Burchard v. Braunmühl

Kleines Skript “Algorithmieren und Progammieren”

zur Vorlesung Informatik für Ingenieure Modul I (SS 01)

Neben dem Erlernen einer Progammiersprache gehört zum Programmieren auch die Fähigkeit Programme zu entwerfen. Hat man eine Aufgabe, die ein Rechner übernehmen soll, so muß man dem Rechner eine genaue Vorschrift übergeben, wie er diese Aufgabe lösen soll. Das bedeutet:

1. Wir müssen erst einmal selbst eine Lösung gefunden haben.
2. Darüber hinaus soll die Lösung effizient sein, d.h. der Rechner soll mit wenig Zeit und Speicher auskommen.
3. Die Lösung muß mit Hilfe der Programmiersprache formuliert werden.

Schritt 3 erfordert die Fähigkeit des Programmierens, d.h. Algorithmen in einer Programmiersprache auszudrücken. Das Erlernen einer Programmiersprache geht über das Kennenlernen der Elemente der Sprache hinaus. Es gilt in dieser Sprache komplizierte Sachverhalte zu formulieren, und umgekehrt die Möglichkeiten der Sprache beim Entwurf einer Lösung zu berücksichtigen.

Algorithmieren beschäftigt sich mit Schritt 1 und 2. Zu einem Problem, einer Aufgabe muß eine Lösungsverfahren gefunden werden, das von einem Rechner ausgeführt werden kann. Dazu muß man natürlich seinen Rechner und seine Programmiersprache kennen, denn das sind die Werkzeuge, mit denen wir arbeiten. Aber die Kenntnis der Werkzeuge allein liefert noch keine Lösung. Also geht es darum, Lösungsstrategien und algorithmische Methoden kennenzulernen, die einem beim Finden einer Lösung helfen können. Da es soviele verschiedene Aufgaben gibt, kann man freilich nicht erwarten, daß es für alle fertige Lösungen gibt, die man nur zu suchen hat.

Wie erlernt man das methodische Herangehen an ein Problem? Einmal kann man einfach viele exemplarische Lösungen kennenlernen und so ein Gefühl für die Art bekommen, wie solchen Lösungen gestrickt sind. Mit dem richtigen Instinkt für eine günstige Herangehensweise hat man eine gute Chance auch neue Aufgaben zu bewältigen. Noch besser ist, wenn sich für eine Klasse von Aufgaben schon eine Theorie der Methoden entwickelt hat. Dann ist man nicht auf den Instinkt angewiesen, sondern kann sich auf klare Strategien stützen. Methoden wollen aber verstanden sein. Reines Auswendiglernen von Methoden hilft wenig. Letztlich geht es um eine Schulung des algorithmischen Denkens. Geschult wird dieses Denken durch Training, in dem man häufig in diesen Bahnen denkt. Der Gegenstand hierbei hat exemplarischen Charakter. Er sollte das Denken herausfordern, aber er muß nicht notwendig im Berufsleben ein Rolle spielen. Wer weiß schon, was ihm in seinem späteren Berufsleben abgefordert werden wird.

Das Programmieren ist eine Sache der Übung. Es gibt zahllose Programmierbücher, die alle schnell veralten, weil sie sich in der Regel auf die aktuelle Version einer Programmiersprache beziehen. Dieses Skript ist dem Algorithmieren gewidmet. Wir wollen das Algorithmieren, das Entwerfen von algorithmischen Lösungsstrategien, anhand von zwei Aufgaben darstellen, die auch in der Praxis eine große Rolle spielen: dem Sortieren von vergleichbaren Elementen und dem Suchen von Worten in Texten. Beim Sortieren wollen wir verschiedene algorithmische Prinzipien kennenlernen und wie man den Aufwand eine Strategie berechnet, beim Suchen wollen wir zeigen, wie hilfreich eine Theorie sein kann, auch wenn sie nicht für diesen speziellen Zweck entwickelt worden ist.

1 Sortieren

Sortieren setzt voraus, daß die zu sortierenden Elemente vergleichbar sind, d.h. daß sie aus einer Menge stammen, der eine (lineare) Ordnung zugrunde liegt. Dies ist z.B. für die ganzen Zahlen (Integer) der Fall, die wir daher als Repräsentanten sortierbarer Elemente nehmen. Grundlegend für die hier vorgestellten Algorithmen ist, daß wir über die Elemente nichts weiter wissen, als daß sie aus einer Ordnung stammen und daß wir entscheiden können, welches Element im Sinne dieser Ordnung größer oder kleiner ist.

Es gibt auch Sortieralgorithmen, die Eigenschaften von Elementen nutzen, die wir hier nicht kennen. So kann man z.B. die Menge der Worte über einem Alphabet linear ordnen, so daß man sagen kann welches von zwei Worten größer ist, aber man weiß von einem Wort darüber hinaus, wie es aufgebaut ist. Daraus kann man Nutzen ziehen, wie es etwa Bucket-Sort tut. So kommt man zu schnelleren Verfahren, als es bei Sortieren nur durch Vergleich möglich ist. Man kann beweisen, daß kein Sortierverfahren, das nur Vergleiche nützt, mit weniger als $n \log(n)$ Vergleichen auskommt. Setzt man allerdings andere Techniken ein, so kann man den Aufwand unter diese Zeitschranke drücken.

Wir wollen Aussagen über den Aufwand dieser Verfahren machen. Dazu zählen wir die Anzahl der benötigten Vergleiche. Natürlich finden in einem Sortierprogramm auch andere Operationen statt, die Zeit oder Speicherplatz brauchen. Aber wir kennen weder den Rechner, noch die Programmiersprache, noch den Programmierer. Wir wissen nicht, wer unsere Verfahren implementiert, wie, in welcher Sprache und auf welcher Plattform. So können wir nur Aussagen machen, die umabhängig von diesen Umständen sind. Das ist auf jeden Fall die Zahl der benötigten Vergleiche. Wir können allerdings davon ausgehen, daß dies ein gutes Maß ist. Man sollte davon ausgehen, daß Vergleichen und Tauschen, die teuersten Operationen in einem solchen Programm sind. Man wird nicht öfter tauschen als vergleichen und ein gutes Programm sollte keinen sonstigen Aufwand treiben, der den Aufwand aller Vergleiche und Tauschoperationen übersteigt.

1.1 Die Exponentialfunktion und der Logarithmus

Zunächst ein paar Bemerkungen über Funktionen, die wir hier so viel verwenden: die Exponentialfunktionen und die Logarithmen.

Am häufigsten benützen wir die Exponentialfunktion 2^n zur Basis 2. Die ersten 11 Zweierpotenzen sind $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$. Es ist nützlich, diese auswendig zu kennen. Die Exponentialfunktion ist eine arithmetische Funktion, d.h. eine Funktion auf den natürlichen Zahlen ($2^n : \mathbb{N} \longrightarrow \mathbb{N}$). Gewohnter für uns ist die Exponentialfunktion 10^n,weil sie Grundlage unseres Dezimalsystems ist ($546 = 5 \cdot 10^2 + 4 \cdot 10^1 + 6 \cdot 10^0$). Da $2^{10} \sim 10^3$, kann man die Zweierpotenzen und Zehnerpotenzen durch Multiplikation-Division der Exponenten mit 3 grob miteinander vergleichen. Die Regel $x^a x^b = x^{a+b}$ sollte man kennen.

Bekannter ist die Exponentialfunktion e^n zur Basis e. Die Eulerkonstante $e \sim 2,72$ ist irrational, weil sie sich nicht als Bruch zweier ganzer Zahlen darstellen läßt und transzendent, weil sie nicht - wie die ebenfalls irrationale Zahl $\sqrt{2}$ - Nullstelle eines ganzzahligen Polynoms ist. Sie spielt in vielen Anwendungen wie den komplexen Zahlen, dem Wachstum von Populationen,radiaktivem Zerfall und der Zinsrechnung eine Rolle. e ist der Limes der Folge $\left(1 + \frac{1}{n}\right)^n$ für $n = 1, 2, 3, ...$ oder der Reihe $\sum_{n=1}^{\infty} \frac{1}{n!}$. Die Ableitung der reellen Funktion e^x ist selbst wieder e^x, d.h. die Funktion e^x hat an der Stelle x gerade die Steigung e^x. Die Reihe dieser Funktion ist $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.

Die Umkehrfunktion der Exponentialfunktion ist der Logarithmus, den es auch zu den verschiedenen Basen gibt. Der Logarithmus zur Basis e - wir schreiben ihn $\ln x$ - wird der 'natürliche Logarithmus' genannt, hat die Ableitung $\frac{1}{x}$ und wird durch die Reihe $\ln x = \sum_{i=1}^{\infty}(-1)^{i-1}\frac{(x-1)^i}{i}$ beschrieben. Es ist $e^{\ln x} = x = \ln(e^x)$. Den Logarithmus zur Basis b schreiben wir $\log_b x$. Zur Umrechnung der Basen gilt: $\log_a x = \frac{\log_b x}{\log_b a}$ bzw. $\log_a x = \log_b x \cdot \log_a b$. Es gilt $\log(ab) = \log a + \log b$ und $\log n^m = m \log n$.

Im Unterschied zur Analysis, wo der Logarithmus eine reelle Funktion ist, verwenden wir hier den Logarithmus als eine arithmetische Funktion. Der Logarithmus zur Basis b ist dann definiert durch

$$\log_b n = \max\left\{i \mid b^i \leq n\right\}$$

Kennt man schon den reellen Logarithmus, so kann man natürlich den Wert des arithmetischen auch definieren als die größte natürliche Zahl unter dem Wert des reellen Logarithmus, also den reellen Logarithmus abgerundet. Für den arithmetischen Logarithmus $\log_2 n$ zur Basis 2 schreiben wir auch einfach nur $\log n$.

Beispiel 1
$\log 1 = 0$, $\log 2 = 1$, $\log 3 = 1$, $\log 4 = 2$, $\log 5 = 2$, $\log 6 = 2$, $\log 7 = 2$, $\log 8 = 3$.

Beispiel 2
1. Die Länge der Binärdarstellung einer natürlichen Zahl n ist gerade $\log n+1$ (damit meinen wir $(\log n) + 1$, die 1-stellige Operation log geht vor $+$).
2. Die natürliche Zahl n kann man $\log n$-mal halbieren (unter weglassen des Restes) bis man auf 1 anlangt.

Notation 3
Die Exponentialfunktion wächst so schnell, daß Menschen sich intuitiv fast immer irren, auch wenn sie eigentlich über diese Funktion Bescheid wissen.

1. EinKalif versprach seinem Schachgegner, ihm im Falle seines Sieges jeden Wunsch zu erfüllen. Der wünschte sich ein Weizenkorn auf das erste Feld des Schachbretts und auf jedes weitere Feld die doppelte Menge von Körnern. Der Kalif versprach ihm diesen Wunsch leichten Herzens, bis er darüber aufgeklärt wurde, das diese letztlich $2^{64} - 1$ Körner eine Billionen Tonnen Weizen bedeuteten, eine Menge, die in der Geschichte der Menschheit bis dahin noch nicht geerntet worden war.
2. Nach einer Legende befindet sich im großen Tempel von Benares unter der Kuppel die Mitte der Welt. Dorthin legte Gott bei der Erschaffung der Welt eine Goldplatte mit drei diamantenen Nadeln, jede eine Elle hoch. Auf der ersten liegt ein Turm von 64 sich verjüngenden Scheiben aus purem Gold. Dieses ist Brahmas Turm. Seit Anfang der Zeiten sind die Priester des Tempels damit beschäftig, die goldenen Scheiben von der ersten Nadel auf die zweite zu legen. Dabei dürfen sie die dritte Nadel benutzen, aber unter keiner Scheibe darf eine kleinere liegen. Sobald der ganze Turm auf der zweiten Nadel liegt, wird der Tempel zu Staub zerfallen und die Welt mit einem Donnerschlag untergehen.
   Wenn die Priester jede Sekunde eine Platte bewegen, so benötigen sie 2^{64} Sekunden, das sind 584 Milliarden Jahre. Es mag wohl sein, daß nach dieser Zeit das Universum in sich zusammenstürzt, unser Sonnensystem allerdings wird 4 Milliarden Jahre nicht überleben.

1.2 Sortieren durch Aufsteigen: Bubble-Sort

Wir stellen nun zunächst zwei "dumme" Sortierverfahren vor (BubbleSort und MinSort), die im schlimmsten Fall (worst case) quadratischen Aufwand benötigen (Bei n Elementen können bis zu n^2 Vergleiche nötig sein.) Danach kommen intelligentere Verfahren, die alle in irgendeinem Sinn besser sind, die uns aber vor allem verschiedene Techniken demonstrieren, die auf dem Gebiet der "Effizienten Algorithmen" Bedeutung gewonnen haben.

Idee:

Man stelle sich den Array aus Integern vor als eine aufrecht gestellte Wassersäule, in der die Integer wie Luftblasen schweben. Die großen Luftblasen steigen hoch - durch die kleineren hindurch - und werden erst durch noch größere aufgehalten. Nach einiger Zeit werden sich die Luftblasen von selbst der Größer nach geordnet haben: die kleinen unten, die großen oben.

Beispiel 4
Eine Säule von 4 Integern. In drei Durchgängen wandert die 4, die 3, die schießlich die 2 an ihren Platz.

1	4	4	4
2	1	3	3
3	2	1	2
4	3	2	1

Beispiel 5
Eine Säule von 6 Integern wird 6-mal durchlaufen, wobei jede Blase soweit aufsteigt, bis sie unter einer größeren hängen bleibt. Jeder Durchgang benötigt 5 Vergleiche, 6 Durchgänge müssen wir machen. Im letzten Durchgang passiert zwar nichts mehr, aber er ist nötig, um sicherzustellen, daß wir fertig sind. Denn das Abbruchkriterium ist gerade, daß in einem Durchgang kein Wandern (Tauschen) mehr stattfindet. Es sind also im schlimmsten Fall $5 \cdot 6 = 30$ Vergleiche nötig. Der muß aber nicht eintreten, denn wenn die Folge schon in unserem Sinne sortiert ist, so genügt ein einziger Durchgang, also 5 Vergleiche, um festzustellen, daß wir fertig sind. Das ist der günstigste Fall (best case).

