Programm:*

```
Anker, p: Ref_BinBaum;
begin ...; Anker := null;
    for i in 1..n loop
        p := Anker; q:=p;
        while p /= null loop  q:=p;
          if p.inhalt < A(i) then p:= p.R; else p:=p.L; end if;
          p := new Ref_BinBaum'(A(i), null, null);
          if q.inhalt < A(I) then q.R := p; else q.L := p; end if;
    end loop; end loop:
    Inorder (Anker); ...
end;
```



Der Rumpf der Prozedur "Inorder" muss hier lauten:

```
if b /= null then  Inorder (b.L);
                   put (b.inhalt);
                   Inorder (b.R);
end if;
```

Somit wird die sortierte Folge ausgegeben.

*Aufwand:* Im worst case (wenn die Folge der A(i) bereits sortiert ist), kann der Baum die Tiefe O(n) erreichen. Das Einsortieren dauert dann $O(n^2)$ Schritte, der Inorder-Durchlauf O(n) Schritte, insgesamt also $O(n^2)$ Schritte. Im Mittel werden aber nur O(n log(n)) Schritte benötigt, wie wir später bei der Analyse des Quicksort-Algorithmus sehen werden.

*Zusätzlicher Platzbedarf:* Nur für die Inorder-Rekursion, siehe dort; bis zu O(n) im worst case.



### 1.6.4.5 Transitive Hülle eines Graphen

Die transitive Hülle gibt für je zwei Knoten u und v direkt an, ob es einen Weg von u nach v gibt oder nicht. Gegeben sei also ein gerichteter Graph G = (V, E) mit $V = \{x_1, ..., x_n\}$.

Üblicherweise berechnet man die transitive Hülle über die Adjazenzmatrix (siehe 1.3.3.4). Wir betrachten hier den *gerichteten Fall:* Die Adjazenzmatrix $A = (a_{i,j})$ ist definiert durch $a_{i,j} = 1$, falls $(x_i, x_j) \in E$, und $a_{i,j} = 0$ sonst (i, j = 1, ..., n).

Gesucht ist die Adjazenzmatrix D der transitiven Hülle (zur Definition blättere ca. 9 Folien zurück):
$D = (d_{i,j})$ ist definiert durch
$d_{i,j} = 1$, falls es einen gerichteten Weg von $x_i$ nach $x_j$ gibt, und $d_{i,j} = 0$ sonst (i, j = 1, ..., n).

*Vorgehen:*

Prüfe für alle i, j = 1, ..., n, ob es ein k gibt, so dass ein Weg von Weg von $x_i$ nach $x_j$ über $x_k$ existiert.

Warshall-Algorithmus zur Berechnung der transitiven Hülle:

```
type adj is array (1..n, 1..n) of 0..1;  A, D: adj; ...
begin   ... ;  D := A;
        for i in 1..n loop D(i,i) := 1; end loop;
        for k in 1..n loop
           for i in 1..n loop
              for j in 1..n loop
                 if D(i,k)=1 and D(k,j)=1 then D(i,j) := 1; end if;
        end loop;  end loop;  end loop; ...
end;
```

*Hinweis:* k muss in der äußersten Schleife stehen!



Wie beweist man, dass dieses Verfahren tatsächlich die transitive Hülle berechnet?

Durch Induktion. Die Induktionsannahme lautet:
Besitzt k am Ende der beiden inneren Schleifen den Wert m, so gilt
D(i,j) = 1 $\Leftrightarrow$ Es gibt $r \geq 0$ und einen Weg $(x_i, u_1, u_2, ..., u_r, x_j)$, so dass für alle s=1, .., r gilt: $u_s \in \{x_1, x_2, ..., x_m\}$.
Man sieht dann leicht ein, dass diese Aussage nach Durchlauf der beiden inneren Schleifen auch für m+1 gilt. Die transitive Hülle erfüllt genau diese Aussage für k = n, d.h., am Ende ist D die Adjazenzmatrix für die transitive Hülle.

*Aufwand:* $O(n^3)$, da drei ineinander geschachtelte Schleifen.

*Zusätzlicher Platzbedarf:* Nur für die Matrix D: $O(n^2)$ Plätze.



## Gliederung des Kapitels


**1.6 Komplexität von Algorithmen und Programmen**

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




## 1.6.5 Andere Rechenmodelle

Bei den Rechenmodellen hängt es davon ab, welche "Mächtigkeit" das Modell besitzen soll. Gängige Modelle, die beliebige Algorithmen ausführen können (die so mächtig wie Turingmaschinen sind; man sagt, "die *turingmächtig* sind"), sind die Registermaschine, der Lambda-Kalkül, prädikatenlogische Kalküle, Markov-Algorithmen usw.

Deutlich eingeschränkt in ihren Fähigkeiten sind die "linear beschränkten Automaten" (= Turingmaschinen mit O(n) Speicherzellen auf dem Band), die Keller- oder Pushdownautomaten (neben Ein- und Ausgabeband besitzen sie einen Keller als Speicherband) und endliche Automaten oder endliche Akzeptoren ("EA" oder engl. "FA") ohne zusätzlichen Speicher, jeweils deterministisch und nichtdeterministisch.

Wir stellen alle Modelle kurz vor, hierbei etwas genauer die EAs.

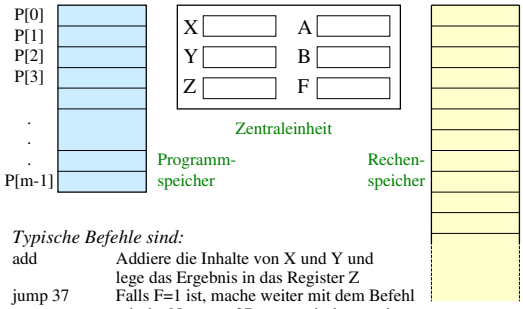


### 1.6.5.1 Veranschaulichung von Registermaschinen

Eine Registermaschine ist wie ein kleiner Mikroprozessor aufgebaut. Sie besteht aus

1. einer *Zentraleinheit* mit 6 Registern (Speicherzellen): 3 Register X, Y und Z für arithmetische und logische Operationen, ein Adressregister A für den Zugriff auf Rechenspeicherzellen, ein Befehlsregister B, in dem die Adresse des auszuführenden Befehls steht, und ein "Flag"-Register F, in dem das Ergebnis von Operationen abgelegt wird;
2. einem *Programmspeicher* P, in dem nacheinander die Befehle des abzuarbeitenden endlichen Programms stehen;
3. einem (unendlich langen) *Rechenspeicher* R mit durchnummerierten Speicherzellen, die jeweils eine ganze Zahl aufnehmen können.






