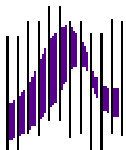
Grundlagen der Informatik II

Wolfgang Ertel

WS 2004/05



FH Ravensburg-Weingarten
Hochschule für Technik und Sozialwesen

Inhaltsverzeichnis

1 Einfache Sortieralgorithmen **3**
1.1 Sortieren durch Einfügen . . . 3
1.2 Quicksort . . . 8
1.3 Sortieren mit Bäumen (Heapsort) . . . 13
1.4 Sortieren in linearer Zeit . . . 18

2 Algorithmen auf Graphen **20**
2.1 Einführung, Repräsentation, Algorithmen . . . 20
2.2 Kürzeste Wege . . . 48
2.3 Das Problem des Handlungsreisenden . . . 51
2.4 Planare Graphen . . . 53

3 Formale Sprachen und Maschinenmodelle **55**
3.1 Grundlagen . . . 55
3.2 Grammatiken . . . 57
3.3 Chomsky-Hierarchie . . . 61
3.4 Endliche Automaten . . . 62
3.5 Reguläre Ausdrücke . . . 65
3.6 Der Lexical Analyzer LEX . . . 67
3.7 YACC: Yet Another Compiler Compiler . . . 67
3.8 Kellerautomaten . . . 69
3.9 Turingmaschinen . . . 72
3.10 Zusammenfassung zu Sprachen und Maschinenmodellen . . . 75

4 Übungen **76**
4.1 Sortieren . . . 76
4.2 Graphen . . . 78
4.3 Formale Sprachen und Automaten . . . 80

5 Lösungen zu den Übungsaufgaben **85**
5.1 Sortieren . . . 85
5.2 Graphen . . . 89
5.3 Formale Sprachen und Automaten . . . 92

Literaturverzeichnis **98**

Kapitel 1

Einfache Sortieralgorithmen

1.1 Sortieren durch Einfügen

Beispiel 1.1

7	**4**	8	3	5	1
4	7	**8**	3	5	1
4	7	8	**3**	5	1
3	4	7	8	**5**	1
3	4	5	7	8	**1**
1	3	4	5	7	8

Definition 1.1 Eine Liste $A = (A_1, \ \ldots, \ A_n)$ heisst sortiert, wenn für alle $i = 1, \ldots, n-1$ gilt $A_i \leq A_{i+1}$.

1.1.1 Algorithmus (grob)

Von links nach rechts: eine Zahl x wählen und verschieben nach links bis $A_{i-1} \leq A_i = x \leq A_{i+1}$ erfüllt ist.

```
For i:= 2 To n Do
    x:= a[i];
    "füge x am richtigen Platz ein"
```

1.1.2 Algorithmus als PASCAL-Programm

```
Program sortieren(input,output);
CONST n = 8000;

TYPE
   index = 0..n;
   item = RECORD
              key : Integer;
              data : String[20]
           END;
itemarray = ARRAY [0..n] OF item;

VAR a : itemarray;
    i,j : Index;
```

Code	Rechenzeit worst case ($T_{min}(n)$)	Rechenzeit best case ($T_{max}(n)$)
`Procedure ins_sort(var a : itemarray);`		
`VAR`		
`i,j : index;`		
`x : item;`		
`BEGIN`		
`FOR i := 2 TO n DO`	$(n-1) \cdot (I+C)$	$(n-1) \cdot (I+C)$
`BEGIN`		
`x := a[i]; a[0] := x; j := i-1;`	$(n-1) \cdot (3M+I)$	$(n-1) \cdot (3M+I)$
`WHILE x.key < a[j].key DO`	$\sum_{i=2}^{n} i \cdot C$	$(n-1) \cdot C$
`BEGIN`		
`a[j+1] := a[j];`	$\sum_{i=2}^{n} i \cdot (M+I)$	0
`j := j-1;`	$\sum_{i=2}^{n} i \cdot (M+2I)$	0
`END;`		
`a[j+1] := x`	$(n-1) \cdot (I+M)$	$(n-1) \cdot (I+M)$
`END`		
`END;`		

Durch kopieren von $a(i)$ auf die Variable x und auf $a(0)$ wird die Terminierung der Schleife sichergestellt. Sollte die Zahl $a(i)$ also kleiner als alle Elemente davor in der Liste sein und ganz nach links verschoben werden müssen würde der Schleifenindex i ohne diese Terminierung einen negativen Wert annehmen. Da die Liste aber bei $a(0)$ aufhört, würde dies zu einem undefinierten Zustand führen. Diese Terminierung führt zu einem weiteren Schleifendurchlauf. Dieser Durchlauf fällt aber nicht weiter ins Gewicht, da es sich um einen konstanten Aufwand handelt. Dagegen wird im Vergleich zu einer zusätzlichen Bedingung im Kopf der While-Schleife ein Aufwand proportional zu n eingespart.

1.1.3 Worst - Case Rechenzeit

Im Programmcode eingetragen ist für jede Programmzeile die Laufzeit. Als Parameter treten rechnerabhängigen Größen I (Increment), M (Move) und C (Compare) auf:
I = Rechenzeit für eine Zähloperation
M = Rechenzeit für eine Zuweisungsoperation
C = Rechenzeit für eine Vergleichsoperation

Diese Zeiten sind konstant, d.h. nicht von n abhängig und deshalb für die Berechnung der Komplexität unbedeutend. Für die Berechnung von exakten Laufzeiten sind sie jedoch sehr wichtig.

$$\begin{aligned} T_{max}(n) &= (n-1) \cdot (I + C + 3M + I + M + I) + \sum_{i=2}^{n} (i \cdot C + i \cdot (M + 2I)) \\ &= (3I + 4M + C) \cdot n - (3I + 4M + C) + (2I + M + C) \cdot \sum_{i=2}^{n} i \end{aligned}$$

mit

$$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{n^2 + n - 2}{2} = \frac{n^2}{2} + \frac{n}{2} - 1$$

erhalten wir

$$T_{max}(n) = \overbrace{(I + \frac{M}{2} + \frac{C}{2})}^{a} \cdot n^2 + \overbrace{(4I + \frac{9}{2}M + \frac{3}{2}C)}^{b} \cdot n \overbrace{-(5I + 5M + 2C)}^{c}$$
$$= a \cdot n^2 + b \cdot n + c$$

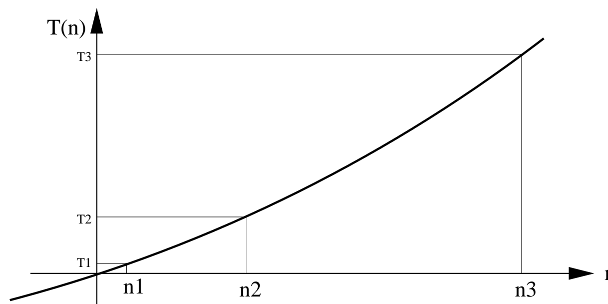
Wenn die Konstanten nicht interessieren, kann die Berechnung der Worst-Case Rechenzeit vereinfacht werden:

$$\begin{array}{rcl} (n-1) \cdot (I + C) & \rightarrow & (n-1) \cdot c_1 \\ (n-1) \cdot (3M + I) & \rightarrow & (n-1) \cdot c_2 \\ \sum_{i=2}^{n} i \cdot C & \rightarrow & (\sum_{i=2}^{2} i) \cdot c_3 \\ \vdots & \vdots & \vdots \end{array}$$

Ergebnis: $a \cdot n^2 + b \cdot n + c$

Bestimmung von a, b und c

1.) Messe drei Zeiten für verschiedene $n = n_1, n_2, n_3$
$\longrightarrow (n_1, T_1), (n_2, T_2), (n_3, T_3)$



2.) Einsetzen in Parabelgleichung:
$T_1 = a \cdot n_1^2 + b \cdot n_1 + c$
$T_2 = a \cdot n_2^2 + b \cdot n_2 + c$
$T_3 = a \cdot n_3^2 + b \cdot n_3 + c$

3.) Auflösen des linearen Gleichungssystems nach a, b und c (siehe Übung 1).

Beispiel 1.2 Vergleich von 3 verschiedenen Komplexitäten

n	Algorithmus 1 $T(n) = \log n \cdot c$	Algorithmus 2 $T(n) = c \cdot n$	Algorithmus 3 $T(n) = c \cdot n^2$	Algorithmus 4 $T(n) = c \cdot n^3$	Algorithmus 5 $T(n) = c \cdot 2^n$
10	1 s	10 s	100 s	1 000 s	1024 s
100	2 s	100 s	10 000 s	1 000 000 s	$\approx 10^{30}$ s
1000	3 s	1000 s	1 000 000 s	1 000 000 000 s	$\approx 10^{300}$ s

Die Werte wurden mit einem c von einer Sekunde errechnet. Eine bessere Hardware würde nur dieses c verbessern und dadurch die Rechenzeit nicht wesentlich beeinflussen. Bei großen n ist die **Komplexität** viel ausschlaggebender als die Konstante c.

1.1.4 Best-Case Rechenzeit

Ist die Liste A vorsortiert, so wird die While-Schleife nicht durchlaufen und das Programm ist viel schneller.
Analog zu den Berechnungen im Worst-Case kann man auch im Best-Case die Zeiten aus der Tabelle aufaddieren und man erhält

$$T_{min}(n) = d \cdot n + e.$$

1.1.5 Einschub: Asymptotik

Beschreibung des asymptotischen Verhaltens von Rechenzeiten $T(n)$ (oder Speicherplatz oder anderer Funktionen) für $n \to \infty$.
Idee: Vernachlässigung von konstanten Faktoren. Nur die Abhängigkeit von n für große n ist wichtig.

Definition 1.2 Seien die Funktionen $f : \mathbb{N} \to \mathbb{R}^+$, $g : \mathbb{N} \to \mathbb{R}^+$ gegeben. Dann schreibt man:

Asymptotische obere Schranke:

$$f(n) = O(g(n)) \quad \Leftrightarrow \quad \exists\, c \in \mathbb{R}^+ : \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$

$$f(n) = o(g(n)) \quad \Leftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Asymptotische untere Schranke:

$$f(n) = \Omega(g(n)) \quad \Leftrightarrow \quad \exists\, c \in \mathbb{R}^+ : \lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$$

$$f(n) = \omega(g(n)) \quad \Leftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Asymptotische enge (harte) Schranke:

$$f(n) = \Theta(g(n)) \quad \Leftrightarrow \quad \exists\, c_1, c_2 \in \mathbb{R}^+ : \ c_1 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c_2$$

$O, \Omega, \Theta, o, \omega$ sind Relationen auf Funktionen, wie z.B. $<, \leq$ auf reellen Zahlen.
Analogien:

Funktionen $f : \mathbb{N} \to \mathbb{R}^+$	reelle Zahlen
$f(n) = O(g(n))$	$a \leq b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = \omega(g(n))$	$a > b$
$f(n) = \Theta(g(n))$	$a = b$

Satz 1.1

$$T(n) = \Theta(g(n)) \quad \Leftrightarrow \quad T(n) = \Omega(g(n)) \text{ und } T(n) = O(g(n))$$

Die bisher berechneten Ergebnisse lassen sich nun also formulieren wie folgt:

Satz 1.2 Beim Sortieren durch Einfügen gilt

$$\begin{aligned} T_{min}(n) &= \Theta(n) \\ T_{max}(n) &= \Theta(n^2) \\ T(n) &= \Omega(n) \\ T(n) &= O(n^2) \end{aligned}$$

1.1.6 Schwachstellen und mögliche Verbesserungen

Die Suche nach der Einfügestelle für das Element x soll verbessert werden.

Suche in Listen

Aufgabe 1: Suche ein Element x in einer **beliebigen** Liste.

optimale Lösung: lineare Suche mit $T_{max}(n) = \Theta(n)$

Aufgabe 2: Suche ein Element x in einer **sortierten** Liste $A[1 \ldots n]$

optimale Lösung: Bisektion

Der Bereich links von x ist bereits sortiert. Man findet die richtige Stelle für x indem man die Liste links von x mehrfach halbiert.

Definition 1.3 $\lfloor x \rfloor := \max\{y \in \mathbb{Z} | y \leq x\}$. $\lfloor x \rfloor$ ist also die größte ganze Zahl kleiner oder gleich x. Analog sei $\lceil x \rceil = \min\{y \in \mathbb{Z} | y \geq x\}$.

Der Bisektionsalgorithmus

```
a = 1
b = n
m = ⌊(a+b)/2⌋
while A[m] ≠ x & b > a
   If x < A[m]
       Then b = m
       Else a = m
   m = ⌊(a+b)/2⌋
If A[m] == x
   Then print (''Hurra '',x,''gefunden an position '',m)
   Else print (''Schade, '',x,''nicht in A'')
```

Komplexität

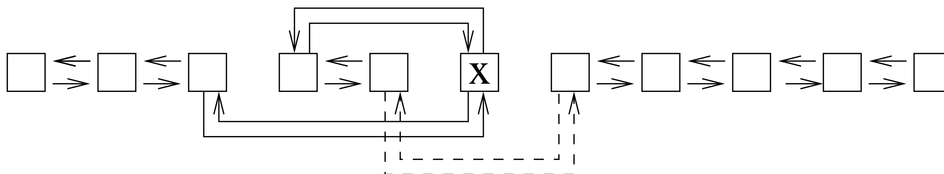
Sei die Arraylänge $n = 2^k$. Dann sind höchstens k Wiederholungen der While-Schleife erforderlich. Also ist die Rechenzeit proportional zu k, d.h. $T(n) = c \cdot k$.

$$\begin{aligned} n &= 2^k \\ lnn &= k \cdot \ln 2 \\ k &= \frac{\ln\ n}{\ln\ 2} \end{aligned}$$

Also gilt für die Bisektion

$$T(n) = c \cdot \frac{ln\ n}{ln\ 2} = c \cdot \log_2 n = O(\log n)$$

Erläuterung: An $\log_2 x = \frac{ln\ x}{ln\ 2} = c \cdot ln\ x \leftarrow \log n$, erkennt man, daß sich alle Logarithmen nur um einen kostanten Faktor unterscheiden.
Das Suchen der Einfügestelle ist mit logarithmischem Aufwand möglich.
Aber: verschieben der O(n) Arrayelemente im Array kostet linearen Aufwand.
Idee: verwende dynamische Datenstruktur als verkettete Liste.



Verschieben der Arrayelemente ist damit unnötig. x wird direkt an der richtigen Stelle eingefügt.
Aber: die Bisektion ist auf einer verketteten Liste nicht anwendbar.
Daher: für Sortieren durch Einfügen bleibt $T_{max}(n) = \Theta(n^2)$

1.2 Quicksort

Beispiel 1.3 Es soll die Liste $5, 3, 2, 6, 4, 1, 3, 7$ sortiert werden. Dies erfolgt nach dem Prinzip *divide and conquer* durch rekursiv wiederholtes Aufteilen und bearbeiten, wie in folgender Tabelle zu sehen:

	5	3	2	6	4	1	3	7		x = 5 (Pivotelement)
$\upharpoonleft_i$									$\upharpoonleft_j$	
	5	3	2	6	4	1	3	7		
	$\upharpoonleft_i$			$\longleftrightarrow$			$\upharpoonleft_j$			
	3	3	2	6	4	1	5	7		
				$\upharpoonleft_i$	$\leftrightarrow$	$\upharpoonleft_j$				
	3	3	2	1	4	6	5	7		
					$\upharpoonleft_j$	$\upharpoonleft_i$				
	3	3	2	1	4	**6**	5	7		$x = 3$, $x = 6$
	$\upharpoonleft_i$	$\longleftrightarrow$		$\upharpoonleft_j$		$\upharpoonleft_i \leftrightarrow$	$\upharpoonleft_j$			
	1	3	2	3	4	5	**6**	7		
		$\upharpoonleft_i \leftrightarrow$	$\upharpoonleft_j$			$\upharpoonleft_j$	$\upharpoonleft_i$			
	1	2	**3**	3	4	5	6	7		
		$\upharpoonleft_j$	$\upharpoonleft_i$				$\upharpoonleft_i \upharpoonleft_j$			
	1	2	3	3	4	5	6	7		
	$\upharpoonleft_i \upharpoonleft_j$		$\upharpoonleft_i \leftrightarrow$	$\upharpoonleft_j$						
	1	2	3	**3**	4	5	6	7		
			$\upharpoonleft_j$	$\upharpoonleft_i$						
	1	2	3	3	4	5	6	7		
				$\upharpoonleft_i \upharpoonleft_j$						
	1	2	3	3	4	5	6	7		

Erläuterung: Der erste Wert der Liste ist das Pivotelement (ausgezeichnetes, besonderes Element). Zwei Indizes i und j durchlaufen die Liste nach folgendem Kriterium:
i sucht von links nach einem Wert $\geq x$ und j von rechts nach einem Wert $\leq$ x. Dann werden i und j vertauscht. Das setzt sich solange fort bis sich beide Indizes überkreuzen oder gleich sind. Dann wird die Liste rechts von j geteilt und auf beiden Hälften rekursiv von vorne begonnen.

1.2.1 Der Algorithmus

Seien A=Liste, p=Anfangsindex, r=Endeindex. Dann ist die rekursive Struktur gegeben durch:

```
Quicksort(A,p,r)
   if p<r
      then q=Partition (A,p,r)
         Quicksort (A,p,q)
         Quicksort (A,q+1,r)
```

Das Aufteilen erfolgt mittels

```
Partition (A,p,r)
   x=A[p]
   i=p-1
   j=r+1
   while TRUE do
      repeat j=j-1
         until A[j] ≤ x
      repeat i=i+1
         until A[i]≥ x
      if i<j
         then vertausche A[i] mit A[j]
         else Return(j)
```

Der erste Aufruf von Quicksort auf einer Liste der Länge n erfolgt durch

```
Quicksort(A,1,length(A)).
```

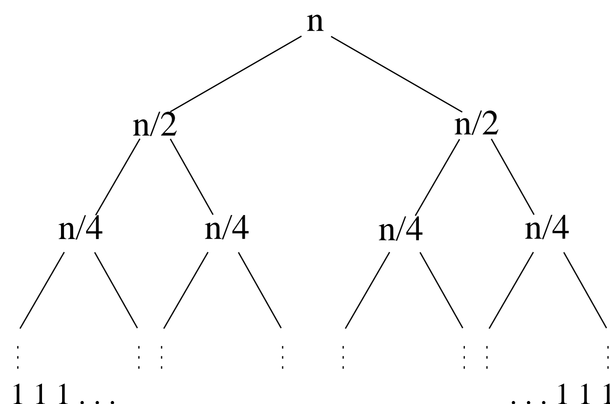
1.2.2 Analyse

Laufzeit von Partition

auf Array $A[p...r]$ mit $n = r - p + 1$

$$T(n) = \Theta(n)$$

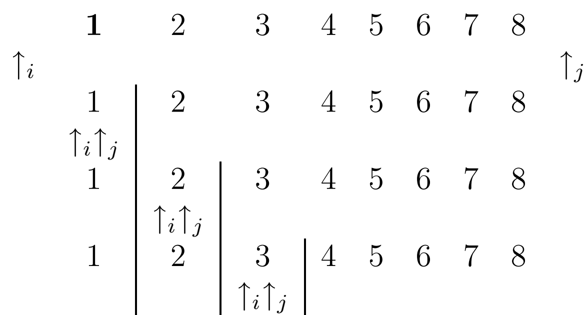
Laufzeit von Quicksort (Best Case)



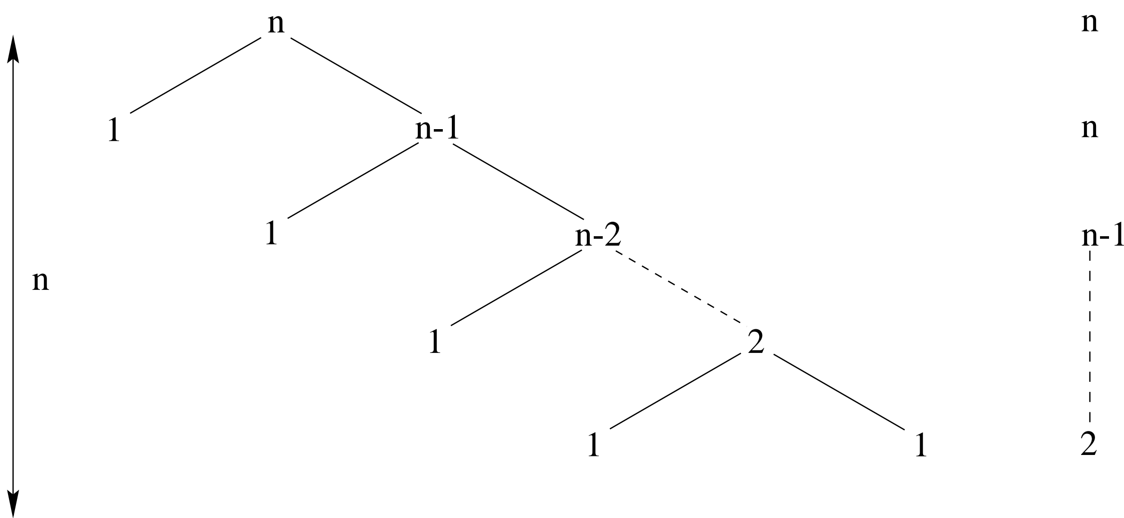
Komplexität auf jeder Ebene ist $\Theta(n) \Longrightarrow$ Komplexität gesamt: $\Theta(n \cdot \log_2 n)$

Zahl der Ebenen im Rekursionsbaum? Der Baum wächst in der Breite exponentiell:
$n = 2^k \rightarrow k =$ Tiefe des Baumes
$\log_2 n = \log_2 2^k = k$
$T_{min}(n) = c \cdot n \cdot \log_2 n = \Theta(n \log n)$

Laufzeit von Quicksort (Worst Case, sortierte Liste)



Rekursionsbaum



$$
\begin{aligned}
T_{max}(n) &= c_2 \cdot (\sum_{i=2}^{n} i + n) \\
&= c_2 \cdot (\frac{n(n+1)}{2} - 1 + n) \\
&= c_2 \cdot (\frac{n^2}{2} + \frac{3}{2}n - 1) \\
&= \Theta(n^2)
\end{aligned}
$$

Für die Rechenzeit T(n) gilt:

$c_1 \cdot n \log n \leq T(n) \leq c_2 \cdot n^2$

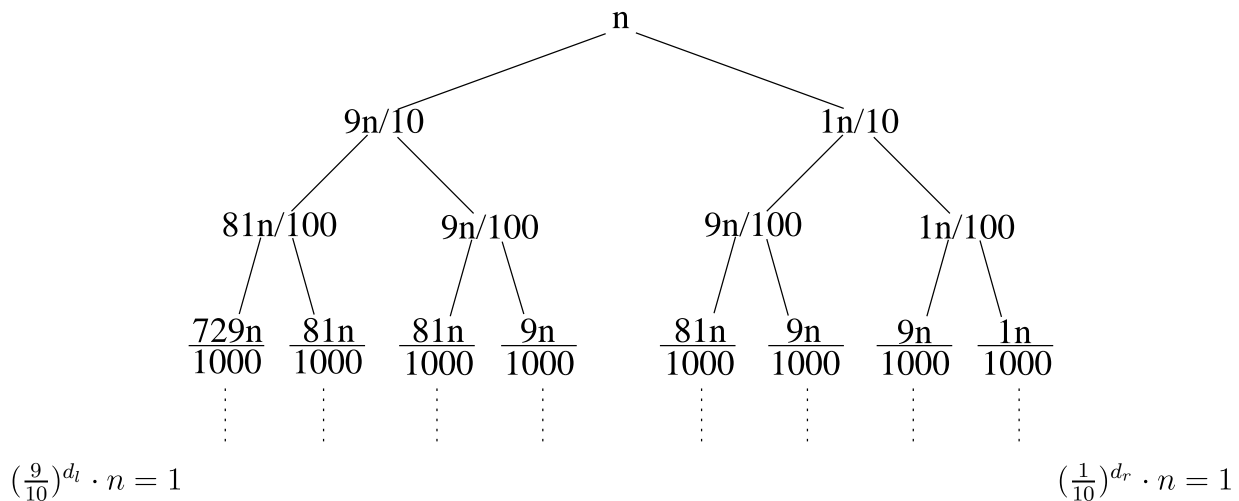
Idee zur Verhinderung des Worst Case: Liste vorher zufällig permutieren.
Begründung: Die Wahrscheinlichkeit für das Erzeugen einer sortierten Liste ist sehr klein.
$\frac{1}{10^n}$ falls nur 10 Zahlen erlaubt sind.
Allgemeine Wahrscheinlichkeit: $\frac{1}{M^n}$ für M unterschiedliche zu sortierende Werte.