1	6	6	6	6	6
6	1	5	5	5	5
2	5	1	4	4	4
5	2	4	1	3	3
3	4	2	3	1	2
4	3	3	2	2	1

Nach dem i-ten Durchgang ist das i-t größte Element an seinem richtigen Platz

Aufwand:

Auf Grund der obigen Überlegungen benötigen wir, um n Elemente zu sortieren, höchstens n Durchgänge von je n Vergleichen, das sind insgesamt höchstens n^2 Vergleiche. Da nach dem i-ten Durchgang das i-t größte Element an seinem richtigen Platz steht, kann man freilich auch mithalten, daß man im i-ten Durchgang nur die $n - i$ unteren Felder überprüfen muß. Dann benötigen wir insgesamt höchstens $\sum_{i=1}^{n} i = 1 + \cdots + n = \frac{1}{2}\left(n^2 + n\right)$ Vergleiche.

Anmerkung 6
Die Summe der ersten n Zahlen ist $\sum_{i=1}^{n} i = 1 + \cdots + n = \frac{1}{2}\left(n^2+n\right)$. Denn:
ist n gerade, so ist

$$\begin{aligned} 1+\cdots+n &= (1+(n-1)) + (2+(n-2)) + \cdots + \left(\frac{n}{2} + \left(n - \frac{n}{2}\right)\right) + n - \frac{n}{2} \\ &= \frac{n}{2}\cdot n + n - \frac{n}{2} = \frac{1}{2}\left(n^2+n\right). \end{aligned}$$

ist n ungerade, so ist

$$\begin{aligned} 1+\cdots+n &= (1+(n-1)) + (2+(n-2)) + \cdots + \left(\frac{n-1}{2} + \left(n - \frac{n-1}{2}\right)\right) + n \\ &= \frac{n-1}{2}\cdot n + n = \frac{1}{2}\left(n^2+n\right). \end{aligned}$$

Programm ohne Prozedur:

```
program bubsort_1;

const n=?;
type arr=array[1..n] of integer;
var A:arr; temp,i:integer; nochmal:boolean;

begin
    repeat
        nochmal:=false;
        for i:=1 to n-1 do
            if A[i]>A[i+1] then begin
                temp:= A[i]; A[i]:=A[i+1]; A[i+1]:=temp;
                nochmal:=true;
            end;
    until nochmal=false;
end.
```

Programm mit Funktion

```
program bubsort_2;

const n=?;
type arr=array[1..n] of integer;
var a:arr; i:integer;

procedure tausch(i,j:integer);
var temp:  integer;
begin temp:= A[i]; A[i]:=A[j]; A[j]:=temp end;

function blubber:  boolean;
var i:integer;
begin
    result:=false;
    for i:=1 to n-1 do
        if A[i]>A[i+1] then begin tausch(i,i+1); result:=true end;
end;
```

```
begin
    while blubber do
end.
```

1.3 Sortieren durch Minimumsuche: Min-Sort

Idee:

Wir lassen einen Positionszeiger durch den Array wandern und setzen an die aktuelle Position jeweils das kleinste Element des Restarrays rechts von dieser Position. Um dieses kleinste Element zu finden, wandern wir mit einem zweiten Zeiger von der aktuellen Position bis zum Ende des Arrays und merken uns das jeweils kleinste bisher gefundene Element.

Beispiel:

6	1	5	2	4	3	5 Vergleiche
1	6	5	2	4	3	4 Vergleiche
1	2	5	6	4	3	3 Vergleiche
1	2	3	6	4	5	2 Vergleiche
1	2	3	4	6	5	1 Vergleiche
1	2	3	4	5	6	0 Vergleiche
						15 Vergleiche

Aufwand:

Offenbar müssen wir bei n Elementen $(n-1)$-mal das Minimum suchen. Beim ersten Mal kostet uns dies $n-1$ Verlgeiche, beim 2. Mal $n-2$, beim i-ten Mal $n-i$ Vergleiche. D.h. die Zahl der Vergleiche ist die Summe der Zahlen von 1 bis $n-1$, das ist gerade $\frac{1}{2}(n^2-n)$. Wir sehen, der Aufwand ist der gleiche wie bei Bubble-Sort. Aber er ist immer so hoch, denn selbst, wenn die Folge schon fertig sortiert vorliegt, würde unser Verfahren dies nicht feststellen, bis es alle Vergleiche durchgeführt hat.

Programm:

```
program minsort;

const n = ?;
type arr = array[1..n] of integer;
var A: arr; i,j:  integer;

procedure suchmin(anfang:integer; var minpos:integer);
var j,min:integer;
begin
    min:= A[anfang]; minpos:= anfang;
    for j:=anfang to n do
        if A[j]<min then begin min:=A[j]; minpos:=j end;
end;

procedure tausch(i,j:integer);
var temp:  integer;
begin temp:=A[i]; A[i]=:A[j]; A[j]:=temp end;
```

```
begin
   for i:=1 to n-1 do begin suchmin(i,j); tausch(i,j) end;
end.
```

1.4 Sortieren durch Mischen: Merge-Sort

Idee:

Dies ist nun eines der intelligenten Verfahren. Es basiert auf dem "Divide-and-Conquer"-Prinzip: man zerlegt das Originalproblem in zwei Probleme halber Größe, löst die beiden Teilprobleme und setzt dann aus den beiden Lösungen die Lösung des Originalproblems zusammen. Beim Lösen der Teilprobleme geht man genauso vor, so daß man ein rekursives Verfahren erhält.

Der Aufwand bei einem solchen Verfahren läßt sich besonders elegant durch eine Rekursionsgleichung beschreiben: Hat das Problem die Größe n und ist n eine Zweierpotenz, läßt sich unser Problem in der Größe also immer genau halbieren, so ist der Aufwand unseres Problems im worst case beschreibbar durch

$$T(n) = \begin{cases} a & \text{falls } n = 1 \\ 2T(\frac{n}{2}) + cn & \text{falls } n > 1 \end{cases},$$

wobei a der Aufwand für ein Problem der Größe 1 ist, und cn der Aufwand, der nötig ist, um aus den beiden Lösungen der Teilprobleme der Größe $\frac{n}{2}$ die Lösung des Originalproblems der Größe n herzustellen. Zunächst sei n eine Zweierpotenz ($n = 2^k$). Dann ist

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + cn \\ &= 2(2T(\tfrac{n}{4}) + c\tfrac{n}{2}) + cn \\ &= 4T(\tfrac{n}{4}) + 2cn \\ &= 8T(\tfrac{n}{8}) + 3cn \\ &= \ldots\ldots\ldots\ldots\ldots\ldots \\ &= 2^k T(\tfrac{n}{2^k}) + kcn \\ &= an + cn \log n. \end{aligned}$$

Ist $2^k < n \leq 2^{k+1}$, so ist $T(2^k) \leq T(n) \leq T(2^{k+1})$ und somit $dn \log n \leq T(n) \leq en \log n$ für gewisse Konstante d und e.

Beim MergeSort teilt man den Array von n Elementen in 2 Teilarrays von $n/2$ Elementen, sortiert diese und mischt (engl. to merge) nun die Elemente der sortierten Teilarrays so ineinander, daß die Elemente sortiert bleiben. Dies geschieht, indem man die jeweils ersten Elemente der beiden Teilarrays vergleicht, das kleinere der beiden löscht und in den großen Array schreibt. Dies wiederholt man, bis einer der Teilarrays leer ist. Dann kann man den Rest des anderen (sortierten) Teilarrays einfach in den großen Array abschreiben. Man sieht, daß der Aufwand beim Zusammensetzen der Teillösungen bedeutsam ist: er erfordert bis zu $n-1$Vergleiche, wenn der resultierende Array die Länge $n$hat.

Im Fall des Merge-Sort sähe unsere Rekursionsgleichung also wie folgt aus:

$$T(n) = \begin{cases} 1 & \text{für } n = 2 \text{ (sortieren)} \\ 2\,T\left(\frac{n}{2}\right) + n - 1 & \text{sonst (mischen)} \end{cases}$$

Sei $n = 2^{k+1}$.Dann ist

$$\begin{aligned}
T(n) &= 2\,T\left(\tfrac{n}{2}\right) + n - 1 \\
&= 2\left(2\,T\left(\tfrac{n}{4}\right) + \tfrac{n}{2} - 1\right) + n - 1 \\
&= 4\,T\left(\tfrac{n}{4}\right) - 2n - 3 \\
&= 4\left(2\,T\left(\tfrac{n}{8}\right) + \tfrac{n}{4} - 1\right) + 2n - 3 \\
&= 8\,T\left(\tfrac{n}{8}\right) + 3n - 7 \\
&\vdots \\
&= 2^k\,T\left(\tfrac{n}{2^k}\right) + k \cdot n - \left(2^k - 1\right) \\
&= \tfrac{n}{2} \cdot 1 + (\log n - 1) \cdot n - \tfrac{n}{2} + 1 \\
&= n\,(\log n - 1) + 1
\end{aligned}$$

Beispiel:

Wir beginnen mit einem Array der Größe 16. Der wird halbiert in zwei der Größe 8, diese wieder in 4 der Größe 4 und diese in 8 der Größe 2. Diese kleinen Arrays sortieren wir, und nun mergen wir nur noch die sortierten Arrays zu neuen sortierten Arrays doppelter Größe und diese wieder zu sortierten Arrays doppelter Größe usw, bis wir wieder einen Array der Größe 16 haben.

16								teile		
8				8				teile		
4		4		4		4		teile		
2	2	2	2	2	2	2	2	sortiere	$8\,(2-1)=8$	Vgle
2	2	2	2	2	2	2	2	mische	$4\,(4-1)=12$	Vgle
4		4		4		4		mische	$2\,(8-1)=14$	Vgle
8				8				mische	$1\,(16-1)=15$	Vgle
16										
								insges	$4 \cdot 16 - (16-1) = 49$	Vgle

Aufwand:

Man erkennt die Verallgemeinerung: die Zahl der Vergleiche bei n Elementen ist offenbar: ($k = \log n$)

$\sum_{i=0}^{k-1} 2^i \left(2^{k-i} - 1\right) = \sum_{i=0}^{k-1} 2^k - 2^i = \sum_{i=0}^{k-1} 2^k - \sum_{i=0}^{k-1} 2^i = k2^k - \left(2^k - 1\right) = n(\log n - 1) + 1$

Um uns klar zu machen, wie groß der Fortschritt von n^2 zu $n \log n$ ist, betrachten wir die Zahl $n = 2^{30}$, was etwa ein Milliarde ist, dann ist $n^2 = 2^{60}$, also etwa eine Milliarde Milliarden (wir nennen das im Deutschen eine Trillion), aber $n \log n = 30 \cdot 2^{30}$, also nur etwa 30 Milliarden. Der Unterschied zwischen 30 und einer Milliarde wird jedem auffallen. Einer, der nur 30 DM besitzt, wird glauben, daß er nichts besitzt, wenn er mit einem Milliardär spricht. Entsprechend groß ist der Unterschied zwischen 30 Milliarden und einer Milliarde Milliarden. Die Funktion $n \log n$ wächst nur wenig mehr als n selbst, n^2 dagegen hat ein Wachstum, das schon die Grenzen unserer Vorstellung erreicht.

Programm

```
program mergesort;}
```

```
const c=?;
type arr=array[1..c+1] of integer;
```

{wir verlängern den Array um ein Feld, in das wir die größte darstellbare Zahl schreiben, ein Trick, der uns die Implementierung etwas abkürzen soll}

```
var v,temp:arr; i:integer;
{wir verwenden einen temporären Array temp, in den wir die Teilarrays hineinmergen}

procedure merge(i1,i2,j1,j2:integer);
{i1, i1 sind die Grenzen des ersten Arrays, j1, j2, die des zweiten}
var i,j,k,l:integer;
begin
    i:=i1; j:=j1; l:=j2-i1+1;
    for k:=1 to l do
        if v[i]<v[j]
            then begin temp[k]:=v[i]; if i<i2 then i:=i+1 else i:=c+1; end
            else begin temp[k]:=v[j]; if j<j2 then j:=j+1 else j:=c+1; end;
            {wenn i auf das Ende des Teilarrays stößt, dann setzen wir i auf Feld c+1, wo die
            größte denkbare Zahl steht. Dann ist v[i] immer größer v[j], d.h. der Rest des
            Teilarrays, in dem j steht wird in temp geschrieben.}
    for k:=1 to l do v[i1+k-1]:=temp[k];
end;

procedure msort(a,e:integer);
var a1,a2,e1,e2,t:integer;
begin
    if e-a>1
        then begin
            a1:=a; e1:=a + (e-a) div 2; a2:=e1+1; e2:=e;
            msort(a1,e1); msort(a2,e2);
            merge(a1,e1,a2,e2);
        end
        else if v[a]>v[e] then begin t:=v[a]; v[a]:=v[e]; v[e]:=t end;
end;

begin
    v[c+1]:=32767; msort(1,c);
end.
```

Wie wir gesehen haben, ist Merge-Sort bedeutend schneller als die beiden obigen Verfahren, hat aber den Nachteil, daß es einen temporären zweiten Array, also den doppelten Speicherplatz braucht.