*Typische Befehle sind:*

| | |
|---|---|
| add | Addiere die Inhalte von X und Y und lege das Ergebnis in das Register Z |
| jump 37 | Falls F=1 ist, mache weiter mit dem Befehl mit der Nummer 37, sonst mit dem nächsten |
| LeftShift X | Schiebe den Inhalt von X eine Stelle nach links und füge eine 0 an |
| comp(Y > Z) | if der Inhalt von Y ist größer als der von Z then F:=1 else F:=0 fi |
| read X | X := $R_{<A>}$, d.h., sei a der Inhalt von A, so weise zu X:=$R_a$ |
| load A,4 | A := 4 (Wertzuweisung einer Konstanten an ein Register) |

*Der übliche Befehlssatz einer Registermaschine* lautet (hierbei bezeichnen V und V' beliebige Register, c ist eine ganze Zahl, $b \in \mathbb{N}_0$, $R_k$ ist die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ ist eine Vergleichsoperation; außer beim Jump-Befehl wird nach jedem Befehl B um 1 erhöht):

| Befehl | Bedeutung | Befehl | Bedeutung |
|---|---|---|---|
| load V,c | V := c | copy V,V' | V := V' |
| read V | V := $R_{<A>}$ | write V | $R_{<A>}$ := V |
| succ | X := X+1 | add | X := Y+Z |
| sub | X := Y-Z | shift | X := X div 2 |
| comp (σ) | if XσY then F:=1 else F:=0 fi | | |
| jump b | if F=1 then B:=b else B:=B+1 fi | | |
| stop | Anhalten, Ende der Berechnung | | |

Diese Befehle finden sich bei allen Mikroprozessoren; diese haben in der Regel aber noch viel mehr Befehle, insbesondere bzgl. der Adressierung, der Verschiebungen von Daten, der eingebauten Kellermechanismen und der Unterbrechungsbefehle. Die Ausformulierung von Algorithmen als Registermaschine ähnelt daher der Maschinenprogrammierung.



*Beispiel:* Test auf Teilbarkeit durch 2 (die Zahl $n \geq 0$ stehe anfangs in $R_0$). Naheliegende Lösung: Subtrahiere von n ständig die Zahl 2. Die Zahl n sei hier der Variablen I zugewiesen. Dann lautet das Programmstück: **while I>1 do I:=I-2 od;** Anschließend steht in I der Rest der Division von n durch 2.
Übertrage dieses Programmstück in den Befehlssatz der Registermaschine (I ⇔ Register Y, Zahl 2 ⇔ Z, anfangs steht n in der Rechenspeicherzelle $R_0$, am Ende stehe das Ergebnis in $R_0$, sei a der Inhalt von A, so bezeichnet $R_{<A>}$ den Inhalt der Rechenspeicherzelle $R_a$). Man erhält folgendes Registermaschinenprogramm:

| Nummer | Befehl | Erläuterung |
|---|---|---|
| 0: | load A,0 | A := 0 |
| 1: | read Y | Y := n (= $R_0$ = $R_{<A>}$) |
| 2: | load Z,2 | Z := 2 |
| 3: | load X,1 | X := 1 |
| 4: | comp ( ≥ ) | teste, ob X ≥ Y ist (beachte: in X steht eine 1) |
| 5: | jump 10 | if 1 ≥ Y then weiter bei Befehl mit Nummer 10 fi |
| 6: | sub | X := Y-Z (also X:=Y-2) |
| 7: | copy Y,X | Y := X |
| 8: | load F,1 | F:=1 (um einen Sprung nach 3 zu erzwingen) |
| 9: | jump 3 | weitermachen beim Befehl mit der Nummer 3 |
| 10: | write Y | in A steht noch 0, also: $R_0$ := Y |
| 11: | stop | |



Betrachtet man Registermaschinen, die nur auf ganzen Zahlen arbeiten, so muss man zuvor prüfen, ob n < 0 ist oder nicht. Wir fügen daher vor das oben angegebene Programmstück noch die Anweisung if I < 0 then I := 0-I fi und übertragen dieses Programm wiederum in ein Registermaschinenprogramm. So erhalten wir:

Man kann sich überzeugen, dass jedes Ada-Programm in ein solches Registermaschinenprogramm übersetzt werden kann und dass sich dieser Prozess automatisieren lässt (Compiler). Solche einfachen Sprachen heißen in der Praxis "Maschinensprachen" oder "Maschinencode"; sie werden meist in Form von Assemblern ein wenig lesbarer und komfortabler gemacht.

```
0: load A,0
1: read Y
2: load X,0
3: comp (≤)
4: jump 9
5: copy Z,Y
6: load Y,0
7: sub
8: copy Y,X
9: load Z,2
10: load X,1
11: comp (≥)
12: jump 17
13: sub
14: copy Y,X
15: load F,1
16: jump 10
17: write Y
18: stop
```



Die verschiedenen Typen von Registermaschinen unterscheiden sich in ihren (elementaren) Datentypen und in ihren Befehlssätzen. Meist fügt man noch ein Eingabeband, das nur gelesen werden darf, und ein Ausgabeband, das nur beschrieben werden darf, hinzu (vgl. Kellerautomat und endlicher Automat unten).

Wie bei Turingmaschinen kann man für dieses Rechenmodell, das über das Register A einen wahlfreien Zugriff auf die Rechenspeicherzellen erlaubt, Komplexitätsmaße und -klassen definieren. Hiermit lassen sich Aussagen beweisen, wie wir sie in 1.6.3.9 und 1.6.3.10 vorgestellt haben.

Die Komplexitätsmaße von Registermaschinen geben in der Regel die Verhältnisse von Programmen recht gut wieder. Sie werden daher vielen theoretischen Untersuchungen zugrunde gelegt.



In den Programmen von Registermaschinen gibt es nur zwei Kontrollstrukturen:

- Übergang zum nächsten Befehl (dies entspricht dem ";" in Ada). Dies wird dadurch realisiert, dass nach jeder Ausführung eines Befehls B um 1 erhöht wird.
- Bedingter Sprung (falls F=1) zum Befehl mit der Nummer b (jump b), sofern die Nummer b im Programm vorkommt (anderenfalls Fehlerabbruch).

Man kann alle strukturierten Anweisungen in höheren Programmiersprachen durch diese beiden Kontrollstrukturen simulieren. In Ada sind Sprünge mit dem Schlüsselwort goto zugelassen. Anstelle der Nummern verwendet man *Marken*, dies sind Bezeichner, die in <<...>> eingeschlossen und vor eine Anweisung gesetzt werden



if X < Y then Z := 1; else X := Y; end if; X := X+1; ...

wird im Registerprogramm zu (willkürlich wurde 20 als Nummer des ersten Befehls gewählt)

```
20: comp (<)      vergleiche X mit Y bzgl. "kleiner"
21: jump 25       springe zum then-Zweig
22: load Z,1      Z := 1 (else-Zweig ausführen)
23: load F,1      bereite einen "unbedingten" Sprung vor
24: jump 26       überspringe den then-Zweig
25: copy X,Y      X := Y (then-Zweig ausführen)
26: succ          X := X+1
```

In Ada kann man dieses Programmstück direkt nachbilden:

```
F := X < Y; if F then goto THEN_ZWEIG; end if;
Z := 1; goto DANACH;
<<THEN_ZWEIG>>  X := Y;
<<DANACH>>  X := X+1; ...
```

while X < Y loop X := X+1; end loop; Z:=X;...

wird im Registerprogramm zu

| | |
|---|---|
| 30: comp (≥) | vergleiche X mit Y bzgl. "kleiner" |
| 31: jump 25 | überspringe die Schleife |
| 32: succ | X := X+1 (Schleifenrumpf) |
| 33: load F,1 | bereite einen "unbedingten" Sprung vor |
| 34: jump 30 | zurück zur Schleifen-Bedingung |
| 35: copy Z,X | Z := X ... |

In Ada lautet dieses Programmstück:

<<Schleife>> F := X ≥ Y; if F then goto danach; end if;
X := X+1; goto Schleife;
<<danach>> Z := X; ...