Beispiel 1.4 32 Bit Integer Zahlen: $M = 2^{32}$, $n = 10^6$, $\frac{1}{M^n} = \frac{1}{(2^{32})^{10^6}} = \frac{1}{2^{32 \cdot 10^6}}$

Folgerung: Die Wahrscheinlichkeit, dass ein randomisiertes Quicksort den Worst-Case trifft ist **sehr** klein. Aber die zufällige Permutation der Liste vor dem Sortieren kostet Rechenzeit. Daher:

Zufällige Wahl des Pivotelements
ersetze `x = A[p]` durch `x = A[random (p,r)]`
Average-Case-Analyse: Berechnung der mittleren Laufzeit des Algorithmus z.B. Quicksort auf einer repräsentativen Menge von Eingaben.

Beispiel 1.5 Die Average-Case-Analyse für Quicksort ist sehr schwierig. Daher wird hier der ungünstige Fall eines konstanten Aufteilungsverhältnisses $1 : n$ am Beispiel $1 : 9$ behandelt.



$(\frac{9}{10})^{d_l} \cdot n = 1$ $\qquad$ $(\frac{1}{10})^{d_r} \cdot n = 1$

$$d_l \cdot \log \frac{9}{10} + \log n = 0$$

$$d_l = \frac{\log n}{\log \frac{10}{9}}$$

$$T(n) < \frac{c}{\log \frac{10}{9}} \cdot n \log n$$

Tiefe $k = \frac{\log n}{\log \frac{10}{9}} \approx 6,6 \cdot \log_{10} n$

Aufwand auf jeder Ebene $\leq n \quad \Rightarrow \quad T(n) = O(n \cdot \log n)$. Da die komplexität von Quicksort im Best-Case auch $n \cdot \log n$ ist, gilt:

Satz 1.3 Quicksort besitzt im Average-Case die Komplexität $T(n) = \Theta(n \cdot \log n)$

Bemerkung: Der Worst-Case tritt bei Quicksort sehr selten auf. Er kommt bei vorsortierten Listen vor, diese sind in der Praxis allerdings häufig.

Verbesserung: zufällige Wahl des Pivot-Elements, d.h. Ersetzung von Partition (A,p,r) durch:

```
Random-Partition (A,p,r)
   i=Random (p,r)
      vertausche A[p] mit A[i]
      return Partition (A,p,r)
```

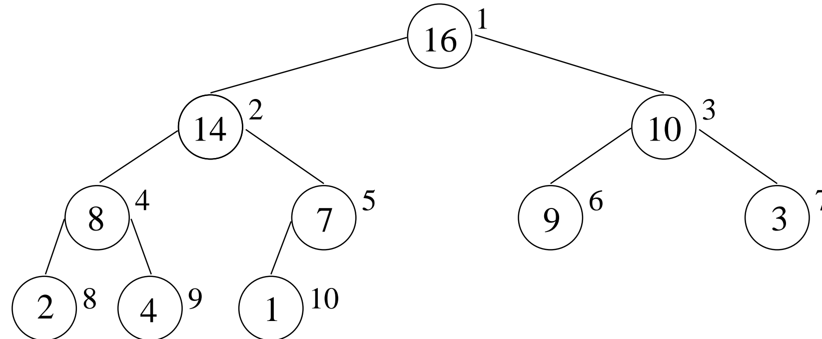
Random (p,r) liefert zufällig mit konstanter Wahrscheinlichkeit eine Zahl aus {p, p+1, ...,r}.
Bemerkung: Durch Randomisierung gibt es **keine Eingabe** mehr, für die der Algorithmus **immer Worst-Case Verhalten** zeigt.

1.3 Sortieren mit Bäumen (Heapsort)

Beispiel 1.6

i :	1	2	3	4	5	6	7	8	9	10
$A[i]$:	16	14	10	8	7	9	3	2	4	1

Darstellung von A als Baum:



Keine neue Datenstruktur nötig, sondern nur Funktionen parent, left, right mit den Eigenschaften

$$\begin{aligned} \text{parent}(i) &= \lfloor \tfrac{i}{2} \rfloor && \text{(Vorgänger)} \\ \text{left}(i) &= 2i && \text{(linker Nachfolger)} \\ \text{right}(i) &= 2i+1 && \text{(rechter Nachfolger)} \end{aligned}$$

Definition 1.4 Ein Array mit den Funktionen left, right, parent heisst **heap**, wenn gilt

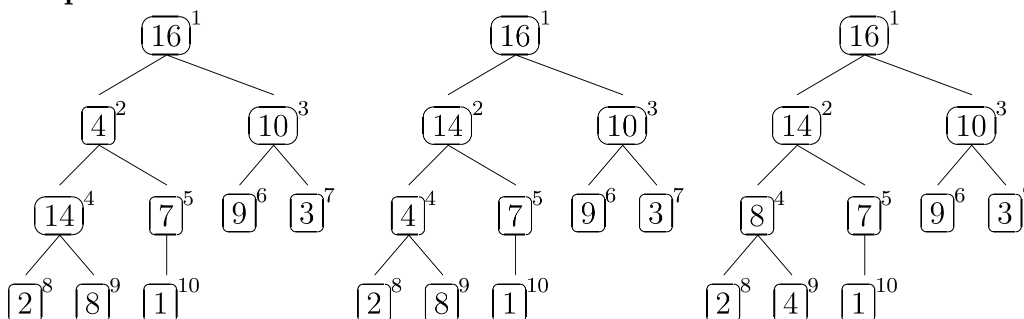
$$A[parent(i)] \geq A[i]$$

Definition 1.5 Die **Höhe eines Knotens** i in einem Baum ist gleich der Zahl der Knoten im längsten Pfad von i zu einem Blattknoten. Die **Höhe des Baumes** ist gleich der Höhe des Wurzelknotens.

Der folgende Algorithmus verwandelt einen Binärbaum in einen Heap unter der Voraussetzung, dass beide Unterbäume schon Heaps sind.

```
Heapify(A,i)
   vertausche A[i] mit seinem größten Nachfolger
   A[largest] falls A[largest] > A[i]
   heapify (A,largest)
```

Beispiel 1.7



Der Algorithmus

```
Heapify(A,i)
l = Left(i)
r = Right(i)
if l ≤ heapsize(A) AND A[l] > A[i]
   then largest = l
   else largest = i
if r ≤ heapsize(A) AND A[r] > A[largest]
   then largest = r
if largest ≠ i
   then vertausche A[i] mit A[largest]
      Heapify(A, largest)
```

Laufzeit von heapify

Vergleich von $A[i]$ mit $A[left(i)]$ und $A[right(i)]$: konstante Zeit= $\Theta(1)$

Rekursion

maximale Zahl von Knoten in einem der Unterbäume left(i), right(i) = $\frac{2}{3}n$, wenn n = Zahl der Unterknoten in i.

Rekurrenzrelation

$$T(n) \leq T(\frac{2}{3}n) + \Theta(1)$$

Das Auflösen der Rekurrenzrelation erfolgt mit Hilfe des Mastertheorems:

Satz 1.4 (Mastertheorem) Seien $a \geq 1$, $b > 1$ Konstanten und $f : R_+ \to R_+$, $T : R_+ \to R_+$ Funtionen mit

$$T(n) = aT(n/b) + f(n)$$

wobei n/b für die ganzzahlige Division ($\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$) steht. Dann kann $T(n)$ asymptotisch beschränkt werden durch

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{falls} \quad \exists_{\varepsilon>0} : \quad f(n) = O(n^{\log_b a-\varepsilon}) \\ \Theta(n^{\log_b a} \log\ n) & \text{falls} \quad f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{falls} \quad \exists_{\varepsilon>0} : \quad f(n) = \Omega(n^{\log_b a+\varepsilon}) \text{ und} \\ & \qquad\quad \exists_{c<1}\exists_{n_0}\forall_{n\geq n_0} : \quad af(n/b) \leq cf(n) \end{cases}$$

Für Heapsort gilt $T(n) \leq T(\frac{2}{3}n) + \Theta(1)$. Für den Worst Case gilt

$$T_{max}(n) = 1 \cdot T\left(\frac{n}{3/2}\right) + c$$

Für die Anwendung des Mastertheorems gilt dann also $a = 1$, $b = \frac{3}{2}$, $f(n) = 1$ sowie

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = 1$$

Es ist der zweite Fall anwendbar, denn $f(n) = \Theta(n^{\log_b a})$. Also gilt für heapify

$$T_{max}(n) = \Theta(1 \cdot \log n) = \Theta(\log n).$$

Beispiel 1.8 Anwendung des Master-Theorems auf Quicksort (Best Case):

$T(n) = 2\,T(\frac{n}{2}) + c \cdot n$
$a = 2,\ b = 2,\ f(n) = c \cdot n$
$n^{\log_b a} = n^{\log_2 2} = n$
$f(n) = \Theta(n)$
$\Rightarrow \quad$ 2. Fall: $T(n) = \Theta(n \cdot \log n)$

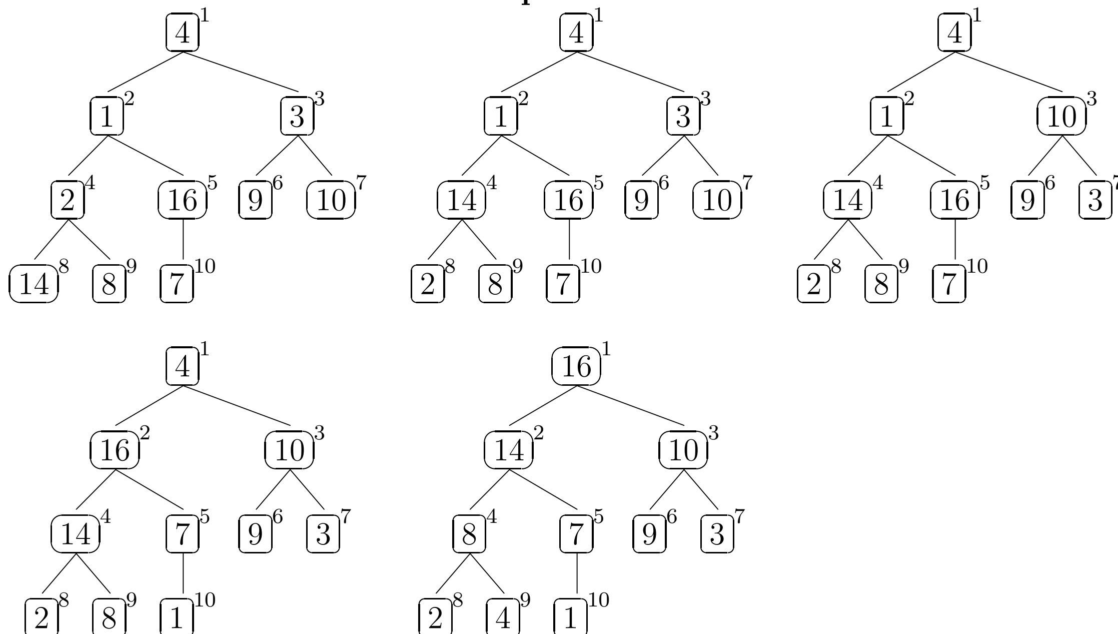
Beispiel 1.9 wie oben, jedoch mit anderem $f(n)$:

$T(n) = 2T(\frac{n}{2}) + c \cdot \log n$
$a = 2,\ b = 2,\ f(n) = c \cdot \log n$
$n^{\log_b a} = n$
$f(n) = c \cdot \log n = O(n^{(1-\varepsilon)})$
$\Rightarrow \quad$ 1. Fall: $T(n) = \Theta(n)$

1.3.1 Erzeugen eines Heap

```
Build-Heap(A)
   for i = ⌊length(A)/2⌋ downto 1
      do heapify(A,i)
```

Beispiel 1.10



Analyse

obere Schranke

Jeder der $\Theta(n)$ Aufrufe von heapify kostet $O(\log n)$ Zeit. Also gilt

$$\Longrightarrow T(n) = O(n \log n).$$

bessere Schranke

Die Laufzeit von heapify hängt von der Höhe des Knotens im Baum ab. Die meisten Knoten haben sehr niedrige Höhe!

$$\text{(Zahl der Knoten auf Höhe } h) \leq \lceil \frac{n}{2^{h+1}} \rceil$$

Die Laufzeit von heapify für Knoten der Höhe h wächst linear mit h. Also $T_{heapify}(h) = O(h) = O(\log(\text{Anzahl Knoten im Unterbaum}))$

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O(2n) = O(n)$$

Die vorletzte Gleichung gilt weil

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2,$$

was man in der Formelsammlung findet.

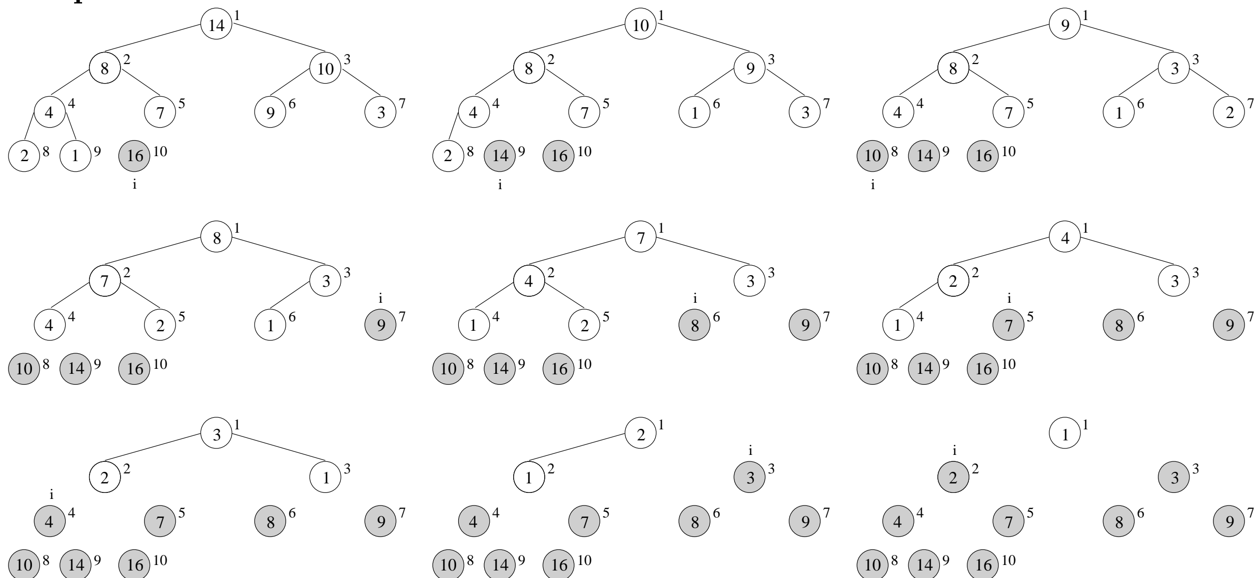
1.3.2 Der Heapsort - Algorithmus

Voraussetzung: Build-Heap setzt maximales Element an die Wurzel, d.h. in $A[1]$

Idee

Wiederhole folgende Schritte bis der Heap nur noch aus einem Element besteht: 1.) vertausche $A[1]$ mit $A[n]$
2.) lösche $A[n]$ aus dem Heap
3.) Heapify (A,1)

Beispiel 1.11



Algorithmus

```
Heapsort(A)
   Build-Heap(A)
   for i= length(A) downto 2
      do vertausche A[1] mit A[n]
         heapsize(A)= heapsize(A)-1
         Heapify(A,1)
```

Analyse

Build-Heap: $T(n) = O(n)$
for-Schleife: $(n-1)$ mal Heapify mit $T(n) = O(\log n)$ und Vertauschen mit $O(1)$

$$\begin{aligned} T(n) &\leq c_1 \cdot n + (n-1) \cdot [c_2 \cdot \log n + c_3] \\ &= c_2 \cdot n \log n + \underbrace{(c_1 + c_3) \cdot n - c_2 \log n - c_3}_{\text{vernachlässigbar für } n \to \infty} \\ &= O(n \log n) \end{aligned}$$

1.3.3 Priority-Queues als Anwendung der Heap-Struktur

Bei der Verwaltung von Warteschlangen mit Aufträgen unterschiedlicher Priorität muß als jeweils nächster Auftrag immer ein Auftrag mit höchster Priorität ausgewählt werden. Dies kann zum Beispiel erreicht werden, indem man die Aufträge in einer sortierten Liste speichert. Allerdings ist dann die Komplexität zum Speichern der Aufträge in der Liste linear in der Länge n.
Verwendet man dagegen einen Heap zur Verwaltung der Warteschlangen von Aufträgen, so ist die Komplexität zum speichern und holen von Aufträgen im Heap logarithmisch in n.

Beispiel 1.12 Einfügen eines Elements x im Heap

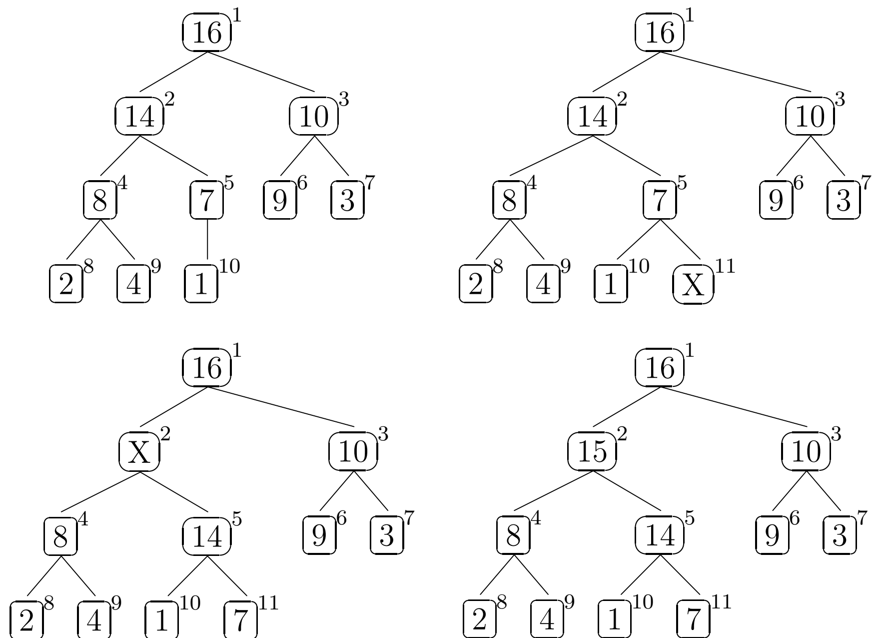
Idee

1.) neues Blatt erzeugen
2.) Pfad vom Blatt zur Wurzel durchsuchen und x an der richtigen Stelle einfügen.

Algorithmus

```
Heap-Insert(A, key)
     heapsize(A) = heapsize(A)+1
     i=heapsize(A)
     while i>1 AND A[parent(i)] < key
          do A[i] = A[parent(i)]
               i=parent(i)
          A[i]=key
```

Beispiel 1.13 Anwendung von `Heap-Insert` zum Einfügen des Elements $X = 15$



Analyse

$$T(n) = O(\log n).$$

1.4 Sortieren in linearer Zeit

Unter ganz bestimmten Bedingungen ist Sortieren sogar in linearer Zeit möglich, wie man an folgendem Beispiel erkannt.

Idee

Wenn alle in der Liste A vorkommenden Sortierschlüssel aus einer bekannten endlichen Menge sind, kann man durch einfaches Abzählen der Häufigkeiten aller in A vorkommenden Elemente eine Tabelle S der Häufigkeiten erstellen. Danach wird die Liste A überschrieben mit den Elementen in der richtigen Reihenfolge (siehe Beispiel).

Beispiel 1.14 Zu sortieren sei $A = (2, 1, 4, 5, 4, 7, 3, 1, 4, 4, 1)$. Als Häufigkeitstabelle S erhält man

Sortierschlüssel	1	2	3	4	5	6	7
Häufigkeit	3	1	1	4	1	0	1

Daraus erhält man einfach die sortierte Liste $A = (1, 1, 1, 2, 3, 4, 4, 4, 4, 5, 7)$.

Komplexität

Mit $m = |S|$ entsteht beim Abzählen ein linearer Aufwand und beim Zurückschreiben auch und man erhält

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot m + c_3 \cdot n \\ &= \Theta(n + m) \end{aligned}$$

Dieser Algorithmus ist besonders interessant, wenn die Zahl m der verwendeten Sortierschlüssel klein ist, zum Beispiel beim Sortieren einer großen Adressdatei nach Postleitzahlen. Das hier verwendete Verfahren setzt voraus, dass die Menge der verwendeten Schlüssel bekannt ist. Daher ist der aufwändige Vergleichen der Sortierschlüssel nicht nötig. Ist diese Voraussetzung jedoch nicht erfüllt und auch sonst kein Zusatzwissen über die zu sortierenden Daten vorhanden, so ist Sortieren in linearer Zeit nicht möglich und man kann für die Komplexität folgende untere Schranke angeben:

Satz 1.5 Wenn ein Sortieralgorithmus für beliebige Listen nur mit Vergleichen und Kopieren arbeitet und keine Zusatzinformation über die Liste erhält, dann gilt:

$$T(n) = \Omega(n \log n)$$

Kapitel 2

Algorithmen auf Graphen

2.1 Einführung, Repräsentation, Algorithmen

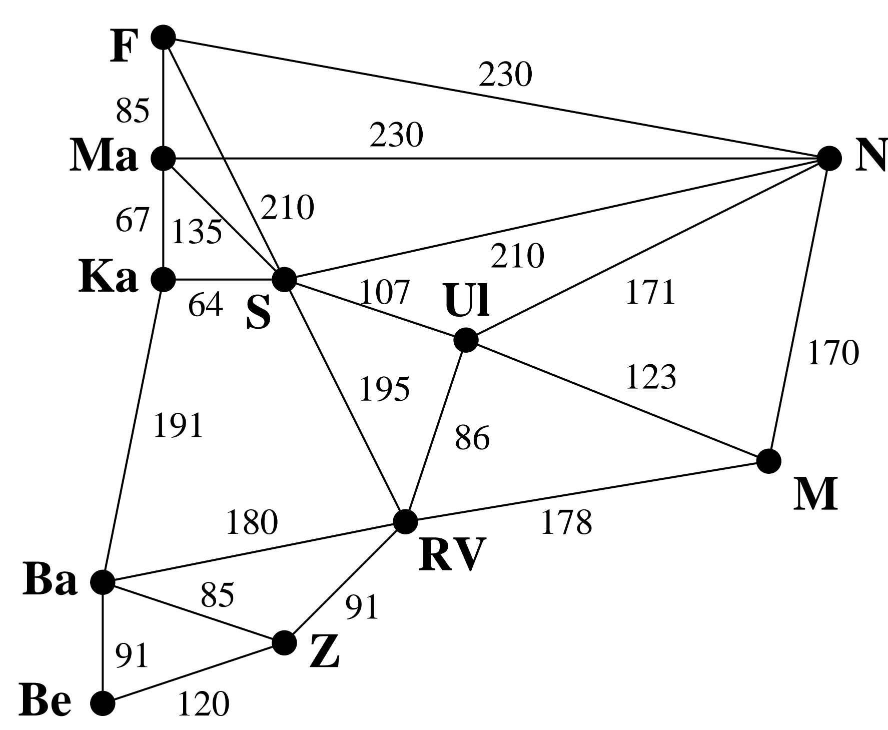
Die folgenden 25 Seiten Einführung zum Thema Graphen sind entnommen aus dem Vorlesungsskript *Praktische Informatik II* von Prof. Dr. W. Hesse, Univ. Marburg (http://www.mathematik.u

Definition 2.1 Ein **Hamilton-Kreis** (hamiltonian circle) in einem Graphen G ist ein geschlossener Weg, in dem jeder Konten aus G genau einmal vorkommt. Eine **Clique** in einem (ungerichteten) Graphen ist ein vollvernetzter Teilgraph.