Lochkartenmaschinen

Merge-Sort hatte übrigens früher eine weitere praktische Bedeutung. In der Zeit, als man noch Lochkarten benützte, um seine Programme zu schreiben und dann als Job beim Operator abgab, damit der das Programm auf dem "Großrechner" rechnen ließ, benützte man Maschinen, die große Lochkartenstapel durchblättern, lesen und mischen konnten. Hier wurde Merge-Sort benützt, um einen Stapel Lochkarten zu sortieren. Dies ging auf folgende Weise (n sei wieder eine Zweierpotenz):

```
for i = 1 to log n do begin
    halbiere Stapel 1;
    lege die untere Hälfte von Stapel 1 in Stapel 2 und die obere in Stapel 3;
    while Stapel 2 und Stapel 3 nicht leer do begin
        mische die jeweils 2^i untersten Karten von Stapel 2 und Stapel 3
```

```
        geordnet in Stapel 1
```
{entnehme immer die kleinere der beiden untersten Karten von Stapel 2 und 3 und legen sie
in Stapel 3, bis in einem der Stapel 2^i Karten entnommen sind. Dann lege den noch nicht
entnommenen Rest der 2^i Karten des anderen Stapels in Stapel 3}
```
    end;
end.
```

Nach dem i-ten Durchlauf der Schleife liegen in Stapel 1 sortierte Blöcke der Länge 2^i. $\log n$-mal mußte der Operator den Stapel in der Maschine teilen und umsetzen, die die beiden Teilstapel dann durchblätterte und wieder zu einem mischte. Das ging aber noch, denn bei einem Stapel von 1000 Karten, mußte er dies 10-mal tun, und größere Stapel hatte man kaum.

Beispiel:

1			2			5			8
3			1			4			7
5			7			2			6
8			3			1			5
2	2	1	5	5	2	8	8	5	4
7	7	3	4	4	1	7	7	4	3
4	4	5	8	8	7	6	6	2	2
6	6	8	6	6	3	3	3	1	1
1	2	3	1	2	3	1	2	3	1

1.5 Sortieren durch Pivotieren: Quicksort

Idee:

Dies ist ebenfalls eine Divide-and-Conquer-Strategie. Aber im Gegensatz zu Merge-Sort ist nicht garantiert, daß wir das Problem halbieren. Darum ist der Aufwand im schlechtesten Fall (worst case) auch proportional zu n^2 und nicht zu $n \log n$. Allerdings ist der Aufwand im Durchschnitt (average case) proportional zu $n \log n$ und in zahlreichen Meßreihen schnitt Quicksort besser ab als Merge-Sort. Außerdem benötigt Merge-Sort einen temporären zweiten Array, Quicksort aber nicht. Damit ist auch der Speicherbedarf von Quicksort geringer.

Man wählt unter den Elementen des zu sortierenden Arrays irgendein Element p - ein Pivotelement - das dazu dient, die übrigen Elemente des Arrays aufzuteilen in solche, die kleiner oder gleich p sind und solche, die größer p sind. Das günstigste Element wäre der Median, d.h. das Element, für das es ebenso viele kleinere, wie größere Elemente im Array gibt, das also unsere Arrayelemente in zwei gleich große Teilmengen aufteilt. Da wir aber nicht wissen, welches Element der Median ist, können wir auch jedes beliebige Element nehmen, etwa das erste im Array. Wie gut es ist, hängt ganz von der zu sortierenden Folge ab. Es sei jedoch darauf hingewiesen, daß es sich lohnt, sich etwas mehr Mühe mit dem Pivotelement zu geben. So erziehlt man bessere Ergebnisse, wenn man als Pivotelement den Median der ersten 5 Elemente des Arrays wählt. Der Mehraufwand ist gering im Vergleich zum Gewinn.

Hat man den Array umgeschrieben in 2 Arrays - der eine enthält die Elemente kleiner oder gleich p (aber nicht das Pivotelement selbst), der andere die größer p - so sortiert man diese Teilarrays und legt sie aneinander: erst der Array mit den kleineren Elementen, dann das Pivotelement

selbst und schließlich den zweiten Array. Dann haben wir einen sortierten Array vorliegen. Hier braucht man also nicht zu mischen, aber dafür ist das Teilen aufwendig: man braucht $n - 1$ Vergleiche um einen Array der Länge n zu teilen.

Das Sortieren der Teilarrays geht genauso, so daß wir wieder ein rekursives Verfahren haben. Allerdings benützen wir keinen temporären Array um diese Teilung vorzunehmen, sondern wir vertauschen die Elemente in unserem Array so, daß zuerst die Elemente kleiner oder gleich p kommen, dann p selbst und dann die übrigen. Dafür benützen wir 2 Zeiger. Ein Zeiger L läuft auf dem Array von links nach rechts und sucht das erste Element, das größer p ist. Bis zu dieser Stelle finden wir nur Elemente kleiner oder gleich p vor, die in der richtigen linken Hälfte des Arrays liegen und nicht bewegt werden müssen. Genauso läuft eine Zeiger R von rechts nach links über den Array und sucht das erste Element, das kleiner oder gleich p ist. Auch hier gilt: rechts von der gefundenen Position liegen nur Elemente größer p, die somit schon in der richtigen Hälfte des Arrays liegen. Das Element unter L aber ist größer p und sollte nicht links von dem Element unter R liegen, das ja kleiner oder gleich p ist. Also tauschen wir diese beiden Elemente. Danach laufen die Zeiger L und R weiter aufeinander zu, bis wieder ein Tausch fällig wird oder sie sich kreuzen. Nach dem Kreuzen der beiden Zeiger sind wir mit dem Teilen fertig (unter R liegt ein Element kleiner oder gleich p). Was bleibt, ist das Pivotelement zwischen die beiden Teilfolgen zu schieben. Dies geschieht, indem wir das Pivotelement, d.h. das erste Element des Arrays mit dem unter R liegenden Element tauschen.

Dasselbe wird nun mit den gewonnenen Teilarrays wiederholt, bis die Teilarrays eine Länge kleiner oder gleich 2 haben. Diese kleinen Arrays sortieren wir, wenn nötig durch einen Vergleich. Dann sind wir fertig, denn alle Teilarrays liegen schon in der richtigen Folge hintereinander.

Programm

```
program quicksort;

const n = ?;
var a:  = array[1..n] of integer;
i,j :  integer;

procedure tausch(i,j:  integer);
var temp:  integer;
begin
    temp := a[i];
    a[i] := a[j];
    a[j] := temp;
end;

procedure pivot(i,j:integer; var m:integer);
    {i,j sind die Arraygrenzen, m das Feld, das den Array teilt}
var p,L,R: integer;
    {p Pivotelement; Zeiger L sucht Element > p, Zeiger R sucht Elelement ≤ p}
begin
    p:= a[i]; L:= i; R:= j+1;
    repeat L:= L+1 until (a[L] >p ) or (L >= j); {L wandert nach rechts}
    repeat R:= R-1 until a[R] <= p; {R wandert nach links}
    while L <R do begin {solange sich die Zeiger nicht kreuzen}
        tausch(L,R);
        repeat L:= L+1 until a[L] >p;
```

```
        {L wandert nach rechts bis zum nächsten Element > p. Das gibt es wegen tausch(L,R);}
        repeat R:= R-1 until a[R] <= p; {R wandert nach links}
    end;
    m:= R; {m ist das trennende Feld, die "Mitte" }
    tausch(i,m); {das Pivotelement kommt zwischen die beiden Teilarrays}
end;

procedure quicksort(i,j:integer);
var m:  integer;
begin
    if j-i = 1 then if a[i] >a[j] then tausch(i,j);
    if j-i >1 then begin
        pivot(i,j,m);
        quicksort(i,m-1);
        quicksort(m+1,j);
    end;
    {wenn j-i = 1 oder -1, dann ist der Array schon sortiert}
end;

begin
    quicksort(1,n);
end.
```

Aufwand

Man sieht schnell, daß der Aufwand etwa für einen Array mit umgekehrt sortierter Folge quadratisch in der Länge des Arrays ist: Das Pivotelement ist immer entweder das kleinste oder das größte Element des Arrays. Daher wird der Array überhaupt nicht geteilt, er wird nur kleiner, weil wir das Pivotelement herausnehmen. Es hat sich aber herausgestellt, daß dieses Verfahren dennoch im Durchschnitt sehr gut ist, besser als die anderen bekannten Sortieralgorithmen, die nur mit Vergleichen arbeiten.

Average case: Wenn wir annehmen, daß alle Ordnungen der zu sortierenden Folgen gleichwahrscheinlich sind, so können wir die Aufwände bei allen n möglichen Unterteilungen des Arrays durch das Pivotelement aufsummieren und durch n teilen, um einen mittleren Aufwand zu erhalten. So erhalten wir folgende Rekursionsformel (das erste n steht für die n Vergleiche, die wir beim Teilen des Arrays durch das Pivotelement benötigen):

$$t(0) = t(1) = 0 \text{ (hier ist nichts zu sortieren)}$$
$$t(n) \leq n + \frac{1}{n}\sum_{k=1}^{n}(t(k-1) + t(n-k)) = n + \frac{2}{n}\sum_{k=0}^{n-1} t(k) \text{ für } n \geq 2,$$

d.h. der Wert von $t(n)$ wird durch die Werte $t(k)$ für Argumente $k < n$ beschränkt. Wir behaupten, daß dann $t(n) \leq cn\log n$ sein muß, wenn nur $c \geq \frac{2}{\log e} \approx 1,4$.

Dazu folgende Fakten:

$$\log n = \ln n \cdot \log e$$

und

$$\int_{x=0}^{n} x \ln x \,\mathrm{d}\, x = \frac{n^2}{2}\left(\ln n - \frac{1}{2}\right)$$

Wir beweisen obige Behauptung durch Induktion über n.

1. Für $n < 2$ gilt die Behauptung. Ebenso gilt
2. für $n = 2$:
$$t(2) \leq 2 + \frac{2}{2}\left(t(0) + t(1)\right) \leq 2 \leq 2c\log 2.$$
3. Wir nehmen an, daß die Behauptung richtig sei für alle $k < n$.
4. Dann ist
$$\sum_{k=0}^{n-1} t(k) \leq c\sum_{k=0}^{n-1} k\log k \leq c\log e\sum_{k=0}^{n} k\ln k \leq c\log e\int_{x=0}^{n} x\ln x\,\mathrm{d}\,x$$
$$= c\log e\left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) = c\frac{n^2}{2}\log n - \frac{c\log e}{4}n^2$$
Also ist
$$t(n) \leq n + cn\log n - \frac{1}{2}cn\log e \leq cn\log n,\ \text{da}\ \frac{c\log e}{2} > 1.$$

1.6 Baum

Definition 7 (Digraph)
Sei Q eine Menge (die Menge der Knoten) und $K \subseteq Q \times Q$ (die Menge der Kanten). Dann heißt die Struktur $G = (Q, K)$ gerichteter Graph oder Digraph.

Ist (p, q) eine Kante aus K, so nennen wir p *direkten Vorgänger* von q und q *direkten Nachfolger* von p. Eine Folge benachbarter Kanten, genauer $\pi = (q_1\, q_2)\,(q_2\, q_3)\ \ldots\ (q_n\, q_{n+1})$ mit $(q_i\, q_{i+1}) \in K\ \ (i = 1, \ldots, n)$ heißt *Pfad der Länge* n von Knoten q_1 nach Knoten q_{n+1}. (Der Pfad der Länge 0 führt von einem Knoten p zu ihm selbst). p heißt *Vorgänger* von q und q *Nachfolger* von p wenn es einen Pfad von p nach q gibt (ein Knoten ist immer sein eigener Vorgänger und Nachfolger wegen des Pfades der Länge 0).

Man kann sowohl den Knoten, als auch den Kanten Objekte (vom Typ integer, character, string, boolean etc.) zuordnen. Man kann sich vorstellen, daß die Knoten - wie beim Array - Felder sind, in die diese Objekte eingeschrieben werden können. Die Kanten können ebenfalls mit Objekten (z.B. Zeichen) belegt werden (die man sich an die Kanten geschrieben denkt). Formal hat der Graph dann noch ein Knotenbelegungsfunktion $f_Q : Q \longrightarrow A$, wenn A die Menge der Objekte ist, die man in die Knoten schreiben kann, und eine Kantenbelegungsfunktion $f_K : K \longrightarrow B$, wenn B die Menge der Objekte ist, die man an die Kanten schreiben kann. Ist q ein Knoten, so ist $f_Q\,(q)$ das in den Knoten eingeschriebene Objekt - wir können es auch den Inhalt des Knoten q nennen.

Beispiel 8
$G = (\{1, 2, 3, 4\}, \{(1, 2), (1, 3), (1, 4), (2, 3)\})$

Definition 9 (Baum)
Ein Digraph ist ein *Baum* gdw. gilt:

1. es existiert genau ein Knoten ohne Vorgänger (wir nennen ihn die Wurzel des Baums),
2. jeder Knoten ist von der Wurzel aus durch einen und nur einen Pfad erreichbar.

Ein paar Begriffe zum Baum:

Vater	=	direkter Vorgänger	*Vorfahre*	=	Vorgänger
Sohn	=	direkter Nachfolger	*Nachfahre*	=	Nachfolger
Brüder		haben denselben Vater	*Blatt*	=	Knoten ohne Sohn

Tiefe eines Knotens q	=	Länge des Pfades von der Wurzel zu q
Höhe des Baumes	=	Länge des längsten Pfades (= maximale Tiefe eines Knotens)
Ebene i des Baums	=	die Menge der Knoten der Tiefe i

Anmerkung 10
Wir können uns einen Baum so gezeichnet denken, daß die Knoten derselben Tiefe auf einer Ebene stehen.

Definition 11 (orientiert)
Ein Baum heißt *orientiert,* wenn die Söhne jedes Knotens linear geordnet sind ($\leq_B$) (im Bild meistens von links nach rechts).

Definition 12 (Knoten-Nummerierung)
In einem orientierten Baum kann man auf mehrere Weisen eine Ordnung der Knoten definieren.

Depth First:	$a \leq_{DF} b$	gdw.	1)	a Vorfahre von b oder
			2)	es existiert Vorfahre a' von a und b' von b mit $a' \leq_B b'$.
Breadth First:	$a \leq_{BF} b$	gdw.	1)	Tiefe $(a) <$ Tiefe (b) oder
			2)	Tiefe $(a) =$ Tiefe (b) und es existieren Vorfahren a' von a und b' von b mit $a' \leq_B b'$.

Breiten-Nummerierung eines Baums: Die Nummerierung seiner Knoten in ihrer Breadth-First Reihenfolge (die Knoten derselben Tiefe von links nach rechts und die Knoten der Tiefe i vor denen der Tiefe $i+1$).