*Hinweise zu Sprüngen in Ada:*

1. Generell soll man Sprünge vermeiden, weil Fehler in solchen Programmen nur schwer zu entdecken sind und entsprechende Software kaum gepflegt oder angepasst werden kann.
2. Sprünge dürfen nie in Strukturen hineinführen. Z.B.: Es ist verboten, von außen in den then- oder else-Zweig, in einen Schleifenrumpf, in einen Block, in einen Prozedurrumpf, in einen exception_handler usw. zu springen.
3. Umgekehrt sind jedoch Sprünge aus Strukturen heraus in hierarchisch übergeordnete Strukturen erlaubt (Sprünge aus einem exception_handler hinaus müssen allerdings den umgebenden Block / die Prozedur verlassen).
4. Regel: *Vermeiden Sie Sprünge!*

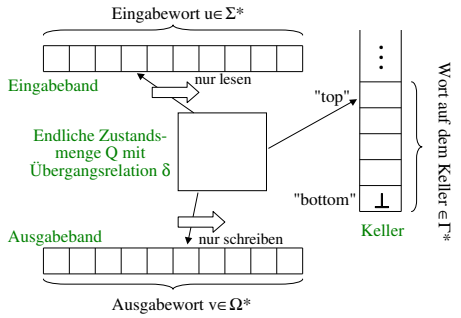


1.6.5.2: Linear beschränkte Automaten sind Turingmaschinen, deren k Bänder nur so lang sind wie die Länge des Eingabewortes. Sie beschreiben also genau die Komplexitätsklasse
DSpace$^{TM}$(n), sofern die Maschine deterministisch ist, bzw.
NSpace$^{TM}$(n) im nichtdeterministischen Fall.

Fast alle Probleme, die in der Praxis auftreten, gehören zur Klasse NSpace$^{TM}$(n). Diese Klasse kann genau die Probleme lösen, die sich mit kontextsensitiven Grammatiken erzeugen lassen (vgl. 1.1.5.17). Die Laufzeit dieser Maschinen ist in der Regel exponentiell mit der Länge der Eingabe, so dass man in der Praxis nur eingeschränkte Maschinen verwendet.



1.6.5.3: Die Keller- oder Pushdownautomaten besitzen ein Eingabe- und ein Ausgabeband sowie als Arbeitsspeicher ein Keller-Band.






1.6.5.3.a Definition (nichtdeterministischer Kellerautomat)

$A=(Q,\Sigma,\Omega,\Gamma,\delta,Q_0,F,\perp)$ heißt Keller- oder Pushdownautomat ⇔

1. Q ist eine endliche nicht-leere Menge (Zustandsmenge),
2. Σ ist eine endliche nicht-leere Menge (Eingabealphabet),
3. Ω ist eine endliche nicht-leere Menge (Ausgabealphabet),
4. Γ ist eine endliche nicht-leere Menge (Kelleralphabet),
5. $Q_0 \in Q$ ist die Menge der Anfangszustände (meist einelementig),
6. $F \subseteq Q$ ist die Menge der Endzustände,
7. $\perp \in \Gamma$ ist das Bottomsymbol (zeigt an, ob der Keller leer ist)
8. $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^* \times \Omega^*$, endliche Menge, ist die Überführungsrelation.

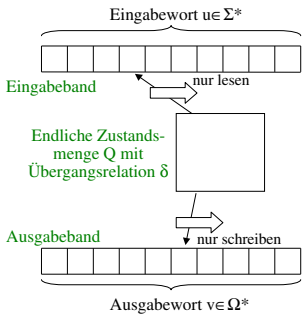
*Hinweis:* A heißt deterministisch ⇔
(1) Wenn es ein $(q,\varepsilon,\alpha,q',z,w) \in \delta$ gibt, so gibt es kein anderes $(q,a,\alpha,q'',z'',w'') \in \delta$ (also mit gleichem q und α) für alle $a \in \Sigma \cup \{\varepsilon\}$.
(2) Liegt Fall (1) nicht vor, so gibt es zu jedem Tripel $(q,a,\alpha) \in Q \times \Sigma \times \Gamma$ höchstens ein Tripel $(q',z,w) \in Q \times \Gamma^* \times \Omega^*$ mit $(q,a,\alpha,q',z,w) \in \delta$.



1.6.5.3.b: Aussagen und Hinweise:

1. Alle Probleme, die Kellerautomaten bearbeiten können, liegen in DTime$^{TM}$($n^3$) und in NTime$^{TM}$(n).
2. Kellerautomaten können genau die Probleme bearbeiten, die man mit kontextfreien Grammatiken (also mit der BNF) beschreiben kann.
3. Nur die Programmiersprachen, deren Syntaxanalyse sich im Wesentlichen mit deterministischen Kellerautomaten durchführen lässt, sind für die Praxis interessant.
4. Kellerautomaten werden für (hierarchisch aufgebaute) Probleme mit Klammerstrukturen eingesetzt.

Lässt man nun noch das Kellerband weg, so erhält man den "endlichen Automaten", also eine Maschine, die die Eingabe von links nach rechts liest, die synchron hierzu ein Ausgabeband beschreibt und die nur *endlich viel Information* in ihrer Zustandsmenge speichern kann.






1.6.5.4 Definition (nichtdeterministischer endlicher Automat)

$A = (Q, \Sigma, \Omega, \delta, Q_0, F)$ heißt endlicher Automat (engl.: finite state machine, finite automaton) $\Leftrightarrow$

1. $Q$ ist eine endliche nicht-leere Menge (Zustandsmenge),
2. $\Sigma$ ist eine endliche nicht-leere Menge (Eingabealphabet),
3. $\Omega$ ist eine endliche nicht-leere Menge (Ausgabealphabet),
4. $Q_0 \in Q$ ist die Menge der Anfangszustände,
5. $F \subseteq Q$ ist die Menge der Endzustände,
6. $\delta \subseteq Q \times \Sigma \times Q \times \Omega^*$, endliche Menge, ist die Überführungsrelation.

A heißt deterministisch, wenn es zu jedem Paar $(q,a) \in Q \times \Sigma$ höchstens ein Paar $(q',w) \in Q \times \Omega^*$ mit $(q,a,q',w) \in \delta$ gibt.
A heißt endlicher Akzeptor, wenn $\delta \subseteq Q \times \Sigma \times Q$ gilt und es genau einen Anfangszustand gibt.