2.2 Kürzeste Wege

Beispiel 2.1 Entfernungstabelle einiger süddeutscher Städte

	Ravensb.	Ulm	Münch.	Stuttg.	Karlsr.	Zürich	Basel	Bern	Frankf.	Mannh.	Nürnbg.
Abkürzung	RV	U	M	S	Ka	Z	Ba	Be	F	Ma	N
Ravensburg	0	86	178	195	–	91	180	–	–	–	–
Ulm		0	123	107	–	–	–	–	–	–	171
München			0	–	–	–	–	–	–	–	170
Stuttgart				0	64	–	–	–	210	135	210
Karlsruhe					0	–	191	–	–	67	–
Zürich						0	85	120	–	–	–
Basel							0	91	–	–	–
Bern								0	–	–	–
Frankfurt									0	85	230
Mannheim										0	230
Nürnberg											0



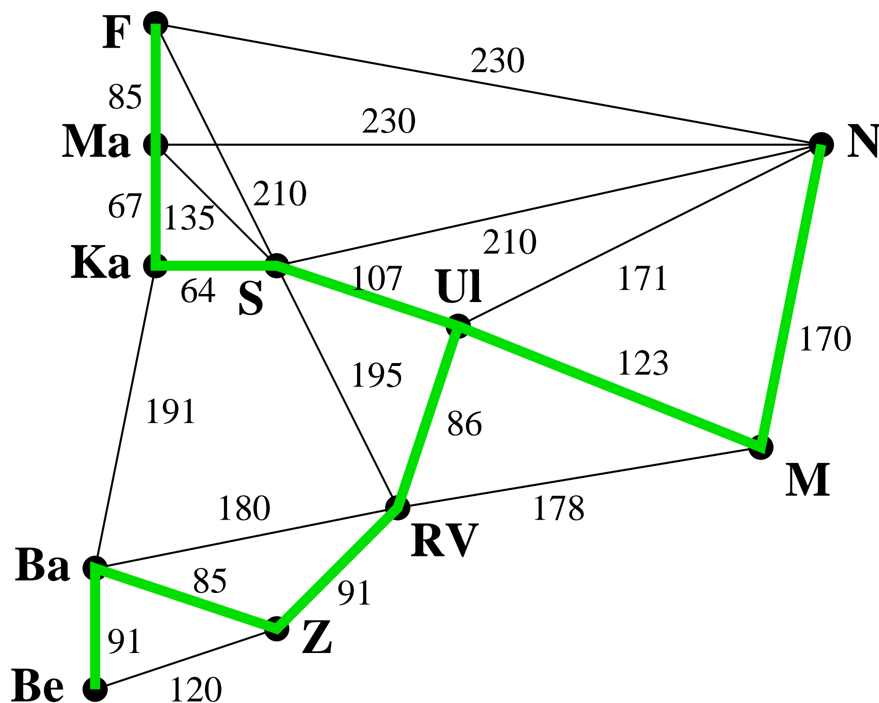
Satz 2.1 (Kruskal-Algorithmus) Für einen ungerichteten, zusammenhängenden, bewerteten Graphen findet der folgende Algorithmus einen minimal aufspannenden Baum T:

1. Setze $T = \{\}, i = 0$

2. Sortiere die Kanten nach ihrem Gewicht.

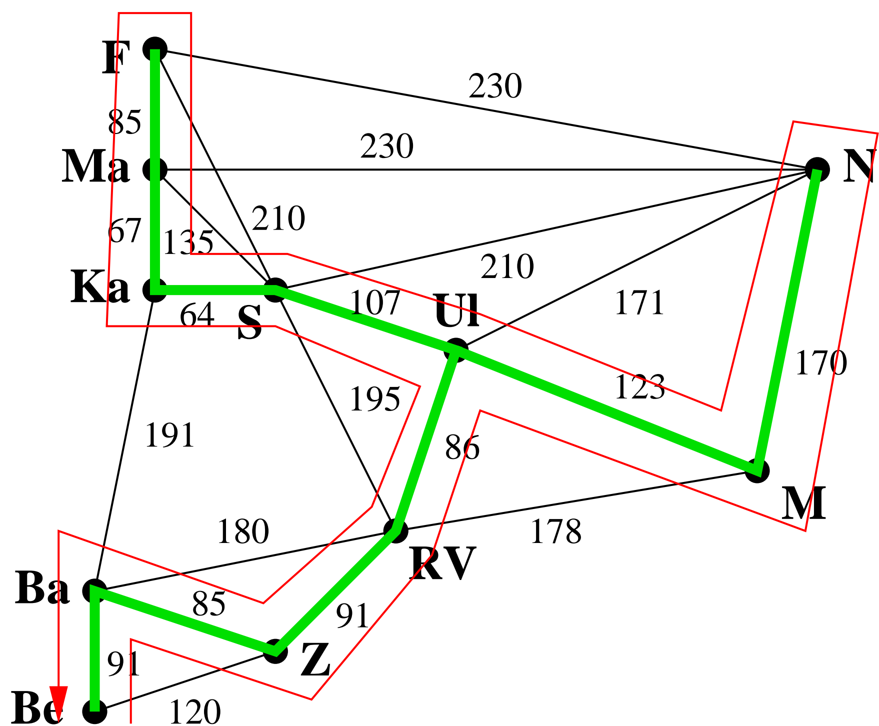
3. Wähle aus den noch nicht gewählten Kanten die mit dem kleinsten Gewicht. Falls diese Kante in T keinen Zyklus erzeugt, erweitere T um diese Kante. Wiederhole Schritt 3 bis alle Kanten verbraucht sind.

Beispiel 2.2 Wir wenden nun den Kruskal-Algorithmus auf obiges Beispiel an und erhalten folgenden minimal aufspannenden Baum:

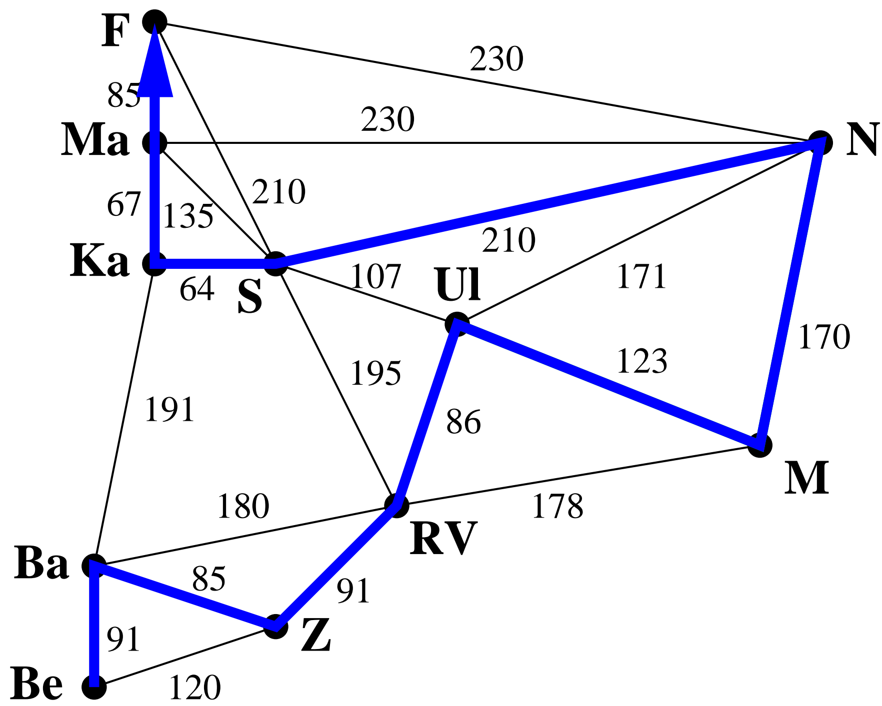


> **Definition 2.2** Ein **Preorder-Tree-Walk** traversiert einen Baum folgendermassen: Beginnend mit dem Wurzelknoten werden von links nach rechts rekursiv alle Nachfolgerknoten besucht.

Beispiel 2.3 Angewendet auf den minimal aufspannenden Baum mit Wurzelknoten *Bern* ergibt sich:



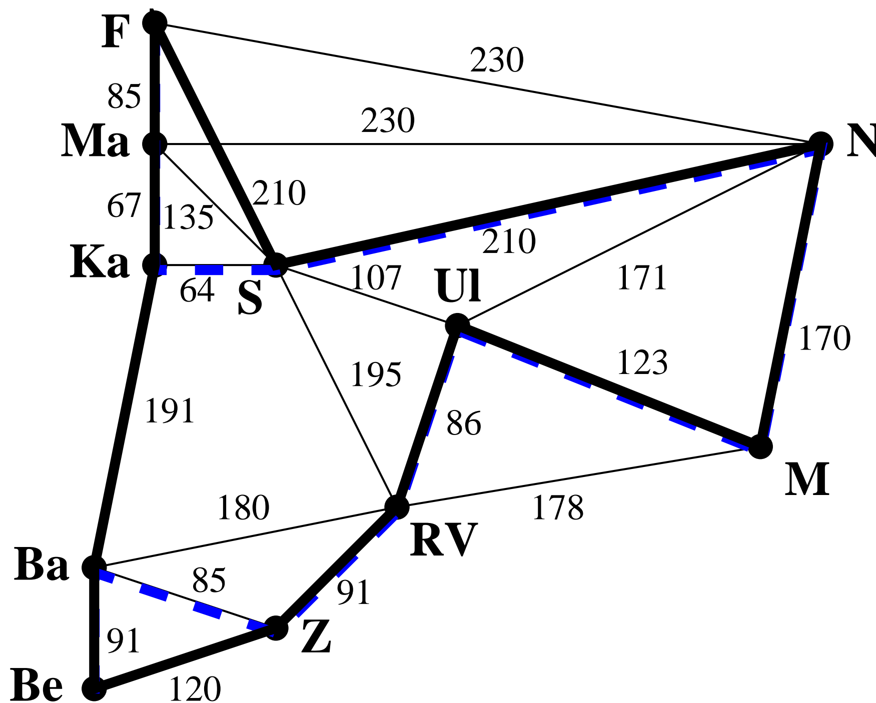
Listet man nun alle besuchten Städte in der besuchten Reihenfolge und löscht alle Wiederholungen so ergibt sich ein einfacher Pfad, der alle Knoten besucht.



Beispiel 2.4

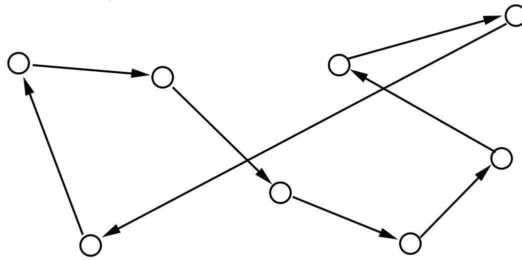
Im Fall eines voll vernetzten Graphen läßt sich dieser Pfad durch eine Verbindung von Frankfurt nach Bern zu einem Hamilton-Kreis schließen und man erhält eine Näherungslösung für das TSP-Problem. Dieses **heuristische** Vorgehen nennt man die **Minumum-Spanning-Tree-Heuristik**.
Ist der Graph nicht vollständig vernetzt, wie im Beispiel, so ist es durchaus möglich, dass es keinen Hamilton-Kreis gibt.

Beispiel 2.5 Der minimal aufspannende Baum liefert jedoch auch hier einen guten Ausgangspunkt für eine Näherung zur Lösung des TSP-Problems.



2.3 Das Problem des Handlungsreisenden

(traveling salesman problem, TSP)



gegeben: Menge von Städten $s_1, \ldots, s_n$ und eine Entfernungstabelle d mit $d_{ij} = |s_i - s_j| =$ Entfernung zwischen Stadt i und Stadt j.

gesucht: Eine Permutation (Vertauschung) Π von $(s_1, \ldots, s_n)$, so dass

$$\sum_{i=1}^{n-1} d_{\Pi(i),\Pi(i+1)} + d_{\Pi(n),\Pi(1)}$$

minimal ist. Es ist also eine Route gesucht, die jede Stadt genau einmal besucht und am Ende wieder im Ausgangspunkt endet.

Beispiel 2.6 Symmetrische Entfernungstabelle für vier Städte:

d	s_1	s_2	s_3	s_4
s_1	0	120	237	44
s_2	120	0	5	340
s_3	237	5	0	720
s_4	44	340	720	0

Definition 2.3 Eine bijektive Abbildung einer Menge M auf sich selbst heißt Permutation.

Beispiel 2.7 Eine Permutation der Menge $\{1, \ldots, 6\}$

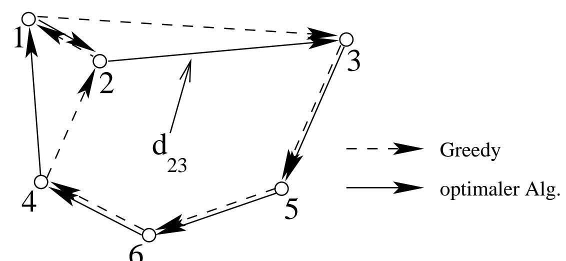
i	1	2	3	4	5	6
$\Pi(i)$	4	2	1	3	5	6

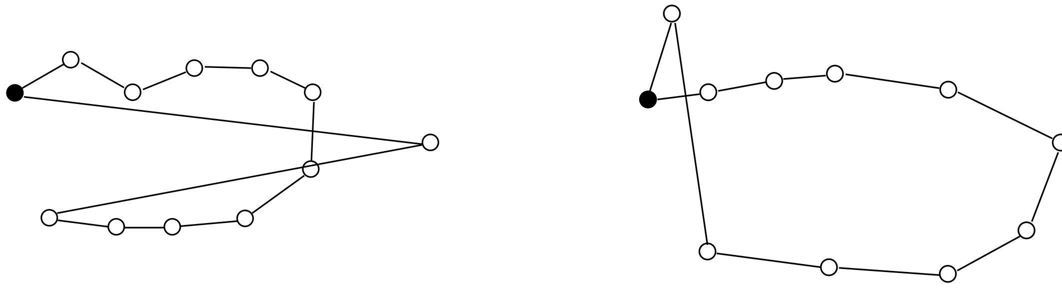
2.3.1 Der Greedy - Algorithmus

Idee

Starte mit Stadt Nr. 1 und wähle als nächste Stadt immer die mit dem geringsten Abstand. "Greedy" heißt auf deutsch "gierig". Dieses Attribut trifft auf den Algorithmus sehr gut zu, denn er greift gierig immer nach der "nächst besten" Stadt ohne auf das Gesamtresultat zu schauen.

Beispiel 2.8
Startet in nebenstehendem Beispiel der Greedy-Algorithmus zum Beispiel bei Stadt 4, so findet er die nicht optimale Tour $(4, 2, 1, 3, 5, 6, 4)$. Die Permutation Π für diese Tour ist in obigem Beispiel 2.7 angegeben. Ähnliches gilt für die beiden nacholgenden Beispiele, wenn der Greedy-Algorithmus etwa jeweils bei der markierten Stadt beginnt.





Algorithmus

```
Greedy_TSP(D)
   For i=1 To n Do
      tour[i]=i
   tour[n+1]=1
   For i=2 To n-1 Do
      next=argmin{D[i-1,j]|j ∈ tour[i...n]}
      vertausche tour[i] mit tour[next]
   Return(tour)
```

Komplexität

$$T(n) = \Theta(n^2)$$

Optimaler Algorithmus OPTIMAL_TSP

Idee

berechne für **jede** Tour $\sum_{i=1}^{n} d_{tour[i],tour[i+1]}$ und wähle eine Tour mit minimalem Wert.

Komplexität

ohne Beschränkung der Allgemeinheit ist die erste Stadt frei wählbar.

zweite Stadt: $n-1$ Möglichkeiten

dritte $n-2$ Möglichkeiten

...

n-te Stadt: eine Möglichkeit

$$T(n) = c \cdot (n-1)! = \Theta((n-1)!)$$

$$(n-1)! = (n-1)(n-2)\ldots 1 \leq (n-1)^{n-1}$$

$$n! \leq n^n = n \cdot n \ldots n \Longrightarrow T(n) = O(n^{n-1})$$

2.3.2 Stirling'sche Formel

$$
\begin{aligned}
n! &\approx (\frac{n}{e})^n \cdot \sqrt{2\Pi n} \quad \text{für} \quad n \to \infty \\
\Rightarrow \quad T(n) &= \Theta((\frac{n}{e})^{n-1} \cdot \sqrt{n})
\end{aligned}
$$

$\Longrightarrow$ überexponentielles Wachstum

Resultate

1.) Fachleute sind überzeugt, daß es keinen polynomiellen Alg. für TSP gibt!

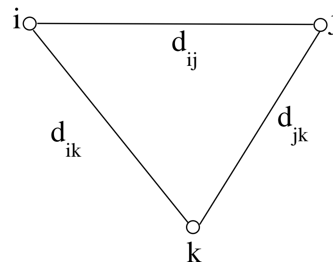
2.) und dass es sogar für fast optimale Lösungen keinen polynomiellen Alg. gibt.

3.) für TSP mit Dreiecksungleichung gibt es einen fast optimalen Alg. mit $T(n) = O(n^2)$

Dreiecksungleichung:

$$\forall i, j, k \quad d_{ij} \leq d_{ik} + d_{kj}$$

Diese Gleichung besagt, dass die direkte Verbindung zwischen zwei Städten immer minimale Länge hat. Sie gilt zum Beispiel für Luftlinienverbindungen zwischen Städten. Für die Entfernungen auf Straßen gilt sie jedoch nicht immer.



Wachstum der Rechenzeit von OPTIMAL_TSP

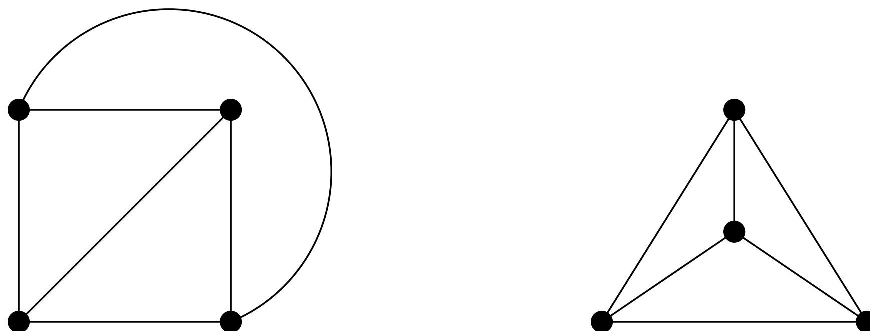
Folgende Tabelle veranschaulicht das extrem schnelle Wachstum der Rechenzeit von OPTIMAL_TSP. Heute ist das TSP-Problem mit optimierten Algorithmen für etwa 30 Städte lösbar. Wie man an der Tabelle erkennt, ist der Aufwand für 40 Städte etwa um den Faktor 10^{16} höher und damit also unerreichbar.

n	(n-1)!	(n-1)!
10	362880	$3.6 \cdot 10^5$
20	121645100408832000	$1.2 \cdot 10^{17}$
30	8841761993739701954543616000000	$8.8 \cdot 10^{30}$
40	20397882081197443358640281739902897356800000000	$2.0 \cdot 10^{46}$
50	608281864034267560872252163321295376887552831379210240000000000	$6.1 \cdot 10^{62}$

2.4 Planare Graphen

Definition 2.4 Läßt sich ein Graph in der Ebene darstellen, ohne dass sich Kanten kreuzen, so nennt man ihn **planar**. Die Menge der durch die Kanten begrenzten Flächenstücke wird mit A bezeichnet.

Beispiel 2.9 Zwei planare Darstellungen des vollständigen Graphen mit vier Knoten:



Für diesen Graphen gilt $|A| = 4$.

Satz 2.2 (Eulerformel) Für jeden planaren zusammenhängenden Graphen $G = (V, E)$ gilt

$$|V| - |E| + |A| = 2.$$

Beweis: Falls G ein Baum ist gilt $|V| = |E| + 1$ und $|A| = 1$ und es ergibt sich

$$|V| - |E| + |A| = |E| + 1 - |E| + 1 = 2.$$

Für den Fall, dass G kein Baum ist zeigen wir die Behauptung per Induktion über die Zahl der Kanten $n = |E|$. Für $n = 0$ besteht der Baum aus einem Knoten, d.h. es gilt $|V| = 1$, $|E| = 0$, $|A| = 1$ und die Formel stimmt.
Wir nehmen an, die Formel gilt für $n > 0$. Nun fügen wir eine Kante hinzu und erzeugen den neuen Graphen $G' = (V', E')$ mit $|V'| = |V|$ und $|E'| = |E| + 1$. Die neue Kante schließt einen Kreis und es gilt $|A'| = |A| + 1$, woraus

$$|V'| - |E'| + |A'| = |V| - (|E| + 1) + |A| + 1 = |V| - |E| + |A| = 2$$

folgt. $\square$

Kapitel 3

Formale Sprachen und Maschinenmodelle

3.1 Grundlagen

Man muss sich Sprachen vorstellen wie einen Legobaukasten. Die Buchstaben des Alphabets entsprechen elementaren Bausteinen und die Worte, beziehungsweise Sätze entsprechen gebauten Objekten. Solche Mengen lassen sich mehr oder weniger einfach beschreiben. Zum Beispiel die Menge der Objekte, die nur aus roten Steinen gebaut sind. Oder die Menge der Objekte bei denen auf einem Basisstein nur Steine oben draufgesetzt werden dürfen, aber nicht daneben. Was ist, wenn ich das fertige Objekt auf den Kopf stelle? Muß dann die Forderung immer noch erfüllt sein?
Um solche Unklarheiten auszuschließen, werden wir bei den Sprachen ganz formal vorgehen. Wir werden Spielgregeln in Form von Grammatiken zum Aufbau von Sprachen angeben. Mit diesen Spielregeln können dann nur noch Worte aus einer bestimmten Sprache erzeugt (abgeleitet) werden.
Hier stellen sich sofort einige für den Informatiker sehr wichtige und interessante Fragen:

- Läßt sich jede formale Sprache durch eine Grammatik beschreiben?
- Wenn ich eine Grammatik G habe, die eine Sprache L definiert, wie kann ich erkennen, ob ein Wort zu dieser Sprache gehört oder nicht?
- Etwas konkreter: Ist es möglich, für eine konkrete Programmiersprache L in endlicher Zeit zu entscheiden, ob ein vorgegebener Text ein Programm dieser Sprache darstellt oder nicht. Diese Aufgabe ist der Syntaxcheck des Compliers.
- Ist diese Entscheidung vielleicht sogar effizient möglich, das heißt, auch für große Programme in kurzer Zeit?
- Wenn ja, wie macht man das?
- Kann man vielleicht sogar automatisch Fehler in Programmen erkennen, wie zum Beispiel Endlosschleifen?
- Kann man überprüfen, ob ein Programm korrekt ist?