Tiefen-Nummerierung eines (endlichen) Baumes: Die Nummerierung seiner Knoten in ihrer Depth-First Reihenfolge. Dabei läuft man die Pfade des Baumes ab, so weit links wie möglich und so tief wie möglich:

```
procedure num(p);
begin
    nummeriere p (d.h.  gib ihm die kleinste noch nicht vergebene Nummer);
    for all Söhne q von p in ihrer Reihenfolge do num(q);
end.
```

oder wer es lieber iterativ formulieren will:

```
nummeriere die Wurzel;
if ein Sohn existiert then begin
    gehe zum kleinsten Sohn;
    repeat
        gehe zum Vater;
        while nicht nummerierter Sohn existiert do begin
            gehe zum kleinstem nicht nummerierten Sohn und nummeriere ihn;
        end;
    until Vater existiert nicht;
end.
```

Definition 13 (binärer Baum)
Ein Baum ist *binär*, wenn jeder Knoten höchstens 2 Söhne hat.

Definition 14 (vollständiger binärer Baum)
Ein binärer Baum der Höhe h heißt *vollständig*, wenn alle Knoten einer Tiefe $< h$ genau 2 Söhne haben.

Definition 15 (fast vollständiger binärer Baum)
Ein orientierter binärer Baum der Höhe h heißt *fast vollständig*, wenn gilt:

1. alle Knoten einer Tiefe $< h-1$ haben genau 2 Söhne,
2. hat ein Knoten p der Tiefe $h-1$ weniger als 2 Söhne, so haben alle Knoten q mit $q >_{BF} p$ keinen Sohn.

Lemma 16
Ein binärer Baum der Höhe h hat höchstens $2^{h+1}-1$ Knoten.

Beweis: Unter den binären Bäumen der Höhe h hat der vollständige die meisten Knoten. Der hat 2^i Knoten der Tiefe i, also insgesamt $2^0+2^1+2^2+2^3+...+2^h = 2^{h+1}-1$ Knoten. ■

Anmerkung 17

1. Die Zahl der Blätter eines vollständigen binären Baum ist um 1 größer als die Zahl der übrigen Knoten.
2. Ist B ein fast vollständiger binärer Baum mit n Knoten und Höhe h, so gilt

$$\begin{aligned} 2^h &\leq n < 2^{h+1} \\ h &\leq \log n < h+1 \\ h &= \lfloor \log n \rfloor \end{aligned}$$

d.h. ein fast vollständig binärer Baum der Größe n hat die Höhe $\lfloor \log n \rfloor$.

Die Datenstruktur Array besteht aus einer Folge aneinanderliegender Felder, die benannt werden durch die Zahlen $1, ..., n$. Eine solche Zahl i, die ein Feld des Arrays bezeichnet, nennen wir die *Adresse* des Feldes.

Die *Datenstruktur binärer Baum* besteht aus Feldern, die in Art eines binären Baumes zusammenhängen; d.h. zu einem Feld, gibt es in der Regel das Vaterfeld und die Sohnfelder. Als Adresse eines Baumfeldes kann man seine Nummer in der Breiten-Nummerierung (Breadth-First) wählen. Betrachten wir in dieser Nummerierung das i-te Baumfeld, so ist der Vater das (i div 2)-te Feld, der linke Sohn das 2i-te Feld und der rechte Sohn das (2i+1)-te Feld.

Wie die Felder des Arrays haben auch die Felder eines Baums Inhalte (formal: den Knoten (Feldern) des Baums sind Objekte, Werte zugeordnet. Ist B ein Baum, so schreiben wir - wie beim Array - $B[i]$ für den Inhalt des i-ten Feldes. Uns interessieren hier Inhalte, die man ordnen, sortieren kann, die man paarweise vergleichen kann, für die also paarweise $\leq$ oder $\geq$ gilt (z.B. Integer).

Definition 18 (Array-Repräsentation eines fast vollständigen binären Baums)
Fassen wir die Breiten-Nummern der Felder eines binären vollständigen Baums der Höhe h als Feldnummern eines Arrays auf, so erhalten wir einen Array der Länge $2^{h+1}-1$, der unseren Baum repäsentiert. In diesem Array stehen die Felder in der Reihenfolge ihrer Breitennummerierung nebeneinander (d.h. der Baum ist ebenenweise in den Array geschrieben). Ist der binäre Baum nur fast vollständig, so ist die letzte Ebene nicht vollständig, d.h. der Array hat eine Länge n mit $2^h \leq n < 2^{h+1}$.

Beispiel 19
Betrachte den binären vollständigen Baum

$$B = (\{1,2,3,4,5,6,7\}, \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}).$$

Die Benennung der Knoten entspreche ihrer Breitennummerierung. Die Belegung der Felder sei

$$B[1] = 40,\ B[2] = 80,\ B[3] = 35,\ B[4] = 90,\ B[5] = 45,\ B[6] = 50 \text{ und } B[7] = 70.$$

Dargestellt als Array ist das

$$B = (40, 80, 35, 90, 45, 50, 70)\,.$$

Anmerkung 20
Obwohl wir dies hier nicht benützen, bemerken wir doch, daß man binäre Bäume auch dann durch einen Array repräsentieren kann, wenn sie nicht fast vollständig sind, d.h. wenn es Knoten gibt, die nur einen Sohn haben, indem man den Baum mit dummy-Feldern vervollständigt, die keinen Inhalt bekommen: B_1 Wurzelfeld, B_{2i} = dummy-Feld, falls B_i keine Söhne hat oder ein dummy-Feld ist und B_{2i} = erster Sohn von B_i sonst. B_{2i+1} = dummy-Feld, falls B_i nicht zwei Söhne hat oder ein dummy-Feld ist und B_{2i+1} = zweiter Sohn von B_i sonst.

Beispiel 21
Baum $B_1 = (\{A, B, C, D\}, \{(A, B), (B, C), (C, D)\})$. Da alle Knoten höchstens einen Sohn haben, ist die Breitennummerierung klar. Die Inhalte der Knoten seien gerade ihrer Breitennummern. Dann ist die Arrayrepräsentation:
Array $B_1 = (1, 2, -, 4, -, -, -, 8, -, -, -, -, -, -, -)$

Baum $B_2 = (\{A, B, C, D, E, F, G\}, \{(A, B), (A, C), (B, D), (C, E), (E, F), (E, G)\})$. Es sei $B \leq_B C$ und $F \leq_B G$. Dann ist die Breitennummerierung festgelegt. Wieder seien die Inhalte der Knoten ihre Breitennummeren. Dann ist die Arrayrepräsentation:
Array $B_2 = (1, 2, 3, 4, -, 5, -, -, -, -, -, 6, 7, -, -)$

Anmerkung 22
Man sollte sich klarmachen, daß es bei fast vollständigen binären Bäumen in der Notation der Felder keinen Unterschied macht, ob wir uns die Felder in einer Baumstruktur oder in einer Arraystruktur verknüpft denken. Das i-te Feld hat in beiden Fällen denselben Namen i. Der Unterschied liegt in unserer anschaulichen Vorstellung und daß wir "Vater" von Feld i sagen statt Feld $(i \operatorname{div} 2)$, "linker Sohn" statt Feld $2i$ und "rechter Sohn" statt Feld $(2i + 1)$. Man kann sich jeden Array als fast vollständigen binären Baum von Feldern vorstellen und umgekehrt. Wir wollen daher auch bei einem Array von Wurzel, Blatt, Vater eines Feldes und Sohn eines Feldes reden.

Definition 23 (Heap, Fast-Heap)
Ein Array B ist ein *Heap*, wenn der Inhalt eines Feldes immer größergleich den Inhalten seiner Söhne ist, soweit es sie gibt. Für einen Heap B gilt also: $B[i] \geq B[2i]$ wenn Feld i einen Sohn besitzt und außerdem $B[i] \geq B[2i + 1]$, wenn Feld i zwei Söhne besitzt. Ein Array B ist ein *Fast-Heap*, wenn die Heapbedingung für alle Knoten außer der Wurzel gilt.

1.7 Heap-Sort

1.7.1 Die Idee des Heap-Sort

1. Wir machen aus dem zu sortierenden Array erst einmal einen Heap.
2. Dann nehmen wir den Inhalt der Wurzel heraus, der der größte Inhalt des Arrays überhaupt ist. In das leere Wurzelfeld schreiben wir jetzt den Inhalt des rechtesten Blattfeldes (letzten Arrayfeldes), das wir damit löschen. Das Ergebnis ist kein Heap mehr, weil der Inhalt der Wurzel möglicherweise kleiner als der Inhalt eines der Söhne ist. Da das der einzige Fehler ist, ist dies ein Fast-Heap.
3. Aus diesem Fast-Heap machen wir wieder einen richtigen Heap und beginnen wieder mit 2.

Dies machen wir solange, bis der Array nur noch die Länge 2 hat. Die nach und nach herausgenommenen Inhalte schreiben wir in dieser Reihenfolge von rechts nach links in die freigewordenen Blattfelder, so daß wir die gewüschte Ordnung bekommen.

Wir müssen also erklären, wie man aus einem beliebig gefüllten Array und wie man aus einem Fast-Heap einen Heap macht.

1.7.2 Mache aus einem Baum einen Heap

Methode: Wir gehen die Baumfelder in ihrer Breiten-Nummerierung durch und schieben den Inhalt im Baum solange in Richtung Wurzel hoch, bis das Vaterfeld einen größergleichen Inhalt hat oder wir die Wurzel erreicht haben.

Pidgin-Algorithmus:

```
procedure fitin(i)
begin
    j = i;
    while (j ≠ 1 und Inhalt von Vater von Feld j kleiner als Inhalt von Feld j) do
    begin
        tausche die Inhalte von Feld j und Vater von Feld j;
        j := Nummer von Vater(j);
    end;
end;

for i = 2 to n do fitin(i).
```

Beispiel 24

$$A = (40, \mathbf{80}, 35, 90, 45, 50, 70)$$
$$A = (80, 40, \mathbf{35}, \mathbf{90}, 45, 50, 70)$$
$$A = (90, 80, 35, 40, \mathbf{45}, \mathbf{50}, 70)$$
$$A = (90, 80, 50, 40, 45, 35, \mathbf{70})$$
$$A = (90, 80, 70, 40, 45, 35, 50)$$

Laufzeit (worste case):

Der Baum hat die $\log n$ Ebenen 0, 1, ..., $\log n - 1$. In der Ebene i liegen 2^i Knoten, deren Werte

höchstens i-mal nach oben wandern können, das gibt insgesamt höchstens

$$\sum_{i=0}^{\log n-1} i2^i \leq \log n \cdot 2^{\log n} = n \cdot \log n$$

Anhebungen bzw. Vertauschungen bzw. Vergleiche. [Der *average case* ist *const* $\cdot\, n$, also linear.]

Anmerkung 25
Zur Summe $\sum_{i=0}^{\log n-1} i \cdot 2^i = (n-1) \cdot 2^{n+1} + 2$

Es gilt

1. $2^1 + \cdots\cdots\cdots + 2^n = 2^{n+1} - 2$
2. $2^i + \cdots\cdots\cdots + 2^n = 2^{n+1} - 2 - \left(2^i - 2\right) = 2^{n+1} - 2^i$

Somit ist $\sum_{i=0}^{\log n-1} i \cdot 2^i =$

$$\begin{array}{cccccccccll}
2^1 & +2^2 & +2^3 & +2^4 & \dots & +2^i & +\cdots & +2^n & = & 2^{n+1} - 2^1 & \text{(Summe Zeile 1)} \\
 & +2^2 & +2^3 & +2^4 & & & +\cdots & +2^n & = & 2^{n+1} - 2^2 & \text{(Summe Zeile 2)} \\
 & & +2^3 & +2^4 & & \vdots & & \vdots & & \vdots & \\
 & & & +2^4 & & \vdots & & \vdots & & \vdots & \\
 & & & & & \vdots & & \vdots & & \vdots & \\
 & & & & & +2^i & & +2^n & = & 2^{n+1} - 2^i & \text{(Summe Zeile } i\text{)} \\
 & & & & & & & \vdots & & \vdots & \\
 & & & & & & & +2^n & = & 2^{n+1} - 2^n & \text{(Summe Zeile } n\text{)} \\
 & & & & & & & & \text{insgesamt} & n\,2^{n+1} - \left(2^{n+1} - 2\right) & \\
 & & & & & & & & & = (n-1)\;2^{n+1} + 2 &
\end{array}$$

Algorithmus

```
program
var A : array [1 ... n] of integer;

procedure fitin(i : integer)
var j : integer
begin
    j := i; temp = A[i];
    while (j > 1 and A[j div 2] < temp) do begin
        A[j] := A[j div 2]; j := j div 2
    end;
    A[j] := temp
end;

begin for i = 2 to n do fitin(i) end.
```

1.7.3 Mache aus einem Fast-Heap einen Heap

Input: A : array $[1 \dots n]$ of integer (Fast-Heap)

Output: A sortiert als Heap

Methode: Inhalt der Wurzel wandert nach unten durch Tausch mit dem jeweils größeren Wert der Söhne bis kein Sohn mehr einen größeren Inhalt hat.

Pidgin-Algorithmus:

```
procedure adjust(n : integer)

begin
    j = 1;
    while (ein Sohn von Feld j einen Inhalt größer dem von Feld j hat) do begin
        tausche Inhalt von Feld j mit dem größeren Inhalt der Söhne;
        j := Nummer des Sohns mit größerem Inhalt
    end;
end;
```

Algorithmus:

```
procedure adjust(n); {n = Länge Array}

var i, j : integer; stop: boolean;
begin
    stop := false; j := 1;
    while (stop = false and 2j ≤ n) do begin
        i:=j; j:=2j;
        if j ≠ n and A[j] < A[j+1] then j = j+1;
        if A[i] < A[j] then tausch(i, j) else stop := true;
        end;
    end;

procedure tausch(i,j:integer);
var temp:  integer;
begin temp:= A[i]; A[i]:=A[j]; A[j]:=temp end;
```

Beispiel 26 (Adjust)

35	80	70	40	45	50
80	35	70	40	45	50
80	45	70	40	35	50

Aufwand: Bei einem Baum der Höhe h sind das $2h$ Vergleiche, d.h. bei einem Array der Länge n höchstens $2\log n$ Vergleiche.