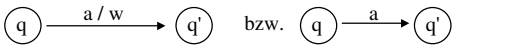


Die Arbeitsweise eines endlichen Automaten A ist sehr einfach: Anfangs befindet man sich in einem Anfangszustand und es steht ein Wort $u \in \Sigma^*$ auf dem Eingabeband. Zu dem jeweiligen Zustand q und dem nächsten Eingabezeichen a suche ein Tupel $(q,a,q',w) \in \delta$; gehe dann in den Zustand q' und drucke w ans Ende des Ausgabebandes. Falls es kein solches Tupel gibt, brich ab. Wenn man auf diese Weise die gesamte Eingabe gelesen hat und sich am Ende in einem Endzustand (aus F) befindet, dann steht auf dem Ausgabeband das Resultat der Eingabe $Res_A(u) = v$. Im Falle des Akzeptors spielt die Ausgabe keine Rolle, sondern nur die Frage, ob man am Ende in einem Endzustand ist oder nicht.

Der endliche Automat funktioniert also wie eine 2-Band-Turingmaschine, die das Eingabeband nur von links nach rechts lesen und synchron hierzu auf das zweite Band die Ausgabe von links nach rechts drucken darf.



*Grafische Darstellung:*
Die Zustände werden durch Kreise dargestellt, die Übergänge durch Kanten, an die die Eingabe und die Ausgabe, getrennt durch "/" geschrieben werden. Wenn $(q,a,q',w) \in \delta$ bzw. beim Akzeptor $(q,a,q') \in \delta$ ist, so zeichnet man dies in der Form




Anfangszustände erhalten einen Pfeil


und Endzustände werden doppelt umkringelt.


Dadurch werden Automaten sehr anschaulich. Da sie zugleich formal definiert sind, lassen sie sich auch maschinell sehr gut verarbeiten.



*Was ist die Bedeutung eines endlichen Automaten?*

Interpretation als Automat:
Zu jeder Eingabe gibt es eine Menge von Ausgaben, die bei dieser Eingabe möglich wären:
$L(A) = \{(u,v) \mid$ Es gibt eine Folge von Übergängen, die einen Anfangszustand aus $Q_0$ bei der Eingabe $u \in \Sigma^*$ in einen Endzustand aus F überführen und hierbei die Ausgabe $v \in \Omega^*$ erzeugen $\} \subseteq \Sigma^* \times \Omega^*$.

Falls A deterministisch ist, so gibt es zu jedem $u \in \Sigma^*$ höchstens ein $v \in \Omega^*$ mit $(u,v) \in L(A)$. In diesem Fall wird L(A) zu einer (partiellen) Abbildung $Res_A: \Sigma^* \to \Omega^*$.

L(A) heißt die von A realisierte Relation, $Res_A$ heißt die von A realisierte Abbildung oder die Resultatsfunktion von A.



*Was ist die Bedeutung eines endlichen Akzeptors?*

Interpretation als Akzeptor:

Jede Eingabe u überführt den Akzeptor in einen Endzustand oder nicht. Demnach definiert man die von A erkannte Sprache:
$L(A) = \{u \in \Sigma^* \mid$ Es gibt eine Folge von Übergängen, die einen Anfangszustand aus $Q_0$ bei der Eingabe u in einen Endzustand aus F überführen $\} \subseteq \Sigma^*$.

Wir werden im Folgenden vor allem Beispiele zu Akzeptoren angeben und nur ein Beispiel zu einem deterministischen Automaten.

*Beispiel a:* Gegeben sei der Akzeptor $A = (Q, \Sigma, \delta, q_0, F)$ mit

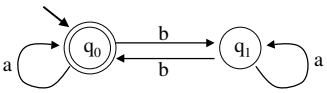
| $\delta$ | a | b |
|---|---|---|
| $q_0$ | $q_0$ | $q_1$ |
| $q_1$ | $q_2$ | $q_0$ |

$Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$ und $F = \{q_0\}$.

Gesucht wird die Menge
$L(A) = \{u \in \Sigma^* \mid$ u wird vollständig eingelesen und dann befindet sich A im Endzustand $q_0\}$.

In diesem Beispiel ist
$L(A) = \{u \in \Sigma^* \mid$ die Anzahl der b in u ist gerade$\}$.

Skizze:




■



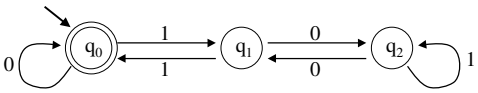
*Beispiel b:* Gegeben sei der Akzeptor $A^{(3)} = (Q, \Sigma, \delta^{(3)}, q_0, F)$

| $\delta^{(3)}$ | 0 | 1 |
|---|---|---|
| $q_0$ | $q_0$ | $q_1$ |
| $q_1$ | $q_2$ | $q_0$ |
| $q_2$ | $q_1$ | $q_2$ |

mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$ und $F = \{q_0\}$.

Gesucht wird die Menge
$L(A^{(3)}) = \{u \in \Sigma^* \mid$ u wird vollständig eingelesen und dann befindet sich $A^{(3)}$ im Endzustand $q_0\}$.

Skizze:






*Wie lautet die Menge $L(A^{(3)})$ ?*

Hierzu betrachten wir die Beziehungen der Eingaben. Es gilt:

Wenn der Akzeptor nach Eingabe von u im Zustand $q_0$ ist, dann führt auch u0 in den Endzustand $q_0$.
Wenn der Akzeptor nach Eingabe von u im Zustand $q_1$ ist, dann führt u1 in den Endzustand $q_0$.
Wenn der Akzeptor nach Eingabe von u im Zustand $q_2$ ist, dann führt u01 in den Endzustand $q_0$.

Dies erinnert an die Teilbarkeit durch die Zahl 3:
Wenn u (binär dargestellt) durch 3 teilbar ist, dann ist auch u0 durch 3 teilbar.
Wenn u (binär dargestellt) durch 3 den Rest 1 lässt (d.h. u = 3k+1), dann ist $u1 = (3k+1)\cdot 2 + 1 = 3\cdot(2\cdot k+1)$ durch 3 teilbar.
Wenn u (binär dargestellt) durch 3 den Rest 2 lässt (d.h. u = 3k+2), dann ist $u01 = (3k+2)\cdot 4 + 1 = 3\cdot(4\cdot k+3)$ durch 3 teilbar.



Tatsächlich ist die vom Akzeptor erkannte Sprache
$L(A^{(3)}) = \{u \in \{0,1\}^* \mid$ u ist die Binärdarstellung einer Zahl, die durch 3 teilbar ist$\}$.

Der Beweis ergibt sich aus der Tatsache, dass sich der Akzeptor $A^{(3)}$ genau dann im Zustand $q_i$ befindet, wenn die bis dahin gelesene Eingabe, aufgefasst als binär dargestellte Zahl, bei der Division durch 3 den Rest i besitzt. Führende Nullen sind hierbei zugelassen.

Dies lässt sich unmittelbar durch Auflisten aller Fälle nachweisen. Damit ist $L(A^{(3)})$ charakterisiert.