Die Beantwortung dieser Fragen ist Bestandteil des Gebiets der *formalen Sprachen und Automaten.* Um es vorweg zu nehmen, wir werden bis auf die erste und die letzten beiden Fragen teilweise oder ganz positive Antworten liefern.

Fangen wir bei den elementaren Bausteinen an.

|| **Definition 3.1** Ein Alphabet Σ ist eine endliche nicht leere Menge von Zeichen.

Sprachen sind noch einfacher als Lego-Baukästen. Es gibt genau vier Möglichkeiten, zwei Alphabetzeichen a und b miteinander zu verknüpfen, nämlich aa, ab, ba, oder bb. Diese Verknüpfung heißt Konkatenation und ist nicht vertauschbar. Damit kann man beliebig lange endliche Worte bauen, ähnlich wie bei den Legos.

Definition 3.2 Die Menge Σ^* aller **Worte** ist wie folgt rekursiv definiert.

- $\Sigma \subset \Sigma^*$ und auch das **leere Wort** ε ist in Σ^* enthalten.
- Für jedes Wort $w \in \Sigma^*$ und jedes Zeichen $x \in \Sigma$ ist auch $wx \in \Sigma^*$. wx ist die Zeichenkette, die entsteht, wenn man das Zeichen x an das Wort w anhängt.

Jede Teilmenge von Σ^* wird **Sprache** genannt.

Beispiel 3.1

$$\begin{aligned} \Sigma &= \{0,1\} \\ \Sigma^* &= \{0,1,\varepsilon,00,01,10,11,001,000,011,010,\ldots\} \end{aligned}$$

Beispiel 3.2

$$\begin{aligned} \Sigma &= \{+,-,\cdot,/,(,),0,1,2,3,4,5,6,7,8,9,x,y,a,b\} \\ \Sigma^* &= \{\ldots,)+-\cdot 567ax,\ldots\} \\ Terme &= \{x,y,a,b,(a),\ldots\} \\ Terme &\subset \Sigma^* \end{aligned}$$

Die Menge aller korrekten arithmetischen Terme ist eine kleine Teilmenge von Σ^*. Ein Affe, der zufällig auf einer entsprechenden Tastatur tippt würde viele Versuche benötigen, um einen korrekten Term zu erzeugen. Die Wahrscheinlichkeit für das Zustandekommen eines vorgegebenen Terms der Länge 40 wäre etwa $\frac{1}{2^{40}\cdot 10^{40}} \approx \frac{1}{10^{53}}$.

Definition 3.3 Sei $w \in \Sigma^*$ und $n \in \mathbb{N}_0$. Dann ist w^n das durch n-fache Wiederholung von w entstandene Wort. w^0 ist also das leere Wort ε.

Definition 3.4 Für eine endliche Zeichenmenge M ist M^* die Menge aller Zeichenketten die aus Elementen in M gebildet werden können. Das leere Wort gehört zu M^* dazu. Die Menge $M^+ = M^* \backslash \varepsilon$ enthält dagegen nur Worte mit mindestens einem Zeichen.

Beispiel 3.3 Sei $\Sigma = \{a,b,c\}$. Dann sind

$$\begin{aligned} &\emptyset, \\ &\{aa,ab,aaa\}, \\ &\{a^n | n \in \mathbb{N}_0\} = \{\varepsilon,a,aa,aaa,aaaa,\ldots\}, \\ &\{(ab)^n | n \in \mathbb{N}_0\} = \{\varepsilon,ab,abab,ababab,abababab,\ldots\}, \\ &\{a^n b^n | n \in \mathbb{N}_0\} = \{\varepsilon,ab,aabb,aaabbb,aaaabbbb,\ldots\} \end{aligned}$$

Teilmengen von Σ^* und somit Sprachen über dem Alphabet Σ.

Lemma 3.1 Für jedes endliche Alphabet A ist Σ^* abzählbar. Die Menge aller Sprachen über A ist überabzählbar.

Beweis: als Übung
Die interessanten Sprachen sind unendlich. Zum Beispiel sind alle Programmiersprachen unendlich, denn wir wollen nicht die Länge von Programmen beschränken.

3.2 Grammatiken

Besonders interessant sind strukturierte Sprachen. Eine Sammlung von zufällig erzeugten Wörtern ist für die meisten Anwendungen nicht sehr hilfreich. "Struktur" heißt hier, dass sich die Sprache endlich beschreiben läßt. Wir werden Grammatiken verwenden um Sprachen zu beschreiben.
Aus dem Sprachunterricht in der Schule ist die Grammatik der deutschen Sprache bekannt. Ein Satz der deutschen Sprache kann zum Beispiel bestehen aus <*Subjekt*> <*Prädikat*> <*Objekt*> und <*Subjekt*> wiederum kann ersetzt werden durch <*Artikel*><*Substantiv*>. Damit ist also *Die Studentin spielt Schach* ein wohlgeformter Satz entsprechend der einfachen angegebenen Grammatik. Jede Programmiersprache besitzt eine Grammatik.

Beispiel 3.4 Die (unendliche) Menge der arithmetischen Terme wie zum Beispiel $x \cdot (x + a \cdot (b - 12))$ läßt sich durch folgende Regelgrammatik charakterisieren:

$$\begin{array}{lcl}
<\mathit{Term}> & \rightarrow & <\mathit{Term}> + <\mathit{Term}> \\
<\mathit{Term}> & \rightarrow & <\mathit{Term}> - <\mathit{Term}> \\
<\mathit{Term}> & \rightarrow & <\mathit{Term}> \;/\; <\mathit{Term}> \\
<\mathit{Term}> & \rightarrow & <\mathit{Term}> \cdot <\mathit{Term}> \\
<\mathit{Term}> & \rightarrow & (<\mathit{Term}>) \\
<\mathit{Term}> & \rightarrow & <\mathit{Var}> \\
<\mathit{Term}> & \rightarrow & <\mathit{Konst}> \\
<\mathit{Var}> & \rightarrow & \text{x — y} \\
<\mathit{Konst}> & \rightarrow & \text{a — b — } <\mathit{Zahl}> \\
<\mathit{Zahl}> & \rightarrow & <\mathit{Zahl}><\mathit{Ziffer}> \text{ — } <\mathit{Ziffer}> \\
<\mathit{Ziffer}> & \rightarrow & \text{0—1—2—3—4—5—6—7—8—9}
\end{array}$$

Hier steht das Zeichen | für "oder", das heißt, eine Regel $S \rightarrow u|v$ steht für die zwei Regeln $S \rightarrow u$ und $S \rightarrow v$.

Definition 3.5 Eine **Grammatik** ist ein 4-Tupel

$$G = (V, \Sigma, P, S)$$

mit

- V als endliche nichtleere Menge der **Variablen**.
- Σ als Menge der **Konstanten** oder **Terminalalphabet** und $V \cap \Sigma = \emptyset$.
- $P \subset (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ als endliche Menge der **Produktionsregeln**.
- $S \in V$ ist die **Startvariable**.

Definition 3.6 Die in Beispiel 3.4 und im Folgenden verwendete Art der Darstellung von Grammatikregeln wird nach ihren Erfindern **Backus-Naur-Form** oder kurz **BNF** genannt.

Beispiel 3.5 Mit

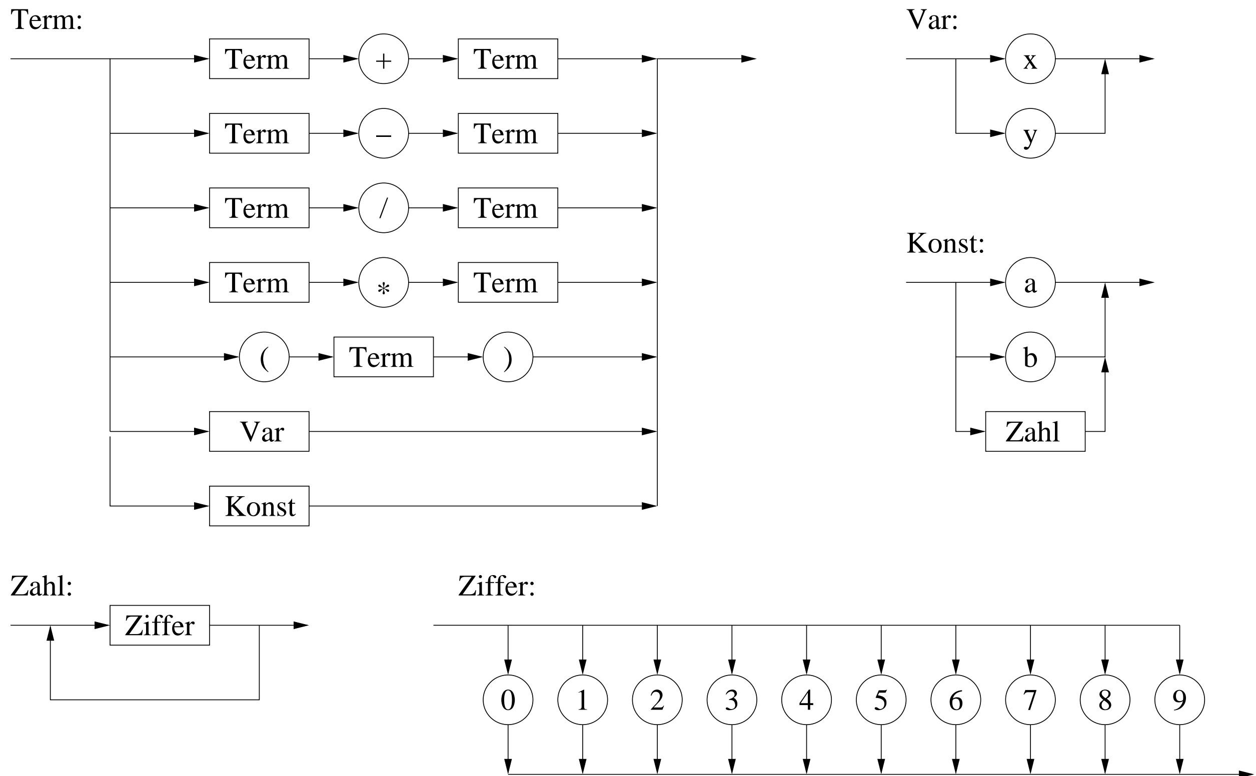
$$\begin{aligned} G \;=(\;& \{<\mathit{Term}>, <\mathit{Var}>, <\mathit{Konst}>, <\mathit{Zahl}>, <\mathit{Ziffer}>\}, \\ & \{x, y, a, b, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), +, -, \cdot, /\}, \\ & P, <\mathit{Term}>) \end{aligned}$$

und P als Menge der Regeln aus Beispiel 3.4 ergibt sich also eine Grammatik mit den angegebenen Variablen, Konstanten und $<\mathit{Term}>$ als Startsymbol.
Durch sukzessives Anwenden einer der Regeln aus P beginnend mit dem Startsymbol kann man den obigen Term $x \cdot (x + a \cdot (b - 12))$ **ableiten**:

$$\begin{aligned} <\mathit{Term}> &\Rightarrow <\mathit{Term}> \cdot <\mathit{Term}> \\ &\Rightarrow <\mathit{Var}> \cdot <\mathit{Term}> \\ &\Rightarrow x\cdot <\mathit{Term}> \\ &\Rightarrow x \cdot (<\mathit{Term}>) \\ &\Rightarrow x \cdot (<\mathit{Term}> + <\mathit{Term}>) \\ &\Rightarrow x \cdot (<\mathit{Var}> + <\mathit{Term}>) \\ &\Rightarrow x \cdot (x+ <\mathit{Term}>) \\ &\Rightarrow x \cdot (x+ <\mathit{Term}> \cdot <\mathit{Term}>) \\ &\Rightarrow x \cdot (x+ <\mathit{Konst}> \cdot <\mathit{Term}>) \\ &\Rightarrow x \cdot (x + a\cdot <\mathit{Term}>) \\ &\Rightarrow x \cdot (x + a \cdot (<\mathit{Term}>)) \\ &\Rightarrow x \cdot (x + a \cdot (<\mathit{Term}> - <\mathit{Term}>)) \\ &\Rightarrow x \cdot (x + a \cdot (<\mathit{Konst}> - <\mathit{Term}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b- <\mathit{Term}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b- <\mathit{Konst}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b- <\mathit{Zahl}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b- <\mathit{Zahl}><\mathit{Ziffer}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b- <\mathit{Ziffer}><\mathit{Ziffer}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b - 1 <\mathit{Ziffer}>)) \\ &\Rightarrow x \cdot (x + a \cdot (b - 12)) \end{aligned}$$

Eine äquivalente Darstellung von Regelgrammatiken in grafischer Form bieten die **Syntaxdiagramme**, die wir hier nicht formal einführen. Ein Beispiel soll genügen:

Beispiel 3.6 Syntaxdiagramm für Terme



Definition 3.7 Eine Folge von Wörtern $(w_0, w_1, \ldots, w_n)$ mit $w_0 = S$ und $w_n \in \Sigma^*$ heißt **Ableitung** von w_n, falls für $i \geq 1$ jedes der Wörter w_i aus w_{i-1} entstanden ist durch Anwendung einer Regel aus P auf ein Teilwort von w_{i-1}. Für einen Teilschritt schreibt man $w_{i-1} \Rightarrow w_i$. Ist ein Wort w durch einen oder mehrere Teilschritte aus u ableitbar, so schreibt man $u \Rightarrow^* w$.

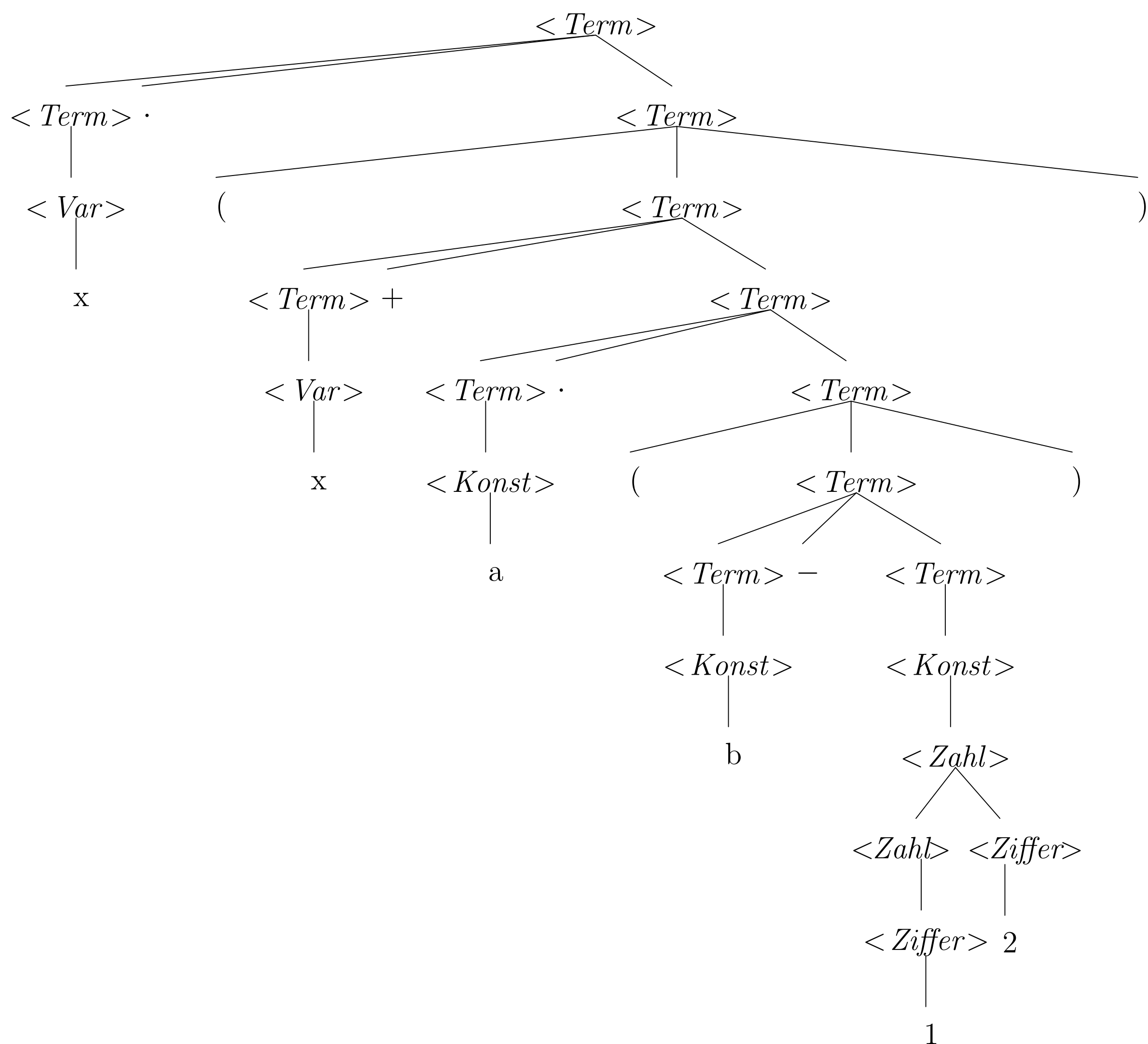
Obige Grammatik erzeugt die (unendliche) Menge der Terme als Teilmenge von Σ^*. Allgemein definiert man

Definition 3.8 Die durch G *erzeugte bzw. definierte Sprache* ist

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}.$$

Für kontextfreie Sprachen erhält man eine einfachere Darstellung der möglichen Ableitungen eines Wortes mit Hilfe von **Syntaxbäumen**.

Beispiel 3.7 Der Syntaxbaum zur Ableitung aus Beispiel 3.5 hat folgende Gestalt:



Definition 3.9 Eine Grammatik G heißt mehrdeutig, wenn es zu einem Wort $\omega \in L(G)$ mehrere verschiedene Syntaxbäume gibt. Sie heisst eindeutig, wenn es nur einen Syntaxbaum gibt.

Wir werden nun die Grammatiken formaler Sprachen einteilen in verschiedene Klassen, die sogenannte Chomsky-Hierarchie mit dem Ziel, zu verstehen, wie sich die Eigenschaften der zugehörigen Sprachen verändern, wenn, ausgehend von den einfachsten (regulären) Grammatiken, immer komplexere Regeln erlaubt sind.

3.3 Chomsky-Hierarchie

Definition 3.10 Jede Grammatik $G = (V, \Sigma, P, S)$ entsprechend Definition 3.5 ist vom **Typ 0**. Die Menge der Typ-0-Grammatiken ist also gleich der Menge aller Grammatiken.

Typ	Bezeichnung	erlaubte Regeltypen, $(w \in (V \cup \Sigma)^+,\ u \in (V \cup \Sigma)^*)$
0	Grammatik	$w \to u$.
1	kontextsensitiv	$w \to u$ mit $\|w\| \leq \|u\|$.
2	kontextfrei	$A \to u$, $A \to \varepsilon$, mit $A \in V$, d.h. auf der linken Seite aller Regeln kommt genau eine Variable vor.
3	regulär	$A \to a$, $A \to aB$, $A \to \varepsilon$, d.h. auf der rechten Seite der Regeln steht entweder ein Terminalsymbol oder ein Terminalsymbol gefolgt von einer Variablen.

Eine Sprache ist vom Typ t, wenn es eine Grammatik vom Typ t gibt mit $L(G) = L$.

Beispiel 3.8 Die Sprache aus Beispiel 3.4 ist offensichtlich eine kontextfreie Grammatik, das heißt sie ist vom Typ 2. Sie ist aber keine Typ-3-Grammatik. (Warum?)

Beispiel 3.9 Die Sprache $\{a^n b^n | n \in \mathbb{N}\}$ ist kontextfrei und läßt sich durch die Grammatik $G = (\{S\}, \{a, b\}, P, S)$ beschreiben mit

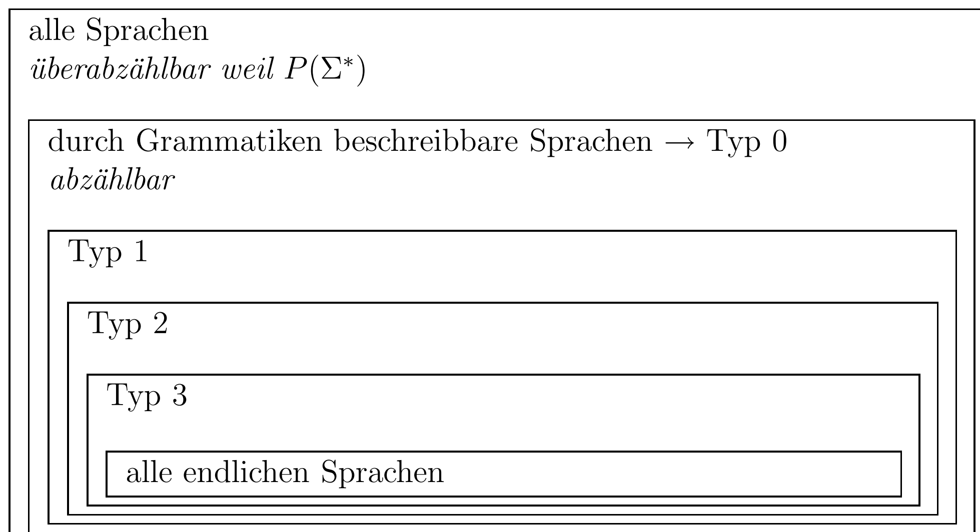
$$P = \{ \quad S \to aSb, \quad S \to ab \quad \}.$$

Diese Sprache ist nicht regulär.

Beispiel 3.10 Die Sprache $\{a^n b^m | n \in \mathbb{N}, m \in \mathbb{N}\}$ ist regulär und läßt sich durch die Grammatik $G = (\{S, T\}, \{a, b\}, P, S)$ beschreiben mit

$$P = \{ \quad S \to aS, \quad S \to aT, \quad T \to bT, \quad T \to b \quad \}.$$

Die Chomsky-Hierarchie der verschiedenen Sprachklassen ist in folgendem Mengendiagramm dargestellt:



Beispiel 3.11

$$\begin{aligned}\Sigma &= \{a, b\} \\ G_1 &= (\{S\}, \Sigma, P, S) \\ P &= \{S \to aS, S \to bS, S \to \varepsilon\}\end{aligned}$$

Offenbar läßt sich aus dieser Grammatik jedes Wort $w \in \Sigma^*$ ableiten, also gilt $L(G_1) = \Sigma^*$. Diese Aussage läßt sich wie folgt verallgemeinern.