1.7.4 Heapsort

Alles zusammen liefert uns jetzt das Verfahren Heapsort:

Gegeben: A : array $[1 ... n]$ of integer.

Aufgabe: Sortieren des Arrays

Pidgin-Algorithmus:

1. Mache aus $A := A[1..n]$ einen Heap. {Dann ist das größte Element an der Wurzel $A[1]$. [Aufwand $n \log n$]}

2. Tausche erstes und letztes Element {Das größte Element steht schon an der letzten Stelle $A[n]$,und wir betrachten jetzt den Array $A[1 .. n-1]$}
   Setze $n := n - 1$.
3. Mache aus Fast-Heap $A[1 .. n]$ einen Heap {Adjust) [Aufwand $2 \log n$]}. Gehe zu 2., falls $n > 1$.

Algorithmus:

```
procedure heapsort

begin
    for i = 2 to n do fitin(i); {Anfangsheap}
    for i = n downto 2 do begin
        tausch(1, i); {Fast-Heap}
        adjust(i − 1); {Heap}
    end;
end;
```

Aufwand:

Bei einem Array der Länge n ist die Zahl der Vergleiche höchstens $n \log n + 2n \log n = 3n \log n$.

Beispiel 27

	4	**8**	3	9	5	6	7	fitin(2)
	8	4	**3**	**9**	5	6	7	fitin(3) , fitin(4)
	9	8	3	4	**5**	**6**	7	fitin(5) , fitin(6)
	9	8	6	4	5	3	**7**	fitin(7)
Heap	9	8	7	4	5	3	6	tausche(1, 7)
FastHeap	6	8	7	4	5	3	9	adjust(6)
Heap	8	6	7	4	5	3	9	tausche(1, 6)
FastHeap	3	6	7	4	5	8	9	adjust(5)
Heap	7	6	3	4	5	8	9	tausche(1, 5)
FastHeap	5	6	3	4	7	8	9	adjust(4)
Heap	6	5	3	4	7	8	9	tausche(1, 4)
FastHeap	4	5	3	6	7	8	9	adjust(3)
Heap	5	4	3	6	7	8	9	tausche(1, 3)
FastHeap	3	4	5	6	7	8	9	adjust(2)
Heap	4	3	5	6	7	8	9	tausche(1, 2)
FastHeap	3	4	5	6	7	8	9	adjust(1)
sortiert	3	4	5	6	7	8	9	

1.8 Mindestaufwand beim Sortieren nur durch Vergleich

Satz 28 (Untere Schranke)
Jeder Sortieralgorithmus, der nur mit Vergleichen arbeitet, braucht im worst-case const$\cdot n \cdot \log n$ Vergleiche.

Beweis: Sei S ein Algorithmus, der nur durch Vergleich sortiert. S sortiert einen Array $A[1..n]$ durch Umordnen der Inhalte des Arrays: der Inhalt $A[i]$ von Feld i wird in Feld $\pi(i)$ verschoben. Hierbei ist π eine bijektive Funktion, d.h. nie werden die Inhalte zweier verschiedener Felder in dasselbe Feld verschoben. Eine solche Funktion nennen wir eine *Permutation* (Umordnung).

Es gibt $n!$ viele Möglichkeiten, einen Array der Länge n umzuordnen. (den Inhalt $A[1]$ können wir in eines von n Feldern schieben, haben also n Möglichkeiten. $A[2]$ können wir in eines der $n-1$ noch leeren Felder schieben, haben also $n-1$ Möglichkeiten, $A[i]$ schieben wir in eines der noch freien $n-i+1$ Felder und $A[n]$ schließlich in das letzte noch freie Feld, hier haben wir nur noch $n-n+1=1$ Möglichkeiten).

Ein Beispiel einer Permutation ist:
$A[1..4] = (99, 37, 22, 54)$. Sortiert $A'[1..4] = (22, 37, 54, 99)$. Die entsprechende Permutation schreiben wir:

$$\pi = \begin{pmatrix} 1\,2\,3\,4 \\ 4\,2\,1\,3 \end{pmatrix}$$

Gegeben seien also n Integer, die in einer der $n!$ möglichen Anordnungen in den Array $A[1..n]$ eingeschrieben seien. Der Algorithmus S sortiert jede dieser $n!$ verschiedenen Anordnungen in dieselbe sortierte Reihenfolge. S beginnt mit einem ersten Vergleich $(A[i]\ ?\ A[j])$. Je nachdem, wie die Antwort ausfällt ("ja" oder "nein") - was wieder mit der Anordnung der Werte in A abhängt - wird getauscht und werden 2 andere Felder miteinander verglichen. Der Ausgang dieses zweiten Vergleichs bestimmt wieder, welche Felder im dritten Vergleich herangezogen werden, usw.

Wir können uns also einen binären Vergleichsbaum definieren: jeder Knoten ist mit einem Vergleich $(A[i]\ ?\ A[j])$ belegt, die beiden ausgehenden Kanten mit den Ergebnissen des Vergleichs "ja" bzw. "nein". Die Wurzel enthält den ersten Vergleich, die beiden Söhne die Vergleiche, die durchgeführt werden, wenn der erste positiv bzw. negativ ausgegangen ist, usw. Je nach Ergebnis der Vergleiche, werden wir einem Pfad in dem Baum entlang gehen, bis kein Vergleich mehr durchgeführt wird. Der so erreichte Knoten ist dann ein Blatt unseres Baumes. Es entspricht dem Ende der Sortierung.

Bei jedem zu sortierenden Array A wird der Algorithmus S eine Folge von Vergleichen durchführen, die gerade einem Pfad von der Wurzel zu einem Blatt in dem Vergleichsbaum entspricht. Die Vergleichsfolge bei einem anderen Array B wird ebenfalls einem solchen Pfad im Vergleichsbaum entsprechen und zwar einem anderen, da derselbe Pfad auch zu denselben Vertauschungen führen würde. Nie aber können verschieden angeordnete Arrays mit derselben Folge von Vertauschungen sortiert werden.

Da es $n!$ viele verschieden angeordnete Arrays mit unseren n Integers gibt, muß es auch $n!$ viele verschiedene Pfade von der Wurzel zu einem Blatt in unserem Vergleichsbaum geben, d.h. es muß $n!$ viele Blätter geben. Daraus können wir nun schließen, wie lang ein Pfad im worst case sein kann, was bedeutet, wie viele Vergleiche im schlimmsten Fall gemacht werden müssen, um einen Array der Länge n zu sortieren.

Ein binärer Baum der Höhe h hat höchstens 2^h Blätter. Soll er also $n!$ Blätter haben, muß die Höhe des Vergleichsbaums mindestens $\log n!$ sein, d.h. es muß einen Pfad der Länge $\log n!$ geben. Nun ist

$$n! = 1 \cdot 2 \cdot \dots \cdot \frac{n}{2} \cdot \dots \cdot n \geq \frac{n}{2} \cdot \dots \cdot n \geq \frac{n}{2} \cdot \dots \cdot \frac{n}{2} = \left(\frac{n}{2}\right)^{\frac{n}{2}},$$

$$\text{d.h. } \log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \left(\frac{n}{2}\right) = \frac{1}{2}(n \log n - 2n) = \frac{1}{2} n \left(\log n - 2\right) \geq c \cdot n \log n$$

für eine Konstante $c \leq \frac{1}{3}$ und genügend große n. ■

2 Suchen in Texten

Gegeben sei ein Text. Wenn wir die Leerstellen im Text ebenfalls als ein besonderes Symbol "Blank" (i.Z. □) auffassen, können wir uns unter einem Text auch eine Folge von Zeichen (String) vorstellen. Der Einfachheit halber gehen wir davon aus, daß das Textende mit dem Zeichen # markiert ist.

In diesem String wollen wir nun die Vorkommen von vogegebenen Schlüsselworten finden. Für ein einzelnes Schlüsselwort ist es nicht schwer eine Prozedur zu schreiben, die an einer Stelle des Textes anhält, an der das Schlüsselwort vorkommt.

Beispiel 29
Das Schlüsselwort sei "erde", der Textstring t

```
procedure find(var i:integer);      {suche ab Stelle i das Schlüsselwort "erde"}
var
begin
    z := 0;
    while t[i] ≠ # do begin
        zz := 0;
        if (z = 0) and (t[i] = e) then zz := 1;
        if (z = 1) and (t[i] = r) then zz := 2;
        if (z = 2) and (t[i] = d) then zz := 3;
        if (z = 3) and (t[i] = e) then zz := 4;
        if zz = 4 then exit;     {Vorkommen an der Stelle i gefunden}
        if (z = 0 or zz ≠ 0) then i := i + 1;
        z := zz;
    end;
    i := −1;      {kein Vorkommen gefunden}
end;
```

Die Variable z zählt, den wievielten Buchstaben von 'erde' wir bereits gefunden haben.

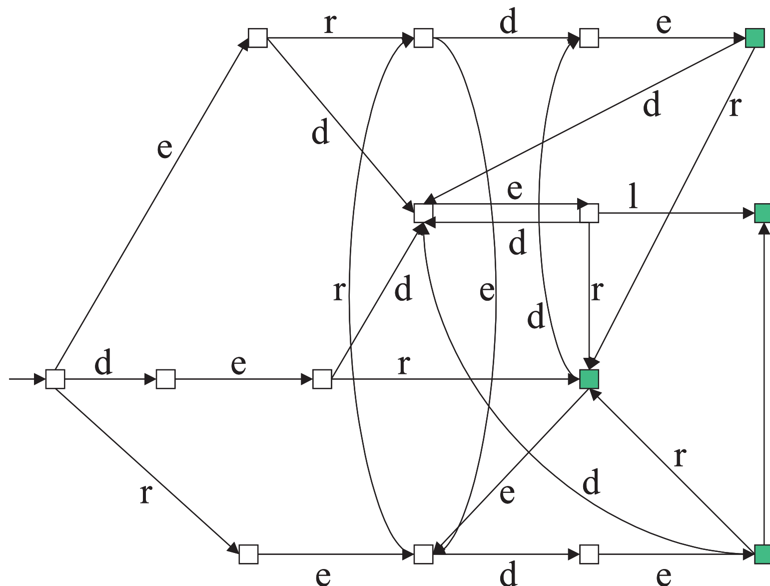
Ein Nachteil dieser Prozedur ist, daß überlappende Vorkommen von 'erde' wie in 'erderderderde' nicht gefunden werden. Außerdem wird nur ein Schlüsselwort gefunden. Schließlich funktioniert diese Prozedur z. B. nicht bei 'rorot' in dem Text 'rororot'. Es wird unsere Aufgabe sein, das Suchprogramm außerdem komfortabler zu machen, so daß viele Schlüsselworte, Wortgruppen und logische Verknüpfungen von Worten gefunden werden.

Dazu müssen wir etwas ausholen:

2.1 Digraphen

Das nächste Bild stellt einen Digraphen dar (directed graph, gerichteter Graphen siehe Def. 7), dessen Kanten (Pfeile) mit Zeichen beschriftet sind, dessen Knoten (Kästchen) markiert (hier

dunkel) sein können und der einen einzelnen Startknoten (kleiner Pfeil) besitzt. Der Graph ist als vollständig zu betrachten, d.h. von jedem Knoten soll für jedes Zeichen ein entsprechender Pfeil abgehen. Aus Gründen der Übersichtlichkeit haben wir alle Pfeile weggelassen, die in den Startknoten oder in einen seiner direkten Nachfolger zurückführen. Bezeichnen wir diese direkten Nachfolger, die vom Startknoten aus mit den Eingaben e, d bzw. r erreicht werden mit [e], [d] bzw. [r], so führen alle fehlenden e-Pfeile in [e], alle fehlenden d-Pfeile in [d], alle fehlenden r-Pfeile in [r] und alle übrigen fehlenden Pfeile in den Startknoten.



Der Text wird gelesen und - beginnend beim Startknoten (kleiner Pfeil) - läuft man je nach gelesenem Zeichen eine Kante entlang zum nächsten Knoten. Immer wenn ein markierter (dunkler) Knoten erreicht ist, endet der bisher gelesene Text auf ein Schlüsselwort. In diesem Fall sind die Schlüsselworte: 'erde', 'edel', 'rede' und 'der'. Der Text 'reder' wird also zweimal einen markierten Knoten durchlaufen, einmal wenn er auf 'rede' endet und beim nächsten Zeichen gleich wieder, wenn er auf 'der' endet. Läßt man also den Text durch diesen Graphen laufen, und löst jedesmal ein Ereignis (Signal) aus, wenn ein markierter Knoten erreicht wird, so finden wir so jedes Vorkommen eines unserer Schlüsselworte im Text.

Es ist nicht schwer, ein Programm zu schreiben, daß genau dies nachmacht - vorausgesetzt wir haben den obigen Graphen vorliegen. Die Frage ist, wie man einen solchen Graphen herstellt. Im obigen Fall, wo wir es mit einer endlichen Auflistung von Schlüsselworten zu tun haben, können wir einen direkten Zugang zur Konstruktion des Graphen angeben, der wenig Theorie erfordert, aber etwas komplizierter ist. Danach werden wir einen indirekteren Zugang beschreiben, der Fakten aus der Theorie der Finiten Automaten nützt, und einfacher wirkt, wenn diese Fakten bekannt sind.