■



*Beispiel c:* Gegeben sei der Akzeptor $A = (Q, \Sigma, \delta, q_0, F)$ mit

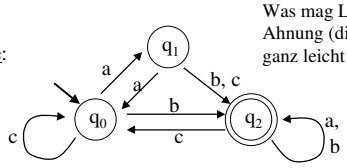
| $\delta$ | a | b | c |
|---|---|---|---|
| $q_0$ | $q_1$ | $q_2$ | $q_0$ |
| $q_1$ | $q_0$ | $q_2$ | $q_2$ |
| $q_2$ | $q_2$ | $q_2$ | $q_0$ |

$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b, c\}$ und $F = \{q_2\}$.

Gesucht wird die Menge
$L(A) = \{u \in \Sigma^* \mid$ u wird vollständig eingelesen und dann befindet sich A im Endzustand $q_2\}$.

Was mag L(A) sein? Keine Ahnung (dies ist auch nicht ganz leicht zu sehen).

Skizze:




■



Wenn der endliche Automat $A=(Q,\Sigma,\Omega,\delta,Q_0,F)$ deterministisch ist, dann kann man (wie bei Turingmaschinen) $\delta$ als (partielle) Funktion $\delta: Q \times \Sigma \to Q \times \Omega^*$ auffassen. Man füge nun einen "Fangzustand" $q_f$ zu Q hinzu, zu dem alle nicht erfassten Übergangsmöglichkeiten führen, wodurch man den Automaten $A_v$ erhält: $A_v=(Q \cup \{q_f\},\Sigma,\Omega,\delta_v,Q_0,F)$, wobei $\delta_v$ eine Erweiterung von $\delta$ ist mit $\delta_v(q,a) = (q_f, \varepsilon)$ für alle $(q,a) \in (Q \cup \{q_f\}) \times \Sigma$, für die $\delta$ nicht definiert ist. Solch einen Automaten mit totaler Übergangsfunktion $\delta$ nennt man vollständig definiert. $A_v$ verhält sich genauso wie A, liest aber jede Eingabe stets vollständig ein.

Analog kann man jeden nichtdeterministischen Automaten vervollständigen, so dass wir also stets annehmen können, dass es zu jedem Paar $(q,a) \in Q \times \Sigma$ mindestens ein Paar $(q',w) \in Q \times \Omega^*$ mit $(q,a,q',w) \in \delta$ gibt.

Meist wird der endliche Automat nur für $\delta \subseteq Q \times \Sigma \times Q \times \Omega$ definiert, d.h., zu jedem gelesenen Eingabezeichen wird genau ein Ausgabezeichen erzeugt. Im deterministischen Fall gilt dann $\delta: Q \times \Sigma \to Q \times \Omega$, was man mit zwei Abbildungen $\delta: Q \times \Sigma \to Q$ und $\gamma: Q \times \Sigma \to \Omega$ darstellt. Weiterhin verlangt man meist, dass der Automat genau einen Anfangszustand $q_0$ besitzt. Das zugehörige deterministische Modell $A=(Q,\Sigma,\Omega,\delta,\gamma,q_0,F)$ bezeichnet man als Mealy-Automaten.

Hängt die Funktion $\gamma$ nicht von der Eingabe, sondern nur vom Zustand ab, d.h. $\gamma: Q \to \Omega$, so spricht man von einem Moore-Automaten. Mealy- und Moore-Automaten werden meist dem Entwurf von Schaltwerken zugrunde gelegt

Wer aus dieser klassischen Theorie kommt, bezeichnet unsere Automaten dann als *"verallgemeinerte"* Automaten.



Endliche Automaten gehören zu den einfachsten Algorithmen. Sie arbeiten nur mit einem Eingabeband, das sie zeichenweise von links nach rechts lesen. Dabei drucken sie in jedem Schritt ein Wort (dies kann auch das leere Wort ε sein) aus $\Omega^*$ auf das Ausgabeband. Nach genau so vielen Schritten, wie die Eingabe lang ist, ist die Arbeit des endlichen Automatens beendet. Die exakte Arbeitsweise lässt sich leicht als Programm formulieren:

q := irgendein Zustand aus $Q_0$;
while not end_of_file do read (a);
    wähle ein Paar (q',w) mit $(q,a,q',w) \in Q \times \Sigma \times Q \times \Omega^*$;
    *(falls dies nicht existiert => Abbruch);* q := q'; write (w)
od;
if $q \in F$ then write (" Erfolgreiche Rechnung ")
else write (" Erfolglose Rechnung ") fi



1.6.5.5: Beispiel Morse-Code

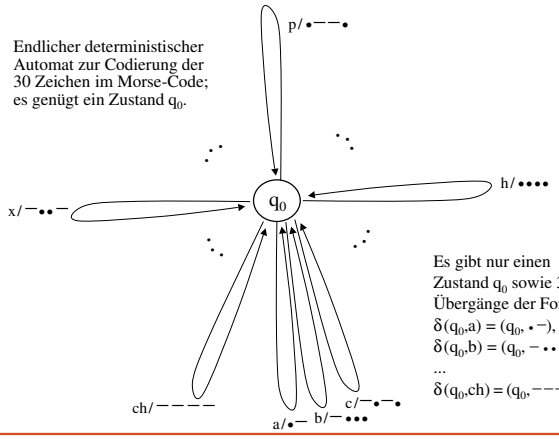
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **a** | •− | **b** | −••• | **c** | −•−• | **d** | −•• | **e** | • |
| **f** | ••−• | **g** | −−• | **h** | •••• | **i** | •• | **j** | •−−− |
| **k** | −•− | **l** | •−•• | **m** | −− | **n** | −• | **o** | −−− |
| **p** | •−−• | **q** | −−•− | **r** | •−• | **s** | ••• | **t** | − |
| **u** | ••− | **v** | •••− | **w** | •−− | **x** | −••− | **y** | −•−− |
| **z** | −−•• | **ä** | •−•− | **ö** | −−−• | **ü** | ••−− | **ch** | −−−− |

Dieser Code besteht aus zwei Zeichen • "kurz" und − "lang". Faktisch gibt es aber noch ein drittes Zeichen ◊ "Pause", mit dem man feststellt, wann eine kurz-lang-Folge zu Ende ist.

Die Codierung {a, b, ..., z, ä, ö. ü, ch}* → {•, −, ◊}* lässt sich mit einem deterministischen endlichen Automaten mit nur einem Zustand realisieren; für die Umkehrung ("Decodierung") braucht man 31 Zustände.



Endlicher deterministischer Automat zur Codierung der 30 Zeichen im Morse-Code; es genügt ein Zustand $q_0$.




Es gibt nur einen Zustand $q_0$ sowie 32 Übergänge der Form
$\delta(q_0,a) = (q_0, \bullet-)$,
$\delta(q_0,b) = (q_0, -\bullet\bullet\bullet)$,
...
$\delta(q_0,ch) = (q_0, ----)$.