Satz 3.1 Sei $\Sigma = \{c_1, \ldots, c_n\}$. Dann ist Σ^* eine Typ-3-Sprache und jede endliche Teilmenge von Σ^* ist vom Typ 3.

Beweis:

1. Teil:

$$\begin{aligned}\Sigma &= \{c_1, \ldots, c_n\} \\ G_1 &= (\{S\}, \Sigma, P, S) \\ P &= \{S \to c_1 S, S \to c_2 S, \ldots, S \to c_n S, S \to \varepsilon\}\end{aligned}$$

es folgt:

$$L(G_1) = \Sigma^*$$

Weil alle Regeln aus P Typ 3 - Regeln sind ist Σ^* vom Typ 3.

2. Teil:

Sei die endliche Sprache $L = \{w_1, w_2, \ldots, w_n\}$ gegeben. Die Grammatikregeln für das Wort $w_i = c_{i1} c_{i2} \ldots c_{ik}$ sind:

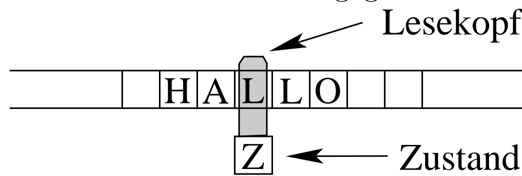
$$\begin{aligned}P_i &= \{S \to c_{i1} S_{i1}, S_{i1} \to c_{i2} S_{i2}, \ldots, S_{il_i} \to \varepsilon\} \\ G &= (V, \Sigma, P, S) \\ \Sigma &= \{c_{11}, \ldots, c_{1l_1}, c_{21}, \ldots, c_{2l_2}, \ldots, c_{n1}, \ldots, c_{nl_n}\} \\ V &= \{S, S_{11}, \ldots, S_{1l_1}, \ldots, S_{n1}, \ldots, S_{nl_n}\} \\ P &= \cup_{i=1}^{n} P_i\end{aligned}$$

3.4 Endliche Automaten

Nun kennen wir einige reguläre Sprachen und deren Regelgrammatik. Mit Hilfe der Grammatik lassen sich alle Worte der Sprache erzeugen. Nun stellen wir uns die Frage ob es eine möglichst einfache und effiziente Rechenmaschine gibt, mit der man für ein beliebiges Wort w entscheiden kann, ob dieses zu einer vorgegebenen regulären Sprache gehört.

Definition 3.11 Die Aufgabe, zu entscheiden, ob ein Wort w zu einer Sprache L gehört, heißt **Wortproblem**.

Das Wortproblem für reguläre Sprachen kann durch **endliche Automaten** effizient gelöst werden. Anschaulich ist ein endlicher Automat ein Rechenelement, welches auf einem Eingabeband beginnend mit dem ersten Zeichen das eingegebene Wort Zeichen für Zeichen liest.



Hierbei kann er seinen internen Zustand entsprechend von Regeln abhängig vom Eingabezeichen wechseln. Die Zahl der Zustände ist endlich. Erreicht der Automat nach Lesen des letzten Zeichens einen Endzustand, so hat er das Wort erkannt. Formal wird der Automat wie folgt definiert:

Definition 3.12 Ein endlicher Automat M besteht aus einem 5-, bzw. 7-Tupel

$$M = (Z, \Sigma, \delta, z_0, E)$$

bzw.

$$M = (Z, \Sigma, \delta, z_0, E, \gamma, \Theta)$$

mit

$$\begin{aligned} Z &: \text{ endliche Zustandsmenge} \\ \Sigma &: \text{ endliches Eingabealphabet, } \Sigma \cap Z = \phi \\ \delta &: \ Z \times \Sigma \to \mathcal{P}(Z), \text{ die Zustandsübergangsfunktion} \\ z_0 &: \text{ Startzustand} \\ E &: \text{ Menge der Endzustände} \\ \gamma &: \ Z \times \Sigma \to \Theta, \text{ die Ausgabefunktion} \\ \Theta &: \text{ Ausgabealphabet} \end{aligned}$$

Definition 3.13 Ein Wort $w = w_1 \dots w_n$ mit $w_i \in \Sigma$ wird **akzeptiert** von dem endlichen Automaten M genau dann wenn M gestartet im Startzustand auf w_1 nach n Anwendungen der Funktion δ, d.h. nach Lesen von w_n, einen Endzustand $z \in \Sigma$ erreichen kann.
Die von M **akzeptierte** (erkannte) Sprache $L(M)$ ist

$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$$

Satz 3.2 Eine Sprache L wird von einem endlichen Automaten genau dann erkannt, wenn sie regulär (Typ 3) ist.

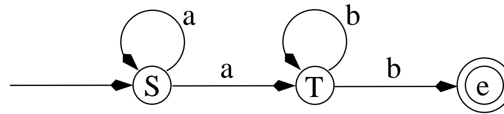
Beispiel 3.12 Die reguläre Sprache $L = \{a^n b^m \mid n \in \mathbb{N}, m \in \mathbb{N}\}$ wird erzeugt durch die Regelmenge

$$P = \{S \to aS, S \to aT, T \to bT, T \to b\}$$

Die Zustandsübergangsfunktion δ des zugehörigen Automaten $M = (\{S, T, e\}, \{a, b\}, \delta, S, \{e\})$ ist gegeben durch die **Zustandsübergangstabelle**

δ	S	T	e
a	$\{S, T\}$		
b		$\{T, e\}$	

Man beachte, dass die Zustandsübergangsfunktion δ nicht eindeutig ist, denn zum Beispiel kann der Automat nach Lesen eines a im Zustand S nach S oder nach T übergehen. Dies zeichnet den nichtdeterministischen Automaten aus.
Der zugehörige Zustandsgraph ist



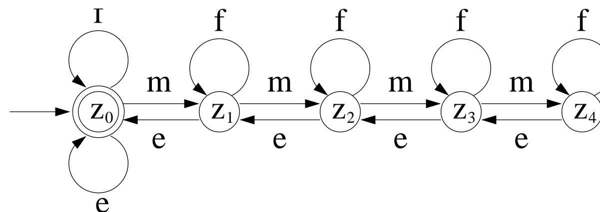
Beispiel 3.13 Es soll ein Getränkeautomat mit Hilfe eines endlichen Automaten programmiert werden. Der Automat kann mit bis zu 4 Dosen Mineralwasser gefüllt werden. Wenn eine 1-Euro-Münze eingegeben wird, soll er eine Dose Wasser ausgeben. Bei Eingabe einer anderen Münze soll er die eingegebene Münze wieder ausgeben, aber kein Getränk. Wenn der Automat leer ist soll er anhalten und per Funk den Service benachrichtigen.
Ein endlicher Automaten (mit Ausgabe) für diese Aufgabe ist

$$(\{z_0, z_1, z_2, z_3, z_4\}, \{e, f, m\}, \delta, z_0, \{z_0\}, \gamma, \{e, f, m\})$$

wobei δ und γ gegeben sind durch

δ, γ	z_0	z_1	z_2	z_3	z_4
m	z_1, ε	z_2, ε	z_3, ε	z_4, ε	
e	z_0, e	z_0, m	z_1, m	z_2, m	z_3, m
f	z_0, f	z_1, f	z_2, f	z_3, f	z_4, f

Das Zustandsdiagramm zu diesem Automaten sieht so aus:



Dieser Automat akzeptiert alle Eingabesequenzen (Worte), die zum leeren Automaten (d.h. zu z_0) führen.

Definition 3.14 Beim nichtdeterministischen endlichen Automaten (NFA) sind (im Gegensatz zum deterministischen endlichen Automaten (DFA)) für jeden Zustand Z und Eingabezeichen a mehrere Regeln

$$z, a \rightarrow z_1$$
$$\vdots$$
$$z, a \rightarrow z_n$$

erlaubt.

Bemerkung:
Aus der Zustandsübergangsfunktion δ wird eine Relation.

Beispiel 3.14 An der Sprache $L = \{a^n b^m | n \in \mathbb{N}, m \in \mathbb{N}\}$ erkennt man schön, wie die Regelgrammatik in einfacher Weise in einen nichtdeterministischen Automaten übersetzt werden kann:

Regelgrammatik	Automat
$P = \{S \rightarrow aS$	$\delta = \{S, a \rightarrow S$
$S \rightarrow aT$	$S, a \rightarrow T$
$T \rightarrow bT$	$T, b \rightarrow T$
$T \rightarrow b\,\}$	$T, b \rightarrow E\,\}$

Hier stellt sich die Frage, ob es vielleicht auch einen deterministischen Automaten gibt, der diese Sprache erkennt. Allgemein lautet die Frage: Sind nichtdeterministische Automaten mächtiger ist als deterministische. Der folgende Satz beantwortet beide Fragen.

Satz 3.3 NFAs und DFAs sind gleich mächtig, d.h. zu jedem NFA gibt es einen DFA, der die gleiche Sprache erkennt.

3.5 Reguläre Ausdrücke

Reguläre Ausdrücke dienen wie reguläre Grammatiken der Beschreibung von Typ-3-Sprachen.

Definition 3.15 Reguläre Ausdrücke zum Alphabet Σ sind:

1.) $\emptyset = \{\}$

2.) ε

3.) a, falls $a \in \Sigma$

4.) sind α, β reguläre Ausdrücke, so auch $\alpha\beta$, $\alpha|\beta$ und α^*

hierbei steht $\alpha|\beta$ für α oder β, α^* für beliebig viele Wiederholungen von α (auch 0 Wiederholungen).

Beispiel 3.15 $aa{*}bb{*}$ beschreibt $\{a^n b^m / n \in \mathbb{N}, m \in \mathbb{N}\}$. Für $aa*$ schreibt man kürzer $a+$. Der Operator + steht also für beliebig viele Wiederholungen, aber mindestens eine.

Beispiel 3.16 Sei $\Sigma = \{., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

$$0.(0|1|2|3|4|5|6|7|8|9)+$$

beschreibt die Menge aller Dezimalzahlen mit 0 vor dem Dezimalpunkt.

Da wir im Folgenden das Werkzeug Lex bzw. Flex vorstellen werden, wird die Definition der regulären Ausdrücke gekürzt entnommen aus der Linux-Manual-Page zu FLex. Die Rangfolge der Operatoren entspricht deren Priorität.

```
x           match the character ’x’
.           any character (byte) except newline
[xyz]       a "character class"; in this case, the pattern
              matches either an ’x’, a ’y’, or a ’z’
[abj-oZ]    a "character class" with a range in it; matches
              an ’a’, a ’b’, any letter from ’j’ through ’o’,
              or a ’Z’
[^A-Z]      a "negated character class", i.e., any character
              but those in the class.  In this case, any
              character EXCEPT an uppercase letter.
[^A-Z\n]    any character EXCEPT an uppercase letter or
              a newline
r*          zero or more r’s, where r is any regular expression
r+          one or more r’s
r?          zero or one r’s (that is, "an optional r")
r{2,5}      anywhere from two to five r’s
r{2,}       two or more r’s
r{4}        exactly 4 r’s
{name}      the expansion of the "name" definition
            (see above)
"[xyz]\"foo"
            the literal string: [xyz]"foo
\X          if X is an ’a’, ’b’, ’f’, ’n’, ’r’, ’t’, or ’v’,
              then the ANSI-C interpretation of \x.
              Otherwise, a literal ’X’ (used to escape
              operators such as ’*’)
\0          a NUL character (ASCII code 0)
\123        the character with octal value 123
\x2a        the character with hexadecimal value 2a
(r)         match an r; parentheses are used to override
              precedence (see below)
rs          the regular expression r followed by the
              regular expression s; called "concatenation"
r|s         either an r or an s
^r          an r, but only at the beginning of a line (i.e.,
              which just starting to scan, or right after a
              newline has been scanned).
r$          an r, but only at the end of a line (i.e., just
              before a newline).
```

Mit dieser erweiterten Notation für reguläre Ausdrücke lassen sich Dezimalzahlen (siehe Beispiel 3.16) einfacher beschreiben mit

$$0.[0123456789]+$$

oder noch einfacher durch

$$0.[0-9]+$$

Bevor wir mit der praktischen Anwendung von regulären Ausdrücken fortfahren noch ein wichiger Satz.

Satz 3.4 Jede reguläre Sprache ist durch einen regulären Ausdruck beschreibbar. Umgekehrt definiert jeder reguläre Ausdruck eine reguläre Sprache.

3.6 Der Lexical Analyzer LEX

LEX, bzw. FLEX ist ein sogenannter Lexical Analyzer. Er kann dazu verwendet werden, für reguläre Sprachen das Wortproblem zu lösen, das heisst, zu entscheiden, ob ein vorgegebenes Wort zu einer Sprache L gehört. LEX kann daneben sogar noch erkannte Wörter ersetzen entsprechend definierten Regeln.
Das Ersetzen von Ausdrücken der Form

```
Tel.: 0751/501-721
```

durch Ausdrücke der Form

```
Phone: ++49-751-501-721
```

kann durch folgendes LEX-Programm erfolgen.

Die Datei `tel.x`:

```
%option noyywrap
%%
Tel\.?:[ \t]+0    printf("Phone: ++49-");
[0-9](\/|-)[0-9]  printf("%c-%c", yytext[0],yytext[2]);
\n                printf("\n");
```

Anwendung des fertigen Programms `tel` liefert:

```
> tel
Tel.: 0751/501-9721
Phone: ++49-751-501-9721
Tel.: 075qt1/501-9721
Phone: ++49-75qt1-501-9721
quit
quit
^D
```

Aufruf von LEX mit Quelldatei `tel.x`:

```
flex -otel.c tel.x
cc -lfl tel.c -o tel
tel
```

der generelle Aufbau eines LEX-Programms (und auch eines YACC-Programms) ist:

Definitionen
%%
Regeln
%%
Funktionen

3.7 YACC: Yet Another Compiler Compiler

YACC ist ein Programmgenerator, der aus einer kontextfreien Grammatik ein Programm generiert, das die Korrektheit der eingegebenen Worte prüft, das heisst das Wortproblem entscheidet. YACC wird hauptsächlich dazu verwendet, Parser für Programmiersprachen automatisch zu erzeugen. Er kann also nicht nur die Syntax von Programmiersprachen checken, sondern auch Code generieren. Dies führt jedoch hier zu weit.

3.7.1 Ein Beispiel mit LEX und YACC

Die Datei term.x

```
#include "y.tab.h"
%%
\(                return(yytext[0]);
\)                return(yytext[0]);
[\+\-\*\/]        return(OP);
"[ \t]+"          ;
[a-zA-Z][a-zA-Z0-9_]* return(BEZ);
[0-9]+            return(INTEGER);
[0-9]+\.[0-9]+     return(GLEITPKTZ);
\n                return('\0');
```

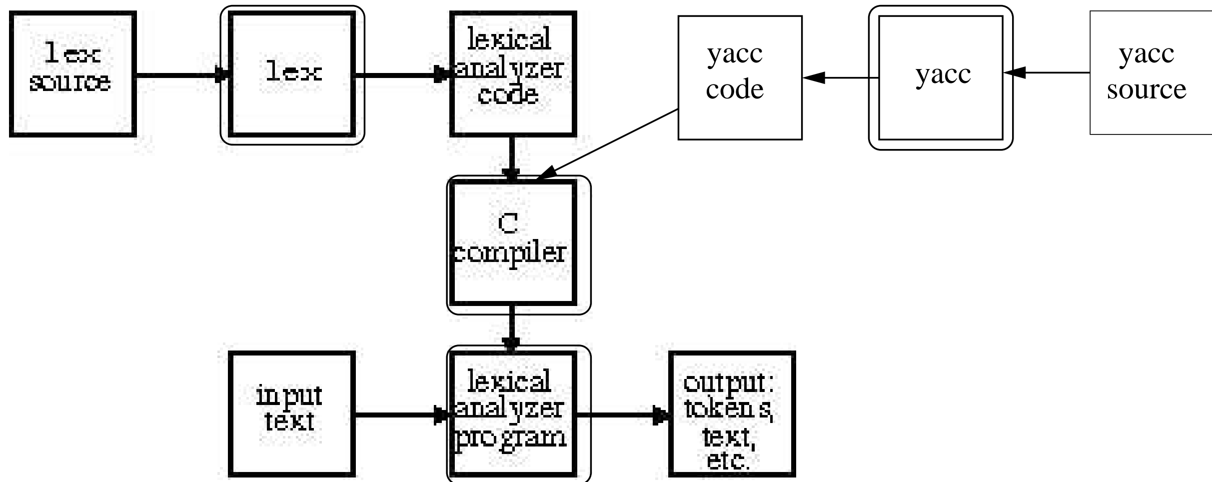
Die BISON-Eingabedatei term.y

```
%token BEZ OP INTEGER GLEITPKTZ
%%
term    : INTEGER
          | GLEITPKTZ
          | BEZ
          | term OP term
          | '(' term ')'
          | error {printf("Term nicht wohlgeformt!\n");}

%%
yyerror (s)  /* Called by yyparse on error */
 char *s;
{
  printf ("%s\n", s);
}

#include "lex.yy.c"

main()
{
  /*  yydebug = 1;*/
  yyparse();
}
```

Die Datei makefile

```
term: lex.yy.c term.tab.c
        cc term.tab.c -lfl -o term

term.tab.c: term.y
        bison term.y

lex.yy.c: term.x
        flex term.x
```

Ablauf und Zusammenspiel von LEX, YACC und CC:



3.8 Kellerautomaten

Wir starten mit einem Beispiel an dem man erkennt, dass schon recht einfache Sprachen von einem endlichen Automaten nicht erkannt werden können.

Beispiel 3.17

$$L = \{a^n b^n | n \varepsilon \mathbb{N}\}$$

Grammatik für L:

$$P = \{s \to aSb, S \to ab\}$$

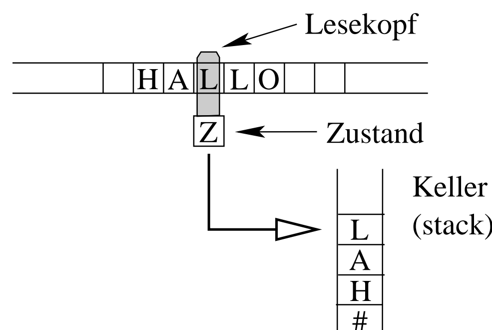
L ist eine Typ-2-Sprache. Daher gibt es keine reguläre Grammatik für L und auch keinen endlichen Automaten, der L erkennt.

Beispiel 3.18 Beschränkt man allerdings die Zahl n der a-s und a-s, so gibt es einen endlichen Automaten, der die Sprache erkennt. Sei also

$$L' = \{a^n b^n | n = 1, \ldots, 100\}.$$

Diese Sprache wird erkannt von einem Automaten mit Endzustand E und folgenden Zustandsübergängen

$$\begin{array}{lll} A_0, a \to A_1 & & B_0, b \to E \\ A_1, a \to A_2 & A_1, b \to B_1 & B_1, b \to B_0 \\ A_2, a \to A_3 & A_2, b \to B_2 & B_2, b \to B_1 \\ \ldots & \ldots & \ldots \\ A_{98}, a \to A_{99} & A_{99}, b \to B_{99} & B_{99}, b \to B_{98} \end{array}$$



Definition 3.16 Ein Kellerautomat K besteht aus einem 6-Tupel

$$K = (Z, \Sigma, \Gamma, \delta, z_0, \#)$$

mit

$$\begin{array}{rcl} Z & : & \text{endliche Zustandsmenge} \\ \Sigma & : & \text{endliches Eingabealphabet, } \Sigma \cap Z = \emptyset \\ \Gamma & : & \text{endliches Kelleralphabet, } \Sigma \cap Z = \emptyset \\ \delta & : & Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_e(Z \times \Gamma^*) \text{, die Zustandsübergangsfunktion}^a \\ z_0 \in Z & : & \text{Startzustand} \\ \# \in \Gamma & : & \text{unterstes Kellerzeichen} \end{array}$$

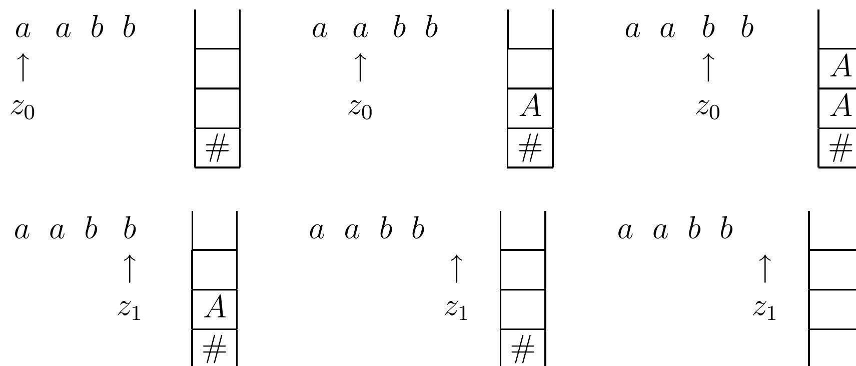
[a] $\mathcal{P}_e$ steht für die Menge aller endlichen Teilmengen.