2.1.1 Direkter Zugang

Wir betrachten obigen Graphen, mit dem wir die Schlüsselworte 'erde', 'edel', 'rede' und 'der' finden. Zunächst erkennen wir, daß es für jeden Schlüsselwortanfang einen Knoten gibt, in den dieser hineinläuft. Insbesondere gibt es für jedes Schlüsselwort einen eigenen Pfad vom Startzustand in einen Endzustand. Dies ist sozusagen das Gerüst des Graphen, in dem schon

alle Knoten vorhanden sind. Unser Ziel ist ein Graph mit folgender Eigenschaft: Laufen wir den Kanten des Graphen entlang entsprechend den gelesenen Zeichen unseres Textes, so befinden wir uns in einem Knoten, der dem Schlüsselanfang u entspricht genau dann, wenn u gerade der längste Schlüsselwortanfang ist, auf den unser bisher gelesener Text endet. Dazu gehen wir wie folgt vor:

1. Sei SW die Menge unserer Schlüsselworte und SWA die Menge der Anfänge der Schlüsselworte (einschließlich des leeren Wortes ε und der Schlüsselworte selbst). Für jeden Schlüsselwortanfang $v \in$ SWA schaffen wir einen Knoten, den wir mit $[v]$ bezeichnen. Der Startknoten ist $[\varepsilon]$. Ist a ein Symbol und va auch aus SWA, so führen wir einen a-Pfeil von $[v]$ nach $[va]$. Der so angegebene Graph bildet das Gerüst unseres eigentlichen Graphen. Jeder Schlüsselwortanfang v führt vom Startzustand aus in seinen Knoten $[v]$. Befinden wir uns im Knoten $[v]$, so signalisiert dies, daß wir jetzt den Anfang v eines Schlüsselwortes gelesen haben. Endknoten sind die Knoten, in denen wir ein ganzes Schlüsselwort gelesen haben. Wenn allerdings ein Schlüsselwort u ein Teilwort des Schlüsselwortes v ist, also etwa $v = xuy$ für zwei Wort x, y, dann muß nicht nur der Knoten $[u]$, sondern auch der Knoten $[xu]$ ein Endknoten werden. Denn, haben wir xu gelesen, so endet der Text offenbar auch auf das Schlüsselwort u. Damit ist die Menge der Endknoten $\{[u] \mid u \in$ SWA und u endet auf ein Schlüsselwort aus SW $\}$
2. Unser Ziel ist, daß wir im Knoten $[u]$ sind, wenn u der längste Schlüsselwortanfang ist, auf den unser bisher gelesener Text endet. Natürlich lesen wir in einem Knoten nicht immer das gewünschte nächste Zeichen eines Schlüsselwortes. Auch für die anderen Zeichen, muß es Kanten geben, die von diesem Knoten ausgehen, damit wir nicht hängen bleiben. So muß es vom Startknoten aus nicht nur einen e-Pfeil, einen d-Pfeil und einen r-Pfeil geben, sondern für jeden anderen Buchstaben ebenfalls einen Pfeil. (In unserem Bild vermissen wir dies, weil alle Pfeile weggelassen worden sind, die in den Startknoten zurückführen, was die meisten sind.)
3. Wohin soll ein Pfeil, der nicht zum Gerüst gehört, führen. Wir möchten gern im Knoten $[u]$ sein, wenn u der längste Schlüsselwortanfang ist, auf den unser bisher gelesener Text endet. Lesen wir also im Knoten $[u]$ im Text ein Zeichen a und ist ua kein Schlüsselwortanfang, so suchen wir aus den Endstücken von ua den längsten Schlüsselwortanfang v heraus, so daß v also das längste Endstück von ua ist, das selbst ein Schlüsselwortanfang ist. Dann führen wir den a-Pfeil in den Knoten $[v]$ (oft wird dies ε sein, d.h. der Pfeil führt in den Startknoten zurück).
   Nehmen wir z.B. den Knoten $[er]$, den wir auf dem 'erde'-Pfad vom Startzustand aus erreichen, wenn wir ein e und dann ein r einlesen. Lesen wir weiter ein d ein, so kommen wir zum Knoten $[erd]$. Lesen wir aber ein e, so haben wir jetzt insgesamt 'ere' gelesen. Kein Schlüsselwort beginnt mit 'ere', aber es gibt ein Schlüsselwort, das mit 're' anfängt. Also sollte der e-Pfeil in den Knoten $[re]$ gehen, der signalisiert, daß bisher 're' gelesen worden ist.

Ist u der längste Schlüsselwortanfang, auf den der bisher gelesene Text endet, so befinden wir uns im Knoten $[u]$. Endet u auf ein Schlüsselwort (oder ist selbst ein Schlüsselwort), so ist $[u]$ nach obiger Definition ein Endknoten. Immer wenn wir einen Endknoten erreichen, haben wir ein Schlüsselwort gefunden. Umgekehrt, endet der bisher gelesene Text auf ein Schlüsselwort v, so müssen wir in einem Knoten $[u]$ gelaufen sein, dessen Namen u auf v endet und der damit ein Endknoten ist.

Auf diese Weise erhalten wir den obigen Graphen, wenn wir dort noch die fehlenden Pfeile ergänzen.

Für die, die sich in der Theorie der Finiten Automaten etwas auskennen, formulieren wir dies formaler:

Gegeben sei eine endliche Menge SW von Schlüsselworten und ein Text T über dem Alphabet Σ*. SWA sei die Menge der Anfänge von Schlüsselworten aus SW. Natürlich ist auch SWA endlich. Für* $x \in \mathrm{SWA}$ *sei* $[x]$ *die Menge aller Worte* w*, für die* x *der längste Schlüsselwortanfang ist, auf den sie enden, d.h.* $[x] = \{w \mid w \text{ endet auf } x \text{ und, endet } w \text{ auf } y\text{, so } |x| > |y|\}$ *. Dann ist* $K = \{[x] \mid x \in \mathrm{SWA}\}$ *eine endliche Partion auf* Σ^**, denn jedes Wort endet auf ein* $x \in \mathrm{SWA}$ *(auch* $\varepsilon \in \mathrm{SWA}$*), ist also in einer Klasse* $[x]$*, für jedes Wort* w *ist aber die Klasse, in der es liegt, eindeutig durch das längste Endstück bestimmt, das Schlüsselwortanfang ist (*$w \in [x] \Longrightarrow \forall\, y \in \mathrm{SWA}\colon w \notin [y]$*).*

Die durch K *induzierte Äquivalenz* $\sim$ *ist auch eine Rechtskongruenz: Sind* x *und* y *äquivalent, liegen sie also in derselben Klasse* $[u]$*, so ist* u *für beide der längste Schlüsselwortanfang, auf den sie enden. Für jedes* $z \in \Sigma^*$ *gilt dann aber: ist* v *der längste Schlüsselwortanfang auf den* xz *endet, so gilt dies auch für* yz*, da* v *ein Ende von* uz *sein muß, weil* u *sonst nicht maximal gewesen wäre. Da die Menge der Worte, die auf ein Schlüsselwort enden, gerade die Vereinigung der Klassen* $[x]$ *ist, für die* x *auf ein Schlüsselwort endet (*$L = \bigcup\{[x] \mid x \text{ endet auf ein Schlüsselwort}\}$*), induziert diese Partition einen finiten Automaten für L, d.h. einen DFA der* w *akzeptiert genau dann, wenn* w *auf ein Schlüsselwort endet.*

Dies war schon etwas kompliziert. Es wird erheblich einfacher und allgemeiner, wenn wir die Anfangsgründe der Theorie der Finiten Automaten zu Hilfe nehmen. Darum folgender Exkurs:

2.2 Finite Automaten

Solche speziellen Digraphen mit Kanten, die mit (einzelnen) Zeichen belegt sind, mit einem Startknoten und markierten Endknoten, nennen wir auch "finite Automaten". Formal definiert man sie so:

Definition 30 (NFA)
Ist Q eine endliche Menge (wir nennen sie die Menge der *Zustände*) und Σ ebenfalls eine endliche Menge (wir nennen sie das *Inputalphabet*), gibt es einen ausgezeichneten Zustand $s \in Q$ (wir nennen ihn den *Startzustand*) und eine ausgezeichnete Teilmenge $F \subseteq Q$ von Q (wir nennen sie die Menge der *Endzustände*) und ist schließlich $\delta \subseteq Q \times \Sigma \times Q$ eine Menge von Tripeln (wir nennen sie die *Befehlsmenge*), so bezeichnet man die Struktur

$$M = (Q, \Sigma, \delta, s, F)$$

einen *(nichtdeterministischen) finiten Automaten.*

Definition 31
Die von einem finiten Automaten *erkannte Sprache* ist die Menge aller Worte, die von dem Startzustand des Automaten in einen Endzustand führen, genauer: für die es einen Pfad vom Startknoten in einen Endknoten gibt, der mit den Zeichen dieses Wortes (in ihrer Reihenfolge) beschriftet ist.

Man erkennt den zugehörigen Digraphen sofort wieder, wenn man die Zustände als Knoten und die Befehle als Kanten ansieht. Wir wollen daher 'Zustand' und 'Knoten', ebenso wie 'Kante' und 'Befehl' synonym verwenden.

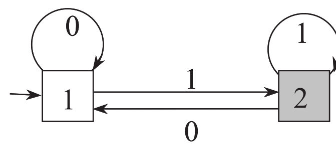
Da wir nicht ausgeschlossen haben, daß von einem Knoten, mehrere gleichbeschriftete Pfeile ausgehen, nennen wir den finiten Automaten nichtdeterministisch (es ist nicht determiniert, welchen Pfeil wir wählen sollen, wenn wir im Text ein Zeichen 'a' lesen und uns in einem Knoten befinden, von dem aus mehrere verschieden a-Pfeile ablaufen. Es ist klar, das dies für uns nicht wünschenswert ist.

Wir wünschen uns *deterministische* finite Automaten, so daß immer klar ist, welchem Pfeil man folgen muß. Zudem wünschen wir uns auch *vollständige* Automaten, bei denen in jedem Zustand und für jedes Zeichen ein entsprechender Pfeil abgeht, so daß wir immer weitermachen können, und nicht stoppen müssen, weil es für das gelesene Zeichen gar keinen Pfeil gibt.

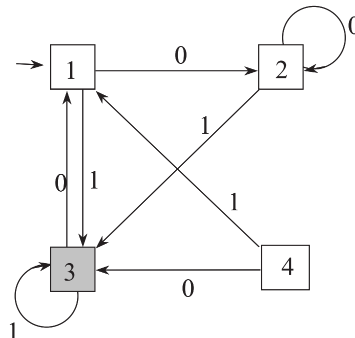
Definition 32
Ein finiter Automat heißt *deterministisch*, wenn von keinem Knoten aus zwei gleichbeschriftete Pfeile ausgehen. Er heißt *vollständig*, wenn von jedem Knoten aus für jedes Zeichen ein entsprechend beschrifteter Pfeil abgeht.

Einige Beispiele:

Beispiel 33
Dieser finite Automat ist deterministisch und vollständig und er erkennt gerade alle Worte mit den Zeichen 0 und 1, die auf eine 1 enden.



Beispiel 34
Dieser finite Automat ist ebenfalls deterministisch und vollständig und erkennt dieselbe Sprache.



An diesen Beispielen kann man erkennen, erstens, daß verschiedene Automaten doch dieselbe Sprache erkennen können, und zweitens, daß man für eine bestimmte Sprache mehr oder weniger umständliche Automaten angeben kann. Es gibt Verfahren, die es ermöglichen, den kleinsten Automaten für eine gegebene Sprache zu finden. Wir wollen uns aber hier begnügen, überhaupt einen Automaten für eine Sprache zu finden, in unserem Falle für die Sprache aller Worte, die auf ein Schlüsselwort enden.

2.2.1 Verfahren für endliche viele Schlüsselwörter:

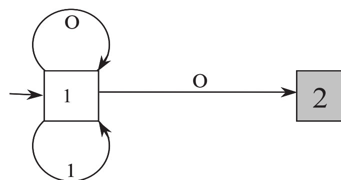
Gegeben sei also eine endliche Menge SW von Schlüsselwörtern und ein Text T. Sei SWA die Menge aller Schlüsselwortanfänge, d.h. aller Wörter, die den Beginn eines Schlüsselwortes bilden

(einschließlich des leeren Wortes ε und der Schlüsselwörter selbst). Die Methode ist:

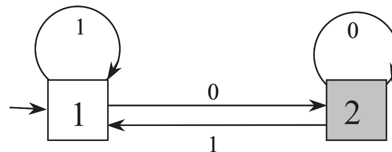
1. Baue einen nichtdeterministischen finiten Automat, der alle Worte erkennt, die auf ein Schlüsselwort enden.
2. Konstruiere daraus einen äquivalenten deterministischen Automaten.
3. Mache daraus ein Programm.

Beispiel 35
Gegeben ist ein Text (String), der nur aus 0'en und 1'en besteht. Gesucht wird jede Stelle im Text, an der eine 0 steht. Das Schlüsselwort ist also 0, und die Worte, die auf ein Schlüsselwort enden, sind in diesem Fall, alle Worte, die auf 0 enden.

Folgender nichtdeterministische finite Automat erkennt offenbar diese Worte: der Startzustand ist 1, der Endzustand ist 2 und die Befehlsmenge ist $\delta = \{(1,0,1),(1,1,1),(1,0,2)\}$.



Als deterministischen äquivalenten finiten Automaten erhalten wir: der Startzustand ist 1, der Endzustand ist 2 und die Befehlsmenge ist $\delta = \{(1,0,2),(1,1,1),(2,0,2),(2,1,1)\}$.



Das Programm dazu sieht dann so aus:

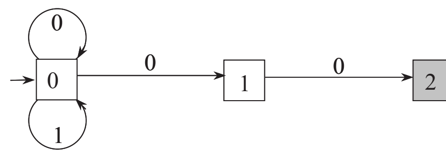
```
program
var z, zz integer
begin z := 1; i := 1
    while t[i] ≠ # do begin
        if (z = 1) ∧ (t[i] = 0) then zz := 2;
        if (z = 1) ∧ (t[i] = 1) then zz := 1;
        if (z = 2) ∧ (t[i] = 0) then zz := 2;
        if (z = 2) ∧ (t[i] = 1) then zz := 1;
        z := zz;
        if z = 2 then 'piep';
        i := i + 1;
    end;
end;
```

2.2.2 Potenzautomat

Wie macht man aus einem nichtdeterministischem finiten Automaten einen deterministischen, der dieselbe Sprache erkennt? Sei M der NFA, zudem wir einen DFA M' bauen wollen. Man kann sich vorstellen, daß der Zustand eines Automaten Informationen über das bisher eingelesene Wort trägt.