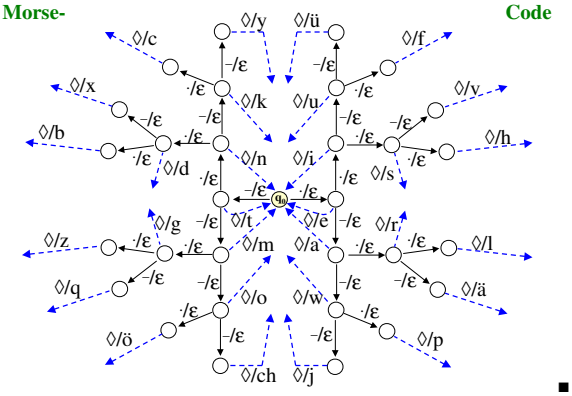


Um zu entscheiden, zu welchem Zeichen eine Folge aus "•" und "−" gehört, muss man die Folge einlesen, bis das Pausenzeichen "◊" auftritt. Dann kann man eindeutig das gesuchte Zeichen ermitteln und ausgeben. Zu dem Zweck muss man sich jedoch alle Zeichenfolgen bis zur Länge 4 merken; dies sind $2^1 + 2^2 + 2^3 + 2^4 = 30$ Möglichkeiten. Zusammen mit dem Anfangszustand benötigt man daher 31 Zustände für die Decodierung.

Der zugehörige endliche deterministische Automat lautet $A_{Morse} = (Q, \Sigma, \Omega, \delta, \{q_0\}, \{q_0\})$ mit $Q = \{q_0, q_1, q_2, ..., q_{30}\}$, $\Sigma=\{\bullet,-,\Diamond\}$, $\Omega=\{a, b, ..., z, ä, ö. ü, ch\}$, und die Übergangsfunktion $\delta$ folgt aus der Zeichnung auf der nächsten Folie (die Nummerierung der Zustände ist belanglos).



**Morse-** **Code**




■

*Alle* gestrichelten Linien führen zum Anfangszustand in der Mitte.

1.6.5.6: "Reguläre Sprachen": Eine Menge $L \subseteq \Sigma^*$ heißt "endlich akzeptierbar" oder regulär, wenn es einen endlichen Akzeptor gibt, der diese Sprache akzeptiert.

Hierbei ist es egal, ob dieser Akzeptor deterministisch oder nichtdeterministisch ist, da man beweisen kann, dass man zu jedem nichtdeterministischen Akzeptor A einen deterministischen Akzeptor A' mit L(A) = L(A') konstruieren kann.

Weiterhin kann man beweisen, dass jede reguläre Sprache rechtslinear ist und umgekehrt, vgl. Definition 1.1.5.17 und daran anschließende Aussagen.

Der Begriff "reguläre Sprache" besitzt eine ähnlich zentrale Bedeutung, wie dies die Begriffe "aufzählbare Sprache" und "entscheidbare Sprache" haben. Siehe viele weitere Veranstaltungen im Laufe Ihres Studiums.



## 1.6.5.7 Nichtdeterminismus

Bereits in den 70er Jahren wurden die "bewachten Anweisungen" (guarded commands) vorgeschlagen:

Schema:

```
if Bedingung1 => Anweisung1;
[] Bedingung2 => Anweisung2;
. . .
[] Bedingungk => Anweisungk
fi
```

Bedeutung: Alle Bedingungen werden ausgewertet. Falls keine Bedingung true ist, so wird die Anweisung übersprungen, anderenfalls wird genau eine "Anweisung i", deren "Bedingung i" zu true ausgewertet wurde, nichtdeterministisch ausgewählt und ausgeführt.



Wir betrachten das Standardbeispiel ggT. In 1.1.2.4 wurden folgende Eigenschaften aufgeführt:

(1) $ggT(a,b) = ggT(b,a)$ für alle $a,b \in \mathbb{N}_0$,

(2) $ggT(a,0) = ggT(a,a) = a$ für alle $a \in \mathbb{N}_0$,

(3) $ggT(a,b) = ggT(a-b,b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a,b) = ggT(a+b,b)$ für alle $a,b \in \mathbb{N}_0$,

(4) $ggT(a,b) = ggT(b, a \bmod b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$.

Hieraus kann man sofort ein nichtdeterministisches Programm zur Berechnung des ggT angeben: Wähle ständig irgendeine der obigen Gleichungen aus und führe die zugehörige Wertzuweisung aus, bis irgendwann einmal Fall (2) eintritt.

Das Programm mit bewachten Anweisungen lautet:



```
while B ≠ 0 do
    if true => H := A; A := B; B := H
    [] A ≥ B => A := A - B
    [] true => A := A + B
    [] A ≥ B => H := A mod B; A := B; B := H
    fi
od;
put (A);
```

Hier ist keine "Strategie" vorgegeben, wie man zum Ende der Schleife kommen kann. Wie beim Nichtdeterminismus üblich lautet die Aussage nur: Wenn es eine Möglichkeit gibt, das Programmstück zu beenden, so wähle eine solche und fahre anschließend mit den nachfolgenden Anweisungen fort.



Mit dem Nichtdeterminismus kann man "schwierige" Probleme leicht beschreiben, z.B. das *Binpacking-Problem:*
Es sollen n Gegenstände, die die Gewichte $g_1, g_2, ..., g_n$ besitzen, so in m Behälter gepackt werden, dass in jedem Behälter das Maximalgewicht Max nicht überschritten wird.

*Anwendungen:*

1. Es sollen n Kisten, die jeweils $g_1, g_2, ..., g_n$ Tonnen wiegen, mit m LKWs transportiert werden, wobei jeder LKW die maximale Zuladung Max Tonnen besitzt. Geht dies?
2. Kann man n Produkte mit m Maschinen in insgesamt Max Zeiteinheiten herstellen, wobei das i-te Produkt $g_i$ Zeiteinheiten zu seiner Herstellung benötigt?
3. Optimierungsproblem:Verteile möglichst viele von n Goldstücken der Gewichte $g_1, g_2, ..., g_n$ so auf m Personen, die jede höchstens Max Gewichtseinheiten tragen können, dass das Gewicht der zurückbleibenden Goldstücke minimal ist.



Formalisierung des BPP als Entscheidungsproblem:

Definition 1.6.5.8: Binpacking Problem (BPP)

Gegeben: natürliche Zahlen n, m, Max, $g_1, g_2, ..., g_n$.
Gesucht: eine Abbildung $f: \{1, 2, ..., n\} \rightarrow \{1, 2, ..., m\}$, so dass für jedes i ($1 \leq i \leq m$) gilt:

$$\sum_{f(j)=i} g_j \leq Max.$$

f(j) = i bedeutet also: Der j-te Gegenstand wird in den i-ten Behälter gelegt.

Um das Problem, ob es zu n, m, Max, $g_1, g_2, ..., g_n$ ein solches f gibt, zu lösen, kann man deterministisch alle $m^n$ Möglichkeiten durchprobieren. Nichtdeterministisch lässt sich eine Lösung dagegen leicht beschreiben.