Beispiel 3.19 Kellerautomat K für $L = \{a^n b^n | n \in \mathbb{N}\}$

$$K = \{\{z_0, z_1\}, \{a, b\}, \{A, B, \#\}, \delta, z_0, \#\}$$

$$\begin{array}{rcl} \delta = \{z_0, a, \# & \to & z_0, A\#\} \\ z_0, a, A & \to & z_0, AA \\ z_0, b, A & \to & z_1, \varepsilon \\ z_1, b, A & \to & z_1, \varepsilon \\ z_1, \varepsilon, \# & \to & z_1, \varepsilon \end{array}$$

Folgende Sequenz von Konfigurationen veranschaulicht die Arbeit von K:



$$L = \{a^n b^n c^m | n \in \mathbb{N}_0,\ m \in \mathbb{N}_0\}$$
$$\delta' = \delta \cup \{z_1, c, \# \to z_1, \#\}$$

Beispiel 3.20 Palindrome sind Worte, die in der Mitte gespiegelt sind. Wir betrachten nun die Sprache aller Palindrome über dem Alphabet $\{a, b\}$, wobei die Mitte des Wortes jeweils durch ein \$-Zeichen markiert ist. Sei also

$$\Sigma = \{a, b, \$\} \quad \text{und} \quad L = \{a_1 \ldots a_n \$ a_n \ldots a_1 | a_i \in \{a, b\}, n \in \mathbb{N}_0\}$$

Die Sprache wird erzeugt durch die Typ-2-Grammatik $G = (V, \Sigma, P, S)$ mit

$$P = \{S \to \$|aSa|bSb\}$$

L ist keine Typ 3 Sprache, denn wie oben gezeigt gibt es keinen endlichen Automaten zu dieser Sprache. Folgender **deterministischer Kellerautomat** erkennt L:

Eingabezeichen Kellerzeichen	s_0	s_1
$a, \#$	$s_0, A\#$	
$b, \#$	$s_0, B\#$	
$\$, \#$	$s_1, \#$	
a, A	s_0, AA	s_1, ε
a, B	s_0, AB	
b, A	s_0, BA	
b, B	s_0, BB	s_1, ε
$\$, A$	s_1, A	
$\$, B$	s_1, B	
$\varepsilon, \#$		s_1, ε

Beispiel 3.21 Lassen wir die Mittenmarkierung weg, so ergibt sich

$$\Sigma = \{a, b\} \quad \text{und} \quad L = \{a_1 \ldots a_n a_n \ldots a_1 | a_i \in \Sigma, n \in \mathbb{N}_0\}$$

mit der Grammatik

$$P = \{S \to aSa|bSb|\varepsilon\}.$$

Hier kann man nun die Mitte des Wortes nicht mehr in einem deterministischen Durchlauf erkennen. Daher ist ein **nichtdeterministischer Kellerautomat** gefordert:

Eingabezeichen, Kellerzeichen	s_0	s_1
$a, \#$	$s_0, A\#; s_1, A\#$	
$b, \#$	$s_0, B\#; s_1, B\#$	
a, A	$s_0, AA; s_1, AA$	s_1, ε
a, B	$s_0 AB; s_1 AB$	
b, A	$s_0, BA; s_1 BA$	
b, B	$s_0, BB; s_1, BB$	s_1, ε
$\varepsilon, \#$	s_1, ε	s_1, ε

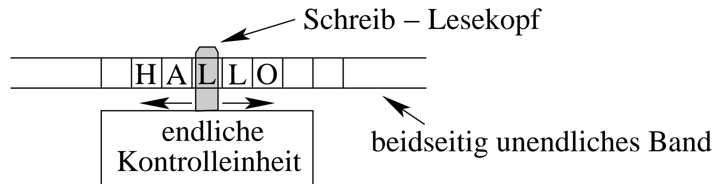
Die Semantik des Erkennens eines Wortes durch einen deterministischen Kellerautomaten definieren wir wie folgt:

Definition 3.17 Ein (nichtdeterministischer) Kellerautomat K erkennt, bzw. akzeptiert ein Wort $w = w_1 \ldots w_n$ genau dann, wenn es eine Folge von Zustandsübergängen gibt, so dass nach Lesen von w_n der Keller ganz leer ist. Hierbei muß K gestartet werden auf w_1.

Bemerkung: An diesem Beispiel erkennt man, dass nichtdeterministische Kellerautomaten mächtiger sind als deterministische, d.h. es gibt Sprachen, die von nichtdeterministischen Kellerautomaten erkannt werden, aber nicht von deterministischen.

3.9 Turingmaschinen

Eine der vielen großen Erfindungen des genialen britischen Mathematikers Alan Turing ($\approx$ 1940) war die Definition eines Modells für eine universelle Rechenmaschine, welche alle Funktionen berechnen kann, die wir uns intuitiv als berechenbar vorstellen. Die sogenannte Turingmaschine besitzt endliche viele Zustände und arbeitet auf einem beidseitig unendlichen Band. Ihre Mächtigkeit erhält sie durch die Möglichkeit Zeichen auf dem Band zu überschreiben sowie den Schreib-Lesekopf nach rechts oder links zu bewegen.



Beispiel 3.22 Turingmaschine T, die eine Eingabe $x \in \{0,1\}^*$ als Binärzahl interpretiert und 1 hinzuaddiert. Ein leeres Bandfeld bezeichnen wir im Folgenden mit $\sqcup$.

$$M = (\{z_0, z_1, z_2, z_e\}, \{0,1\}, \{0,1,\sqcup\}, \delta, z_0, \sqcup, \{z_e\})$$

$$\begin{array}{rcl}
z_0, 0 & \rightarrow & z_0, 0, R \\
z_0, 1 & \rightarrow & z_0, 1, R \\
z_0, \sqcup & \rightarrow & z_1, \sqcup, L \\
z_1, 0 & \rightarrow & z_2, 1, L \\
z_1, 1 & \rightarrow & z_1, 0, L \\
z_1, \sqcup & \rightarrow & z_e, 1, N \\
z_2, 0 & \rightarrow & z_2, 0, L \\
z_2, 1 & \rightarrow & z_2, 1, L \\
z_2, \sqcup & \rightarrow & z_e, \sqcup, R
\end{array}$$

Die Maschine bewegt sich zuerst nach rechts bis zum Ende der Binärzahl. Dann erfolgt die eigentliche Addition im Zustand z_1 mit Bewegung nach links. Bei der ersten null ist die Addition abgeschlossen und im Zustand z_2 läuft T zum Anfang der Zahl, wo sie terminiert.

Definition 3.18 Ein Turingmaschine besteht aus einem 7-Tupel

$$T = (Z, \Sigma, \Gamma, \delta, z_0, \sqcup, E)$$

mit

Z	:	endliche Zustandsmenge
Σ	:	endliches Eingabealphabet, $\Sigma \cap Z = \emptyset$
Γ	:	endliches Arbeitsalphabet, mit $\Sigma \subset \Gamma$
δ	:	$Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$, die Zustandsübergangsfunktion bei deterministischen Turingmaschinen
δ	:	$Z \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$, die Zustandsübergangsfunktion bei nichtdeterministischen Turingmaschinen
z_0	:	Startzustand, $z_0 \in Z$
$\sqcup$	:	Das Blank (Leerzeichen), wobei $\sqcup \in \Gamma - \Sigma$
E	:	Menge der Endzustände mit $E \subseteq Z$

Definition 3.19 Ein Wort $w = w_1 \dots w_n$ wird von einer Turingmaschine T akzeptiert, wenn sie, gestartet auf w_1 in einem Endzustand hält.

$$L(T) = \{w \,\varepsilon\, \Sigma^* | M \text{ akzeptiert } w\}$$

Satz 3.5 Turingmaschinen akzeptieren genau die Typ-0-Sprachen.

Man könnte aufgrund dieses Satzes verleitet sein, zu glauben, dass Turingmaschinen das Wortproblem (Definition 3.11) lösen. Dies ist aber falsch. Man vergleiche hierzu zum Beispiel den kleinen aber subtilen Unterschied in den Definitionen 3.17 und 3.19. Der Kellerautomat akzeptiert ein Wort "genau dann wenn ...", die Turingmaschine hingegen akzeptiert ein Wort "wenn ...". Über den Fall, dass die Turingmaschine nicht in einem Endzustand hält, macht die Definition keine Aussage. Warum?

Beispiel 3.23 Besonders einfach sind Turingmaschinen, die unendlich viele 1-en schreiben:

$$z_0, \sqcup \to z_0, 1, R$$

Viel schwieriger ist es, möglichst viele, aber endlich viele Einsen zu schreiben.

3.9.1 Fleißige Biber

folgende Ausführungen sind teilweise entnommen aus http://www.wuerzburg.de/gym-fkg/SCHULE/FACH
Die Turingmaschine ist natürlich hoffnungslos ineffektiv bei der Bearbeitung von konkreten Problemen. Sie wird meist nur als theoretisches Konzept verwendet. Dabei hat sie allerdings große Bedeutung. So kann mit ihrer Hilfe zum Beispiel das zentrale Problem der Informatik, das Halteproblem (leider negativ) beantwortet werden.

So dachte sich der ungarische Mathematiker Tibor Rado 1962 das busy-beaver-Problem aus. Die gestellte Frage lautet:

Definition 3.20 Busy- Beaver- Problem:
Gesucht ist eine Turingmaschine mit dem Arbeitsalphabet $\{1, \sqcup\}$ und einer vorgegebenen Anzahl von Zuständen. Das Turingband ist leer. Wie viele Zeichen kann sie maximal schreiben?

Bemerkung: Es ist kein Problem, eine Turingmaschine zu entwerfen, die unendliche viele Zeichen schreibt (s.o.). Aber die Turingmaschine soll ja **irgendwann anhalten**. Das macht das Problem so schwierig. Bei der Anzahl der Zustände der fleißigen Biber werden die Endzustände nicht mitgezählt.

Beispiel 3.24 Dieser Busy Beaver mit 2 Zuständen schreibt 4 Einsen:

$$\begin{aligned} z_0, \sqcup &\rightarrow z_1, 1, R \\ z_0, 1 &\rightarrow z_1, 1, L \\ z_1, \sqcup &\rightarrow z_0, 1, L \\ z_1, 1 &\rightarrow z_e, 1, R \end{aligned}$$

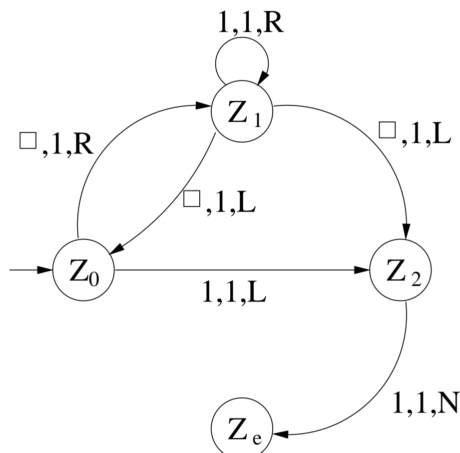
Beispiel 3.25 Busy Beaver mit 3 Zuständen, der 6 Einsen schreibt:

	$\sqcup$	1
z_0	$z_1, 1, R$	$z_2, 1, L$
z_1	$z_2, 1, R$	$z_e, 1, N$
z_2	$z_0, 1, L$	$z_1, \sqcup, L$

Beispiel 3.26 Noch ein fleißiger Biber mit 3 Zuständen:

$$\begin{aligned} z_0, \sqcup &\rightarrow z_1, 1, R \\ z_0, 1 &\rightarrow z_2, 1, L \\ z_1, \sqcup &\rightarrow z_0, 1, L \\ z_1, 1 &\rightarrow z_1, 1, R \\ z_2, \sqcup &\rightarrow z_1, 1, L \\ z_2, 1 &\rightarrow z_e, 1, N \end{aligned}$$

Der gleiche Biber als Automat dargestellt:



Man kann zeigen, dass eine Turingmaschine mit einem Zustand maximal ein Zeichen schreiben kann, eine mit zwei Zuständen maximal vier Zeichen, eine mit drei Zuständen maximal sechs Zeichen, eine mit vier Zuständen maximal dreizehn Zeichen.
Folgende Tabelle aus [6] listet einige aktuelle bekannte Ergebnisse über fleißige Biber auf. Hier ist n die Zahl der Zustände (ohne Endzustand), $\Sigma(n)$ die maximale Zahl geschriebener Einsen und $S(n)$ die maximale Zahl von Rechenschritten solch einer Maschine.

n	$\Sigma(n)$	$S(n)$	Quelle
1	1	1	Lin und Rado
2	4	6	Lin und Rado
3	6	21	Lin und Rado
4	13	107	Brady
5	≥ 4098	$\geq 47,176,870$	Marxen und Buntrock
6	$> 1.29 \cdot 10^{865}$	$> 3 \cdot 10^{1730}$	Marxen und Buntrock

Interessant ist offenbar auch folgende Frage: Welche Turingmaschine mit n Zuständen - ohne Endzustand - macht möglichst viele Arbeitschritte, stoppt dann und hinterlässt ein leeres Band?
Die Funktion (n), die angibt, wie gross die maximale Zahl von Zeichen ist, die eine Turingmaschine mit n Zuständen (ohne Endzustand) ausgeben kann, ist zwar wohldefiniert, aber nicht durch eine Turingmaschine und somit überhaupt nicht berechenbar!

3.10 Zusammenfassung zu Sprachen und Maschinenmodellen

Vergleich von Sprachtypen und Maschinenmodellen:

Chomsky-Typ	Beschreibung	Maschinen-Modell	Komplexität d. Wortproblems
0	Regelgrammatiken	Turingmaschine	unlösbar / halbentscheidbar
1	kontextsensitive Grammatik	linear beschränkter Automat (TM)	$O(a^n)$ (exponentiell)
2	kontextfreie Grammatik	Kellerautomat (nichtdeterminist.)	$O(n^3)$
3	reguläre Grammatiken / reguläre Ausdrücke	endlicher Automat	$\Theta(n)$

Satz 3.6 Church'sche These: Die Menge, der durch Turingmaschine berechenbaren Funktionen entspricht genau der Menge aller intuitiv berechenbaren Funktionen.

Die Church'sche These ist kein Satz im strengen Sinne, denn der Begriff intuitiv widersetzt sich einem Beweis.

Satz 3.7 Die Turingmaschine ist gleich mächtig wie der von-Neumann-Rechner, das heißt, dass jedes Problem, das ein von-Neumann-Rechner löst auch von einer Turingmaschine gelöst werden kann und umgekehrt.

Damit ist die Menge der Berechnungsprobleme, die von Turingmaschinen gelöst werden können gleich der Menge der Berechnungsprobleme, die mit einer "klassischen" Programmiersprache (wie zum Beispiel C) gelöst werden können. Man nennt eine derartige Programmiersprache **Turing-mächtig**.

Kapitel 4

Übungen

4.1 Sortieren

Sortieren durch Einfügen

Aufgabe 1

a) Programmieren Sie “Sortieren durch Einfügen” in C für Integerzahlen - Arrays.

b) Testen Sie das Programm mit sortierten, zufälligen und umgekehrt sortierten Arrays auf Korrektheit.

c) Bestimmen Sie für sortierte, zufällige und umgekehrt sortierte Arrays die Parameter a, b und c von $T(n) = a \cdot n^2 + b \cdot n + c$. Da in der Sprache C die Länge statischer Arrays auf 250000 beschränkt ist, sollten Sie mit dynamischen Arrays arbeiten. Das geht (mit C++) z.B. so:

```
/* Deklaration Array */
int* a;
n = 300000000
/* Speicherplatz reservieren */
a = new int[n];
```

d) Bestimmen Sie die theoretische Rechenzeit für ein Array der Länge $5 \cdot 10^9$.

e) Warum wird in der Praxis $T(5 \cdot 10^9)$ viel größer sein als oben errechneter Wert?

f) Durch welche Massnahme kann man die berechnete Rechenzeit tatsächlich erreichen?

Quicksort

Aufgabe 2

Skizzieren Sie den Rekursionsbaum von Quicksort für ein konstantes Aufteilungsverhältnis von 1:k und leiten Sie aus diesem Baum $T(n) = O(n \lg n)$ ab.

Aufgabe 3

Programmieren Sie Quicksort für einfache Arrays von Integer-Zahlen, indem Sie die Funktion Partition wie in der Vorlesung beschrieben implementieren. Zeigen Sie empirisch, daß für zufällig sortierte Arrays $T_{\text{rand}}(n) = \Theta(n \lg n)$ und für vorsortierte Arrays $T_{\max}(n) = \Theta(n^2)$ gilt.

Aufgabe 4

Beweisen Sie, daß für Sortieralgorithmen gilt $T_{min}(n) = \Omega(n)$.

Aufgabe 5

Gegeben sei folgender Programmteil eines C-Programms:

```
for(i = 1; i <= n; i++)
    for(j = 1; j <= n; j++)
        for(k = 1; k <= j; k++)
            z = z+1;
```

a) Berechnen Sie die Laufzeit $T(n)$ und geben Sie die (asymptotische) Zeitkomplexität an.

b) Welchen Wert hat z nach Verlassen der äußersten Schleife für $n = 100$?

Aufgabe 6

Gegeben sei die rekursive Funktion *fib*:

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \quad \text{für } n \geq 2. \end{aligned}$$

a) Berechnen Sie $fib(5)$.

b) Zeichnen Sie den Rekursionsbaum für *fib*.

c) Bestimmen Sie anhand des Rekursionsbaumes die Komplexität von *fib*.

Heapsort

Aufgabe 7

Zeigen Sie, daß gilt

$$\sum_{k=0}^{\infty} kq^k = \frac{q}{(1-q)^2}.$$

Tip: Entweder Sie schreiben die Reihe gliedweise in geeigneter Form und wenden dann die Formel für die geometrische Reihe an, oder Sie starten indem Sie die Formel für die geometrische Reihe differenzieren.

Aufgabe 8

Warum wird der Schleifenindex i in Zeile 2 von Build-Heap von $\lfloor \text{length}[A]/2 \rfloor$ bis 1 erniedrigt und nicht von 1 bis $\lfloor \text{length}[A]/2 \rfloor$ erhöht?

Aufgabe 9

Stellen Sie den Ablauf von Build-Heap für $A = (5, 3, 17, 10, 84, 19, 6, 22, 9)$ grafisch dar, ähnlich wie in Beispiel 1.10.

Aufgabe 10

Stellen Sie den Ablauf von HEAPSORT für $A = (5, 13, 2, 25, 7, 17, 20, 8, 4)$ grafisch dar, ähnlich wie in Beispiel 1.11.

Aufgabe 11

Benutzen Sie die Master-Methode zur Berechnung der Laufzeit der Suche durch Bisektion mit der Rekurrenzgleichung $T(n) = T(n/2) + \Theta(1)$.

Aufgabe 12

Geben Sie mit Hilfe des Master-Theorems asymptotische Schranken für folgende Rekurrenzen an:

a) $T(n) = 4T(n/2) + n$

b) $T(n) = 4T(n/2) + n^2$

c) $T(n) = 4T(n/2) + n^3$.

4.2 Graphen

Aufgabe 13

Geben Sie für den Süddeutschlandgraphen die zugehörige Adjazenzliste an.

Aufgabe 14

Erzeugen Sie zwei möglichst unterschiedliche aufspannende Bäume für den SFBay-Graphen indem Sie in Palo Alto starten.

Aufgabe 15

Verwenden Sie den Algorithmus von Dijkstra für das Single-Source-Shortest-Path-Problem beim SFBay-Graphen mit Start in Palo Alto.

Aufgabe 16

Erzeugen Sie einen minimal aufspannenden Baum mit dem Algorithmus von Kruskal für den SFBay-Graphen.

Aufgabe 17

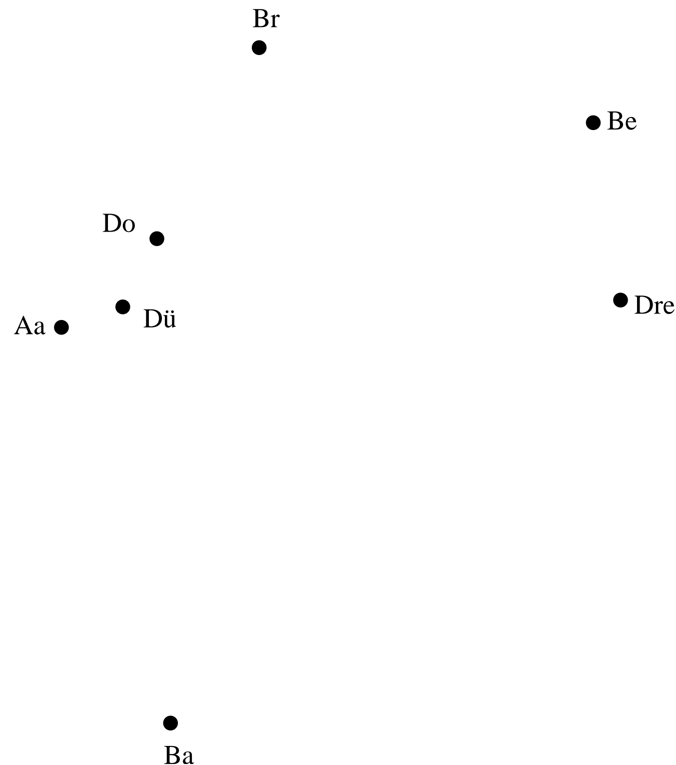
Gegeben ist folgende Entfernungstabelle deutscher Städte:

	Aachen	Basel	Berlin	Bremen	Dortmund	Dresden	Düsseldorf
Aachen							
Basel	545						
Berlin	650	875					
Bremen	370	775	400				
Dortmund	155	555	495	235			
Dresden	645	745	200	490	515		
Düsseldorf	90	550	560	285	70	580	

a) Bestimmen Sie mit dem Greedy-Algorithmus eine Lösung für das TSP-Problem. Starten Sie in Düsseldorf. Geben Sie die Tour sowie deren Länge an.

b) Bestimmen Sie einen Minimum Spanning Tree mit dem Kruskal-Algorithmus. Zeichnen Sie diesen in die Landkarte (unten) ein.

c) Verwenden Sie die Minimum-Spanning-Tree-Heuristik mit Wurzelknoten Bremen zur Bestimmung einer Lösung des TSP-Problems. Zeichnen Sie die Lösung in die Karte ein und geben Sie Tour sowie Länge an.



Aufgabe 18

Versuchen Sie, die Minimum-Spanning-Tree-Heuristik zur Lösung des TSP-Problems beim SFBay-Graphen anzuwenden. Vergleichen Sie das Ergebnis mit dem des Greedy-Algorithmus bei Start in San Franzisko oder anderen Städten. Warum treten hier Probleme auf?

Aufgabe 19

Bestimmen Sie eine optimale Tour für das TSP-Problem beim SFBay-Graphen.

Aufgabe 20

Für einen planaren (ebenen) ungerichteten Graphen sei A die Menge der durch die Kanten des Graphen begrenzten Flächenstücke. Überprüfen Sie an allen bisher verwendeten Graphen (V, E) die Gültigkeit der Euler-Formel

$$|V| - |E| + |A| = 2.$$

Problem des Handlungsreisenden

Aufgabe 21

Es soll ein Programm entworfen werden, das bei gegebener Abstandsmatrix für vollständig vollständige Graphen eine optimale Tour findet.

a) Schreiben Sie ein Programm für OPTIMAL_TSP mit 4 Städten

b) Schreiben Sie ein Programm für OPTIMAL_TSP mit n Städten

4.3 Formale Sprachen und Automaten

Aufgabe 22

Gegeben sei die Grammatik ([7], S. 15)

$$\begin{aligned} G &= (V, \Sigma, P, S)\ , \text{ wobei:} \\ V &= \{S, B, C\} \\ \Sigma &= \{a, b, c\} \\ P &= \{S \to aSBC, S \to aBC, CB \to BC, \\ &\quad aB \to ab, bB \to bb, bC \to bc, cC \to cc\} \end{aligned}$$

Konstruieren Sie eine Ableitung für *aabbcc*. Welchen Chomski-Typ hat diese Grammatik?

Aufgabe 23

Geben sie eine möglichst einfache Grammatik an, die alle Zeichenketten der Form
ab, *abab*, *ababab*, ... erzeugt. Welchen Chomski-Typ hat diese Grammatik?

Aufgabe 24

Geben sie eine möglichst einfache Grammatik an, die alle Zeichenketten der Form
ab, *aabb*, *aaabbb*, ... erzeugt. Welchen Chomski-Typ hat diese Grammatik?

Aufgabe 25

Geben sie eine möglichst einfache Grammatik an, die alle Zeichenketten der Form
abba, *ababbaba*, *abababbababa*, ... erzeugt. Zeichnen Sie den Syntaxbaum für das Wort *ababbaba*. Welchen Chomski-Typ hat diese Grammatik?

Aufgabe 26

Definieren Sie eine Grammatik, die einfache Programme folgender Art beschreibt. Es gibt im Programmrumpf nur Wertzuweisungen, Terme sowie den print-Befehl

```
function plusplus(x,y,z)
   var int u,v;
   var float w;

   u = x + y;
   v = x * (z+y);
   w = x / z;
   print(u,v,w)
end
```

Aufgabe 27

a) Zeigen Sie, dass die Menge aller C-Programme unendlich ist.

b) Geben sie eine obere Schranke für die Zahl der C-Programme der Länge n an.