Hat M den Startzustand q_0, so geben wir auch M' diesen Startzustand (wir schreiben ihn $\{q_0\}$). Können wir in M vom Startzustand q_0 aus mit dem Zeichen a mehrere andere Zustände $q_1, ..., q_k$ erreichen, so gehen wir in M' mit dem Zeichen a in einen Zustand $\{q_1, ..., q_k\}$, was man sich so vorstellen kann, daß wir gleichzeitig in allen Zuständen $q_1, ..., q_k$ sind. Im Zustand $\{q_1, ..., q_k\}$ gehen wir mit dem Zeichen b in den Zustand, der aus allen Zuständen von M besteht, die von einem der Zustände $q_1, ..., q_k$ aus mit dem Zeichen b erreicht würden. Das sind nämlich die Zustände, in denen wir nach dem Lesen von a und b sein könnten. Und so machen wir immer weiter, bis für jeden neu geschaffenen Zustand klar ist, wie man von ihm weiterkommt.

Beispiel 36
Wir wollen einen Automaten konstruieren, der die Sprache $L = \{w \in \{0,1\}^* \mid w \text{ endet auf } 00\}$ erkennt. Einen nichtdeterministischen finiten Automaten anzugeben ist nicht schwer:



Wir wollen dazu jetzt einen deterministischen Automaten konstruieren, der dieselbe Sprache erkennt. Die Befehlstafel des obigen Automaten und des äquivalenten Automaten kann man so schreiben:

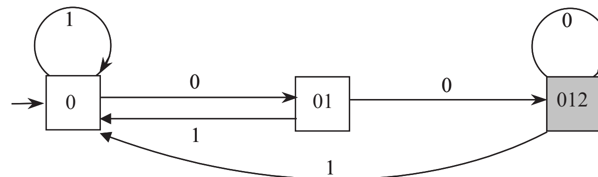
NFA

δ	0	1
0	0,1	0
1	2	-
2	-	-

DFA

δ	0	1
$\{0\}$	$\{0,1\}$	$\{0\}$
$\{0,1\}$	$\{0,1,2\}$	$\{0\}$
$\{0,1,2\}$	$\{0,1,2\}$	$\{0\}$

Dann ist der Graph des deterministischen finiten Automaten:



Formal kann man diese Konstruktion so festhalten: Ist $M = (Q, \Sigma, \delta, q_0, F)$ der NFA, so ist folgender DFA äquivalent dazu:

$$\begin{aligned} M' &= \left(\mathcal{P}(Q), \Sigma, \delta', \{q_0\}, F'\right) \quad \text{, wobei} \\ \mathcal{P}(Q) &= \{P \mid P \subseteq Q\}, \\ F' &= \{P \subseteq Q \mid \exists p \in P : p \in F\} \text{ und} \\ \delta' &= \left\{\left(P, a, P'\right) \mid P, P' \subseteq Q,\ P' = \{q \in Q \mid \exists p \in P : (p, a, q) \in \delta\}\right\}, \end{aligned}$$

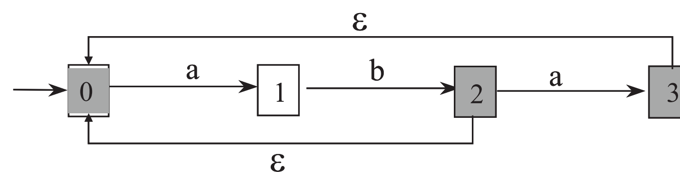
d.h. alle a-Pfeile, die von einem Zustand von P aus gehen, werden aufgesammelt und ihre Zielzustände in der Menge P' zusammengefaßt. Dieses P' ist dann der Zielzustand des a-Pfeils, der von P ausläuft. Weil man die Menge $\mathcal{P}(Q)$ aller Teilmengen von Q die Potenzmenge von Q nennt, heißt dieser Automat M' auch der Potenzautomat von M. Er ist das deterministische Äquivalent von M.

2.2.3 ε-NFA

Den nichtdeterministischen finiten Automaten haben wir definiert, weil er zu einer gegebenen Aufgabe sehr viel leichter zu konstruieren ist. Dies werden wir im folgenden auch noch sehen. Um uns das Leben noch leichter zu machen, lassen wir jetzt auch Pfeile zu, die nicht beschriftet sind, die man also entlanggehen kann, ohne daß ein zusätzliches Zeichen im Text gelesen wird. Man nennt so etwas auch einen spontanen Zustandsübergang, da er nicht durch ein eingehendes Zeichen ausgelöst wird. Diese finiten Automaten nennen wir ε-Automaten (ε-NFA's), weil ε für das leere Wort steht und man sich angewöhnt hat, an diese unbeschrifteten Pfeile das Zeichen ε zu schreiben, um eben anzudeuten, daß beim Durchlaufen dieses Pfeils nur das leere Wort - also kein Zeichen - hinzukommt. Man schreibt also etwas an den Pfeil, um zu sagen, daß man nichts an ihn schreiben möchte. In Formeln allerdings ist es hilfreich mit diesem ε zu operieren.

Wie wir sehen werden, ist dies für die Konstruktion von finiten Automaten sehr nützlich. Aber wieder gilt, daß es bei der Umsetzung in ein Programm eigentlich nicht erwünscht ist, da solche Pfeile wieder eine Uneindeutigkeit ins Spiel bringen: soll man dem unbeschrifteten Pfeil folgen oder soll man dem Pfeil folgen, der dem nächsten Zeichen im Text entspricht? Darum gilt - wie oben - diese Automaten benützen wir nur zum Entwurf. Sie müssen wieder in DFA's gewandelt werden.

Beispiel 37
Wir wollen einen Automaten konstruieren, der die Sprache $L = \{ab, aba\}^*$ erkennt, alsodas Leere Wort ε und alle Wörter, die sich aus den Wörtern ab und aba zusammensetzen lassen (z.B. $ababaab$). Die Idee ist einfach: wir bauen einen Automaten für die Worte ab und aba - das ist einfach eine Kette von 4 Zuständen und drei Pfeilen, der Reihe nach mit a, b, a beschrieben. Der Zustand, der durch ε, der, der durch ab, und der, der durch aba erreicht wird, wird Endzustand. Dann führen wir von jedem der beiden Endzustände noch einen ε-Pfeil in den Startzustand zurück, so daß das nächste Wort aus $\{ab, aba\}$ in einen Endzustand führt:

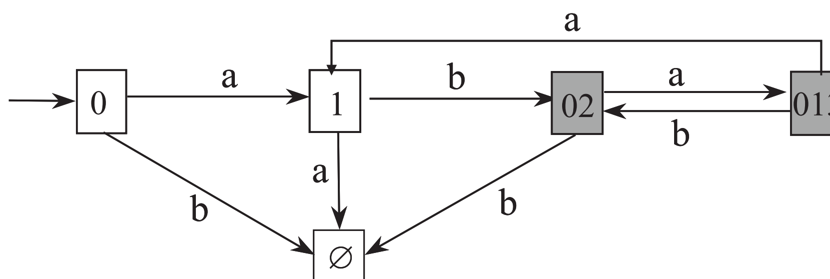


Um aus einem ε-NFA einen äquivalenten deterministischen finiten Automaten (DFA) zu machen, gehen wir wie im nichtdeterministischen Fall vor: wir bauen den Potenzautomaten. Die neuen Zustände sind Mengen von alten Zuständen. Ist P ein neuer Zustand, so gehen wir mit einem a-Pfeil in den Zustand P' über, der alle Zustände enthält, in die man mit einem a-Pfeil oder mit einem Pfad aus einem a-Pfeil und eventuell mehreren folgenden ε-Pfeilen von einem der Zustände von P aus gelangt. Der neue Startzustand besteht aus allen Zuständen ε-NFA, die vom Startzustand des ε-NFA aus durch reine ε-Pfade erreicht werden. Die Endzustände des DFA sind die, die einen Endzustand des ε-NFA enthalten.

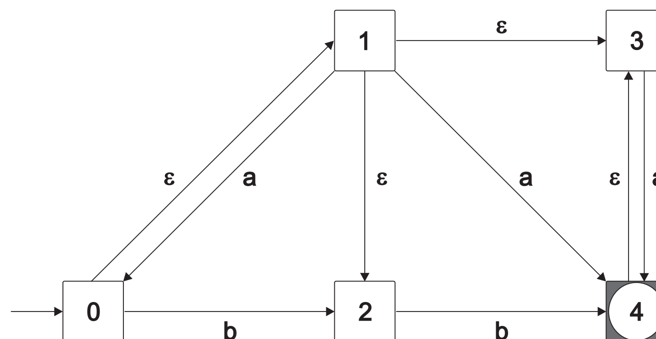
Beispiel 38
Um unser obiges Beispiel weiterzuführen, erstellen wir die Befehlstafel des DFA:

	δ	a	b
s, f	$\{0\}$	$\{1\}$	$\varnothing$
	$\{1\}$	$\varnothing$	$\{0, 2\}$
f	$\{0, 2\}$	$\{0, 1, 3\}$	$-$
f	$\{0, 1, 3\}$	$\{1\}$	$\{0, 2\}$
	$\varnothing$	$\varnothing$	$\varnothing$

Dies gibt folgenden Graphen:



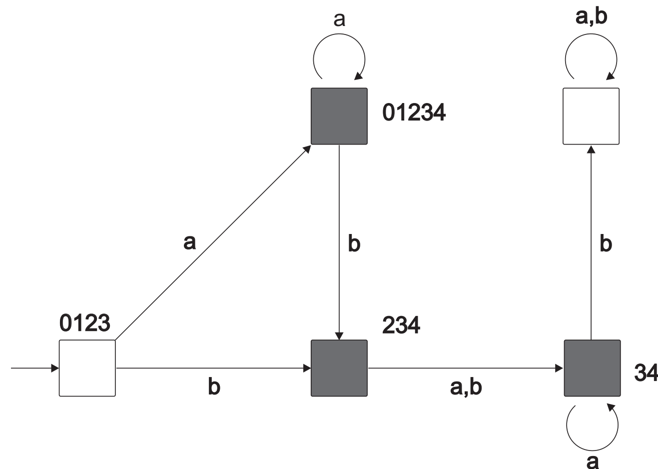
Beispiel 39
Geben sei folgender ε-NFA:



Die Wandlung in einen äquivalenten DFA mittels der Potenzautomatenkonstruktion führt zu folgender Befehlstafel:

δ	a	b
0123	01234	234
01234	01234	234
234	34	34
34	34	$\varnothing$
$\varnothing$	$\varnothing$	$\varnothing$

und zu folgendem Graphen:



2.3 Reguläre Ausdrücke

Nun haben wir alle Mittel, um auch eine komplexe Wortsuche zu programmieren. Suchen wir eines von mehreren Schlüsselworten, so entspricht das einer ODER-Verknüpfung der Schlüsselworte (wir suchen 'erde' oder 'edel' oder 'rede' oder 'der'). Man könnte aber auch nach Stellen suchen, an denen mehrere der Schlüsselworte vorkommen. Dies entspräche einer UND-Verknüpfung (wir suchen eine Stelle, bis zu der 'erde' und 'edel' und 'der' vorgekommen sind). Wir können auch Worte suchen, die mit einer bestimmten Silbe anfangen oder enden, oder die eine bestimmte Silbe enthalten.

Um seinen Wunsch dem Programm mitzuteilen, muß man einen Ausdruck eingeben, der formuliert, was man will. Ein solcher Ausdruck besteht aus Worten und Verknüpfungssymbole (genau wie ein arithmetischer Ausdruck aus Zahlsymbolen und Operationssymbolen wie $+$ und $\times$ besteht). Wir wollen hier Ausdrücke vorschlagen, die nur drei Operationssymbole enthalten, mit denen man aber, wie man sich vergewissern kann, sehr viele Suchwünsche formulieren kann.

Die Operationen, die wir benützen, sind

1. die Vereinigung von Wortmengen (wir verwenden das Zeichen $+$ dafür)
2. die Konkatenation von Wortmengen (dafür schreiben wir $\cdot$)
3. die Iteration von Wortmengen (dafür schreiben wir *)

Zunächst müssen wir genau definieren, was diese Operationen auf Wortmengen bewirken:

1. Die Vereinigung $A+B$ von zwei Mengen A und B ist die Menge, die gerade alle Elemente von A und alle Elemente von B enthält.
2. Die Konkatenation $A \cdot B$ ist die Menge, die aus allen Worten besteht, die man erhält, wenn man hinter ein Wort von A ein Wort von B schreibt.
3. Die Iteration A^* ist die Menge, die aus allen Worten besteht, die man erhält, wenn man beliebig viele Worte aus A hintereinanderschreibt (das schließt auch ein, daß man null Worte hintereinanderschreibt, d.h. das leere Wort ε gehört immer zu A^*).

Wenn wir also irgendein Vorkommen von 'erde' oder 'edel' oder 'rede' oder 'der' suchen, also Textstellen, die auf eines dieser Schlüsselwort enden (und wenn A unser Alphabet ist), so können wir die Menge dieser Worte (Textanfänge) durch den Ausdruck $A^* \cdot$ (erde+edel+rede+der)

beschreiben. Suchen wir einen Textanfang, in dem ‘erde’ und ‘edel’ vorgekommen ist, so können wir dies durch den Ausdruck: $A^* \cdot$ erde $\cdot$ $A^* \cdot$ edel $+$ $A^* \cdot$ edel $\cdot$ $A^* \cdot$ erde). Wollen wir im Text immer hinter einem Wort stoppen, das das Zeichen r enthält, so können wir dies durch den Ausdruck $A^* \cdot r \cdot (A - \{\Box\})^* \cdot \Box$ (wobei $\Box$ das Zeichen für eine Leerstelle ist, das wir als zu unserem Alphabet A gehörig ansehen wollen.)