Nichtdeterministisches Programmstück nach der Methode "guess and check" (raten und dann nachprüfen):

```
var n, m, Max: natural;
begin  read(n, m, Max); ...
var g: array [1..n] of natural; f: array [1..n] of natural;
    B: array [1..m] of natural; Fehler: Boolean; i, j: natural;
begin   < Lies die Gewichte g ein >;
    Fehler := false;  for i:=1 to n do B[i] := 0 od;
raten (guess)
    for j := 1 to n do
        if true => f[j] := 1; [] true => f[j] := 2; ...
        [] true => f[j] := m-1; [] true => f[j] := m  fi  od;
prüfen (check)
    for j := 1 to n do B[f[j]] := B[f[j]] + g[j];
        if B[f[j]] > Max then Fehler := true fi od;
    if not Fehler then < gib die Lösung f aus > fi
end  end;
```



1.6.5.9 Backtracking

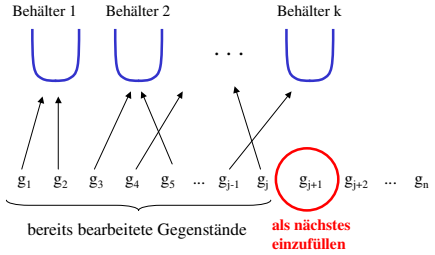
Beachten Sie, dass bei einem nichtdeterministischen Programm der Misserfolg nicht beachtet wird. Nur im Erfolgsfall (also der Fall "Es existiert eine Lösung") wird eine Lösung ausgegeben.

Ein nichtdeterministisches Programm dient der Klärung des Problems; für die Ermittlung einer Lösung ist es in der Praxis nicht hilfreich. Um nun eine Lösung zu finden, geht man systematisch alle Möglichkeiten durch: Die verschiedenen Möglichkeiten werden baumartig aufgeschrieben und systematisch durchlaufen und abgeprüft.

Ein solches systematisches Durchprobieren aller Möglichkeiten bezeichnet man als Backtracking = "Rückverfolgen durch den Lösungsbaum".

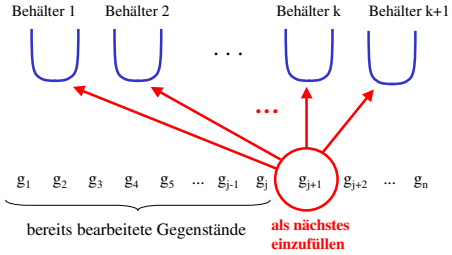


Backtracking geht stets von einer vorhandenen Situation aus und reduziert das Problem auf einfachere Probleme. Eine typische Situation beim Füllen der m Behälter ist: Man hat die ersten j Gegenstände bereits konfliktfrei in die ersten k Behälter gefüllt und muss nun den (j+1)-ten Gegenstand einfüllen.






Wir können den (j+1)-ten Gegenstand nun einem der bereits vorhandenen Behälter 1, 2, ..., k zuordnen oder einen neuen Behälter (k+1) verwenden, sofern k+1 ≤ m, d.h., k < m ist.




Alle diese k+1 Fälle probieren wir im Backtracking durch. Die folgende Prozedur vollzieht genau die obige Rekursion nach.



*Entwurf der rekursiven Prozedur (es fehlen noch Details):*

```
procedure BTBPP (j, k: natural);
var i: natural;
begin
    if  j = n  then  ..... < Lösung gefunden > .....
    else  for i:=1 to k do
        if  Gegenstand (j+1) passt noch in Behälter i  then
            BTBPP (j+1, k)  fi  od;
        if k < m then BTBPP (j+1, k+1) fi
    fi
end;
```

Um zu überprüfen, ob der Gegenstand (j+1) noch in den Behälter i passt, verwenden wir wieder das globale Feld B: array[1..m] of natural, in welchem wir die Summe der Gewichte aller Gegenstände, die aktuell im jeweiligen Behälter liegen, notieren. Der Gegenstand (j+1) darf in den Behälter i nur gelegt werden, wenn dadurch das Maximalgewicht nicht überschritten wird, also nur, wenn  B[i]+g[j+1] ≤ Max  ist.



So erhalten wir die Verfeinerung des Prozedurschemas (man beachte, dass vor und nach jedem rekursiven Aufruf das Gewicht des jeweiligen Behälters B[i] angepasst werden muss):

```
procedure BTBPP (j, k: natural);
var i: natural;
begin
    if  j = n  then  < Lösung gefunden >
    else  for i:=1 to k do
            if  B[i] + g[j+1] ≤ Max  then
                B[i]:=B[i]+g[j+1];  BTBPP (j+1, k);
                B[i]:=B[i]-g[j+1]  fi  od;
            if k < m then B[k+1]:=g[j+1];  BTBPP (j+1, k+1);
                B[k+1]:=0  fi
    fi
end;
```

Diese Prozedur stellt bisher nur fest, ob eine Lösung existiert, aber sie gibt keine mögliche Zuordnung zu den Behältern aus. Hierfür müssen wir uns noch zu jedem Gegenstand merken, in welchen Behälter er gelegt wurde. Dies geschieht in dem globalen Feld f: array [1..n] of natural. Dieses Feld beschreibt genau die Abbildung f aus Definition 1.6.5.8.

Wir müssen nicht alle $m^n$ möglichen Abbildungen f durchzuprobieren. Wir können annehmen, dass der Gegenstand mit der Nummer 1 stets im Behälter 1 liegt, der Gegenstand mit der Nummer 2 stets in einem der Behälter 1 oder 2 usw. Durch Umnummerieren der Behälter lässt sich also stets erreichen, dass $f[j] \leq j$ für alle j=1,2,...,n gilt. Speziell ist dann f[1]=1. In der Prozedur stellen wir automatisch sicher, dass $f[j] \leq j$ für alle weiteren j gilt, indem für den Gegenstand j+1 neben den bereits betrachteten Behältern nur der Behälter k+1 (und nicht k+2, k+3 usw.) ausprobiert wird. Weil m die höchste Nummer eines Behälters ist, brauchen wir also nur Abbildungen f zu betrachten mit:

$f(j) \leq \text{Min}(j,m)$, wobei Min(j,m) das Minimum der beiden Zahlen j und m ist.



So erhalten wir BTBPP (Backtrackingprozedur für das Binpackingproblem):

```
procedure BTBPP (j, k: natural);
< Beachte: Die Gegenstände 1 bis j sind bereits eingefügt, wobei k Behälter verwendet wurden >
var i: natural;
begin
    if  j = n  then  < Lösung gefunden, drucke das Feld f aus >
    else  for i:=1 to k do
              if  B[i] + g[j+1] ≤ Max  then
                  f[j+1]:=i; B[i]:=B[i]+g[j+1]; BTBPP (j+1, k);
                  B[i]:=B[i]-g[j+1]; f[j+1]:=0  fi  od;
          if k < m then f[j+1]:=k+1; B[k+1]:=g[j+1];
                  BTBPP (j+1, k+1); B[k+1]:=0; f[j+1]:=0  fi
    fi
end;
```



Globale Variablen sind wiederum:

```
Max, m, n: natural;
g, f: array[1..n] of natural;
B: array[1..m] of natural;
```

Der Aufruf der Prozedur BTBPP lautet, sofern bereits alle Daten in die globalen Variablen Max, m, n und g eingelesen wurden:

```
if (n<1) or (m<1) then
        < Abbruch, da keine Lösung zu suchen ist > fi; ...
for j:=1 to n do if g[i]>Max then
        < Abbruch, da keine Lösung möglich > fi od; ...
for i:=1 to m do f[i]:=0 od;
for j:=1 to n do B[j]:=0 od;
B[1]:=g[1]; f[1]:=1; BTBPP(1,1);
```



Manche der obigen Anweisungen erscheinen überflüssig (z.B. f[j+1]:=0 oder for i:=1 to n do f[i]:=0 od;). Wir haben sie dennoch aufgeführt, da sie eventuell sehr hilfreich werden können, wenn Fehler auftreten oder wenn die Prozedur noch von anderen Programmen aufgerufen wird.