Aufgabe 28

Gegeben sei die Grammatik $G = (V, \Sigma, P, S)$ mit $V = \{S, A, B\}$, $\Sigma = \{a, b, c\}$ und

$$P = \{ \begin{array}{lll} S \to aA, & A \to aA, & B \to bB, \\ S \to bA, & A \to bB, & B \to c, \\ & A \to c & \end{array} \}$$

a) Geben Sie eine Ableitung an für *abbbbc*.

b) Geben Sie alle Worte der Länge 4 der zugehörigen Sprache $L = L(G)$ an.

c) Geben Sie einen regulären Ausdruck für die Sprache L an.

d) Zeichnen Sie den Zustandsgraphen eines endlichen Automaten, der L akzeptiert.

e) Geben Sie den endlichen Automaten als Formel an.

f) Welchen Chomsky Typ hat diese Sprache? (Begründung!)

Aufgabe 29

Geben Sie eine Grammatik an für die Sprache (Buch S. 25) $L = \{a^i b^j c^k \mid i = j \text{ oder } j = k\}$
Zeigen sie daß diese Grammatik mehrdeutig ist.

Aufgabe 30

Geben Sie reguläre Ausdrücke an für

a) groß geschriebene Worte wie z.B. `Hallo`, aber nicht `HALLO`.

b) groß geschriebene Worte mit mindestens 3 und höchstens 10 Buchstaben.

c) Gleitpunktzahlen mit beliebig vielen Stellen vor dem Komma und mindestens einer Stelle nach dem Komma.

d) Datumsangaben der Form `21.10.1999` oder `1.1.2000` oder `1.1.‘00` oder `21.10.‘99`. Nicht erlaubt sind unzulässige Werte wie z.B. `33.44.‘99` oder `121.10.1999`

Aufgabe 31

Beschreiben Sie die durch folgende regulären Ausdrücke definierten Sprachen und geben Sie Beispiele an.

a) `\\(index|color|label|ref)\{[^\}]*\}`

b) `%.*$`

c) `\\section*?\{.*\}`

d) `[A-Za-z0-9]+@[A-Za-z0-9]+(\.[A-Za-z0-9]+){1,6}`

Aufgabe 32

Schreiben Sie ein Lex-Programm, das in einer Datei alle Datumsangaben vom Format $< Tag > . < Mon > . < Jahr >$ in das Format $< Mon > - < Tag > - < Jahr >$ übersetzt. $< Tag >$ und $< Mon >$ sind zu verstehen wie in Aufgabe 30.

Aufgabe 33

Konstruieren Sie mit Lex und Yacc einen Parser für die Grammatik in Aufgabe 26.

Aufgabe 34

Es soll ein Getränkeautomat mit Hilfe eines endlichen Automaten programmiert werden. Der Automat kann mit bis zu 4 Dosen Mineralwasser, 4 Dosen Limo und 4 Dosen Bier gefüllt werden. Wenn eine 1-Euro-Münze eingegeben wird, soll er eine Dose des gewählten Getränks ausgeben. Bei Eingabe einer anderen Münze soll er die eingegebene Münze wieder ausgeben, aber kein Getränk. Wenn von einer Getränkesorte alle Dosen ausgegeben sind, soll er anhalten und per Funk den Service benachrichtigen.

a) Geben Sie einen endlichen Automaten (mit Ausgabe) für diese Aufgabe an.

b) Zeichnen Sie ein Zustandsdiagramm zu diesem Automaten.

c) Geben Sie einen regulären Ausdruck für diese Sprache an.

d) Geben Sie eine reguläre Grammatik an, welche die für diesen Automaten erlaubte Eingabesprache akzeptiert.

Aufgabe 35

Es soll eine Fußgängerampel mit Hilfe eines endlichen Automaten programmiert werden. Die Ampel hat die zwei Zustände rot und grün (aus der Sicht des Fahrzeugs). Im Zustand rot akzeptiert die Ampel Signale von der Kontaktschleife auf der Straße und schaltet dann auf Grün. Im Zustand Grün akzeptiert die Ampel Signale vom Fußgängertaster und schaltet auf Rot. Alle anderen Eingaben ignoriert der Automat.

a) Geben Sie einen endlichen Automaten für diese Aufgabe an.

b) Zeichnen Sie ein Zustandsdiagramm zu diesem Automaten.

c) Geben Sie einen regulären Ausdruck für diese Sprache an.

d) Geben Sie eine reguläre Grammatik für diesen Automaten an.

e) Zeigen Sie, dass diese Ampelschaltung bei geringem Verkehrsaufkommen das Mehrheitsprinzip exakt erfüllt, das heisst, das Verhältnis aus rot- zu grün-Zuständen ist gleich dem Verhältnis aus Fußgängerzahl zu Autofahrerzahl.

Aufgabe 36

konstruieren sie (deterministische oder nichtdeterministische) endliche Automaten für folgende durch reguläre Ausdrücke gegebenen Sprachen:

a) `[0-9]*\.[0-9]+`

b) `\\section*?\{.*\}`

Aufgabe 37

a) Entwerfen sie für das Alphabet {a,b} einen Kellerautomaten, der alle Worte der Form $x_1, x_2, \ldots x_n \$ y_1, y_2, \ldots y_m$ erkennt, wobei die Zahl der a-s vor und nach dem “$”-Zeichen gleich groß sein soll.

b) Geben Sie für diese Sprache eine BNF Grammatik an.

Aufgabe 38

a) Ändern Sie den Automaten aus Aufgabe 37 so ab, daß er nur Worte erkennt, für die $n = m$ ist.

b) Geben Sie für die geänderte Sprache auch eine BNF Grammatik an.

Aufgabe 39

Entwerfen Sie sie für das Alphabet {a,(,)} einen Kellerautomaten, der genau die korrekt geklammerten Ausdrücke erkennt.

Aufgabe 40

Gegeben sei die BNF-Grammatik $G = (\{S, T, Z\}, \{a, [,]\}, P, T)$ mit

$$P = \{ \begin{array}{l} T \rightarrow [S[STS]S] \mid \varepsilon \\ S \rightarrow ZS \mid Z \mid \varepsilon \\ Z \rightarrow a \end{array} \}$$

a) Geben Sie eine Linksableitung an für $[a[a]a]$.

b) Geben Sie in einer Tabelle alle Worte der Längen 1 bis 6 der zugehörigen Sprache $L = L(G)$ an.

c) Beschreiben Sie die Sprache L in ein bis zwei Sätzen.

d) Geben Sie einen Kellerautomaten an, der L akzeptiert.

e) Welchen Chomsky Typ hat diese Sprache? (Begründung!)

Aufgabe 41

Konstruieren Sie eine Turingmaschine M zur Berechnung der Parität des Eingabewortes $w \in \{0, 1\}^*$. Die Parität p eines Wortes w ist Null wenn w eine gerade Zahl von Einsen enthält und Eins sonst. M startet in der Konfiguration $\ldots \square z_0 w \square \ldots$ mit Startzustand z_0 und stoppt im Endzustand z_e in der Konfiguration $\ldots \square w \square z_e p \square \ldots$.

Aufgabe 42

Entwerfen Sie für folgende Aufgaben je eine Turingmaschine:

a) Löschen des gesamten Bandes, d.h. alle Einsen und Nullen werden durch $\square$ ersetzt.

b) Invertieren der Eingabe, d.h. jede Null wird zur Eins und umgekehrt.

c) Multiplikation der Eingabe mit 2.

d) Kopieren der Eingabe, d.h. die Maschine erzeugt aus der Startkonfiguration

$$\ldots \square\square\square z_0 x_1 x_2 \ldots x_n \square\square\square \ldots$$

die Stopkonfiguration

$$\ldots \square\square\square z_0 x_1 x_2 \ldots x_n \square x_1 x_2 \ldots x_n \square\square\square \ldots$$

Aufgabe 43

Suchen Sie im Internet nach Simulatoren für Turingmaschinen sowie nach Online Beschreibungen, etc.

Aufgabe 44

Entwerfen Sie möglichst fleißige Biber mit 2, 3, und 4 Zuständen.

Kapitel 5

Lösungen zu den Übungsaufgaben

5.1 Sortieren

Sortieren durch Einfügen

Aufgabe 1

a) Lösung des Programms

b) siehe Programm

c) Tabelle zur Ermittlung der Koeffizienten a,b,c über LGS aus Messungen

n	absteigend	aufsteigend	zufällig
20 000	80 960 ms	28 940 ms	54 870 ms
10 000	27 460 ms	14 440 ms	20 980 ms
5 000	10 490 ms	7 200 ms	8 790 ms
2 500	4 400 ms	3 570 ms	4 010 ms

absteigend (Worst Case):
$c = 40$

$b = \frac{7220-30}{5000} = 1,438$

$a = \frac{80960-40-28760}{400000000} = 1,304 \cdot 10^{-4}$

aufsteigend (Best Case):
$c = -33,3$

$b = \frac{7205+25}{5000} = 1,446$

$a = \frac{28940-28920+12,5}{400000000} = 8,125 \cdot 10^{-8}$

zufällig (Average Case):
$c = -230$

$b = \frac{7262,5+172,5}{5000} = 1,487$

$a = \frac{54870-29740+230}{400000000} = 6,34 \cdot 10^{-5}$

d) Worst Case: $1,30 \cdot 10^{-4} \cdot (5 \cdot 10^9)^2 + 1,438 \cdot 5 \cdot 10^9 + 40 = 3,26 \cdot 10^{15} ms \approx 103374\ Jahre$

Best Case: $8,125 \cdot 10^{-8} \cdot (5 \cdot 10^9)^2 + 1,446 \cdot 5 \cdot 10^9 - 33,\bar{3} = 2,04 \cdot 20^{12} ms \approx 65\ Jahre$

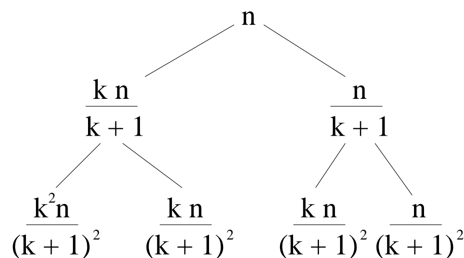
Zufällig: $6,34 \cdot 10^{-5} \cdot (5 \cdot 10^9)^2 + 1,487 \cdot 5 \cdot 10^9 - 230 = 1,59 \cdot 10^{15} ms \approx 50260\ Jahre$

e) In der Praxis wird der Wert viel größer wie der oben errechnete Wert, da es Hardwaregrenzen wie z.B. den Hauptspeicher gibt. Wenn dieser sein Maximum an Kapizität erreicht hat, muß er Daten auf die Festplatte auslagern, dies kostet Zeit.

f) Durch die Anpassung der Hardware. D.h. den Speicher ausreichend erweitern, und den Adressbus vergrössern.

Quicksort

Aufgabe 2



Tiefe der untersten Ebene:

$\left(\frac{k}{k+1}\right)^{dl} \cdot n = 1$

Rechenaufwand pro Ebene $\leq n$

$dl \cdot \log \frac{k}{k+1} + \log n = 0$

$dl = \frac{\log n}{\log \frac{k+1}{k}} \Rightarrow \frac{c}{\frac{k+1}{k}} \cdot \log n \cdot n$

$\Rightarrow T(n) = O(n \cdot \log n)$

Aufgabe 3

Messungen:

n	T(n) [sec]
2000000	1.1
4000000	2.47
6000000	3.5
8000000	4.76
10000000	5.84
15000000	9.41
20000000	12.72
40000000	26.98
80000000	55.46

Für $n \to \infty$ gilt $T(n) = Cn \ln n$, also $C = T(n)/n \ln n$
Man erhält also die folgende Tabelle und schließt daraus $C \approx 3.810^{-8}$ sec, d.h. $T(n) \approx 3.810^{-8} n \ln n sec.$

n	$T(n)/(n \ln n)$ [sec]
2000000	$3.79084\ 10^{-8}$
4000000	$4.06202\ 10^{-8}$
6000000	$3.73757\ 10^{-8}$
8000000	$3.74333\ 10^{-8}$
10000000	$3.62326\ 10^{-8}$
15000000	$3.7966\ 10^{-8}$
20000000	$3.78318\ 10^{-8}$
40000000	$3.85332\ 10^{-8}$
80000000	$3.80958\ 10^{-8}$

Aufgabe 4 Weil der Algorithmus auf jedes Element mindestens einmal zugreifen muß, um zu überprüfen, ob es an der richtigen Stelle steht. Ansonsten ist nicht garantiert, daß die gesamte Liste sortiert ist.

Aufgabe 5

a)

$$T(n) = n \cdot \sum_{j=1}^{n} j = n \cdot \frac{n(n+1)}{2} = \frac{n}{2}(n^2+n) = \frac{1}{2}(n^3+n^2) = \Theta(n^3)$$

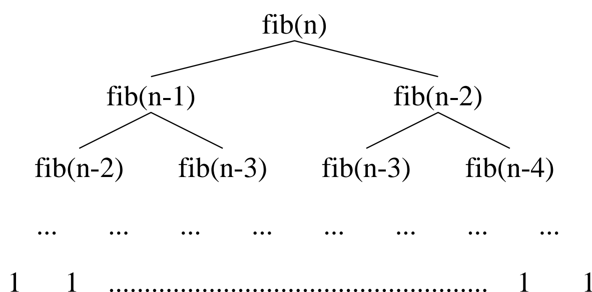
b)

$$\frac{1}{2}(10^6+10^4) = \frac{1}{2}1.01 \cdot 10^6 = 0.505 \cdot 10^6 = 505000$$

Aufgabe 6

a) $\mathit{fib}(5) = 8$

b)



c) Tiefe des Baumes: n, Zahl der Blätter: 2^n
Zahl der Knoten auf Ebene k: 2^k
Zahl der Knoten gesamt:

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

Heapsort

Aufgabe 7 Formel für geometrische Reihe:

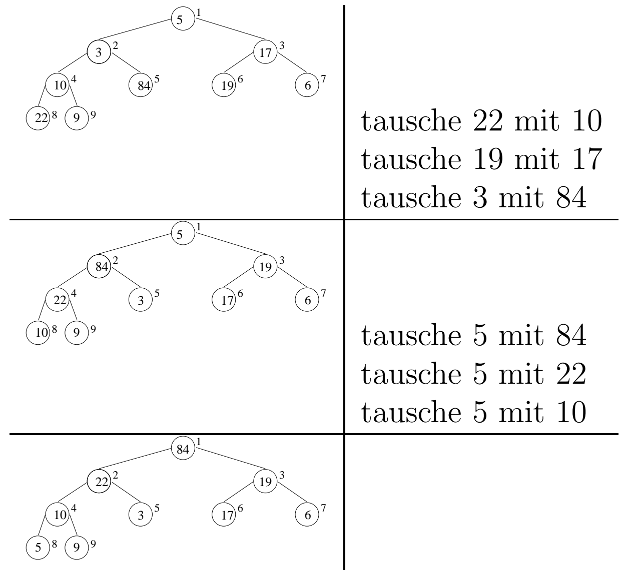
$$\sum_{k=0}^{\infty} q^k = \frac{1}{1-q}$$

differenziert:

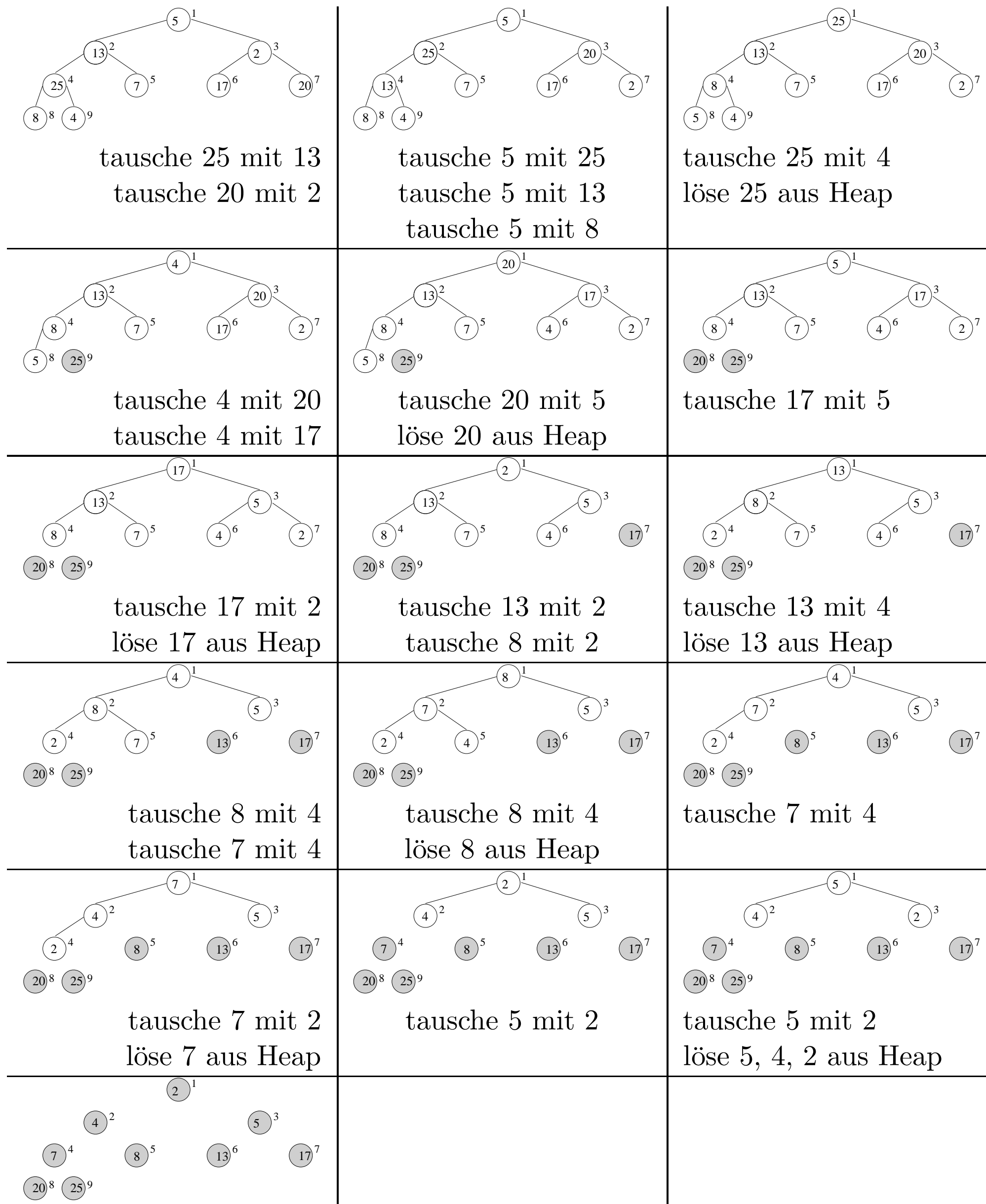
$$\sum_{k=0}^{\infty} k \cdot q^{k-1} = \frac{1}{(1-q)^2} \Rightarrow \sum_{k=0}^{\infty} k \cdot q^k = \frac{q}{(1-q)^2}$$

Aufgabe 8 Heapify setzt voraus, dass die Unterbäume schon Heaps sind. Darum muß der Baum von unten nach oben in einen Heap umgewandelt werden.

Aufgabe 9



Aufgabe 10



Aufgabe 11 $f(n) = 1n^{\log_2 1} = n^0 = 1 \Rightarrow Fall2 \Rightarrow T(n) = \Theta(n^0 \cdot \log n) = \Theta(\log n)$

Aufgabe 12

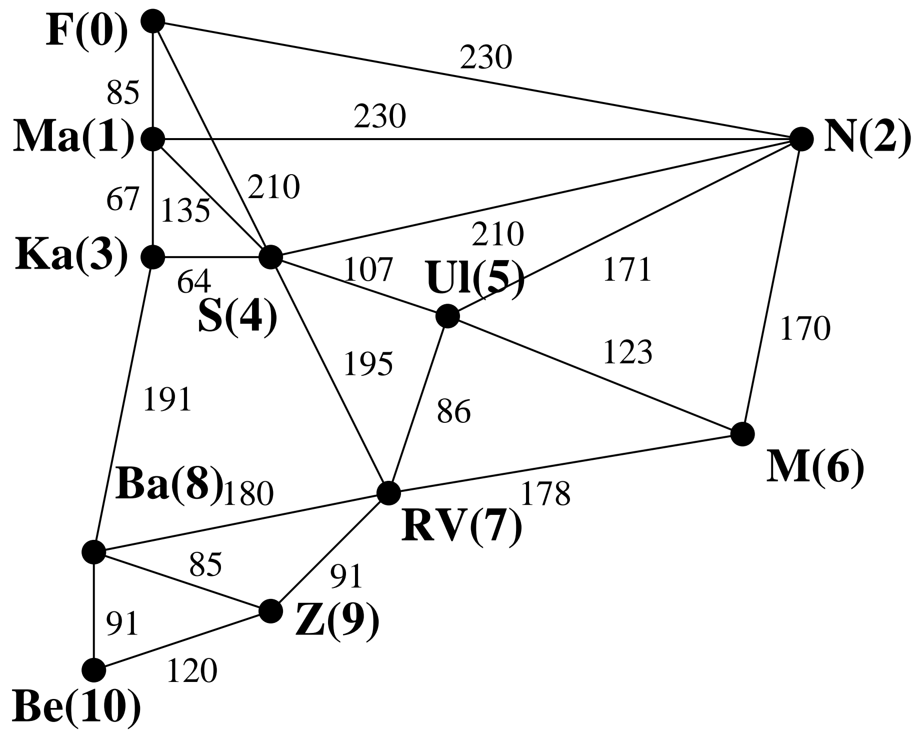
a) $f(n) = n \qquad n^{\log_2 4} = n^2 \qquad \Rightarrow$ Fall 1 $\qquad T(n) = \Theta(n^2)$

b) $f(n) = n^2 \qquad n^{\log_2 4} = n^2 \qquad \Rightarrow$ Fall 2 $\qquad T(n) = \Theta(n^2 \cdot \log n)$

c) $f(n) = n^3 \qquad n^{\log_2 4} = n^2 \qquad \Rightarrow$ Fall 3 $\qquad T(n) = \Theta(n^3)$