Wir wollen diese Ausdrücke, die wir reguläre Ausdrücke nennen wollen, noch formal exakt definieren. Dabei müssen wir zwischen einem Wortausdruck und der von ihm beschriebenen Wortmenge unterscheiden. Ist R ein Ausdruck, so wollen wir die durch R beschriebene Wortmenge mit LR bezeichnen.

Definition 40 (regulärer Ausdruck (RA) über Σ)
Wir definieren dies induktiv über den Aufbau:

1. Φ ist ein regulärer Ausdruck, der für die Menge $L\Phi = \varnothing$ steht.
2. Ist $a \in \Sigma$, so ist (a) ein regulärer Ausdruck, der für die Menge $L(a) = \{a\}$ steht.
3. Sind R und S reguläre Ausdrücke, so auch
   $(R \cdot S)$, wobei $L(R \cdot S) = LR \cdot LS$
   $(R + S)$, wobei $L(R + S) = LR \cup LS$ und
   (R^*), wobei $L(R^*) = (LR)^*$

Wie bei den arithmetischen Audrücken, wollen wir uns auch hier einer saloppen Schreibweise bedienen und Klammern sparen durch Prioritätsregeln der Operationszeichen: es geht $*$ vor $\cdot$ und $\cdot$ vor $+$. So schreiben wir

$$(1^*01)^*\ 00\ (0+1)^* \text{ statt}$$
$$\left\langle \left\{ \left[((1)^*) \cdot ((0) \cdot (1)) \right]^* \right] \cdot ((0)\,(0)) \right\} \cdot [((0) + (1))^*] \right\rangle \ .$$

Beispiel 41
Ein regulärer Ausdruck, der die Menge $\{\varepsilon\}$ beschreibt, also die Wortmenge, die nur das leere Wort ε enthält, wäre zum Beispiel (Φ^*) ,denn $L(\Phi^*) = (L\Phi)^* = \varnothing^* = \{\varepsilon\}$ (da die leere Menge $\varnothing$ keine Worte enthält, können wir auch keine Worte aus $\varnothing$ hintereinanderschreiben, genauer wir können nur null Worte hintereinanderschreiben, was das leere Wort ε ergibt)

Beispiel 42

$R_1 = (ab + aba)^*$ $\quad LR_1 = \{ab,\, aba\}^*$
$R_2 = (0+1)^*\, 111\, (0+1)^*$ $\quad LR_2 = \{w \mid w \text{ enthält } 111\}$

Definition 43
Wir nennen zwei reguläre Ausdrücke R und S *äquivalent* (i.Z. $R \equiv S$), wenn sie dieselbe Wortmenge beschreiben, wenn also $LR = LS$.

Beispiel 44

$R_3 := (1 + 01)^*\ 00\ (0 + 1)^*$
$R_4 := (0 + 1)^*\ 00\ (0 + 1)^*$

Es gilt: $R_3 \equiv R_4$

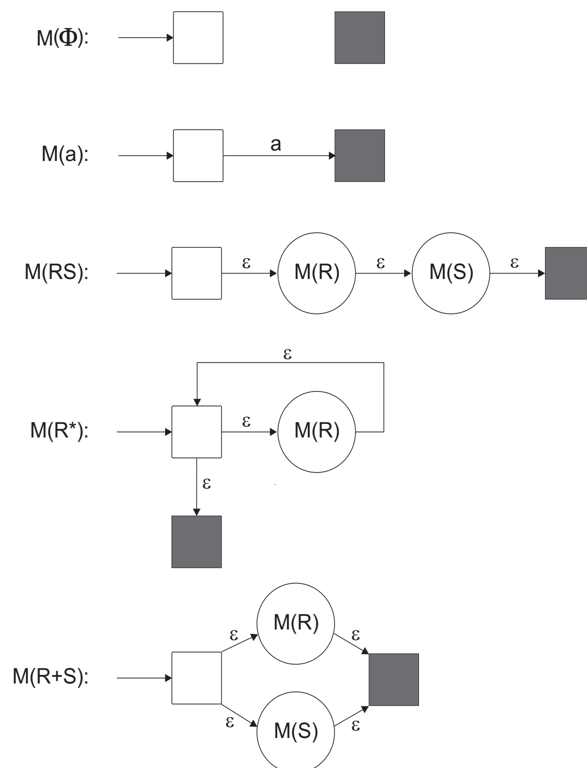
Beweis: Wir müssen zeigen, daß $LR_3 = LR_4$. Dazu zeigen wir erst $LR_3 \subseteq LR_4$ und dann $LR_4 \subseteq LR_3$.

$LR_3 \subseteq LR_4$ dürfte klar sein, da jedes Wort, das aus 1 und 01 zusammengesetzt ist, natürlich ein Wort ist, das aus 0 und 1 aufgebaut ist.

$LR_4 \subseteq LR_3$: Die Worte aus LR_4 sind gerade die Worte aus 0 und 1, die das Teilwort 00 enthalten. Betrachten wir ein solches Wort, und den Anfang dieses Wortes bis zum ersten Vorkommen von 00. Der Anfang dieses Wortes kann jetzt jedes Wort aus 0 und 1 sein, das nicht 00 enthält und auf 1 endet, d.h. jedes Wort, das aus 1 oder 01 zusammengesetzt ist. Das sind aber gerade die Worte aus $L\left((1+01)^*\right)$ .Somit besteht jedes Wort aus LR_4 aus einem Anfang aus $L\left((1+01)^*\right)$, gefolgt von dem Teilwort 00 und dies wieder gefolgt von einem beliebigen Wort aus 0 und 1, also aus $L\left((0+1)^*\right)$. ■

Satz 45 (Kleenesche Synthese)
Zu jedem regulären Ausdruck R existiert ein ε-NFA, der LR erkennt.

Beweis: (Induktion über den Aufbau von R). Wir geben den Automaten für den regulären Ausdruck Φ und für die regulären Ausdrücke (a) der einzelnen Zeichen $a \in \Sigma$. Unter der Annahme, daß wir die Automaten $\mathrm{M}(R)$ und $\mathrm{M}(S)$ für die regulären Ausdrücke R und S schon gefunden haben, zeigen wir wie die Automaten für $(R \cdot S), (R+S)$ und (R^*) aussehen. Man beachte, daß die so konstruierten Automaten immer nur einen Endzustand haben. Ein Pfeil zwischen zwei Automaten soll einen Pfeil vom Endzustand des einen in den Startzustand des anderen Automaten bedeuten.



■

2.4 Komplexe Wortsuche

Jetzt sind wir soweit, eine komplexere Suche nach Worten zu formulieren und in ein Programm umzusetzen. Das sind vier Schritte:

1. Bildung des regulären Ausdrucks R, der unseren Wunsch beschreibt,

2. Erstellen eines ε-NFA, der die Worte von LR erkennt,
3. Wandlung des ε-NFA in einen DFA und
4. Implementation dieses DFA in ein Suchprogramm.

Zu 4. wollen wir noch erklären, wie eine Implementation eines DFA aussehen kann:

Gegeben sei ein DFA $M = (Q, \Sigma, \delta, s, F)$ mit $Q = \{1, ..., e\}$, $s = 1$, $F = \{e\}$ und

$$\delta := \{(p_i, a_i, q_i) \mid i = 1, ..., k\,;\; p_i, q_i \in Q,\; a_i \in \Sigma\}\,.$$

Dann realisiert folgende Prozedur das Suchen eines der Worte aus $L(M)$ mittels des DFA M. Wir nehmen dabei an, daß wir durch früheres Suchen schon bis an die Stelle i im Text gelangt seien, und im DFA M inzwischen den Zustand q erreicht haben. Diese beiden Parameter i und q müssen der Prozedur also mitgegeben werden. Beim ersten Aufruf der Prozedur ist $i = 1$ und $q = 1$. Wenn wir weitersuchen wollen, so wird die Prozedur wieder aufgerufen.

```
procedure find(var i, q : integer)
var z, zz : integer
begin
    z := q;
    while t[i] ≠ # do begin
        if (z = p1) ∧ (t[i] = a) then zz := q1; {dies entspricht dem ersten Befehl von M}
        ⋮
        ⋮
        if (z = pk) ∧ (t[i] = ak) then zz := qk; {dies entspricht dem letzten Befehl von M}
        z := zz
        if z = e then exit; {Ein Endzustand wurde erreicht, die Prozedur wird verlassen.
                             An der Position i endet ein Schlüsselwortes}
        i := i + 1
    end;
    i := −1; { i = −1 signalisiert, daß kein Schlüsselwort mehr vorgekommen ist}
end;
```

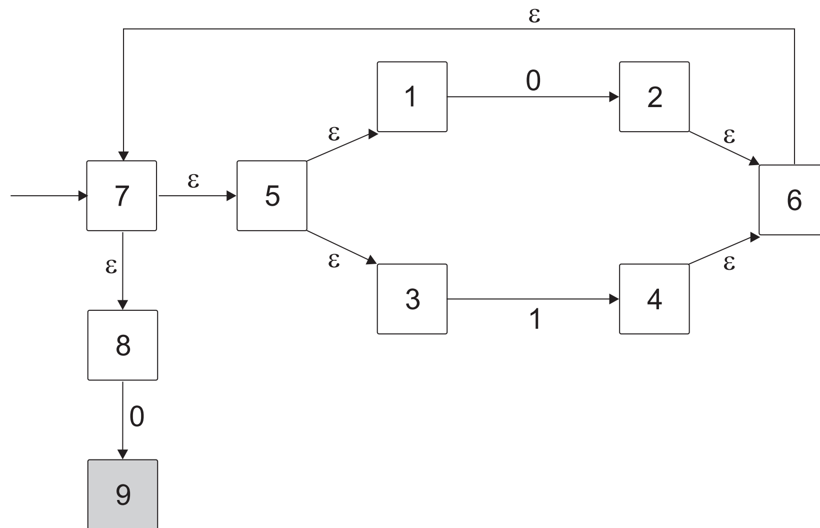
Anmerkung 46
Diese Prozedur funktioniert nur, wenn der vorliegende DFA vollständig ist, wie wir dies für unsere Textsuche erwarten. Natürliche können wir sie leicht so abändern, daß sie auch einen unvollständigen DFA realisiert. Z.B. können wir die Variable zz am Anfang der while-Schleife jeweils auf zurücksetzen -1 setzen und die Prozedur mit exit verlassen, wenn die Zustandsvariable z am Ende eines Durchlaufs der while-Schleife auf -1 steht.

Wir wollen uns die vier Schritte der Suche an einem Beispiel verdeutlichen. Dieses Beispiel ist so einfach, daß man dafür leicht direkt ein Suchprogramm schreiben kann und nicht diesen umständlich wirkenden Weg einschlagen würde. Dennoch wollen wir unseren kanonischen Weg an einem so einfachen Beispiel demonstrieren, weil ein lohnenderes Beispiel sehr leicht unübersichtlich würde.

Beispiel 47
Suche in einem Text aus 0 und 1 eine Null.

1. Ein regulärer Ausdruck für die Menge der Worte über 0 und 1, die mit 0 enden, ist $(0 + 1)^*\,0$.

2. Wir bilden zu diesem regulären Ausdruck nach obigem Verfahren einen ε-NFA, der die so beschriebenen Worte erkennt.



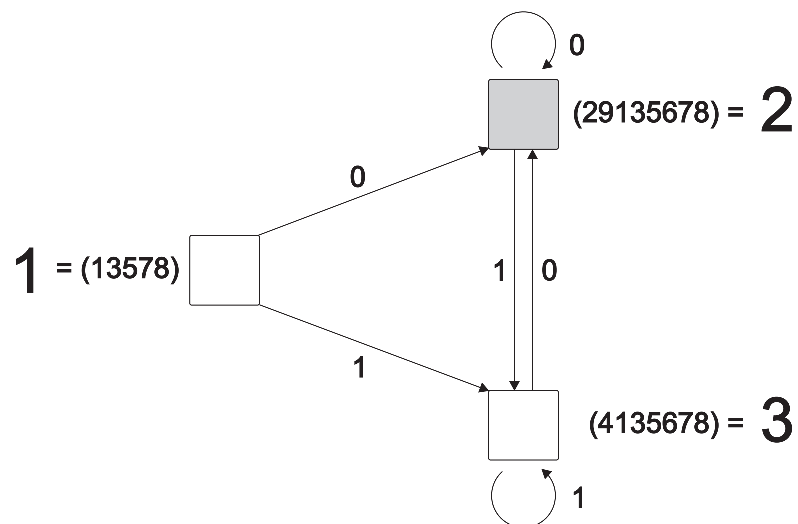
3. Hieraus bilden wir einen äquivalenten deterministischen finiten Potenzautomaten

	δ	0	1
s	$\{13578\}$	$\{2,9,1,3,5,6,7,8\}$	$\{4,1,3,5,6,7,8\}$
f	$\{2,9,1,3,5,6,7,8\}$	$\{2,9,1,3,5,6,7,8\}$	$\{4,1,3,5,6,7,8\}$
	$\{4,1,3,5,6,7,8\}$	$\{2,9,1,3,5,6,7,8\}$	$\{4,1,3,5,6,7,8\}$

Wenn wir diese drei Zustände umbenennen, so haben wir die Befehlsmenge

$$\delta = \{(1,0,2)\,,\,(1,1,3)\,,\,(2,0,2)\,,\,(2,1,3)\,,\,(3,0,3)\,,\,(3,1,3)\}\,,$$

was uns folgenden Graphen liefert:



4. Schließlich schreiben wir eine Prozedur, die diesen DFA realisiert:

```
procedure find(var i, q : integer)
var z, zz : integer
```

```
begin
    z := q;
    while t[i] ≠ # do begin
        if (z = 1) ∧ (t[i] = 0) then zz := 2;
        if (z = 1) ∧ (t[i] = 1) then zz := 3;
        if (z = 2) ∧ (t[i] = 0) then zz := 2;
        if (z = 2) ∧ (t[i] = 1) then zz := 3;
        if (z = 3) ∧ (t[i] = 0) then zz := 3;
        if (z = 3) ∧ (t[i] = 1) then zz := 3;
        z := zz;
        if z = 2 then exit;
        i := i + 1
    end;
    i := −1;
end;
```