Die Übertragung nach Ada ist nun sehr einfach.

*Aufgabe:* Schreiben Sie das Programm für die Lösung des Binpacking-Problems fertig und testen Sie es an einigen Daten. Messen Sie Zeit und Speicherplatz. Machen Sie sich das systematische Durchprobieren mittels Backtracking klar und üben Sie diese Technik an dem in 1.6.6 genannten Erbschaftsproblem.



*Hinweise:* Nichtdeterministische Sprachelemente sind in imperativen Programmiersprachen unüblich. In Ada gibt es eine nichtdeterministische Auswahl ("select-Anweisung" mit den Schlüsselwörtern select, or und else, zusammen mit task, accept, entry, delay und terminate), allerdings wurde sie auf die Kommunikation zwischen Prozessen beschränkt, siehe später in Teil 2 der Vorlesung.

In prädikativen Programmiersprachen (wie PROLOG) gehören der Nichtdeterminismus als Beschreibung und das deterministische Backtracking als Lösungsmethode zu den Grundprinzipien. Dies wird in der Vorlesung "Einführung in die Informatik III" genauer behandelt.



## 1.6.6 Historisches

Dass Algorithmen lange Laufzeiten haben können, ist seit altersher bekannt. Der Euklidische Algorithmus oder das Newtonverfahren zur Berechnung von Nullstellen sind relativ schnell arbeitende Verfahren. Die Gaußsche Elimination zur Lösung linearer Gleichungssysteme benötigt $O(n^3)$ Schritte, wobei n die Zahl der Variablen ist. Dies ist für die Berechnung per Hand meist schon zu aufwändig.

Die Erfindung von Rechenmaschinen wurde sicher durch die zeitraubenden Rechnungen, vor allem im technischen Bereich, beflügelt. Systematische Untersuchungen zur Komplexität begannen in den 1950er Jahren. Das exponentielle Wachstum mancher Algorithmen führte zu der Frage, ob man die Komplexitätsklasse **P** charakterisieren und ihr Verhältnis zur Klasse **NP**, in der viele praktische Probleme liegen, klären könne.

1971 zeigte S. Cook, dass es in **NP** Probleme mit folgender Eigenschaft gibt: Liegt nur eines dieser Probleme in **P**, so fallen die Klassen **P** und **NP** zusammen. Diese Probleme sind unter dem Begriff "NP-vollständige Probleme" bekannt geworden. Man kennt mittlerweile rund 10.000 solcher Probleme.

Hierzu zählen Zuordnungs- und Optimierungsprobleme wie die Erstellung von Stundenplänen oder optimale Standorte oder Rundreisen mit vorgegebenen Eigenschaften. Das einfachste dieser Probleme ist das "Erbschaftsproblem" (engl.: "partition problem"): Gegeben seien n natürliche Zahlen $g_1, g_2, ..., g_n$, gibt es hierzu eine Menge von Indizes $I=\{i_1, i_2, ..., i_r\} \subseteq \{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G, \quad \text{wobei } G = \left(\sum_{i=1}^{n} g_i\right) / 2 \quad ?$$

Kann man also die n Zahlen in zwei Teilmengen zerlegen, deren Summe gleich ist?



Falls Sie einmal an diesem einfach aussehenden Problem üben wollen, so stellen Sie für folgende 20 Zahlen, deren Summe 33.480.070 ist (d.h. G = 16.740.035), fest, ob es eine Teilmenge gibt, deren Summe gleich G ist:

1.976.834, 1.864.558, 1.755.621, 1.575.931, 2.169.504, 1.567.429, 2.001.571, 1.682.544, 1.289.337, 1.223.752, 1.884.283, 1.671.449, 1.400.530, 1.547.733, 1.338.626, 1.438.792, 2.010.563, 1.422.589, 1.863.866, 1.794.558

Mittlerweile ist die "Komplexitätstheorie", die eng mit der "Algorithmik" zusammenhängt, ein etablierter Zweig der Informatik. Auch wenn hier viele abstrakte Erkenntnisse gewonnen wurden, so hat man keine Techniken entwickeln können, um zu beweisen, dass es keine Turingmaschinen (oder Programme) geben kann, die gewisse Probleme in polynomieller Zeit lösen.



Schon um 1955 gelang es dagegen, eine Klasse von Problemen zu entdecken, die sich in Realzeit durch Schaltwerke mit endlichem Gedächtnis (= EA) lösen lassen. Ausgangspunkt waren Arbeiten von Kleene über Nervennetze.

E. F. Moore veröffentlichte 1956 "Gedanken Experiments on Sequential Machines" (in den Automata Studies, Princeton University, Princeton, N.J.) und G. H. Mealy untersuchte "A Method for Synthesizing Sequential Circuits" (Bell System Technical Journal 34,1955). Beide Arbeiten waren grundlegend für die Beschreibung von Schaltwerken auf höherer Ebene (= endliche Automaten) und für die Synthese von Hardwarebausteinen. Ab 1959 wurden diese regulären Probleme exakt klassifiziert.

Man erkannte rasch, dass man komplexere Rechenmodelle benötigte: Kellerautomaten, Stackautomaten, linear beschränkte und weitere Automaten wurden ab 1961 definiert und analysiert.



Immer wieder stieß man auf das Problem, nichtdeterministische Verfahren deterministisch simulieren zu müssen. Es entstanden hunderte von Klassen und Hierarchien, ohne dass man essentielle Hinweise zur Lösung des P=NP-Problems erhielt.

Heute weiß man von sehr vielen Problemen ziemlich genau, in welcher Komplexitätsklasse sie liegen; z.B. lässt sich die Frage, ob eine Zahl eine Primzahl ist, in polynomieller Zeit (bzgl. der Länge der eingegebenen Zahl) lösen. Doch wir wissen noch viel zu wenig über die (Nicht-) Existenz mathematischer Räume, in denen man sehr effizient rechnen und zwischen denen man relativ schnell hin- und herschalten kann.

Alle diese Fragen werden durch parallele und verteilte Algorithmen noch verschärft. Hier kommen neue Komplexitätsklassen ins Spiel, auf die wir erst im Hauptstudium eingehen werden.