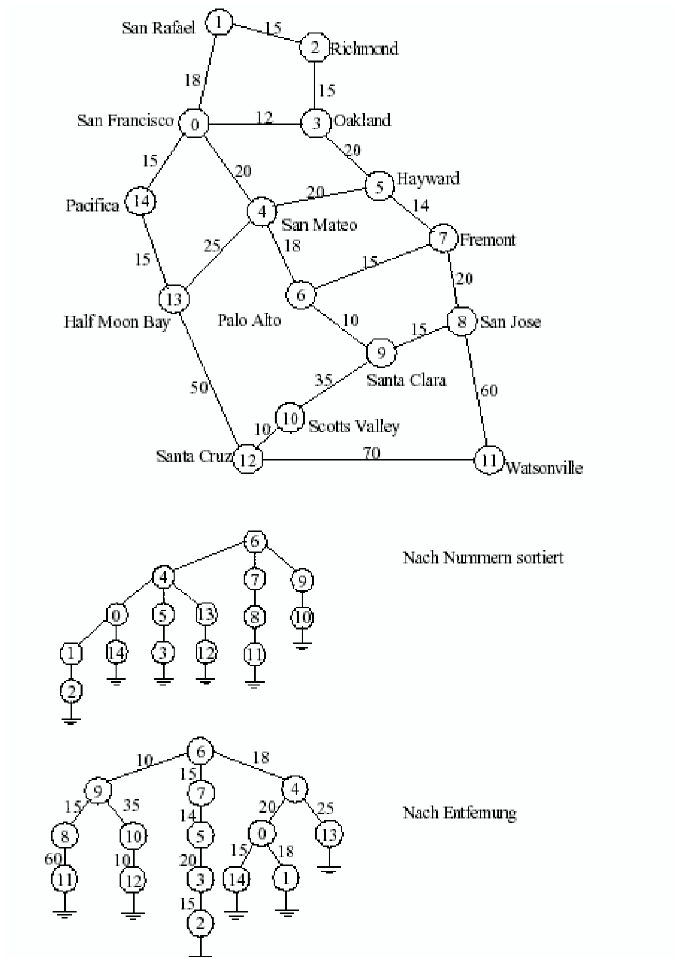
5.2 Graphen

Aufgabe 13



0 F	1 Ma	2 N	3 Ka	4 S	5 Ul	6 M	7 RV	8 Ba	9 Z	10 Be
1	0	0	1	0	2	2	4	3	7	8
2	2	1	4	1	3	5	5	7	8	9
4	3	4	8	2	6	7	6	9	10	
	4	5		3	7		8	10		
		6		5			9			
				7						

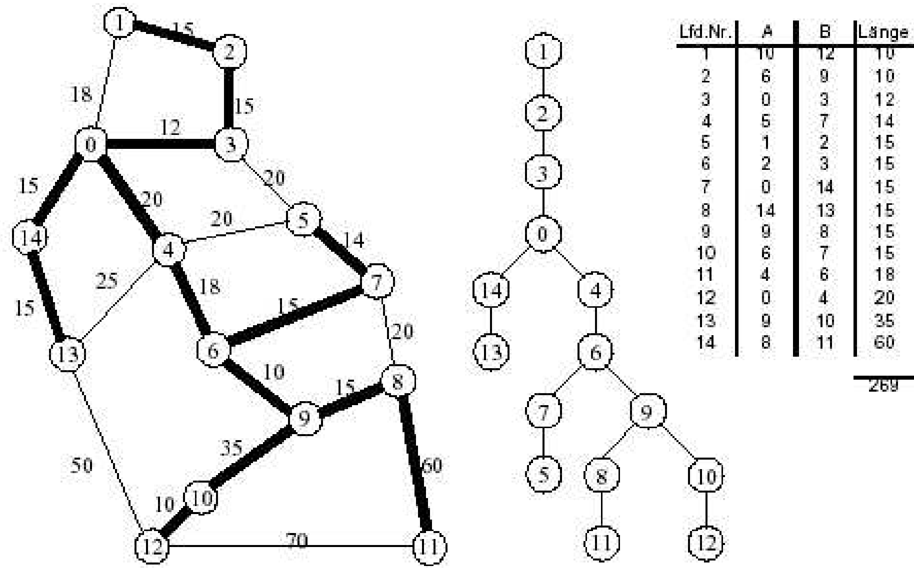
Aufgabe 14



Aufgabe 15

Nr	Name	Entfernung
6	Palo Alto	0
9	Santa Clara	10
7	Fremont	15
4	San Manteo	18
8	San Jose	25
5	Hayward	29
0	San Francisco	38
13	Half Moon Bay	43
10	Scotts Valley	45
3	Oakland	49
14	Pacifica	53
12	Santa Cruz	55
1	San Rafael	56
2	Richmond	64
11	Watsonville	85

Aufgabe 16



Lfd.Nr.	A	B	Länge
1	10	12	10
2	6	9	10
3	0	3	12
4	5	7	14
5	1	2	15
6	2	3	15
7	0	14	15
8	14	13	15
9	9	8	15
10	6	7	15
11	4	6	18
12	0	4	20
13	9	10	35
14	8	11	60
			269

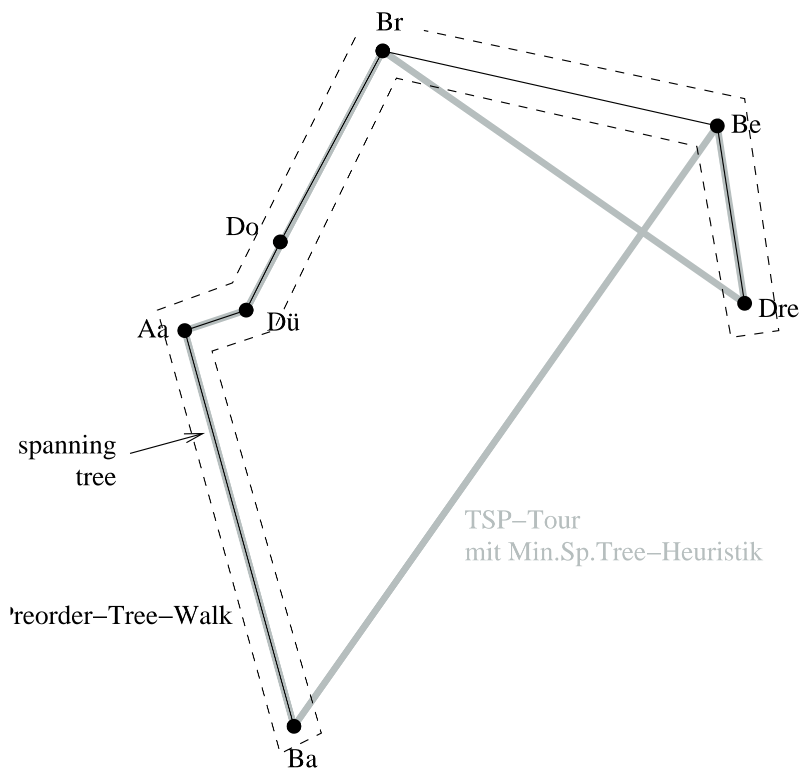
Suchen der kürzesten Kante, dann der zweitkürzesten,
dann der drittkürzesten usw.
Wichtig: Es darf dabei kein Zyklus entstehen.

Aufgabe 17 Süddeutschlandgraph:

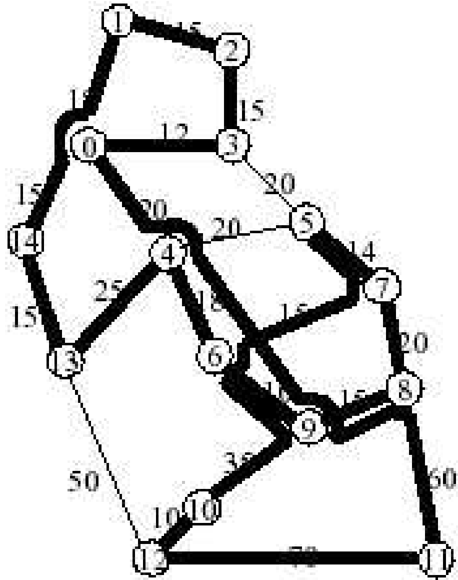
a) Dü,Do,Aa,Br,Ber,Dre,Bas,Dü, Gesamtentfernung: 2490

b) siehe unten

c) Dü,Do,Br,Dre,Ber,Bas,Aa,Dü, Gesamtentfernung: 2505, siehe unten



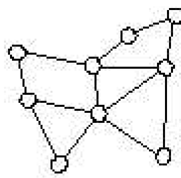
Aufgabe 18 Da es nur recht wenige Verbindungen in diesem Graph gibt, wundert es nicht, dass der Greedy-Algorithmus versagt.

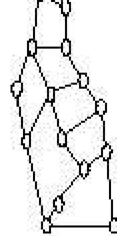


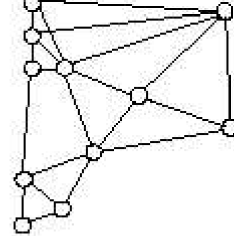
Aufgabe 19 Die Lösung auf das TSP-Problem beim SFBay-Graphen ist eindeutig und intuitiv:

Die optimal Tour:
$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 11 \rightarrow 12 \rightarrow 10 \rightarrow 9 \rightarrow 6 \rightarrow 4 \rightarrow 13 \rightarrow 14 \rightarrow 0 \rightarrow 1$

Aufgabe 20







V = Knoten; E = Kanten; A = Flächen;

Problem des Handlungsreisenden

Aufgabe 21 einen Lösungsansatz findet ihr unter http://zebra.fh-weingarten.de/~scheremet/gin

5.3 Formale Sprachen und Automaten

Aufgabe 22 Ableitung:

S ⇒ aSBC ⇒ aaBCBC ⇒ aabCBC ⇒ aabBCC ⇒ aabbCC ⇒ aabbcc

Chomsky Typ 1

Aufgabe 23 $P = \{S \rightarrow aB, B \rightarrow bS, B \rightarrow b\}$ Chomsky Typ 3

Aufgabe 24 $P = \{S \rightarrow aSb, S \rightarrow ab\}$ Chomsky Typ 2

Aufgabe 25 $P = \{S \rightarrow abSba, S \rightarrow abba\}$ Chomsky Typ 2

Aufgabe 26

$$
\begin{array}{rcl}
< Funkt > & \rightarrow & < Kopf >< Rumpf > end \\
< Kopf > & \rightarrow & function < Name > (< Liste >) \mid function < Name > (< Liste >) < Var \\
< Liste > & \rightarrow & < Name >, < Liste > | < Name > \\
< Vardekl > & \rightarrow & < Varzeile >< Vardekl > | < Varzeile > \\
< Varzeile > & \rightarrow & var < Typ >< Liste > \\
< Typ > & \rightarrow & int \mid float \\
< Rumpf > & \rightarrow & < Anweisung >; < Rumpf > | < Anweisung > \\
< Anweisung > & \rightarrow & < Wertzuweisung > | < Drucken > \\
< Drucken > & \rightarrow & print(< Liste >) \\
< Wertzuweisung > & \rightarrow & < Name >=< Term > \\
< Name > & \rightarrow & < Buchstabe >< N - Rest > \\
< N - Rest > & \rightarrow & < Zeichen >< N - Rest > | < \Sigma > \\
< Zeichen > & \rightarrow & < Buchstabe > | < Ziffer > \\
< Buchstabe > & \rightarrow & A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
< Ziffer > & \rightarrow & 0 \mid 1 \mid 2 \mid \ldots \mid 9
\end{array}
$$

Für die Definition von $< Term >$ siehe Beispiel 3.4 auf Seite 57.

Aufgabe 27

a) Die Menge der C-Programme ist unendlich, da schon die Klasse aller C-Pogramme, welche nur aus beliebig vielen Wiederholungen von `printf(''x'');` bestehen unendlich ist.

```
main ()
{
    printf(''x'');
    ...
    printf(''x'');
}
```

b) 128 erlaubte Tastaturzeichen. Dann gibt es 128^n Worte der Länge n.

$|\{$C-Programm der Länge $n\}|$ ¡ 128^n

Aufgabe 28

a) $S \Rightarrow aA \Rightarrow abB \Rightarrow abbB \Rightarrow abbbB \Rightarrow abbbbB \Rightarrow abbbbc$

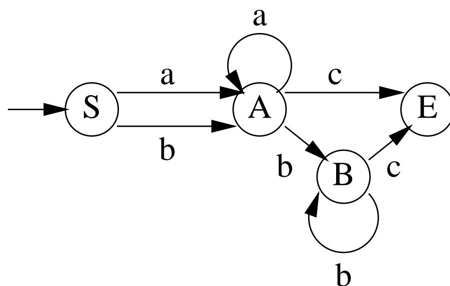
b) $aaac,\ aabc,\ abbc,\ baac,\ babc,\ bbbc$

c) $(a|b)(a * |b*)b * c$

d) $Z = \{S, A, B, E\}$, $\Sigma = \{a, b, c\}$, $E = \{E\}$

$$
\begin{aligned}
\delta(S, a) &= A \\
\delta(S, b) &= A \\
\delta(A, a) &= A \\
\delta(A, b) &= B \\
\delta(A, c) &= E \\
\delta(B, b) &= B \\
\delta(B, c) &= E
\end{aligned}
$$

e)



f) Chomsky Typ 3 (kann man mit einem regulären Ausdruck beschreiben)

Aufgabe 29

$$
\begin{aligned}
P = \{&S \rightarrow AB, S \rightarrow CD, \\
&A \rightarrow aAb, A \rightarrow ab, \\
&B \rightarrow cB, B \rightarrow c, \\
&C \rightarrow aC, C \rightarrow a, \\
&D \rightarrow bDc, D \rightarrow bc\}
\end{aligned}
$$

Die Grammatik ist mehrdeutig, da für ein Wort z.B *abc* mehrere Lösungsbäume existieren.

Aufgabe 30

a) [A-Z][a-z]*

b) [A-Z][a-z\-]{2,9}

c) [0-9]*\.[0-9]+

d) ([1-9]|([1-2][0-9])|3[01])\.([1-9]|(1[0-2]))\.([1-9]{1,4}|’[1-9]{1,2})

Aufgabe 31

a) \index{xxx} \color{xxx} \label{xxx} \ref{xxx}

b) %<text> am Zeilenende.

c) \section{<text>}, \section*{<text>}

d) <text1>@<text2>.<text3>.<text4> ... bis maximal <text6>

Aufgabe 32 Variante 1:

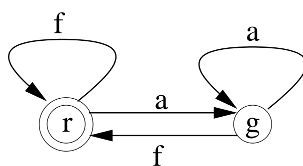
```
%option noyywrap
%%
[1-9]\.[1-9]\.[1-9]{4}  printf("%c-%c-%c%c%c%c", yytext[2],yytext[0],
        yytext[4],yytext[5],yytext[6],yytext[7]);

([1-2][0-9]|3[0-2])\.[1-9]\.[1-9]{4} printf("%c-%c%c-%c%c%c%c", yytext[3],
        yytext[0],yytext[1],yytext[5],yytext[6],yytext[7],yytext[8]);

[1-9]\.1[12]\.[1-9]{4}  printf("%c%c-%c-%c%c%c%c", yytext[2],yytext[3],
        yytext[0],yytext[5],yytext[6],yytext[7],yytext[8]);

([1-2][0-9]|3[0-2])\.1[12]\.[1-9]{4}    printf("%c%c-%c%c-%c%c%c%c", yytext[3],
        yytext[4],yytext[0],yytext[1],yytext[6],yytext[7],yytext[8],yytext[9]);
.
```

Aufgabe 34 Beim Erstellen dieses Automaten ist darauf zu achten, dass man mit mehreren Zuständen (über Indizes kennzeichnen) die Lösung findet.

Aufgabe 35

a) $(\{r, g\}, \{e, f, a\}, \delta, z_0, \{z_0\})$ wobei δ gegeben ist durch:

$$\begin{aligned} r, f &\to r \\ r, a &\to g \\ g, f &\to r \\ r, f &\to r \end{aligned}$$

b) 

c) [ab]*

d) $S \to Sa, \quad S \to Sb, \quad S \to a, \quad S \to b$

Aufgabe 36

a) $(\{z_0, z_1, z_e\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}, \delta, z_0, \{z_e\})$ wobei δ gegeben ist durch:

$$\begin{array}{lllll} z_0, 0 \to z_0 & z_0, 1 \to z_0 & z_0, 2 \to z_0 & \ldots & z_0, 9 \to z_0 \\ z_0, . \to z_1 & & & & \\ z_1, 0 \to z_e & z_1, 1 \to z_e & z_1, 2 \to z_e & \ldots & z_1, 9 \to z_e \\ z_e, 0 \to z_e & z_e, 1 \to z_e & z_e, 2 \to z_e & \ldots & z_e, 9 \to z_e \end{array}$$

b) $(\{z_0, z_1, z_2, z_3, z_{11}, ..., z_e\}, ASCII - Zeichensatz, \delta, z_0, \{z_e\})$ wobei δ gegeben ist durch:

$$\begin{array}{lllll}
z_0, \backslash \rightarrow z_1 & & & & \\
z_1, s \rightarrow z_2 & z_2, e \rightarrow z_3 & z_3, c \rightarrow z_4 & ... & z_7, n \rightarrow z_8 \\
z_8, * \rightarrow z_9 & & & & \\
z_9, \{ \rightarrow z_{10} & & & & \\
z_8, \{ \rightarrow z_{10} & & & & \\
z_{10}, 0 \rightarrow z_{11} & z_{10}, 1 \rightarrow z_{11} & z_{10}, 2 \rightarrow z_{11} & ... & z_{10}, Z \rightarrow z_{11} \\
z_{11}, 0 \rightarrow z_{11} & z_{11}, 1 \rightarrow z_{11} & z_{11}, 2 \rightarrow z_{11} & ... & z_{11}, Z \rightarrow z_{11} \\
z_{11}, \} \rightarrow z_e & & & & \\
z_e, 0 \rightarrow z_{11} & z_e, 1 \rightarrow z_{11} & z_e, 2 \rightarrow z_{11} & ... & z_e, Z \rightarrow z_{11} \\
z_1, 0 \rightarrow z_e & z_1, 1 \rightarrow z_e & z_1, 2 \rightarrow z_e & ... & z_1, 9 \rightarrow z_e \\
z_e, 0 \rightarrow z_e & z_e, 1 \rightarrow z_e & z_e, 2 \rightarrow z_e & ... & z_e, 9 \rightarrow z_e
\end{array}$$

Aufgabe 40

a) $T \Rightarrow [S[STS]S] \Rightarrow [Z[STS]S] \Rightarrow [a[STS]S] \Rightarrow [a[ZTS]S] \Rightarrow [a[aTS]S] \Rightarrow [a[aTS]S] \Rightarrow [a[aS]S] \Rightarrow [a[aZ]S] \Rightarrow [a[a]S] \Rightarrow [a[a]Z] \Rightarrow [a[a]a]$

b)

Länge	Worte
1	–
2	–
3	–
4	[[]]
5	[[a]], [a[]], [[]a]
6	[aa[]], [[aa]], [[]aa], [a[a]], [a[]a], [[a]a]

c) Die Worte in L haben folgende Eigenschaften: Zuerst kommt eine gerade Anzahl öffnender Klammern und dann die gleiche Anzahl schliessender Klammern. Zwischen je zwei der Klammern können beliebig viele a's stehen.

d) Kellerautomat $K = \{\{z_0, z_1, z_2, z_3\}, \{a, [,]\}, \{K\}, \delta, z_0, \#\}$, wobei δ definiert ist durch:

$$\begin{array}{rcl}
z_0, [, \# & \rightarrow & z_1, \# \\
z_0, [, K & \rightarrow & z_1, K \\
z_1, [, \# & \rightarrow & z_0, K\# \\
z_1, [, \# & \rightarrow & z_2, K\# \\
z_1, [, K & \rightarrow & z_0, KK \\
z_1, [, K & \rightarrow & z_2, KK \\
z_2,], K & \rightarrow & z_3, K \\
z_3,], K & \rightarrow & z_2, \varepsilon \\
z_2,], K & \rightarrow & z_2, \varepsilon
\end{array}$$

e) L ist eine kontextfreie Sprache (Typ 2), denn die linken Seiten der Regeln in G bestehen nur aus je einer Variablen. Sie ist nicht regulär, denn ein endlicher Automat mit n Zuständen kann sich nicht mehr als n Paare von Klammern merken.

Aufgabe 41

$$z_0, 0 \to z_0, 0, R \qquad z_0, \square \to z_2, \square, R$$
$$z_0, 1 \to z_1, 1, R \qquad z_1, \square \to z_3, \square, R$$
$$z_1, 0 \to z_1, 0, R \qquad z_2, \square \to z_e, 0, N$$
$$z_1, 1 \to z_0, 1, R \qquad z_3, \square \to z_e, 1, N$$

Aufgabe 42

a)

		Eingabez.	
		0	1
Zust.	A	B1R	B1L
	B	A1L	C0L
	C	e1R	D1L
	D	D1R	A0R

Aufgabe 43 http://chicory.stanford.edu/~berezin/turing/

Aufgabe 44

Busy-Beaver mit 2 Zuständen:

		Eingabez.	
		0	1
Zust.	A	B1R	B1L
	B	A1L	H1R

$$\ldots\square\square\overset{A}{\square}\square\square\ldots$$
$$\vdash \ldots\square\square 1\overset{B}{\square}\square\square\ldots$$
$$\vdash \ldots\square\square\overset{A}{1}1\square\square\ldots$$
$$\vdash \ldots\square\square\overset{B}{\square}11\square\square\ldots$$
$$\vdash \ldots\square\square\overset{A}{\square}111\square\square\ldots$$
$$\vdash \ldots\square\square 1\overset{B}{1}11\square\square\ldots$$
$$\vdash \ldots\square\square 11\overset{H}{1}1\square\square\ldots$$

Busy-Beaver mit 3 Zuständen:

		Eingabez.	
		0	1
Zust.	A	B1R	C1L
	B	C1R	H1R
	C	A1L	B1L

Busy-Beaver mit 4 Zuständen:

		Eingabez.	
		0	1
Zust.	A	B1R	B1L
	B	A1L	C0L
	C	e1R	D1L
	D	D1R	A0R

Literaturverzeichnis

[1] M. Brill. *Mathematik für Informatiker*. Hanser Verlag, 2001. Sehr gutes Buch, das auch diskrete Mathematik beinhaltet.

[2] V. Claus and Schwill A. *Duden Informatik*. Bibliographisches Institut & F.A. Brockhaus AG, 1988. Ein gutes Nachschlagewerk zur Informatik allgemein.

[3] T.H. Cormen, Ch.E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass, 1994. Sehr gute Einführung in die Analyse von Algorithmen.

[4] A. K. Dewdney. *New Turing Omnibus*. W.H. Freeman & Company, 1993.

[5] C. Horn and O. Kerner. *Lehr- und Übungsbuch Informatik, Band 2: Theorie der Informatik*. Fachbuchverlag Leipzig, 2000. Eines der wenigen Bücher auf FH-Niveau, aber viele Fehler und unübersichtlich.

[6] H. Marxen. Busy beaver. `http://www.drb.insel.de/~heiner/BB/index.html`, 2001. 3.9.1

[7] U. Schöning. *Theoretische Informatik kurzgefasst*. Spektrum Akademischer Verlag, 1992. Gutes Buch, aber etwas zu theoretisch für die FH. 22

[8] R. Sedgewick. *Algorithmen*. Addison-Wesley, Bonn, 1995. Übersetzung d. engl. Originals, empfehlenswert.

[9] R. Socher. *Theoretische Grundlagen der Informatik*. Fachbuchverlag Leipzig, 2003. Gutes Buch auf FH-Niveau über formale Sprachen.

[10] P. Tittmann. *Graphentheorie*. Fachbuchverlag Leipzig, 2003. Sehr gutes Buch mit vielen Beispielen. Leider fehlen die Wegesuchalgorithmen.

[11] I. Wegener. *Kompendium Theoretische Informatik – eine Ideensammlung*. Teubner Verlag, 1996. Als Ergänzung zur Vorlesung und zum besseren Verständnis hervorragend geeignet. Ohne Formeln wird ein Überblick vermittelt.

[12] N. Wirth. *Algorithmen und Datenstrukturen*. Teubner-Verlag, Stuttgart, 1983 (3. Auflage). Ein Klassiker, vom Erfinder der Sprache PASCAL.